Unit-III

Memory Management : Requirements on the primary memory, mapping the address space to primary memory, dynamic memory for data structures, Memory allocation (Fixed partition Memory allocation strategy), Dynamic address Relocation, Memory Manger Strategies (Swapping, Virtual Memory, Shared Memory Multiprocessors). Virtual Memory : Address translation paging, Static and dynamic paging agorithms.

3) Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

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

Exercise

Part I (Very Short Answer)

1. What is situation of deadlock?
2. What is a mutual exclusion?
3. What is deadlock avoidance?
4. What is deadlock detection?
5. What do you mean by roll back.

Part II (Short Answer)

6. .Write in brief the necessary conditions of deadlock.
7. Write in brief about deadlock prevention.
8. Write in brief about process

Part-III (Long Answer)

9. Explain Resource allocation graphs with example.
10. Explain deadlock prevention in all four conditions.

■■■

Chapter

7

Memory Management

In a multiprogramming computer the operating system resides in part of memory and the rest is used by multiple processes. The task of subdividing the memory not used by the operating system among the various processes is called memory management.

In this chapter, we discuss various ways to manage memory. The memory management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, many algorithms require hardware support, although recent designs have closely integrated the hardware and operating system.

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them. Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses.

7.1 Memory Management Requirements

- Relocation - programs not loaded into a fixed point in memory but may reside in various areas.
- Protection - each process should be protected against unwanted interference by other processes either accidentally or intentionally. All memory references generated by a process must be checked at run time to ensure that they only reference memory that has been allocated to that process.
- Sharing - Protection mechanisms must be flexible enough to allow a number of processes to access the same area of memory. For example if a number of processes are executing the same program, it is advantageous to allow each program to access the same copy of the program.

- Logical Organization - Programs are normally organized into modules some of which are non modifiable and some of which contain data that may be modified. How does the operating system organize RAM which is a linear address space to reflect this logical structure?
- Physical Organization - Computer memory is organizes into a least two levels: main memory and secondary memory. The task of moving information between the two levels of memory should be the systems responsibility (not the programmers).

7.2 Memory Management Basics

- Program must be brought into memory and placed within a process for it to be run.
- Input queue - collection of processes on the disk that are waiting to be brought into memory to run the program.
- Normally, a process is selected from the input queue and is brought into the memory for execution. During execution of a process, it accesses instruction and data from the memory.
- Most systems allow a user process to be loaded in any part of the memory. This affects the addresses that a user program can access.
- User programs go through several steps before being run.

Addresses may be represented during these steps.

- Addresses in the source program are generally symbolic (variable name).
- A compiler typically binds these addresses to relocatable addresses (in terms of offsets).
- The linkage editor or loader binds relocatable addresses to absolute addresses (physical addresses).

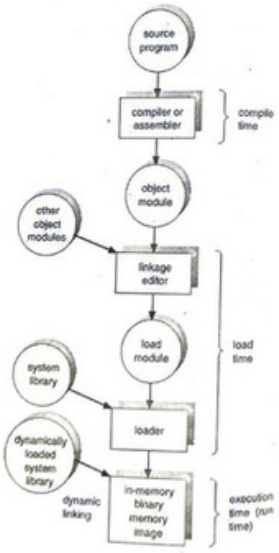


Figure 7.2 : Memory Management Basic

7.2.1 Loading and Loader

A loader is responsible to place a load module in main memory at some starting address. There are three approaches, which can be used for loading.

- **Absolute loading** - A given module is always loaded into the same memory location. All references in the load module must be absolute memory addresses. The address binding can be done at programming time, compile time or assembly time.
- **Relocatable loading** - The loader places a module in any desired location of the main memory. To make it possible, the compiler or assembler must generate relative addresses.
- **Dynamic loading** - Routine is not loaded until it is called resulting in better memory-space utilization (unused routine is never loaded). It is useful when large amounts of code are needed to handle infrequently occurring cases. No special support from the operating system is required implemented through program design. The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

7.2.2 Linking and Linker

The function of a linker is to take as input a collection of object modules and produce a load module consisting of an integrated set of programs and data modules to be passed to the loader. In each object module, there may be symbolic reference to location in other modules. The linking can be done either statically or dynamically.

- Static linking - A linker generally creates a single load module that is the contiguous joining of all the object modules with references properly changed. This is called static
- Dynamic linking - Linking is postponed until execution time. All external references are not resolved until the CPU executes the external call.

Small piece of code is a stub that is used to locate the appropriate memory resident library routine. Stub replaces itself with the address of the routine, and executes the routine. Operating system needed to check if routine is in processes' memory address. Dynamic linking is particularly useful for libraries.

7.2.3 Binding of Instructions and Data to Memory

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue. The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure 8.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as count). A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

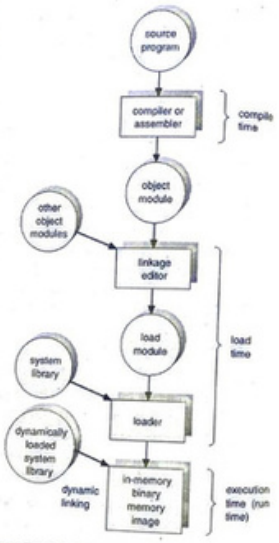


Figure 7.2.3 : Memory Management Basic

Address binding of instructions and data to memory addresses can happen at three different stages.

- Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- Load time: The compiler must generate re-allocate-able code if memory location is not known at compile time. The final binding is delayed until load time.
- Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

7.2.4 Logical vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

- Logical address - generated by the CPU; also referred to as virtual address.
- Physical address - address seen by the memory unit.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time addresses binding scheme results in differing logical and

physical addresses. In this case, we usually refer to the logical address as a virtual address. We use logical address and virtual address interchangeably in this text. The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

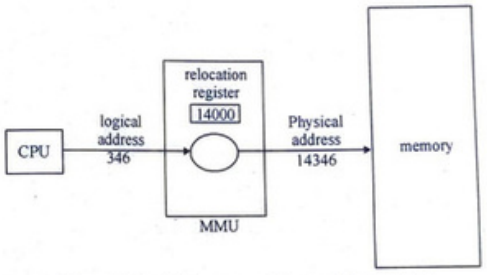


Figure 7.2.4 : Address space Memory Management

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

7.2.5 Memory Management Unit - (MMU)

- MMU is a hardware device that maps virtual to physical address at run-time.
- In a simple MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical addresses.

7.2.6 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine, first checks to see whether the other routine has been loaded or not. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used(and hence loaded) may be much smaller.

7.2.7 Dynamic Linking and Shared Library

Some operating systems support only static linking, in system language libraries are treated like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

7.3 Memory Allocation

Before discussing memory allocation further, we must discuss the issue of memory mapping and protection. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process. The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver [or other operating-system service] is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called transient operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution. To protect the operating system code and data by the user processes as well as protect user processes from one another using relocation register and limit register.

7.3.1 Single Partition Allocation

In this scheme Operating system is residing in low memory and user processes are executing in higher memory.

Advantages

- It is simple.
- It is easy to understand and use.

Disadvantages

- It leads to poor utilization of processor and memory.
- Users job is limited to the size of available memory.

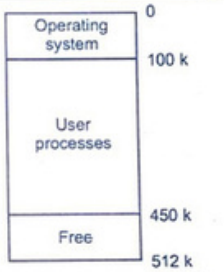


Figure 7.3.1 : Partition Allocation

7.3.2 Multiple-partition Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed sized partitions. There are two variations of this.

7.3.2.1 Fixed Equal-size Partitions

It divides the main memory into equal number of fixed sized partitions, operating system occupies some fixed portion and remaining portion of main memory is available for user processes.

Advantages

- Any process whose size is less than or equal to the partition size can be loaded into any available partition.
- It supports multiprogramming.

Disadvantages

- If a program is too big to fit into a partition use overlay technique.
- Memory use is inefficient, i.e., block of data loaded into memory may be smaller than the partition. It is known as internal fragmentation.

7.3.2.2 Fixed Variable Size Partitions

By using fixed variable size partitions we can overcome the disadvantages present in fixed equal size partitioning.

With unequal-size partitions, there are two ways to assign processes to partitions.

Use multiple queues:- For each and every process one queue is present, as shown in the figure below. Assign each process to the smallest partition within which it will fit, by using the scheduling queues, i.e., when a new

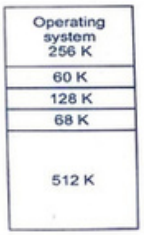


Figure 7.3.2 : (a) Size Partitions

process is to arrive it will put in the queue it is able to fit without wasting the memory space, irrespective of other blocks queues.

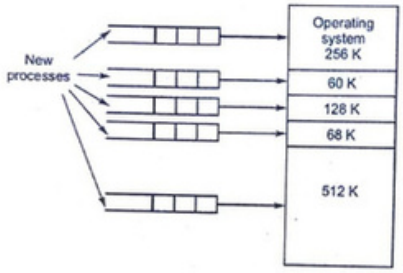


Figure 7.3.2 : (a) Size Partition

Advantages

- Minimize wastage of memory.

Disadvantages

- This scheme is optimum from the system point of view. Because larger partitions remains unused.

Use single queue:- In this method only one ready queue is present for scheduling the jobs for all the blocks irrespective of size. If any block is free even though it is larger than the process, it will simply join instead of waiting for the suitable block size.It is depicted in the figure below:

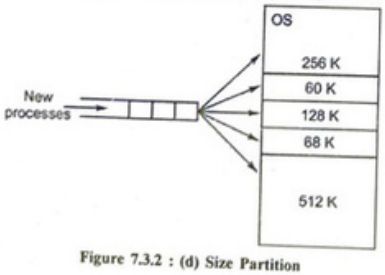


Figure 7.3.2 : (d) Size Partition

Advantages

- It is simple and minimum processing overhead.

Disadvantages

- The number of partitions specified at the time of system generation limits the number of active processes.
- Small jobs do not use partition space efficiently.

7.4 Contiguous Allocation Techniques

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one common method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management (Section 8.6). In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole.

Eventually as you will see, memory contains a set of holes of various sizes. As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can complete for CPU time. When a process terminates, it releases its memory which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied -that is, no available block of memory (or hole) is large enough to hold that process. The operating system

can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met. In general as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts.

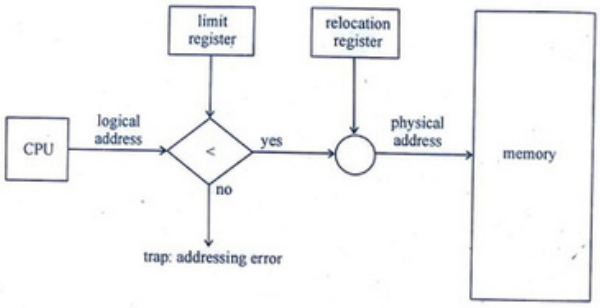


Figure 7.4 : Contigous Allocaiton

One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

- First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

7.5 Swapping

- A process needs to be in the memory to be executed. However, a process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Swapping needs a backing store - fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Roll out, roll in - swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped. Note that context switching time in this scheme is quite high.
- If we want to swap out a process, we must be sure that it is completely idle.
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps. A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.

When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called roll out, roll in. Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the

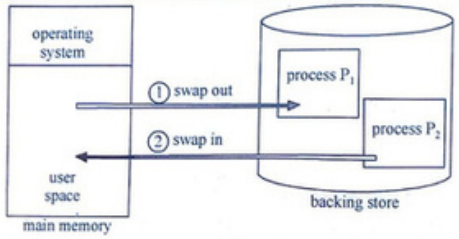


Figure 7.5 : Swapping of two processes using a disk as a backing store.

CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

7.6 Virtual Memory

In computing, virtual memory is a memory management technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Main storage as seen by a process or task appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit or MMU, automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.

Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory.

This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occured in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

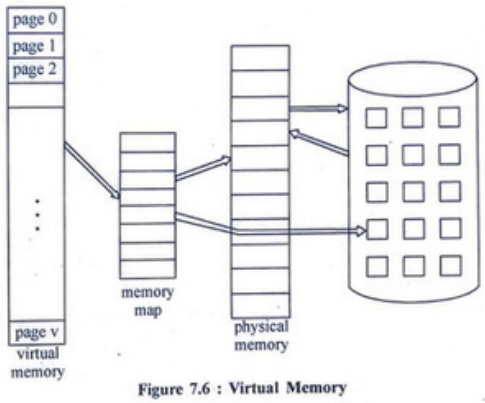


Figure 7.6 : Virtual Memory

- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 9.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of

physical memory available; she can concentrate instead on the problem to be programmed.

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address; say, address 0, and exists in contiguous memory, as shown in Figure 9.2. Recall from Chapter 8, though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure, that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls.

The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing (Section 8.4.4). This leads to the following benefits:

1. System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes. Typically, a library is mapped read-only into the space of each process that is linked with it.
2. Similarly, virtual memory enables processes to share memory. Recall from previous Chapter that two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure.
3. Virtual memory can allow pages to be shared during process creation with the fork() system call thus speeding up process creation.

7.6.1 Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to load pages only as they are

needed. This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

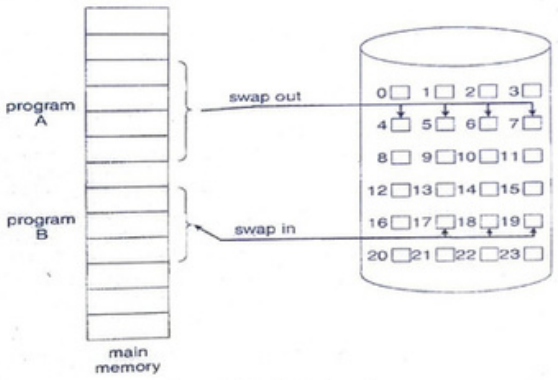


Figure 7.6.1 : Demand paging

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

7.7 Shared Memory Multiprocessors

The hardware evolution has reached the point where it becomes extremely difficult to further improve the performance of superscalar processors by either exploiting more instruction-level parallelism (ILP) or using new semiconductor technologies. The effort to increase processor performance by exploiting ILP follows the law of diminishing returns: new, more complex optimizations tend to cost more in terms of silicon as well as design effort and provide smaller and smaller performance gains. In addition, aggressive use of speculative caching and execution

techniques in modern superscalar's leads to poor energy efficiency - an important concern in both embedded systems with limited battery capacity and in server systems, where heat dissipation is a problem of growing importance. The natural solution is to rely on thread-level parallelism (TLP) rather than ILP to further increase the computational power of computer systems. The following forms of TLP are currently being used: explicit multithreading, chip-level multiprocessing (CMP), symmetric multiprocessing (SMP), asymmetric multiprocessing (ASMP), non-uniform memory access multiprocessing (NUMA), and clustered multiprocessing. With the exception of clustered multiprocessors, all of the above architectures provide all cores in the system with access to a shared physical address space. The shared memory organization has three major advantages over simpler private memory organization. First, because in shared-memory systems communication does not have to interfere with computation and because access to shared memory can be streamlined using hardware caching, shared memory provides an extremely efficient low-latency high-bandwidth communication mechanism. Second, shared memory provides a natural communication abstraction well understood by most developers. Third, the shared memory organization allows multithreaded or multi-process applications developed for uni-processors to run on shared-memory multiprocessors with minimal or no modifications. The goal of this report in to give an overview of issues and tradeoffs involved in memory hierarchy design for shared memory multiprocessors.

7.8 Segmentation and Paging

At its very roots virtual addressing is applied one of two ways: either via segmentation or paging. Segmentation involves the relocation of variable sized segments into the physical address space. Generally these segments are contiguous units, and are referred to in programs by their segment number and an offset to the requested data. Although a segmentation approach can be more powerful to a programmer in terms of control over the memory, it can also become a burden, as suggested by [1]. Efficient segmentation relies on programs that are very thoughtfully written for their target system. Even assuming best case scenarios, segmentation can lead to problems, however.

External fragmentation is the term coined for pieces of memory between segments, which may collectively provide a useful amount of memory, but are rendered useless by their non-contiguous nature. Since segmentation relies on memory that is located in single large blocks, it is very possible that enough free space is available to load a new module, but cannot be utilized. Segmentation may also suffer from internal fragmentation if segments are not variable-sized, where memory above the segment is not used by the program but is still "reserved" for it. Contrarily, paging provides a somewhat easier interface for programs, in that its operation tends to be more automatic and thus transparent. Each unit of transfer, referred to as a page, is of a fixed size and swapped by the virtual memory manager outside of the program's control. Instead of utilizing a segment/offset addressing approach, as seen in segmentation, paging uses a linear sequence of virtual addresses which are mapped to physical memory as necessary. Due to this addressing approach, a single program may refer to a series of many non-contiguous segments. Although some internal

fragmentation may still exist due to the fixed size of the pages, the approach virtually eliminates external fragmentation. The advantages of paging over segmentation generally outweigh their disadvantages.

7.9 Static Paging Algorithms

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

- Find the location of the desired page on the disk.
- Find a free frame:
  a. If there is a free frame, use it.
  b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
  c. Write the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the newly freed frame; change the page and frame tables.
- Restart the user process.

All paging algorithms function on three basic policies: a fetch policy, a replacement policy, and a placement policy. In the case of static paging, [1] describes the process with a shortcut: the page that has been removed is always replaced by the incoming page; this means that the placement policy is always fixed. Since we are also assuming demand paging, the fetch policy is also a constant; the page fetched is that which has been requested by a page fault.

Optimal Replacement Theory: In a best case scenario the only pages replaced are those that will either never be needed again, or have the longest number of page requests before they are referenced. This "perfect" scenario is usually used only as a benchmark by which other algorithms can be judged, and is referred to as either Belady's Optimal Algorithm or Perfect Prediction (PP). Such a feat cannot be accomplished without full prior knowledge of the reference stream, or a record of past behavior that is incredibly consistent. Although usually a pipe dream for system designers, it can be seen in very rare cases, such as large weather prediction programs that carry out the same operations on consistently sized data.

7.9.1 Random Replacement

On the flip-side of complete optimization is the most basic approach to page replacement: simply choosing the victim, or page to be removed, at random. Each page frame involved has an equal chance of being chosen, without taking into consideration the reference stream or locality principals. Due to its random nature, the behavior of this algorithm is quite obviously, random and unreliable. With most reference streams this method produces an unacceptable number of page faults, as well as victim pages being thrashed unnecessarily. Better performance can almost always

be achieved by employing a different algorithm. Most systems stopped experimenting with this method as early as the 1960's.

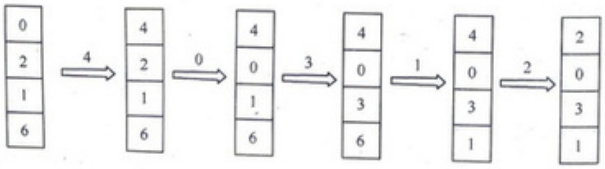
7.9.2 First-In, First-Out (FIFO)

First-in, first-out is as easy to implement as Random Replacement, and although its performance is equally unreliable or worse, its behavior does follow a predictable pattern. Rather than choosing a victim page at random, the oldest page (or first-in) is the first to be removed. Conceptually FIFO to a limited size queue, with items being added to the queue at the tail. When the queue fills (all of the physical memory has been allocated), the first page to enter is pushed out of head of the queue. Similar to Random Replacement, FIFO blatantly ignores trends, and although it produces less page faults, still does not take advantage of locality trends unless by coincidence as pages move along the queue.

A modification to FIFO that makes its operation much more useful is First-In Not-Used First-Out (FINUFO). The only modification here is that a single bit is used to identify whether or not a page has been referenced during its time in the FIFO queue. This utility, or referenced bit, is then used to determine if a page is identified as a victim. If, since it has been fetched, the page has been referenced at least once, its bit becomes set. When a page must be swapped out, the first to enter the queue whose bit has not been set is removed; if every active page has been referenced, a likely occurrence taking locality into consideration, all of the bits are reset. In a worst-case scenario this could cause minor and temporary thrashing, but is generally very effective given its low cost.

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses : × × × × × × × × ×



Fault Rate = 9/12 = 0.75

7.9.3 Least Recently Used (LRU)

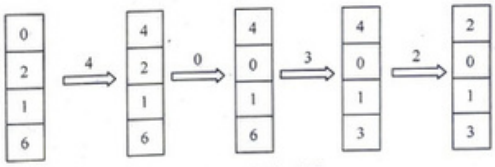
We have seen that an algorithm must use some kind of behavior prediction if it is to be efficient. One of the most basic page replacement approaches uses the usage of a page as an

indication of its "worth" when searching for a victim page: the Least Recently Used (LRU) Algorithm. LRU was designed to take advantage of "normal" program operation, which generally consists of a series of loops with calls to rarely executed code. In terms of the virtual addressing and pages, this means that the majority of code executed will be held in a small number of pages; essentially the algorithm takes advantage of the locality principal.

As per the previous description of locality, LRU assumes that a page recently referenced will most likely be referenced again soon. To measure the "time" elapsed since a page has been a part of the reference stream, a backward distance is stored [2]. This distance must always be greater than zero, the point for the current position in the reference stream, and can be defined as infinite in the case of a page that has never been referenced. Thus, the victim page is defined as the one with the maximum backward distance; if two or more points meet this condition, a page is chosen arbitrarily.

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses : × × × × × × × ×



Fault Rate = 8/12 = 0.67

7.9.4 6.9.3 Least Recently Used (LRU)

Often confused with LRU, Least Frequently Used (LFU) selects a page for replacement if has not been used often in the past. Instead of using a single age as in the case of LRU, LFU defines a frequency of use associated with each page. This frequency is calculated throughout the reference stream, and its value can be calculated in a variety of ways. The most common frequency implementation begins at the beginning of the page reference stream, and continues to calculate the frequency over an ever-increasing interval. Although this is the most accurate representation of the actual frequency of use, it does have some serious drawbacks. Primarily, reactions to locality changes will be extremely slow [1]. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less

than the pages associated with the previous program. Since the context has changed, and the pages swapped out will most likely be needed again soon (due to the new program's principal of locality), a period of thrashing will likely occur.

- Page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

7.10 Dynamic Paging Algorithms

All of the static page replacement algorithms considered have one thing in common: they assumed that each program is allocated a fixed amount of memory when it begins execution, and does not request further memory during its lifetime. Although static algorithms will work in this scenario, they are hardly optimized to handle the common occurrence of adjusting to page allocation changes. This can lead to problems when a program rapidly switches between needing relatively large and relatively small page sets or localities. Depending on the size of the memory requirements of a program, the number of page faults may increase or decrease rapidly; for Stack Algorithms, we know that as the memory size is decreased, the number of page faults will increase. Other static algorithms may become completely unpredictable. Generally speaking, any program can have its number of page faults statistically analyzed for a variety of memory allocations. At some point the rate of increase of the page faults (derivative of the curve) will peak; this point is sometimes referred to as the hysteresis point. If the memory allocated to the program is less than the hysteresis point, the program is likely to thrash its page replacement. Past the point, there is generally little noticeable change in the fault rate, making the hysteresis the target page allocation.

Since a full analysis is rarely available to a virtual memory controller, and that program behavior is quite dynamic, finding the optimal page allocation can be incredibly difficult. A variety of methods must be employed to develop replacement algorithms that work hand-in-hand with the locality changes present in complex programs. Dynamic paging algorithms accomplish this by attempting to predict program memory requirements, while adjusting available pages based on reoccurring trends. This policy of controlling available pages is also referred to as "prefetch" paging, and is contrary to the idea of demand paging. Although localities (within the scope of a set of operations) may change, states, it is likely that within the global locality (encompassing the smaller clusters), locality sets will be repeated. This idea of a "working set" of localities, and is the basis for most modern operating systems' replacement algorithms.

Exercise

Part I (Very Short Answer)

1. What is MMU?
2. What is a Linker?

3. What is a Loader?
4. What is Demand paging?

Part II (Short Answer)

5. Write in brief the requirements of memory management.
6. Write in brief about virtual memory.
7. Compare linker and loader

Part III (Long Answer)

8. Explain the static paging algorithms in detail with example.
9. Explain virtual memory on detail.
10. Explain fixed and variable partition allocation of memory.

■■■