3.3 DEAD LOCKS

3.3.1. System Model

● For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.

● Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.

● By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".

● Some categories may have a single resource.

● In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence: 1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. Example: system calls open( ), malloc( ), new( ), and request( ). 2. Use - The process uses the resource.

Example: prints to the printer or reads from the file.

3. Release - The process relinquishes the resource. so that it becomes available for other processes.

Example: close( ), free( ), delete( ), and release( ).

● For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)

● A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.) 3.3.2. Deadlock Characterization Necessary Conditions: There are four conditions that are necessary to achieve deadlock:

Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released. Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process. No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it. Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition

implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Resource-Allocation Graph In some cases deadlocks can be understood more clearly through the use of Resource-Allocation Graphs, having the following properties: *A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.) *A set of processes, { P1, P2, P3, . . ., PN } *Request Edges - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available. *Assignment Edges - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj. Note that a request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.) For example:

*If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.) See the example in Figure 7.2 above.

*If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists. *If a resource category contains more than one instance, then the presence of a cycle in the resourceallocation graph indicates the possibility of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:

3.3.3. Methods for Handling Deadlocks

Generally speaking there are three ways of handling deadlocks: Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process,

depending on the particular algorithm.) Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.

If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

3.3.4. Deadlock Prevention Deadlocks can be prevented by preventing at least one of the four required conditions: Mutual Exclusion Shared resources such as read-only files do not lead to deadlocks. Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process. Hold and Wait To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

i. Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later. ii. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it. iii. Either of the methods described above can lead to starvation if a process requires one or more popular resources. No Preemption Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible. i. One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to reacquire the old resources along with the new resources in a single request, similar to the previous discussion. ii. Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting. iii. Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives. Circular Wait i. One way to avoid circular wait is to number all resources, and to require that processes request resources only

in strictly increasing (or decreasing) order. ii. In other words, in order to request resource Rj, a process must first release all Ri such that i >= j. iii. One big challenge in this scheme is determining the relative ordering of the different resources

Deadlock Avoidance The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions. This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.) In some algorithms the scheduler only needs to know the maximum number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order. When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted. A resource allocation state is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system. Safe State i. A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state. ii. More formally, a state is safe if there exists a safe sequence of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.) iii. If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

Fig: Safe, unsafe, and deadlocked state spaces. For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

Maximum Needs Current Allocation P0 10 5 P1 4 2 P2 9 2

i: What happens to the above table if process P2 requests and is granted one more tape drive? ii. Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm i. If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs. ii. In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges, noted by dashed lines, which point from a process to a resource that it may request in the future. iii. In order for this technique to work, all claim edges must be added to the graph for any

particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.) iv. When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge. v. This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect. Consider for example what happens when process P2 requests resource R2:

Fig: Resource allocation graph for dead lock avoidance The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

Fig: An Unsafe State in a Resource Allocation Graph

Banker's Algorithm i. For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen. ii. The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.) iii. When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system. iv. When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely. Data Structures for the Banker's Algorithm Let n = number of processes, and m = number of resources types. N Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available n Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rjn Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rjn Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task Need [i,j] = Max[i,j] – Allocation [i,j] Safety Algorithm 1. Let Work and Finish be vectors of length m and n, respectively. Initialize: Work = Available Finish [i] = false for i = 0, 1, …, n- 1

2. Find and i such that both:

(a) Finish [i] = false

(b) Needi <= Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process Pi

Request = request vector for process Pi . If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. If Requesti <= Needi

go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Requesti £ Available, go to step 3. Otherwise Pi must wait, since resources are not available

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

lIf safe Þ the resources are allocated to Pi lIf unsafe Þ Pi must wait, and the old resource-allocation state is restored An Illustrative Example Consider the following situation:

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria. Example: P1 Request (1,0,2) Check that Request £ Available (that is, (1,0,2) <= (3,3,2) Þ true Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement. 3.3.6. Deadlock Detection i. If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow. ii. In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted. 6.1 Single Instance of Each Resource Type i. If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a wait-for graph. ii. A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below. iii. An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

As before, cycles in the wait-for graph indicate deadlocks. This algorithm must maintain the wait-for graph, and periodically search it for cycles. Several Instances of a Resource Type Available:

A vector of length m indicates the number of available resources of each type. Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. Request: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj . Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:

(a) Work = Available(b) For i = 1,2, …, n,

if Allocationi != 0, then

Finish[i] = false;

otherwise,

Finish[i] = true

2. Find an index i such that both:

(a) Finish[i] == false

(b) Requesti <=Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish[i] == false, for some i, 1 <= i <=n, then the system is in deadlock state.

Moreover, if Finish[i] == false, then Pi is deadlocked Algorithm requires an order of O(m x n2) operations to detect whether the system is in deadlocked state. Example of Detection Algorithm

Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T0:

Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

Detection-Algorithm Usage i. When should the deadlock detection be done? Frequently, or infrequently? The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.) ii. There are two obvious approaches, each with trade-offs: 1.Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently. 2.Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock

recovery can be more complicated and damaging to more processes. 3.(As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock checks periodically (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.) 7. Recovery From Deadlock There are three basic approaches to recovery from deadlock: i. Inform the system operator, and allow him/her to take manual intervention. ii. Terminate one or more processes involved in the deadlock iii. Preempt resources. Process Termination 1. Two basic approaches, both of which recover resources allocated to terminated processes:

i. Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary. ii. Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step. 2. In the latter case there are many factors that can go into deciding which processes to terminate next: i. Process priorities. ii. How long the process has been running, and how close it is to finishing. iii. How many and what type of resources is the process holding. (Are they easy to preempt and restore?) iv. How many more resources does the process need to complete. v. How many processes will need to be terminated vi. Whether the process is interactive or batch.

Resource Preemption When preempting resources to relieve deadlock, there are three important issues to be addressed: Selecting a victim - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above. Rollback - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.) Starvation - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

MEMORY MANAGEMENT

Memory management is concerned with managing the primary memory. Memory consists of array of bytes or words each with their own address. The instructions are fetched from the memory by the CPU based on the value program counter. Functions of memory management:

- Keeping track of status of each memory location.
- Determining the allocation policy.
- Memory allocation technique.
- De-allocation technique.

Address Binding: Programs are stored on the secondary storage disks as binary executable files. When the programs are to be executed they are brought in to the main memory and placed within a process. The collection of processes on the disk waiting to enter the main memory forms the input queue. One of the processes which are to be executed is fetched from the queue and placed in the main memory. During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space. During execution the process will go through different steps and in each step the address is represented in different ways. In source program the address is symbolic. The compiler converts the symbolic address to re-locatable address. The loader will convert this re-locatable address to absolute address. Binding of instructions and data can be done at any step along the way: Compile time:-If we know whether the process resides in memory then absolute code can be generated. If the static address changes then it is necessary to re-compile the code from the beginning. Load time:-If the compiler doesn't know whether the process resides in memory then it generates the re-locatable code. In this the binding is delayed until the load time. Execution time:-If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.

Logical versus physical address:

The address generated by the CPU is called logical address or virtual address. The address seen by the memory unit i.e., the one loaded in to the memory register is called the physical address. Compile time and load time address binding methods generate some logical and physical address. The execution time addressing binding generate different logical and physical address. Set of logical address space generated by the programs is the logical address space. Set of physical address corresponding to these logical addresses is the physical address space. The mapping of virtual address to physical address during run time is done by the hardware device called memory management unit

(MMU). The base register is also called re-location register. Value of the re-location register is added to every address generated by the user process at the time it is sent to memory.

Dynamic re-location using a re-location registers

The above figure shows that dynamic re-location which implies mapping from virtual addresses space to physical address space and is performed by the hardware at run time. Re-location is performed by the hardware and is invisible to the user dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.

Dynamic Loading:

For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory utilization. In dynamic loading the routine or procedure will not be loaded until it is called. Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.

Advantage: Gives better memory utilization. Unused routine is never loaded. Do not need special operating system support. This method is useful when large amount of codes are needed to handle in frequently occurring cases.

Dynamic linking and Shared libraries:

Some operating system supports only the static linking. In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time. With dynamic linking a “stub” is used in the image of each library referenced routine. A “stub” is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present. When “stub” is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory. This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library. More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the programs that are compiled with the new version are

affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called “shared library”.

3.1.1 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping. Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding. If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time. Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run. Swapping is constant by other factors: To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped.

3.1.2 Contiguous memory allocation:

One of the simplest method for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions. In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. When process terminates, the memory partition becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied. When the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory. When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available. If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes. The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:

 First bit:-Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process. Best bit:-It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process. Worst fit:-It allocates the largest hole to the process request. It searches for the largest hole in the entire list. Operating Systems

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes. All these algorithms suffer from fragmentation.

Memory Protection:

Memory protection means protecting the OS from user process and protecting process from one another. Memory protection is provided by using a re-location register, with a limit register. Re-location register contains the values of smallest physical address and limit register contains range of logical addresses.

(Re-location = 100040 and limit = 74600). The logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in re-location register. When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch. Since every address generated by the CPU is checked against these register we can protect the OS and other users programs and data from being modified.

Fragmentation: Memory fragmentation can be of two types: Internal Fragmentation External Fragmentation Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition. Eg:-If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes. External Fragmentation may be either minor or a major problem. One solution for over-coming external fragmentation is compaction. The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the relocation is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time. Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing the process to be allocated physical memory whenever the latter is available.

3.1.3 Segmentation Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack, heap etc. A logical address is a collection of segments. Each segment has a name and length. The address specifies both the segment name and the offset within the segments. The users specify address by using two quantities: a segment name and an offset. For simplicity the segments are numbered and referred by a segment number. So the logical address consists of <segment number, offset>.

Hardware support: We must define an implementation to map 2D user defined address in to 1D physical address. This mapping is affected by a segment table. Each entry in the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment resides and limit specifies the length of the segment.

The use of segment table is shown in the above figure: Logical address consists of two parts: segment number‘s’ and an offset‘d’ to that segment. The segment number is used as an index to segment table. The offset’d’ must be in between 0 and limit, if not an error is reported to OS. If legal the offset is added to the base to generate the actual physical address. The segment table is an array of base limit register pairs.

Protection and Sharing: A particular advantage of segmentation is the association of protection with the segments. The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal access to memory like attempts to write in to read-only segment. Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it. Segments are shared when the entries in the segment tables of two different processes points to same physical location. Sharing occurs at the segment table. Any information can

be shared at the segment level. Several segments can be shared so a program consisting of several segments can be shared. We can also share parts of a program. Advantages: Eliminates fragmentation. x Provides virtual growth. Allows dynamic segment growth. Assist dynamic linking. Segmentation is visible.

Differences between segmentation and paging:- Segmentation:

• Program is divided in to variable sized segments. x User is responsible for dividing the program in to segments. • Segmentation is slower than paging. • Visible to user. • Eliminates internal fragmentation. • Suffers from external fragmentation. • Process or user segment number, offset to calculate absolute address.

Paging:

• Programs are divided in to fixed size pages. • Division is performed by the OS. • Paging is faster than segmentation. • Invisible to user. • Suffers from internal fragmentation. • No external fragmentation. • Process or user page number, offset to calculate absolute address.

3.1.4 Paging

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation. Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

Basic Method: Physical memory is broken in to fixed sized blocks called frames (f). Logical memory is broken in to blocks of same size called pages (p). When a process is to be executed its pages are loaded in to available frames from backing store. The blocking store is also divided in to fixed-sized blocks of same size as memory frames. The following figure shows paging hardware:

Logical address generated by the CPU is divided in to two parts: page number (p) and page offset (d). The page number (p) is used as index to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit. The page size is defined by the hardware. The size of

a power of 2, varying between 512 bytes and 10Mb per page. If the size of logical address space is 2^m address unit and page size is 2^n, then high order m-n designates the page number and n low order bits represents page offset.

Eg:-To show how to map logical memory in to physical memory consider a page size of 4 bytes and physical memory of 32 bytes (8 pages). Logical address 0 is page 0 and offset 0. Page 0 is in frame 5. The logical address 0 maps to physical address 20. [(5*4) + 0]. logical address 3 is page 0 and offset 3 maps to physical address 23 [(5*4) + 3]. Logical address 4 is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to physical address 24 [(6*4) + 0]. Logical address 13 is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to physical address 9 [(2*4) + 1]. Hardware Support for Paging: The hardware implementation of the page table can be done in several ways: The simplest method is that the page table is implemented as a set of dedicated registers. These registers must be built with very high speed logic for making paging address translation. Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.

If the page table is large then the use of registers is not visible. So the page table is kept in the main memory and a page table base register [PTBR] points to the page table. Changing the page table requires only one register which reduces the context switching type. The problem with this approach is the time required to access memory location. To access a location [i] first we have to index the page table using PTBR offset. It gives the frame number which is combined with the page offset to produce the actual address. Thus we need two memory accesses for a byte.

The only solution is to use special, fast, lookup hardware cache called translation look aside buffer [TLB] or associative register. LB is built with associative register with high speed memory. Each register contains two paths a key and a value.

When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is return and searching is fast. TLB is used with the page table as follows: TLB contains only few page table entries. When a logical address is generated by the CPU, its page number along with the frame number is added to TLB. If the page number is found its frame memory is used to access the actual memory. If the page number is not in the TLB (TLB miss) the memory reference to the page table is made. When the frame number is obtained use can use it to access the memory. If the TLB is full of entries the OS must select anyone for replacement. Each time a new page table is selected the TLB must be flushed [erased] to ensure that next executing

process do not use wrong information. The percentage of time that a page number is found in the TLB is called HIT ratio. Protection:

Memory protection in paged environment is done by protection bits that are associated with each frame these bits are kept in page table. x One bit can define a page to be read-write or read-only. To find the correct frame number every reference to the memory should go through page table. At the same time physical address is computed. The protection bits can be checked to verify that no writers are made to read-only page. Any attempt to write in to read-only page causes a hardware trap to the OS. This approach can be used to provide protection to read-only, read-write or execute-only pages. One more bit is generally added to each entry in the page table: a valid-invalid bit.

A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or valid page. If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal addresses are trapped by using the valid-invalid bit. The OS sets this bit for each page to allow or disallow accesses to that page.

3.1.5 Structure Of Page Table

a. Hierarchical paging:

Recent computer system support a large logical address apace from 2^32 to 2^64. In this system the page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One simple solution to this problem is to divide page table in to smaller pieces. There are several ways to accomplish this division.

One way is to use two-level paging algorithm in which the page table itself is also paged. Eg:-In a 32 bit machine with page size of 4kb. A logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset.

b. Hashed page table: Hashed page table handles the address space larger than 32 bit. The virtual page number is used as hashed value. Linked list is used in the hash table which contains a list of elements that hash to the same location. Each element in the hash table contains the following three fields: Virtual page number x Mapped page frame value x Pointer to the next element in the linked list Working: Virtual page number is taken from virtual address. Virtual page number is hashed in

to hash table. Virtual page number is compared with the first element of linked list. Both the values are matched, that value is (page frame) used for calculating the physical address. If not match then entire linked list is searched for matching virtual page number. Clustered pages are similar to hash table but one difference is that each entity in the hash table refer to several pages.

c. Inverted Page Tables: Since the address spaces have grown to 64 bits, the traditional page tables become a problem. Even with two level page tables. The table can be too large to handle. An inverted page table has only entry for each page in memory. Each entry consisted of virtual address of the page stored in that read-only location with information about the process that owns that page. Each virtual address in the Inverted page table consists of triple <process-id , page number , offset >. The inverted page table entry is a pair <process-id , page number>. When a memory reference is made, the part of virtual address i.e., <process-id , page number> is presented in to memory sub-system. The inverted page table is searched for a match. If a match is found at entry I then the physical address <i , offset> is generated. If no match is found then an illegal address access has been attempted. This scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. If the whole table is to be searched it takes too long.

Advantage:

• Eliminates fragmentation. • Support high degree of multiprogramming. • Increases memory and processor utilization. • Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantage:

• Page address mapping hardware increases the cost of the computer. • Memory must be used to store the various tables like page tables, memory map table etc. • Some memory will still be unused if the number of available block is not sufficient for the address space of the jobs to be run.

d. Shared Pages:

Another advantage of paging is the possibility of sharing common code. This is useful in timesharing environment. Eg:-Consider a system with 40 users, each executing a text editor. If the text editor is

of 150k and data space is 50k, we need 8000k for 40 users. If the code is reentrant it can be shared. Consider the following figure

If the code is reentrant then it never changes during execution. Thus two or more processes can execute same code at the same time. Each process has its own copy of registers and the data of two processes will vary. Only one copy of the editor is kept in physical memory. Each user's page table maps to same physical copy of editor but date pages are mapped to different frames. So to support 40 users we need only one copy of editor (150k) plus 40 copies of 50k of data space i.e., only 2150k instead of 8000k.

3.2 Virtual Memory Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages noncontiguously in memory. However the entire process still had to be stored in memory somewhere. In practice, most real processes do not need all their pages, or at least not all at once, for several reasons: o Error handling code is not needed unless that specific error occurs, some of which are quite rare. o Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice. o Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-) The ability to load only the portions of processes that were actually needed (and only when they were needed) has several benefits: o Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer. o Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput. o Less I/O is needed for swapping processes in and out of RAM, speeding things up. Figure below shows the general layout of virtual memory, which can be much larger than physical memory:

Fig: Diagram showing virtual memory that is larger than physical memory

 Figure below shows virtual address space, which is the programmer's logical view of process memory storage. The actual physical layout is controlled by the process's page table. Note that the address space shown in Figure is sparse - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

Fig: Virtual address space Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits: o System libraries can be shared by mapping them into the virtual address space of more than one process. o Processes can also share virtual memory by mapping the

same block of memory to more than one process. o Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

Demand Paging The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a lazy swapper, although a pager is a more accurate term.

Fig: Transfer of a paged memory to contiguous disk space

Basic Concepts The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.) Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.) If the process only ever accesses pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded in to memory.

Fig: Page table when some pages are not in main memory. On the other hand, if a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps: 1. The memory address requested is first checked, to make sure it was a valid memory request. 2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in. 3. A free frame is located, possibly from a free-frame list. 4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.) 5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference. 6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

Fig: Steps in handling a page fault In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as pure demand paging. In theory each instruction could generate multiple page faults. In practice this is very rare, due to locality of reference. The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends

of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins. Performance of Demand Paging Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?

 There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a page fault rate of p, (on a scale from 0 to 1), the effective access time is now: (1 - p) * (200) + p * 8000000 = 200 + 7,999,800 * p which clearly depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses. A subtlety is that swap space is faster to access than the regular file system, because it does not have to go through the whole directory structure. For this reason some systems will transfer an entire process from the file system to swap space before starting up the process, so that future paging all occurs from the (relatively) faster swap space. Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. This approach is used by both Solaris and BSD Unix. Copy-on-Write The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.

Fig: Before process 1 modifies page C

Fig: After process 1 modifies page C Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared. Pages used to satisfy copy-on-write duplications are typically allocated using zero-fillon-demand, meaning that their previous contents are zeroed out before the copy proceeds. Some systems provide an alternative to the fork( ) system call called a virtual memory fork, vfork( ). In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence this

addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime. Page Replacement In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process. However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider: 1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.) 2. Put the process requesting more pages into a wait queue until some free frames become available. 3. Swap some process out of memory completely, freeing up its page frames.

4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as page replacement, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.

\

Fig: Ned for page replacement

Basic Page Replacement The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows: 1. Find the location of the desired page on the disk, either in swap space or in the file system. 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 an existing frame to be replaced, known as the victim frame. c) Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory. 3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change. 4. Restart the process that was waiting for this page.

Fig: Page replacement. Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a modify bit, or dirty bit to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. It should come as no surprise

that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set. There are two major requirements to implement a successful demand paging system. We must develop a frame-allocation algorithm and a page-replacement algorithm. The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available. The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance. Algorithms are evaluated using a given string of memory accesses known as a reference string, which can be generated in one of (at least) three common ways: 1. Randomly generated, either evenly distributed or with some distribution curve based on observed system behavior. This is the fastest and easiest approach, but may not reflect real performance well, as it ignores locality of reference.

2. Specifically designed sequences. These are useful for illustrating the properties of comparative algorithms in published papers and textbooks, (and also for homework and exam problems. :-)) 3. Recorded memory references from a live system. This may be the best approach, but the amount of data collected can be enormous, on the order of a million addresses per second. The volume of collected data can be reduced by making two important observations: i. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations. ii. Successive accesses within the same page can be treated as a single page request, because all requests after the first are guaranteed to be page hits. (Since there are no intervening requests for other pages that could remove this page from the page table.) Example: So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4) As the number of available frames increases, the number of page faults should decrease, as shown in below Figure:

Fig: Graph of page faults versus number of frames

FIFO Page Replacement A simple and obvious page replacement strategy is FIFO, i.e. first-in-first-out. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:

Fig: FIFO page-replacement algorithm Although FIFO is simple and easy, it is not always optimal, or even efficient. An interesting effect that can occur with FIFO is Belady's anomaly, in which increasing the number of frames available can actually increase the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames. Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:

Fig: Page-fault curve for FIFO replacement on a reference string Optimal Page Replacement The discovery of Belady's anomaly lead to the search for an optimal page-replacement algorithm, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly. Such an algorithm does exist, and is called OPT or MIN. This algorithm is simply "Replace the page that will not be used for the longest time in the future." For example, Figure 9.14 shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9. Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.) Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms. In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.

Fig: Optimal page-replacement algorithm LRU Page Replacement The prediction behind LRU, the Least Recently Used, algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest load time, and the latter looks at the oldest use time.) Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.) Below figure illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

Fig: LRU page-replacement algorithm LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used: 1. Counters. Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered. 2. Stack. Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure. Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for every memory access. Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of pagereplacement algorithms called stack algorithms, which can never exhibit Belady's anomaly. A stack algorithm is one in which the pages kept in memory for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.

Fig: Use of a stack to record the most recent page references

LRU-Approximation Page Replacement Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary. However many systems offer some degree of HW support, enough to approximate LRU fairly well. (In the absence of ANY hardware support, FIFO might be the best available choice.) In particular, many systems provide a reference bit for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail. Additional-Reference-Bits Algorithm Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int. o At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit. o The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared. o At any given time, the page with the smallest value for the reference byte is the LRU page. Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform. Second-Chance Algorithm The second chance algorithm is essentially a FIFO, except the reference bit is used to give pages a

second chance at staying in the page table. o When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner. o If a page is found with its reference bit not set, then that page is selected as the next victim. o If, however, the next page in the FIFO does have its reference bit set, then it is given a second chance: The reference bit is cleared, and the FIFO search continues. If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table. If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.

 If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement. As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely. This algorithm is also known as the clock algorithm, from the hands of the clock moving around the circular queue.

Fig: Second-chance (clock) page-replacement algorithm Enhanced Second-Chance Algorithm The enhanced second chance algorithm looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes: 1. (0, 0) - Neither recently used nor modified. 2. (0, 1) - Not recently used, but modified. 3. (1, 0) - Recently used, but clean. 4. (1, 1) - Recently used and modified. This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc. The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible. Counting-Based Page Replacement There are several algorithms based on counting the number of references that have been made to a given page, such as: o Least Frequently Used, LFU: Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to rightshift the counters periodically, yielding a time-decaying average reference count. o Most Frequently Used, MFU: Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them. In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well. Page-Buffering Algorithms There are a number of page-buffering algorithms that can be used in

conjunction with the aforementioned algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms: Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step. Keep a list of modified pages, and when the I/O system is otherwise idle, have it write these pages out to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim. Keep a pool of free frames, but remember what page was in it before it was made free. Since the data in the page is not actually cleared out when the page is freed, it can be made an active page again without having to load in any new data from disk. This is useful when an algorithm mistakenly replaces a page that in fact is needed again soon. Applications and Page Replacement Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management. Sometimes such programs are given a raw disk partition to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit. Allocation of Frames We said earlier that there were two important tasks in virtual memory management: a pagereplacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

Minimum Number of Frames The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction. If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch. Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also. The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process. Allocation Algorithms Equal Allocation - If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in

a free-frame buffer pool. Proportional Allocation - Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S. Variations on proportional allocation could consider priority of process rather than just their size. Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.) Global versus Local Allocation One big question is whether frame allocation (page replacement) occurs on a local or global level. With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process. With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not. Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.

 Global page replacement is overall more efficient, and is the more commonly used approach. Non-Uniform Memory Access The above arguments all assume that all memory is equivalent, or at least has equivalent access times. This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory. In these latter systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards. The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times. The presence of threads complicates the picture, especially when the threads get loaded onto different processors. Solaris uses an lgroup as a solution, in a hierarchical fashion based on relative latency. For example, all processors and RAM on a single board would probably be in the same lgroup. Memory assignments are made within the same lgroup if possible, or to the next nearest lgroup otherwise. (Where "nearest" is defined as having the lowest access time.) Thrashing If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling. But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults. A process that is spending more time paging than executing is said to be thrashing. Cause of Thrashing Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low. The problem is that when memory filled up and processes

started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt. Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)

Fig: Thrashing To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is? The locality model notes that processes typically access memory references in a given locality, making lots of references to the same general area of memory before moving periodically to a new locality, as shown in Figure 9.19 below. If we could just keep as many frames as are involved in the current locality, then page faulting would occur primarily on switches from one locality to another. (E.g. when one function exits and another is called.) Working-Set Model The working set model is based on the concept of locality, and defines a working set window, of length delta. Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in below Figure:

Fig: Working-set model

 The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed. The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched. The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references: o For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit. o Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared. o If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set. o Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead. Page-Fault

Frequency A more direct approach is to recognize that what we really want to control is the pagefault rate, and to allocate frames based on this directly measurable value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes. (I suppose a page-replacement strategy could be devised that would select victim frames based on the process with the lowest current page-fault frequency.)

Fig: Page-fault frequency

Memory-mapped files Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as memory-mapping a file. Basic Mechanism Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system. Note that file writes are made to the memory page frames, and are not immediately written out to disk. (This is the purpose of the "flush( )" system call, which may also be needed for stdout in some cases. See the time killer program for an example of this.) This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes. Some systems provide special system calls to memory map files and use direct disk access otherwise. Other systems map the file to process address space if the special system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case. File sharing is made possible by mapping the same file to the address space of more than one process, as shown in below Figure. Copy-on-write is supported, and mutual exclusion techniques may be needed to avoid synchronization problems.

Fig: Memory-mapped files.

 Shared memory can be implemented via shared memory-mapped files (Windows), or it can be implemented through a separate process (Linux, UNIX.) Shared Memory in the Win32 API Windows implements shared memory using shared memory-mapped files, involving three basic steps: 1. Create a file, producing a HANDLE to the new file. 2. Name the file as a shared object, producing a HANDLE to the shared object. 3. Map the shared object to virtual memory address space, returning its base address as a void pointer (LPVOID). This is illustrated in below Figure

Fig: Shared memory in Windows using memory-mapped I/O Memory-Mapped I/O All access to devices is done by writing into (or reading from) the device's registers. Normally this is done via special I/O instructions. For certain devices it makes sense to simply map the device's registers to addresses in the process's virtual address space, making device I/O as fast and simple as any other memory access. Video controller cards are a classic example of this. Serial and parallel devices can also use memory mapped I/O, mapping the device registers to specific memory addresses known as I/O Ports, e.g. 0xF8. Transferring a series of bytes must be done one at a time, moving only as fast as the I/O device is prepared to process the data, through one of two mechanisms: o Programmed I/O (PIO), also known as polling. The CPU periodically checks the control bit on the device, to see if it is ready to handle another byte of data. o Interrupt Driven. The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

Allocating Kernel Memory Previous discussions have centered on process memory, which can be conveniently broken up into page-sized chunks, and the only fragmentation that occurs is the average half-page lost to internal fragmentation for each process (segment.) There is also additional memory allocated to the kernel, however, which cannot be so easily paged. Some of it is used for I/O buffering and direct access by devices, example, and must therefore be contiguous and not affected by paging. Other memory is used for internal kernel data structures of various sizes, and since kernel memory is often locked (restricted from being ever swapped out), management of this resource must be done carefully to avoid internal fragmentation or other waste. (I.e. you would like the kernel to consume as little memory as possible, leaving as much as possible for user processes.) Accordingly there are several classic algorithms in place for allocating kernel memory structures. Buddy System The Buddy System allocates memory using a power of two allocator. Under this scheme, memory is always allocated as a power of 2 (4K, 8K, 16K, etc), rounding up to the next nearest power of two if necessary. If a block of the correct size is not currently available, then one is formed by splitting the next larger block in two, forming two matched buddies. (And if that larger size is not available, then the next largest available size is split, and so on.) One nice feature of the buddy system is that if the address of a block is exclusively ORed with the size of the block, the resulting address is the address of the buddy of the same size, which allows for fast and easy coalescing of free blocks back into larger blocks. o Free lists are maintained for every size block. o If the necessary block size is not available upon request, a free block from the next largest size is split into two buddies of the desired size. (Recursively splitting larger size blocks if necessary.) o When a block is freed, its buddy's address is calculated, and the free list for that size block is checked to see if the buddy is also free. If it is, then the two buddies are coalesced into one larger free block, and the process is repeated with successively larger free lists. o See the (annotated) Figure below for an example.

Fig: Buddy System Allocation

Slab Allocation Slab Allocation allocates memory to the kernel in chunks called slabs, consisting of one or more contiguous pages. The kernel then creates separate caches for each type of data structure it might need from one or more slabs. Initially the caches are marked empty, and are marked full as they are used. New requests for space in the cache is first granted from empty or partially empty slabs, and if all slabs are full, then additional slabs are allocated. (This essentially amounts to allocating space for arrays of structures, in large chunks suitable to the size of the structure being stored. For example if a particular structure were 512 bytes long, space for them would be allocated in groups of 8 using 4K pages. If the structure were 3K, then space for 4 of them could be allocated at one time in a slab of 12K using three 4K pages. Benefits of slab allocation include lack of internal fragmentation and fast allocation of space for individual structures Solaris uses slab allocation for the kernel and also for certain user-mode memory allocations. Linux used the buddy system prior to 2.2 and switched to slab allocation since then.

Fig: Slab Allocation

Other Considerations Prepaging The basic idea behind prepaging is to predict the pages that will be needed in the near future, and page them in before they are actually requested. If a process was swapped out and we know what its working set was at the time, then when we swap it back in we can go ahead and page back in the entire working set, before the page faults actually occur. With small (data) files we can go ahead and prepage all of the pages at one time. Prepaging can be of benefit if the prediction is good and the pages are needed eventually, but slows the system down if the prediction is wrong. Page Size There are quite a few trade-offs of small versus large page sizes: Small pages waste less memory due to internal fragmentation. Large pages require smaller page tables. For disk access, the latency and seek times greatly outweigh the actual data transfer times. This makes it much faster to transfer one large page of data than two or more smaller pages containing the same amount of data. Smaller pages match locality better, because we are not bringing in data that is not really needed. Small pages generate more page faults, with attending overhead. The physical hardware may also play a part in determining page size. It is hard to determine an "optimal" page size for any given system. Current norms range from 4K to 4M, and tend towards larger page sizes as time passes. TLB Reach TLB Reach is defined as the amount of memory that can be reached by the pages listed in the TLB. Ideally the working set would fit within the reach of the TLB. Increasing the size of the TLB is an obvious way of increasing TLB reach, but TLB memory is very expensive and also draws lots of power. Increasing page sizes increases TLB reach, but also leads to increased fragmentation loss. Some systems provide multiple size pages to

increase TLB reach while keeping fragmentation low. Multiple page sizes requires that the TLB be managed by software, not hardware. Inverted Page Tables

 Inverted page tables store one entry for each frame instead of one entry for each virtual page. This reduces the memory requirement for the page table, but loses the information needed to implement virtual memory paging. A solution is to keep a separate page table for each process, for virtual memory management purposes. These are kept on disk, and only paged in when a page fault occurs. (I.e. they are not referenced with every memory access the way a traditional page table would be.) Program Structure Consider a pair of nested loops to access every element in a 1024 x 1024 twodimensional array of 32-bit ints. Arrays in C are stored in row-major order, which means that each row of the array would occupy a page of memory. If the loops are nested so that the outer loop increments the row and the inner loop increments the column, then an entire row can be processed before the next page fault, yielding 1024 page faults total. On the other hand, if the loops are nested the other way, so that the program worked down the columns instead of across the rows, then every access would be to a different page, yielding a new page fault for each access, or over a million page faults all together. Be aware that different languages store their arrays differently. FORTRAN for example stores arrays in column-major format instead of row-major. This means that blind translation of code from one language to another may turn a fast program into a very slow one, strictly because of the extra page faults. I/O Interlock and Page Locking There are several occasions when it may be desirable to lock pages in memory, and not let them get paged out: Certain kernel operations cannot tolerate having their pages swapped out. If an I/O controller is doing direct-memory access, it would be wrong to change pages in the middle of the I/O operation. In a priority based scheduling system, low priority jobs may need to wait quite a while before getting their turn on the CPU, and there is a danger of their pages being paged out before they get a chance to use them even once after paging them in. In this situation pages may be locked when they are paged in, until the process that requested them gets at least one turn in the CPU.

Figure :The reason why frames used for I/O must be in memory. Operating-System Examples (Optional) Windows Windows uses demand paging with clustering, meaning they page in multiple pages whenever a page fault occurs. The working set minimum and maximum are normally set at 50 and 345 pages respectively. (Maximums can be exceeded in rare circumstances.) Free pages are maintained on a free list, with a minimum threshold indicating when there are enough free frames available. If a page fault occurs and the process is below their maximum, then additional pages are allocated. Otherwise some pages from this process must be replaced, using a local page replacement algorithm. If the amount of free frames falls below the allowable threshold, then working set

trimming occurs, taking frames away from any processes which are above their minimum, until all are at their minimums. Then additional frames can be allocated to processes that need them. The algorithm for selecting victim frames depends on the type of processor: On single processor 80x86 systems, a variation of the clock (second chance) algorithm is used. On Alpha and multiprocessor systems, clearing the reference bits may require invalidating entries in the TLB on other processors, which is an expensive operation. In this case Windows uses a variation of FIFO.

Solaris Solaris maintains a list of free pages, and allocates one to a faulting thread whenever a fault occurs. It is therefore imperative that a minimum amount of free memory be kept on hand at all times. Solaris has a parameter, lotsfree, usually set at 1/64 of total physical memory. Solaris checks 4 times per second to see if the free memory falls below this threshold, and if it does, then the page out process is started. Pageout uses a variation of the clock (second chance) algorithm, with two hands rotating around through the frame table. The first hand clears the reference bits, and the second hand comes by afterwards and checks them. Any frame whose reference bit has not been reset before the second hand gets there gets paged out. The Pageout method is adjustable by the distance between the two hands, (the handspan), and the speed at which the hands move. For example, if the hands each check 100 frames per second, and the handspan is 1000 frames, then there would be a 10 second interval between the time when the leading hand clears the reference bits and the time when the trailing hand checks them. The speed of the hands is usually adjusted according to the amount of free memory, as shown below. Slowscan is usually set at 100 pages per second, and fastscan is usually set at the smaller of 1/2 of the total physical pages per second and 8192 pages per second.

Fig: Solaris Page Scanner Solaris also maintains a cache of pages that have been reclaimed but which have not yet been overwritten, as opposed to the free list which only holds pages whose current contents are invalid. If one of the pages from the cache is needed before it gets moved to the free list, then it can be quickly recovered. Normally page out runs 4 times per second to check if memory has fallen below lotsfree. However if it falls below desfree, then page out will run at 100 times per second in an attempt to keep at least desfree pages free. If it is unable to do this for a 30-second average, then Solaris begins swapping processes, starting preferably with processes that have been idle for a long time. If free memory falls below minfree, then page out runs with every page fault. Recent releases of Solaris have enhanced the virtual memory management system, including recognizing pages from shared libraries, and protecting them from being paged out. 3