COMP 242 Class Notes [1]
Handout 1: Introduction

Prasun Dewan

1. INTRODUCTION

Readings: Chapter 1, Comer; Chopter 1, Solomon.

An operating system is a set of procedures that provide facilities to:

—allocate resources to processes, and

—hide the details of the physical machine and provide a more pleasant virtual machine.

We shall call these facilities the **resource** and **beautification** facilities respectively.

1.1 Resource Facilities

An understanding of resource facilities requires definitions of both processes and resources.

1.1.1 *Processes.* While the notion of a process exists in all operating systems, different operating systems support processes in different ways. Therefore a detailed definition of processes is operating system-dependent. Here we present an intuitive operating system-independent explanation of a process. We shall see later a detailed explanation of processes supported by the Xinu Operating System.

The idea of a **process** is related to the idea of a program. A program describes data and the code to manipulate them. For instance a C program consists of declarations of variables (data) and procedures (code) that manipulate them. A process is the result of *executing* a program. For instance, when a C program is run, space is allocated in memory for the code and data described by the program and a new process is created. The process executes the main procedure of the program, which may call other procedures. The process is removed from the system when (and if) the computation started by the main procedure terminates or external factors (such as the system going down or the user typing a special character) cause termination.

A system may consist of several processes executing concurrently. These processes may be the execution instances of (i.e. result of executing) the same or different programs. For instance, in a typical interactive system, at any point there are several execution instances of editor, mailer, and command interpreter programs. Each of these processes represents a separate 'thread of control' and is associated with a separate stack. Executions of these threads in a single processor system are *interleaved*, as we shall see later.

1.1.2 *Resources.* A **resource** is a commodity needed by a process to do its work. The computer hardware provides a number of fundamental resources. A

[1] These notes are derived from several sources, which include, primarily, Doug Comer's two books on Xinu, Raphael Finkel's "An Operating Systems Vade Mecum", Prentice Hall, 1986, and Singhal and Shivratari's Advanced Operating Systems," McGraw Hall 1996. Copyright by P. Dewan. All rights reserved.

process needs *memory space* for its code and data, *processor time* to execute its instructions, and *I/O facilities* to accept data and produce results. In addition to these fundamental resources, the operating system introduces new resources. For example, *files* are provided to store permanent data. Facilities may be provided for interprocess communication. Other higher level resources may be built out of these fundamental resources.

The following analogy may explain the process-resource concept. Consider a theater that shows several one-actor plays simultaneously. The theater corresponds to the computer and the actors to the processes. The resources are the props used by the actors. As actors need props, they request them from the director (operating system). The director's job is to satisfy the following contradictory goals:

—to let each actor have whatever props are needed
—to be fair in giving props to each actor.

The director also is responsible to the owners of the theater (that is, the owners of the computer), who have invested a considerable sum in its resources. This responsibility also has several goals:

—to make sure the props (resources) are used as much as possible
—to finish as many plays (processes) as possible.

1.2 Beautification Facilities

An operating system also provides facilities to hide the details of the hardware. It constructs higher level resources out of lower level ones. For instance, the hardware provides terminal interrupts for input from and output to a terminal. An operating system provides higher level facilities to read and write characters, which relieve a user program from the task of handling interrupts from the terminal. Similarly, the hardware provides an unstructured disk. The operating system provides a higher level structure consisting of files.

1.3 What an Operating System is Not

An operating system is not:

—the hardware, which provides the fundamental resources managed by the operating system,
—a programming language, though a programming language is needed to write an operating system,
—a set of utility programs such as editors, mailers, compilers and loaders, which use the facilities provided by an operating system.

1.4 Services provided by an Operating System

The following are examples of services provided by an operating system:

—Context Switching & Scheduling, which allocate a process CPU time to execute its instructions.
—Memory Management, which deals with allocating memory to processes.
—Interprocess Communication, which deals with facilities to allow concurrently running processes to communicate with each other.

—File Systems, which provide higher level files out of low level unstructured data on a disk.

—High level I/O facilities, which free a process from the low-level details of interrupt handling.

2. CLASSIFYING OPERATING SYSTEMS

An operating system can be classified according to several criteria, some of which are given below. The criteria given in this section have to do with the functionality of the OS. Later we will look at the structural/implementation criteria for classifying operating systems.

2.1 Number and Coupling among Processors

One important criterion is the number and the coupling among the processors managed by the operating system. The simplest case is the traditional **uniprocessor** operating system, which manages a single processor. Operating systems that manage multiple, interconnected processors can be distinguished based on the "coupling" among the processors. The level of coupling depends on both the hardware and the software connecting the processors.

The tightest-coupling case is the **multiprocessor** operating system, manages several processors with separate caches but sharing a common memory. The memory is typically divided into several modules which can be accessed independently. Thus several (non-conflicting) memory requests can be serviced concurrently. A switching network is used to direct requests from processors to the correct memory module. The presence of multiple processors significantly changes the synchronization and scheduling components of the operating system.

Long-haul networks are collections of widely scattered computers connected by a common communication network. Communication in such systems is relatively slow (9.6 to 56 Kbps) and unreliable and typically through telephone lines, microwave links, and satellite channels. Long-haul networks provide several network-based services to their users such as mail, news, talk, and WWW access. The notion of a long-haul network requires the ability to communicate among remote processes. It also requires schemes for overcoming the lack of speed and reliability of the network and also the absence of a global clock.

A **local-area network** consists of independent computers confined to small geographical area. Since this area is small, the communication network can be fast (3-100 Mbps) and reliable. It offers, in addition to the services offered by long-haul networks, *sharing* of expensive devices such as printers and large secondary store. The challenge here is to unify services offered by multiple computers and address the problem of naming these services. For instance, in the case of the file service, the challenge is to provide a unified file system that spans multiple computers.

Local-area (and long-haul) networks run independent, possibly different, operating systems that offer limited services for sharing. An alternative approach is to use the same hardware configuration but present a single, **multicomputer**, operating system that manages all the computers. The main challenge here is dynamic placement and migration of processes running on different computers and implementation of distributed shared memory.

These four kinds of systems are basically four points in a continuous coupling spectrum. One can imagine a variety of intermediate points base on how much sharing is provided among the different processors.

2.2 Support for Mobility

We have assumed above that coupled computers are always connected, are placed at fixed locations, and can easily meet their power needs. These assumptions do not hold for battery-powered mobile computers (e.g. laptops) that are sporadically connected to stationary and mobile computers. To support such computers, we must revisit scheduling so that CPU energy used is reduced, and distributed synchronization since processes on different mobile computers may not be connected to each other.

2.3 Real-time Support

An operating systems can also be classified by the extent of the support for real-time services. Most operating systems support priorities, which may be used to allow applications with nearer deadlines to get precedence. **Real-time operating systems** provide more elaborate, higher-level support for meeting deadlines that automatically schedules processes based on task durations and deadlines. Things get tricky when shared objects are accessed by real-time processes, since ideally, we do not want a process with a later deadline to block a process with an earlier deadline. Solutions exist if we make certain assumptions about behaviors of real-time systems.

2.4 Database Support

Operating systems are often competitors of database systems for the management of secondary store and synchronization of processes. Traditional operating systems provide simple forms of persistent data and synchronization, leaving complex support to database systems. **Database operating systems** provide more elaborate support - in particular they support transactions.

3. STRUCTURE OF AN OPERATING SYSTEM

An operating system is composed of a **kernel**, possibly some **servers**, and posssibly some user-level libraries. The kernel provides operating system services through a set of procedures, which may be invoked by user processes through **system calls**. System calls look like procedure calls when they appear in a program, but transfer control to the kernel when they are invoked at run time. (*read* is an example of a system call in Unix.)

In some operating systems, the kernel and user-processes run in a single (physical or virtual) address space. In these systems, a system call is simply a procedure call. In more robust systems, the kernel runs in its own address space and in a special privileged mode in which it can execute instructions not available to ordinary user processes. In such systems, a system call invokes a trap as discussed below.

A **trap** is a hardware event that invokes a **trap handler** in the kernel. The trap indicates, in appropriate hardware status fields, the kind of trap. Traps may be generated *implicitly* by the hardware to signal events such as division by zero and address faults (which we will discuss later), or *explicitly* by a process when it executes a special instruction. Explicit or user-initiated traps are used to handle system calls. A system call stores the name of the call and its arguments on the stack and generates a user-initiated trap. The trap handler in the kernel knows, from the type of the trap, that it is a user-initiated trap asking for a system call, finds the name of the systems call, and calls the appropriate kernel procedure to handle the call passing it the arguments stored on the stack.

We will later consider the various techniques for structuring the kernel. As we shall see, kernels may be layered, object-oriented, or decomposed into kernel processes.

Not all operating services have to be provided by the kernel. Modern operating systems also define **servers**, which are user processes that offer operating system services to other processes. These services are invoked by clients through interprocess communication (IPC) primitives. We shall see later the rationale for transferring functionality from the kernel to servers. We shall also see the minimum functionality that needs to be supported in the kernel. In **micorkernel-based** systems, the kernel provides this minimum functionality.

The cost of invoking system calls and IPC primitives is more than the cost of invoking a simple procedure call when multiple address spaces are supported by the system. Therefore, as we shall see later, some of the traditional OS functionality is sometimes also provided by user-level libraries.

3.1 Policy and Mechanism in an Operating System

In this course, we shall distinguish between **policy** and **mechanism**. Policies are ways to choose which activities to perform. Mechanisms are the implementations that enforce policies, and often depend to some extent on the hardware on which the operating system runs. For instance, a processes may be granted resources using the first come, first serve policy. This policy may be implemented using a queue of requests. Often the kernel provides mechanisms that are used to implement policies in servers.

3.2 The XINU Approach

There are several approaches to OS design, depending on the policies and mechanisms supported by the OS. In this course we shall study in detail the policies and mechanisms of one such approach, implemented in the XINU operating system. We have chosen XINU in this course because its implementation has been carefully thought and documented in a text book. Moreover, once you master Xinu, you will have no difficulty understanding Comer's text books on networking which show how networking support can be added to Xinu. The XINU approach is fairly typical of current OS approaches. However, it is only one OS approach and provides limited functionality. Therefore, variations of and embellishments to it will be studied through reading and programming assignments and class lectures.

In Xinu, there is no difference between a system call and an ordinary procedure call. A user program is linked to the kernel and can thus invoke kernel procedures through ordinary procedure calls. Moreover, there is no notion of servers and thus all operating system services are provided by the kernel. However, we shall see later how Xinu may be extended to support servers.

An important concept in XINU is **layering**. The different components of the system are partitioned into layers. At the heart of the layered organization is the raw machine. Building out from this core, higher layers of software provide more powerful primitives, and shield the user from the machine underneath. Each layer of the system provides an abstract service, and uses the services of lower level layers. The following are the XINU layers, going from inside to outside: the hardware, memory manager, process manager, process coordination, interprocess communication, real-time clock manager, device manager and device drivers, intermachine network communication, file system, and finally the user programs that use the virtual machine underneath. In the rest of this course, we shall study these layers in detail.

Layering has the advantage that it provides a step-by-step or modular approach to operating system design and implementation. Each layer may be added incrementally, and high level layers have to be concerned only with the definition of the services provided by lower level layers, and not with their implementation.

4. MACHINE AND RUN-TIME ENVIRONMENT

Refer to chapter 2 of the Comer text for the details.

5. XINU PROCESSES AND CONCURRENCY

Consider the following program consisting of three procedures:

```
#include <conf.h>

main ()
{
    int prA(), prB();

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

prA()
{
    while (1)
        putc(CONSOLE, 'A');
}

prB()
{
    while (1)
        putc(CONSOLE, ‘B’);
}
```

The statement ‘#include ¡conf.h¿’ inserts a file of XINU declarations in the source program. These declarations include, among other things, a definition of the constant CONSOLE, and the procedures ‘resume’, ‘create’, and ‘putc’ used by the program.

Consider what happens when this program is executed. A process is created, which starts executing the main procedure. This procedure in turn executes the two resume-create statements. Each of these spawns a new process.

The statement 'create(prA....)' asks XINU to create a new process that executes the procedure 'prA' (just as the initial process executes the procedure 'main'). The other arguments to create specify such things as the stack size needed (recall that each process is associated with a separate stack), a scheduling priority (the OS uses this to allocate processor time to the process), process name ('proc1'), the count of arguments to the process (which are the arguments to the procedure 'prA' and are zero in number). Create sets up the process, leaving it ready to run, but temporarily suspended. It returns the *process id* of the new process, which is an integer that identifies the created process so that it may be referenced later. In the example, the main procedure passes this process id to 'resume', which starts (unsuspends) the process so it begins executing. The process executes the procedure 'prA', which uses the XINU facility 'putc' to write the character 'A' on the screen.

Similarly, the statement 'resume(create (prB...))' creates a process that executes the procedure 'prB', which prints 'Bs' on the screen.

It is important to distinguish between a procedure call and calls to 'create' and 'resume:

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 to proceed concurrently, as discussed in the following section.

Thus in the example, the main program terminates its computation after executing the create and resume procedure (when it reaches the end of the main procedure), while the two processes remain churning out 'As' and 'Bs' forever.

5.1 Concurrent Execution of Processes

Modern operating systems allow several processes to execute *concurrently.* It is not difficult to imagine several independent processes each being concurrently executed statement-by-statement on a different machine. But what does it mean for an operating system to provide concurrent processing on a single processor, which can execute only one statement at a time?

On a single processor, the OS, to create the illusion of concurrency, switches a single processor among several processes, allowing it to run one for only a few thousandths of a second before moving on to another. When viewed by a human, these processes appear to run concurrently.

Thus in the example, the execution of the three processes is interleaved. As a result, the output on the screen is a mixture of 'As' and 'Bs'.

5.2 Shared Memory in XINU

In XINU, each process has its own copy of local variables, formal parameters, and procedure calls, but all processes share the same set of global (external) variables. Consider a simple example of two processes that need to communicate with each other through a shared integer, 'n'.

```
#include <conf.h>
```

```
int n = 0;   /* external variable shared by all processes */

main()
{
    int produce(), consume();
    resume( create(consume, 200, 20, "cons", 0) );
    resume( create(produce, 200, 20, "prod", 0) );
}

produce()
{
    int i;

    for( i=1; i<=2000; i++ )
        n++;
};

consume()
{
    int i;
    for( i=1; i<=2000; i++ )
        printf("n is %d", n);
}
```

The process executing ‘produce’ loops 2000 times, incrementing ‘n’, we call this process the *producer*. The process executing ‘consume’ also loops 2000 times; it prints the value of ‘n’ in decimal. We call this process the *consumer*.

5.3 Synchronization and Semaphores

Assume it is required that the consumer print all the numbers produced by the producer. Does the above program achieve this goal?

The answer, surprisingly, is no. While execution of the producer and the consumer is interleaved, no guarantees are made about the relative execution speeds of the two processes, which the operating system is free to choose. Thus the programmer can make no assumptions about the way in which processor time is allocated to each process.

XINU provides **semaphores** to **synchronize** processes. A semaphore consists of an integer value, initialized when the semaphore is created. The system call *wait* decrements the semaphore and causes the suspended process to delay if the result is negative. *Signal* performs the opposite action, incrementing the semaphore and allowing a waiting process to continue. Semaphores are created dynamically with the call *screate*, which takes the initial count as an argument, and returns an integer by which the semaphore is known.

The following program illustrates the use of semaphores to achieve our goal of printing every value produced:

```
int n = 0;
```

```
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(consumed, produced)
{
    int i;

    for( i=1; i<=2000; i++ )
    {
        wait(consumed);
        n++;
        signal(produced);
    }
}

cons2(consumed, produced)
{
    int i;

    for ( i=1; i<=2000; i++) =
    {
        wait(produced);
        printf("n is %d", n);
        signal(consumed);
    }
```

In the above example, two semaphores 'consumed' and 'produced' are created by calls to *screate*., The producer waits before the consumer prints a value of 'n'. Likewise, the consumer prints a new value of 'n' only after it has been incremented by the producer. Thus the consumer prints all values of *n* from 0 through 1999.

5.4 Mutual Exclusion

We now consider **mutual exclusion**, which ensures that two concurrent processes do not access shared data at the same time.

The following example illustrates the need for mutual exclusion:

```
int a[2];
int n = 0;

main()
{
    resume( create(AddChar, 200, 20, "AddChar1", 1, 'a') );
    resume( create(AddChar, 200, 20, "AddChar2", 1, 'b') );
}

AddChar(item)
char item;
{
    a[n++] = item;
}
```

The purpose of the program is to spawn two processes, each of which adds an item to the global array *a*, which represents a set whose maximum size is 2. After all processes are completed, the result should be either:

```
n =  2;
a[1] = 'a';
a[2] = 'b';
```

or

```
n = 2;
a[1] = 'b';
a[2] = 'a';
```

The order in which the two elements are added is not important because of the set semantics.

Is one of the two results guaranteed by the above program? Certainly, if the assignment statement:

```
a[n++] = item;
```

is executed atomically, (that is executed as one unit by the processor) one of the two results is guaranteed. However, if they are not executed atomically, an invalid result may be produced, as shown below.

Assume that the statement:

```
a[n++] = item;
```

in process *AddChar1* is translated into the following three atomic statements:

```
temp1 = n;
n++;
a[temp1] = item;
```

Similarly, assume that

```
a[n++] = item
```

in process *AddChar2* is translated into:

```
temp2 = n;
n++;
a[temp2] = item
```

Now assume that the OS interleaves the execution of the two statements in the following manner:

```
/* begin executing a[n++] = item in process AddChar1 */
temp1 = n;

/* switch to a[n++] = item in process AddChar2 */
temp2 = n;
n++;
a[temp2] = item;

/* execute rest of a[n++] = item in process AddChar1 */
n++;
a[temp1] = item;
```

The result is:

```
n = 2;
a[0] = 'a';
a[1] = ?;
```

since both *temp1* and *temp2* are assigned the initial value of *n*.

This error can be corrected by using a semaphore to disallow concurrent access to the shared data. The semaphore has an initial count of 1. Before accessing shared data, each process executes *wait* on the semaphore, and calls *signal* after it has completed. Thus the statement 'a[n++] = item' is replaced by

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

where *mutex* is the semaphore used for mutual exclusion.

6. HEAVYWEIGHT VS LIGHTWEIGHT PROCESSES

As we have seen above, Xinu processes all execute in the same address space, and do not incur the overhead of a trap when making a system call. Such processes are called **lightweight processes** (**lwps**), since they have little associated state, making context switches, process creation, and interprocess communication relatively inexpensive. These processes are to contrasted with Unix-like **heavyweight processes** (**hwps**), which run in separate address spaces and trap to the kernel to make a system call. Lightweight and heavyweight processes are complementary concepts in that one can run multiple lightweight processes inside a heavyweight process. In fact, your assignments will be doing exactly this, creating Xinu lwps within a Unix hwp.