OPS Processes

Handout Introduction

These notes were originally written using (Tanenbaum, 1992) but later editions (Tanenbaum, 2001; 2008) contain the same information.

Introduction to Processes

The concept of a process is fundamental to an operating system and they can be viewed as an abstraction of a program. Although to be strict we can say that a program (i.e. an algorithm expressed in some suitable notation) has a process that executes the algorithm and has associated with it input, output and a state.

Computers nowadays can do many things at the same time. They can be writing to a printer, reading from a disc and scanning an image. The computer (more strictly the operating system) is also responsible for running many processes, usually, on the same CPU. In fact, only one process can be run at a time so the operating system has to share the CPU between the processes that are available to be run, whilst giving the illusion that the computer is doing many things at the same time.
This approach can be directly contrasted with the first computers. They could only run one program at a time and although this may seem archaic now, it does have its advantages and it makes most of this part of the course irrelevant and we can simply ignore the issues raised.

Therefore, the main point of this part of the course is to consider how an operating system deals with processes when we allow many to run in ***pseudoparallelism***.

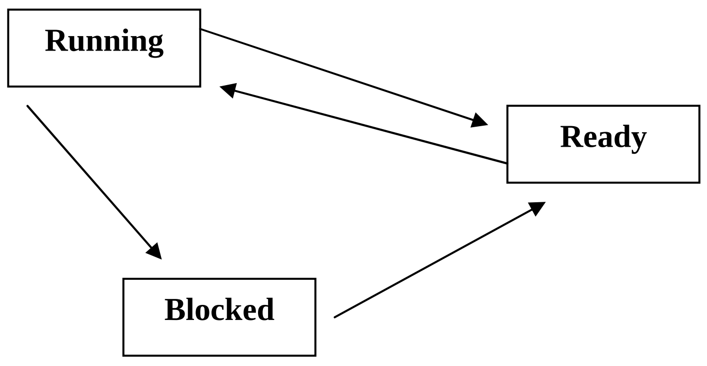
Although, with a single CPU, we know the computer can only execute a single process at a given moment in time. It is important to realise that this is the case. It is also important to realise that one process can have an effect on another process which is not currently running; as we shall see later.

Process States

A process may be in one of three states

Running.	Only one process can be running at any one time (assuming a single processor machine). A running process is the process that is actually using the CPU at that time.
Ready.	A process that is ready is runnable but cannot get access to the CPU due to another process using it.
Blocked.	A blocked process is unable to run until some external event has taken place. For example, it may be waiting for data to be retrieved from a disc.

A state transition diagram can be used to represent the various states and the transition between those states.



You can see from this that a running process can either be blocked (i.e. it needs to wait for an external event) or it can go to a ready state (for example, the scheduler allows another process to use the CPU).
A ready process can only move to a running state whilst a blocked process can only move to a ready state.

It should be apparent that the job of the scheduler is concerned with deciding which one of the processes in a ready state should be allowed to move to a running state (and thus use the CPU).

Process Control Blocks (PCB)

Assuming we have a computer with one processor, it is obvious that the computer can only execute one process at a time. However, when a process moves from a running state to a ready or blocked state it must store certain information so that it can remember where it was up to when it moves back to a running state. For example, it needs to know which instruction it was about to execute, which record it was about to read from its input file and the values of various variables that it was using as working storage.
Each process has associated with it a process table. This is used to hold all the information that a process needs in order to restart from where it left off when it moves from a ready state to a running state.
Different systems will hold different information in the process block.
This table shows a typical set of data

Process Management
Registers
Program Counter
Program Status Word
Stack Pointer
Process State
Time when process started
CPU time used
Time of next alarm
Process id

You will notice that, as well as information to ensure the process can start again (e.g. Program Counter), the process control block also holds accounting information such as the time the process started and how much CPU time has been used.
You should note that this is only a sample of the information held. There will be other information, not least of all concerned with the files being used and the memory the process is using.

Race Conditions

It is sometimes necessary for two processes to communicate with one another. This can either be done via shared memory or via a file on disc. It does not really matter.
We are not discussing the situation where a process can write some data to a file that is read by another process at a later time (maybe days, weeks or even months). We are talking about two processes that need to communicate at the time they are running.
Take, as an example, one *type* of process (i.e. there could be more than one process of this type running) that checks a counter when it starts running. If the counter is at a certain value, say *x*, then the process terminates as only *x* copies of the process are allowed to run at any one time.
This is how it works

- The process starts
- The counter, *i*, is read from the shared memory
- If the i = *x* the process terminates else *i = i + 1*
- *i* is written back to the shared memory

Sounds okay. But consider this scenario

- Process 1, P_1, starts
- P_1 reads the counter, i_1, from the shared memory. Assume $i_1 = 3$ (that is three processes of this type are already running)
- P_1 gets interrupted and is placed in a ready state
- Process 2, P_2, starts
- P_2 reads the counter, i_2, from the shared memory; $i_2 = 3$
- Assume $i_2 < x$ so $i_2 = i_2 + 1$ (i.e. 4)
- i_2 is written back to shared memory
- P_2 is moved to a ready state and P_1 goes into a running state
- Assume $i_1 < x$ so $i_1 = i_1 + 1$ (i.e. 4)
- i_1 is written back to the shared memory

We now have the situation where we have five processes running but the counter is only set to four

This problem is known as a ***race condition***.

Critical Sections

One way to avoid race conditions is not to allow two processes to be in their critical sections at the same time (by critical section we mean the part of the process that accesses a shared variable). That is, we need a mechanism of ***mutual exclusion***. Some way of ensuring that one processes, whilst using the shared variable, does not allow another process to access that variable.
In fact, to provide a good solution to the problem of race conditions we need four conditions to hold.

1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about the speed or the number of processors.
3. No process running outside its critical section may block other processes
4. No process should have to wait forever to enter its critical section.

As we shall see, it is difficult to devise a method that meets all these conditions.

Implementing Mutual Exclusion with Busy Waiting

Disabling Interrupts

Perhaps the most obvious way of achieving mutual exclusion is to allow a process to disable interrupts before it enters its critical section and then enable interrupts after it leaves its critical section.
By disabling interrupts the CPU will be unable to switch processes. This guarantees that the process can use the shared variable without another process accessing it.
But, disabling interrupts, is a major undertaking. At best, the computer will not be able to service interrupts for, maybe, a long time (who knows what a process is doing in its critical section?). At worst, the process may never enable interrupts, thus (effectively) crashing the computer.
Although disabling interrupts might seem a good solution its disadvantages far outweigh the advantages.

Lock Variables

Another method, which is obviously flawed, is to assign a lock variable. This is set to (say) 1 when a process is in its critical section and reset to zero when a processes exits its critical section.
It does not take a great leap of intuition to realise that this simply moves the problem from the shared variable to the lock variable.

Strict Alternation

Process 0

```
While (TRUE) {
      while (turn != 0); // wait
      critical_section();
      turn = 1;
      noncritical_section();
}
```

Process 1

```
While (TRUE) {
      while (turn != 1); // wait
      critical_section();
      turn = 0;
      noncritical_section();
}
```

These code fragments offer a solution to the mutual exclusion problem.
Assume the variable *turn* is initially set to zero.
Process 0 is allowed to run. It finds that *turn* is zero and is allowed to enter its critical region. If process 1 tries to run, it will also find that *turn* is zero and will have to wait (the while statement) until *turn* becomes equal to 1.
When process 0 exits its critical region it sets *turn* to 1, which allows process 1 to enter its critical region.
If process 0 tries to enter its critical region again it will be blocked as *turn* is no longer zero.
However, there is one major flaw in this approach. Consider this sequence of events.

- Process 0 runs, enters its critical section and exits; setting turn to 1. Process 0 is now in its non-critical section. Assume this non-critical procedure takes a long time.
- Process 1, which is a much faster process, now runs and once it has left its critical section turn is set to zero.
- Process 1 executes its non-critical section very quickly and returns to the top of the procedure.
- The situation is now that process 0 is in its non-critical section and process 1 is waiting for turn to be set to zero. In fact, there is no reason why process 1 cannot enter its critical region as process 0 is not in its critical region.

What we can see here is violation of one of the conditions that we listed above (number 3). That is, a process, not in its critical section, is blocking another process.
If you work through a few iterations of this solution you will see that the processes must enter their critical sections in turn; thus this solution is called ***strict alternation***.

Peterson's Solution

A solution to the mutual exclusion problem that does not require strict alternation, but still uses the idea of lock (and warning) variables together with the concept of taking turns is described in (Dijkstra, 1965). In fact the original idea came from a Dutch mathematician (T. Dekker). This was the first time the mutual exclusion problem had been solved using a software solution.
(Peterson, 1981), came up with a much simpler solution.

The solution consists of two procedures, shown here in a C style syntax.

```
int No_Of_Processes;                // Number of processes
int turn;                           // Whose turn is it?
int interested[No_Of_Processes];    // All values initially FALSE

void enter_region(int process) {
   int other;                       // number of the other process

   other = 1 – process;             // the opposite process
   interested[process] = TRUE;      // this process is interested
   turn = process;                  // set flag
   while(turn == process && interested[other] == TRUE); // wait
}

void leave_region(int process) {
   interested[process] = FALSE;     // process leaves critical region
}
```

A process that is about to enter its critical region has to call enter_region. At the end of its critical region it calls leave_region.
Initially, both processes are not in their critical region and the array *interested* has all (*both* in the above example) its elements set to false.
Assume that process 0 calls enter_region. The variable *other* is set to one (the other process number) and it indicates its interest by setting the relevant element of *interested*. Next it sets the *turn* variable, before coming across the while loop. In this instance, the process will be allowed to enter its critical region, as process 1 is not interested in running.

Now process 1 could call enter_region. It will be forced to wait as the other process (0) is still interested. Process 1 will only be allowed to continue when *interested[0]* is set to false which can only come about from process 0 calling leave_region.

If we ever arrive at the situation where both processes call enter region at the same time, one of the processes will set the *turn* variable, but it will be immediately overwritten.
Assume that process 0 sets *turn* to zero and then process 1 immediately sets it to 1. Under these conditions process 0 will be allowed to enter its critical region and process 1 will be forced to wait.

Test and Set Lock (TSL)

If we are given assistance by the instruction set of the processor we can implement a solution to the mutual exclusion problem. The instruction we require is called test and set lock (TSL). This instructions reads the contents of a memory location, stores it in a register and then stores a non-zero value at the address. This operation is guaranteed to be indivisible. That is, no other process can access that memory location until the TSL instruction has finished.

This assembly (like) code shows how we can make use of the TSL instruction to solve the mutual exclusion problem.

```
enter_region:
   tsl  register, flag ; copy flag to register and set flag to 1
   cmp  register, #0   ;was flag zero?
   jnz  enter_region   ;if flag was non zero, lock was set , so loop
   ret                 ;return (and enter critical region)

leave_region:
   mov  flag, #0       ; store zero in flag
   ret                 ;return
```

Assume, again, two processes.

Process 0 calls enter_region. The tsl instruction copies the flag to a register and sets it to a non-zero value. The flag is now compared to zero (cmp - compare) and if found to be non-zero (jnz – jump if non-zero) the routine loops back to the top. Only when process 1 has set the flag to zero (or under initial conditions), by calling leave_region, will process 0 be allowed to continue.

Comments

Of all the solutions we have looked at, both Peterson's and the TSL solutions solve the mutual exclusion problem.
However, both of these solutions have the problem of ***busy waiting***. That is, if the process is not allowed to enter its critical section it sits in a tight lop waiting for a condition to be met. This is obviously wasteful in terms of CPU usage.
It can also have, not so obvious disadvantages.
Suppose we have two processes, one of high priority, *h*, and one of low priority, *l*. The scheduler is set so that whenever *h* is in ready state it must be run. If *l* is in its critical section when *h* becomes ready to run, *l* will be placed in a ready state so that *h* can be run. However, if *h* tries to enter its critical section then it will be blocked by *l*, which will never be given the opportunity of running and leaving its critical section. Meantime, *h* will simply sit in a loop forever.
This is sometimes called the ***priority inversion problem***.

Sleep and Wakeup

In this section, instead of a process doing a busy waiting we will look at procedures that send the process to ***sleep***. In reality, it is placed in a blocked state. The important point is that it is not using the CPU by sitting in a tight loop.
To implement a sleep and wakeup system we need access to two system calls (SLEEP and WAKEUP). These can be implemented in a number of ways. One method is for SLEEP to simply block the calling process and for WAKEUP to have one parameter; that is the process it has to wakeup.
An alternative is for both calls to have one parameter, this being a memory address which is used to match the SLEEP and WAKEUP calls.

The Producer-Consumer Problem

This problem is outlined later in this handout. You should read this now.
To implement a solution to the problem using SLEEP/WAKEUP we need to maintain a variable, *count*, that keeps track of the number of items in the buffer
The producer will check count against *n* (maximum items in the buffer). If *count* = *n* then the producer sends itself the sleep. Otherwise it adds the item to the buffer and increments *n*.
Similarly, when the consumer retrieves an item from the buffer, it first checks if *n* is zero. If it is it sends itself to sleep. Otherwise it removes an item from the buffer and decrements *count*.

The calls to WAKEUP occur under the following conditions.

- Once the producer has added an item to the buffer, and incremented count, it checks to see if *count* = 1 (i.e. the buffer was empty before). If it is, it wakes up the consumer.
- Once the consumer has removed an item from the buffer, it decrements count. Now it checks count to see if it equals *n*-1 (i.e. the buffer was full). If it does it wakes up the producer.

Here is the producer and consumer code.

```
int BUFFER_SIZE = 100;
int count = 0;

void producer(void) {
 int item;
 while(TRUE) {
  produce_item(&item);                   // generate next item
  if(count == BUFFER_SIZE) sleep ();     // if buffer full, go to sleep
  enter_item(item);                      // put item in buffer
  count++;                               // increment count
  if(count == 1) wakeup(consumer);       // was buffer empty?
 }
}

void consumer(void) {
 int item;
 while(TRUE) {
  if(count == 0) sleep ();               // if buffer is empty, sleep
  remove_item(&item);                    // remove item from buffer
  count--;                               // decrement count
  if(count == BUFFER_SIZE - 1) wakeup(producer); // was buffer full?
  consume_item(&item);                   // print item
 }
}
```

This seems logically correct but we have the problem of race conditions with *count*.
The following situation could arise.

- The buffer is empty and the consumer has just read count to see if it is equal to zero.
- The scheduler stops running the consumer and starts running the producer.
- The producer places an item in the buffer and increments count.
- The producer checks to see if count is equal to one. Finding that it is, it assumes that it was previously zero which implies that the consumer is sleeping – so it sends a wakeup.
- In fact, the consumer is not asleep so the call to wakeup is lost.
- The consumer now runs – continuing from where it left off – it checks the value of count. Finding that it is zero it goes to sleep. As the wakeup call has already been issued the consumer will sleep forever.
- Eventually the buffer will become full and the producer will send itself to sleep.
- Both producer and consumer will sleep forever.

One solution is to have a ***wakeup waiting bit*** that is turned on when a wakeup is sent to a process that is already awake. If a process goes to sleep, it first checks the wakeup bit. If set the bit will be turned off, but the process will not go to sleep.
Whilst seeming a workable solution it suffers from the drawback that you need an ever increasing number wakeup bits to cater for larger number of processes.

Semaphores

In (Dijkstra, 1965) the suggestion was made that an integer variable be used that recorded how many wakeups had been saved. Dijkstra called this variable a ***semaphore***. If it was equal to zero it indicated that no wakeup's were saved. A positive value shows that one or more wakeup's are pending.
Now the sleep operation (which Dijkstra called DOWN) checks the semaphore to see if it is greater than zero. If it is, it decrements the value (using up a stored wakeup) and continues. If the semaphore is zero the process sleeps.

The wakeup operation (which Dijkstra called UP) increments the value of the semaphore. If one or more processes were sleeping on that semaphore then one of the processes is chosen and allowed to complete its DOWN.
Checking and updating the semaphore must be done as an ***atomic*** action to avoid race conditions.

Here is an example of a series of Down and Up's. We are assuming we have a semaphore called *mutex* (for mutual exclusion). It is initially set to 1. The subscript figure, in this example, represents the process, p, that is issuing the Down.

```
Down$_1$(mutex)     // p1 enters critical section (mutex = 0)
Down$_2$(mutex)     // p2 sleeps (mutex = 0)
Down$_3$(mutex)     // p3 sleeps (mutex = 0)
Down$_4$(mutex)     // p4 sleeps (mutex = 0)
Up(mutex)        // mutex = 1 and chooses p3
  Down$_3$(mutex)   // p3 completes its down (mutex = 0)
Up(mutex)        // mutex = 1 and chooses p2
  Down$_2$(mutex)   // p2 completes its down (mutex = 0)
Up(mutex)        // mutex = 1 and chooses p2
  Down$_1$(mutex)   // p1 completes its down (mutex = 0)
Up(mutex)        // mutex = 1 and chooses p4
  Down$_4$(mutex)   // p4 completes its down (mutex = 0)
```

From this example, you can see that we can use semaphores to ensure that only one process is in its critical section at any one time, i.e. the principle of mutual exclusion.

We can also use semaphores to synchronise processes. For example, the produce and consume functions in the producer-consumer problem. Take a look at this program fragment.

```
int BUFFER_SIZE = 100;
typedef int semaphore;

semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;

void producer(void) {
 int item;
 while(TRUE) {
  produce_item(&item);      // generate next item
  down(&empty);             // decrement empty count
  down(&mutex);             // enter critical region
  enter_item(item);         // put item in buffer
  up(&mutex);               // leave critical region
  up(&full);                // increment count of full slots
 }
}

void consumer(void) {
 int item;
 while(TRUE) {
  down(&full);              // decrement full count
  down(&mutex);             // enter critical region
  remove_item(&item);       // remove item from buffer
  up(&mutex);               // leave critical region
  up(&empty);               // increment count of empty slots
  consume_item(&item);      // print item
 }
}
```

The *mutex* semaphore (given the above example) should be self-explanatory.
The *empty* and *full* semaphore provide a method of synchronising adding and removing items to the buffer. Each time an item is removed from the buffer a *down* is done on *full*. This decrements the semaphore and, should it reach zero the consumer will sleep until the producer adds another item. The consumer also does an *up* an *empty*. This is so that, should the producer try to add an item to a full buffer it will sleep (via the *down* on *empty*) until the consumer has removed an item.

Inter-Process Communication Problems

In this section we describe a few classic inter process communication problems. If you have the time and the inclination you might like to try and write a program which solves these problems.

The Producer-Consumer Problem

Assume there is a producer (which produces goods) and a consumer (which consumes goods). The producer, produces goods and places them in a fixed size buffer. The consumer takes the goods from the buffer.
The buffer has a finite capacity so that if it is full, the producer must stop producing.
Similarly, if the buffer is empty, the consumer must stop consuming.
This problem is also referred to as the ***bounded buffer*** problem.

The type of situations we must cater for are when the buffer is full, so the producer cannot place new items into it. Another potential problem is when the buffer is empty, so the consumer cannot take from the buffer.

The Dining Philosophers Problem

This problem was posed by (Dijkstra, 1965).
Five philosophers are sat around a circular table. In front of each of them is a bowl of food. In between each bowl there is a fork. That is there are five forks in all.
Philosophers spend their time either eating or thinking. When they are thinking they are not using a fork. When they want to eat they need to use two forks. They must pick up one of the forks to their right or left. Once they have acquired one fork they must acquire the other one. They may acquire the forks in any order. Once a philosopher has two forks they can eat. When finished eating they return both forks to the table.
The question is, can a program be written, for each philosopher, that never gets stuck, that is, a philosopher is waiting for a fork forever.

The Readers Writers Problem

This problem was devised by (Courtois et al., 1971). It models a number of processes requiring access to a database. Any number of processes may read the database but only one can write to the database.
The problem is to write a program that ensures this happens.

The Sleeping Barber Problem

A barber shop consists of a waiting room with *n* chairs. There is another room that contains the barbers chair. The following situations can happen.

- If there are no customers the barber goes to sleep.
- If a customer enters and the barber is asleep (indicating there are no other customers waiting) he wakes the barber and has a haircut.
- If a customer enters and the barber is busy and there are spare chairs in the waiting room, the customer sits in one of the chairs.
- If a customer enters and all the chairs in the waiting room are occupied, the customer leaves.

The problem is to program the barber and the customers without getting into race conditions.

Process Scheduling

Scheduling Objectives

If we assume we only have one processor and there are two or more processes in a ready state, how do we decide which process to schedule next? Or more precisely, which ***scheduling algorithm*** does the ***scheduler*** use in deciding which process should be moved to a running state?
These questions are the subject of this section.

In trying to schedule processes, the scheduler tries to meet a number of objectives

1.	**Fairness**	**:**	Ensure each process gets a fair share of the CPU
2.	**Efficiency**	**:**	Ensure the CPU is busy 100% of the time. In practise, a measure of between 40% (for a lightly loaded system) to 90% (for a heavily loaded system) is acceptable
3.	**Response Times**	**:**	Ensure interactive users get good response times
4.	**Turnaround**	**:**	Ensure batch jobs are processed in acceptable time
5.	**Throughput**	**:**	Ensure a maximum number of jobs are processed

Of course, the scheduler cannot meet all of these objectives to an optimum level. For example, in trying to give interactive users good response times, the batch jobs may have to suffer.
Many large companies, that use mainframes, address these types of problems by taking many of the scheduling decisions themselves. For example, a company the lecturer used to work for did not allow batch work during the day. Instead they gave the TP (Transaction Processing) system all the available resources

so that the response times for the users (many of which were dealing with the public in an interactive way) was as fast as possible.
The batch work was run overnight when the interactive workload was much less, typically only operations staff and technical support personnel.
However, these type of problems are likely to increase as the world becomes “smaller.” If a company operates a mainframe that is accessible from all over the world then the concept of night and day no longer hold and there may be a requirement for TP access 24 hours a day and the batch work somehow has to be fitted in around this workload.

Preemptive Scheduling

A simple scheduling algorithm would allow the currently active process to run until it has completed. This would have several advantages

1. We would no longer have to concern ourselves with race conditions as we could be sure that one process could not interrupt another and update a shared variable.
2. Scheduling the next process to run would simply be a case of taking the highest priority job (or using some other algorithm, such as FIFO algorithm).

Note : We could define completed as when a process decides to relinquish control of the CPU. The process may not have completed but it would only relinquish control when it was safe to do so (e.g. not in the middle of updating a shared variable).

But the disadvantages of this approach far outweigh the advantages.

- A rogue process may never relinquish control, effectively bringing the computer to a standstill.
- Processes may hold the CPU too long, not allowing other applications to run.

Therefore, it is usual for the scheduler to have the ability to decide which process can use the CPU and, once it has had its slice of time then it is placed into a ready state and the next process allowed to run. This type of scheduling is called ***preemptive scheduling***.
This disadvantage of this method is that we need to cater for race conditions as well as having the responsibility of scheduling the processes.

Typical Process Activity

There is probably no such thing as an average process but studies have been done on typical processes. It has been found that processes come in two varieties.
Processes which are I/O bound, and only require the CPU in short bursts. Then there are processes that require the CPU for long bursts. Of course, a process can combine these two attributes. At the start of its processing it is I/O bound so that it only requires short bursts of the CPU. Later in its processing, it is heavily I/O bound so that (if it was allowed) it would like the CPU for long periods.
An important aspect, when scheduling is to consider the *CPU Bursts* that are required. That is how long the process needs the CPU before it will either finish or move to a blocked state. From a scheduling point of view we need not concern ourselves with processes that are waiting for I/O. As far as the scheduler is concerned, this is a good thing, as it is one less process to worry about.
However, the scheduler needs to be concerned about the burst time. If a process has a long burst time it may need to have access to the CPU on a number of occasions in order to complete its burst sequence.
The problem is that the scheduler cannot know what burst time a process has before it schedules the process. Therefore, when we look at the scheduling algorithms below, we can only look at the effect of the burst time and the effect it has on the average running time of the processes (it is usual to measure the effect of a scheduling policy using average figures).

First Come – First Served Scheduling (FCFS)

An obvious scheduling algorithm is to execute the processes in the order they arrive and to execute them to completion. In fact, this simply implements a non-preemptive scheduling algorithm.
It is an easy algorithm to implement. When a process becomes ready it is added to the tail of ready queue. This is achieved by adding the Process Control Block (PCB) to the queue.

When the CPU becomes free the process at the head of the queue is removed, moved to a running state and allowed to use the CPU until it is completed.
The problem with FCFS is that the average waiting time can be long. Consider the following processes

Process	Burst Time
P1	27
P2	9
P3	2

P1 will start immediately, with a waiting time of 0 milliseconds (ms). P2 will have to wait 27ms. P3 will have to wait 36ms before starting. This gives us an average waiting time of 21ms (i.e. (0 + 27 + 36) /3).

Now consider if the processes had arrived in the order P2, P3, P1. The average waiting time would now be 6.6ms (i.e. (0 + 9 + 11) /3).
This is obviously a big saving and all due to the fact the way the jobs arrived. It can be shown that FCFS is not generally minimal, with regard to average waiting time and this figure varies depending on the process burst times.
The FCFS algorithm can also have other undesirable effects. A CPU bound job may make the I/O bound (once they have finished the I/O) wait for the processor. At this point the I/O devices are sitting idle.
When the CPU bound job finally does some I/O, the mainly I/O processes use the CPU quickly and now the CPU sits idle waiting for the mainly CPU bound job to complete its I/O.
Although this is a simplistic example, you can appreciate that FCFS can lead to I/O devices and the CPU both being idle for long periods.

Shortest Job First (SJF)

Using the SJF algorithm, each process is tagged with the length of its next CPU burst. The processes are then scheduled by selecting the shortest job first.

Consider these processes, P1..P4. Assume they arrived in the order P1..P4.

Process	Burst Time	Wait Time
P1	12	0
P2	19	12
P3	4	31
P4	7	35

If we schedule the processes in the order they arrive then the average wait time is 19.5 (78/4). If we run the processes using the burst time as a priority then the wait times will be 0, 4, 11 and 23; giving an average wait time of 9.50.

In fact, the SJF algorithm is provably optimal with regard to the average waiting time. And, intuitively, this is the case as shorter jobs add less to the average time, thus giving a shorter average.

The problem is we do not know the burst time of a process before it starts.

For some systems (notably batch systems) we can make fairly accurate estimates but for interactive processes it is not so easy.
One approach is to try and estimate the length of the next CPU burst, based on the processes previous activity.

To do this we can use the following formula

$T_{n+1} = at_n + (1 - a)T_n$

Where

a,	$0 <= a <= 1$
T_n,	stores the past history
t_n,	contains the most recent information

What this formula allows us to do is weight both the history of the burst times and the most recent burst time. The weight is controlled by a.
If $a = 0$ then $T_{n+1} = T_n$ and recent history (the most recent burst time) has no effect. If $a = 1$ then the history has no effect and the guess is equal to the most recent burst time.

A value of 0.5 for a is often used so that equal weight is given to recent and past history.

This formula has been reproduced on a spreadsheet (which I hope to make available on the WWW site associated with this course) so that you can experiment with the various values.

Priority Scheduling

Shortest Job First is just a special case of *priority* scheduling. Of course, we can use a number of different measures as priority.
Another example of setting priorities based on the resources they have previously used is as follows.

Assume processes are allowed 100ms before the scheduler preempts it. If a process only used, say 2ms, then it is likely to be a job that is I/O bound and it is in our interest to allow this job to run as soon as it has completed I/O – in the hope that it will go away and do some more I/O; thus making effective use of the processor as well as the I/O devices.
If a job used all its 100ms we might want to give this job a lower priority, in the belief that we can get smaller jobs completed first before we allow the longer jobs to run.

One method of calculating priorities based on this reasoning is to use the formula

$1 / (n / p)$

Where

n,	is the last CPU burst for that process
p,	is the CPU time allowed for each process before it is preempted (100ms in our example)

Plugging in some real figures we can assign priorities as follows

CPU Burst Last Time	Processing Time Slice	Priority Assigned
100	100	1
50	100	2
25	100	4
5	100	20
2	100	50
1	100	100

Another way of assigning priorities is to set them externally. During the day interactive jobs may be given a high priority and batch jobs are only allowed to run when there are no interactive jobs. Another alternative is to allow users who pay more for their computer time to be given higher priority for their jobs.

One of the problems with priority scheduling is that some processes may never run. There may always be higher priority jobs that get assigned the CPU. This is known as ***indefinite blocking*** or ***starvation***.
One solution to this problem is called ***aging***. This means that the priority of jobs are gradually increased until even the lowest priority jobs will become the highest priority job in the system. This could be done, for example, by increasing the priority of a job after it has been in the system for a certain length of time.

Round Robin Scheduling (RR)

The processes to be run are held in a queue and the scheduler takes the first job off the front of the queue and assigns it to the CPU (so far the same as FCFS).
In addition, there is a unit of time defined (called a ***quantum***). Once the process has used up a quantum the process is preempted and a context switch occurs. The process which was using the processor is placed at the back of the ready queue and the process at the head of the queue is assigned to the CPU.
Of course, instead of being preempted the process could complete before using its quantum. This would mean a new process could start earlier and the completed process would not be placed at the end of the queue (it would either finish completely or move to a blocked state whilst it waited for some interrupt, for example I/O).

The average waiting time using RR can be quite long. Consider these processes (which we assume all arrive at time zero).

Process	**Burst Time**
P1	24
P2	3
P3	3

Assume a quantum of 4ms is being used. Process 1 will run first and will be preempted after 4ms. Next process 2 will run for 3ms, followed by process 3 (for 3ms) Following this process 1 will run to completion, taking five quantum.
The waiting time is 17ms (process 2 had to wait 4ms, process 3 had to 7ms and process 1 had to wait 6ms), giving an average of 5.66ms. If we compare this to SJF, the average is only 3ms (9/3).

But, the main concern with the RR algorithm is the length of the quantum. If it is too long then processes will never get preempted and we have the equivalent of FCFS. If we switch processes after every ms then we make it appear as if every process has its own processor that runs at 1/*n* the speed of the actual processor.
In fact, this ignores the effect of context switching. If the quantum is too small then the processor will spend large amounts of time context switching, and not processing data.
Say, for example, we set the quantum to 5ms and it takes 5ms to execute a process switch then we are using half the CPU capability simply switching processes.
A quantum of around 100ms is often used.

Multilevel Queue Scheduling

We have previously mentioned that there are two typical types of processes in a computer system. These are interactive jobs and batch jobs. By their nature interactive jobs tend to be shorter and batch jobs tend to be longer.
Most companies will try to schedule interactive jobs before batch jobs (at least during the day) so that the response time to their users and customers is kept to a minimum.
One way of achieving this is to set up different queues to cater for different process types.
Each queue may have its own scheduling algorithm. The background queue will typically use the FCFS algorithm, whilst the interactive queue may use the RR algorithm.

Furthermore, the scheduler now has to decide which queue to run (as well as using the scheduling algorithm with each queue). There are two main methods. Higher priority queues can be processed until they are empty before the lower priority queues are executed.

Alternatively, each queue can be given a certain amount of the CPU. Maybe, the interactive queue could be assigned 80% of the CPU, with the batch queue being given 20%.

It should also be noted that there can be many other queues. Many systems will have a ***system queue***. This queue will contain processes that are important to keep the system running. For this reason these processes normally have the highest priority.

Multilevel Feedback Queue Scheduling

In multilevel queue scheduling we assign a process to a queue and it remains in that queue until the process is allowed access to the CPU. That is, processes do not move between queues. This is a reasonable scheme as batch processes do not suddenly change to an interactive process and vice versa. However, there may be instances when it is advantageous to move process between queues. Multilevel feedback queue scheduling allows us to do this.

Consider processes with different CPU burst characteristics. If a process uses too much of the CPU it will be moved to a lower priority queue. This will leave I/O bound and (fast) interactive processes in the higher priority queue(s).

Assume we have three queues (Q0, Q1 and Q2). Q0 is the highest priority queue and Q2 is the lowest priority queue.
The scheduler first executes process in Q0 and only considers Q1 and Q2 when Q0 is empty. Whilst running processes in Q1, if a new process arrived in Q0, then the currently running process is preempted so that the Q0 process can be serviced.

Any job arriving is put into Q0. When it runs, it does so with a quantum of 8ms (say). If the process does not complete, it is preempted and placed at the end of the Q1 queue. This queue (Q1) has a time quantum of 16ms associated with it. Any processes not finishing in this time are demoted to Q2, with these processes being executed on a FCFS basis.

The above description means that any jobs that require less than 8ms of the CPU are serviced very quickly. Any processes that require between 8ms and 24ms are also serviced fairly quickly. Any jobs that need more than 24ms are executed with any spare CPU capacity once Q0 and Q1 processes have been serviced.

In implementing a multilevel feedback queue there are various parameters that define the scheduler.

- The number of queues
- The scheduling algorithm for each queue
- The method used to demote processes to lower priority queues
- The method used to promote processes to a higher priority queue (presumably by some form of aging)
- The method used to determine which queue a process will enter

If you are interested in scheduling algorithms then you might like to implement a multilevel feedback queue scheduling algorithm. Once implemented you can mimic any of the other scheduling algorithms we have discussed (e.g. by defining only one queue, with a suitable quantum and the RR algorithm we can generalise to the RR algorithm).

Two Level Scheduling

In our discussions of scheduling algorithms we have assumed that the process are all available in memory so that the context switching is fast. However, if the computer is low on memory then some processes may be swapped out to disc. If a process on disc is scheduled next the context switch will take a long time compared to a context switch to a process in memory.
Therefore, it is sensible to schedule only those processes in memory. This is the responsibility of a top level scheduler.

A second scheduler is invoked periodically to remove processes from memory to disc and move some processes on disc to memory. This scheduler will use various parameters to decide which processes to move. Amongst these could be

- How long has it been since a process has been swapped in or out?
- How much CPU time has the process recently had?
- How big is the process (on the basis that small ones do not get in the way)?
- What is the priority of the process?

Guaranteed Scheduling

The scheduling algorithms we have considered make no guarantees about the jobs that are run. In some circumstances we might need to make certain promises (and live up to them).
In this day and age it is common to have a Service Level Agreement (SLA) with your customers. Part of the SLA may specify that the average response time has to be (say) two seconds. This means that the process, when it arrives in the ready queue must be serviced in (say) 0.5 of a second. (The two second response time is from the user pressing the return key/mouse button and the information being displayed on the screen – therefore, other factors, such as the network, have to be taken into account).

In order to meet these times the support teams will manipulate the various scheduling algorithm parameters to ensure they meet the SLA. If the average response time starts to get near the time specified in the SLA then they will recommend various courses of actions, in the worst case it will mean upgrading the computer.

Another form of guaranteed scheduling is described in (Tanenbaum, 1992).

Evaluation of Process Scheduling Algorithms

This topic is not covered in (Tanenbaum, 1992). It is covered in (Silberschatz, 1994) (pages 152 – 158) although these notes are complete in their own right.

In the section above we looked at various scheduling algorithms. But how do we decide which one to use? The first thing we need to decide is how we will evaluate the algorithms. To do this we need to decide on the relative importance of the factors we listed above (Fairness, Efficiency, Response Times, Turnaround and Throughput). Only once we have decided on our evaluation method can we carry out the evaluation.

Deterministic Modeling

This evaluation method takes a predetermined workload and evaluates each algorithm using that workload. Assume we are presented with the following processes, which all arrive at time zero.

Process	Burst Time
P1	9
P2	33
P3	2
P4	5
P5	14

Which of the following algorithms will perform best on this workload?
First Come First Served (FCFS), Non Preemptive Shortest Job First (SJF) and Round Robin (RR). Assume a quantum of 8 milliseconds.
Before looking at the answer (supplied as a separate document), try to calculate the figures for each algorithm.

The advantages of deterministic modeling is that it is exact and fast to compute. The disadvantage is that it is only applicable to the workload that you use to test. As an example, use the above workload but assume P1 only has a burst time of 8 milliseconds. What does this do to the average waiting time?
Of course, the workload might be typical and scale up but generally deterministic modeling is too specific and requires too much knowledge about the workload.

Queuing Models

Another method of evaluating scheduling algorithms is to use queuing theory. Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process. We can now generate these times with a certain distribution.
We can also generate arrival times for processes (arrival time distribution).
If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.

Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.

One useful formula is ***Little's Formula***.

$n = \lambda \mathrm{w}$

Where

n	is the average queue length
λ	is the average arrival rate for new processes (e.g. five a second)
w	is the average waiting time in the queue

Knowing two of these values we can, obviously, calculate the third. For example, if we know that eight processes arrive every second and there are normally sixteen processes in the queue we can compute that the average waiting time per process is two seconds.

The main disadvantage of using queuing models is that it is not always easy to define realistic distribution times and we have to make assumptions. This results in the model only being an approximation of what actually happens.

Simulations

Rather than using queuing models we simulate a computer. A Variable, representing a clock is incremented. At each increment the state of the simulation is updated. Statistics are gathered at each clock tick so that the system performance can be analysed.
The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.
Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation. This can often provide good results and good comparisons over a range of scheduling algorithms.
However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

Implementation

The best way to compare algorithms is to implement them on real machines. This will give the best results but does have a number of disadvantages.

- It is expensive as the algorithm has to be written and then implemented on real hardware.
- If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.
- If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

References

- Courtois P., J., Heymans F. and Parnas D. L. 1971. Concurrent Control with Readers and Writers. Communications of the ACM, Vol. 10, pp. 667-668
- Dijkstra E. W. 1965. Co-operating Sequential Processes. Programming Languages, Genuys, F. (ed), London :Academic Press
- Peterson G., L. 1981. Myths about the Mutual Exclusion Problem. Information Processing Letters, Vol 12, No. 3
- Silberachatz A., Galvin P. 1994. Operating System Concepts (4th Ed). Addison-Wesley Publishing Company
- Tanenbaum, A., S. 1992. Modern Operating Systems (1st ed.). Prentice Hall.
- Tanenbaum, A., S. 2001. Modern Operating Systems (2nd ed.). Prentice Hall.
- Tanenbaum, A., S. 2008. Modern Operating Systems (3rd ed.). Prentice Hall.