



Operating Systems Cheat Sheet

Operating Systems (City University of Hong Kong)

Operating Systems Cheat Sheet

1 COMMON DEFINITIONS

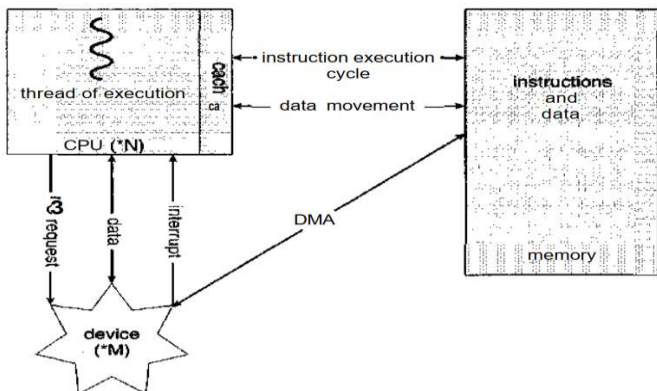
TERM	DEFINITION
WINDOWS OPERATING SYSTEM ENVIRONMENT	Windows systems. The primary programming environment for Windows systems is the Win32 API (application programming interface)
POSIX	POSIX (which stands for <i>Portable Operating System Interface</i>) represents a set of standards implemented primarily for UNIX-based operating systems
JAVA VIRTUAL MACHINE	Java is a widely used programming language with a rich API and built-in language support for thread creation and management. Java programs run on any operating system supporting a Java virtual machine (or JVM).
OPERATING SYSTEM	A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being systems programs and application programs
INTERRUPT <i>Except</i> Exits a process, with an interrupt msg	The occurrence of an event is usually signalled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a system call (also called a monitor call)
BOOTSTRAP PROGRAM	For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or bootstrap program , tends to be simple. They are written in ROM, known as firmware
FAULT TOLERANCE AND GRACEFUL DEGRADATION	Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation . Some systems go beyond graceful degradation and are called fault tolerant , because they can suffer a failure of any single component and still continue operation.
MODERN COMPUTER	 <p>The diagram illustrates the components and data flow of a modern computer system. It shows a CPU with an internal cache and a main memory. A device is connected to the CPU via a bus. Data flows from the device to the CPU cache, then to the main memory, and back to the device. The CPU also interacts with the cache and memory for instruction execution and data movement. DMA (Direct Memory Access) is shown as a path for data movement between the device and memory.</p>
MASTER SLAVE PROCESSING	Some systems use asymmetric multiprocessing , in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.
SYMMETRIC MULTIPROCESSING	symmetric multiprocessing (SMP) , in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master-slave relationship exists between processors.
BLADE SERVERS	Lastly, blade servers are a recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis.
CLUSTERED SYSTEM	Another type of multiple-CPU system is the clustered system . Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work and in that they are composed of two or more individual systems coupled together. Eg LAN

Figure 1.5 How a modern computer system works.

Process -> usme usually memory is not shared.

Thread-> its shares the memory occupeid by the process

DISTRIBUTED LOCK MANAGER

PROCESS

SWAPPING, VIRTUAL MEMORY

Sometimes, a process is used very often when it stops, computer stores it into swap memory. Why? loading is easier

TRAP

DUAL MODE OPERATION

FILE

CACHING

Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**.

Time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a **process**.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through **swapping**, where processes are swapped in and out of main ^{RAM} memory to the disk. A more common method for achieving this goal is **virtual memory**, a technique that allows the execution of a process that is not completely in memory.

The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory

A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. Thus, we need two separate **modes** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).

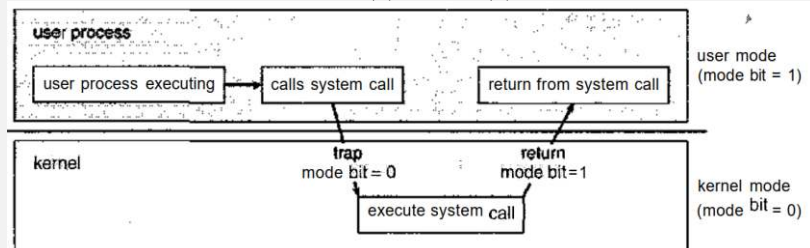


Figure 1.8 Transition from user to kernel mode.

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media and accesses these files via the storage devices.

Caching is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache; if it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon

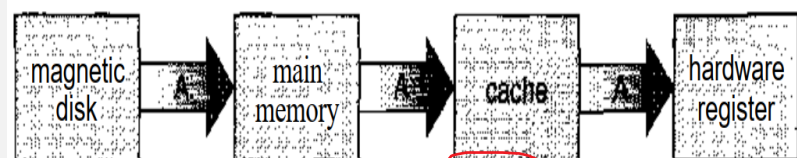


Figure 1.10 Migration of integer A from disk to register.

PERFORMANCE OF STORAGE LEVEL

Level	1	2	3	4
Name	registers	cache	ram	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5000 - 10,000	1000 - 5000	20 - 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Figure 1.9 Performance of various levels of storage.

CACHE COHERENCY

Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware problem

PROTECTION

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system

COMPUTE SERVER SYSTEM

The compute-server system provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client.

2 OPERATING SYSTEM STRUCTURES

2.1 DEFINITION

TERM	DEFINITION
SYSTEM CALLS	System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.
APPLICATION PROGRAMMING INTERFACE (API)	The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Win32 API for Windows systems, the POSIX API for POSIX-based systems (which includes virtually all versions of UNIX, Linux, and Mac OS X), and the Java API for designing programs that run on the Java virtual machine
POLICY AND MECHANISM	Mechanisms determine <i>how</i> to do something; policies determine <i>what</i> will be done
VIRTUAL MACHINE	The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

All the features in VM is same as that of original OS

This document is available free of charge on



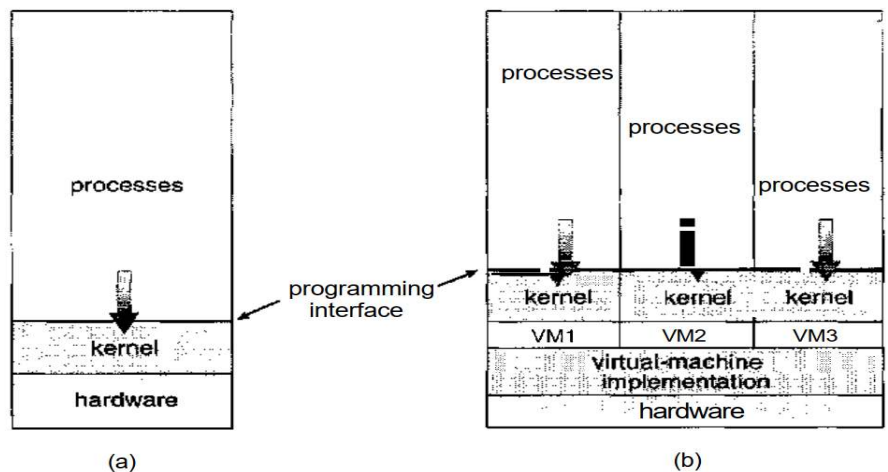


Figure 2.15 System models. (a) Nonvirtual machine. (b) Virtual machine.

VIRTUAL DISKS

Physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine, because the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks—termed *minidisks* in IBM's VM operating system—that are identical in all respects except size. The system implements each minidisk by allocating as many tracks on the physical disks as the minidisk needs. Obviously, the sum of the sizes of all minidisks must be smaller than the size of the physical disk space available.

GARBAGE COLLECTION

The practice of reclaiming memory from objects no longer in use and returning it to the system.

JUST IN TIME COMPILER

The bytecodes for the method are turned into native machine language for the host system

SYSTEM GENERATION

Operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation** (SYSGEN).

BOOTSTRAP PROGRAM

On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution

2.2 SUB-ROUTINES

SUB-ROUTINE

WORK

EXECUTION OF RM FILENAME.TXT

The UNIX command to delete a file

```
rm file.txt
```

would search for a file called rm, load the file into memory, and execute it with the parameter file. txt. The function associated with the rm command would be defined completely by the code in the file rm

OPEN FILE

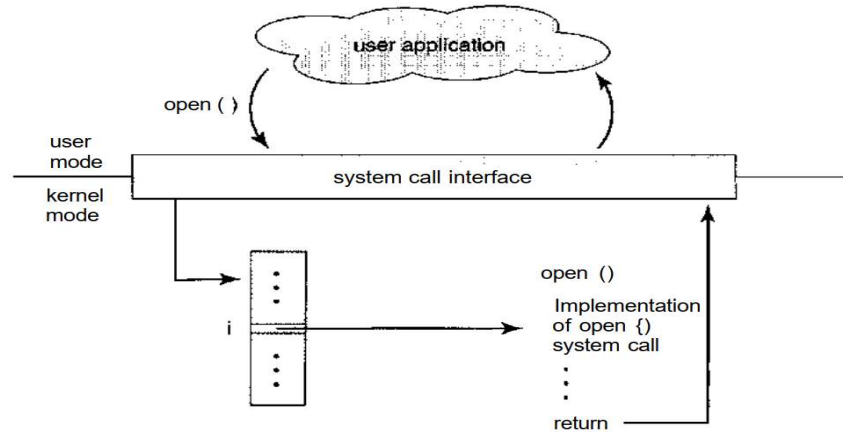


Figure 2.3 The handling of a user application invoking the open() system call.

PARAMETERS PASSING

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in *registers*. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block*, or table, in memory, and the address of the block is passed as a parameter in a register. Parameters also can be placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system.

MS DOS EXECUTION

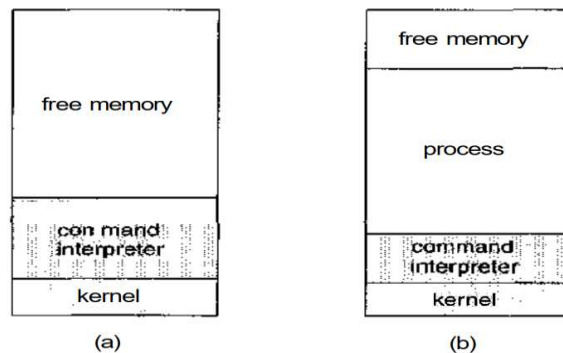


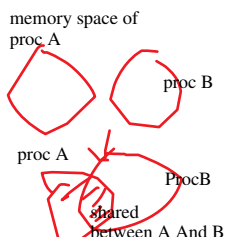
Figure 2.7 MS-DOS execution. (a) At system startup. (b) Running a program.

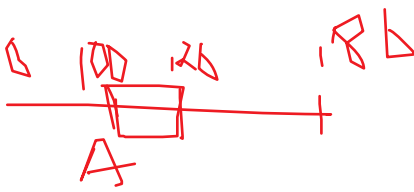
MULTITASKING SYSTEM

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs.

MESSAGE PASSING

There are two common models of interprocess communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name*, and this





name is translated into an identifier by which the operating system can refer to the process. The `get host id` and `get processid` system calls do this translation. The identifiers are then passed to the general purpose open and close calls provided by the file system or to specific open connection and close connection system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept connection` call. Most processes that will be receiving connections are special-purpose *daemons*, which are systems programs provided for that purpose. They execute a wait for connection call and are awakened when a connection is made. The source of the communication, known as the *client*, and the receiving daemon, known as a *server*, then exchange messages by using read message and write message system calls. The close connection call terminates the communication.

SHARED MEMORY PASSING

In the shared-memory model, processes use shared memory create and shared memory attach system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by the processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

VIRTUAL MACHINE KERNAL MODE COMMANDS

When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

JAVA VIRTUAL MACHINE

- 1) Java objects are specified with the class construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (.class) file that will run on any implementation of the JVM.
- 2) The class loader loads the compiled . class files from both the Java program and the Java API for execution by the Java interpreter
- 3) After a class is loaded, the verifier checks that the . class file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access.

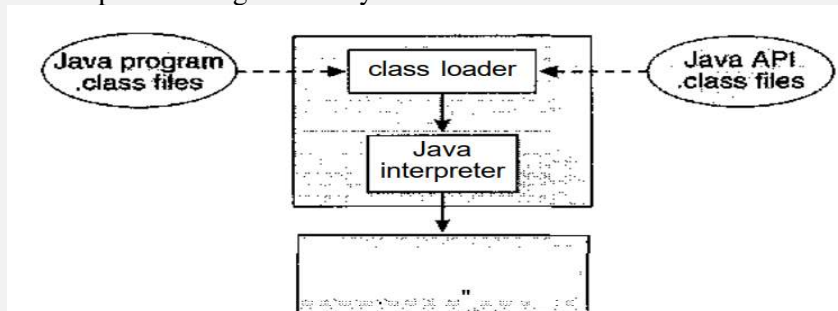


Figure 2.17 The Java virtual machine.

.NET COMMON LANGUAGE RUNTIME

At the core of the .NET Framework is the Common Language Runtime (CLR). Programs written in languages such as C# (pronounced *C-sharp*)

and VB.NET are compiled into an intermediate, architecture-independent language called iMicrosoft Intermediate Language (MS-IL). These compiled files, called assemblies, include MS-IL instructions and metadata. They have a file extension of either .EXE or .DLL. Upon execution of a program, the CLR loads assemblies into what is known as the **Application Domain**. As instructions are requested by the executing program, the CLR converts the MS-IL instructions inside the assemblies into native code that is specific to the underlying architecture using just-in-time compilation.

SYSTEM BOOT

BOOT BLOCK

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

The bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**.

3 CHAPTER 3

3.1 DEFINITION

TERM

BATCH SYSTEM AND TIME-SHARED SYSTEM PROCESS

DEFINITION

A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*

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.

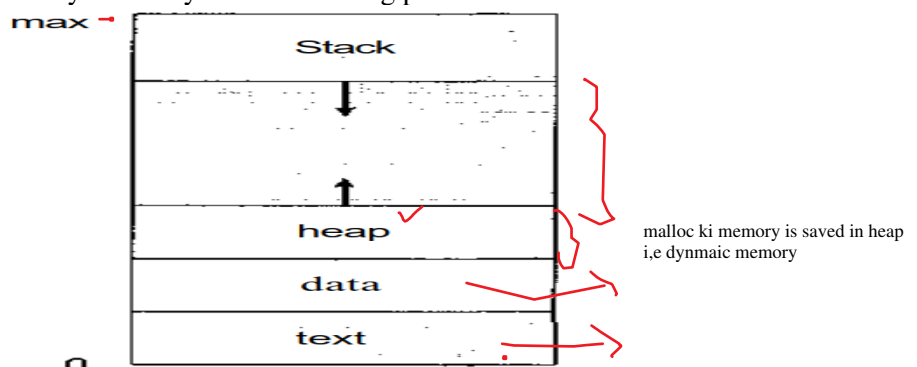


Figure 3.1 Process in memory.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**), whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources.

- **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.

PROGRAM AND PROCESS AND EXECUTABLE FILE

PROCESS STATES

New proc created
ran the first segment
had to stop to take input

Only one process can be *running* on any processor at any instant.

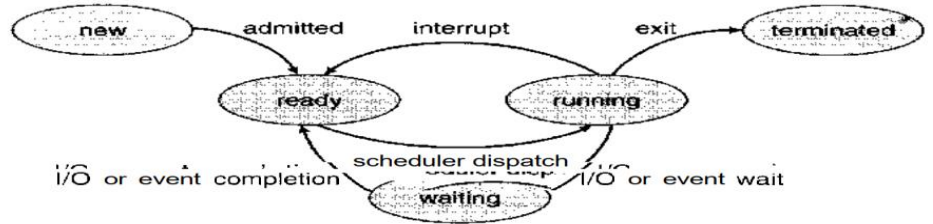


Figure 3.2 Diagram of process state.

PROCESS CONTROL BLOCK OR CONTEXT

- **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 state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **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 information as the value of the base and limit registers, 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 members, 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.

SHORT AND LONG TERM SCHEDULER

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them. Long term schedulers doesn't exist in time-sharing system such as UNIX or windows instead medium-term schedulers exist

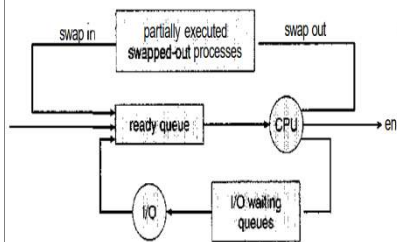


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

SWAPPING

It can be advantageous to remove processes from memory (and from active contention for the CPU) to reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

INDEPENDENT AND COPERATING PROCESS

A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by the other processes executing in the system

BUFFERING

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length n ; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity.** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

SOCKET

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number

3.2 SUB-ROUTINES

SUB-ROUTINE

WORK

CPU PROCESS EXECUTION

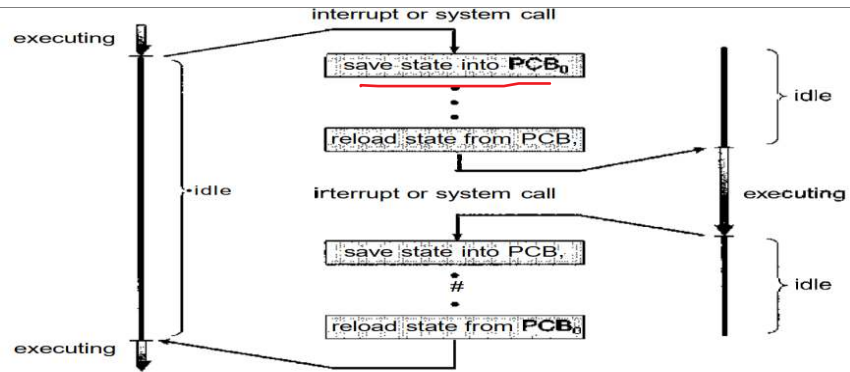
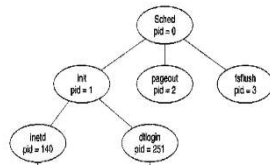


Figure 3.4 Diagram showing CPU switch from process to process.

PROCESS CREATION

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes. Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique **process identifier (or pid)**



When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses

WINDOWS XP COMMUNICATION

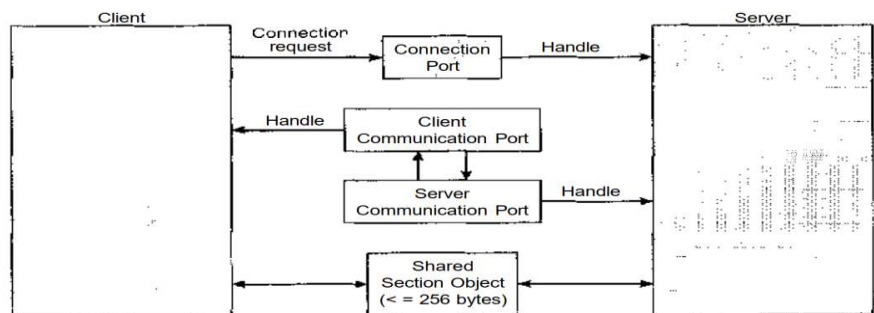


Figure 3.17 Local procedure calls in Windows XP.

CLIENT SERVER SOCKET COMMUNICATION

When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to

establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80)

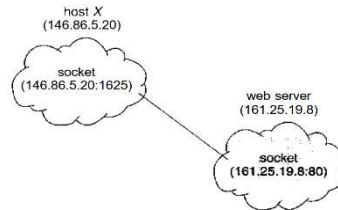


Figure 3.18 Communication using sockets.

on the web server.

REMOTE PROCEDURE CALLS(RPC)

Teamviewer

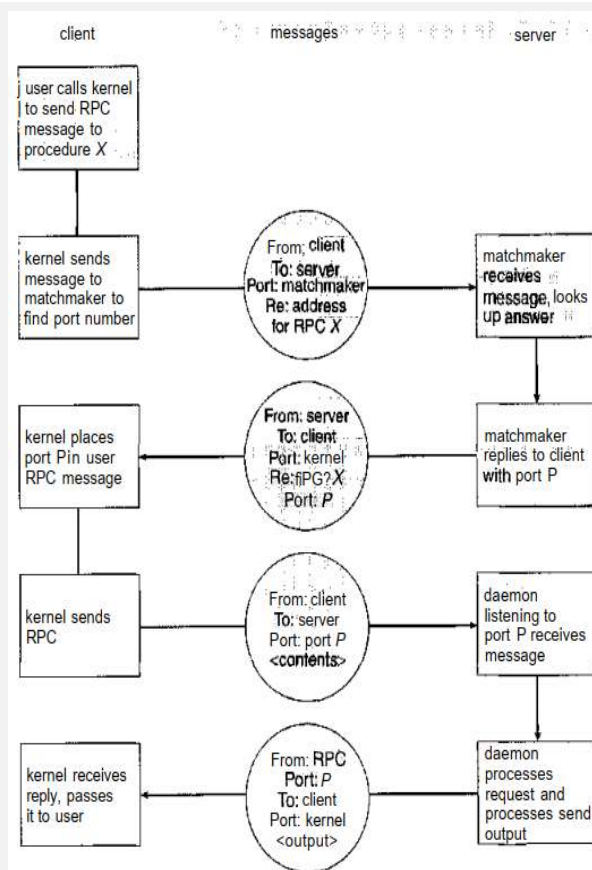


Figure 3.21 Execution of a remote procedure call (RPC).

Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

The RPC system hides the details that allow communication to take place by providing a **stub** on the client side. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and **marshals** the parameters. Parameter marshalling involves packaging the parameters into a form that

can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

3.3 PROCESS SCHEDULING

- 1) As processes enter the system, they are put into a **job queue**, which consists of all processes in the system
- 2) The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. Stored as linked list
- 3) When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

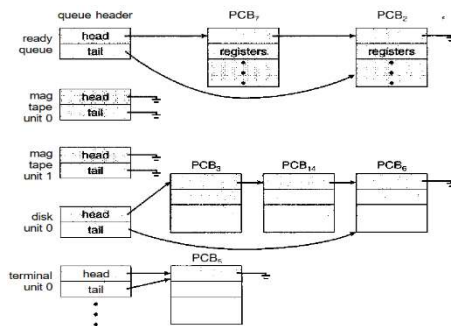


Figure 3.6 The ready queue and various I/O device queues.

4) Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device **queue**. Each device has its own device queue.

5) A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.

- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.

3.4 PROCESS SPLIT

3.4.1 Execution Difference

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

3.4.2 Space Difference

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

3.4.3 Fork

A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: The return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent

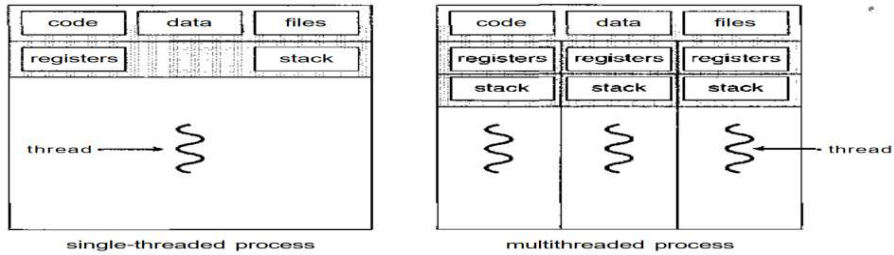
3.4.4 Create Process

Windows Processes are created in the Win32 API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

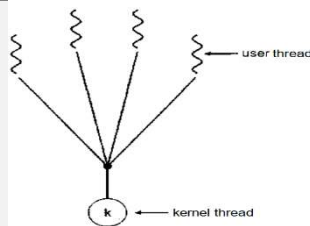
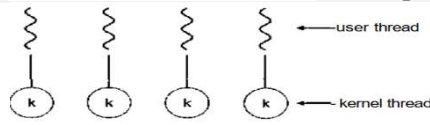
4 THREADING

4.1 DEFINITION

TERM	DEFINATION
THREADING	A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

	 <p style="text-align: center;">single-threaded process multithreaded process</p> <p style="text-align: center;">Figure 4.1 Single-threaded and multithreaded processes.</p>
THREAD CANCELLATION	<p>A thread that is to be cancelled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:</p> <ol style="list-style-type: none"> 1. Asynchronous cancellation. One thread immediately terminates the target thread. 2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
USER DEFINED HANDLERS	<p>Every signal has a default signal handler that is run by the kernel when handling that signal. This default action can be overridden by a user-defined signal handler that is called to handle the signal. Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program</p>
THREAD POOLING	<p>Idea behind a thread pool is to create a number of threads at process start-up and place them into a <i>pool</i>, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request to service. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.</p>
LWP(LIGHT WEIGHT PROCESSOR)	<p>To the user-thread library, the LWP appears to be a <i>virtual processor</i> on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks</p>
SCHEDULER ACTIVATIONS	<p>Scheme for communication between the user-thread library and the kernel is known as scheduler activation. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.</p>
UPCALL	<p>The kernel must inform an application about certain events. This procedure is known as an upcall. Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor. One event that triggers an upcall occurs when an application thread is about to block</p>

4.2 DIFFERENT MODELS OF THREADING

MODEL	VIEW
MANY-TO-ONE	 <p style="text-align: center;">Figure 4.2 Many-to-one model.</p> <p>Many-to-one model (Figure 4.2) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors</p>
ONE-TO-ONE	 <p style="text-align: center;">Figure 4.3 One-to-one model.</p> <p>The one-to-one model (Figure 4.3) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The overhead of creating kernel threads can burden the performance of an application</p>

MANY-TO-MANY

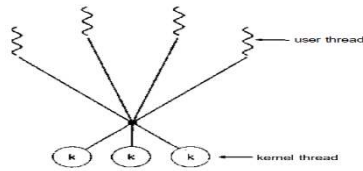


Figure 4.4 Many-to-many model.

The many-to-many model (Figure 4.4) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine

COMPARISON BETWEEN ALL

Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

4.3 SIGNAL HANDLING

4.3.1 Approach

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled

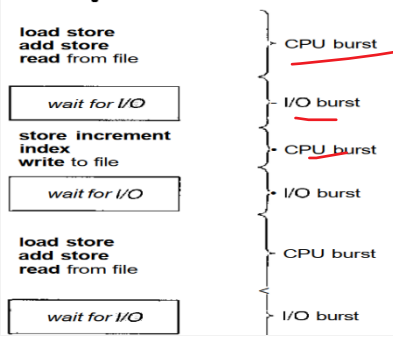
4.3.2 Asynchronous and synchronous Signals

SIGNAL	WORK
SYNCHRONOUS	Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal
ASYNCHRONOUS	When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Typically, an asynchronous signal is sent to another process.
SIGNAL DELIVERY	<ol style="list-style-type: none">1. Deliver the signal to the thread to which the signal applies.2. Deliver the signal to every thread in the process.3. Deliver the signal to certain threads in the process.4. Assign a specific thread to receive all signals for the process

5 CPU MULTI-PROGRAMMING

5.1 DEFINITION

TERM	DEFINATION
------	------------

CPU AND I/O BURSTS	 <p>Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution</p>
CPU DECISIONS SITUATION	<p>CPU-scheduling decisions may take place under the following four circumstances:</p> <ol style="list-style-type: none"> 1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes) 2. When a process switches from the running state to the ready state (for example, when an interrupt occurs) 3. When a process switches from the waiting state to the ready state (for example, at completion of I/O) 4. When a process terminates
PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING	<p>Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. Above 1 & 4 are eg. In Preemptive current running process can be stopped for execution to allow other to execute. Eg 2 & 3 in above</p>
DISPATCHER	<p>Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:</p> <ul style="list-style-type: none"> • Switching context • Switching to user mode • Jumping to the proper location in the user program to restart that program
TURNAROUND TIME	<p>The interval from the time of submission of a process to the time of completion is the <i>turnaround time</i>. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O</p>
WAITING TIME	<p><i>Waiting time</i> is the sum of the periods spent waiting in the ready queue.</p>
THROUGHPUT	<p>If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called <i>throughput</i>.</p>
RESPONSE TIME	<p>The time from the submission of a request until the first response is produced. This measure, called <i>response time</i>, is the time it takes to start responding, not the time it takes to output the response.</p>
PROCESSOR AFFINITY	<p>Because of the high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as processor affinity, meaning that a process has an affinity for the processor on which it is currently running</p>
SOFT AND HARD AFFINITY	<p>Soft: OS tries to keep in the same processor but doesn't guarantee it Hard: OS provides system calls that allows a process to specify if she doesn't wish to migrate</p>
LOAD BALANCING	<p>Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. Necessary when private queue of Processors is present</p>
PUSH AND PULL MIGRATION	<p>There are two general approaches to load balancing: push migration and pull migration. With push migration, a specific task periodically checks the load on each processor and-if it finds an imbalance-evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor</p>
LOGICAL PROCESSORS AND HYPERTHREADING	<p>Create multiple logical processors on the same physical processor, presenting a view of several logical processors to the operating system. Each logical processor has its own architecture state, which includes general-purpose and machine-state registers. Furthermore, each logical processor is responsible for its own interrupt handling, meaning that interrupts are delivered to-and handled by-logical processors rather than physical ones. Otherwise, each logical processor shares the resources of its physical processor, such as cache memory and buses.</p>

PROCESS CONTENTION SCOPE AND SYSTEM CONTENTION SCOPE

When library schedules a thread to a available LWP -> Process contention Scope.
To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.

5.2 SUB-ROUTINES

SUB-ROUTINE

WORK

SOLARIS SCHEDULING

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	40	49	59

Priority After Waking from sleep
Supports interactive process
Decreases CPU-bound priority when time quantum expires

Figure 5.11 Solaris dispatch table for interactive and time-sharing threads.

5.3 SCHEDULING ALGORITHMS

5.3.1 First Come First Serve(FCFS)

- Process which comes first is allocated the CPU first.
- Managed using linked list FIFO queue
- Non-Preemptive
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3 , and are served in FCFS order, we get the result shown in the following Gantt chart:

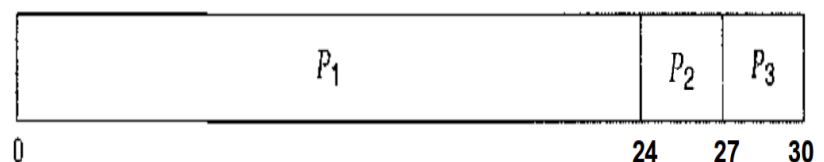


Figure 1 The waiting time is 0 milliseconds for process P_i , 24 milliseconds for process P_n , and 27 milliseconds for process P_j . Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds

5.3.2 Shortest Job First scheduling

- This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

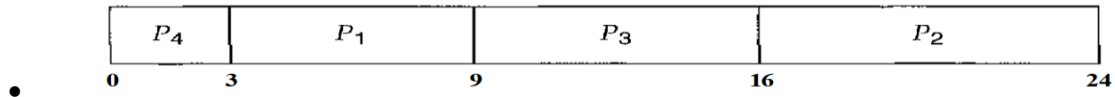


Figure 2 The waiting time is 3 milliseconds for process P_1 , 16 milliseconds for process P_2 , 9 milliseconds for process P_3 , and 0 milliseconds for process P_4 . Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. FCTS = 10 ms

- There is no way to know next CPU burst time, so we have to apply heuristics
- We expect that the next CPU burst will be similar in length to the previous ones. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
- The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the n th CPU

burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for a , $0 \leq a \leq 1$, define

$$\tau_{n+1} = a t_n + (1 - a) \tau_n.$$

5.3.3 Shortest Remaining Time Scheduling

- This time we also consider what is the time of execution left by the current execution process
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A pre-emptive SJF algorithm will preempt the currently executing process, whereas a non-pre-emptive SJF algorithm will allow the currently running process to finish its CPU burst. Pre-emptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**

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

Process	Start Time	End Time
P_1	0	1
P_2	1	5
P_4	5	10
P_1	10	17
P_3	17	26

Figure 4 Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5$ milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

5.3.4 Priority Scheduling

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

Process	Start Time	End Time
P_2	0	1
P_5	1	6
P_1	6	16
P_3	16	18
P_4	18	19

Figure 3: Avg wait time: 8.2 ms, Priority can be defined internally or externally.

- Starvation problem: low priority may never run.
- A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time

5.3.5 Round Robin Scheduling

- Preemption is added in FCTS to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

Process	Burst Time
P_1	24
P_2	3
P_3	3

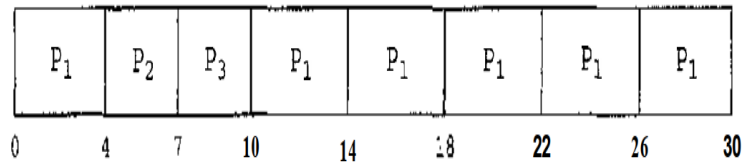


Figure 5 If we use a time quantum of 4 milliseconds, then process P_i gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2 . Since process P_i does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P_3 . Once each process has received 1 time quantum, the CPU is returned to process P_i for an additional time quantum. The average waiting time is $17/3 = 5.66$ milliseconds.

5.3.6 Multi-Level FeedBack-Queue Scheduling

- Multi level queues based on priority
- If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

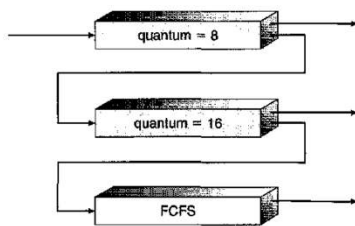


Figure 5.7 Multilevel feedback queues.

- Consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure 5.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

6 SYNCHRONISATION AND DEADLOCKS

6.1 DEFINITION

TERM	DEFINITION
PETERSON'S SOLUTION	<p>Use two data structures, int turn and Boolean flag[2]; turn indicates whose turn is to enter the critical section and. Flag array indicates process is ready to enter critical section</p> <pre> do { If the process who changes turn last enters critical section flag[i] = TRUE; turn = j; while (flag[j] && turn == j); critical section flag[i] = FALSE; remainder section } while (TRUE); </pre> <p>Figure 6.2 The structure of process P_i in Peterson's solution.</p> <pre> do { acquire lock critical section releaselock remainder section } while (TRUE); </pre>
SEMAPHORES	<p>A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().</p>


```

wait(S) {
    while S <= 0
        ; // no-op
    S--;
}

signal(S) {
    S++;
}

```

Figure 6 Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a waitQ operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal () operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

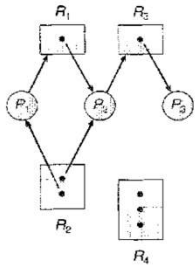
BINARY SEMAPHORE AND COUNTING SPINLOCK	Counting is unbounded whereas binary is 0 and 1. On some systems, binary semaphores are known as mutex locks , as they are locks that provide <i>mutual</i> exclusion While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock. Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock
WAKEUP OPERATION	To avoid spinlock() the process can <i>block</i> itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state.
DEADLOCKED	Waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
DEADLOCK CONDITIONS	<ol style="list-style-type: none"> 1) Mutual Exclusion: Only one proc can use the resource 2) Hold and Wait: process waiting for additional resources that other process are having 3) No Preemption: Resources can't be pre-empted 4) Circulat Wait: X_i waiting for X
RESOURCE ALLOCATION GRAPH	<p>In This example,</p> <ul style="list-style-type: none"> • Dots represent the number of instances of the resource, If dots in R4 is 3 represent 3 parallel process can run on it • Directed edge from Process to Resource indicates process requested the resource • Directed edge from Resource dot to Process indicates that resource instance has be allocated to the process 
CONDITIONS OF A DEADLOCK IN THE GRAPH	<ul style="list-style-type: none"> • If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. • If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock
RESOURCE ALLOCATION STATES	The resource-allocation <i>state</i> is defined by the number of available and allocated resources and the maximum demands of the processes.
SAFE STATES	A state is <i>safe</i> if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $1 \leq j < i$
WAIT-FOR GRAPH	An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$

Figure 7.2 Resource-allocation graph.

6.2 BANKERS' ALGORITHM

6.2.1 Data Structure

N = no of process and m = number of resources.

- *Available*: vector of length m indicating available resources
- *Max*: A matrix of $n \times m$ indicating maximum resources process i requires of resource j
- *Allocation*: A matrix of $n \times m$ defines current allocation
- *Need*: A $n \times m$ matrix indicating remaining required resources for each process

6.2.2 Safety Algorithm

- 1) Set $Work_m = Available$ and $Finish[i]_n = False$
- 2) Find i such that $Finish[i] == False$ and $Need_i \leq Work$. If no such i exist goto 4
- 3) $Work = Work + Allocation$; $Finish[i] = True$ go to step 2
- 4) If $Finish[i] == True \forall i$ then system is in safe state

6.2.3 Request Resource Algorithm

- 1) $Request_i$ = vector for process p currently requested
- 2) if $Request_i \leq Need_i$ goto step 2. Otherwise raise an error, process exceed its maximum claim
- 3) if $Request_i \leq Available$ goto step 3. Otherwise P_i must wait since resource are not available.
- 4) Have the system pretend to have allocated the requested resources to process P - by modifying the state as follows

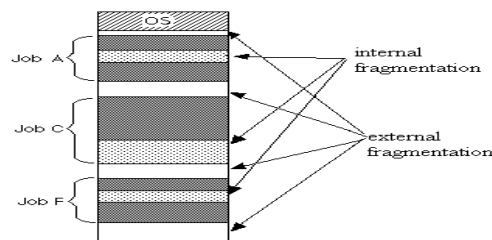
$$\begin{aligned} Available &= Available - Request \\ Allocation &= Allocation + Request \\ Need_i &= Need_i - Request \end{aligned}$$

7 MEMORY MANAGEMENT

7.1 DEFINITION

TERM	DEFINITION
LIMIT AND BASE REGISTER	The base register holds the smallest legal physical memory address; the limit register specifies the size of the range for a process.
ADDRESS BINDING	Addresses in the program are usually relative and symbolic(count) in nature, A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another
ABSOLUTE AND RELOCATABLE CODE	Compile time Addresses of variable:-> Absolute Code [Absolute address] Load time Address of variable -> Relocatable code [Absolute address] Execution time: If process can be moved during its execution from one place to another [Virtual]
LOGICAL AND PHYSICAL ADDRESS SPACE	An address generated by the CPU is commonly referred to as a logical address , whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address .
DYNAMIC LINKED LIBRARIES	This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
SWAPPING	A process, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
EXTERNAL FRAGMENTATION	External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes

INTERNAL

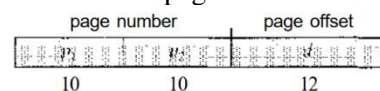


VALID INVALID BIT

One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page

HEIARCHIAL PAGING

A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where p_1 is an index into the outer page table and p_2 is the displacement within the page of the outer page table.

SEGMENTATION

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.

7.2 SUB-ROUTINES

SUB-ROUTINE

WORK

USER PROGRAM

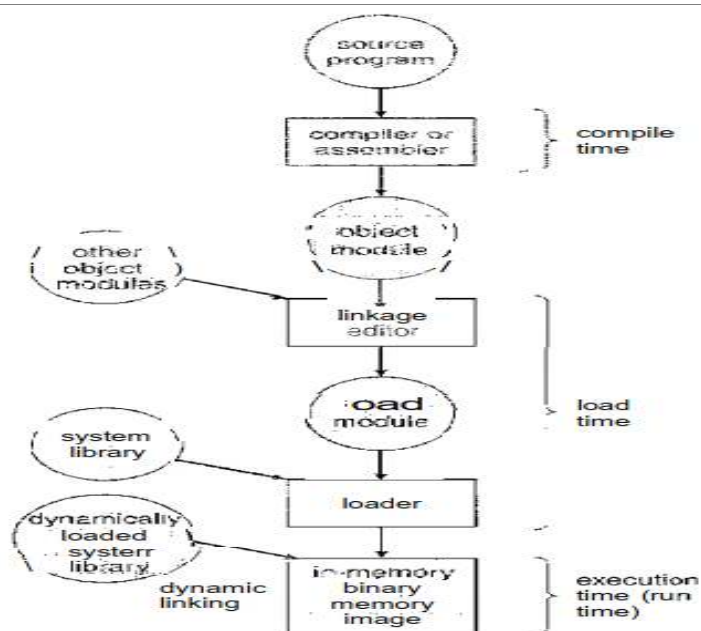


Figure 8.3 Multistep processing of a user program.

MEMORY MANAGEMENT DYNAMIC RELOCATION

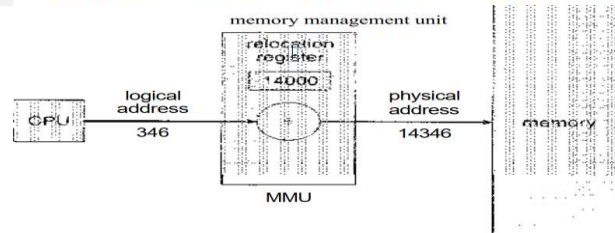


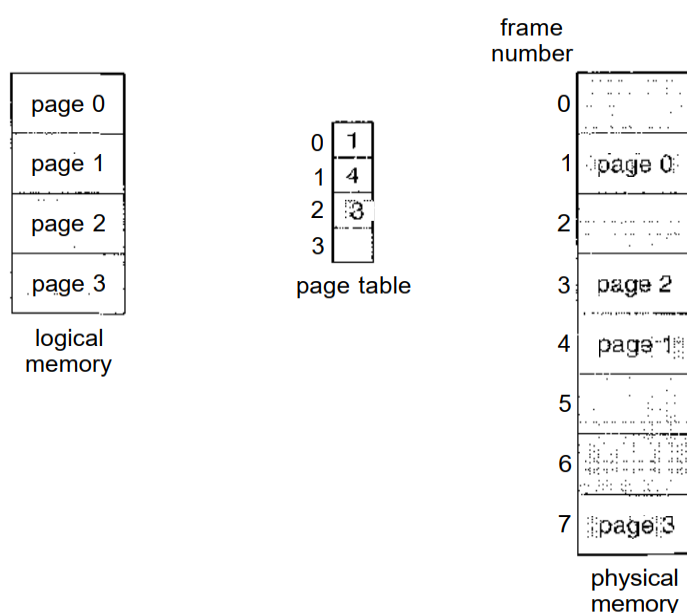
Figure 8.4 Dynamic relocation using a relocation register.

FIXED PARTITION MEMORY

In the fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. When a process arrives and needs memory, we search for a hole

DYNAMIC STORAGE	large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests
FITTING IN DYNAMIC STORAGE	<p>General dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes</p> <ul style="list-style-type: none"> • First fit. Allocate the <i>first</i> hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough. • Best fit. Allocate the <i>smallest</i> hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole. • Worst fit. Allocate the <i>largest</i> hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

7.3 PAGING



Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

Internal fragmentation possible but not external.

Figure 8.8 Paging model of logical and physical memory.

7.3.1 32-bit Architecture

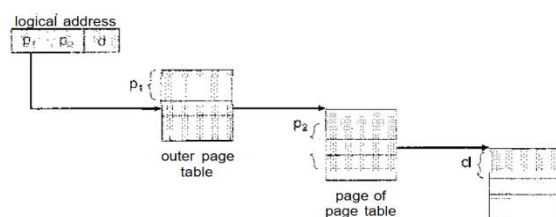


Figure 8.15 Address translation for a two-level 32-bit paging architecture.



Figure 7 The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page.

7.3.2 64-bit Architecture

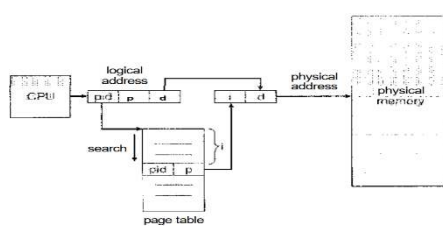


Figure 8.17 Inverted page table.

An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory