Systemorientierte Informatik III

Einführung in Betriebssysteme

Prof. Dr. Willem-Paul de Roever

WS 2003/2004

0 Introduction

1 The Xinu-Approach

1.1 Operating Systems

- The software that controls processes, manages resources and communicates with external devices like disks and printers is called an **operating system**.
- Operating systems allow **multiple users** to **share the machine simultaneously**, protect data from unauthorized access, and keep **dozens of independent devices operating correctly**. The operating system is itself a program that is executed by the same processor that executes user's programs – if the machine is executing a user's program, the operating system is **inactive**.
- An operating system manages to provide **reasonably high-level services** with **unreasonably low-level hardware**.
- Operating systems **hides** the **low-level details** of the real machine, and provide the high-level services of an abstract machine.

Operating Systems: Layering

- Research identified the **abstract services common to all operating systems**, and explored their variations. The **basic operating system components** that carry these machine-independent abstractions have been formulated.
- Researchers discovered a technique called **layering** the organized components which simplifies system design and eases implementation.
- At the heart of the layered organization is the **raw machine**. Subsequent layers of software provide **more powerful primitives**, and **shield** the user from the machine beneath.
- Each layer of the system provides **an abstract resource**, implemented **in terms of the abstract services** provided by lower level layers.

1.2 Our Approach

- The Xinu-approach is a **practical approach**, showing the details of a real system; starting with a micro computer and proceeding step-by-step through the construction of a **layered system**.
- You can see how an entire system fits together, there's **no mystery** about any part of the implementation, it's possible to experiment with the system.
- The **programs and code** form an **integral part** of the book and the lectures, beginning with a layering scheme and following it consistently.
- The design ends with a **complete, working system** that supports multiple processes and a file system.

1.3 What an Operating System is *not*

1. **A language or a compiler:** One needs no special language or compiler to write an operating system.
2. **Command interpreter:** in modern systems command interpreters can be chosen by users – or be written to meet their needs.
3. **Library of commands:** Programs that edit files, send mails, compile programs are just utility programs.

1.4 An Operating System viewed from the Outside

- The essence of an operating system are the **services** it provides to user programs; programs access these services by making **system calls**.
- System calls look like **procedure calls** appearing in ordinary programs, but **transfer to the operating system** routines when invoked at run-time.
- These system calls establish a **boundary** between the running program and the operating system, an operating system can even be **described by its services**.
- The XINU operating system described in COMER's book we reads characters from a keyboard, displays characters on a terminal, operate timers, saves files on disks, relays messages between programs, ...

1.4.1 The Xinu Small Machine Environment

Sometimes computers are **too small to compile** operating systems. How to get an operating system on the computer?

- There's a minimum configuration on the microcomputer, called the **client**.
- The entire system is prepared on a larger machine, called a **host**.
- Compilation is done on host with a **cross-compiler** that runs on the host and produces code for the client.
- A **downloader** copies the memory image to the client (over a serial connection).
- The execution proceeds on the client without the help of the client.

Otherwise, one version of the operating system can be used to create the next version, this is called **bootstraping**.

1.4.2 Xinu services

Programs running under Xinu access services by **calling operating system routines** through system calls. For example the system routine `putc` writes a character on an I/O device. It takes tow arguments: the **device identifier** and the **character to write**.

```
/* ex1.c - main */
#include <conf.h>

/*--------------------------------------------
 * main -- write "hi" on the console
 *--------------------------------------------
*/
main()
{
      putc(CONSOLE, 'h'); putc(CONSOLE, 'i');
      putc(CONSOLE, '\r'); putc(CONSOLE, '\n');
}
```

Example `ex1.c`

The code on the previous slide introduces several **conventions**:

- `ex1.c` is the **name** of the file, the **comment** `/* ex1.c - main */` gives the name followed by the procedure defined by thefile, in this case `main`.
- `#include <conf.h>` includes a file of **configuration declarations** in the source program. This config file contains a definition for `CONSOLE`, witch refers to a **terminal** connected to the computer through which the user interacts. For now we only need to know that this include statement **must appear** in every program that uses **device names**.
- Four characters are written to the terminal: `"h"`, `"i"`, carriage return (`"\r"`) and a line feed (`"\n"`). The latter two are **control characters**.

1.4.3 Concurrent Processing

- Conventional programs are called **sequential** because the programmer imagines a machine with a single processor executing the code **statement-by-statement** without interruption or delay. In contrast, **concurrent processing** means that many computations proceed **“at the same time”**.
- The most visible **concurrency**, multiple independent programs executing simultaneously, is a **grand illusion**! To create that illusion, the operating system **switches** a **single processor** among **multiple programs**, allowing it to execute one for **only few milliseconds** before moving on to another one. This is done at the lowest level of the operating system by the **scheduler**.
- Viewed by a human the processes appear to proceed concurrently. This technique is called **multiprogramming**. Interactive multiprogramming systems are called **timesharing systems**, when the policy used to switch the processor around gives all users equal amounts of CPU time.

1.4.4 The Distinction between Programs and Processes

- The operating system switches the CPU among many computations, called **processes (jobs, tasks)**.
- At any time **several processes** may be executing. It may be that no two of them are executing the same program, but it may also be that all are executing the same statement of the same program.
- Since the processor is switched amongst processes, one process may **overtake** another; **no guarantee** is made about their **relative speeds**.
- Therefore the system procedures must be designed so that cooperation between processes function correctly, independently of their relative speeds; the following example illustrates this:

```
/* ex2.c - main, prA, prB */

#include <conf.h>

/*------------------------------------------------
* main -- example of creating processes in Xinu
*------------------------------------------------
*/
main()
{
  int prA(), prB();

  resume( create(prA, 200, 20, "proc 1", 0));
  resume( create(prB, 200, 20, "proc 2", 0));
}
```

```
/*-----------------------------------
* prA -- repeatedly print 'A'
*        without ever terminating
*-----------------------------------
*/
prA()
{
  while(1)
    putc(CONSOLE, 'A');
}

/*-----------------------------------
* prB -- repeatedly print 'B'
*        without ever terminating
*------------------------------------
*/
prB()
{
  while(1)
    putc(CONSOLE, 'B');
}
```

Explanation for example `ex2.c` (1/2)

- The operating system starts up a **single process** executing the user's main program called `main()`.
- When a process creates a new one by calling the system procedure `create`, the original process **continues** to execute and the new process is ready to execute **concurrently**.
- That single process **executes a call** to system procedure `create`, passing the address of `prA` (and `prB`) as first argument – the other **arguments** of `create` identify the stack space needed for the process created, its priority, the name of the process, the number of arguments and the process' arguments. Each call to `create` forms a **new process** that begins executing instructions at the address specified by its first argument.

Explanation for example `ex2.c` (2/2)

- `create` sets up the process, ready to run, but **temporarily suspended**. It returns the **process id** (an integer that identifies the created process) of the new process to its caller. The system procedure `resume` starts (unsuspends) that process so that it **begins executing**.
- In this example, the first new process prints `A` continuously, the second one prints `B` continuously. Because he processes execute concurrently, the output is a **mixture** of `A`s and `B`s.
- What happens to the initial process executing the main programm? It **exists** after the second call to `resume`, **not affecting** the newly created processes at all.

The Distinction between normal Procedure Calls and Process Creation

- A procedure call **does not return** until the called procedure completes.
- `create` and `resume` **return to the caller** after starting the process, allowing execution of both the calling procedure and the named procedure (e.g. `prA`, `prB`) to proceed **concurrently**.

Example `ex3.c`

- The next example shows that **independent processes** need **not** execute **independent code**. A single process begins executing the main program, calling `create` twice to start new processes; both execute code for the same procedure `prnter(ch)`.
- This points to the **difficulty** involved in designing operating systems: Not only must each process execute **correctly by itself**, it should also not **interfere with other processes** (unless expressly stated), no matter how many processes execute simultaneously.
- Although processes share code, they usually have some **local variables**. `create` allocates a independent set for each process, as indicated. This processes are passed **different arguments**, although executing the **same code**.
- The first new process created by a call to `create` is passed `A` as argument, so it begins execution with formal parameter `ch` set to `A`. The second new process begins with `ch` set to `B`. Although these procedures execute the same code, each has **its own copy** of `ch`, just as recursive invocations of procedures habe their own copy of formal parameters. The output is a **mixture** of `A`s and `B`s.

```
/* ex3.c - main, prntr */

#include <conf.h>
/*------------------------------------------------------------
 * main -- example of 2 processes executing the same code
 * concurrently
 *------------------------------------------------------------
*/
main()
{
    int prntr();
    resume( create(prntr, 200, 20, ``print A'', 1, 'A'));
    resume( create(prntr, 200, 20, ``print A'', 1, 'A'));
}

/*------------------------------------------------------------
 * prntr -- print a chracter
 *------------------------------------------------------------
*/
prtr( ch )
    char  ch;
{
    while ( 1 )
            putc(CONSOLE, ch);
}
```

Some more differences between Programs and Processes

1. A program consist of **code executed by a single process**. In sharp contrast, processes are **not uniquely associated with a piece of code:** multiple processes can execute the same code simultaneously.
2. Storage for local variables and procedure arguments is **associated with the process executing the procedure**, not with the code in which they appear. Therefore, each process has its own stack of local variables, formal parameters and procedure calls.

1.4.5 Process exit

- In the above example (`ex3.c`), the initial process ceased when it reached the **end of the code** of the main program; this is referred to as **process exit**. Other processes exit by **reaching the end of the procedures** in which they start (or by returning from it). Once a process exits, it **disappears forever**.
- Just like a sequential program, each process has **its own stack** of procedure calls. Whenever it executes a call, the called procedure is pushed onto the stack; whenever it return from that procedure, it is popped of the stack. Process exit occurs only when the process **pops its last procedure** (or main program) off the stack.
- System call `kill(P)` **terminates** the process with process identifier `P`. So a process may exit by calling `kill(getpid())` – the call to `getpid()` returns its process id, witch is passed on to `kill`.

1.4.6 Shared memory

- In XINU each process has its own copy of **local variables**, **formal parameters** and **procedure calls**. But processes can also share a set of **global (external) variables** – all processes have access to those variables.
- Example `x4.c` demonstrates this: It's called the **producer-consumer-problem**. In the code, global variable `n`, it is a shared integer, initialized to `0`.
  - The process executing `produce` loops 2,000 times, increasing `n`; it is called the **producer**.
  - The process executing `consume` also loops 2,000 times; it prints the value of `n` in decimal. We call this process the **consumer**.
- **What happens** when `ex4.c` is run under XINU? Will it print all values of `n`?

?

Producer-Consumer-Problem `x4.c`

- **Actually,** it prints a low value of n, say 0, and then 2000. **Why?**
- Even though the two processes run concurrently, they do not require the **same amount of time**. The **producer is fast**, however the consumer quickly fills the available output buffers for printing the values of `n`. So the consumer has to **wait for the output device** to send characters to the console before it can proceed. **While the consumer waits the producer runs**.
- A constraint is: How can the programmer **synchronize** producer and consumer, so that the **consumer receives every datum produced**?
  - The **producer** must **wait for the consumer** to access the datum.
  - The **consumer** must **wait for the producer** to manufacture it.

```
/* ex4.c - main, produce, consume */

int n=0; /* external variables are shared by all processes */

/*------------------------------------------------------
 * main -- example of unsynchronised producer
 * and consumer processes
 *------------------------------------------------------
 */
main() {
     int produce(), consume();
     resume( create(consume, 200, 20, "cons", 0));
     resume( create(consume, 200, 20, "prod", 0));
}

/*------------------------------------------------------
 * produce -- increment n 2000 times and exit
 *------------------------------------------------------
 */
produce() {
        int i;
        for( i=1 ; i<=2000 ; i++)
                n++;
}

/*------------------------------------------------------
 * consume -- print n 2000 times and exit
 *------------------------------------------------------
 */
consume() {
        int i;
        for( i=1 ; i<=2000 ; i++ )
                printf("n is \%d \n", n);
}
```

1.4.7 Synchronization

- However the mechanism for **implementing this waiting** of producer and consumer, called **synchronization**, should be **designed carefully**.
- **In a single processor system, no process should use the CPU while waiting for another process.**
- Executing instructions while waiting for another process is called **busy waiting**. It should be excluded on single-processor implementations because it causes **unnecessary delays**. XINU avoids busy waiting by supplying coordination primitives called **semaphores** and system calls `wait(s)` and `signal(s)` operating on a semaphore `s`:
  - The system call `wait(s)` decrements the semaphore `s` and causes the calling process to wait if the result is negative.
  - The system call `signal(s)` increments `s` – if there's a process waiting for `s`, it's allowed to continue.

Producer-Consumer-Problem `ex5.c`

- To synchronize producer and consumer, two semaphores are needed:
  - One for leaving the **consumer** wait on a value produced.
  - One for leaving the **producer** wait until the consumer is finished with the value just produced.
- Semaphore `s` is created by system call `screate(n)` which takes the desired **initial count** as an argument, returning an integer `s` by which the semaphore is known.
- The modified version of the producer-consumer-problem `ex5.c` using two semaphores now prints **all values from 0 to 1999**.

```
/* ex5.c - main, prod2, cons2 */

int n = 0;  /* n assigned an intial value of zero */
/*------------------------------------------------------
 * main -- synchronized producer and consumer processes
 *------------------------------------------------------
 */
main() {
   int  prod2(), cons2();
   int  produced, consumed;
   consumed = screate(0);
   produced = screate(1);
   resume( create(cons2, 200, 20, "cons", 2, consumed, produced));
   resume( create(prod2, 200, 20, "prod", 2, consumed, produced));
}

/*----------------------------------------------------
 * prod2 -- increments n 2000 times (with semaphore)
 *----------------------------------------------------
 */
prod2(consumed, produced) {
  int i;
  for(i=1 ; i<=2000 ; i++){
      wait(consumed);
      n++;
      signal(produced);
   }
}

/*----------------------------------------------------
 * cons2 -- print n 2000 times (with semaphore)
 *----------------------------------------------------
 */
cons2(consumed, produced) {
  int i;
  for( i=1 ; i<=2000 ; i++){
      wait(produced);
      printf("n is \%d \n", n);
      signal(consumed);
      }
}
```

1.4.8 Mutual exclusion

- Assume two processes willing to print lines. Now a printer can **only serve one process at a time**. Clearly we do not want that these lines are interleaved on the printer – we want each of these processes to have **exclusive access** to the printer, as long as their file is printed.
- These two processes should engage in **mutual exclusion** when cooperating so that **only one of them obtains access** to a resource at a given time. This can also be realized using **semaphores**.
- For example, look at the problem of **updating a list or an array** which is **shared** among a number of processes: Allowing simultaneous access would result in an **undefined outcome** of the operation – so any process operating on that list or array should obtain **exclusive access** to it to **guarantee mutual exclusion**.

Mutual exclusion example

Assuming that a process created a semaphore `mutex` using `screate` before any process calls `additem`, the procedure `additem` can be used to **safely** update an array.

```
/* ex6.c - additem */

int mutex;  /* assume initialized with screat */
int a[100];
int n=0;

/*-------------------------------------------
 * additem -- obtain exclusive access to
 * array 'a' and add item to it
 *-------------------------------------------
*/ additem(item) {

       wait(mutex);
       a[n++] = item;
       signal(mutex);

}
```

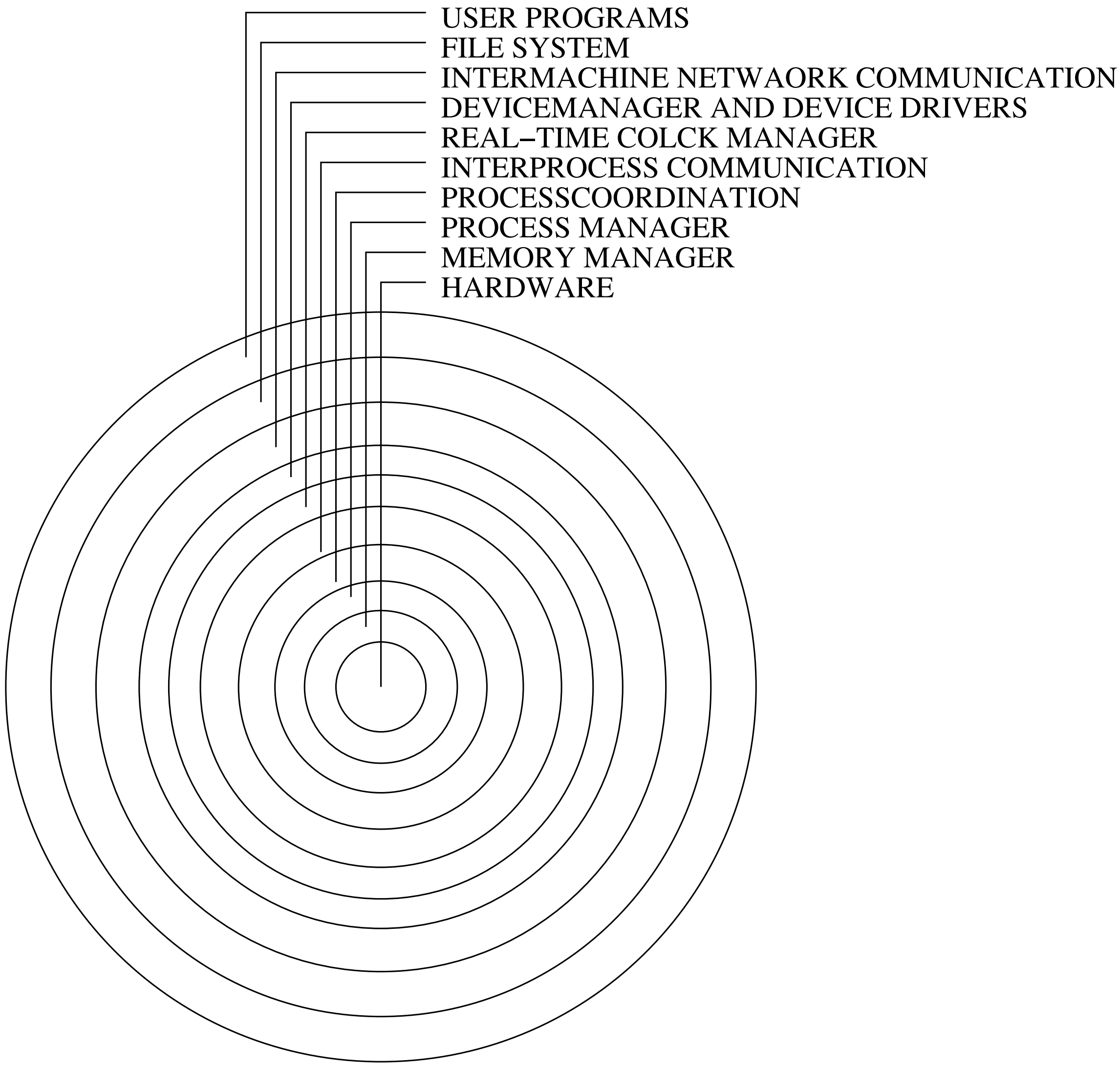
1.5 A layered Operating System

- Xinu is designed in layers. The system functions are partitioned in roughly eight major components, organizing those components into a layered hierarchy.
- Layering the components of an operating system is only done to **make their design and implementation cleaner**. It does not describe **control-flow**. Once finished, the operating system can be called using system procedures in any order.
- The **hardware** lies beneath the lowest layer, the **user's programs** run above these eight layers.

The layering of components in Xinu

1. At the system's heart is the **scheduler** and **context switch**: They switch the CPU among processes **ready to run**.
2. The next layer constitutes the rest of the **process manager** – providing primitives to create, kill, suspend and resume processes.
3. In the process coordination layer **semaphores** are implemented.
4. Message passing is implemented in the **interprocess communication layer**.
5. The **real-time clock manager** schedules processes and delays them (for a given time).
6. On the top of the real-time clock layer lies a layer of **device manager** and **device drivers**, where the device-independent input and output routines are implemented.
7. The layer above the device manager implements machine-to-machine communication, so called **intermachine communication**.
8. The **file system** is controlled by the last layer.

The layering of components in Xinu



2 An overview of the Machine and Run-Time Environment

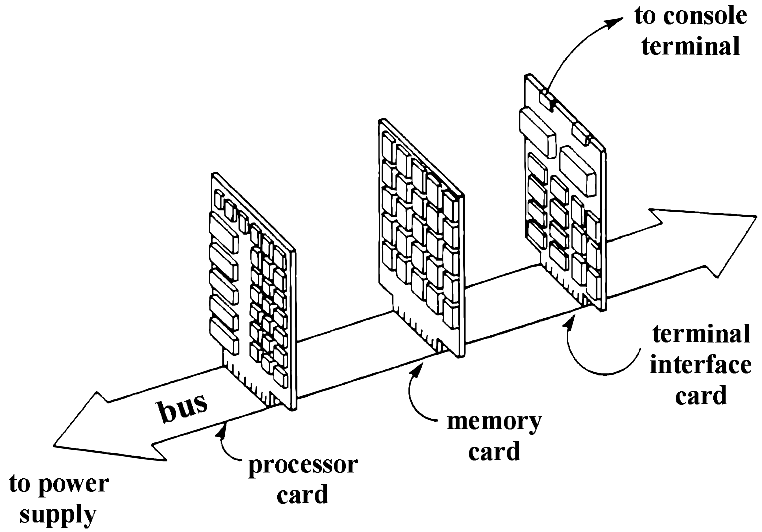
2.1 The Machine

- The machine discussed here is a **Digital Equipment Corporation LSI 11/2** 16-bit microprocessor. This is a microcomputer version of the PDP 11.
- We will discuss:

processor	communication devices	mechanisms such as
memory	asynchronous comm. devices	– stack
architecture	disk storage devices	– vectored interrupts
		– device addressing

2.1.1 Physical Organization of the LSI 11/2

- The LSI 11/2 is constructed from **printed circuit boards** which are slotted into the **sockets of a backplane**.



- Sockets are wired together to form a **bus** (the Q-bus) consisting of **power lines** (attached to all board) and **lines linking up the sockets for inter-board communication**: signals travel to the board on one contact and away on another. One board contains the **11/2 processor** itself (at the beginning of the bus), other boards contain memory or device interfaces for I/O devices.

Communication between boards

- A board communicates with another board by **passing signals across the bus**. E.g., when the processor board needs to **write into memory**, it places **address + data** on the bus for the memory board to retrieve and store.
- Memory is **logically contiguous**: each board contains switches (hardwired jumpers) that can be changed – it is possible to configure **two identical memory boars** so that one responds to **low memory addresses** and the other to **high addresses**. Similarly, I/O devices interface boards contain such switches to represent two distinct devices.

Communication between boards

- The physical order of boards along the Q-bus determines their **priority** (this is used later): Signals enter a board on one connector and leave it on another one, so the board can decide whether to **intercept the signal** or **pass it on down the bus**.
- Example:
  - Two boards are waiting for service.
  - The CPU sends a **grant signal** down the bus.
  - It **passes from board to board** until it reaches a board ready for service.
  - The first board with a device waiting responds to the signal **without passing it on**.
  - Boards further down the bus **wait** until the CPU finishes servicing the first board and reissue the grant signal.

2.1.2 Logical Organization of the LSI 11/2

- The Operating system is concerned with the logical organization of the machine, not its physical organization.

2.1.3 The Registers of the LSI 11/2

- The LSI 11/2 processor contains eight 16-bit **registers** (called $R_0, \ldots, R_5, SP$ and PC) and one 16-bit **processor status word** (PS):

Notation	Register	Use	
R_0	0	general purpose	
R_1	1	general purpose	
R_2	2	general purpose	
R_3	3	general purpose	
R_4	4	previous display	$\longleftarrow$ { points to the previous display, i.e. to the calling procedure's frame
R_5	5	current display	$\longleftarrow$ { points to the current display, i.e. to the currently active called procedure
PC	7	program counter	
PS	status	processor status	

2.1.4 The Address Space

- Memory is divided into 8-bit quantities called **bytes** – also called characters – with the byte being the **smallest addressable unit**.
- Most instructions operate on two bytes (a **word**), the adressed byte and the next higher byte:

address of byte:	addressed byte	$\longleftarrow$ address of the word
address of byte +1:	next higher byte	

- The 11/2 adresses are formulated in **octal**, there are $64k$ Bytes of memory (with $k = 1024$): 2^{16} Bytes $=$ 2^{15} Words.

The Address Space

	$000\,0000_8$:	for interrupt			
		and exception			
	$000\,0777_8$:	vectors	$8^3 - 1 = 511$		
	$000\,1000_8$:	„real“ available	$8^3 = 512$	$\longleftarrow$	this is where programs
$\nearrow$		memory			and data start
address in octals					
$\searrow$					stack grows from high
	$015\,7777_8$:	$\Uparrow$ stack $\Uparrow$	$56k$ Bytes	$\longleftarrow$	to low locations
	$016\,0000_8$:	address	$1 \cdot 8^5 + 6 \cdot 8^4$		
		space for			
	$017\,7777_8$:	devices	$1 \cdot 8^5 + (8^5 - 1)$		

2.1.5 Processor Status Word

- The 16-bit processor status word is divided into four parts:

15 through 8	7 through 5	4	3 through 0
...	**Processor Priority**	**Trace Mode**	**Condition Codes**

- Bits 7 – 5 are called the **processor mask** (or interrupt mask), they are used for setting the processor's priority (ranges from 0 to 7):
  - processor priority of 0 to 3 means external interrupts enabled,
  - processor priority of 4 to 7 means external interrupts disabled.

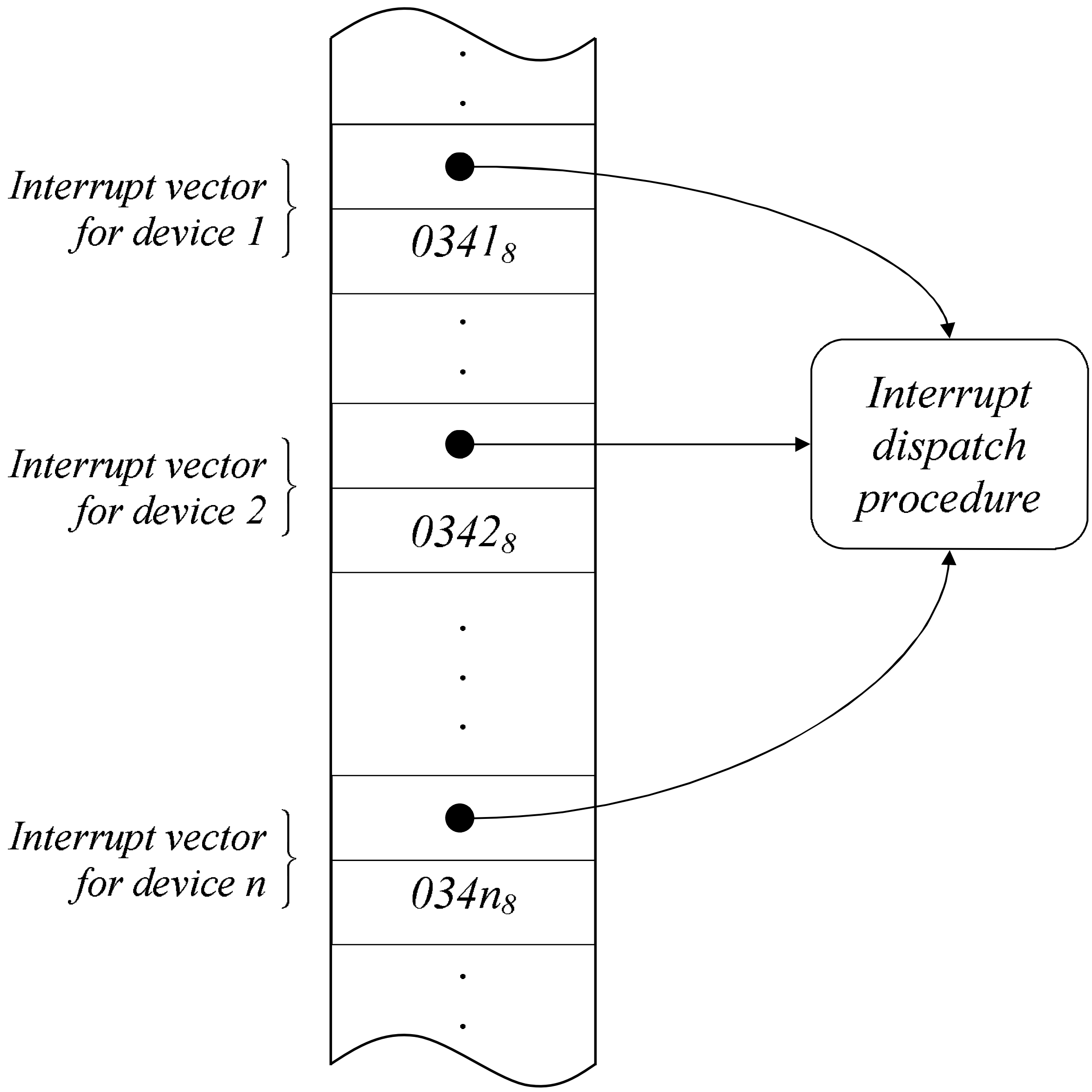
  When needed to disable interrupts, the interrupt level is set to 7 by setting PS's value to $0340_8 = (1 + 2 + 4) \cdot 2^5$.

- Bits 3 – 0 are used for **condition codes**: They are set when e.g. add-instruction produces an arithmetic result, or when comparison is done; records whether result is negative or zero, whether overflow or carry occured; used for conditional bracing instructions.

2.1.6 Vectored interrupts

- The LSI 11/2 uses the **vectored interrupts scheme** for handling exceptions and interrupts from external devices.
- Whenever an external device must communicate with the processor the device **places a signal** on the **interrupt bus line**. If the processor runs with **interrupts enabled**, it checks the interrupt line after executing **each instruction**!
- To handle an interrupt, the processor sends an **acknowledgement** over the bus, requesting the interrupting device to return an **interrupt vector address (IVA)**. The first device with a pending request receives the acknowledgement and responds by returning its interrupt vector address.
- When the CPU receives the address v of the interrupt vector from the Q-bus:
  - the processor pushes the current value of PC and PS on the stack,
  - loads a new PC and PS from two words in memory starting at location v and
  - continues execution at the new location addressed by the PC.

Vectored interrupts



Vectored interrupts

- Each device is assigned a unique IVA, enabling the system software to identify the device, i.e. distinguish between different devices.
- The *PC* stored at v points to an **interrupt service routine** (interrupt handler) for the interrupting device.
- The interrupt „acts“ like a procedure call **inserted** (invisibily) by the hardware in **between two instructions** in user's code.
- The processor executes the code in the interrupt routine and then returns to the place where the user's process was interrupted.

Vectored interrupts

- To make the interrupt invisible (transparent) to the running program, the interrupt handler must **save and restore the state of the machine**. Saving the state of the 11/2 means saving $R_0 - R_6$, since the hardware saves $R_7 = PC$ and PS.
- In practice the stack pointer R_6 needs **not** to be saved, provided the address of the interrupt routine is **popped off from the stack before returning**, whenever this address has been pushed on the stack before, thus restoring the stack to its original position.
- To prevent an interrupt routine from being interrupted by itself the PS in the IVA specifies a priority level 7, i.e. as soon as **interrupt processing** begins, the processor operates with **interrupts disabled**.

Vectored interrupts

- Interrupt processing ends when the processor executes a **„return from interrupt"** instruction (`rti`, `rtt`), thereby restoring the old values of the PC and PS (before executing the interrupt) are restored from the stack in a **single step**, thereby the SP is restored to its **original value**.
- This amounts to **reversing** the actions taken by the processor when it detected the interrupt, allowing processing to continue where it was interrupted **as if nothing had happened**!

2.1.7 Exceptions

- **Exceptional conditions** are handled like interrupts. Examples for exceptions are: reference to nonexistent memory, execution of invalid instruction code, power failure or other errors detected by the hardware.
- Similarly as above the PC and PS are pushed onto stack and new values for PC and PS are loaded from memory. Exception vectors are **permanently** assigned to memory locations $0 - 27$. Unlike interrupts, they **cannot be disabled** by changing the priority level!

2.1.8 Asynchronous Communication

- An **asynchronous communication device**, the **serial line unit**, sends and receive characters (to/from) a conventional terminal. At least one board in any 11/2 system contains such a device, connecting the computer to a terminal.
- The Cable connecting the device to terminal consists of three wires: one carries data into the computer, one carries data away from the computer and one is the ground voltage wire.
- Signals are **serial** because they travel down the line one bit at-a-time – and they are **asynchronous** because the transmitter sends a character whenever one is available, regardless of the state of the receiver. I.e., there is no synchronization between the transmitter and receiver to control the start of a character.

Asynchronous Communication

- The work to **convert characters** into a **series of pulses** or to covert a series of pulses into characters is carried out by a single integrated circuit (IC) controlled by a precise clock called **Universal Asynchronous Receiver and Transmitter (UART)**.
- **Errors** occur when a receiver can not make **sense** out of the signals it receives. The receiver itself holds **one character at a time**. The CPU must extract **each character** before subsequent characters arrive or the receiver indicates an **overrun condition** and replaces the earlier character with the latest one.

2.1.9 LSI 11/2 Asynchronous Serial Line Hardware

- Conceptually: asynchronous transmitters and receivers are grouped into pairs consisting of one receiver and one transmitter to model two-way communication.

2.1.10 Addressing a Serial Line Unit

- The CPU communicates with a SLU by reading and writing to addresses **beyond the end of real memory**. Boards containing device interfaces watch the bus for addresses corresponding to their devices, and respond accordingly.
- From the programmer's point of view, each SLU contains four 16-bit registers:
  - `RCSR`: receiver control and status registers
  - `RBUF`: received data buffer
  - `XCSR`: transmitter control and status registers
  - `XBUF`: transmitter data buffer
- To communicate with the device, a program **reads or writes the address corresponding to the device register address**, just as it reads or writes to a real memory address.
- Device registers are **not** storage locations! Rather, the interface board containing the device **intercepts** the transfer and **interprets** it as an instruction to control the device or to transfer data between the device and the CPU. So I/O is a side-effect of manipulating certain locations in the address space.

Arrangement of Serial Line Unit Registers

	high-order byte								*low-order byte*							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RCSR	–	–	–	–	–	–	–	–	**D**	**I**	–	–	–	–	–	–
	(unused)								***receiver control (see below)***		*(unused)*					
RBUF	**E**	**V**	**F**	**P**	**0**	**0**	**0**	**0**	*	*	*	*	*	*	*	*
	receiver errors								***character***							
XCSR	–	–	–	–	–	–	–	–	**T**	**I**	–	–	–	–	–	**B**
	(unused)								***transmitter control (see below)***		*(unused)*					
XBUF	–	–	–	–	–	–	–	–	*	*	*	*	*	*	*	*
	(unused)								***character***							

RCSR	Bit 7 (**D**)	1 ≙ character received from SLU
RCSR	Bit 8 (**I**)	1 ≙ SLU posts interrupt when character received
XCSR	Bit 7 (**T**)	1 ≙ free to transmit, character has been sent
XCSR	Bit 8 (**I**)	1 ≙ SLU has received character (finished sending) and new character can be written

Examples of addressing a Serial Line Unit

- **Writing** a character to the `XBUF` register causes the SLU to capture that character and **transmitting** it down to the SLU. Once the board accepts a character, it operates **independently** of the CPU (processing continues while the transmitter sends the character).
- The CPU receives a character received over the SLU by **reading from the `RBUF` address**:

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
error occured	overflow error	framing error	parity error	(unused)	**character**

- `RBUF` is **read only** – write instructions directed to that address are ignored. Accessing RBUF has a **side effect**: the receiver is cleared an enabled to receive the next character.

2.1.11 Polled vs. Interrupt-Driven I/O

$\longrightarrow$ discussed later in chapter 13

2.2 Disk Storage Organization

$\longrightarrow$ discussed later in chapter 17

2.3 The C Run-Time Environment

- Operating Systems are written in **high-level language** to make them **easier** to write, understand, debug, and move to other machines.
- However, sometimes **machine assembly language** procedures are introduced because **machine quantities** must be manipulated, e.g., for saving and restoring the machine's registers by context-switching code, interrupt routines, manipulating the process table, implementing semaphores, operating on the real-time clock, and so on.

The C Run-Time Environment

- The C-Compiler expects each program to run in an address space as the following:

↓ 0	↓`_etext`	↓`_edata`	↓`_end`
text	**data**	**bss**	**free space** ⟵ ··· **stack**

  - **text** segment contains text of main program and all its procedures
  - **data** segment contains all initialized data
  - **bss** segment contains all uninitialized data
  - the **stack** occupies the highest part of address space and grows downwards
  - **heap storage** ist allocated from `_end` upwards, i.e., it grows from the bottom of the free space upwards.
- Symbols `_etext`, `_edata`, `_end` refer to global variables inserted into object code by the loader. They are initialized to the first address beyond **text**, **data** and **bss** segments, respectively.
- The running program can find out how much free memory remains between the end of loaded part and the current top of the stack by taking the address of `_end`.

The C Run-Time Environment

- When XINU runs multiple processes, it allocates a **separate stack** from the **highest available free space** for each process:

↓ 0	↓`_etext`	↓`_edata`	↓`_end`			
text	**data**	**bss**	**free**	**stack** # 3	**stack** # 2	**stack** # 1

- **Stack overflow** in one process destroys stacks of other processes – techniques to prevent this from happening are discussed in chapter 15.

- Stack space is **returned** to the free list whenever a process exits.

Conventions for translating procedures in the C-compiler

- Consider a **calling procedure** A and a **called procedure** B:

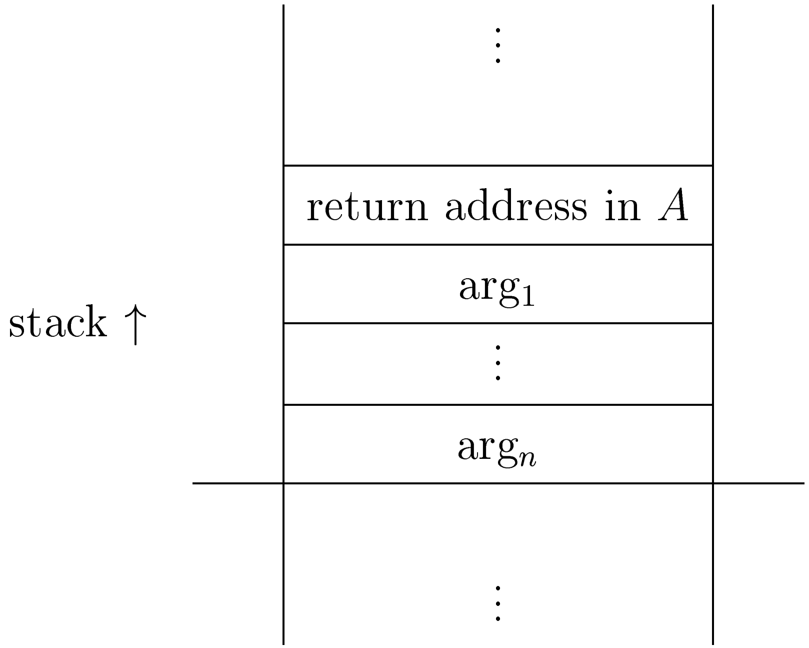
```
proc A:
    ...; B(arg_1, ..., arg_n); ...
```

- How is the code of B within the body of A implemented in C's compiler?
  1. Values of actual **parameters** of B are pushed on stack (in **reverse** order).
  2. **Return address** in A, i.e., of the instruction following the call to B, is pushed on the stack.
  3. **Flow control** branches to B.

  The latter two are carried out by calling a **single instruction**: `jsr pc,` $address_B$.

- Calling procedure A is also responsible for **popping** B**'s arguments** from the stack after the called procedure B returns!

Image of Stack



Conventions for the called procedure B

- The called procedure B is **responsible** for
  - **saving and restoring the machine state**, i.e., the contents of the registers
  - **popping the return address** from the stack before returning
- Saving the contents of the used registers is done by calling the assembly language routine `csv` (**C-registers save routine**).
- Restoring the contents of the used registers is done by calling the assembly language routine `cret` (**C-registers restore routine**).
- Hardware provides a **single instruction** `rts pc` that pops an address from the stack and returns to that address.

Schematic translation of calling procedure A

(1) . . . code for A . . .

(2) code for putting the arguments of B's call on the stack in reverse order

(3) `jsr pc,` $address_B$

— this is where B is executed —
(original value of PC points to 4.)

→(4) code for popping B's arguments from the stack

(5) . . . code for A . . .

(6) `rts pc` (to return to caller of A)

Schematic translation of called procedure B

(1) `move r5, r0` (code for pushing the old value of R_5 on the stack)

(2) `move pc+2, r5` (assigning to R_5 the return address in B after calling `csv`)

(3) `jsr r0, csv` (jumping to `csv`)

(4) . . . code for B . . .

— Assumption: after executing fragment (4) of B's code,
the stack has the same contents, and the values of SP
and R_5 are the same es before executing fragment (4) —

(5) `rts pc` (pops address of `cret` from the stack and returns to `cret`)

Image of Stack

	⋮	
	address of `cret`	done by `csv`
	old value of R_2	done by `csv`
	old value of R_3	done by `csv`
	old value of R_4	done by `csv`
stack $\uparrow$	old value of R_5	`jsr r0, csv`
	return address in A	`jsr pc,` $address_B$
	arg_1	
	⋮	
	arg_n	
	⋮	

C-registers save routine `csv` and C-registers restore routine `cret`

- To save the machine registers, the compiler inserts a call to `csv`, an assembly language routine. `csv`
  - **saves the registers** and
  - **jumps back** to called routine B.
- When code for B is finished, compiler inserts a call to assembly language routine `cret` to restore the values of the machine's registers and the stackpointer SP ($= R_6$) and to return control to the original caller A. `cret`
  - restores **old values of machine registers**
  - restores the **stack pointer**
  - **returns control** to original caller

C-registers save routine `csv` and C-registers restore routine `cret`

`csv:`

(1) $R_0 := R_5$ (by assumption R_5 contains return address in B, and stack has old value of R_5, the return address in A, and B's arguments on it)

(2) $R_5 := SP$ (R_5 stores value of SP, i.e., address of B's frame)

(3) R_4, R_3, R_2 pushed on stack in that order

(4) `jsr pc, (r0)` (old value of PC, i.e., address of `cret`, pushed on stack and control branches to address stored in R_0, i.e., return address in B)

`cret:`

(1) $R_2 := R_5$ (put copy of called frame pointer in R_2, see (2) above in `csv`)

(2) restore R_4, R_3, R_2 in that order

(3) $SP := R_5$ (restore stack pointer)

(4) `mov (sp)+, r5` (restore R_5 on stack by call to `csv` procedure entry)

(5) `rts pc` (return to A)

C-registers save routine csv and C-registers restore routine cret

```
/* csv.s - csv, cret */

/* C register save: upon entry here, procedure A has called B   */
/* and B has called csv to save registers. r5 contains return   */
/* address in B. The stack has old r5, return address in A, and */
/* arguments on it. C return: cret (below) is used to restore   */
/* regs when the called proc. exits                             */

      .glob csv, cret
/*-------------------------------------------------------------*/
/* csv  -- C register save routine
/*-------------------------------------------------------------*/
csv:
      mov   r5,r0     / r0 not saved at call (C convention)
      mov   sp,r5     / r5 points to called routine's frame
      mov   r4,-(sp)  / push r4 - r2 on stack
      mov   r3,-(sp)
      mov   r2,-(sp)  / jsr pushes PC onto stack hoes to
      jsr   pc,(r0)   / address in r0 (originally in r5)

/*-------------------------------------------------------------*/
/* cret  -- C register restore routine
/*-------------------------------------------------------------*/
cret:
      mov   r5,r2     / put copy of called frame ptr in r2
      mov   -(r2),r4  / reload r4 - r2 from start of frame
      mov   -(r2),r3
      mov   -(r2),r2
      mov   r5,sp     / restore SP
      mov   (sp)+,r5  / restore r5 on stack by call
                      / to csv procedure entry
      rts   pc        / return to caller
```

3 List and Queue Manipulation

- List processing is fundamental for operating system since one needs:
  - Lists of processes to be **scheduled (ready for scheduling)**
  - Lists of processes **waiting on a semaphore**
  - Lists of processes ordered by **priority (time)** in order to implement **time-sharing (sleep list)**.
- We need
  - **FIFO queues** (ordered by **time of insertion**) and
  - **priority queues** (ordered by **time to wake up** in view of time-sharing).

Operations on FIFO and Priority Queues

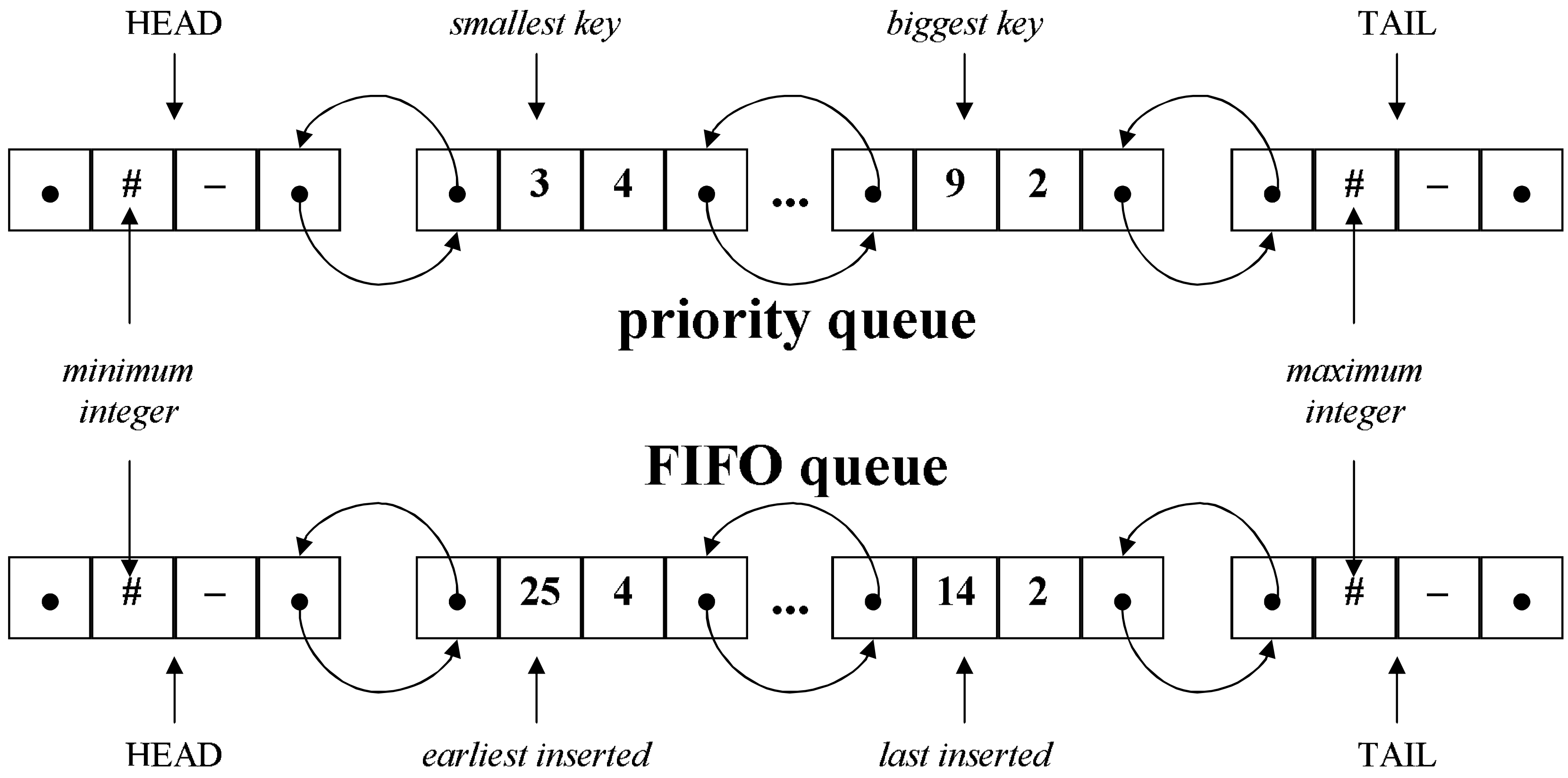
- Operations needed on FIFO and priority queues are:
  - **inserting** an item at the **tail** of a list or within an **ordered** list,
  - **removing** an item at the **head** of a list or from an **ordered** list,
  - allocating a **new list**.
- Programming these operations is **simple** because **only one process at a time** executes these operations. Thus the reader can think of the code as a sequential program.

  This is due to the fact that these operations occur only in **critical sections** (programmed by setting the priority of the process status register PS in the associated process to 7, hence making the process deaf to external interrupts).

Implementation of FIFO and Priority Queues

- All lists are **doubly linked**, i.e.
  - each list has both a **head** and a **tail**
  - each node points to its **successor** as well as **predecessor**
  - predecessor of head and successor of tail point to the empty list ($\texttt{EMPTYLIST} = -1$)
- Each node contains a **key**, headnode contains `MININT` as key, tailnode `MAXINT`.
- Elements are extracted from a **FIFO queue** by **removing** them from the **head**, and they are **inserted** at its **tail**.
- If the list is a **priority queue**, `getfirst(head)` removes item with **smallest key**, and `getlast(tail)` removes item with **biggest key**. Hence priority queue insertion uses a loop starting at head of the priority queue.

FIFO and Priority Queues



3.1 Linked Lists of Processes

- In the lists of processes,
  1. the items to be stored are process identifiers,
  2. now there are `NPROC` processes,
  3. the process priority is $\geq$ `MININT` and $\leq$ `MAXINT`.
- Optimization is possible because a process **appears on at most one list at any time** by **using process identifier values as array indices**.
- The q-**array** consists conceptually of **two parts**:
  1. **nodes for each process** in its `[0 ... NPROC-1]`-part and
  2. a pair of **head and tail nodes** for every separate list in its `[NPROC ... NQENT]`-part.
- The q-array is defined **external**; it's an array of `qent` structures as declared in file `q.h`.

The *q*-Array

array index	*key*	*next*	*previous*
0			
1			
2	14	33	4
3			
4	25	2	32
...			
`NRPOC-1`			
`NPROC`			
...			
32	`MININT`	4	-1
33	`MAXINT`	-1	2
...			

Logical Pictures for Operations

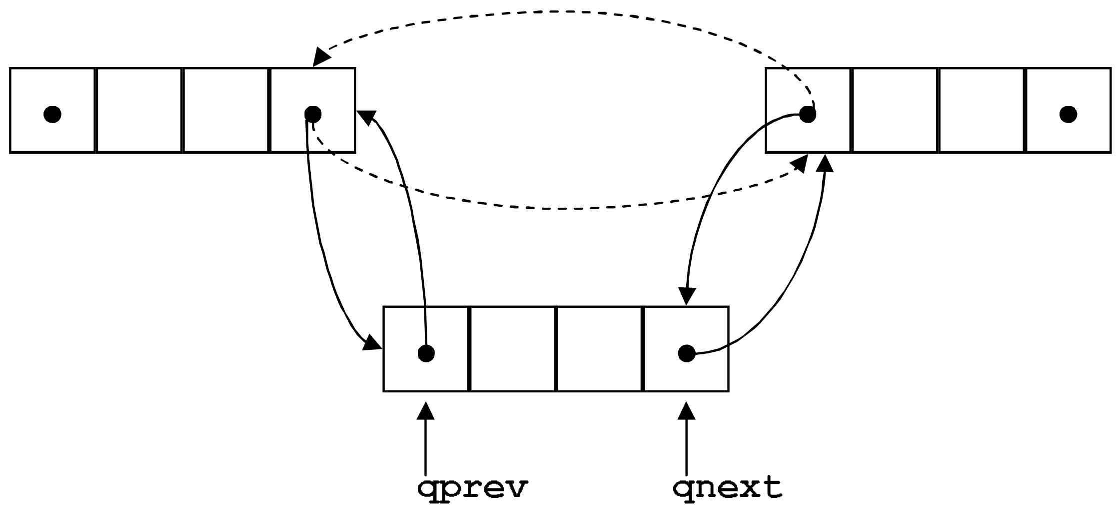
- In order to understand the procedure `enqueue` – for inserting an element at the tail of the list – consider the figure on slide no. 9 and **check the correspondence instruction by instruction**!
- In order to understand the procedure `dequeue` - for removing an item from a list - consider the figure on slide no. 10 and again, **check the correspondence** between `dequeue` and the picture **instruction by instruction**!
- In order to undestand the procedure `newqueue` – for creating a new, empty q-structure – consider the figures on slide no. 13.
- A similar ficture can be found on slide no. 11 for picturing the operation of inserting a node in an ordered list by procedure `insert`.

3.2 Implementation of the q-Structure

```
/* q.h - firstid, firstkey, isempty, lastkey, nonempty */
/* q structure declarations, constants, and inline procedures               */

#ifndef     NQENT
#define     NQENT       NPROC + NSEM + NSEM + 4 /* for ready & sleep         */
#endif

struct  qent   {              /* one for each process plus two for each list */
        short  qkey;          /* key on which the queue is ordered           */
        short  qnext;         /* pointer to next process or tail             */
        short  qprev;         /* pointer to previous process or head         */
        };

extern  struct  qent q[];
extern  int     nextqueue;

/* inline list manipulation procedures */
#define isempty(list)   (q[(list)].qnext >= NPROC)
#define nonempty(list)  (q[(list)].qnext < NPROC)
#define firstkey(list)  (q[q[(list)].qnext].qkey)
#define lastkey(tail)   (q[q[(tail)].qprev].qkey)
#define firstid(list)   (q[(list)].qnext)

#define EMPTY -1              /* equivalent of null pointer                  */
```

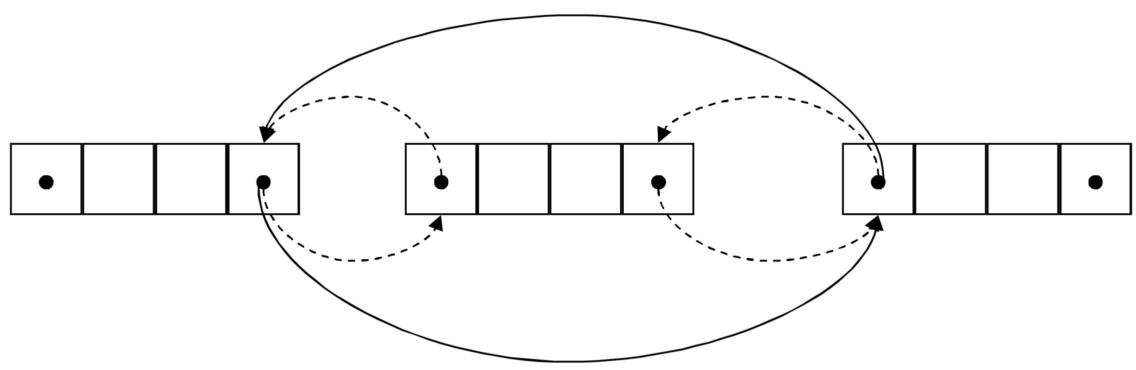
3.2.2 FIFO Queue Manipulation: enqueue

```
/*------------------------------------------------------------------------
 * enqueue  --  insert an item at the tail of a list
 *------------------------------------------------------------------------
 */
int     enqueue(item, tail)
        int     item;                   /* item to enqueue on a list    */
        int     tail;                   /* index in q of list tail      */
{
        struct  qent    *tptr;          /* points to tail entry         */
        struct  qent    *mptr;          /* points to item entry         */

        tptr = &q[tail];
        mptr = &q[item];
        mptr->qnext = tail;
        mptr->qprev = tptr->qprev;
        q[tptr->qprev].qnext = item;
        tptr->qprev = item;
        return(item);
}
```



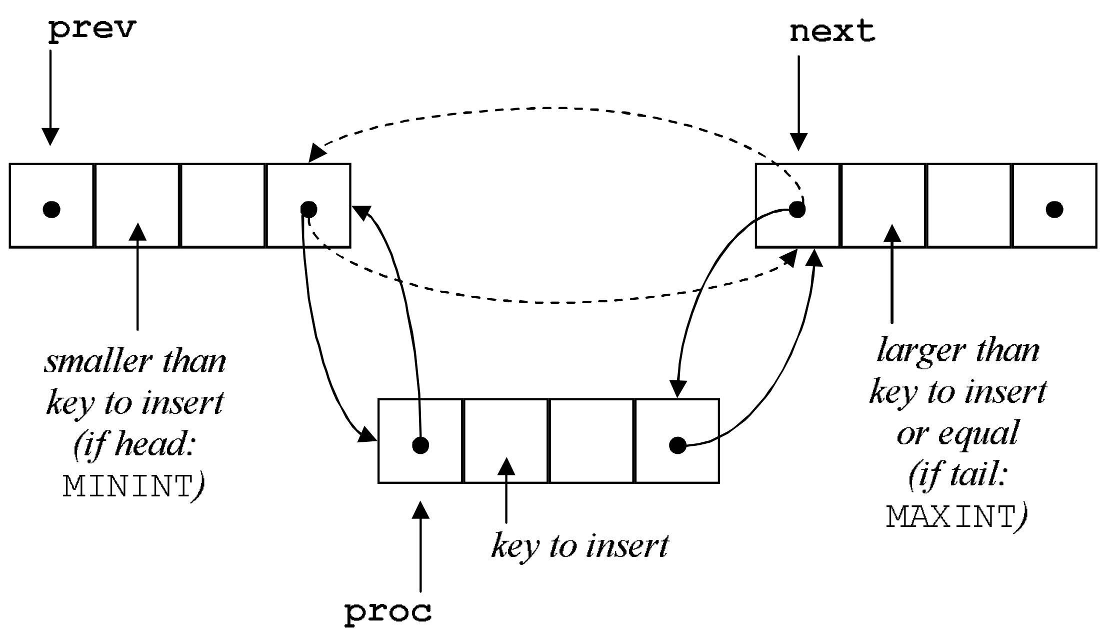
FIFO Queue Manipulation: dequeue

```
/*------------------------------------------------------------------------
 *  dequeue  --  remove an item from the head of a list and return it
 *------------------------------------------------------------------------
 */
int     dequeue(item)
        int     item;
{
        struct  qent    *mptr;          /* pointer to q entry for item  */

        mptr = &q[item];
        q[mptr->qprev].qnext = mptr->qnext;
        q[mptr->qnext].qprev = mptr->qprev;
        return(item);
}
```



3.3 Priority Queue Manipulation

```
/*------------------------------------------------------------------------
 * insert.c  --  insert an process into a q list in key order
 *------------------------------------------------------------------------
 */
int     insert(proc, head, key)
        int     proc;                   /* process to insert            */
        int     head;                   /* q index of head of list      */
        int     key;                    /* key to use for this process  */
{
        int     next;                   /* runs through list            */
        int     prev;

        next = q[head].qnext;
        while (q[next].qkey < key)      /* tail has maxint as key       */
                next = q[next].qnext;
        q[proc].qnext = next;
        q[proc].qprev = prev = q[next].qprev;
        q[proc].qkey  = key;
        q[prev].qnext = proc;
        q[next].qprev = proc;
        return(OK);
}
```

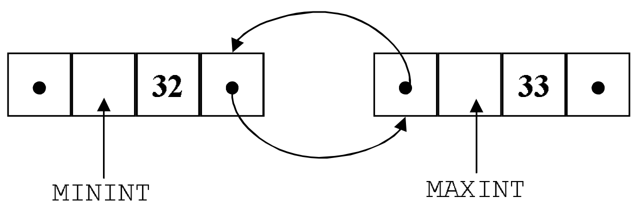


Priority Queue Manipulation

```
/*------------------------------------------------------------------------
 * getfirst  --  remove and return the first process on a list
 *------------------------------------------------------------------------
 */
int     getfirst(head)
        int     head;                   /* q index of head of list      */
{
        int     proc;                   /* first process on the list    */

        if ((proc=q[head].qnext) < NPROC)
                return( dequeue(proc) );
        else
                return(EMPTY);
}


/*------------------------------------------------------------------------
 * getlast  --  remove and return the last process from a list
 *------------------------------------------------------------------------
 */
int     getlast(tail)
        int     tail;                   /* q index of tail of list      */
{
        int     proc;                   /* last process on the list     */

        if ((proc=q[tail].qprev) < NPROC)
                return( dequeue(proc) );
        else
                return(EMPTY);
}
```

3.4 List Initialization

```
/*------------------------------------------------------------------------
 * newqueue  --  initialize a new list in the q structure
 *------------------------------------------------------------------------
 */
int     newqueue()
{
        struct  qent    *hptr;          /* address of new list head     */
        struct  qent    *tptr;          /* address of new list tail     */
        int     hindex, tindex;         /* head and tail indexes        */

        hptr = &q[ hindex=nextqueue++]; /* assign and rememeber queue   */
        tptr = &q[ tindex=nextqueue++]; /* index values for head&tail   */
        hptr->qnext = tindex;
        hptr->qprev = EMPTY;
        hptr->qkey  = MININT;
        tptr->qnext = EMPTY;
        tptr->qprev = hindex;
        tptr->qkey  = MAXINT;
        return(hindex);
}
```



array index	*key*	*next*	*previous*
NPROC			
...			
32	MININT	33	−1
33	MAXINT	−1	32
...			

4 Scheduling and Context Switching

Context Switching

- Operating system achieves **the illusion of concurrent processing** by rapidly switching one processor among several **computations**
- **Context switching** at the heart of the processor juggling act: consists of
  1. stopping the current computation
  2. saving informations needed for restarting it later and
  3. starting a possibly different one.
- **Why is this job difficult**? The CPU **can not be stopped** at all – it continues to execute code until it switches to another process!
- This chapter describes basic context switch mechanism showing how a process saves its **state information**, **chooses** another process to run and **returns control** to that process. It describes also the basic **data structure**, the **process table**, holding all information about processes.

4.1 Process Table

- Information about processes are kept in the **process table** **`proctab`**, an array of `pentry`-structure:
  - There's **one entry for each process** in this table.
  - Because only one process is **running** at a time, one of these entries is **out-of-date**, since corresponding with the currently active process.
  - The other entries correspond to processes witch are **temporarily halted**.
- Exactly **what information must be saved** in `proctab`? All values that will be **destroyed when another process runs**.
- E.g., since in XINU each process has **its own segment of the stack**, no copy of the stack needs to be saved. However a new process will change the contents of the **machine registers**, so this must be saved. Also: information used to control the process and account for its resource.
- Processes are referenced by their **process id**, witch is the index of the saved-state-info in `proctab`; e.g. `pbase`, `pstklen`, and `plimit` are used to **free memory space** when process completes.

Process Table in proc.h

```
#ifndef     NPROC                   /* set the number of processes          */
#define     NPROC       30          /*  allowed if not already done         */
#endif

/* miscellaneous process definitions */

#define     PNREGS      9           /* size of saved register area          */
#define     PNMLEN      8           /* length of process "name"             */
#define     NULLPROC    0           /* id of the null process; it           */
                                    /*  is always eligible to run           */

#define     isbadpid(x) (x<=0 || x>=NPROC)

/* process table entry */

struct  pentry {
        char   pstate;              /* process state: PRCURR, etc.          */
        short  pprio;               /* process priority               / see */
        short  pregs[PNREGS];       /* saved regs. R0-R5,SP,PC,PS <--|  pg. */
        short  psem;                /* semaphore if process waiting   \ 24  */
        short  pmsg;                /* message sent to this process         */
        short  phasmsg;             /* nonzero iff pmsg is valid            */
        short  pbase;               /* base of run time stack               */
        short  pstklen;             /* stack length                         */
        short  plimit;              /* lowest extent of stack               */
        char   pname[PNMLEN];       /* process name                         */
        short  pargs;               /* initial number of arguments          */
        short  paddr;               /* initial code address                 */
};

extern  struct pentry proctab[];
extern  int    numproc;             /* currently active processes  \  see   */
extern  int    nextproc;            /* search point for free slot   | page  */
extern  int    currpid;             /* currently executing process /  196   */
```

4.2 Process States

- Process states are needed for **checking validity** of **operations performed on a process**.
- The following process states are defined in `proc.h`:

```
/* process state constants */

#define  PRCURR   '\001'  /* process is currently running  -- chapter  4 */
#define  PRFREE   '\002'  /* process slot is free          -- chapter  4 */
#define  PRREADY  '\003'  /* process is on ready queue     -- chapter  4 */
#define  PRRECV   '\004'  /* process waiting for message   -- chapter  7 */
#define  PRSLEEP  '\005'  /* process is sleeping           -- chapter 10 */
#define  PRSUSP   '\006'  /* process is suspended          -- chapter  5 */
#define  PRWAIT   '\007'  /* process is on semaphore queue -- chapter  6 */
```

4.3 Selecting a Ready Process

- A process is classified as **ready** (`PRREADY`) when it is **eligible for CPU** but **not currently executing**.
- The **single process served by the CPU** is classified as **current** (`PRCURR`).
- Switching context involves **selecting a process** from those that are ready or current and **giving control** to selected process.
- The software implementing the **policy to select a process** from among those that are ready or current is called **scheduler**.

Selecting a Ready Process: Xinu's Policy

- XINU's policy for selecting a process is:

  At any time, the highest priority process eligible for CPU service is executing!

  Among processes with equal priority, scheduling is round-robin – i.e., all processes with equal priority are selected one-after-another so that **all members of this set are executed before any member has a second opportunity**.

- **Priorities** are **positive integers** kept in `pprio`-field of process table entry.
- All ready processes appear in the **ready list ordered by priority** (a priority queue in the *q*-structure). Global variables `rdyhead` and `rdytail` point to head and tail of this *q*-structure.
- In XINU, the **current process** is **not on ready list**, its process identifier is given by the global variable `CURRPID`.

Selecting a Ready Process: Xinu's Policy

- What happens to **currently executing process** during a context switch? – Often, it **remains eligible to use CPU**, even when control is temporarily passed to another process (e.g., as a result of round-robin scheduling). Then its current **process state changes** to **PRREADY** and it is moved to ready list for later CPU service.
- How dose the **rescheduler**, `resched`, decide whether to move current process to ready list? – If **current process** is **not eligible** to use CPU, the system routines assign to its `pstate`-field a desired next state before calling `resched`, see e.g. in `suspend` (see page 69) or in `kill` (see page 71).
- When `resched` prepares to **switch context**, it checks `pstate`-field of current process and makes it ready for execution by CPU, only if its `pstate` is `PCURRENT`.

Selecting a Ready Process: `resched`

The XINU scheduler **`resched`**

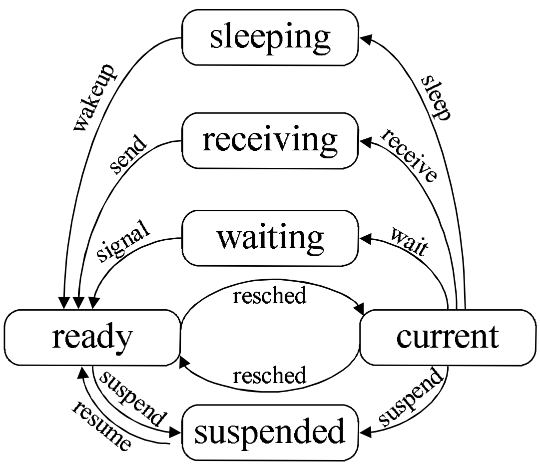
- selects a process to run
- changes process table for that entry
- removes new process from ready list and makes it current
- updates `CURRPID`
- resets preemption counter (to give the new process maximal amount of slot time to run, see pages 73 and 125 – it's set to `QUANTUM` = 10)
- calls `ctxsw(optr <- pregs, nptr <- pregs)` to save and reset the machine registers

Selecting a Ready Process: resched

```
/*------------------------------------------------------------------------
 * resched  --  reschedule processor to highest priority ready process
 *
 * Notes:   Upon entry, currpid gives current process id.
 *          Proctab[currpid].pstate gives correct NEXT state for
 *                current process if other than PRREADY.
 *------------------------------------------------------------------------
 */
int   resched()
{
        register struct pentry  *optr;   /* pointer to old process entry */
        register struct pentry  *nptr;   /* pointer to new process entry */

        /* no switch needed if current process priority higher than next */

        if ( ( (optr= &proctab[currpid])->pstate == PRCURR) &&
           (lastkey(rdytail)<optr->pprio))
              return(OK);

        /* force context switch */

        if (optr->pstate == PRCURR) {
              optr->pstate = PRREADY;
              insert(currpid,rdyhead,optr->pprio);
        }

        /* remove highest priority process at end of ready list */

        nptr = &proctab[ (currpid = getlast(rdytail)) ];
        nptr->pstate = PRCURR;           /* mark it currently running    */
#ifdef  RTCLOCK
        preempt = QUANTUM;               /* reset preemption counter     */
#endif
        ctxsw(optr->pregs, nptr->pregs);

        /* The OLD process returns here when resumed. */
        return(OK);
}
```

Which processes call `resched`?

- **Answer:** The process which is executed unter XINU **on this exact moment**. See process state diagram on pg. 128:



`sleep10(n)`	pg. 129
`wakeup()`	pg. 133
`receive()`	pg. 97
`send(...,...)`	pg. 96
`wait(sem)`	pg. 85
`signal(sem)`	pg. 86
`sdelete(sem)`	pg. 89
`suspend(pid)`	pg. 69
`resume(pid)`	pg. 67
`kill(pid)`	pg. 71

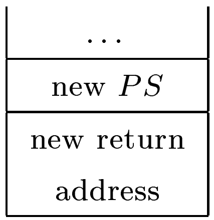
- So `resched()` is a **normal procedure**, and calling it results in executing it **like a normal procedure**.
- However, `ctxsw` is not a normal procedure call in that after executing it one doesn't return (immediately) to the next statement in `resched()`.

```
int resched() {
...
  ctxsw(optr->pregs,nptr->pregs);
  return(OK);
}
```

- **So when is the return address in `resched()` reached?**

Context Switching: `ctxsw`

- The code of `ctxsw(...,...)` is **machine dependent** because machine registers should be saved by it.
- *PC* must be changed **last** to give the CPU the opportunity **executing the new process** (and not earlier), once information about old process is stored in `proctab` and stack.
- On the 11/2 the `rtt` instruction pops **both** *PS* **and** *PC* **from the stack**, and reload them **in one step**:



`rtt` operates on these two: reloads *PS* and *PC* with their values, and pops them

- So after saving registers associated with the old process in the **register save area** of that process, `ctxsw` **switches stack** to new process stack, restore the values of new registers, and, after having pushed **new values of** *PS* **and** *PC* **on stack** it resets the *PS* and *PC* in one go by calling `rtt`.

Context Switching: `ctxsw`

- Convention for calling procedures (see pg. 37):

  Calling procedure pushes the call's actual arguments on the stack in reverse order, pushes the return address on the stack, and then braches to the called routine.

- So when calling `ctxsw(optr<-pregs,nptr<-pregs)` (i.e. after `jsr pc, _ctxsw` was called in translation of `resched()`) the stack looks like this:

...	
Address of save area with **new** registers and PS	$\leftarrow \quad SP+4$
Address of save area with **old** registers and PS	$\leftarrow \quad SP+2$
return address of `return(OK);`	$\leftarrow \quad SP$

Context Switching: `ctxsw`

All values of the **nine registers** $(R_0, \ldots, R_5, SP, PC, PS)$ **of the old process are saved**:

- $R_0 := *2(SP)$ – where $*x$ is the address of the area that x points to, i.e., the value of R_0 is stored in R_0-position in **old register-save-area**.
- $R_0 := 2(SP)$; $R_0 := R_0 + 2$ – so R_0 points to the R_1-position in old register-save-area.
- R_1 to R_5 are saved in old register-save-area.
- $SP := SP + 2$ – **after return from hibernation** stack pointer should point to stack segment with **parameters of `ctxsw`** since these are popped in calling procedure's translation. So the stack pointer is now **prepared** for later return from hibernation, i.e. SP no longer points to return address of `return(OK)` but to address of `ctxsw`'s arguments.
- Now the prepared stack pointer is saved; the return address of `return(OK)` (now contents of $SP - 2$, this leads to `-(sp)`) is put to old register-save-area; finally `mfps` puts the processor status word PS to old register-save-area.

$\Rightarrow$ Now the values of SP, PC and PS we need when **returning to `return(OK)`** are all **assembled in the old register-save-area**!

Context Switching: `ctxsw`

The result of saving all nine registers:

…		
R_0 ⋮ R_5		
old $SP + 2$	←	saved SP
contents of old SP	←	saved PC
old PS	←	saved PS

Now the **registers of the new process are restored**:

- $R_0 := SP + 4$ – so R_0 holds address of **new register-save-area**
- Now the values of R_1 to R_5 are restored.
- SP is restored (now we have actually switched stacks)
- PS and PC are loaded from new register-save-area and pushed onto the new stack
- R_0 is restored from new register-save-area
- **`rtt`** – this instructions pops PS and PC of the new stack (and thereby SP is reset correctly) all at once!

– only now context switch has taken place –

It is interesting to note that alle processes call **`resched()`** to perform context switching, and **`resched()`** calls `ctxsw`, so all suspended processes resume at the same place – just after the call to `ctxsw`. However, each process has its own stack of procedure calls, so the return from **`resched()`** will take them in various directions.

Context Switching: `ctxsw`

```
/*------------------------------------------------------------------------
/* ctxsw  --  actually perform context switch, saving/loading registers
/*------------------------------------------------------------------------
  / The stack contains three items upon entry to this routine:
  /   SP+4 =>     address of 9 word save area with new registers + PS
  /   SP+2 =>     address of 9 word save area for old registers + PS
  /   SP   =>     return address
  / The saved state consists of: the values of R0-R5 upon entry, SP+2,
  / PC equal to the return address, and the PS (i.e., the PC and SP are
  / saved as if the calling process had returned to its caller).
      .globl      _ctxsw      / declare the routine name global
_ctxsw:                       / entry point to context switch
      mov   r0,*2(sp)         / Save old R0 in old register area
      mov   2(sp),r0          / Get address of old register area
      add   $2,r0             /  in R0; increment to saved pos. of R1
      mov   r1,(r0)+          / Save registers R1-R5 in successive
      mov   r2,(r0)+          /  locations of the old process
      mov   r3,(r0)+          /  register save area.  (r0)+ denotes
      mov   r4,(r0)+          /  indirect reference and, as a side
      mov   r5,(r0)+          /  effect, incrementing r0 to next word.
      add   $2,sp             / move sp beyond the return address,
                              /   as if a return had occurred.
      mov   sp,(r0)+          / save stack pointer
      mov   -(sp),(r0)+       / Save caller's return address as PC
      mfps  (r0)              / Save processor status beyond registers
      mov   4(sp),r0          / Pick up address of new registers in R0
                              / Ready to load registers for the new
                              /   process and abandon the old stack.
      mov   2(r0),r1          / Load R1-R5 and SP from the saved area
      mov   4(r0),r2          /   for the new process.
      mov   6.(r0),r3         / NOTE: dot following a number makes it
      mov   8.(r0),r4         /   decimal; all others are octal
      mov   10.(r0),r5
      mov   12.(r0),sp        / Have now actually switched stacks
      mov   16.(r0),-(sp)     / Push new process PS on new process stack
      mov   14.(r0),-(sp)     / Push new process PC on new process stack
      mov   (r0),r0           / Finally, load R0 from new area
      rtt                     / Load PC, PS, and reset SP all at once
```

4.4 Null Process

- `resched()` resumes that at least one process is available on the ready queue – it does not bother to verify whether ready list is empty, i.e.:
  `resched()` can only switch context from one process to another, so at least one process must always remain on ready queue.
- To ensure that a ready process always exists XINU creates an extra process, the **null process**, when it initializes the system. It has `pid` zero and `pprio` zero; its code contains of an infinite loop.
- Because user processes have all priority greater than 0, the scheduler switches to the null process **only when no user process ready to run** – so the null process is currently running when other processes **deadlock**. But even then the **real time clock interrupt routine** decreases real time clock until new process, if any, get ready to execute when their time expires. These are then put on ready queue, after which `resched()` is called.

4.5 Making A Process Ready

- Making a process **eligible for CPU service** occurs so often that a special procedure has been designed to do so: `ready(pid,resched)`.
- In principle putting a process on the ready list should result in a **call to** `resched()` to make sure that the **process with the highest priority is running**
- However, sometimes this results in a too heavy overhead in execution time when many processes are put on the ready queue (for instance see process wakeup in realtime interrupt processing). Then all these processes are put **in one go** on the ready list **without rescheduling** after which `resched()` is called **once** to make sure that the process with the highest priority is running.
- This construction is made possible by providing `ready(...,...)` with a **boolean** argument `RESCHEDNO`/`RESCHEDYES` determining whether `resched()` is called.

Making A Process Ready

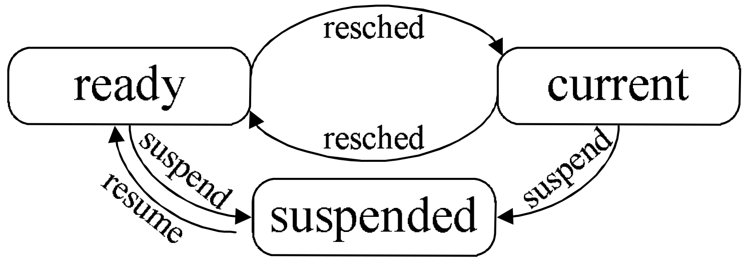
```
/* ready.c - ready */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*------------------------------------------------------------------------
 * ready  --  make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
int   ready (pid, resch)
      int   pid;             /* id of process to make ready      */
      int   resch;           /* reschedule afterward?            */
{
      register struct pentry  *pptr;

      if (isbadpid(pid))
            return(SYSERR);
      pptr = &proctab[pid];
      pptr->pstate = PRREADY;
      insert(pid,rdyhead,pptr->pprio);
      if (resch)
            resched();
      return(OK);
}
```

5 More Process Management: Resumption, Suspension, Process Creation & Termination

5.1 Process Suspension and Resumption

- To temporarily stop a process from executing and later start it explicitly, a new process state is introduced: **suspended animation**, abbreviated to **suspended**.
- This process state is, e.g., used in paging systems, to install pages referenced in user programs in **primary** (code) memory from **secondary** (disk) memory. During this operation the process is **suspended**.

Process Suspension and Resumption



- `suspend` removes a process from the ready list or stops a currently executing process; parameter: process id.
- `resume` moves a process from the suspended state to the ready list explicitly.
- The currently executing process can suspend itself by calling `suspend(getpid())`.
- `create` creates a new, independent process **as if it had been stopped during running**, so `ctxsw` can return to it; this mechanism is called **pseudocall**.

Process Suspension and Resumption:

resume

```
/* resume.c - resume */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 * resume  --  unsuspend a process, making it ready; return the priority
 *------------------------------------------------------------------------
 */
SYSCALL resume(pid)
        int     pid;
{
        STATWORD ps;
        struct  pentry  *pptr;          /* pointer to proc. tab. entry  */
        int     prio;                   /* priority to return           */

        disable(ps);
        if (isbadpid(pid) || (pptr= &proctab[pid])->pstate!=PRSUSP) {
                restore(ps);
                return(SYSERR);
        }
        prio = pptr->pprio;
        ready(pid, RESCHYES);
        restore(ps);
        return(prio);
}
```

Process Suspension and Resumption: suspend

```
/* suspend.c - suspend */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  suspend  --  suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
SYSCALL suspend(pid)
        int     pid;                    /* id of process to suspend     */
{
        STATWORD ps;
        struct  pentry  *pptr;          /* pointer to proc. tab. entry  */
        int     prio;                   /* priority returned            */

        disable(ps);
        if (isbadpid(pid) || pid==NULLPROC ||
         ((pptr= &proctab[pid])->pstate!=PRCURR && pptr->pstate!=PRREADY)) {
                restore(ps);
                return(SYSERR);
        }
        if (pptr->pstate == PRREADY) {
                pptr->pstate = PRSUSP;
                dequeue(pid);
        }
        else {
                pptr->pstate = PRSUSP;
                resched();
        }
        prio = pptr->pprio;
        restore(ps);
        return(prio);
}
```

The need for turning interrupts off when accessing the central data structures of Xinu

- `resume` calls `resched`, and hence changes the **process table** (i.e., the entry for the resumed process), the *q*-**structure** (since within it the ready list is implemented).
- What happens when another process would be simultaneously changing the process table? **The final state of the process table is not clear!** This situation is, in general, called **race condition**:

```
{ x = 0 }    x := 1 || x := 2     { x = ? }
```

- To prevent such situation from happening: this uncertainly is prohibited by **disabling interrupts while accessing the process table** (or other certain data structure of the operating system).

5.2 System calls

- The precaution `resume` takes to verify that a process is legal, make it a **general purpose routine** that can be invoked by (unexperienced) users. So `resume` is a **system call**.
- **System calls define the exterior of the operating system** by providing an **interface** through which the user accesses all system services. They should protect the internal system from illegal use.
- There is one process table in the system, shared by all its processes. **How can a process be sure** that no other process is trying to change the process table **at the same time**?
  - It should not call `resched()`, because rescheduling switches control to another process and this change the process table.
  - The system should not react to interrupts, since interrupt routines can call `resched` as well!

System calls

- How can `resched()` be called? See process table on page 128: by calling `sleep`/`wakeup`, `receive`/`send`, `wait`/`signal`, `suspend`/`resume`.
- However this kind of call presupposes that `resched()` is called **in parallel to the current process**.
- Interleaving the instructions of this concurrent (parallel) process in between the instructions of the concurrent process, implies that, e.g., **the real-time clock procedure has interrupted** the current process.
- However, this is impossible when the current process is **deaf to interrupts**. For then the real-time clock interrupt is temporarily postponed.
- To prevent Interrupts `resume` invokes procedure `disable(ps)` to set the process priority high enough (`034i`) to prevent interrupts. To restore the situation before `disable(ps)` was called, one calls `restore(ps)`. But does this necessarily imply that the **process reacts to interrupts again**?

System calls

- As we will see in chapter 9, **high level interrupt routines** enable the occurrence of **other interrupts** by calling `resched()` if they need the CPU to switch temporarily to a new context, i.e., a process with interrupt enabled.

  (E.g., to make room in a character buffer when reading characters from a terminal, see chapters 11 and 12, or to read in characters to an output buffer when the output buffer is empty and characters should be printed.)

- To **prevent inconsistencies**, the shared data structures should be **left in a valid sate** before calling **any routines that switches context**. For example, the following states are **invalid**:
  - If `resched()` is called in between `pptr->pstate != PRCURR` and `pptr->pstate != PRREADY`, this might result in setting state to `PRCURR`
  - `resched` occurring in between `dequeue(pid)` and `pptr->pstate = PRSUSP` could result in `enqueue(pid)`
  - Interrupts occurring in between `pptr->pstate = PRSUSP` and `resched` might result in `pptr->pstate = PRCURR`

5.3 Process Termination

- `suspend` freezes processes, but leaves them in the system so they can be resumed later. System call `kill` stops a process immediately and **removes it from the system completely**.
- First, `kill` checks its argument `pid` to ensure it corresponds to a **valid active process**. Then it **decrements** `numproc`, it **frees the stack space** of this process (by calling procedure `freestk`), unlinks the process from the **ready list** (`dequeue`) and frees the **process table entry** (by assigning its state field `PRFREE`).
- Since the process no longer appears on the ready list, it will **never regain control of CPU**.
- A process can **kill itself** by calling `kill(getpid())`. Its state is made **PRFREE** and `resched()` is called – this process can not be started again because it does no longer appear on the ready list.
- If the last active process is killed, the procedure `xdone()` is called.

Process Termination: kill

```
/* kill.c - kill */

#include "conf.h"
#include "kernel.h"
#include "proc.h"
#include "sem.h"
#include "mem.h"

/*------------------------------------------------------------------------
 * kill  --  kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
SYSCALL kill(pid)
        int     pid;                    /* process to kill              */
{
        struct  pentry  *pptr;          /* points to proc. table for pid*/
        char    ps;                     /* saved processor status       */

        disable(ps);
        if (isbadpid(pid) || (pptr= &proctab[pid])->pstate==PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        freestk(pptr->pbase, pptr->pstklen);
        switch (pptr->pstate) {
          case PRCURR:  pptr->pstate = PRFREE;  /* suicide */
                        resched();

          case PRWAIT:  semaph[pptr->psem].semcnt++;
                                                /* fall through */
          case PRSLEEP:
          case PRREADY: dequeue(pid);

          default:      pptr->pstate = PRFREE;
        }
        restore(ps);
        return(OK);
}
```

5.4 Kernel Declaration

- The code in `kernel.c` defines in-line procedures `disable` and `restore` used above, as well as other variables and symbolic constants used throughout XINU.
- Constants like `INITSTK`, `INITPRIO` and `INITNAME` are used on page 196 when creating the null process:

  > *Once initialization is complete and a process has been created to execute the user's program, the null process just falls into an infinite loop, giving* ***`resched`*** *a process to schedule when no user processes are ready to run.*

Kernel Declarations

```
/* kernel.h - disable, enable, halt, restore, isodd, min */

/* Symbolic constants used throughout Xinu */

typedef char        Bool;           /* Boolean type                */
#define FALSE       0               /* Boolean constants           */
#define TRUE        1
#define NULL        (char *)0       /* Null pointer for linked lists*/
#define SYSCALL     int             /* System call declaration     */
#define LOCAL       static          /* Local procedure declaration */
#define INTPROC     int             /* Interrupt procedure  "      */
#define PROCESS     int             /* Process declaration         */
#define RESCHYES    1               /* tell ready to reschedule    */
#define RESCHNO     0               /* tell ready not to resch.    */
#define MININT      0100000         /* minimum integer (-32768)    */
#define MAXINT      0077777         /* maximum integer             */
#define SP          6               /* reg. 6 is stack pointer     */
#define PC          7               /* reg. 7 is program counter   */
#define PS          8               /* proc. status in 8th reg. loc */
#define MINSTK      40              /* minimum process stack size  */
#define NULLSTK     300             /* process 0 stack size        */
#define DISABLE     0340            /* PS to disable interrupts    */
#define OK           1              /* system call ok              */
#define SYSERR      -1              /* system call failed          */

/* Initialization constants */

#define INITARGC    1               /* initial process arcs        */
#define INITSTK     500             /* initial process stack       */
#define INITPRIO    20              /* initial process priority    */
#define INITNAME    "main"          /* initial process name        */
#define INITRET     userret         /* processes return address    */
#define INITPS      0               /* initial process PS          */
#define INITREG     0               /* initial register contents   */
#define QUANTUM     10              /* clock ticks until preemption */

/* Miscellaneous utility inline functions */

#define isodd(x)        (01&(int)(x))
#define disable(ps)     asm("mfps ~ps");asm("mtps $0340")
#define restore(ps)     asm("mtps ~ps") /* restore interrupt status    */
#define enable()        asm("mtps $000")/* enable interrupts           */
#define pause()         asm("wait")     /* machine "wait for interr."  */
#define halt()          asm("halt")     /* machine halt instruction    */

extern  int     rdyhead, rdytail;
extern  int     preempt;
```

5.5 Process Creation

- System call `create` creates a new, independent process. So it lays down an **exact image of the process as if it has been stopped while running**, so `ctxsw` in `resched` can switch to it; it fills a free (unused) slot in the **process table** and allocates space for the **process' stack**. The initial process stack looks like this:

...
arg_n
...
arg_1
returnaddress

- The return address is used **after** the created process has terminated, so it contains the address of procedure `userret` which calls `kill(getpid())`.
- We refer to the initial process stack as a **pseudo-call** because `create` carefully pushes values on it to **simulate a procedure call** (in C: **arguments and a return address**). The process begins executing the code, obeying the calling conventions for accessing arguments and allocating local variables – so **it behaves exactly as if it had been called from another procedure**.

Process Creation: create

```
/*------------------------------------------------------------------------
 *  create  -  create a process to start running a procedure
 *------------------------------------------------------------------------
 */
SYSCALL create(procaddr,ssize,priority,name,nargs,args)
        int     *procaddr;              /* procedure address            */
        int     ssize;                  /* stack size in words          */
        int     priority;               /* process priority > 0         */
        char    *name;                  /* name (for debugging)         */
        int     nargs;                  /* number of args that follow   */
        int     args;                   /* arguments (treated like an   */
                                        /* array in the code)           */
{
        int     pid;                    /* stores new process id        */
        struct  pentry  *pptr;          /* pointer to proc. table entry */
        int     i;
        int     *a;                     /* points to list of args       */
        int     *saddr;                 /* stack address                */
        char    ps;                     /* saved processor status       */
        int     INITRET();
        disable(ps);
        ssize = roundew(ssize);
        if ( ssize < MINSTK || ((saddr=getstk(ssize)) == SYSERR ) ||
                (pid=newpid()) == SYSERR || isodd(procaddr) ||
                priority < 1 ) {
                restore(ps);
                return(SYSERR);
        }
        numproc++;
        pptr = &proctab[pid];
        pptr->pstate = PRSUSP;
        for (i=0 ; i<PNMLEN && (pptr->pname[i]=name[i])!=0 ; i++)
                ;
        pptr->pprio = priority;
        pptr->pbase = (short)saddr;
        pptr->pstklen = ssize;
        pptr->plimit = (short)(saddr - ssize + 1);
        pptr->pargs = nargs;
        for (i=0 ; i<PNREGS ; i++)
                pptr->pregs[i]=INITREG;
        pptr->pregs[PC] = pptr->paddr = (short)procaddr;
        pptr->pregs[PS] = INITPS;
        a = (&args) + (nargs-1);        /* point to last argument       */
        for ( ; nargs > 0 ; nargs--)    /* machine dependent; copy args */
                *saddr-- = *a--;        /* onto created process' stack  */
        *saddr = (int)INITRET;          /* push on return address       */
        pptr->pregs[SP] = (int)saddr;
        restore(ps);
        return(pid);
}
```

Process Creation: newpid and userret

```
/*------------------------------------------------------------------------
 * newpid  --  obtain a new (free) process id
 *------------------------------------------------------------------------
 */
LOCAL   newpid()
{
        int     pid;                    /* process id to return         */
        int     i;

        for (i=0 ; i<NPROC ; i++) {     /* check all NPROC slots        */
                if ( (pid=nextproc--) <= 0)
                        nextproc = NPROC-1;
                if (proctab[pid].pstate == PRFREE)
                        return(pid);
        }
        return(SYSERR);
}


/* userret.c - userret */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 * userret  --  entered when a process exits by return
 *------------------------------------------------------------------------
 */
userret()
{
        kill( getpid() );
}
```

5.6 Utility Procedures

- Following code shows the implementation of the system calls: `getpid`, `getprio` and `chprio`
- `getpid` allows a process to obtain its process id – `userret` shows one reason why a procedure needs to know the process id of the process executing it.
- `getprio` allows a process to obtain a process' scheduling priority, and `chprio` allows a process to change a process' priority.

Utility Procedures: getprio and getpid

```
/* getprio.c - getprio */
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 * getprio -- return the scheduling priority of a given process
 *------------------------------------------------------------------------
 */
SYSCALL getprio(pid)
        int     pid;
{
        struct  pentry  *pptr;
        char    ps;

        disable(ps);
        if (isbadpid(pid) || (pptr = &proctab[pid])->pstate == PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        restore(ps);
        return(pptr->pprio);
}

/* getpid.c - getpid */
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 * getpid  --  get the process id of currently executing process
 *------------------------------------------------------------------------
 */
SYSCALL getpid()
{
        return(currpid);
}
```

6 Process Coordination

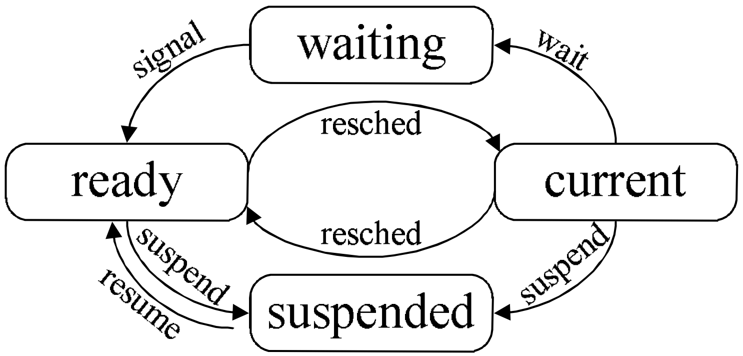
- Process coordination is required to **synchronize the actions of processes** and to **cooperate in sharing resources**.
- In the first Chapter we discussed the **producer-consumer-problem** and **mutual exclusion**.
- The simplest way to synchronize process coordination is provided by **semaphores**:
  - Each semaphore s consists of an **integer count**. The system call `wait(`s`)` is used to **decrement** s, system call `signal(`s`)` is used to **increments** s.
  - If s becomes **negative** when a process executes `wait(`s`)`, that process is **delayed**. Each time `signal(`s`)` is called **one delayed process is ready to run again**.

6.1 Low-level Coordination Techniques

- XINU procedures `ready(`s`)` and `resume(`s`)` employ code for process coordination through the **disabling of interrupts**, because when the process table needs to be updated **only one process** is allowed to do so at every moment.
- However, disabling interrupts has an undesirable effect: it **all but one process**, and **limits** what that process can do (e.g., I/O operations). We need process coordination primitives enabling the coordination of a given subset of processes **without stopping the other processes**, and **without limiting what these processes can do**.

6.2 Implementation of High-Level Coordination Primitives

- When a process needs to **wait** for some condition to become true, the system places it on a **list associated with semaphore** s. So each semaphore has its own **list of waiting processes**.
- To **delay the current process**, `wait(`s`)` enqueues it on the list for s, and calls `resched` allowing another process to run.
- `signal(`s`)` checks that list whenever it is called. When the list of waiting processes associated with semaphore s is **nonempty**, signal **reactivates exactly one process** by moving it back to the ready list.
- This leads to a new process state called `PRWAIT`:



6.2.1 Semaphore Data Structures

- Semaphore info is kept in the **global semaphore table** `SEMAPH`, initialized at system initialization and consisting of `sentry` structures:

```
/* sem.h - isbadsem */

#ifndef NSEM
#define NSEM            45      /* number of semaphores, if not defined */
#endif

#define SFREE   '\01'           /* this semaphore is free               */
#define SUSED   '\02'           /* this semaphore is used               */

struct  sentry  {               /* semaphore table entry                */
        char    sstate;         /* the state SFREE or SUSED             */
        short   semcnt;         /* count for this semaphore             */
        short   sqhead;         /* q index of head of list              */
        short   sqtail;         /* q index of tail of list              */
};
extern  struct  sentry  semaph[];
extern  int     nextsem;

#define isbadsem(s)     (s<0 || s>=NSEM)
```

Implementation of High-Level Coordination Primitives

- `wait(`s`)` is a system procedure courtesy of the interface of XINU. It **decreases** the count of s. If it remains **nonnegative** (≥ 0) wait **returns to the caller**. Otherwise, the calling process is enqueued on the list for s, its state is changed to `PRWAIT`, and `resched` is called to switch to a ready process.
- The call to `signal(`s`)` moves one waiting process back to the ready list, it becomes eligible for CPU use, and is eventually executed.
- From the point of the waiting process, its last action was calling `ctxsw`. Then this call to `ctxsw` returns to `resched`, that returns to `wait`, and that returns to where `wait` had been called!
- A nonnegative semaphore count means the queue is empty, a semaphore count of negative n means the queue contains n waiting processes.

High-Level Coordination: `wait`

```
/* wait.c - wait */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * wait  --  make current process wait on a semaphore
 *------------------------------------------------------------------------
 */
SYSCALL wait(sem)
        int     sem;
{
        char    ps;
        register struct sentry  *sptr;
        register struct pentry  *pptr;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if (--(sptr->semcnt) < 0) {
                (pptr = &proctab[currpid])->pstate = PRWAIT;
                pptr->psem = sem;
                enqueue(currpid,sptr->sqtail);
                resched();
        }
        restore(ps);
        return(OK);
}
```

High-Level Coordination: `signal`

```
/* signal.c - signal */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * signal  --  signal a semaphore, releasing one waiting process
 *------------------------------------------------------------------------
 */
SYSCALL signal(sem)
register int    sem;
{
        register struct sentry  *sptr;
        char    ps;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ((sptr->semcnt++) < 0)
                ready(getfirst(sptr->sqhead), RESCHYES);
        restore(ps);
        return(OK);
}
```

Semaphore Creation and Deletion

- The system **preallocates** the head and tail nodes **for each semaphore list** in the q-structure at system initialization time. System calls `screate` and `sdelete` **allocate and delete semaphores**.

```
/* screate.c - screate, newsem */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * screate  --  create and initialize a semaphore, returning its id
 *------------------------------------------------------------------------
 */
SYSCALL screate(count)
        int     count;                  /* initial count (>=0)          */
{
        char    ps;
        int     sem;

        disable(ps);
        if ( count<0 || (sem=newsem())==SYSERR ) {
                restore(ps);
                return(SYSERR);
        }
        semaph[sem].semcnt = count;
        /* sqhead and sqtail were initialized at system startup */
        restore(ps);
        return(sem);
}
```

Semaphore Creation and Deletion: newsem

```
/*------------------------------------------------------------------------
 * newsem  --  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
LOCAL   newsem()
{
        int     sem;
        int     i;

        for (i=0 ; i<NSEM ; i++) {
                sem=nextsem--;
                if (nextsem < 0)
                        nextsem = NSEM-1;
                if (semaph[sem].sstate==SFREE) {
                        semaph[sem].sstate = SUSED;
                        return(sem);
                }
        }
        return(SYSERR);
}
```

Semaphore Creation and Deletion: `sdelete`

```
/* sdelete.c - sdelete */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * sdelete  --  delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
SYSCALL sdelete(sem)
        int     sem;
{
        char    ps;
        int     pid;
        struct  sentry  *sptr;          /* address of sem to free       */

        disable(ps);
        if (isbadsem(sem) || semaph[sem].sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        sptr->sstate = SFREE;
        if (nonempty(sptr->sqhead)) {   /* free waiting processes       */
                while( (pid=getfirst(sptr->sqhead)) != EMPTY)
                        ready(pid,RESCHNO);
                resched();
        }
        restore(ps);
        return(OK);
}
```

7 Message Passing

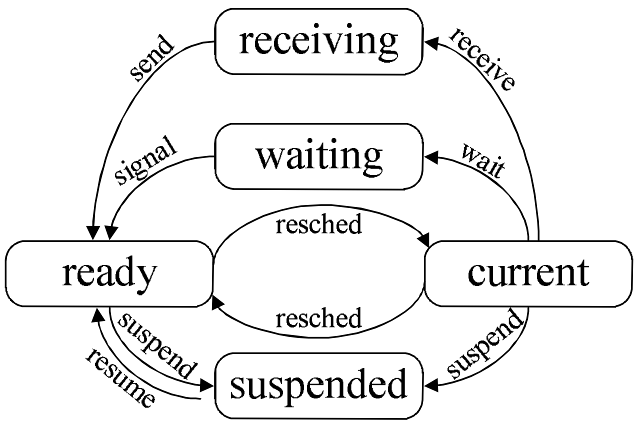
- **Inter-process communication** is an important mechanism in operating systems. Example from UNIX: shell-pipeline – IPC between user processes
- Several mechanisms exist:
  - **semaphores** for coordination/synchronization (see chapter 6)
  - **monitors**: mechanism for obtaining mutual exclusion; high-level programming language constructs, similar to some kind of abstract data type or object (no ordinary system call, see [Hoa74], [BrinchHan75])
  - data transmission by using **shared variables**
  - **message passing**: synchronization and data transmission

7.1 Message Passing in Xinu

- Message passing is a form of **inter-process communication** and **process synchronization and -coordination** (alternative: shared variables).
- Advantage to synchronization by using semaphores: it can be **asynchronous**!
- XINU's implementation is using system calls `send` and `receive`.
- Several variants of message passing are possible:
  - **Direct** message passing
  - Message sending and receiving can be **blocking** or **unblocking**
  - **Rendezvous**: send and receive are blocking
  - **Capacity** of the binding (buffer size): what would happen, if buffer is full?
  - Messages of **determined** or **variable size**
  - Is **more than one receiver** possible?
  - **Specified** sending/receiving process
  - ....

Message Passing in Xinu

- In Xinu there are two forms of message passing: either process-to-process (**direct**) or the message are left at **rendezvous** points (see chapter 14).
- System calls in Xinu:
  - **`send`** (asynchronous) takes a message and a process id as arguments and delivers the message to the specified process,
  - **`receive`** (synchronous) waits for the message to arrive,
  - **`recvclr`** (asynchronous) never waits for a message to arrive.
- A new process state is needed: **receiving** (`PRREC`).



Message Passing in Xinu

- Receiving has a **buffer** of size **1** (= one word), so only the **first message** is received and **all other are lost** because sender does not block.
- **Storage place** for a message: in the **process table** entry – not in the **sender's memory** because the sending process might exit before the message is received, but neither in the **recipient's memory** because it poses a security threat!

```
/* proc.h   see page 55 */


...
struct  pentry {
        char     pstate;   /* process state                  */
        ...
        short    pmsg;     /* 1 message sent to this process */
        short    phasmsg;  /* nonzero => msg is valid        */
        ...
};
...
```

7.2 Implementation of `send`

- During `send`, interrupts are disabled. Uses the process id to access corresponding process table entry
- **Errors** occur when an **invalid process id** is used or the receiving **buffer is full**.
- Else sending a message consists of **passing the message** and setting the **flag** (`phasmsg`) – if the recipient is **waiting** for a message, then reschedule by calling `ready()`.

Implementation of send

```
/* send.c - send */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  send  --  send a message to another process
 *------------------------------------------------------------------------
 */
SYSCALL send(pid, msg)
int     pid;
int     msg;
{
        struct  pentry  *pptr;          /* receiver's proc. table addr. */
        char    ps;

        disable(ps);
        if (isbadpid(pid) || ( (pptr= &proctab[pid])->pstate == PRFREE)
           || pptr->phasmsg) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;               /* deposit message              */
        pptr->phasmsg = TRUE;
        if (pptr->pstate == PRRECV)     /* if receiver waits, start it  */
                ready(pid, RESCHYES);
        else if (pptr->pstate == PRTRECV) {
                unsleep(pid);
                ready(pid, RESCHYES);
        }
        restore(ps);
        return(OK);
}
```

7.3 Implementation of `Receive`

- **Asynchronous receiving** is implemented by `recvclr`: if the process has a message, `recvclr` returns it, else it returns `OK`:

```
/* recvclr.c - recvclr */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  recvclr  --  clear messages, returning waiting message (if any)
 *------------------------------------------------------------------------
 */
SYSCALL recvclr()
{
        char    ps;
        int     msg;

        disable(ps);
        if ( proctab[currpid].phasmsg ) {       /* existing message?    */
                proctab[currpid].phasmsg = FALSE;
                msg = proctab[currpid].pmsg;
        } else
                msg = OK;
        restore(ps);
        return(msg);
}
```

Implementation of `Receive`

- **Synchronous receiving** is implemented by `receive`
  – difference to asynchronous receive: if the process has no message, the receiving process changes into the receiving state and calls **`resched`**.

```
/* receive.c - receive */

#include <conf.h> #include <kernel.h> #include <proc.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return it
 *------------------------------------------------------------------------
 */
SYSCALL receive() {
        struct  pentry  *pptr;
        int     msg;
        char    ps;

        disable(ps);
        pptr = &proctab[currpid];
        if ( !pptr->phasmsg ) {         /* if no message, wait for one  */
                pptr->pstate = PRRECV;
                resched();
        }
        msg = pptr->pmsg;               /* retrieve message             */
        pptr->phasmsg = FALSE;
        restore(ps);
        return(msg);
}
```

8 Memory Managment

- The operating system keeps track of **location** and **size** of the **available free space**, **allocating** it on demand, and **recovering** it when processes complete.
- In large operating system different from XINU, the demand for memory by processes may **exceed** the total amount of core/primary memory available, hence the system must **multiplex** real memory among processes waiting for use it.
- Memory multiplexing can take form of:
  - **Swapping**: entire processes are written to secondary (disk) storage when they are not using the CPU.
  - **Paging**: each program is divided into small fixed-size pieces called **pages**. The system keeps only the most recently referenced pages in primary storage (i.e., main memory).
- Programmers **don't notice** swapping and paging because the system hides it by appropriate hardware.

Memory Management

- In addition to pages, paged system supply each process with its own independent address space. This **virtual** address space can be **larger than real memory** because the paging system keeps the virtual image on disk, moving **only the small subset** of referenced pages to main memory
- When a process references location i in its address space, the hardware consults **memory mapping tables** to see whether the page with location i resides in main memory or not.
- If not, the operating system **suspends** the process, loads the requested page from secondary to primary memory, making room for it when needed, and when it's there, **resumes execution** of the process.
- Memory management requires hardware support. When done well, the operating system can portion memory in such a way that hardware **prevents** a process from **reading or writing memory allocated to another process**. This constitutes an especially important aspect for security.

8.1 Memory Management on the 11/2

- The whole program resides in **main memory**!
- Storage layout **when Xinu begins**:

↓ 0	↓_etext	↓_edata	↓_end
text	**data**	**bss**	**free space** ⟵ · · · **stack**

8.2 Dynamic Memory Requirements in Xinu

- Storage layout **during execution**:

↓ 0	↓_etext	↓_edata	↓_end				
text	**data**	**bss**	**heap**	**free**	**stack** # 3	**stack** # 2	**stack** # 1

8.3 Low-Level Memory Management Procedure

- System calls `getstk` and `freestk` are used by `create` and `kill`:
  - in `create`: ...`saddr = getstk(ssize)`... – see page 75
  - in `kill`: ...`freestk(pptr->pbase, pptr->pstklen)`... – see page 77
- `getstk` **obtains a block of memory** and returns its highest address, at which the stack of the created process initially starts.
- `create` records **size and location** of the allocated space in the process table entry of a process – later `kill` returns this stack space to **free memory** space by calling `freestk` which has two arguments: highest **address** of block being returned and its **size**.
- XINU guarantees that the stack space allocated to a process is **released** at process exit.

Low-Level Memory Management Procedure

- For the **heap no deallocation procedures** are built into Xinu– deallocation must be done on a process-by-process basis:

  A process must release storage that it allocates from the heap before it exit by itself (i.e., this doesn't happen automatically).

- **Allocation** on the heap is done by procedure `getmem`, **deallocation** on the heap is done by procedure `freemem`. Both system calls reference blocks by their **lowest address**.

- `freestk` is implemented using `freemem`:

```
#define freestk(p,len) freemem((unsigned)(p) - (unsigned)(roundew(len)) \
                                  + (unsigned)sizeof(int), roundew(len))
```

  This converts counting an address from the top to counting from its bottom address:

...	1001	1000	999	998	997	996	...
		↑	length 4		↑		

8.4 The Location of Allocated Storage

- **Separating** heap from the stack storage works best when a **single process** executes: if (single) **stack overflow** occurs, the process runs into the free space located between stack and heap:

heap	**free space** $\longleftarrow$	**stack**

- A problem arises when more than one process executes: **Stack overflow** that causes **corruption of data**... an exercise: **suggest a solution**!

8.5 Implementation Of Xinu Memory Management

- The free memory `freemem` is organized as follows: **Blocks of free memory** are **linked together** on a **list with global variable `memlist`** pointing to the first free block.
- Blocks on this free list are ordered by increasing addresses. While on the free list, each block contains in its first 2 words a **pointer `*mnext` to the next free block** and the **size `mlen` of the current block**:

```
struct  mblock  {
        struct  mblock  *mnext;
        unsigned mlen;
        };
```

- `mnext` points to the next block or contains `NULL`, `mlen` contains the length of the current block in bytes **including the two-word-header**! So only blocks of **two words or more** can be linked into the free list, i.e., of four bytes or more, making the number of bytes a multiple of four.
- Record `memlist` has the same form as all other blocks on the free list: it's of type `struct mblock`!

Implementation Of Xinu Memory Management

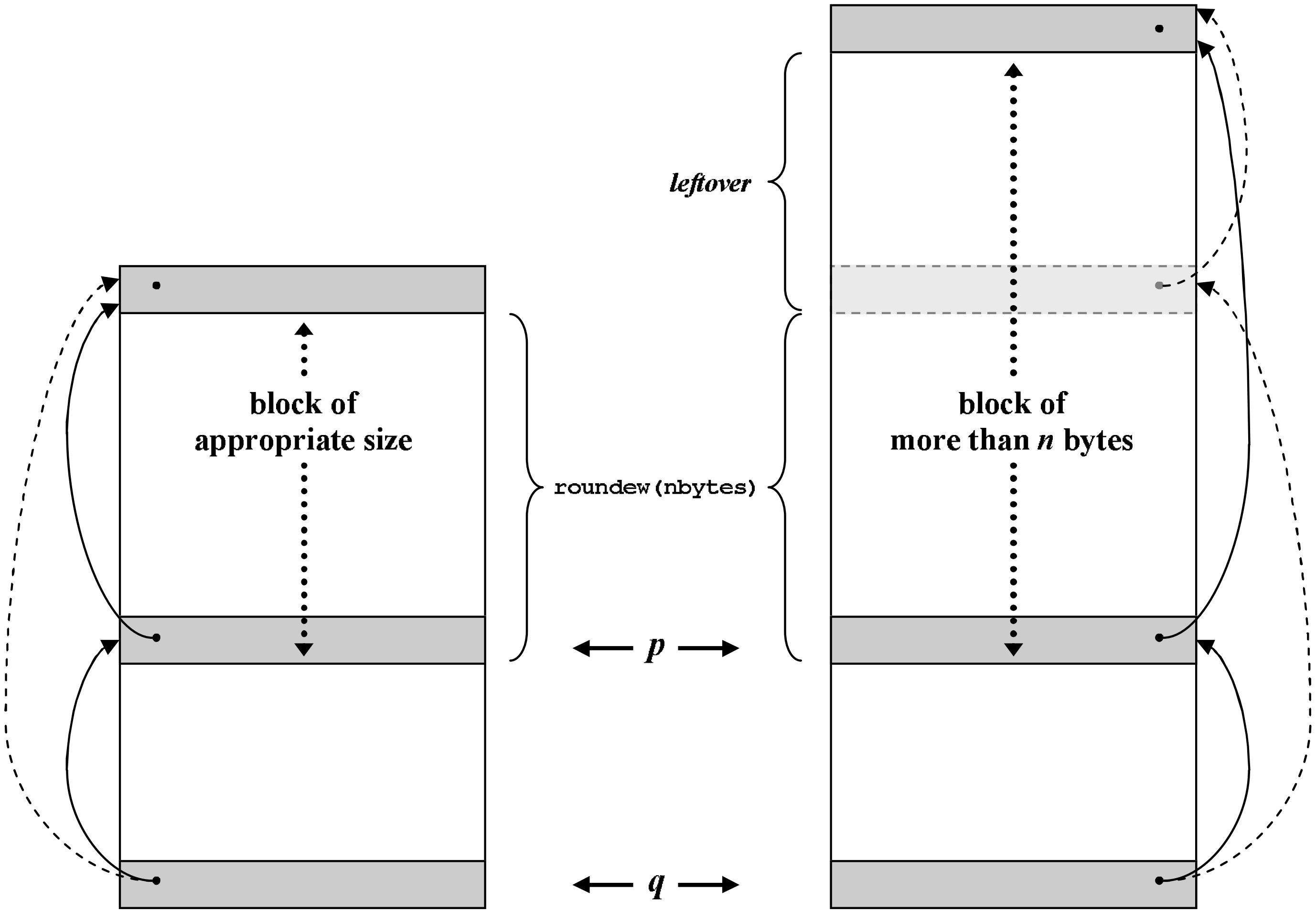
```
/* mem.h - freestk, roundew, truncew */

/*----------------------------------------------------------------------
 * roundew, truncew - round or truncate address to next even word
 *----------------------------------------------------------------------
 */
#define roundew(x)      (int *)( (3 + (int)(x)) & (-3) )
#define truncew(x)      (int *)( ((int)(x)) & (-3) )
                       /* round or truncate an even number of words    */
                       /* by making number of bytes a multiple of four */

/*----------------------------------------------------------------------
 *  freestk  --  free stack memory allocated by getstk
 *----------------------------------------------------------------------
 */
#define freestk(p,len)  freemem((unsigned)(p)                       \
                                - (unsigned)(roundew(len))          \
                                + (unsigned)sizeof(int),            \
                                roundew(len) )

struct  mblock  {
        struct  mblock  *mnext;
        unsigned mlen;
        };
extern  struct  mblock  memlist;        /* head of free memory list     */
extern  int     *maxaddr;               /* max memory address           */
extern  int     etext;                  /* address beyond text segment  */
extern  int     edata;                  /* address beyond data segment  */
extern  int     end;                    /* address beyond loaded memory */
```

8.5.1 Allocating Heap Storage

- Procedure `*getmem(nbytes)` allocates heap storage returning the **lowest** integer address. `getmem` searches the memory list to find the **first block of memory large enough** to contain `roundew(nbytes)` bytes of free storage.
- Because the list of free blocks is **singly-linked**, **two pointers** p and q with q trailing behind p, are used to search for a block of appropriate size. When p points to a block of suitable size, q points to its **predecessor** on the list (possibly its head, `memlist`).

Allocating Heap Storage

- Two cases may occur while searching:



Allocating Heap Storage

```
/* getmem.c - getmem */

#include <conf.h>
#include <kernel.h>
#include <mem.h>

/*------------------------------------------------------------------------
 * getmem  --  allocate heap storage, returning lowest integer address
 *------------------------------------------------------------------------
 */
SYSCALL *getmem(nbytes)
        unsigned nbytes;
{
        char    ps;
        struct  mblock  *p, *q, *leftover;

        disable(ps);
        if (nbytes==0 || memlist.mnext==NULL) {
                restore(ps);
                return( (int *)SYSERR);
        }
        nbytes = (unsigned) roundew(nbytes);
        for (q= &memlist,p=memlist.mnext ; p!=NULL ; q=p,p=p->mnext)
                if ( p->mlen == nbytes) {
                        q->mnext = p->mnext;
                        restore(ps);
                        return( (int *)p );
                } else if ( p->mlen > nbytes ) {
                        leftover = (struct mblock *)( (unsigned)p + nbytes );
                        q->mnext = leftover;
                        leftover->mnext = p->mnext;
                        leftover->mlen = p->mlen - nbytes;
                        restore(ps);
                        return( (int *)p );
                }
        restore(ps);
        return( (int *)SYSERR );
}
```

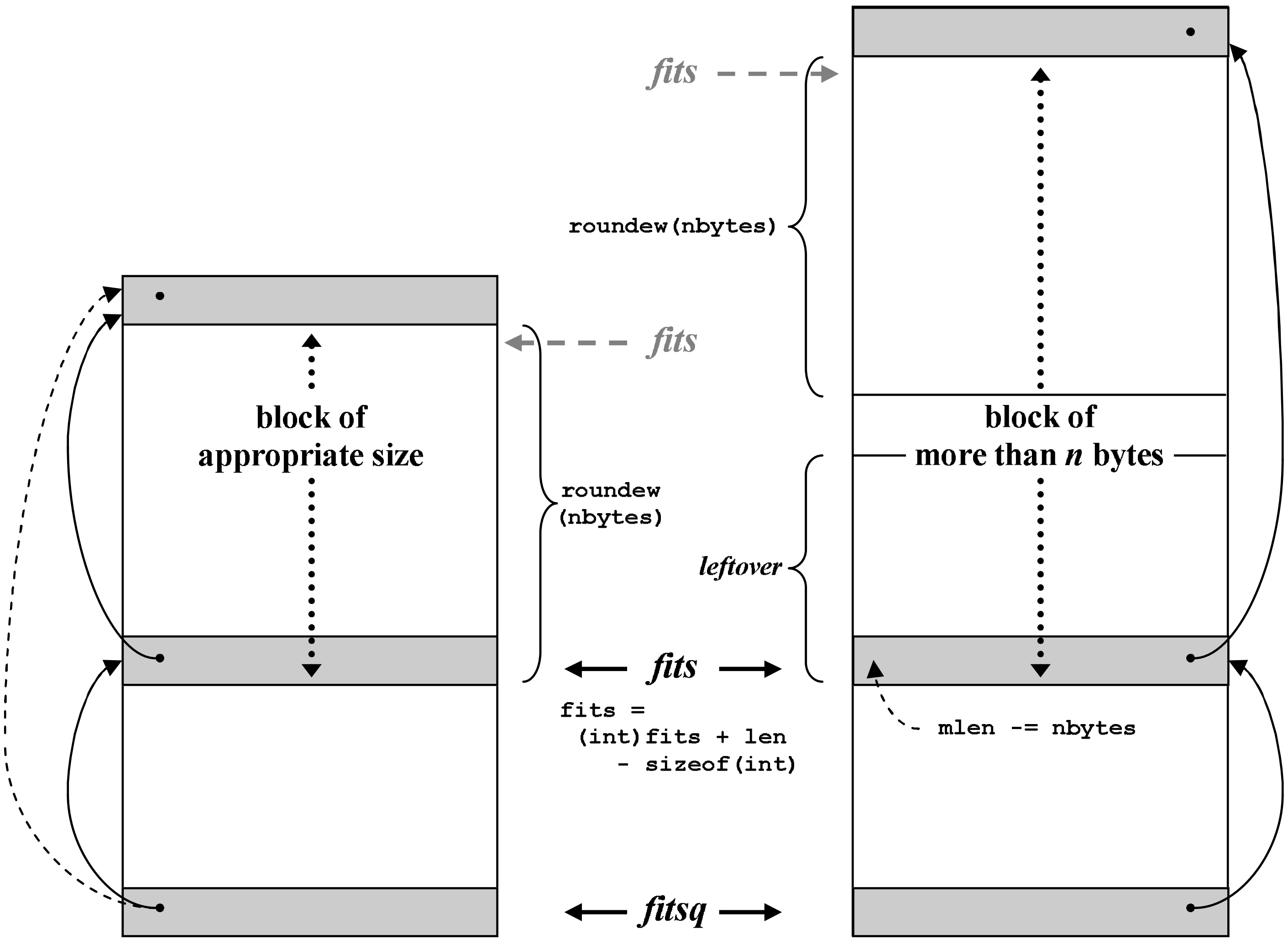
8.5.2 Allocating Stack Storage

- `getstk` must search the entire list of free blocks because the list is kept in ascending order by block address, and the desired block is the one with the highest address satisfying the request.
- `fits` and `fitsq` record the values of p and q each time a block satisfies the request; when search is completed: `fits` points to the last free block that satisfied the request, and `fitsq` points to its predecessor.
- `*getstk(nbytes)` makes full use of C's powerful `for`-statement, see page 382 for its semantics – `for(S1; exp; S2) { S3 }` corresponds in PASCAL with

```
S1; while exp <> 0 do begin S3; S2 end
```

Allocating Stack Storage

- Two cases may occur while searching:

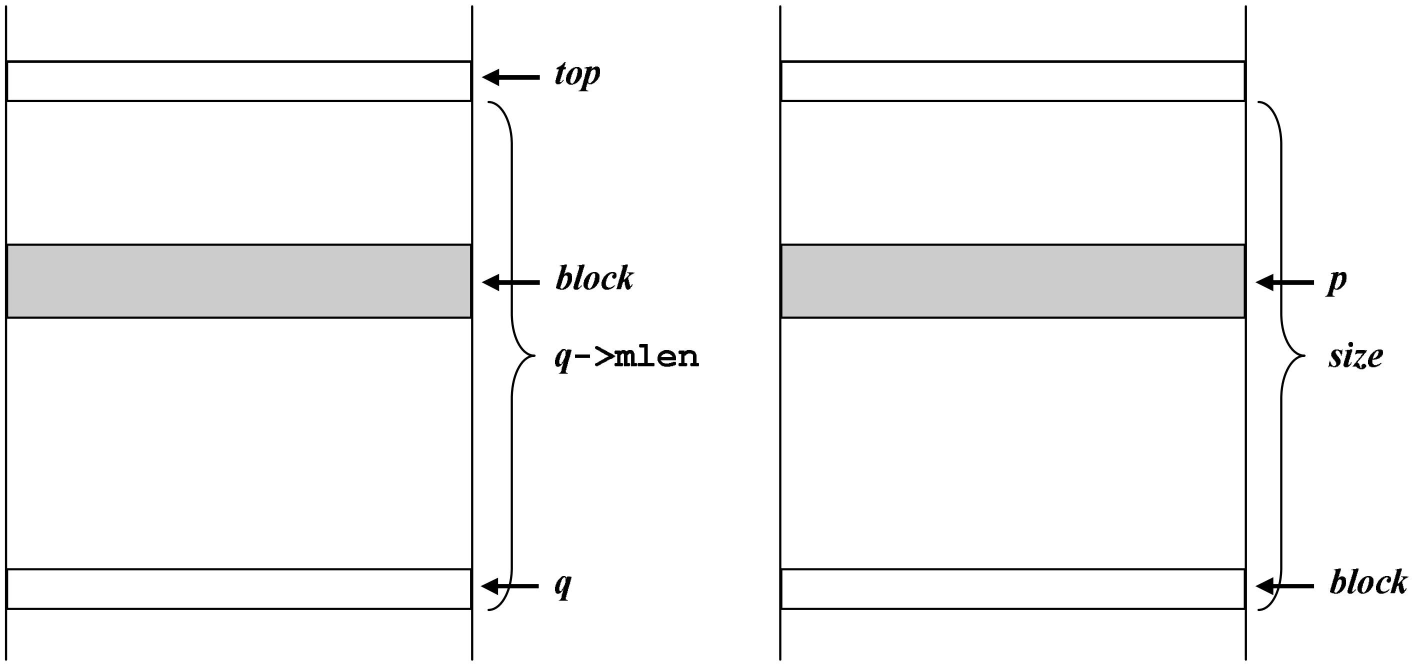

fits
roundew(nbytes)
fits
block of
appropriate size
roundew
(nbytes)
block of
more than *n* bytes
leftover
fits
fits =
(int)fits + len
- sizeof(int)
mlen -= nbytes
fitsq

Allocating Heap Storage

```
/*------------------------------------------------------------------------
 * getstk  --  allocate stack memory, returning address of topmost int
 *------------------------------------------------------------------------
 */
SYSCALL *getstk(nbytes)
        unsigned  int nbytes;
{
        char    ps;
        struct  mblock  *p, *q; /* q follows p along memlist            */
        struct  mblock  *fits, *fitsq;
        unsigned len;

        disable(ps);
        if (nbytes == 0) {
                restore(ps);
                return( (int *)SYSERR );
        }
        nbytes = (unsigned)roundew(nbytes);
        fits = NULL;
        q = &memlist;
        for (p = q->mnext ; p != NULL ; q = p,p = p->mnext)
                if ( p->mlen >= nbytes) {
                        fitsq = q;
                        fits = p;
                }
        if (fits == NULL) {
                restore(ps);
                return( (int *)SYSERR );
        }
        if (nbytes == (len = fits->mlen) ) {
                fitsq->mnext = fits->mnext;
        } else {
                fits->mlen -= nbytes;
        }
        fits = ((unsigned)fits) + len - sizeof(int);
        restore(ps);
        return( (int *)fits );
}
```

8.5.3 Releasing Storage

- System call `freemem` returns a block of storage by inserting it at the proper location in the **free list** and **coalescing it** with any **adjacent free blocks**.
- `freemem` stops searching as soon as address of the block to be returned lies between p and q.
- Two cases lead to `SYSERR`: a block starts **within free area** or block intersects "free area":

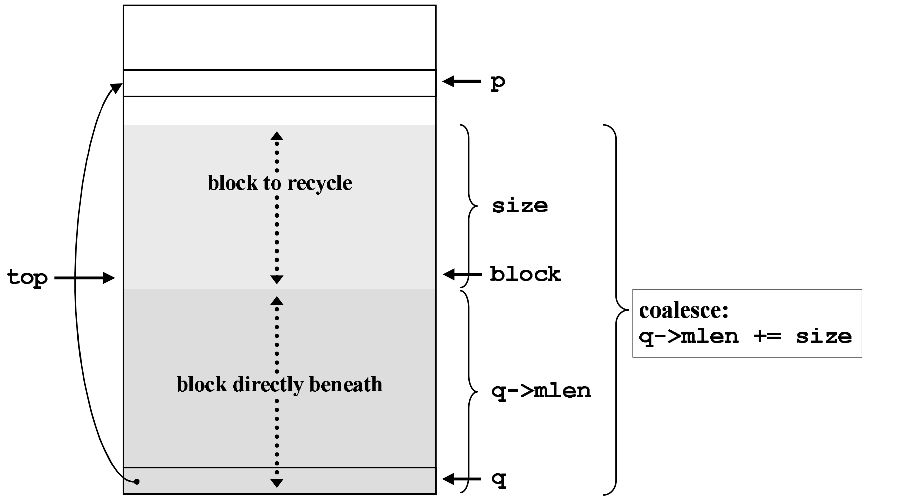

top
block
q->mlen
q
p
size
block

Releasing Storage

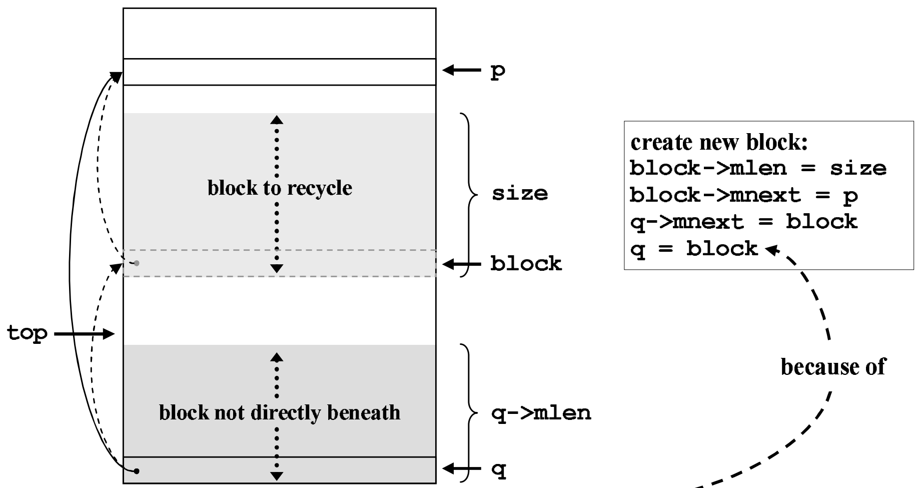
```
/*----------------------------------------------------------------------
 *  freemem  --  free a memory block, returning it to memlist
 *----------------------------------------------------------------------
 */
SYSCALL freemem(block, size)
        struct  mblock  *block;
        unsigned size;
{
        char    ps;
        struct  mblock  *p, *q;
        unsigned top;
        if (size==0 || (unsigned)block>(unsigned)maxaddr
                    || ((unsigned)block)<((unsigned)&end))
                return(SYSERR);
        size = (unsigned)roundew(size);
        disable(ps);
        for( p=memlist.mnext,q= &memlist ; (char *)p!=NULL && p<block ;
                    q=p,p=p->mnext );
        if ((top=q->mlen+(unsigned)q)>(unsigned)block && q!= &memlist ||
                (char *)p!=NULL && (size+(unsigned)block) > (unsigned)p) {
                restore(ps);
                return(SYSERR);
        }
        if ( q!= &memlist && top == (unsigned)block )
                q->mlen += size;
        else {
                block->mlen = size;
                block->mnext = p;
                q->mnext = block;
                q = block;
        }
        if ( (unsigned)( q->mlen + (unsigned)q ) == (unsigned)p) {
                q->mlen += p->mlen;
                q->mnext = p->mnext;
        }
        restore(ps);
        return(OK);
}
```

Releasing Storage

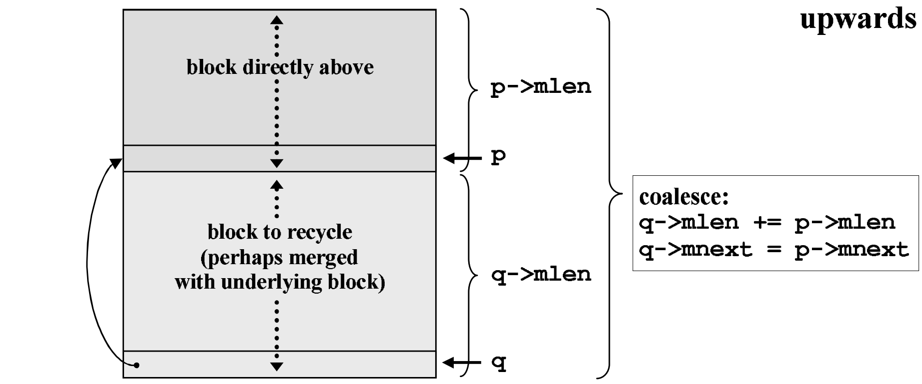
1) a) if `(top==(unsigned)block)` : coalesce downwards



1) b) if `(top!=(unsigned)block)` : create new block



2) if `(q->mlen + (unsigned)q == (unsigned)p)` : coalesce upwards



9 Interrupt Processing

- Interrupt processing takes place in layer 5 of the layered design of Xinu:
  0. chapter 2: **run-time environment** and **hardware**
  1. chapter 8: **memory management** (`getstk`, `freestk`, ...)
  2. chapter 3, 4, 5: **list manipulation** plus **context switching** (`resched`, `ctxsw`, `ready`, process table) and **process management** (`suspend`, `resume`, `kill`, `create`)
  3. chapter 6: process coordination by **semaphores** (`wait`, `signal`, `screate`, `sdelete`)
  4. chapter 7: **asynchronous message passing** (`send`, `receive`)
  5. chapter 9: **interrupt processing** (`intmap` table plus interrupt dispatchers)
- Without an **interrupt mechanism** the operating system cannot guarantee that it will ever **regain control** once user program is started!

9.1 Dispatching Interrupts

- The 11/2 processor uses *vectored interrupts* to handle exceptions and interrupts from external devices.
  - Whenever an external device needs to communicate with the processor, it places a signal on the **interrupt bus line**
  - If the processor is running with **interrupts enabled**, it checks this line **after every time an instruction has been executed** to see whether an interrupt needs processing.
  - To handle the interrupt, the processor sends an acknowledgement over the interrupt bus line.
  - The **first external device** with an pending request receives the acknowledgement and returns its interrupt vector address v along the bus.
  - When the processor receives v, the processor pushes the current PC and PS on the stack, loads the PC and PS from two words in memory, starts at location v and continues executing instructions at the new location.

Dispatching Interrupts

- Each external device is assigned a **unique interrupt vector address** v before it is inserted into the bus, thus enabling the system to distinguish among them. I.e., every device is **identified** by its unique interrupt vector address v.
- The interrupt vector locations contain a valid PC and PS. This PC points to an **interrupt handling routine** (interrupt handler) for the interrupting device.
- The processor execute the code of the **interrupt routine** before **returning** to the place where the user's program has interrupted.
- To make the interrupt **transparent** (i.e., invisible) to the running program, the interrupt routine saves and restores the **stack** of the machine – the values of R_0 to R_6 (since hardware saves PC and PS).

Dispatching Interrupts

- In practice SP $(= R_6)$ **needs no saving** – provided the interrupt routine pops off the stack whenever it pushed the stack before returning (i.e., the stack is restored to its original position).
- To prevent the interrupt routine from **being interrupted itself** by another device, the PS stored in the interrupt vector specifies a priority level of 7, i.e. `034i`. Thus interrupt processing begins with interrupts disabled.
- Interrupt processing ends by executing a **return from interrupt** instruction: `rti` or `rtt`, restoring – in a **single step** – the old PC and PS from the stack and returning the stack pointer SP to its original value.

Handling Exceptions

- **Exceptions** are handled similarly, except that they can **never be turned off** (i.e., disabled).
- Examples of exceptions occur when **nonexistent memory** is addressed, upon an **array-bound** error, execution of **invalid code** or **power failure**.

Dispatching Interrupts

- There is no procedure call instruction `jsr` in the interrupt program for jumping to the interrupt handler. The processor **simulates** a call to the appropriate interrupt routine **in between** the execution of instructions of the user's program.
- Comer uses a two-level strategy for processing interrupts:

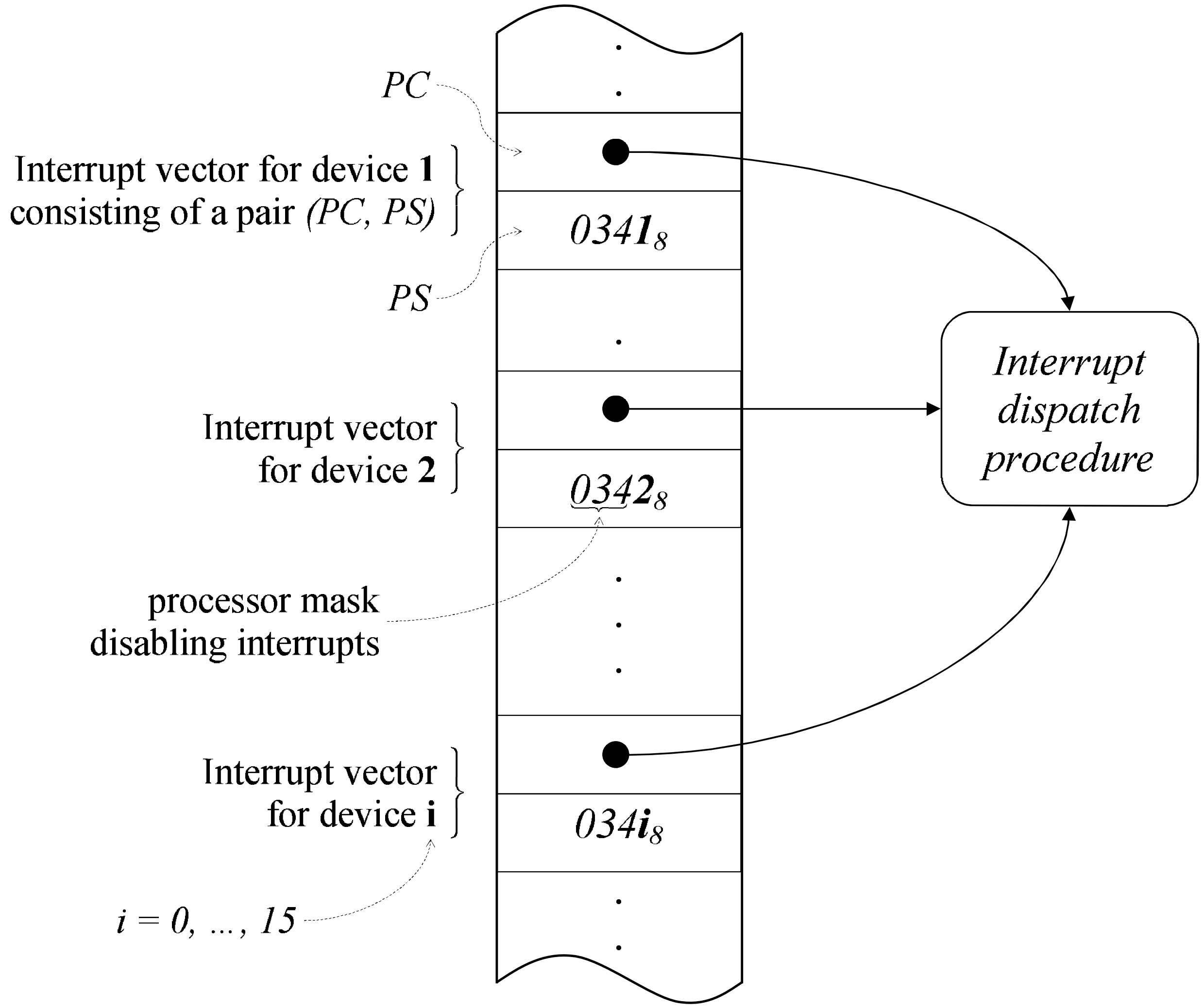
  1st level: Interrupt branch to **low-level interrupt dispatch routines** written in **assembly language**. They save and restore register info, identify the interrupting device, handle return from interrupts and call the second level:

  2nd level: **High-level routines** (for r/t, i/o processing, disc processing) **do the real work**, using the info passed from level 1.
- XINU contains three different interrupt dispatchers:
  - for handling **clock interrupts** – chapter 10
  - for handling **input interrupts** – this chapter
  - for handling **output interrupts** – this chapter

9.2 Input and Output Interrupt Dispatchers

- Devices **connect to computer system** through hardware mechanisms called **controllers**.
- **Controllers** reside on boards plugged into the system bus; their hardware converts digital data to wave forms and signals controlling peripheral devices s.a. terminals and disks
- Xinu allows both input and output operations to the **console** device with output to console **displayed on screen** and input from console **read from keyboard**.
- Input interrupts for characters typed on a terminal's keyboard trap on a **different vector** from **output interrupts** (for characters sent to the display)

Vectored Interrupt Table



Input and Output Interrupt Dispatchers

- **When an interrupt occurs**, the old PC and PS are **put on the stack,** new PC and PS are **loaded from interrupt vector**; while the new PC points to interrupt dispatch routine.
- The **interrupt dispatch routine** saves a copy of PS (because condition codes change lower-order bits of register) and masks off low-order 4 bits to compute an **index into a global interrupt dispatch table** which contains the **addresses of the high-level interrupt handlers** for each connected device (see the code for `intmap` later).
- I/O interrupt routines defined in `intmap` are `getchar()`, `putchar(ch)`, `fgetchar()`, and `fputchar(ch)`. They receive an argument, **identifying the interrupting device**. I/O interrupt dispatchers are combined in file `ioint.s`.

Input and Output Interrupt Dispatchers

Stack image after calling `jsr pc,*(r0)` in `ioinit`-routine:

	Stack		
	. . .		
	old PC	}	stacked by hardware
	old PS		
	new PS	$\longleftarrow$	to identify device number
	old R_0	}	since these are not saved by `csv`-routine and the high-level interrupt routines obey normal procedure call conventions
	old R_1		
	`icode`/`ocode`	$\longleftarrow$	parameter for interrupt routine
$SP \longrightarrow$	return address	$\longleftarrow$	return address of `move 2(sp),r1`

Input and Output Interrupt Dispatchers

After returning from interrupt-routine:

$SP + 12$	$\longrightarrow$	...
		old PC
$SP + 8$	$\longrightarrow$	old PS
		new PS
$SP + 4$	$\longrightarrow$	old R_0
$SP + 2$	$\longrightarrow$	old R_1
SP	$\longrightarrow$	`icode`/`ocode`
		return address

The call to `rtt` is popping old PS and PC from stack, adding 4 to SP, returning to address pointed at by old PC.

Input and Output Interrupt Dispatchers

```
/* io.h -  fgetc, fputc, getchar, isbaddev, putchar */


#define INTVECI inint           /* input interrupt dispatch routine     */
#define INTVECO outint          /* output interrupt dispatch routine    */
extern  int     INTVECI();
extern  int     INTVECO();

struct  intmap  {               /* device-to-interrupt routine mapping  */
        int     (*iin)();       /* address of input interrupt routine   */
        int     icode;          /* argument passed to input routine     */
        int     (*iout)();      /* address of output interrupt routine  */
        int     ocode;          /* argument passed to output routine    */
        };

#ifdef  NDEVS
extern  struct  intmap intmap[NDEVS];
#define isbaddev(f)     ( (f)<0 || (f)>=NDEVS )
#endif
#define BADDEV          -1

/* In-line I/O procedures */

#define getchar()       getc(CONSOLE)
#define putchar(ch)     putc(CONSOLE,(ch))
#define fgetc(unit)     getc((unit))
#define fputc(unit,ch)  putc((unit),(ch))

struct  vector  {
        char    *vproc;         /* address of interrupt procedure       */
        int     vps;            /* saved process status word            */
        };
```

Input and Output Interrupt Dispatchers

```
/* ioint.s - inint, outint */

/* I/O interrupts trap here.  Original PC and PS are on top of the      */
/* stack upon entry.  Low order 4 bits of the current PS contain the    */
/* device descriptor.  Interrupts are disabled.                         */

        .globl  _inint,_outint,_intmap
_outint:                                / Output interrupt entry point
        mfps    -(sp)                   / Save device descriptor from PS
        mov     r0,-(sp)                / Save r0 (csv does not)
        mov     $_intmap+4,r0           / point r0 to output in intmap
        br      ioint                   / Go do common part of code
_inint:                                 / Input interrupt entry point
        mfps    -(sp)                   / Save device code from PS
        mov     r0,-(sp)                / Save r0 (csv does not)
        mov     $_intmap,r0             / point r0 to input in intmap
ioint:                                  / Code common to input & output
        mov     r1,-(sp)                / Save r1 (csv does not)
        mov     4(sp),r1                / Get saved PS in r1
        bic     $177760,r1              / Mask off device descriptor
        ash     $3,r1                   / pick correct entry in intmap
        add     r1,r0                   / Form pointer to intmap entry
        mov     2(r0),-(sp)             / Push "code" from intmap as arg
        jsr     pc,*(r0)                / Call interrupt routine
        mov     2(sp),r1                / Restore r1 and R0 from stack
        mov     4(sp),r0
        add     $8,sp                   / Pop arg, saved r0, r1, and PS
        rtt                             / Return from interrupt
```

9.3 The Rules for Interrupt Processing

1) Interrupt routines examine and modify global data structures such as I/O buffers, so **interference from over processes** must be prevented. This is essentially done in two ways:

- Disabling interrupts, with interrupts still disabled after interrupt routine returns; so only after `ioint` returns are interrupts enabled again.
- Disabling interrupts, but: **high level interrupt routines may enable interrupts** by calling `resched`.

In case CPU switches to a process with interrupts enabled (`send` called, starting the receiver waiting for a message, or `signal` making first process on waiting queue ready to run):

$\Longrightarrow$ Interrupt routines should **leave global data** in a **valid state** before calling `resched()`.

$\Longrightarrow$ **No process enables interrupts** unless it **previously disabled them**.

The Rules for Interrupt Processing

2) **Interrupts should not be disabled too long.** If they do, devices will fail to perform correctly, e.g., when processor does not accept character from an I/O device before the next one arrives, **data will be lost**.

$\Longrightarrow$ Interrupt routines must be designed to **enable further interrupts as quickly as possible**.

3) Interrupt code is executed by **whatever process is running when the interrupt occurs**. This is especially the `NULL` process. Recall that `resched()` requires that there's **at least one process ready to run**, so the `NULL` process must always be current or ready.

$\Longrightarrow$ Interrupt routines can **only call procedures** that **leave the executing process in the current or ready states**.

E.g., `signal` or `send` may be called in an interrupt routine, but not `wait` or `receive`!

9.4 Rescheduling while processing an Interrupt

This section explains **why no infinite stack-overflow can occur because of the combination of interrupt routines with `resched()` calls**:

- Assume a running process P is interrupted. Executing the dispatcher and high-level routine is done by process P, with its PS still **disabling interrupts**.
- Unless the high-level interrupt routines indirectly call `resched()` through which **another process**, Q, with possibly **enabled interrupts** is executed. However, $Q \neq P$ in case Q enables interrupts because P disables interrupts.
- Interrupting Q leads to **disabling interrupts** in Q, but, as a result of a call to `resched()`, may **enable interrupts** of a new process R. Important is that $R \neq Q$ and $R \neq P$.

... **and so on** ...

- So, **no more interrupts** can be in state of being executed than the **number of processes** executed in parallel on the ready queue and executing as current.
- For this to be sure, no procedure should **enable interrupts** unless it **previously disabled them** (when calling `resched()`).

10 Real-Time Clock Management

10.1 Kinds of clocks associated with a computer

- Three kinds of clocks are associated with a computer:
  1. A **central system clock**, controlling the rate the CPU executes instructions.
  2. A **real-time clock**, pulsing regularly an integral number of times each second, signalling the CPU each time a pulse occurs by posting an interrupt.
  3. A **time-of-day clock** like a wristwatch; the CPU controls this clock.
- Unlike the time-of-day clock, a real-time clock does not contain a counter, does not accumulate interrupts (left to the system), and controls the CPU.
- Responsibility for counting interrupts falls upon the system:

  If the CPU takes too long to service a real-time clock interrupt, or if it operates for more than one clock cycle with interrupts disabled, it will **miss the next real-time interrupt**, consequently:
  - ∗ Systems should **service clock interrupts quickly**.
  - ∗ Clock interrupts have **highest priority**.

10.2 Slowing down of the clock rate

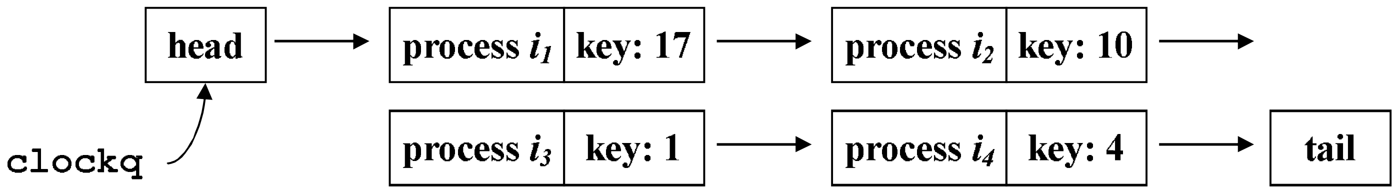
- Even so, the 11/2 processor cannot call a C-procedure on each clock interrupt, or it would spend **most of the time handling them**. So, the **clock rate must be slowed down**.
- How can the clock rate be „effectively " adjusted to prevent the system spending most of the time **handling interrupts**?
  - The clock interrupt handler/dispatcher `clkint` simulates a **slower-rate clock** by dividing the clock rate.
  - The LSI 11/2 real-time clock generates 60 pulses per second. `Clkint` **ignores** 5 clock interrupts in a row before (very quickly!) processing the 6^{th} one.
  - This reduces the **effective clock rate** to **10 pulses per second**, the so-called **tick rate**.
- The **tick rate** is the clock rate assumed by all processes in this chapter.

10.3 The use of real-time clock

- OS's use real-time clocks to compute the **time-slices allotted for the execution of each process**, by scheduling a preemption event.
- The OS also uses the real-time clock to provide processes with **timed delays**: The system maintains a list of processes, ordered by the time they should be **awakened**. When the real-time clock interrupts, it examines this list and wakes up processes for which the **delay has expired**.
- A preemption event is used to prevent processes from running forever. It is set in `resched` by `preempt := QUANTUM`.
- The clock interrupt dispatcher `clkint` **decrements preempt on each tick**, calling `resched` when preempt equals 0.
- `QUANTUM` shouldn't be **too small** (e.g. 2 or 3), causing too much overhead, and not **too large**, slowing down the reaction time. In practice most processes are stopped earlier that `QUANTUM` ticks, by executing `wait` or doing I/O processing.

10.4 Delta List Processing

- To avoid searching through lists, delayed processes are put on the **delta list**, residing in the q-structure. Variable `clockq` contains the q-index of its head.
- At each clock tick, `clkint` examines the **first process** in `clockq`, and calls a high-level interrupt routine `wakeup` to **awake processes when their delay has expired**.
- Processes on the `clockq` are **ordered** by the time at which they should be awakened, with each key telling the number of clock ticks which the process must delay **beyond the preceding one** on the `clockq`.
- E.g. 4 processes, needing to delay 17, 27, 28 and 32 ticks are ordered in the delta list as follows:



- Procedure `insertd(pid, head, key)` inserts process `pid` in delta-list `head`, given its key `key`.

Delta List Processing

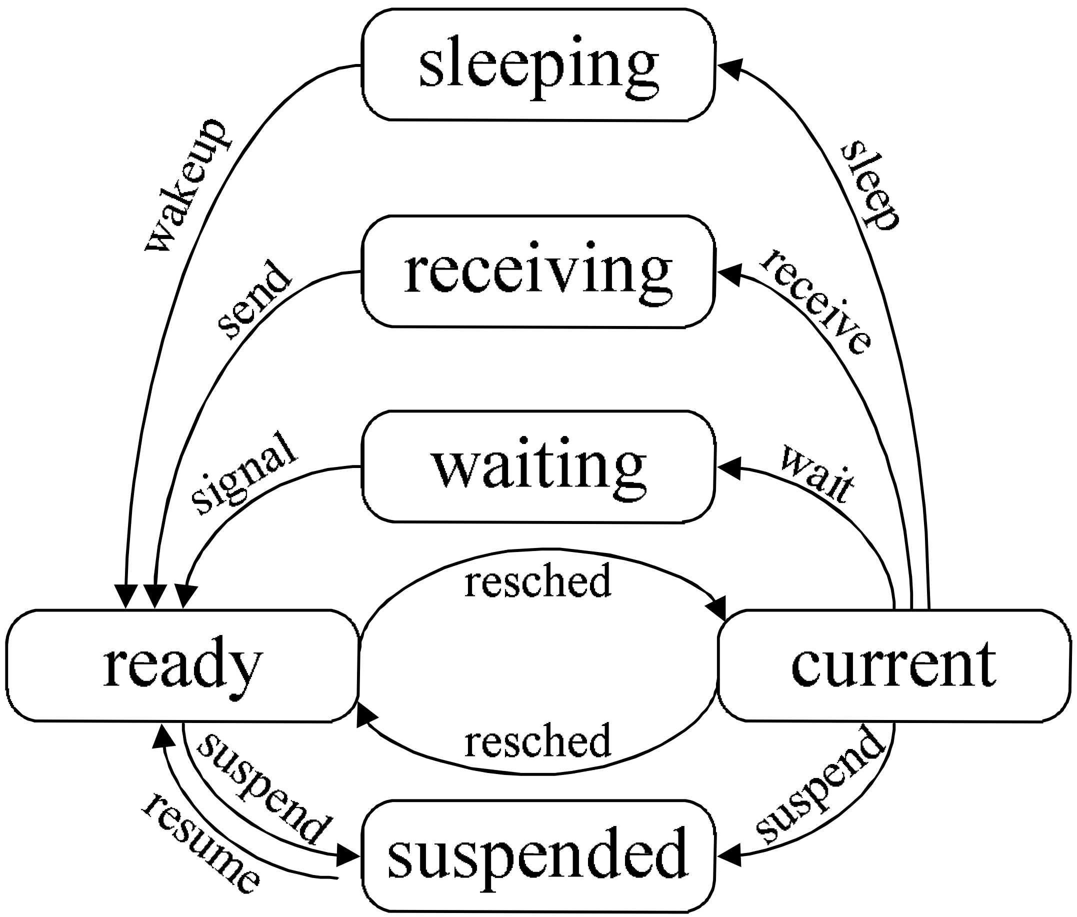
```
/* insertd.c - insertd */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  insertd  --  insert process pid in delta list "head", given its key
 *------------------------------------------------------------------------
 */
insertd(pid, head, key)
        int     pid;
        int     head;
        int     key;
{
        int     next;                   /* runs through list            */
        int     prev;                   /* follows next through list    */

        for(prev=head,next=q[head].qnext ;
            q[next].qkey < key ; prev=next,next=q[next].qnext)
                key -= q[next].qkey;
        q[pid].qnext = next;
        q[pid].qprev = prev;
        q[pid].qkey  = key;
        q[prev].qnext = pid;
        q[next].qprev = pid;
        if (next < NPROC)
                q[next].qkey -= key;
        return(OK);
}
```

10.5 Putting a process to sleep

- System call `sleep(`n`)` delays calling process (i.e. the current one) for n tenths-of-a-second, by moving the current process to the delta list `clockq`.
- This requires introducing a new process state for that moved process: `SLEEPING`. System variables introduced on page 129:
  - `slnempty`: sleeping list is non-empty
  - `clkruns`: if greater than 0 the hardware includes a real-time clock
  - `sltop`: address of the first key on clock list

  Notice that this **does not concern hard real-time**!
- See page 130: `setclkr` sets `clkruns` to 1 if real-time clock exists, otherwise to 0. It simulates an interrupt mechanism whose only purpose is to see **whether the real-time clock is present**, i.e., runs.

New process state SLEEPING



Putting a process to sleep

```
/* sleep10.c - sleep10 */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * sleep10  --  delay the caller for a time specified in tenths of seconds
 *------------------------------------------------------------------------
 */
SYSCALL sleep10(n)
        int n;
{
        char ps;

        if (n < 0  || clkruns==0)
                return(SYSERR);
        disable(ps);
        if (n == 0) {                   /* sleep10(0) -> end time slice */
                ;
        } else {
                insertd(currpid,clockq,n);
                slnempty = TRUE;
                sltop = (int *) & q[q[clockq].qnext].qkey;
                proctab[currpid].pstate = PRSLEEP;
        }
        resched();
        restore(ps);
        return(OK);
}
```

Putting a process to sleep

```
/* sleep.h */

#define CVECTOR 0100/*octal*/   /* location of clock interrupt vector   */

extern  int     clkruns;        /* 1 iff clock exists; 0 otherwise      */
                                /* Set at system startup.               */
extern  int     clockq;         /* q index of sleeping process list     */
extern  int     count6;         /* used to ignore 5 of 6 interrupts     */
extern  int     *sltop;         /* address of first key on clockq      */
extern  int     slnempty;       /* 1 iff clockq is nonempty             */

extern  int     defclk;         /* >0 iff clock interrupts are deferred */
extern  int     clkdiff;        /* number of clock ticks deferred       */
extern  int     clkint();       /* clock interrupt handler              */
```

10.6 Delays measured in seconds

- Size of integer n – 16 bits – **limits** the delay allowed by calling `sleep10(`n`)` to $2^{15} - 1$ tenths of a second $= 55$ minutes.
- System call `sleep(`n`)` measures delays in **seconds** and so allows delaying **up to 9 hours** – see the code in `sleep.c`:

```
/* sleep.c - sleep */
#include <conf.h> <kernel.h> <proc.h> <q.h> <sleep.h>

/*------------------------------------------------------------
 * sleep  --  delay the calling process n seconds
 *------------------------------------------------------------
 */
SYSCALL sleep(n)
        int     n;
{
        char    ps;

        if (n<0 || clkruns==0)
                return(SYSERR);
        if (n == 0) {
                disable(ps);
                resched();
                restore(ps);
                return(OK);
        }
        while (n >= 1000) {
                sleep10(10000);
                n -= 1000;
        }
        if (n > 0)
                sleep10(10*n);
        return(OK);
}
```

10.7 Awaking sleeping processes

- Read the code for `clkint` interrupt dispatcher on page 136 to see:
  - that `clkint` decrements the count of the first key on `clockq` at each tick, until his equals 0, and
  - the high-level interrupt procedure `wakeup()` is called, to put a process with key = 0 on the ready list.
- `wakeup()` assumes that interrupts have been disabled upon entry, as signalled by keyword `INTPROC`:

```
/* wakeup.c - wakeup */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * wakeup  --  called by clock interrupt dispatcher to awaken processes
 *------------------------------------------------------------------------
 */
INTPROC wakeup()
{
        while (nonempty(clockq) && firstkey(clockq) <= 0)
                ready(getfirst(clockq),RESCHNO);
        if ( slnempty = nonempty(clockq) )
                sltop = (int *) & q[q[clockq].qnext].qkey;
        resched();
}
```

10.8 Deferred clock processing

- The **deferred mode** allows the system to accumulate clock ticks in variable `clkdiff` **without initiating events**.
- I.e., it **postpones the reaction** upon real-time interrupts. The clock handler can schedule events that **should have occurred** to the point where it leaves deferred mode and returns to **original mode**.
- A process can place the clock in deferred mode by calling `stopclk`, and return clock to real time mode by calling `strtclk`:
  - `Stopclk` **counts deferral requests** by incrementing `defclk`.
  - `strtclk` counts „restart “ requests by decrementing `defclk`.
  - As long as `defclk` remains **positive** the interrupt handler counts clock times in `clkdiff` **without processing them**.
- `strtclk` „makes up for lost time “ when `defclk` equals 0, by catching up on all events that **should have occurred** while the clock remained deferred: `strtclk` **updates the preemption counter** and abstracts the **accumulated ticks** from the delay of sleeping processes.

Deferred clock processing

```
/* ssclock.c - stopclk, strtclk */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 *  stopclk  --  put the clock in defer mode
 *------------------------------------------------------------------------
 */
stopclk()
{
        defclk++;
}
```

Deferred clock processing

`ssclock.c` continued:

```
/*------------------------------------------------------------------------
 *  strtclk  --  take the clock out of defer mode
 *------------------------------------------------------------------------
 */
strtclk()
{
        char ps;
        int makeup;
        int next;

        disable(ps);
        if ( defclk<=0 || --defclk>0 ) {
                restore(ps);
                return;
        }
        makeup = clkdiff;
        preempt -= makeup;
        clkdiff = 0;
        if ( slnempty ) {
                for (next=firstid(clockq) ;
                    next < NPROC && q[next].qkey < makeup ;
                    next=q[next].qnext) {
                        makeup -= q[next].qkey;
                        q[next].qkey = 0;
                }
                if (next < NPROC)
                        q[next].qkey -= makeup;
                wakeup();
        }
        if ( preempt <= 0 )
                resched();
        restore(ps);
}
```

`for`-loops in C

Look at this `for`-loop in C:

$$\texttt{for}(S_1;\ \mathrm{exp};\ S_2)\ S_3$$

The semantic of this loop is

$$S_1;\ \texttt{while}\ (\mathrm{exp} \neq 0)\ \texttt{do begin}\ S_3;\ S_2\ \texttt{end}$$

Nota Bene: S_1, exp and S_2 are **optional**, if exp is omitted, an **infinite loop** results.

10.9 Clock interrupt processing

- The **assembly language routine** dispatching clock interrupts is called `clkint`. Read it **carefully**!

```
/* clkint.s -  clkint */

/*------------------------------------------------------------------------ /*
clkint  --  real-time clock interrupt service routine
/*------------------------------------------------------------------------
        .globl  _clkint
_clkint:
        dec     _count6                 / Is this the 6th interrupt?
        bgt     clret                   /  no => return
        mov     $6,_count6              /  yes=> reset counter&continue
        tst     _defclk                 / Are clock ticks deferred?
        beq     notdef                  /  no => go process this tick
        inc     _clkdiff                /  yes=> count in clkdiff and
        rtt                             /       return quickly
notdef:
        tst     _slnempty               / Is sleep queue nonempty?
        beq     clpreem                 /  no => go process preemption
        dec     *_sltop                 /  yes=> decrement delta key
        bgt     clpreem                 /       on first process,
        mov     r0,-(sp)                /       calling wakeup if
        mov     r1,-(sp)                /       it reaches zero
        jsr     pc,_wakeup              /       (interrupt routine
        mov     (sp)+,r1                /        saves & restores r0
        mov     (sp)+,r0                /        and r1; C doesn't)
clpreem:
        dec     _preempt                / Decrement preemption counter
        bgt     clret                   /   and call resched if it
        mov     r0,-(sp)                /   reaches zero
        mov     r1,-(sp)                /      (As before, interrupt
        jsr     pc,_resched             /       routine must save &
        mov     (sp)+,r1                /       restore r0 and r1
        mov     (sp)+,r0                /       because C doesn't)
clret:
        rtt                             / Return from interrupt
```

10.10 Clock initialization

- The clock interrupt vector should be **initialized** at system start-up, **before clock interrupts occur**. This is done by calling `clkinit()`.
- Variable `CVECTOR` ($= 0100_8 = 64_{10}$) is defined in `sleep.h` on an earlier slide. Recall: `setclkr()` initializes `clkruns`. Now read `setclkr()`:
  - Observe in setloop: hardware clock R_1 decreased at tact of hardware clock!
  - If after 30.000 ticks of the hardware clock no real-time interrupt occurred, `clkruns` remains zero. If clock interrupt occurs, `setclkr`'s control jumps to `setint`: `clkruns` > 0.
  - `setclkr` returns, after restoring the original real-time clock vector, to the next instruction pointed at by PC.

Clock initialization

```
/* clkinit.c - clkinit */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>

/*
 *------------------------------------------------------------------------
 * clkinit - initialize the clock and sleep queue (called at startup)
 *------------------------------------------------------------------------
 */
clkinit()
{
        int *vector;

        vector = (int *) CVECTOR;       /* set up interrupt vector      */
        *vector++ = clkint;
        *vector = DISABLE;
        setclkr();
        preempt = QUANTUM;              /* initial time quantum         */
        count6 = 6;                     /* 60ths of a sec. counter      */
        slnempty = FALSE;               /* initially, no process asleep */
        clkdiff = 0;                    /* zero deferred ticks          */
        defclk = 0;                     /* clock is not deferred        */
        clockq = newqueue();            /* allocate clock queue in q    */
}
```

Clock initialization

```
/* setclkr.s - setclkr */

CVECTPC =       100                     / clock interrupt vector address
CVECTPS =       102                     /  "      "          "      (of PS)
DISABLE =       340                     / PS that disables interrupts
ENABLE  =       000                     / PS that enables interrupts
COUNT   =       30000.                  / Times to loop (in decimal)

/*------------------------------------------------------------------------
/* setclkr  --  set cklruns to 1 iff real-time clock exists, 0 otherwise
/*------------------------------------------------------------------------
        .globl  _setclkr
_setclkr:
        mov     r1,-(sp)                / save register used
        clr     _clkruns                / initialize for no clock
        mov     *$CVECTPS,-(sp)         / save clock interrupt vector
        mov     *$CVECTPC,-(sp)         /   on caller's stack
        mov     $DISABLE,*$CVECTPS      / set up new interrupt vector
        mov     $setint,*$CVECTPC
        mov     $COUNT,r1               / initialize counter for loop
        reset                           / clear other interrupts, if any
        mtps    $ENABLE                 / allow interrupts
setloop:
        dec     r1                      / loop COUNT times waiting for
        bpl     setloop                 /   a clock interrupt
        mtps    $DISABLE                / no interrupt occurred, so quit
        br      setdone
setint:
        inc     _clkruns                / clock interrupt jumps here
        add     $4,sp                   / pop pc/ps pushed by interrupt
setdone:
        mov     (sp)+,*$CVECTPC         / restore old interrupt vector
        mov     (sp)+,*$CVECTPS
        mov     (sp)+,r1                / restore register
        rts     pc                      / return to caller
```

11 Device Independent Input and Output

- Operating system **control I/O devices** for three reasons:
  - The hardware interface to such devices is **crude**, requiring **complex software packets** for their control, called **device drivers**.
  - Device drivers are **shared resources**, which need to be protected and allocated in a fair and safe way.
  - A **uniform, flexible, high-level interface** should be provided, allowing users to write programs without knowing the **machine configuration**.
- Discussed in this chapter are the selection of a set of **machine-independent high-level I/O primitives** and data structures required to relate these primitives to specific devices
- **How are these primitives selected?**
  - By generating a list of desirable properties; delivering a set of high-level primitives, giving their meaning with relation to certain abstract (classes of) devices, terminal, disks, etc.
  - Build software mapping the abstract devices to instances of that device.

11.1 Properties of the I/O interface

- Should processes **block** while performing I/O operations? Or should they **continue executing** and be **notified** when the operation completes?
  - **Asynchronous operations** are useful for controlling overlap – i.e., more parallelism – of computation and I/O operations.
  - **Synchronous operations** delay input operations until data arrives and output operations until data has been consumed. Their advantage is that users can depend on data **immediately** after an input operation, and change data **immediately** after an output operation.
- Which **format** does data have, and what is the **size** of transferred data?
  - Single-byte transfer (teletype terminals, e.g., the console)
  - Block transfer (a block of many bytes, used for data transfer to/from disks).

11.2 Abstract operations

- `getc()` and `putc( , ch)` deal with **single-character transfer**:
  - `getc()` reads next character from keyboard
  - `putc( , ch)` displays one character on terminal
- `read` and `write` deal with transfer to/from **continuous blocks of memory**:
  - `read` reads a specified number of characters
  - `write` displays several characters within one call
- `control` allows control of the device (driver): e.g., whether the system **echoes** each character as it is typed on the keyboard
- `seek` applies only to **randomly accessible memory (RAM)** to search for a particular position
- `open` and `close` inform device (drivers) that data transfer **will begin or has ended** (applies to disk and file access)
- `init` **initializes** the device and device driver at system start up

11.3 Binding abstract operations to real devices

- The system **maps** these high-level I/O operations to **specific device drivers**: It hides the details of the hardware and the device drivers. E.g., that keyboard and display are **independent**, it makes programs **independent** of the particular hardware configuration.
- The high-level calls of these operations constitute the **environment which the system presents to running programs** – i.e., the programs only perceive the peripheral devices through these abstract calls, see, e.g., §11.5 on page 147.
- The system also maps abstract names, s.a. **console**, to real devices.

Binding abstract operations to real devices

- Coded into the system is a **description of each abstract device**: E.g., the device driver routines which it uses, and the address of the real device to which it corresponds. Compare, e.g., §2.1.9 and §2.1.10 on pages 28 ff.
- When a new device is added to the system, or, e.g., the device addresses are modified, the system must be **altered and recompiled**.
- However, since programs do not contain **direct calls** to these devices and their drivers, and **no device addresses**, programs don't need to be recompiled, as long as the **abstract device descriptors** do not change (e.g., adding graphics facilities would change them).

11.4 Binding I/O calls to device drivers at run-time

- Routines like `read` in the compiled code should map **abstract device descriptors**, s.a., console, to **device driver routines** and real device addresses.
- In XINU, each abstract device is assigned an **integer device descriptor** (0, . . . , 8), at system configuration. E.g., the console has the same device descriptor in all XINU systems.
- After system configuration, device descriptors are bound into the system when it is **compiled**, and these are placed in a **library**. No recompilation is needed unless the system changes (e.g, a new device is added).
- At **run-time** (after compilation) the program calls high-level routines, s.a., `read(descriptor, ..., ...)`, and `putc(descriptor, ch)` having the device descriptor as argument.
- **Device switch table**: `devtab[0 ... 8]` contains an array of `devsw` structures of `devsw` containing the device drivers for that abstract device, and device addresses. A **device descriptor** is an index into the **device switch table** which contains an entry for each value of the device descriptor; read pages 144 to 146.

Device switch table

- Each entry in the device switch table corresponds to a **single device**, containing:
  - `dvnum`, the corresponding entry into the **interrupt dispatch table** `intmap`.
  - Addresses of the **device driver routines** for that device: `dvgetc`, `devputc`, `dvread`, `dvwrite`, `dvcntrl`, `dvseek`, `dvinit`. These routines hold addresses of **driver routines** corresponding to these high-level operations.
  - Device address and other information, since **more than one device** can use the same device driver.
- Device switch table also contains:
  - hardware device addresses `dvcsr`,
  - interrupt vector addresses `dvivec` and `dvovec`,
  - the interrupt routines for input (`dviint`) and output (`dvoint`),
  - buffer pointer `dvioblk` (address of corresponding control block for that device, see, e.g., page 182), and
  - an integer `dvminor` distinguishing among multiple copies of a device.

Device switch table – see page 144

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

#define NULLPTR (char *)0

/* Device table declarations */
struct  devsw   {                       /* device table entry */
        int     dvnum;
        char    *dvname;
        int     (*dvinit)();
        int     (*dvopen)();
        int     (*dvclose)();
        int     (*dvread)();
        int     (*dvwrite)();
        int     (*dvseek)();
        int     (*dvgetc)();
        int     (*dvputc)();
        int     (*dvcntl)();
        int     dvcsr;
        int     dvivec;
        int     dvovec;
        int     (*dviint)();
        int     (*dvoint)();
        char    *dvioblk;
        int     dvminor;
        };

extern  struct  devsw devtab[];         /* one entry per device */
```

Device switch table – see pages 145, 146

`conf.h` continued:

```
/* Device name definitions */

#define CONSOLE     0                   /* type tty      */
#define OTHER       1                   /* type tty      */
#define RINGOIN     2                   /* type dlc      */
#define RINGOOUT    3                   /* type dlc      */
#define DISK0       4                   /* type dsk      */
#define FILE1       5                   /* type df       */
#define FILE2       6                   /* type df       */
#define FILE3       7                   /* type df       */
#define FILE4       8                   /* type df       */

/* Control block sizes */

#define Ntty    2
#define Ndlc    2
#define Ndsk    1
#define Ndf     4

#define NDEVS   9

/* Configuration and Size Constants */

#define MEMMARK                         /* define if memory marking used*/
#define NNETS       1                   /* number of Xinu ring networks */
                                        /*  (remove if there are zero)  */
#define NPROC       10                  /* number of user processes     */
#define NSEM        50                  /* number of semaphores         */
#define RTCLOCK                         /* system has a real-time clock */
#define VERSION     "6.1b (05/22/84)"   /* label printed at startup     */
```

Device switch table – see pages 145, 149

`conf.h` continued:

```
/* Declarations of I/O routines referenced */

extern  int     ttyinit();  /* open e.g. on a console                     */
extern  int     ionull();   /* signifies an unnecessary but innocuous op. */
extern  int     ttyread();
extern  int     ttywrite();
extern  int     ioerr();    /* signifies an illegal operation             */
extern  int     ttycntl();
extern  int     ttygetc();
extern  int     ttyputc();
extern  int     ttyiin();
extern  int     ttyoin();
extern  int     dclinit();
extern  int     dclread();
extern  int     dclwrite();
extern  int     dlccntl();
extern  int     dlcputc();
extern  int     dlciin();
extern  int     dlcoin();
extern  int     dsinit();
extern  int     dsopen();
extern  int     dsread();
extern  int     dswrite();
extern  int     dsseek();
extern  int     dscntl();
extern  int     dsinter(); /* disk interrupt routine */
extern  int     lfinit();
extern  int     lfclose();
extern  int     lfread();
extern  int     lfwrite();
extern  int     lfseek();
extern  int     lfgetc();
extern  int     lfputc();
```

11.5 Implementation of high-level I/O operations

- There is a procedure call in the object code of a program for each of the abstract operations `getc`, `put`, `read` etc. – e.g.,

  `read(descrp, buff, count)`

  - `descrp` denotes the device descriptor: an index into device switch table `devtab`;
  - `buff` is the address of buffer **into which** the data read should be written;
  - `count` is the number of characters to be read.
- `read(..., ..., ...)` calls the driver routine given by field `dvread` of the structure `devtab` corresponding entry to `descrp`.

Implementation of high-level I/O operations

```
/* read.c - read */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  read  -  read one or more bytes from a device
 *------------------------------------------------------------------------
 */
SYSCALL read(descrp, buff, count)
int descrp, count;
char *buff;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvread)(devptr,buff,count) );
}
```

Implementation of high-level I/O operations

```
/* control.c - control */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  control  -  control a device (e.g., set the mode)
 *------------------------------------------------------------------------
 */
SYSCALL control(descrp, func, addr, addr2)
int descrp, func;
char *addr,*addr2;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvcntl)(devptr, func, addr, addr2) );
}
```

Implementation of high-level I/O operations

```
/* getc.c - getc */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  getc  -  get one character from a device
 *------------------------------------------------------------------------
 */
SYSCALL getc(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvgetc)(devptr) );
}
```

Implementation of high-level I/O operations

```
/* init.c - init */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  init  -  initialize a device
 *------------------------------------------------------------------------
 */
init(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvinit)(devptr) );
}
```

Implementation of high-level I/O operations

```
/* putc.c - putc */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  putc  -  write a single character to a device
 *------------------------------------------------------------------------
 */
SYSCALL putc(descrp, ch)
int descrp;
char ch;
{
        struct  devsw   *devptr;

        if (isbaddev    (descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvputc)(devptr,ch) );
}
```

Implementation of high-level I/O operations

```
/* seek.c seek */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  seek  --  position a device (a very common special case of control)
 *------------------------------------------------------------------------
 */
SYSCALL seek(descrp, pos)
int descrp;
long pos;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvseek)(devptr,pos) );
}
```

Implementation of high-level I/O operations

```
/* write.c - write */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  write  -  write 1 or more bytes to a device
 *------------------------------------------------------------------------
 */
SYSCALL write(descrp, buff, count)
        int descrp, count;
        char *buff;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvwrite)(devptr,buff,count) );
}
```

11.6 Opening and closing devices

- Disk devices require programs to **start them up** before a transfer operation, and to **stop them** after the transfer has completed: `open` and `close`.

```
/* open.c - open */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  open  -  open a connection to a device/file (parameters 2 & 3 are optional)
 *------------------------------------------------------------------------
 */
SYSCALL open(descrp, nam, mode)
int     descrp;
char    *nam;
char    *mode;
{
        struct  devsw   *devptr;

        if ( isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvopen)(devptr, nam, mode) );
}
```

Opening and closing devices

```
/* close.c - close */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  close  -  close a device
 *------------------------------------------------------------------------
 */
SYSCALL close(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvclose)(devptr));
}
```

11.7 Null and error entries in `devtab`

- Some entries in the device table are **not meaningful for a particular device**. E.g., `open` and `close` do not apply to consoles. Calling there operations would result in an error for console. How should these entries be filled in? By using:
  - `ioerr` – returns `SYSERR` when called. This signifies an illegal operation.
  - `ionull` – returns `OK`. This signifies an unnecessary but, otherwise, innocuous operation.

Null and error entries in `devtab`

```
/* ioerr.c - ioerr */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  ioerr  -  return an error (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
ioerr()
{
        return(SYSERR);
}

/* ionull.c - ionull */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  ionull  -  do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
ionull()
{
        return(OK);
}
```

11.8 Initialization of the I/O system

- `devtab` is initialized at system configuration time, so it is completely filled in by the time the system is compiled – see pages 154 and 155:

```
/* conf.c (GENERATED FILE; DO NOT EDIT) */

#include <conf.h>

/* device independent I/O switch */

struct  devsw   devtab[NDEVS] = {

/*  Format of entries is:
device-number,
init, open, close,
read, write, seek,
getc, putc, cntl,
device-csr-address, input-vector, output-vector,
iint-handler, oint-handler, control-block, minor-device,
*/

/*  CONSOLE  */

0,
ttyinit, ttyopen, ionull,
ttyread, ttywrite, ioerr,
ttygetc, ttyputc, ttycntl,
0177560, 0060, 0064,
ttyiin, ttyoin, NULLPTR, 0,
```

Initialization of the I/O system

```
/*  OTHER */

1,
ttyinit, ionull, ionull,
ttyread, ttywrite, ioerr,
ttygetc, ttyputc, ttycntl,
0177560, 0300, 0304,
ttyiin, ttyoin, NULLPTR, 1,

/*  RINGOIN  */

2,
dlcinit, ioerr, ioerr,
dlcread, dlcwrite, ioerr,
ioerr, dlcputc, dlccntl,
0176510, 0310, 0314,
dlciin, dlcoin, NULLPTR, 0,

/*  RINGOOUT  */

3,
dlcinit, ioerr, ioerr,
dlcread, dlcwrite, ioerr,
ioerr, dlcputc, dlccntl,
0176510, 0320, 0324,
dlciin, dlcoin, NULLPTR, 1,

/*  DISK0  */

4,
dsinit, dsopen, ioerr,
dsread, dswrite, dsseek,
ioerr, ioerr, dlccntl,
0176510, 0134, 0134,
dsinter, dsinter, NULLPTR, 0,

/*  FILE1  */

5,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 0,

/*  FILE2  */

6,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 1,

/*  FILE3  */

7,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 2,

/*  FILE4  */

8,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 3
        };
```

11.9 Interrupt vector initialization

- **Interrupt vectors** and the **interrupt dispatch table** `intmap` are **initalized at runtime**, using the information in `devtab`, by calling `init(0)`, . . . , `init(8)` (defined on page 149). This is done in §13, see page 197. E.g., `ttyinit` calls

$$\texttt{iosetvec(descrp, }\underbrace{\texttt{incode}}_{\texttt{minor}}\texttt{, }\underbrace{\texttt{outcode}}_{\texttt{minor}}\texttt{)}$$

  where `minor := devtab[descrp.dvminor]` in `ioinit(descrp)`.

- At startup, for each device descriptor the system call `init(k)`:
  - initializes `dvinit` routine,
  - initializes **interrupt vectors** and fills in entries in **interrupt dispatch table** `intmap` (used by interrupt dispatchers to locate the high-level IO interrupt handlers for that device); this is done by calling `iosetvec(descrp, minor, minor)`,
  - initializes the control blocks (see §12) and buffers associated with that device.

Interrupt vector initialization

- This initialization occurs in every device initialization routine. Part of it is collected in the C-procedure `ioinit(descr)`, which calls `iosetvec(descr, minor, minor)`.
- `iosetvec` initializes the `intmap` structure, i.e., interrupt dispatch table entry (see page 116) for that device and fills in **input and output interrupt vectors**.
- The `devtab` table forms the **general framework** linking **interrupts**, **devices** and **device driver routines**.

Interrupt vector initialization

```
/* ioinit.c - ioinit, iosetvec */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ioinit --  standard interrupt vector and dispatch initialization
 *------------------------------------------------------------------------
 */
ioinit(descrp)
int     descrp;
{
        int     ioblck;

        if (isbaddev(descrp) )
                return(SYSERR);
        ioblck = devtab[descrp].dvioblck;
        iosetvec(descrp, ioblck, ioblck);
        return(OK);
}
```

Interrupt vector initialization

`ioinit.c` continued:

```
/*------------------------------------------------------------------------
 *  iosetvec  -  fill in interrupt vectors and dispatch table entries
 *------------------------------------------------------------------------
 */
iosetvec(descrp, incode, outcode)
int     descrp;
int     incode;
int     outcode;
{
        struct  devsw   *devptr;
        struct  intmap  *map;
        struct  vector  *vptr;

        if (isbaddev(descrp))
                return(SYSERR);
        devptr = &devtab[descrp];
        map = &intmap[devptr->dvnum];   /* fill in interrupt dispatch   */
        map->iin =  devptr->dviint;     /*   map with addresses of high-*/
        map->icode = incode;            /*   level input and output     */
        map->iout = devptr->dvoint;     /*   interrupt handlers and     */
        map->ocode = outcode;           /*   minor device numbers       */
        vptr = (struct vector *)devptr->dvivec;
        vptr->vproc = (char *)INTVECI;  /* fill in address of input     */
                                        /* interrupt dispatch routine   */
        vptr->vps = descrp | DISABLE;   /*   vector PC and PS values    */
                                        /* | is C's bitwise or operator */
        vptr = (struct vector *)devptr->dvovec;
        vptr->vproc = (char *)INTVECO;  /* fill in address of output    */
                                        /* interrupt dispatch routine   */
        vptr->vps = descrp | DISABLE;   /*   vector PC and PS values    */
        return(OK);
}
```

11.10 Summary: Recap §11

- Call `read` in a program.
- Compilation results in translation into `read(descrp, buff, count)` – a Xinu routine written in C.
- Executing that routine results in executing the `dvread` routine whose address is stored in the **device switch table** entry of `devtab`.
- That results in posting one (or more) **interrupts** to the device in question.
- Those interrupts result in:
  - storing the original PC and PS on the stack,
  - overwriting these registers with values from the **interrupt vector**, and
  - executing the **interrupt dispatch routines** contained in the interrupt vector.
- The interrupt vector contains the address of the **interrupt dispatch routine** for the device and **operations concerned** plus the **device identifier** (descriptor).

Summary: Recap §11

- After saving the values of those registers that will be used, the interrupt dispatch routine:
  - calls the appropriate **high-level I/O interrupt routines**
  - with appropriate arguments **identifying the device in question**.
- The I/O dispatch routine finds the **addresses of the interrupt handlers** in the interrupt dispatch table `intmap`, in which also the I/O device identifiers are stored.
- These addresses are obtained from the device table **at system initialization**. The I/O interrupt vectors and interrupt dispatch table are initialized through calling `init(0)`, . . . , `init(8)`.
- Calling `init(descriptor)` results in caling the **device-dependent initialization routines**. The **standard** form of such a routine, `ioinit(descrp)`, is given on page 156, which calls `iosetvec`, part of every `dvinit` routine (see, e.g., `ttyinit` on page 182).

12 An Example Device Driver

Overview

- Device driver: a set of procedures **controlling a peripheral hardware device**.
- Device driver routines are partitioned into:
  - **Upper-half** device drivers: called from **user programs**.
  - **Lower-half** device drivers: handling device **interrupts**.

  These two communicate via a shared data structure, the **device control block**.
- In this chapter we discuss a device driver for a **teletype terminal (tty)**, a standard computer terminal with a keyboard managing:
  - output to display screen,
  - input from keyboard,

  using a SLU, an asynchronous serial line unit controller (see §2.1.9 and §2.1.11)

Overview

- Upper-half TTY device routines implement the `read`, `write`, `getc`, `putc` and `control` **device-independent I/O procedures**.
- Lower-half TTY device routines consist of:
  - **Output procedure `ttoin`**: called when transmitter interrupts, sending a character from the out-buffer – i.e., from the queue of characters waiting to be sent.
  - **Input procedure `ttyiin`**: called whenever an input character arrives, puts it in a queue (in-buffer) of incoming characters to be retrieved by upper-half routines.
  - **Initialization procedure `ttyinit`**: fills in the so-called **device control block** and interrupt vectors when system starts.

Introduction

- The **device switch table** `devtab` forms the framework linking interrupts, devices and device driver routines.
- This chapter introduces the device driver routines for a standard computer terminal with a keyboard, called **teletype (TTY)**.
- From the computer's viewpoint, this device consists of a **asynchronous serial line unit (SLU)**, between the keyboard and the system bus. I.e. a computer terminal connected to the computer over an asynchronous serial line unit (SLU) – see §2.1.8 to §2.1.11

12.1 The Device Type TTY

- Each TTY has as input device a keyboard **transmitting characters** (which are typed) to the computer, and an output device for displaying characters received from the computer on the screen.
- The TTY device driver consists of routines which **map operating system routines** for reading/writing characters to **hardware operations** transmitting characters to the display and receiving characters from keyboard.
- In practice, these driver routines communicate with the SLU-controller, requesting it to **send characters** or to **receive chars**.
- The controller's hardware transforms characters to and from **electrical impulses** on lines connecting the SLU to the terminal.

Name	**High-Order Byte**	**Low-Order Bytes**
`RCSR`	unused	receiver control
`RBUF`	receiver error	received data
`XCSR`	unused	transmitter control
`XBUF`	unused	transmitted data buffer

The Device Type TTY

- To minimize the **interference** between I/O devices and running processes, the driver uses **interrupt-driven processing**, to:
  1. transmit characters when the SLU is **idle**,
  2. read characters when these are **received** by the SLU,
  3. handle **errors** in the receiving process (receiver), and
  4. coordinate requests for I/O with the **speed** of the device.
- The latter is important because character transmission times are **many orders of magnitude slower** than processing speed of the CPU!

The Device Type TTY

The TTY driver operates using **parameters**, so it can be used for a variety of terminals in a variety of system configurations. E.g.,

- Several parameters control the **echo** of characters typed on the keyboard onto the screen.
  - Terminals operating in **full-duplex mode** do not display characters typed by the user – such terminals need the system to echo these characters.
  - Terminals operating in **half-duplex mode** display those characters automatically, and don't need the system to echo them.
- There are parameters concerning whether to echo **unprintable control chars** as printable combinations.
- Some parameters are dealing with **moving to a new line**: There are two unprintable characters controlling movement of the cursor:
  - **return** (also called **carriage return**): moving the cursor to the beginning of the current line
  - **newline** (`"\n"`): moving the cursor vertically down by one line

The Device Type TTY

- However, programs like to deal with a **single end-of-line character** to move to a new line. Hence, in practice, terminal must receive **both** `newline` and `return` characters to move to the start of a new line.
- Now, to simplify programming, the TTY driver can interpret both `newline` and `return` as the *move-to-the-start-of-a-new-line* character, depending on the value of several **parameters**:
  - `icrlf`: controls mapping of `return` to *move-to-the-start-of-a-new-line*, when received from the keyboard.
  - `ocrlf`: controls mapping of `newline` to *move-to-the-start-of-a-new-line*, i.e., controls whether the driver inserts `return` whenever a program writes `newline`.

12.2 Upper and Lower Halves of the Device Driver

- The TTY device drivers are partitioned in two sets:
  - **Upper half** device drivers: These are called by user processes through `devtab` in order to read/write characters.
  - **Lower half** device drivers: These are called by the interrupt dispatcher through the `intmap` interrupt dispatch table during interrupts, in order to transfer characters - these do the real work.

  This partition decouples **normal processing** from **hardware interrupts**.

Upper and Lower Halves of the Device Driver

- Every device has its own **circular buffers**:
  - One for **transferring input requests**, connecting high-level calls for actions for input along the SLU, and
  - one for **transferring high-level output requests** to actions for output along the SLU.
- I.e., these buffers record **outgoing data** from the time the user requests them to be sent until the time the device (SLU) receives them; and also record **incoming data** from the time the SLU deposits them until the user program requests them.

Upper and Lower Halves of the Device Driver

These buffers are important for three reasons:

1. The driver can accept incoming characters **before** the user process reads them (since the terminal user can **start typing at any time**).
2. Device such as disks transfer data in **large blocks**.
3. These buffers permit the drivers to perform I/O **concurrently** with user processes. I.e., when a user process writes data, the driver copies them into a buffer, allowing the user to **continue executing** while the driver transfers the data from the buffer to the device.

Device Driver's Upper and Lower Halves

So the TTY driver uses **two circular buffers per terminal**: One for the **output**:

- Output operations issued by the calling (user) program deposit to-be-written characters in the output buffer, and return to the caller.
- Also, these operations are accompanied by setting the interrupt-enable-bit of the SLU to 1 for the output line.
- As a result, the SLU transmitter interrupts whenever it is idle, calling interrupt processing routines in the lower half of the TTY device driver.
- It picks up characters from the output buffer and places them into the output control area, thereby starting their transmission.

And one for the **input**:

- Whenever the SLU receiver interrupts after/when it has received a character, the interrupt dispatcher calls the lower-half input interrupt routine.
- The lower-half interrupt input handler reads the waiting character and deposits it in the circular buffer.
- A process waiting for input from the (empty) circular buffer is started as soon as the next character arrives.

Upper and Lower Halves of the Device Driver

- **Ideally**, the two halves of the device driver communicate through the shared buffer:
  - Upper-half routines **enqueue requests** for data transfer or device control; they do not interact with devices directly.
  - Lower-half routines **transfer data from buffer** or control devices; they do not interact with user program directly.
- **In practice**, the two halves do more than manipulating shared data:
  - An output routine **starts up** the lower-half routine when it deposits output in its buffer.
  - In case the available buffer space is **full** when a process tries to write, the upper and lower halves must **coordinate**, stopping the process until space becomes available.
  - Also these routines start a process waiting for input from an empty buffer, as soon as the **next character arrives**.

12.3 Synchronization of the Upper and Lower Halves

- The synchronization of the upper and lower halves via circular buffers seems at first glance to be a simple instance of **producer-consumer synchronization**, implemented using semaphores; for upper-half output routines **produce** characters **consumed** by the lower-half output routines; and lower-half input routines **produce** characters **consumed** by upper-half input routines.
- Input poses no problem because user processes calling upper-half routines can **wait** for lower half input routines to produce a character. The latter is **signalled** by a `signal` operation, in interrupt mode.
- However, the following violates the rule that **processes in interrupt mode** should always be `current` or `ready`, and never be on an any other list than the ready list:

  For output poses a problem, if output offered by a user process is translated in a `signal` operation in the upper-half routines, since this implies that the lower-half routines should `wait` by a `wait` operation, until that a character becomes available! And that implies that, i.e., the **null process** would be put on a semaphore queue, which is **forbidden**.

A Problem ... and its Solution!

Problem:

- During an interrupt, the process carrying out that interrupt, should always be running (`CURRENT`) or on the ready queue (`READY`), but never be waiting on another queue, i.e. be in another process state. But that is not implied by the above producer/consumer synchronization for output operations!
- For the null process is only allowed to be in the `READY` or `CURRENT` state, since, otherwise, there might be no process ready to run when calling **`resched`**!

Solution for the output:

- Instead of synchronizing the upper and lower halves by the **availability** of output characters, occupying free buffer space, synchronize them by the size of the complement: **The number of free buffer slots**!
- Now the output operation in the upper-halve device driver has to **wait** for free space to become available, until the output operation in the lower-half **signals** that free space has become available, after an output operation in the SLU has interrupted, i.e., has ended.
- In this way, the lower-half **produces** free space, which upper-half **consumes**, when implementing an output operation.

12.4 Control Block and Buffer Declarations

- Each SLU used as TTY has its own pair of input and output **buffers**, and accompanying input and output **semaphores** `isem` (counting number of characters) and `osem` (counting number of free buffer slots).
- This, and other information, is kept in the tty control block structure. There is a control block structure for each tty device. Control blocks also contain, e.g., the control parameters `icrlf` and `ocrlf`.

Control Block and Buffer Declarations

```
/* tty.h */

#define IOCHERR         0200            /* bit set in when an error     */
                                        /* occurred reading the char    */
#define OBMINSP         20              /* min space in buffer before   */
                                        /* processes awakened to write  */
#define EBUFLEN         20              /* size of echo queue           */

/* size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef IBUFLEN
#define IBUFLEN         128             /* num. chars in input queue    */
#endif
#ifndef OBUFLEN
#define OBUFLEN         64              /* num. chars in output queue   */
#endif

/* mode constants */

#define IMRAW           'R'             /* raw mode => nothing done     */
#define IMCOOKED        'C'             /* cooked mode => line editing  */
#define IMCBREAK        'K'             /* honor echo, etc, no line edit*/
#define OMRAW           'R'             /* raw mode => normal processing*/
```

Control Block and Buffer Declarations

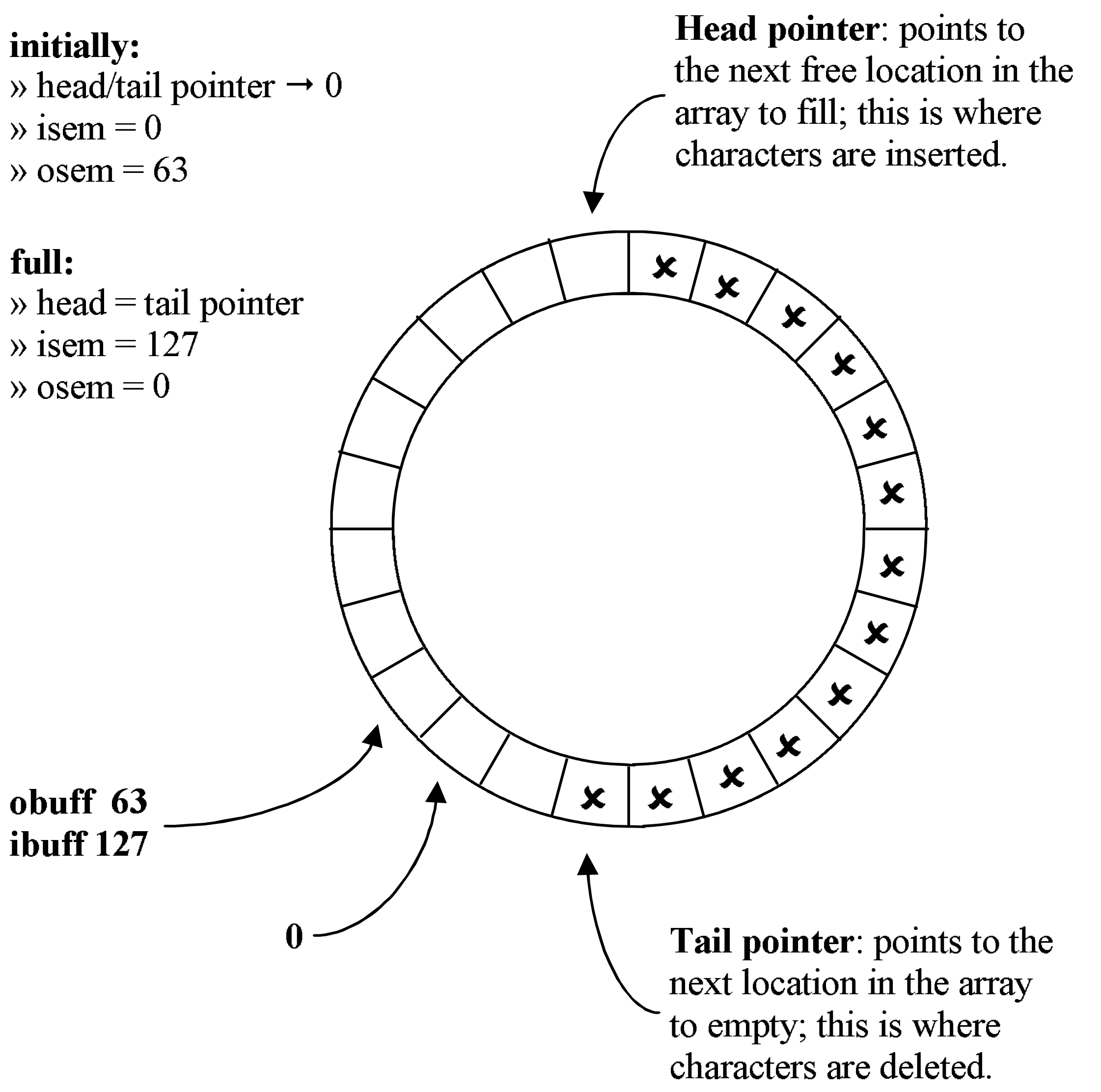
```
struct  tty     {                       /* tty line control block       */
        int     ihead;                  /* head of input queue          */
        int     itail;                  /* tail of input queue          */
        char    ibuff[IBUFLEN];         /* input buffer for this line   */
        int     isem;                   /* input semaphore              */
        int     ohead;                  /* head of output queue         */
        int     otail;                  /* tail of output queue         */
        char    obuff[OBUFLEN];         /* output buffer for this line  */
        int     osem;                   /* output semaphore             */
        int     odsend;                 /* sends delayed for space      */
        int     ehead;                  /* head of echo queue           */
        int     etail;                  /* tail of echo queue           */
        char    ebuff[EBUFLEN];         /* echo queue                   */
        char    imode;                  /* IMRAW, IMCBREAK, IMCOOKED    */
        Bool    iecho;                  /* is input echoed?             */
        Bool    ieback;                 /* do erasing backspace on echo?*/
        Bool    evis;                   /* echo control chars as ^X ?   */
        Bool    ecrlf;                  /* echo CR-LF for newline?      */
        Bool    icrlf;                  /* map '\r' to '\n' on input?   */
        Bool    ierase;                 /* honor erase character?       */
        char    ierasec;                /* erase character (backspace)  */
        Bool    ikill;                  /* honor line kill character?   */
        char    ikillc;                 /* line kill character          */
        int     icursor;                /* current cursor position      */
        Bool    oflow;                  /* honor ostop/ostart?          */
        Bool    oheld;                  /* output currently being held? */
        char    ostop;                  /* character that stops output  */
        char    ostart;                 /* character that starts output */
        Bool    ocrlf;                  /* echo CR/LF for LF ?          */
        char    ifullc;                 /* char to send when input full */
        struct  csr     *ioaddr;        /* device address of this unit  */
};
extern  struct  tty tty[];
```

Control Block and Buffer Declarations

```
#define BACKSP  '\b'                     /* backspace one character pos. */
#define BELL    '\07'                    /* usually an audiable tone     */
#define ATSIGN  '@'
#define BLANK   ' '
#define NEWLINE '\n'                     /* line feed                    */
#define RETURN  '\r'                     /* carriage return              */
#define STOPCH  '\023'                   /* control-S stops output       */
#define STRTCH  '\021'                   /* control-Q restarts output    */
#define UPARROW '^'                      /* usually for visuals like ^X  */

/* ttycontrol function codes */
#define TCSETBRK        1                /* turn on BREAK in transmitter */
#define TCRSTBRK        2                /* turn off BREAK "       "     */
#define TCNEXTC         3                /* look ahead 1 character       */
#define TCMODER         4                /* set input mode to raw        */
#define TCMODEC         5                /* set input mode to cooked     */
#define TCMODEK         6                /* set input mode to cbreak     */
#define TCICHARS        8                /* return number of input chars */
#define TCECHO          9                /* turn on echo                 */
#define TCNOECHO        10               /* turn off echo                */
#define TFULLC          BELL             /* char to echo when buffer full*/
```

Control Block and Buffer Declarations

- **Key components:**
  - input buffer `ibuff` and input semaphore `isem`
  - output buffer `obuff` and output semaphore `osem` (counting the number of free buffer slots) – see the figure below:



Control Block and Buffer Declarations

- There is one **control block structure** per device in array TTY, indexed by the minor device number.
- `Ntty` is the **number of tty devices**: the system configuration program assigns a minor device number from 0 ... `ntty` − 1 to each device.
- The system configuration program also places the minor device number in the **device switch table** `devtab`.
- Both the interrupt device routines in the lower half and the driver routines in the upper half use minor device numbers as index into array TTY.
- Control block contains the information needed for synchronizing the upper and lower half routines.

12.5 Upper-Half TTY Input Routines

- Routines

  `ttygetc` `ttyputc` `ttyread` `ttywrite`

  form basis of upper-half TTY driver routines, corresponding to:

  `getc` `putc` `read` `write`

  which are the high-level I/O operations from §11.

- **Simplest** is `ttygetc`:
  - `ttygetc` retrieves the minor device number from device switch table and uses it as index into array TTY to locate the correct corresponding control block.
  - Then it executes `wait(isem)` until the lower-half input routine deposits a character in **ibuf**.
  - When `wait` returns, `ttygetc` extracts next character from **ibuf** and updates tail pointer using modulo (128) arithmetic and returns.

Upper-Half TTY Input Routines – ttygetc

```
/* ttygetc.c - ttygetc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttygetc - read one character from a tty device
 *------------------------------------------------------------------------
 */
ttygetc(devptr)
struct  devsw   *devptr;
{
        char    ps;
        int     ch;
        struct  tty   *iptr;

        disable(ps);
        iptr = &tty[devptr->dvminor];
        wait(iptr->isem);               /* wait for a character in buff */
        ch = iptr->ibuff[iptr->itail++];
        if (iptr->itail == IBUFLEN)     /* IBUFLEN == 128               */
                iptr->itail = 0;
        restore(ps);
        return(ch);
}
```

Upper-Half TTY Input Routines – ttyread

- `ttyread` has **three parameters**:
  1. a device switch table entry to retrieve the minor device number,
  2. the address of the buffer to which read input data should be output, and
  3. the number of characters to read `nread`.
- **Reading all characters** in `ibuf` is encoded by *number of characters to read equals zero.*
- Two cases are treated:
  1. `nread` $\leq$ number of characters in **ibuf**
  2. `nread` $>$ number of characters in **ibuf**

Upper-Half TTY Input Routines – ttyread

```
/* ttyread.c - ttyread, readcopy */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyread - read one or more characters from a tty device
 *------------------------------------------------------------------------
 */
ttyread(devptr, buff, count)
struct  devsw   *devptr;
int count;
char *buff;
{
        char ps;
        register struct tty *iptr;
        int avail, nread;
```

Upper-Half TTY Input Routines – ttyread

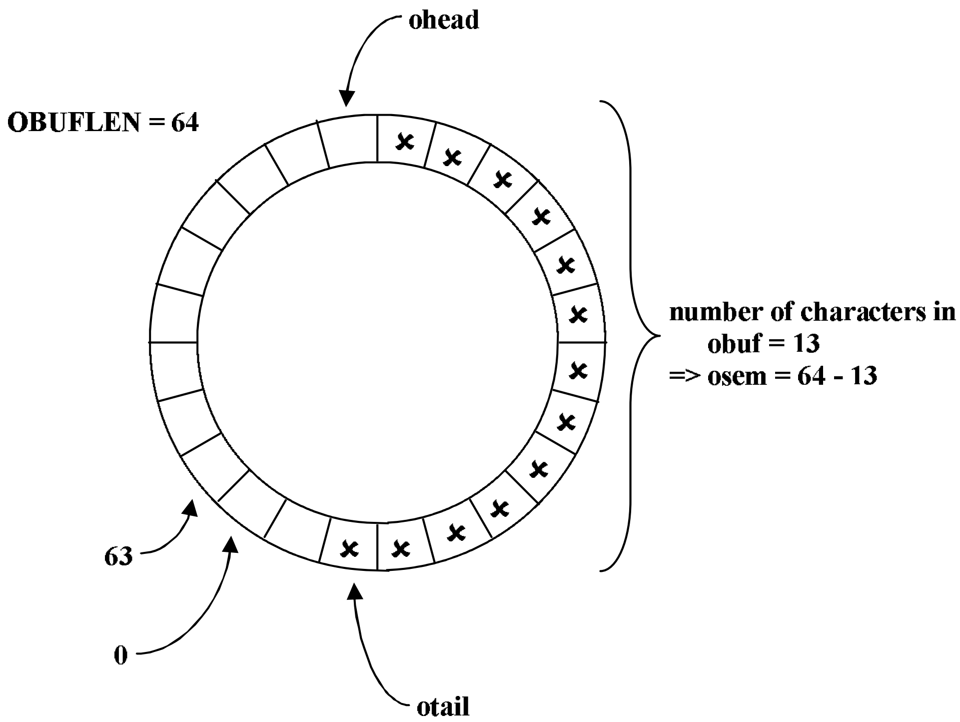
ttyread continued:

```
        if (count < 0)
                return(SYSERR);
        disable(ps);
        avail=scount( (iptr= &tty[devptr->dvminor])->isem );
        if ( (count = (count==0 ? avail : count)) == 0) {
                restore(ps);
                return(0); // ttyread (..., 0) invokes reading
                           // ALL characters available in the buffer!
        }
        nread = count;
        if (count <= avail)
                readcopy(buff, iptr, count);
                   // if there are enough charachters in the buffer
                   // to satisfy the read request, copy them directly
                   // to the user's buffer
        else {
                if (avail > 0) {
                        readcopy(buff, iptr, avail);
                        buff += avail;
                        count -= avail;
                }
                for ( ; count>0 ; count--)
                        *buff++ = ttygetc(devptr);
                   // when the user requests more characters than are
                   // present in the buffer, issue additional calls
                   // of ttygetc
        }
        restore(ps);
        return(nread);
}
```

Upper-Half TTY Input Routines – ttyread

```
/*------------------------------------------------------------------------
 *  readcopy - high speed copy procedure used by ttyread
 *------------------------------------------------------------------------
 */
LOCAL readcopy(buff, iptr, count)
register char *buff;
struct tty *iptr;
int count;
{
        register char *qtail, *qend, *uend /* copy loop variables */

        qtail = &iptr->ibuff[iptr->itail];
        qend  = &iptr->ibuff[IBUFLEN];
        uend = buff + count;
        while ( buff < uend ) {
                *buff++ = *qtail++;
                if ( qtail >= qend )
                        qtail = iptr->ibuff;
        }
        iptr->itail = qtail-iptr->ibuf;
        sreset(iptr->isem, scount(iptr->isem)-count);
}
```

12.6 Upper-Half TTY Output Routines



- `ttyputc`: note `iptr->ocrlf` flag indicates whenever after `ch == NEWLINE` the return-character should be inserted by `ttyputc(devptr, return)`.
- Just before `ttyputc` returns, `ttyputc` enables output - transmitter interrupts in the SLU, to guarantee that lower-half will transfer characters inserted into `obuf` (by setting its control bit to 1; if it is 1 already this causes no harm)
- `ctstat` – see page 170 – identifies the transmitter control and status field `XCSR` of the SLU-register – see page 29. Bit 6 of `XCSR` set to 1 enables it to interrupt when transmitter becomes idle.

Upper-Half TTY Output Routines

- Structure `csr` defines the layout of the SLU device registers in the address space.
- To move data into SLU device registers, `ttyputc` needs to know the address assigned to the SLU upon hardware installation. This address is put at system configuration time in the device switch table `devtab`.
- During device initialization this is copied into the TTY control block field `ioaddr`, by calling `ttyinit` (see page 182) via calling `init(0)`.
- Using the device register address `ioaddr` as pointer, `ttyputc` "writes into the `ctstat` field of the structure" it points to. Since this address is "beyond real memory", it is computed by the hardware and the device output interrupt enable bit is assigned to 1 by assigning `SLUENABLE` $= 0100_8 = 1\,000\,000_2$.
- This is the only way upper-half routines have to awaken lower-half routines to initiate character transfer: By setting bit nr. 6 to 1 whenever current output finishes, SLU will interrupt; calling the interrupt dispatcher, which in its turn calls the lower-half output routine, which finds a character in the buffer and transmits it.

Upper-Half TTY Output Routines – `slu.h`

```
/* slu.h */

/* standard serial line unit device constants */

#define SLUENABLE      0100          /* device interrupt enable bit  */
#define SLUREADY       0200          /* device ready bit             */
#define SLUDISABLE     0000          /* device interrupt disable mask*/
#define SLUTBREAK      0001          /* transmitter break-mode bit   */
#define SLUERMASK      0170000       /* mask for error flags on input*/
#define SLUCHMASK      0377          /* mask for input character     */

/* SLU device register layout and correspondence to vendor's names     */

struct  csr     {
        int     crstat;         /* receiver control and status  (RCSR)  */
        int     crbuf;          /* receiver data buffer         (RBUF)  */
        int     ctstat;         /* transmitter control & status (XCSR)  */
        int     ctbuf;          /* transmitter data buffer      (XBUF)  */
};
```

- `SLUENABLE` $= 0100_8 = 100\,0000_2$
  (device interrupt enable bit)
- `SLUREADY` $= 0200_8 = 1000\,0000_2$
  (device ready bit)
- `SLUERMASK` $= 0170000_8 = 1111\,0000\,0000\,0000_2$
  (mask for input errors)
- `SLUCHMASK` $= 0377_8 = 0000\,0000\,1111\,1111_2$
  (mask for input character)

Upper-Half TTY Output Routines – ttyputc

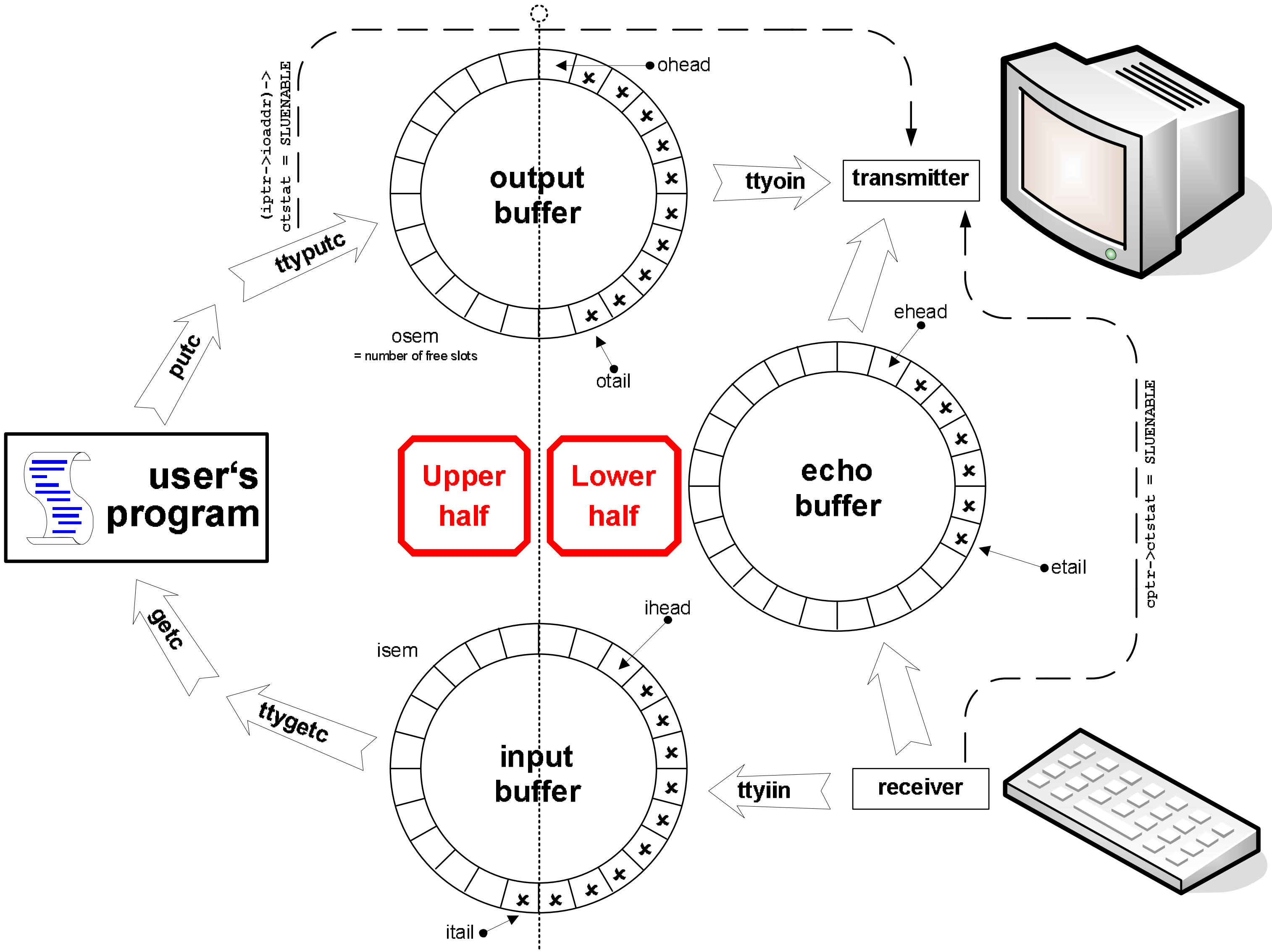
```
/* ttyputc.c - ttyputc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyputc - write one character to a tty device
 *------------------------------------------------------------------------
 */
ttyputc(devptr, ch )
struct  devsw   *devptr;
char    ch;
{
        struct  tty   *iptr;
        char    ps;

        iptr = &tty[devptr->dvminor];
        if ( ch==NEWLINE && iptr->ocrlf )
                ttyputc(devptr,RETURN);
        disable(ps);
        wait(iptr->osem);               /* wait for space in queue      */
        iptr->obuff[iptr->ohead++] = ch;
        if (iptr->ohead >= OBUFLEN)
                iptr->ohead = 0;
        (iptr->ioaddr)->ctstat = SLUENABLE;
        restore(ps);
        return(OK);
}
```

Upper-Half TTY Output Routines

```
/* ttywrite.c - ttywrite, writcopy */

#include <conf.h> <kernel.h> <tty.h> <io.h> <slu.h>

/*------------------------------------------------------------------------
 *  ttywrite - write one or more characters to a tty device
 *------------------------------------------------------------------------
 */
ttywrite(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        register struct tty *ttyp;
        int ncopied;
        char ps;

        if (count < 0)
                return(SYSERR);
        disable(ps);
        ttyp = &tty[devptr->dvminor];
        if ((avail=scount(ttyp->osem)) >= count) /* #freespace >= count */
        {       writcopy(buff, ttyp, count);
                (ttyp->ioaddr)->ctstat = SLUENABLE;
        } else {
                if (avail > 0) {
                        writecopy(buff, ttyp, avail);
                        buff += avail;
                        count -= avail;
                }
                for (; count>0 ; count--)
                        ttyputc(devptr, *buff++);
        }
        restore(ps);
        return(OK);
}
```

Upper-Half TTY Output Routines – ttyputc

```
/*------------------------------------------------------------------------
 *  writcopy - high-speed copy from user's buffer into system buffer
 *------------------------------------------------------------------------
 */
LOCAL writcopy(buff, ttyp, count)
char    *buff;
struct  tty *ttyp;
int     count;
{
        register char   *qhead, *qend, *uend;

        qhead = &ttyp->obuff[ttyp->ohead];
        qend  = &ttyp->obuff[OBUFLEN];
        uend  = buff + count;
        while (buff < uend) {
                *qhead++ = *buff++;
                if ( qhead >= qend )
                        qhead = ttyp->obuff;
        }
        }
        ttyp->ohead = qhead - ttyp->obuff;
        sreset(ttyp->osem, scount(ttyp->osem)-count);
}
```


output
buffer
ohead
osem
= number of free slots
otail
ttyputc
(iptr->ioaddr)->
ctstat = SLUENABLE
ttyoin
transmitter
putc
user's
program
Upper
half
Lower
half
echo
buffer
ehead
etail
optr->ctstat = SLUENABLE
getc
ttygetc
isem
ihead
input
buffer
itail
ttyiin
receiver

12.7 Lower-Half TTY Driver Routines

- The lower-half of the TTY driver performs the real work of operating the device and fielding interrupts. It consists of two procedures: The input interrupt routine, `ttyiin`, and the output interrupt routine, `ttyoin` – first, `ttyoin` is discussed.
- The interrupt dispatcher calls `ttyoin` with interrupts disabled whenever the output device is idle (because only then an interrupt is generated).
- The initilization routine `iosetvec` (page 157) has the dispatcher pass `ttyoin` one argument – the address of TTY control block for the interrupting device `iptr`.
- Processing an output interrupt consists of three cases:
  - either the driver transmits a character from the echo buffer,
  - or a character from the output buffer,
  - or it does nothing at all.

Lower-Half TTY Driver Routines: The Echo Buffer

- If the echo buffer is nonempty, `ttyoin` takes a character from it and deposits the character in the device tansmitter register (see page 170 modeled after the figure on page 29).
- The `ttyoin` checks `oheld` to see wheter output has been stopped (why this is so is explained when discussing `ttyiin`).
  - The input interrupt handler `ttyiin` sets `oheld` when it detects the "stop" character, and cleans `oheld` when it detects a "start" character.
  - When `ttyoin` finds `oheld` set, it turns off interrupts without sending more characters. Another routine, `ttyiin`, must eventually clear `oheld` and enable output device interrups again.
- The lower-half my find the output buffer empty when an interrup occurs if it has sent the last waiting character to the device already. Then `ttyoin` cleans the device interrupt and returns.
- Normal output processing consists of selecting a character from the output buffer, `obuff`, and depositing it in the device transmitter register.

12.7.1 Watermarks and delayed signals

- `ttyoin` uses a technique called watermark processing to **minimize overhead in the interaction between upper and lower halves of the driver**.
- Effect: If free slots $\leq$ `OBMINSP` (= 20), then `ttyiin` stops calling signal (`iptr->osem`) 20 times, only emptying the buffer each time by one character.

  Once the buffer has been emptied of 20 characters, `osem` is increased by 20 (i.e., logically `signal(osem)` is called 20 times) so that 20 more free slots are freed for the user program to continue outputting.

Lower-Half TTY Driver Routines

- **Motivation**: Assume for the movement that the lower-half calls `signal` each time it removes a character from the buffer. Because the processor generates characters **much faster** than the SLU can transmit them, the output buffer usually remains full with a process waiting for the output semaphore.
- When `ttyoin` then removes a character, it signals the semaphore, causing the first waiting process to deposit a character. Because programs often write more than one character at a time, the process that was waiting usually produces quickly another character and ends up waiting for the semaphore again.
- The problem is that this form of reschedlung is rather **expensive**: executing it on every character interrupt deprives other ready processes of CPU time.

Lower-Half TTY Driver Routines

- There are two of them:
  - lower-half input-interrupt routine `ttyiin` (code on later slide).
  - lower-half output-interrupt routine `ttyoin`: Recall that `iptr` points to address of TTY-control block for the interrupting device; also recall that the interrupt dispatcher calls `ttyoin` with interrupts disabled wherever output device driver is idle (because only the interrupt is generated).
- A `ttyoin(iptr)`-call considers three cases:
  - a character from echo buffer `ebuf` must be transmitted
  - the driver should do something because output is currently interrupted, as indicated by `oheld`, until line is full
  - a character from the output buffer is transmitted – two ways to do this:
    * There is more than `OBMINSP` (= 20) free space in `obuf`. Character is put in device register and `signal(osem)` is called to free **one buffer slot**.
    * There is less than `OBMINSP` free space in `obuf`. Chars are still transmitted, but `signal(osem)` is delayed to halt (potentially) calls of `ttyputc`.

Lower-Half TTY Driver Routines

```
/* ttyoin.c - ttyoin */

#include <conf.h> <kernel.h> <tty.h> <io.h> <slu.h>

/*------------------------------------------------------------------------
 *  ttyoin  --  lower-half tty device driver for output interrupts
 *------------------------------------------------------------------------
 */
INTPROC ttyoin(iptr)
        register struct tty   *iptr;
{
        register struct csr     *cptr;
        int     ct;

        cptr = iptr->ioaddr;
        if (iptr->ehead != iptr->etail) {
                cptr->ctbuf = iptr->ebuff[iptr->etail++];
                if (iptr->etail >= EBUFLEN)
                        iptr->etail = 0;
                return;
        }
        if (iptr->oheld) {                      /* honor flow control   */
                cptr->ctstat = SLUDISABLE;
                return;
        }
        if ((ct=scount(iptr->osem)) < OBUFLEN) {
                cptr->ctbuf = iptr->obuff[iptr->otail++];
                if (iptr->otail >= OBUFLEN)
                        iptr->otail = 0;
                if (ct > OBMINSP)
                        signal(iptr->osem);
                else if ( ++(iptr->odsend) == OBMINSP) {
                        iptr->odsend = 0;
                        signaln(iptr->osem, OBMINSP);
                }
        } else
                cptr->ctstat = SLUDISABLE;
}
```

12.7.2 Lower-Half Input Processing

- Input interrupt processing is **more complex** because it caters for **character echo, line editing, processing of input errors** and **buffer overflow**.
- It operates in one of three modes, specified by `imode` field in TTY control block:
  - `raw`: it accumulates characters in `ibuff` without further processing.
  - `cooked`: it does **character echo**; honors suspend or restart output; **accumulates full lines before passing them on to upper-half routines**; honors **input editing** by erasing previous characters or killing entire lines.
  - `cbreak`: honors **control chars** but does **no line editing**.

1. In `raw` mode:
   - `ttyiin` takes **input character** from SLU-receiver unit
   - deposits it in `ibuf`,
   - signals `isem`,
   - if no space remains, `ttyiin` throws character away; error is reported by setting 8^{th} character bit to 1.

Lower-Half Input Processing

2. In `cbreak` mode,
   - `ttyiin` takes input character from SLU-receiver unit,
   - **maps return to newline**,
   - handles output flow control:
     - field `oflow` determines whether driver honours flow control,
     - if it does, driver **suspends** output by setting `oheld` when receiving `ostop` character,
     - starting output when receiving `ostart` character,
     - `ostart` and `ostop` are control characters, not placed in `ibuf` for processing by upper-half input routine `ttygetc`,
     - reports `ibuffer` overflow by sending `ifullc` character in echo buffer, **ringing a bell**, and does character echo.

Lower-Half Input Processing

3. In `cooked` mode, line editing is also performed:
   - accumulating lines in `ibuf` using var `icursor` for counting chars on current line
   - erasing chars when `ierasec` arrives (decrementing `icursor` by 1)
   - when receiving line-kill character `ikillc`, `ttyiin` backs over all chars on current line by `icursor := 0`,
   - calling `erase1` procedure to obliterate chars from display
   - upon receiving `NEWLINE` or `RETURN` char, signalling `isem` `icursor` times

Lower-Half Input Processing

```
/* ttyiin.c ttyiin, erase1, eputc, echoch */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyiin  --  lower-half tty device driver for input interrupts
 *------------------------------------------------------------------------
 */
INTPROC ttyiin(iptr)
        register struct tty     *iptr;  /* pointer to tty block         */
{
        register struct csr *cptr;
        register int    ch;
        Bool    cerr;
        int     ct;

        cptr = iptr->ioaddr;
        if (iptr->imode == IMRAW) {
                if (scount(iptr->isem) >= IBUFLEN) {
                        ch = cptr->crbuf;
                        return;
                }
                if ((ch=cptr->crbuf)&SLUERMASK) /* character error        */
                        iptr->ibuff[iptr->ihead++] = (ch&SLUCHMASK)|IOCHERR;
                else                            /* normal read complete   */
                        iptr->ibuff[iptr->ihead++] = ch & SLUCHMASK;
                if (iptr->ihead >= IBUFLEN)
                        iptr->ihead = 0;
                signal(iptr->isem);
```

Lower-Half Input Processing

```
} else {                                /* cbreak | cooked mode */
        cerr = ((ch=cptr->crbuf)&SLUERMASK) ? IOCHERR : 0;
        ch &= SLUCHMASK;
        if ( ch == RETURN && iptr->icrlf )
                ch = NEWLINE;
        if (iptr->oflow) {
                if (ch == iptr->ostart) {
                        iptr->oheld = FALSE;
                        cptr->ctstat = SLUENABLE;
                        return;
                }
                if (ch == iptr->ostop) {
                        iptr->oheld = TRUE;
                        return; }
        }
        iptr->oheld = FALSE;
        if (iptr->imode == IMCBREAK) {          /* cbreak mode  */
                if (scount(iptr->isem) >= IBUFLEN) {
                        eputc(iptr->ifullc,iptr,cptr);
                        return;
                }
                iptr->ibuff[iptr->ihead++] = ch | cerr;
                if (iptr->ihead >= IBUFLEN)
                        iptr->ihead = 0;
                if (iptr->iecho)
                        echoch(ch,iptr,cptr);
                if (scount(iptr->isem) < IBUFLEN)
                        signal(iptr->isem);
        } else {                                /* cooked mode  */
                if (ch == iptr->ikillc && iptr->ikill) {
                        iptr->ihead -= iptr->icursor;
                        if (iptr->ihead < 0)
                                iptr->ihead += IBUFLEN;
                        iptr->icursor = 0;
                        eputc(RETURN,iptr,cptr);
                        eputc(NEWLINE,iptr,cptr);
                        return;
                }
```

Lower-Half Input Processing

```
                                if (ch == iptr->ierasec && iptr->ierase) {
                                        if (iptr->icursor > 0) {
                                                iptr->icursor--;
                                                erase1(iptr,cptr);
                                        }
                                        return;
                                }
                                if (ch == NEWLINE || ch == RETURN) {
                                        if (iptr->iecho)
                                                echoch(ch,iptr,cptr);
                                        iptr->ibuff[iptr->ihead++] = ch | cerr;
                                        if (iptr->ihead >= IBUFLEN)
                                                iptr->ihead = 0;
                                        ct = iptr->icursor+1; /* +1 for \n or \r*/
                                        iptr->icursor = 0;
                                        signaln(iptr->isem,ct);
                                        return;
                                }
                                ct = scount(iptr->isem);
                                ct = ct < 0 ? 0 : ct;
                                if ((ct + iptr->icursor) >= IBUFLEN-1) {
                                        eputc(iptr->ifullc,iptr,cptr);
                                        return;
                                }
                                if (iptr->iecho)
                                        echoch(ch,iptr,cptr);
                                iptr->icursor++;
                                iptr->ibuff[iptr->ihead++] = ch | cerr;
                                if (iptr->ihead >= IBUFLEN)
                                        iptr->ihead = 0;
                        }
                }
        }
```

Lower-Half Input Processing

```
/*------------------------------------------------------------------------
 *  erase1  --  erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
LOCAL erase1(iptr,cptr)
        struct  tty   *iptr;
        struct  csr    *cptr;
{
        char    ch;

        if (--(iptr->ihead) < 0)
                iptr->ihead += IBUFLEN;
        ch = iptr->ibuff[iptr->ihead];
        if (iptr->iecho) {
                if (ch < BLANK || ch == 0177) {
                        if (iptr->evis) {
                                eputc(BACKSP,iptr,cptr);
                                if (iptr->ieback) {
                                        eputc(BLANK,iptr,cptr);
                                        eputc(BACKSP,iptr,cptr);
                                }
                        }
                        eputc(BACKSP,iptr,cptr);
                        if (iptr->ieback) {
                                eputc(BLANK,iptr,cptr);
                                eputc(BACKSP,iptr,cptr);
                        }
                } else {
                        eputc(BACKSP,iptr,cptr);
                        if (iptr->ieback) {
                                eputc(BLANK,iptr,cptr);
                                eputc(BACKSP,iptr,cptr);
                        }
                }
        } else
                cptr->ctstat = SLUENABLE;
}
```

Lower-Half Input Processing

```
/*------------------------------------------------------------------------
 *  echoch  --  echo a character with visual and ocrlf options
 *------------------------------------------------------------------------
 */
LOCAL echoch(ch, iptr, cptr)
        char    ch;             /* character to echo                    */
        struct  tty   *iptr;    /* pointer to I/O block for this devptr */
        struct  csr     *cptr;  /* csr address for this devptr          */
{
        if ((ch==NEWLINE||ch==RETURN)&&iptr->ecrlf) {
                eputc(RETURN,iptr,cptr);
                eputc(NEWLINE,iptr,cptr);
        } else if ((ch<BLANK||ch==0177) && iptr->evis) {
                eputc(UPARROW,iptr,cptr);
                eputc(ch+0100,iptr,cptr);       /* make it printable    */
        } else {
                eputc(ch,iptr,cptr);
        }
}

/*------------------------------------------------------------------------
 *  eputc - put one character in the echo queue
 *------------------------------------------------------------------------
 */
LOCAL eputc(ch,iptr,cptr)
        char    ch;
        struct  tty   *iptr;
        struct  csr     *cptr;
{
        iptr->ebuff[iptr->ehead++] = ch;
        if (iptr->ehead >= EBUFLEN)
                iptr->ehead = 0;
        cptr->ctstat = SLUENABLE;
}
```

12.8 TTY Control Block Initialization

Procedure `ttyinit` initializes the tty control block and interrupt vectors, given a pointer to the `devtab` structure for that device.

- It calls `iosetvec`: sets interrupt vector and fills in the interrupt dispatch table, passing the address of the appropriate TTY control block as argument to `ttyiin` and `ttyoin`.
- It initializes the control block for `raw` or `cooked` mode, depending on whether or not the line corresponds to the console terminal.
- It creates input and output buffer semaphores and initialize `ibuf`, `obuf`, `ebuf`.
- After parameters, buffers, and interrupt vector have been set, it
  - clears receiver buffer in hardware,
  - enables recipient of interrupts and
  - disables transmitter interrupts.

TTY Control Block Initialization

```
/* ttyinit.c - ttyinit */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyinit - initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
ttyinit(devptr)
        struct  devsw   *devptr;
{
        register struct tty *iptr;
        register struct csr *cptr;
        int      junk, isconsole;

        /* set up interrupt vector and interrupt dispatch table */

        iptr = &tty[devptr->dvminor];
        iosetvec(devptr->dvnum, iptr, iptr);
```

TTY Control Block Initialization

```
        devptr->dvioblk = iptr;                 /* fill tty control blk */
        isconsole = (devptr->dvnum == CONSOLE); /* make console cooked  */
        iptr->ioaddr = devptr->dvcsr;           /* copy in csr address  */
        iptr->ihead = iptr->itail = 0;          /* empty input queue    */
        iptr->isem = screate(0);                /* chars. read so far=0 */
        iptr->osem = screate(OBUFLEN);          /* buffer available=all */
        iptr->odsend = 0;                       /* sends delayed so far */
        iptr->ohead = iptr->otail = 0;          /* output queue empty   */
        iptr->ehead = iptr->etail = 0;          /* echo queue empty     */
        iptr->imode = (isconsole ? IMCOOKED : IMRAW);
        iptr->iecho = iptr->evis = isconsole;   /* echo console input   */
        iptr->ierase = iptr->ieback = isconsole;/* console honors erase */
        iptr->ierasec = BACKSP;                 /*  using ^h            */
        iptr->ecrlf = iptr->icrlf = isconsole;  /* map RETURN on input  */
        iptr->ocrlf = iptr->oflow = isconsole;  /* map RETURN on output */
        iptr->ikill = isconsole;                /* set line kill == @   */
        iptr->ikillc = ATSIGN;
        iptr->iintr = FALSE;
        iptr->ostart = STRTCH;
        iptr->ostop = STOPCH;
        iptr->icursor = 0;
        iptr->ifullc = TFULLC;
        cptr = (struct csr *)devptr->dvcsr;
        junk = cptr->crbuf;                     /* clear receiver and   */
        cptr->crstat = SLUENABLE;               /* enable in. interrupts*/
        cptr->ctstat = SLUDISABLE;              /* disable out.   "     */
}
```

12.9 Device Driver Control

- `ttycntl` implements the higher-level device-independent I/O procedure `control(descrp,func,addr,addr2)` with `descrp` as an index into the device switch table and `func` setting the relevant mode.
- `ttycntl` controls a TTY device by setting its modes, e.g.,
  - `func == TCSETBRK` sets the “break” bit in the SLU transmitter by placing the line in a break state,
  - `func == TCRSTBRK` turns off the break bit in the SLU transmitter and returns the line to normal processing,
  - `func == TCMODER` sets mode field `imode` to `raw`,
  - `func == TCMODEC` sets mode field `imode` to `cooked`,
  - `func == TCMODEK` sets mode field `imode` to `cbreak`,
  - `func == TCECHO` (`TCECHON`) control character echo
  - `func == TCICHARS` allows the user to query driver
  - `func =` ...

Device Driver Control

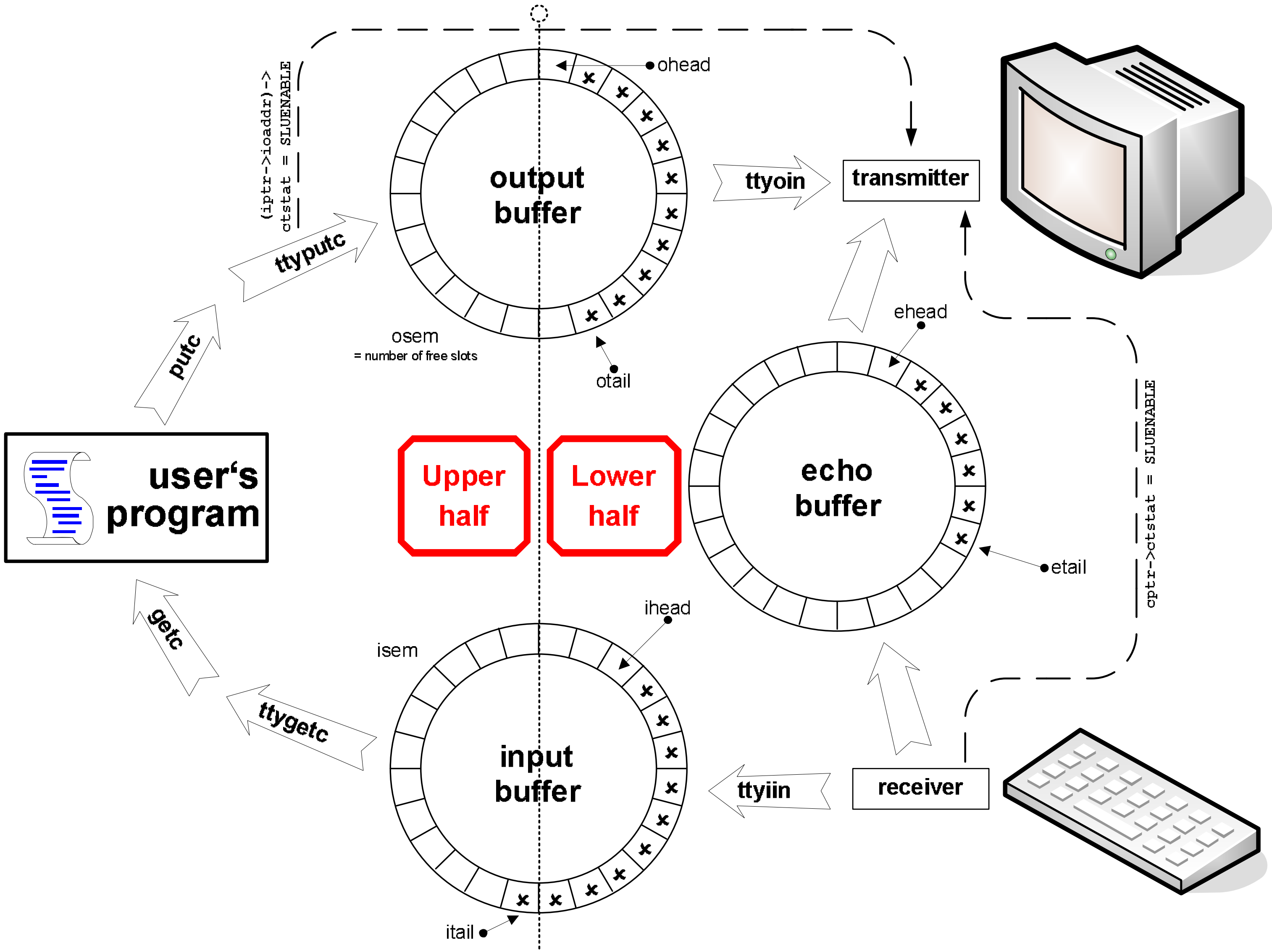
```
/* ttycntl.c - ttycntl */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttycntl  -  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
ttycntl(devptr, func)
struct  devsw   *devptr;
int func;
{
        register struct tty *ttyp;
        char    ch;
        char    ps;

        ttyp = &tty[devptr->dvminor];
```

Device Driver Control

```
        switch ( func ) {
        case TCSETBRK:
                ttyp->ioaddr->ctstat |= SLUTBREAK;
                break;
        case TCRSTBRK:
                ttyp->ioaddr->ctstat &= ~SLUTBREAK;
                break;
        case TCNEXTC:
                disable(ps);
                wait(ttyp->isem);
                ch = ttyp->ibuff[ttyp->itail];
                restore(ps);
                signal(ttyp->isem);
                return(ch);
        case TCMODER:
                ttyp->imode = IMRAW;
                break;
        case TCMODEC:
                ttyp->imode = IMCOOKED;
                break;
        case TCMODEK:
                ttyp->imode = IMCBREAK;
                break;
        case TCECHO:
                ttyp->iecho = TRUE;
                break;
        case TCNOECHO:
                ttyp->iecho = FALSE;
                break;
        case TCICHARS:
                return(scount(ttyp->isem));
        default:
                return(SYSERR);
        }
        return(OK);
}
```

12.10 Summary

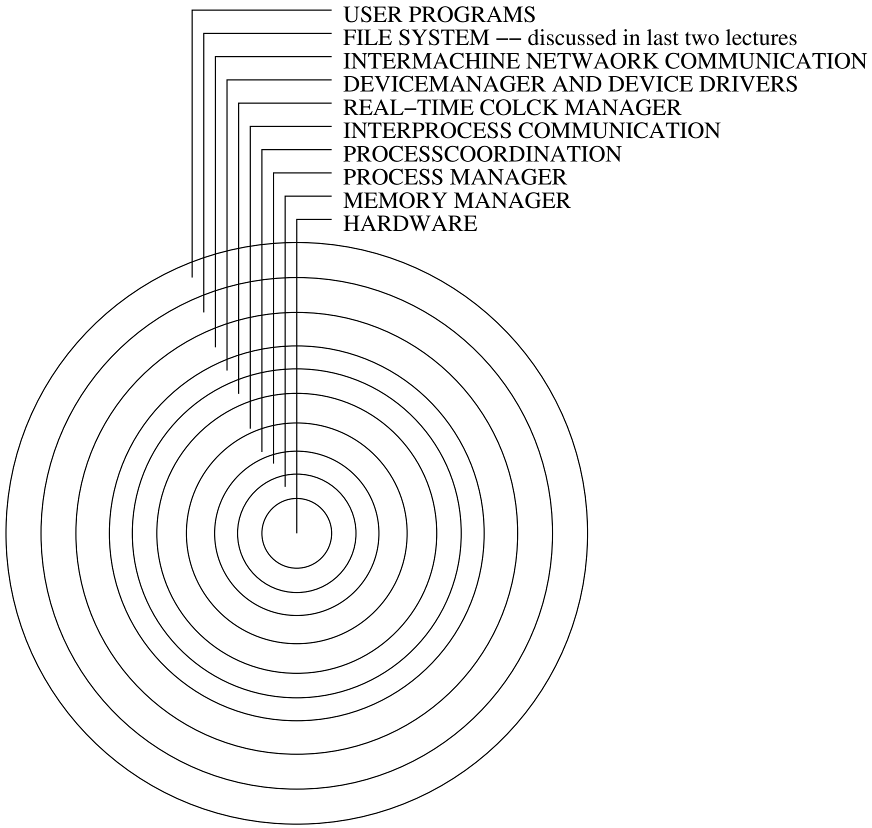
- A device driver consists of a set of procedures to control a peripheral hardware device. The driver routines are partitioned into two halves:
  - an upper half that contains the routines called from user programs, and
  - a lower half that contains the routines that handle device interrupts.
- The TTY driver upper-half routines implement `read`, `write`, `getc`, `putc` and control characters.
- The lower-half output procedure, called whenever the transmitter interrupts, sends characters from the queue of those waiting to be sent (. . . `osem` counts free slots. . .).
- When an input character arrives, the lower-half deposits it in the queue of incoming characters where it can be retrieved by the upper half.
- The driver also contains an initialization procedure that fills in the device control block and interrupt vectors when the system starts.



(iptr->ioaddr)->
ctstat = SLUENABLE
ohead
output buffer
ttyoin
transmitter
ttyputc
putc
osem
= number of free slots
otail
ehead
user's program
Upper half
Lower half
echo buffer
etail
optr->ctstat = SLUENABLE
getc
isem
ihead
ttygetc
input buffer
ttyiin
receiver
itail

13 System Initialization

Till now six layers have been constructed:



Introduction

- Initialization is the **last step of the design process**. Why should it be taken up here?
- Because many microcomputer applications require no more than **we have at hand** – a process manager to support concurrent computation, and the means to transmit info to and from running programs. The rest is just icing on the (layered) cake.
- The discussion of initialization begins with a consideration of **system termination**.

Introduction

- Many microcomputers require no more than what's been discussed in chapters 1 – 12: a process manager to support **concurrent computations**, and the means to transform information to and from **running programs**.
- It makes sense to consider initialization now, because subsequent chapters describe pieces of the system that are **more or less optional**.
- Chapters 1 – 13 describe a so-called **micro-kernel (system kernel)**, upon which basis a more complete and user-friendly operating system can be built. So a micro-kernel represents a "minimal" system – **the rest ist just icing on the (layered) cake**.

13.1 Starting from scratch

- A **crash** occurs when hardware executes an **invalid operation** caused because code or data in the operating system has been destroyed. A crash also means that the **contents of memory** have been **corrupted or lost**!
- How can a machine, devoid of valid programs, spring into action and begin executing again? **It cannot!**
- Somehow a program must be **deposited in memory** before the machine can start. On the oldest computers this happened **by hand**, using switches.
- Later, **standard keyboards** built to that purpose, i.e., special terminals, now micro- and mini-computers, were used to load the **initial program** from tape or disk storage attached to the micro.
- The microcomputer itself has its initial program in **read-only memory** encoded, so it can **restart** without help from other machines.

Starting from scratch

- Once the initial program has been loaded, the CPU can execute the startup program which reads a **larger program** (usually from a special location on a specific disk).
- Then the CPU branches to a larger program which reads the **entire operating system** into memory, and **branches to its beginning**. This process is called **rebooting** the system.
- When the CPU begins executing the operating system code, the system must **initialize devices and system data structures**: process table `proctab`, semaphore table `semaph[]`, the ready list, interrupt dispatch table `intmap[]`, device control blocks, memory, the sleep queue, and so on.
- It must also check for, and repair, **damage** to the linked lists and disk pointers in the system.

13.2 Booting Xinu

Xinu is **downloaded** from another machine called its **host**:

- The host computer generates a **break condition** to halt the 11/2 processor by communicating over the console terminal line (described in the last paragraph of §2.1.11 on page 38).
- The 11/2 responds in **Octal Debugging Technique** (ODT) mode: It sends a **prompt** and recognizes **commands to display** and **change memory locations and registers**.
- The host loads an **initial program** in the 11/2 memory starting at location zero, and starts the 11/2 to execute it.

2.2 Adressing a Serial Line Unit

- Hardware on the interface board maps the four SLU registers into four contigous words in the 11/2's address space. These device registers are not actually storage locations – **see page 29**. . .

- I/O can be viewed as a **side effect of manipulating certain locations in the address space**, e.g., writing a character to the `XBUF` register address causes the SLU to capture that character and start transmitting it on the serial output line.
- Once the board accepts the character, it operates **independent** of the CPU; processing continues while the transmitter sends the character.

2.3 Polled vs. Interrupt-Driven I/O

- How do the CPU and SLu **coordinate**? Coordination is important because the CPU will interfere with the SLU if it accesses the receiver or transmitter data buffer while an I/O operation is in progress.
- I.e., how does the CPU **know** when a character has been received or when the transmitter is idle and ready to send a character? It uses one of two techniques: **Polled** or **interrupt-driven I/O**.
- Polling requires the CPU to **check the device repeatedly** until it finds that a character is waiting. **Polled transmission** uses the same approach – read page 30 of Comer. An example of polled output is the routine `kprintf` in chapter 19 that is used for diagnostic output.
- Polling is **impractical** (except at system boot-up) because it requires the CPU to **attend services constantly**.

Booting Xinu

- The initial boot program (sent from host to the 11/2) **reads characters**, using polled I/O (also described in §2.1.11 – read from third paragraph onwards on page 30 of Comer) and deposits them in memory starting at the highest location.
- When finished, the host sends a **break**, forcing the 11/2 into ODT mode.
- When ODT responds, the host starts the 11/2 executing the second sent **boot program**.
- The host and the second boot program **communicate**, with the host sending "packets" of bytes (one-at-a-time), and the boot program acknowledging receipt or requesting transmission.
- The host either tells the second bootstrap to branch to the **start of Xinu** or to halt and **wait for ODT commands**.
- In most cases, the host directs the second bootstrap to **branch to Xinu** and the 11/2's CPU begins executing start (at 01000_8).

13.3 System startup

- The startup program described below creates the **environment** which a C-program expects; then it jumps to the C-procedure `nulluser`. Its start is located at 01000_8 – the first location beyond the interrupt vectors.
- Basically, startup **disables interrupts** and **establishes an empty stack**.
- Since **memory size is unknown** (the stack should start at the high-end of memory and grows downwards), **finding the memory size is done by calling procedure** `sizmem`.
- To guarantee that `stackptr` SP is valid while `sizmem` runs, startup sets `stackptr` to an **auxiliary stack** called `kernstk` (declared in file `panic.s` in §19), i.e. 300 decimal bytes, set up in the data area, and used when usual stack cannot be used.
- **As its last instruction, start jumps to** `nulluser`.

System startup

```
/* startup.s - start */

DISABLE     =     340               / PS to disable interrupts

/*------------------------------------------------------------------------
/*  Xinu system entry point -- first location beyond interrupt vectors
/*------------------------------------------------------------------------
      .global      start
start:
      mtps  $DISABLE          / Disable interrupts
      mov   $kernstk,sp       / temporary stack pointer (kernstk decl. on
      jsr   pc,sizmem         / _maxaddr set to max address      page 352)
      mov   _maxaddr,sp       / switch stack to high memory
      reset                   / reset bus
      clr   r5                / clear initial r5 for debugging
      clr   r4
      jmp   _nulluser         / Jump to C startup routine
```

13.4 Finding the size of memory

- The highest valid memory address is found by encoding the **highest possible memory address** in `_maxaddr` and then **referencing** successively smaller addresses **until one is found causing no exception**.
- To **detect an invalid address**, `sizmem` relies on the hardware – when **nonexistent memory** is referenced, the **resulting error** causes the CPU to **trap through the exception vector** at location 4 in memory (called the **bus exception**).
- `sizmem` sets an address in the exception vector so each time it references an illegal address the hardware **branches to location** `siztrap`.
- `sizmem` should not destroy the **original data** in the bus exception vector, so it saves its original contents upon entry, by pushing them onto the stack, and restores them later.
- In the code, the Label `sizloop` marks the point at which `sizmem` **tests a memory location**. An exception trap occurs on each reference to an address “beyond real men”, causing a branch to `siztrap`, until a valid address has been reached.
- When a reference succeeds, `sizmem` saves the address in **global variable** `maxaddr`, making it available to other system routines.

Finding the size of memory

```
/* sizmem.s - sizmem */

MAXADDR     =     157776            / Maximum possible mem. address
DISABLE     =     340               / PS to     disable     interrupts
ENABLE      =     000               / PS to     enable interrupts
EXCPPC      =     4                 / exception vector address for
EXCPPS      =     6                 / "memory out of range" errors

/*------------------------------------------------------------------------
/*  sizmem  --  size memory, placing highest valid address in _maxaddr
/*------------------------------------------------------------------------
      .globl      sizmem
sizmem:
      mfps  -(sp)             / save incoming PS
      mtps  $DISABLE          / disable interrupts
      mov   r0,-(sp)          / save registers used
      mov   *$EXCPPC,-(sp)    / save old contents of
      mov   *$EXCPPS,-(sp)    /  exception vector
      mov   $siztrap,*$EXCPPC / set up vector to catch memory
      mov   $DISABLE,*$EXCPPS / exceptions; disable interrupts
      mov   $MAXADDR,r0       / set r0 to highest possible loc
sizloop:
      mov   (r0),(r0)         / reference what r0 points to
      mov   r0,_maxaddr       / no interrupt - memory exists.
      mov   (sp)+,*$EXCPPS    / restore exception vector
      mov   (sp)+,*$EXCPPC
      mov   (sp)+,r0          / restore r0
      mtps  (sp)+             / restore PS
      rts   pc                / return to caller
siztrap:
      add   $4,sp             / pop interrupted PC and PS
      sub   $2,r0             / move to next lower address
      jbr   sizloop           / try again
```

13.5 Initializing system data structures

- `startup.s` and `sizmem.s` do nothing more than **create a valid run-time environment** for C by **setting the `stackptr` to the highest valid memory address**, and **disabling interrupts**.
- A **single** program is running – **startup** – when the CPU jumps to `nulluser`. This is the program that **initializes important datastructures, devices, semaphores, and processes**. The code is found in file `initialize.c` below, where the transformation from program to system takes place.
- **If there is drama in the system, it lies here!**

Initializing system data structures

```
/* initialize.c - nulluser, sysinit */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <sleep.h>
#include <mem.h>
#include <tty.h>
#include <q.h>
#include <io.h>
#include <disk.h>
#include <network.h>

extern  int     main();                 /* address of user's main prog  */

/* Declarations of major kernel variables */

struct  pentry  proctab[NPROC]; /* process table              pg    55 */
int     nextproc;               /* next process slot to use   pg 74,76 */
struct  sentry  semaph[NSEM];   /* semaphore table            pg    84 */
int     nextsem;                /* next semaphore slot to use pg    88 */
struct  qent    q[NQENT];       /* q table (see queue.c)      pg    44 */
int     nextqueue;              /* next slot in q structure   pg    44 */
int     *maxaddr;               /* max memory address (set by sizmem)  */
#ifdef  NDEVS                   /*                           pg     145 */
struct  intmap  intmap[NDEVS];  /* interrupt dispatch table  pg 116,157 */
#endif
struct  mblock  memlist;        /* list of free mem blks.    pg     105 */
#ifdef  Ntty                    /*                           pg     163 */
struct  tty     tty[Ntty];      /* SLU buffers + mode cntrl  pg     163 */
#endif

/* active system status */

int     numproc;                /* number of live user processes       */
int     currpid;                /* id of currently running process     */
int     reboot = 0;             /* non-zero after first boot           */
```

Initializing system data structures

```
//* real-time clock variables and sleeping process queue pointers  */

#ifdef  RTCLOCK   /*                                               */
int     count6;   /* initial.     counts in 60ths of a second 6-0     */
int     defclk;   /* on page      non-zero, then deferring clock count */
int     clkdiff;  /* 137 in       deferred clock ticks                */
int     slnempty; /* clkinit      FALSE if the sleep queue is empty   */
int     *sltop;   /*              address of key part of top entry in */
                                /* the sleep queue if slnempty==TRUE   */
int     clockq;   /* inital.by    head of queue of sleeping processes */
int     preempt;  /* clkinit      preemption counter.  Current process */
                                /* is preempted when it reaches zero;  */
                                /* set in resched; counts in ticks     */
int     clkruns;                /* set TRUE iff clock exists by setclkr */
#else
int     clkruns = FALSE;        /* no clock configured; be sure sleep  */
#endif                          /*   doesn't wait forever              */

int     rdyhead,rdytail;        /* head/tail of ready list (q indexes) */


/*********************************************************************/
/***                          NOTE:                               ***/
/***                                                              ***/
/***   This is where the system begins after the C environment has ***/
/***   been established.  Interrupts are initially DISABLED, and   ***/
/***   must eventually be enabled explicitly.  This routine turns  ***/
/***   itself into the null process after initialization.  Because ***/
/***   the null process must always remain ready to run, it cannot ***/
/***   execute code that might cause it to be suspended, wait for a ***/
/***   semaphore, or put to sleep, or exit.  In particular, it must ***/
/***   not do I/O unless it uses kprintf for polled output.        ***/
/***                                                              ***/
/*********************************************************************/
```

Initializing system data structures

```
/*------------------------------------------------------------------------
 *  nulluser  -- initialize system and become the null process (id==0)
 *------------------------------------------------------------------------
 */
nulluser()                              /* babysit CPU when no one home */
{
        int     userpid;
        char    ps;

        kprintf("\n\nXinu Version %s", VERSION);
        if (reboot++ < 1)
                kprintf("\n");
        else
                kprintf("   (reboot %d)\n", reboot);
        sysinit();                      /* initialize all of Xinu */
        kprintf("%u real mem\n",(unsigned)maxaddr+(unsigned)sizeof(int));
        kprintf("%u avail mem\n",
                (unsigned)maxaddr-(unsigned)(&end)+(unsigned)sizeof(int));
        kprintf("clock %sabled\n\n", clkruns==1?"en":"dis");
        enable();                       /* enable interrupts */

        /* start a process executing the user's main program */

        resume(
                userpid = create(main,INITSTK,INITPRIO,INITNAME,1,0);
        );
        while (TRUE) {                  /* run forever without actually */
                pause();                /*  executing instructions      */
        }
}
```

Initializing system data structures

```
/*------------------------------------------------------------------------
 *  sysinit  --  initialize all Xinu data structures and devices
 *------------------------------------------------------------------------
 */
LOCAL   sysinit()
{
        int     i, j;
        struct  pentry  *pptr;
        struct  sentry  *sptr;
        struct  mblock  *mptr;

        numproc  = 0;                   /* initialize system variables  */
        nextproc = NPROC-1;
        nextsem  = NSEM-1;
        nextqueue= NPROC;               /* q[0..NPROC-1] are processes  */

        memlist.mnext = mptr =          /* initialize free memory list  */
          (struct mblock *) roundew(&end);
        mptr->mnext = (struct mblock *)NULL;/* NULLSTK -- see page 72   */
        mptr->mlen = truncew((unsigned)maxaddr-NULLSTK-(unsigned)&end);

        for (i=0 ; i<NPROC ; i++)       /* initialize process table     */
                proctab[i].pstate = PRFREE;

        pptr = &proctab[NULLPROC];      /* init. null process entry     */
        pptr->pstate = PRCURR;          /* see pages 54/55              */
        for (j=0; j<7; j++)
                pptr->pname[j] = "prnull"[j];
        pptr->plimit = ( (int)maxaddr ) - NULLSTK;
        pptr->pbase = maxaddr;          /* initialized in sizmem pg 193 */
        pptr->paddr = nulluser;
        pptr->pargs = 0;
        currpid = NULLPROC;
```

Initializing system data structures

```
        for (i=0 ; i<NSEM ; i++) {      /* initialize semaphores */
                (sptr = &semaph[i])->sstate = SFREE;  /* page 84 */
                sptr->sqtail = 1 + (sptr->sqhead = newqueue());
        }

        rdytail = 1 + (rdyhead=newqueue());/* initialize ready list */

#ifdef  MEMMARK
        _mkinit();                     /* initialize memory marking, §15 */
#endif
#ifdef  RTCLOCK
        clkinit();                      /* initialize r.t.clock, pg 137 */
#endif
#ifdef  Ndsk
        dskdbp= mkpool(DBUFSIZ,NDBUFF); /* initialize disk buffers */
        dskrbp= mkpool(DREQSIZ,NDREQ);  /* page 240, §15 */
#endif
        for ( i=0 ; i<NDEVS ; i++ )     /* initialize devices */
                init(i);
#ifdef  NNETS                           /* initialize networks */
        netinit();
#endif
        return(OK);
}
```

13.6 nulluser

- `nulluser` is simple: It calls `sysinit` to do the **initialization**. When `sysinit` returns, **the running program has been made into process 0, with interrupts disabled, and no other process existing**.
- After printing a few messages, `nulluser` **enables interrupts** and calls `create` to start a process running the **user's main program**.
- Because the process executing `nulluser` has become the null-process, it **cannot** exit, sleep, wait for a semaphore, or suspend itself.
- After initialization is completed, and a process has been created to execute the user's main program, the `nulluser` process just becomes an **infinite loop**, **capable of reacting to interrupts and giving `resched` a process to schedule to when no user process are ready to run**.
- `nulluser` requires a **special print procedure** `kprintf`, using **polled I/O**, because normal I/O processing would put it in the process state `PRWAIT`, and this is forbidden for the nulluser, because its process state should be *current* or *ready* (see above). `kprintf` is dicussed in §19.4.2.

13.7 Transforming the program into a process

Where is the **program** turned into a **process**?

- Procedure `sysinit` performs the **system initialization**:
  - It initializes the system's data structures, like the semaphore table, q-structure, process table, free memory list etc.
  - It initializes clock routines by calling `clockinit`.
  - Finally, it calls `init` once for each system device.
  - Procedure `init` calls the device initialization routines indirectly through `devtab`.
- Most interesting: it fills in the **process-table fields for process zero**. The real work is done by **two lines**: assigning the process-table fields `PRCURR`, and `currpid = NULLPROC`!
- Only **after these values have been assigned**, **rescheduling becomes possible**: then the program becomes a currently running process that `resched` can identify as process 0.

13.8 The map of low core

- The interrupt vectors are all initialized to a **default value** before interrupt vectors for specific devices are filled in.
- Unexpected interrupts trap to procedure `panic`, which **prints a message** on the console and **halts** the 11/2, through traps using hte **exception vectors**.
- **None** of the initialization code we have seen until now fills in exception or interrupt vector locations other than the clock vector and the I/O vectors specified in the device switch table.
- **How are these locations initialized**? The remaining vector addresses are initialized statically, before execution begins, to an appropriate entry point in `panic`. File `lowcore.s` contains the code.

The map of low core

```
/* lowcore.s - (map of interupt vectors in low part of memory) */

DISABLE     =     340              / PS to     disable     interrupts
ENABLE      =     000              / PS to     enable interrupts

/*-----------------------------------------------------------------------------
/*  absolute location 0  -- fixed interrupt and exception vectors
/*-----------------------------------------------------------------------------
.     =     000^.          / panic because something jumped  / not used for
      jmp   panic0         /          to location zero       / an exception
                                                             / vector address
.     =     004^.          / bus error (e.g., malfunction)   /----------------
      panic;      DISABLE+0                                  / here except.
.     =     010^.            / illegal/reserved instruction  / vector starts
      panic;      DISABLE+1
.     =     014^.            / BPT instruction and T bit
      panic;      DISABLE+2
.     =     020^.            / Input/Output trap
      panic;      DISABLE+3
.     =     024^.            / Power fail
      panic;      DISABLE+4
.     =     030^.            / EMT instruction, emulator trap
      panic;      DISABLE+5
.     =     034^.            / TRAP instruction
      panic;      DISABLE+6
.     =     040^.             /------------------------------------------
      panic;      DISABLE+7   | To distinguish between the various types
.     =     044^.             | of exceptions and traps caused by un-
      panic;      DISABLE+7 <--+ expected interrups, lowcore.s places a
                              | code in the low-order 4 bits of the
 repeat for all               | stored PS, just as the device interrupt
 locations through 774        | dispatcher does
                              \------------------------------------------
.     =     770^.
      panic;      DISABLE+7
.     =     774^.
      panic;      DISABLE+7
```

E.g., the code for `panic0` just prints the message “Panic: branch to location 0” on the console, and halts the processor. Procedure `panic` does more: it also prints the contents of all registers and the top few locations of the stack, §19.

17 A Disk Driver

Introduction

- At the level of device driver routines, a disk is viewed as a **randomly accessible array of blocks** (here, of size 512 bytes).
- The interaction between driver software and the disk device itself is more complicated than for a SLU – therefore, an **extra controller** (the disk-controller) is needed with its own format of controller request records.
- A disk is a random access device, therefore, a disk must **position the disk arm** as well as **transfer data**.
- Disk hardware uses **direct-memory-access-mode (DMA)**, allowing for the **direct transfer** to/from memory blocks **without help from the CPU**. So it does not interrupt the CPU for the transfer of every single character. Hence the CPU can continue operation while a block transfer takes place.

17.1 Operations supplied by Disk Drivers

- At the device driver level, a disk is nothing more than a large array of data blocks that can be accessed **randomly** using three basic operations:
  - select a block: `seek`
  - copy the contents of the selected block from disk to memory: `read`
  - copy the contents of memory to the selected block on disk: `write`
- Format of an instruction: `read(DISKDEV, buff, blocknr)`

  This means read diskblock with number `blocknr` into memory starting at address `buff` from the disk identified by `DISKDEV`.
- As long as calls to read and write include a block position, a **separate seek operation** is unnecessary. So why is operation `seek` needed? **To optimize disk access**!
- Why is optimization necessary? Because the relation between the times taken by an execution of a memory instruction and moving a disk arm is

1 : **300.000**!

Operations supplied by Disk Drivers

- As long as calls to `read` and `write` include a block position, a separate `seek` operation is **unnecessary**.
- In many systems, however, disk controller hardware does not permit **simultaneous transfer** from two platters even through it does allow **simultaneous motion** of all disk arms.
- Operating systems using such hardware can significantly reduce disk access times by moving one disk arm to the desired position while reading a block from another disk. Arm movement is the **most costly part of disk access**, so overlapping `seeks` with computations (and transfer operations on other disks) is an important optimization.
- To ensure that higher layers of software can perform such optimization, a `seek` operation will be included in the design.

Recapitulation of §2.2: Disk Storage Organization

- Disk devices are **slow and awkward**, compared with main memory operations at speeds measured in **nanoseconds**!
- A disk access is on the order of **300,000 times slower** than a memory access. To put this in perspective, imagine that you need to retrieve a pencil before you continue some task. Suppose that it takes 10 seconds to walk to your desk (high-speed memory) and retrieve one. A retrieval operation that took 300,000 times as long would last roughly **34.7 days**. After a few such retrieval operations, you would quickly learn to save pencils in your desk drawer to avoid the trip. Operating systems use an analogous technique – they often contain **complex algorithms** that save **copies of blocks** to avoid unnecessary disk access.

Recapitulation of §2.2: Disk Storage Organization

- So **why use disks** at all? Because main memory is **volatile**, i.e., data “evaporates” when power is off, whereas disks ar **nonvolatile**, because **their data persists** when the system stops and power is off.
- Most significant **difference** with respect to **SLU**: Unlike SLU hardware which interrupts CPU for **each single character** received or transmitted, the disk interface transfers **large blocks** of data directly to or from memory, interrupting **only** when transfer has completed.
- This technique is called **Direct Memory Access (DMA)** and **dramatically increases throughput** while the CPU continues to execute instructions.

Recapitulation of §2.2.1: Disk Interface and Controller Hardware

A disk consists of:

- a **disk drive** – i.e., platters (spindle) which rotate at high speed, a physiscal arm and read/write head(s)
- an **electronic controller** (Xebec) – it contains **microprocessor(s)**, controls positioning of the disk arm and transfer of data, it also senses **error** occuring, it can operate on **multiple** disk drives, but it's not able to transfer data to/from more than one diskplatter **simultaneously**,
- a **host interface** (DTC) – it **connects** controller to system bus, i.e., it passes requests for starting an I/O operation **from processor to controller**, **interrupts the CPU** when controller signals that an operation has completed, and **passes data** between controller and main memory during data transfer.

Recapitulation of §2.2.2: The pieces of a Disk Hardware

- The disk hardware has no notion of **file** or built-in **directory**. The hardware considers each disk device as an **array of 512-byte blocks** (called **sectors**), addressed $0, 1, \dots, n$ (with $n = 1000 \dots 30.000$ anno 1984!).
- Hardware reads data by copying a sector on the disk to a block in memory, and writes data by copying a block from memory onto a sector of the disk.
- Using the disk hardware consists of **passing requests to the host interface**, which carries them out and reports the results. Part of this information constitues a **request for the controller**. The other part consists of a **block to which data is to be transferred**.

Recapitulation of §2.2.3: Interface and Controller Request Formats

- The disk host interface starts at address 0177460_8 (device-`csr`-address) **beyond real memory** and has a 6 word format, including 4 essentially used words (on the 11/2).
- Xebec controller **request record** arrangement:

Byte 0	command
Byte 1	disk unit (0)
Byte 2	high byte of block-number
Byte 3	low byte of block-number
Byte 4	block count (1)
Byte 5	control (0)

- The disk driver discussed in the chapter was built for an *International Technologies, Inc., 5018H Winchester Disk Drive*, a *Data Technology Corporation DTC-11-1 host interface* and a *Xebec Systems Inc. S1410 Winchester Disk Controller*. Despite being manufactured by three vendors, these pieces of equipment work well together.

Recapitulation of §2.2.3: Interface and Controller Request Formats

`CCSR`	Completion Status Register		
`CSR`	Control & Status Register	←	see below
`DAR`	DMA Address Register	←	Buffer address for read/write
`CAR`	Control Address Request Register	←	Adress in memory of request record for the disk controller
`XDAR`	extended `DAR`, not used		
`XCAR`	extended `CAR`, not used		

The interface `CSR` register contains several bits that **report errors** and **control interrupts**:

15	14	13	12 ... 8	7	6	5 ... 2	1	0
E		P		D	I		F	G
error bit		parity error		operation completed	interrupts enabled		reset controler	go-bit

- If bit 6 (I) is set, interface will interrupt whenever an operation is completed.
- If go-bit 0 (G) is set, then I/O operation is started.

Summary of §17

1. **Low-level** disk drivers implement 4 operations (`read`, `write`, `seek` and `control`). **Upper-half** routines merely **enqueue requests for services**.

   Whenever an data-transfer-operation completes, a lower-half routines takes the **next pending request from queue**, starts up hardware performing that operation, and wakes up the process waiting for **previous operation to end**.

2. Driver reduces time to honor requests by recording them to **minimize arm movement**. When enqueuing a request, upper-half routines insert it between two adjacent existing requests if the arm will **pass over the requested block** while **traveling** from one block to the other. While this heuristic does not **guarantee** minimum access time for a set of requests, it **works well in practice**.

3. Because **copying data** for each transfer takes too much time, the driver accepts output requests, and returns **asynchronously** to caller – **without having coped the data** in the output buffer into the system buffers – **a dangerous strategy**, unusable for a general-purpose disk interface. However the routines developed in this chapter are used **safely**, without causing errors, inside the system procedures for **handling files** in the next chapter.

17.2 Controller Requests and Interface Register Descriptions

The Xebec Disc Controller layout is described by structure `xbdcb` on pages 284/285, corresponding to the figure of the request record layout just seen.

```
/* xebec.h */
/* Xebec S1410 5.25 inch Winchester Disk Controller */

struct      xbdcb {                    /* Xebec disk controller regs.     */
      char  xop;                       /* command class/opcode            */
      char  xunit;                     /* unit address bits/no of disk unit*/
      char  xmaddr;                    /* middle-order address bits       */
      char  xladdr;                    /* low-order address bits          */
      char  xcount;                    /* block count: no of block on disk */
      char  xcntl;                     /* control field                   */
      char  xres1;                     /* reserved field #1               */
      char  xres2;                     /* reserved field #2               */
};

/* Xebec controller operation codes */
#define     XOTDR       0              /* Test Drive Ready                */
#define     XORCAL      1              /* Recalibrate                     */
#define     XONONO      2              /* Reserved (is a no-no)           */
#define     XORSS       3              /* Request Sense Status            */
#define     XOFMTD      4              /* Format Drive                    */
#define     XOCTF       5              /* Check Track Format              */
#define     XOFMTT      6              /* Format Track                    */
#define     XOFMTBT     7              /* Format Bad Track                */
#define     XOREAD      8              /* Read                            */
#define     XONO2       9              /* Reserved (is also a no-no)      */
#define     XOWRITE     10             /* Write                           */
#define     XOSEEK      11             /* Seek                            */
#define     XOINIT      12             /* Initialize Drive Character.*/
#define     XOREBEL     13             /* Read ECC Burst Error Length*/
#define     XORAMD      14             /* Perform RAM Diagnostic Test*/
#define     XRETRY      0              /* retry control byte              */
```

Controller Requests and Interface Register Descriptions

- Notice the many operations listed in `xebec.h`! For us, only `XOREAD`, `XOWRITE` and `XOSEEK` are interesting.
- The DTC host interface parallels the figure of the **host interface device register layout**:

```
/* dtc.h */

/* Data Technology Corp. DTC-11-1 disk controller host interface */

struct  dtc     {                           /* controller interface regs.   */
        int     dt_ccsr;                    /* command completion status    */
        int     dt_csr;                     /* control and status register  */
        char    *dt_dar;                    /* data address register        */
        struct  xbdcb   *dt_car;            /* command address register     */
        int     dt_xdar;                    /* extension of dar (not used)  */
        int     dt_xcar;                    /* extension of car (not used)  */
};

/* bits in the dtc csr register */
#define DTGO    0000001                     /* "go" bit - start interface   */
#define DTRESET 0000002                     /* "force reset" bit            */
#define DTINTR  0000100                     /* enable interface interrupt   */
#define DTDONE  0000200                     /* command done                 */
#define DTERROR 0100000                     /* some error occurred          */
```

- Notice that the bits in figure 2.13 correspond to operations:

`DTGO` – set bit nr. 1		`DTRESET` – set bit nr. 2	
`DTINTR` – set bit nr. 6		`DTDONE` – set bit nr. 7	
`DTERROR` – set bit nr. 15			

17.3 The List of Pending Disk Requests

- Disk driver routines are partitioned into two sets: Upper- and Lower-half routines, communication through a **shared data structure**, the **disk control block**, which includes a **list of pending requests**.
- Upper-half routines **enqueue requests** for operations, **start** the device when it is idle, and then **wait** for the operation to complete.
- Lower-half routines are interrupt handlers, which are called when an **operation completes**, then they **start the next operation** and awaken the process waiting for the request just completed.

The List of Pending Disk Requests

```
/* disk.h - dssync, dsdirec */

#include <xebec.h>                          /* disk controller constants    */
#include <dtc.h>                            /* disk interface constants     */

typedef unsigned        DBADDR;             /* disk data block addresses    */
#define DBNULL          (DBADDR)0177777 /* null disk block address      */

struct  dsblk   {                           /* disk driver control block    */
        struct  dtc     *dcsr;              /* disk interface csr address   */
        struct  dreq    *dreqlst;           /* list of pending requests     */
        int     dnum;                       /* device number of this disk   */
        int     dibsem;                     /* i-block mut.exclus.sem.  (*) */
        int     dflsem;                     /* free list "   "     "    (*) */
        int     ddirsem;                    /* directory "   "     "    (*) */
        int     dnfiles;                    /* num. of currently open files */
        char    *ddir;                      /* address of in-core directory */
        struct  xbdcb   ddcb;               /* holds command that interface */
                                            /*   sends to disk controller   */
};
extern  struct  dsblk   dstab[];

struct  dreq    {                           /* node in list of requests     */
        DBADDR  drdba;                      /* disk block address to use    */
        int     drpid;                      /* process id making request    */
        char    *drbuff;                    /* buffer address for read/write*/
        char    drop;                       /* operation: READ/WRITE/SEEK   */
        int     drstat;                     /* returned status OK/SYSERR    */
        struct  dreq    *drnext;            /* ptr to next node on req. list*/
};
```

(∗) Multiple processes access the file system concurrently. Also a single process can access an arbitrary number of files concurrently. However, access of each file is restricted to at most one process. These conditions require for their implementation the intro of **three semaphores**.

The List of Pending Disk Requests

`disk.h` continued:

```
#define DRNULL  (struct dreq *) 0       /* null pointer in request list */
#define DIRBLK  0                       /* block used to hold directory */
#define DONQ    2                       /* status if request enqueued   */
#define DBUFSIZ 512                     /* size of disk data block      */
#define DREQSIZ sizeof(struct dreq)     /* size of disk request node    */
#ifndef NDBUFF
#define NDBUFF  10                      /* number of disk data buffers  */
#endif
#define NDREQ   10                      /* number of disk request buf.  */
#define DREAD   XOREAD                  /* read command in dreq.drop    */
#define DWRITE  XOWRITE                 /* write   "                    */
#define DSEEK   XOSEEK                  /* seek    "                    */
#define DSYNC   XOTDR                   /* sync    "  (test-disk-ready) */

extern  int     dskrbp;                 /* disk request node buffer pool*/
extern  int     dskdbp;                 /* disk data block buffer pool  */

/* disk control function codes */

#define DSKSYNC 0                       /* synchronize (flush all I/O   */
                                        /*          at system shutdown) */
#define dssync(ddev)    control((ddev),DSKSYNC);

#define dsdirec(ddev)   ((struct dir *)devtab[ddev].dvioblk->ddir)
```

The List of Pending Disk Requests

- For disks, the required information for transfer between upper- and lower-level routines resides in **control blocks**.
- The disk driver control block consists of an array `dstab[]`, consisting of structures `dsblk`, containing, among others:
  - an address of `CSR` (0177460_8 – taken from device switch table),
  - a list of pending requests `*dreqlist`,
  - a device number of the disk, `dnum`,
  - three `semaphores` for implementing mutual exclusion because of the possibility for concurrent access by processes to disk, caused by file operations,
  - `dnfiles` – the numer of currently open files, and
  - a 6 byte command, that the interface sends to the controller of the disk, structures `xbdcb` (for Xebec disk controller block) `ddcb`.

The List of Pending Disk Requests

- The format of each entry in the list of requests is of structure `dreq`, consisting of fields for:
  - diskblock address to use, `drdba`,
  - identifier of requesting process (because of need for notification), `drpid`,
  - buffer address for read/write, `*drbuff`,
  - operation to-be-performed, `drop`, and
  - returned status, `drstat` (`OK`/`SYSERR`/`DONQ`).
- There are two constants: `DIRBLK` (block used to hold directory) and `DONQ` (status when request is enqueued).
- The disk request-node buffer pool is accessed by `dskrbp`, the disk data block buffer pool by `dskdbp`.

The List of Pending Disk Requests

- The disk driver maintains the following invariant:

  The first request on the pending request list is always the one the hardware is performing; if the list is empty, the hardware is idle.

- The list of pending request, i.e., `dreqlst`, is a **singly linked list** consisting of structures `dreq`.

- Field `drstat` **communicates** status information from lower-half to upper-half: The lower-half records information about **errors** occuring during the operation; the upper-half extracts that information and **passes it on to the caller**.

17.4 Enqueuing disk requests

- The disk driver processes I/O requests in the order in which they occur on the **request list**. When one completes, it is **removed from the list** and the **next one** is started.
- When procedure `dskenq` adds a request for block B to the existing list of requests, it schedules it to be performed between requests i and $i + 1$, **if the disk arm will pass over block B on its way from i to $i + 1$!**
- If no such pair $(i, i + 1)$ exists, `dskenq` adds this new request to the end of list.

Enqueuing disk requests

```
/* dskenq.c - dskenq */

#include <conf.h> <kernel.h> <disk.h>

/*------------------------------------------------------------------------
 *  dskenq  --  enqueue a disk request and start I/O if disk not busy
 *------------------------------------------------------------------------
 */
dskenq(drptr, dsptr)
        struct  dreq    *drptr;         /* add. of to-be-inserted req.  */
        struct  dsblk   *dsptr;         /* add. diskdriver contr. block */
{
        struct  dreq    *p, *q;         /* q follows p through requests */
        DBADDR  block;
        int     st;

        if ( (q=dsptr->dreqlst) == DRNULL ) {    /* an empty list means */
                dsptr->dreqlst = drptr;          /* the device is idle  */
                drptr->drnext = DRNULL;
                dskstrt(dsptr);                  /* start up the disk   */
                return(DONQ);               /* status if request enqueued */
        }
        block = drptr->drdba;
        for (p = q->drnext ; p != DRNULL ; q=p,p=p->drnext) {
           if (p->drdba==block && (st=dskqopt(p, q, drptr)!=SYSERR))
                        return(st);
           if ( (q->drdba <= block && block < p->drdba) ||   /* [..b..) */
                (q->drdba >= block && block > p->drdba)) {   /* (..b..] */
                        drptr->drnext = p;     /* insert between q and p */
                        q->drnext = drptr;     /* observe that multiple  */
                        return(DONQ);          /* requests for the same  */
                }                              /* block are stored in    */
        }                                      /* FIFO order             */
        drptr->drnext = DRNULL;
        q->drnext = drptr;
        return(DONQ);
}
```

17.5 Optimizing the Request Queue

- In case of a **read operation** to be enqueued for which there is a **write for the same block** in the list of pending requests, the read can be **instantaneously performed**, by copying the write request buffer directly into the read request buffer, and allow the reading process to continue.
- Optimizations are also possible for double or more **writes to same block**: the first one can be discarded! This is done by procedure `dskqopt`.

Optimizing the Request Queue

```
/* dskqopt.c - dskqopt */

#include <conf.h>
#include <kernel.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dskqopt  --  optimize requests to read/write/seek to the same block
 *------------------------------------------------------------------------
 */
dskqopt(p, q, drptr)
struct  dreq    *p, *q, *drptr;
{
        char    *to, *from;
        int     i;
        DBADDR  block;

        /* By definition, sync requests cannot be optimized.  Also,     */
        /* cannot optimize read requests if already reading.            */

        if (drptr->drop==DSYNC || (drptr->drop==DREAD && p->drop==DREAD))
                return(SYSERR);

        if (drptr->drop == DSEEK) {     /* ignore extraneous seeks      */
                freebuf(drptr);
                return(OK);
        }

        if (p->drop == DSEEK) {         /* replace existing seeks       */
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p);
                return(OK);
        }
```

Optimizing the Request Queue

`dskqopt.c` continued:

```
        if (p->drop==DWRITE && drptr->drop==DWRITE) {   /* dup write    */
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p->drbuff);
                freebuf(p);
                return(OK);
        }

        if (drptr->drop==DREAD && p->drop==DWRITE) {    /* satisfy read */
                to = drptr->drbuff;
                from = p->drbuff;
                for (i=0 ; i<DBUFSIZ ; i++)
                        *to++ = *from++;
                return(OK);
        }

        if (drptr->drop==DWRITE && p->drop==DREAD) {    /* sat. old read*/
                block = drptr->drdba;
                from = drptr->drbuff;
                for (; p!=DRNULL && p->drdba==block ; p=p->drnext) {
                        q->drnext = p->drnext;
                        to = p->drbuff;
                        for (i=0 ; i<DBUFSIZ ; i++)
                                *to++ = *from++;
                        p->drstat = OK;
                        ready(p->drpid, RESCHNO);
                }
                drptr->drnext = p;
                q->drnext = drptr;
                resched();
                return(OK);
        }
        return(SYSERR);
}
```

17.6 Starting a Disk Operation

- Procedure `dskstrt` **starts** a disk operation by building a **controller request structure** in the `ddcb`-field of the disk control block, feeding that request to the controller through the interface.
- Recall that the first request on that line is the one **hardware should perform**. Once finished – as indicated by an interrupt – the **second one** is started, and the process waiting for termination of that first request is notified.

Starting a Disk Operation

```
/* dskstrt.c - dskstrt */

#include <conf.h>
#include <kernel.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dskstrt  --  start an I/O operation on a disk device
 *------------------------------------------------------------------------
 */
dskstrt(dsptr)
        struct  dsblk   *dsptr;    /* diskdriver control block address  */
{
        struct  xbdcb   *xptr;     /* address of controller interface   */
        struct  dtc     *dtptr;    /* ptr to disk contr. host interface */
        struct  dreq    *drptr;    /* pointer to the request list       */

        /* build command for controller */

        drptr = dsptr->dreqlst;    /* list of pending req., 1st record  */
        xptr = & dsptr->ddcb;      /* address of contrl request block   */
        xptr->xop    = (char) drptr->drop;             /* opcode       */
        xptr->xunit  = (char) 0;                       /* top addr bits*/
        xptr->xmaddr = (char) ((drptr->drdba>>8)&0377); /* mid addr bits*/
        xptr->xladdr = (char) (drptr->drdba & 0377);    /* low addr bits*/
        xptr->xcount = (char) 1;                       /* num of blocks*/
        xptr->xcntl  = (char) XRETRY;                  /* retry code   */

        /* feed command to controller through interface */

        dtptr = dsptr->dcsr;  /* hw addr. of disk device host interface */
        dtptr->dt_dar = drptr->drbuff; /* buffer addr. for an operation */
        dtptr->dt_car = xptr;    /* addr. of ctrl. req. block filled in */
        dtptr->dt_xdar = dtptr->dt_xcar = 0; /* ext. addr. regs cleared */
        dtptr->dt_csr = DTINTR | DTGO; /* enable interface interrupt +  */
}                                      /* start interface by setting    */
                                       /* "go"-bit to 1                 */
```

17.7 Driver Initialization

- Procedure `dsinit`
  - fills in the **disk control block**,
  - fills in the corresponding **interrupt vectors**, and the corresponding entries into the **interrupt dispatch table** by calling

    `iosetvec(devptr->dvnum, dsptr, dsptr)`,

  - sets up a read command for the **disk directory** to be read, and
  - then **starts** the interface up.
- Because the system executes the initialization routine **before interrupts are enabled**, `dsinit` must **poll** to determine when the read operation completes!

Driver Initialization

```
/* dsinit.c - dsinit */

#include <conf.h>
#include <kernel.h>
#include <disk.h>

#ifdef  Ndsk
struct  dsblk   dstab[Ndsk];
#endif
int     dskdbp, dskrbp;

/*------------------------------------------------------------------------
 *  dsinit  --  initialize disk drive device
 *------------------------------------------------------------------------
 */
dsinit(devptr)
        struct  devsw   *devptr;    /* ptr to device switch table entry */
{
        struct  dsblk   *dsptr;     /* ptr to disk driver control block */
        struct  dtc     *dtptr;     /* ptr to disk contr. host interface*/
        int     status;
        char    ps;

        disable(ps);
        devptr->dvioblk = dsptr = &dstab[ devptr->dvminor ];
        dsptr->dcsr    = devptr->dvcsr;  /* address of the host interf. */
        dsptr->dreqlst = DRNULL;         /* pending requests: NONE!     */
        dsptr->dnum    = devptr->dvnum;  /* corresponding entry in      */
        dsptr->dibsem  = screate(1);     /*   interrupt dispatch table  */
        dsptr->dflsem  = screate(1);
        dsptr->ddirsem = screate(1);
        dsptr->dnfiles = 0;
        dsptr->ddir    = getbuf(dskdbp); /* addr. of in-core-mem for dir*/
        iosetvec(devptr->dvnum, dsptr, dsptr);         /* see page 157 */
```

Driver Initialization

- `dvioblk` should contain address or control block
- `dcsr` should contain address of host interface
- `dreqlst` is `DRNULL`!
- `dnum` should contain device number of *this* disk
- `dibsem` contains semaphore dealing with access to the **index block list** in the file system
- `dflsem` contains semaphore dealing with access to the **free list of data blocks** in the file system
- `ddirsem` contains semaphore dealing with **directory access** in the file system
- `dnfiles` is the number of open files, initially 0
- `ddir` is the address for **incore memory** for the data directory: `getbuf(dskdbp)` with `dskdbp` being the disk data block buffer pool
- `iosetvec(..., ..., ...)` initializes the interrupt dispatch table and interrupt vectors

Driver Initialization

`dsinit.c` continued:

- Next the Xebec controller interface structure for a **read operation** is initialized. The host interface is initialized and the interface is started setting the **go**-bit.
- The procedure goes in a while loop, checking whether the operation is not yet completed by polling the `DTDONE`-bit, until it is 1, since no interrupt mechanism is available.

```
    /* read directory block: setup read command then start interface */
    dsptr->ddcb.xop    = (char) XOREAD;
    dsptr->ddcb.xunit  = (char) 0;         /* fill in fields of disk */
    dsptr->ddcb.xcntl  = (char) XRETRY;    /*   driver control block */
    dsptr->ddcb.xmaddr = (char) ((DIRBLK>>8)&(0377));
    dsptr->ddcb.xladdr = (char) (DIRBLK&0377);
    dsptr->ddcb.xcount = (char) 1;
    dtptr = dsptr->dcsr;   /* ptr to host interface in control block */
    dtptr->dt_dar   = dsptr->ddir;    /* address of disk directory   */
    dtptr->dt_car   = &dsptr->ddcb;   /* contr. cmnd register passed */
    dtptr->dt_xdar  = dtptr->dt_xcar = NULL;
    dtptr->dt_csr   = DTGO;     /* host interf. starts I/O operation */
    while ( ( (status=dtptr->dt_csr) & DTDONE) == 0)
            ;                      /* poll until operation completed */
    if (status & DTERROR)
            panic("Disk error");
    restore(ps);
    return(OK);
}
```

17.8 Upper-Half Read Routine

The upper-half input routine:

- builds a **request node**,
- inserts that node in **pending request list**,
- waits until lower-half routine signals **completion** of that request,
- then **starts up** the suspended process making that request, and
- deallocates the used request block to free memory.

Upper-Half Read Routine

```
/* dsread.c - dsread */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dsread  --  read a block from a disk device
 *------------------------------------------------------------------------
 */
dsread(devptr, buff, block)
        struct  devsw   *devptr;
        char    *buff;
        DBADDR  block;
{
        struct  dreq    *drptr;
        int     stat;
        char    ps;

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        drptr->drdba = block;           /* disk block address           */
        drptr->drpid = currpid;         /* process making request       */
        drptr->drbuff = buff;           /* memory buffer into which read */
        drptr->drop = DREAD;
        if ( (stat=dskenq(drptr, devptr->dvioblk)) == DONQ) {
                suspend(currpid);       /* wait for lower half          */
                stat = drptr->drstat;   /* OK/SYSERR                    */
        }
        freebuf(drptr);                 /* deallocate this request block */
        restore(ps);
        return(stat);
}
```

17.9 Upper-Half Output Routine

- The upper-half output routine is **asynchronous**!
- I.e., the operation **does not wait until output is completed** because of considerations of **speed**.

! **Danger: One shouldn't overwrite the to be outputted buffer until the operation is completed!**

- Indeed, this is done in XINU: **Only** XINU **file system routines** call disk driver, taking care that output is buffered **before** calling the driver.

17.10 Implementation of the Upper-Half Output Routine

```
/* dswrite.c - dswrite */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dswrite  --  write a block (system buffer) onto a disk device
 *------------------------------------------------------------------------
 */
dswrite(devptr, buff, block)
        struct  devsw   *devptr;
        char    *buff;
        DBADDR  block;
{
        struct  dreq    *drptr;
        char    ps;

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        drptr->drbuff = buff;
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drop = DWRITE;
        dskenq(drptr, devptr->dvioblk);
        restore(ps);
        return(OK);
}
```

17.11 The Upper-Half Seek Routine

```
/* dsseek.c - dsseek */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dsseek  --  schedule a request to move the disk arm
 *------------------------------------------------------------------------
 */
dsseek(devptr, block)
        struct  devsw   *devptr;
        DBADDR  block;
{
        struct  dreq    *drptr;
        char    ps;

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drbuff = NULL;
        drptr->drop = DSEEK;

        /* enqueued with normal policy like other read/write requests */

        dskenq(drptr, devptr->dvioblk);
        restore(ps);
        return(OK);
}
```

17.12 Lower-Half Disk Driver Routine

- The procedure `dsinter` is called whenever device interface interrupts to signal **completion of an operation**.
- Then `dsinter` **removes the first request** from the head of pending request list (the operation that just finished) and **starts up** the next operation if one exists.
- `dsinter` also disposes of the request node just removed.
- If just-satisfied request was input request, `dsinter` awakens/unsuspends the reading process (allowing it to return to its caller). Otherwise, it deallocates request node itself because the operation was `WRITE` (in this case it also deallocates data buffer) or `SEEK`.

Lower-Half Disk Driver Routine

```
/* dsinter.c - dsinter */
#include <conf.h> <kernel.h> <disk.h>

/*------------------------------------------------------------------------
 *  dsinter  --  process disk interrupt (DTC interface; XEBEC controller)
 *------------------------------------------------------------------------
 */
INTPROC dsinter(dsptr)
        struct  dsblk   *dsptr;   /* disk driver control block address  */
{
        struct  dtc     *dtptr;   /* ptr to controller host interface   */
        struct  dreq    *drptr;   /* ptr to request node                */

        dtptr = dsptr->dcsr;      /* address of host interface          */
        drptr = dsptr->dreqlst;   /* record address of 1st request node */
        if (drptr == DRNULL) {
                panic("Disk interrupt when disk not busy");
                return;
        }
        if (dtptr->dt_csr & DTERROR)    /* depending on result of op.   */
                drptr->drstat = SYSERR; /* just completed: dsinter      */
        else                            /* resumes calling process or   */
                drptr->drstat = OK;     /* deallocates request node     */
        if ( (dsptr->dreqlst=drptr->drnext) != DRNULL)
                dskstrt(dsptr);  /* start next op if another req exists */
        switch (drptr->drop) {

                case DREAD:
                case DSYNC:    /* resume process that requested the op. */
                        ready(drptr->drpid, RESCHYES);
                        return;
                case DWRITE:
                        freebuf(drptr->drbuff); /* deallocate data buff */
                        /* fall through */
                case DSEEK:
                        freebuf(drptr);      /* deallocate request node */
        }
}
```

17.13 Flushing pending Requests

- Because write operations are treated **asynchronously**, at **system shutdown** all **pending requests** must be finished before system can be shutdown, indeed.
- This is done by calling the high-level routine `CONTROL(device, DSKSYNC)`. **This operation suspends the caller until all existing requests have been satisfied**. This is implemented by `dskcntl(..., ...)`, which calls `dsksync(devptr)`.
- All `dskcntl` does is to put request node with operation `DSYNC` at the **end of the request list**; suspends `currpid` and waits till it is started up again (by `dsinter`).
- Then it checks the status of the job just completed, frees the node and returns `status`.

Flushing pending Requests

```
/* dscntl.c - dscntl */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dscntl  --  control disk driver/device
 *------------------------------------------------------------------------
 */
dscntl(devptr, func)
        struct  devsw   *devptr;
{
        int     stat;
        char    ps;

        disable(ps);
        switch (func) {

                case DSKSYNC:
                        stat = dsksync(devptr);
                        break;

                default:
                        stat = SYSERR;
                        break;
        }
        restore(ps);
        return(stat);
}
```

Flushing pending Requests

```
/* dsksync.c - dsksync */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

/*------------------------------------------------------------------------
 *  dsksync  --  wait for all outstanding disk requests before returning
 *------------------------------------------------------------------------
 */
dsksync(devptr)
struct  devsw   *devptr;
{
        struct  dreq    *drptr, *p, *q;
        int     stat;

        if ( (q=(devptr->dvioblk)->dreqlst) == DRNULL )
                return(OK);
        drptr = (struct dreq *) getbuf(dskrbp); /* dsksync allocates    */
        drptr->drdba = 0;                       /* a request node and   */
        drptr->drpid = currpid;                 /* fills it             */
        drptr->drbuff = NULL;
        drptr->drop = DSYNC;
        drptr->drnext = DRNULL;

        /* place at end of request list */

        for (p=q->drnext ; p!=DRNULL ; q=p,p=p->drnext)
                ;
        q->drnext = drptr;      /* store that request at end of dqlist */
        drptr->drstat = SYSERR;
        suspend(currpid);       /* suspend calling process until all   */
        stat = drptr->drstat;   /* requests have been carried out and  */
        freebuf(drptr);         /* process returns -- free req. node   */
        return(stat);
}
```

18 File Systems

18.1 What is a File System?

- A File System is a software that manages **permanent data** object, i.e., objects that **exist longer** than the programs that create and use them.
- Files are kept on **secondary storage** like **disks**, and organized by directories.
- Conceptually, each file consits of a **sequence of data objects** (e.g., a sequence of zero or more bytes).
- Operations on files:
  - `create` and `delete` a file
  - `open` a file, given its name
  - `read` the next object from an open file
  - `write` objects in an open file
  - `close` a file
  - `seek` a specified location within a file

What is a File System?

- A file system provides an **abstract name space** and high-level operations to manipulate objects in that space. The abstract space consists of the set of **valid** file names.
- Names in the abstract space need not correspond to conventional files on secondary storage. They may refer to **devices**, to **services** that the system supplies, or files that reside on another machine.
- E.g., the name `printer` may correspond to a service that prints a copy of data written to it. A name like `foo:bar` may correspond to disk file `bar` accessed through network `foo`.
- Consider general purpose program `cp` that reads data from one file and writes them to another file. If the abstract name space contains names of **devices** and **remote** files, `cp` can be used to
  - print contents of a file on a console terminal (a device),
  - accept input from a console and write it on a disk file, and
  - copy the contents from a file on a remote machine to one on a local machine.

What is a File System?

- Here, a file consists of zero or more **bytes**. Any further structure must be enforced by user-level programs.
- Xinu-Operations on files:
  - Opening a named file (see page 311) **connects** the executing process with the **data on the disk**, and establishes a **pointer to the first byte** on this file.
  - `getc` and `read` **retrieve bytes** of data from file and **advance the pointer**.
  - `putc` and `write` change bytes on disk file and **move pointer** along, possibly **extending** the file.
  - `seek` moves the pointer to the **specified location** in a file.
  - `close` **detaches** the running process from the (disk) file.

18.2 Disk an File Servers

- File systems on diskless machines (**workstations**) pass requests over a network to a **server** machine.

18.3 A Local File System

- Local Files are files that reside on a disk connected to the machine: in this book **only local files** are discussed.

A Local File System

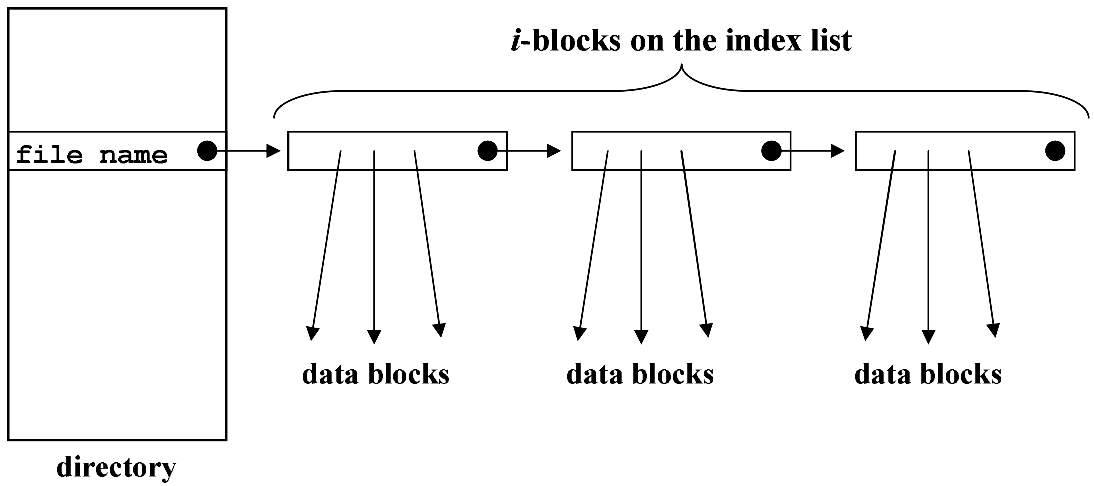
- **Concurrency control** in Xinu consists of the following three elements:
  - Allow multiple processes to access an arbitrary member of files **concurrently**.
  - Allow a single process to access an arbitrary number of files **concurrently**.
  - **Restrict** access to each file to **at most one process**.
- This explains the presence of **three semaphores** in each disk control bock (see page 295):
  - `dibsem`: for accessing list of index blocks (to prevent interference, i.e. writing processes obtain exclusive access to **index area**)
  - `dflsem`: mutual exclusion semaphore for controlling access to the **freelist of index blocks**
  - `ddirsem`: mutual exclusion semaphore controlling access to **in-core directory**

18.4 Data Structures for the File System

- The figure below shows how a disk is partitioned into **three separate areas**:

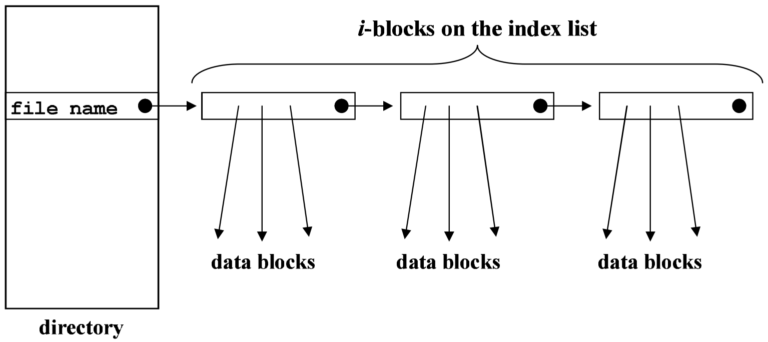
directory	**index**	**data area**
data block 0	from data block 1 onwards	beyond

- The following figure shows how these three separate areas are logically implemented:



Data Structures for the File System

- The **Directory** (page 322) contains:
  - pointers to the list of **free index blocks** `d_filst`,
  - the list of **free data blocks** `d_fblst`,
  - the names of all files and corresponding pointers to the index list of each file `d_files[NFDES]`,
  - the number of bytes in each file `fdlen`.
- **Index Blocks** (see page 314):
  - Each file on the disk has its own index list: pointed at by a pointer, `fdiba`, to a singly linked list of index blocks (of 64 bytes) containing pointer to data blocks.



Data Structures for the File System

- **Data Area**:
  - Disk blocks (of 512 bytes) in data area are called **data blocks** (hence 8 index blocks fit in one data block).
  - The file system **allocates** unused data blocks from a **free list** `dskdbp` (for disk data-block buffer pool) and **returns** them when the file is deleted.
  - Data blocks contain **no pointers to link them together** when accessed by index blocks; that info resides in the **index blocks of that file**.
  - However, when data blocks reside on the free list `dskdp`, they **do not contain a pointer field** to link them into a singly linked list!

18.5 The Index Manager

```
/* iblock.h - ibtodb, ibdisp */

typedef int             IBADDR;         /* iblocks addressed 0,1,2,...  */

#define IBLEN           29              /* # d-block ptrs in an i-block */
#define IBNULL          -1              /* null pointer in i-block list */
#define IBAREA          1               /* start if iblocks on disk     */
#define IBWDIR          TRUE            /* ibnew: write directory       */
#define IBNWDIR         FALSE           /* ibnew: don't write directory */

struct  iblk            {               /* index block layout           */
        long            ib_byte;        /* first data byte indexed by   */
                                        /*  this index block            */
        IBADDR          ib_next;        /* address of next i-block      */
        DBADDR          ib_dba[IBLEN];  /* ptrs to data blocks indexed  */
};

#define ibtodb(ib)      (((ib)>>3)+IBAREA)/* iblock to disk block addr. */
#define ibdisp(ib)      (((ib)&07)*sizeof(struct iblk))
```

The Index Manager

- Each i-block contains an **array of pointers to data blocks**. This array contains addresses of 29 (`IBLEN`) data blocks, each of which is 512 bytes long.
- How does the software know where to find an i-block given its address?
  - It must know where the index area **starts** on the disk, an
  - **how many** i-blocks are contained in each physical disk block
- In our design, the disk directory occupies disk block zero, so the index area starts at disk block one. Thus, i-block zero through seven lie in physical block one.
- Procedure **`ibtodb`** contains code that converts an i-block-address to the correct physical disk address. Procedure **`ibdisp`** converts an i-block address into the byte displacement within its physical block.

18.6 Operations on i-Blocks

18.6.1 Clearing an i-Block

- The file system initialization routine links all i-block into the free list when it builds an empty file system at system start.
- As the file system allocates i-blocks from the free list, it needs to clean old information. This is done by procedure `ibclean`, below.

```
/*------------------------------------------------------------------------
 *  ibclear  --  clear in-core copy of an iblock
 *------------------------------------------------------------------------ */
ibclear(ibptr, ibbyte)
struct  iblk    *ibptr;
long    ibbyte;
{
        int     i;
        ibptr->ib_byte = ibbyte;
        for (i=0 ; i<IBLEN ; i++)
                ibptr->ib_dba[i] = DBNULL;
        ibptr->ib_next = IBNULL;
}
```

18.6.2 Reading an i-Block

- To read an i-block, the system maps its address to a **physical disk block address**, reads the physical disk block, and copies the appropriate area from the physical block into the desired location.
- File `ibget.c` contains the code, as below; it gets an i-block from a disk `diskdev`, given its number `inum` and writes its contents into a buffer starting with address `loc`.
- `ibger` **allocates space** for the physical disk block from the system buffer pool `dskdbp`. After it reads the physical block and extracts the desired i-block, it **releases the storage** with `freebuf`.

Reading an i-Block

```
/* ibget.c - ibget */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  ibget  --  get an iblock from disk given its number
 *------------------------------------------------------------------------
 */
ibget(diskdev, inum, loc)
int     diskdev;
IBADDR  inum;
struct  iblk    *loc;
{
        char    *from, *to;
        int     i;
        char    *buff;

        buff = getbuf(dskdbp);
        read(diskdev, buff, ibtodb(inum));
        from = buff + ibdisp(inum);
        to = (char *)loc;
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        freebuf(buff);
}
```

18.6.3 Writing an i-Block

- i-blocks from several files may occupy the same physical block.
- Because processes my try to write i-blocks into the **same** physical block **concurrently**, to prevent interference, writing processes **must obtain exclusive access** to the index area through a semaphore, `dibsem`, the i-block mutual exclusion semaphore in the disk control block (see page 287).
- `ibput` writes an i-block back to disk given its number.
- `ibput` allocates space for a physical disk block from the system buffer pool `dskdbp` (for disk data-block pool), reads the appropriate physical block, copies in the changed i-block, and writes the physical block back to disk.

Writing an i-Block

```
/* ibput.c - ibput */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  ibput  --  write an iblock back to disk given its number
 *------------------------------------------------------------------------
 */
ibput(diskdev, inum, loc)
int     diskdev;
IBADDR  inum;
struct  iblk    *loc;
{
        DBADDR  dba;
        char    *buff;
        char    *to, *from;
        int     i;
        int     ibsem;

        dba = ibtodb(inum);
        buff = getbuf(dskdbp);
        ibsem = ((struct dsblk *)devtab[diskdev].dvioblk)->dibsem;
        wait(ibsem);
        read(diskdev, buff, dba);
        to = buff + ibdisp(inum);
        from = (char *)loc;
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        write(diskdev, buff, dba);
        signal(ibsem);
        return(OK);
}
```

18.6.4 Allocating *i*-Blocks from the Free List

- Procedure `ibnew` allocates a new *i*-block from the free list of *i*-blocks on a disk.
- The file system **allocates** an *i*-block from the free list whenever it needs one for an index, and returns *i*-blocks to the free list when it **deallocates** a file.
- Procedure `ibnew` obtains the **next free *i*-block** and returns its identification.
- Concurrency control complicates what is, otherwise, a simple procedure; `ibnew` first **obtains exclusive use of the free list** by waiting for the **free list mutual exclusion semaphore** `dflsem` associated with the disk `diskdev`.
- Then `ibnew` retrieves the identification of the first free *i*-block from the directory. It **updates the free list** by reading the first *i*-block and making the directory entry point to its successor.
- After changing that directory, it writes a copy back to the disk. Finally, `ibnew` cleans the allocated *i*-block and writes it to disk.

Allocating i-Blocks from the Free List

```
/* ibnew.c - ibnew */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  ibnew  --  allocate a new iblock from free list on disk
 *------------------------------------------------------------------------
 */
ibnew(diskdev, writedir)
int     diskdev;
Bool    writedir;
{
        struct  dir     *dirptr;
        struct  iblk    iblock;
        IBADDR  inum;
        int     i;
        int     sem;

        sem =  ((struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        dirptr = dsdirec(diskdev);
        wait(sem);
        inum = dirptr->d_filst;
        ibget(diskdev, inum, &iblock);
        dirptr->d_filst = iblock.ib_next;
        if (writedir)
                write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
        ibclear(&iblock, 0L);
        ibput(diskdev, inum, &iblock);
        return(inum);
}
```

18.6.5 Returning i-Blocks to the Free List

- When a file is deleted, its index and data blocks must be **returned to their appropriate free list**. Procedure `iblfree` performs this task.
- `iblfree` takes an i-block address as argument, and releases the i-blocks on that list (together with **all associated data blocks**).
- Thus, a single call to `iblfree` releases all the space used by a file.

Returning *i*-Blocks to the Free List

```
/*------------------------------------------------------------------------
 *  iblfree  --  free a list of iblocks given the number of the first
 *------------------------------------------------------------------------
 */
iblfree(diskdev, iblist)
int     diskdev;
IBADDR  iblist;
{
        IBADDR  ilast;
        struct  iblk    iblock;
        struct  dir     *dirptr;
        int     sem;
        DBADDR  dba;
        int     j;

        if (iblist == IBNULL)
                return(OK);
        dirptr = dsdirec(diskdev);
        ibget(diskdev, iblist, &iblock);
        for (ilast=iblist ; iblock.ib_next!=IBNULL ;) {
                for (j=0 ; j<IBLEN ; j++)
                        if ( (dba=iblock.ib_dba[j]) != DBNULL)
                                lfsdfree(diskdev, dba);
                ilast = iblock.ib_next;
                ibget(diskdev, ilast, &iblock);
        }
        for (j=0 ; j<IBLEN ; j++)
                if ( (dba=iblock.ib_dba[j]) != DBNULL)
                        lfsdfree(diskdev, dba);
        sem = ( (struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        wait(sem);
        iblock.ib_next = dirptr->d_filst;
        dirptr->d_filst = iblist;
        ibput(diskdev, ilast, &iblock);
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
        return(OK);
}
```

18.7 The Directory Structure

- The directory contains an entry for each file on this disk.
- The file entry structure includes the file's name and the address of the first i-block on the file's index list.
- The directory also contains the total number of i-blocks on this disk, as well as pointers to the list of free blocks.

The Directory Structure

```
/* dir.h */

#define FDNLEN  10                      /* length of file name + 1      */
#define NFDES   28                      /* number of files per directory*/

struct  fdes    {                       /* description of each file     */
        long    fdlen;                  /* length in bytes              */
        IBADDR  fdiba;                  /* first index block            */
        char    fdname[FDNLEN];         /* zero-terminated file name    */
};

struct  dir     {                       /* directory layout             */
        int     d_iblks;                /* i-blocks on this disk        */
        DBADDR  d_fblst;                /* pointer to list of free blks */
        IBADDR  d_filst;                /* pointer to list of free iblks*/
        int     d_id;                   /* disk identification integer  */
        int     d_nfiles;               /* current number of files      */
        struct  fdes    d_files[NFDES]; /* descriptions of the files    */
};

struct  freeblk {                       /* shape of block on free list  */
        DBADDR  fbnext;                 /* address of next free block   */
};
```

18.8 Using the Device Switch Table for Files

- File system software establishes connections between **running processes** and **files on disk** so that read/write operations can be mapped to the correct location on a disk.
- To do so, we use the **device switch table** machinery (see pages 154+155), i.e., file pseudo devices 5, 6, 7, 8 – corresponding to pseudo devices `file1`, ..., `file4`. The term **pseudo** refers to the fact that a file is not a real peripheral device; hence it has no lower-half device drivers.
- A set of pseudo devices `file1`, ..., `file4` is added to the device switch table `devtab` such that each pseudo device controls **one** file dynamically. Pseudo devices perform as operations `read`, `getc`, `write`, `putc` and `close`.

Using the Device Switch Table for Files

- When a process **opens** a file, the file system searches for a currently unused pseudo device, sets up the **control block** for that file plus device; and returns the pseudo device identifier to the caller.
- After opening the file, the process uses that **device identifier** to write, read etc. These high level operations are **mapped by `devtab` to driver routines** for pseudo devices.
- Differently from the upper-half/lower-half mechanism for implementing drivers, **only upper-half routines** exist here. Upper-half routines call the disk operations discussed in chapter 17 – see control block for pseudo device files on pages 324.

Using the Device Switch Table for Files

```
/* file.h */

/* Local disk layout: disk block 0 is directory, then index area, and   */
/* then data blocks.  Each disk block (512 bytes) in the index area     */
/* contains 8 iblocks, which are 64 bytes long.  Iblocks are referenced */
/* relative to 0, so the disk block address of iblock k is given by     */
/* truncate(k/8)+1.  The offset of iblock k within its disk block is    */
/* given by 64*remainder(k,8).  The directory entry points to a linked  */
/* list of iblocks, and each iblock contains pointers to IBLEN (29) data*/
/* blocks. Index pointers contain a valid data block address or DBNULL. */

#include <iblock.h>
#include <file.h>

struct  flblk   {                       /* file "device" control block  */
        int     fl_id;                  /* file's "device id" in devtab */
        int     fl_dev;                 /* file is on this disk device  */
        int     fl_pid;                 /* process id accessing the file*/
        struct  fdes    *fl_dent;       /* file's in-core dir. entry    */
        int     fl_mode;                /* FLREAD, FLWRITE, or both     */
        IBADDR  fl_iba;                 /* address of iblock in fl_iblk */
        struct  iblk    fl_iblk;        /* current iblock for file      */
        int     fl_ipnum;               /* current iptr in fl_iblk      */
        long    fl_pos;                 /* current file position (bytes)*/
        Bool    fl_dch;                 /* has fl_buff been changed?    */
        char    *fl_bptr;               /* ptr to next char in fl_buff  */
        char    fl_buff[DBUFSIZ];       /* current data block for file  */
};

#ifdef  Ndf
extern  struct  flblk   fltab[];
#endif
```

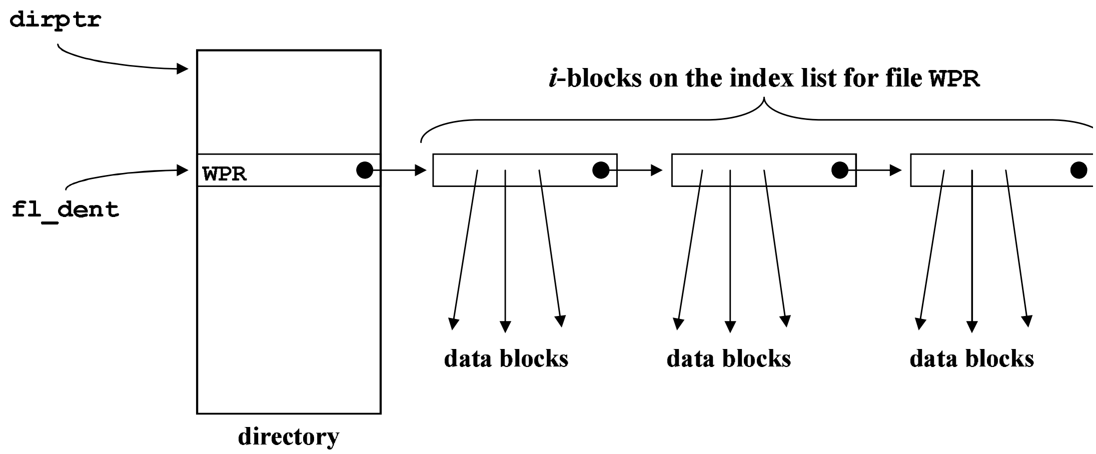
Using the Device Switch Table for Files

- Upper-half routines for pseudo-device disk files call
  - `dsread(devptr, ...)`,
  - `dswrite(devptr, ...)`,
  - `dsseek(devptr, ...)`, and
  - `dscntl(devptr, ...)`.
- However, these peudo-devices should also execute `getc`, `putc`, and `close`.
- Since there's no `dsgetc`, `dsputc` at our disposal, within control block for pseudo-devices **extra administration is required** in the form of addresses, indexes and pointers, for **relating the current positions in open files**, e.g., the cursor, together with **all pointers in the chain** leading from the **in-core directory** file descriptor to the **cursor** – see page 324.
- This extra information resides in the **pseudo-file control block** array `fltab[]`.

Using the Device Switch Table for Files

- We display schematically an in-core directory for a particular disk with device number `ddev` (in device switch table); its address is

  `dirptr = dev[ddev].dvioblk->ddir`



  `fl_denk` is a pointer to file `WPR`'s **in-core directory entry**.

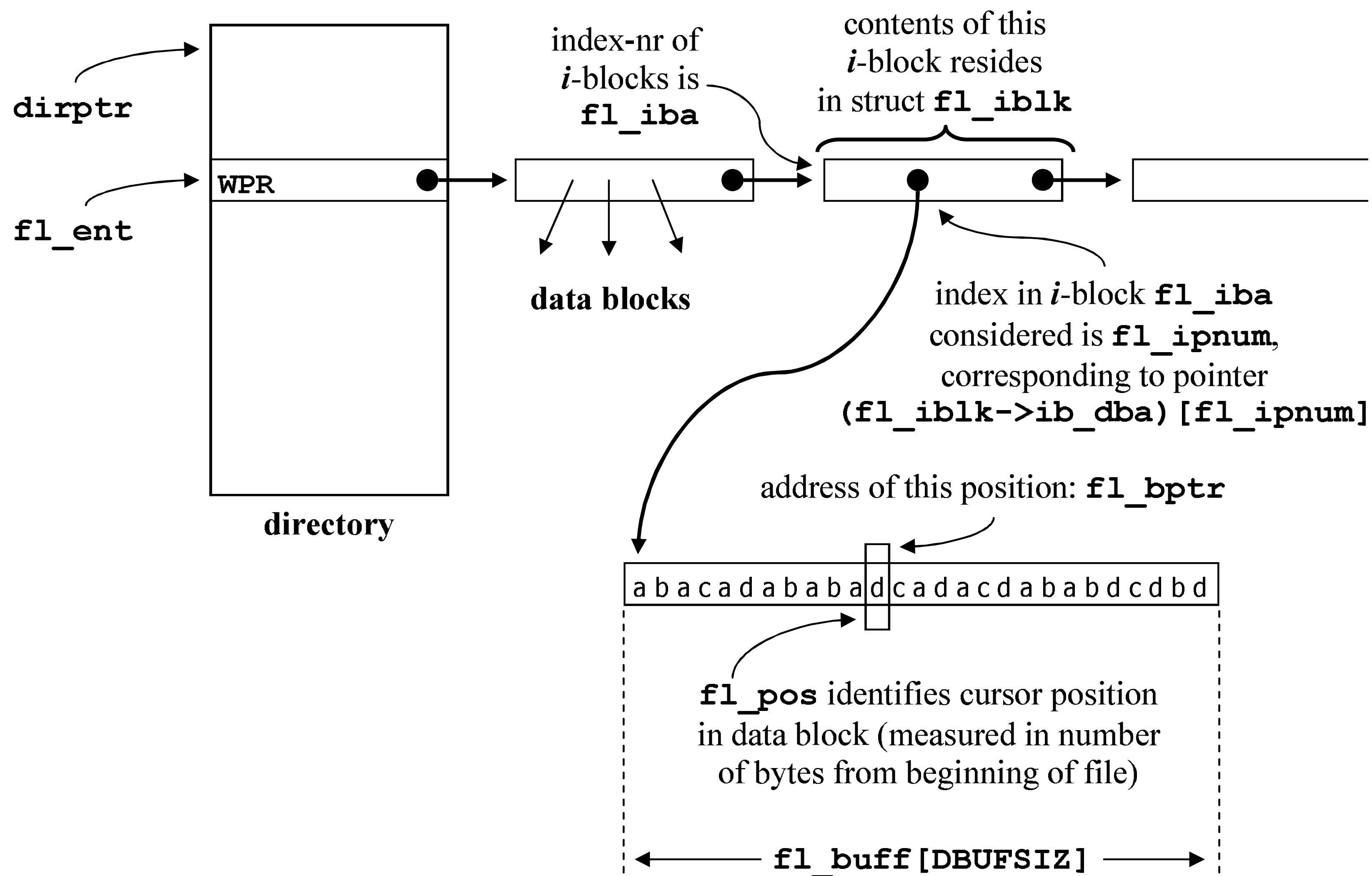
- Operations on file `WPR` take place on pseudo-device `fl_id` and relate to physical operation on disk `fl_dev`.

- The process accessing file `WPR`, `fl_pid`, **must be known**, e.g., to wake this process up after suspending for a read operation on disk `fl_dev`.

Using the Device Switch Table for Files

- Opening a file establishes a **cursor** that points to the **beginning** of the file.
- As processes read or write data, the cursor **moves through the file**.
- Field `fl_pos` gives the **current cursor position** measured in bytes from the beginning of the file.
- The file-control block **contains info** to enable upper-half routines to **retrieve and modify data** found at current cursor position:
  - `fl_iblk` contains a copy of the *i*-block that indexes the current position.
  - Field `fl_ipnum` identifies pointer to data block containing the current position.
  - Data block in `fl_buff` includes current position.
  - Field `fl_bptr` points to character in the buffer that is found at current position.

Using the Device Switch Table for Files



18.9 Establishing a Pseudo-Device

Making a **connection** between a **running program** and a **named file** involves:

- **Searching the directory** to see if the name is valid – this is done by calling `dfdsrch`: searches disk directory for position of given file name.
- **Allocating a pseudo device** – done by calling `dfalloc`: allocate a device table entry for a disk file and return its id.
- **Initializing the control block** and the cursor to the beginning of the file – done by calling `dsopen`: open/create a file on the specified disk device.

Establishing a Pseudo-Device

`open(4, name, mode)` – in user program (with `mode` being read/write mode)

⇒ `dsopen(devptr, name, mode)` – open/create a file on the specified disk device on a pseudo-device, whose identity is returned to caller

⇒ `dsdsrch(dsptr, name, mbits)` – search disk directory of disk control block pointed at by `dsptr`, for position of file name „name“

⇒ `dfalloc()` – allocate a device table entry for a disk file, return its id

Establishing a Pseudo-Device

- In Xinu, the user can specify, when opening a file, whether the file should be **old or new** (i.e., whether it must or must not exist), and whether it will be read or written.
- Such specifications are made by parsing a **mode string** to the procedure that opens files.
- To parse the mode string and **convert it** to a single integer, the open routine calls `dfckmd`, shown below.
- `dfckmd` returns an integer with **mode bits** set according to the symbolic constants in `file.h`.
- Once the mode string has been parsed, the **directory can be searched**. Procedure `dfdsrch` uses the mode integer created by `dfckmd`.
- It **searches for the specified filename**, creating a new file if the mode bits allow creation and the name does not exist.

Establishing a Pseudo-Device

```
/*------------------------------------------------------------------------
 *  dfckmd  --  parse file mode argument and generate actual mode bits
 *------------------------------------------------------------------------
 */
dfckmd(mode)
char    *mode;
{
        int     mbits;
        char    ch;

        mbits = 0;
        while (ch = *mode++)
                switch (ch) {

                        case ´r´:   if (mbits&FLREAD) return(SYSERR);
                                    mbits |= FLREAD;
                                    break;

                        case ´w´:   if (mbits&FLWRITE) return(SYSERR);
                                    mbits |= FLWRITE;
                                    break;

                        case ´o´:   if (mbits&FLOLD || mbits & FLNEW)
                                        return(SYSERR);
                                    mbits |= FLOLD;
                                    break;

                        case ´n´:   if (mbits&FLOLD || mbits & FLNEW)
                                        return(SYSERR);
                                    mbits |= FLNEW;
                                    break;

                        default:    return(SYSERR);
                }
        if ((mbits&FLREAD) == (mbits&FLWRITE)) /* default: allow R + W */
                mbits |= (FLREAD|FLWRITE);
        return(mbits);
}
```

Establishing a Pseudo-Device

```
/*------------------------------------------------------------------------
 *  dfdsrch  --  search disk directory for position of given file name
 *------------------------------------------------------------------------
 */
struct  fdes    *dfdsrch(dsptr, filenam, mbits)
struct  dsblk   *dsptr;
char    *filenam;
int     mbits;
{
        struct  dir     *dirptr;
        struct  fdes    *fdptr;
        int     len;
        int     inum;

        if ( (len=strlen(filenam))<=0 || len>=FDNLEN)
                return((struct fdes *)SYSERR);
        dirptr = dsdirec(dsptr->dnum);
        for (i=0 ; i<dirptr->d_nfiles ; i++)
                if (strcmp(filenam, dirptr->d_files[i].fdname) == 0)
                        if ( (mbits&FLNEW) != 0)
                                return((struct fdes *)SYSERR);
                        else
                                return(&dirptr->d_files[i]);
        wait(dsptr->ddirsem);
        if ( (mbits&FLOLD) || dirptr->d_nfiles >= NFDES) {
                signal(dsptr->ddirsem);
                return((struct fdes *)SYSERR);
        }
        inum = ibnew(dsptr->dnum, IBNWDIR);
        fdptr = &(dirptr->d_files[dirptr->d_nfiles++]);
        fdptr->fdlen = 0L;
        strcpy(fdptr->fdname, filenam);
        fdptr->fdiba = inum;
        write(dsptr->dnum, dskbcpy(dirptr), DIRBLK);
        signal(dsptr->ddirsem);
        return(fdptr);
}
```

Establishing a Pseudo-Device

- `dfdsrch` first checks whether the name is **valid**. Then it **searches** the directory.
- If a match is found, it uses the mode bits to determine whether an **old file is allowed**. If no match is found, `dfdsrch` uses the mode bits to determine whether a **new file is allowed**.
- If a new file is needed, `dfdsrch` **creates one** by adding the new name to the directory and allocating an i-block for the file on the free list.
- Once a file has been found or created, a **connection** can be established by allocating a pseudo-device and initializing its control block.
- Procedure `dfalloc` **allocates** the pseudo-device. Allocation is straight-forward. `dfalloc` searches the array of pseudo-device control blocks looking for an **unused device**.
- When it finds an unused device, `dfalloc` marks the device busy by storing the caller's process id in the process id field. It then returns the index of the file control block it reserved.

Establishing a Pseudo-Device

```
/* dfalloc.c - dfalloc */

/*------------------------------------------------------------------------
 *  dfalloc  --  allocate a device table entry for a disk file; return id
 *------------------------------------------------------------------------
 */
#ifdef  Ndf
dfalloc()               /* assume exclusion for dir. provided by caller */
{
        int     i;

        for (i=0 ; i<Ndf ; i++)
                if (fltab[i].fl_pid == 0) {
                        fltab[i].fl_pid = getpid();
                        return(i);
                }
        return(SYSERR);
}
#endif
```

Establishing a Pseudo-Device

- Procedure `dsopen` uses the three procedures described above to **make a connection** between a **running program** and a **disk file**.
- It calls `dfckmd` to check the **mode string** and to convert it into an integer, `dfdsrch` to search the directory, and `dfalloc` to allocate a pseudo-device for the file.
- Finally, `dsopen` fills in the **file control block** by setting the current position to zero and reading the first *i*-block from the file's index list. It **returns the file's device id** to the caller so it can be used in operations like `read` or `write`.
- Because the directory maps names to disk files, the **open operation** is associated **with the disk file**, **not with the individual files**.

Establishing a Pseudo-Device

```
/* dsopen.c - dsopen */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  dsopen  --  open/create a file on the specified disk device
 *------------------------------------------------------------------------
 */
#ifdef  Ndf
dsopen(devptr, filenam, mode)
struct  devsw   *devptr;
char    *filenam;
char    *mode;
{
        struct  dir     *dirptr;
        struct  flblk   *flptr;
        struct  fdes    *fdptr;
        DBADDR  dba;
        int     mbits, findex;
        int     retcode;
        char    ps;

        disable(ps);
        dirptr = dsdirec(devptr->dvnum);
        if ( (mbits=dfckmd(mode)) == SYSERR)
                retcode = SYSERR;
        else if( (int)(fdptr=dfdsrch(devptr->dvioblk,filenam,mbits))
                == SYSERR)
                retcode = SYSERR;
        else if ( (findex=dfalloc()) == SYSERR)
                retcode = SYSERR;
```

Establishing a Pseudo-Device

File `dsopen.c` continued:

```
    else {
        flptr = &fltab[findex];
        flptr->fl_dev = devptr->dvnum;
        flptr->fl_dent = fdptr;
        flptr->fl_mode = mbits & FLRW;
        flptr->fl_iba = fdptr->fdiba;
        ibget(flptr->fl_dev, flptr->fl_iba, &(flptr->fl_iblk));
        flptr->fl_pos = 0L;
        flptr->fl_dch = FALSE;
        dba = flptr->fl_iblk.ib_dba[flptr->fl_ipnum = 0];
        if (dba != DBNULL) {
            read(flptr->fl_dev, flptr->fl_buff, dba);
            flptr->fl_bptr = flptr->fl_buff;
        } else
            flptr->fl_bptr = &flptr->fl_buff[DBUFSIZ];
        retcode = flptr->fl_id;
    }
    restore(ps);
    return(retcode);
}
#endif
```

Example of Call Chain involved in Reading a File

- We **trace the call chain** involved in reading a file, identified by a control block with address `devptr`, into a buffer starting at address `buff`, involving `count` number of characters.
- Observe how first procedures in the **upper-half pseudo-device driver** routines are called. Using the file pseudo device control block `flptr` as intermediary, the **disk-related routines** are called.
- First routines in the **upper-half** of the disk device driver are called. Then a write-request node is put into the `dreqlist` of the disk driver control block. This leads to calling procedures from the lower-half disk device driver.
- This interface uses the disk through the dtc disk host interface at address 0177460_8, calling the Xebec disk controller, and, ultimately, the disk hardware.

Example of Call Chain involved in Reading a File

`read(devptr, buff, count)` – in user program with `devptr` pointing to control block for previously opened disk file

⇒ `lfread(devptr, buff, count)` – read from a previously opened disk file

⇒ `lfgetc(devptr)` – get next character from (buffered) disk file

⇒ `lfsflush(flptr)` – flush data and i-block for a file

⇒ file pseudo device control block

These steps happen in the (upper half) pseudo-device driver routines. Next: the disk-related routines are called.

Example of Call Chain involved in Reading a File

⇒ `write(4, dskcpy(... fl_buff), dba)`

⇒ `dswrite(devptr, buff, block)` – write a block (system buffer) onto a disk device, whose control block is pointed to by `devptr`

⇒ *write-request node* – put into `dreqlist` of disk driver control block; here's the interface between the upper- and lower-half disk driver routines

⇔ `dsinter(dsptr)` – `dtptr->dt_csr = DTGO` – initialize disk driver and read disk directory into main memory

⇔ `dskstrt(dsptr)` – `dtptr->dt_csr == DTINTR | DTGO`

⇒ dtc disk host interface – at address 0177460_8

⇔ Xebec disk controller

⇔ disk hardware

18.10 Pseudo-Device Driver Routines

- Instead of starting hardware, the upper-half file routines carry out requests by performing input and output operations on the particular disk device at hand.
- Procedure `lfsnewd` allocates a data block from the free list almost exactly the same way `ibnew` allocates an index block.
- Procedure `lfsdfree` reverses the actions of `lfsnewd` by returning a data block to the free list.

Pseudo-Device Driver Routines

```
/* lfsnewd.c - lfsnewd */

#define  DFILLER  '+'

/*------------------------------------------------------------------------
 *  lfsnewd  --  allocate a new data block from free list on disk
 *------------------------------------------------------------------------
 */
lfsnewd(diskdev, flptr)
int     diskdev;
struct  flblk   *flptr;
{
        struct  iblock  *ibptr;
        struct  dir     *dirptr;
        struct  freeblk *fbptr;
        char    *buf;
        int     sem;
        DBADDR  dba;
        int     i;

        dirptr = dsdirec(diskdev);
        fbptr = (struct freeblk *) (buf = flptr->fl_buff);
        sem = ((struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        wait(sem);
        dba = dirptr->d_fblst;
        read(diskdev, fbptr, dba);
        dirptr->d_fblst = fbptr->fbnext;
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
        for (i=0 ; i<DBUFSIZ ; i++)
                *buf++ = DFILLER;
        write(diskdev, dskbcpy(fbptr), dba);
        return(dba);
}
```

Pseudo-Device Driver Routines

```
/* lfsdfree.c - lfsdfree */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  lfsdfree  --  free a data block given its address
 *------------------------------------------------------------------------
 */
lfsdfree(diskdev, dba)
int     diskdev;
DBADDR  dba;
{
        struct  dir     *dirptr;
        int     dirsem;
        struct  freeblk *buf;

        dirptr = dsdirec(diskdev);
        dirsem = devtab[diskdev].dvioblk->dflsem;
        buf = (struct freeblk *)getbuf(dskdbp);
        wait(dirsem);
        buf->fbnext = dirptr->d_fblst;
        dirptr->d_fblst = dba;
        write(diskdev, buf, dba);
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(dirsem);
        return(OK);
}
```

18.10.1 Index Manager Routines

- The file system **cannot afford** to write a data block **every time** the program changes one character. Instead, it **waits** until the block has been **filled** and a new one is needed, before writing the block to the disk.
- Changed blocks must also be written when the user **closes the file** (even if its block has not been filled), and when the user positions the file pointer outside the current block by calling `seek`.
- Of course, the file system **should not copy the buffer** to disk unless it has been changed, because disk access **require much time**.
- Procedure `lfsflush` writes data from the buffer to disk **if the data has been changed**. It computes the disk block address of the current buffer by looking in the *i*-block (field `fl_iblk`), and writes a copy of the buffer to that address.
- Procedure `lfsetup` positions a file at the specified location by **finding the correct index block** and **data block**.

Index Manager Routines

```
/* lfsflush.c - lfsflush */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  lfsflush  --  flush data and i-block for a file
 *------------------------------------------------------------------------
 */
lfsflush(flptr)
struct  flblk   *flptr;
{
        DBADDR  dba;

        if (!flptr->fl_dch)
                return(SYSERR);
        dba = flptr->fl_iblk.ib_dba[flptr->fl_ipnum];
        write(flptr->fl_dev, dskbcpy(flptr->fl_buff), dba);
        flptr->fl_dch = FALSE;
        return(OK);
}
```

Index Manager Routines

File `dsopen.c` continued:

```
/* lfsetup.c - lfsetup */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  lfsetup  --  set up appropriate iblock and data block in memory
 *------------------------------------------------------------------------
 */
lfsetup(diskdev, flptr)
int     diskdev;
struct  flblk   *flptr;
{
        struct  iblk    *ibptr;
        int     displ, i;
        long    ibrange;
        IBADDR  nextib;
        DBADDR  dba;

        ibrange = (long) (IBLEN * DBUFSIZ);
        ibptr = &flptr->fl_iblk;
        if (flptr->fl_pos < ibptr->ib_byte) {
                flptr->fl_iba = (flptr->fl_dent)->fdiba;
                ibget(diskdev, flptr->fl_iba, ibptr);
        }
```

Index Manager Routines

File `lfsetup.c` continued:

```
	while (ibptr->ib_byte+ibrange <= flptr->fl_pos) {
		if (ibptr->ib_next == IBNULL) {
			ibptr->ib_next = ibnew(diskdev, IBWDIR);
			ibput(diskdev, flptr->fl_iba, ibptr);
			flptr->fl_iba = ibptr->ib_next;
			ibclear(ibptr, (long)ibptr->ib_byte+ibrange);
			ibput(diskdev, flptr->fl_iba, ibptr);
		} else {
			flptr->fl_iba = ibptr->ib_next;
			ibget(diskdev, flptr->fl_iba, ibptr);
		}
	}
	displ = (int) (flptr->fl_pos - ibptr->ib_byte);
	for (flptr->fl_ipnum=0 ; displ>=DBUFSIZ ; displ-=DBUFSIZ)
		flptr->fl_ipnum++;
	flptr->fl_bptr = flptr->fl_buff + displ;
	if ( (dba=ibptr->ib_dba[flptr->fl_ipnum]) == DBNULL) {
		ibptr->ib_dba[flptr->fl_ipnum] = lfsnewd(diskdev,flptr);
		ibput(diskdev, flptr->fl_iba, ibptr);
	} else
		read(diskdev, flptr->fl_buff, dba);
	flptr->fl_dch = FALSE;
}
```

18.10.2 The Pseudo-Device `seek` Routine

- The `seek` operation **moves the current file position** to a given location without transferring data.
- `lfseek` implements the seek operation.

The Pseudo-Device seek Routine

```
/* lfseek.c - lfseek */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  lfseek  --  seek to a specified position of a file
 *------------------------------------------------------------------------
 */
lfseek(devptr, offset)
struct  devsw   *devptr;
long    offset;
{
        struct  flblk   *flptr;
        int     retcode;
        char    ps;

        disable(ps);
        flptr = (struct flblk *)devptr->dvioblk;
        if (flptr->fl_mode & FLWRITE) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
        } else if (offset > (flptr->fl_dent)->fdlen) {
                        restore(ps);
                        return(SYSERR);
        }
        flptr->fl_pos = offset;
        retcode = lfsetup(flptr->fl_dev, flptr);
        restore(ps);
        return(retcode);
}
```

18.10.3 The Pseudo-Device `getc` Routine

- Once a file has been opened and the index is in place, input from the file is trivial. It consists of extracting the next character from the buffer and advancing the buffer pointer.
- Procedure `lfgetc` performs that operation.

The Pseudo-Device getc Routine

```
/* lfgetc.c - lfgetc */

/*------------------------------------------------------------------------
 *  lfgetc  --  get next character from (buffered) disk file
 *------------------------------------------------------------------------
 */
lfgetc(devptr)
struct  devsw   *devptr;
{
        struct  flblk   *flptr;
        char    nextch;
        char    ps;

        disable(ps);
        flptr = (struct flblk *)devptr->dvioblk;
        if (flptr->fl_pid!=currpid || !(flptr->fl_mode&FLREAD)) {
                restore(ps);
                return(SYSERR);
        }
        if (flptr->fl_pos >= (flptr->fl_dent)->fdlen) {
                restore(ps);
                return(EOF);
        }
        if (flptr->fl_bptr >= &flptr->fl_buff[DBUFSIZ]) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
                lfsetup(flptr->fl_dev, flptr);
        }
        nextch = *(flptr->fl_bptr)++;
        flptr->fl_pos++;
        restore(ps);
        return(nextch);
}
```

18.10.4 The Pseudo-Device putc Routine

```
/* lfputc.c - lfputc */

/*------------------------------------------------------------------------
 *  lfputc  --  put a character onto a (buffered) disk file
 *------------------------------------------------------------------------
 */
lfputc(devptr, ch)
struct  devsw   *devptr;
char    ch;
{
        struct  flblk   *flptr;
        char    ps;

        disable(ps);
        flptr = (struct flblk *) devptr->dvioblk;
        if (flptr->fl_pid != currpid || !(flptr->fl_mode&FLWRITE)) {
                restore(ps);
                return(SYSERR);
        }
        if (flptr->fl_bptr >= &flptr->fl_buff[DBUFSIZ]) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
                lfsetup(flptr->fl_dev, flptr);
        }
        if (flptr->fl_pos >= (flptr->fl_dent)->fdlen)
                (flptr->fl_dent)->fdlen++;
        flptr->fl_pos++;
        *(flptr->fl_bptr)++ = ch;
        flptr->fl_dch = TRUE;
        restore(ps);
        return(OK);
}
```

18.10.5 The Pseudo-Device read Routine

```
/* lfread.c - lfread */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <dir.h>

/*------------------------------------------------------------------------
 *  lfread  --  read from a previously opened disk file
 *------------------------------------------------------------------------
 */
lfread(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        int     done;
        int     ichar;

        if (count < 0)
                return(SYSERR);
        for (done=0 ; done < count ; done++)
                if ( (ichar=lfgetc(devptr)) == SYSERR)
                        return(SYSERR);
                else if (ichar == EOF ) {       /* EOF before finished */
                        if (done == 0)
                                return(EOF);
                        else
                                return(done);
                } else
                        *buff++ = (char) ichar;
        return(done);
}
```

18.10.6 The Pseudo-Device `write` Routine

```
/* lfwrite.c - lfwrite */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  lfwrite  --  write 'count' bytes onto a local disk file
 *------------------------------------------------------------------------
 */
lfwrite(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        int     i;

        if (count < 0)
                return(SYSERR);
        for (i=count; i>0 ; i--)
                if (lfputc(devptr, *buff++) == SYSERR)
                        return(SYSERR);
        return(count);
}
```

18.10.7 The Pseudo-Device `close` Routine

- When a process finishes using a file, it calls `close` to flush unwritten buffers out to disk, and detach the file from the process.
- Procedure `lfclose` implements the `close` operation.
- It calls `lfsflush` to write data from buffer if needed, and **assigns the process id field in the file control block to zero** to indicate that the **pseudo-device can be used again**.
- After it writes the data blocks to disk, `lfclose` writes a copy of the directory to disk in case the **file legth** recorded in the directory entry has changed.

The Pseudo-Device `close` Routine

```
/* lfclose.c - lfclose */

/*------------------------------------------------------------------------
 *  lfclose  --  close a file by flushing output and freeing device slot
 *------------------------------------------------------------------------
 */
lfclose(devptr)
struct  devsw   *devptr;
{
        struct  dsblk   *dsptr;
        struct  dir     *dirptr;
        struct  flblk   *flptr;
        int     diskdev;
        char    ps;

        disable(ps);
        flptr = (struct flblk *) devptr->dvioblk;
        if (flptr->fl_pid != currpid) {
                restore(ps);
                return(SYSERR);
        }
        diskdev = flptr->fl_dev;
        dsptr = (struct dsblk *)devtab[diskdev].dvioblk;
        dirptr = (struct dir *) dsptr->ddir;
        if ( (flptr->fl_mode & FLWRITE) && flptr->fl_dch)
                lfsflush(flptr);
        flptr->fl_pid = 0;
        dsptr->dnfiles--;
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        restore(ps);
        return(OK);
}
```

18.10.8 The Pseudo-Device Initialization Routine

- Although procedure `dsopen` initializes most of the entries in the file control block when it opens a file, some additional initialization is required at system startup.
- The initialization routine assigns field `fl_id` in the file control block **the device identifier** that the **high level routines** call to access the file.
- It also initializes field `fl_pid` to zero, indicating that the pseudo identifier is **not in use**.
- The pseudo-device initialization routine also places a pointer to the file control block in the device witch table (field `dvioblk`) so the upper-half routines can find the correct control block.
- These details are handled by procedure `lfinit`.

The Pseudo-Device Initialization Routine

```
/* lfinit.c - lfinit */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

#ifndef Ndf
#define Ndf     1
#endif
struct  flblk   fltab[Ndf];

/*------------------------------------------------------------------------
 *  lfinit  --  mark disk file 'device' available at system startup
 *------------------------------------------------------------------------
 */
lfinit(devptr)
struct  devsw   *devptr;
{
        struct  flblk   *flptr;

        devptr->dvioblk = flptr = &fltab[devptr->dvminor];
        flptr->fl_pid = 0;
        flptr->fl_id = devptr->dvnum;
        return(OK);
}
```

18.11 Summary

- This chapter concentrates on the part of the file system software that **manages data files on secondary storage**. To keep the interface to files the same as the interface to devices, the software is organized into a **pseudo-device driver** that supports `read`, `write`, `getc`, `putc`, `seek` and `close` operations.
- When a process opens a file, the software establishes a connection to it through the device switch table, allowing high-level I/O routines to **access the file driver just as they access hardware device drivers**.
- Multiple files grow concurrently by using an index to keep track of the data blocks associated with each file. The **index** for a file consists of a **linked list** of nodes called ***i*-blocks**, where each one contains pointers to a set of data blocks.
- When a file is opened, the driver software **reads its first index block** into memory. It also reads a copy of the first data block if the file is non-empty. Subsequent accesses or changes affect the **buffer in memory**.
- Only when the file position **moves outside the current buffer** does the file system copy the buffer back to disk and read another.