1.1 Operating system

- The software that controls processing, manages resources and communicate with external devices like disks and printers is called 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 ranges 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 hight-level services of an abstract machine.

1.2 Comer's approach

Comer's book is a guide to the design and implementation of layered operating systems.

It begins with a sample microcomputer and develops every chapter a new layer illustrating the details with programs.

The design ends with a complete, working system that supports multiple processes and a file system.

1.3 What an operating system is not

- An operating system is not a language or compiler.
- It is not a command interpreter, in modern system command interpreter can be chosen by users, or be written to meet their needs.
- An operating system is not library of commands.Modern systems extend the freedom to replace commands to users so they can tailor the computing environments according to their individual tastes.

1.4 An operating system viewed from the outside

- The essence of an operating system lies in the services it provides to user programs. Programs access these services by making system calls.
- System calls look like procedure calls appearing in ordinary program, but transfer to operating system routines when invoked at run-time.
- As a set, the system calls establish a well-defined boundary between the running program and the operating system.
- They define the services that the operating system provides and the interface to those services.

1.4.1 The Xinu operating system described in Comer's book basically:

- reads characters from a keyboard, and displays characters on a terminal
- manages multiple simultaneous computations
- operate timers
- saves files on disk storage devices
- relays messages between programs.

1.4.2 The Xinu small machine environment

Xinu runs on a small, slow microcomputer without using external storage devices.

The entire system is prepared on a larger machine, called a *host*, and downloaded onto microcomputer.

To run a program :

- the user compiles it on the host using a *cross-complier* (a cross-compiler produces code for the micro)
- combines it with code for Xinu using a *cross-loader* (a cross-loader produces an exact memory image for the micro by coping the combining the compiler's output with previously compiled for Xinu)
- after a memory image has been produced, a down-loader copies that image from the host machine into the memory of the micro over a standard serial connection.
- after the micro memory has been filled, execution proceeds on the micro independent of the host.

1.4.3 Xinu services

Programs running under Xinu access services by calling operating system routines. 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.
In example ex1.c a procedure called *"main"*, written in C, is illustrated.

```
/* 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');
}
```

The code on the previous slide produces several conventions :

- `#include <conf.h>` inserts a file of configuration declarations in the source program.
- a definition for a *CONSOLE*, witch usually refers to a terminal connected to the micro.
- This program writes four characters to the terminal: ”h ” , ” i ” , carriage return, and a line feed. The latter two are control characters.
- *ex1.c* is the name of the file.
- The comment `/* ex1.c - main */` gives the name of the file followed by the procedure defined by the file, in this case *main*

1.4.4 Concurrent processing

- Conventional programs are called *sequential* because the programmer imagines a machine executing the code statement-by-statement.
- Operating systems support *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 the illusion:
  - the operating system switch a single processor among multiple programs, allowing it to execute one for a few thousandths of second before moving on to another.
  - Viewed by a human the programs appear to proceed concurrently. This technique is called *multiprogramming*.
- Interactive multiprogramming systems are called *time sharing systems*, when the policy used to switch the processor around gives all users equal amounts of CPU time.
- The operating system switches the CPU among many computations called *processes (jobs, tasks)*

1.4.5 The distinction between programs and Processes

- At any time several processes (computations) may be executing. It may be that no two of them are executing the same program.
- Since one processor is switched among several processes, one process may create another; no guarantee is made about their "speeds" .
- Therefor the system procedures must be designed so that cooperation between processes proceed correctly, independently of their relative speeds.
- For example , the code from `ex2.c` consists of a main program and two procedures, *prA* and *prB*:

```
/* 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');
}
```

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

Exploration

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

The distinction between normal procedure calls and process creation is :

- A procedure call dose 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 to proceed concurrently.

In example `ex2.c`, the first new process prints 'A' and the second new process prints 'B'.

- Because processes execute concurrently, the output is mixture of 'A's and 'B's.
- The process executing the main program exits after the second call to resume.

Example `ex3.c` shows :

- Independent programs need not execute independent code.
- A single program begins executing the main program, calling *create* twice to start new processes
- both execute code for the same procedure *prntr(ch)* :

```
/* 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);
}
```

Exploration (1)

Tow processes running concurrently without affection each other. although they execute the same piece of code.
This shows the notion of process and the notion of program:

- A process consists of code executed by a single process.
- In contrast, processes are not associated with a single piece of code.
- Multiple processes can execute the same code simultaneously.

This shows some hint of the difficulty involved in designing operating systems:

- each piece must be designed to operate correctly by itself,
- it must be guaranteed that it dose not interfere with other pieces,
- no matter how many processes execute simultaneously.

Exploration (2)

Although processes share code, they usually have some local variables:

- *Create* allocates an independent set of arguments for each process, as indicated.
- Thus 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 processes execute the same code,they each has its own copy of 'ch', just an recursive invocation of a procedure have their own copy of formal parameters.

The output is a mixture of 'A's and 'B's.
This points out another difference between program and process.
A second deference between programs and processes is :

- Storage for local variables and procedure arguments is associated with the process executing the procedure, not in the code in witch they appear.
- There for each process has its own stack of:
  - local variables
  - formal parameters
  - procedure calls

1.4.6 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 witch 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 returns from that procedure, it is popped off the stack.
- Process exit occurs only when the process pops its last procedure (or main program) off the stack.
- System call *kill(P)* terminate process *P*.
- E.g., a process exits by calling *kill(getpid())* (a call to *getpid()* returns its process identifier, witch is passed on to *kill*.

1.4.7 Shared memory

In Xinu each process has its own copy of :

- local variables
- formal parameters
- procedure calls.

But also processes can share a set of global (external) variables, as it shown below in example `x4.c`.

In the code global variable *n*, it is a shared integer, initialized to zero.

The process executing *produce* loops 2000 times, increasing *n*; it is called the *producer*.

The process executing *consume* also loops 2000 times; it prints the value of n in decimal. We call this process the *consumer*

What happen when `ex4.c` is run under Xinu ?
Will is print all the values of n ?
Actually, it prints a low value of n, sa 0, and then 2000.

Why?

- Even though the tow processes run concurrently, they do not require the same amount of time
  - the producer is fast
  - the consumer quickly fills the available output buffers
  - the consumer has to wait for the output device to send characters to the console
  - while the consumer waits the producer runs.

A constraint is :
How can the programmer *synchronize* producer and consumer, so that the consumer receive 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);
}
```

Exploration

- In *ex4.c* global variable *n*, it is a shared integer, initialized to zero.
- The process executing *produce* loops 2000 times, increasing *n*; it is called the *producer*.
- The process executing *consume* also loops 2000 times; it prints the value of n in decimal. We call this process the *consumer*
- What happen when `ex4.c` is run under Xinu ?
- Will is print all the values of n ?
  Actually, it prints a low value of n, sa 0, and then 2000.

So: "How can the programmer synchronize the producer and consumer that the consumer receives every datum produced ?"

However this mechanism for implementing this waiting, called *synchronization*, should be designed carefully :

In a single processor system,no process should use the CPU while waiting for another.
Executing instruction while waiting for another process is called *busy waiting.*

- Xinu avoid busy waiting by supplying primitive called *semaphores*, and system calls , *wait(s)* and *signal(s)*,operate on them.
  - The system call *wait(s)* decrements a semaphore and causes the process to delay if the result is negative.
  - The system call *signal(s)* increments a semaphore, allowing a waiting process to continue.
- To synchronize producer and consumer, tow semaphores are needed:
  - one for leaving the consumer wait on a value produced
  - and 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)* witch takes the desired initial count as an argument, returning an integer by witch the semaphore is known.(see `ex5.c`)

The consumer now prints all values 0,...,1999.

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

int n = 0;  /* n assigned an intial value of zero */
/*-------------------------------------------------
 * main -- producer and consumer processes synchrinized
 * with semaphores
 *-------------------------------------------------
 */
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, waiting for it
 * to be consumed
 *-------------------------------------------------
 */
prod2(consumed, produced)
{
  int i;
  for(i=1 ; i<=2000 ; i++){
      wait(consumed);
      n++;
      signal(produced);
   }
}

/*-------------------------------------------------
 * cons2 -- print n 2000 times, waiting for it to be
 * produced
 *-------------------------------------------------
 */
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 tow processes willing to print lines. Now a printer can only receive one process at a time.

Clearly we do not want that these lines are under-leaved on the printer. That is, we want each of these processes to have exclusive access to the printer.

These tow processes should engage in mutual exclusion when they cooperate so that only one of them obtains access to a resource at a given time

This can also be realized using *semaphores*

For example, updating a list or an array witch is shared among a number of processes:

- allowing semultaniously access would result an undefined access of the operation - so any process operating on that list or an array should obtain exclusive access to it to guarantee mutual exclusion. (see `ex6.c`)

```
/* 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);

}
```

The assumption here is that a process create a semaphore *mutex* using *screate* before any process called *additem*.

1.5 A layered operating system

Xinu is designed in layers

The system functions are portioned in roughly 8 components organizing those component into a layered hierarchy (see figure 1.1).

- At the system's heart lies the scheduler and context switch:
  They switch the CPU among processes ready to run.
- In the next layer constitute the rest of the process manager
- In the process coordination layer semaphores are implemented
- Next come the procedures for the real-time clock manager
- 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.
- Above the device manager layer a layer implements machine-to-machine communication, so called inter machine communication.
- and the layer above implements the file system.
- The layer for user programs lies on the top of this layered system.

See figure 1.1

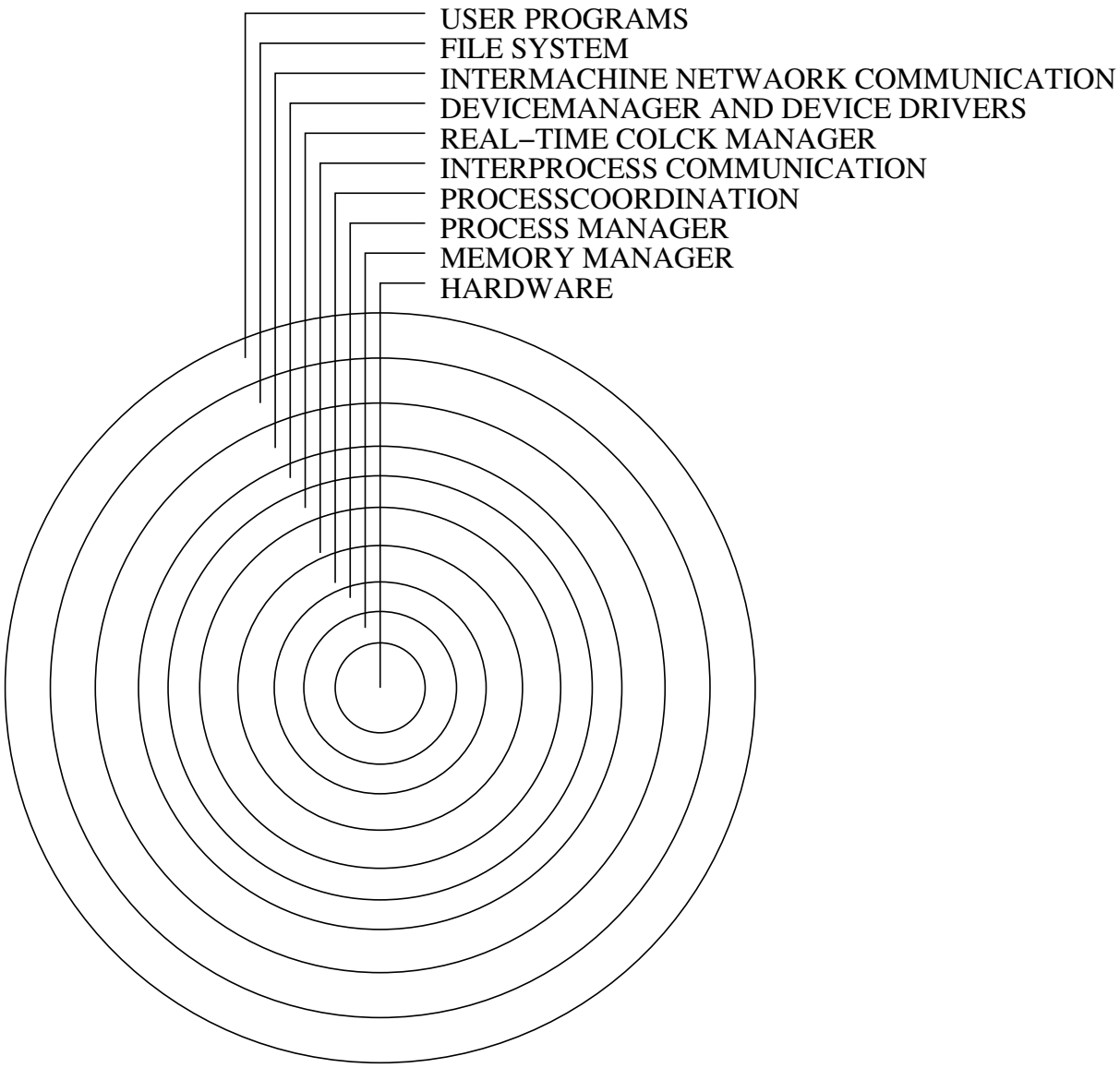

Figure 1.1: The layering of components in Xinu