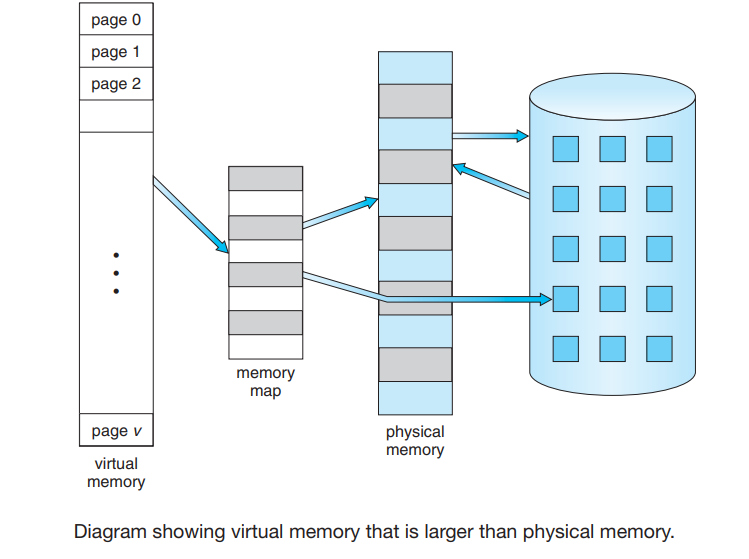
**UNIT-IV PART-I**

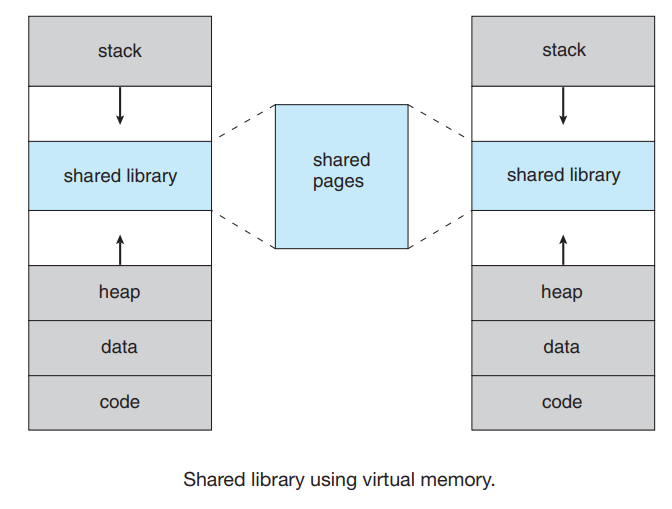
**Virtual Memory Management**

**Introduction**

* Memory-management has of one basic requirement: **The instructions being executed must be in physical memory.**
* Unfortunately, this requirement **it limits the size of a program to the size of physical memory.**
* A **careful examination** of real programs **shows us that, in many cases, the entire program is not needed.**
* **For instance**, consider the following: Programs often have **code to handle unusual error conditions**. If errors are not there, this code is almost never executed.
* The ability to execute a program that is only partially in memory would **confer many benefits**:
* A program would **no longer be constrained by the amount of physical memory** that is available.
* Because each user program could take less physical memory, **more programs could be run** at the same time, with a corresponding increase in CPU utilization and throughput
* **Less I/O would be needed** to load or swap user programs into memory
* **Virtual memory is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of process from random access memory to disk storage.**
* Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.
* Virtual memory is large when compared to physical memory.

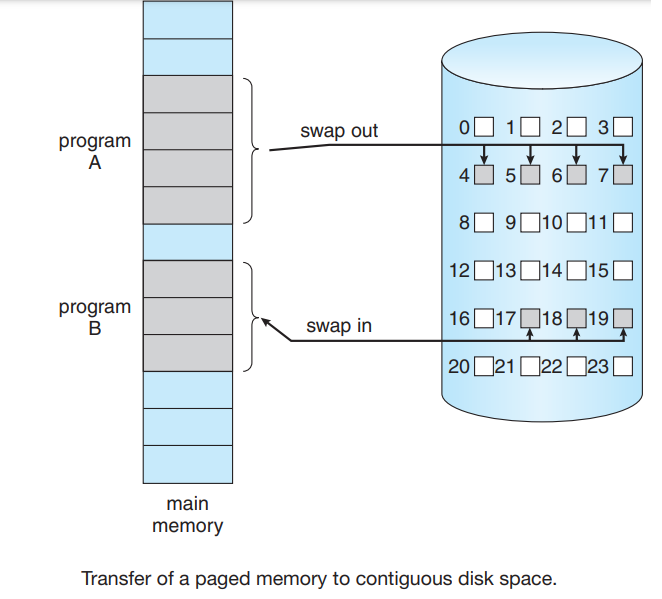


* The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
* In the above figure that we allow the heap to grow upward in memory as it is used for dynamic memory allocation.
* Similarly, we allow for the stack to grow downward in memory through successive function calls.
* In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing.

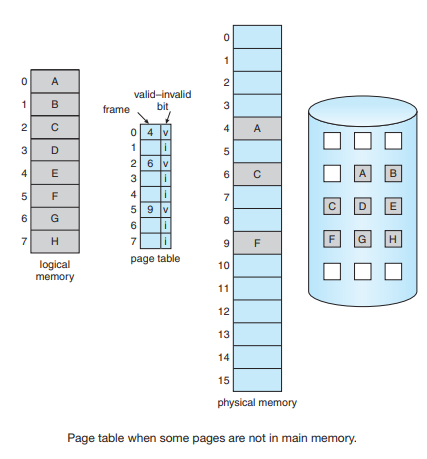


**Demand Paging**

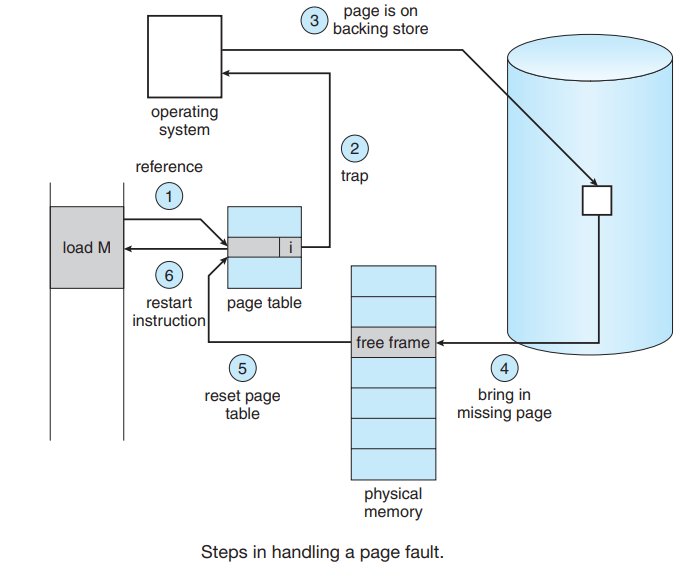
* Consider how an executable program might be loaded from disk into memory.
* One option is to load the entire program in physical memory at program execution time.
* However, a problem with this approach is that we may not initially need the entire program in memory.
* An alternative strategy is to load pages of executing program only as they are needed. This technique is known as demand paging
* With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory
* A demand-paging system is similar to a paging system with swapping
* In demand paging we use a lazy pager. A lazy pager never swaps a page into memory unless that page will be needed.



* With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.



* When the bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid or is valid but is currently on the disk.
* **A page invalid will have no effect if the process never attempts to access that page.**
* **But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault.**
* The procedure for handling this page fault is straightforward

****

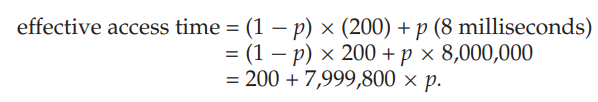
1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory. **(SeIf t validation bit = v)**
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory
7. If process executes with no more faults. This scheme is pure demand paging: never bring a page into memory until it is required

**Performance of Demand Paging**

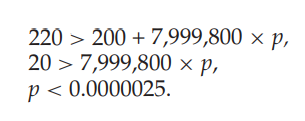
* Demand paging can significantly affect the performance of a computer system.
* As long as we have no page faults, the effective access time is equal to the memory access time.
* If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired page.
* To see why, let’s compute the effective access time for a demand-paged memory. For most computer systems, the memory-access time, denoted by ma, ranges from 10 to 200 nanoseconds.
* Let p be the probability of a page fault (0 ≤ p ≤ 1). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

****

* To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:
* Trap to the operating system.
* Save the user registers and process state.
* Determine that the interrupt was a page fault.
* Check that the page reference was legal and determine the location of the page on the disk.
* Issue a read from the disk to a free frame
* While waiting, allocate the CPU to some other user (CPU scheduling, optional).
* Receive an interrupt from the disk I/O subsystem (I/O completed).
* Save the registers and process state for the other user (if step 6 is executed).
* Determine that the interrupt was from the disk.
* Correct the page table and other tables to show that the desired page is now in memory.
* Wait for the CPU to be allocated to this process again.
* Restore the user registers, process state, and new page table, and then resume the interrupted instruction.
* Example: With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is



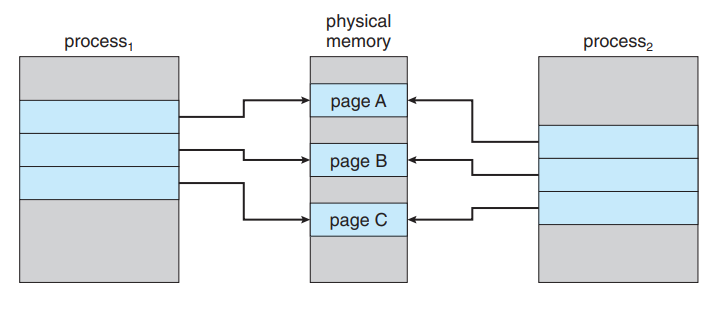
* If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds (1 millisecond = 1000 microseconds). The computer will be slowed down by a factor of 40 because of demand paging!
* If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:



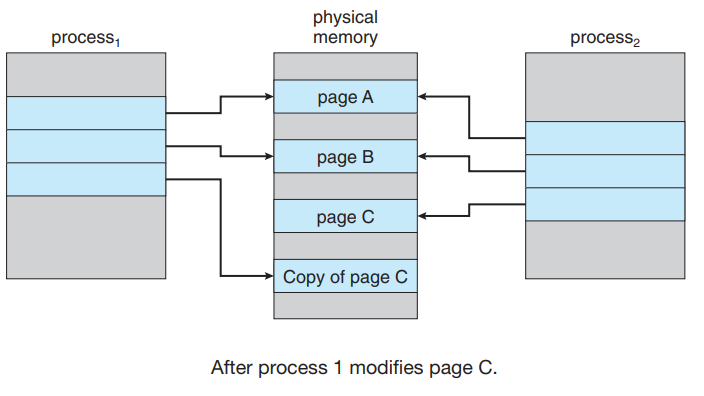
* That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer than memory access time to page-fault.

**Copy-on-Write**

* Many child processes invoke the **exec()** system call immediately after creation, the copying of the parent’s address space may be unnecessary.
* Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages.
* These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, or create a copy of the shared page is created.
* Copy-on-write is illustrated in the following figures

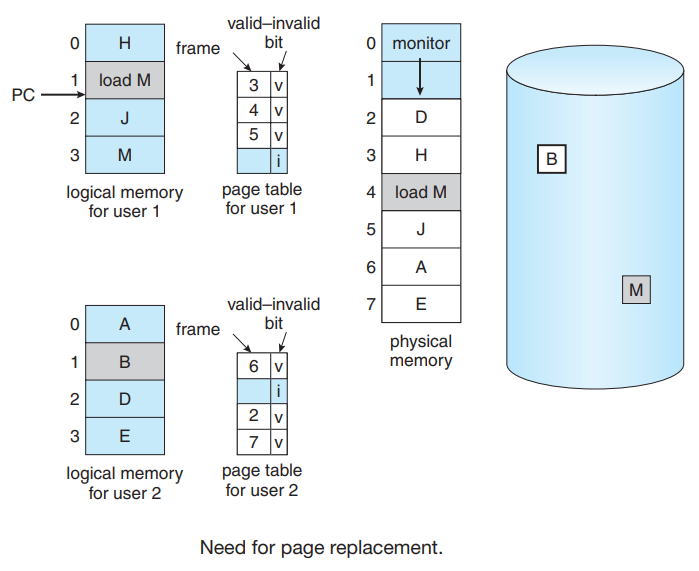


* For example, assume that the child process attempts to modify a page
* The operating system will create a copy of this page, mapping it to the address space of the child process.
* The child process will then modify its copied page and not the page belonging to the parent process.



**Page replacement**

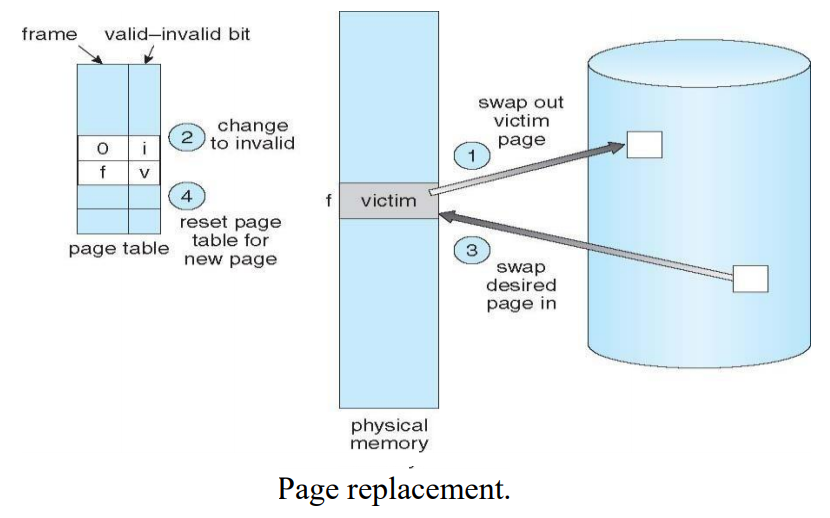
* While a user process is executing, a page fault occurs.
* The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use (see Figure below).



* The operating system has several options at this point.
  + It could terminate the user process.
  + However, demand paging is the operating system’s attempt to improve the computer system’s utilization and throughput.
  + The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.( This option is a good one in certain circumstances)
* Here, we discuss the most common solution: **page replacement.**

**Basic Page Replacement**

* Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it.
* We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (see the Figure).

****

* We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   * If there is a free frame, use it.
   * If there is no free frame, use a page-replacement algorithm to select a victim frame.
   * Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

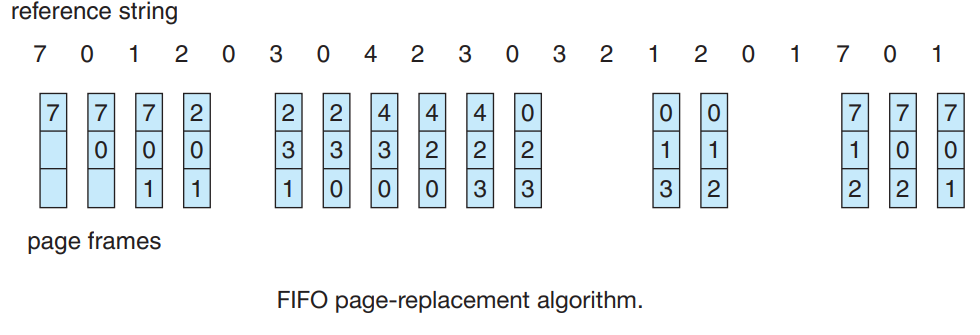
* Notice that, if no frames are free, two page transfers (one out and one in) are required. **This situation** effectively **doubles the page-fault service time** and increases the effective access time accordingly
* We must solve two major problems to implement demand paging: we must develop a frame-allocation algorithm and a page-replacement algorithm.
* There are many different page-replacement algorithms.
* We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.
  + String is just page numbers, not full addresses
  + Repeated access to the same page does not cause a page fault
  + Results depend on number of frames available
* The string of memory references is called a reference string
* For example: the reference string of referenced page numbers is **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

**Page replacement algorithm**

* FIFO Page Replacement
* Optimal Page Replacement
* LRU Page Replacement
* LRU-Approximation Page Replacement
  + Additional-Reference-Bits Algorithm
  + Second-Chance Algorithm
  + Enhanced Second-Chance Algorithm
* 5. Counting-Based Page Replacement
  + Least frequently used (LFU)
  + Most frequently used (MFU)
* 6. Page-Buffering Algorithms

**FIFO Page Replacement**

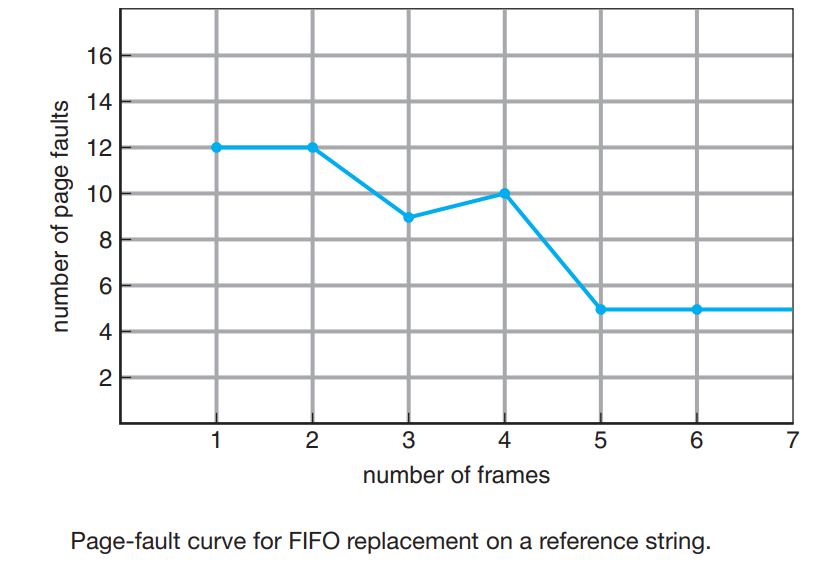
* The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
* When a page must be replaced, the first page entered in to frames is chosen.
* **Example:** Consider Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
* Consider 3 frames (3 pages can be in memory at a time per process)



* For our example reference string, our three frames are initially empty.
* The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
* The next reference (2) replaces page 7, because page 7 was brought in first.
* Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
* The first reference to 3 results in replacement of page 0, since it is now first in line.
* Because of this replacement, the next reference, to 0, will fault.
* Page 1 is then replaced by page 0.
* This process continues as shown in Figure above. Every time a fault occurs, we show which pages are in our three frames.
* **Totally there are fifteen faults altogether.**
* The FIFO page-replacement algorithm is easy to understand and program.
* However, its performance is not always good. **It has increases the page-fault rate and slows process execution**

**Example2:**

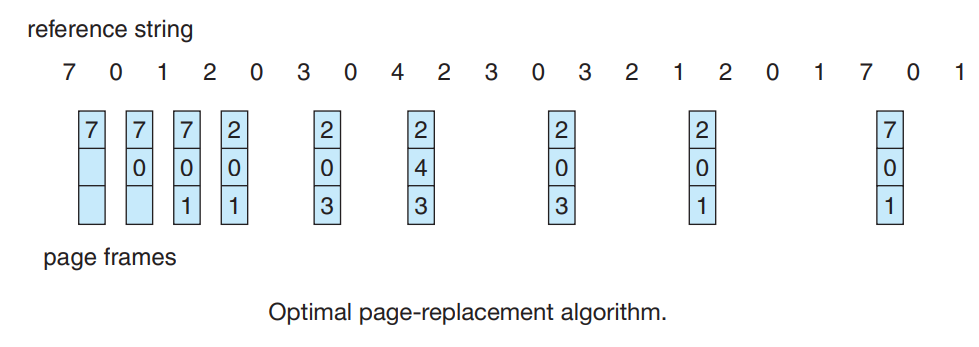
* Consider reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  + With 3 frames you will have 9 page faults
  + With 4 frames you will have 10 page faults
* The number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as **Belady’s anomaly**
* Figure below shows the curve of page faults for this reference string versus the number of available frames.



**Optimal Page Replacement**

* Optimal page-replacement algorithm—the algorithm that has the lowest page-fault rate of all algorithms and will **never suffer from Belady’s anomaly**.
* It is simply this: **Replace the page that will not be used for the longest period of time in the future.**

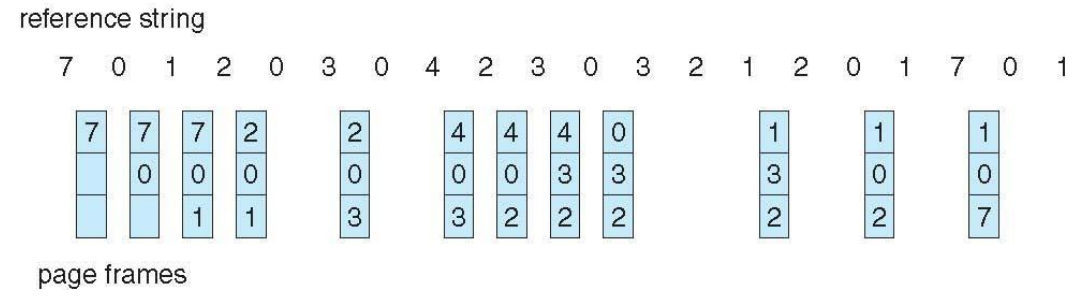
**Example:** Consider Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 [check this string to identify page that is not used for so long time]

****

* For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure above.
* The first three references cause faults that fill the three empty frames.
* The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
* The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
* With only **nine page faults**, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.
* Unfortunately, the **optimal page-replacement algorithm is difficult to implement**, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-schedulin g algorithm)

**Least Recently Used (LRU) Algorithm**

* In LRU replace the page that has not been used for the longest period of time.
* When we want to replace the page we need to consider past references.
* LRU replacement **associates with each page the time** of that page’s last use.
* When a page must be replaced, **LRU chooses the page that has not been used for the longest period of time**.



* Totally we have 12 page faults – LRU better than FIFO but worse than optimal

🡪**A. Implementation**

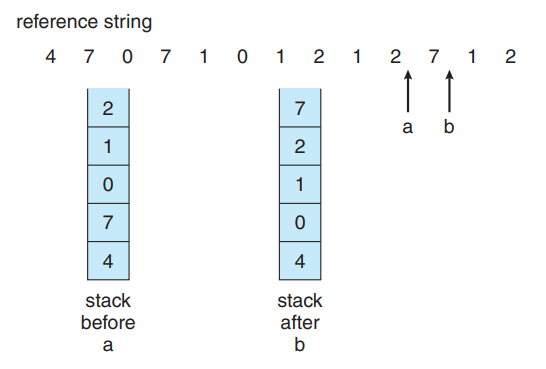
* The major problem is how to implement LRU replacement.
* An LRU page-replacement algorithm may require substantial hardware assistance.
* The problem is to determine an order for the frames defined by the time of last use.
* Two implementations are feasible:
* Using Counters
* Using Stack

**(i) Using Counters:**

* A logical counter (called time of use field) is associated with each page-table entry.
* This counter is incremented for every memory reference.
* We replace the page with the smallest time value.
* This scheme requires a search of the page table to find the LRU page and a write to for each memory access

**(i) Using Stack:**

* Another approach to implementing LRU replacement is to keep a stack of page numbers.
* Whenever a page is referenced, it is removed from the stack and put on the top.
* In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom (see the figure).



* Because **entries must be removed from the middle of the stack**, it is **best to implement** this approach by using a **doubly linked list**
* Like optimal replacement, LRU replacement **does not suffer from Belady’s anomaly.**

**LRU-Approximation Page Replacement**

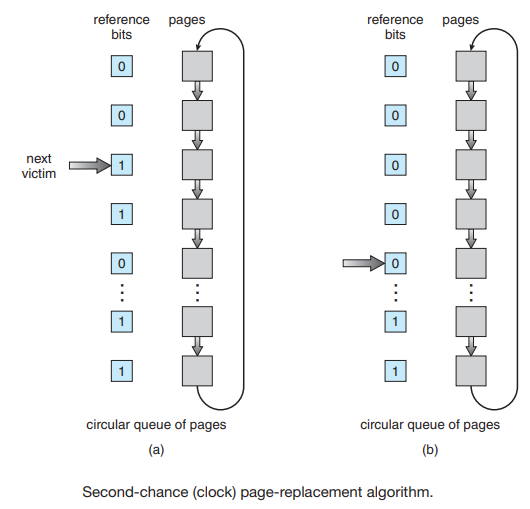
* The **updating of the counter or stack** must be done for every memory reference, it would **slow every memory reference** by a factor of at least ten
* LRU **need sufficient hardware support**. In fact, **some systems provide no hardware support,**

**(i) Additional-Reference-Bits Algorithm**

* The reference bit for a page is set by the hardware whenever that page is referenced
* Reference bits are associated with each entry in the page table.
* Initially, all bits are cleared (to 0) by the operating system.
* As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.
* We can determine which pages have been used and which have not been used by examining the reference bits

**(ii) Second-Chance Algorithm**

* When a page has been selected, **we inspect its reference bit.**
* **If** the value is **0**, we proceed to **replace this page**;
* But **if** the reference bit is **set to 1**, we **give the page a second chance** and move on to select the next FIFO page.
* When a **page gets a second chance**, its **reference bit is cleared**, and **its arrival time is reset to the current time.**
* Thus, **a page** that is **given a second chance will not be replaced until all other pages** have been replaced
* **One way to implement the second-chance algorithm** (sometimes referred to as the clock algorithm) is as a circular queue.

****

* A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit
* As it advances, it clears the reference bits (see the Figure above). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
* Second-chance replacement degenerates to FIFO replacement if all bits are set.

**(iii) Enhanced Second-Chance Algorithm**

* Improve algorithm by using reference bit and modify bit
* Take ordered pair (reference, modify)
  + (0, 0) neither recently used not modified – best page to replace
  + (0, 1) not recently used but modified – not quite as good, must write out before replacement
  + (1, 0) recently used but clean – probably will be used again soon
  + (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
* When page replacement is called for; instead of examining reference bit set to 1, we examine the class to which that page belongs

**Counting-Based Page Replacement**

* There are many other algorithms that can be used for page replacement.
* For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

**(i) Least frequently used (LFU) page-replacement algorithm**

* The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced.
* The reason for this selection is that an actively used page should have a large reference count.
* A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again.
* Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
* One solution is to shift the counts right by 1 bit at regular intervals,

**(ii) Most frequently used (MFU) page-replacement algorithm**

* The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

**Page-Buffering Algorithms**

* **Systems** commonly keep a pool of **free frames.**
* When a **page fault occurs**, a victim frame is chosen as before, the **desired page is read into a free frame** from the pool **before the victim is written out**[victim page will be placed in buffer].
* This procedure allows the process to **restart as soon as possible**, without waiting for the victim page to be written out.
* When the victim is later written out, its frame is added to the free-frame pool.
* An **expansion of this idea** is to maintain a **list of modified pages.**
* A **modified page** is selected and is **written to the disk**. Its modify **bit is then reset.**
* This scheme **increases the probability that a page** will be **clean** when it is **selected for replacement** and will not need to be written out.
* Another modification is to **keep a pool of free frames but to remember which page was in each frame**.
* The **old page can be reused directly** from the free-frame pool if it is needed before that frame is reused.

**Applications and Page Replacement**

* All of these algorithms have OS guessing about future page access
* Some applications have better knowledge – i.e. databases
* Memory intensive applications can cause double buffering
  + OS keeps copy of page in memory as I/O buffer
  + Application keeps page in memory for its own work
* Operating system can given direct access to the disk, getting out of the way ofthe applications
* Raw disk mode o Bypasses buffering, locking, etc

**Frame allocation**

How do we allocate the fixed amount of free memory among the various processes?

If we have 93 free frames and two processes, how many frames does each process get?

* Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames.
* The operating system may take 35 KB, leaving 93 frames for the user process.
* Under pure demand paging, all 93 frames would initially be put on the free-frame list.
* When a user process started execution, it would generate a sequence of page faults.
* The first 93 page faults would all get free frames from the free-frame list.
* When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
* When the process terminated, the 93 frames would once again be placed on the free-frame list.

**Minimum Number of Frames**

* We must also allocate at least a minimum number of frames for the following reasons.
  + One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
  + When a page fault occurs before an executing instruction is complete, the instruction must be restarted
* The minimum number of frames is defined by the computer architecture

**Allocation Algorithms**

**(i)Equal Allocations**

* The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames.
* For instance, if there are 93 frames and five processes, each process will get 18 frames.
* The three leftover frames can be used as a free-frame buffer pool. This scheme is called equal allocation.

**(ii) Proportional allocation**

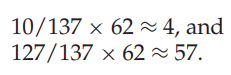
* Allocate available memory to each process according to its size(s)
* Let the total size of the virtual memory for process **pi** be **si** , and define

****

* Then, if the total number of available frames is m, we allocate **ai** frames to process **pi** , where **ai** is approximately

****

* Example: We have 62 frames(m) to be allocated b/w between two processes, P1 with size(s1) 10 pages and p2 with size 127 pages, Then number of frames allocated to each process is given as



* Process P1 is allocated with 4 frames and P2 allocated with 57 frames, respectively.

**Global versus Local Allocation**

* With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: global replacement and local replacement.
* Global replacement **allows a process to select a replacement frame from the set of all frames**, even if that frame is currently allocated to some other process
* That is, one process can take a frame from another.
* **Local replacement** requires that **each process select from only its own** set of allocated frames.

**Non-uniform memory access (NUMA)**

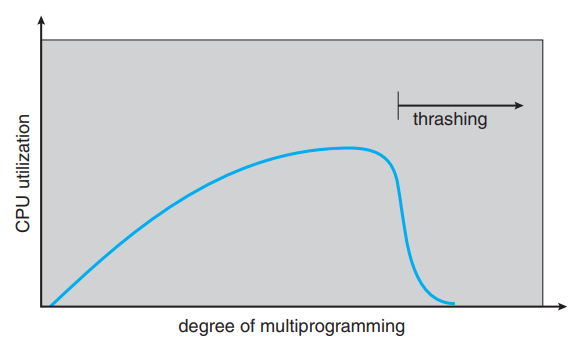
* **Computer systems are made up of variety of system boards,** each containing multiple CPUs and some memory.
* The CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system.
* Systems in which **memory access times vary significantly** are known collectively as non-uniform memory access (NUMA) systems

**Thrashing**

* The high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.
* High paging activity means, Consider process does not have the number of frames it needs to support pages
* At this point, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again
* This situation is called **trashing**. Thrashing results in severe performance problems.

**Cause of Thrashing**

* The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue
* As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. This situation is called trashing
* This phenomenon is illustrated in Figure below



**Soultion1:**

* At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

**Solution2:**

* We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm).
* With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved.

**Soultion3:**

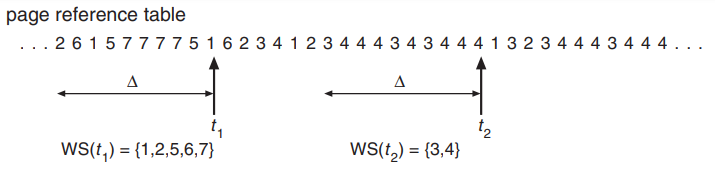
* To prevent thrashing, we must provide a process with as many frames as it needs.
* But how do we know how many frames it “needs”?
* There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

**🡪Locality**

* The locality model states that, as a process executes, it moves from locality to locality.
* A locality is a set of pages that are actively used together
* A program is generally composed of several different localities, which may overlap.
* For example, when a function is called, it defines a new locality. In this locality, there will pages for its instructions, its local variables, and a subset of the global variables
* Suppose we allocate enough frames to a process to accommodate its current locality.
* It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
* If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash,

**Working-Set Model**

* The working-set model is based on the assumption of locality
* This model uses a parameter, , to define the working-set window.
* The set of pages in the most recent page references is the working set
* For example, given the sequence of memory references shown in Figure 9.20, if = 10 memory references, then the working set at time t1 is {1, 2, 5, 6, 7}. By time t2, the working set has changed to {3, 4}.



* The most important property of the working set, then, is its size.
* If we compute the working-set size, WSSi , for each process in the system, we can then consider that



* Where D is the total demand for frames.
* Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur
* The operating system monitors the working set of each process
* If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend
* This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.

**Page-Fault Frequency**

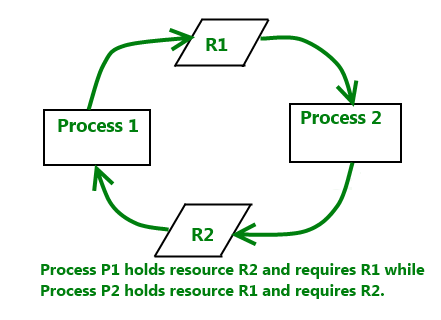
* The working-set model is a clumsy way to control thrashing.
* A strategy that uses the page-fault frequency (PFF) takes a more direct approach.
* Thrashing has a high page-fault rate. When it is too high, we know that the process needs more frames.
* Conversely, if the page-fault rate is too low, then the process may have too many frames.
* We can establish upper and lower bounds on the desired page-fault rate
* If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.
* If the page-fault rate falls below the lower limit, we remove a frame from the process.

**PART-II**

**DEADLOCKS**

**4.1 Introduction**

* A **deadlock** is a situation where in two or more process are waiting for each other in order to be finished but none of the processes are willing to give up the resources that other process needs.

****

**4.2 RESOURCES**

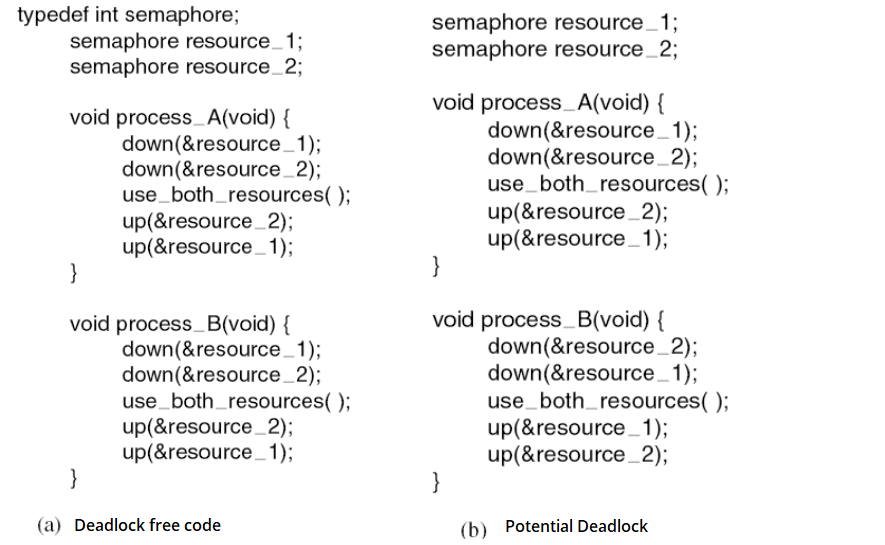
* A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database).
* Deadlocks can occur when processes have been granted exclusive access to devices, data records, files, and so forth.

**4.2.1 Preemptable and Non-preemptable Resources**

* A preemptable resource is one that can be taken away from the process
* **Memory is an example of a preemptable resource.**
* Consider, for example, a system with 256 MB of user memory, one printer, and two processes that each want to print something.
  + Process A requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.
  + Process B now runs and tries, unsuccessfully, to acquire the printer but acquires memory, we now have a deadlock situation,
  + Fortunately, it is possible to preempt (take away) the memory from B by swapping it out and swapping A in. Now A can run, do its printing, and then release the printer. No deadlock occurs
* A non preemptable resource, in contrast, is one that cannot be taken away from processes. E.g CD ROM
* Suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD.
* In general, deadlocks involve non preemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved

**4.2.2 Resource Acquisition**

* For some kinds of resources, such as records in a database system, it is up to the user processes to manage resource usage themselves.
* One way of allowing user management of resources is to associate a semaphore with each resource.
* Now let us consider a situation with two processes, A and B, and two resources.



* In Fig.(a), one of the processes will acquire the first resource before the other one.
* That process will then successfully acquire the second resource and do its work.
* If the other process attempts to acquire resource 1 before it has been released, the other process will simply block until it becomes available.
* In Fig.(b), If process A acquires resource 1 and process B acquires resource 2. Each one will now block when trying to acquire the other one. Neither process will ever run again. This situation is a deadlock.

**Conditions for resource deadlocks**

* The following four conditions must hold for there to be a (resource) deadlock:

1. **Mutual exclusion condition**. Each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition**. Processes currently holding resources that were granted earlier can request new resources.
3. **No preemption condition**. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. **Circular wait condition.** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

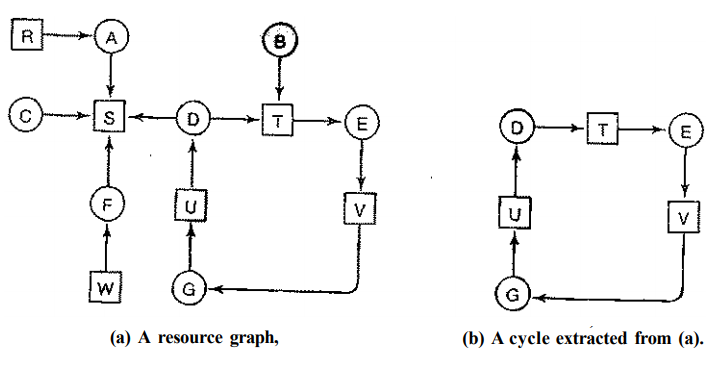
* All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.

**4.5 Deadlock detection and recovery**

* A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring.
* Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact

**4.5.1 Deadlock Detection with One Resource of Each Type**

* Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive.
* For such a system, we can construct a resource graph of the sort illustrated in below Figure
* If this graph contains one or more cycles, a deadlock exists.
* As an example, consider a system with seven processes, A though G, and six resources, R through W.
* state of which resources are currently owned and which ones are currently being requested is as follows:



* Process A holds R and wants S.
* Process B holds nothing but wants T.
* Process C holds nothing but wants S.
* Process D holds U and wants S and T.
* Process E holds T and wants V.
* Process F holds W and wants S.
* Process G holds V and wants U.
* We can construct the resource graph to detect deadlock
* This graph contains one cycle, we can see the processes D, E, and G are all deadlocked
* Processes A, C, and F are not deadlock because S can be allocated to any one of them, which then finishes and returns
* It is relatively simple to pick out the deadlocked processes by eye, but we need a formal algorithm for detecting deadlocks.
* Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exists
* It uses one dynamic data structure, L, a list of nodes, as well as the list of arcs
* The algorithm operates by carrying out the following steps as specified:

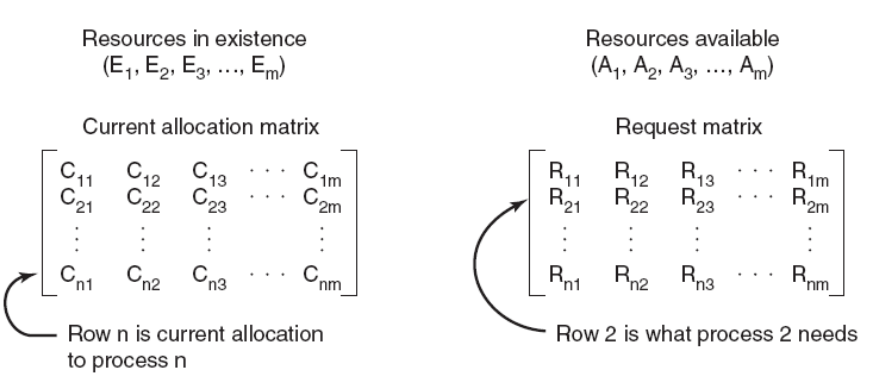
1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3

**Example:**

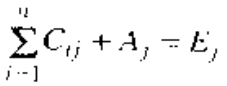
* To see how the algorithm works in practice, let us use it on the graph of Fig. (a).
* The order of processing the nodes is arbitrary, so let us just inspect them from left to right, top to bottom,
* Start running the algorithm from R, then successively A, B, C, S, D, T, E, F, and so forth. If we hit a cycle, the algorithm stops.
* We start at R and initialize L to the empty list.
* Then we add R to the list and move to the only possibility, A, and add it to L, giving L = [R, A].
* From A we go to S, giving L = [R, A, S]. S has no outgoing arcs, so it is a dead end, forcing us to backtrack to A.
* Since A has no unmarked outgoing arcs, we backtrack to R, completing our inspection of R.
* Now we restart the algorithm starting at A, resetting L to the empty list. This search, too, quickly stops, so we start again at B.
* From B we continue to follow outgoing arcs until we get to D, at which time L - [B, T, E, V, G, U, D].
* Now we must make a (random) choice. If we pick S we come to a dead end and backtrack to D.
* The second time we pick T and update L to be [B, T, E, V, G, U,D,T], at which point we discover the cycle and stop the algorithm.

**Deadlock Detection with Multiple Resources of Each Type**

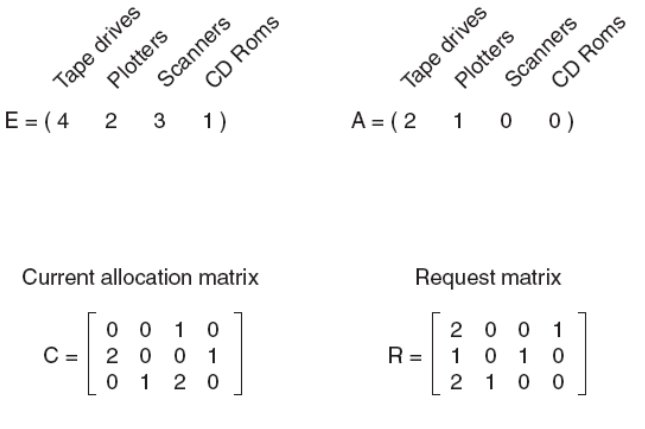
* When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks.
* We will now present a matrix-based algorithm for detecting deadlock among n processes**, P1,P2…… Pn**
* Let the number of resource classes be m, with **E1 resources of class 1**, E2 resources of class 2, and generally, Ei resources of class i (1 < i < m).
* **E** is the existing resource vector. It gives the total number of instances of each resource in existence.
* For example, if class 1 is tape drives, then **E1 = 2** means the system has two tape drives.
* At any instant, some of the resources are assigned and are not available. Let A be the available resource vector, If both of our two tape drives are assigned, A1 will be 0.
* Now we need two arrays, **C,** the current allocation matrix, and **R**, the request matrix.
* The i-th row of **C** tells how many instances of each resource class Pj currently holds.
* Thus **Cij** is the number of instances of resource j that are held by process i.
* Similarly, **Rij** is the number of instances of resource **j** that **Pi** wants. These four data structures are shown in fig below



* if we **add up all the instances of the resource j that have been allocated** and **to** this add **all the instances that are available**, the **result is** **the number of instances of that resource class that exist.**

****

* The deadlock detection algorithm is based on comparing vectors. Let us define the relation **A ≤ B** on two vectors A and B to mean that each element of **A** is less than or equal to the corresponding element of **B.**
* Mathematically, **A≤ B** holds if and only if , **Ai ≤ Bi for 1≤ i ≤ m**
* Each process is **initially said to be unmarked**. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked.
* When the **algorithm terminates, any unmarked processes are known to be deadlocked.**
* The deadlock detection algorithm can now be given as follows.
  1. Look for an unmarked process, Pi, for which the i-th row of R is less than or equal to A.
  2. If such a process is found, add the i-th row of C to A, mark the process, and go back to step 1.
  3. If no such process exists, the algorithm terminates.
* When the algorithm finishes, all the unmarked processes, if any, are deadlocked
* As an example of how the deadlock detection algorithm works, consider Fig. below.



* Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive.
* Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners.
* Each process needs additional resources, as shown by the R matrix .
* **To run the deadlock detection algorithm**, we look for a process whose resource request can be satisfied.
* The first one cannot be satisfied because there is no CD-ROM drive available.
* The second cannot be satisfied either, because there is no scanner free.
* Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

A = (2,2, 2, 0)

* At this point process 2 can run and return its resources, giving

A=(4, 2, 2 ,1)

* Now the remaining process can run. There is no deadlock in the system.

**Note:** Suppose that process 2 needs a CD-ROM drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system is deadlocked.

**Recovery from Deadlock**

* Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again.

**(i) Recovery through Preemption**

* In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
* The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource

**(ii) Recovery through Rollback**

* If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes check pointed periodically
* Check pointing a process means that its state is written to a file so that it can be restarted later.
* The checkpoint contains not only the memory image, but also the resource state.In other words, which resources are currently assigned to the process
* When a deadlock is detected, it is easy to see which resources are needed.
* To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting one of its earlier checkpoints.
* All the work done since the checkpoint is lost

**(iii) Recovery through Killing Processes**

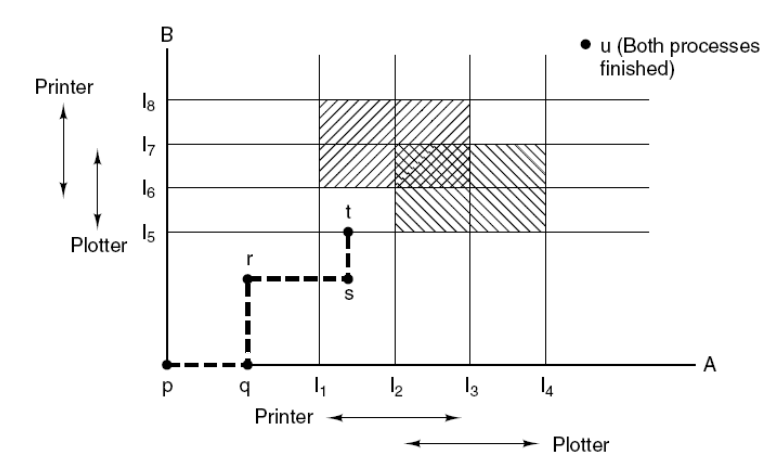
* One possibility is to kill a process in the cycle. If this does not help, it can be repeated until the cycle is broken.
* Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources.
* In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs

**Deadlock Avoidance**

* The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe.
* Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time?
* The answer is yes ----we can avoid deadlocks, but only if certain information is available in advance.

**Resource Trajectories**

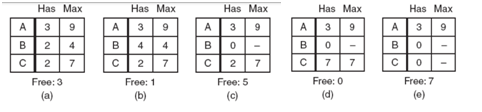
* The main algorithms for doing deadlock avoidance are based on the concept of safe states.
* In Figure below, we see a model for dealing with two processes and two resources, for example, a printer and a plotter.
* The horizontal axis represents the number of instructions executed by process A. The vertical axis represents the number of instructions executed by process B.
* At I1 A requests a printer; at l2 it needs a plotter.
* The printer and plotter are released at I3 and I4
* Process B needs the plotter from I5 to I6 and the printer from I6 to I7



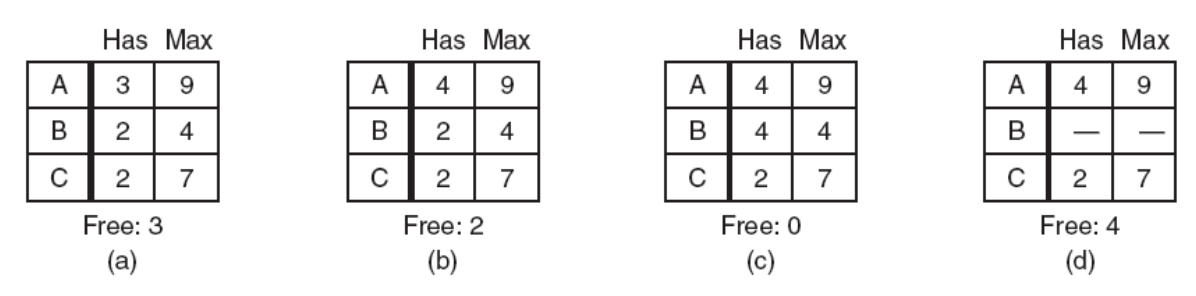
* Every point in the diagram represents a joint state of the two processes.
  + Initially, the state is at p, with neither process having executed any instructions
  + If the scheduler chooses to run A first, we get to the point q, in which A has executed some number of instructions, but B has executed none.
  + At point q the trajectory becomes vertical, indicating that the scheduler has chosen to run B.
  + When A crosses the I1 line on the path from r to s, it requests and is granted the printer.
  + When B reaches point t, it requests the plotter.
* The region with lines slanting from southwest to northeast represents both processes having the printer.
* Similarly, the region shaded the other way represents both processes having the plotter
* It will eventually a deadlock when it process A and process B gets to the intersection at I2 & I6.
* The important thing to see here is that at point t, B is requesting a resource. The system must decide whether to grant it or not.
* If the grant is made, the system will enter an unsafe region and eventually deadlock.
* To avoid the deadlock, B should be suspended until A has requested and released the plotter.

**Safe and Unsafe States**

* State is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately.



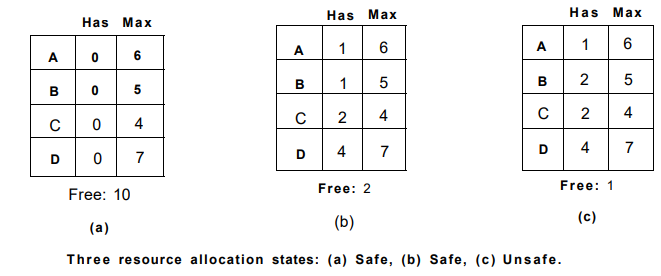
* We have a state in which A has three instances of the resource but may need as many as nine eventually.
* B currently has two and may need four altogether, later.
* Similarly, C also has two but may need an additional five.
* A total of 10 instances of the resource exist, so with seven resources already allocated, there are three still free.
* The scheduler could simply run B. When B completes, we get the state of Fig. (c). Then the scheduler can run C, leading eventually to Fig. (d).
* When C completes, we get Fig. (e). Now A can get the six instances of the resource it needs and also complete.
* Thus the state of Fig. (a) is safe because the system, by careful scheduling, can avoid deadlock.
* Now suppose we have the initial state shown in Fig. 6-10(a), but this time A requests and gets another resource, giving Fig. (b).
* The scheduler could run B until it asked for all its resources, as shown in Fig. (c).



* Eventually, B completes and we get the situation of Fig. (d). At this point we are stuck.
* We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion.
* Thus the allocation decision that moved the system from Fig. (a) to Fig. (b) went from a safe state to an unsafe state.
* **It is worth noting that an unsafe state is not a deadlocked state.**
* Furthermore, it is possible that A might release a resource before asking for any more, allowing C to complete and avoiding deadlock altogether.

**The Banker's Algorithm for a Single Resource**

* A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the banker's algorithm and is an extension of the deadlock detection algorithm
* In this model small-town banker might deal with a group of customers to whom he has granted lines of credit.
* What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied.
* If granting the request leads to a safe state, it is carried out
* In Fig. (a) we see four customers, A, B, C, and D, each of whom has been granted a certain number of credit units
* The banker knows that not all customers will need their maximum credit immediately
* In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system

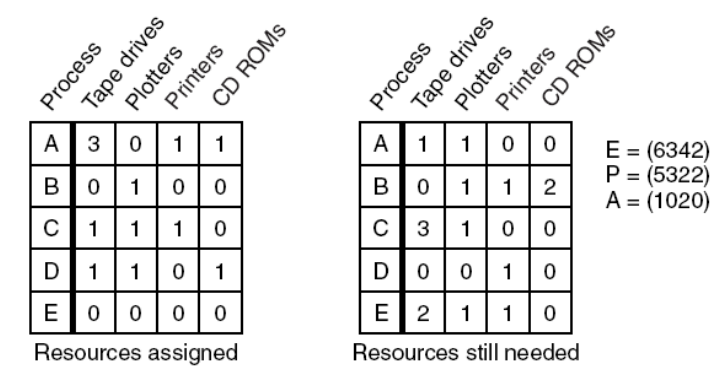


* At a certain moment, the situation is as shown in Fig.(b). This state is safe because with two units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources.
* With four units in hand, the banker can let either D or B have the necessary units, and so on.
* Consider what would happen if a request from B for one more unit were granted in Fig. (b).
* We would have situation Fig. (c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock.

**The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed**

**The Banker's Algorithm for Multiple Resources**

* The banker's algorithm can be generalized to handle multiple resources.
* In below Figure , we see two matrices shown. The one on the left shows how many of each resource are currently assigned to each of the five processes.



* The matrix on the right shows how many resources each process still needs in order to complete.
* These matrices are just C and R from
* The three vectors at the right of the figure show the existing resources, E, the possessed resources, P, and the available resources, A, respectively.
* From E we see that the system has six tape drives, three plotters, four printers, and two CDROM drives
* Of these, five tape drives, three plotters, two printers, and two CDROM drives are currently assigned.
* The algorithm for checking to see if a state is safe can now be stated.
  + Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion
  + Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the A vector.
  + Repeat steps 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resource needs can be met
* Now let us get back to the above figure. The current state is safe.
* Suppose that process B now makes a request for the printer.
* This request can be granted because the resulting state is still safe. process D can finish, and then processes A or E, followed by the rest)
* Now imagine that after **giving B one of the two remaining printers, E wants the last printer.** Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to **deadlock.**

**DEADLOCK PREVENTION**

* Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock?
* The answer is to go back to the four conditions (Mutual exclusion, no preemption, Hold & wait, Circular wait,) and ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible

**Attacking the Mutual Exclusion Condition**

* First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks.
* However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos.
* We can follow two things
* **Shareable resources**, in contrast, **do not require mutually exclusive access** and thus **cannot be involved in a deadlock** (i.e.,read-only files). A process never needs to wait for a shareable resource.
* Providing spooling space and Spooled devices (e.g., printer)
* In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.

**Attacking the Hold and Wait**

* Each process requests all resources before starting execution.
* If everything is available, all resources (requested by the process) are located and finish its job.
* If some resources are not available, no resources are allocated to the process.
* Problems

❖ May not know required resources at start of run

❖ Ties up resources other processes could be using

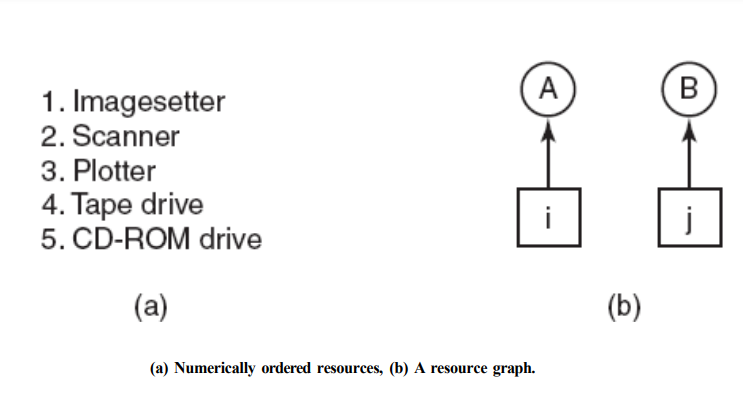
* An alternative protocol allows a process to request resources only when it has none.
* A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
* A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once

**Attacking the No preemption**

* 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 currently being held are preempted.
* The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources

**Attacking the No CircularWait**

* The circular wait can be eliminated in several ways.
* One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one.
* Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Figure below.



* A process can request resources whenever they want, but all requests must made in numerical order.
* Each process can request resources only in an increasing order of enumeration.
* With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig (b).
* We can get a deadlock only if A requests resource j and B requests resource i.
* Assuming i and j are distinct resources, they will have different numbers.
* If i > j, then A is allowed to request j because that is lower than what it already has.
* If i < j, then B is not allowed to request i because that is lower than what it already has. Either way, deadlock is impossible.
* Summary of approaches to deadlock prevention.

