Unit 5 - Memory Management

Memory accesses and memory management are a very important part of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.

The emergence of multitasking OSes adds to the complexity of memory management, because as processes are swapped in and out of the CPU so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.

Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.

Basic Hardware

With memory management, our main concerns would be faster access to data as well as being able to protect the operating system from access by user processes and, in addition, to protect user processes from one another.

Hardware support for faster access

The memory unit sees only a stream of memory addresses. From the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care.

The CPU can only access its registers and main memory. Therefore, any data stored there must first be transferred into the main memory chips before the CPU can work with it. Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.

Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the CPU if it were not for an intermediary fast memory ***cache*** built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.

Hardware support for protection

User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures 8.1 and 8.2 below.

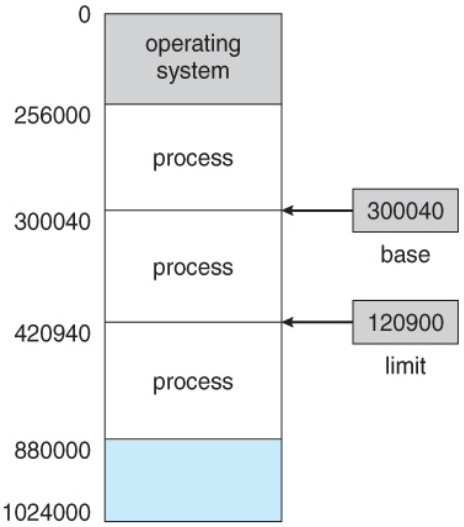


Figure 8.1 - A base and a limit register define a logical address space

Protection is provided by using two registers.

i. **Base** Register-holds the smallest legal physical memory address

ii. **Limit** Register-specifies the size of the range

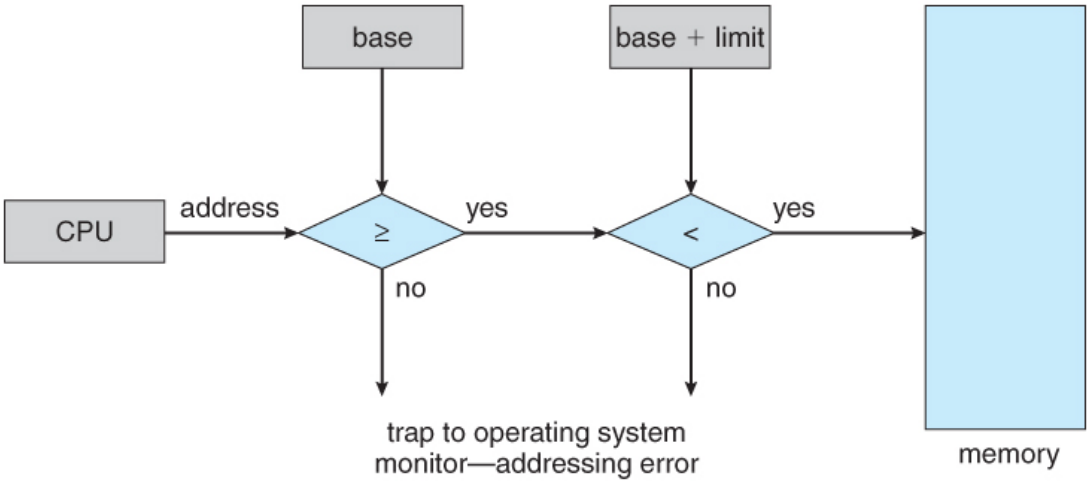


Figure 8.2 - Hardware address protection with base and limit registers

Every memory access made by a user process is ***checked against these two registers***. A program executing in user mode ***attempt to access OS memory or other user's memory results in a trap to the OS***, which treats the attempt as a fatal error.

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The operating system, executing in kernel mode, is given unrestricted access to both operating system memory and users' memory. This provision allows the operating system to swap user code and data in and out of the memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.

Address Binding

1. Describe execution time binding [4]
2. What do you mean by address binding? Explain with necessary steps the binding of instructions and data to memory addresses [8]

User programs typically refer to memory addresses with symbolic names and these symbolic names must be mapped or ***bound*** to physical memory addresses.

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. **The process of associating program instructions and data to physical memory addresses is called *address binding*, or *relocation*.**

Memory Management Unit is a special hardware that performs address binding

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

- Compile time
- Load time
- Execution time

- **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load

address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.

- **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.

- **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware support for address map. (ex: base and limit registers).and is the method implemented by most modern OSes.

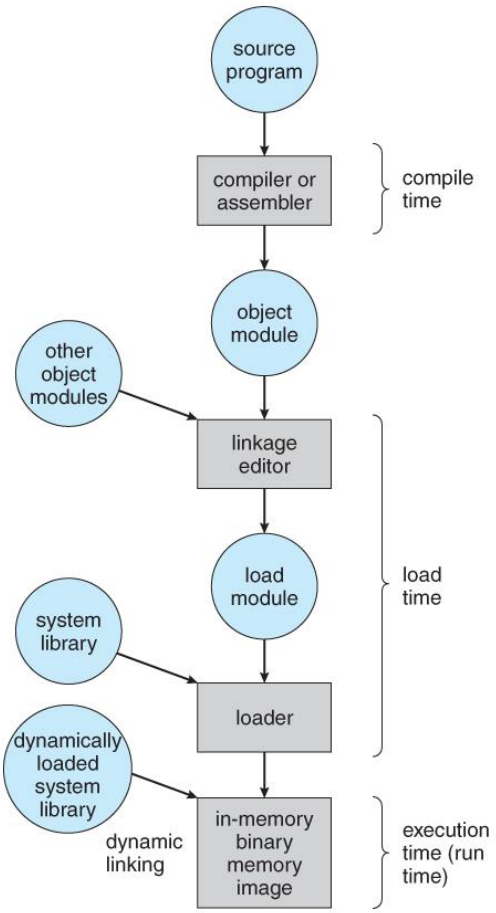


Figure 3: Multistep processing of a user program.

Logical Versus Physical Address Space

The address generated by the CPU is a **logical address**, whereas the address actually seen by the memory hardware is a **physical address**.

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

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. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (MMU).

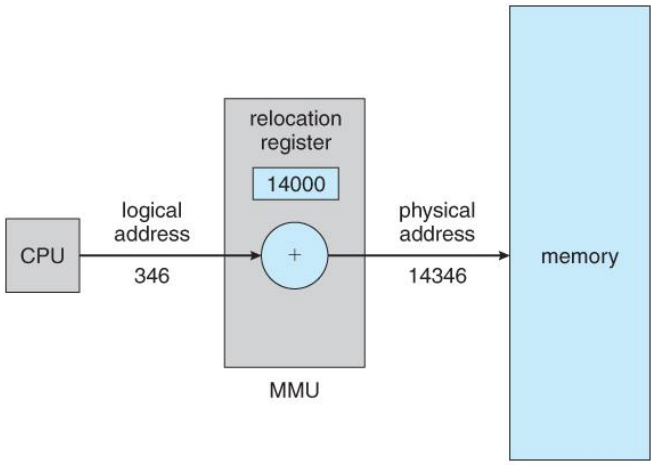


Figure4: Dynamic relocation using a relocation register.

The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.

The value in the relocation or base register is added to every address generated by a user process at the time it is sent to memory. For ex, if the base is at 14000, then an attempt to address location 0 is dynamically relocated to location 14000; an access to location 300 is mapped to 14300. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Dynamic Loading

Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that *unused routines need never be loaded*, *reducing total memory usage* and *generating faster program start up times*. The ***downside*** is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

- Useful when large amounts of code are needed to handle infrequently occurring cases (ex: error handling code, rarely used features, etc)

- No special support from the operating system is required; implemented by application programs or libraries

Dynamic Linking and Shared Libraries

Static linking

- Codes (including libraries) are combined into the binary program image
- Every program that included a certain routine from a library would have to have their own copy of that routine linked into their executable code
- Consume more disk space and main memory

Dynamic linking

- With dynamic linking, only a stub is linked into the executable module, containing references to the actual library module linked in at run time. This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
- The stub contains code to locate the appropriate memory-resident library routine
- ***The first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access***

Another advantage of dynamic linking is that if code section of the library routines is reentrant, (read-only, execute-only), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it

An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates. *When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes*. However ***if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system***. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.

Swapping

1. Let the user process size be 1MB, and the data transfer rate from memory to disk be 5MB per second. Determine the time required to transfer the program data transfer from disk to memory. If average latency is 8ms, determine the total swap time [4]
2. Determine the swap time for a user process of size 4MB with a disk transfer rate of 10 MB per second and latency time is 12 msec [3]
3. What is swapping? Does this increase the Operating systems overhead? Justify.

A process must be loaded into memory in order to execute. If there is *not enough memory* available to keep all running processes in memory at the same time, then *some processes who are not currently using the CPU may have their memory swapped out* to a fast local disk called the ***backing store.***

Swapping is evident in algorithms such as Round Robin and Priority scheduling, where processes are swapped in and out of memory by the memory manager at the end of a time quantum or to accommodate space for a higher priority process.

===== Will the process be swapped back to the same location? =======

If **compile-time** or **load-time** address binding is used, then *processes must be swapped back into the same memory* location from which they were swapped out. If **execution time binding** is used, then the *processes can be swapped back into any available location.*

===== Where are the swapped out processes stored? =======

Swapping requires a backing store, usually implemented as a fast disk. The swap space should be large enough to accommodate copies of all processes and must provide direct access to them. Whenever CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks if the next process is in the memory. If it is not in memory and there is no free space to load the process into memory, dispatcher swaps out a process from the memory and swaps in the desired process.

===== Swapping. An overhead =======

Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.

The major part of swap time is transfer rate. The ***total transfer time is directly proportional to the amount of memory swapped.***

To reduce swapping transfer overhead, it is desired to *transfer as little information* as possible, which requires that the system know how much memory a process is using, as opposed to how much it might use. Programmers can help with this by freeing up dynamic memory that they are no longer using.

Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again.

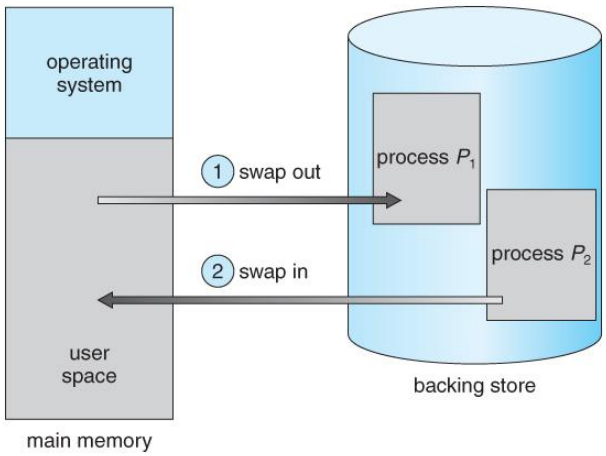


Figure: Swapping of two processes using a disk as a backing store.

Contiguous Memory Allocation

One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.

Memory Mapping and Protection

The system shown in Figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

In the contiguous memory allocation, each process is contained in a single contiguous section of memory. A relocation register (contains starting address for a process) and limit register (contains the range of physical addresses)marks the address space assigned for that process; The MMU maps the logical address dynamically by adding the value in the relocation register. If the generated address is a valid address (i.e. in between the base and limit range), access to that address in the memory is permitted.

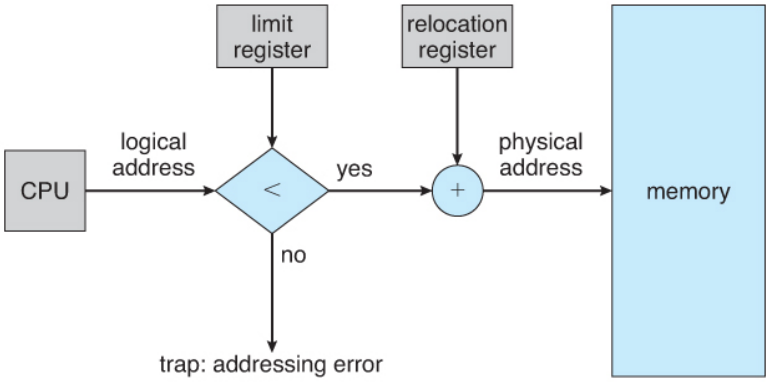


Figure : Hardware support for relocation and limit registers.

- 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.

Contiguous Memory Allocation

1. Given memory partitions of 270, 150, 600 (KB) in order, how would each of first-fit, best-fit, worst-fit algorithms place the processes of 100, 212 and 270 KB (in order) [6]
2. Given memory partitions of 100, 500, 200, 300, 600 (KB) in order, how would each of first-fit, best-fit, worst-fit algorithms place the processes of 212, 417, 112, 426 (KB) in order. Which algorithm makes most efficient usage of memory [6]
3. What is dynamic storage allocation? Explain the commonly used strategies for dynamic storage allocation [12]

One method of allocating contiguous memory is to *divide all available memory into equal sized partitions* (fixed sized partition), and to assign each process to their own partition. This **restricts** both the ***number of simultaneous processes*** and the ***maximum size of each process***, and is no longer used.

An alternate approach is to *keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size* whenever a process needs to be loaded into memory.

Contiguous memory allocation is a particular application of the **Dynamic Storage Allocation** problem, which is concerned with how to satisfy a request for memory 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* to hold the process.
  - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process.
  - Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
- **Best fit**.
  - Allocate the *smallest hole that is big enough*.
  - Maintaining a sorted *free list* can speed up the process of finding the right hole.
  - This strategy produces the *smallest leftover hole* and will therefore be wasted
- **Worst fit**.
  - *Allocate the largest hole*.
  - A *sorted free list* will speed up the task of searching the largest hole
  - This strategy produces the *largest leftover hole*, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests..

Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

Fragmentation

1. Differentiate between
   a. Internal and External fragmentation
   b. Paging and Segmentation
   c. Logical and physical address space
   d. First fit and best fit algorithms
2. Explain internal and external fragmentation with a neat diagram [5]
3. Explain internal and external fragmentation with an example [6]
4. Explain the external and internal fragmentation occurred in contiguous memory allocation scheme [4]
5. What is fragmentation? Differentiate internal and external fragmentation. How are they overcome? [5]

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometime that there are a lot of small free slots in memory which are unusable because they are too small to accommodate any process. This problem of the memory being broken down into small unusable pieces is known as ***Fragmentation***.

There are 2 types of memory fragmentation:

- Internal Fragmentation
- External fragmentation

External Fragmentation

External fragmentation exists when *there is enough total memory* space to satisfy a request, *but the available spaces are not contiguous*. Storage is fragmented into a ***large number of small holes***

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.

Internal Fragmentation

Internal fragmentation ***occurs in fixed partition memory allocation***.

This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size..

In internal fragmentation, *memory is internal to a partition but is not being used*.

Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.

==========Methods to overcome fragmentation==========

Internal Fragmentation:

The only method to overcome internal fragmentation is to avoid fixed size partitions and create blocks that matches the exact size of the process.

External Fragmentation:

i) If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses. Clearly, this approach is not applicable if processes are not relocatable.

ii) Another solution is to *allow logical address space of the processes to be non-contiguous*, thus allowing a process to be allocated physical memory wherever the latter is available

Two complementary techniques achieve this solution:

- **paging**
- **segmentation**

Paging

1. What is paging? Explain [7]
2. Explain the fragmentation problem and the paging scheme to avoid this problem[8]
3. Describe the paged memory allocation scheme [8]
4. Explain the concept of paging with an example. In paging, what is the average case and worst case internal fragmentation per process? [10]
5. With the help of a neat diagram, explain the hardware support required for paging. What is the effective access time in a paged memory [8]
6. What is paging and swapping [4]
7. Explain the paging scheme wrt storage management. With a neat diagram, explain the paging hardware with TLB [8]
8. What is locality of reference? Differentiate paging and segmentation. [5]
9. With a supporting paging hardware, explain in detail the concept of paging with an example for 32 byte memory with 4 byte pages, with a process being 16 bytes. How many bits are reserved for page number and page offset in the logical address? Suppose the logical address is 5, calculate the corresponding physical address, after populating memory and page table [10]
10. Consider a system with 32 bit logical address space. If the page size is 4 KB and each entry in the page table is 2 bytes long, how much memory is required for the page table? What is the size of the main memory [6]
11. Consider a logical address space of 8 pages of 1024 bytes each mapped onto a physical memory 32 frame. How may bits are there in logical and physical address [4]

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

Basic Method

The basic idea behind paging is to *divide physical memory into* a number of *equal sized blocks called frames*, and to divide a programs *logical memory space into blocks of the same size called pages*.

Any page (from any process) can be placed into any available frame.

A **page table** maintains the mapping of logical pages to physical frames, therefore allowing different parts of a process (pages) to be shuffled in non-contiguous parts of memory.

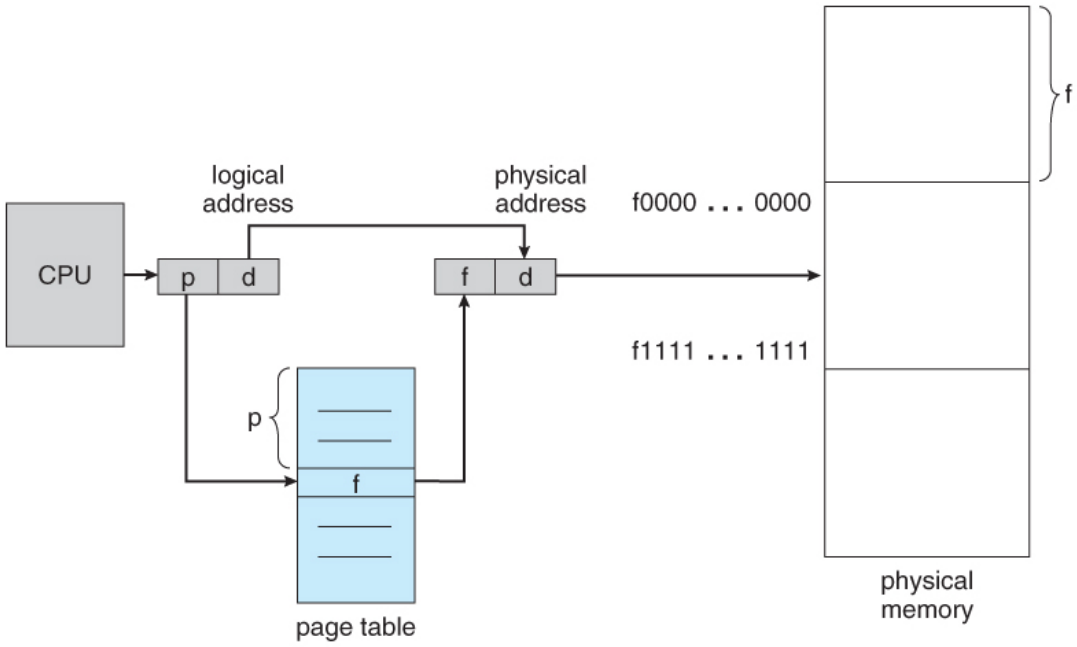


Figure: Paging hardware.

- Every address generated by the CPU is divided into two parts: a **page number** (*p*) and a **page offset** (*d*).
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

- The paging model of memory is shown in below Fig.

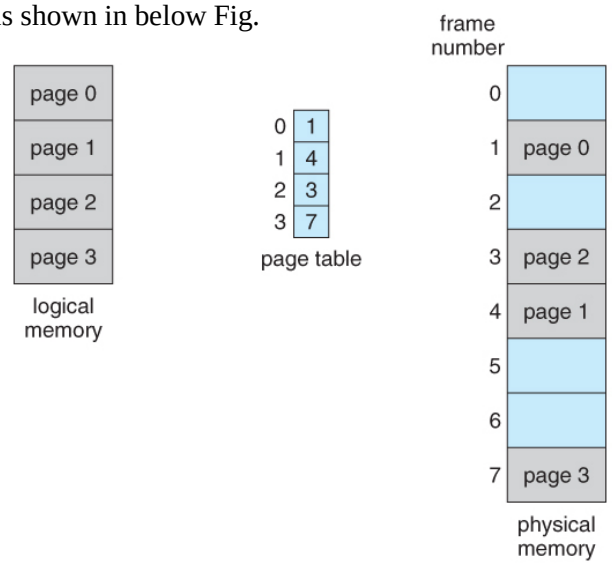


Figure: Paging model of logical and physical memory.

Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of 2, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.

12. Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. What is dynamic storage allocation? Explain the commonly used strategies for dynamic storage allocation [12]

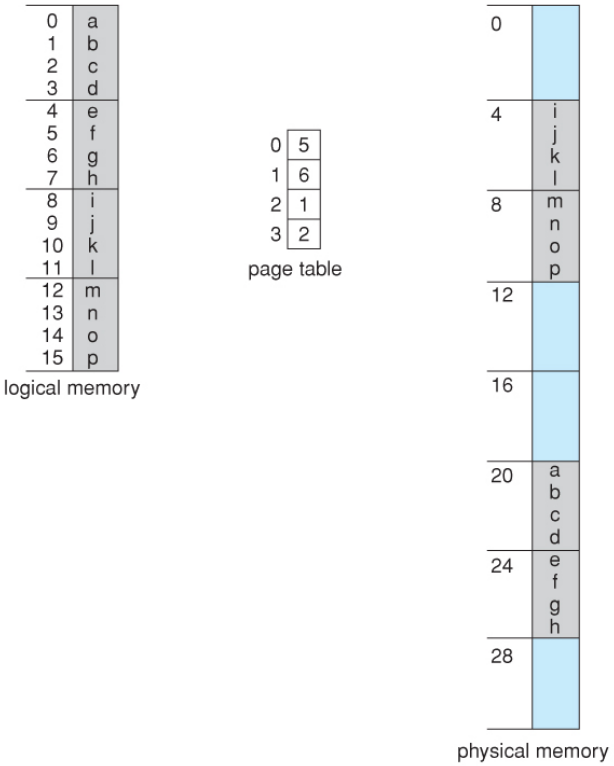


Figure: Paging example for a 32-byte memory with 4-byte pages

==========Paging and Fragmentation==========

- Using paging is *similar to using a table of base (or relocation) registers*, one for each frame of memory.
- When we use a paging scheme, we have *no external fragmentation:* Any free frame can be allocated to a process that needs it.
- However, we may ***have some internal fragmentation.*** Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting ***on the average half a page of memory per process***. In the worst case a process might need n full frames and 1 additional frame to store one byte.

==========Memory Allocation with paging==========

- When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame.
- The OS maintains a ***frame table*** containing details of total frames, free frames, occupied frames, etc.
- If sufficient frames are available in the free frame list, they are allocated to this arriving process and the page table of the process is updated accordingly

Figure 10: Free frames (a) before allocation and (b) after allocation.

- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.

==========Protection with Paging==========

- The logical addresses are translated into physical addresses by the address-translation hardware. This *mapping is hidden from the user* and is controlled by the OS.
- Processes are *blocked from accessing anyone else's memory* because all of their *memory requests are mapped through their page table*. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system *must keep track of each individual process's page table*, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU.

Paging Hardware Support

Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.

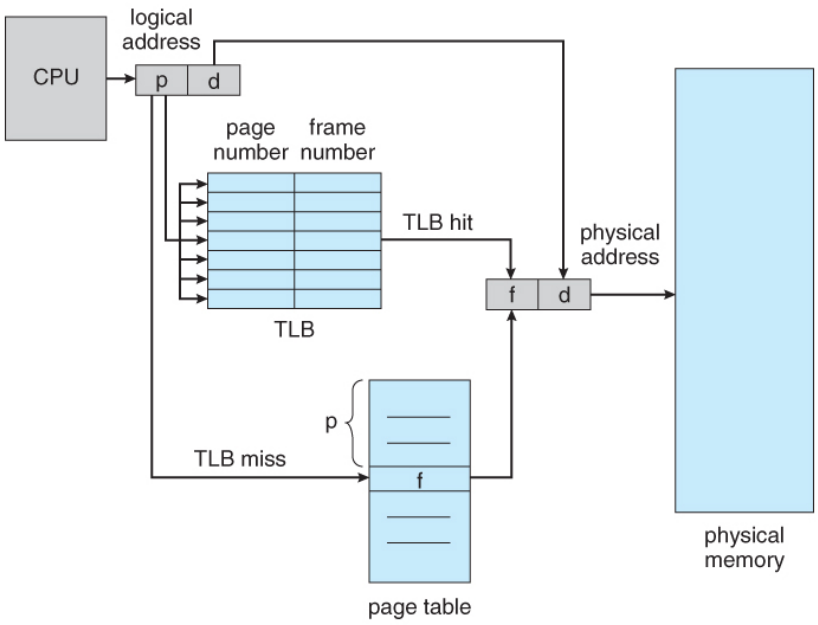
- In the simplest case, the page table is implemented ***as a set of dedicated registers***. If the page table is too large, then use of registers is not satisfactory.
- If the page table is too large, then it is ***kept in the main memory***, and a page table register (PTBR) points to the page table.
    - Process switching is fast, because only the single register needs to be changed.
    - However **memory access just got half as fast** [***memory access reduced by a factor of 2***], because ***every memory access now requires two memory accesses*** - One to fetch the frame number from memory and then another one to access the desired memory location.
    - The SOLUTION to this problem is to use a very special high-speed memory device called the **translation look-aside buffer, TLB.**

Translation Look-aside Buffer [TLB]

1. Describe TLB. Explain how it improves memory access time with an example [5]
2. Mention the problem with a simple paging scheme. How is TLB used to solve this problem? Explain with supporting hardware diagram and with an example [8]
3. Why are TLB's important? In a simple paging scheme, what information is stored in TLB? Explain.[8]
4. Assume we have a paged memory scheme with associative registers[TLB] to hold the most active page entries. If the page table is normally held in memory, and memory access time is one microsecond, what would be the effective access time if 85% of all memory references find their entries in associative registers. Assume that associative register access time is 0 [4]
5. Consider a paging system with a TLB and page table stored in memory. If hit ratio of TLB is 80% and it takes 20 nanoseconds to search TLB and 100 nanoseconds to search memory, find the effective memory access time [4]
6. For a memory access system with a TLB, we have TLB hit ratio of 0.9, memory access time of 110 nanoseconds; TLB search time of 20 nano seconds. Calculate effective memory access time [5]

TLB is a high speed associative memory. It can search an entire table for a key value ***in parallel***, and if it is found anywhere in the table, then the corresponding lookup value is returned.

Each entry in the TLB consists of two parts: **a key and a value**.



The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.

==========Address translation with TLB=========

The TLB contains only a few of the page table entries. When a logical address is generated by the CPU its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. If the page number is not in the TLB known as **TLB miss**, a memory reference to the page table must be made. When the frame number is obtained, it can be used to access memory.

Some TLB's allow entries to be wired down, that is they cannot be removed from the TLB. TLB entries for kernel code are wired down

Some TLB's store address space identifiers (ASIDS) in each TLB entry. An ASID uniquely identifies each process and is used to provide address space protection for that process. An ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be flushed or erased to ensure that the next executing process does not use the wrong translation information.

==========Effective Access Time==========

- The percentage of time that the desired information is found in the TLB is termed the **hit ratio**. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 20ns to search the TLB and 100ns to access the memory, then a mapped-memory access takes 120ns (20+100=120) when the page number is in the TLB.
- If we fail to find the page number in the TLB (20ns), then we must first access memory for the page table and frame number (100ns) and then access the desired byte in memory (100ns)
- Hence the total memory access time is 220ns (20+100+100=220).

→ To find the **effective memory access time**, each case is weighted by its probability.

→ effective access time= 0.80 ×120 +0.20 × 220

= 140ns.

Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A ***bit or bits can be added to the page table*** to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then *each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.*
- ***Valid / invalid bits*** can be added to "mask off" entries in the page table that are ***not in use*** by the current process. When this bit is set to "valid", the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to "invalid", the page is not in the process's logical address space.
- Some systems provide hardware in the form of a **page table length register** (PTLR) to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the OS.

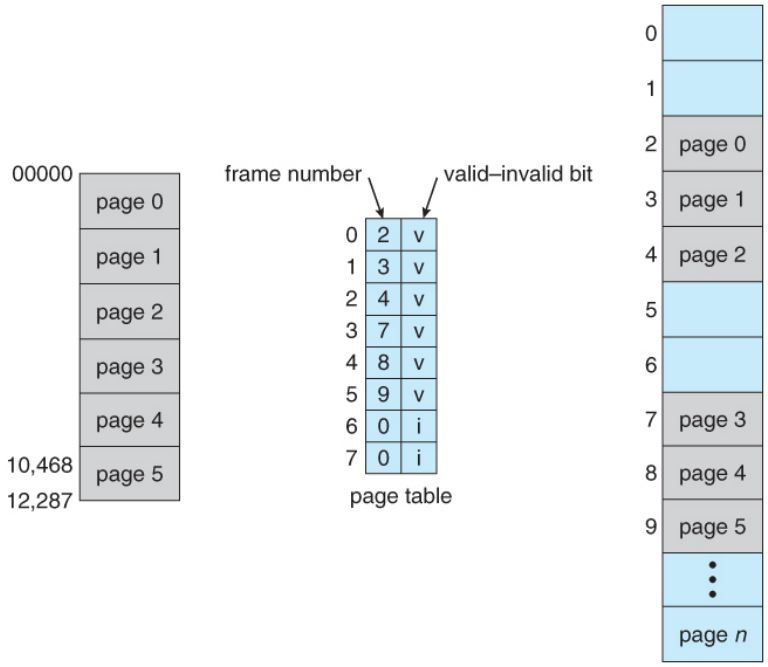


Figure 12: Valid (v) or invalid (i) bit in a page table.

Shared Pages

- An advantage of paging is the possibility of sharing common code.

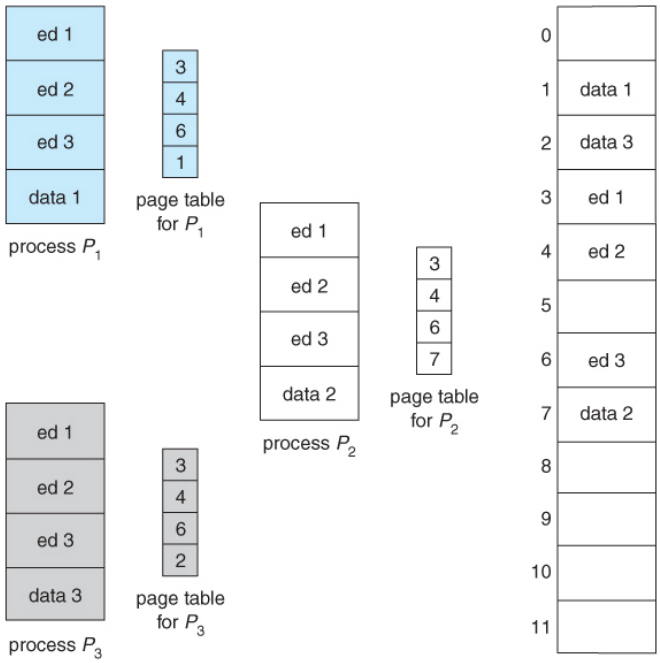


Figure: Sharing of code in a paging environment.

- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- If the code is re-entrant code (or read-only code), it can be shared (to be shareable, the code must be re-entrant).
- Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB.

Structure of the Page Table

- Common techniques for structuring page table

Hierarchical Paging

Most modern computer systems support a large logical address space. In such an environment, ***the page table itself becomes excessively large***. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces and store it as separate pages in memory. This also requires the need to have a master page table to keeps track of the starting addresses of the pages containing the parts of the page table

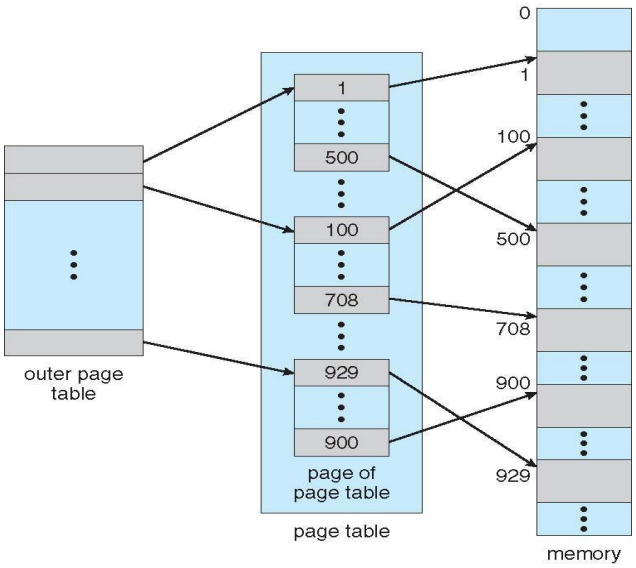
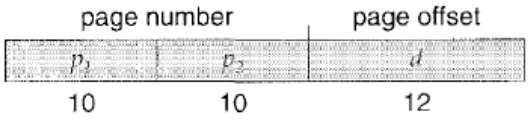


Figure 15: A two-level page-table scheme

Consider an example of a 32-bit machine with a page size of 4 KB (2^{12}). The logical address generated by the CPU is 32 bits. The lower 12 bits would represent the page offset and remaining 20 bits can be used as index to page table. Because we page the page table, the page number is further divided into a l0-bit page number and a 10-bit page offset.



The first part P1 identifies an index to the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)

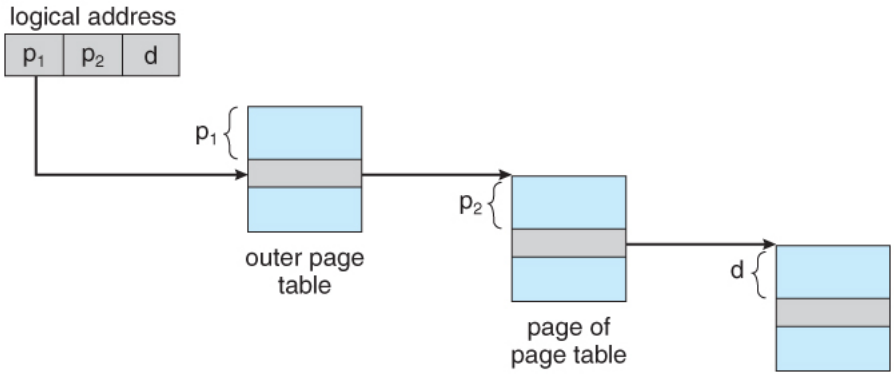


Figure: Address translation for a two-level 32-bit paging architecture

Because address translation works from the outer page table inward, this scheme is also known as a **forward mapped page table.**

Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table,** with the hash value being the virtual page number. The virtual page number p is given as input to a hash function that generates a number which will be given as input to the page table. More than one virtual page numbers may hash to the same value. Hence, each entry in the hash table contains a linked list of elements that hash to the same location. Each element consists of three fields:

- The virtual page number
- The value of the mapped page frame
- Pointer to the next element in the linked list

It works as follows –

The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field1 in the first element in the linked list. If there is a match, the corresponding page frame (field2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

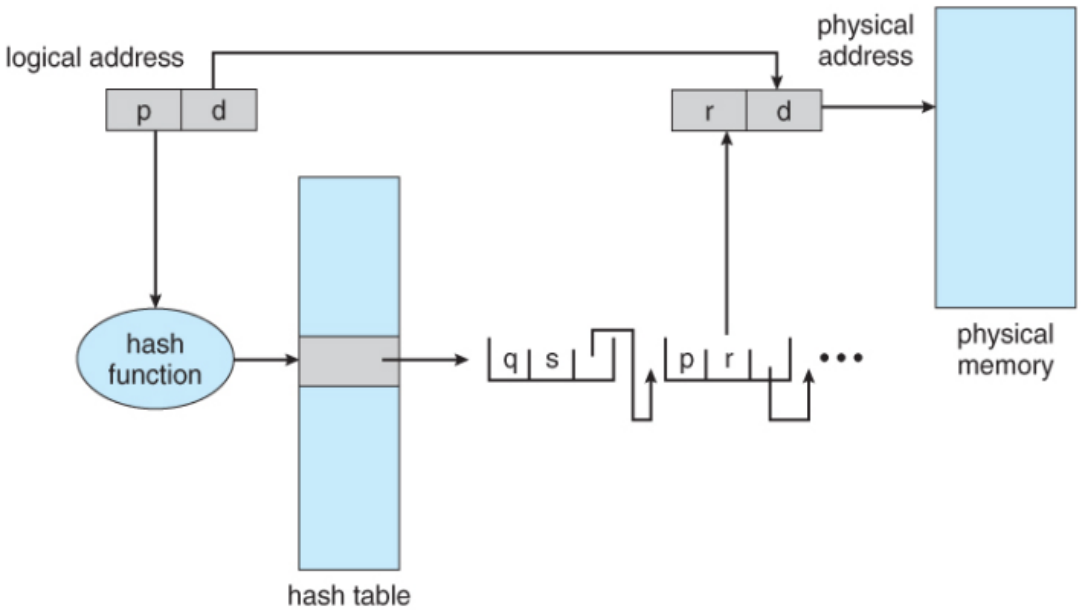


Figure: Hashed page table

A variation of this scheme for 64 bit address spaces uses **clustered page tables** which are similar to hashed page tables except that each entry in the hash table refers to several pages rather than a single page. Therefore, a single page table entry can store the mappings for multiple physical page frames. Clustered page tables are useful for **sparse** address spaces where memory references are non contiguous and scattered throughout the address space.

Inverted Page Tables

Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (i.e. there is one entry per frame instead of one entry per page.)

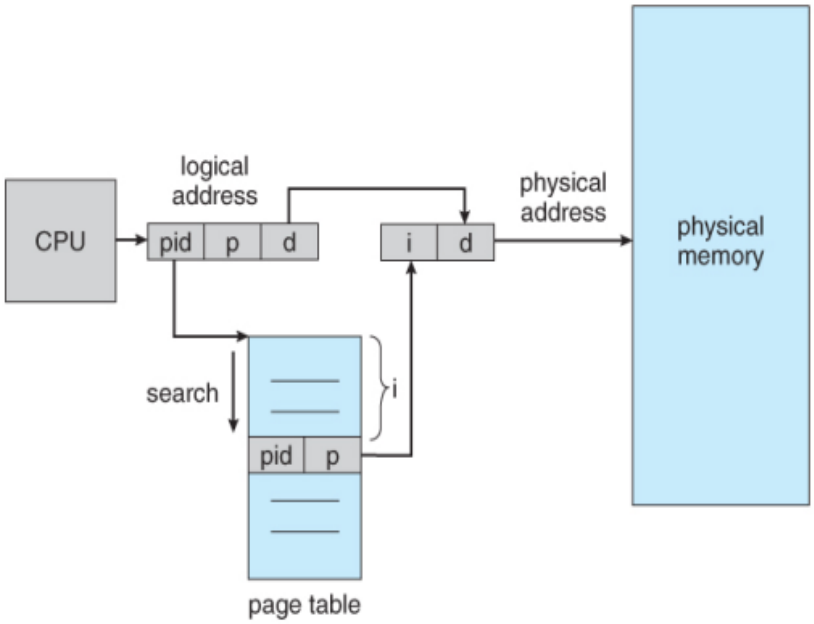


Figure 18: Inverted page table

Each entry consists of ***the virtual address*** *of the page* stored in that real memory location; with information about the ***process that owns that page***.

Thus, only one page table is in the system and it has only entry for each page of physical memory. Inverted page table often require that an address space identifier be stored on each entry of the page table since the table usually contains several different address spaces mapping physical memory. Storing the address space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

Access to an inverted page table may require to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speed up the search process.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is implemented as multiple virtual addresses that are mapped to one physical address. This method cannot be used with inverted page tables because there is only one virtual page entry for every physical page, one physical page cannot have two or more shared virtual addresses. A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address.

Segmentation

1. Explain segmented memory management [6]
2. On a system using simple segmentation, compute the physical address for each logical address. Logical address is given in the following segment table. If address generated is segmentation fault, indicate it as "segment fault".

Segment	Base	Limit
0	330	124
1	876	211
2	111	99
3	498	302

i) 0,9 ii) 2,78 iii)3, 222 iv)0,111

3. Explain the basic concepts of segmentation wrt memory management [6]

Most users (programmers) do not think of their programs as existing in one continuous linear address space. Rather they tend to think of their memory in multiple segments, each dedicated to a particular use, such as code, data, the stack, the heap, etc.

Memory segmentation supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.

For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure:

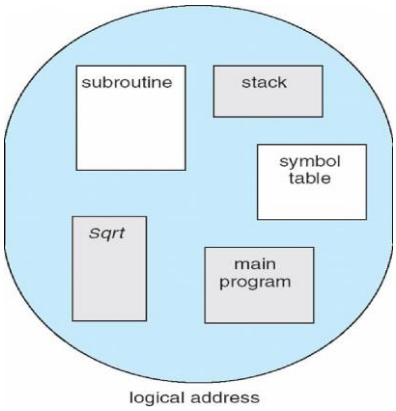


Figure : User's view of a program.

Segmentation is a memory management scheme that supports the user view of memory. A logical address space is a collection of segments. ***Each segment has a name and a length***. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities – a segment name and an offset.

Segments are numbered and are referred to by a segment number. ***Thus a logical address consists of a two tuple: <segment – number, offset>***

The user program is compiled and the compiler automatically constructs segments reflecting the input program. Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Hardware

Even though, the user can now refer to objects in the program by a two dimensional address, the actual physical memory is a one dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user defined addresses into one dimensional physical address. This mapping is affected by a segment table.

Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory whereas the segment limit specifies the length of the segment.

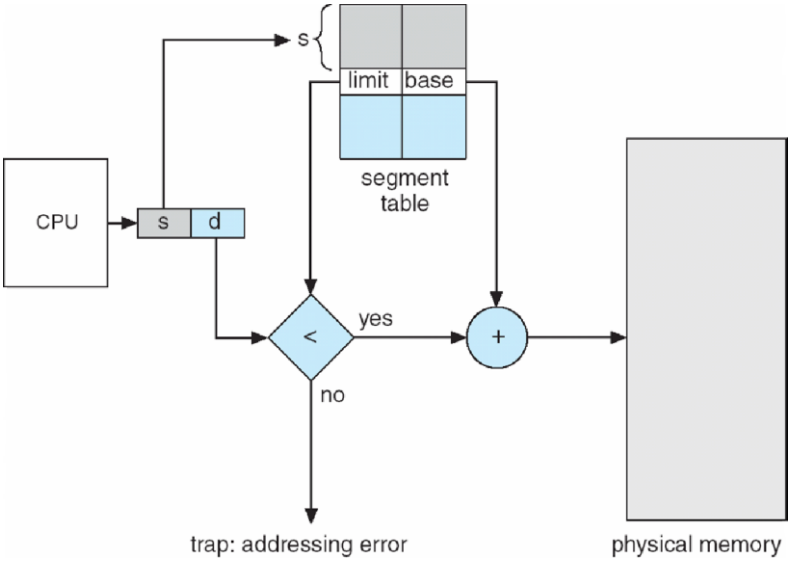


Figure: Segmentation hardware.

A logical address consists of two parts: a segment number s, and an offset into that segment, d. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, a trap is returned. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base – limit register pairs.

Module 3 DEADLOCKS

Processes need resources to complete execution. In a multi-programmed environment, many processes may be competing for finite number of resources.

A process requests for resources for execution. If resources are not available at that time, the process enters into waiting state until the resource becomes available. Sometimes a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

Definition: A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.

Example : Consider 2 processes: 1 & 2 both of which need a printer and tape drive to begin execution.

Process-1 requests the printer, gets it
Process-2 requests the tape unit, gets it
Process-1 requests the tape unit, waits
Process-2 requests the printer, waits

} Process-1 and Process-2 are deadlocked!

System Model

1. With the help of a system model, explain deadlock and explain the necessary conditions for deadlocks to occur [6]

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types.

If a system has two CPUs, then the resource type *CPU* has two instances.

If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**: The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release:** The process releases the resource.

A **system table** *records whether each resource is free or allocated*; for each resource that is allocated, *the table also records the process to which it is allocated*. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Deadlock Characterization

1. What is deadlock? Explain the necessary conditions for its occurrence. [10M]
2. What are the necessary conditions for deadlock to occur? [6M]
3. Define deadlock. List and elaborate the necessary condition for deadlock to occur.[10M]
4. What is deadlock? Explain the various methods for dealing with deadlock problem [12M]

A DEADLOCK OCCURS **IF AND ONLY IF** THE FOLLOWING FOUR CONDITIONS **HOLD SIMULTANEOUSLY** IN A SYSTEM:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode (that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released).
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set {P0, P1, ..., Pn } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *Pn*, and Pn is waiting for a resource held by *Po*.

Resource-Allocation Graph

1. What is a deadlock? Explain how resource allocation graph can be used to check for deadlock in a system. [8M]
2. Explain the resource allocation graph [6m]
3. Explain with an example how resource allocation graph is used to describe the deadlock [6M]
4. Why is deadlock more critical than starvation in a multiprogrammed environment? Describe resource allocation graph (i) with deadlock (ii) without deadlock

Draw a resource-allocation graph for the following situation and check if the system is in deadlock and explain.

Process P_1 is (holding) using resource R_2 and waiting for resource R_1
P_2 is using R_1 and waiting for R_3, R_4 and R_5
P_3 is using R_4 and waiting for R_5
P_4 is using R_5 and waiting for R_2
P_5 is using R_3 **(08 Marks)**

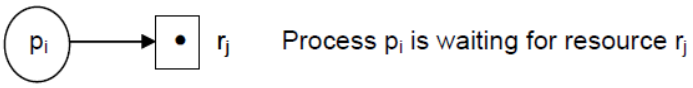
5.
6. Deadlock exists if cycle exists. Yes or No. Justify your answer with suitable example [8]
7. Explain how resource allocation graph is used to describe deadlocks [5]
8. Given 3 processes A,B and C, three resources x,y and z and following events, i) A requests x ii) A requests y iii) B requests y iv) B requests z v) C requests z vi) C requests x vii) C requests y. Assume that requested resources should always be allocated to the request process if it is available. Draw the resource allocation graph for the sequences. And also mention whether it is a deadlock? If it is, how to recover the deadlock. [8M]

9. What is resource allocation graph? Explain how RAG is useful in describing a deadly embrace by considering your own example [8]

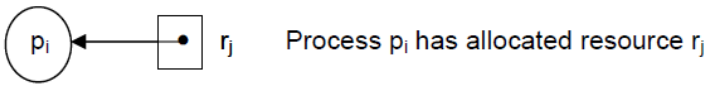
Resource-Allocation Graph [RAG]

Deadlocks can be *described more precisely in terms of a directed graph* called system resource-allocation graph. The RAG consists of a set of vertices P={P1,P2 ,…,P n} of processes and R={R1,R2,…,Rm} of resources.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ **(***called as* ***request edge*****).**



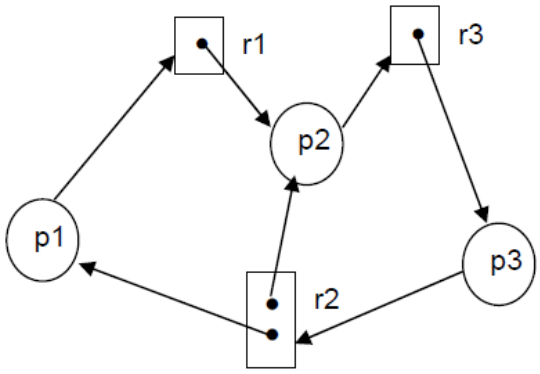
A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ *(called an* ***assignment edge****).*



Pictorially, we represent ***each process Pi as a circle*** and each resource type ***Rj as a rectangle***. Since resource type *Rj may have more than one instance*, we represent ***each such instance as a dot within the rectangle***. Note that a request edge points to only the rectangle *Rj*, whereas an assignment edge must also designate one of the dots in the rectangle.

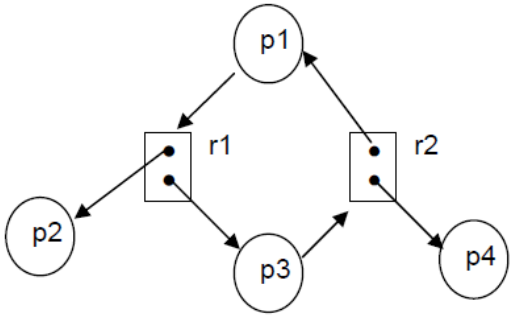
When process *Pi* requests an instance of resource type *Rj*, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

- If the resource allocation graph contains no cycles then there is no deadlock in the system at that instance.
- If the resource allocation graph contains a cycle then a deadlock **may exist.**
- If there is a cycle, and the cycle involves only resources which have a single instance, then a deadlock **has occurred**.



Resource-allocation graph with a deadlock.

There are three cycles, so a deadlock may exist (p1, p2 and p3 are deadlocked).



Resource-allocation graph with a cycle but no deadlock.

There is a cycle, however there is no deadlock. If p4 releases r2, r2 may be allocated to p3, which breaks the cycle.

Methods for Handling Deadlocks

1. Briefly describe the methods for handling deadlocks [4]

Deadlocks can be dealt in one of the three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock Prevention

Explain the methods for deadlock prevention [8M]

Deadlock Prevention provides a set of methods for ***ensuring that at least one of the necessary conditions*** (Mutual exclusion, Hold and wait, No preemption and Circular wait) ***cannot hold***. These methods prevent deadlocks by constraining how requests for resources can be made.

a. **Mutual exclusion**:
   - Sharable resources (ex: read only file) do not require mutually exclusive access and thus cannot be involved in a deadlock. A process never needs to wait for a sharable resource. Thus a deadlock can be prevented **by keeping most resources in shareable mode**.
   - However, We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources (ex: printers) are naturally non-sharable.

b. **Hold and wait**: To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, **whenever a process requests a resource, it does not hold any other resources**.

➔ ***One protocol to ensure that hold-and-wait condition never occurs requires each process to request and get all of its resources before it begins execution.***

➔ ***An alternative protocol allows a process to request resources only when it has none.*** A process may request some resources and use them. Before it can use any additional resources, it must release all the resources that it is currently allocated.

Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

→ Both protocols cause low resource utilization and starvation.

Many resources are allocated but most of them are unused for a long period of time. A process that requests several commonly used resources causes many others to wait indefinitely.

c. **No Preemption:** The third necessary condition for deadlocks is that there be no pre-emption of resources that have already been allocated.

To ensure that this condition does not hold, we can use the following protocol.

- ❖ If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), ***then all resources the process is currently holding are preempted***. The preempted resources are added to the list of resources for which the process is waiting.
- ❖ Alternatively,
  - ➢ if a process requests some resources, we first *check whether they are available*. If they are, we allocate them.
  - ➢ If they are not, we *check whether they are allocated to some other process that is waiting for additional resources*. If so, we ***preempt the desired resources from the waiting process and allocate them to the requesting process***.
  - ➢ If the resources are `neither available nor held by a waiting process, the requesting process must wait`. While it is waiting, some of its resources may be preempted, but only if another process requests them. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

d. **Circular Wait:** One protocol to ensure that the circular wait condition never holds is "***Impose a linear ordering of all resource types***." Then, *each process can only request resources in an increasing order of priority.*

- ➢ In other words, in order *to request resource Rj*, a process ***must first release all Ri such that i >= j***.
- ➢ One big *challenge* in this scheme is ***determining the relative ordering*** of the different resources

`Let` *R* `= { R1, R2, ... , Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F:` *R* `–›N, where` *N* `is the set of natural numbers. For example, if the set of resource`

types *R* includes tape drives, disk drives, and printers, then the function *F* might be defined as follows:

F (tape drive) = 1

F (disk drive) = 5

F (printer) = 12

We can now consider the following protocol to prevent deadlocks:

Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -say, R_i. After that, the process can request instances of resource type *Rj* if and only if F(Rj) > F(R_i).

Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources R_i such that F(Ri) ≥ F(Rj).

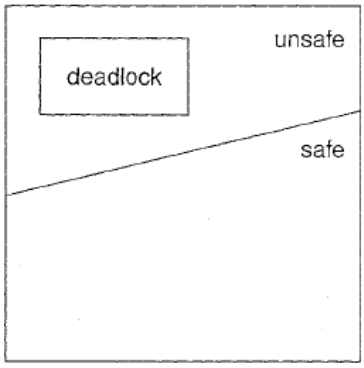
If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance

1. Explain the deadlock avoidance algorithm [8M]
2. "A safe state is not a deadlock state but a deadlock state is an unsafe state". Explain [4]
3. Define the terms: safe state and safe sequence. Give an algorithm that is used to determine whether a system is in safe state [10]
4. What is the difference between deadlock prevention and deadlock avoidance methods of dealing with a dead lock problem?

Deadlock Avoidance

- Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states.
- The algorithm will ***dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources***.
- *When a process requests a resource that is already available, the system must decide whether that resource can immediately be allocated or not.* The resource is immediately allocated only if it leaves the system in a ***safe state***.
- *A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.* More formally, a system is in a safe state only if there exists a **safe sequence**. If no such sequence exists, then the system state is said to be *unsafe.*
- A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.
- In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on.
- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.



Safe, unsafe, and deadlocked state spaces.

- An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

Avoidance Algorithms

1. **Resource-Allocation-Graph Algorithm**
2. **Banker's Algorithm**

Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph for deadlock avoidance.

In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge.**

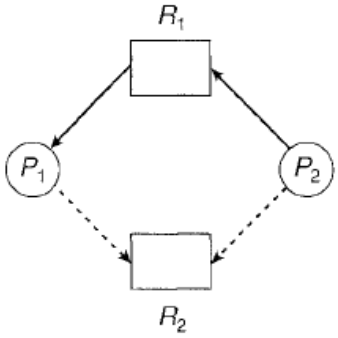
A claim edge *Pi ->Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process *Pi* requests resource *Rj,* the claim edge is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge is reconverted to a claim edge *Pi ->Rj.*

Suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi -> Rj* to an assignment edge *Rj ->Pi* does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where *n* is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied.

Consider the resource-allocation graph shown.

Suppose that *P2* requests R2 . Although R2 is currently free, we cannot allocate it to *P2,* since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.



Resource-allocation graph for deadlock avoidance.

Banker's Algorithm

1. What is Bankers algorithm? Explain.[10M]
2. Write and explain Banker's algorithm for deadlock avoidance[10M]
3. Write the resource request algorithm [3]
4. Explain banker's algorithm using the example given below A system has 5 processes, p1,p2,p3,p4 and p5.There are 3 types of resources r1,r2 and r3.there are 10 instances of r1, 5 instances of r2 and 7 instances of r3. At time t0, the situation is as follows;

Process	allocation of			maximum requirement of		
	r1	r2	r3	r1	r2	r3
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	2	2	2
P5	0	0	2	4	3	3

Is the system in a safe state at time t0? Suppose now a time t1, process p2 requests one additional instance of resource type r1 and 2 instances of resource type r3, supposing the request if granted will the system be in a safe state?

5. Consider the following snapshot of a system [10M]

Process	allocation				**Max**			
	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2
P1	1	0	0	0	1	7	5	0
P2	1	3	5	4	2	3	5	6
P3	0	6	3	2	0	6	5	2
P4	0	0	1	4	0	6	5	6
P5								

Available

A	B	C	D
1	5	2	0

Answer the following questions using Bankers Algorithm

i) What is the content of array node

ii) Is the system in a safe state ?

iii) If a request from process P1 arrives for (0,4,2,0) can the request be immediately granted ??

Bankers Algorithm

→ The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

→ *The banker's algorithm* is applicable for system with multiple instances of each resource type.

When a new process enters the system, it must *declare the maximum number of instances of each resource type that it may need*. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm.

The following data structures are needed, where *n* is the number of processes in the system and *m* is the number of resource types:

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

Safety Algorithm

→ Algorithm for finding out whether or not a systern is in a safe state.

This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, \ldots, n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == true$ for all i, then the system is in a safe state.

Resource-Request Algorithm

→ Algorithm for determining whether requests can be safely granted.

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

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for *Requesti*, and the old resource-allocation state is restored.

Deadlock Detection

1. Give deadlock detection algorithms for both single and multiple instances of the resources [10]
2. What is wait for graph? How is it useful in detecting deadlocks? [5]

b. For the following resource-allocation graph, write the corresponding wait – for graph. **(04 Marks)**

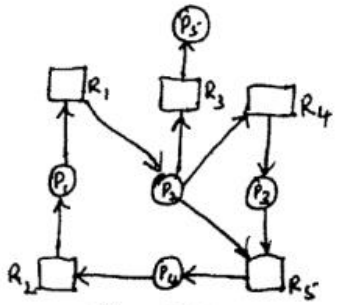


Fig. Q4(b)

3.
4. Explain the different approaches towards a deadlock detection problem when [12M]
   a. there is a single instance of a resource type
   b. Multiple instances of a resource type
5. Write and explain deadlock detection algorithm and explain how it differs from Banker's algorithm. [12M]

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- *An algorithm that examines the state of the system to determine whether a deadlock has occurred*
- *An algorithm to recover from the deadlock*

Detection Algorithms

1. **Algorithm for resource type with single instance**
2. **Algorithm for resource type with several instances.**

Single Instance of Each Resource Type

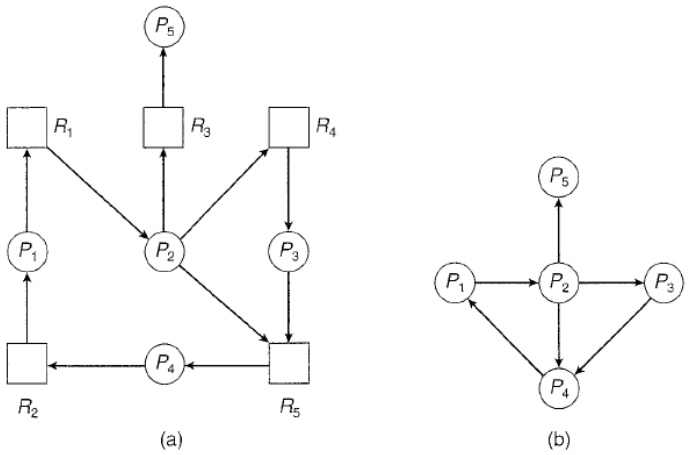
- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for* graph**.

We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs. An edge *Pi* ->*Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* -> *Rq* and *Rq*-> *Pj* for some resource *Rq.*

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.

An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



(a) Resource-allocation graph. (b) Corresponding wait-for graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

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

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n\text{-}1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

NOTE: REFER CLASS NOTES FOR PROBLEMS ON DEADLOCK DETECTION

Detection-Algorithm Usage

When should the deadlock detection be done? Frequently, or infrequently?

The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)

There are two obvious approaches, each with trade-offs:

- ✓ Do deadlock detection ***after every resource allocation*** which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
- ✓ Do deadlock detection ***only when there is some clue that a deadlock may have occurred***, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes deadlock recovery can be more complicated and damaging to more processes.

- ✓ Do deadlock checks ***periodically*** (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock.

Recovery from Deadlock

1. How does a system recover from deadlock?
2. What are the 2 options for breaking a deadlock? Explain each clearly [7]
3. Discuss the issues related with recovery from deadlock [6]

When a detection algorithm determines that a deadlock exists, several alternatives are available.

- ✓ One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- ✓ Another possibility is to let the system *recover* from the deadlock automatically.

There are two options for breaking a deadlock

- ➢ One is simply to abort one or more processes to break the circular wait.
- ➢ The other is to preempt some resources from one or more of the deadlocked processes.

1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**. This method clearly will *break the deadlock cycle, but at great expense*; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time** until the deadlock cycle is eliminated. This method incurs *considerable overhead*, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

2. Resource Pre-emption

- To eliminate deadlocks using resource pre-emption, we successively pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Following factors should be considered during preemption:

- **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has consumed during its execution.
- **Rollback.** If we preempt a resource from a process, what should be done with that process? It cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.
- **Starvation.** How do we ensure that starvation will not occur i.e., how can we guarantee that resources will not always be preempted from the same process?