LECTURE 4- Memory Management

Introduction:

Ideally what is desired is an infinitely large, inexpensive, fast memory that is also nonvolatile. Unfortunately technology does not provide for such.

The memory hierarchy is based on storage capacity, speed, and cost.

Lower in the hierarchy, storage capacities have lesser speed and cost. Consequently, most computers have a memory hierarchy, with a small amount of very fast, inexpensive, volatile cache memory, some number of megabytes of medium-speed, medium-price volatile main memory (RAM) and hundreds or thousands of megabytes of slow, cheap, non-volatile disk storage. It is the work of the operating systems to co-ordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its role is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and de-allocate it when they are done. It will also manage swapping between main memory and disk when main memory is not enough to hold all the processes.

Memory management

Memory management is one of the most important services provided by the operating system. An operating system has five major responsibilities for managing memory:

1. **Process isolation** – The operating system should prevent independent processes from interfering with the data and code segments of each other.
2. **Automatic allocation and management -** Programs should be dynamically allocated across the memory depending on the availability.
3. **Modular programming support -** The programmers should be able to define program modules and also be able to dynamically create/destroy/alter the size of modules.
4. **Protection and access control -** Different programs should be able to co-operate in sharing some memory space.
5. **Long-term storage -** Users and applications may require means for storing information for extended periods of time. This is implemented using a file system.

3.2 Basic memory management

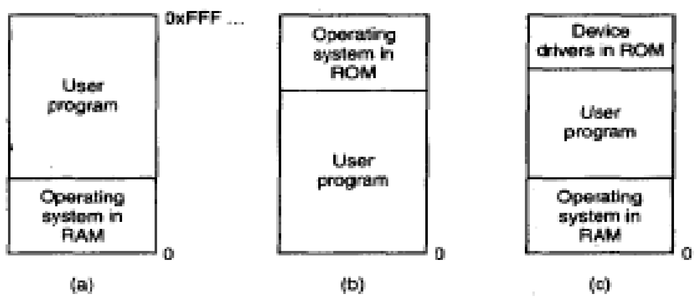
Memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk during execution (swapping and paging) and those that do not. Swapping and paging are caused by lack of sufficient main memory to hold all the programs at once.

3.2.1 Mono-programming without swapping or paging

The simplest possible memory management scheme is to run one program at a time, sharing the memory between the program and the operating system. In these scheme, there are three variations:

1. The operating system may be at the bottom of memory in RAM,
2. or Os at the top in ROM, or
3. Device drivers may at the top in ROM and the rest of the system in RAM.

The later model is common with small MS-DOS systems. On IBM PCs, the portion of the system in ROM is called the BIOS (Basic Input Output System).

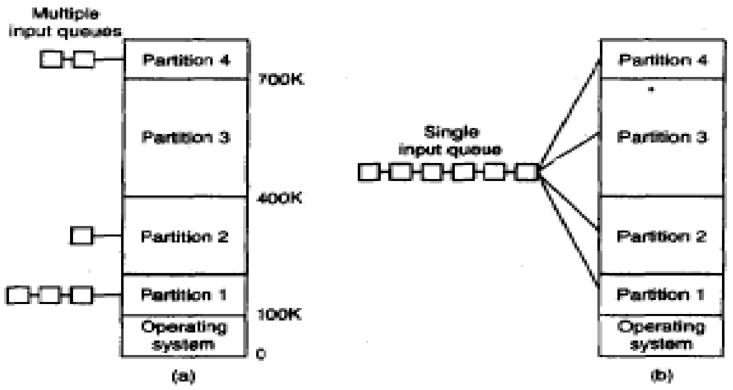


When the system is organized like this, only one process can be running. The operating system copies the requested program from disk to memory and executes it. When the operating system receives another command, it loads a new program into memory and overwrites the previous one.

Multiprogramming with fixed partitions

Mono programming is common with simple operating systems. It is desirable to have multiple processes running at once. On timesharing systems, having multiple processes in memory at the same time means that when one process is blocked waiting for I/O to finish, another one can use the CPU.

One way of achieving multiprogramming is to divide memory into unequal partitions. Partitioning can be done manually when the system is started up. When a job is started, it is put in the input queue for the smallest partition that is large enough to hold it. Since partitions are fixed, any unused space in the partition is lost.



(a) fixed memory partitions with separate queues(b) fixed memory partitions with single queues

The disadvantage this model is how to sort incoming jobs into separate queues especially when large partitions are empty and small partitions full. Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits it.

Note that this algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas, it is desirable to give small jobs the best service and not the worst.

A solution is to have at least one small partition around. Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than *k* times. Each time it is skipped over it is given one point. When it has gained *k* points, it may not be skipped again.

Relocation and protection

Multiprogramming introduces relocation and protection. Different jobs will now have to be run at different addresses. When a program is linked (main program, user-written procedures and library procedures are combined into a single address space), the linker must know at what address the program will start in memory.

Suppose the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker. If this program is loaded in partition 1, that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to 100K + 100. If the program is loaded into partition 2, it must be carried out as call to 200K + 100. This problem is known as the **relocation** problem.

Relocation during loading does not solve the protection problem. A malicious program can construct a new instruction and jump to it. In multi-user systems, it is undesirable to let processes read and write memory belonging to other users.

Hardware protects the base and limit registers from user programs.

3.3 Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.

The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

```
2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds
```

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. 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.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

- Page table requires extra memory space, so may not be good for a system having small RAM.

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment.

Virtual Memory

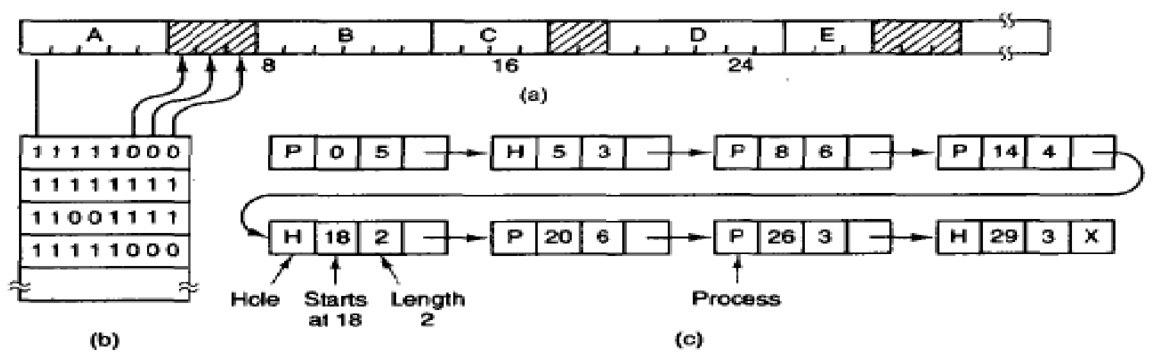
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.
The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.
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 occurred 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.
- 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.

Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware.

Memory Management with Bit Maps

When memory is allocated dynamically, the operating system must manage it. There are two ways to keep track of memory usage: **bit maps** and **linked lists**. With bit maps, memory is divided into allocation units. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied.



Memory management with bit maps

The size of the allocation unit is an important design issue.

- The smaller the allocation unit the larger the bit maps.
- If the allocation unit is large, the bit map will be smaller, but appreciable memory will be wasted in the last unit if the process size is not exact multiple of the allocation unit.

A bit map provides a simple way to keep track of memory in a fixed amount of memory because the size of the bit map depends on the size of memory and size of allocation unit.

Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a **process** or **hole** between two processes. Each entry in the list specifies a hole (H) or a process (P), the address at which it starts, the length and a pointer to the next entry. In this example, the segment list is kept sorted by address.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created or swapped in process. We assume that the memory manager knows how much memory to allocate.

The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for unused memory, except for the unlikely case of a perfect fit.

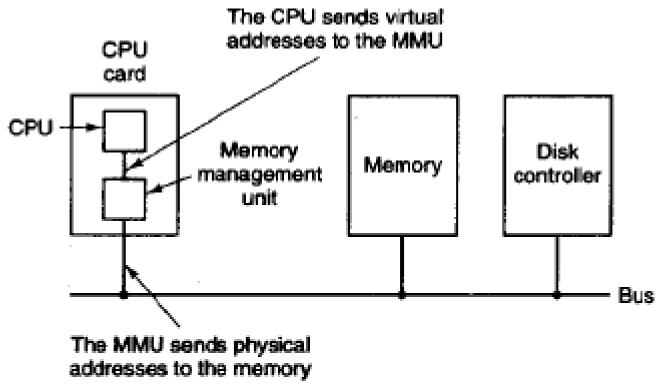
Another variation of first fit is the **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of the beginning as with the previous algorithm.

The **best-fit** algorithm searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that may be needed later, best fit, tries to find a hole that is close to the actual size needed. Best fit is slower than first fit and also results in more wasted memory because it tends to fill memory with tiny useless holes.

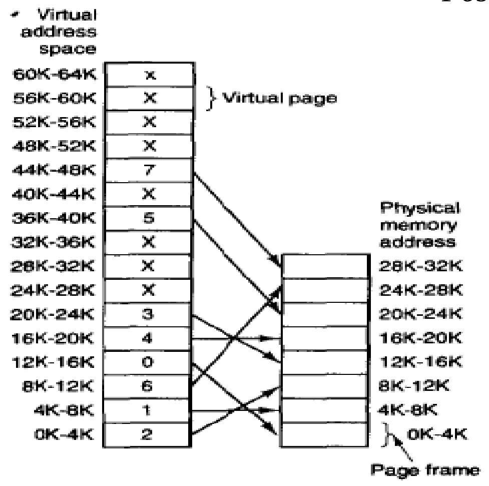
To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, i.e. take the largest available hole, so that the hole broken off will be big enough to be useful.

Paging

Most virtual memory systems use a technique called **paging**. On any computer, there exists a set of addresses that programs can produce. Addresses can be generated using indexing or base registers. These program-generated addresses are called **virtual addresses**.
On computers without virtual memory, the virtual address is put directly onto memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to memory bus. Instead they go to a Memory Management Unit (MMU). MMU maps the virtual addresses onto physical memory addresses.


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

Position of MMU



relation between virtual address and physical memory addresses

The virtual address space is divided into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are usually exactly the same size. Transfers between memory and disk are always in units of a page.

If a program tries to use an unmapped page, the MMU notices that the page is unmapped and causes the CPU to trap to the operating system. The trap is called a page fault. The operating system then picks one page frame and writes its content back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map and restarts the trapped instruction.

The page number is used as an index to the page table, yielding the number of the page frame corresponding to that virtual page. If the present/absent bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number is found in the page table is copied to the high-order 3 bits of the output register along with the 12-bit offset, which is unmodified. The 15-bit output register is then put onto the memory bus as the physical memory address.

Page Replacement Algorithms

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Optimal Page Replacement Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

Not recently Used (NRU) Page Replacement Algorithm

- 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.

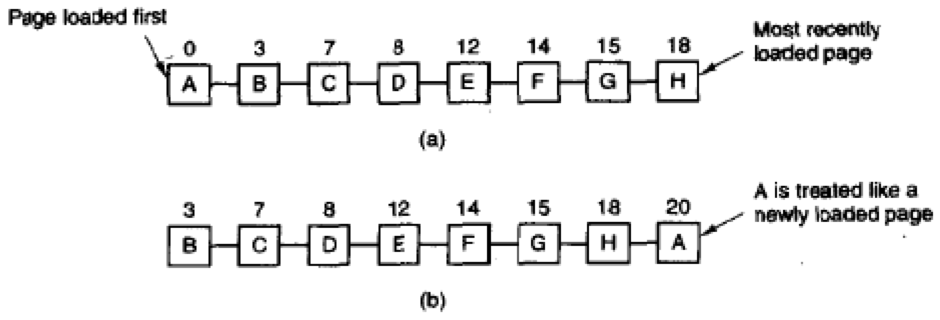
The First-In, First-Out (FIFO) Page Replacement Algorithm

- 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.

The second Chance Page Replacement Algorithm.

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced. If the R bit is 1, the bit is cleared and the page is put at the end of the list of pages, and its load time is updated as though it has just arrived in memory. The search continues.

The algorithm can be illustrated using the following example. Pages A through H are kept on a linked list sorted by the time they arrived in memory. Suppose that a page fault occurs at time 20. The oldest page is A, which arrived at time 0, when the process started. If A has the R bit cleared, it is evicted from memory. On the other hand if the R bit is set, A is put on to the end of the list and its load time reset to current time (20). The R bit is also cleared and the search for a suitable page continues with B.



The second chance looks for an old page that has not been referenced in the previous clock interval. If all have been referenced then second chance degenerates into a pure FIFO. Suppose that all the pages have their R bits set. One by one the operating system moves the pages to the end of the list and clears their R bit. Eventually, it comes back to page A, which now has its R bit cleared. At this point A is evicted. Thus the algorithm always terminates.

The Least recently Used (LRU) Page Replacement Algorithm

- The 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.