Unit II: Memory Management

1. What is no memory abstraction?

- The simplest memory abstraction is no abstraction at all. Early mainframe computers (before 1960), early minicomputers (before 1970), and early personal computers (before 1980) had no memory abstraction. Every program simply saw the physical memory. When a program executed an instruction like **MOV REGISTER1, 1000** the computer just moved the contents of physical memory location 1000 to REGISTER. In this way the model of memory presented to the programmer was just physical memory, a set of addresses from 0 to some maximum, each addresses corresponding to a cell containing some number of bits, commonly eight.
- Under these conditions, it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.
- Even with the model of memory being just physical memory, many options are possible. Three variations are demonstrated in Figure 1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Figure 1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Figure 1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Figure 1(c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used on some handheld computers and embedded systems. The third model was used by early personal computers (e.g., running MS- DOS), where the portion of the system in the ROM is called the BIOS (Basic Input Output System). Models (a) and (c) have the disadvantage that a bug in the user program can wipe out the operating system, maybe with disastrous results (such as garbling the disk).
- When the system is organized in this way, usually only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

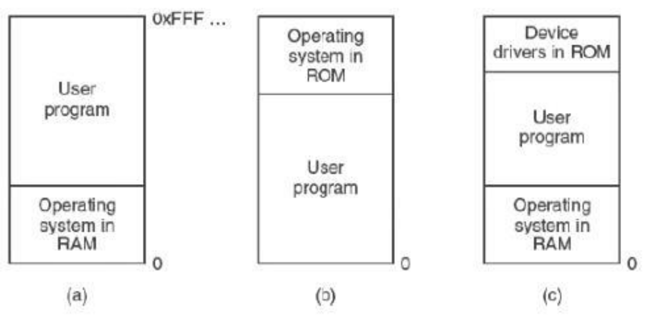


Figure 1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

One way to get some parallelism in a system with no memory abstraction is to program with multiple threads. Since all threads in a process are supposed to see the same memory image, the fact that they are forced to is not a problem. While this idea works, it is of limited use since what people often want is unrelated programs to be running at the same time, something the threads abstraction does not provide. Moreover, any system that is so primitive as to provide no memory abstraction is unlikely to provide a threads abstraction.

Running Multiple Programs without a Memory Abstraction

Nevertheless, even with no memory abstraction, it is possible to run multiple programs at the same time. What the operating system has to do is save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts.

Address Space

Two problems must be solved to allow various applications to be in memory at the same time without their interfering with each other: protection and relocation. A better solution is to invent a new abstraction for memory: the address space. Just as the process concept creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in. An address space is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

Base and Limit Registers

This simple solution uses a particularly simple version of dynamic relocation. What it does is map each process address space onto a different part of physical memory in a simple way.

2. Write a note on Swapping.

- If the physical memory of the computer is large enough to hold all the processes, the schemes explained so far will more or less do. But in practice, the total amount of RAM required by all the processes is often much more than can fit in memory. On a typical Windows or Linux system, something like 40-60 processes or more may be started up when the computer is booted.
- Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called swapping, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called virtual memory, allows programs to run even when they are only partially in main memory. Below we will examine swapping; in "VIRTUAL MEMORY"
- The operation of a swapping system is shown in Figure 2. In the beginning, only process A is in memory. Then processes B and C are created or swapped in from disk. In Figure 2(d) A is swapped out to disk. Then D comes in and B goes out. In the end A comes in again. Since A is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution. For instance, base and limit registers would work fine here.

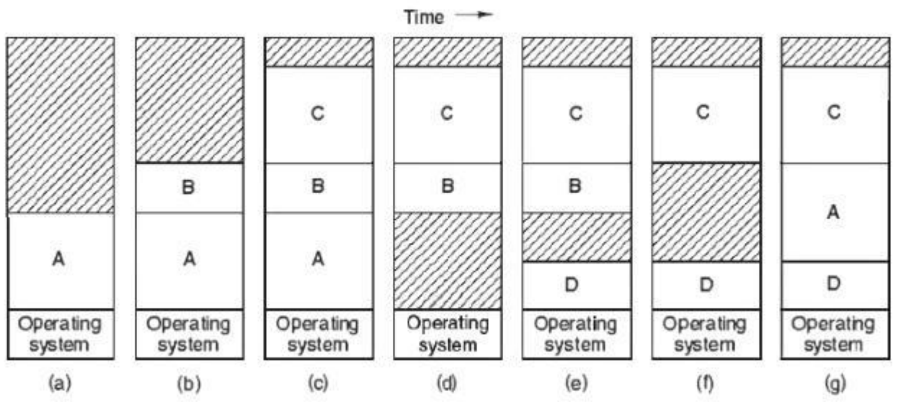


Figure 2. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

- When swapping makes multiple holes in memory, it is possible to merge them all into one big one by moving all the processes downward as far as possible. This technique is known as memory compaction. It is normally not done because it requires a lot of CPU time. For example, on a 1-GB machine that can copy 4 bytes in 20 nsec, it would take about 5 sec to compact all of memory.
- A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that

never changes, then the allocation is simple: the operating system allocates exactly what is required, no more and no less.

- If it is expected that most processes will grow as they run, it is perhaps a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Figure 3(a) we see a memory configuration in which space for growth has been allocated to two processes.

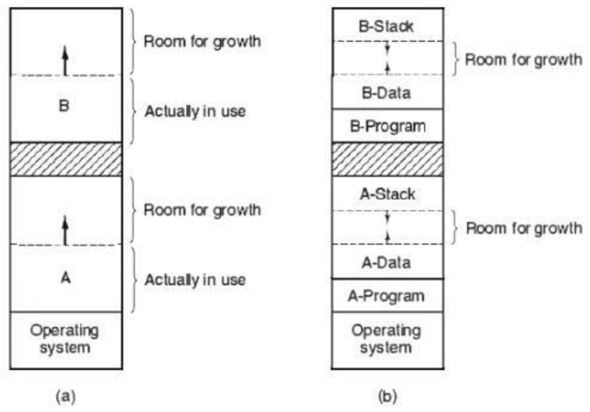


Figure 3. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

When memory is allocated dynamically, the operating system must manage it. Generally, there are two methods to keep track of memory usage: bitmaps and free lists. In this section and the next one we will study these two methods.

<u>Memory Management with Bitmaps</u>

With a bitmap, memory is divided into allocation units as small as a few words and as large as numerous kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 1 illustrates part of memory and the corresponding bitmap.

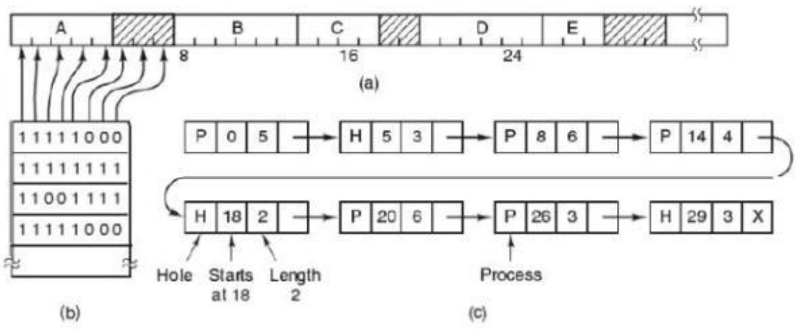


Figure 1. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

<u>Memory Management with Linked Lists</u>

A different way of keeping track of memory is to maintain a linked list of assigned and free memory segments, where a segment either includes a process or is an empty hole between two processes. The memory of Figure 1(a) is represented in Figure 1(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

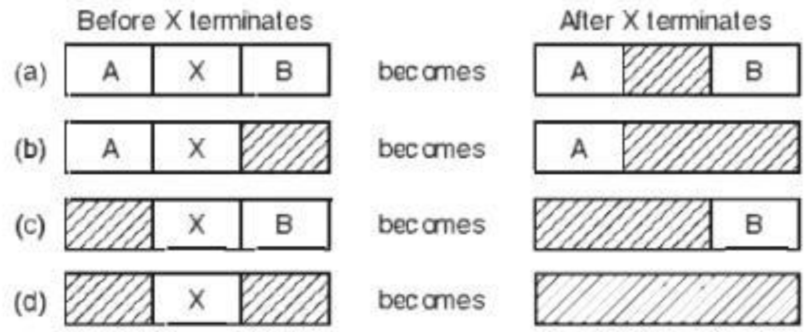


Figure 2. Four neighbor combinations for the terminating process, X.

3. With the suitable examples discuss First fit, Best fit, Worst fir strategies for memory allocation.

1. <u>First Fit</u>

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

<u>Advantage</u>

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

2. **Best Fit**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

3. **Worst fit**

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage

Reduces the rate of production of small gaps.

Disadvantage

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

4. **Next fit**

Next fit is a modified version of first fit. It begins as first fit to find a free partition. When called next time it starts searching from where it left off, not from the beginning.

4. Explain contiguous memory allocation.

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible.

- The memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. With this approach each process is contained in a single contiguous section of memory.
- One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. 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. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided.
- The contiguous memory allocation scheme can be implemented in operating systems with the help of two registers, known as the base and limit registers. When a process is executing in main memory, its base register contains the starting address of the memory location where the process is executing, while the amount of bytes consumed by the process is stored in the limit register. A process does not directly refer to the actual address for a corresponding memory location. Instead, it uses a relative address with respect to its base register. All addresses referred by a program are considered as virtual addresses. The CPU generates the logical or virtual address, which is converted into an actual address with the help of the memory management unit (MMU). The base address register is used for address translation by the MMU. Thus, a physical address is calculated as follows:

  Physical Address = Base register address + Logical address/Virtual address

- The address of any memory location referenced by a process is checked to ensure that it does not refer to an address of a neighboring process. This processing security is handled by the underlying operating system.
- One disadvantage of contiguous memory allocation is that the degree of multiprogramming is reduced due to processes waiting for free memory.

5. What is Virtual Memory? Give its advantages and Disadvantages.

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.

In real scenarios, most processes never need all their pages at once, for following reasons :

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain features of certain programs are rarely used.

Benefits of having Virtual Memory:

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

Virtual memory is a feature of an operating system (OS) that allows a computer to compensate for shortages of physical memory by temporarily transferring pages of data from random access memory (RAM) to disk storage. Eventually, the OS will need to retrieve the data that was moved to temporarily to disk storage -- but remember, the only reason the OS moved pages of data from RAM to disk storage to begin with was because it was running out of RAM. To solve the problem, the operating system will need to move *other* pages to hard disk so it has room to bring back the pages it needs right away from temporary disk storage. This process is known as *paging* or *swapping* and the temporary storage space on the hard disk is called a page file or a swap file.

Swapping, which happens so quickly that the end user doesn't know it's happening, is carried out by the computer’s memory manager unit (MMU). The memory manager unit may use one of several algorithms to choose which page should be swapped out, including Least Recently Used (LRU), Least Frequently Used (LFU) or Most Recently Used (MRU).

In a virtualized computing environment, administrators can use virtual memory management techniques to allocate additional memory to a virtual machine (VM) that has run out of resources. Such virtualization management tactics can improve VM performance and management flexibility.

6. What is the difference between a physical address and a virtual address?

Logical Address

- Logical address is the address generated by the CPU. From the perspective of a program that is running, an item seems to be located in the address provided by the logical address.
- Application programs that are running on the computer do not see the physical addresses. They always work using the logical addresses. The logical address space is the set of logical addresses generated by a program.
- Logical addresses need to be mapped to physical addresses before they are used and this mapping is handled using a hardware device called the Memory Management Unit (MMU). There are several mapping schemes used by the MMU.
- In the simplest mapping scheme, the value in the relocation register is added to each logical address produced by application programs before sending them to the memory. There are also some other complex methods that are used to generate the mapping.
- Address binding (i.e. allocating instructions and data in to memory addresses) can happen in three different times. Address binding can happen in compile time if the actual memory locations are known in advance and this would generate the absolute code in compile time.
- Address binding can also happen at load time if the memory locations are not known in advance. For this, re-locatable code needs to be generated at compile time.
- Furthermore, address binding can happen at execution time. This requires hardware support for address mapping. In compile time and load time address binding, logical and physical addresses are the same. But in execution time address binding, they are different.

Physical Address

Physical address or the real address is the address seen by the memory unit and it allows the data bus to access a particular memory cell in the main memory. Logical addresses generated by the CPU when executing a program are mapped in to physical address using the MMU. For example, using the simplest mapping scheme, which adds the relocation register (assume that the value in the register is y) value to the logical address, a logical address range from 0 to x would map to a physical address range y to x+y. this is also called the physical address space of that program. All the logical addresses need to be mapped in to physical addresses before they can be used.

Difference between a Logical Address and a Physical Address

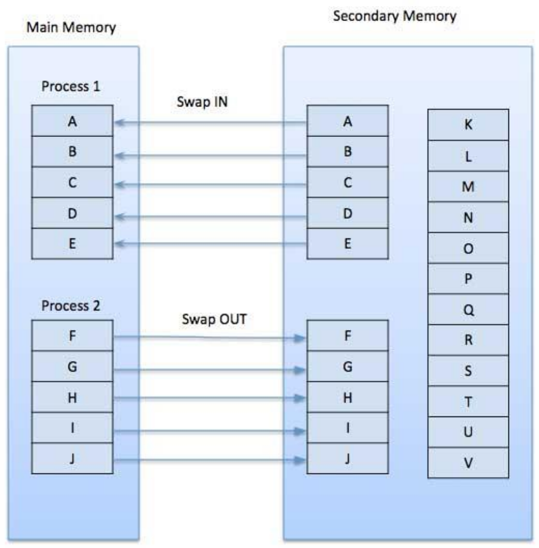
Logical address is the address generated by the CPU (from the perspective of a program that is running) whereas physical address (or the real address) is the address seen by the memory unit and it allows the data bus to access a particular memory cell in the main memory. All the logical addresses need to be mapped in to physical addresses before they can be used by the MMU. Physical and logical addresses are same when using compile time and load time address binding but they differ when using execution time address binding.

7. Explain Page Tables.

- Paging is a memory management technique in which the memory is divided into fixed size pages. Paging is used for faster access to data. When a program needs a page, it is available in the main memory as the OS copies a certain number of pages from your storage device to main memory. Paging allows the physical address space of a process to be noncontiguous.
- OS performs an operation for storing and retrieving data from secondary storage devices for use in main memory. Paging is one of such memory management scheme. Data is retrieved from storage media by OS, in the same sized blocks called as pages. Paging allows the physical address space of the process to be non contiguous. The whole program had to fit into storage contiguously.
- Paging is to deal with external fragmentation problem. This is to allow the logical address space of a process to be noncontiguous, which makes the process to be allocated physical memory.
- Paging is a method of writing data to, and reading it from, secondary storage for use in primary storage, also known as main memory. Paging plays a role in memory management for a computer's OS (operating system).
- In a memory management system that takes advantage of paging, the OS reads data from secondary storage in blocks called pages, all of which have identical size. The physical region of memory containing a single page is called a frame. When paging is used, a frame does not have to comprise a single physically contiguous region in secondary storage. This approach offers an advantage over earlier memory management methods, because it facilitates more efficient and faster use of storage.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program’s pages out to the disk or any of the new program’s pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program’s pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Table:

- The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.
- Thus the purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the

virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Structure of a Page Table Entry

The size varies from computer to computer, but 32 bits is a common size. The most important field is the Page frame number. After all, the goal of the page mapping is to output this value. Next to it we have the Present/absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

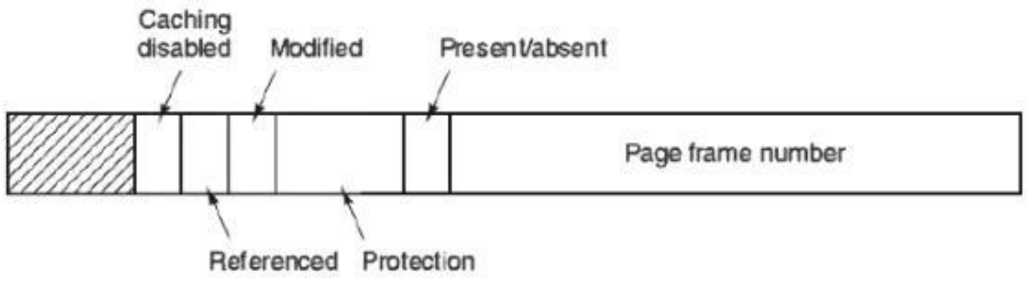


Figure 1. A typical page table entry.

- The Protection bits tell what kinds of access are allowed. In the simplest form, this field includes 1 bit, with 0 for read/write and 1 for read only. A more complicated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.
- The Modified and Referenced bits keep track of page usage. When a page is written to, the hardware automatically sets the Modified bit.
- The Referenced bit is set whenever a page is referenced, either for reading or writing.

8. Explain design issues with paging system.

In any paging system, two main issues must be faced:

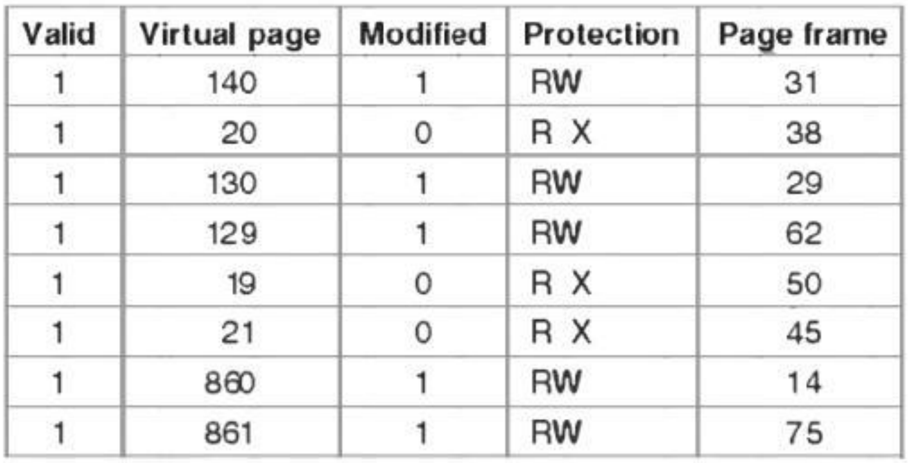
1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

- The first point is a result of the fact that the virtual-to-physical mapping must be done on every memory reference. All instructions must eventually come from memory and many of them reference operands in memory as well. Therefore, it is necessary to make one, two, or sometimes more page table references per instruction. If an instruction execution takes, say, 1 nsec, the page table lookup must be done in under 0.2 nsec to avoid having the mapping become a major bottleneck.

- The second point follows from the fact that all modern computers use virtual addresses of at least 32 bits, with 64 bits becoming gradually more common. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to consider. With 1 million pages in the virtual address space, the page table must have 1 million entries, and remember that each process needs its own page table (because it has its own virtual address space).
- The need for large, fast page mapping is a major restriction on the way computers are built. The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number, as shown in "Paging" Figure 3. When a process is started up, the operating system loads the registers with the process page table, taken from a copy kept in main memory. During process execution, no more memory references are required for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is unbearably expensive if the page table is large. Another is that having to load the full page table at every context switch hurts performance.
- At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the virtual-to-physical map to be changed at a context switch by reloading one register. Certainly, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction, making it very slow.

<u>Translation Lookaside Buffers</u>

We shall now consider widely implemented schemes for speeding up paging and for handling large virtual address spaces, starting with the earlier. The starting point of most optimization techniques is that the page table is in memory. Potentially, this design has a massive impact on performance.



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 1. A TLB to speed up paging.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffers 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.

9. Write a note on Page Tables for Large Memories.

TLBs can be used to speed up virtual address to physical address translation over the original page-table-in-memory scheme. But that is not the only problem we have to deal with. Another problem is how to deal with very large virtual address spaces. Below we will discuss two ways of dealing with them.

Multilevel Page Tables

- As a first approach, look at the use of a multilevel page table. A simple example is illustrated in Figure 1. In Figure 1(a) we have a 32-bit virtual address that is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit Offset field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. Particularly, those that are not required should not be kept around. Assume, for instance, that a process needs 12 megabytes, the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a huge hole that is not used.
- In Figure 1(b) we see how the two-level page table works in this example. On the left we have the top-level page table, with 1024 entries, corresponding to the 10-bit PT1 field. When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes.

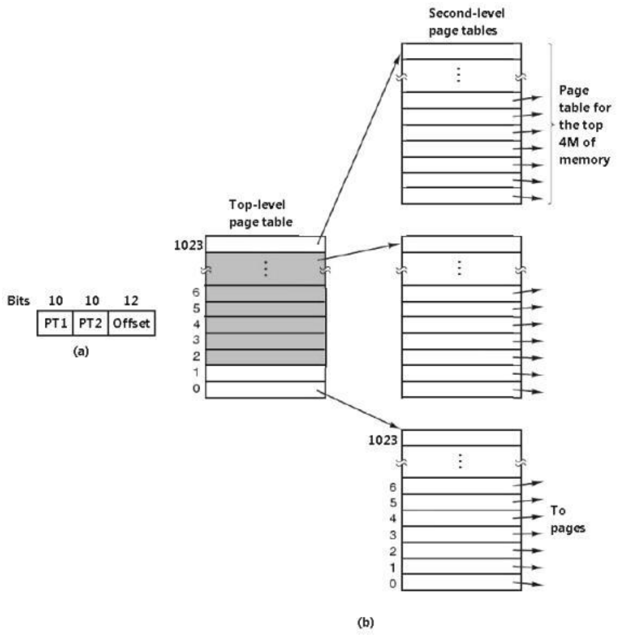


Figure 1. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

- The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

10. Explain any two 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,

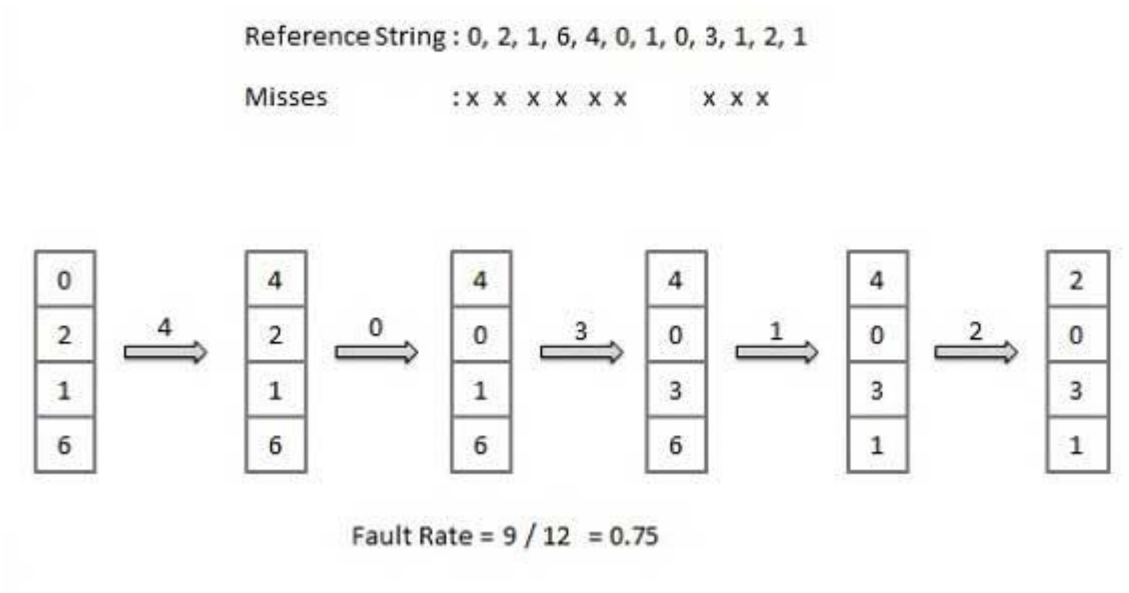
Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm

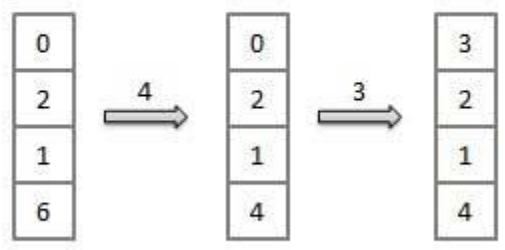
- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.


Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses : x x x x x x x x x
0 2 1 6
4
4 2 1 6
0
4 0 1 6
3
4 0 3 6
1
4 0 3 1
2
2 0 3 1
Fault Rate = 9 / 12 = 0.75

<u>Optimal Page algorithm</u>

- 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.

```
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses           : x  x  x  x  x              x
```

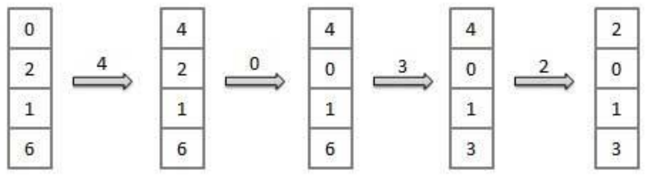


Fault Rate = 6 / 12 = 0.50

<u>Least Recently Used (LRU) algorithm</u>

- 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           : x  x  x  x  x  x        x     x
```



Fault Rate = 8 / 12 = 0.67

Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) 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.

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

The Second-Chance Page Replacement Algorithm

- A simple alteration 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 immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. The operation of this algorithm, called second chance, is shown in Figure 1. In Figure 1(a) we see pages A through H kept on a linked list and sorted by the time they arrived in memory.

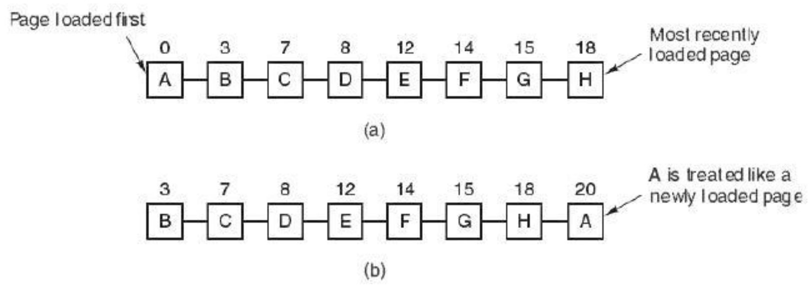


Figure 1. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is continually moving pages around on its list. A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Figure 2. The hand points to the oldest page.

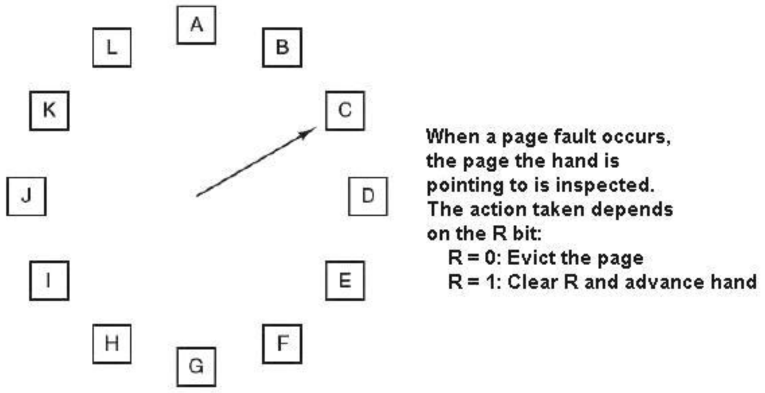

A
B
C
D
E
F
G
H
I
J
K
L
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Figure 2. The clock page replacement algorithm.

11.Explain Working Set Page Replacement Algorithm with an example.

- In a multiprogramming system, processes are often moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 100, or even 1000 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.
- Thus, many paging systems try to keep track of each process working set and make sure that it is in memory before letting the process run. This approach is called the working set model (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages before letting processes run is also called prepaging. Note that the working set changes over time.
- It has long been known that most programs do not reference their address space equally, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, w(k, t), is the working set. Because the k = 1 most recent references must have used all the pages used by the k > 1 most recent references, and possibly others, w(k, t) is a monotonically non decreasing function of k. The limit of w(k, t) as k becomes large is finite because a program cannot reference more pages than its address space includes, and few programs will use every single page. Figure 1 shows the size of the working set as a function of k.

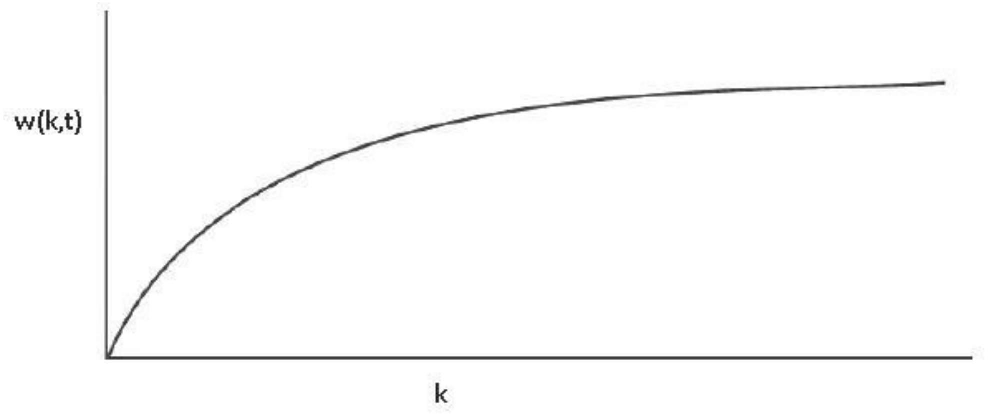


Figure 1. The working set is the set of pages used by the k most recent memory references. The function w(k, t) is the size of the working set at time t.

- To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and remove it. To implement such an algorithm, we need a particular way of determining which pages are in the working set. By definition, the working set is the set of pages used in the k most recent memory references (some authors use the k most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of k must be chosen in advance. Once some value has been selected, after every memory reference, the set of pages used by the most recent k memory references is uniquely determined.

<u>Working set page replacement algorithm:</u>

- Working set is the set of pages which is currently being used by a process. If the entire working set is in the memory, there will not be any page faults. The concept of this algorithm is to make sure that the working set of the process is in the memory before a process is running. On the occasion of a page fault, a page is located which is not in the working set and then it is removed. When there is a page fault entire page table is scanned to find a page for removal. Say, R is the referenced bit and t is the number of clock ticks.

1) For an entry if R = 1, current virtual time is set to "Time of last use field" in page table. This shows that the page was in use during page fault and will not be removed.
2) If R = 0 and Age is calculated by: (Current virtual time – Time of last use)
a) If Age > t, then the page is not in working set and is removed.
b) If Age ≤ t, then the page is in the working set and is not removed.

12.Explain WSClock Page Replacement Algorithm with an example.

The main working set algorithm is unwieldy, since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm, that is based on the clock algorithm but also uses the working set information, is called WSClock (Carr and Hennessey, 1981). Due to its simplicity of implementation and good performance, it is commonly used in practice.

WSClock page replacement algorithm:

It is an improvement of Working set page replacement algorithm. Here, pages are put in a circular list just like in a clock. For each entry, Time of last use field and value of referenced bit, R is checked.

1) For an entry if R = 1, page is not removed. R bit is set to 0 and clock hand is moved to next page.

2) If R = 0 and Age is calculated by: (Current virtual time – Time of last use)

a) If Age > t, then the page is not in working set and is replaced by a new page.

b) If Age ≤ t, then the page is in the working set and is not removed. Clock hand is moved to next page.

13. What is Local versus Global Allocation Policies?

- Consider Figure 1(a). In this figure, three processes, A, B, and C, make up the set of runnable processes. Assume A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A, or should it consider all the pages in memory? If it looks only at A's pages, the page with the lowest age value is A5, so we get the situation of Figure 1(b).
- However, if the page with the lowest age value is removed without regard to whose page it is, page B3 will be selected and we will get the situation of Figure 1(c). The algorithm of Figure 1(b) is said to be a local page replacement algorithm, whereas that of Figure 1(c) is said to be a global algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically assign page frames among the runnable processes. In this way the number of page frames allocated to each process varies in time.

- Normally, global algorithms work better, particularly when the working set size can vary over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are plenty of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to allocate to each process. One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in microseconds, on the other hand the aging bits are a crude measure spread over a number of clock ticks.

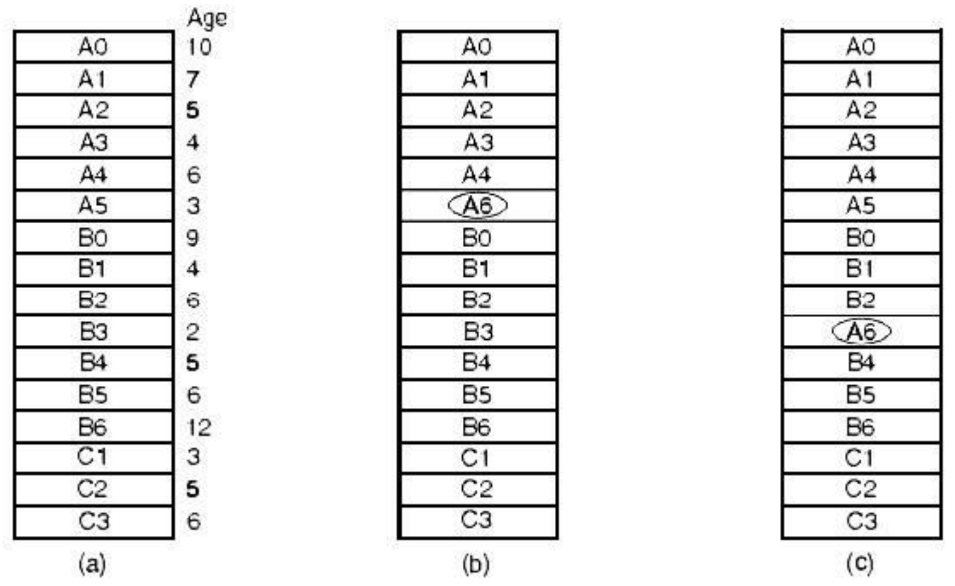

Age
A0 10
A1 7
A2 5
A3 4
A4 6
A5 3
B0 9
B1 4
B2 6
B3 2
B4 5
B5 6
B6 12
C1 3
C2 5
C3 6
(a)
A0 A1 A2 A3 A4 A6 B0 B1 B2 B3 B4 B5 B6 C1 C2 C3
(b)
A0 A1 A2 A3 A4 A5 B0 B1 B2 A6 B4 B5 B6 C1 C2 C3
(c)

Figure 1. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement

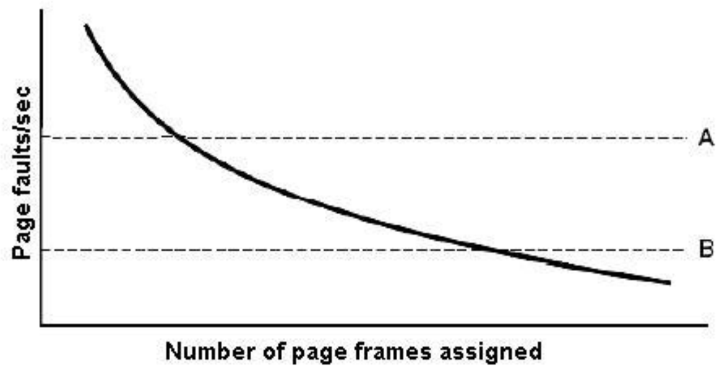


Figure 2. Page fault rate as a function of the number of page frames assigned

- Measuring the page fault rate is straightforward: just count the number of faults per second, probably taking a running mean over past seconds as well. One easy way to do this is to add the number of page faults during the immediately preceding second to the current running mean and divide by two. The dashed line marked A corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked B corresponds to a page fault rate so low that we can assume the process has too much memory. In this case page frames may be taken away from it. Therefore, PFF tries to keep the paging rate for each process within acceptable bounds.

Load Control

- Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can come about that the system thrashes. Actually, whenever the combined working sets of all processes exceed the capacity of memory, thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory. In this case there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes.

13. What is mean by I-Space and D-Space?

Separate Instruction and Data Spaces

- The majority of computers have a single address space that holds both programs and data, as shown in Figure 1(a). If this address space is large enough, everything works fine. On the other hand, it is often too small, forcing programmers to stand on their heads to fit everything into the address space.

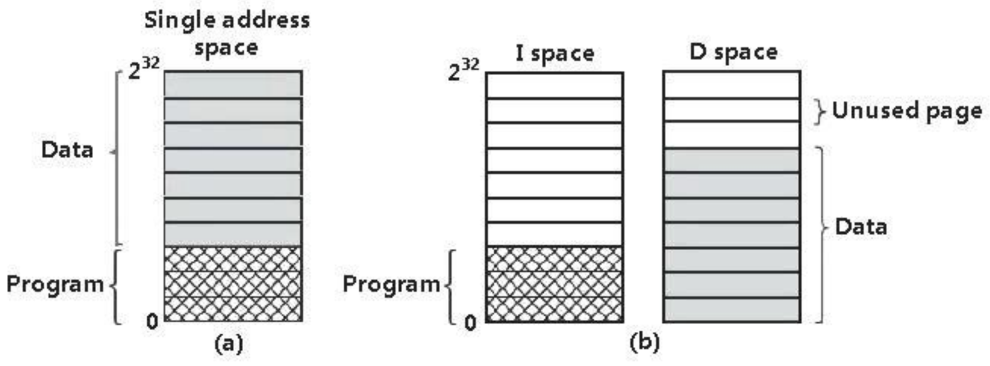


Figure 1. (a) One address space. (b) Separate I and D spaces.

- With this design, both address spaces can be paged, independently from one another in a computer. Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table. Likewise, references to data must go through the D-space page table. Other than this distinction, having separate I- and D-spaces does not introduce any special complications and it does double the available address space.

Shared Pages

- Sharing is another design issue. In a large multiprogramming system, it is common for various users to be running the same program at the same time. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. Particularly, pages that are read-only, such as program text, can be shared, but data pages cannot.
- If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their I-space but different page tables for their D-spaces. Normally in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Figure 2. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more difficult.

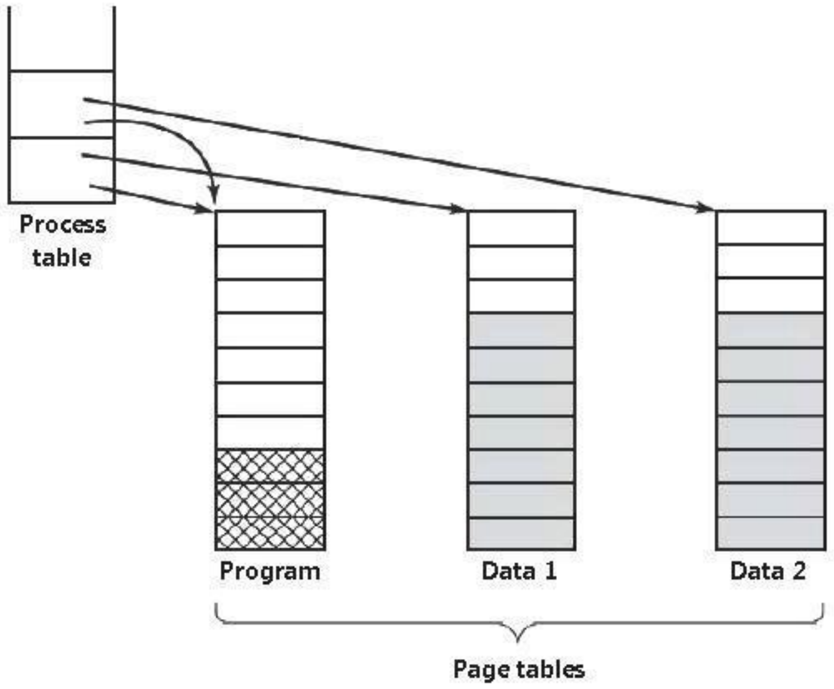


Figure 2. Two processes sharing the same program sharing its page table.

- When two or more processes share some code, a problem takes place with the shared pages. Assume that processes A and B are both running the editor and sharing its pages. If the scheduler decides to remove A from memory, removing all its pages and filling the empty page frames with some other program will cause B to generate a large number of page faults to bring them back in again.

15. Explain cleaning policy.

- Paging works best when there are lots of free page frames that can be claimed as page faults take place. If every page frame is full, and moreover customized, before a new page can be brought in, an old page must first be written to disk. To make sure a plentiful supply of free page frames, many paging systems have a background process, called the paging daemon, that sleeps most of the time but is awakened from time to time to inspect the state of memory. If too few page frames are free, the paging daemon begins selecting pages to expel using some page replacement algorithm. If these pages have been customized since being loaded, they are written to disk.
- In any event, the preceding contents of the page are remembered. In the event one of the evicted pages is required again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames. Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is required. At the very least, the paging daemon makes sure that all the free frames are clean, so they need not be written to disk in a big hurry when they are needed.
- One method to implement this cleaning policy is with a two-handed clock. The front hand is controlled by the paging daemon. When it points to a dirty page, that page is written back to disk and the front hand is advanced. When it points to a clean page, it is just advanced. The back hand is used for page replacement, as in the standard clock algorithm. Only now, the chance of the back hand hitting a clean page is increased due to the work of the paging daemon.

<u>Virtual Memory Interface</u>

- Sharing of pages can also be used to implement a high-performance message-passing system. Usually, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

16. Write a note on implementation issues.

- Implementers of virtual memory systems have to make selections among the major theoretical algorithms, such as second chance versus aging, local versus global page allocation, and demand paging versus prepaging. But they also have to be aware of a number of practical implementation issues as well. In this section we will consider a few of the common problems and some solutions.

Operating System Involvement with Paging

- There are four times when the operating system has paging-related work to do: process creation time, process execution time, page fault time, and process termination time. We will now briefly study each of these to see what has to be done.
- When a new process is created in a paging system, the operating system has to determine how large the program and data will be (initially) and create a page table for them. Space has to be allocated in memory for the page table and it has to be initialized. The page table need not be resident when the process is swapped out but has to be in memory when the process is running.
- When a process is scheduled for implementation, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process. The new process page table has to be made current, usually by copying it or a pointer to it to some hardware register(s).
- When a page fault takes place, the operating system has to read out hardware registers to find out which virtual address caused the fault. From this information, it must compute which page is required and locate that page on disk. It must then find an available page frame to put the new page, removing some old page if need be.
- When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can only be released when the last process using them has ended.

Instruction Backup

- When a program references a page that is not in memory, the instruction causing the fault is stopped partway through and a trap to the operating system takes place. After the operating system has fetched the page required, it must restart the instruction causing the trap. This is easier said than done.

Locking Pages in Memory

- Although we have not discussed I/O much in this section, the fact that a computer has virtual memory does not mean that I/O is absent. Virtual memory and I/O interact in subtle ways.

Backing Store

- The simplest algorithm for allocating page space on the disk is to have a special swap partition on the disk, or even better on a separate disk from the file system (to balance the I/O load). Most UNIX systems work like this. This partition does not have a normal file system on it, which removes all the overhead of converting offsets in files to block addresses. Instead, block numbers relative to the start of the partition are used throughout.

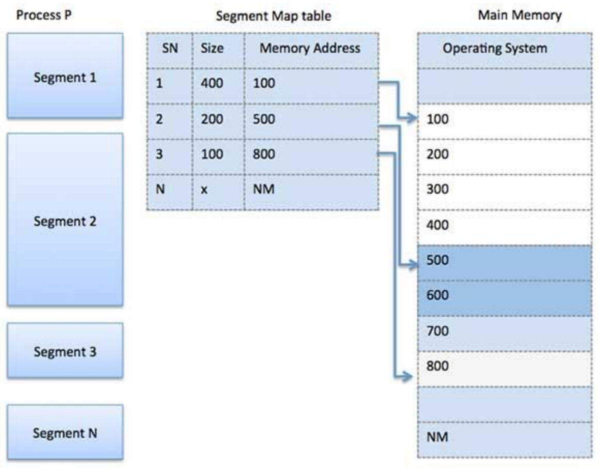
17. Explain the basic concept of segmentation.

- 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. A reference to a memory location includes a value that identifies a segment and an offset.



- The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For various problems, having two or more separate virtual address spaces may be much better than having only one. For instance, a compiler has many tables that are built up as compilation proceeds, possibly including:

1. The source text being saved for the printed listing (on batch systems).

2. The symbol table, containing the names and attributes of variables.

3. The table containing all the integer and floating-point constants used.

4. The parse tree, containing the syntactic analysis of the program.

5. The stack used for procedure calls within the compiler.

- Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be assigned contiguous chunks of virtual address space, as in Figure 1. Examine what happens if a program has a much larger than usual number of variables but a normal amount of everything else. The chunk of address space assigned for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

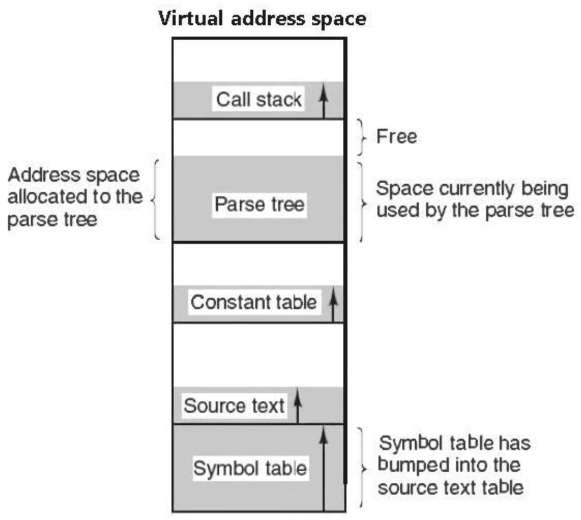

Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Figure 1. In a one-dimensional address space with growing tables, one table may bump into another.

18. Explain any one type of Segmentation with paging.

- If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages that are really required have to be around. Many significant systems have supported paged segments. In this section we will explain the first one: MULTICS. In the next one we will describe a more recent one: the Intel Pentium.

- MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 218 segments (more than 250,000), each of which could be up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it is being used) with the advantages of segmentation (ease of programming, modularity, protection, sharing).

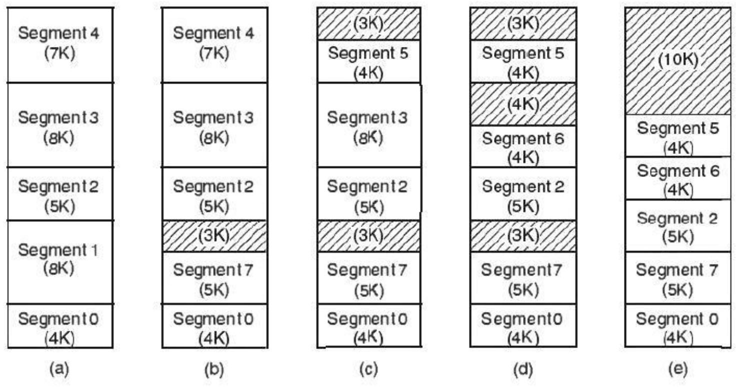


Figure 1. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

- Each MULTICS program has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory.
- Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory). An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page
- When a memory reference occurs, the following algorithm is carried out.

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

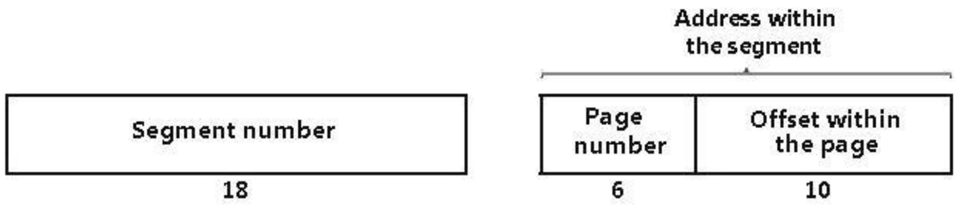


Figure 3. A 34-bit MULTICS virtual address.