UNIT 1

What is an operating system? Hard to define precisely, because operating systems arose historically as people needed to solve problems associated with using computers.

Much of operating system history driven by relative cost factors of hardware and people. Hardware started out fantastically expensive relative to people and the relative cost has been decreasing ever since. Relative costs drive the goals of the operating system.

Goals of an Operating System

- The primary goal of an operating system is thus to make the computer system convenient to use.
- The secondary goal is to use the computer hardware in an efficient manner.

Components of a Computer System

- An operating system is an important part of almost every computer system.
- A computer system can be divided roughly into four components.
    i. Hardware
    ii. Operating system
    iii. The application programs
    iv. Users

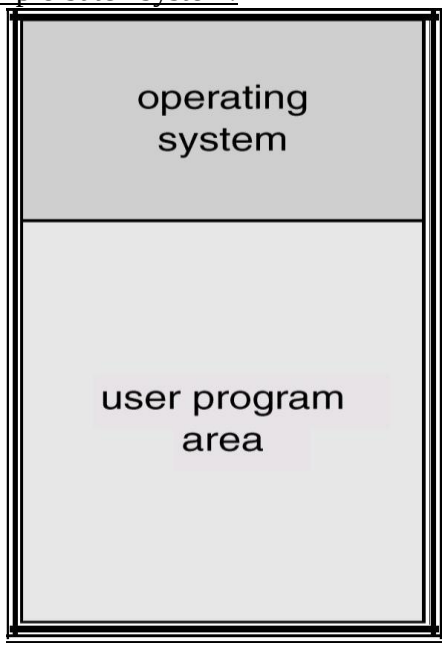
Abstract view of the components of a computer system.

- Operating system can be viewed as a resource allocator.
- The OS acts as the manager of the resources (such as CPU time, memory space, file storage space, I/O devices) and allocates them to specific programs and users as necessary for tasks.
- An operating system is a control program. It controls the execution of user programs to prevent errors and improper use of computer.

Mainframe Systems

- Early computers were physically enormous machines run from a console.
- The common input devices were card readers and tape drives.
- The common output devices were line printers, tape drives, and card punches.
- The user did not interact directly with the computer systems.
- Rather, the user prepared a job - which consisted of the program, the data, and some control information about the nature of the job (control cards)-and submitted it to the computer operator.
- The job was usually in the form of punch cards.
- The operating system in these early computers was fairly simple.
- Its major task was to transfer control automatically from one job to the next.
- The operating system was always resident in memory

- Memory layout for a simple batch system.

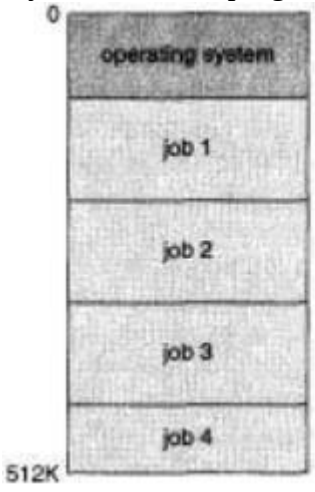


- A batch operating system, thus normally reads a stream of separate jobs.
- When the job is complete its output is usually printed on a line printer.
- The definitive feature of batch system is the lack of interaction between the user and the job while the job is executing.
- Spooling is also used for processing data at remote sites.

Multiprogrammed Systems

- A pool of jobs on disk allows the OS to select which job to run next, to increase CPU utilization.
- Multiprogramming increases CPU utilization by organizing jobs such that the CPU always has one to execute.
- The idea is as follows: The operating system keeps several jobs in memory simultaneously. This set of jobs is a subset of the jobs kept in the job pool.
  The operating system picks and begins to execute one of the jobs in the memory.

Memory layout for a multiprogramming system.



Time-Sharing Systems

- Time sharing (or multitasking) is a logical extension of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.
- A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to her use, even though it is being shared among many users.

Desktop Systems

- As hardware costs have decreased, it has once again become feasible to have a computer system dedicated to a single user. These types of computer systems are usually referred to as personal computers(PCS). They are
  microcomputers that are smaller and less expensive than mainframe computers.
- Operating systems for these computers have benefited from the development of operating systems for mainframes in several ways.

Multiprocessor Systems

- **Multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems)** have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

- Multiprocessor systems have three main advantages.
  - **Increased throughput.**
  - **Economy of scale.**
  - **Increased reliablility.**
- **If** functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether. This ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation.** Systems designed for graceful degradation are also called **fault tolerant.**

- Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.
- The most common multiple-processor systems now use **symmetric multiprocessing** (SMP), in whch each processor runs an identical copy of the operating system, and these copies communicate with one another as needed.

- Some systems use **asymmetric multiprocessing,** in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

Distributed Systems

- In contrast to the tightly coupled systems, the processors do not share memory or a clock. Instead , each processor has its own local memory.
- The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. These systems are usually referred to as loosely coupled systems, or distributed systems.

Advantages of distributed systems

- Resource Sharing
- Computation speedup
- Reliability
- Communication

Real-Time Systems

- Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems. A real-time system has well-defined, fixed time constraints.
- Real-time systems come in two flavors: hard and soft.
- A hard real-time system guarantees that critical tasks be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems.
- A less restrictive type of real-time system is a soft real-time system, where a critical real-time task gets priority over other tasks, and retains that priority until it completes.
- Soft real-time systems, however, have more limited utility than hard real-time systems. They are useful, in several areas, including multimedia, virtual reality, and advanced scientific projects.

Operating System Components

There are eight major operating system components.They are :

Process management
- Main-memory management
- File management
- I/O-system management
- Secondary-storage management
- Networking
- Protection system
- Command-interpreter system

(i) Process Management

- A **process** can be thought of as a program in execution. A batch job is a process. A time shared user program is a process.
- A process needs certain resources-including CPU time, memory, files, and I/O devices-to accomplish its task.
- A program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity, with a **program counter** specifying the next instruction to execute.
- A process is the unit of work in a system.
- The operating system is responsible for the following activities in connection with process management:
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Providing mechanisms for deadlock handling

(ii) Main – Memory Management

- Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address.
- Main memory is a repository of quickly accessible data shared by the CPU and I/O devices.
- To improve both the utilization of the CPU and the speed of the computer's response to its users, we must keep several programs in memory.
- The operating system is responsible for the following activities in connection with memory management:
  - Keeping track of which parts of memory are currently being used and by whom.
  - Deciding which processes are to be loaded into memory when memory space becomes available.
  - Allocating and deallocating memory space as needed.

(iii)File Management

- File management is one of the most visible components of an operating system.
- The operating system is responsible for the following activities in connection with file management:
  - Creating and deleting files
  - Creating and deleting directories
  - Supporting primitives for manipulating files and directories
  - Mapping files onto secondary storage
  - Backing up files on stable (nonvolatile) storage media

(iv)I/O System management

- One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. This is done using the I/O subsystem.
- The I/O subsystem consists of
  - A memory-management component that includes buffering, caching, and spooling
  - A general device-driver interface
  - Drivers for specific hardware devices

(v) Secondary storage management

- Because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost,the computer system must provide **secondary storage** to back up main memory.
- The operating system is responsible for the following activities in connection with disk management:
  - Free-space management
  - Storage allocation
  - Disk scheduling

(vi)Networking

- **A distributed system** is a collection of processors that do not share memory, peripheral devices, or a clock.
- Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines, such as high-speed buses or networks.
- The processors in the system are connected through a **communication network,** which can be configured in a number of different ways.

(vii) Protection System

- Various processes must be protected from one another's activities. For that purpose, mechanisms ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.
- Protection is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.
- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

(viii) Command-Interpreter System

- One of the most important systems programs for an operating system is the command interpreter.
- It is the interface between the user and the operating system.
- Some operating systems include the command interpreter in the kernel. Other operating systems, such as MS-DOS and UNIX, treat the command interpreter as a special program that is running when a job is initiated, or when a user first logs on (on time-sharing systems).

- Many commands are given to the operating system by control statements.
- When a new job is started in a batch system, or when a user logs on to a time-shared system, a program that reads and interprets control statements is executed automatically.
- This program is sometimes called the **control-card interpreter** or the **command-line interpreter**, and is often known as the **shell**.
-

Operating-System Services

The OS provides certain services to programs and to the users of those programs.

(ix) **Program execution:** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

(x) **I/O operations:** A running program may require I/O. This I/O may involve a file or an I/O device.

(xi) **File-system manipulation:** The program needs to read, write, create, delete files.

(xii) **Communications :** In many circumstances, one process needs to exchange information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing

on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network.

(xiii) **Error detection:** The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

(xiv) **Resource allocation:** Different types of resources are managed by the Os. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.

(xv) **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.

(xvi) **Protection:** The owners of information stored in a multiuser computer system may want to control use of that information. Security of the system is also important.

System Calls

(ξϖιι) System calls provide the interface between a process and the operating system.

(ξϖιιι) These calls are generally available as assembly-language instructions.

(ξιξ) System calls can be grouped roughly into five major categories:

- Process control

- file management
- device management
- information maintenance

5. communications.

1. Process Control

- end,abort
- load, execute
- Create process and terminate process
- get process attributes and set process attributes.
- wait for time, wait event, signal event
- Allocate and free memory.

File Management

- Create file, delete file
- Open , close
- Read, write, reposition
- Get file attributes, set file attributes.

Device Management

- Request device, release device.
- Read, write, reposition
- Get device attribtues, set device attributes
- Logically attach or detach devices

Information maintenance

- Get time or date, set time or date
- Get system data, set system data
- Get process, file, or device attributes
- Set process, file or device attributes

Communications

- Create, delete communication connection
- Send, receive messages
- Transfer status information
- Attach or detach remote devices

Two types of communication models

(a) Message passing model

- Shared memory model

System Programs

- System programs provide a convenient environment for program development and execution.

- They can be divided into several categories:

1. **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
2. **Status information:** The status such as date, time, amount of available memory or diskspace, number of users or similar status information.
3. **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
4. **Programming-language support:** Compilers, assemblers, and interpreters for common programming languages are often provided to the user with the operating system.
5. **Program loading and execution:** The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.
6. **Communications:** These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems. (email, FTP, Remote log in)
7. **Application programs:** Programs that are useful to solve common problems, or to perform common operations.
   Eg. Web browsers, database systems.

Operating System Structures

1. **Simple Structure**
2. **Layered Approach**
3. **Microkernel**

Simple Structure

Many commercial systems do not have a well-defined structure. Frequently, such operating systems started as small, simple, and limited systems, and then grew beyond their original scope.

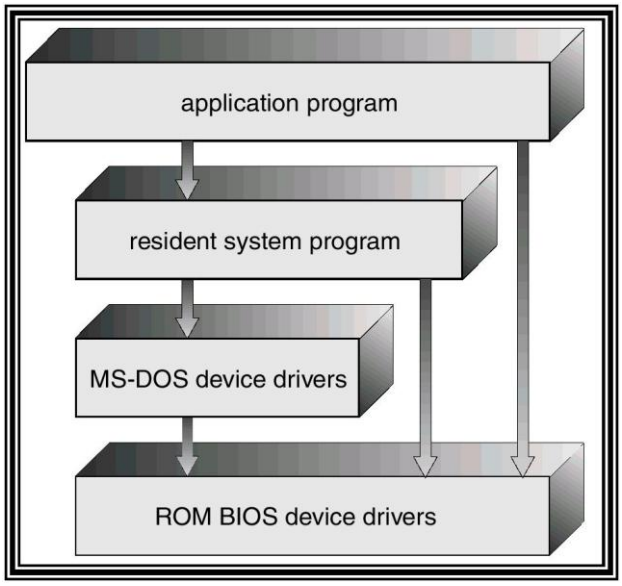
Example 1: MS-DOS.

It provides the most functionality in the least space.In MS-DOS, the interfaces and levels of functionality are not well separated.

For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives.

(-) Such freedom leaves MS-DOS vulnerable to malicious programs causing entire system crashes when user program fail.

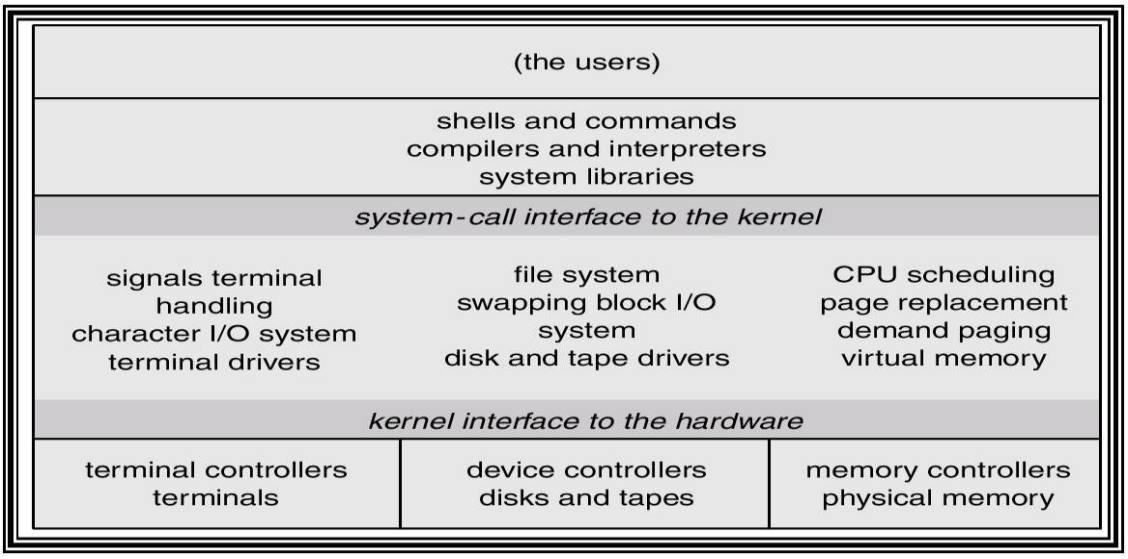
(-) MS-DOS was also limited by the hardware



Example 2: UNIX

UNIX is another system that was initially limited by hardware functionality. It consists of two separable parts:

1. **Systems programs** – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
2. **The kernel**
   - Consists of everything below the system-call interface and above the physical hardware
   - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.



Layered Approach

- Given proper hardware support, OS may be broken into smaller, more appropriate pieces.
- The modularization of a system can be done in many ways.
- One method is the layered approach, in which the operating system is broken up into a number of layers (or levels), each built on top of lower layers.
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- The main advantage of the layered approach is modularity.
- The modularity makes the debugging & verification easy.

.

UNIT 2

A process is an execution stream in the context of a particular process state.

An execution stream is a sequence of instructions.

Process state determines the effect of the instructions. It usually includes (but is not restricted to):

Registers

Stack

Memory (global variables and dynamically allocated memory)

Open file tables

Signal management information

Key concept: processes are separated: no process can directly affect the state of another process.

Process is a key OS abstraction that users see - the environment you interact with when you use a computer is built up out of processes.

The shell you type stuff into is a process.

When you execute a program you have just compiled, the OS generates a process to run the program.

Your WWW browser is a process.

Organizing system activities around processes has proved to be a useful way of separating out different activities into coherent units.

Two concepts: uniprogramming and multiprogramming.

Uniprogramming: only one process at a time. Typical example: DOS. Problem: users often wish to perform more than one activity at a time (load a remote file while editing a program, for example), and uniprogramming does not allow this. So DOS and other uniprogrammed systems put in things like memory-resident programs that invoked asynchronously, but still have separation problems. One key problem with DOS is that there is no memory protection - one program may write the memory of another program, causing weird bugs.

Multiprogramming: multiple processes at a time. Typical of Unix plus all currently envisioned new operating systems. Allows system to separate out activities cleanly.

Multiprogramming introduces the resource sharing problem - which processes get to use the physical resources of the machine when? One crucial resource: CPU. Standard solution is to use preemptive multitasking - OS runs one process for a while, then takes the CPU away from that process and lets another process run. Must save and restore process state. Key issue: fairness. Must ensure that all processes get their fair share of the CPU.

How does the OS implement the process abstraction? Uses a context switch to switch from running one process to running another process.

How does machine implement context switch? A processor has a limited amount of physical resources. For example, it has only one register set. But every process on the machine has its own set of registers. Solution: save and restore hardware state on a context switch. Save the state in Process Control Block (PCB). What is in PCB? Depends on the hardware.

Registers - almost all machines save registers in PCB.

Processor Status Word.

What about memory? Most machines allow memory from multiple processes to coexist in the physical memory of the machine. Some may require Memory Management Unit (MMU) changes on a context switch. But, some early personal computers switched all of process's memory out to disk (!!!).

Operating Systems are fundamentally event-driven systems - they wait for an event to happen, respond appropriately to the event, then wait for the next event. Examples:

User hits a key. The keystroke is echoed on the screen.

A user program issues a system call to read a file. The operating system figures out which disk blocks to bring in, and generates a request to the disk controller to read the disk blocks into memory.

The disk controller finishes reading in the disk block and generates and interrupt. The OS moves the read data into the user program and restarts the user program.

A Mosaic or Netscape user asks for a URL to be retrieved. This eventually generates requests to the OS to send request packets out over the network to a remote WWW server. The OS sends the packets.

The response packets come back from the WWW server, interrupting the processor. The OS figures out which process should get the packets, then routes the packets to that process.

Time-slice timer goes off. The OS must save the state of the current process, choose another process to run, the give the CPU to that process.

When build an event-driven system with several distinct serial activities, threads are a key structuring mechanism of the OS.

A thread is again an execution stream in the context of a thread state. Key difference between processes and threads is that multiple threads share parts of their state. Typically, allow multiple threads to read and write same memory. (Recall that no processes could directly access memory of another process). But, each thread still has its own registers. Also has its own stack, but other threads can read and write the stack memory.

What is in a thread control block? Typically just registers. Don't need to do anything to the MMU when switch threads, because all threads can access same memory.

Typically, an OS will have a separate thread for each distinct activity. In particular, the OS will have a separate thread for each process, and that thread will perform OS activities on behalf of the process. In this case we say that each user process is backed by a kernel thread.

When process issues a system call to read a file, the process's thread will take over, figure out which disk accesses to generate, and issue the low level instructions required to start the transfer. It then suspends until the disk finishes reading in the data.

When process starts up a remote TCP connection, its thread handles the low-level details of sending out network packets.

Having a separate thread for each activity allows the programmer to program the actions associated with that activity as a single serial stream of actions and events. Programmer does not have to deal with the complexity of interleaving multiple activities on the same thread.

Why allow threads to access same memory? Because inside OS, threads must coordinate their activities very closely.

If two processes issue read file system calls at close to the same time, must make sure that the OS serializes the disk requests appropriately.

When one process allocates memory, its thread must find some free memory and give it to the process. Must ensure that multiple threads allocate disjoint pieces of memory.

Having threads share the same address space makes it much easier to coordinate activities - can build data structures that represent system state and have threads read and write data structures to figure out what to do when they need to process a request.

One complication that threads must deal with: asynchrony. Asynchronous events happen arbitrarily as the thread is executing, and may interfere with the thread's activities unless the programmer does something to limit the asynchrony. Examples:

An interrupt occurs, transferring control away from one thread to an interrupt handler.

A time-slice switch occurs, transferring control from one thread to another.

Two threads running on different processors read and write the same memory.

Asynchronous events, if not properly controlled, can lead to incorrect behavior. Examples:

Two threads need to issue disk requests. First thread starts to program disk controller (assume it is memory-mapped, and must issue multiple writes to specify a disk operation). In the meantime, the second thread runs on a different processor and also issues the memory-mapped writes to program the disk controller. The disk controller gets horribly confused and reads the wrong disk block.

Two threads need to write to the display. The first thread starts to build its request, but before it finishes a time-slice switch occurs and the second thread starts its request. The combination of the two threads issues a forbidden request sequence, and smoke starts pouring out of the display.

For accounting reasons the operating system keeps track of how much time is spent in each user program. It also keeps a running sum of the total amount of time spent in all user programs. Two threads increment their local counters for their processes, then concurrently increment the global counter. Their increments interfere, and the recorded total time spent in all user processes is less than the sum of the local times.

So, programmers need to coordinate the activities of the multiple threads so that these bad things don't happen. Key mechanism: synchronization operations. These operations allow threads to control the timing of their events relative to events in other threads. Appropriate use allows programmers to avoid problems like the ones outlined above.

We first must postulate a thread creation and manipulation interface. Will use the one in Nachos:

```
class Thread {
public:
        Thread(char* debugName);
        ~Thread();
        void Fork(void (*func)(int), int arg);
        void Yield();
        void Finish();
}
```

The Thread constructor creates a new thread. It allocates a data structure with space for the TCB.

To actually start the thread running, must tell it what function to start running when it runs. The Fork method gives it the function and a parameter to the function.

What does Fork do? It first allocates a stack for the thread. It then sets up the TCB so that when the thread starts running, it will invoke the function and pass it the correct parameter. It then puts the thread on a run queue someplace. Fork then returns, and the thread that called Fork continues.

How does OS set up TCB so that the thread starts running at the function? First, it sets the stack pointer in the TCB to the stack. Then, it sets the PC in the TCB to be the first instruction in the function. Then, it sets the register in the TCB holding the first parameter to the parameter. When the thread system restores the state from the TCB, the function will magically start to run.

The system maintains a queue of runnable threads. Whenever a processor becomes idle, the thread scheduler grabs a thread off of the run queue and runs the thread.

Conceptually, threads execute concurrently. This is the best way to reason about the behavior of threads. But in practice, the OS only has a finite number of processors, and it can't run all of the runnable threads at once. So, must multiplex the runnable threads on the finite number of processors.

Let's do a few thread examples. First example: two threads that increment a variable.

```
int a = 0;
void sum(int p) {
a++;
printf("%d : a = %d\n", p, a);
}
void main() {
Thread *t = new Thread("child");
t->Fork(sum, 1);
sum(0);
}
```

The two calls to sum run concurrently. What are the possible results of the program? To understand this fully, we must break the sum subroutine up into its primitive components.

sum first reads the value of a into a register. It then increments the register, then stores the contents of the register back into a. It then reads the values of of the control string, p and a into the registers that it uses to pass arguments to the printf routine. It then calls printf, which prints out the data.

The best way to understand the instruction sequence is to look at the generated assembly language (cleaned up just a bit). You can have the compiler generate assembly code instead of object code by giving it the -S flag. It will put the generated assembly in the same file name as the .c or .cc file, but with a .s suffix.

```
la      a, %r0
ld      [%r0],%r1
add     %r1,1,%r1
st      %r1,[%r0]

ld      [%r0], %o3 ! parameters are passed starting with %o0
mov     %o0, %o1
la      .L17, %o0
call    printf
```

So when execute concurrently, the result depends on how the instructions interleave. What are possible results?

0 : 1	0 : 1
1 : 2	1 : 1
1 : 2	1 : 1
0 : 1	0 : 1
1 : 1	0 : 2
0 : 2	1 : 2
0 : 2	1 : 2
1 : 1	0 : 2

So the results are nondeterministic - you may get different results when you run the program more than once. So, it can be very difficult to reproduce bugs. Nondeterministic execution is one of the things that makes writing parallel programs much more difficult than writing serial programs.

Chances are, the programmer is not happy with all of the possible results listed above. Probably wanted the value of a to be 2 after both threads finish. To achieve this, must make the increment operation atomic. That is, must prevent the interleaving of the instructions in a way that would interfere with the additions.

Concept of atomic operation. An atomic operation is one that executes without any interference from other operations - in other words, it executes as one unit. Typically build complex atomic operations up out of sequences of primitive operations. In our case the primitive operations are the individual machine instructions.

More formally, if several atomic operations execute, the final result is guaranteed to be the same as if the operations executed in some serial order.

In our case above, build an increment operation up out of loads, stores and add machine instructions. Want the increment operation to be atomic.

Use synchronization operations to make code sequences atomic. First synchronization abstraction: semaphores. A semaphore is, conceptually, a counter that supports two atomic operations, P and V. Here is the Semaphore interface from Nachos:

```
class Semaphore {
public:
        Semaphore(char* debugName, int initialValue);
        ~Semaphore();
        void P();
        void V();
}
```

Here is what the operations do:

Semphore(name, count) : creates a semaphore and initializes the counter to count.

P() : Atomically waits until the counter is greater than 0, then decrements the counter and returns.

V() : Atomically increments the counter.

Here is how we can use the semaphore to make the sum example work:

```
int a = 0;
Semaphore *s;
void sum(int p) {
int t;
s->P();
a++;
t = a;
s->V();
printf("%d : a = %d\n", p, t);
}
void main() {
Thread *t = new Thread("child");
s = new Semaphore("s", 1);
t->Fork(sum, 1);
sum(0);
}
```

We are using semaphores here to implement a mutual exclusion mechanism. The idea behind mutual exclusion is that only one thread at a time should be allowed to do something. In this case, only one thread should access a. Use mutual exclusion to make operations atomic. The code that performs the atomic operation is called a critical section.

Semaphores do much more than mutual exclusion. They can also be used to synchronize producer/consumer programs. The idea is that the producer is generating data and the consumer is consuming data. So a Unix pipe has a producer and a consumer. You can also think of a person typing at a keyboard as a producer and the shell program reading the characters as a consumer.

Here is the synchronization problem: make sure that the consumer does not get ahead of the producer. But, we would like the producer to be able to produce without waiting for the consumer to consume. Can use semaphores to do this. Here is how it works:

```
Semaphore *s;
void consumer(int dummy) {
while (1) {
        s->P();
        consume the next unit of data
}
}
void producer(int dummy) {
while (1) {
        produce the next unit of data
        s->V();
}
}
void main() {
s = new Semaphore("s", 0);
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

In some sense the semaphore is an abstraction of the collection of data.

In the real world, pragmatics intrude. If we let the producer run forever and never run the consumer, we have to store all of the produced data somewhere. But no machine has an infinite amount of storage. So, we want to let the producer to get ahead of the consumer if it can, but only a given amount ahead. We need to implement a bounded buffer which can hold only N items. If the bounded buffer is full, the producer must wait before it can put any more data in.

```
Semaphore *full;
```

```
Semaphore *empty;
void consumer(int dummy) {
while (1) {
        full->P();
        consume the next unit of data
        empty->V();
}
}
void producer(int dummy) {
while (1) {
        empty->P();
        produce the next unit of data
        full->V();
}
}
void main() {
empty = new Semaphore("empty", N);
full = new Semaphore("full", 0);
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

An example of where you might use a producer and consumer in an operating system is the console (a device that reads and writes characters from and to the system console). You would probably use semaphores to make sure you don't try to read a character before it is typed.

Semaphores are one synchronization abstraction. There is another called locks and condition variables.

Locks are an abstraction specifically for mutual exclusion only. Here is the Nachos lock interface:

```
class Lock {
public:
        Lock(char* debugName);              // initialize lock to be FREE
        ~Lock();                            // deallocate lock
        void Acquire(); // these are the only operations on a lock
        void Release(); // they are both *atomic*
}
```

A lock can be in one of two states: locked and unlocked. Semantics of lock operations:

Lock(name) : creates a lock that starts out in the unlocked state.

Acquire() : Atomically waits until the lock state is unlocked, then sets the lock state to locked.

Release() : Atomically changes the lock state to unlocked from locked.

In assignment 1 you will implement locks in Nachos on top of semaphores.

What are requirements for a locking implementation?

Only one thread can acquire lock at a time. (safety)

If multiple threads try to acquire an unlocked lock, one of the threads will get it. (liveness)

All unlocks complete in finite time. (liveness)

What are desirable properties for a locking implementation?

Efficiency: take up as little resources as possible.

Fairness: threads acquire lock in the order they ask for it. Are also weaker forms of fairness.

Simple to use.

When use locks, typically associate a lock with pieces of data that multiple threads access. When one thread wants to access a piece of data, it first acquires the lock. It then performs the access, then unlocks the lock. So, the lock allows threads to perform complicated atomic operations on each piece of data.

Can you implement unbounded buffer only using locks? There is a problem - if the consumer wants to consume a piece of data before the producer produces the data, it must wait. But locks do not allow the consumer to wait until the producer produces the data. So, consumer must loop until the data is ready. This is bad because it wastes CPU resources.

There is another synchronization abstraction called condition variables just for this kind of situation. Here is the Nachos interface:

```
class Condition {
public:
        Condition(char* debugName);
        ~Condition();
        void Wait(Lock *conditionLock);
        void Signal(Lock *conditionLock);
        void Broadcast(Lock *conditionLock);
}
```

Semantics of condition variable operations:

Condition(name) : creates a condition variable.

Wait(Lock *l) : Atomically releases the lock and waits. When Wait returns the lock will have been reacquired.

Signal(Lock *l) : Enables one of the waiting threads to run. When Signal returns the lock is still acquired.

Broadcast(Lock *l) : Enables all of the waiting threads to run. When Broadcast returns the lock is still acquired.

All locks must be the same. In assignment 1 you will implement condition variables in Nachos on top of semaphores.

Typically, you associate a lock and a condition variable with a data structure. Before the program performs an operation on the data structure, it acquires the lock. If it has to wait before it can perform the operation, it uses the condition variable to wait for another operation to bring the data structure into a state where it can perform the operation. In some cases you need more than one condition variable.

Let's say that we want to implement an unbounded buffer using locks and condition variables. In this case we have 2 consumers.

```
Lock *l;
Condition *c;
int avail = 0;
void consumer(int dummy) {
while (1) {
        l->Acquire();
        if (avail == 0) {
        c->Wait(l);
        }
        consume the next unit of data
        avail--;
        l->Release();
}
}
void producer(int dummy) {
while (1) {
        l->Acquire();
        produce the next unit of data
        avail++;
        c->Signal(l);
        l->Release();
```

```
}
}
void main() {
l = new Lock("l");
c = new Condition("c");
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
Thread *t = new Thread("consumer");
t->Fork(consumer, 2);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

There are two variants of condition variables: Hoare condition variables and Mesa condition variables. For Hoare condition variables, when one thread performs a Signal, the very next thread to run is the waiting thread. For Mesa condition variables, there are no guarantees when the signalled thread will run. Other threads that acquire the lock can execute between the signaller and the waiter. The example above will work with Hoare condition variables but not with Mesa condition variables.

What is the problem with Mesa condition variables? Consider the following scenario: Three threads, thread 1 one producing data, threads 2 and 3 consuming data.

> Thread 2 calls consumer, and suspends.
>
> Thread 1 calls producer, and signals thread 2.
>
> Instead of thread 2 running next, thread 3 runs next, calls consumer, and consumes the element. (Note: with Hoare monitors, thread 2 would always run next, so this would not happen.)
>
> Thread 2 runs, and tries to consume an item that is not there. Depending on the data structure used to store produced items, may get some kind of illegal access error.

How can we fix this problem? Replace the if with a while.

```
void consumer(int dummy) {
while (1) {
        l->Acquire();
        while (avail == 0) {
        c->Wait(l);
        }
        consume the next unit of data
        avail--;
        l->Release();
}
```

}

> In general, this is a crucial point. Always put while's around your condition variable code. If you don't, you can get really obscure bugs that show up very infrequently.

In this example, what is the data that the lock and condition variable are associated with? The avail variable.

People have developed a programming abstraction that automatically associates locks and condition variables with data. This abstraction is called a monitor. A monitor is a data structure plus a set of operations (sort of like an abstract data type). The monitor also has a lock and, optionally, one or more condition variables. See notes for Lecture 14.

The compiler for the monitor language automatically inserts a lock operation at the beginning of each routine and an unlock operation at the end of the routine. So, programmer does not have to put in the lock operations.

Monitor languages were popular in the middle 80's - they are in some sense safer because they eliminate one possible programming error. But more recent languages have tended not to support monitors explicitly, and expose the locking operations to the programmer. So the programmer has to insert the lock and unlock operations by hand. Java takes a middle ground - it supports monitors, but also allows programmers to exert finer grain control over the locked sections by supporting synchronized blocks within methods. But synchronized blocks still present a structured model of synchronization, so it is not possible to mismatch the lock acquire and release.

Laundromat Example: A local laudromat has switched to a computerized machine allocation scheme. There are N machines, numbered 1 to N. By the front door there are P allocation stations. When you want to wash your clothes, you go to an allocation station and put in your coins. The allocation station gives you a number, and you use that machine. There are also P deallocation stations. When your clothes finish, you give the number back to one of the deallocation stations, and someone else can use the machine. Here is the alpha release of the machine allocation software:

```
allocate(int dummy) {
while (1) {
        wait for coins from user
        n = get();
        give number n to user
}
}
deallocate(int dummy) {
while (1) {
        wait for number n from user
        put(i);
}
```

```
}
main() {
for (i = 0; i < P; i++) {
        t = new Thread("allocate");
        t->Fork(allocate, 0);
        t = new Thread("deallocate");
        t->Fork(deallocate, 0);
}
}
```

The key parts of the scheduling are done in the two routines get and put, which use an array data structure a to keep track of which machines are in use and which are free.

```
int a[N];
int get() {
for (i = 0; i < N; i++) {
        if (a[i] == 0) {
        a[i] = 1;
        return(i+1);
        }
}
}
void put(int i) {
a[i-1] = 0;
}
```

It seems that the alpha software isn't doing all that well. Just looking at the software, you can see that there are several synchronization problems.

The first problem is that sometimes two people are assigned to the same machine. Why does this happen? We can fix this with a lock:

```
int a[N];
Lock *l;
int get() {
l->Acquire();
for (i = 0; i < N; i++) {
        if (a[i] == 0) {
        a[i] = 1;
        l->Release();
        return(i+1);
        }
}
l->Release();
}
void put(int i) {
```

```
l->Acquire();
a[i-1] = 0;
l->Release();
}
```

So now, have fixed the multiple assignment problem. But what happens if someone comes in to the laundry when all of the machines are already taken? What does the machine return? Must fix it so that the system waits until there is a machine free before it returns a number. The situation calls for condition variables.

```
int a[N];
Lock *l;
Condition *c;
int get() {
l->Acquire();
while (1) {
        for (i = 0; i < N; i++) {
        if (a[i] == 0) {
        a[i] = 1;
        l->Release();
        return(i+1);
        }
        }
        c->Wait(l);
}
}
void put(int i) {
l->Acquire();
a[i-1] = 0;
c->Signal();
l->Release();
}
```

What data is the lock protecting? The a array.

When would you use a broadcast operation? Whenever want to wake up all waiting threads, not just one. For an event that happens only once. For example, a bunch of threads may wait until a file is deleted. The thread that actually deleted the file could use a broadcast to wake up all of the threads.

Also use a broadcast for allocation/deallocation of variable sized units. Example: concurrent malloc/free.

```
Lock *l;
Condition *c;
char *malloc(int s) {
```

```
l->Acquire();
while (cannot allocate a chunk of size s) {
        c->Wait(l);
}
allocate chunk of size s;
l->Release();
return pointer to allocated chunk
}
void free(char *m) {
l->Acquire();
deallocate m.
c->Broadcast(l);
l->Release();
}
```

Example with malloc/free. Initially start out with 10 bytes free.

Time	Process 1	Process 2	Process 3
	malloc(10) - succeeds	malloc(5) - suspends lock	malloc(5) suspends lock
1		gets lock - waits	
2			gets lock - waits
3	free(10) - broadcast		
4		resume malloc(5) - succeeds	
5			resume malloc(5) - succeeds
6	malloc(7) - waits		
7			malloc(3) - waits
8		free(5) - broadcast	
9	resume malloc(7) - waits		
10			resume malloc(3) - succeeds

What would happen if changed c->Broadcast(l) to c->Signal(l)? At step 10, process 3 would not wake up, and it would not get the chance to allocate available memory. What would happen if changed while loop to an if?

You will be asked to implement condition variables as part of assignment 1. The following implementation is INCORRECT. Please do not turn this implementation in.

```
class Condition {
private:
        int waiting;
        Semaphore *sema;
}
void Condition::Wait(Lock* l)
{
waiting++;
l->Release();
sema->P();
l->Acquire();
}
void Condition::Signal(Lock* l)
{
if (waiting > 0) {
        sema->V();
        waiting--;
}
}
```

Why is this solution incorrect? Because in some cases the signalling thread may wake up a waiting thread that called Wait after the signalling thread called Signal.

THREADS

We first must postulate a thread creation and manipulation interface. Will use the one in Nachos:

```
class Thread {
  public:
    Thread(char* debugName);
    ~Thread();
    void Fork(void (*func)(int), int arg);
    void Yield();
    void Finish();
}
```

The Thread constructor creates a new thread. It allocates a data structure with space for the TCB.

To actually start the thread running, must tell it what function to start running when it runs. The Fork method gives it the function and a parameter to the function.

What does Fork do? It first allocates a stack for the thread. It then sets up the TCB so that when the thread starts running, it will invoke the function and pass it the correct parameter. It then puts the thread on a run queue someplace. Fork then returns, and the thread that called Fork continues.

How does OS set up TCB so that the thread starts running at the function? First, it sets the stack pointer in the TCB to the stack. Then, it sets the PC in the TCB to be the first instruction in the function. Then, it sets the register in the TCB holding the first parameter to the parameter. When the thread system restores the state from the TCB, the function will magically start to run.

The system maintains a queue of runnable threads. Whenever a processor becomes idle, the thread scheduler grabs a thread off of the run queue and runs the thread.

Conceptually, threads execute concurrently. This is the best way to reason about the behavior of threads. But in practice, the OS only has a finite number of processors, and it can't run all of the runnable threads at once. So, must multiplex the runnable threads on the finite number of processors.

Let's do a few thread examples. First example: two threads that increment a variable.

```
int a = 0;
void sum(int p) {
  a++;
  printf("%d : a = %d\n", p, a);
}
void main() {
  Thread *t = new Thread("child");
  t->Fork(sum, 1);
  sum(0);
}
```

The two calls to sum run concurrently. What are the possible results of the program? To understand this fully, we must break the sum subroutine up into its primitive components.

sum first reads the value of a into a register. It then increments the register, then stores the contents of the register back into a. It then reads the values of of the control string, p and a into the registers that it uses to pass arguments to the printf routine. It then calls printf, which prints out the data.

The best way to understand the instruction sequence is to look at the generated assembly language (cleaned up just a bit). You can have the compiler generate assembly code instead of object code by giving it the -S flag. It will put the generated assembly in the same file name as the .c or .cc file, but with a .s suffix.

```
    la      a, %r0
```

```
ld      [%r0],%r1
add     %r1,1,%r1
st      %r1,[%r0]

ld      [%r0], %o3 ! parameters are passed starting with %o0
mov     %o0, %o1
la      .L17, %o0
call    printf
```

So when execute concurrently, the result depends on how the instructions interleave. What are possible results?

0 : 1	0 : 1
1 : 2	1 : 1
1 : 2	1 : 1
0 : 1	0 : 1
1 : 1	0 : 2
0 : 2	1 : 2
0 : 2	1 : 2
1 : 1	0 : 2

So the results are nondeterministic - you may get different results when you run the program more than once. So, it can be very difficult to reproduce bugs. Nondeterministic execution is one of the things that makes writing parallel programs much more difficult than writing serial programs.

Chances are, the programmer is not happy with all of the possible results listed above. Probably wanted the value of a to be 2 after both threads finish. To achieve this, must make the increment operation atomic. That is, must prevent the interleaving of the instructions in a way that would interfere with the additions.

Concept of atomic operation. An atomic operation is one that executes without any interference from other operations - in other words, it executes as one unit. Typically build complex atomic operations up out of sequences of primitive operations. In our case the primitive operations are the individual machine instructions.

More formally, if several atomic operations execute, the final result is guaranteed to be the same as if the operations executed in some serial order.

In our case above, build an increment operation up out of loads, stores and add machine instructions. Want the increment operation to be atomic.

Use synchronization operations to make code sequences atomic. First synchronization abstraction: semaphores. A semaphore is, conceptually, a counter that supports two atomic operations, P and V. Here is the Semaphore interface from Nachos:

```
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    void P();
    void V();
}
```

Here is what the operations do:

> Semphore(name, count) : creates a semaphore and initializes the counter to count.
>
> P() : Atomically waits until the counter is greater than 0, then decrements the counter and returns.
>
> V() : Atomically increments the counter.

Here is how we can use the semaphore to make the sum example work:

```
int a = 0;
Semaphore *s;
void sum(int p) {
  int t;
  s->P();
  a++;
  t = a;
  s->V();
  printf("%d : a = %d\n", p, t);
}
void main() {
  Thread *t = new Thread("child");
  s = new Semaphore("s", 1);
  t->Fork(sum, 1);
  sum(0);
}
```

We are using semaphores here to implement a mutual exclusion mechanism. The idea behind mutual exclusion is that only one thread at a time should be allowed to do something. In this case, only one thread should access a. Use mutual exclusion to make operations atomic. The code that performs the atomic operation is called a critical section.

Semaphores do much more than mutual exclusion. They can also be used to synchronize producer/consumer programs. The idea is that the producer is generating data and the consumer is consuming data. So a Unix pipe has a producer and a consumer. You can also think of a person typing at a keyboard as a producer and the shell program reading the characters as a consumer.

Here is the synchronization problem: make sure that the consumer does not get ahead of the producer. But, we would like the producer to be able to produce without waiting for the consumer to consume. Can use semaphores to do this. Here is how it works:

```
Semaphore *s;
void consumer(int dummy) {
  while (1) {
    s->P();
    consume the next unit of data
  }
}
void producer(int dummy) {
  while (1) {
    produce the next unit of data
    s->V();
  }
}
void main() {
  s = new Semaphore("s", 0);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

In some sense the semaphore is an abstraction of the collection of data.

In the real world, pragmatics intrude. If we let the producer run forever and never run the consumer, we have to store all of the produced data somewhere. But no machine has an infinite amount of storage. So, we want to let the producer to get ahead of the consumer if it can, but only a given amount ahead. We need to implement a bounded buffer which can hold only N items. If the bounded buffer is full, the producer must wait before it can put any more data in.

```
Semaphore *full;
Semaphore *empty;
void consumer(int dummy) {
  while (1) {
    full->P();
    consume the next unit of data
```

```
    empty->V();
  }
}
void producer(int dummy) {
  while (1) {
    empty->P();
    produce the next unit of data
    full->V();
  }
}
void main() {
  empty = new Semaphore("empty", N);
  full = new Semaphore("full", 0);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

An example of where you might use a producer and consumer in an operating system is the console (a device that reads and writes characters from and to the system console). You would probably use semaphores to make sure you don't try to read a character before it is typed.

Semaphores are one synchronization abstraction. There is another called locks and condition variables.

Locks are an abstraction specifically for mutual exclusion only. Here is the Nachos lock interface:

```
class Lock {
  public:
    Lock(char* debugName);              // initialize lock to be FREE
    ~Lock();                            // deallocate lock
    void Acquire(); // these are the only operations on a lock
    void Release(); // they are both *atomic*
}
```

A lock can be in one of two states: locked and unlocked. Semantics of lock operations:

Lock(name) : creates a lock that starts out in the unlocked state.

Acquire() : Atomically waits until the lock state is unlocked, then sets the lock state to locked.

Release() : Atomically changes the lock state to unlocked from locked.

In assignment 1 you will implement locks in Nachos on top of semaphores.

What are requirements for a locking implementation?

Only one thread can acquire lock at a time. (safety)

If multiple threads try to acquire an unlocked lock, one of the threads will get it. (liveness)

All unlocks complete in finite time. (liveness)

What are desirable properties for a locking implementation?

Efficiency: take up as little resources as possible.

Fairness: threads acquire lock in the order they ask for it. Are also weaker forms of fairness.

Simple to use.

When use locks, typically associate a lock with pieces of data that multiple threads access. When one thread wants to access a piece of data, it first acquires the lock. It then performs the access, then unlocks the lock. So, the lock allows threads to perform complicated atomic operations on each piece of data.

Can you implement unbounded buffer only using locks? There is a problem - if the consumer wants to consume a piece of data before the producer produces the data, it must wait. But locks do not allow the consumer to wait until the producer produces the data. So, consumer must loop until the data is ready. This is bad because it wastes CPU resources.

There is another synchronization abstraction called condition variables just for this kind of situation. Here is the Nachos interface:

```
class Condition {
  public:
    Condition(char* debugName);
    ~Condition();
    void Wait(Lock *conditionLock);
    void Signal(Lock *conditionLock);
    void Broadcast(Lock *conditionLock);
}
```

Semantics of condition variable operations:

Condition(name) : creates a condition variable.

Wait(Lock *l) : Atomically releases the lock and waits. When Wait returns the lock will have been reacquired.

Signal(Lock *l) : Enables one of the waiting threads to run. When Signal returns the lock is still acquired.

Broadcast(Lock *l) : Enables all of the waiting threads to run. When Broadcast returns the lock is still acquired.

All locks must be the same. In assignment 1 you will implement condition variables in Nachos on top of semaphores.

Typically, you associate a lock and a condition variable with a data structure. Before the program performs an operation on the data structure, it acquires the lock. If it has to wait before it can perform the operation, it uses the condition variable to wait for another operation to bring the data structure into a state where it can perform the operation. In some cases you need more than one condition variable.

Let's say that we want to implement an unbounded buffer using locks and condition variables. In this case we have 2 consumers.

```
Lock *l;
Condition *c;
int avail = 0;
void consumer(int dummy) {
  while (1) {
    l->Acquire();
    if (avail == 0) {
      c->Wait(l);
    }
    consume the next unit of data
    avail--;
    l->Release();
  }
}
void producer(int dummy) {
  while (1) {
    l->Acquire();
    produce the next unit of data
    avail++;
    c->Signal(l);
    l->Release();
  }
}
void main() {
  l = new Lock("l");
```

```
  c = new Condition("c");
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 2);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

There are two variants of condition variables: Hoare condition variables and Mesa condition variables. For Hoare condition variables, when one thread performs a Signal, the very next thread to run is the waiting thread. For Mesa condition variables, there are no guarantees when the signalled thread will run. Other threads that acquire the lock can execute between the signaller and the waiter. The example above will work with Hoare condition variables but not with Mesa condition variables.

What is the problem with Mesa condition variables? Consider the following scenario: Three threads, thread 1 one producing data, threads 2 and 3 consuming data.

> Thread 2 calls consumer, and suspends.
>
> Thread 1 calls producer, and signals thread 2.
>
> Instead of thread 2 running next, thread 3 runs next, calls consumer, and consumes the element. (Note: with Hoare monitors, thread 2 would always run next, so this would not happen.)
>
> Thread 2 runs, and tries to consume an item that is not there. Depending on the data structure used to store produced items, may get some kind of illegal access error.

How can we fix this problem? Replace the if with a while.

```
void consumer(int dummy) {
  while (1) {
    l->Acquire();
    while (avail == 0) {
      c->Wait(l);
    }
    consume the next unit of data
    avail--;
    l->Release();
  }
}
```

> In general, this is a crucial point. Always put while's around your condition variable code. If you don't, you can get really obscure bugs that show up very infrequently.

In this example, what is the data that the lock and condition variable are associated with? The avail variable.

People have developed a programming abstraction that automatically associates locks and condition variables with data. This abstraction is called a monitor. A monitor is a data structure plus a set of operations (sort of like an abstract data type). The monitor also has a lock and, optionally, one or more condition variables. See notes for Lecture 14.

The compiler for the monitor language automatically inserts a lock operation at the beginning of each routine and an unlock operation at the end of the routine. So, programmer does not have to put in the lock operations.

Monitor languages were popular in the middle 80's - they are in some sense safer because they eliminate one possible programming error. But more recent languages have tended not to support monitors explicitly, and expose the locking operations to the programmer. So the programmer has to insert the lock and unlock operations by hand. Java takes a middle ground - it supports monitors, but also allows programmers to exert finer grain control over the locked sections by supporting synchronized blocks within methods. But synchronized blocks still present a structured model of synchronization, so it is not possible to mismatch the lock acquire and release.

Laundromat Example: A local laudromat has switched to a computerized machine allocation scheme. There are N machines, numbered 1 to N. By the front door there are P allocation stations. When you want to wash your clothes, you go to an allocation station and put in your coins. The allocation station gives you a number, and you use that machine. There are also P deallocation stations. When your clothes finish, you give the number back to one of the deallocation stations, and someone else can use the machine. Here is the alpha release of the machine allocation software:

```
allocate(int dummy) {
  while (1) {
    wait for coins from user
    n = get();
    give number n to user
  }
}
deallocate(int dummy) {
  while (1) {
    wait for number n from user
    put(i);
  }
}
main() {
  for (i = 0; i < P; i++) {
    t = new Thread("allocate");
    t->Fork(allocate, 0);
```

```
    t = new Thread("deallocate");
    t->Fork(deallocate, 0);
  }
}
```

The key parts of the scheduling are done in the two routines get and put, which use an array data structure a to keep track of which machines are in use and which are free.

```
int a[N];
int get() {
  for (i = 0; i < N; i++) {
    if (a[i] == 0) {
      a[i] = 1;
      return(i+1);
    }
  }
}
void put(int i) {
  a[i-1] = 0;
}
```

It seems that the alpha software isn't doing all that well. Just looking at the software, you can see that there are several synchronization problems.

The first problem is that sometimes two people are assigned to the same machine. Why does this happen? We can fix this with a lock:

```
int a[N];
Lock *l;
int get() {
  l->Acquire();
  for (i = 0; i < N; i++) {
    if (a[i] == 0) {
      a[i] = 1;
      l->Release();
      return(i+1);
    }
  }
  l->Release();
}
void put(int i) {
  l->Acquire();
  a[i-1] = 0;
  l->Release();
}
```

So now, have fixed the multiple assignment problem. But what happens if someone comes in to the laundry when all of the machines are already taken? What does the machine return? Must fix it so that the system waits until there is a machine free before it returns a number. The situation calls for condition variables.

```
int a[N];
Lock *l;
Condition *c;
int get() {
l->Acquire();
while (1) {
        for (i = 0; i < N; i++) {
        if (a[i] == 0) {
        a[i] = 1;
        l->Release();
        return(i+1);
        }
        }
        c->Wait(l);
}
}
void put(int i) {
l->Acquire();
a[i-1] = 0;
c->Signal();
l->Release();
}
```

What data is the lock protecting? The a array.

When would you use a broadcast operation? Whenever want to wake up all waiting threads, not just one. For an event that happens only once. For example, a bunch of threads may wait until a file is deleted. The thread that actually deleted the file could use a broadcast to wake up all of the threads.

Also use a broadcast for allocation/deallocation of variable sized units. Example: concurrent malloc/free.

```
Lock *l;
Condition *c;
char *malloc(int s) {
  l->Acquire();
  while (cannot allocate a chunk of size s) {
    c->Wait(l);
  }
  allocate chunk of size s;
```

```
  l->Release();
  return pointer to allocated chunk
}
void free(char *m) {
  l->Acquire();
  deallocate m.
  c->Broadcast(l);
  l->Release();
}
```

Example with malloc/free. Initially start out with 10 bytes free.

Time	Process 1	Process 2	Process 3
	malloc(10) - succeeds	malloc(5) - suspends lock	malloc(5) suspends lock
1		gets lock - waits	
2			gets lock - waits
3	free(10) - broadcast		
4		resume malloc(5) - succeeds	
5			resume malloc(5) - succeeds
6	malloc(7) - waits		
7			malloc(3) - waits
8		free(5) - broadcast	
9	resume malloc(7) - waits		
10			resume malloc(3) - succeeds

What would happen if changed c->Broadcast(l) to c->Signal(l)? At step 10, process 3 would not wake up, and it would not get the chance to allocate available memory. What would happen if changed while loop to an if?

You will be asked to implement condition variables as part of assignment 1. The following implementation is INCORRECT. Please do not turn this implementation in.

```
class Condition {
  private:
```

```
    int waiting;
    Semaphore *sema;
}
void Condition::Wait(Lock* l)
{
  waiting++;
  l->Release();
  sema->P();
  l->Acquire();
}
void Condition::Signal(Lock* l)
{
  if (waiting > 0) {
    sema->V();
    waiting--;
  }
}
```

Why is this solution incorrect? Because in some cases the signalling thread may wake up a waiting thread that called Wait after the signalling thread called Signal.

DEADLOCKS

You may need to write code that acquires more than one lock. This opens up the possibility of deadlock. Consider the following piece of code:

```
Lock *l1, *l2;
void p() {
  l1->Acquire();
  l2->Acquire();
  code that manipulates data that l1 and l2 protect
  l2->Release();
  l1->Release();
}
void q() {
  l2->Acquire();
  l1->Acquire();
  code that manipulates data that l1 and l2 protect
  l1->Release();
  l2->Release();
}
```

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. How long will they wait? Forever.

This case is called deadlock. What are conditions for deadlock?

> Mutual Exclusion: Only one thread can hold lock at a time.
>
> Hold and Wait: At least one thread holds a lock and is waiting for another process to release a lock.
>
> No preemption: Only the process holding the lock can release it.
>
> Circular Wait: There is a set t_1, ..., t_n such that t_1 is waiting for a lock held by t_2, ..., t_n is waiting for a lock held by t_1.

How can p and q avoid deadlock? Order the locks, and always acquire the locks in that order. Eliminates the circular wait condition.

Occasionally you may need to write code that needs to acquire locks in different orders. Here is what to do in this situation.

> First, most locking abstractions offer an operation that tries to acquire the lock but returns if it cannot. We will call this operation TryAcquire. Use this operation to try to acquire the lock that you need to acquire out of order.
>
> If the operation succeeds, fine. Once you've got the lock, there is no problem.
>
> If the operation fails, your code will need to release all locks that come after the lock you are trying to acquire. Make sure the associated data structures are in a state where you can release the locks without crashing the system.
>
> Release all of the locks that come after the lock you are trying to acquire, then reacquire all of the locks in the right order. When the code resumes, bear in mind that the data structures might have changed between the time when you released and reacquired the lock.

Here is an example.

```
int d1, d2;
// The standard acquisition order for these two locks
// is l1, l2.
Lock *l1, // protects d1
     *l2; // protects d2
// Decrements d2, and if the result is 0, increments d1
void increment() {
  l2->Acquire();
  int t = d2;
  t--;
  if (t == 0) {
```

```
    if (l1->TryAcquire()) {
      d1++;
    } else {
      // Any modifications to d2 go here - in this case none
      l2->Release();
      l1->Acquire();
      l2->Acquire();
      t = d2;
      t--;
      // some other thread may have changed d2 - must recheck it
      if (t == 0) {
        d1++;
      }
    }
    l1->Release();
  }
  d2 = t;
  l2->Release();
}
```

> This example is somewhat contrived, but you will recognize the situation when it occurs.

There is a generalization of the deadlock problem to situations in which processes need multiple resources, and the hardware may have multiple kinds of each resource - two printers, etc. Seems kind of like a batch model - processes request resources, then system schedules process to run when resources are available.

In this model, processes issue requests to OS for resources, and OS decides who gets which resource when. A lot of theory built up to handle this situation.

Process first requests a resource, the OS issues it and the process uses the resource, then the process releases the resource back to the OS.

Reason about resource allocation using resource allocation graph. Each resource is represented with a box, each process with a circle, and the individual instances of the resources with dots in the boxes. Arrows go from processes to resource boxes if the process is waiting for the resource. Arrows go from dots in resource box to processes if the process holds that instance of the resource. See Fig. 7.1.

If graph contains no cycles, is no deadlock. If has a cycle, may or may not have deadlock. See Fig. 7.2, 7.3.

System can either

> Restrict the way in which processes will request resources to prevent deadlock.

Require processes to give advance information about which resources they will require, then use algorithms that schedule the processes in a way that avoids deadlock.

Detect and eliminate deadlock when it occurs.

First consider prevention. Look at the deadlock conditions listed above.

Mutual Exclusion - To eliminate mutual exclusion, allow everybody to use the resource immediately if they want to. Unrealistic in general - do you want your printer output interleaved with someone elses?

Hold and Wait. To prevent hold and wait, ensure that when a process requests resources, does not hold any other resources. Either asks for all resources before executes, or dynamically asks for resources in chunks as needed, then releases all resources before asking for more. Two problems - processes may hold but not use resources for a long time because they will eventually hold them. Also, may have starvation. If a process asks for lots of resources, may never run because other processes always hold some subset of the resources.

Circular Wait. To prevent circular wait, order resources and require processes to request resources in that order.

Deadlock avoidance. Simplest algorithm - each process tells max number of resources it will ever need. As process runs, it requests resources but never exceeds max number of resources. System schedules processes and allocates resoures in a way that ensures that no deadlock results.

Example: system has 12 tape drives. System currently running P0 needs max 10 has 5, P1 needs max 4 has 2, P2 needs max 9 has 2.

Can system prevent deadlock even if all processes request the max? Well, right now system has 3 free tape drives. If P1 runs first and completes, it will have 5 free tape drives. P0 can run to completion with those 5 free tape drives even if it requests max. Then P2 can complete. So, this schedule will execute without deadlock.

If P2 requests two more tape drives, can system give it the drives? No, because cannot be sure it can run all jobs to completion with only 1 free drive. So, system must not give P2 2 more tape drives until P1 finishes. If P2 asks for 2 tape drives, system suspends P2 until P1 finishes.

Concept: Safe Sequence. Is an ordering of processes such that all processes can execute to completion in that order even if all request maximum resources. Concept: Safe State - a state in which there exists a safe sequence. Deadlock avoidance algorithms always ensure that system stays in a safe state.

How can you figure out if a system is in a safe state? Given the current and maximum allocation, find a safe sequence. System must maintain some information about the resources and how they are used. See OSC 7.5.3.

Avail[j] = number of resource j available
Max[i,j] = max number of resource j that process i will use
Alloc[i,j] = number of resource j that process i currently has
Need[i,j] = Max[i,j] - Alloc[i,j]

Notation: A<=B if for all processes i, A[i]<=B[i].

Safety Algorithm: will try to find a safe sequence. Simulate evolution of system over time under worst case assumptions of resource demands.

1: Work = Avail;
  Finish[i] = False for all i;
2: Find i such that Finish[i] = False and Need[i] <= Work
  If no such i exists, goto 4
3: Work = Work + Alloc[i]; Finish[i] = True; goto 2
4: If Finish[i] = True for all i, system is in a safe state

Now, can use safety algorithm to determine if we can satisfy a given resource demand. When a process demands additional resources, see if can give them to process and remain in a safe state. If not, suspend process until system can allocate resources and remain in a safe state. Need an additional data structure:

Request[i,j] = number of j resources that process i requests

Here is algorithm. Assume process i has just requested additional resources.

1: If Request[i] <= Need[i] goto 2. Otherwise, process has
  violated its maximum resource claim.
2: If Request[i] <= Avail goto 3. Otherwise, i must wait
  because resources are not available.
3: Pretend to allocate resources as follows:
  Avail = Avail - Request[i]
  Alloc[i] = Alloc[i] + Request[i]
  Need[i] = Need[i] - Request[i]
  If this is a safe state, give the process the resources. Otherwise,
  suspend the process and restore the old state.

When to check if a suspended process should be given the resources and resumed? Obvious choice - when some other process relinquishes its resources. Obvious problem - process starves because other processes with lower resource requirements are always taking freed resources.

See Example in Section 7.5.3.3.

Third alternative: deadlock detection and elimination. Just let deadlock happen. Detect when it does, and eliminate the deadlock by preempting resources.

Here is deadlock detection algorithm. Is very similar to safe state detection algorithm.

1: Work = Avail;
  Finish[i] = False for all i;
2: Find i such that Finish[i] = False and Request[i] <= Work
  If no such i exists, goto 4
3: Work = Work + Alloc[i]; Finish[i] = True; goto 2
4: If Finish[i] = False for some i, system is deadlocked.
  Moreover, Finish[i] = False implies that process i is deadlocked.

When to run deadlock detection algorithm? Obvious time: whenever a process requests more resources and suspends. If deadlock detection takes too much time, maybe run it less frequently.

OK, now you've found a deadlock. What do you do? Must free up some resources so that some processes can run. So, preempt resources - take them away from processes. Several different preemption cases:

> Can preempt some resources without killing job - for example, main memory. Can just swap out to disk and resume job later.
>
> If job provides rollback points, can roll job back to point before acquired resources. At a later time, restart job from rollback point. Default rollback point - start of job.
>
> For some resources must just kill job. All resources are then free. Can either kill processes one by one until your system is no longer deadlocked. Or, just go ahead and kill all deadlocked processes.

In a real system, typically use different deadlock strategies for different situations based on resource characteristics.

This whole topic has a sort of 60's and 70's batch mainframe feel to it. How come these topics never seem to arise in modern Unix systems?

Implementing Synchronization Operations

How do we implement synchronization operations like locks? Can build synchronization operations out of atomic reads and writes. There is a lot of literature on how to do this, one algorithm is called the bakery algorithm. But, this is slow and cumbersome to use. So, most machines have hardware support for synchronization - they provide synchronization instructions.

On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts. So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section. Guaranteed atomicity. It is also fairly efficient. Early versions of Unix did this.

Why not just use turning off interrupts? Two main disadvantages: can't use in a multiprocessor, and can't use directly from user program for synchronization.

Test-And-Set. The test and set instruction atomically checks if a memory location is zero, and if so, sets the memory location to 1. If the memory location is 1, it does nothing. It returns the old value of the memory location. You can use test and set to implement locks as follows:

> The lock state is implemented by a memory location. The location is 0 if the lock is unlocked and 1 if the lock is locked.
>
> The lock operation is implemented as:
>
> while (test-and-set(l) == 1);
>
> The unlock operation is implemented as: *l = 0;
>
> The problem with this implementation is busy-waiting. What if one thread already has the lock, and another thread wants to acquire the lock? The acquiring thread will spin until the thread that already has the lock unlocks it.

What if the threads are running on a uniprocessor? How long will the acquiring thread spin? Until it expires its quantum and thread that will unlock the lock runs. So on a uniprocessor, if can't get the thread the first time, should just suspend. So, lock acquisition looks like this:

```
while (test-and-set(l) == 1) {
  currentThread->Yield();
}
```

> Can make it even better by having a queue lock that queues up the waiting threads and gives the lock to the first thread in the queue. So, threads never try to acquire lock more than once.

On a multiprocessor, it is less clear. Process that will unlock the lock may be running on another processor. Maybe should spin just a little while, in hopes that other process will release lock. To evaluate spinning and suspending strategies, need to come up with a cost for each suspension algorithm. The cost is the amount of CPU time the algorithm uses to acquire a lock.

There are three components of the cost: spinning, suspending and resuming. What is the cost of spinning? Waste the CPU for the spin time. What is cost of suspending and resuming?

Amount of CPU time it takes to suspend the thread and restart it when the thread acquires the lock.

Each lock acquisition algorithm spins for a while, then suspends if it didn't get the lock. The optimal algorithm is as follows:

> If the lock will be free in less than the suspend and resume time, spin until acquire the lock.
>
> If the lock will be free in more than the suspend and resume time, suspend immediately.
>
> Obviously, cannot implement this algorithm - it requires knowledge of the future, which we do not in general have.

How do we evaluate practical algorithms - algorithms that spin for a while, then suspend. Well, we compare them with the optimal algorithm in the worst case for the practical algorithm. What is the worst case for any practical algorithm relative to the optimal algorithm? When the lock become free just after the practical algorithm stops spinning.

What is worst-case cost of algorithm that spins for the suspend and resume time, then suspends? (Will call this the SR algorithm). Two times the suspend and resume time. The worst case is when the lock is unlocked just after the thread starts the suspend. The optimal algorithm just spins until the lock is unlocked, taking the suspend and resume time to acquire the lock. The SR algorithm costs twice the suspend and resume time -it first spins for the suspend and resume time, then suspends, then gets the lock, then resumes.

What about other algorithms that spin for a different fixed amount of time then block? Are all worse than the SR algorithm.

> If spin for less than suspend and resume time then suspend (call this the LT-SR algorithm), worst case is when lock becomes free just after start the suspend. In this case the the algorithm will cost spinning time plus suspend and resume time. The SR algorithm will just cost the spinning time.
>
> If spin for greater than suspend and resume time then suspend (call this the GR-SR algorithm), worst case is again when lock becomes free just after start the suspend. In this case the SR algorithm will also suspend and resume, but it will spin for less time than the GT-SR algorithm
>
> Of course, in practice locks may not exhibit worst case behavior, so best algorithm depends on locking and unlocking patterns actually observed.

Here is the SR algorithm. Again, can be improved with use of queueing locks.

notDone = test-and-set(l);

```
if (!notDone) return;
start = readClock();
while (notDone) {
  stop = readClock();
  if (stop - start >= suspendAndResumeTime) {
    currentThread->Yield();
    start = readClock();
  }
  notDone = test-and-set(l);
}
```

There is an orthogonal issue. test-and-set instruction typically consumes bus resources every time. But a load instruction caches the data. Subsequent loads come out of cache and never hit the bus. So, can do something like this for inital algorithm:

```
while (1) {
  if !test-and-set(l) break;
  while (*l == 1);
}
```

Are other instructions that can be used to implement spin locks - swap instruction, for example.

On modern RISC machines, test-and-set and swap may cause implementation headaches. Would rather do something that fits into load/store nature of architecture. So, have a non-blocking abstraction: Load Linked(LL)/Store Conditional(SC).

Semantics of LL: Load memory location into register and mark it as loaded by this processor. A memory location can be marked as loaded by more than one processor.

Semantics of SC: if the memory location is marked as loaded by this processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.

Here is how to use LL/SC to implement the lock operation:

```
  while (1) {
    LL  r1, lock
    if (r1 == 0) {
      LI r2, 1
      if (SC r2, lock) break;
    }
  }
```

Unlock operation is the same as before.

Can also use LL/SC to implement some operations (like increment) directly. People have built up a whole bunch of theory dealing with the difference in power between stuff like LL/SC and test-and-set.

```
while (1) {
  LL   r1, lock
  ADDI r1, 1, r1
  if (SC r2, lock) break;
}
```

Note that the increment operation is non-blocking. If two threads start to perform the increment at the same time, neither will block - both will complete the add and only one will successfully perform the SC. The other will retry. So, it eliminates problems with locking like: one thread acquires locks and dies, or one thread acquires locks and is suspended for a long time, preventing other threads that need to acquire the lock from proceeding.

CPU Scheduling

What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.

By the way, the world went through a long period (late 80's, early 90's) in which the most popular operating systems (DOS, Mac) had NO sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directs them to run another process. Why was this true? More recent systems (Windows NT) are back to having sophisticated CPU scheduling algorithms. What drove the change, and what will happen in the future?

Basic assumptions behind most scheduling algorithms:

There is a pool of runnable processes contending for the CPU.

The processes are independent and compete for resources.

The job of the scheduler is to distribute the scarce resource of the CPU to the different processes ``fairly'' (according to some definition of fairness) and in a way that optimizes some performance criteria.

In general, these assumptions are starting to break down. First of all, CPUs are not really that scarce - almost everybody has several, and pretty soon people will be able to afford lots. Second, many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.

How do processes behave? First, CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.

> IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
>
> CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.
>
> One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why? The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.

When look at CPU burst times across the whole system, have the exponential or hyperexponential distribution in Fig. 5.2.

What are possible process states?

> Running - process is running on CPU.
>
> Ready - ready to run, but not actually running on the CPU.
>
> Waiting - waiting for some event like IO to happen.

When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:

> When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
>
> When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
>
> When process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
>
> When a process terminates.

How to evaluate scheduling algorithm? There are many possible criteria:

> CPU Utilization: Keep CPU utilization as high as possible. (What is utilization, by the way?).

Throughput: number of processes completed per unit time.

Turnaround Time: mean time from submission to completion of process.

Waiting Time: Amount of time spent ready to run but not running.

Response Time: Time between submission of requests and first response to the request.

Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.

Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.

Difference between long and short term scheduling. Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.

Let's start looking at several vanilla scheduling algorithms.

First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.

Consider performance of FCFS algorithm for three compute-bound processes. What if have 4 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is

Waiting Time? (24 + 27) / 3 = 17

Turnaround Time? (24 + 27 + 30) = 27.

Throughput? 30 / 3 = 10.

What about if processes come in order P2, P3, P1? What is

Waiting Time? (3 + 3) / 2 = 6

Turnaround Time? (3 + 6 + 30) = 13.

Throughput? 30 / 3 = 10.

Shortest-Job-First (SJF) can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?

For long term scheduler running on a batch system, user will give an estimate. Usually pretty good - if it is too short, system will cancel job before it finishes. If too long, system will hold off on running the process. So, users give pretty good estimates of overall running time.

For short-term scheduler, must use the past to predict the future. Standard way: use a time-decayed exponentially weighted average of previous CPU bursts for each process. Let T_n be the measured burst time of the nth burst, s_n be the predicted size of next CPU burst. Then, choose a weighting factor w, where $0 <= w <= 1$ and compute $s_{n+1} = w\ T_n + (1 - w)s_n$. s_0 is defined as some default constant or system average.

w tells how to weight the past relative to future. If choose $w = .5$, last observation has as much weight as entire rest of the history. If choose $w = 1$, only last observation has any weight. Do a quick example.

Preemptive vs. Non-preemptive SJF scheduler. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.

Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?

Priority Scheduling. Each process is given a priority, then CPU executes process with highest priority. If multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.

Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?

Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.

What about interactive systems? Cannot just let any process run on the CPU until it gives it up - must give response to users in a reasonable time. So, use an algorithm called round-robin scheduling. Similar to FCFS but with preemption. Have a time quantum or time slice. Let the

first process in the queue run until it expires its quantum (i.e. runs for as long as the time quantum), then run the next process in the queue.

Implementing round-robin requires timer interrupts. When schedule a process, set the timer to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do a context switch. Save the state of the running process and run the next process.

How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?

What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.

What about having a really small quantum supported in hardware? Then, you have something called multithreading. Give the CPU a bunch of registers and heavily pipeline the execution. Feed the processes into the pipe one by one. Treat memory access like IO - suspend the thread until the data comes back from the memory. In the meantime, execute other threads. Use computation to hide the latency of accessing memory.

What about a really big quantum? It turns into FCFS. Rule of thumb - want 80 percent of CPU bursts to be shorter than time quantum.

Multilevel Queue Scheduling - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.

Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.

A simple example of a multilevel feedback queue scheduling algorithm. Have 3 queues, numbered 0, 1, 2 with corresponding priority. So, for example, execute a task in queue 2 only when queues 0 and 1 are empty.

A process goes into queue 0 when it becomes ready. When run a process from queue 0, give it a quantum of 8 ms. If it expires its quantum, move to queue 1. When execute a process from queue 1, give it a quantum of 16. If it expires its quantum, move to queue 2. In queue 2, run a RR scheduler with a large quantum if in an interactive system or an FCFS scheduler if in a batch system. Of course, preempt queue 2 processes when a new process becomes ready.

Another example of a multilevel feedback queue scheduling algorithm: the Unix scheduler. We will go over a simplified version that does not include kernel priorities. The point of the algorithm is to fairly allocate the CPU between processes, with processes that have not recently used a lot of CPU resources given priority over processes that have.

Processes are given a base priority of 60, with lower numbers representing higher priorities. The system clock generates an interrupt between 50 and 100 times a second, so we will assume a value of 60 clock interrupts per second. The clock interrupt handler increments a CPU usage field in the PCB of the interrupted process every time it runs.

The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.

CPU usage field = CPU usage field / 2

Priority = CPU usage field / 2 + base priority

So, when a process does not use much CPU recently, its priority rises. The priorities of IO bound processes and interactive processes therefore tend to be high and the priorities of CPU bound processes tend to be low (which is what you want).

Unix also allows users to provide a ``nice" value for each process. Nice values modify the priority calculation as follows:

Priority = CPU usage field / 2 + base priority + nice value

So, you can reduce the priority of your process to be ``nice" to other processes (which may include your own).

In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.

Anomalies and system effects associated with schedulers.

Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.

How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.

Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.

If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why not? Because threads don't hold resources and prevent other threads from accessing them.

Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.

In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system.

UNIT 3

Introduction to Memory Management

Point of memory management algorithms - support sharing of main memory. We will focus on having multiple processes sharing the same physical memory. Key issues:

> Protection. Must allow one process to protect its memory from access by other processes.
>
> Naming. How do processes identify shared pieces of memory.
>
> Transparency. How transparent is sharing. Does user program have to manage anything explicitly?
>
> Efficiency. Any memory management strategy should not impose too much of a performance burden.

Why share memory between processes? Because want to multiprogram the processor. To time share system, to overlap computation and I/O. So, must provide for multiple processes to be resident in physical memory at the same time. Processes must share the physical memory.

Historical Development.

> For first computers, loaded one program onto machine and it executed to completion. No sharing required. OS was just a subroutine library, and there was no protection. What addresses does program generate?
>
> Desire to increase processor utilization in the face of long I/O delays drove the adoptation of multiprogramming. So, one process runs until it does I/O, then OS lets another process run. How do processes share memory? Alternatives:
>
> > Load both processes into memory, then switch between them under OS control. Must relocate program when load it. Big Problem: Protection. A bug in one process can kill the other process. MS-DOS, MS-Windows use this strategy.
> >
> > Copy entire memory of process to disk when it does I/O, then copy back when it restarts. No need to relocate when load. Obvious performance problems. Early version of Unix did this.
> >
> > Do access checking on each memory reference. Give each program a piece of memory that it can access, and on every memory reference check that it stays within its address space. Typical mechanism: base and bounds registers. Where is check done? Answer: in hardware for speed. When OS runs process, loads the base and bounds registers for that process. Cray-1 did this. Note: there is now a translation process. Program generates virtual addresses that get translated into physical addresses. But, no longer have a protection problem: one process cannot access another's memory, because it is outside its address space. If it tries to access it, the hardware will generate an exception.

End up with a model where physical memory of machine is dynamically allocated to processes as they enter and exit the system. Variety of allocation strategies: best fit, first fit, etc. All suffer from external fragmentation. In worst case, may have enough memory free to load a process, but can't use it because it is fragmented into little pieces.

What if cannot find a space big enough to run a process? Either because of fragmentation or because physical memory is too small to hold all address spaces. Can compact and relocate processes (easy with base and bounds hardware, not so easy for direct physical address machines). Or, can swap a process out to disk then restore when space becomes available. In both cases incur copying overhead. When move process within memory, must copy between memory locations. When move to disk, must copy back and forth to disk.

One way to avoid external fragmentation: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.

So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:

Extracts page number.

Extracts offset.

Translate page number to physical page frame id.

Accesses data at offset in physical page frame.

How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?

Extract page number.

Extract offset.

Check that page number is within address space of process.

Look up page number in page table.

Add offset to resulting physical page number

Access memory location.

With paging, still have protection. One process cannot access a piece of physical memory unless its page table points to that physical page. So, if the page tables of two processes point to different physical pages, the processes cannot access each other's physical memory.

Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.

But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. This kind of memory loss is called internal fragmentation.

Introduction to Paging

Basic idea: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.

So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:

Extracts page number.

Extracts offset.

Translate page number to physical page frame id.

Accesses data at offset in physical page frame.

How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?

Extract page number.

Extract offset.

Check that page number is within address space of process.

Look up page number in page table.

Add offset to resulting physical page number

Access memory location.

Problem: for each memory access that processor generates, must now generate two physical memory accesses.

Speed up the lookup problem with a cache. Store most recent page lookup values in TLB. TLB design options: fully associative, direct mapped, set associative, etc. Can make direct mapped larger for a given amount of circuit space.

How does lookup work now?

Extract page number.

Extract offset.

Look up page number in TLB.

If there, add offset to physical page number and access memory location.

Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

Like any cache, TLB can work well, or it can work poorly. What is a good and bad case for a direct mapped TLB? What about fully associative TLBs, or set associative TLB?

Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.

But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. Book calls this internal fragmentation.

How do processes share memory? The OS makes their page tables point to the same physical page frames. Useful for fast interprocess communication mechanisms. This is very nice because it allows transparent sharing at speed.

What about protection? There are a variety of protections:

Preventing one process from reading or writing another process' memory.

Preventing one process from reading another process' memory.

Preventing a process from reading or writing some of its own memory.

Preventing a process from reading some of its own memory.

How is this protection integrated into the above scheme?

Preventing a process from reading or writing memory: OS refuses to establish a mapping from virtual address space to physical page frame containing the protected memory. When program attempts to access this memory, OS will typically generate a fault. If user process catches the fault, can take action to fix things up.

Preventing a process from writing memory, but allowing a process to read memory. OS sets a write protect bit in the TLB entry. If process attempts to write the memory, OS generates a fault. But, reads go through just fine.

Virtual Memory Introduction.

When a segmented system needed more memory, it swapped segments out to disk and then swapped them back in again when necessary. Page based systems can do something similar on a page basis.

Basic idea: when OS needs to a physical page frame to store a page, and there are none free, it can select one page and store it out to disk. It can then use the newly free page frame for the new page. Some pragmatic considerations:

> In practice, it makes sense to keep a few free page frames. When number of free pages drops below this threshold, choose a page and store it out. This way, can overlap I/O required to store out a page with computation that uses the newly allocated page frame.
>
> In practice the page frame size usually equals the disk block size. Why?
>
> Do you need to allocate disk space for a virtual page before you swap it out? (Not if always keep one page frame free) Why did BSD do this? At some point OS must refuse to allocate a process more memory because has no swap space. When can this happen? (malloc, stack extension, new process creation).

When process tries to access paged out memory, OS must run off to the disk, find a free page frame, then read page back off of disk into the page frame and restart process.

What is advantage of virtual memory/paging?

> Can run programs whose virtual address space is larger than physical memory. In effect, one process shares physical memory with itself.
>
> Can also flexibly share machine between processes whose total address space sizes exceed the physical memory size.
>
> Supports a wide range of user-level stuff - See Li and Appel paper.

Disadvantages of VM/paging: extra resource consumption.

> Memory overhead for storing page tables. In extreme cases, page table may take up a significant portion of virtual memory. One Solution: page the page table. Others: go to a more complicated data structure for storing virtual to physical translations.
>
> Translation overhead.

File System Implementation

Discuss several file system implementation strategies.

First implementation strategy: contiguous allocation. Just lay out the file in contiguous disk blocks. Used in VM/CMS - an old IBM interactive system. Advantages:

Quick and easy calculation of block holding data - just offset from start of file!

For sequential access, almost no seeks required.

Even direct access is fast - just seek and read. Only one disk access.

Disadvantages:

Where is best place to put a new file?

Problems when file gets bigger - may have to move whole file!!

External Fragmentation.

Compaction may be required, and it can be very expensive.

Next strategy: linked allocation. All files stored in fixed size blocks. Link together adjacent blocks like a linked list. Advantages:

No more variable-sized file allocation problems. Everything takes place in fixed-size chunks, which makes memory allocation a lot easier.

No more external fragmentation.

No need to compact or relocate files.

Disadvantages:

Potentially terrible performance for direct access files - have to follow pointers from one disk block to the next!

Even sequential access is less efficient than for contiguous files because may generate long seeks between blocks.

Reliability -if lose one pointer, have big problems.

FAT allocation. Instead of storing next file pointer in each block, have a table of next pointers indexed by disk block. Still have to linearly traverse next pointers, but at least don't have to go to disk for each of them. Can just cache the FAT table and do traverse all in memory. MS/DOS and OS/2 use this scheme.

Table pointer of last block in file has EOF pointer value. Free blocks have table pointer of 0. Allocation of free blocks with FAT scheme is straightforward. Just search for first block with 0 table pointer.

Indexed Schemes. Give each file an index table. Each entry of the index points to the disk blocks containing the actual file data. Supports fast direct file access, and not bad for sequential access.

Question: how to allocate index table? Must be stored on disk like everything else in the file system. Have basically same alternatives as for file itself! Contiguous, linked, and multilevel index. In practice some combination scheme is usually used. This whole discussion is reminiscent of paging discussions.

Will now discuss how traditional Unix lays out file system.

First 8KB - label + boot block. Next 8KB - Superblock plus free inode and disk block cache.

Next 64KB - inodes. Each inode corresponds to one file.

Until end of file system - disk blocks. Each disk block consists of a number of consecutive sectors.

What is in an inode - information about a file. Each inode corresponds to one file. Important fields:

> Mode. This includes protection information and the file type. File type can be normal file (-), directory (d), symbolic link (l).
>
> Owner
>
> Number of links - number of directory entries that point to this inode.
>
> Length - how many bytes long the file is.
>
> Nblocks - number of disk blocks the file occupies.
>
> Array of 10 direct block pointers. These are first 10 blocks of file.
>
> One indirect block pointer. Points to a block full of pointers to disk blocks.
>
> One doubly indirect block pointer. Points to a block full of pointers to blocks full of pointers to disk blocks.
>
> One triply indirect block pointer. (Not currently used).
>
> So, a file consists of an inode and the disk blocks that it points to.

Nblocks and Length do not contain redundant information - can have holes in files. A hole shows up as block pointers that point to block 0 - i.e., nothing in that block.

Assume block size is 512 bytes (i.e. one sector). To access any of first 512*10 bytes of file, can just go straight from inode. To access data farther in, must go indirect through at least one level of indirection.

What does a directory look like? It is a file consisting of a list of (name,inode number) pairs. In early Unix Systems the name was a maximum of 14 characters long, and the inode number was 2 bytes. Later versions of Unix removed this restriction, and each directory entry was variable length and also included the length of the file name.

Why don't inodes contain names? Because would like a file to be able to have multiple names.

How does Unix implement the directories . and ..? They are just names in the directory. . points to the inode of the directory, while .. points to the inode of the directory's parent directory. So, there are some circularities in the file system structure.

User can refer to files in one of two ways: relative to current directory, or relative to the root directory. Where does lookup for root start? By convention, inode number 2 is the inode for the top directory. If a name starts with /, lookup starts at the file for inode number 2.

How does system convert a name to an inode? There is a routine called namei that does it.

Do a simple file system example, draw out inodes and disk blocks, etc. Include counts, length, etc.

What about symbolic links? A symbolic link is a file containing a file name. Whenever a Unix operation has the name of the symbolic link as a component of a file name, it macro substitutes the name in the file in for the component.

What disk accesses take place when list a directory, cd to a directory, cat a file? Is there any difference between ls and ls -F?

What about when use the Unix rm command? Does it always delete the file? NO - it decrements the reference count. If the count is 0, then it frees up the space. Does this algorithm work for directories? NO - directory has a reference to itself (.). Use a different command.

When write a file, may need to allocate more inodes and disk blocks. The superblock keeps track of data that help this process along. A superblock contains:

the size of the file system

number of free blocks in the file system

list of free blocks available in the file system

index of next free block in free block list

the size of the inode list

the number of free inodes in the file system

a cache of free inodes

the index of the next free inode in inode cache

The kernel maintains the superblock in memory, and periodically writes it back to disk. The superblock also contains crucial information, so it is replicated on disk in case part of disk fails.

When OS wants to allocate an inode, it first looks in the inode cache. The inode cache is a stack of free inodes, the index points to the top of the stack. When the OS allocates an inode, it just decrements index. If the inode cache is empty, it linearly searches inode list on disk to find free inodes. An inode is free iff its type field is 0. So, when go to search inode list for free inodes, keep looking until wrap or fill inode cache in superblock. Keep track of where stopped looking - will start looking there next time.

To free an inode, put it in superblock's inode cache if there is room. If not, don't do anything much. Only check against the number where OS stopped looking for inodes the last time it filled the cache. Make this number the minimum of the freed inode number and the number already there.

OS stores list of free disk blocks as follows. The list consists of a sequence of disk blocks. Each disk block in this sequence stores a sequence of free disk block numbers. The first number in each disk block is the number of the next disk block in this sequence. The rest of the numbers are the numbers of free disk blocks. (Do a picture) The superblock has the first disk block in this sequence.

To allocate a disk block, check the superblock's block of free disk blocks. If there are at least two numbers, grab the one at the top and decrement the index of next free block. If there is only one number left, it contains the index of the next block in the disk block sequence. Copy this disk block into the superblock's free disk block list, and use it as the free disk block.

To free a disk block do the reverse. If there is room in the superblock's disk block, push it on there. If not, write superblock's disk block into free block, then put index of newly free disk block in as first number in superblock's disk block.

Note that OS maintains a list of free disk blocks, but only a cache of free inodes. Why is this?

Kernel can determine whether inode is free or not just by looking at it. But, cannot with disk block - any bit pattern is OK for disk blocks.

Easy to store lots of free disk block numbers in one disk block. But, inodes aren't large enough to store lots of inode numbers.

Users consume disk blocks faster than inodes. So, pauses to search for inodes aren't as bad as searching for disk blocks would be.

Inodes are small enough to read in lots in a single disk operation. So, scanning lists of inodes is not so bad.

Synchronizing multiple file accesses. What should correct semantics be for concurrent reads and writes to the same file? Reads and writes should be atomic:

If a read execute concurrently, read should either observe the entire write or none of the write.

Reads can execute concurrently with no atomicity constraints.

How to implement these atomicity constraints? Implement reader-writer locks for each open file. Here are some operations:

Acquire read lock: blocks until no other process has a write lock, then increments read lock count and returns.

Release read lock: decrements read lock count.

Acquire write lock: blocks until no other process has a write or read lock, then sets the write lock flag and returns.

Release write lock: clears write lock flag.

Obtain read or write locks inside the kernel's system call handler. On a Read system call, obtain read lock, perform all file operations required to read in the appropriate part of file, then release read lock and return. On Write system call, do something similar except get write locks.

What about Create, Open, Close and Delete calls? If multiple processes have file open, and a process calls Delete on that file, all processes must close the file before it is actually deleted. Yet another form of synchronization is required.

How to organize synchronization? Have a global file table in addition to local file tables. What does each file table do?

Global File Table: Indexed by some global file id - for example, the inode index would work. Each entry has a reader/writer lock, a count of number of processes that have file open and a bit that says whether or not to delete the file when last process that has file open closes it. May have other data depending on what other functionality file system supports.

Local File Table: Indexed by open file id for that process. Has a pointer to the current position in the open file to start reading from or writing to for Write and Read operations.

For your nachos assignments, do not have to implement reader/writer locks - can just use a simple mutual exclusion lock.

What are sources of inefficiency in this file system? Are two kinds - wasted time and wasted space.

Wasted time comes from waiting to access the disk. Basic problem with system described above: it scatters related items all around the disk.

Inodes separated from files.

Inodes in same directory may be scattered around in inode space.

Disk blocks that store one file are scattered around the disk.

So, system may spend all of its time moving the disk heads and waiting for the disk to revolve.

The initial layout attempts to minimize these phenonmena by setting up free lists so that they allocate consecutive disk blocks for new files. So, files tend to be consecutive on disk. But, as use file system, layout gets scrambled. So, the free list order becomes increasingly randomized, and the disk blocks for files get spread all over the disk.

Just how bad is it? Well, in traditional Unix, the disk block size equaled the sector size, which was 512 bytes. When they went from 3BSD to 4.0BSD they doubled the disk block size. This more than doubled the disk performance. Two factors:

Each block access fetched twice as much data, so amortized the disk seek overhead over more data.

The file blocks were bigger, so more files fit into the direct section of the inode index.

But, still pretty bad. When file system first created, got transfer rates of up to 175 KByte per second. After a few weeks, deteriorated down to 30 KByte per second. What is worse, this is only about 4 percent (!!!!) of maxmimum disk throughput. So, the obvious fix is to make the block size even bigger.

Wasted space comes from internal fragmentation. Each file with anything in it (even small ones) takes up at least one disk block. So, if file size is not an even multiple of disk block size, there will be wasted space off the end of the last disk block in the file. And, since most files are small, there may not be lots of full disk blocks in the middle of files.

Just how bad is it? It gets worse for larger block sizes. (so, maybe making block size bigger to get more of the disk transfer rate isn't such a good idea...). Did some measurements on a file system at Berkeley, to calculate size and percentage of waste based on disk block size. Here are some numbers:

Space Used (Mbytes)	Percent Waste	Organization
775.2	0.0	Data only, no separation between files
828.7	6.9	Data + inodes, 512 byte block
866.5	11.8	Data + inodes, 1024 byte block
948.5	22.4	Data + inodes, 2048 byte block
1128.3	45.6	Data + inodes, 4096 byte block

Notice that a problem is that the presence of small files kills large file performance. If only had large files, would make the block size large and amortize the seek overhead down to some very small number. But, small files take up a full disk block and large disk blocks waste space.

In 4.2BSD they attempted to fix some of these problems.

Introduced concept of a cylinder group. A cylinder group is a set of adjacent cylinders. A file system consists of a set of cylinder groups.

Each cylinder group has a redundant copy of the super block, space for inodes and a bit map describing available blocks in the cylinder group. Default policy: allocate 1 inode per 2048 bytes of space in cylinder group.

Basic idea behind cylinder groups: will put related information together in the same cylinder group and unrelated information apart in different cylinder groups. Use a bunch of heuristics.

Try to put all inodes for a given directory in the same cylinder group.

Also try to put blocks for one file adjacent in the cylinder group. The bitmap as a storage device makes it easier to find adjacent groups of blocks. For long files redirect blocks to a new cylinder group every megabyte. This spreads stuff out over the disk at a large enough granularity to amortize the seek time.

Important point to making this scheme work well - keep a free space reserve (5 to 10 percent). Once above this reserve, only supervisor can allocate disk blocks. If disk is almost

completely full, allocation scheme cannot keep related data together and allocation scheme degenerates to random.

Increased block size. The minimum block size is now 4096 bytes. Helps read bandwidth and write bandwidth for big files. But, don't waste a lot of space for small files? Solution: introduce concept of a disk block fragment.

Each disk block can be chopped up into 2, 4, or 8 fragments. Each file contains at most one fragment which holds the last part of data in the file. So, if have 8 small files they together only occupy one disk block. Can also allocate larger fragments if the end of the file is larger than one eighth of the disk block. The bit map is laid out at the granularity of fragments.

When increase the size of the file, may need to copy out the last fragment if the size gets too big. So, may copy a file multiple times as it grows. The Unix utilities try to avoid this problem by growing files a disk block at a time.

Bottom line: this helped a lot - read bandwidth up to 43 percent of peak disk transfer rate for large files.

Another standard mechanism that can really help disk performance - a disk block cache. OS maintains a cache of disk blocks in main memory. When a request comes, it can satisfy request locally if data is in cache. This is part of almost any IO system in a modern machine, and can really help performance.

How does caching algorithm work? Devote part of main memory to cached data. When read a file, put into disk block cache. Before reading a file, check to see if appropriate disk blocks are in the cache.

What about replacement policy? Have many of same options as for paging algorithms. Can use LRU, FIFO with second chance, etc.

How easy is it to implement LRU for disk blocks? Pretty easy - OS gets control every time disk block is accessed. So can implement an exact LRU algorithm easily.

How easy was it to implement an exact LRU algorithm for virtual memory pages? How easy was it to implement an approximate LRU algorithm for virtual memory pages?

Bottom line: different context makes different cache replacement policies appropriate for disk block caches.

What is bad case for all LRU algorithms? Sequential accesses. What is common case for file access? Sequential accesses. How to fix this? Use free-behind for large sequentially accessed files - as soon as finish reading one disk block and move to the next, eject first disk block from the cache.

So what cache replacement policy do you use? Best choice depends on how file is accessed. So, policy choice is difficult because may not know.

Can use read-ahead to improve file system performance. Most files accessed sequentially, so can optimistically prefetch disk blocks ahead of the one that is being read.

Prefetching is a general technique used to increase the performance of fetching data from long-latency devices. Can try to hide latency by running something else concurrently with fetch.

With disk block caching, physical memory serves as a cache for the files stored on disk. With virtual memory, physical memory serves as a cache for processes stored on disk. So, have one physical resource shared by two parts of system.

How much of each resource should file cache and virtual memory get?

> Fixed allocation. Each gets a fixed amount. Problem - not flexible enough for all situations.
>
> Adaptive - if run an application that uses lots of files, give more space to file cache. If run applications that need more memory, give more to virtual memory subsystem. Sun OS does this.

How to handle writes. Can you avoid going to disk on writes? Possible answers:

> No - user wants data on stable storage, that's why he wrote it to a file.
>
> Yes - keep in memory for a short time, and can get big performance improvements. Maybe file is deleted, so don't ever need to use disk at all. Especially useful for /tmp files. Or, can batch up lots of small writes into a larger write, or can give disk scheduler more flexibility.
>
> In general, depends on needs of the system.

One more question - do you keep data written back to disk in the file cache? Probably - may be read in the near future, so should keep it resident locally.

One common problem with file caches - if use file system as backing store, can run into double caching. Eject a page, and it gets written back to file. But, disk blocks from recently written files may be cached in memory in the file cache. In effect, file caching interferes with performance of the virtual memory system. Fix this by not caching backing store files.

An important issue for file systems is crash recovery. Must maintain enough information on disk to recover from crashes. So, modifications must be carefully sequenced to leave disk in a recoverable state at all times.

Monitors

Monitors: A high-level data abstraction tool that automatically generates atomic operations on a given data structure. A monitor has:

Shared data.

A set of atomic operations on that data.

A set of condition variables.

Monitors can be imbedded in a programming language: Mesa/Cedar from Xerox PARC.

Typical implementation: each monitor has one lock. Acquire lock when begin a monitor operation, and Release lock when operation finishes. Optimization: reader/writer locks. Statically identify operations that only read data, then allow these read-only operations to go concurrently. Writers get mutual exclusion with respect to other writers and to readers. Standard synchronization mechanism for accessing shared data.

Advantages: reduces probability of error (never forget to Acquire or Release the lock), biases programmer to think about the system in a certain way (is not ideologically neutral). Trend is away from encapsulated high-level operations such as monitors toward more general purpose but lower level synchronization operations.

Bounded buffer using monitors and signals

Shared State data[10] - a buffer holding produced data. num - tells how many produced data items there are in the buffer.

Atomic Operations Produce(v) called when producer produces data item v. Consume(v) called when consumer is ready to consume a data item. Consumed item put into v.

Condition Variables bufferAvail - signalled when a buffer becomes available. dataAvail - signalled when data becomes available.

```
monitor {
  Condition *bufferAvail, *dataAvail;
  int num = 0;
  int data[10];

  Produce(v) {
    while (num == 10) { /* Mesa semantics */
      bufferAvail->Wait();
    }
    put v into data array
```

```
  num++;
  dataAvail->Signal(); /* must always do this? */
                /* can replace with broadcast? */
 }
 Consume(v) {
  while (num == 0) { /* Mesa Semantics */
   dataAvail->Wait();
  }
  put next data array value into v
  num--;
  bufferAvail->Signal(); /* must always do this? */
                 /* can replace with broadcast? */
 }
}
```

The best way to understand monitors is that there is a syntactic transformation that inserts the lock operations.

```
 Condition *bufferAvail, *dataAvail;
 int num = 0;
 int data[10];
 Lock *monitorLock;

 Produce(v) {
  monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
  while (num == 10) { /* Mesa semantics */
   bufferAvail->Wait(monitorLock);
  }
  put v into data array
  num++;
  dataAvail->Signal(monitorLock); /* must always do this? */
                /* can replace with broadcast? */
  monitorLock->Release(); /* Release monitor lock after perform operation */
 }
 Consume(v) {
  monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
  while (num == 0) { /* Mesa Semantics */
   dataAvail->Wait(monitorLock);
  }
  put next data array value into v
  num--;
  bufferAvail->Signal(monitorLock); /* must always do this? */
                 /* can replace with broadcast? */
  monitorLock->Release(); /* Release monitor lock after perform operation */
 }
        }
```

SEGMENTATION

Programs need to share data on a controlled basis. Examples: all processes should use same compiler. Two processes may wish to share part of their data.

Program needs to treat different pieces of its memory differently. Examples: process should be able to execute its code, but not its data. Process should be able to write its data, but not its code. Process should share part of memory with other processes, but not all of memory. Some memory may need to be exported read-only, other memory exported read/write.

Mechanism to support treating different pieces of address space separately: segments. Program's memory is structured as a set of segments. Each segment is a variable-sized chunk of memory. An address is a segment,offset pair. Each segment has protection bits that specify which kind of accesses can be performed. Typically will have something like read, write and execute bits.

Where are segments stored in physical memory? One alternative: each segment is stored contiguously in physical memory. Each segment has a base and a bound. So, each process has a segment table giving the base, bounds and protection bits for each segment.

How does program generate an address containing a segment identifier? There are several ways:

Top bits of address specify segment, low bits specify offset.

Instruction implicity specifies segment. I.E. code vs. data vs. stack.

Current data segment stored in a register.

Store several segment ids in registers; instruction specifies which one.

What does address translation mechanism look like now?

Find base and bound for segment id.

Add base to offset.

Check that offset < bound.

Check that access permissions match actual access.

Reference the generated physical address.

How can this be fast enough? Several parts of strategy:

Segment table cache stored in fast memory. Typically fully associative.

Full segment table stored in physical memory. If the segment id misses in the cache, get from physical memory and reload the cache.

OS may need to reference data from any process. How is this done? One way: OS runs with address translation turned off. Reserve certain parts of physical memory to hold OS data structures (buffers, PCB's, etc.).

How do user and OS communicate? Via shared memory. But, OS must manually apply translation any time user program gives it a pointer to data. Example: Exec(file) system call in nachos.

What must OS do to manage segments?

Keep copy of segment table in PCB.

When create process, allocate space for segments, fill in base and bounds registers.

When switch contexts, switch segment information state in hardware. Examples: may invalidate segment id cache.

What about memory management? Segments come in variable sized chunks, so must allocate physical memory in variable sized chunks. Can use a variety of heuristics: first fit, best fit, etc. All suffer from fragmentation (external fragmentation).

What to do when must allocate a segment and it doesn't fit given segments that are already resident? Have several options:

Can compact segments. Copy segments to contiguous physical memory locations so that small holes are collected into one big hole. Notice that this changes the physical memory locations of segment's data. What must OS do to implement new translation?

Can push segments out to disk. But, must provide a mechanism to detect a reference to the swapped segment. When the reference happens, will then reload from disk.

What happens when must enlarge a segment? (This can happen if user needs to dynamically allocate more memory). If lucky, there is a hole above the segment and can just increment bound for that segment. If not, maybe can move to a larger hole where the new size fits. If not, may have to compact or swap out segments.

Protection: How does one process ensure that no other process can access its memory? Make sure OS never creates a segment table entry that points to same physical memory.

Sharing: How do processes share memory? Typically at segment level. Segment tables of the two processes point to the same physical memory.

What about protection for a shared segment? What if one process only wants other processes to read segment? Typically have access bits in segment table and OS can make segment read only in one process.

Naming: Processes must name segments created and manipulated by other processes. Typically have a name space for segments; processes export segments for other processes to use under given names. In Multics, had a tree structured segment name space, and segments were persistent across process invocations.

Efficiency: It is efficient. The segment table lookup typically does not impose too much time overhead, and segment tables tend to be small with not much memory overhead.

Granularity: allows processes to specify which memory they share with other processes. But if whole segment is either resident or not, limits the flexibility of OS memory allocation.

Advantages of segmentation:

Can share data in a controlled way with appropriate protection mechanisms.

Can move segments independently.

Can put segments on disk independently.

Is a nice abstraction for sharing data. In fact, abstraction is often preserved as a software concept in systems that use other hardware mechanisms to share data.

Problems with segmentation:

Fragmentation and complicated memory management.

Whole segment must be resident or not. Allocation granularity may be too large for efficient memory utilization. Example: have a big segment but only access a small part of it for a long time. Waste memory used to hold the rest.

Potentially bad address space utilization if have fixed size segment id field in addresses. If have few segments, waste bits in segment field. If have small segments, waste bits in offset field.

Must be sure to make offset field large enough. See 8086.

UNIT4
Disk Scheduling

The processes running on a machine may have multiple outstanding requests for data from the disk. In what order should requests be served?

First-Come-First-Served. This is how nachos works right now. As processes arrive, they queue up for the disk and get their requests served in order. In current version of nachos, queueing happens at the mutex lock.

What is wrong with FCFS? May have long swings from one part of disk to another. It makes sense to service outstanding requests from adjacent parts of disk sequentially.

Shortest-Seek-Time-First. Disk scheduler looks at all outstanding disk requests, and services the one closest to where the disk head currently is. Sort of like Shortest-Job-First task scheduling.

What is the problem with SSTF? Starvation. A request for a remote part of the disk may never get serviced.

SCAN algorithm. Head goes from one end of disk to another. Reverses direction when hits end of disk and goes back the other way. Eliminates starvation problem. Minor variant: C-SCAN, which goes all the way back to front of disk when it hits the end, sort of like a raster scan in a display.

LOOK algorithm. Like scan, but reverse direction when hit the last request in the current direction. C-LOOK is the circular variant of LOOK.