Chapter 1
Operating System Concepts

What is an Operating System?

If we just build a computer, using its basic physical components, then we end up with a lot of assembled metal, plastic and silicon. In this state the computer is useless. To turn it into one of the most useful tools we need software. We need applications that allow us to write letters, write software, perform numerical modeling, calculate cash flow forecasts etc etc. But, if all we have are just the applications, then each programmer has to deal with the complexities of the hardware. If a program requires data from a disc, the programmer would need to know how *every* type of disc worked and then be able to program at a low level in order to extract the data. In addition, the programmer would have to deal with all the error conditions that could arise. For example, it is a lot easier for a programmer to say READ NEXT RECORD than have to worry about: spinning the motor up, moving the read/write heads, waiting for the correct sector to come around and then reading the data.

It was clear, from an early stage in the development of computers, that there needed to be a "layer of software" that sat between the hardware and the software, to hide the user from such complexities, and to hide the 'breakable' parts of the computer from human error or stupidity. Thus we can define operating system as *"It is the system software that acts as a interface between computer hardware and users and provides easy interface to the users by hiding underlying complexities of computer hardware "*

Two views of an operating system

Now we are going to look at two views of an operating system. In another word we can categorize functions of an operating system into two categories

OS as Resource Manager: One view considers the operating system as a ***resource manager***. In this view the operating system is seen as a way of providing the users of the computer with the resources they need at any given time. Some of these resource requests may not be able to be met (memory, CPU usage etc.) but, the operating system is able to deal with problems such as these. For example consider the situation where more than one process is requesting CPU. If we have single CPU it can be assigned to only one process at a time. OS is responsible for when to provide CPU to which process called CPU scheduling. Similarly other resources are also managed by CPU.

OS as Extended Machine: Another view of an operating system sees it as a way of not having to deal with the complexity of the hardware. If we have operating system, we can read data easily from disc by issuing a command such as READ. But, if we don't have OS we have to deal with low level complexities f disc to read data from it. We should know whether the floppy disc is spinning, what type of recording method we should use, What error codes are used etc etc. Operating system hides all these complexities from us simple minded users and provides convenient interface. So in this view of the machine, the operating system can be seen as an *extended machine* or a *virtual machine*.

History of Operating Systems

In this section we take a brief look at the history of operating systems which is almost the same as looking at the history of computers. You are probably aware that Charles Babbage is attributed with designing the first digital computer, which he called the ***Analytical Engine***. It is unfortunate that he never managed to build the computer as, being of a mechanical design; the technology of the day could not produce the components to the needed precision. Of course, Babbage's machine did not have an operating system, but would have been incredibly useful all the same for it's era for generating nautical navigation tables.

First Generation (1945-1955) : Like many developments, the first digital computer was developed due to the motivation of war. During the Second World War many people were developing automatic calculating machines. These first generation computers filled entire rooms with thousands of vacuum tubes. Like the analytical engine **they did not have an operating system**, they did not even have programming languages and programmers had to physically wire the computer to carry out their intended instructions. The programmers also had to book time on the computer as a programmer had to have dedicated use of the machine.

Second Generation (1955-1965): Vacuum tubes proved very unreliable and a programmer, wishing to run his program, could quite easily spend all his/her time searching for and replacing tubes that had blown. The mid fifties saw the development of the transistor which, as well as being smaller than vacuum tubes, was much more reliable. It now became feasible to manufacture computers that could be sold to customers willing to part with their money. Of course, the only people who could afford computers were large organizations who needed large air conditioned rooms in which to place them. Now, instead of programmers booking time on the machine, the computers were under the control of computer operators. Programs were submitted on punched cards that were placed onto a magnetic tape. This tape was given to the operators who ran the job through the computer and delivered the output to the expectant programmer.

As computers were so expensive methods were developed that allowed the computer to be as productive as possible. One method of doing this (which is still in use today) is the concept of ***batch jobs***. Instead of submitting one job at a time, many jobs were placed onto a single tape and these were processed one after another by the computer. The ability to do this can be seen as the first real operating system .

Third Generation (1965-1980): The third generation of computers is characterized by the use of Integrated Circuits as a replacement for transistors. This allowed computer manufacturers to build systems that users could upgrade as necessary. Up until this time, computers were single tasking. The third generation saw the start of ***multiprogramming***. That is, the computer could *give the illusion* of running more than one task at a time. Being able to do this allowed the CPU to be used much more effectively. When one job had to wait for an I/O request, another program could use the CPU. The concept of multiprogramming led to a need for a more complex operating system. One was now needed that could schedule tasks and deal with all the problems that this brings. In implementing multiprogramming, the system was confined by the amount of physical memory that was available (unlike today where we have the concept of virtual memory). Another feature of third generation machines was that they implemented ***spooling***. This allowed reading of punch cards onto disc as soon as they were brought into the computer room. This eliminated the need to store the jobs on tape, with all the problems this brings. Similarly, the output from jobs could also be stored to disc, thus allowing programs that produced output to run at the speed of the disc, and not the printer.

Although, compared to first and second generation machines, third generation machines were far superior but they did have a downside. Up until these point programmers were used to giving their job to an operator and watching it run. This problem led to the concept of ***time sharing***. This allowed programmers to access the computer from a terminal and work in an interactive manner. Obviously, with the advent of multiprogramming, spooling and time sharing, operating systems had to become a lot more complex in order to deal with all these issues.

Fourth Generation (1980-present): The late seventies saw the development of Large Scale Integration (LSI). This led directly to the development of the personal computer (PC). These computers were (originally) designed to be single user, highly interactive and provide **graphics capability**. One of the requirements for the original PC produced by IBM was an operating system and, Bill Gates supplied MS-DOS on which he made his fortune. In addition, mainly on non-Intel processors, the UNIX operating system was being used. It is still (largely) true today that there are "mainframe" operating systems (such as VME which runs on ICL mainframes) and "PC" operating systems (such as MS-Windows and UNIX), although the distinctions are starting to blur. Mainly, we can say that **Graphical User Interface (GUI)** became popular in 3rd generation computers.

Fifth Generation (Sometime in the future): If you look through the descriptions of the computer generations you will notice that each have been influenced by new hardware that was developed (vacuum tubes, transistors, integrated circuits and LSI). The fifth generation of computers may be the first that breaks with this tradition and the advances in software will be as important as advances in hardware. One view of what will define a fifth generation computer is one that is able to interact with humans in a way that is natural to us. No longer will we use mice and keyboards but we will be able to talk to computers in the same way that we communicate with each other. In addition, we will be able to talk in any language and the computer will have the ability to convert to any other language. Computers will also be able to reason in a way that imitates humans. Just being able to accept (and understand!) the spoken word and carry out reasoning on that data requires many things to come together before we have a fifth generation computer. For example, advances need to be made in AI (Artificial Intelligence) so that the computer can mimic human reasoning. It is also likely that computers will need to be more powerful. Maybe parallel processing will be required. Maybe a computer based on a non-silicon substance may be needed to fulfill that requirement (as silicon has a theoretical limit as to how fast it can go). This is one view of what will make a fifth generation computer. At the moment, as we do not have any, it is difficult to provide a reliable definition.

Types of Operating systems

All of this history and development has left us with a wide variety of operating systems, not all of which are widely known.

Mainframe Operating Systems: A mainframe with 1000 disks and thousands of gigabytes of data is not unusual. Mainframes are normally used as web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions. The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need heavy amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing.

A batch system is one that processes routine jobs without any interactive user present. A claim processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction processing systems handle large numbers of small requests; for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360

Real-Time Operating Systems: Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system.**

Another kind of real-time system is a **soft real-time** system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category.

Personal Computer Operating Systems: Job of personal computer operating system is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, Internet access etc. Personal computer operating systems are so widely known to the people who use computers but only few computer users knows about other types of operating systems. Common examples of PC operating systems are Windows 2008, Windows 2007, the Macintosh operating system, Linux, Ubuntu etc.

Server Operating Systems: Server operating systems run on servers, which are very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Web sites use servers to store the Web pages and handle the incoming requests. Some Examples of typical server operating systems are UNIX and Windows 2007 server, Sun Solaris etc.

Review of Computer Hardware

An operating system is closely related to the hardware of the computer it runs on. It extends the computer's instruction set and manages its resources. To work, it must be well known about the hardware, at least, about how the hardware appears to the programmer.

Processors: The "brain" of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is given below:

- ✓ Fetch
- ✓ Decode
- ✓ Execute

Fetching loads instruction from memory that is pointed by program counter. Decoding determines type of instruction, operations, and operands. Execution carried out the specified operation on operands. And then fetch, decode, and execute subsequent instructions. Each CPU has a specific set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Because reading an instruction or data word from memory takes much longer than executing an instruction, all CPUs contain some registers inside it to store currently used variables and temporary results.

Most computers have several special registers that are visible to the programmer. One of these is the **program counter,** which contains the memory address of the next instruction to be fetched. Another register is the **stack pointer,** which points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure's stack frame holds input parameters, local variables, and temporary variables that are not kept in registers. Yet another register is the **PSW (Program Status Word).** This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. The PSW plays an important role in system calls and I/O. The operating system must be aware of all the registers. In timesharing systems, the operating system will often stop the running program to (re)start another one. Every time it stops a running program, the operating system must save all the registers so they can be restored when the program runs later.

Many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it was executing instruction n , it could also be decoding instruction $n + 1$ and fetching instruction $n + 2$. Such an organization is called a **pipeline.**

Instruction n	Fetch	Decode	Execute		
Instruction n+1		Fetch	Decode	Execute	
Instruction n+2			Fetch	Decode	Execute

Figure: Three Stage Pipelining

Pipelines cause compiler writers and operating system writers great headaches because they expose the complexities of the underlying machine to them. Even more advanced than a pipeline design is a **superscalar** CPU. In this design, multiple execution units are present, for example, one for integer arithmetic, one for floating-point arithmetic, and one for Boolean operations. Two or more instructions are fetched at once, decoded, and dumped into a holding buffer until they can be executed. As soon as an execution unit is free, it looks in the holding buffer to see if there is an instruction it can handle, and if so, it removes the instruction from the buffer and executes it.

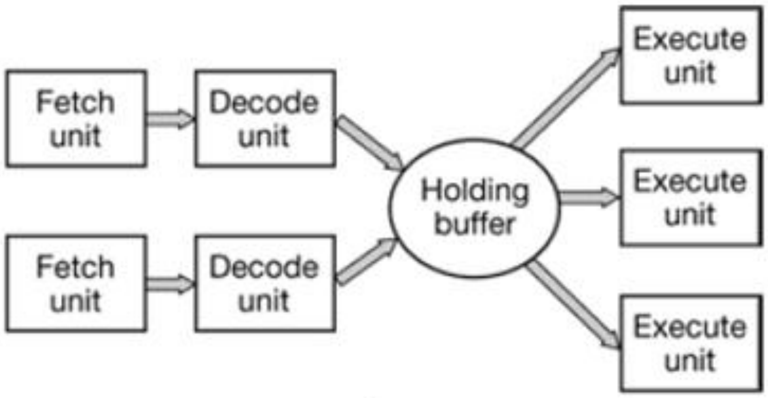
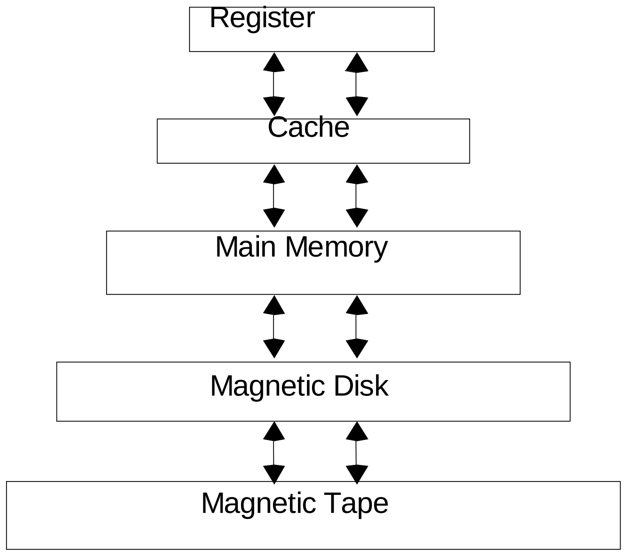


Figure: Superscalar CPU

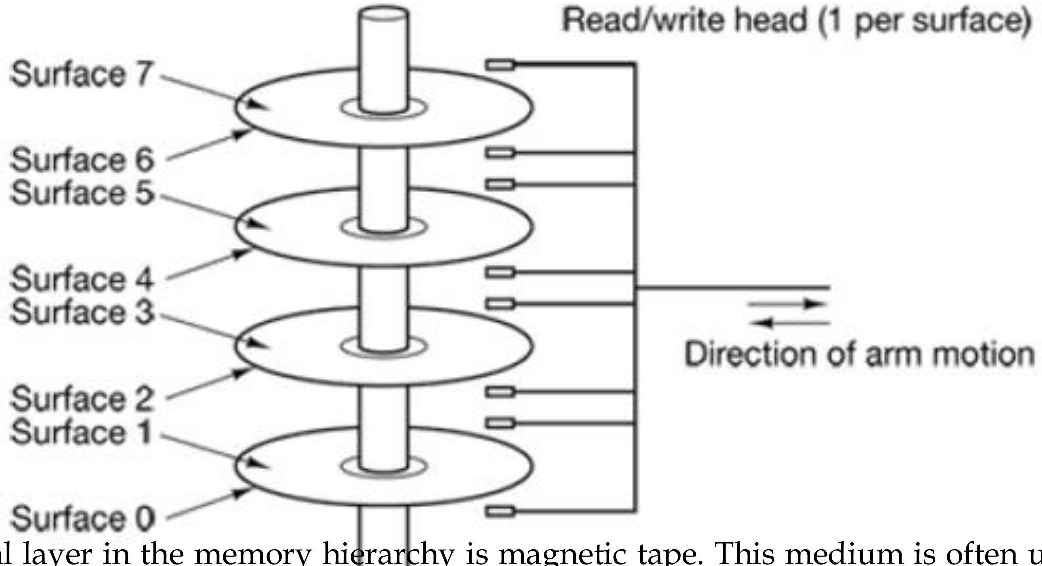
Most CPUs have two operating modes, kernel mode and user mode. Usually a bit in the PSW controls the mode. When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware. In contrast, user programs run in user mode, which permits only a subset of the instructions to be executed and a subset of the features to be accessed. Generally, all instructions involving I/O and memory protection are disallowed in user mode. Setting the PSW mode bit to kernel mode-is also forbidden, of course.

Memory: The second major component in any computer is the memory. Ideally, a memory should be extremely faster so that CPU is not held up by the memory and should be large enough. No current technology satisfies these goals, so the memory system is designed as hierarchy.



The top layer consists of the **registers** internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. The storage capacity available in them is typically 32 x 32-bits on a 32-bit CPU and 64 x 64-bits on a 64-bit CPU. Generally size of registers is less than 1 KB. Next layer in the hierarchy is the **cache memory**, which is mostly controlled by the hardware. When the program needs to read a memory word, the cache hardware checks to see if the data needed is in the cache. If it is, called a **cache hit,** the request is satisfied from the cache and no memory request is sent over the bus to the main memory. Cache hits normally take about two clock cycles. Cache misses have to go to memory, with a substantial time penalty. Cache memory is limited in size due to its high cost. Some machines have two or even three levels of cache, each one slower and bigger than the one before it.

Main memory comes next. This is the workhorse of the memory system. Main memory is often called **RAM (Random Access Memory).** All CPU requests that cannot be satisfied out of the cache go to main memory. Next in the hierarchy is **magnetic disk** (hard disk). Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. This low speed is due to the fact that a disk is **a mechanical device**. A disk consists of one or more metal platters that rotate at 5400, 7200, or 10,800 rpm. A **mechanical arm** is pivoted over the platters. Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track.** Together, all the tracks for a given arm position form a **cylinder.** Each track is divided into some number of sectors, typically 512 bytes per sector. The arm can move from one cylinder to another cylinder which takes some time in msec. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5 msec to 10 msec, depending on the drive's rpm. Once the sector is under the head, reading or writing occurs at a rate of 5 MB/sec on low-end disks to 160 MB/sec on faster ones.



The final layer in the memory hierarchy is magnetic tape. This medium is often used as a backup for disk storage and for holding very large data sets. To access a tape, it must first be put into a tape reader. Then the tape may have to be spooled forwarded to get to the requested block. The big plus of tape is that it is exceedingly cheap per bit and removable, which is important for backup tapes that must be stored off-site in order to survive fires, floods, earthquakes, etc.

Bus: Physically, bus is a set of wires used for moving data, instruction, and control signals from one component of a computer system to another component. Each component of the computer is connected to these buses. Broadly we can divide buses into following categories:

- ✓ Address Bus
- ✓ Data Bus
- ✓ Control Bus

a) **Address bus:** Address bus is used to specify the address of the memory location to be accessed. CPU reads data or instructions from memory locations by specifying the address of its location or CPU write data to the memory locations by specifying the memory address. Address bus is unidirectional. This means, address bus carries memory location in only one direction, from CPU to memory, both for read and write operation. The width of address bus determines the maximum possible memory of the system.

b) **Data Bus:** Actual data is transferred via the **data bus**. In case of read operations, CPU sends an address to memory; the memory will send data via the data bus in return to the CPU. But In case of write operation, CPU sends an address via address bus and data via data bus and then specified data is written in specified memory location. Therefore data bus is bidirectional.

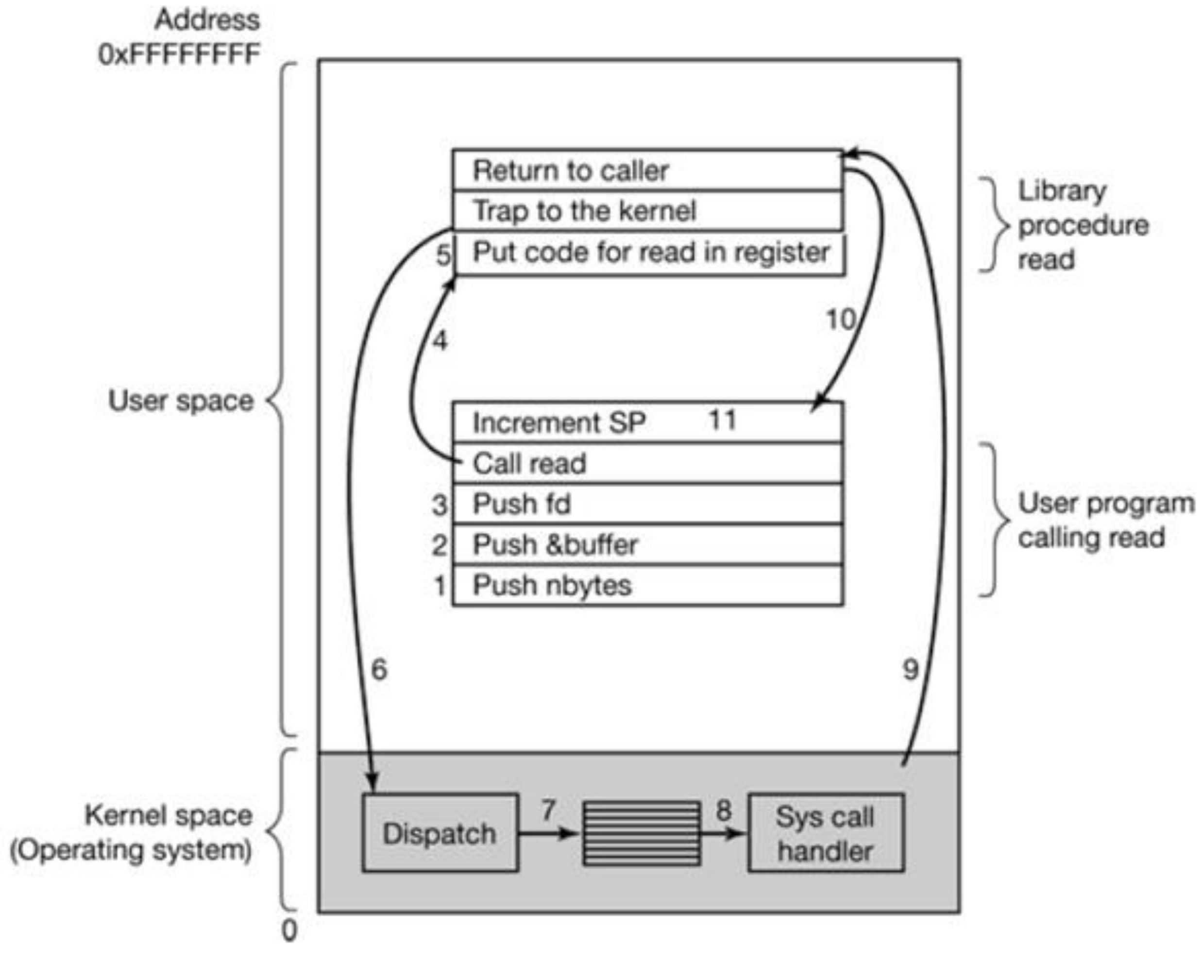
c) **Control Bus:** Path for sending the control signals generated by Control Unit is called control bus. Data and Address bus is shared by all the components of the system through the control bus. If CPU needs to perform write operation on memory it sends 'write' signal via control bus but if CPU needs to perform read operation on memory 'read' signal is sent via control bus. Control bus is also unidirectional because only CPU sends control signals to other devices.

System Calls

System calls are the interface between the operating system and the user programs. Access to the operating system is done through system calls. Each system call has a procedure associated with it so that calls to the operating system can be done in a similar way as that of normal procedure call. When we call one of these procedures it places the parameters into registers and informs the operating system that there is some work to be done. When a system call is made TRAP instruction (sometimes known as a ***kernel call*** or a ***supervisor call***) is executed. This instruction switches the CPU from ***user mode*** to ***kernel*** (or ***supervisor***) mode. The operating system now carries out the work, which includes validating the parameters. Eventually, the operating system will have completed the work and will return a result in the same way as a user written function written in a high level language. Making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well. An example of an operating system call (via a procedure) is

count = read(file, buffer, nbytes);



System calls are performed in a series of steps. To make this concept clearer, let us examine the read call written above. Calling program first pushes the parameters onto the stack, as shown in steps 1-3 before calling the *read* library procedure, which actually makes the read system call. Then the actual call to the library procedure (step 4) is made. This instruction is the normal procedure call instruction used to call all procedures. The library procedure, possibly written in assembly language, typically puts the system call number in specified register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The kernel code examines the system call number and then dispatches to the correct system call handler (step 7). At that point the system call handler runs (step 8). Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10). To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Since, normally the system stack grows downward, stack pointer is incremented exactly enough to remove the parameters pushed before the call to *read.*

The Shell

The operating system is the mechanism that carries out the system calls requested by the various parts of the system. Tools such as compilers, Linkers, Loaders, editors etc are not part of the operating system. Similarly, the ***shell*** is not part of the operating system. The shell is the part of operating systems such as UNIX and MS-DOS where we can type commands to the operating system and receive a response. Shell is also called the ***Command Line Interpreter*** (CLI) or the "C" prompt. Shell is the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh, csh, ksh,* and *bash.*

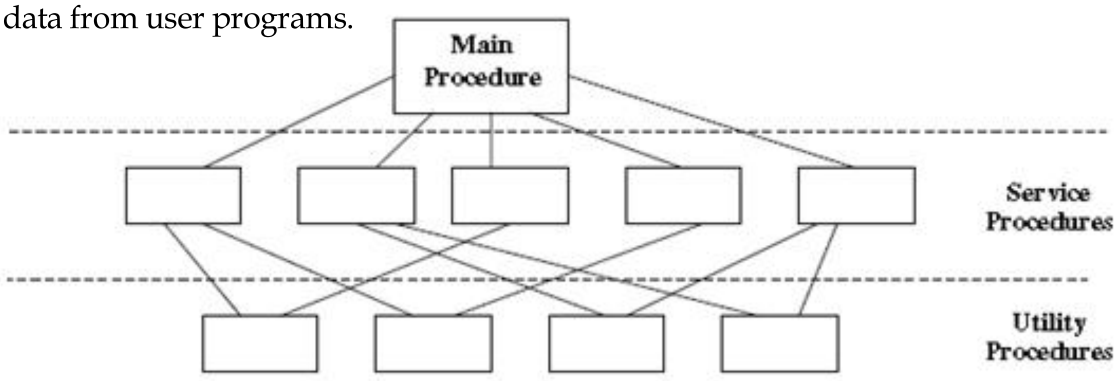
The shell makes heavy use of operating system features and thus serves as a good example of how the system calls can be used. When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt,** a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types **"date"** command, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line. A more complicated command is

```
cat file1 file2 file3 | sort > /dev/lp &
```

This command concatenates three files and pipes them to the sort program. It then redirects the sorted file to a line printer. The ampersand "&" at the end of the command instructs UNIX to issue the command as a background job. This results in the command prompt being returned immediately, whilst another process carries out the requested work. Above command makes a series of system calls to the operating system in order to satisfy the whole request.

Operating System Structure

Monolithic Systems: One possible approach is to have no structure to the operating system at all i.e. it is simply a collection of procedures. Each procedure has a well defined interface and any procedure is able to call any other procedure. The operating system is constructed by compiling all the procedures into one huge monolithic system. There is no concept of encapsulation, data hiding or structure amongst the procedures. This organization suggests a basic structure for the operating system: A main program that invokes the requested service procedure, A set of service procedures that carry out the system calls, and A set of utility procedures that help the service procedures. In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs.



Layered Systems: In 1968 E. W. Dijkstra (Holland) and his students built the "THE" operating system that was structured into layers. It can be viewed as a generalization of the model shown above, but this model had six layers.

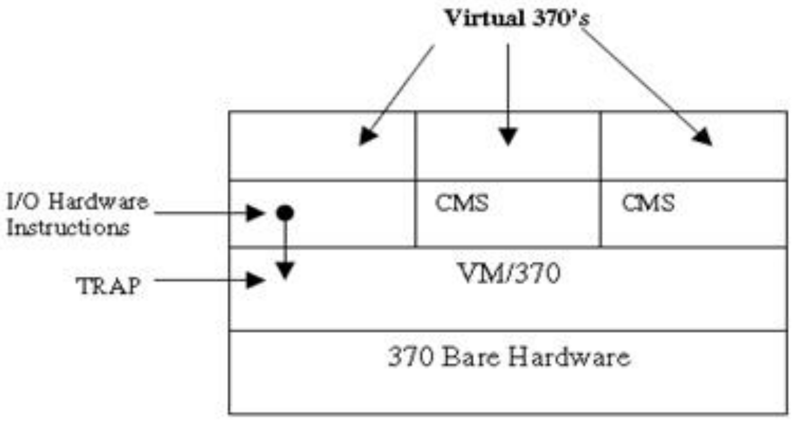
- ✓ **Layer 0** was responsible for the multiprogramming aspects of the operating system. It decided which process was allocated to the CPU. It dealt with interrupts and performed the context switches when a process change was required.

- ✓ **Layer 1** was concerned with allocating memory to processes.
- ✓ **Layer 2** deals with inter-process communication and communication between the operating system and the console.
- ✓ **Layer 3** manages all I/O between the devices attached to the computer. This included buffering information from the various devices.
- ✓ **Layer 4** was where the user programs were stored.
- ✓ **Layer 5** was the overall control of the system (called the system operator).

As you move through this hierarchy (from 0 to 5) you do not need to worry about the aspects you have "left behind." For example, high level user programs (level 4) do not have to worry about where they are stored in memory or if they are currently allocated to the processor or not, as these are handled in low level 0-1.

Virtual Machines: Virtual machines mean different things to different people. For example, if you run an MS-DOS prompt from with Windows 2008 you are running, what Microsoft call, a virtual machine. It is given this name as the MS-DOS program is fooled into thinking that it is running on a machine that it has sole use of. ICL's mainframe operating system is called VME (Virtual Machine Environment). The idea is that when you log onto the machine a VM (Virtual Machine) is built and it looks as if you have the computer all to yourself.

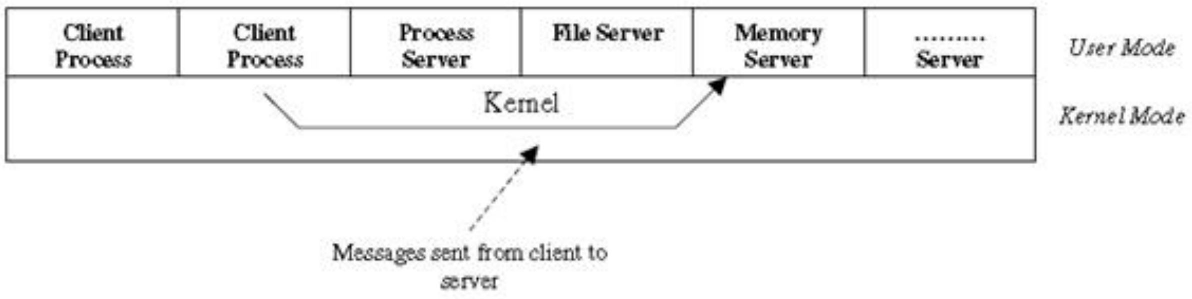
One of the first operating system (VM/370) was able to provide a virtual machine to each user. In addition, each user was able to run different operating systems if they so desired. Main concept of the system was that the bare hardware was "protected" by VM/370 (called a ***virtual machine monitor***). This provided access to the hardware when needed by the various processes running on the computer. Unlike other operating systems, instead of simply providing an extension of the hardware that hides the complexities of the hardware, VM/370 provided an exact copy of the hardware, which included I/O, interrupts and user/kernel mode. Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Any instructions to the hardware are trapped by VM/370, which carried out the instructions on the physical hardware and the results returned to the calling process. The diagram below shows a model of the VM/370 computer. Note that CMS is a "Conversational Monitor System" and is just one of the many operating systems that can be run - in this case it CMS is a single user OS intended for interactive time sharing.



Client-Server Model: One of the recent advances in computing is the idea of a client/server model. A server provides services to any client that requests it. This model is heavily used in distributed systems where a central computer acts as a server to many other computers. The server may be something as simple as a print server, which handles print requests from clients. Or, it could be relatively complex, and the server could provide access to a database which only it is allowed to access directly.

Operating systems can be designed along similar lines. Take, for example, the part of the operating system that deals with file management. This could be written as a server so that any process which requires access to any part of the filing system asks the file management server to carry out a request, which presents the calling client with the results. Similarly, there could be servers which deal with memory management, process scheduling etc. The benefits of this approach include

- ✓ It can result in a minimal kernel. This results in easier maintenance as not so many processes are running in kernel mode. All the kernel does is providing the communication between the clients and the servers.
- ✓ As each server is managing one part of the operating system, the procedures can be better structured and more easily maintained.
- ✓ If a server crashes it is less likely to bring the entire machine down as it will not be running in kernel mode. Only the service that has crashed will be affected.



Chapter 2
Process Management

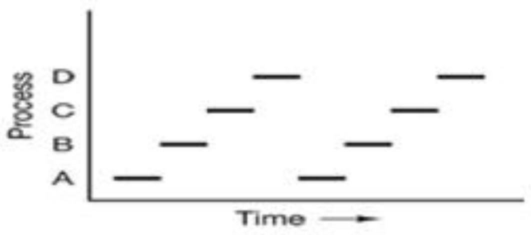
Processes

Process is a program that is ready for execution in CPU. When a program is loaded into memory, it becomes ready for execution and competes with other processes to access CPU. Thus when a program is loaded into memory it becomes process. Computers nowadays can do many things at the same time. They can be writing to a printer, reading from a disc and scanning an image. Operating system is responsible for running many processes, usually, on the same CPU. In fact, only one process can be run at a time so the operating system has to share the CPU between the processes that are available to be run, while giving the illusion that the computer is doing many things at the same time. This approach can be directly contrasted with the first computers. They could only run one program at a time.

Now, the main point of this part is to consider how an operating system deals with processes when we allow many to run. To achieve true **parallelism** we must have multiprocessor system where n processors can execute n programs simultaneously. True parallelism cannot be achieved with single CPU. In single processor CPU switched from one process to another process rapidly. In 1 sec CPU can serve 100's of processes giving the illusion of parallelism to a user which is called ***pseudo-parallelism***.

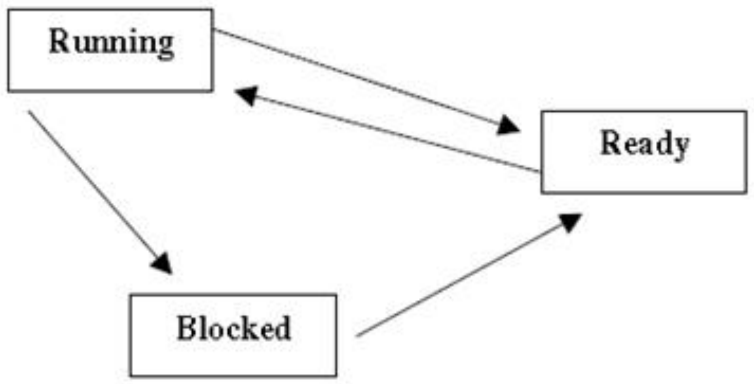
The Process Model

All the runnable programs on the computer including the operating system, is organized into a number of **sequential processes**. Every process includes the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process. To understand the system of CPU switching from program to program and execution of programs, consider four processes each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it has finished its turn on CPU, the physical program counter is saved in the process logical program counter in memory. Same process is also repeated for another process.



Process States

A process may be in one of three states. A state transition diagram can be used to represent the various states and the transition between those states.

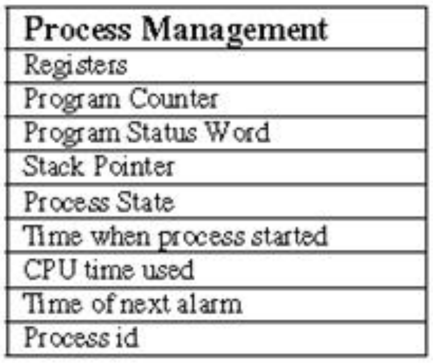


- ✓ **Running:** Only one process can be running sate at any one time (assuming a single processor machine). Running process is the process that is actually using the CPU at that time.
- ✓ **Ready:** A process that is ready is runnable but cannot get access to the CPU due to another process using it. Any number of processes can be in ready state at the same time.
- ✓ **Blocked:** A blocked process is unable to run until some external event has taken place. For example, it may be waiting for data to be retrieved from a disc. Blocked process cannot use CPU even if CPU has to do nothing.

We can see from above transition diagram that a running process can either move to blocked sate or ready state. Running process is moved to blocked state when it needs to wait for an external event such as input from keyboard or it can go to a ready state when the scheduler allows another process to use the CPU. A ready process can only move to a running state when scheduler decides to provide CPU to that process. But, a blocked process can only move to a ready state when the external event for which it is waiting occurs.

Implementation of Processes

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



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

We can notice that, as well as information to ensure the process can start again (e.g. Program Counter), the process control block also holds accounting information such as the time the process started and how much CPU time has been used. This is only a sample of the information held. There will be other information, at least all process entries must also include files being used and the memory the process is using.

Threads

One definition of a process is that it has an address space and a single thread of execution. Processes are based on two independent concepts: resource grouping and execution. Sometimes it would be beneficial if two (or more) processes could share the same address space (i.e. same variables and registers etc) and run parts of the process in parallel. This is what threads do. **Multithreading** is one of the technique for achieving multitasking.

Threads are like mini-processes that operate within a single process. Each thread has its own program counter and stack so that it knows where it is. Apart from this they can be considered the same as processes, with the exception that they share the same address space. This means that all threads from the same process have access to the same global variables and the same files. Threads are also called **light weight** processes. The table given below shows the various items that a process has, compared to the items that each thread has.

Per Thread Items	Per Process Items
☞ Program Counter	☞ Address Space
☞ Stack	☞ Global Variables
☞ Register Set	☞ Open Files
☞ Child Thread	☞ Child Process
☞ State	☞ Timers
	☞ Signals
	☞ Semaphores
	☞ Accounting Information

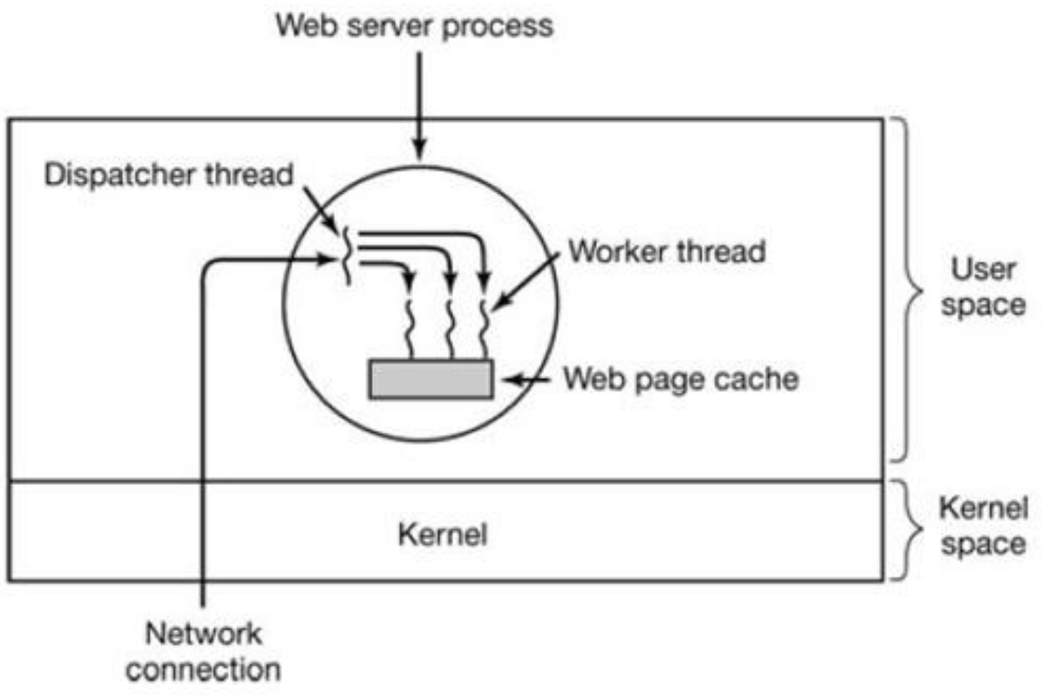
Thread Usage

- ✓ Some application needs to perform multiple activities at once that share common data and files. Thus reason behind need of multiple threads can be listed as below:
- ✓ Ability of parallel entities, within a process, to all share common variables and data.
- ✓ Since threads do not have their own address space, it is easier to create and destroy threads than processes. In fact thread creation is around 100 times faster than process creation
- ✓ Thread switching has much less overhead than process switching because threads have few information attached with them.
- ✓ Performance gain when there is a lot of computing and a lot of I/O as tasks can overlap. Pure CPU bound thread application no advantage though
- ✓ Threads very effective on multiple CPU systems for concurrent processing because with multiple CPUs true parallelism can be achieved

For example, assume we have a server application running. Its purpose is to accept messages and then act upon those messages. Consider the situation where the server receives a message and, in processing that message, it has to issue an I/O request. Whilst waiting for the I/O request to be satisfied it goes to a blocked state. If new messages are received, whilst the process is blocked, they cannot be processed until the process has finished processing the last request. One way we could achieve our objective is to have two processes running. One process deals with incoming messages and another process deals with the requests that are raised such as I/O. However, this approach gives us two problems

- ✓ We still have the problem, in that either of the processes could become blocked. ✓ The two processes will have to update shared variables. This is far easier if they share the same address space.

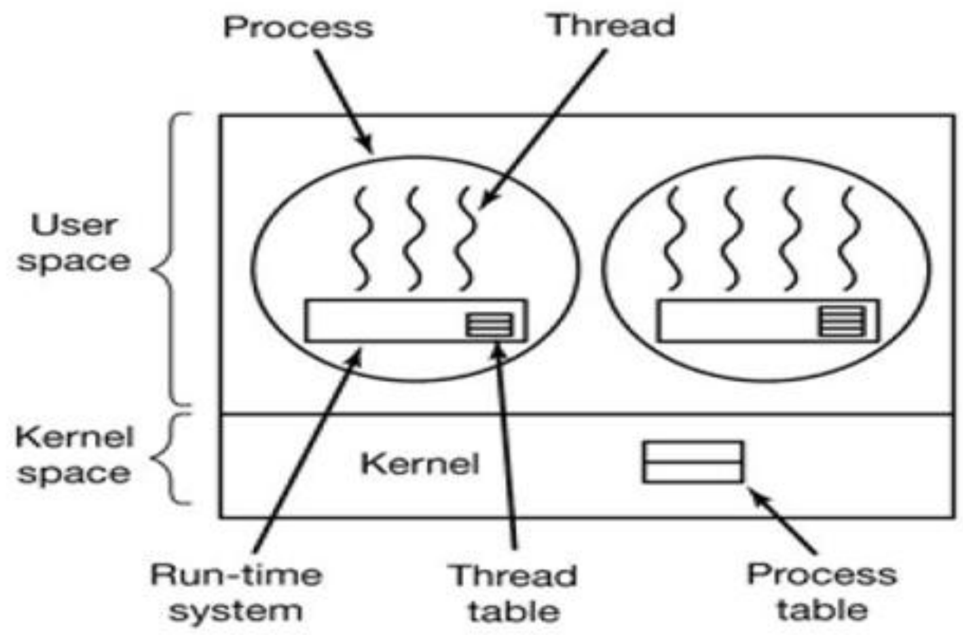
The answer to these types of problems is to use ***threads.*** Now consider yet another example of where threads are useful: a server for a World Wide Web site. Requests for pages come in and the requested page is sent back to the client. One way to organize the Web server is to use a process with dispatcher thread and worker thread. The dispatcher thread reads incoming requests for work from the network. After examining the request, the dispatcher wakes up the sleeping worker, moving it from blocked state to ready state. When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.



Implementation of threads in User Space

There are two main ways to implement a threads package: in user space and in the kernel. The choice is moderately controversial, and a hybrid implementation is also possible. One method is to put the threads package entirely in user space. . If we put threads entirely in user space, the kernel knows nothing about them rather it just thinks that it is serving single threaded processes. User level threads run on top of a run-time system, which is a collection of procedures that manage threads. Some of the procedures are *thread_create* , *thread_exit* , *thread_wait* , *thread_yield* etc.

When threads are implemented in user space Each process needs to maintain its own thread table and are analogous to the kernel's process table. A thread table contains thread program counter, thread stack pointer, thread registers, thread state i.e. ready, blocked etc. Unlike processes though, when a *thread_yield* is called it saves the thread information in the thread table itself and can instruct the thread scheduler to call another thread. Just local procedures used, so more efficient/faster than a kernel call.



Advantages of implementing threads in user space

- ✓ User-level threads package can be implemented on an operating system that does not support threads.
- ✓ Threads permit each process to have its own scheduler. This means processes can customize scheduling algorithm.
- ✓ With user level threads no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

Disadvantages of implementing threads in user space

- ✓ Allowing threads to use blocking system calls causes other threads in the same address space to be blocked.
- ✓ If a thread causes a page fault, the kernel blocks the entire process until the disk I/O is complete, even though other threads might be runnable. This is because kernel doesn't know anything about threads in user space.
- ✓ Within a single process, there are no clock interrupts, making it impossible to schedule threads in the fashion that takes turns. No other thread in that process will ever run unless the first thread voluntarily gives up the CPU

One possible solution to the problem of threads running forever is to have the run-time system request a clock signal (interrupt) once a second to give it control, but periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial.

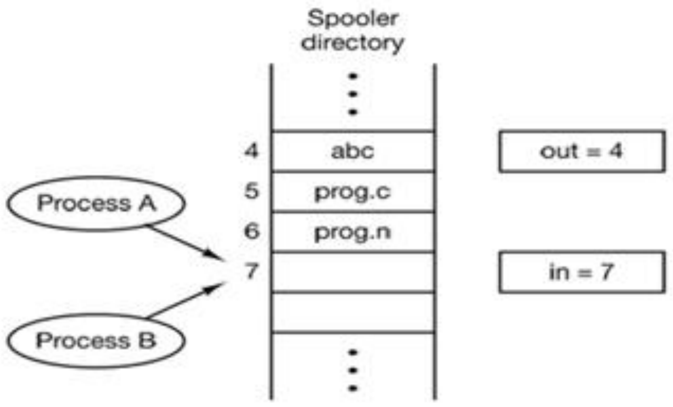
Interprocess Communication

The mechanism of passing information from one process to another process is called IPC. Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts.

Race Conditions

It is sometimes necessary for two processes to communicate with one another. This can either be done via shared memory or via a file on disc. We are not discussing the situation where a process can write some data to a file that is read by another process at a later time e.g. days, weeks or months. We are talking about two processes that need to communicate at the time they are running. **Race conditions** are situations where two or more processes reads or writes some shared data at the same time and the final result is incorrect. Final result may be different according to order of completion of competing processes.

To see how interprocess communication works in practice, let us consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon,** periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory. Consider the situation given in figure below, where *in* and *out* are two shared variables.

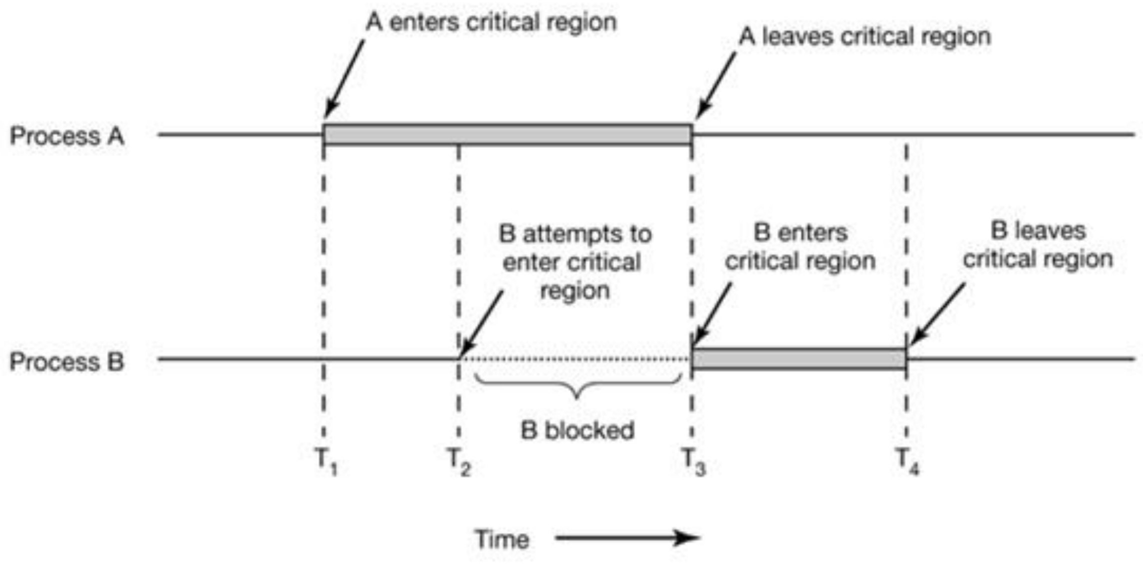


Process *A* reads value of *in* (i.e. 7) but before inserting the name of the document to be printed in index 7, a clock interrupt occurs and the CPU decides to schedule process *B*. Now, process *B* also reads *in,* and also gets a 7. At this instant both processes think that the next available slot is 7. Process *B* now continues to run and stores the name of its file in slot 7 and updates *in* to be an 8. After some time process *A* runs again, starting from the place it left off. It looks at value of *in* stored in its local variable and finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it increments value of *in* cached by (which is 8) it and sets *in* to 8. The printer daemon will not notice anything wrong, but process *B* will never receive any output. Situations like this are called **race conditions**.

Critical Sections

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

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



Implementing Mutual Exclusion

Mutual Exclusion with Busy Waiting

Busy waiting means if the process is not allowed to enter its critical section it sits in a tight loop waiting for a condition to be met. This is obviously wasteful in terms of CPU usage. Busy waiting is useful in systems where CPU have to do no other things busy waiting.

a) **Disabling Interrupts:** Perhaps the most obvious way of achieving mutual exclusion is to allow a process to disable interrupts before it enters its critical section and then enable interrupts after it leaves its critical section. By disabling interrupts the CPU will be unable to switch processes. This guarantees that the process can use the shared variable without another process accessing it. But, it is unwise to give user processes the power to turn off interrupts. At best, the computer will not be able to service interrupts for the time period a process is in its critical section. At worst, the process may never enable interrupts, thus crashing the computer. Although disabling interrupts might seem a good solution its disadvantages overshadow the advantages.

b) **Lock Variables:** Lock variable is software solution to achieve mutual exclusion. It is a binary variable whose value is either zero or 1. Initially zero is assigned to lock variable. Value 0 indicates no process is in critical region and value 1 indicates that some process is doing processing inside the critical region. When a process wants to enter critical region it tests lock variable. If it is 1, it waits for lock to be 0 by sitting in tight loop otherwise it sets lock to 1 and enter into critical section. And process resets lock to zero when it exits from its critical region.

   This scheme simply moves the problem from the shared variable to the lock variable. It again suffers from race condition problem. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time

c) **Strict alternation**

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

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

This approach also suffers from one major flaw. Consider following sequence of events:

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

From this observation we can see that strict alteration violates the condition "No process running outside its critical section may block other processes"

d) **Peterson's Solution:** It is a solution to the mutual exclusion problem that does not require strict alternation, but still uses the idea of lock variables together with the concept of taking turns. The solution consists of two procedures, shown below:

```
#define N      2                        // Number of processes
int turn;                               // Whose turn is it?
int interested[N];                      // All values initially FALSE
void enter_region(int process)
{
    other = 1 - process;                // the opposite process
    interested[process] = TRUE;         // this process is interested
    turn = process;                     // set flag
    while(turn == process && interested[other] == TRUE); // wait
}
void leave_region(int process)
{
    interested[process] = FALSE; // process leaves critical region
}
```

Initially, both processes are not in their critical region and the array *interested* has all its elements set to false. Assume that process 0 calls e**nter_region**. The variable *other* is set to one (the other process number) and it indicates its interest by setting the relevant element of *interested* to true. Next it sets the *turn* variable, before coming across the while loop. In this instance, the process will be allowed to enter its critical region, as process 1 is not interested in running. Now process 1 could call **enter_region**. It will be forced to wait as the other process (i.e process 0) is still interested. Process 1 will only be allowed to continue when *interested[0]* is set to false which is possible only when process 0 calls **leave_region**.

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

e) Test and Set Lock (TSL)

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

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

Assume, again, two processes. Process 0 calls enter_region. The tsl instruction copies the flag to a register and sets flag to a non-zero value. The flag is now compared to zero and if found to be non-zero the routine loops back to the top. Only when process 1 has set the flag to zero (or under initial conditions), by calling leave_region, process 0 will be allowed to enter into critical region.

Comments on Busy Waiting Solutions

Of all the solutions we have looked at, both Peterson's and the TSL solutions solve the mutual exclusion problem. However, both of these solutions suffers from following two problems:

- ✓ **Busy Waiting Problem:** If the process is not allowed to enter its critical section it sits in a tight lop waiting for a condition to be met. This is obviously wasteful in terms of CPU usage.
- ✓ **Priority Inversion Problem:** Suppose we have two processes, one of high priority h, and one of low priority l. Scheduler is set so that whenever h is in ready state it must be run. If l is in its critical section when h becomes ready to run, l will be placed in a ready state so that h can be run. However, if h tries to enter its critical section then it will be blocked by l, which will never be given the opportunity of running and leaving its critical section. Meantime, h will simply sit in a loop forever. This is sometimes called the priority inversion problem.

Sleep and Wakeup

One of the simplest solutions to achieve mutual exclusion without busy waiting is the pair sleep and wakeup system calls. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.

The Producer-Consumer Problem

Producer-consumer problem is also known as the **bounded-buffer** problem. Two processes called producer and consumer share a common, fixed size buffer. Producer puts information into the buffer, and consumer, takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep until consumer has removed one or more items and wakes up it. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. This approach sounds simple enough, but it leads to the race conditions. Consider the following code fragment:

```
#define N 100 /* number of slots in the buffer */ int
count = 0; /* number of items in the buffer */ void
producer (void)
{
        int item;
        while (TRUE) /* repeat forever */
        {
                item = produce_item(); /* generate next item */
                if (count == N) sleep(); /* if buffer is full, go to sleep */
                insert_item(item); /* put item in buffer */
                count = count + 1; /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* Wakeup consumer*/
        }
}
void consumer(void)
{
        int item;
        while (TRUE) /* repeat forever */
        {
                if (count == 0) sleep(); /* if buffer is empty, got to sleep */
                item = remove_item(); /* take item out of buffer */
                count = count - 1; /* decrement count of items in buffer */
                if (count == N - 1) wakeup(producer); /* wakeup producer */
                consume_item(item);
        }
}
```

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

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

Semaphore

Concept of semaphore was devised by Dijkstra in 1965. Semaphore is an integer variable that is used to record number of wakeups that had been saved. If it is equal to zero it indicates that no wakeup's are saved. A positive value shows that one or more wakeup's are pending. The DOWN operation (equivalent to sleep) checks the semaphore to see if it is greater than zero. If it is, it decrements the value (using up a stored wakeup) and continues. If the semaphore is zero the process sleeps. The UP operation(Equivalent to wakeup) increments the value of the semaphore. If one or more processes were sleeping on that semaphore then one of the processes is chosen and allowed to complete its DOWN. Checking and updating the semaphore must be done as an ***atomic*** action to avoid race conditions. Producer Consumer problem can be solved by using semaphore as below:

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

```
    while (TRUE) /* TRUE is the constant 1 */
    {
        item = produce_item(); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex);/* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) /* infinite loop */
    {
        down(&full);/* decrement full count */
        down(&mutex);/* enter critical region */
        item a= remove_item();/* take item from buffer */
        up(&mutex);/* leave critical region */
        up(&empty);/* increment count of empty slots */
        consume_item(item);/* do something with the item */
    }
}
```

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores.**

The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed. The *full* and *empty* semaphores are needed to guarantee synchronization. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty.

Monitors

With semaphores interprocess communication looks easy in first sight. But we should be too much careful about order of down operation. Suppose that the two downs in the producer's code were reversed in order, so *mutex* was decremented before *empty*. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on *mutex*. Since *mutex* is already 0, consumer is also blocked too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is *called* a deadlock.

Above problem shows that one subtle error causes everything to come to a grinding halt. To make it easier to write correct programs, Hoare and Brinch Hansen proposed a higher-level synchronization primitive called a **monitor** . A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. When a process calls a monitor procedure, the first few instructions of the procedure will check to see, if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. A skeleton of the producer-consumer problem with monitors is given below:

```
monitor ProducerConsumer
        function insert (item : integer );
        function remove : integer ;


    end monitor ;

    procedure producer ;
        begin
        while true do
        begin
        item = produce_item ;
        ProducerConsumer .insert (item )
        end;
    end ;
    procedure consumer ;
        begin
        while true do
        begin
        item = ProducerConsumer .remove ;
        consume_item (item )
        end;
    end ;
```

Operations wait and signal *are* very similar to sleep and wakeup, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just before the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.

Message Passing

This method of interprocess communication uses two primitives, send and receive , which, like semaphores and unlike monitors, are system calls rather than language constructs.

- ✓ send(destination, &message): It sends a message to a given destination

- ✓ receive (source, &message): It receives a message from a given source. If no message is available, the receiver can block until one arrives.

Design Issues for Message Passing Systems

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example

- ✓ Messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.
- ✓ If the message itself is received correctly but the acknowledgement is lost, the sender will retransmit the message, so the receiver will get it twice. Usually, his problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.
- ✓ Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous.
- ✓ **Authentication** is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?
- ✓ When the sender and receiver are on the same machine, copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor.

The Producer-Consumer Problem with Message Passing

We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of *N* messages are used. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. If the producer works faster than the consumer, all the messages will be filled up, and the producer will be blocked, waiting for an empty message to come back. If the consumer works faster, then the reverse happens: all the messages will be empties and the consumer will be blocked, waiting for a full message.

```
#define N 100 /* number of slots in the buffer */
void producer(void)
```

```
{
          int item;
          message m;              // message buffer
          while (TRUE)
          {
                    item = produce_item( );       // generate something to put in buffer
                    receive(consumer, &m);       // Receive an empty message, if any
                    build_message (&m, item);   // construct a message to send
                    send(consumer, &m);          // send item to consumer
          }
}

void consumer(void)
{
          int item, i;
          message m;
          for (i = 0; i < N; i++)
          send(producer, &m);          // send N empty messages
          while (TRUE)
          {
                    receive(producer, &m);       //get message containing item
                    item = extract_item(&m);    // extract item from message
                    send(producer, &m);          // send back empty reply
                    consume_item(tem);           // do something with the item
          }
}
```

Classical IPC Problems

The Producer-Consumer Problem: Assume there are a producer (which produces goods) and a consumer (which consumes goods). The producer produces goods and places them in a fixed size buffer. The consumer takes the goods from the buffer. The buffer has a finite capacity so that if it is full, the producer must stop producing. Similarly, if the buffer is empty, the consumer must stop consuming. This problem is also referred to as the *bounded buffer* problem. The type of situations we must cater for are when the buffer is full, so the producer cannot place new items into it. Another potential problem is when the buffer is empty, so the consumer cannot take from the buffer.

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

The Readers Writers Problem: This problem was devised by. It models a number of processes requiring access to a database. Any number of processes may read the database but only one can write to the database. The problem is to write a program that ensures this happens.

The Sleeping Barber Problem: A barber shop consists of a waiting room with *n* chairs. There is another room that contains the barber's chair. The following situations can happen.

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

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

Scheduling

When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler** and the algorithm it uses is called the **scheduling algorithm.**

Compute-Bound Vs I/O-Bound Processes

Some processes spend most of their time in computing. Such processes are called **compute-bound** processes. But some spend most of their time waiting for I/O. such processes are called **I/O bound** processes. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits. As CPUs get faster, processes tend to get more I/O bound. This effect occurs because CPUs are improving much faster than disks. As a consequence, the scheduling of I/O-bound processes is likely to become a more important subject in the future. The basic idea here is that if an I/O-bound process wants to run, it should get a chance quickly so it can issue its disk request and keep the disk busy.

Preemptive vs Non-preemptive Scheduling

A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks or until it voluntarily releases the CPU. Even if it runs for hours, it will not be forceably suspended. Scheduling the next process to run would simply be a case of taking the highest priority job or using some other algorithm, such as FIFO algorithm.

In contrast, a **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). If no clock is available, nonpreemptive scheduling is the only option.

Categories of Scheduling Algorithms

In different environments different scheduling algorithms are needed. This situation arises because different application areas have different goals. The scheduler optimized for one may not be optimal for another area. Three environments worth distinguishing are

- ✓ **Batch:** In batch systems, there are no users impatiently waiting at their terminals for a quick response. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process are often acceptable. This approach reduces process switches and thus improves performance.
- ✓ **Interactive:** In an environment with interactive users, preemption is essential to keep one process from monopolizing the CPU and denying service to the others. Preemption is needed to prevent this behavior.
- ✓ **Real time:** In real time systems normally preemption is not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand.

Scheduling Algorithm Goals

All systems

- ✓ Fairness - Under all circumstances, fairness is important. Processes having similar priority should get should get comparable service. Giving one process much more CPU time than an equivalent one is not fair.
- ✓ Policy enforcement - seeing that stated policy is carried out
- ✓ Balance - All parts of the system must be busy when possible. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second than if some of the components are idle,

Batch systems

- ✓ Throughput - **Throughput** is the number of jobs per hour that the system completes. All things considered, finishing 50 jobs per hour is better than finishing 40 jobs per hour.
- ✓ Turnaround time - **Turnaround time** is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output. Here the rule is: Small is Beautiful.
- ✓ CPU utilization - On the big mainframes where batch systems run, the CPU is still a major expense. Thus computer center managers feel guilty when it is not running all the time therefore one goal of batch system is to keep the CPU busy all the time

Interactive systems

- ✓ Response time - **Response time** is the time between issuing a command and getting the result. One main goal of interactive systems is to minimize response time.
- ✓ Proportionality - When a request that is perceived as complex takes a long time, users accept that, but when a request that is perceived as simple takes a long time, users get irritated. Proportionality measures how well the users' expectations are met.

Real-time systems

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

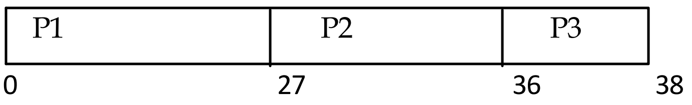
Scheduling Algorithms

First Come - First Served Scheduling (FCFS)

An obvious nonpreemptve scheduling algorithm is to execute the processes in the order they arrive and to execute them to completion. It is an easy algorithm to implement. When a process becomes ready it is added to the tail of ready queue and when the CPU becomes free the process at the head of the queue is removed, moved to a running state and allowed to use the CPU until it is completed. The problem with FCFS is that the average waiting time can be long. Consider the following processes

Process	**Burst Time**	**Arrival time**
P1	27	1
P2	9	2
P3	2	4

Gantt chart



Average waiting =((0 + 27 +36) /3)=21

Average turnaround time = ((27+ 36 +38) /3)= 33.66

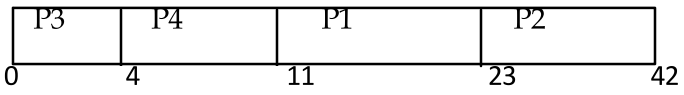
FCFS algorithm has some undesirable effects. A CPU bound job may make the I/O bound (once they have finished the I/O) wait for the processor. At this point the I/O devices are sitting idle. Again, when the CPU bound job finally does some I/O, the I/O-bound processes use the CPU quickly and now the CPU sits idle waiting for the CPU bound job to complete its I/O. Thus FCFS can lead to I/O devices and the CPU both being idle for long periods.

Shortest Job First

SJF is another non-preemptive batch algorithm that assumes the run times are known in advance. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest **job first**. SJF is the optimal scheduling algorithm that gives minimum average waiting time. The problem of SJF is we do not know the burst time of a process before it starts. For some systems (notably batch systems) we can make fairly accurate estimates but for interactive processes it is not so easy. Consider the following processes.

Process	Burst Time	Arrival Time
P1	12	0
P2	19	0
P3	4	0
P4	7	0

Gantt chart



Average waiting =((0 + 4 + 11 + 23) /4)= 9.5
Average turnaround time = ((4+ 11 + 23 + 42) /4)= 20

Challenge!!!
Construct Gantt chart and then calculate average waiting time and average turnaround time for processes given in table below.

Process	Burst Time	Arrival Time
P1	8	0
P2	19	0
P3	4	3
P4	7	5

Shortest Remaining Time Next (SRTF)
A preemptive version of shortest job first is **shortest remaining time next.** With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

Round-Robin Scheduling

One of the oldest, simplest, fairest, and most widely used algorithms is **round robin.** Each process is assigned a time interval, called its **quantum,** which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes. When the process uses up its quantum, it is put on the end of the list.

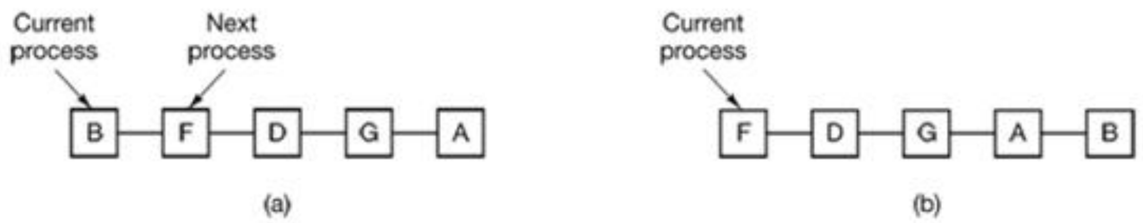
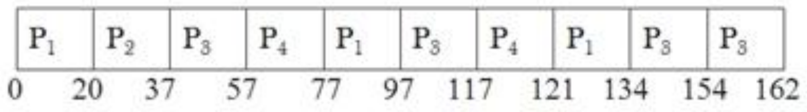


Figure Round-robin scheduling. (a) The list of runnable processes. (b) The list of runna processes after *B* uses up its quantum.

The interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time. Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. Large value of quantum may cause Round Robin scheduler may cause it to behave like FCFS. Typically RR gives higher average turnaround time, but better response time. Consider the following processes with time quantum set to 20.

Process	**Burst Time**	**Arrival Time**
P1	53	0
P2	17	0
P3	68	0
P4	24	0

Gantt chart



Avg. turnaround time= (134+37+162+121)/4 = 113.5
Avg. waiting time=(81+20+94+97)/4=73

Challenge!!!
Construct Gantt chart and then calculate average waiting time and average turnaround time for processes given in table below. {Assume Quantum size=5}

Process	**Burst Time**	**Arrival Time**
P_1	9	0
P_2	17	1
P_3	23	2
P_4	12	16

Priority Scheduling
Round robin scheduling makes the implicit assumption that all processes are equally important. On a PC, there may be multiple processes, some more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time. The need to take external factors into account leads to **priority scheduling.** The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

One of the problems with priority scheduling is that some processes may never run. There may always be higher priority jobs that get assigned the CPU. This is known as ***indefinite blocking*** or ***starvation***. One solution to this problem is scheduler may decrease the priority of the currently running process at each clock tick. If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

Multiple Queue Scheduling
There are two typical processes in a system, *interactive jobs* which tend to be shorter and *batch jobs* which tend to be longer. We can set up different queues to cater for different process types. Each queue may have its own scheduling algorithm - the background queue will typically use the FCFS algorithm while the interactive queue may use the RR algorithm. The scheduler has to decide which queue to run. Either higher priority queues can be processed until they are empty before the lower priority queues are executed or each queue can be given a certain amount of the CPU. There could be other

queues in addition to the two mentioned. Multilevel Queue Scheduling assigns a process to a queue and it remains in that queue. It may be advantageous to move processes between queues (multilevel feedback queue scheduling). If we consider processes with different CPU burst characteristics, a process which uses too much of the CPU will be moved to a lower priority queue. We would leave I/O bound and (fast) interactive processes in the higher priority queues.

Working

Assume three queues (Q_0, Q_1 and Q_2)

- ✓ Scheduler executes Q_0 and only considers Q_1 and Q_2 when Q_0 is empty ✓ A Q_1 process is preempted if a Q_0 process arrives
- ✓ New jobs are placed in Q_0
- ✓ Q_0 runs with a quantum of 8ms
- ✓ If a process Q_0 uses full quantum it is placed at the end of the Q_1 queue ✓ Q_1 has a time quantum of 16ms associated with it
- ✓ Any processes preempted in Q_1 are moved to Q_2, which is FCFS

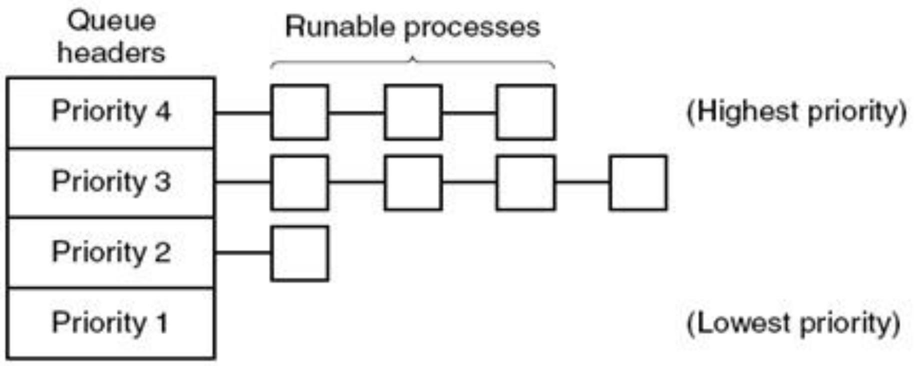


Figure: A scheduling algorithm with four priority classes

Chapter 3
Process Deadlocks
Deadlock Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a **deadlock**. System is deadlocked if there is a set of processes such that every process in the set is waiting for another process in the set. Or "A *set of procedures is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*" for example:

✓ Process A makes a request to use the scanner ✓
Process A given the scanner
✓ Process B requests the CD writer
✓ Process B given the CD writer
✓ Now A requests the CD writer
✓ A is denied permission until B releases the CD writer ✓
Alas now B asks for the scanner
✓ Result a DEADLOCK!

A system consists of a finite number of resources to be distributed among a number of processes. These resources are partitioned into several types, each of which consists of some number of identical resources. A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

☞ **Request:** If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

☞ **Use:** The process can operate on the resource.

☞ **Release:** The process releases the resource.

Deadlock Characterization

Necessary Condition: According to Coffman, a deadlock can arise if the following four conditions hold simultaneously.

☞ **Mutual exclusion:** Only one process at time can use the resource. If a process requests the recourse that is used by another process, the requesting process must be delayed until the resource has been released.

☞ **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

☞ **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task.

☞ **Circular wait:** A set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

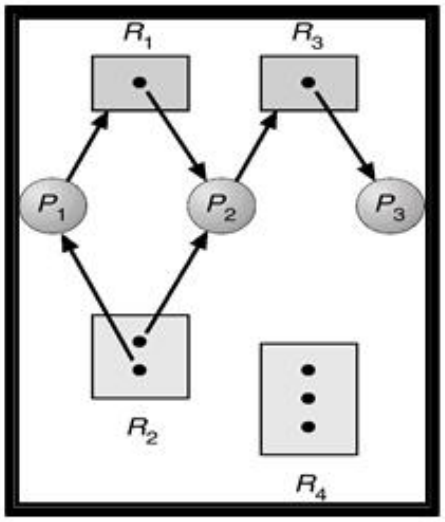
Resource - Allocation Graph: Deadlocks can be described more precisely in terms of a directed graph called a system **resource-allocation graph**. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices V is partitioned two different types of nodes:

☞ $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

☞ $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by P_i □ R_j and it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. This directed edge is called **request edge**. A directed edge from resource type R_j to process P_i is denoted by R_j □ P_i and it signifies that an instance of resource type R_j has been allocated to process P_i. This directed edge is called an **assignment edge**.

Pictorially, we represent each process as a circle, and each resource type as a square. Since resource type may have more than one instance, we represent each such instance as a dot within the square. A request edge points to only the square, whereas an assignment edge must also designate one of the dots in the square. The figure below shows resource-allocation graph.



This resource-allocation graph depicts the following situation.

☞ The set P, R, and E

- ✓ P = {P1, P2, P3}
- ✓ R = {R1, R2, R3, R4}
- ✓ E = {P1 R1, P2 R3, R1 P2, R2 P2, R2 P1, R3 P3} ☞

Resource instances:

- ✓ One instance of resource type R1
- ✓ Two instances of resource type R2
- ✓ One instance of resource type R3
- ✓ Three instances of resource type R4 ☞

Process states:

- ✓ Process p1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1
- ✓ Process P2 is holding an instance of R1 and R2, and is waiting for an resource type R3
- ✓ Process P3 is holding an instance of R3

If graph contains no cycles, no deadlock occurs. But, if the graph contains a cycle and only one instance per resource is available, then deadlock occurs. Also, if the graph contains a cycle and there are several instances per resource type, then there is possibility of deadlock. The figure (a) below shows the resource allocation graph with a deadlock and the figure (b) shows the resource allocation graph with a cycle but no deadlock.

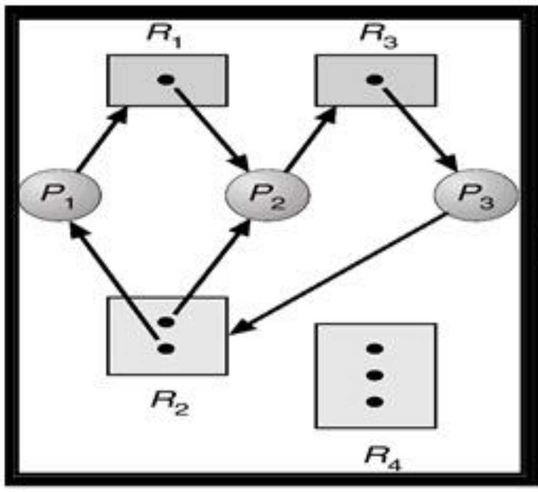
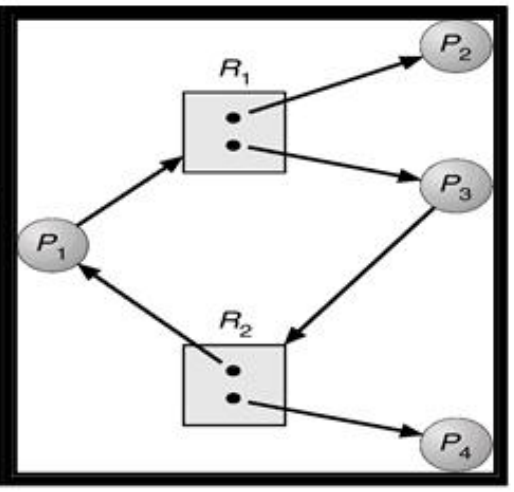


Figure (a):Deadlock



Figure(b):No Deadlock

Handling Deadlocks

We can deal with deadlock problems in one of the four ways:

a. **Ignore the Problem (Ostrich Algorithm):** We can ignore the deadlock problem altogether. If the deadlock occurs, the system will stop functioning and will need to be restarted manually. This approach is used in systems in which deadlocks occur infrequently and if it is not a life critical system. This method is cheaper than other methods. This solution is used by most operating systems, including UNIX.
b. **Deadlock prevention:** Deadlocks can be prevented deadlocks by constraining how requests for resources can be made in the system and how they are handled. The goal is to ensure that at least one of the Coffman's necessary conditions for deadlock can never hold.
c. **Deadlock avoidance**: To avoid deadlocks the system dynamically considers every request and decides whether it is safe to grant it at this point. The system requires additional apriori information regarding the overall potential use of each resource for each process.
d. **Deadlock Detection and Recovery:** We can allow the system to enter a deadlock state, detect it, and recover - If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock.

Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions.

Elimination of "Mutual Exclusion" Condition: The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition: One approach to achieve this is that a process request be granted all of the resources it needs at once, prior to execution. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of "No-preemption" Condition: This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of "Circular Wait" Condition: the circular wait condition, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as below:

1	≡	Card reader
2	≡	Printer
3	≡	Plotter
4	≡	Tape drive
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

Deadlock Avoidance

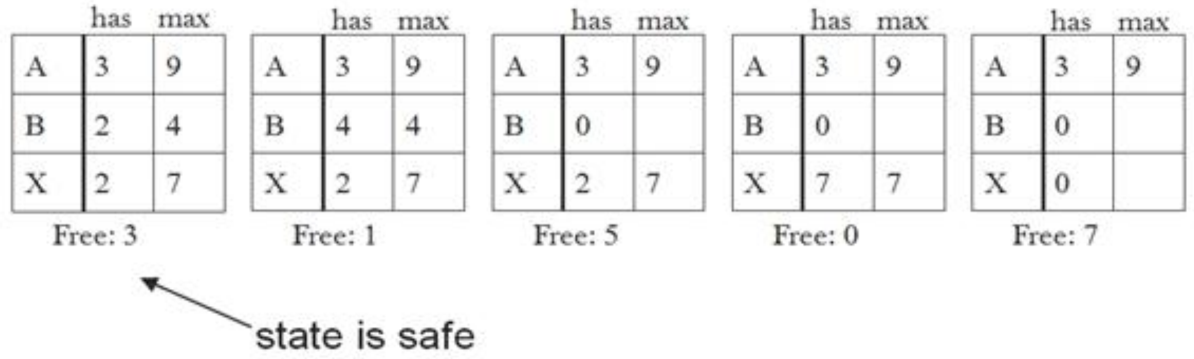
Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will cause the system to enter an *unsafe* state. The system then only grants requests that will lead to *safe* states. In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time:

- ✓ Resources currently available
- ✓ Resources currently allocated to each process
- ✓ Resources that will be required and released by these processes in the *future*

It is possible for a process to be in an unsafe state but for this not to result in a deadlock. The notion of safe/unsafe states only refers to the *ability* of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock. One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance. However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is impossible practically.

Safe and Unsafe States: When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if *there exists a sequence <P_1, P_2, …, P_n> of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.*

Example: Assume we have 10 resources with single instance of each



	has	max
A	3	9
B	2	4
X	2	7

Free: 3

	has	max
A	3	9
B	4	4
X	2	7

Free: 1

	has	max
A	3	9
B	0	
X	2	7

Free: 5

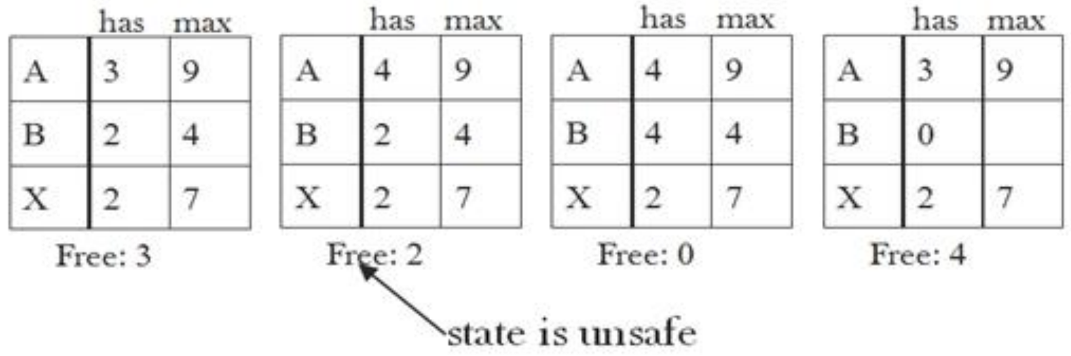
	has	max
A	3	9
B	0	
X	7	7

Free: 0

	has	max
A	3	9
B	0	
X	0	

Free: 7

In the above figure, if we look at last matrix, process A can also continue and complete its work. From the situation depicted in first matrix it is possible for each of the process to complete it's work. One possible sequence is shown in above figure (i.e B,C,A). Thus the state depicted in first matrix is **"Safe State"**.



	has	max
A	3	9
B	2	4
X	2	7

Free: 3

	has	max
A	4	9
B	2	4
X	2	7

Free: 2

	has	max
A	4	9
B	4	4
X	2	7

Free: 0

	has	max
A	3	9
B	0	
X	2	7

Free: 4

In the above figure, if we look at last matrix, none of the processes (i.e. process A or X) can continue and complete its work. This situation is called deadlock. From the situation depicted in second matrix it is not possible for each of the process to complete it's work. No possible sequence exists so that processes can complete their work. Thus the state depicted in second matrix is **"Unsafe State"**.

Banker'sAlgorithm: The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Dijkstra. The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm avoids deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may claim during its life. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Bankers Algorithm for Single Instance of Each Resource:

1. Read maximum resources need by each process
2. Read number of available resources
3. Construct Allocated vector
4. Construct Need vector
5. Accept resource request from a process
6. If granting request leads to unsafe sate
    - Deny or postpone the request
7. Else
    - Grant requested resources
    - If process has all resources needed by it

      Available =Available + Resources held by the process
8. Repeat step 5-7 until all processes are fisnished

Example

Assume that we have 10 resources available. What happens if processes request resources in following order? Assume Four processes A,B,C, and D.

- Process 'A' request one resource
- Process 'B' request one resource
- Process 'C' request two resource
- Process 'D' request four resource
- Process 'B' request one resource

Assume that initially processes holds no resources.

Solution

Initially:

Process	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free=10

Process 'A' request one resource: Leads to safe sate therefore **grant** resources

Process	Has	Max
A	1	6
B	0	5
C	0	4
D	0	7

Free=9

Process 'B' request one resource: **:** Leads to safe sate therefore **grant** resources

Process	Has	Max
A	1	6
B	1	5
C	0	4
D	0	7

Free=8

Process 'C' request two resource: **:** Leads to safe sate therefore **grant** resources

Process	Has	Max
A	1	6
B	1	5
C	2	4
D	0	7

Free=6

Process 'D' request four resource: **:** Leads to **safe** sate therefore **grant** resources

Process	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free=2

Process 'B' request one resource: **:** Leads to **unsafe** sate therefore **deny** request

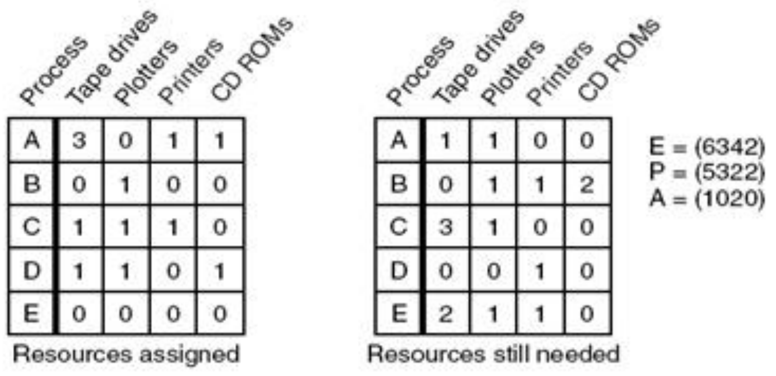
Process	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free=9

B**ankers Algorithm for Multiple Instances of Each Resource Type:** Let n be the number of processes in the system and m be the number of resource types. Then we need the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type R_j available.
- **Max:** A $n \times m$ matrix defines the maximum demand of each process. If Max[i,j] = k, then P_i may request at most k instances of resource type R_j.
- **Allocation:** An n×m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process P_i is currently allocated k instance of resource type R_j.
- **Need:** An n×m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then P_i may need k more instances of resource type R_j to complete task.

Note: Need = Max - Allocation.



Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Here,

E=>Existing Resources, P=Possessed Resources, A=Available Resources

Algorithm for checking to see if a state is safe:

1. Look for row, R, whose unmet resource needs all ≤ A. If no such row exists, system will eventually deadlock since no process can run to completion.
2. Assume process of row chosen requests all resources it needs and finishes. Mark process as terminated, add all its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes marked terminated (initial state was safe) or no process left whose resource needs can be met (there is a deadlock).

Example

Considering the following initial state calculate need matrix.

Table: Resources Assigned

Process	Tape Drives	Plotters	Printers	CD Drives
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Table: Max Resources Needed

Process	Tape Drives	Plotters	Printers	CD Drives
A	4	1	1	1
B	0	2	1	2
C	4	2	1	0
D	1	1	1	1
E	2	1	1	0

Available (A)={1,0,2,0}   Existing(E)={6,3,4,2}

What happens if processes requests resources in following order?

- Process 'B' requests one Printer.
- Process 'E' requests one Printer.

Solution:

Since, Need = Max - Allocation

Need matrix can be given as below:

Resource Needed

Process	Tape Drives	Plotters	Printers	CD Drives
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

- If Process 'B' requests one Printer, Situation becomes like below:

Table: Resources Assigned

Process	Tape Drives	Plotters	Printers	CD Drives
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Table: Resources Needed

Process	Tape Drives	Plotters	Printers	CD Drives
A	1	1	0	0
B	0	0	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Available (A)={1,0,1,0}

System is again in **safe** state, since all processes can complete their task if resources are assigned in sequence {D, E, A, B, C}. Therefore printer is **granted** to process B

- **If Process 'E' requests one Printer, Situation becomes like below:**

Table: Resources Assigned

Process	**Tape Drives**	**Plotters**	**Printers**	**CD Drives**
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	1	0

Available (A)={1,0,0,0}

Table: Max Resources Needed

Process	**Tape Drives**	**Plotters**	**Printers**	**CD Drives**
A	1	1	0	0
B	0	0	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	0	0

Now the system is in **unsafe** state, since No process can continue with available resources. Therefore request for printer is **denied** to process E.

Deadlock Detection and Recovery

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide: An algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock. Deadlock detection algorithm can be studied in following two sections

- Deadlock Detection with one Resource of Each Type
- Deadlock Detection with Multiple Resource of Each Type

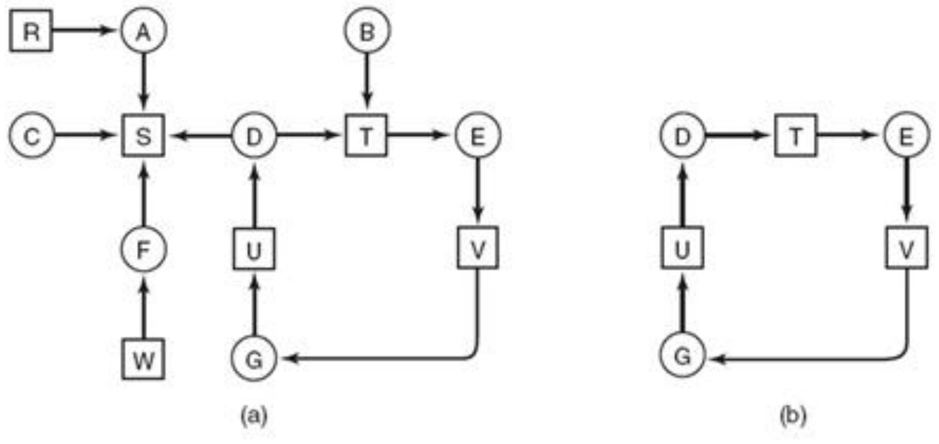
Deadlock Detection with one Resource of Each Type: The deadlock detection algorithm for systems with a single instance of each resource type is based on the graph theory. The idea is to find cycles in the resource allocation graph, which represents the circular-wait condition If cycle is found, it indicates the existence of deadlocks.

Algorithm for detecting deadlock:

1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, designate all arcs as unmarked.

3. Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.
4. From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

Example



Working of algorithm

1. [R, A]
2. [R, A, S] - since S has no outgoing branch (dead-end), back track to A, then to R and Re-start in new section with an empty list [ ] and move to try node C
3. [C,S] - Again, since S has no outgoing branch (dead-end), back track to C, and then Re-start in new section with an empty list [ ] and move to try node B
4. [B]
5. [B, T] ..... [B, T, E] .....[B, T, E, V, G, U, D] D can goto S or T; S we have seen before so dead-end, and backtrack to D, and then it tries T
6. [B, T, E, V, G, U, D, T] - T has appeared twice, so we have a "DEADLOCK"

Deadlock Detection with Multiple Resources of Each Type: The resourceallocation graph scheme is not applicable deadlock detection in a system with multiple instances of each resource type. The algorithm here employs several time-varying data structures that are similar to those used in the banker's algorithm.

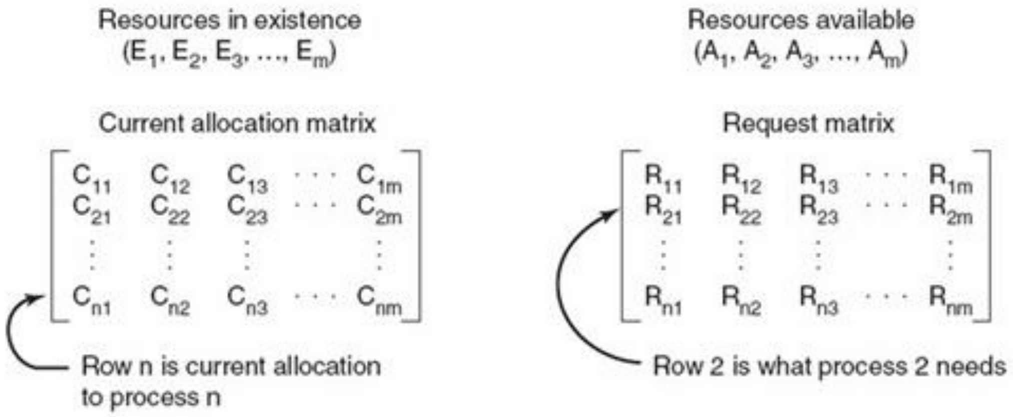

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$
Current allocation matrix
$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$
Row n is current allocation to process n
Resources available
$(A_1, A_2, A_3, \ldots, A_m)$
Request matrix
$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$
Row 2 is what process 2 needs

Fig: Four data structures needed by the deadlock detection algorithm

- ✓ **Existing:** A vector of length m indicates the number of existing resources of each type.
- ✓ **Available:** A vector of length m indicates the number of available resources of each type.
- ✓ **Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process.
- ✓ **Request:** An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P_i is requesting k more instances of resource type. R_j.

Deadlock detection algorithm:

1. Look for an unmarked process, P_i , for which the i-th row of *R* is less than or equal to *A*.
2. If such a process is found, add the *i-th* row of *C* to *A*, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

Example

Consider three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive. Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the *R* matrix.

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array} \qquad A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 1 & 0 & 0) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no CD-ROM drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

A = (2 2 2 0)

At this point process 2 can run and return its resources, giving

A = (4 2 2 1)

Now the remaining process can run. Thus there is no deadlock in the system.

Now consider a minor variation of the situation given above. Suppose that process 2 needs a CD-ROM drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system is deadlocked.

Recovery from Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially attractive, however.

Recovery through Preemption: In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible.

Recovery through Rollback: When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost. In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery through Killing Processes: The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken. Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources.

Chapter 4

Memory Management

Introduction to Memory Management

One of the main tasks of an operating system is to manage the computer's memory. This includes many responsibilities, for example:

1. Being aware of what parts of the memory are in use and which parts are not.
2. Allocating memory to processes when they request it and de-allocating memory when a process releases its memory.
3. Moving data from memory to disc, when the capacity of physical memory becomes full, and vice versa.
4. Managing hierarchical memory e.g. volatile cache -> RAM ->Disk storage

Mono-programming vs. Multi-programming

If we only allow a single process in memory at a time we can make life simple for ourselves. Using this model we do not have to worry about swapping processes out to disc when we run out of memory. Nor do we have to worry about keeping processes separate in memory. All we have to do is load a process into memory, execute it and then unload it before loading the next process. In this day and age mono-programming is unacceptable as multi-programming is not only expected by the users of a computer but it also allows us to make more effective use of the CPU. For example, we can allow a process to use the CPU whilst another process carries out I/O or we can allow two people to run interactive jobs and both receive reasonable response times.

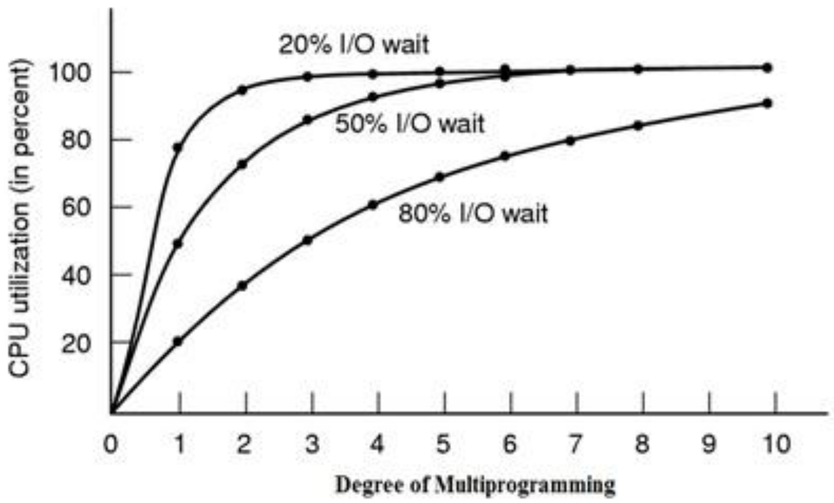
We could allow only a single process in memory at one instance in time and still allow multiprogramming. This means that a process, when in a running state, is loaded in memory. When a context switch occurs the process is copied from memory to disc and then another process is loaded into memory. This method allows us to have a relatively simple memory module in the operating system but still allows multi-programming. The drawback with the method is the amount of time a context switch takes. If we assume that a quantum is 100ms and a context switch takes 200ms then the CPU spends a disproportionate amount of time switching processes. We could increase the amount of time for a quantum but then the interactive users will receive poor response times as processes will have to wait longer to run.

Modeling Multiprogramming

If we have five processes that use the processor twenty percent of the time (spending eighty percent doing I/O) then we should be able to achieve one hundred percent CPU utilization. Of course, in reality, this will not happen as there may be times when all five processes are waiting for I/O. However, it seems reasonable that we will achieve better than twenty percent utilization that we would achieve with mono-programming.

We can build a model from a probabilistic viewpoint. Assume that that a process spends *p* percent of its time waiting for I/O. With *n* processes in memory the probability that all *n* processes are waiting for I/O (meaning the CPU is idle) is p_n. The CPU utilization is then given by:

CPU Utilization = 1 - p_n



You can see that with an I/O wait time of 20%, almost 100% CPU utilization can be achieved with four processes. If the I/O wait time is 90% then with ten processes, we only achieve just above 60% utilization. The important point is that, as we introduce more processes the CPU utilization rises.

Multi-programming with Fixed Partitions

If we use multi-programming, we next need to decide how to organize the available memory in order to make effective use of the resource. One method is to divide the memory into fixed sized partitions. These partitions can be of different sizes but once a partition has taken on a certain size then it remains at that size. There is no provision for changing its size.

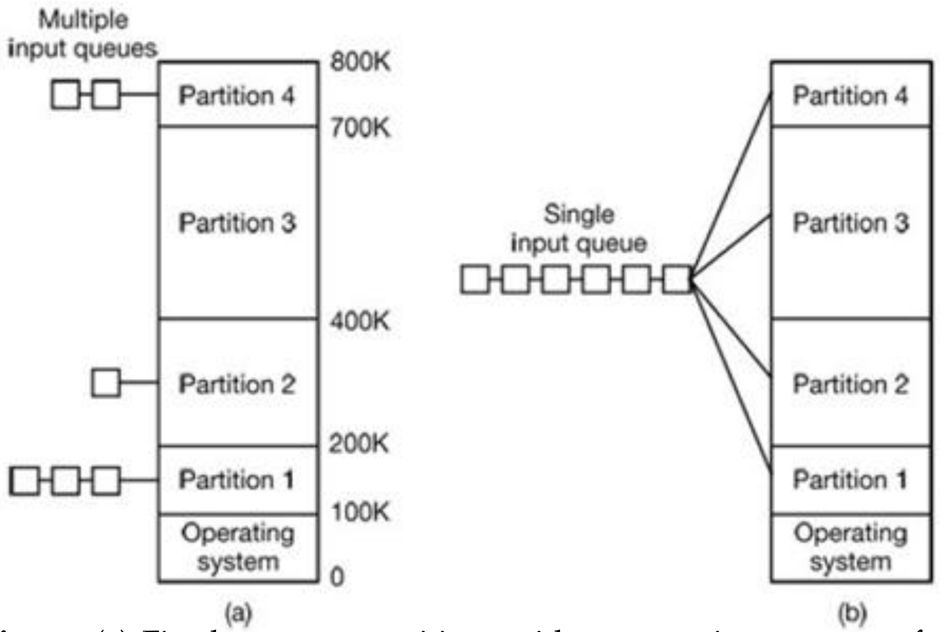


Figure: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

The diagram above shows how this scheme might work. The memory is divided into four partitions. When a job arrives it is placed in the input queue for the smallest partition that will accommodate it. There are a few drawbacks to this scheme.

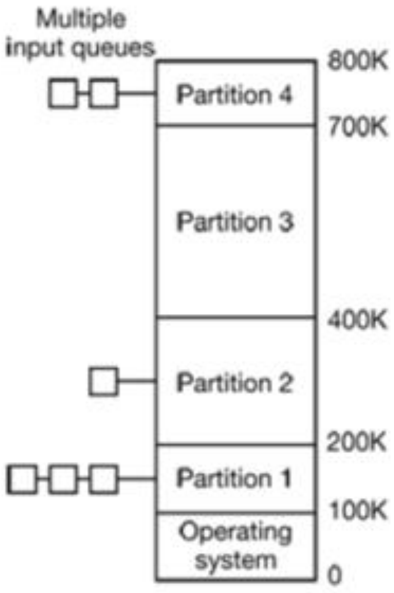
- ✓ As the partition sizes are fixed, any space not used by a particular job is lost.
- ✓ It is possible that small jobs have to wait to get into memory, even though plenty of memory is free.

To cater for the last problem we could have a single input queue where all jobs are held. When a partition becomes free we search the queue looking for the first job that fits into the partition. Since it is undesirable to waste a large partition on a small job, an alternative search strategy is to search the entire input queue looking for the largest job that fits into the partition. This has the advantage that we do not waste a large partition on a small job but has the disadvantage that smaller jobs are discriminated against. Smaller jobs are typically interactive jobs which we normally want to service first. To ensure small jobs do get run we could have at least one small partition or ensure that small jobs only get skipped a certain number of times.

Relocation and Protection

As soon as we introduce multiprogramming we have two problems that we need to address.

Relocation: When a program is run it does not know in advance what location it will be loaded at. Therefore, the program cannot simply generate static addresses. Instead, they must be made relative to where the program has been loaded. For example, suppose that the first instruction is a call to a procedure at absolute address 50 within the binary file produced by the linker. If this program is loaded at address 100K, that instruction will jump to absolute address 50, which is inside the operating system. What is needed is a call to 100K + 50.



Protection: Once you can have two programs in memory at the same time there is a danger that one program can write to the address space of another program. This is obviously dangerous and should be avoided.

In order to cater for relocation we could make the loader modify all the relevant addresses as the binary file is loaded. This scheme suffers from the following problems:

- ✓ The program cannot be moved, after it has been loaded without again modifying all addresses
- ✓ Using this scheme does not help the protection problem as the program can still generate *illegal* addresses
- ✓ The program needs to have some sort of map that tells the loader which addresses need to be modified.

A solution, which solves both the relocation and protection problem, is to equip the machine with two registers called the ***base*** and ***limit*** registers. The base register stores the start address of the partition and the limit register holds the length of the partition. Any address that is generated by the program has the base register added to it. In addition, all addresses are checked to ensure they are within the range of the partition. An additional benefit of this scheme is that if a program is moved within memory, only its base registers needs to be altered. This is obviously a lot quicker than having to modify every address reference within the program.

Multiprogramming with Variable Partitions

Memory is a scare resource and, as we have discussed, fixed partitions can be wasteful of memory. Again, in a timesharing situation where many people want to access the computer, more processes will need to be run than can be fitted into memory at the same time. Therefore it would be a good idea to move towards ***variable partition*** sizes. That is partitions that can change size as the need arises. Variable partitions can be summed up as follows...

- ✓ The number of partition varies.
- ✓ The sizes of the partitions varies
- ✓ The starting addresses of the partitions varies

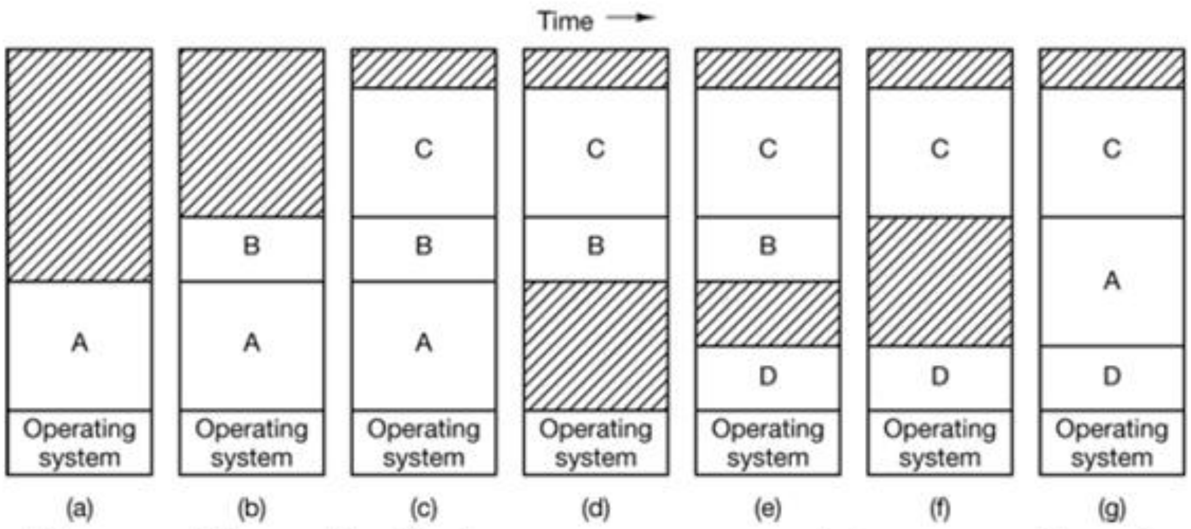


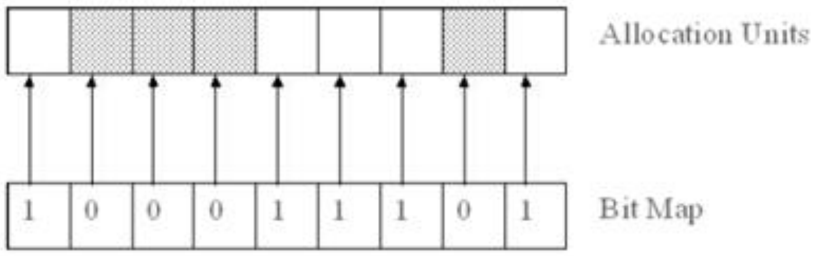
Figure Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Variable partitions are good for much more effective memory management system but it makes the process of maintaining the memory much more difficult. For example, as memory is allocated and deallocated *holes* will appear in the memory and it will become fragmented. Eventually, there will be *holes* that are too small to have a process allocated to it. We could simply *shuffle* all the memory being used downwards called ***memory compaction***, thus closing up all the holes. But this could take a long time and, for this reason it is not usually done frequently.

Variable partitions should be used with **swapping**. This is the strategy that consists of bringing in each process in its entirety, running it for a while, and then putting it back on the disk.

Memory Management with Bitmaps

Under this scheme the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.



The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory due to internal fragmentation. For a memory of size 1 GB and allocation unit of 4byte, size of bitmap will be 32 MB.

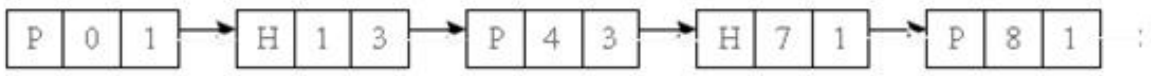
The other problem with a bit map memory scheme is when we need to allocate memory to a process. Assume the allocation size is 4 bytes. If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes. This is a slow operation and for this reason bit maps are not often used.

Memory Management with Linked Lists

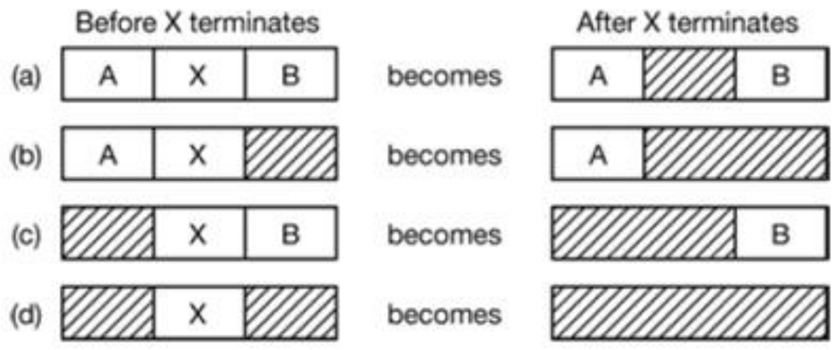
Free and allocated memory can be represented as a linked list. Each entry in the list holds the following data

- ✓ P or H : for Process or Hole
- ✓ Starting segment address
- ✓ The length of the memory segment
- ✓ The next pointer is not shown but assumed to be present

The memory shown above as a bit map can be represented as a linked list as follows.



In the list above, processes follow holes and vice versa. But, it does not have to be this way. It is possible that two processes can be next to each other and we need to keep them as separate elements in the list so that if one process ends we only return the memory for that process. Consecutive holes, on the other hand, can always be merged into a single list entry. This leads to the following observations when a process terminates and returns its memory.



- ✓ In the first option we simply have to replace the P by an H, other than that the list remains the same.
- ✓ In the second option we merge two list entries into one and make the list one entry shorter.
- ✓ Option three is effectively the same as option 2.
- ✓ For the last option we merge three entries into one and the list becomes two entries shorter.

In order to implement this scheme it is normally better to have a doubly linked list so that we have access to the previous entry.

Memory Allocation Strategies

When we need to allocate memory, storing the list in segment address order allows us to implement various strategies.

- ✓ **First Fit:** This algorithm searches along the list looking for the first segment that is large enough to accommodate the process. The segment is then split into a hole and a process. This method is fast as the first available hole that is large enough to accommodate the process is used.

- ✓ **Next Fit:** It is minor variation of first fit. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

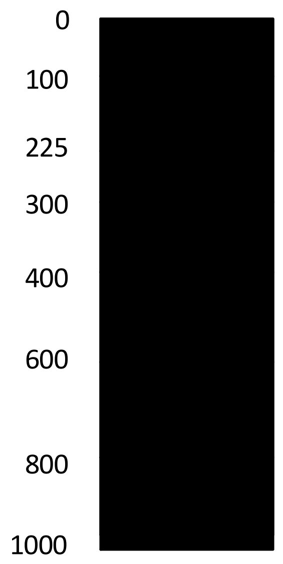
- ✓ **Best Fit:** Best fit searches the entire list and uses the smallest hole that is large enough to accommodate the process. The idea is that it is better not to split up a larger hole that might be needed later. Best fit is slower than first fit as it must search the entire list every time. It has also be shown that best fit performs worse than first fit as it tends to leave lots of small gaps.

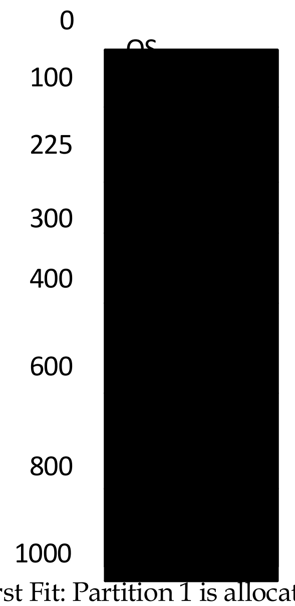
- ✓ **Worst Fit:** As best fit leaves many small, useless holes it might be a good idea to always use the largest hole available. The idea is that splitting a large hole into two will leave a large enough hole to be useful. It has been shown that this algorithm is not very good either.

These algorithms can all be speeded up if we maintain two lists; one for processes and one for holes. This allows the allocation of memory to a process to be speeded up as we only have to search the hole list. The downside is that list maintenance is complicated. If we allocate a hole to a process we have to move the list entry from one list to another. However, maintaining two lists allows us to introduce another optimization. If we hold the hole list in size order (rather than segment address order) we can make the best fit algorithm stop as soon as it finds a hole that is large enough. In fact, first fit and best fit effectively becomes the same algorithm.

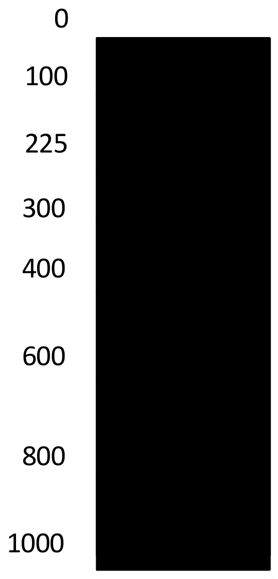
The Quick Fit algorithm takes a different approach to those we have considered so far. Separate lists are maintained for some of the common memory sizes that are requested. For example, we could have a list for holes of 4K, a list for holes of size 8K etc. One list can be kept for large holes or holes which do not fit into any of the other lists. Quick fit allows a hole of the right size to be found very quickly, but it suffers in that there is even more list maintenance.

Example
Consider a memory given below. If a new process 'D' of size 50 k requests memory, which partition should be allocated to the process with first fit strategy? Which partition should be allocated to the process, if we use best fit or worst fit strategy?

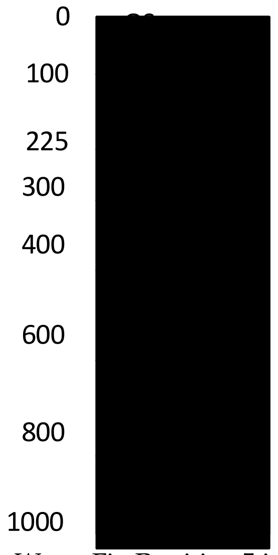




First Fit: Partition 1 is allocated
125k memory is wasted if fixed
partition is used



Best Fit: Partition 2 is allocated
15k memory is wasted if fixed
partition is used



Worst Fit: Partition 5 is allocated
200k memory is wasted if fixed
partition is used

Virtual Memory

The swapping methods we have looked at above are needed so that we can allocate memory to processes when they need it. But what happens when we do not have enough memory? The solution usually adopted was to split the program into pieces, called **overlays**. Overlay 0 would start running first. When it was done, it would call another overlay.

The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed. Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into logical sections had to be done by the programmer. This was time consuming, boring and open to error.

Solution to above problem is to use concept of virtual memory that is devised by Fotheringham, 1961. Virtual memory is the technique that gives a computer ability to run programs even if the amount of physical memory is not sufficient to allow the program and all its data to reside in memory at the same time. For example, we can run a 2048K program on a 256K machine. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. We can also use virtual memory in a multiprogramming environment. For example, we can run twelve programs in a machine that could run only four programs without virtual memory.

Paging

In a computer system that does not support virtual memory, when a program generates a memory address it is placed directly on the memory bus which causes the requested memory location to be accessed. On a computer that supports virtual memory, the address generated by a program goes via a ***memory management unit*** (MMU). This unit maps virtual addresses to physical addresses.

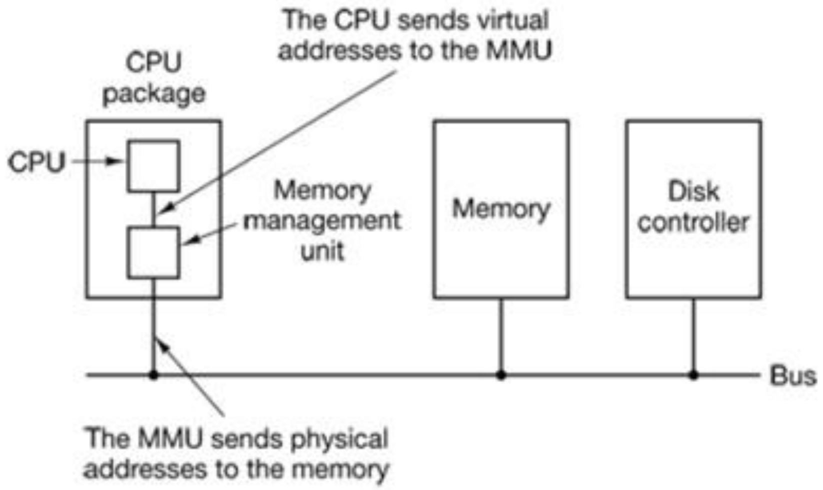
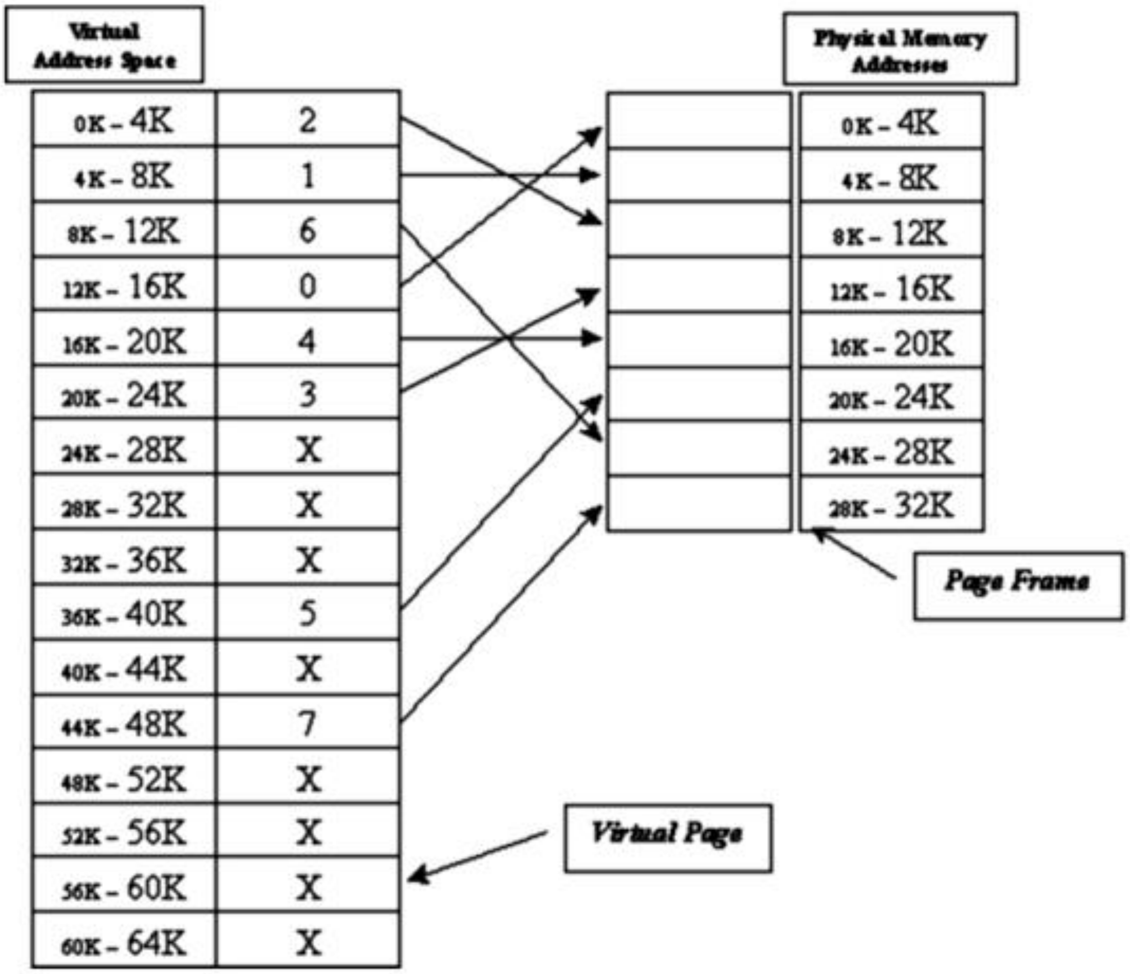

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure . The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was in years gone by.

This diagram below shows how virtual memory operates. The computer in this example can generate 16-bit addresses. That is addresses between 0 and 64K (0-65536). The problem is the computer only has 32K of physical memory so although we can write programs that can access 64K of memory; we do not have the physical memory to support that. We obviously cannot fit 64K into the physical memory available so we have to store some of it on disc. The virtual memory is divided into ***pages***. The physical memory is divided into ***page frames***. The size of the virtual pages and the page frames are the same size (4K in the diagram above). Therefore, we have sixteen virtual pages and eight physical pages. Transfers between disc and memory are done in pages.

Virtual addresses generated by CPU can be logically divided into two parts page number and offset. Page number is used to identify the page of virtual memory to which virtual address belongs and offset is used to identify the particular by within the given virtual page. Similarly, physical addresses generated by MMU can be logically divided into two parts frame number and offset.



Now let us consider what happens when a program generates a request to access a memory location 8192.This address is sent to the MMU. The MMU recognizes that this address falls in virtual page 2 (because 8192/4096=2). The MMU looks at its page mapping and sees that page 2 maps to physical page 6. The MMU translates 8192 to the relevant address in physical address 24576 (because 6*4096= 24576). This address is output by the MMU and the memory board simply sees a request for address 24576. If a virtual memory address is not on a page boundary (as in the above example) then the MMU also has to calculate an offset (in fact, there is always an offset - in the above example it was zero). Since virtual address space is large than physical address space, some of the virtual pages are not mapped to physical page frames. When CPU generates the virtual address whose page number is not mapped to the page frame number in physical memory, **page fault** is generated and the page needs to be read from disk and stored in memory.

Question Using the diagram above, work out the physical page and physical address that are generated by the MMU for the virtual address 45060.

Solution

Given,

Page size= Frame Size= 4 KB=4096 bytes

Virtual Address=45060

Now,

Page Number= 45060/4096=11

Offset=45060%4096=4

Thus,

Frame Number= 7

Physical Address=4096*7+4=28676

Page Tables

Page table is a data structure that holds information (such as page number, offset, present/absent bit, modified bit etc) about virtual pages. The purpose of the page table is to map virtual pages onto page frames. Thus, mathematically we can say that the page table is a function, with the virtual page number as argument and the physical frame number as result.

The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (it any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory. This process of converting virtual address into physical address is shown schematically below:

The way we have described how virtual addresses map to physical addresses is how it works but we still have a couple of problems to consider.

- ☞ **The page table can be large:** Assume a computer uses virtual addresses of 32 bits (very common) and the page size is 4K. This results in over 1 million pages (2^{32} / 4096 = 1048576). If size of an entry is 4 byte then the space needed by a page table is 1048576*4=4194304 byte (4 MB). Again, each process needs its own page table (because it has its own virtual address space).
- ☞ **The mapping between logical and physical addresses must be fast:** A typical instruction requires two memory accesses (to fetch the instruction and to fetch the data that the instruction will operate upon). Therefore, it requires two page table

accesses in order to find the right physical location. Assume an instruction takes 10ms. The mapping of the addresses must be done in a fraction of this time (around 20% of instruction time) in order for this part of the system not to become a bottleneck.

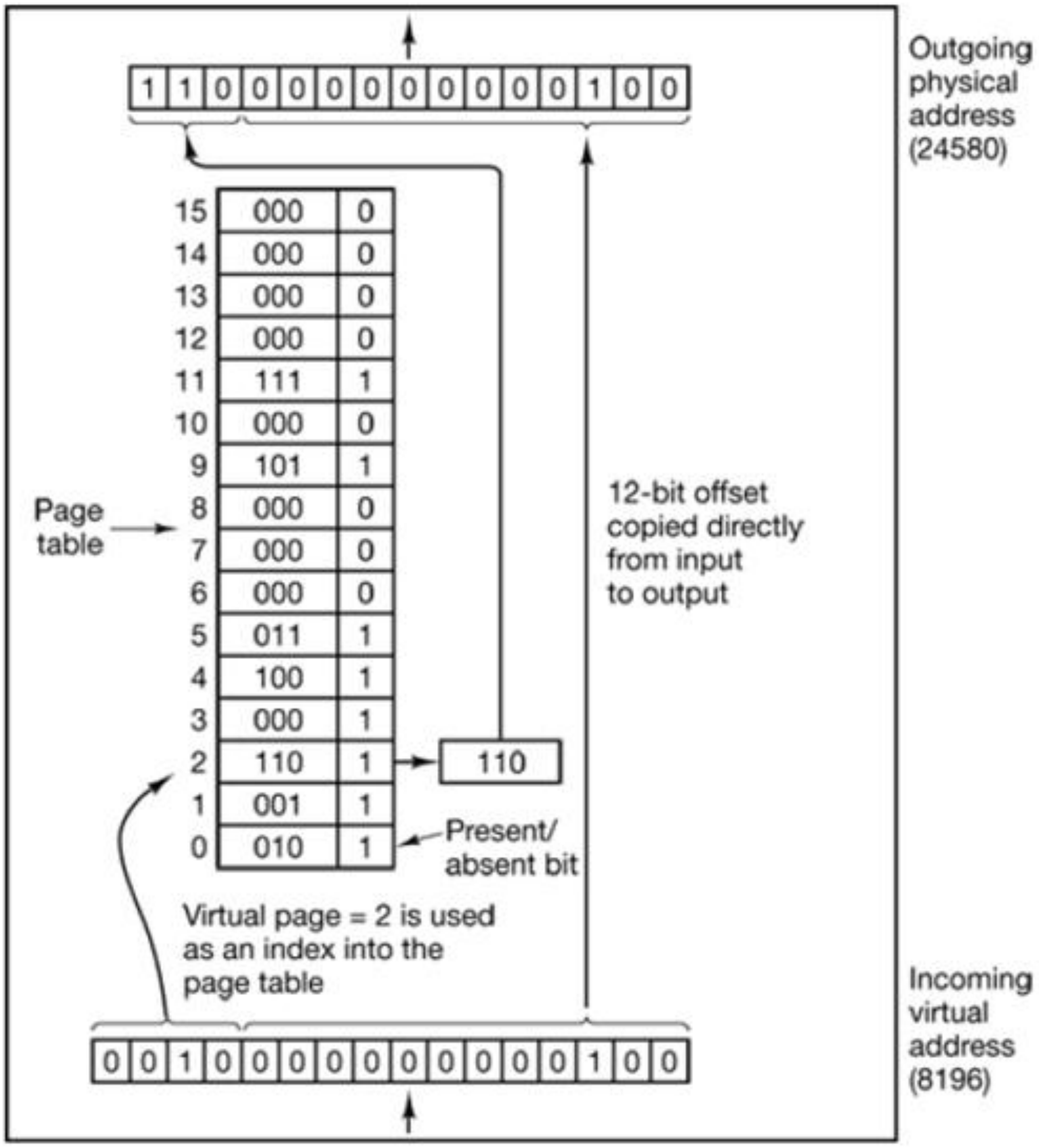
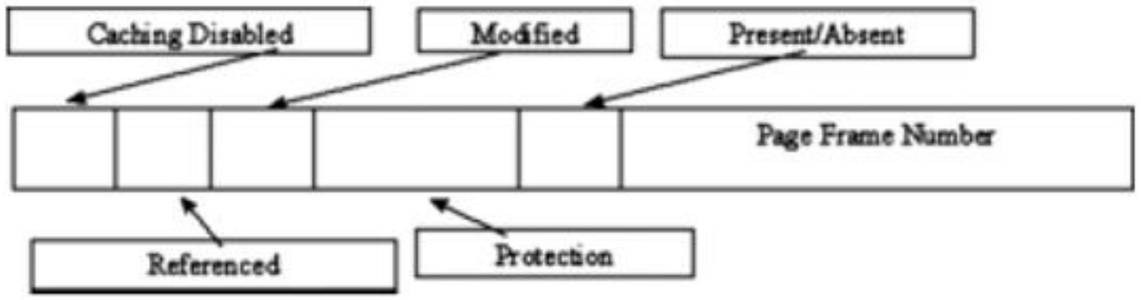


Figure The internal operation of the MMU with 16 4-KB pages.

Structure of Page Table

The above diagram shows typical entries of a page table. Although the exact entries are operating system dependent, the various components are described below.



- ☞ ***Page Frame Number:*** This is the number of the physical page that this page maps to. As this is the whole point of the page, this can be considered the most important part of the page frame entry.
- ☞ ***Present/Absent Bit:*** This indicates if the mapping is valid. A value of 1 indicates the physical page, to which this virtual page relates is in memory. A value of zero indicates the mapping is not valid and a page fault will occur if the page is accessed.
- ☞ ***Protection:*** The protection bit could simply be a single bit which is set to 0 if the page can be read and written and 1 if the page can only be read. If three bits are allowed then each bit can be used to represent read, write and execute (R,W,E).
- ☞ ***Modified:*** This bit is updated if the data in the page has been modified. This bit is used when the data in the page is evicted. If the modified bit is set, the data in the page frame needs to be written back to disc. If the modified bit is not set, then the data can simply be evicted, in the knowledge that the data on disc is already up to date.
- ☞ ***Referenced:*** This bit is updated if the page has been referenced. This bit can be used when deciding which page should be evicted (we will be looking at its use later).
- ☞ ***Caching Disabled:*** This bit allows caching to be disabled for the page. This is useful if a memory address maps onto a device register rather than to a memory address. In this case, the register could be changed by the device and it is important that the register is accessed, rather than using the cached value which may not be up to date.

Page Fault Handling

Page fault occurs when the program tries to use an unmapped page. When CPU presents virtual address to the MMU, it calculates virtual page number and looks at that index of page table. First of all MMU checks for present/absent bit, if the bit value is zero (unmapped) MMU will cause trap to the operating system. This trap is called a

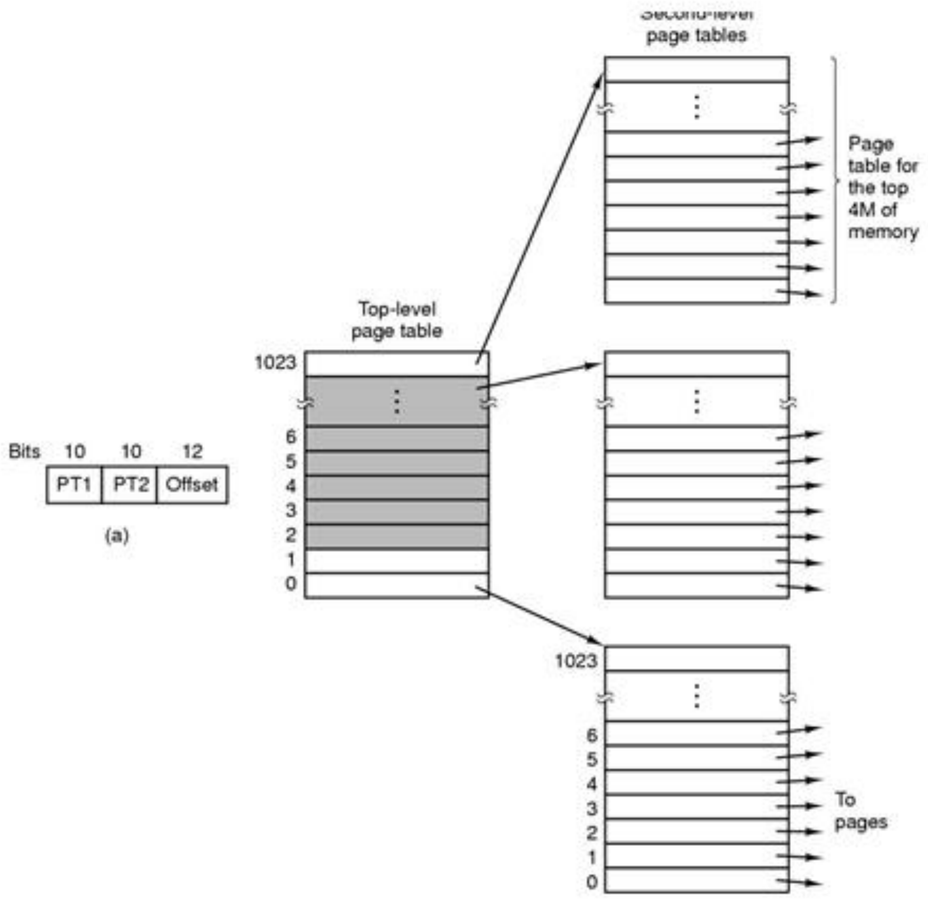
page fault. The operating system would decide to evict one of the currently mapped pages and use that for the page that has just been referenced. The sequence of events would go like this.

- ☞ The hardware traps to the kernel, saving the program counter on the stack. On most machines, some information about the state of the current instruction is saved in special CPU registers.
- ☞ An assembly code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it. This routine calls the operating system as a procedure.
- ☞ The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed.
- ☞ Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access. If not, the process is sent a signal or killed. Otherwise system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to select a victim.
- ☞ If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.
- ☞ As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.
- ☞ When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in normal state.
- ☞ The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.
- ☞ The faulting process is scheduled, and the operating system returns to the assembly language routine that called it.
- ☞ This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

Multilevel Page Tables

If the virtual address space is large, the page table will be large. Nowadays 32 bit addresses are used and 64 bits becoming more common. With 4-KB page size, 32-bit address space has 2^{20}=1048576 pages which require 2^{20}=1048576 page table entries. Again each process needs its own page table. In such situation large amount of memory space is occupied by page table. To solve this problem multi-level page tables can be used.

Main idea behind page tables is to avoid keeping the entire page table in memory because it is too big. For this hierarchy of page tables can be used. The hierarchy is a page table of page tables. For example, 32-bit virtual address can be partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4 KB and there are a total of 2^{20} such pages. Now for a process, if top level of page table contains: Entry 0 points to pages for program text, Entry 1 points to pages for data, and Entry 1023 points to pages for stack, Only 4 page tables need to be kept into memory.



When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M. The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

Consider an example, for the virtual address 4206596, how physical address is calculated?
One entry in top level page table represents 4M
Thus,

Top level index=4206596/4194304=1
Offset for top level page table= (4206596%4194304)=12292

One entry in second level page table represents 4K
Thus,

2nd Level PT Index=12292/4096= 3
Offset=12292%4096=4

Now the entry in index 3 of selected second level page table contains the page frame number of the page containing virtual address 4206596. If that page is not in memory, the *Present/absent* bit in the page table entry will be zero, causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address. This address is put on the bus and sent to memory.

Page Replacement Algorithms

In the discussion above we said that when a page fault occurs we evict a page that is currently mapped and replace it with the page that we are currently trying to access. In doing this, we need to write the page we are evicting to disc, if it has been modified. However, we have not said how we decide which page to evict. One of the easiest methods would be to choose a mapped page at random but this is likely to lead to degraded system performance. This is because the page chosen has a reasonable chance of being a page that will need to be used again in the near future. Therefore, it will be evicted (and may have to be written to disc) and then brought back into memory;

maybe on the next instruction. In this section we describe some of the page replacement algorithms.

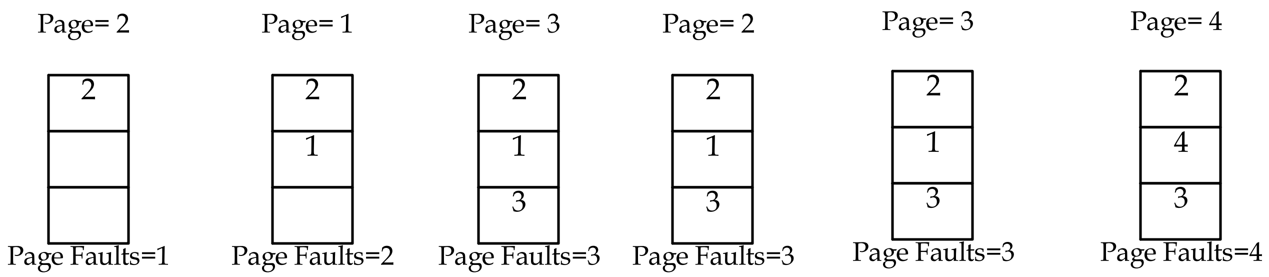
Optimal Page Replacement Algorithm

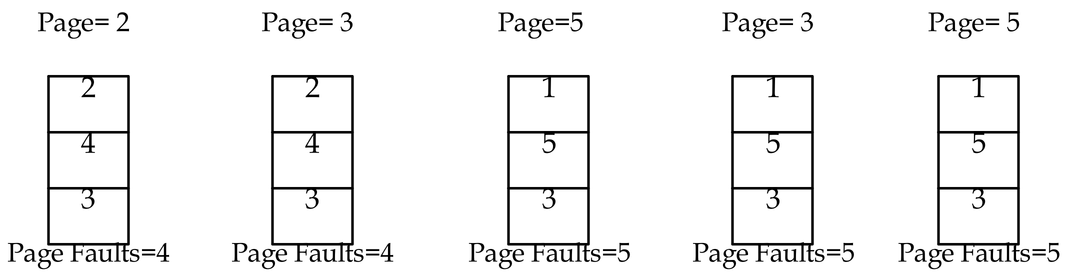
Ideally we want to select an algorithm with the lowest page-fault rate. Such an algorithm exists, and is called the optimal algorithm. Main idea behind page replacement algorithm is to replace the page that will not be used for the longest time. This means optimal page replacement algorithm replaces the page with the greatest forward distance in the reference string.

Example

Assume Memory Size=3

Reference String: 2 1 3 2 3 4 2 3 5 3 5





Note: In third last step we can evict either page 2 or page 4 randomly

The problem associated with optimal page replacement algorithm is that we cannot look into the future and decide which page to evict. So this algorithm is not practically feasible. But, if we cannot implement the algorithm then why bother discussing it? In fact, we can implement it, but only after running the program to see which pages we should evict and at what point. Once we know that we can run the optimal page replacement algorithm. We can then use this as a measure to see how other algorithms perform against this ideal.

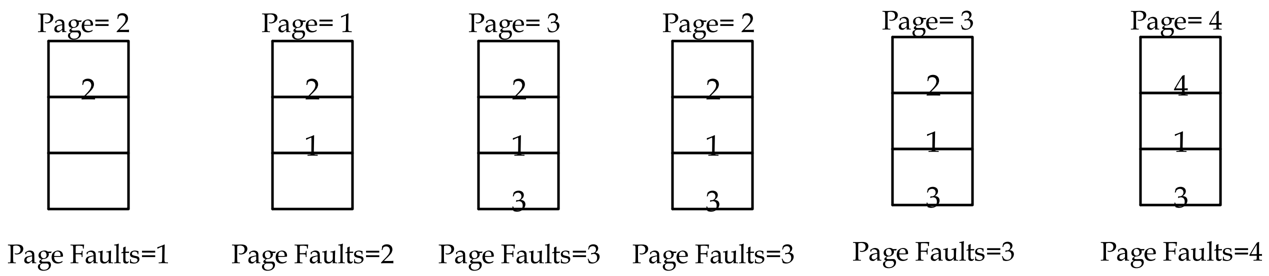
First-In First-Out (FIFO)

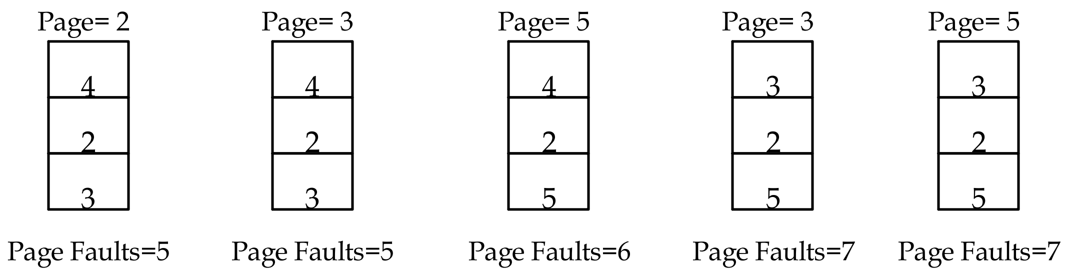
This algorithm simply maintains the pages as a linked queue, with new pages being added to the end of the list. When a page fault occurs, the page at the head of the list (the oldest page) is evicted. Whilst simple to understand and implement a simple FIFO algorithm do not lead to good performance as a heavily used page is just as likely to be evicted as a lightly used page.

Example

Assume Memory Size=3

Reference String: 2 1 3 2 3 4 2 3 5 3 5





Not Recently Used (NRU)

NRU is the page replacement algorithm that makes the use of Reference (R) and Modified (M) bit of page table to make page replacement decision. Main concept behind NRU page replacement policy is to give more priority to clean pages than dirty pages while making the decision of page eviction. When a page is referenced R (reference) bit is set and when a page is written to, the M (modified) bit is set. When a process is started up all R and M bits are cleared (set to zero). Periodically (e.g. on each clock interrupt) the R bit is cleared. This allows us to recognize which pages have been recently referenced. When a page fault occurs (so that a page needs to be evicted), the pages are inspected and divided into four categories based on their R and M bits.

Class 0 : *Not Referenced, Not Modified*
Class 1 : *Not Referenced, Modified*
Class 2 : *Referenced, Not Modified*
Class 3 : *Referenced, Modified*

The Not-Recently-Used (NRU) algorithm removes a page at random from the lowest numbered class that has entries in it. Therefore, pages which have not been referenced or modified are removed in preference to those that have not been referenced but have been modified.

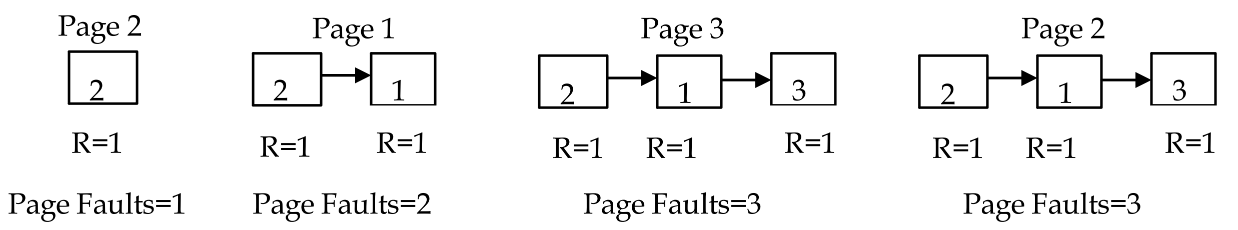
The Second Chance Page Replacement Algorithm

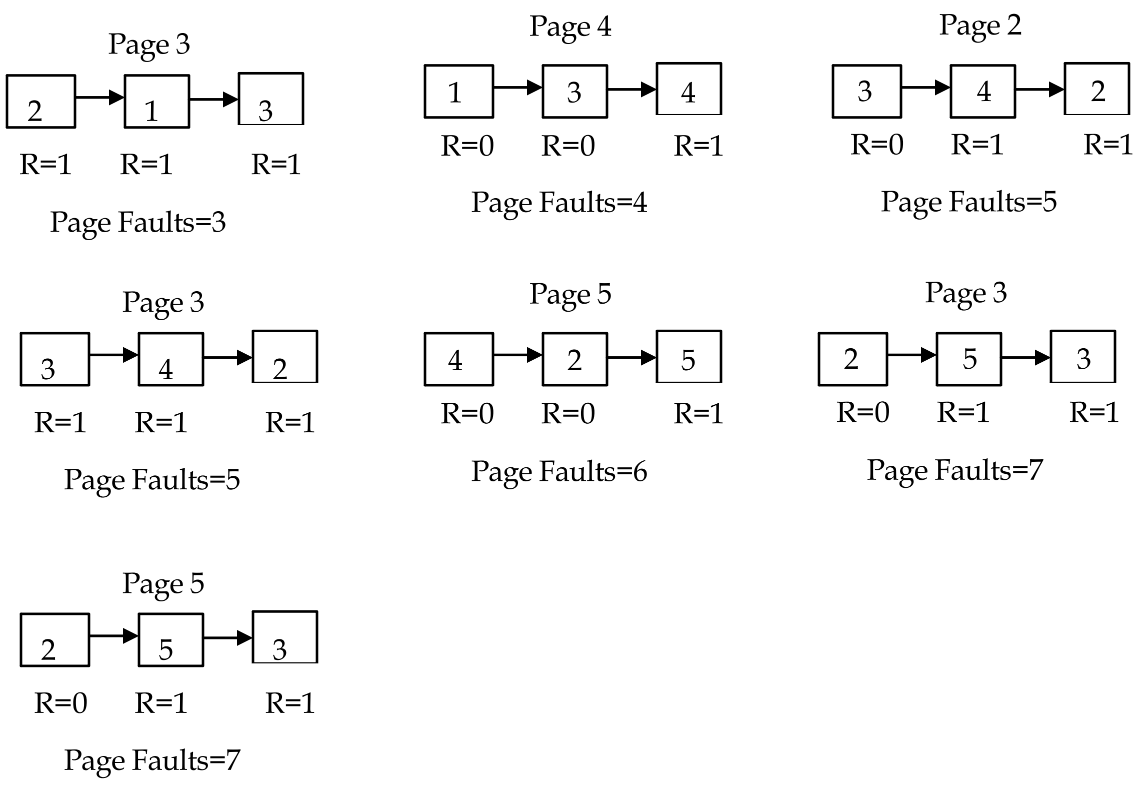
The second chance (SC) algorithm is a modification of the FIFO algorithm. When a page fault is occurred the page at the front of the linked queue is inspected. If it has not been referenced (i.e. it reference bit is clear), it is evicted. If its reference bit is set, then it is placed at the rear (end) of the queue and its reference bit reset. The next page is then inspected. In this way, a page that has been referenced will be given a second chance. Again, when page hit occurs the page is placed at the rear of queue. In effect we should keep the linked queue sorted on the basis of load time. R-bit is set when a page is referenced.

In the worst case, SC operates in the same way as FIFO. Take the situation where the linked queue consists of pages which all have their reference bit set. The first page, call it *a*, is inspected and placed at the end of the queue, after having its R bit cleared. The other pages all receive the same treatment. Eventually page *a* reaches the head of the queue and is evicted as its reference bit is now clear. Therefore, even when all pages in a queue have their reference bit set, the algorithm will always terminate.

Example

Assume Memory Size=3
Reference String: 2 1 3 2 3 4 2 3 5 3 5





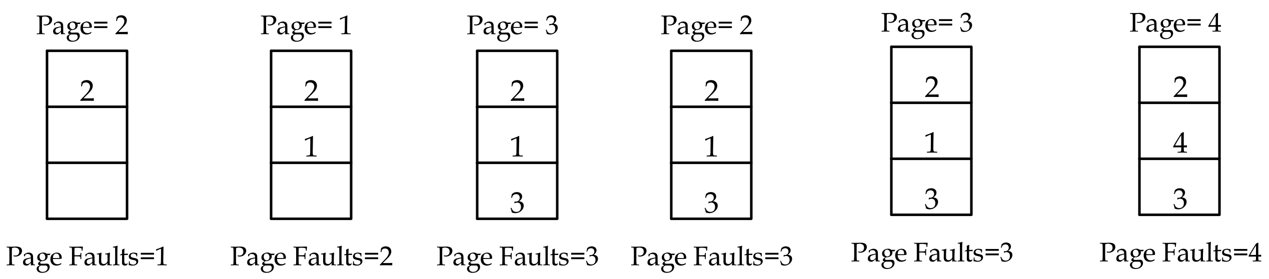
Least Recently Used Page Replacement (LRU) Algorithm

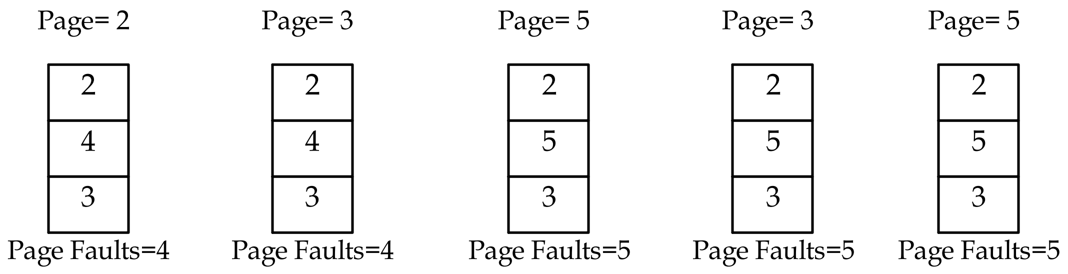
LRU page replacement algorithm is the best approximation of optimal page replacement algorithm on the based on observation of past behavior. Main idea behind LRU algorithm is that if a page has recently been used then it is likely that it will be used again in the near future. Conversely, a page that has not been used for some time is unlikely to be used again in the near future. Therefore LRU evicts the page that has not been used for the longest amount of time in the future. This means LRU replaces the page with the greatest backward distance in the reference string. Main advantage of LRU is that it can adopt along with change in working set. LRU is the best choice for the memory references having strong locality.

Example

Assume Memory Size=3

Reference String: 2 1 3 2 3 4 2 3 5 3 5





Although LRU is theoretically very good, it is expensive to implement in reality. To implement LRU, it is necessary to maintain a linked queue of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty of LRU is that the list must be updated on every memory reference, even if at the time of page hit. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation. Performing such time consuming operations at the time of page hit also cannot be considered as good idea.

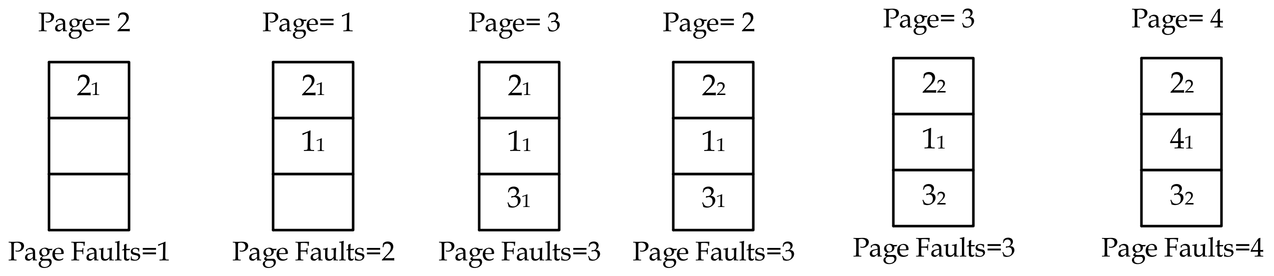
Least Frequently Used (LFU) Page Replacement

Concept behind LFU page replacement policy is to replace the page which has been referenced least often. For each page in the reference string, we need to keep a *reference count.* All reference counts start at 0 and are incremented every time a page is referenced. Main drawback of LFU policy is that it cannot adopt according to change in working set because it never forgets anything. LFU can make memory polluted by keeping the pages in memory that are not in use currently but have large frequency count due to their heavy use in the past. In the example given below, numbers shown in superscript are frequency counts of pages.

Example

Assume Memory Size=3

Reference String: 2 1 3 2 3 4 2 3 5 3 5



Page= 2	Page= 3	Page= 5	Page= 3	Page= 5
2_3	2_3	2_3	2_3	2_3
4_1	4_1	5_1	5_1	5_2
3_2	3_3	3_3	3_4	3_4
Page Faults=4	Page Faults=4	Page Faults=5	Page Faults=5	Page Faults=5

Clock Page Replacement

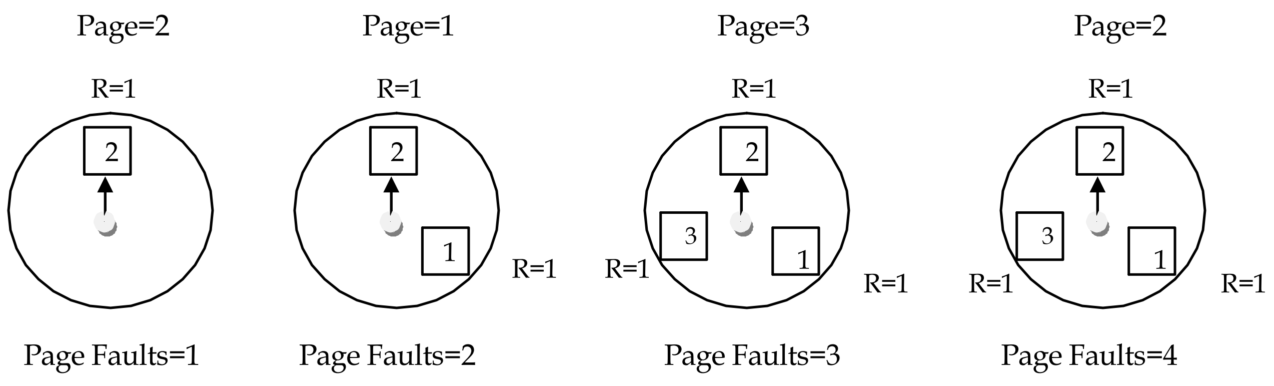
Clock page replacement policy is modified version of second chance page replacement algorithm. It differs from second chance policy only in its implementation. Whilst second chance is a reasonable algorithm it suffers in the amount of time it has to devote to the maintenance of the linked queue. It is more efficient to hold the pages in a circular list and move the pointer rather than moving page from front of the queue to the rear of the queue. It is called the clock page algorithm as it can be visualized as a clock face with the hand (pointer) pointing at the oldest page.

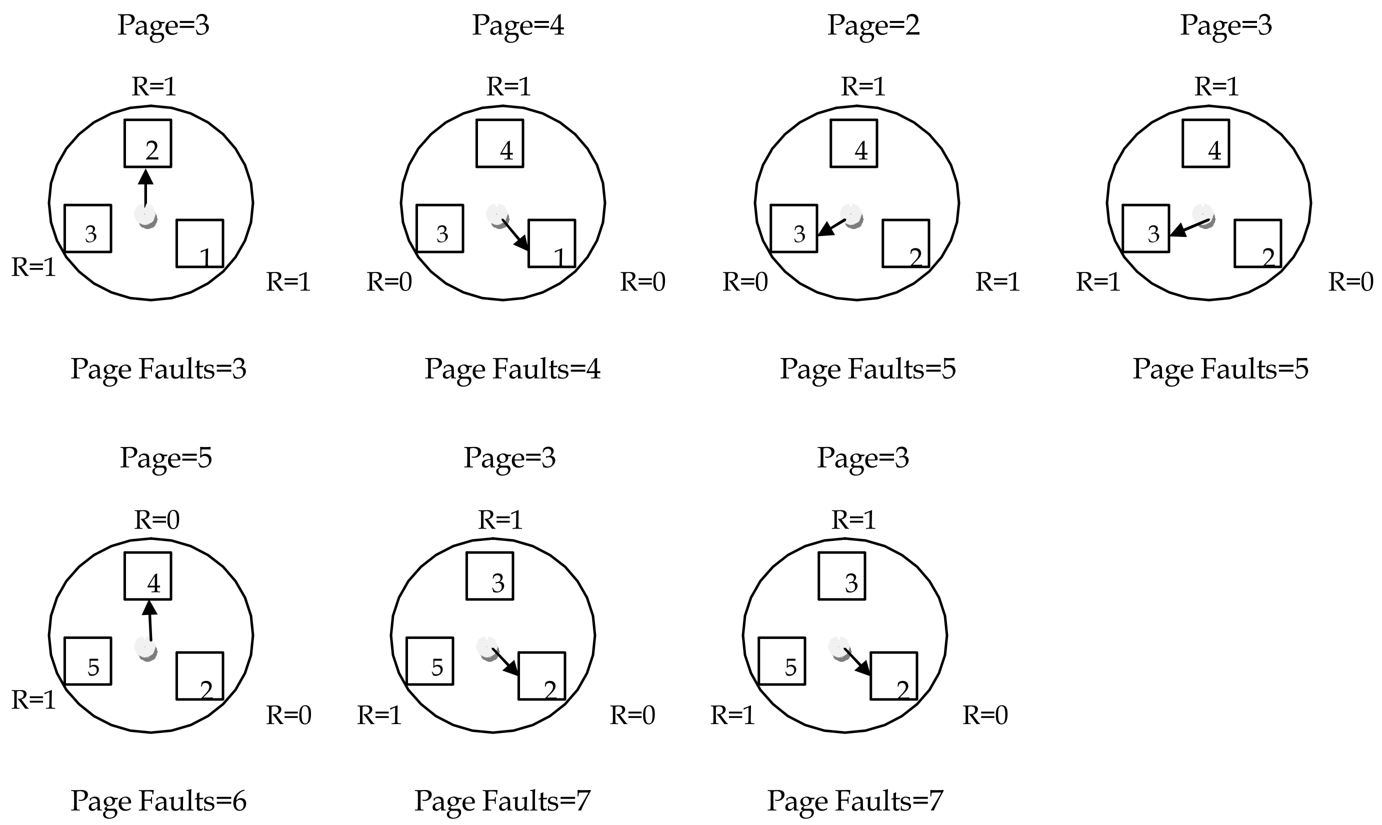
When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with *R* = 0.

Example

Assume Memory Size=3

Reference String: 2 1 3 2 3 4 2 3 5 3 5





Modeling Page Replacement

Belady's Anomaly

Belady's Anomaly states that having more page frames in memory is not necessarily the best course of action. He demonstrated that in a particular example with FIFO there are more page faults with four pages in memory than with three. This situation is called Bledy'd Anomaly. Consider the following sequence of pages, which is assumed to occur on a system with no pages loaded initially that uses FIFO.

Reference String: 0 1 2 3 0 1 4 0 4 1 2 3 4

If we have 3 frames above reference string results in 9 page faults. But, if we have 4 frames this generates 10 page faults.

Stack Algorithms

The set of pages in memory for 4 page frames would also be in with 5 page frames. There would be an additional page in memory, but the set of pages in memory for n page frames is also in memory for n+1 page frames. Algorithms that meet this criterion are called stack algorithms. These algorithms do not suffer from Belady's anomaly Mathematically, these algorithms have the property

$$M(m,r) \subseteq M(m+1, r)$$

This means that pages in memory (M) in the top section (m) at the r point of a reference string are a subset of the top part of memory when it is one larger at the same point in the reference string. Neither LRU nor the optimal algorithm suffers from Belady's anomaly.

Segmentation

Paged virtual memory discussed earlier is one-dimensional in which virtual addresses goes from 0 to some maximum address, one address after another. For many problems, having more separate virtual address spaces may be much better than having one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including:

a) The source text
b) The symbol table that contains the names and attributes of variables
c) The table containing all integers and real constants
d) The parse tree that contains the syntactic analysis of the program
e) The stack used for procedure calls within the compiler

The first four tables grow continuously as compilation proceeds. The last table grows and shrinks unpredictably during compilation. In a one-dimensional memory, the five tables are allocated in contiguous chunks of virtual address space. In this situation if a program has a very large number of variables, the chunk of address space for the symbol table may fill up, but other tables have plenty of room and compiler might discontinue due to too many variables. The solution of this problem is to use segmentation.

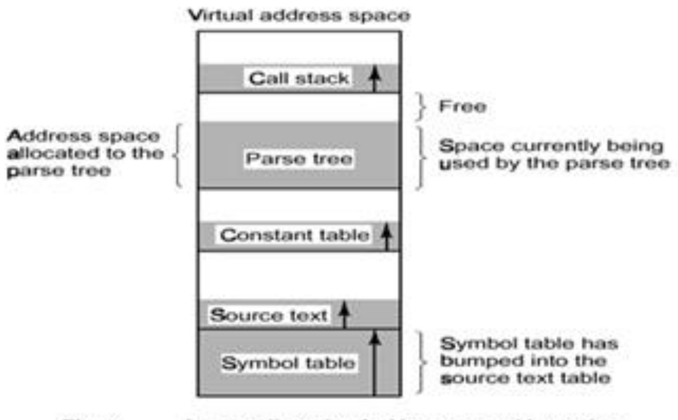

Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Figure In a one-dimensional address space with growing tables, one table may bump into another.

The solution is to use segmentation - i.e. to set aside completely independent address spaces of virtual memory e.g. one virtual memory space for each compile table. Each segment of virtual memory may be a different size, or even change during execution.

A segment is a logical entity. It might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it doesn't contain a mixture of different types. Segmentation is the mechanism that provides the machine with multiple completely independent address spaces. Thus segmentation provides two-dimensional address space. Each segment consists of linear sequence of addresses, from 0 to some maximum. Different segments have different lengths from 0 to maximum allowed. Segment lengths may change during execution. Length of a stack segment may be increased whenever something is pushed on the stack and decreased whenever something is popped off the stack. Each segment constitutes a separate address space; different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it because there is nothing else in its address space to bump into. Segments can rarely fill up because they're usually very large.

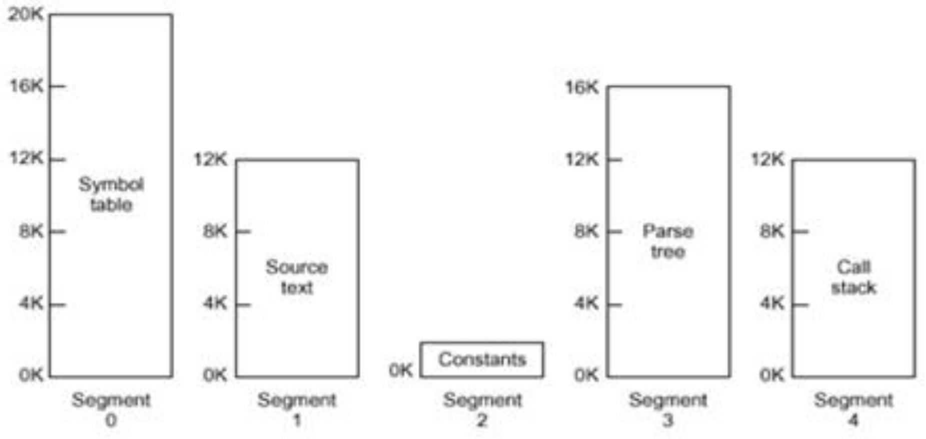
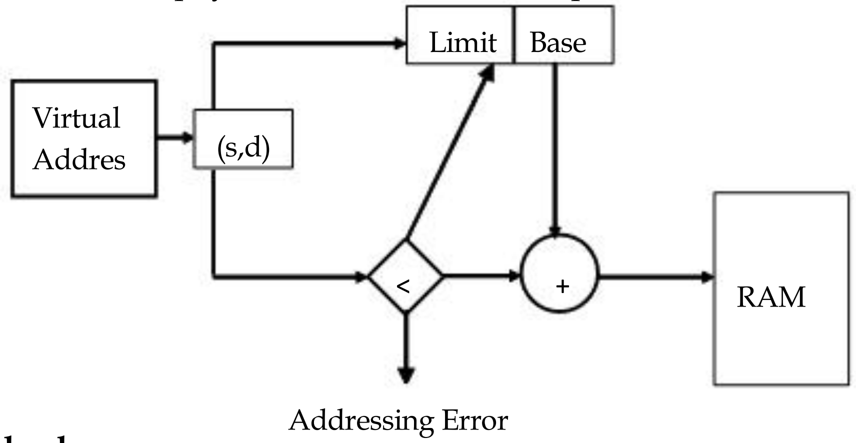


Figure A segmented memory allows each table to grow or shrink independently of the other tables.

To specify an address in this segmented or 2-dimensional memory, the program must supply a 2-part virtual address: 1) segment number(s) and 2) an address within the segment called offset (s). When a user program is compiled, the compiler automatically constructs segments reflecting the input program and segments are numbered. In case of segmentation virtual address is converted into physical address with the help of segment table. Each entry in the segment table has:

1. valid/invalid bit
2. Segment base address (starting address in physical memory)
3. Segment limit (maximum segment length)

Virtual address to physical address translation process is shown in figure below:



Drawbacks

Since segments require contiguous memory allocation external fragmentation may occur after the system has been running for a while. Memory will be divided up into a number of chunks, some containing segments and some containing holes. The figure below shows external fragmentation or checkerboarding.

Figure (a) shows initial memory configuration with 5 segments. Figure (b) shows the situation after replacing segment 1 of 8k by segment 7 of 5k. Thus it creates hole of 3k. In figure (c) another hole of 3k is created after replacing segment 4 of 7k by segment 5 of 4k. Similarly in figure (d), segment 3 of 8k is replaced by segment 6 of 4k and thus creating a hole of 4k.

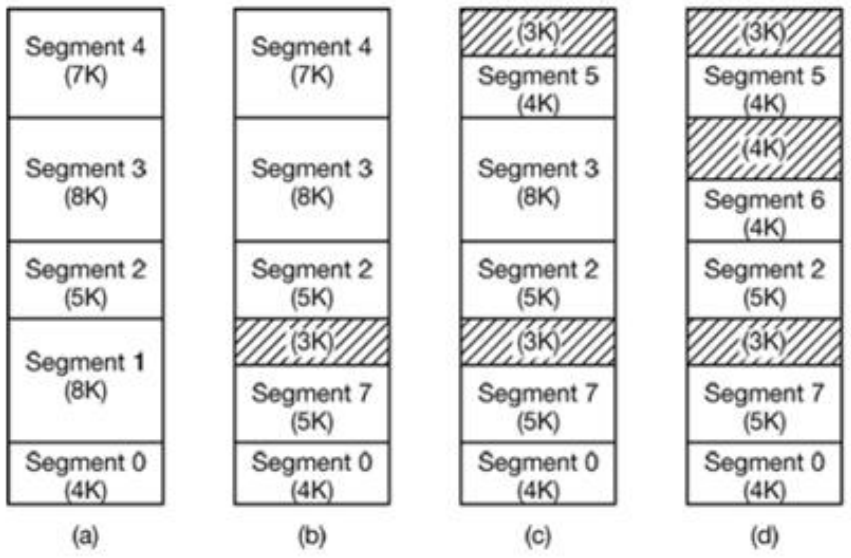


Figure: External Fragmentation

Segmentation with Paging: MULTICS

What happens if size of some segments is larger than size of available physical memory? It is impossible to load such segments into memory. Solution to this problem is to use paged segmentation. By using the idea of paging it is now possible to keep currently used pages of a segment in memory by leaving rest of the pages of the segment in secondary storage and bringing them into memory as needed by using the concept of swapping. Several significant systems have supported paged segments. MULTICS is the first system that uses this concept.

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in figure below. When a memory reference occurs, the following algorithm is carried out.

- ✓ The segment number is used to find the segment descriptor.
- ✓ A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.
- ✓ The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
- ✓ The offset is added to the page origin to give the main memory address where the word is located.
- ✓ The read or store finally takes place.

File Management

File allows data to be stored between processes. It allows us to store large volumes of data and allows more than one process to access the data at the same time.

File Naming

Different operating systems have different file naming conventions. MS-DOS only allows an eight character filename (and a three character extension). This limitation also applies to Windows 3.1. Most UNIX systems allow file names up to 255 characters in length. Modern Windows allows up to 255 characters too, subject to a maximum of 260 characters if one includes the pathname. There are restrictions as to the characters that can be used in filenames e.g. ? and * are forbidden. Some operating systems distinguish between upper and lower case characters. To MS-DOS, the filename ABC, abc, and AbC all represent the same file, UNIX sees these as different files.

Filenames are made up of two parts separated by a full stop. The part of the filename up to the full stop is the actual filename. The part following the full stop is often called a file extension. In MSDOS the extension is limited to three characters. UNIX and Windows 95/NT allow longer extensions. They are used to tell the operating system what type of data the file contains. It associates the file with a certain application. UNIX allows a file to have more than one extension associated with it.

File Structures

Files are stored as a sequence of bytes. It is up to the program that accesses the file to interpret the byte sequence. Files Structures can be categorized into four types:

- ☞ **Byte Sequence:** File is unstructured sequence of bytes: It provides the maximum flexibility. User programs can put anything they want in their files and name them any way that is convenient.
- ☞ **Record Sequence (fixed record):** In this model, a file is a sequence of fixed length records each with some internal structure. Central idea of a file being a sequence of records is the idea that the read operation returns and the write operation overwrites or appends one record.
- ☞ **Tree Structures:** In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

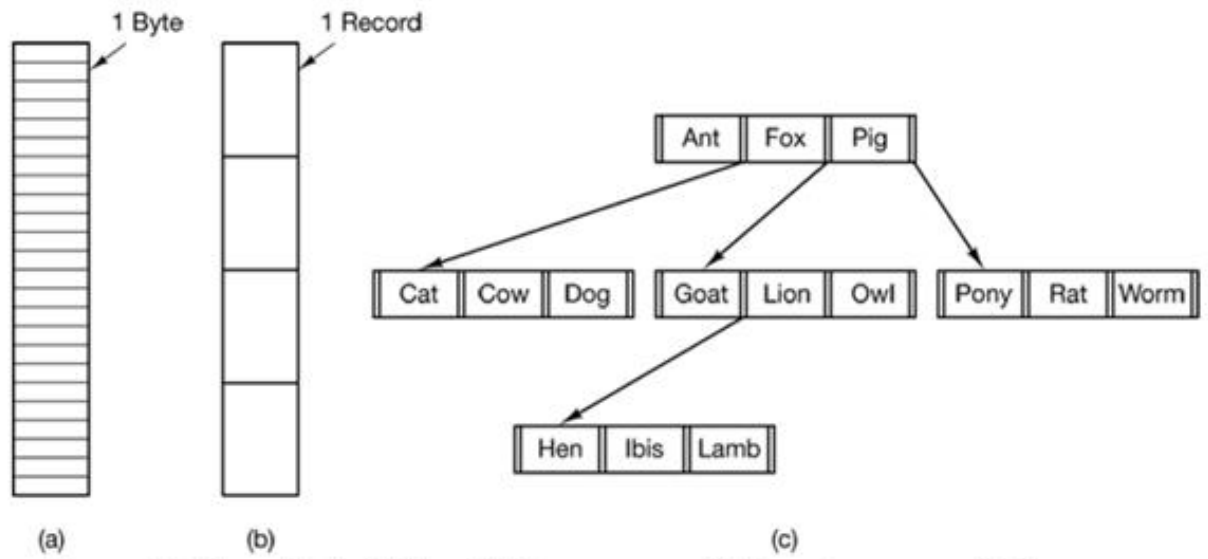


Figure 6-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

File Types
Many operating systems support several types of files. Unix and Windows, have regular files and directories. Regular files are the ones that contain user information generally in ASCII form. Directories are system files for maintaining the structure of the file system. Character special files are related to Input/output and used to model serial I/O devices such as terminals, printers and networks. Block special files are used to Model disks.

File Access
Early operating systems provided only one kind of file access: sequential access. In these system, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files were convenient when the storage medium was magnetic tape, rather than disk. Files whose bytes or records can be read in any order are called random access files. Two methods are used form specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. This allows the system to use different storage techniques for the two classes. Where as in modern operating systems all the files are automatically random access.

File Attributes
Every file has a name and its data. In addition all operating systems associate other information with each file such as the date and time the file was created and the file's size. The list of attributes varies considerably from system to system. Attributes such as protection, password, creator and owner tell who may access it and who may not. The

flags are bits or short fields that control or enable some specific property. The record length, key, position and key length fields are only present in files whose records can be looked up using a key. The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purpose. The current size tells how big the file is at present.

File operations: Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. The few of them of the most common system calls relating to files are:

- **Create:** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
- **Delete:** When the file is no longer needed, it has to be deleted to free up disk space.
- **Open:** Before using a file, a process must open, the purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
- **Close:** When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.
- **Read:** Data re read from file. Usually, the bytes a come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.
- **Write:** Data are written to the file, again, usually at the current position. If the current position is end of the file then the file size gets increased.
- **Append**: This call is a restricted from of write. It can only add data to the end of the file.
- **Seek:** For random access files, a method is needed to specify from where to take the data.
- **Get attributes**: Processes often need to read file attributes to do their work.
- **Set attributes:** Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example.
- **Rename:** It frequently happens that a user needs to change the name of an existing file.

Directories

Allow like files to be grouped together and allow operations to be performed on a group of files which have something in common. For example, copy the files or set one of their attributes. They allow files to have the same filename (as long as they are in different directories). This allows more flexibility in naming files. A typical directory entry contains a number of entries; one per file. All the data (filename, attributes and disc addresses) can be stored within the directory. Alternatively, just the filename can be stored in the directory together with a pointer to a data structure which contains the other details.

Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. This directory is sometimes called **root directory.** On early personal computers, this system was common. The world's first supercomputer, the CDC 6600, also had only a single directory for all files.

Example

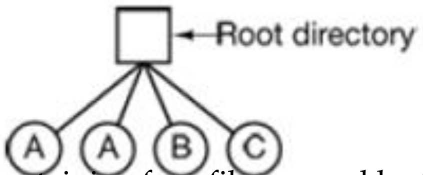
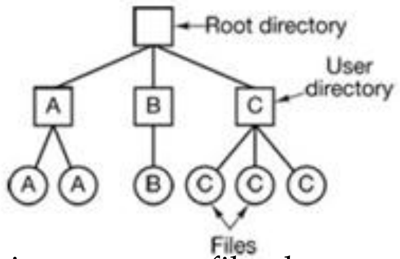


Figure: Root directory containing four files owned by three different users

This system is not applicable in multiple user environments because multiple users may want to create file with same name which is not supported. But it is applicable only in small systems that are only used by few users such as system in a car that is used to keep profile of drivers. Main advantage of single level directory systems is that they are easy to implement.

Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, two-level directory systems can be used. Here, users are given their own directories within root directory.



In this design when a user tries to open a file, the system needs to identify the user in order to know which directory to search. Therefore some kind of login procedure is needed. Two-level directory systems normally stored systems binary executable programs in separate directory. Users can access their own directory without any restriction. But if users want to access files of other users they need to provide calls like open ("Ayan/x"). This might be the call to open a file *x* in the directory of user Ayan.

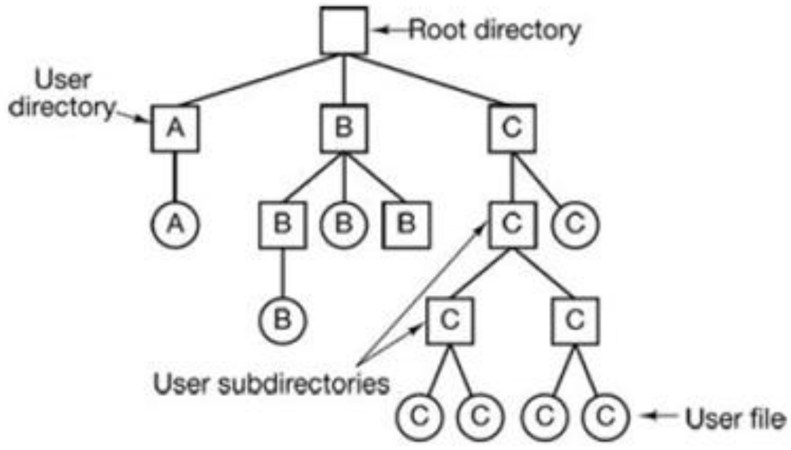
Drawback

- Users cannot make sub-directories with their directory.

Hierarchical Directory Systems

The two-level hierarchy is not satisfactory for users with a large number of files who want to store theses files in different subdirectories by making logical groups of files. For example, a user may want to store music files in one directory, personal data into another directory, study materials in third directory and so on.

Hierarchical directory systems allow users to group their files in natural way by using tree like hierarchy. Here, users can create any number of sub-directories within their directory and can also create subdirectories of subdirectories.



File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block** , and execute it.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. Often the file system will contain some of the items shown in figure below. The first one is the **superblock.** It contains all the key parameters about the file system. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system etc. Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

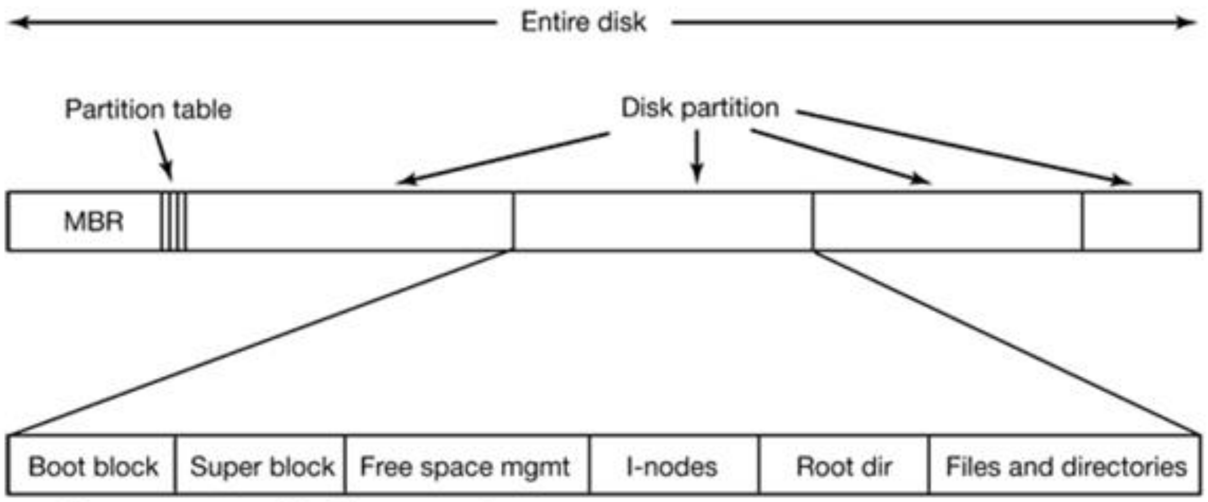


Figure A possible file system layout.

Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. Some of them are discussed below:

Contiguous allocation

Allocate N contiguous blocks to a file. If a file size is 100K and the block size is 2K then 50 contiguous blocks are required to store the file. This strategy is very simple to implement as keeping track of the blocks allocated to a file is reduced to storing the first block that the file occupies and its length.

Advantages

- Simple to implement because we need to store only starting block number and file length
- Read/Write operations can be very fast as the file can be read or written as a contiguous file due to only one seek operation is required.

Drawback

- Fragmentation is main problem of this approach. Due to insertion and deletion of files large number of small wholes may be created and in the worst case we may not be able to store a large file in disk even if we have plenty of free space.

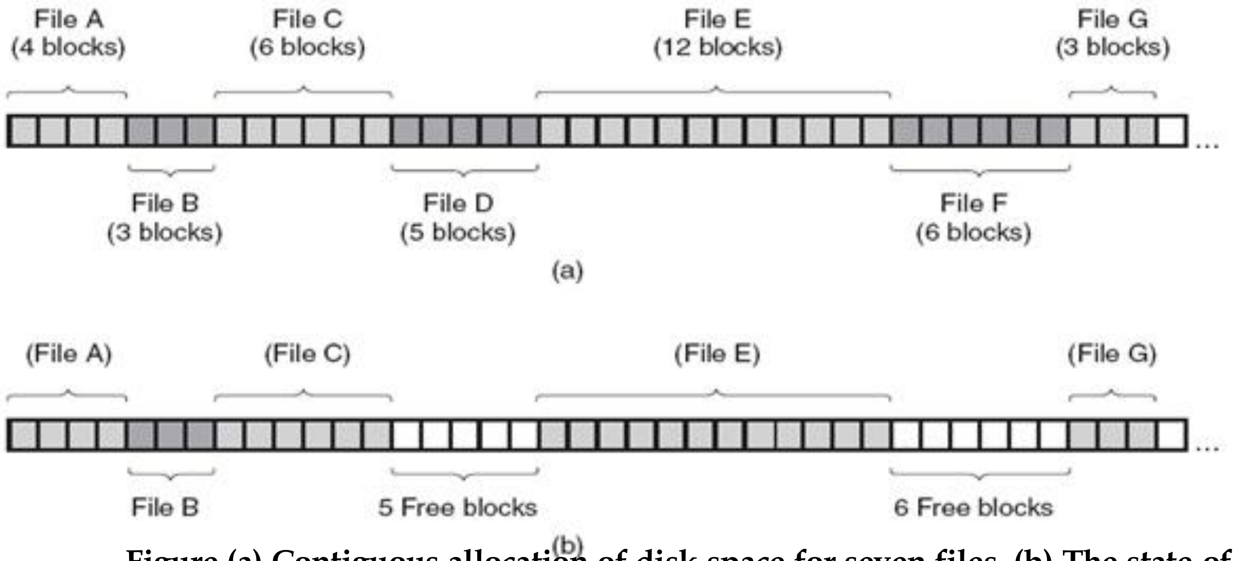


Figure (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files D and F have been removed.

Linked List Allocation

In this strategy blocks of a file are represented using linked lists. All that needs to be held is the address of the first block that the file occupies. Each block contains data and a pointer to the next block. The size of the file does not have to be known beforehand (unlike a contiguous file allocation scheme). When more space is required for a file, any block can be allocated from the free block list.

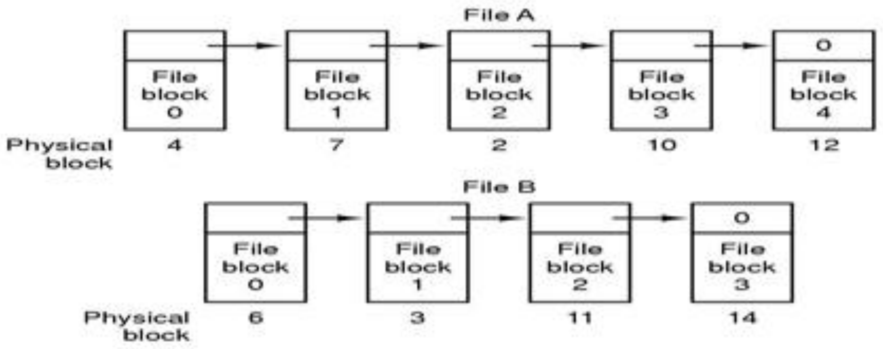


Figure : Storing a file as a linked list of disk blocks.

Advantages

- Every disk block can be used in this method. No space is lost due to disk fragmentation (except for internal fragmentation in the last block).
- It is sufficient for the directory entry to merely store the disk address of the first block.

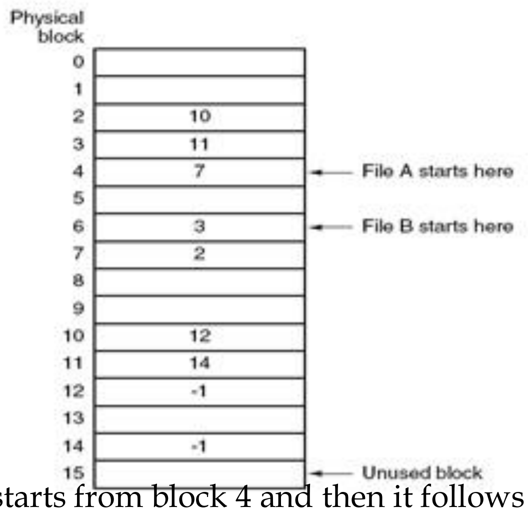
Disadvantage

- Random access is difficult as it needs many disk reads to access a random point in the file

- Space is lost within each block due to the pointer. This does not allow the number of bytes to be a power of two. This is not fatal, but does have an impact on performance.
- Reliability could be a problem. It only needs one corrupt block pointer and the whole system might become corrupted.

Linked List Allocation Using a Table in Memory

Disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Such table is called file allocation table (FAT).



In the above table File A starts from block 4 and then it follows the blocks 7, 2, 10, and 12. Special marker such as (-1) invalid block number can be used to indicate the end of file blocks.

Advantages

- Using this organization, the entire block is available for data.
- Random access is much easier. The chain is entirely in memory, so it can be followed without making any disk references.
- It is sufficient for the directory entry to keep a single integer (the starting block number).

Disadvantage

- Entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries. If each entry requires 4 bytes. Thus the table will take up 80 MB of main memory all the time. Thus it is not suitable for larger disks.

Example

Consider a disk of 100 GB. If block size is 2 KB, calculate the size of FAT assuming that each entry in FAT takes 4 bytes.

Solution

Size of Disk= 100 GB= 100 x 2^{20} KB=104857600 KB

Size of a block =2 KB

Thus,

Number of blocks= 104857600/2=52428800

=>

Number of Entries in FAT=52428800

Since,

Size of an entry= 4 byte

=>

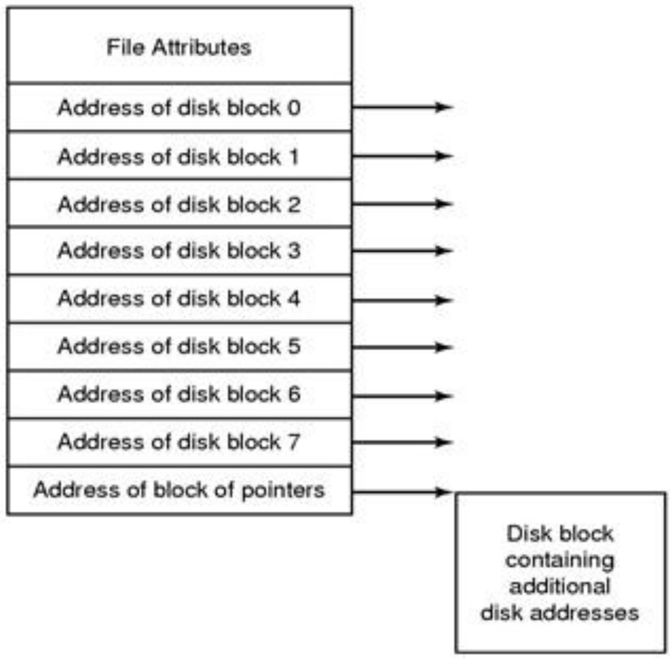
Size of FAT (File Allocation Table)= 52428800 x 4 byte = 200 MB

I-nodes

Another method of accessing file blocks utilizes a data structure called an i-node (index node). All the attributes for the file are stored in an i-node entry, which is loaded into memory when the file is opened. The i-node also contains a number of direct pointers to disc blocks. I-node need only be in memory when the corresponding file is open unlike file allocation table which grows linearly with the disk.

Advantage

- Space needed bin memory is directly proportional to number of files opened not the size of disk. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the i-nodes for the open files is only kn bytes.



Disadvantage

- If i-nodes have room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disks block addresses.

Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. Some of the directory operations are listed below:

-**Create:** A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system

- **Delete:** A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.
- **Opendir:** Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
- **Closedir:** When a directory has been read, it should be closed to free up internal table space.
- **Readdir:** This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
- **Rename**: In many respects, directories are just like files and can be renamed the same way files can be.
- **Link**: Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link** .

-**Unlink:** A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path /usr/ast/mailbox means that the root directory contains a subdirectory usr , which in turn contains a subdirectory ast , which contains the file mailbox . Absolute path names always start at the root directory and are unique.

The other kind of name is the **relative path name.** This is used in conjunction with the concept of the **working directory** (also called the **current directory)**. A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is /usr/ast , then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox.

Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. For example, for a contiguous allocation scheme, the directory entry will contain the first disc block and length (number of blocks). The same is true for linked list allocations. For an i-node implementation the directory entry contains the i-node number. The main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

One of the main issues in directories is where the attributes (such as file's owner and creation time etc) should be stored. Two options are available:

- Store attributes directly in the directory entry.
- Store attributes in the i-nodes

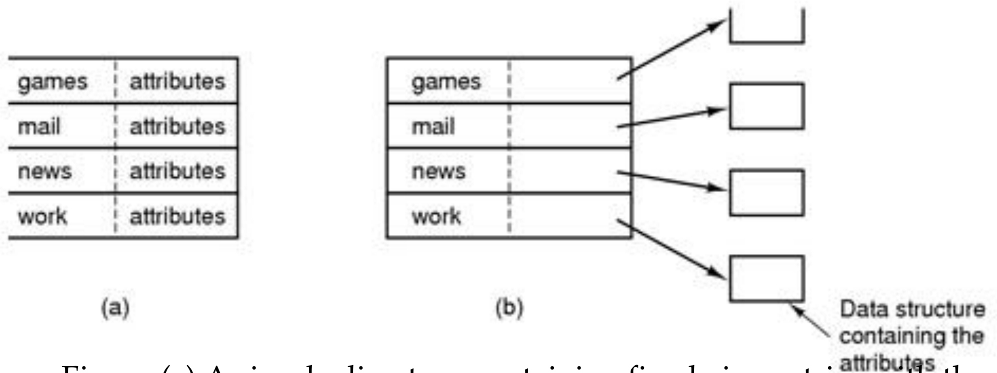


Figure (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

Another major issue about directories is about storing names of files. The simplest approach is to set a limit on file name length, typically 255 characters. This approach is simple, but wastes a great deal of directory space, since few files have such long names. Normally, There are two ways of handling long file names in a directory:

- In-line (variable length) where each filename terminated with a special character. Compacting memory is feasible.
- Store file names in a heap

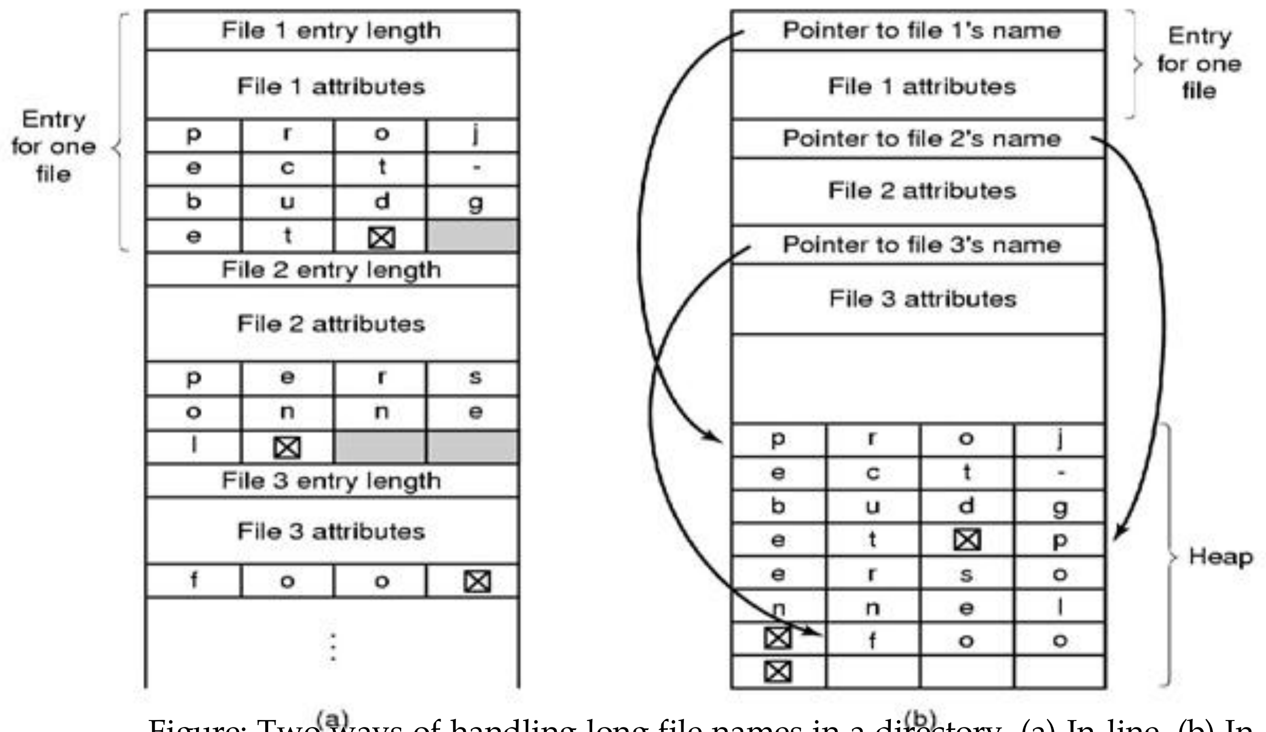


Figure: Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

Shared Files

In many situations we need to share files between multiple users. For example, when several users are working together on a project, they often need to share files. It is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Thus, it is better to represent file system by using DAG (directed acyclic graph) rather than tree structure as shown in the figure below:

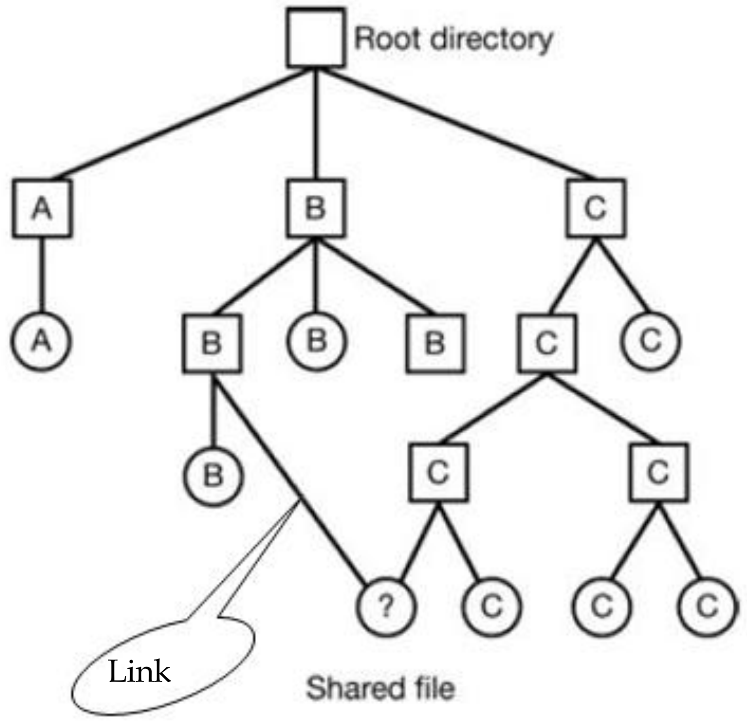


Figure: File System representation using DAG

It is not a good idea to make a copy if a file is being shared. If directories contain disk addresses problem may occur in synchronizing the change made to the file by multiple users. If one users appends new block in the file new blocks will be listed only in the directory of the user doing the append. Following two approaches can be used to solve this problem.

- Directory entry that only points to i-nodes
- Directory entry that points to link file

In the first approach, disk blocks are not listed in directories rather directory entry points to the little data structure (i-node in unix) associated with the file itself. Creating a link does not change the ownership, but it does increase the link count in the i-node. Main problem with this approach is that If owner of the shared file tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, another user may have a directory entry pointing to an invalid i-node. One solution to this is to only remove owner's directory entry. But again If the system has quotas, owner will continue to be billed for the file until another user decides to remove it.

In the second approach, when another user say B want to share the file owned by used C. System creates a LINK file and then maker entry of the LINK file in B's directory. LINK file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**. Extra overhead is the main problem associated with this approach. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses.

Disk Space Management

Whatever block size we choose, then every file must occupy this amount of space as a minimum. If we pick a very large block size, then wastage of space is high. This is because no two files can share the same block (only one file can reside on it). If block size is smaller than most files then files are split up over several blocks and we spend more time seeking. There is a compromise between a block size, fast access and wasted space. The usual compromise is to use a block size of 512 bytes, 1K bytes or 2K bytes.

Keeping Track of Free Blocks

There are two widely used techniques for keeping the track of free blocks: Using bitmap and using linked list of blocks. In case of bitmap, there is a bit for each block on the disc. If the bit is 1 then the block is free. If the bit is zero, the block is in use. Thus a disc with n blocks requires a bit map with n entries.

Second approach uses linked list of disk blocks where each block in the list contains as many free disk block numbers as will fit. Therefore it is also called freelist. Generally bit maps require a lesser number of blocks than a linked list. Only when the disk is nearly full does the linked list implementation require fewer blocks.

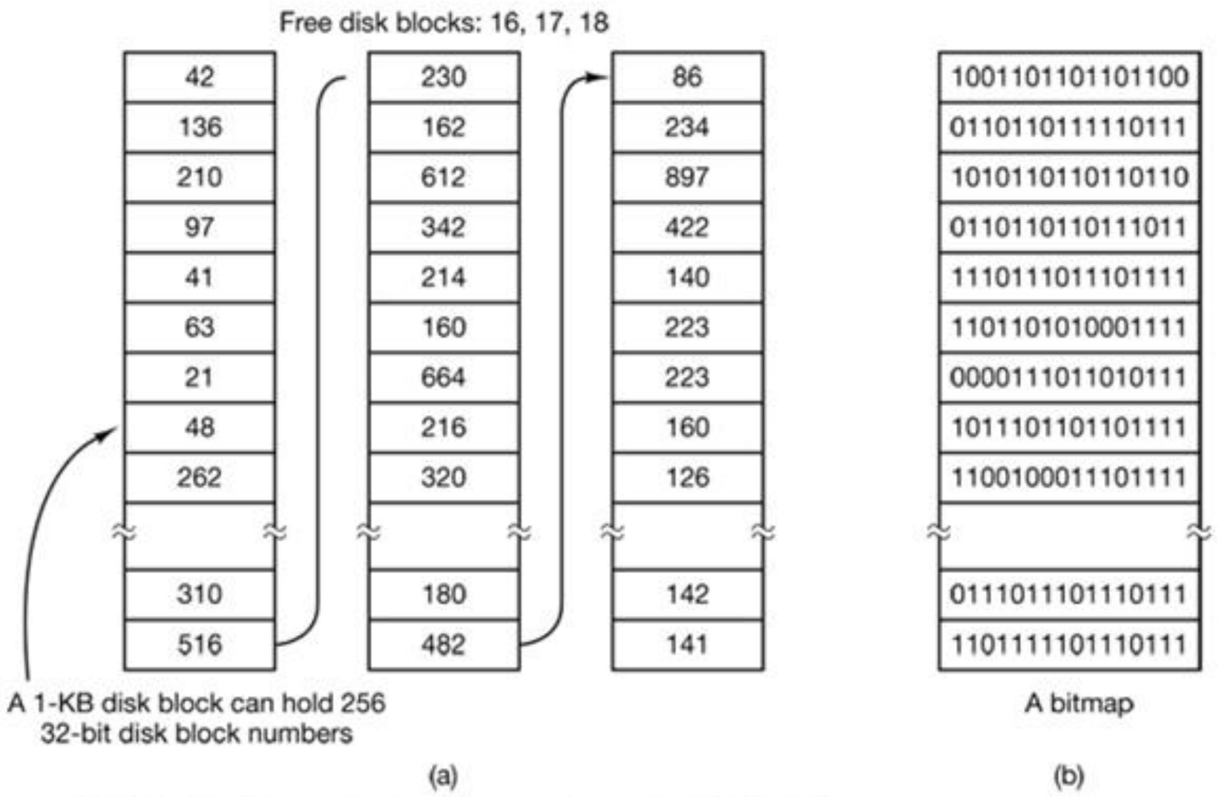


Figure (a) Storing the free list on a linked list. (b) A bitmap.

Example

Consider a 40MB disc with 2K blocks. Calculate the number of blocks needed to hold the disc bitmap. If we require 16-bit to hold a disc block number (i.e disk block number ranges from 0-65535), what will be number of blocks needed by linked list of free blocks.

Solution

Case for Bitmap

Disk Size= 40 MB = 40 x 1024 KB

Thus,

Number of blocks=(40x1024)/2=20 x 1024

Since every block needs 1-bit in bitmap

=>

Size of Bitmap= 20 x 1024 bit=2560 byte= 2.5 KB

Since block size is 2 KB

=>

2 blocks are used to hold the bitmap

Case for Freelist

Block Size= 2K=2 x 1024 byte

Number of bits needed to store block number=16-bit=2 byte

Thus,

Number of blocks that can be stored in a block= (2 x 1024)/2=1024

Since one of the addresses is used to store address of the next block in the free list

=>

Number of blocks that can be stored in a block=1024-1=1023

We know that

Size of Disk=40 MB = 40 x 1024 KB

Since,

Sixe of block= 2K

=>

Number of blocks in Disc= (40 x 1024)/ 2= 20 x 1024

Thus,

Number of blocks needed to store free list= (20 x 1024)/1023=20.019=21