Semaphores

Hardware solutions to synchronization problems are not easy to generalize to more complex problems. To overcome this difficulty we can use a synchronization tool called a semaphore. A **semaphore** S is an integer variable that, apart from initialization is accessible only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait) and V (for signal).

The main disadvantage of the semaphore discussed in the previous section is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

To overcome the need for busy waiting, we can modify the definition of semaphore and the wait and signal operations on it. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute. the busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Problems with Semaphores

Here are some key points about the use of semaphores:

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- The wait(S) and signal(S) operations are scattered among several processes. Hence, it is difficult to understand their effects.
- Usage of semaphores must be correct in all the processes.
- One bad (or malicious) process can fail the entire system of cooperating processes.

Incorrect use of semaphores can cause serious problems. We now discuss a few of these problems.

Deadlocks and Starvation

A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set. Starvation is infinite blocking caused due to unavailability of resources. These problems are due to programming errors because of the tandem use of the wait and signal operations. The solution to these problems is higher-level language constructs such as critical region (region statement) and monitor.

There are two kinds of semaphores:

- **Counting semaphore** whose integer value can range over an unrestricted integer domain.
- **Binary semaphore** whose integer value cannot be > 1; can be simpler to implement.

Classic Problems of Synchronization

The three classic problems of synchronization are:

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

assumes that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. This code can be interpreted as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

Readers Writers Problem

A data object (such as a file or a record) is to be shared among several concurrent processes. Some of these processes, called **readers**, may want only to read the content of the shared object whereas others, called **writers**, may want to update (that is to read and write) the shared object. Obviously, if two readers access the data simultaneously, no adverse effects will result. However, if a writer and some other process (whether a writer or some readers) access the shared object simultaneously, chaos may ensue. To ensure these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to the readerswriters problem.

The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The **second readers-writers problem** requires that once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading. A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of her neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. One

simple solution is to represent each chopstick by a semaphore. A philosopher tires to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Several possibilities that remedy the deadlock situation discussed in the last lecture are listed. Each results in a good solution for the problem.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

High-level Synchronization Constructs

Critical regions

All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. we describe one fundamental highlevel synchronization construct—the **critical region**. We assume that a process consists of some local data, and a sequential program that can operate on the data. Only the sequential program code that is encapsulated within the same process can access the local data. That is, one process cannot directly access the local data of another process. Processes can however share global data.

Monitors

Another high-level synchronization construct is the monitor type. A **monitor** is characterized by local data and a set of programmer-defined operators that can be used to access this data; local data be accessed only through these operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization construct explicitly. While one process is active within a monitor, other processes trying to access a monitor wait outside the monitor. the monitor construct as defined so far is not powerful enough to model some synchronization schemes. For this purpose we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct** (also called **condition variable**). A programmer who needs to write her own tailor made synchronization scheme can define one or more variables of type condition.

Monitor-based Solution for the Dining Philosophers Problem

Each philosopher before starting to eat must invoke the pickup operation. This operation ensures that the philosopher gets to eat if none of its neighbors are eating. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown operation and may start to think. The putdown operation checks if a neighbor (right or left—in this order) of the leaving philosopher

wants to eat. If a neighboring philosopher is hungry and neither of that philosopher's neighbors is eating, then the leaving philosopher signals it so that she could eat. In order to use this solution, a philosopher i must invoke the operations pickup and putdown in the following sequence: It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. Here's an example:

- System has 2 tape drives.
- P1 and P2 each hold one tape drive and each needs another one.

System Model

A system consists of a finite number of resources to be distributed among a finite number of cooperating processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, disk drive, file are examples of resource types. A system with two identical tape drives is said to have two instances of the resource type disk drive. If a process requests an instance of a resource type, the allocation of any instance of that type will satisfy the request. If it will not, then the instances are not identical and the resource type classes have not been defined properly. 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 in order to carryout its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests a needed resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can use the resource.
3. **Release:** The process releases the resource.

Deadlock Characterization

The following four conditions must hold simultaneously for a deadlock to occur:

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **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.
3. **No preemption**: Resources cannot be preempted. That is, after using it a process releases a resource only voluntarily.
4. **Circular wait**: A set $\{P_0, P_1 \ldots P_n\}$ 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, and so on, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

Resource Allocation Graphs

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 is portioned into two different types of nodes $P=\{P_0, P_1 \ldots P_n\}$, the set of the active processes in the system, and $R=\{R_0, R_1 \ldots R_n\}$, the set consisting of all resource types in the system. A directed edge from a process P_i to resource type R_j signifies that process P_i requested an instance of R_j and is waiting for that resource. A directed edge from R_j to P_i signifies that an instance of R_j has been allocated to P_i. Given the definition of a resource allocation graph, it can be shown that if the graph contains no cycles, then no process is deadlocked. If the graph contains cycles then:

- If only one instance per resource type, then a deadlock exists.
- If several instances per resource type, possibility of deadlock exists.

Deadlock Handling

We can deal with deadlocks in a number of ways:

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover from deadlock.
- Ignore the problem and pretend that deadlocks never occur in the system.

These three ways result in the following general methods of handling deadlocks:

1. Deadlock prevention: is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how processes can request for resources.

2. Deadlock Avoidance: This method of handling deadlocks requires that processes give advance additional information concerning which resources they will request and use during their lifetimes. With this information, it may be decided whether a process should wait or not.

3. Allowing Deadlocks and Recovering: One method is to allow the system to enter a deadlocked state, detect it, and recover.

Deadlock Prevention

By ensuring that one of the four necessary conditions for a deadlock does not occur, we may prevent a deadlock.

Mutual exclusion

The mutual exclusion condition must hold for non-sharable resources, e.g., printer. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock, e.g., read-only files. Also, resources whose states can be saved and restored can be shared, such as a CPU. In general, we cannot prevent deadlocks by denying the mutual exclusion condition, as some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold and wait condition does not occur in a system, we must guarantee that whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol requires a process to request

resources only when it has none. A process may request some resources and use them. But it must release these before requesting more resources.

The two main disadvantages of these protocols are: firstly, resource utilization may be low, since many resources may be allocated but unused for a long time. Secondly, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No preemption

To ensure that this condition does not hold we may use the protocol: if a process is holding some resources and requests another that cannot be allocated immediately to it, then all resources currently being held by the process are preempted. These resources are implicitly released, and added to the list of resources for which the process is waiting. The process will be restarted when it gets all its old, as well as the newly requested resources.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing ordering of enumeration.

Let R={ R_1, R_2, R_3 }be resource types. We assign to each a unique integer, which allows us to compare two resources and to determine whether one precedes another in our ordering.

Deadlock Avoidance

One method for avoiding deadlocks is to require additional information about how resources may be requested. Each request for resources by a process requires that the system consider the resources currently available, the resources currently allocated to the process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested by each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

Safe State

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally a system is in a safe state only if there exists a safe sequence. A sequence of processes <P_1, P_2… P_n> is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus all the resources held by all the P_j with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When P_i terminates, P_{i+1} can obtain its needed resources and terminate. If no such sequence exists, then the system is said to be unsafe. If a system is in a safe state, there can be no deadlocks. An unsafe state is not a deadlocked state; a deadlocked state is conversely an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock state.

Deadlock avoidance makes sure that a system never enters an unsafe state. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. A dashed line is used to represent a claim edge. When P_i requests resource R_j the claim edge is converted to a request edge.

Banker's Algorithm

In this algorithm, when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need, i.e., each process must a priori claim maximum use of various system resources. This number may not exceed the total number of instances of resources in the system, and there can be multiple instances of resources. When a process requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise the process must wait until some other process releases enough resources. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures in the Banker's algorithm:

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

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize
Work = Available and Finish[i] = false fori = 1, 2, …, n.
2. Find an i such that both
a) Finish[i] = = false
b) $Need_i$ <= Work
If no such i exists go to step 4.
3. Work = Work + $Allocation_i$
Finish[i] = true
Go to step 2
4. If Finish[i] = = true for all i, then the system is in a safe mode.

This algorithm may require an order of m x n^2 operations to decide whether a state is safe.

Resource Request Algorithm

Let $Request_i$ be the request vector for process P_i. if $Request_i$ [j]=k, then process P_i wants k instances of resource R_j. When a request for resources is made by process P_i the following actions are taken:

1. If $Request_i$ <= $Need_i$ go to step 2. Otherwise, raise an error condition since the process has exceeded its maximum claim.
2. If $Request_i$ <= Available, go to step 3. Otherwise P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
Availabe = Available-$Request_i$;
$Allocation_i$ = $Allocation_i$ + $Request_i$;
$Need_i$ = $Need_i$ –$Request_i$;
Invoke the Safety algorithm. If the resulting resource allocation graph is safe, the transaction is completed. Else, the old resource allocation state is restored and process P_i must wait for $Request_i$.

Deadlock Detection

If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm then a deadlock may occur. In this environment, the system must provide:
 An algorithm that examines (perhaps periodically or after certain events) the state of the system to determine whether a deadlock has occurred
 A scheme to recover from deadlocks

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource allocation graph, called a **wait-for graph**. We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. a deadlock exists in the system if and only if the wait for graph contains a cycle. To detect deadlocks the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

Several Instances of a Resource Type

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock detection algorithm described next is applicable to such a system. It uses the following data structures:
 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.
The algorithm is:
1) Let Work and Finish be vectors of length m and n respectively. Initialize Work=Available. For i=1, 2,… , n if Allocation[i] ≠ 0 the Finish[i]=false; otherwise Finish[i]=true
2) Find an index i such that both
a. Finish[i] = = false
b. $Request_i \leq$ Work

c. If no such i exists go to step 4.
3) Work=Work + $Allocation_i$
a. Finish[i]=true
b. Go to step 2.
4) If Finish[i] = = false, for some i, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if Finish[i] = = false, then P_i is deadlocked.

Detection Algorithm Usage

When should we invoke the deadlock detection algorithm? The answer depends n two factors:
1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens? Hence the options are:
 Every time a request for allocation cannot be granted immediately—expensive but process causing the deadlock is identified, along with processes involved in deadlock
 Periodically, or based on CPU utilization
 Arbitrarily—there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock

There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods the system reclaims all resources allocated to the terminated process.
 Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
 Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.
Aborting a process may not be so easy. If a process was in the midst of updating a file, terminating it will leave the system in an inconsistent state. If the partial termination method is used, then given a set of deadlocked processes, we must determine which process should be terminated in an attempt to break the deadlock. This determination is a policy decision similar to CPU scheduling problems. The question is basically an economic one, we should abort those processes the termination of which will incur the minimum cost.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:
1. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include

such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as the victim. As a result this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process is picked as a victim only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Memory Management

Basic Concepts

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of its program counter and other memory management registers such as segment registers in Intel CPUs. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, e.g., first fetches an instruction from memory, which is then decoded and executed. Operands may have to be fetched from memory. After the instruction has been executed, the results are stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated or what they are for (instructions or data).

Memory Hierarchy

The memory hierarchy includes:

- Very small, extremely fast, extremely expensive, and volatile CPU registers
- Small, very fast, expensive, and volatile cache
- Hundreds of megabytes of medium-speed, medium-price, volatile main memory
- Hundreds of gigabytes of slow, cheap, and non-volatile secondary storage
- Hundreds and thousands of terabytes of very slow, almost free, and non-volatile Internet storage (Web pages, Ftp repositories, etc.)

Memory Management

The purpose of memory management is to ensure fair, secure, orderly, and efficient use of memory. The task of memory management includes keeping track of used and free memory space, as well as when, where, and how much memory to allocate and deallocate. It is also responsible for swapping processes in and out of main memory

Source to Execution

Translation of a source program in a high-level or assembly language involves compilation and linking of the program. This process generates the machine language executable code (also known

as a binary image) for the give source program. To execute the binary code, it is loaded into the main memory and the CPU state is set appropriately.

Address Binding

Usually a program resides on a disk as a binary executable or script file. The program must be brought into the memory it to be executed. The collection of processes that is waiting on the disk to be brought into the memory for execution forms the **input queue.** The normal procedure is to select one of the processes in the input queue and to load that process into the memory. As the process is executed, it accesses instructions and data from memory. Eventually the process terminates and its memory space is become available for reuse.

Addresses may be bound in different ways during these steps. Addresses in the source program are generally symbolic (such as an integer variable *count*). Address can be bound to instructions and data at different times, as discussed below briefly.

 Compile time: if you know at compile where the process will reside in memory, the **absolute addresses** can be assigned to instructions and data by the compiler.

 Load time: if it is not known at compile time where the process will reside in memory, then the compiler must generate **re-locatable code**. In this case the final binding is delayed until load time.

 Execution time: if the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this to work.

Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address,** where as an address seen by the memory unit–that is, the one loaded into the **memory-address register** of the memory–is commonly referred to as the **physical address.** In essence, logical data refers to an instruction or data in the process address space where as the physical address refers to a main memory location where instruction or data resides. The compile time and load time binding methods generate identical logical and physical addresses, where as the execution time binding method results in different physical and logical addresses. In this case we refer to the logical address as the **virtual address.** The set of all logical addresses generated by a program form the **logical address space** of a process; the set of all physical addresses corresponding to these logical addresses is a **physical address space** of the process. The total size of physical address space in a system is equal to the size of its main memory. The logical address is translated into the corresponding physical address by adding the logical address to the value of the relocation register, The run-time mapping from virtual to physical addresses is done by a piece of hardware in the CPU, called the **memory management unit (MMU).**

In the following example, we show the logical address for a program instruction and computation of physical address for the given logical address.

- Logical address (16-bit)
  IP = 0B10h
  CS = D000h
- Physical address (20-bit)
  CS * 24 + IP = D0B10h

Various techniques for memory management

Dynamic Loading

To obtain better memory space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on a disk in a re-locatable format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded or not. If not, the re-locatable linking loader is called to load the desired routine into the memory and to update the program's address tables to reflect this change. The control is then passed to the newly loaded routine. The advantage of dynamic loading is that an unused routine is never loaded. This means that potentially less time is needed to load a program and less memory space is required. However the run time activity involved in dynamic loading is a disadvantage. Dynamic programming does not require special support from the operating system.

Dynamic Linking and Shared Libraries

The concept of dynamic linking is similar to that of dynamic loading. Rather than the loading being postponed until execution time, linking is postponed until run-time. This feature is usually used with system libraries. Without this facility, all programs on a system need to have a copy of their language library included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library-routine reference. This *stub* is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. During execution of a process, stub is replaced by the address of the relevant library code and the code is executed .If library code is not in memory, it is loaded at this time. Programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**. Dynamic linking requires potentially less time to load a program. Less disk space is needed to store binaries. However it is a time-consuming run-time activity, resulting in slower program execution. Dynamic linking requires help from the operating system.

Overlays

To enable a process to be larger than the amount of memory allocated to it, we can use **overlays.** The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

concept of overlays with the example of a two-pass compiler. Here are the various specifications:

- 2-Pass assembler/compiler
- Available main memory: 150k
- Code size: 200k
  - Pass 1 …….. 70k
  - Pass 2 …….. 80k
  - Common routines …... 30k
  - Symbol table …… 20k

Swapping

A process needs to be in the memory to be executed. A process, however, can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution. Backing store is a fast disk large enough to accommodate copies of all memory images for all users; it must provide direct access to these memory images. The system maintains a *ready queue* of all processes whose memory images are on the backing store or in memory and are ready to run. A variant of this swapping policy can be used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manger can swap out the lower-priority process so that it can load and execute the higher- -priority process. When the higher--priority process finishes, the lower--priority process can be swapped back in and continued. This technique is called **roll out, roll in**.

Cost of Swapping

Process size	= 1 MB
Transfer rate	= 5 MB/sec
Swap out time	= 1/5 sec
	= 200 ms
Average latency	= 8 ms
Net swap out time	= 208 ms
Swap out + swap in	= 416 ms

Contiguous memory allocation

The main memory must accommodate both operating system and the various user spaces. Thus memory allocation should be done efficiently. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. The operating system may be placed in the high memory or the low memory. The position of the interrupt vector usually affects this decision. Since the interrupt vector is often in the low memory, programmers place the OS in low memory too.

 It is desirable to have several user processes residing in the memory at the same time. In contiguous memory allocation, each process is contained in a single contiguous section of memory.

The **base** (re-location) and **limit** registers are used to point to the smallest memory address of a process and its size, respectively.

Multiprogramming with Fixed Tasks (MFT)

In this technique, memory is divided into several fixed-size partitions. Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded in the free partition. When the process terminates, the partition becomes available for another process.

MFT can have large **internal fragmentation**, i.e., wasted space *inside* a Region

Each process has a single ``segment'' (we will discuss segments later)

- No sharing between processes.
- No dynamic address translation.
- At load time must ``establish addressability''.

MFT with multiple queues involves load-time address binding. In this technique, there is a potential for wasted memory space i.e. an empty partition but no process in the associated queue. However in MFT with single queue there is a single queue for each partition. The queue is searched for a process when a partition becomes empty. *First-fit*, *best-fit*, *worst-fit* space allocation algorithms can be applied here.

Multiprogramming with Variable Tasks (MVT)

This is the generalization of the fixed partition scheme. It is used primarily in a batch environment. Here are the main characteristics of MVT.

- Both the number and size of the partitions change with time.
- Job still has only one segment (as with MFT) but now can be of any size up to the size of the machine and can change with time.
- A single ready list.
- Job can move (might be swapped back in a different place).
- This is dynamic address translation (during run time).
- Must perform an addition on every memory reference (i.e. on every address translation) to add the start address of the partition.
- **Eliminates internal fragmentation**.
- Find a region the exact right size (leave a hole for the remainder).

Introduces external fragmentation, i.e., holes *outside* any region

External fragmentation

As processes come and go, *holes* of free space are created in the main memory. External Fragmentation refers to the situation when free memory space exists to load a process in the memory but the space is not contiguous. Compaction eliminates external fragmentation by shuffling memory contents (processes) to place all free memory into one large block. The cost of compaction is slower execution of processes as compaction takes place.

Paging

two Paging is a memory management scheme that permits the physical address space of a process to be noncontiguous. It avoids the considerable problem of fitting the various sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower so compaction is impossible. Physical memory is broken down into fixed-sized blocks, called **frames**, and logical memory is divided into blocks of the same size, called **pages**. The size of a page is a power of 2, the typical page table size lying between 1K and16K. It is important to keep track of all free frames. In order to run a program of size *n* pages, we find *n* free frames and load program pages into these frames. In order to keep track of a program's pages in the main memory a **page table** is used. Thus when a process is to be executed, its pages are loaded into any available memory frames from the backing store.

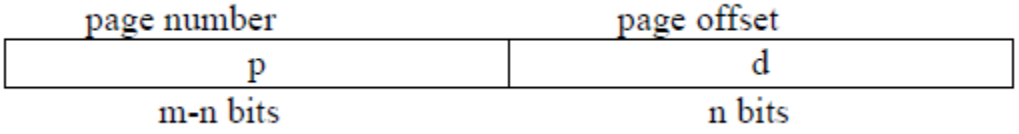
b) Mapping paging in the logical into the frames in the physical address space and keeping this mapping in the page table

Every **logical address** generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page table contains the base address (frame number) of each page in physical memory. The frame number is combined with the page offset to obtain the physical memory address of the memory location that contains the object addressed by the corresponding logical address.

Paging itself is a form of dynamic relocation. When we use a paging scheme, we have no external fragmentation; however we may have **internal fragmentation**.

Addressing in Paging

The page size is defined by the CPU hardware. If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words) , then the high-order *m-n* bits of a logical address designate the page number and the *n* low order bits designate offset within the page. Thus, the logical address is as follows:



Implementation of Page table

Implementation of Page table

 In the CPU registers

This is OK for small process address spaces and large page sizes. It has the advantage of having *effective memory access time* (Teffective) about the same as memory access time (Tmem).

 In the main memory

A page table base register (PTBR) is needed to point to the page table. With page table in main memory, the effective memory access time, Teffective, is 2Tmem , which is not acceptable because it would slow down program execution by a factor of two.

 In the translation look-aside buffer (TLB)

A solution to this problem is to use special, small, fast lookup hardware, called translation look-aside buffer (TLB), which typically has 64–1024 entries. Each entry is (key, value). The key is searched for in parallel; on a hit, value is returned. The (key,value) pair is (p,f) for paging. For a logical address, (p,d), TLB is searched for p. If an entry with a key p is found, we have a hit and f is used to form the physical address. Else, page table in the main memory is searched.

Performance of Paging

We discuss performance of paging in this section. The performance measure is the effective memory access time. With part of the page table in the TLB and the rest in the main memory, the effective memory access time on a hit is $T_{mem} + T_{TLB}$ and on a miss is $2T_{mem} + T_{TLB}$.

If HR is hit ratio and MR is miss ratio, the effective access time is given by the following equation

$$T_{effective} = HR\ (T_{TLB} + T_{mem}) + MR\ (T_{TLB} + 2T_{mem})$$

Protection under Paging

Memory protection in paging is achieved by associating protection bits with each page. These bits are associated with each page table entry and specify protection on the corresponding page. The primary protection scheme guards against a process trying to access a page that does not belong to its address space. This is achieved by using a valid/invalid (v) bit. This bit indicates whether the page is in the process address space or not. If the bit is set to invalid, it indicates that the page is not in the process's logical address space. Illegal addresses are trapped by using the valid-invalid bit and control is passed to the operating system for appropriate action. One bit can define the page table to be read and write or read only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read only page. An attempt to write to a read-only page causes a hardware trap to the operating system (memory-protection violation).

Structure of the Page Table

As logical address spaces become large (32-bit or 64-bit), depending on the page size, page table sizes can become larger than a page and it becomes necessary to page the page the page table. Additionally, large amount of memory space is used for page table. The following schemes allow efficient implementations of page tables.

 Hierarchical / Multilevel Paging
 Hashed Page Table
 Inverted Page Table

Hierarchical/Multilevel Paging

Most modern computers support a large logical address space: (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. This page table cannot fit in one page. One solution is to page the page table, resulting in a 2-level paging. A page table needed for keeping track of pages of the page table— called the outer page table or page directory. In the 32-bit machine described above, we need to partition p into two parts, p1 and p2. p1 is used to index the outer page table and p2 to index the inner page table. Thus the logical address is divided into a page number consisting of 20 bits and a page offset of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page number, and a 10-bit page offset. This is known as **two-level paging**.

Hashed Page Table

This is a common approach to handle address spaces larger then 32 bits .Usually open hashing is used. Each entry in the linked list has three fields: page number, frame number for the page, and pointer to the next element—(p, f, next). The page number in the logical address (specified by p) is hashed to get index of an entry in the hash table. This index is used to search the linked list associated with this entry to locate the frame number corresponding to the given page number. The advantage of hashed page tables is smaller page tables.

Inverted Page Table

Usually each process has a page table associated with it. The page table has one entry for each page in the address space of the process. For large address spaces (32-bit and above), each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the mapping of logical address spaces of processes onto the physical memory. A solution is to use an inverted page table. An **inverted page table** has one entry for each real page (frame) of memory. Each entry consists of the virtual address of the page stored in the in that real memory location, with information about the process that own the page.

Sharing in Paging

Another advantage of paging is the possibility of *sharing* common code. Reentrant (readonly) code pages of a process address can be shared. If the code is reentrant, it never changes during execution. Thus two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process.

Segmentation

Segmentation is a memory management scheme that supports programmer's view of memory. A logical-address space is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, global variables, stack, and symbol table. Each segment has a name and length. The addresses specify both the segment name and the offset within the segment. a logical address consists of a two tuple:

<segment-number, offset> or <s,d>

The segment table maps the two-dimensional logical addresses to physical addresses. Each entry of a segment table has a *base* and a *segment limit*. The segment base contains the starting physical

address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

There are two more registers, relevant to the concept of segmentation:

 Segment-table base register (STBR) points to the segment table's location in memory.

 Segment-table length register (STLR) indicates number of segments used by a program.

Sharing of Segments

Another advantage of segmentation is sharing of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point o the same physical location. The sharing occurs at segment level, thus, any information defined as a segment can be shared.

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging except that the segments are of *variable* length; pages are all the same size. Thus memory allocation is a dynamic storage allocation problem, usually solved with a best fit or worst fit algorithm.

Protection

A particular advantage of segmentation is the association of protection with segments. Because the segments represent a semantically defined portion of the program, it is likely that ll the entries will be used the same way. Hence, some segments are instructions, whereas other segments are data. In a modern architecture, instructions are non-self modifying so they can be defined as read only. Or execute only. The memory mapping hardware will check the protection bits associated with each segment-table entry top prevent illegal access to memory, such as attempts to write into a read only segment. The bits associated with each entry in the segment table, for the purpose of protection are:

 Validation bit : if the validation bit is 0, it indicates an illegal segment

 Read, write, execute bits

Issues with Segmentation

Segmentation may then cause external fragmentation (i.e. total memory space exists to satisfy a space allocation request for a segment, but memory space is not contiguous), when all blocks of memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes 178 available or until compaction creates a larger hole. Since segmentation is by nature a dynamic relocation algorithm, we can compact memory whenever we want. If we define each process to be one segment, this approach reduces to the variable sized partition scheme. T the other extreme, every byte could be put in its own segment and relocated separately. This eliminates external fragmentation altogether, however every byte would need a base register for its relocation, doubling memory use. The next logical step- fixed sized, small segments, is paging i.e. paged segmentation.

Paged Segmentation

In paged segmentation, we divide every segment in a process into fixed size pages. We need to maintain a page table per segment CPU's memory management unit must support both segmentation and paging.

address translation in the protected mode.

Protected Mode

- 248 bytes virtual address space
- 232 bytes linear address space
- Max segment size = 4 GB
- Max segments / process = 16K
- Six CPU registers allow access to six segments at a time
- Selector is used to index a segment descriptor table to obtain an 8-byte segment descriptor entry. Base address and offset are added to get a 32-bit linear address, which is partitioned into p1, p2, and d for supporting 2-level paging.

Virtual Memory Basic Concept

An examination of real programs shows that in many cases the existence of the entire program in memory is not necessary:

- Programs often have code to handle unusual error conditions. Since these errors seldom occur in practice, this code is almost never executed.
- Arrays, lists and tables are often allocated more memory than they actually need.
- Certain options of a program may be used rarely.

Even in cases where the entire program is needed, it may not be all needed at the same time. The ability to execute a program that is only partially in memory confers many benefits.

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Virtual Memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming easier because the programmer need not worry about the amount of physical memory, or about what code can be placed in overlays; she can concentrate instead on the problem to be programmed. In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by several different processes through page sharing. The sharing of pages further allows performance improvements during process creation. Virtual memory is commonly

implemented as demand paging. It can also be implemented in a segmentation system. One benefit of virtual memory is efficient process creation. Yet another is the concept of memory mapped files.

Demand Paging

A demand paging system is similar to a paging system with swapping. Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages rather than as one large contiguous address space, use of swap is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. Thus the term pager is used in connection with demand paging.

Page Fault

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault trap. The paging hardware in translating the address through the page table will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk transfer overhead and memory requirements) rather than an invalid address error as a result of an attempt to use an illegal memory address. The procedure for handling a page fault is straightforward:

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was valid or invalid memory access.
2. If the reference was invalid we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

Page Fault

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault trap. The paging hardware in translating the address through the page table will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk transfer overhead and memory requirements) rather than an invalid address error as a result of an attempt to use an illegal memory address. The procedure for handling a page fault is straightforward:

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was valid or invalid memory access.

2. If the reference was invalid we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.
In addition to this hardware, additional architectural constraints must be imposed. A crucial one is the need to be able to restart any instruction after a page fault. In most cases this is easy to meet, a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again, and then fetch the operand. A similar problem occurs in machines that use special addressing modes, including auto increment and auto decrement modes. These addressing modes use a register as a pointer and automatically increment or decrement the register. Auto decrement automatically decrements the register before using its contents as the operand address; auto increment increments the register after using its contents. Thus the instruction
MOV (R2) +, -(R3)
Copies the contents of the location pointed to by register2 into that pointed to by register3. Now consider what will happen if we get a fault when trying to store into the location pointed to by register3. To restart the instruction we must reset the two registers to the values they had before we started the execution of the instruction

Performance of demand paging

A page fault causes the following sequence to occur:
1. Trap to the operating system
2. Save the user registers and process states
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on disk
5. Issue a read from the disk to a free frame:
a. Wait in a queue for this device until the read request is serviced
b. Wait for the device seek and/or latency time
c. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user (CU scheduling; optimal)
7. Interrupt from the disk (I/O completed)
8. Save the registers and process state for the other user (if step 6 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state and new page table

In any case we are faced with three major components of the page fault service time:

1. Service the page fault interrupt
2. Read in the page
3. Restart the process

It is important to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault. It is important to keep the page fault rate low in a demand-paging system. Otherwise the effective access time increases, slowing process execution dramatically. One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. It is therefore possible for the system to gain better paging throughput by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. Another option is to demand pages from the from the file system initially, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.

Performance of Demand Paging with Page Replacement

When there is no free frame available, page replacement is required, and we must select the pages to be replaced. This can be done via several replacement algorithms, and the major criterion in the selection of a particular algorithm is that we want to minimize the number of page faults. The victim page that is selected depends on the algorithm used, it might be the least recently used page, or the most frequently used etc depending on the algorithm.

Process Creation and Virtual Memory

Paging and virtual memory provide other benefits during process creation, such as copy on write and memory mapped files.

Copy on Write fork()

Demand paging is used when reading a file from disk into memory and such files may include binary executables. However, process creation using fork() may bypass initially the need for demand paging by using a technique similar to page sharing. This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to newly created processes.

Recall the fork() system call creates a child process as a duplicate of its parent.

Traditionally fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Alternatively we can use a technique known as copy on write. This works by allowing the parent and child processes to initially share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Using the copy-on-write technique it is obvious that only the pages that are modified by either process are copied; all non modified pages may be shared by the parent and the child processes. Note that only pages that may be modified are marked as copy-on-write. Pages

that cannot be modified (i.e. pages containing executable code) may be shared by the parent and the child.

When it is determined a page is gong to be duplicated using copy-on-write it is important to note where the free page will be allocated from. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or for managing copy-on-write pages. Operating systems typically allocate these pages using a technique known as zero-fill-ondemand. Zero-fill-on-demand pages have been zeroed out before allocating, thus deleting the previous contents on the page. With copy-on-write the page being copied will be copied to a zero-filled page. Pages allocated for the stack or heap are similarly assigned zero-filled pages.

vfork()

Several versions of UNIX provide a variation of the fork() system call—vfork() (for virtual memory fork). vfork() operates differently than fork() with copy on write. With vfork() the parent process is suspended and the child process uses the address space of the parent. Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with caution, ensuring that the child process does not modify the address space of the parent.

Linux Implementation

In Linux, shared pages are marked read-only after fork(). If either process tries to modify a shared page, a page fault occurs and the page is copied. The other process (who later faults on write) discovers it is the only owner; so no copying takes place. In other words, Linux implementation of fork() is based on the "copy-on-write" semantics.

Memory Mapped files

Consider a sequential read of a file on disk using the standard system calls open(), read(), write(). Every time the file is accessed requires a system call and disk access. Alternatively we can use the virtual memory techniques discussed so far to treat file I/O as routine memory accesses. This approach is known as memory mapping a file, allowing a part of the virtual address space to be logically associated with a file. Memory mapping a file is possible by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds using ordinary demand paging resulting in a page fault. However, a page sized portion of the file is read from the file system into a physical page. Subsequent reads and writes to the file are handled as routine memory accesses, thereby simplifying file access and usage by allowing file manipulation through memory rather than the overhead of using the read() and write() system calls.

Memory-Mapped Files in Solaris 2

Some operating systems provide memory mapping only through a specific system call and treat all other file I/O using the standard system calls. However, some systems choose to memory map a file regardless of whether a file was specified as a memory map or not. For example: Solaris 2 treats all file I/O as memory mapped, allowing file access to take place in memory, whether a file has been specified as memory mapped using mmap() system call or not.

mmap() System Call

In a UNIX system, mmap() system call can be used to request the operating system to memory map an opened file.

Page replacement

While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list: All memory is in use. This means that if no free frame is available on a page fault, we replace a page in memory to load the desired page. The page-fault service routine is modified to include page replacement. We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory. The modified page fault service routine is:

1. Find the location of the desired page on the disk
2. Find a free frame

a) If there is a free frame use it.

b) If there is no free frame, use a page replacement algorithm to select a victim frame.

3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

We can reduce overhead by using a *modify* bit (or *dirty* bit). Each page or frame may have a modify bit associated with it in hardware. The modify bit is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement we examine it's modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case we must write that page to the disk. If the modify bit is not set however, the page has not been modified since it was read into memory, and hence we can avoid writing that page to disk.

Page Replacement Algorithms

In general we want a page replacement algorithm with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory we insert t at the tail of the queue. The problem with this algorithm is that it suffers from Belady's anomaly: For some page replacement algorithms the page fault rate may increase as the number of allocated frames increases, whereas we would expect that giving more memory to a process would improve its performance.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page fault rate of all algorithms, and will never suffer from the Belay's algorithm. This algorithm is simply to replace the page that will not be used for the longest period of time. Use of this algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

LRU Page Replacement

If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least recently used algorithms. An LRU page replacement may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counter-based Implementation of LRU

In the simplest case we associate with each page table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page entry for that page. In that way we always have the time of the last reference to each page. We replace the page that has the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access. The times must also be maintained when page tables are changed. Overflow of the clock must be considered.

Stack-based Implementation of LRU

Another approach to implementing the LRU algorithm is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page. Because entities must be removed from the middle of the stack, it is best implementing by a doubly linked list with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement the tail pointer points to the bottom of the stack which is the LRU page.

Belady's Anomaly

This is due to the **Belady's Anomaly** which states that "For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases."

Stack Replacement Algorithms

These are a class of page replacement algorithms with the following property: *Set of pages in the main memory with n frames is a subset of the set of pages in memory with n+1 frames.*
These algorithms do not suffer from Belady's Anomaly. An example is the LRU algorithm.

LRU Approximation Algorithm

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support and other page replacement algorithms must be used. Many systems provide some help however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced. Reference bits are associated with each entry in the page table. Initially all bits are cleared by the operating system. As a user process executes

the bit associated with each page referenced is set to 1 by the hardware. After some time we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the order of use however, but we know which pages were used and which were not used.

Least frequently used algorithm

This algorithm is based on the locality of reference concept— the least frequently used page is not in the current locality. LFU requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average user count.

Most Frequently Used

The MFU page replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used; it will be in the **locality** that has just started.

Page Buffering Algorithm

The OS may keep a pool of free frames. When a page fault occurs a victim page is chosen as before. However the desired page is read into a free frame from the pool before the victim is written out. This allows the process to restart as soon as possible, without waiting for the victim to be written out. When the victim is later written out, its frame is added to the free frame pool. Thus a process in need can be given a frame quickly and while victims are selected, free frames are added to the pool in the background An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out. Another modification is to keep a pool of free frames, but to remember which page was in which frame. Since the frame contents are not modified when a frame is written to disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

Local vs Global Replacement

If process P generates a page fault, page can be selected in two ways:

- Select for replacement one of its frames.
- Select for replacement a frame from a process with lower priority number.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame belongs to some other process; one process can take a frame from another. Local replacement requires that each process select from only its allocated frames.

Allocation of frames

Each process needs a minimum number of frames so that its execution may be guaranteed on a given machine. Let’s consider the MOV X,Y instruction. The instruction is 6 bytes long (16-bit offsets) and might span 2 pages. Also, two pages to handle source and two pages are required to

handle destination (assuming 16-bit source and destination). There are three major allocation schemes:

- **Fixed allocation**

In this scheme free frames are equally divided among processes

- **Proportional Allocation**

Number of frames allocated to a process is proportional to its size in this scheme.

- **Priority allocation**

Priority-based proportional allocation

Thrashing

If a process does not have “enough” pages, the page-fault rate is very high. This leads to low CPU utilization. The operating system thinks that it needs to increase the degree of multiprogramming, because it monitors CPU utilization and find it to be decreasing due to page faults. Thus another process is added to the system and hence thrashing occurs and causes throughput to plunge. A process is **thrashing** if it is spending more time paging (i.e., swapping pages in and out) than executing. Thrashing results in severe performance problems:

- Low CPU utilization
- High disk utilization
- Low utilization of other I/O devices

as the degree of multiprogramming increases CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in and CPU utilization drops sharply. At this point we must decrease the degree of multiprogramming. We can limit the effects of thrashing by using a local replacement scheme. With local replacement if one process starts thrashing it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part. Hence local page replacement prevents thrashing to spread among several processes. However if processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase and effective access time will increase even for a process that is not thrashing. If a process does not have enough frames, it will quickly page fault. At this point, if a free frame is not available, one of its pages must be replaced so that the desired page can be loaded into the newly vacated frame. However since all its pages are in active use, the replaced page will be needed right away. Consequently it quickly faults again and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called **thrashing**. In this case, *only one process is thrashing*. A process is thrashing if it is spending more time paging than executing. Thrashing results on severe performance problems. The operating system monitors CPU utilization and, if CPU utilization is too low, the operating system increases the degree of multiprogramming by introducing one or more new processes to the system. This decreases the number of frames allocated to each process currently in the system, causing more page faults and further decreasing the CPU utilization. This causes the operating system to introduce more processes into the system. As a result CPU utilization drops even further and the CPU scheduler tries to increase the degree

of multiprogramming even more. Thrashing has occurred and system throughput plunges. The page fault rate increases tremendously. As a result the effective memory access time increases. Along with low CPU utilization, there is high disk utilization. There is low utilization of other I/O devices. No work is getting done, because the processes are spending all their time paging and the system spend most of its time servicing page fault. Now *the whole system is thrashing*—the CPU utilization plunges to almost zero, the paging disk utilization becomes very high, and utilization of other I/O devices becomes very low.

Thus in order to stop thrashing, the degree of multiprogramming needs to be reduced. The effects of thrashing can be reduced by using a local page replacement. With local replacement if one process starts thrashing it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process if which they are a part. However, if processes are thrashing they will be in the queue for the paging device most of the time. The average service time for a page fault will increase due to the longer average queue for the paging device. Thus the effective access time will increase even for a process that is not thrashing, since a thrashing process is consuming more resources.

Locality of Reference

The locality model states that as a process executes it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

Working Set Model

The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called the working set. If a page is in active use it will be in the working set. If it no longer being used it will drop from the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality.

The difficulty with the working set model is to keep track of the working set. The working set window is a moving size window. At each memory reference a new reference appears at one end and the oldest reference drops off the other end. We can approximate the working set model with a fixed interval timer interrupt and a reference bit.

Page Fault Frequency

Page fault frequency is another method to control thrashing. Since thrashing has a high page fault rate, we want to control the page fault frequency. When it is too high we know that the process needs more frames. Similarly if the page-fault rate is too low, then the process may have too many frames. The operating system keeps track of the upper and lower bounds on the page-fault rates of processes. If the page-fault rate falls below the lower limit, the process loses frames. If page-fault rate goes above the upper limit, process gains frames. Thus we directly measure and control the page fault rate to prevent thrashing.

Other considerations

Many other things can be done to help control thrashing. We discuss some of the important ones in this section.

Pre-paging

An obvious property of a pure demand paging system is the large number of page faults that occur when a process is started. This situation is the result of trying to get the initial locality into memory. Pre-paging is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed. Pre-paging may be an advantage in some cases. The question is simply whether the cost of using pre-paging is less than the cost of the servicing the corresponding page faults.

Page Size

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. Because each active process must have its own copy of the page table, a large page size is desirable. On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, and continuing till it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated. This causes internal fragmentation and to minimize this, we need a small page size. Another problem is the time required to read or write a page. I/O time is composed of seek, latency and transfer times. Transfer time is proportional to the amount transferred, and this argues for a small page size. However, latency and seek times usually dwarf transfer times, thus a desire to minimize I/O times argues for a larger page size. I/O overhead is also reduced with small page size because locality improves. This is because a smaller page size allows each page to match program locality more accurately. Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. There is no best answer. However the historical trend is towards larger pages.

Program Structure

Demand paging is designed to be transparent to the user program. However, in some cases system performance can be improved if the programmer has an awareness of the underlying demand paging and execution environment of the language used in the program.

Which of the following will improve CPU utilization?

- Install a faster CPU
- Increase degree of multiprogramming
- Decrease degree of multiprogramming
- Install more main memory

Clearly, the system is thrashing, so the first two are not going to help and the last two will help. Think about the reasons of this answer.

2. Which of the following programming techniques and structures are “good” for a demand paged environment? Which are bad? Explain your answer.

- Stack
- Hash table
- Sequential search

- Binary search
- Indirection
- Vector operations

You should try to answer this question on your own. Focus on how the given data structures and techniques access data. Sequential access means "good" for demand paging (because it causes less page faults) and non-sequential access means "bad" for demand paging environment.

The File Concept

A file is a named collection of
related information that is recorded on secondary storage. Data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric or binary. In essence it is a contiguous logical address space.

File Structure

A file has certain defined structure characteristics according to its type. A few common types of file structures are:

None – file is a sequence of words, bytes

Simple record structure

Lines

Fixed length

Variable length

Complex Structures

Formatted document

Relocatable load file

UNIX considers each file to be a sequence of bytes; no interpretation of these bytes is made by the OS. This scheme provides maximum flexibility but little support.

File Attributes

Every file has certain attributes, which vary from one OS to another, but typically consist of these:

Name: The symbolic file name is the only information kept in human-readable form

Type: This information is needed for those systems that support different types.

Location: This location is a pointer to a device and to the location of the file on that device.

Size: The current size of the file (in bytes, words or blocks) and possibly the maximum allowed size are included in this attribute.

Protection: Access control information determines who can do reading , writing, etc.

Owner

Time and date created: useful for security, protection and usage monitoring.

Time and date last updated: useful for security, protection and usage monitoring.

Read/write pointer value

Where are Attributes Stored?

File attributes are stored in the directory structure, as part of the **directory entry** for a file, e.g., in DOS, Windows, or in a separate data structure; in UNIX/Linux this structure is known as the **inode** for the file.

Directory Entry

A file is represented in a directory by its directory entry. Contents of a directory entry vary from system to system. For example, in DOS/Windows a directory entry consists of file name and its attributes. In UNIX/Linux, a directory entry consists of file name and inode number.

File Operations

Various operations can be performed on files. Here are some of the commonly supported operations. In parentheses are written UNIX/Linux system calls for the corresponding operations.

- Create (`creat`) —two steps are necessary to create a file. First, space must be found for the file in the file system. Second, an entry for the new file must be made in the directory.
- Open (`open`) — The open operation takes a file name and searches the directory, copying the directory entry into the open-file table.
- Write (`write`) —To write to a file, we make a system call, specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place.
- Read (`read`) — To read from a file we use a system call that specifies the name of the file, and where (in memory) the next block of the file should be put. The system needs top keep a read pointer to the location in the file where the next read is to take place. The current pointer location is kept as a process **current-file-position pointer**. Both read and write use the same pointer
- Reposition within file (`lseek`) — A directory is searched for the appropriate entry and the current-file-position is set to a given value. This is often known as a file seek.
- Delete (`unlink`) — Search the directory for the named file, and then release the file space and erase the directory entry. File can be deleted using the unlink system call.
- Truncate (`creat`) — A user may want to erase the contents of the file but keep its attributes. This function allows all attributes to be unchanged except for file length., which is set to zero and file space is released. This can be achieved using creat with a special flag
- Close (`close`) — When a file is closed, the OS removes its entry in the open-file table.

File Types: Extensions

A common technique for implementing files is to include the type of the file as part of the file name. The name is split into two parts, a name and an extension, usually separated by a period character. In this way, the user and the OS can tell from the name alone, what the type of a file is. The operating system uses the extension to indicate the type of the file and the type of operations that can be done on that file. In DOS/Windows only a file with .exe, .com, .bat extension can be executed.

The UNIX system uses a crude magic number stored at the beginning of some files to indicate roughly the type of the file-executable program, batch file/shell script, etc. Not all files have magic

numbers, so system features cannot be based solely on this type of information. UNIX does allow file name extension hints, but these extensions are not enforced or depended on by the OS; they are mostly to aid users in determining the type of contents of the file.

File Types in UNIX

UNIX does not support supports seven types of file:

- **Ordinary file**: used to store data on secondary storage device, e.g., a source program(in C), an executable program. Every file is a sequence of bytes.
- **Directory**: contains the names of other files and/or directories.
- **Block-special file**: correspond to block oriented devices such as a disk. They are used to access such hardware devices.
- **Character-special file**: correspond to character oriented devices, such as keyboard
- **Link file** (created with the `ln -s` command): is created by the system when a symbolic link is created to an existing file, allowing you to rename the existing file and share it without duplicating its contents without
- **FIFO** (created with the mkfifo or mknod commands or system calls): enable processes to communicate with each other. A FIFO(name pipe) is an area in the kernel that allows two processes to communicate with each other provided they are running on the same system , but the processes do not have to be related to each other.
- **Socket** (in BSD-compliant systems—socket): can be used by the process on the same computer or on different computers to communicate with each other.

File Access

Files store information that can be accessed in several ways:

Sequential Access

Information in the file is processed in order, one record after the other. A read operation reads the next potion of the file and automatically advances a file pointer which tracks the I/O location. Similarly, a write operation appends to the end of the file and advances to the end of the newly written material. Such a file can be rest to the beginning and on some systems; a program may be able to skip forward or backward, n records.

Direct Access

A file is made up of fixed length logical record that allow program to read and write records in no particular order. For the direct-access method, the file operations must be modified to include the block number as a parameter (read n (n = relative block number), write n for instance). An alternate approach is to retain read next and write next and to add an operation, *position file to n*, where n is the block number. The block number provided by the user to the OS is normally a *relative block number*, an index relative to the beginning of the file.

Directory Structure

It is a collection of directory entries. To manage all the data, first disks are split into one or more partitions. Each partition contains information about files within it. This information is kept within device directory or volume table of contents.

Directory Operations

The following directory operations are commonly supported in contemporary operating systems. Next to each operation are UNIX system calls or commands for the corresponding operation.

- Create — `mkdir`
- Open — `opendir`
- Read — `readdir`
- Rewind — `rewinddir`
- Close — `closedir`
- Delete — `rmdir`
- Change Directory — `cd`
- List — `ls`
- Search

Directory Structure

When considering a particular directory structure we need to consider the following issues:

1. **Efficient Searching**
2. **Naming** – should be convenient to users
   - Two users can have same name for different files
   - The same file can have several different names
3. **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ..)

Single-Level Directory

All files are contained in the same directory, which is easy to support and understand. However when the number of files increases or the system has more than one user, it has limitations. Since all the files are in the same directory, they must have unique names.

Two-Level Directory

There is a separate directory for each user. When a user refers to a particular file, only his own user file directory (UFD) is
searched. Thus different users can have the same file name as long as the file names within each UFD are unique. This directory structure allows efficient searching. However, this structure effectively isolates one user from another, hence provides no
grouping capability.

Tree Directory

Here is the tree directory structure. Each user has his/her own directory (known as user's home directory) under which he/she can create a complete directory tree of his/her own. The tree has a root directory. Every file in the system has a unique pathname. A path name is the path from the root, through al the subdirectories to a specified file. A directory/subdirectory contains a set of files or subdirectories. In normal use, each user has a current directory. The current directory should contain most of the files that are of current interest to the user. When a reference to a file is made, the current directory is searched. If a file is needed that is not in the current directory,

then the user must either specify a path name or change the directory to the directory holding the file(using the cd system call).

UNIX / Linux Notations and Concepts

- Root directory (/)
- Home directory
  - `~, $HOME, $home`
  - `cd ~`
  - `cd`
- Current/working directory (**.**)
  - `pwd`
- Parent of Current Directory (**..**)
- Absolute Pathname
  - Starts with the root directory
  - For example, /etc, /bin, /usr/bin, /etc/passwd, /home/students/ibraheem
- Relative Pathname
  - Starts with the current directory or a user's home directory
  - For example, ~/courses/cs604, ./a.out

Acyclic-Graph Directories

A tree structure prohibits sharing of files. An acyclic graph allows directories to have shared subdirectories and files. The same file may be in two different directories. A shared file is not the same as two copies of the file. Only one actual copy exists, so any changes made by one user are immediately visible to the other. A common way of implementing shared files and directories is to create a new directory entry called a link, which is effectively a pointer to another file or subdirectory. A link can be implemented as an absolute or relative path name. Another problem involves deletion. If the file is removed when anyone deletes it, we may end up with dangling pointers to the now-nonexistent file. Solutions: Another approach is to preserve the file until all references to it are deleted. When a link or a copy of the directory entry is establishes, a new entry is added to the file-reference list. When a link is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty. Since the reference list can be very large we can keep a count of the number of references. A new link or directory increments the **reference count**, deleting a link or entry decrements the count. When the count is 0, the file can be deleted. UNIX uses this solution for hard links. **Backpointers** can also be maintained so we can delete all pointers. One serious problem with using an acyclic-graph structure is ensuring that there are no cycles. A solution is to allow only links to files not subdirectories. Also every time a new link is added use a **cycle detection algorithm** to determine whether it is OK. If cycles are allowed, we want to avoid searching any component twice. A similar problem exists when we are trying to determine when a file can be deleted. A value of 0 in the reference count means no more references to the file/directory can be deleted.

Links in UNIX

UNIX supports two types of links:

- **Hard links**
- **Soft (symbolic) links**

The ln command is used to create both links, ln –s is used to create a soft link

- `ln [options] existing-file new-file`
- ln [options] existing-file-list directory

File System Mounting

A file system is best visualized as a tree, rooted at /. /dev, /etc, /usr, and other directories in the root directory are branches, which may have their own branches, such as /etc/passwd, /usr/local, and /usr/bin. Filling up the root file system is not a good idea, so splitting /var from / is a good idea.

File System Mounting

A file system is best visualized as a tree, rooted at /. /dev, /etc, /usr, and other directories in the root directory are branches, which may have their own branches, such as /etc/passwd, /usr/local, and /usr/bin. Filling up the root file system is not a good idea, so splitting /var from / is a good idea.

Mounting in UNIX

All files accessible in a Unix system are arranged in one big tree, the file hierarchy, rooted at /. These files can be spread out over several devices. The `mount` command serves to attach the file system found on some device to the big file tree. Conversely, the umount command will detach it again. Here is the syntax of the `mount` command `mount -t type device dir`

This command tells the kernel to attach the file system found on *device* (which is of type *type*) at the directory *dir*. The previous contents (if any) and owner and mode of *dir* become invisible. As long as this file system remains mounted, the pathname *dir* refers to the root of the file system on *device*.

File Sharing

Sharing of files on multi-user systems is desirable. People working on the same project need to share information. For instance: software engineers working on the same project need to share files or directories related to the project Sharing may be done through

- **Duplicating files:** Make copies of the file and give them to all team members. This scheme works well if members of the team are to work on these shared files sequentially. If they work on the files simultaneously, the copies become inconsistent and no single copy reflects the works done by all members. However it is simple to implement.
- **Common login** for members of a team: The system admin creates a new user group and gives the member access to the new account. All files and directories created by any team member under this account and are owned by the team. This works well if number of teams is small and teams are stable. However a separate account is needed for the current project and the system administrator has to create a new account for every team
- Setting appropriate **access permissions.** Team members put all shared files under one member's account and the access permissions are set so all the members can access it. This scheme

works well if *only* this team's members form the user group. File access permissions can be changed using the `chmod` system call:

Protection

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus we could provide complete protection by prohibiting access. Alternatively we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access. File owner/creator should be able to control

- What can be done
- By whom

Several types of operations may be controlled:

- Read: read from the file
- Write: write or rewrite to the file
- Execute: Load the file into memory and execute it
- Append: Write new information at the end of the file
- Delete: Delete the file and free its space for possible reuse
- List: List the name and attributes of the file

UNIX Protection

UNIX recognizes three modes of access: **read**, **write**, and **execute** (r, w, x). The execute permission on a directory specifies permission to **search** the directory. The three classes of users are:

- **Owner:** user is the owner of the file
- **Group:** someone who belongs to the same group as the owner
- **Others:** everyone else who has an account on the system

A user's access to a file can be specified by an octal digit. The first bit of the octal digit specifies the read permission, the second bit specifies the write permission, and the third bit specifies the execute permission. A bit value 1 indicates permission for access and 0 indicates no permission.

Default Permissions

The default permissions on a UNIX/Linux system are 777 for executable files and directories and 666 for text files. You can use the `umask` command to set permission bits on newly created files and directories to 1, except for those bits that are set to 1 in the 'mask'. You can use the `chmod` command to set permissions on existing files and
directories.

Sample commands

chmod 700 ~ …….. Set permissions on home directory to 700
chmod 744 ~/file……. Set permissions on ~/file to 744
chmod 755 ~/directory… Set permissions on ~/directory 755
ls –l ~ …….. Display permissions and some other attributes for all files and directories in your home directory
ls –ld ~ …… Display permissions and some other attributes for your home directory
ls –l prog1.c …… Display permissions and some other attributes for prog1.c in your current directory
ls –ld ~/courses …….. Display permissions and some other attributes for your home directory

The **umask** command sets default permissions on newly created files and directories as

(default permissions – mask value)

Here are some sample commands

umask …… Display current mask value (in octal)
umask 022 …. Set mask value to octal 022 (turn off write permission for 'group' and 'others'
touch temp1 .. Create an empty file called temp1
ls –l temp1 …. Display default permissions and some other attributes for the temp1 file

File Control Block

A file control block is a memory data structure that contains most of the attributes of a file. In UNIX, this data structure is called inode (for index node).

In-Memory Data Structures

The following upper-level data structures needed for file system support.

- An in-memory partition table containing information about each mounted partition
- An in-memory directory structure that holds the directory information of recently accessed directories
- The system-wide open file table contains pointer to the FCB (UNIX inode) of each open file as well as read/write pointer
- The FCB for each open file
- The per process file descriptor table contains a pointer to the appropriate entry in the system wide open file table as well as other information

Space Allocation Methods

We now turn to some file system implementation issues, in particular space allocation techniques and free space management methods. Here are the three commonly used methods for file space allocation.

- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. The directory entry for each file contains starting block number and file size (in blocks). Disk addresses define a linear ordering on the disk. With this ordering, assuming only one job is

accessing the disk, accessing b+1 block after block b normally requires no head movement. When head movement is needed it is only one track. Both sequential and direct access can be supported by contiguous allocation. For direct access to block I of a file that starts at block b we can immediately access block b+i. Best-fit, first-fit, or worst-fit algorithms are the strategies used to select a hole from the set of available holes. Neither first fit, nor best fit is clearly best in terms of both time and storage utilization, but first fit is generally faster. These algorithms suffer from the problem of external fragmentation. As files are allocated or deleted, the free disk is broken into little pieces. This situation results in **external fragmentation** of disk (similar to external fragmentation of main memory due to segmentation). Disk defragmenter utility needs to be used for removing external fragmentation.

Linked Allocation

Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. There is no wastage of space. However, a major disadvantage with linked allocation is that it can be used only for sequential access files. To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get back to the *i*th block. Consequently it is inefficient to support a direct access capability for linked allocation files.

Index Allocation

Indexed allocation brings all the pointers to the block together into a disk block, known as the **index block**. Each file has its own index block, which is an array of disk block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block. To read the *i*th block, we use the pointer in the *i*th index-block entry to find and read the desired block Indexed allocation supports direct access without suffering from external fragmentation because any free block on the disk may satisfy a request for more space. Depending on the disk block size and file system size, a file may need more than one index block. In this case there are two ways of organizing index blocks:

Linked scheme (linked list of index blocks)

An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

Multi-level index scheme

The second method of handling multiple index blocks is to maintain multi-level indexing.

UNIX Space Allocation

The UNIX file manager uses a combination of indexed allocation and linked lists for the index table. It maintains 10-15 direct pointers to file blocks, and three indirect pointers (one-level indirect, two-level indirect, and three-level indirect), all maintained in file's inode,

File Allocation Table (FAT)

The file system on an MS-DOS floppy disk is based on **file allocation table** (FAT) file system in which the disk is divided into a reserved area (containing the boot program) and the actual file

allocation tables, a root directory and file space. Space allocated for files is represented by values in the allocation table, which effectively provide a linked list of all the blocks in the file. Each entry is indexed by a block number and value in a table location contains block number for the next file block. First block number for a file is contained in file's directory entry. Special values designate end of file, unallocated and bad blocks.

Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files if possible. To keep track of free disk space, the system maintains a **free-space list.** The free space list records all *free* disk blocks-those not allocated to some file or directory. To create a file we search the free-space list for the required amount of space and allocate the space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free space list.

Bit vector

Frequently, the free space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1;if it is allocated, the bit is 0. This approach is relatively simple and efficient in finding the first free block or n consecutive free blocks on the disk.

Linked list (free list)

Another approach to free space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. The first block contains a pointer to the next free disk block and so on. However this scheme is not efficient. To traverse the list, we must read each block, which requires substantial I/O time. It cannot get contiguous space easily.

Grouping

A modification of free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 blocks of these blocks are actually free. The last block contains addresses of the next n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

Counting

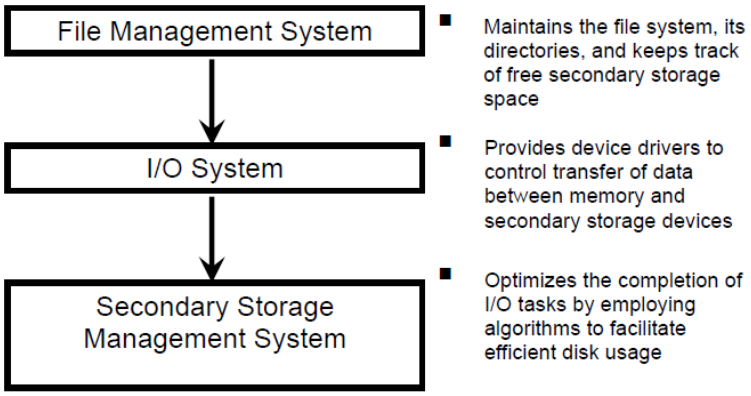
We keep the address of the first free block and the number n of free contiguous blocks that follow the first block in each entry of a block. This scheme is good for contiguous allocation. Although each entry requires more space, the overall list will be shorter.

I/O Operations

A number of I/O operations (inserting, deleting, and reading a file block) needed for the various allocation schemes indicate the goodness of these schemes.

Secondary Storage Management

The following diagram shows the hierarchy of three kernel modules used for mapping user view of directory structure, free space management, file I/O, and secondary storage management. We have discussed some details of the top-most layer. We will not discuss details of the I/O system.



Three layers of file OS kernel used for managing user view of files, file operations, and file storage to disk

Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary storage medium but the access is much slower than for disks. Thus tapes are currently used mainly for backup, for storage of infrequently used information etc. Modern disk drives are addressed as large one dimensional array of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to choose a different logical block size, such as 1024 bytes. The one dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Block 0 is the first sector of the first track on the outermost sector. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to the innermost.

Disk Scheduling

One of the responsibilities of the operating system is to use the computer system hardware efficiently. For the disk drives, meeting this responsibility entails having a fast access time and disk bandwidth. The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector. The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head. The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Some of the popular disk-scheduling algorithms are:

- First-come-first-serve (FCFS)
- Shortest seek time first (SSTF)
- Scan
- Look
- Circular scan (C-Scan)
- Circular look (C-Look)

First Come First Served Scheduling

The simplest form of disk scheduling is FCFS. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

SSTF Scheduling

It seems reasonable to service all the requests close to the current head position, before moving the head far away to service other requests. This assumption is the basis for the shortest seek time first (SSTF) algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

Scan

In the Scan algorithm the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues. The head continuously scans back and forth across the disk.

Look algorithm

This algorithm is a version of SCAN. In this algorithm the arm only goes as far as the last request in each direction, then reverses direction immediately, serving requests while going in the other direction. That is, it looks for a request before continuing to move in a given direction. For the given reques queue, the total head movement (seek distance) for the Look algorithm is 208.

C-Scan and C-Look algorithms

In the C-Scan and C-Look algorithms, when the disk head reverses its direction, it moves all the way to the other end, without serving any requests, and then reverses again and starts serving requests. In other words, these algorithms serve requests in only one direction.