UNIT 2 - PROCESS AND THREADS MANAGEMENT

A **process** is a program in execution.

A process is more than the program code, which is sometimes known as the **text** section.
It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data** section, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure below.

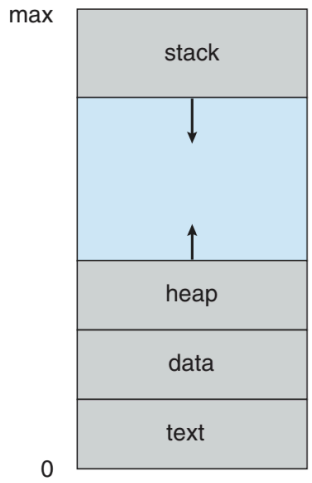


Figure 3.1 Process in memory.

A program is a passive entity, such as a file containing a list of instructions stored on disk
(often called an executable file). In contrast, a process is an active entity,
with a program counter specifying the next instruction to execute and a set
of associated resources. A program becomes a process when an executable file
is loaded into memory.
Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out).
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

• New. The process is being created.

• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

• Terminated. The process has finished execution. These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in the Figure below.

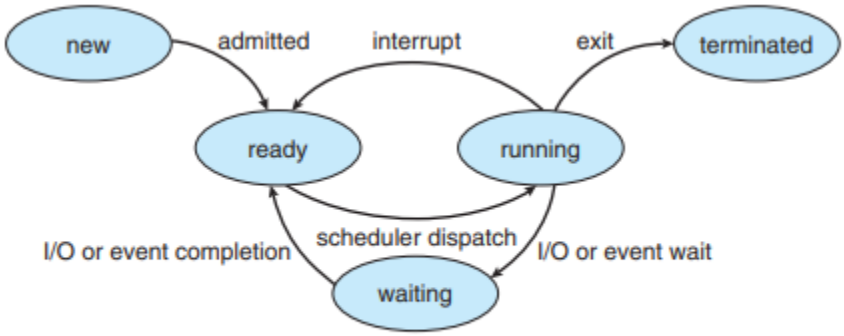


Figure 3.2 Diagram of process state.

Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure It contains many pieces of information associated with a specific process, including these: Process state. The state may be new, ready, running, waiting, halted, and so on.

• **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

• **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward(Figure 3.4).

• **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

• **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
• **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.
In brief, the PCB simply serves as the repository for any information that may vary from process to process.

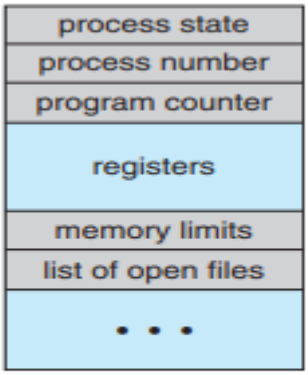


Figure 3.3 Process control block (PCB).

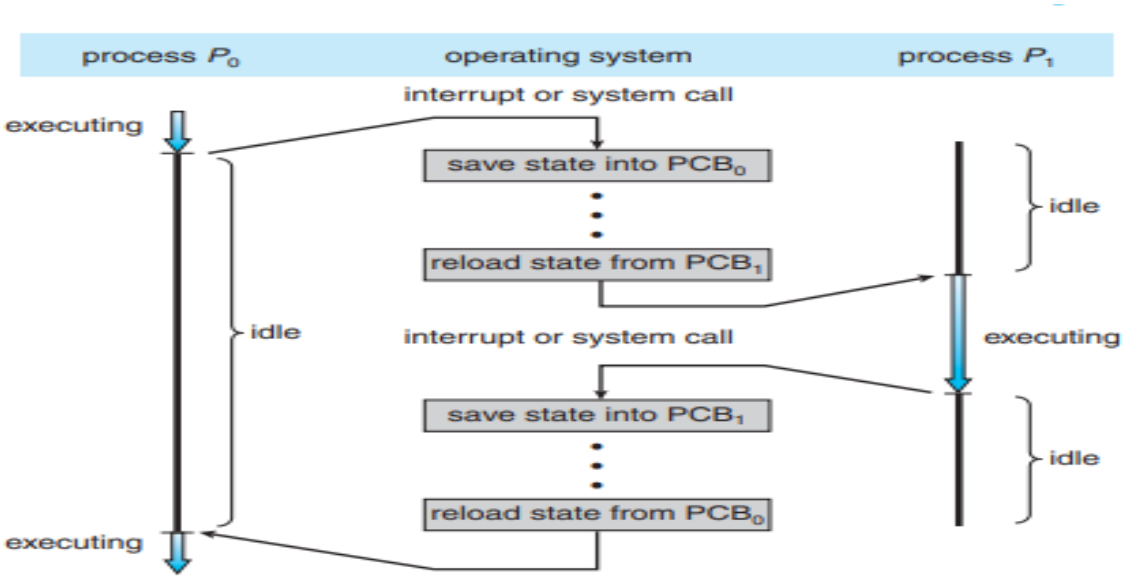


Figure 3.4 Diagram showing CPU switch from process to process.

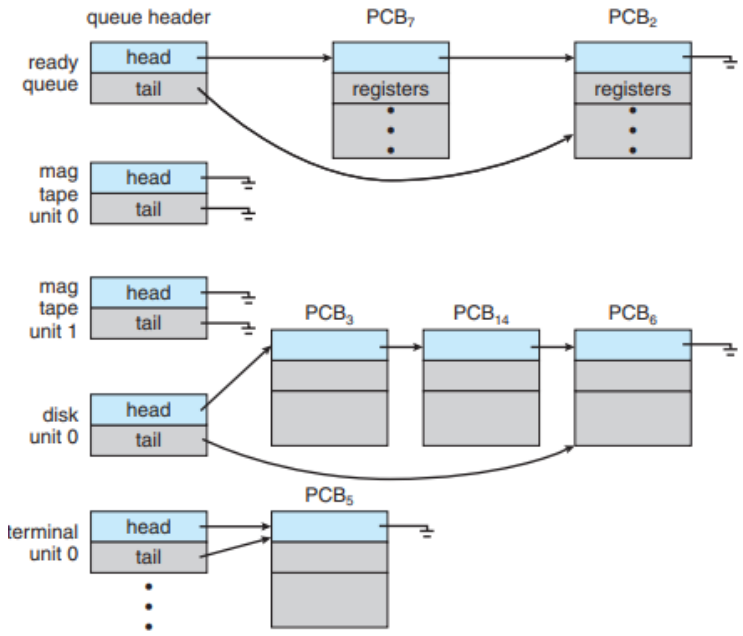
Threads
Modern systems allow a single process to have multiple threads of execution, which execute concurrently.

Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- (Note that these objectives can be conflicting. In particular, every time the system steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work.)

Scheduling Queues

- All processes are stored in the job queue.
- Processes in the Ready state are placed in the ready queue.
- Processes waiting for a device to become available or to deliver data are placed in device queues. There is generally a separate device queue for each device.
- Other queues may also be created and used as needed.



Schedulers

- A long-term scheduler is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The short-term scheduler, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.

- Some systems also employ a medium-term scheduler. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system. See the differences in Figures 3.6 and 3.7 below.
- An efficient scheduling system will select a good process mix of CPU-bound processes and I/O bound processes.

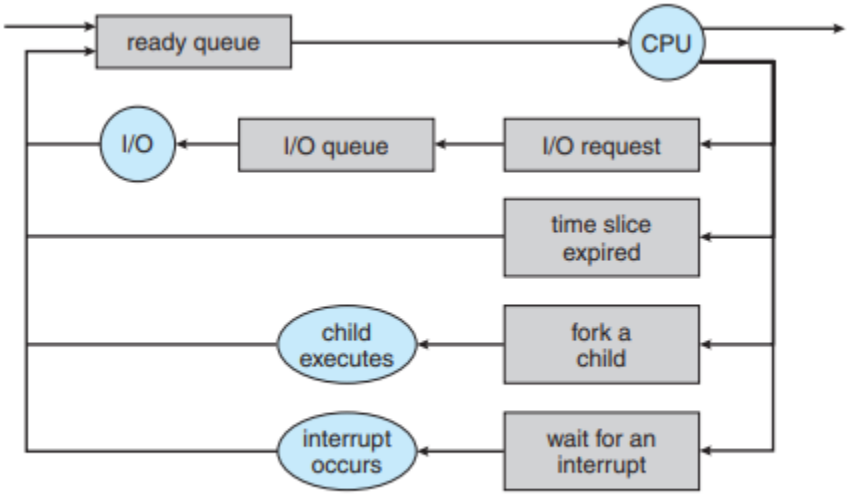


Figure 3.6 Queueing-diagram representation of process scheduling.

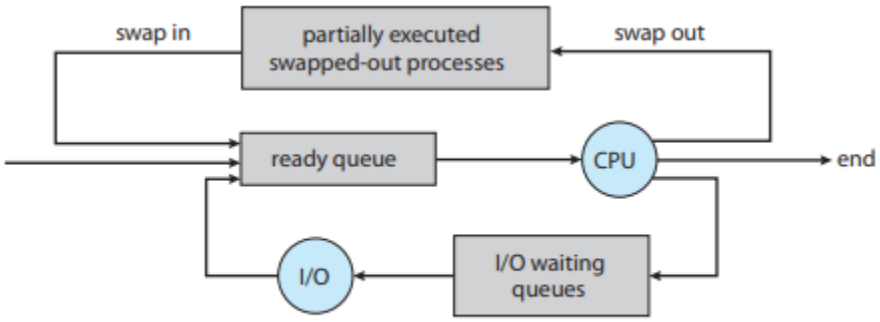


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Context Switch

- Whenever an interrupt arrives, the CPU must do a state-save of the currently running process, then switch into kernel mode to handle the interrupt, and then do a state-restore of the interrupted process.
- Similarly, a context switch occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.

- Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the process control blocks described above.
- Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.
- Some Sun hardware actually has multiple sets of registers, so the context switching can be speeded up by merely switching which set of registers are currently in use. Obviously there is a limit as to how many processes can be switched between in this manner, making it attractive to implement the medium-term scheduler to swap some processes out.

Operations on Processes

Process Creation

- Processes may create other processes through appropriate system calls, such as fork or spawn. The process which does the creating is termed the parent of the other process, which is termed its child.
- Each process is given an integer identifier, termed its process identifier, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed sched, and is given PID 0. The first thing it does at system startup time is to launch init, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.8 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:
- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.
- There are two options for the parent process after creating the child:
    1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
    2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)
- Two possibilities for the address space of the child relative to the parent:

1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the fork system call in UNIX.
2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the spawn system calls in Windows. UNIX systems implement this as a second step, using the exec system call.

- Figures 3.9 and 3.10 below shows the fork and exec process on a UNIX system. Note that the fork system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.

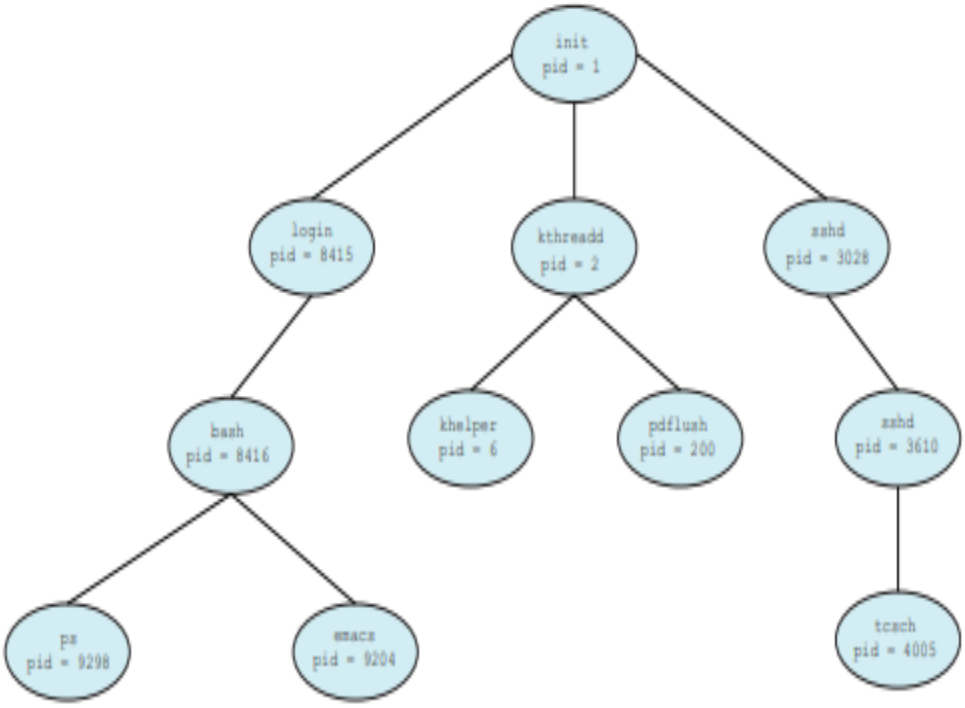


Figure 3.8 A tree of processes on a typical Linux system.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

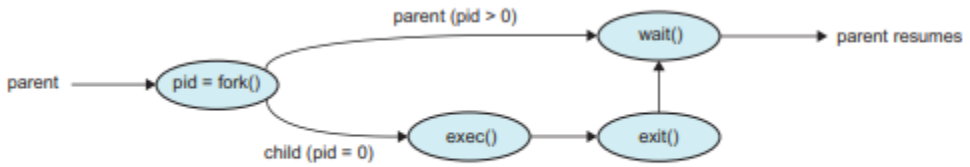


Figure 3.10 Process creation using the `fork()` system call.

Process Termination

Processes may request their own termination by making the exit( ) system call, typically returning an int. This int is passed along to the parent if it is doing a wait( ), and is typically zero on successful completion and some non-zero code in the event of problems.

child code:

int exitCode;

exit(exitCode); // return exitCode; has the same effect when executed from main( )

parent code:

```
pid_t pid;

int status

pid = wait( &status );

// pid indicates which child exited. exitCode in low-order bits of status

// macros can test the high-order bits of status for why it stopped
```

Processes may also be terminated by the system for a variety of reasons, including:

The inability of the system to deliver necessary system resources.

In response to a KILL command, or other un handled process interrupt.

A parent may kill its children if the task assigned to them is no longer needed.

If the parent exits, the system may or may not allow the child to continue without a parent. (On UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them. The UNIX nohup command allows a child to continue executing after its parent has exited.)

When a process terminates, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan. (Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed zombies. These are eventually inherited by init as orphans and killed off. Note that modern UNIX shells do not produce as many orphans and zombies as older systems used to.)

Interprocess Communication

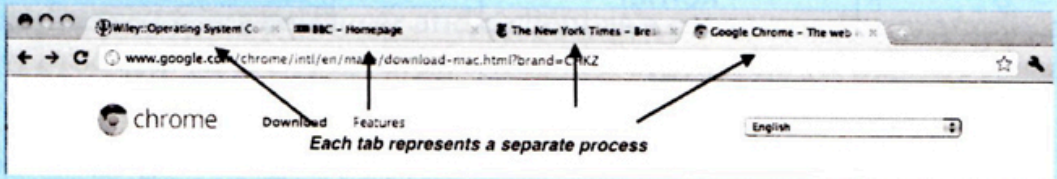
Independent Processes operating concurrently on a system are those that can neither affect other processes or be affected by other processes.

Cooperating Processes are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:

- Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
- Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
- Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
- Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:



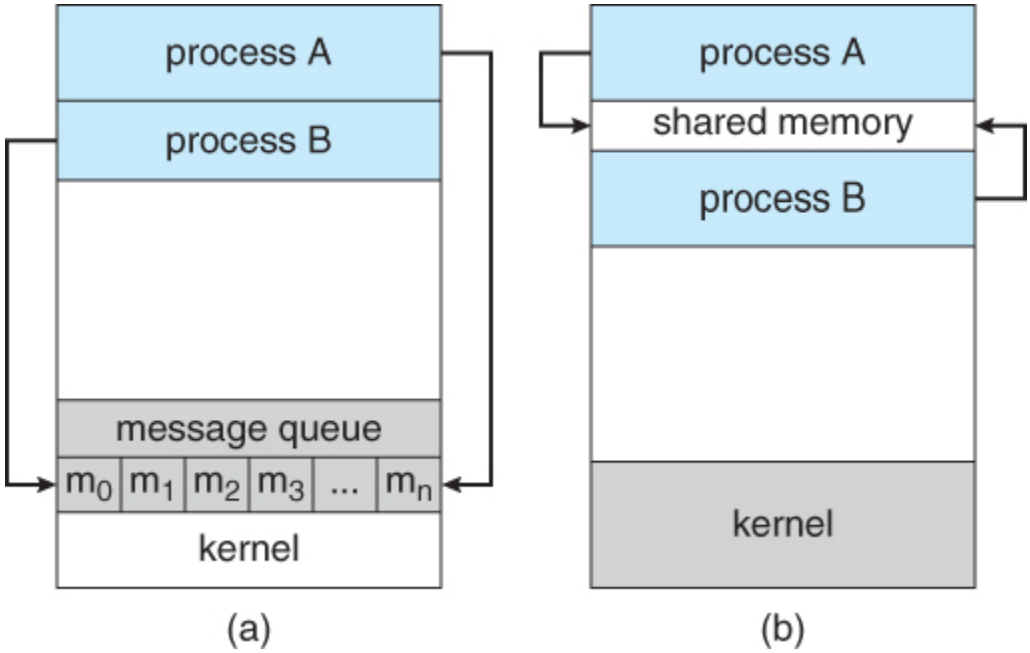
A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites —crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

- Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure below illustrates the difference between the two systems:



- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.

This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.

First the following data is set up in the shared memory area:

#define BUFFER SIZE 10

typedef struct {

. . .

} item;

item buffer[BUFFER SIZE];

int in = 0;

int out = 0;

```
item next_produced;

while (true) {
      /* produce an item in next_produced */

      while (((in + 1) % BUFFER_SIZE) == out)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.12 The producer process using shared memory.

Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

// Code from Figure 3.13

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.13 The consumer process using shared memory.

Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
- Direct or indirect communication (naming)
- Synchronous or asynchronous communication
- Automatic or explicit buffering.

Naming

- With direct communication the sender must know the name of the receiver to which it wishes to send a message.
- There is a one-to-one link between every sender-receiver pair.
- For symmetric communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
- For asymmetric communications, this is not necessary.
- Indirect communication uses shared mailboxes, or ports.
- Multiple processes can share the same mailbox or boxes.
- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
- (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

Synchronization

Either the sending or receiving of messages (or neither or both) may be either blocking or non-blocking.

Buffering

Messages are passed via queues, which may have one of three capacity configurations:

- **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
- **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
- **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

Producer code

```
item nextProduced;

while( true ) {

/* Produce an item and store it in nextProduced */

nextProduced = makeNewItem( . . . );

/* Wait for space to become available */

while( ( ( in + 1 ) % BUFFER_SIZE ) == out )

     ; /* Do nothing */

/* And then store the item and repeat the loop. */

buffer[ in ] = nextProduced;

in = ( in + 1 ) % BUFFER_SIZE;}
```

Consumer code

```
item nextConsumed;

while( true ) {

/* Wait for an item to become available */

while( in == out )

     ; /* Do nothing */

/* Get the next available item */

nextConsumed = buffer[ out ];

out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in nextConsumed

    ( Do something with it ) */

}
```

The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.

We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

Producer Process:

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer Process:

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a race condition. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)
- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part

way through making the update and then the other process butts in, the value of the counter can get left in an incorrect state.

- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

Consumer:

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

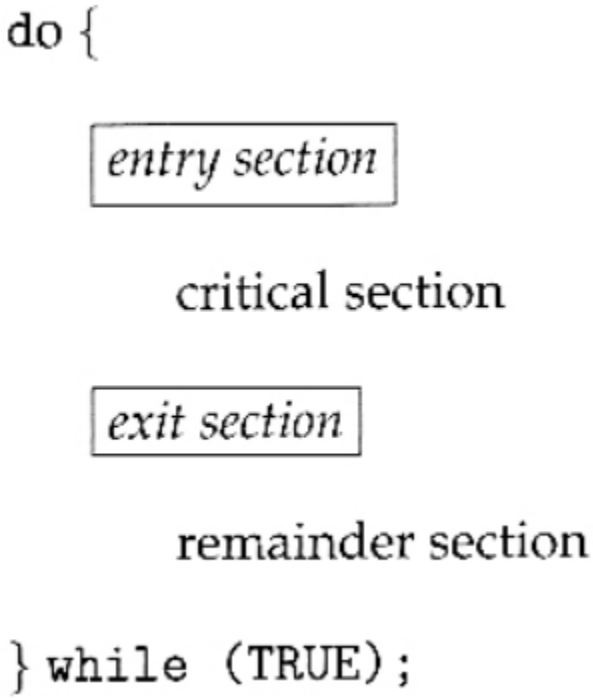
Note that race conditions are notoriously difficult to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-)) Race conditions are also very difficult to reproduce. :-(

Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so let's look at some ways in which this is done, as well as some classic problems in this area.

The Critical-Section Problem

The producer-consumer problem described above is a specific example of a more general situation known as the critical section problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:

- Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
- The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
- The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
- The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.



A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion** - Only one process at a time can be executed in their critical section.
2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the relative speed of one process versus another.

Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:

- Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
- Preemptive kernels allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

Solutions to Critical section problems

1. Two process solution
    1. Strict alternation using boolean variable turn
    2. Using boolean array flag
    3. Peterson’s solution

2. Operating system solution
    1. Counting semaphore
    2. Binary Semaphore

3.Hardware solution
    1. Test and set lock
    2. Disable interrupt

// Shared variable

boolean turn = 0;   // 0 = P0's turn, 1 = P1's turn

Strict Alternation (using boolean turn)

Idea:

A single shared variable turn decides whose turn it is to enter the critical section (CS).

- If turn = 0, process P0 can enter CS.
- If turn = 1, process P1 can enter CS.

After leaving CS, the process changes turn to the other process.

```
// Process P0
while (true) {
    while (turn != 0) {
        // busy wait
    }

    // ---- Critical Section ----

    turn = 1;   // give turn to P1

    // ---- Remainder Section ----
}

// Process P1
while (true) {
    while (turn != 1) {
        // busy wait
    }

    // ---- Critical Section ----

    turn = 0;   // give turn to P0

    // ---- Remainder Section ----
}
```

Explanation of Working

Initialization: Assume turn = 0 (so P0 goes first).
P0 Execution:

P0 finds turn == 0, enters CS. After finishing, P0 sets turn = 1.

P1 Execution:

Now P1 finds turn == 1, enters CS. After finishing, P1 sets turn = 0.

Repeat: They alternate strictly, one after the other.

- Condition Analysis
- Mutual Exclusion ✅
- Only one process can enter CS at a time because turn allows only one process.
- Bounded Waiting ✅
- Each process eventually gets its turn (alternates).
- Progress ❌ (Fails)
- If P0 is in its remainder section (not interested in CS), but turn = 0, then P1 must keep waiting unnecessarily until P0 re-enters CS and then leaves it, setting turn = 1.
  → This is inefficient and wastes CPU time.

  Example of Failure

  Suppose:
  turn = 0.
  P0 is in remainder (not using CS).
  P1 wants to enter CS.
  But since turn = 0, P1 will keep busy waiting (blocked), even though P0 is not competing.

So, Strict Alternation is important as a teaching example because:
It achieves mutual exclusion, but fails the progress requirement, which is why better algorithms like Peterson's solution were developed.

Using Boolean Array flag[]

Here, each process has its own flag[i] that indicates whether it wants to enter the **critical section (CS)**.

Naïve Flag Solution (Fails Mutual Exclusion ❌)

```
// Shared variable
boolean flag[2] = {false, false};   // both initially not interested

// Process P0
while (true) {
   flag[0] = true;   // express desire to enter CS

   while (flag[1]) {
```

```
        // busy wait if P1 also wants CS
    }
    // ---- Critical Section ----

    flag[0] = false;  // done with CS

    // ---- Remainder Section ----
}
// Process P1
while (true) {
    flag[1] = true;   // express desire

    while (flag[0]) {
        // busy wait if P0 also wants CS
    }

    // ---- Critical Section ----

    flag[1] = false;

    // ---- Remainder Section ----
}
```

Problem

- If **both processes set their flag to true simultaneously**, they will both wait forever → **deadlock** (neither can proceed).
- This violates **progress** and may cause **starvation**.

- **Improvement – Peterson's Solution (Flag + Turn ✅)**

Peterson combined **flag[]** with an extra variable **turn** to solve the problem:

```
// Shared variables
boolean flag[2] = {false, false};
int turn;
// Process Pi (i = 0 or 1)
while (true) {
    flag[i] = true;         // express interest
    turn = 1 - i;           // give priority to the other process

    while (flag[1 - i] && turn == 1 - i) {
        // busy wait
    }

    // ---- Critical Section ----
```

```
    flag[i] = false;        // done with CS

    // ---- Remainder Section ----
}
```

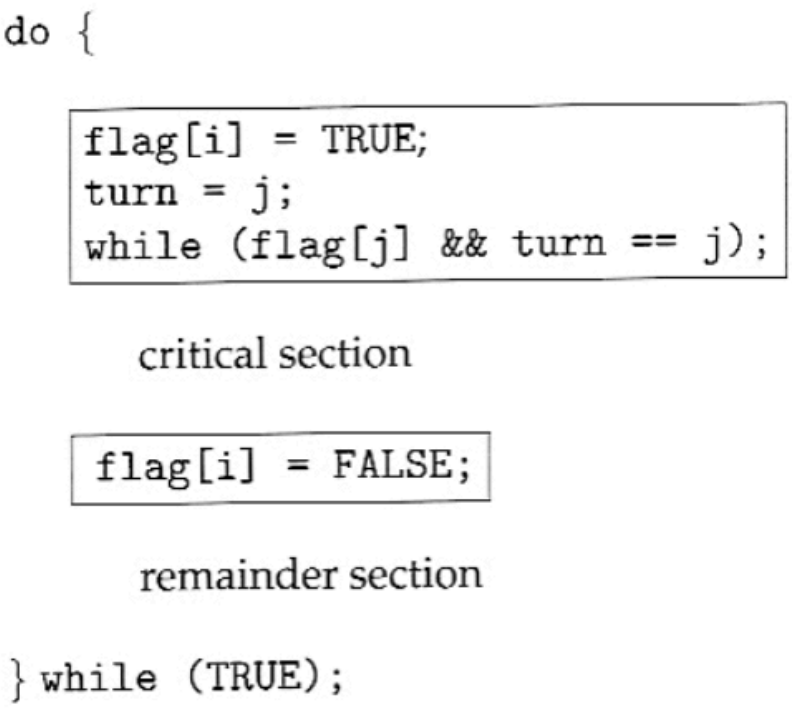
Condition Analysis

- **Naïve Flag Only**
    - Mutual Exclusion ❌ (fails if both set flag simultaneously).
    - Progress ❌ (deadlock possible).
    - Bounded Waiting ❌.
- **Peterson's (Flag + Turn)**
  Mutual Exclusion ✅.
    - Progress ✅.
    - Bounded Waiting ✅.

👉 So, flag[] **alone is not enough** — you need the **turn variable** to break the tie.

Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:
- int turn - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
- boolean flag[2] - Indicates when a process wants to enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.
- In the following diagram, the entry and exit sections are enclosed in boxes.
- In the entry section, process i first raises a flag indicating a desire to enter the critical section.
- Then turn is set to j to allow the other process to enter their critical section if process j so desires.
- The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
- Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.



To prove that the solution is correct, we must examine the three conditions listed above:

1. Mutual exclusion - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
2. Progress - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
3. Bounded Waiting - As each process enters their entry section, they set the turn variable to be the other process's turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

Note that the instruction "turn = j" is atomic, that is it is a single machine instruction which cannot be interrupted

Dekker's Algorithm

the first known correct software solution (for two processes) to the critical section problem. It was proposed by Dutch mathematician Th. J. Dekker in the 1960s.

```
boolean flag[2] = {false, false};   // flag[i] = true means Pi wants to enter CS
int turn = 0;                       // whose turn it is (0 or 1)


while (true) {
    flag[0] = true;                 // P0 wants to enter CS
    while (flag[1]) {               // if P1 also wants CS
        if (turn != 0) {            // but not P0's turn
            flag[0] = false;        // back off
            while (turn != 0);      // wait until turn becomes 0
            flag[0] = true;         // re-enter competition
        }
    }
    // ---- Critical Section ----
    turn = 1;                       // give turn to P1
    flag[0] = false;                // reset interest
    // ---- Remainder Section ----
}
```


```
boolean flag[2] = {false, false};   // flag[i] = true means Pi wants to enter CS
int turn = 0;                       // whose turn it is (0 or 1)
```


Process P0

```
while (true) {
    flag[0] = true;                 // P0 wants to enter CS
```

```
    while (flag[1]) {               // if P1 also wants CS
        if (turn != 0) {            // but not P0's turn
            flag[0] = false;        // back off
            while (turn != 0);      // wait until turn becomes 0
            flag[0] = true;         // re-enter competition
        }
    }

    // ---- Critical Section ----

    turn = 1;                       // give turn to P1
    flag[0] = false;                // reset interest

    // ---- Remainder Section ----
}
```

```
Process P1
while (true) {
    flag[1] = true;                 // P1 wants to enter CS
    while (flag[0]) {               // if P0 also wants CS
        if (turn != 1) {            // but not P1's turn
            flag[1] = false;        // back off
            while (turn != 1);      // wait until turn becomes 1
            flag[1] = true;         // re-enter competition
```

```
        }
    }

    // ---- Critical Section ----

    turn = 0;                    // give turn to P0
    flag[1] = false;             // reset interest

    // ---- Remainder Section ----
}
```

How It Works

1. Each process raises its **flag[i] = true** when it wants to enter CS.
2. If both raise flags:
    - The one whose **turn** it is **waits**, while the other backs off.
    - This avoids deadlock.
3. After leaving CS, a process:
    - Sets **turn = other process**.
    - Resets its own **flag[i] = false**.

Condition Check

- **Mutual Exclusion** ✅
  At most one process can enter CS at a time.
- **Progress** ✅
  If only one process wants CS, it enters immediately.
- **Bounded Waiting** ✅
  The turn variable ensures fairness (no starvation).

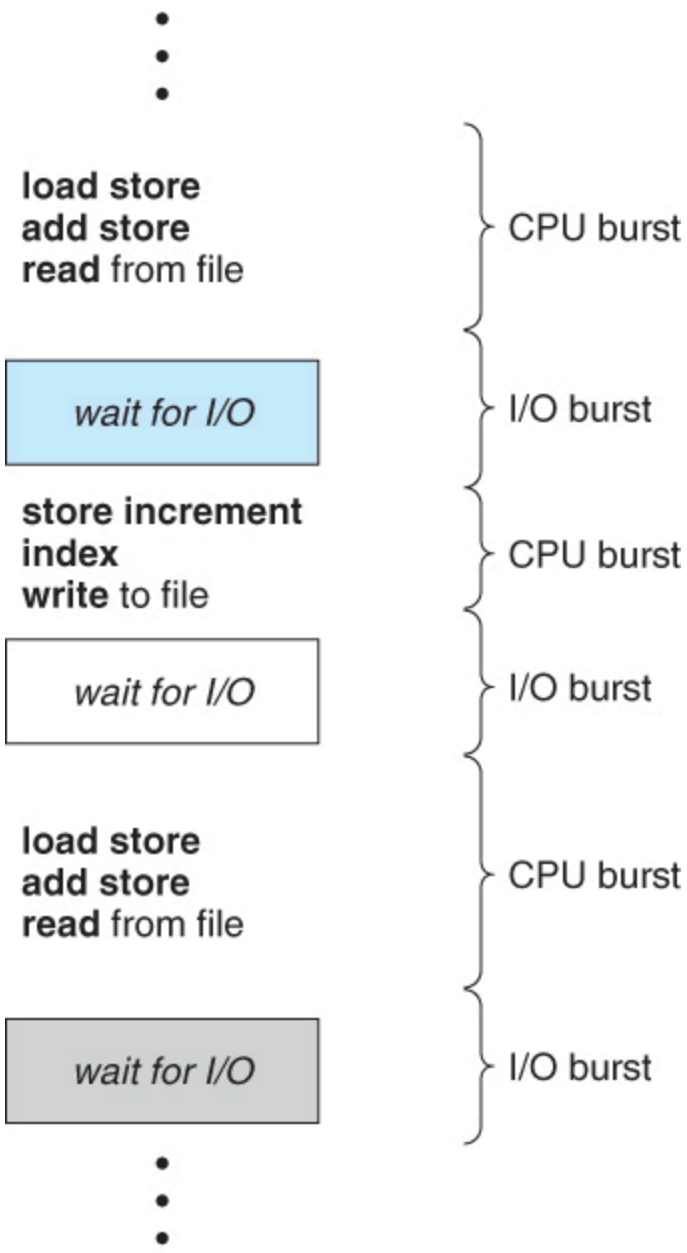
CPU Scheduling

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

6.1.1 CPU-I/O Burst Cycle

Almost all processes alternate between two states in a continuing cycle, as shown in Figure 6.1 below :

- A CPU burst of performing calculations, and
- An I/O burst, waiting for data transfer in or out of the system.



CPU bursts vary from process to process, and from program to program.

CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm, which is the basic subject of this entire chapter.

Preemptive Scheduling

CPU scheduling decisions take place under one of four conditions:

1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
2. When a process switches from the running state to the ready state, for example in response to an interrupt.
3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
4. When a process terminates.

- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be non-preemptive, or cooperative. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be preemptive.
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Chapter 5 examined this issue in greater detail.
- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIXes deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- Some critical sections of code protect themselves from con currency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces

of code that will finish quickly, (usually just a few machine instructions.)

Dispatcher

- The dispatcher is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
    - Switching context.
    - Switching to user mode.
    - Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as dispatch latency.

Scheduling Criteria

- There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:
    - CPU utilization - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
    - Throughput - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
    - Turnaround time - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
    - Waiting time - How much time processes spend in the ready queue waiting their turn to get on the CPU.
    - (Load average - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
    - Response time - The time taken in an interactive program from the issuance of a command to the commence of a response to that command.
- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as

to minimize the maximum response time.
- Sometimes it is more desirable to minimize the variance of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

Scheduling Algorithms

The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

First-Come First-Serve Scheduling, FCFS

FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time.

For example, consider the following three processes:

Process	Burst Time
P_1	24
P_2	3
P_3	3

In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.

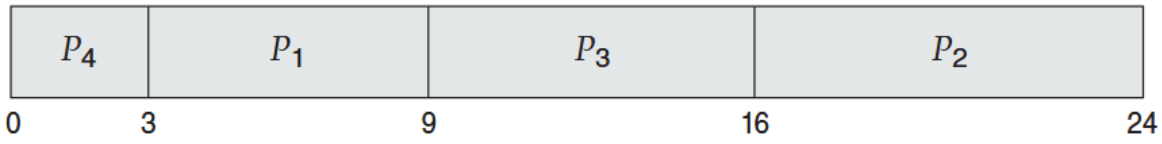
In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.

FCFS can also block the system in a busy dynamic system in another way, known as the convoy effect. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Shortest-Job-First Scheduling, SJF

The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.(Technically this algorithm picks a process based on the next shortest CPU burst, not the overall process time.)For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)

SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?

For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.

Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.

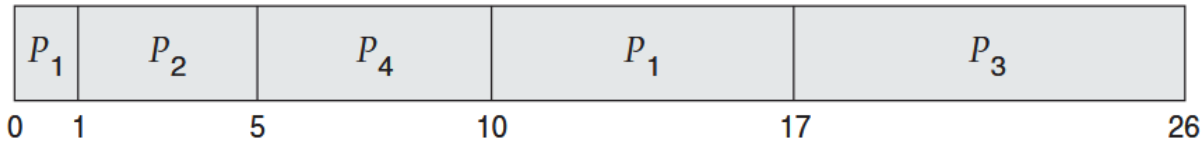
A more practical approach is to predict the length of the next burst, based on some historical measurement of recent burst times for this process. One simple, fast, and relatively accurate method is the exponential average, which can be defined as follows. (The book uses tau and t for their variables, but those are hard to distinguish from one another and don't work well in HTML.)

estimate[i + 1] = alpha * burst[i] + (1.0 - alpha) * estimate[i]

SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as shortest remaining time first scheduling.

For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

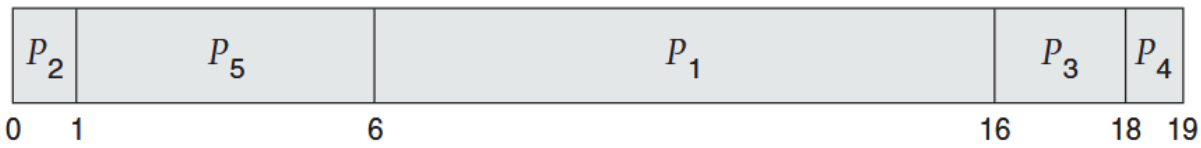
Priority Scheduling

Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)

Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.

For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.

Priority scheduling can be either preemptive or non-preemptive.

Priority scheduling can suffer from a major problem known as indefinite blocking, or starvation, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.

If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)

One common solution to this problem is aging, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Round Robin Scheduling

Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called time quantum.

When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.

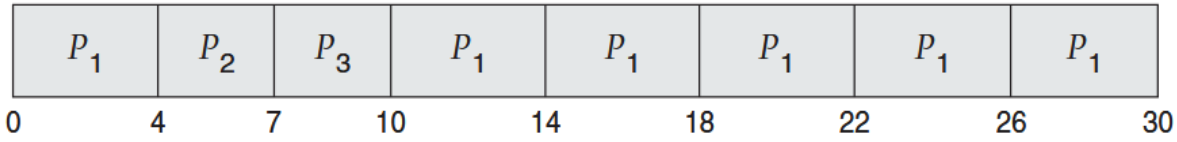
If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.

If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.

The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.

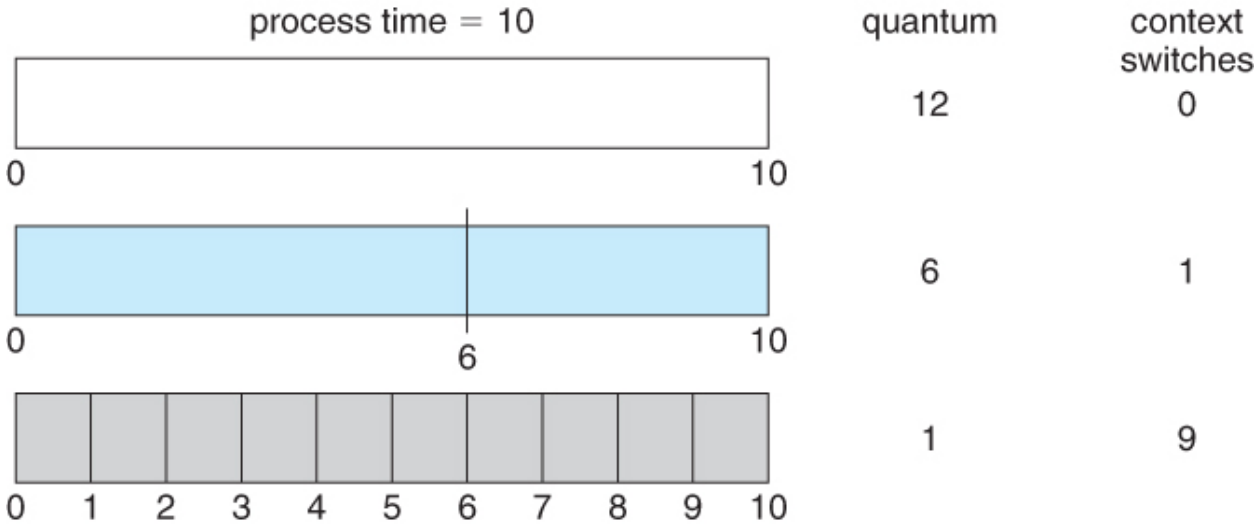
RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms.

Process	Burst Time
P_1	24
P_2	3
P_3	3



The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.

BUT, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. (See Figure 6.4 below.) Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.



In general, turnaround time is minimized if most processes finish their next cpu burst within one quantum time. For example, with three processes of 10 ms bursts each, the average turnaround time for 1 ms quantum is 29, and for 10 ms quantum it reduces to 20. However, if it is made too large, then RR just degenerates to FCFS. A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum.

Multilevel Queue Scheduling

When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.

Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)

Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

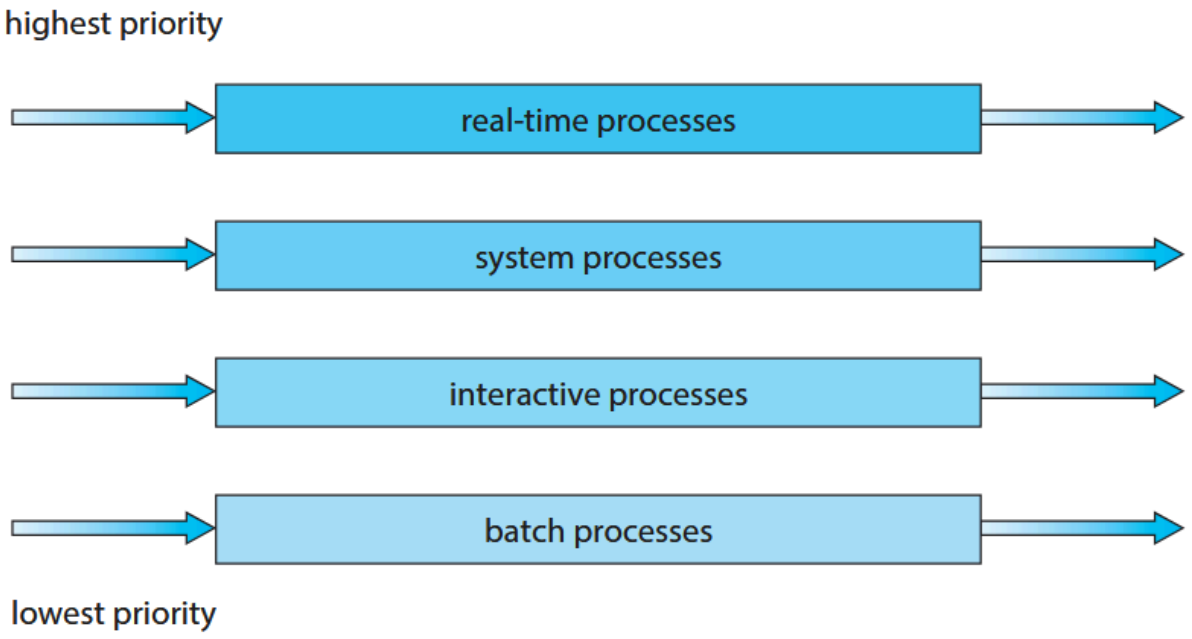


Figure 5.8 Multilevel queue scheduling.

Multilevel Feedback-Queue Scheduling

Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:

If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.

Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.

Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:

The number of queues.

The scheduling algorithm for each queue.

The methods used to upgrade or demote processes from one queue to another. (Which may be different.)

The method used to determine which queue a process enters initially.

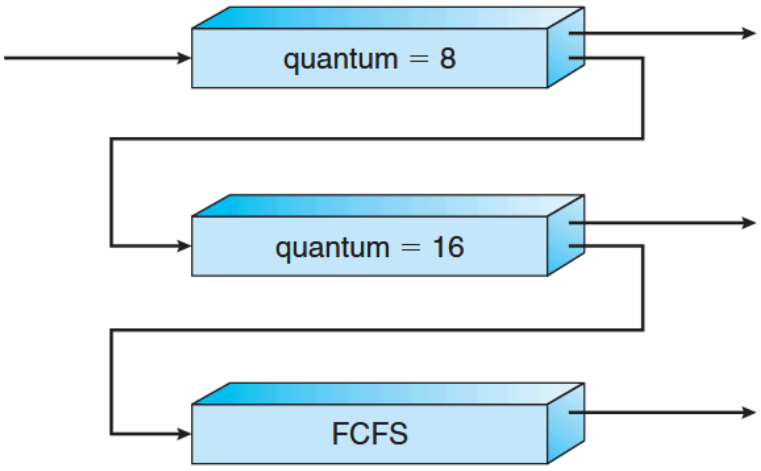


Figure 5.9 Multilevel feedback queues.