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