Chapter 2

Processes:

- Web server when a request comes in the server checks to see if the page needed is in the cache
    - if yes: it is sent back
    - if not, a disk request is started to fetch it

The Process Model:

- all the runnable software on the computer is organized into a number of sequential processes
- A process is just an instance of an executing program
    - including: PC, registers, and variables
    - conceptually each process has its own CPU, in reality the CPU switches back and forth from process to process
- The rapid switching back between processes by the CPU is called multiprogramming
- a single core CPU can run only one process at a time
    - a multicore aclu can each run one process at a time
- Process is an activity of some kind:
    - a program, input, output, and a state

Process Creation:

- Four Principle Events Causes Processes to be created:
    - 1. System initialization
    - 2. Execution of a process-creation system call by a running process
    - 3. A user request to create a new process
    - 4. Initiation of a batch job
- Daemons: processes that stay in the background to handle some activity, such as Email, Web pages, news, printing, ect.
- In nix only one system call to create a new process: fork
    - create an exact clone of the calling process, now there is a parent and a child
    - usually a child process then executes, execs, or a similar system call to change its emory image and run a new program
    - the parent and the child have their own distinct adresss spaces

Process Termination:

- Process with terminate due to:
    - 1. Notmal exit (voluntary)
    - 2. Error exit (voluntary)
    - 3. Fatal error (involuntary)

    - 4. Killed by another process (involuntary)

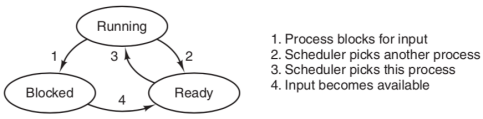
Process Hierarchies:

- in UNIX, a process and all of its children and further descendants form a process group
- processes in UNIX cannot disinherit their children

Process States:

- When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available or the system has decided to allocate the CPU to another process for a while
- Three states a process may be in:
    - 1. Running (actually using the CPU at that instant)
    - 2. Ready (Runnable; temporarily stopped to let another process run)
    - 3. Blocked (unable to run until some external event happens)
- Four Transitions are possible among these three states:

SEC. 2.1 PROCESSES



    - 1. when the OS discovers that a process cannot continue right now
    - 2. when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time
    - 3. occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again
    - 4. occurs when the external event for which a process was waiting happens, such as the arrival of some input

Implementation of Processes:

- To implement the process model, the OS maintains a table, called a process table
    - one entry per process
    - the entry contains the PC, stack pointer, memory allocation, the status of its open files, its accounting and scheduling info, and everything else about the process that must be saved between states

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry.

- all interrupts start by saving the registers, often in the process table entry for the current process
- the info that is pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Modeling Multiprogramming:

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Figure 2-6 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

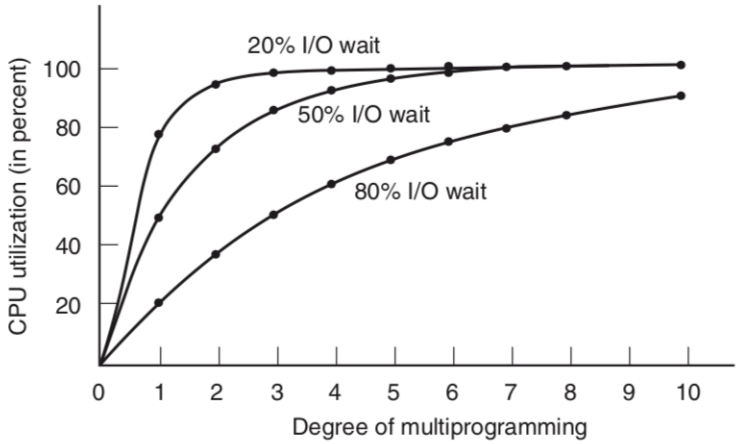


Figure 2-6. CPU utilization as a function of the number of processes in memory.

-
- the above problem assumes that all n processes are independent

Thread Usage:

- process within a process, called threads
- Threads share common memory
- finite stat machine: each computation has a saved state, and there exists some set of events that can occur to change the state.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-10. Three ways to construct a server.

The Classical Thread Model:

- The thread has a PC that keeps track of which instruction to execute next, it has registers to hold its current working variables, it has a stacks that contains the execution history
- threads are the entities scheduled for execution on the CPU
- Threads add to the process model by allowing multiple executions to take place in the same process environment
- multithreading: allowing multiple threads in the same process
- CPU switches rapidly back and forth among the threads, providing the illusion that the threads are running in parallel
- all threads have exactly the same address space, and the share the same global variables

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

POSIX Threads:

- Pthreads: possible to write portable threaded programs
    - certain properties:
        - identifier
        - set of registers ,including PC
        - set of attributes

Implementing Threads in User Space:

- two main places to implement threads:
    - user space
    - the kernel
-
- When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process
- Thread table is managed by the run-time system

- when a thread is blocked it calls run-time system procedure that:
    - checks if the thread needs to be put in blocked state
    - if yes, it stores the thread's register in the thread table
- the run-time system keeps running threads from its own process until the kernel takes the CPU away from it

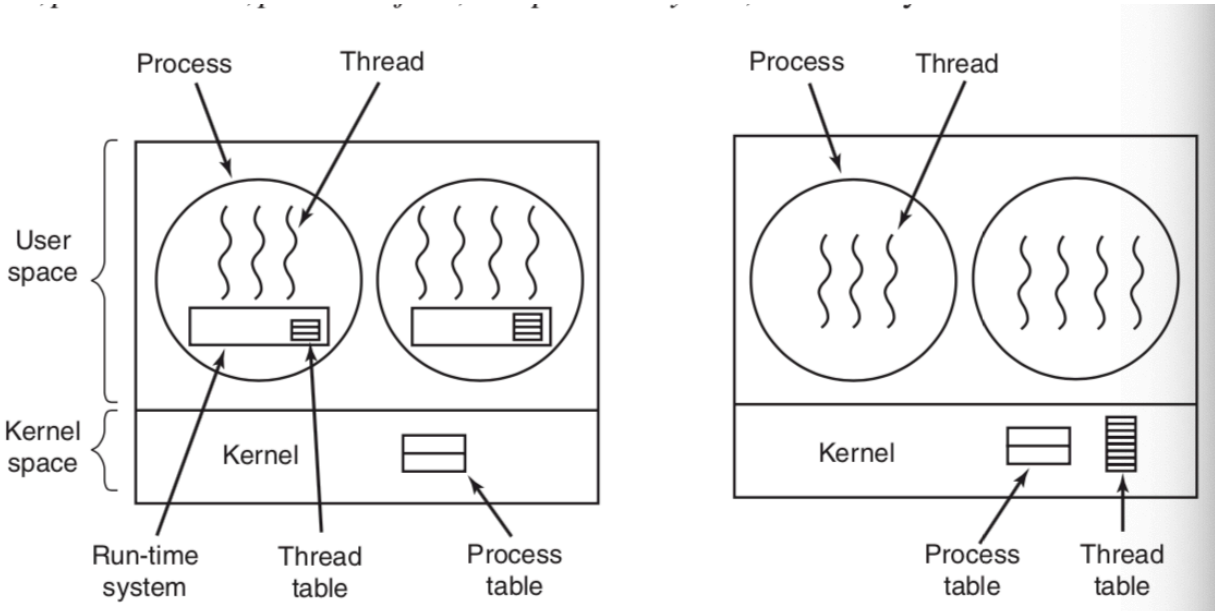


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Implementing Threads in the Kernel:

- the kernel has a thread table that keeps track of all the threads in the system
- when a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table
- when a thread blocks the kernel can run either another thread from the same process or a thread from a different process

Hybrid Implementations:

- combining user-level threads and kernel-level threads
- use kernel-level threads and then multiplex user-level threads onto some or all of them

Scheduler Activations;

- kernel threads are slower than user-level threads
- scheduler activations: the user thread shouldn't have to make special nonblocking

system calls or check in advance if it is safe to make certain system calls
- avoids unnecessary traditions between user and kernel space

Pop-Up Threads:

- pop-up thread: the arrival of a message causes the system to create a new thread to handle the message
- each thread is brand new and doesn't have a history

Making Single-Threaded Code Multithreaded:

- solutions:
    - Prohibit global variables
    - assign each thread its own private global variables
    - to provide each procedure with a jacket that sets a but to mark the library as in use

Interprocess Communication:

- Three issues:
    - 1. how one process can pass info to another
    - 2. making sure two or more processes do not get in each other's way
    - 3. proper sequencing when dependencies are present: if process A produces data and process B prints them, B has to wait until A has produced some data before starting to print

Race Conditions:

- when a process wants to print a file, it enters the file name in a special spooler directory
- printer daemon: periodically checks to see if there are any files to be printed, and if so, it prints them and then removes their names for the directory

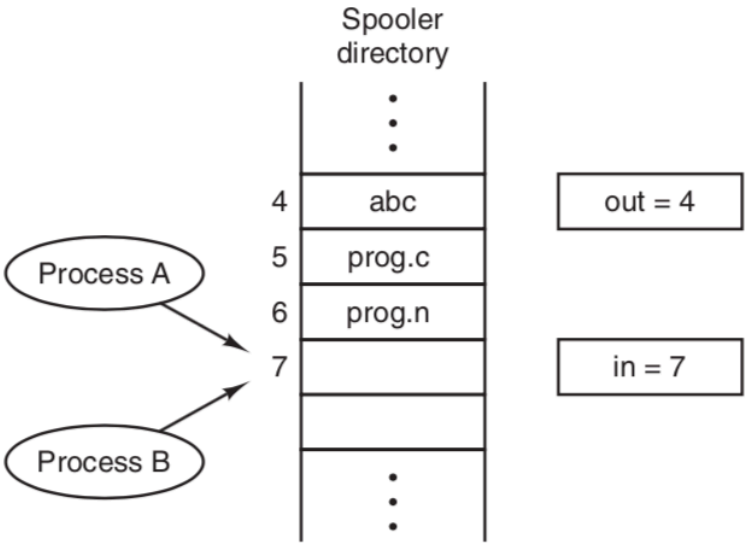


Figure 2-21. Two processes want to access shared memory at the same time.

- Race conditions: when two or more processes are reading or writing some shared data and the final result depends on who runs precisely when

Critical Regions:

- We need mutual exclusions to prevent race conditions.
- mutual exclusions: some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
- we need four conditions for a solution:
    - 1. no two processes may be simultaneously inside their critical regions
    - 2. no assumptions may be made about speeds or the number of CPUs
    - 3. no process running outside its critical region may block any process
    - 4. no process should have to wait forever to enter its critical region

Disabling Interrupts:

- on a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it
    - once a process has disabled interrupts it can examine and update the shared memory without fear that any other process will intervene

Locked Variables:

- consider having a single,, shared (lock) variable, initially 0.
- when a process wants to enter its critical region, if first tests the lock
- if the lock is 0, the process sets it to 1 and enters the critical region

- if the lock is 1, the process just waits until it becomes 0
- 0 means that no process is in its critical region, and a 1 means that some process is in its critical region

Strict Alternation:

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

-
- when process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region
- avoids all races

Peterson's Solutions:

- does not require strict alterations

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Sleep and Wakeup:

- All the solutions above require busy waiting: when a process wants to enter its critical region, it checks to see if the entry is allowed, if not, the process just sits in a tight loop waiting until it is, which is a waste of CPU time
    - can have effects such as the priority inversion problem
- Sleep: is a system call that causes the caller to block, to be suspended until another process wakes it up
- Wakeup: has one parameter, the process to be awakened
- both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups

The Producer-Consumer Problem:

- two processes share a common, fixed-size buffer
- one of them puts info into the buffer and the other one, the consumer, takes it out
- trouble occurs when the producer want to put a new item in the buffer, but its is already full
    - solution: the producer goes to sleep and awakes when the consumer has removed one or more items
    - Or: if the consumer wants to remove an item form the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the biter

and wakes it up

Semaphores:

- semaphores solve lost-wakeup problem
- semaphores: an integer variable to count the number of wakeup saves for future use
- could have the value 0, indicating that no wakeup were saved or some positive value indicating more wakeups were pending
- two operations on semaphores: down and up
- down: checks to see if the value is greater than 0, if so, decrements the value and just continues. if the value is 0, the process is put to sleep without completing the dow for the moment
- up: increments the value of the semaphore addressed

Solving the Producer-Consumer Problem Using Semaphores:

- implement up and down system calls, with OS briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep
- if multiple CPUs are used, each semaphore should be protected by a lock variable

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mutexes:

- mutex: is a shared variable that can be in one of two states: unlocked or locked
- two procedure are used with mutexes:
    - when a thread needs access to a critical region. if the mutex is currently unlocked the call succeeds and the calling thread is free to enter the critical region
    - if the mutex is already locked, the calling threads blocked until the thread in the critical region is finished. If multiple threads are blocked on the mutex, one of

them is chosen at random and allowed to acquire the lock

Futexes:

- “fast user space mutex"
- futex: a feature of Linux that implements basic locking but avoids dropping into the kernel unless it really has to
- futex consists of two parts: a kernel service and a user library
    - kernel service: provides a “wait queue” that allows multiple processes to wait on a lock

Mutexes in Pthreads:

- the basic mechanism uses a mutex variable to guard each critical region
- a thread wishing to enter a critical region first tries to lock the associated mutex
- condition variables: allow threads to block due to some condition not being met
    - allows waiting and blocking to be down atomically
- condition variables and mutexes are always used together

Monitors:

- monitor: is a collections of procedures, variables, and data structures that are all grouped together in a special kind of module or package
- monitors have an important property: only one process can be active in a monitor at any instant
- when a monitor procedure discovers that it cannot continue (e.g. the producer finds the buffer full), it does a wait on some condition variable. This action causes the calling process to block and allowing another process to enter the monitor.

Message Passing:

- uses two primitives, send and receive, which are system calls

Design Issues for Message-Passing Systems:

- messages can be lost by the network
    - solution: the sender and receiver can agree that as soon as a message has been reviewed, the receiver will send back a special acknowledgment message
- authentication is also an issue in message systems: how can the client tell that it is communication with the real file server, and not with an imposter

The Producer-Consumer Problem with Message Passing:

- mailbox: a place to buffer a certain number of messages
- when mailboxes are used the address parameters in the send and receive calls are mailboxes, not processes

- when a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making too for a new one
- rendezvous: eliminate all buffering

Barriers:

- when a process reaches a barrier, it is blocked until all processes have reached the barrier.
- allows groups of processes to synchronize

Avoiding Locks: Read-Copy-Update:

- grace period: any time period in which we know that each thread to be outside the read-side critical section at least once

scheduling:

- the scheduler: the part of the OS that makes the choice about with process will run next
    - uses the scheduling algorithm

Introduction to Scheduling:

- network servers: multiple processes often compete for the CPU
- in addition to picking th right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive:
    - to start with a switch from user mode t kernel mode must occur
    - then the state of the current process must be saved, including storing its registers in the process table

Process Behavior:

- nearly all processes alternate bursts of computing with I/O requests
- compute-bound/CPU-bound: processes that spend most of their time computing
- I/O-bound: processes spend most of their time waiting for I/O
- as CPUs get water, processes tend to get more I/O bound

When to Schedule:

- key issue related to scheduling is when to make scheduling decisions
- variety situations where scheduling is needed:
    - 1. when a new process is created
        - a decision needs to be made whether to run the parent process or the child process
    - 2. a scheduling decision must be made when a process exits
        - that process can no longer run, so some other process must be chosen

from the set of ready processes
    - 3. when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run
    - 4. when an I/O interrupt occurs, a scheduling decision may be made
        - if the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run
        - it is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt
- nonpreemptive scheduling: picks a process to run and then just lets it run until it blocks or voluntarily releases the CPU
- Preemptive scheduling: picks a process and lets it run for a maximum of some fixed time
    - if it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run

Categories of Scheduling Algorithms:

- three environments:
    - 1. Batch
        - there are no users impatiently waiting for a quick response to a short request
        - reduces process switches and thus improves performance
    - 2. Interactive
        - preemption is essential
    - 3. Real Time
        - run only programs that are intended to further the application at hand

Scheduling Algorithm Goals:

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 2-40. Some goals of the scheduling algorithm under different circumstances.

- goal is to keep all parts of the system busy when possible
- Throughput: the number fo jobs per hour that the system completes
- Turnaround time: statistically average time from the moment that a batch job is submitted until the moment it is completed
    - measure how long the average user has to wait for the output
- goal to minimize response time: the time between issuing a command and getting the result

batch systems:
First-Come, First-Served:

- processes are assigned the CPU in the order they request it
- there is a single queue of ready processes
- a single linked list keeps track of all ready processes

Shortest Job First:

- when several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first
- shortest hob first is optimal only when all the jobs are available simultaneously

Shortest Remaining Time Next:

- the scheduler always chooses the process whose remaining run time is the shortest
- run time has to be the known in advanced

Interactive Systems:
Round-Robin Scheduling:

- each process is assigned a time interval, called its quantum, during which it is allowed to run
- if the process is still running at the end of the quantum, the CPU is preempted and given to another process
- if the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks

Priority Scheduling:

- each process is assigned a priority, and the runnable process with the highest priority is allowed to run

Multiple Queues:

- set up priority classes
- process in the highest class were run for one quantum
- process in the next-highest class were run for two quanta
- process in the next one were to run for four quanta …
- whenever a process used up all the quanta allocated to it, it was moved down one class

Shortest Process Next:

- if we regard the execution of each command as a separate “job” then we can minimize overall response time by running time by running the shortest one first
- figuring out which of the currently runnable processes is the shortest one:
    - 1. make estimates based don past behavior and run with shortest estimated time
    - OR 2. estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called aging

Guaranteed Scheduling:

- one promise: if n users are logged in while you are working, you will receive about 1/n of the CPU power
- similarly on a single-user system with n processes running, all things being equal, each one should get 1/n go the CPU cycles
- system must compute the amount of CPU each one is entitled to, the time since creation divided by n

Lottery Scheduling:

- give processes lottery tickets for various system resources, such as CPU time
- whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that tickets gets the resource
- give more lottery tickets to more important processes

Fair-Share Scheduling:

- each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it

Scheduling in Real-Time System:

- real-time system: is one in which time plays an essential role
- hard real time: meaning there are absolute deadlines that must be met
- soft real time: missing an occasional deadline is undesirable, but tolerable
- dividing the program into a number of processes, each of whose behavior is predictable and known in advance
- periodic: occur at regular intervals
- aperiodic: they occur unpredictably

Policy Versus Mechanism:

- problem: schedulers don't always make the right choice
- solution: is to separate the scheduling mechanism from the scheduling policy
- scheduling algorithm is parameterized, and the parameters can be filled in by user processes

Thread Scheduling:

- user-level threads:
    - the kernel is not aware of the existence of threads
    - the kernel picks a process control for a quantum
    - threads can run as long as it wants since there are no thread interrupts
        - not an issue
- Kernel-level threads:
    - kernel picks a particular thread to run
    - thread is given a quantum and is suspended if it exceeds the quantum

Chapter 2 Problems:

2. Suppose that you were to design an advanced computer architecture that did process switching in hardware, instead of having interrupts. What information would the CPU need? Describe how the hardware process switching might work.

You could have a register containing a pointer to the current process-table entry. When I/O completed, the CPU would store the current machine state in the current process-table entry. Then it would go to the interrupt vector for the interrupting device and fetch a pointer to another process-table entry (the service procedure). This process would then be started up

3. On all current computers, at least part of the interrupt handlers are written in assembly language. Why?

Generally, high-level languages do not allow the kind of access to CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.

5. A computer system has enough room to hold five programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?

The chance that all five processes are idle is 1/32, so the CPU idle time is 1/32

7. Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each needing 20 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait

If each job has 50% I/O wait, then it will take 40 minutes to complete in the absence of competition. If run sequentially, the second one will finish 80 minutes after the first one starts. With two jobs, the approximate CPU utilization is $1 - 0.52$. Thus, each one gets 0.375 CPU minute per minute of real time. To accumulate 20 minutes of CPU time, a job must run for 20/0.375 minutes, or about 53.33 minutes. Thus running sequentially the jobs finish after 80 minutes, but running in parallel they finish after 53.33 minutes.

11. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?

No. If a single-threaded process is blocked on the keyboard, it cannot fork.

13. In the text, we described a multithreaded Web server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.

Yes. If the server is entirely CPU bound, there is no need to have multiple threads. It just adds unnecessary complexity. As an example, consider a telephone directory assistance number (like 555-1212) for an area with 1

million people. If each (name, telephone number) record is, say, 64 characters, the entire database takes 64 megabytes and can easily be kept in the server's memory to provide fast lookup.

17. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 12 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

In the single-threaded case, the cache hits take 12 msec and cache misses take 87 msec. The weighted average is $2/3 \times 12 + 1/3 \times 87$. Thus, the mean request takes 37 msec and the server can do about 27 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 12 msec, and the server can handle 83 1/3 requests per second

19. In Fig. 2-15 the thread creations and messages printed by the threads are interleaved at random. Is there a way to force the order to be strictly thread 1 created, thread 1 prints message, thread 1 exits, thread 2 created, thread 2 prints message, thread 2 exists, and so on? If so, how? If not, why not?

Yes, it can be done. After each call to pthread create, the main program could do a pthread join to wait until the thread just created has exited before creating the next thread.

23. Does the busy waiting solution using the *turn* variable (Fig. 2-23) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory?

Yes, it still works, but it still is busy waiting, of course.

29. The producer-consumer problem can be extended to a system with multiple producers and consumers that write (or read) to (from) one shared buffer. Assume that each pro- ducer and consumer runs in its own thread. Will the solution presented in Fig. 2-28, using semaphores, work for this system?

Yes, it will work as is. At a given time instant, only one producer (consumer) can add (remove) an item to (from) the buffer.

31. How could an operating system that can disable interrupts implement semaphores?

To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up, it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of

blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again

37. Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?

. It does not lead to race conditions (nothing is ever lost), but it is effectively busy waiting.

41. Can a measure of whether a process is likely to be CPU bound or I/O bound be deter- mined by analyzing source code? How can this be determined at run time?

In simple cases it may be possible to see if I/O will be limiting by looking at source code. For instance a program that reads all its input files into buffers at the start will probably not be I/O bound, but a problem that reads and writes incrementally to a number of different files (such as a compiler) is likely to be I/O bound. If the operating system provides a facility such as the UNIX ps command that can tell you the amount of CPU time used by a program, you can compare this with the total time to complete execution of the program. This is, of course, most meaningful on a system where you are the only user

43. Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process switch requires a time S, which is effectively wasted (overhead). For round-robin scheduling with quantum Q, give a formula for the CPU efficiency for each of the following:

(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) Q nearly 0

The CPU efficiency is the useful CPU time divided by the total CPU time. When Q ≥ T, the basic cycle is for the process to run for T and undergo a process switch for S. Thus, (a) and (b) have an efficiency of T/(S + T). When the quantum is shorter than T, each run of T will require T/Q process switches, wasting a time ST/Q. The efficiency here is then

T/(T + ST/Q)

which reduces to Q/(Q + S), which is the answer to (c). For (d), we just substitute Q for S and find that the efficiency is 50%. Finally, for (e), as Q → 0 the efficiency goes to 0.

47. Consider a real-time system with two voice calls of periodicity 5 msec each with CPU time per call of 1 msec, and one video stream of periodicity 33 ms with CPU time per call of 11 msec. Is this system schedulable?

Each voice call needs 200 samples of 1 msec or 200 msec. Together they use 400 msec of CPU time. The video needs 11 msec 33 1/3 times a second for a total of about 367 msec. The sum is 767 msec per second of real time so the system is schedulable.

53. Consider a system in which it is desired to separate policy and mechanism for the scheduling of kernel threads. Propose a means of achieving this goal.

The kernel could schedule processes by any means it wishes, but within each process it runs threads strictly in priority order. By letting the user process set the priority of its own threads, the user controls the policy but the kernel handles the mechanism.

13.

14.