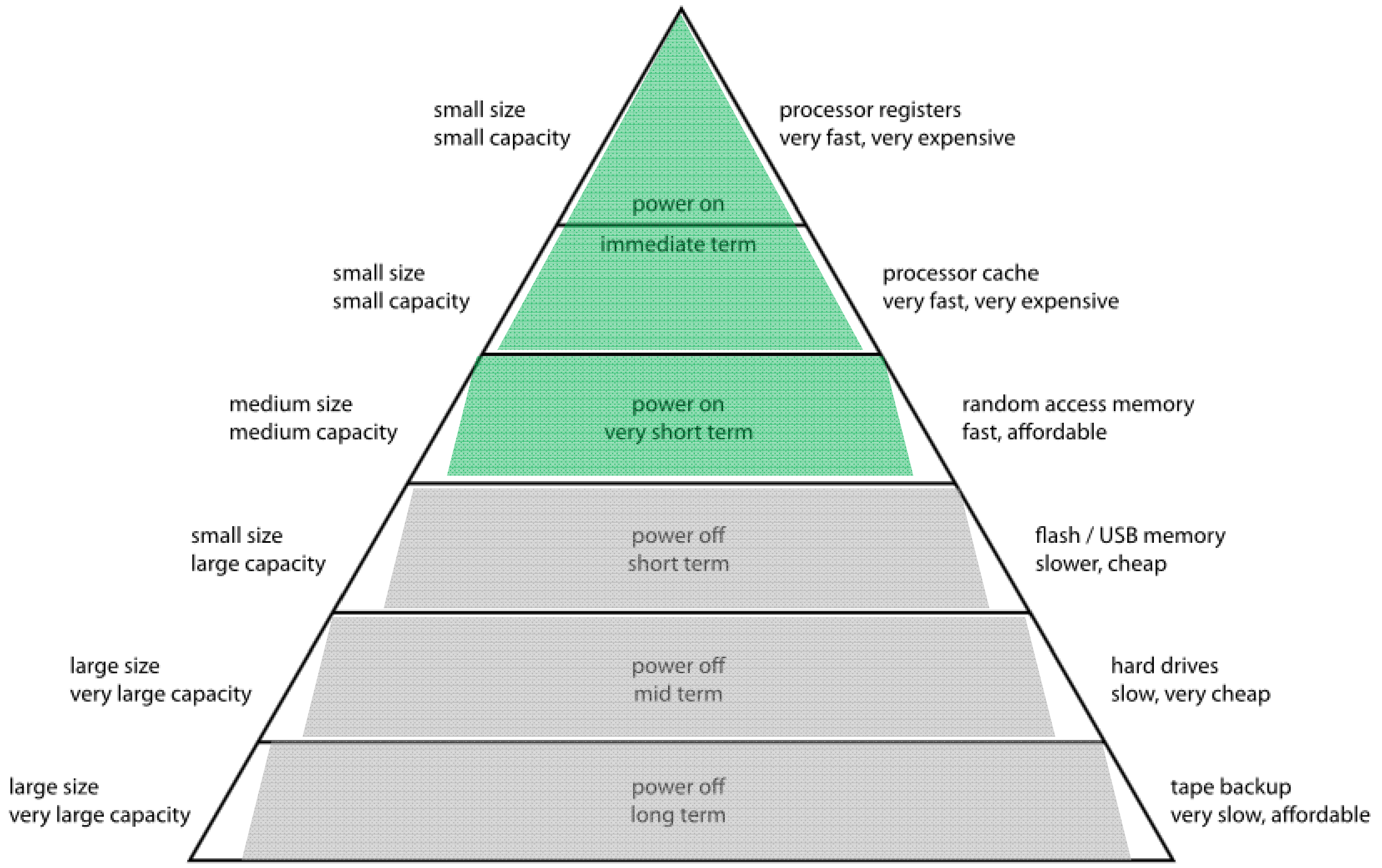
Memory Management

Mangala N

SSDG, C-DAC KP, Bangalore

Computer Memory Hierarchy



Memory Management

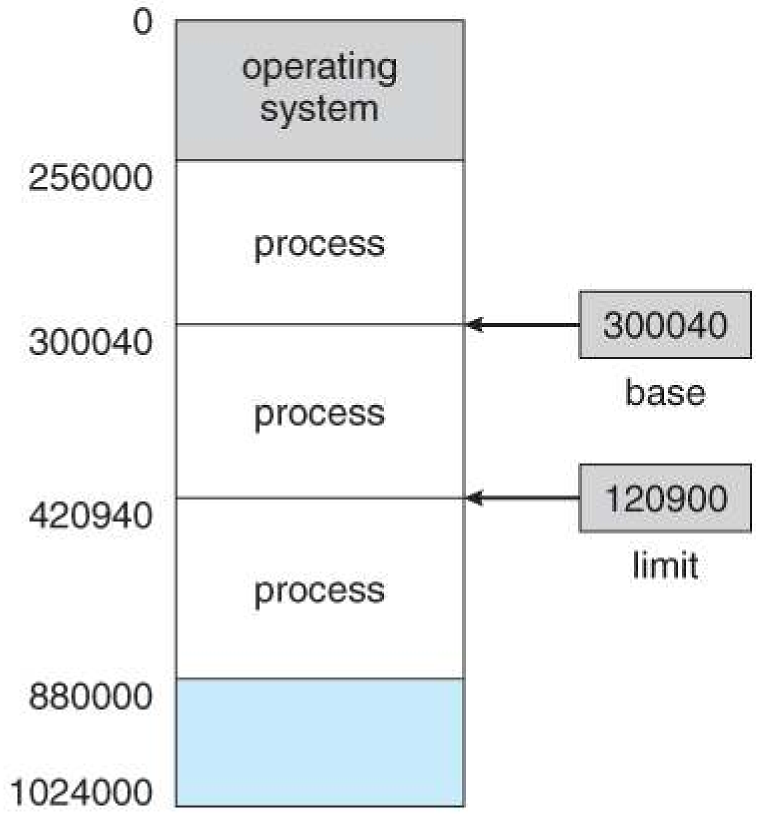
- Computation requires memory (instructions & data)
- Each process gets from the memory a block of memory to use
- But the process itself handles the internal management of that memory
- Program is first converted into a load module; then when the process is started then the load module is loaded from the disk into memory

Memory Management

- The problem the OS has is how best to manage the memory resource, that is how and when to divide the memory into blocks and how and when to allocate the blocks to processes.
- Multiprogramming issues
  - We want to have as many programs in memory as possible, and always the most useful ones
  - Two problems are:
    - Memory Allocation
    - Memory Protection

Process :: Memory

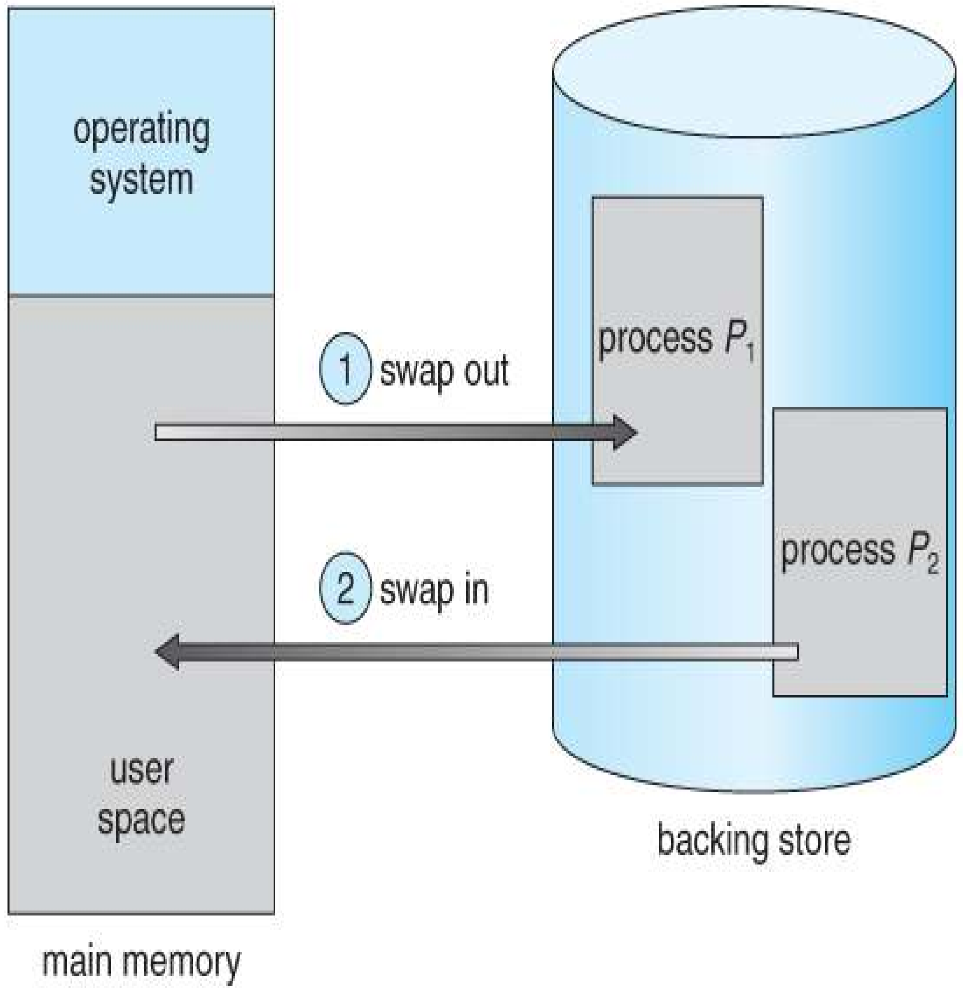
- The memory hardware doesn't know what a particular part of memory is being used for.
- Chunks of memory at a time are transferred from the main memory to the cache, and then access to individual memory locations one at a time from the cache.
- User processes must be restricted so that they only access memory locations that "belong" to that particular process.
- This is usually implemented using a **base register** and a **limit register** for **each process.**
- ***Every*** memory access made by a user process is checked against these two registers.



- The OS has access to all existing memory locations, as this is necessary to swap users' code and data in and out of 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.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes at the same time, some processes who are not currently using the CPU may have their memory swapped out local disk called ***backing store.***
- If compile-time/ load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If runtime binding is used, then processes can be swapped back into any available location.



Address Binding

- Symbolic names such as "i", "count", and "averageTemperature actually reside in memory addresses
- **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)
- **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.

Address generated by CPU -> **Logical Address**
Address seen by the MMU -> **Physical Address**

Load Time Dynamic Linking

- It does not insert the library object modules into the load module.
- Instead it inserts information about where to find the library and where to load them in the program’s address space.
- This takes little longer, since loader has to do more work, but saves disk space that would have been wasted with duplicate copies of library modules
- Loading with load time dynamic linking will only work if the libraries are exactly in the same place in the file system both when they were linked as well as loaded

Runtime Dynamic Linking

- Defer loading until you actually need the module.
- You add *code* to the program to check if a module has been loaded.
- This can be done by accessing the module indirectly with a pointer.
- The pointer can be initially set to interrupt the ***dynamic linker***
- The first time you call a procedure in the module, the call fails because the indirect pointer is not set. This starts the dynamic linker which links the module, loads it into memory from disk and fixes the indirect pointer so that next time the procedure will be called without an interrupt.
- Adv: Easy upgrades and updates

Runtime Dynamic linking contd

- *Dynamic linking* defers linking process until you actually need the module, e.g ELF, MS DLL
- ELF shared libraries can be loaded at any address, so they invariably use position independent code (PIC); can be shared among multiple processes
- An ELF program looks much the same, but in the read-only segment has init and fini routines, an INTERP section to specify the name of the dynamic linker (usually Shared Object ld.so).
- **Starting the dynamic linker**

  When the OS runs the program, it maps in the file's pages as normal, but notes that there's an INTERPRETER section (ld.so) in the executable.
- **Finding the libraries**

  Once the linker's own initializations are done, it finds the names of the libraries required by the program.

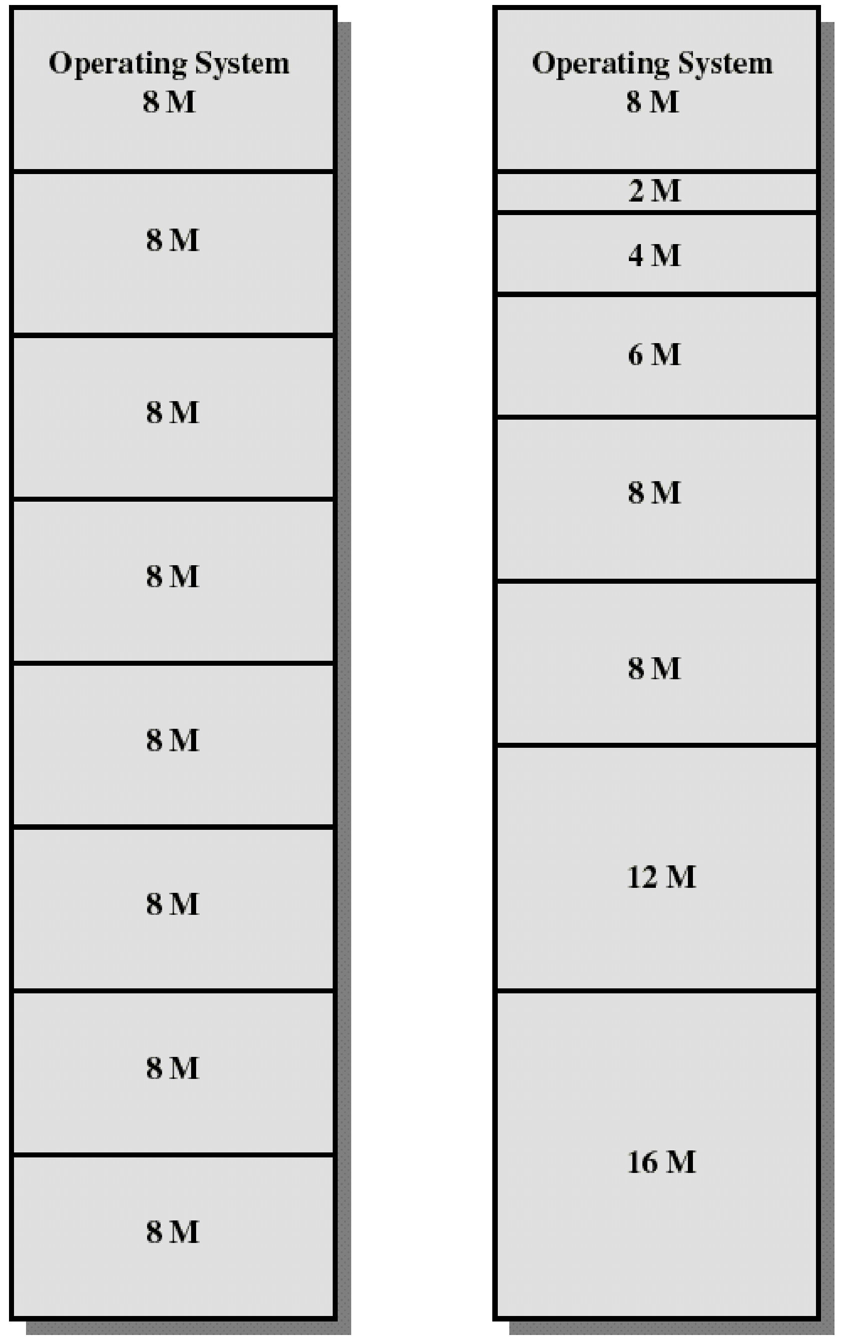
  For each library, the linker finds the library's ELF shared library file, which is in itself a fairly complex process.

Real Memory Management Techniques

- Although the following simple/basic memory management techniques are not used in modern OSs, they lay the ground for a later proper discussion of virtual memory:
  - Fixed/Static Partitioning
  - Variable/Dynamic Partitioning
  - Simple/Basic Paging
  - Simple/Basic Segmentation

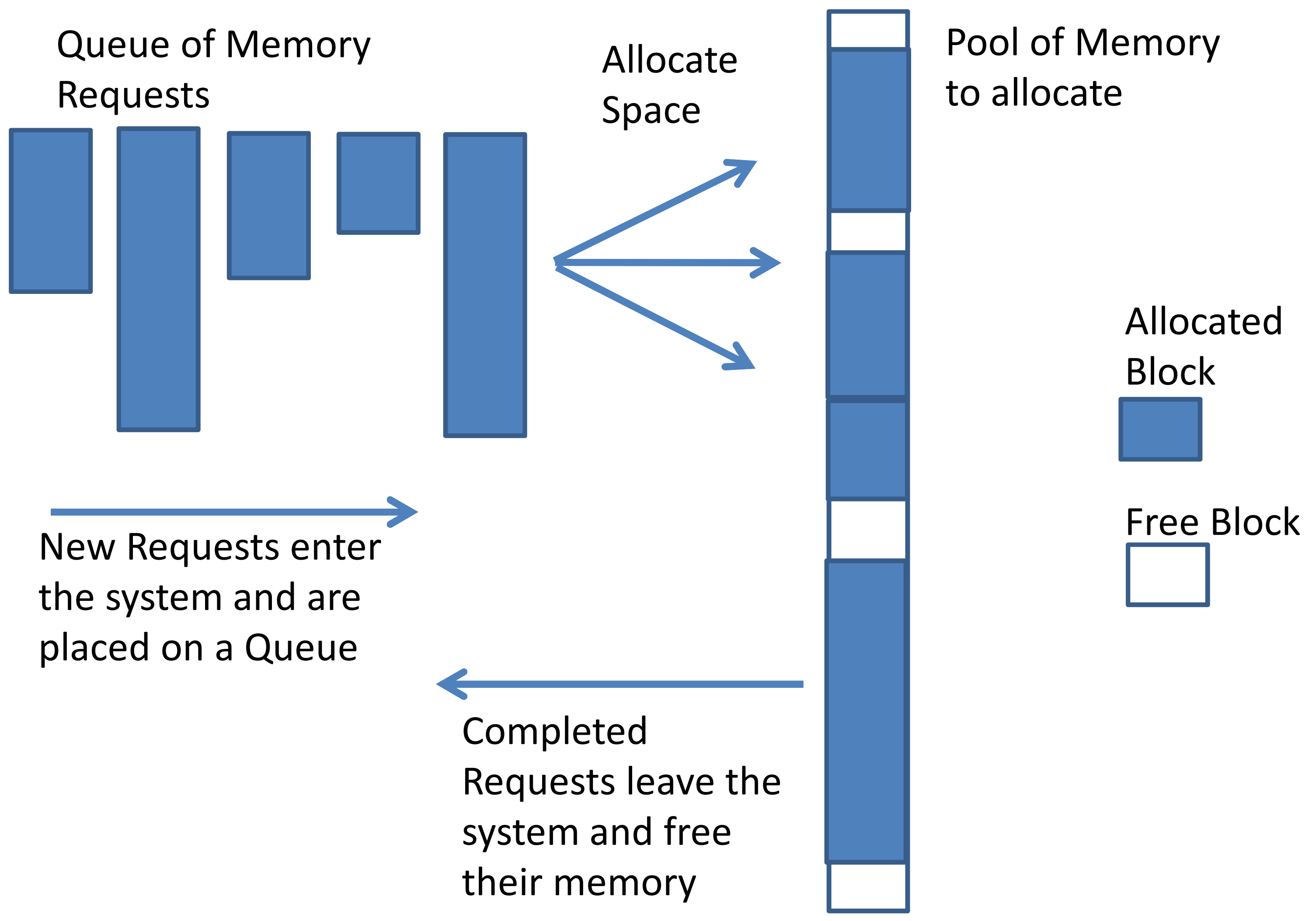
Fixed Partitioning

- Partition main memory into a set of non-overlapping memory regions called partitions.
- Fixed partitions can be of equal or unequal sizes.
- Leftover space in partition, after program assignment, is called internal fragmentation.

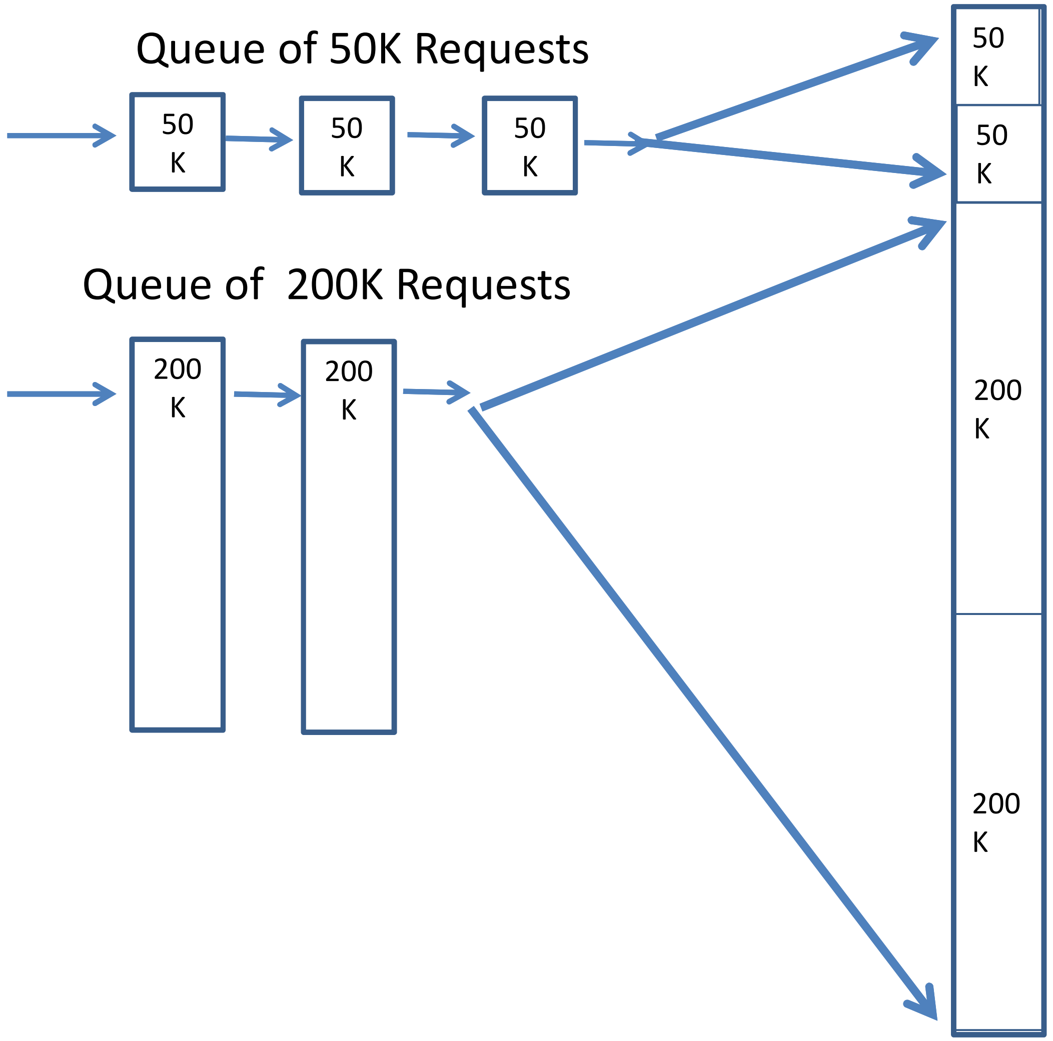


Equal-size partitions   Unequal-size partitions

Memory Management Design Problem


Queue of Memory Requests
Allocate Space
Pool of Memory to allocate
Allocated Block
Free Block
New Requests enter the system and are placed on a Queue
Completed Requests leave the system and free their memory

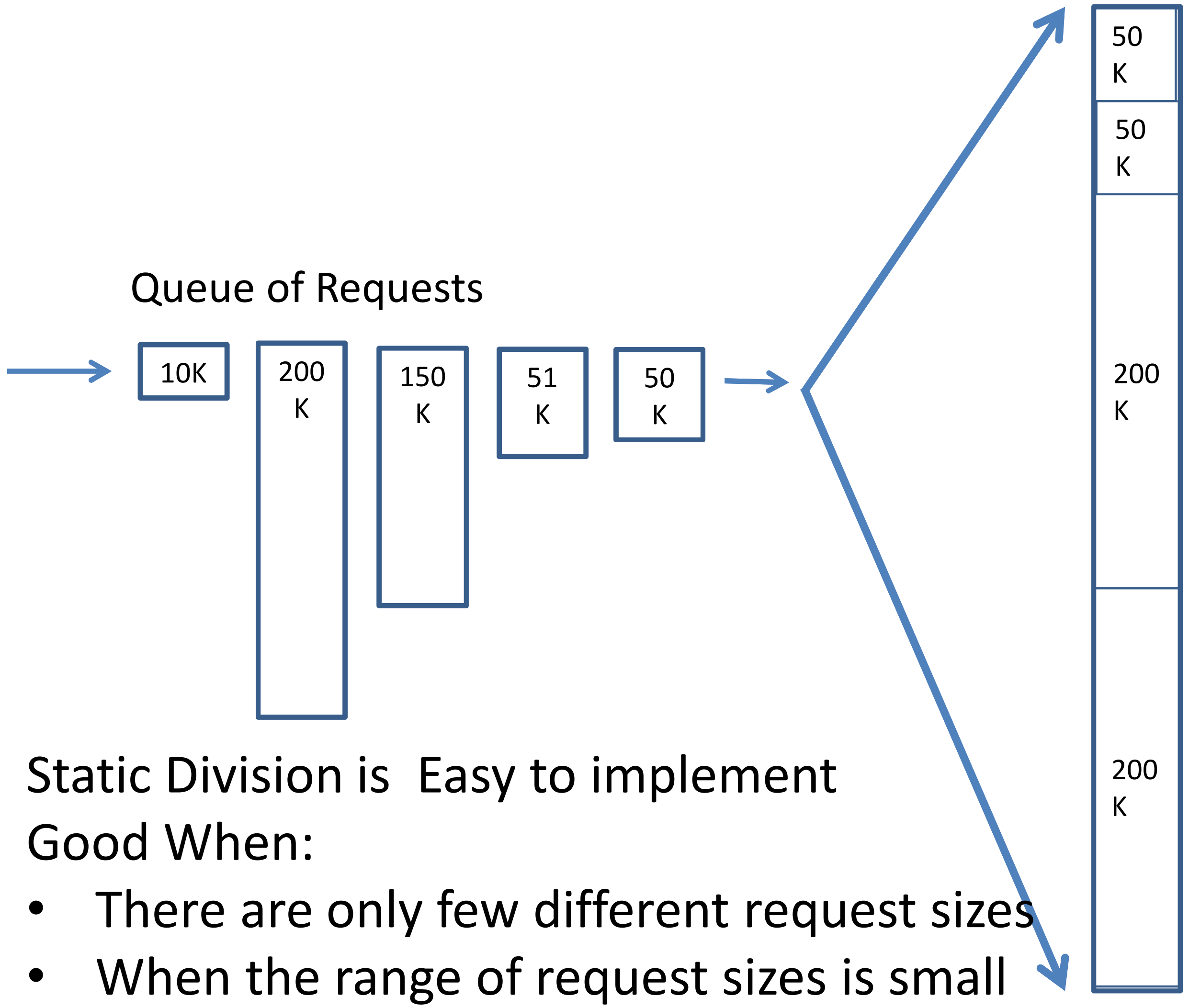
Static Division



Static Specification – memory is divided into partitions prior to processing of any jobs

Fixed-sized requests for memory
Statically divide the memory into blocks of that size and keep them on list for allocation.

Static Division cont'd



Variable-sized requests for memory

Static Division is Easy to implement
Good When:

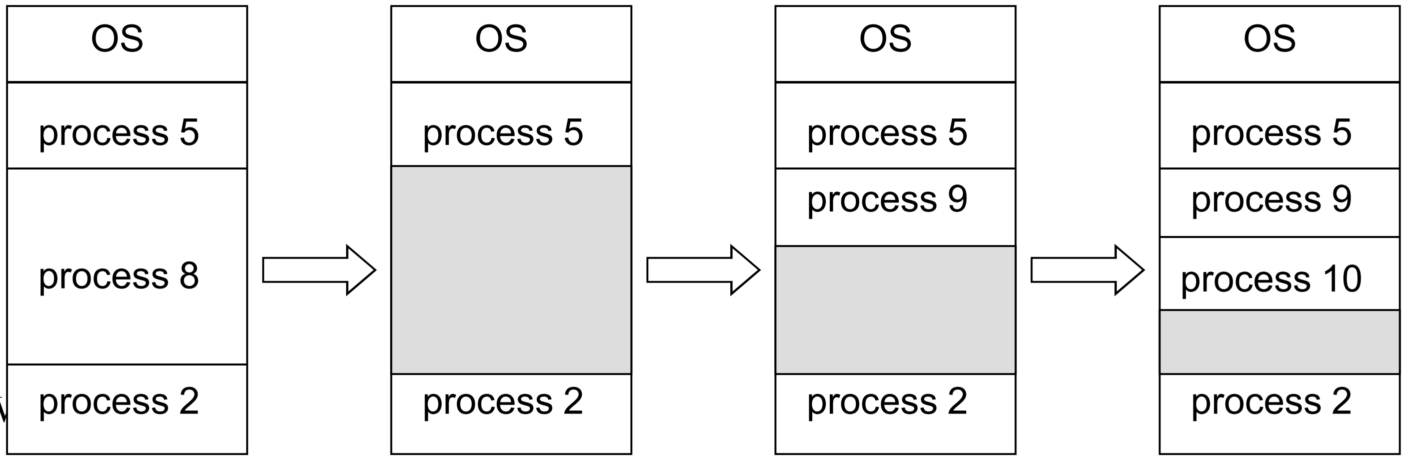
- There are only few different request sizes
- When the range of request sizes is small

Bad When:

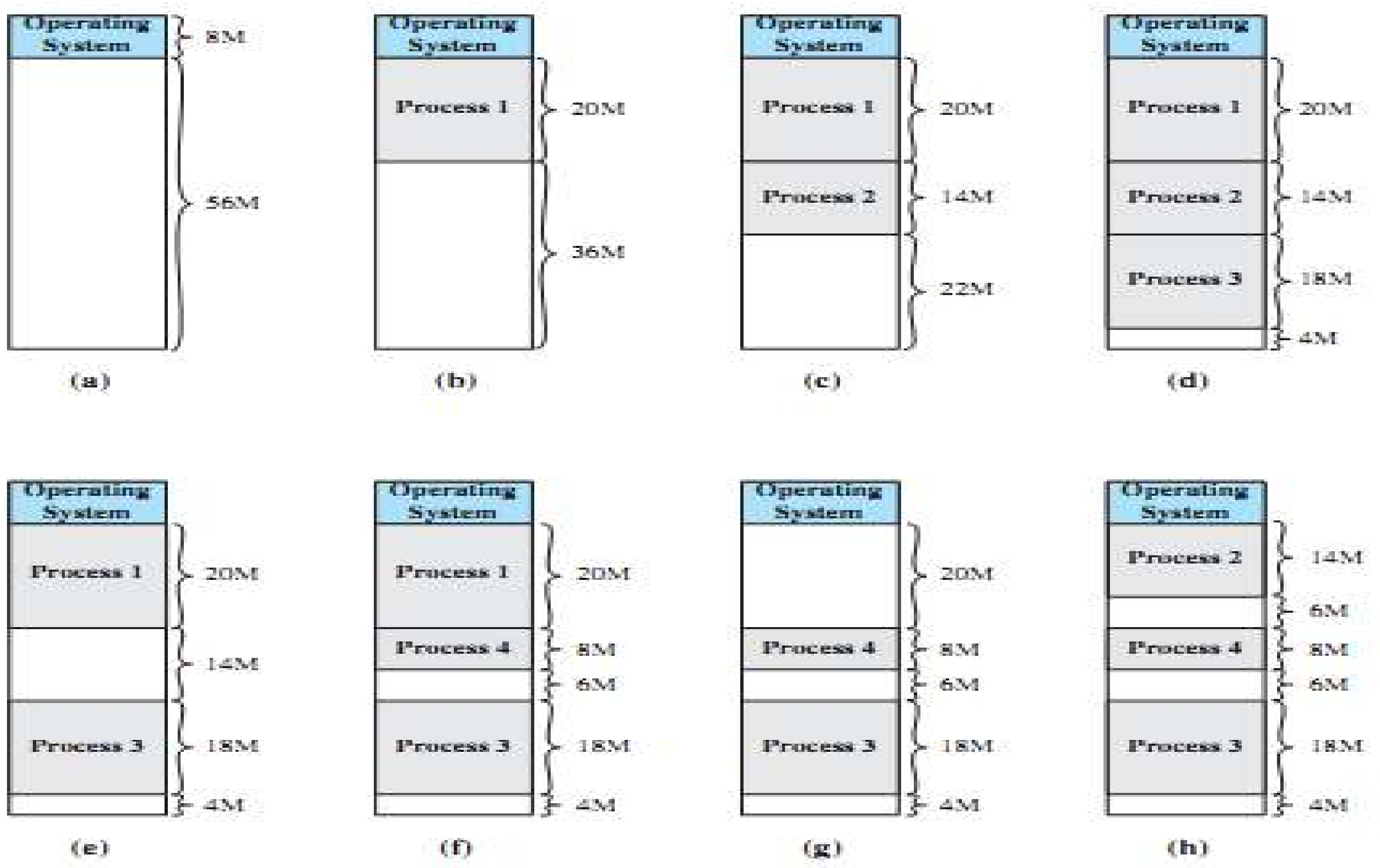
- The range of request sizes is large
- Some requests are very large

Variable Partitioning

- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- *Hole* – block of available memory; holes of various sizes are scattered throughout memory.
- Operating system maintains information about:
  a) allocated partitions b) free partitions (holes)



Variable Partitioning: example



A.

Internal/External Fragmentation

- There are really two types of fragmentation:

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

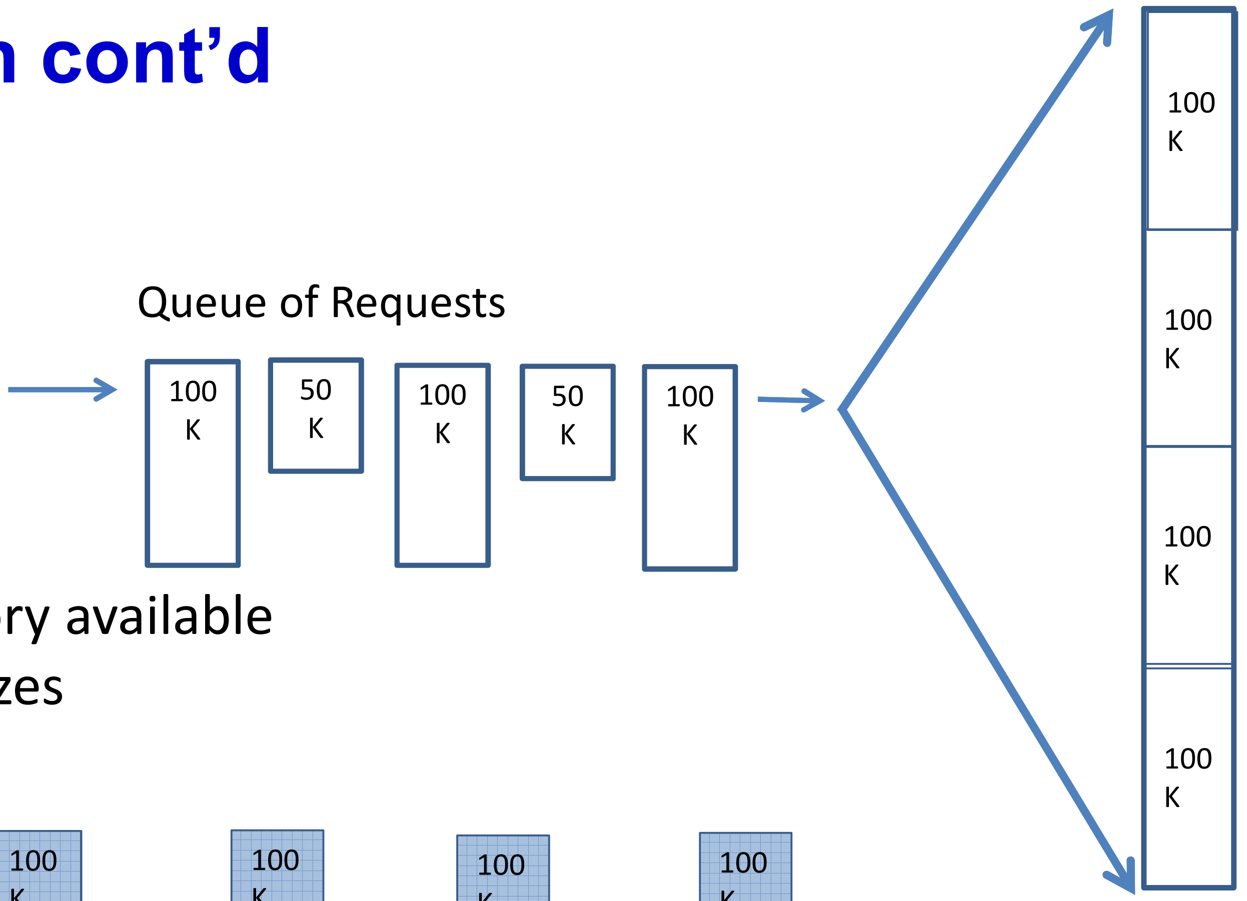
   External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block

Knuth's Buddy System

- A reasonable compromise to overcome disadvantages of both fixed and variable partitioning schemes.
- Memory allocated using **power-of-2 allocation;** Satisfies requests in units sized as power of 2.
- Memory blocks are available in size of 2^{K} where L <= K <= U and where:
  - 2^{L} = smallest size of block allocatable.
  - 2^{U} = largest size of block allocatable (generally, the entire memory available).
- A modified form is used in Unix SVR4 for kernel memory allocation.

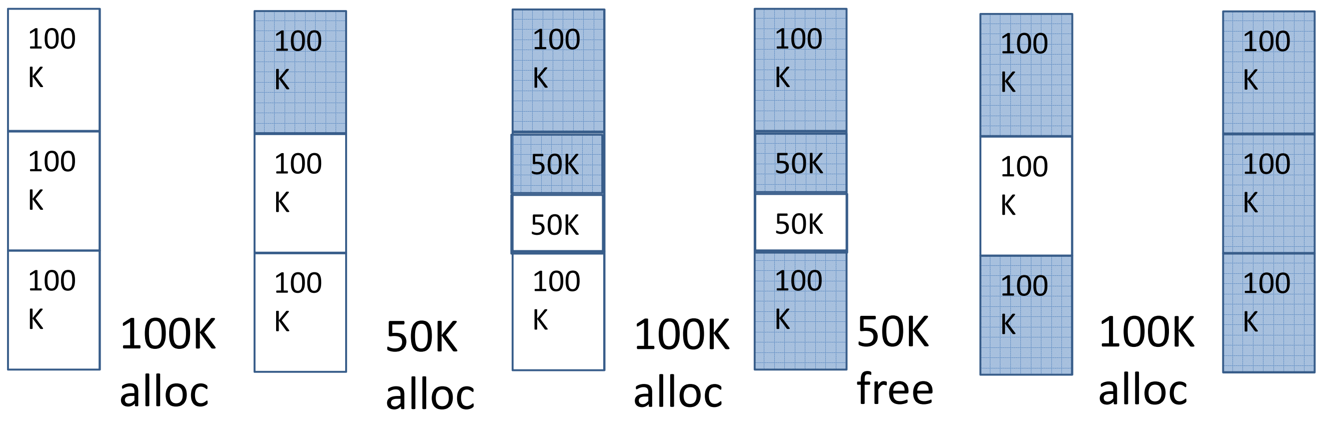
Static Division cont'd

Buddy Systems

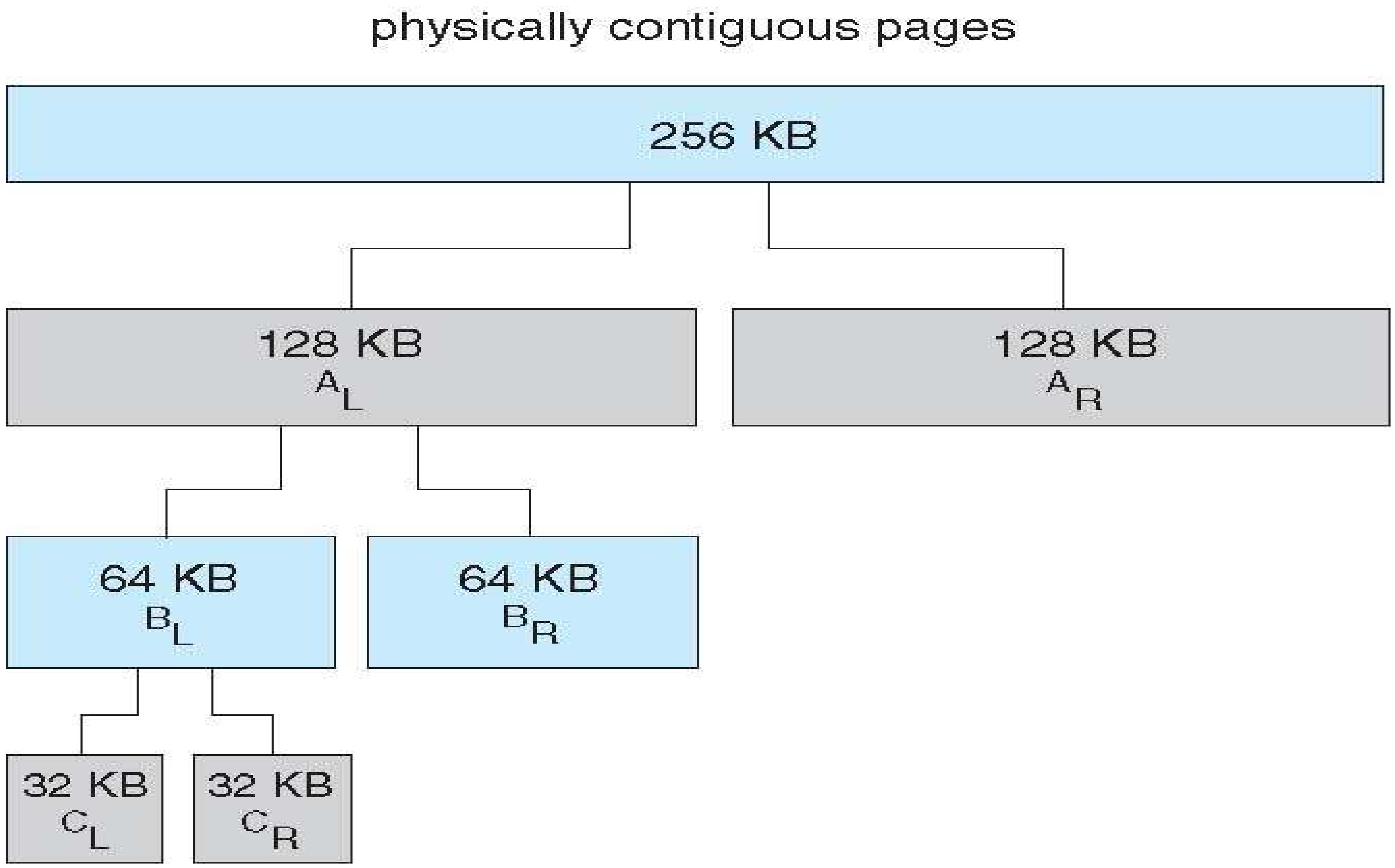


Problem:

- Large block of memory available
- Requests for small sizes

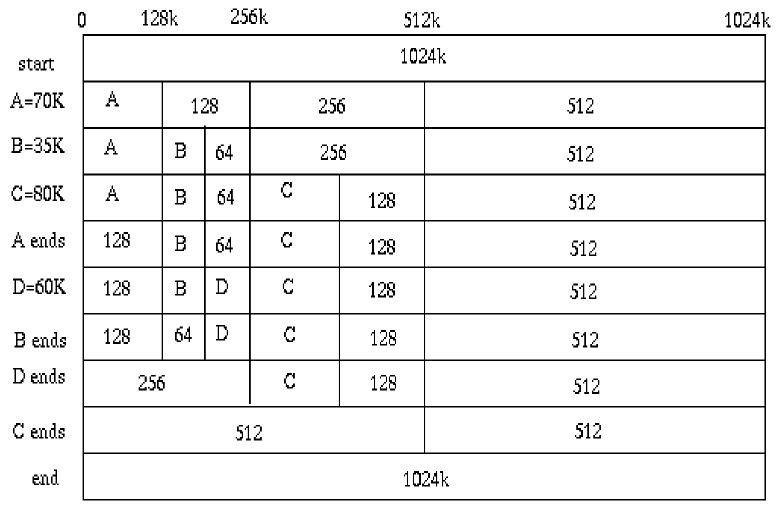


Buddy System Allocation



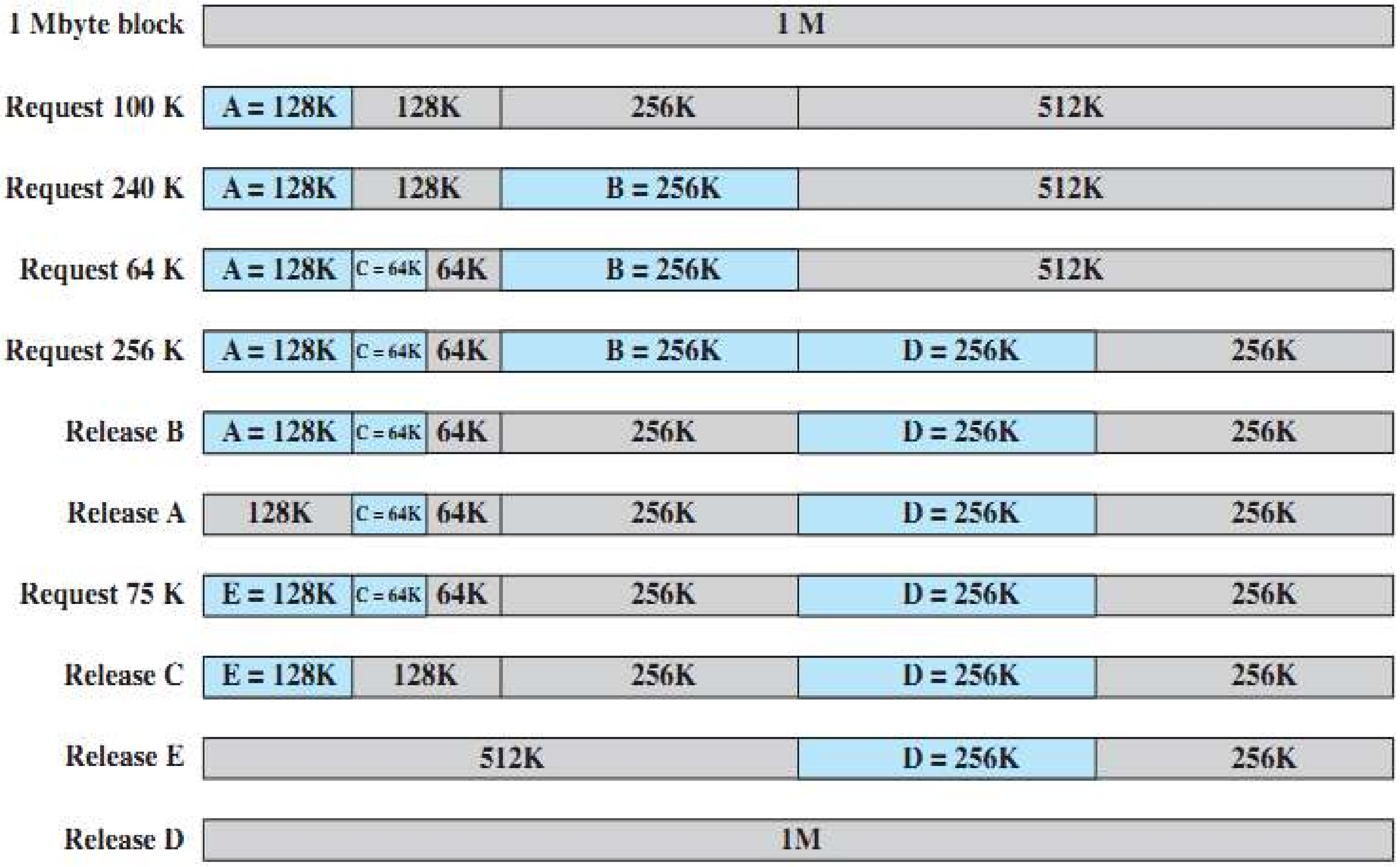
Buddy System cont’d

- Allows us to use larger blocks to satisfy smaller requests
- The generalization of above = buddy system
- Eg. We have 512,000 bytes.
- There will be free block list for each power of two i.e. 512,000 bytes, 256,000 bytes, 128,000 bytes, 64,000 bytes, 32,000 bytes, 16,000 bytes, 8K bytes, 4K bytes, 2K bytes and 1K bytes.



- They all start off empty except 512,000 bytes list = 1.
- When a request is received you round it up to next power of two and lookup the list.
- E.g. 25,000 bytes request will be satisfied by 32,000 bytes block.
- If that block list is empty, then try the next larger power of two and divide a free block into two.
- If you fail to find a block in the list, keep going up the list of or until you run out of lists, in which case it must wait.
- Adv: Handles problem of larger requests
- Disadv: waste memory by rounding to next higher

Example of Buddy System


1 Mbyte block | 1 M
Request 100 K | A = 128K | 128K | 256K | 512K
Request 240 K | A = 128K | 128K | B = 256K | 512K
Request 64 K | A = 128K | C = 64K | 64K | B = 256K | 512K
Request 256 K | A = 128K | C = 64K | 64K | B = 256K | D = 256K | 256K
Release B | A = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Release A | 128K | C = 64K | 64K | 256K | D = 256K | 256K
Request 75 K | E = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Release C | E = 128K | 128K | 256K | D = 256K | 256K
Release E | 512K | D = 256K | 256K
Release D | 1M

A.

Dynamics of Buddy System (1)

- We start with the entire block of size 2^{U}.
- When a request of size S is made:
  - If $2^{U-1} < S <= 2^{U}$ then allocate the entire block of size 2^{U}.
  - Else, split this block into two buddies, each of size 2^{U-1}.
  - If $2^{U-2} < S <= 2^{U-1}$ then allocate one of the 2 buddies.
  - Otherwise one of the 2 buddies is split again.
- This process is repeated until the smallest block greater or equal to S is generated.
- Two buddies are coalesced whenever both of them become unallocated.

Dynamics of Buddy System (2)

- The OS maintains several lists of holes:
  - the i-list is the list of holes of size 2^{i}.
  - whenever a pair of buddies in the i-list occur, they are removed from that list and coalesced into a single hole in the (i+1)-list.
- Presented with a request for an allocation of size k such that 2^{i-1} < k <= 2^{i}:
  - the i-list is first examined.
  - if the i-list is empty, the (i+1)-list is then examined ...

Comments on Buddy System

- Mostly efficient when the size M of memory used by the Buddy System is a power of 2:
  - M = 2^{U} “bytes” where U is an integer.
  - then the size of each block is a power of 2.
  - the smallest block is of size 1.
- On average, internal fragmentation is 25%
  - each memory block is at least 50% occupied.
- Programs are not moved in memory:
  - simplifies memory management.

Static Division cont’d

Power of Two Allocation

- Simpler variation of Buddy System
- Similar to buddy system except that you do not split and combine blocks
- If there are no blocks of particular size, wait for one to get freed or else use larger block (without splitting)
- This is not a good system for dynamic memory allocation

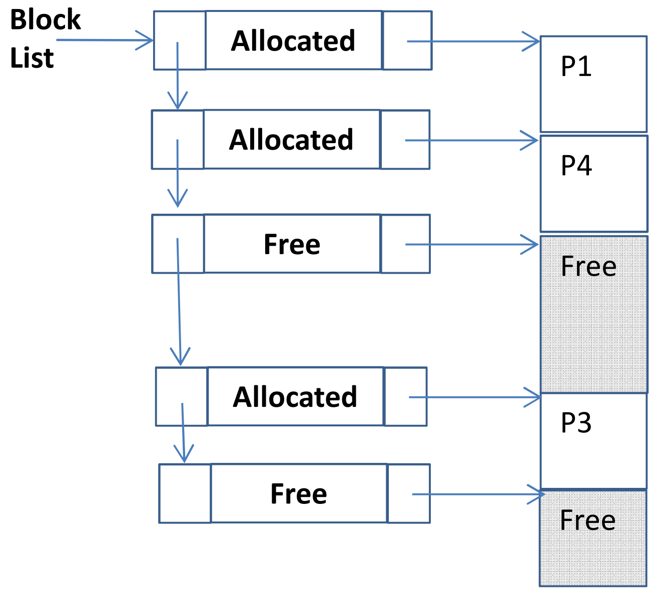
Memory Allocation

- Large memory in system
- Process requests memory and Returns after use
- **Allocator** has to keep track of allocated and free blocks
  - **How does it keep track of the blocks**
  - **Which block should it allocate when a request comes in**

How to Keep Track of the Blocks

The List Method

- Linked list of all the blocks of memory (block list)
- When we need to allocate we go thru the block list to find suitable free block
- Once we identify the free block we can either allocate entire block or divide as per requested size and create free block from remaining
- When a block is freed, we change its status on the free list and combine with surrounding free blocks if any



Where is the block list kept?

Part of the memory is reserved for the block list and the rest of the memory is available for allocation to satisfy requests for memory.

Which Free Block to Allocate

How to satisfy a request of size *n* from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough; this avoids going thru the entire list.

- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole

- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

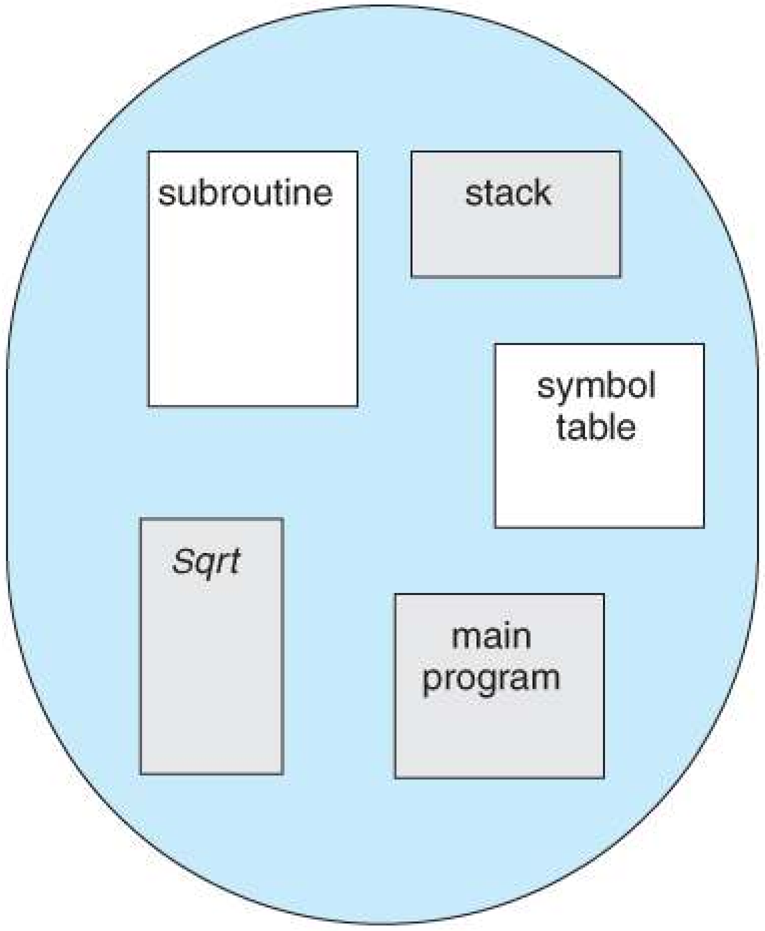
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Concerns

- The methods discussed thus far are not used in modern OS
  - The algorithms mentioned are too slow
  - Modern OS have different requirements / problem to be solved.
  - Paging is more viable

Segmentation

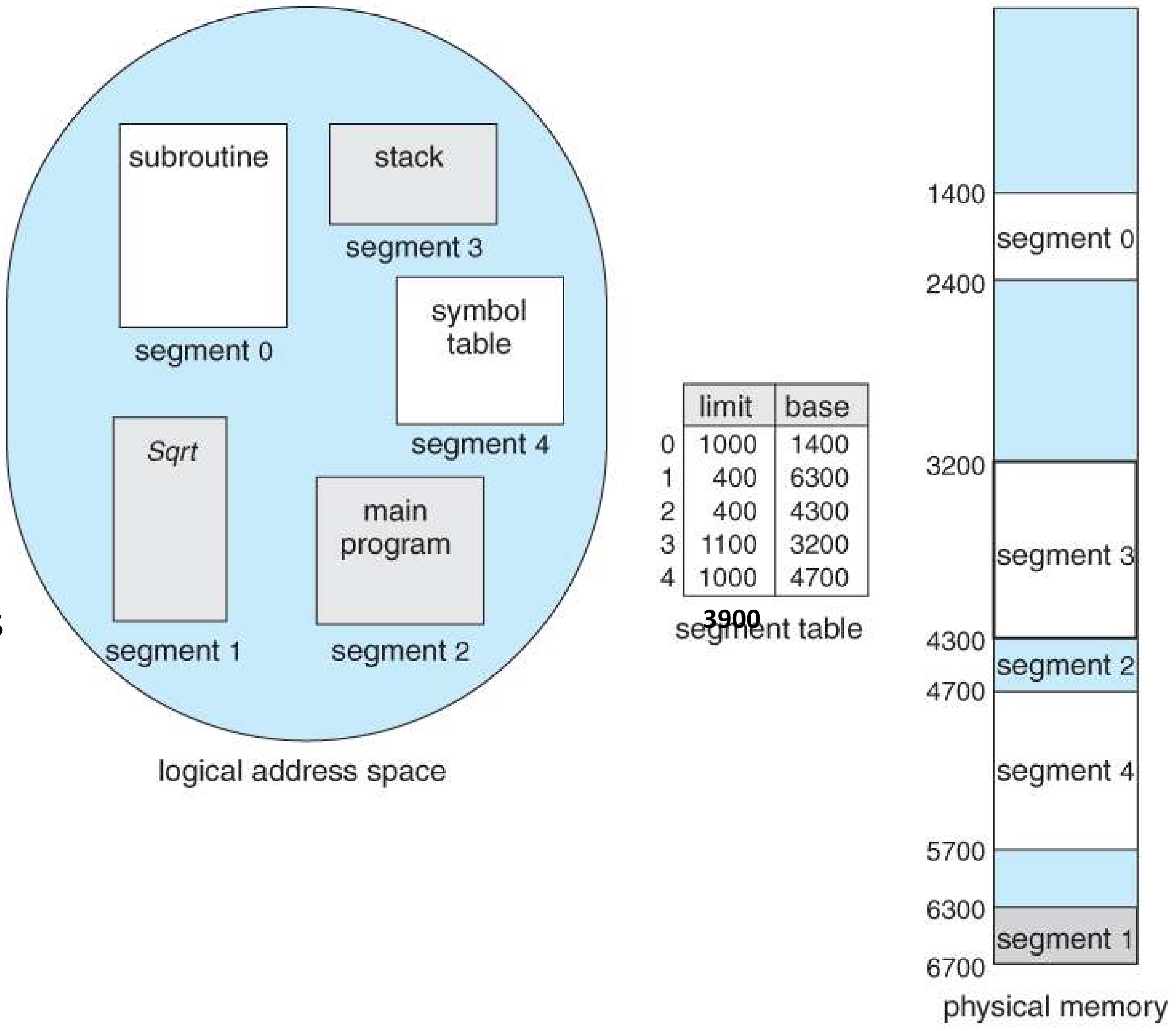
- Most **users** (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a **particular use**, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.



logical address

Segmentation

- A ***segment table*** maps segment-offset addresses to physical addresses
- simultaneously checks for invalid addresses
- each segment is kept in contiguous memory and may be of different sizes
- segmentation can also be combined with paging

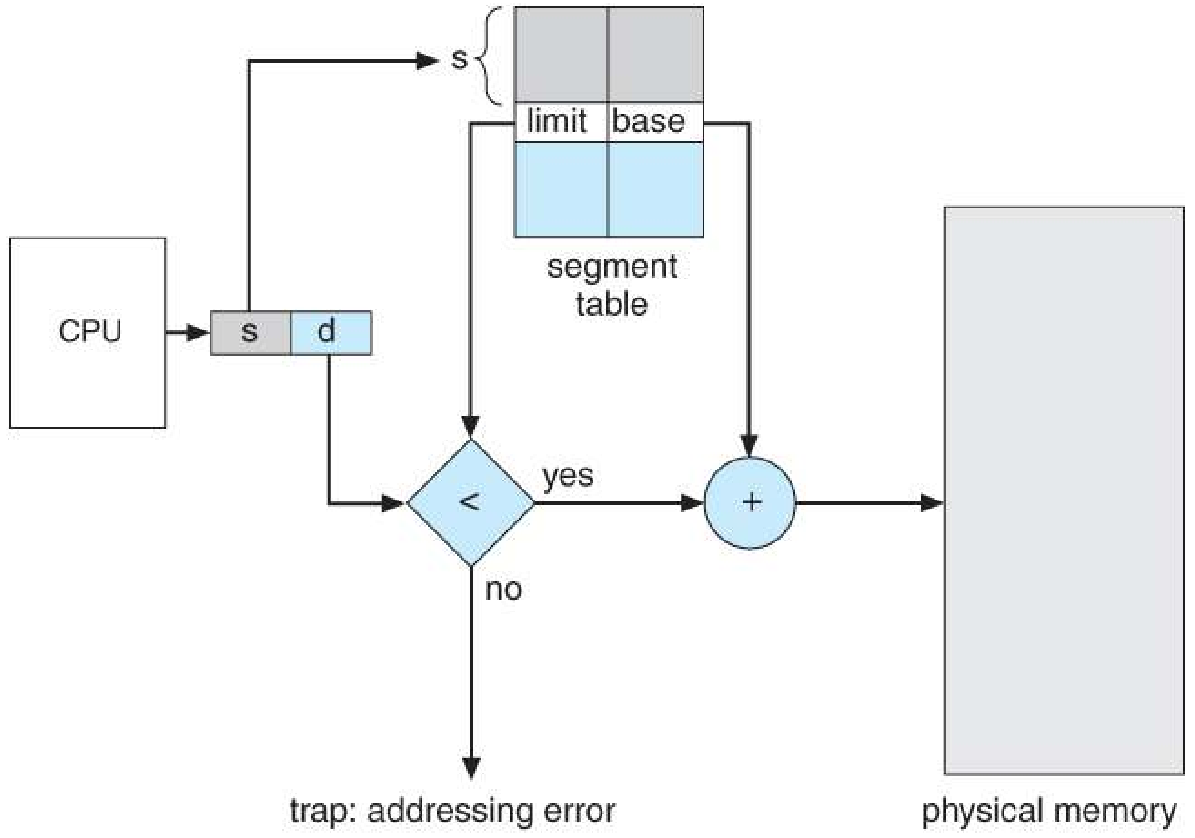


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

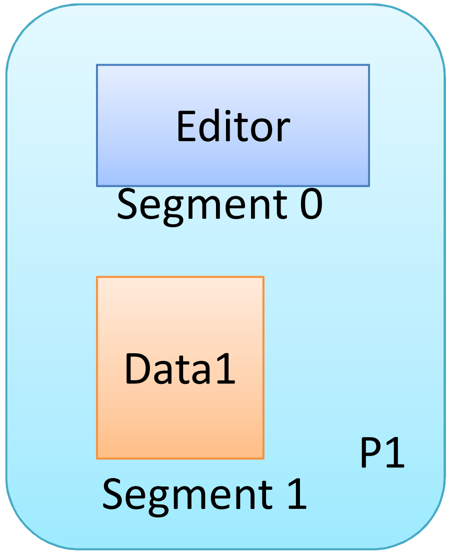
segment table

Segmentation - Hardware Support

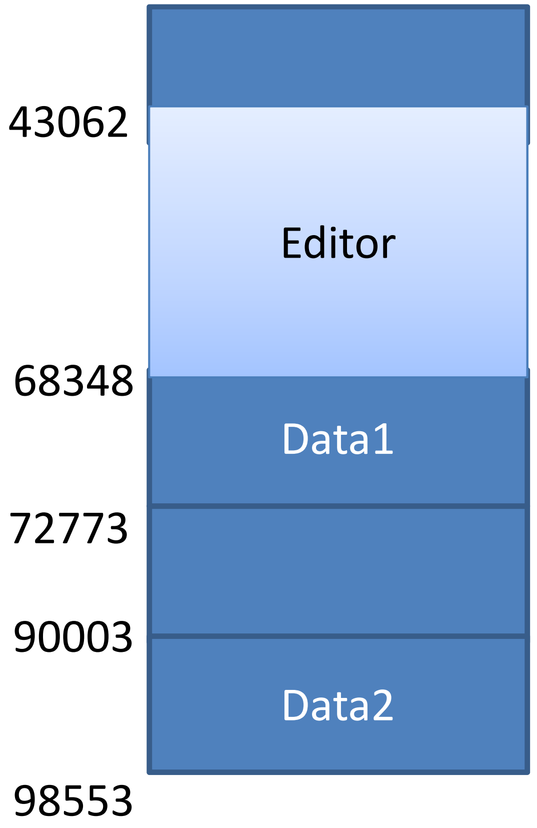
Segment Table

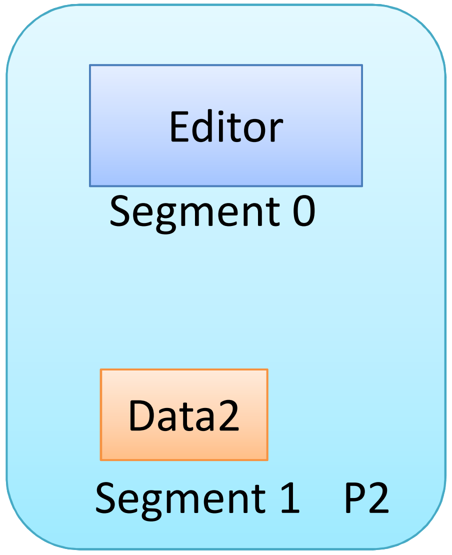


Sharing



Limit	Base
25286	43062
4425	68348





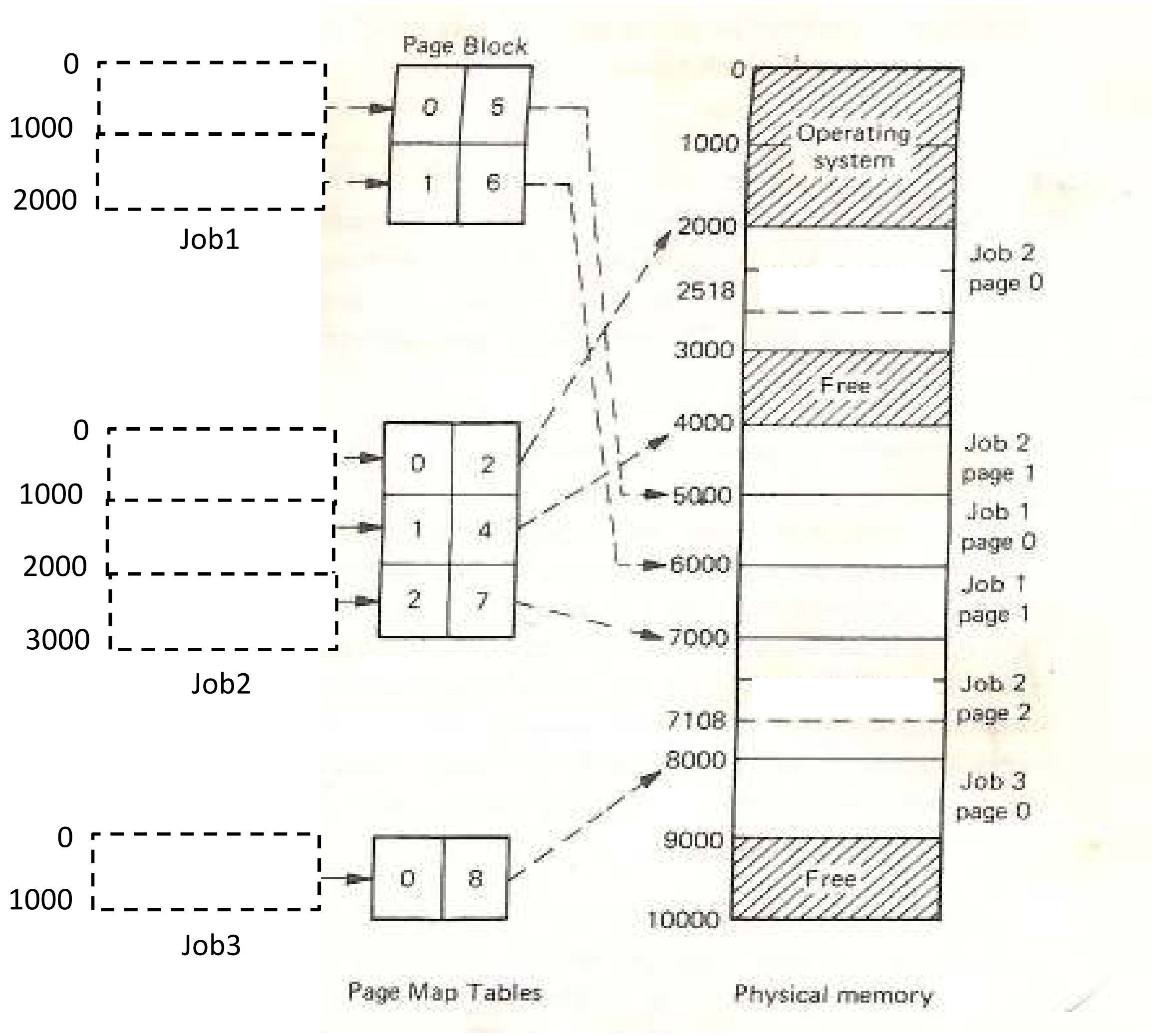
Limit	Base
25286	43062
8850	90003

Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in ***equal sized blocks*** known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.
- 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.

Hardware Support

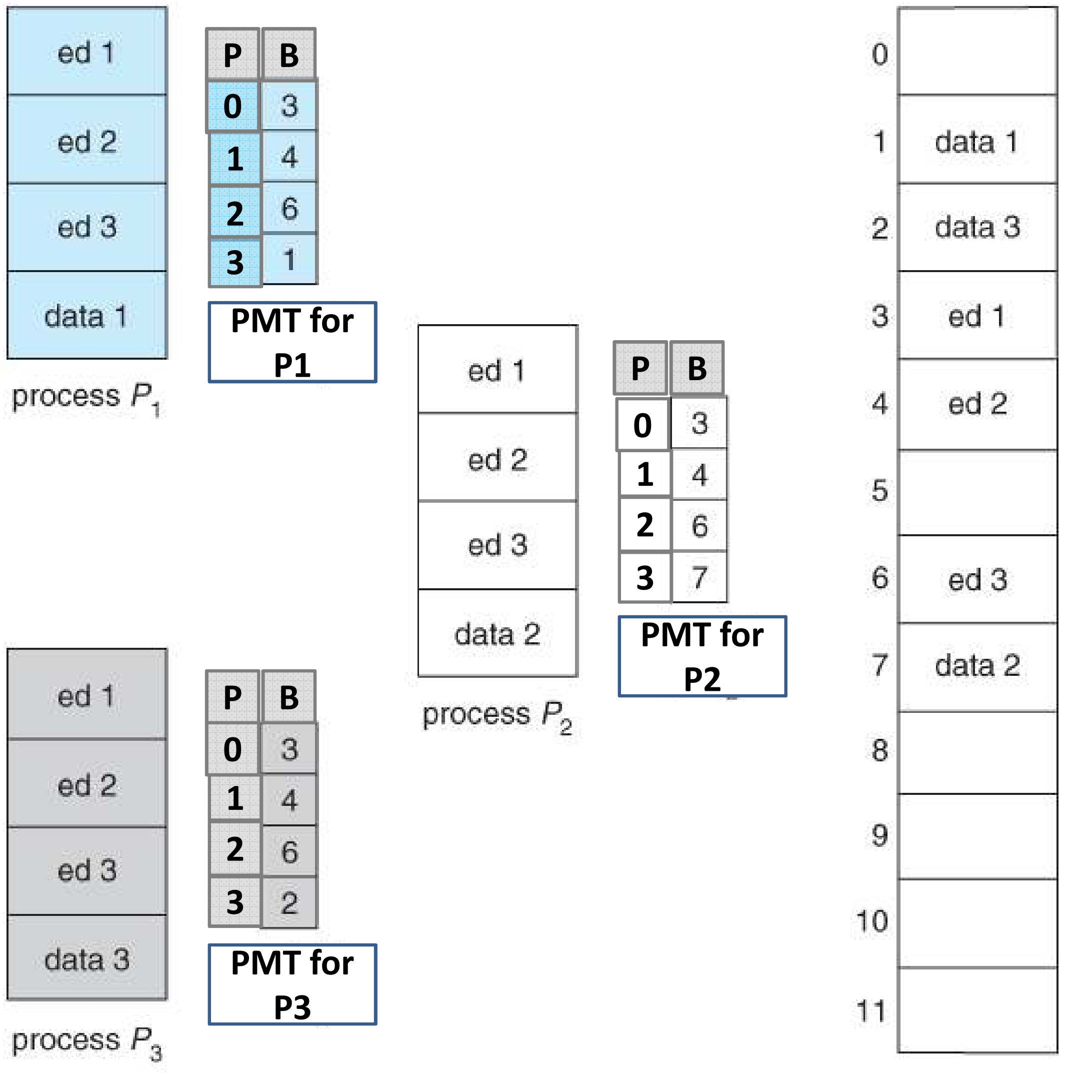
- Using the **Page Map Table**, the page number is replaced by the block number to produce the resultant physical memory address.
- Page size is usually chosen to be a power of two 1024 (1K), … 4096(4K)



Shared Pages

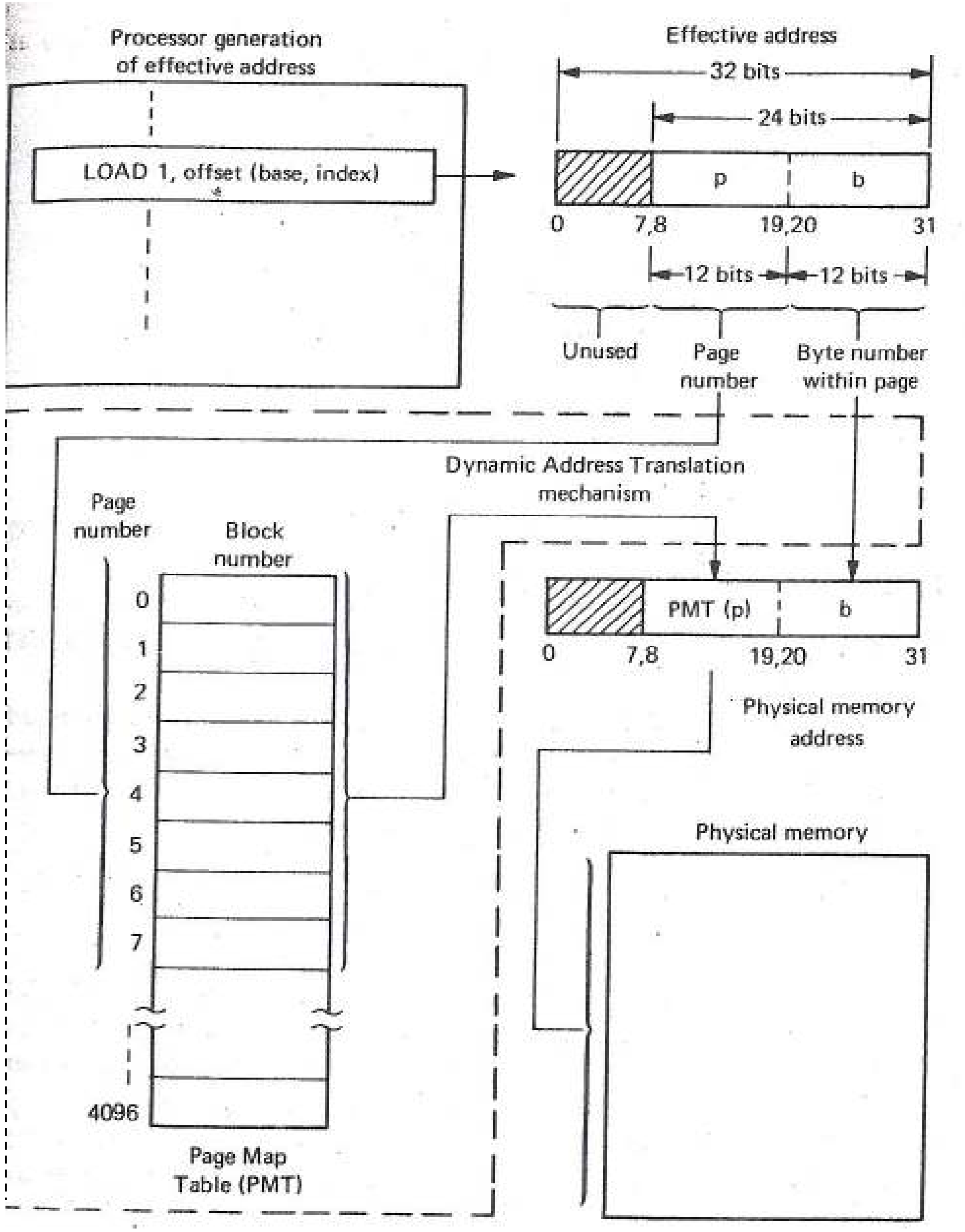
- Paging systems can make it very easy to share blocks; by simply referring same blocks
- It means the code can be shared by multiple processes
- E.g. three different users are running the editor simultaneously, but the code is only loaded into memory one time

Reentrant code - meaning it does not modify the code while it runs, making it safe to re-enter it (e.g. DLL)



Determining Physical Address

- A jobs address space can be 16 Million Bytes (4096 4K pages).
- If the PMT were 4096 entries long , each entry 2 bytes, it would require 8196 bytes.
- The length of PMT should be only as long as needed.



PMTAR

- The length of PMT should be only as long as needed; saves memory
- PMT should be stored in main memory at a known location to the hardware
- However, each time the processor switches job (multiprogramming env) the entire PMT should be replaced! ⬇
- Instead the PMT is allowed to be stored at any location in memory, the specific location being indicated by a special PMT Address Register (PMTAR).
- When the process is switched to a new job, only the PMTAR has to be changed to indicate the location of new job's PMT.
- Process switching is fast, because only the single register needs to be changed ⬆

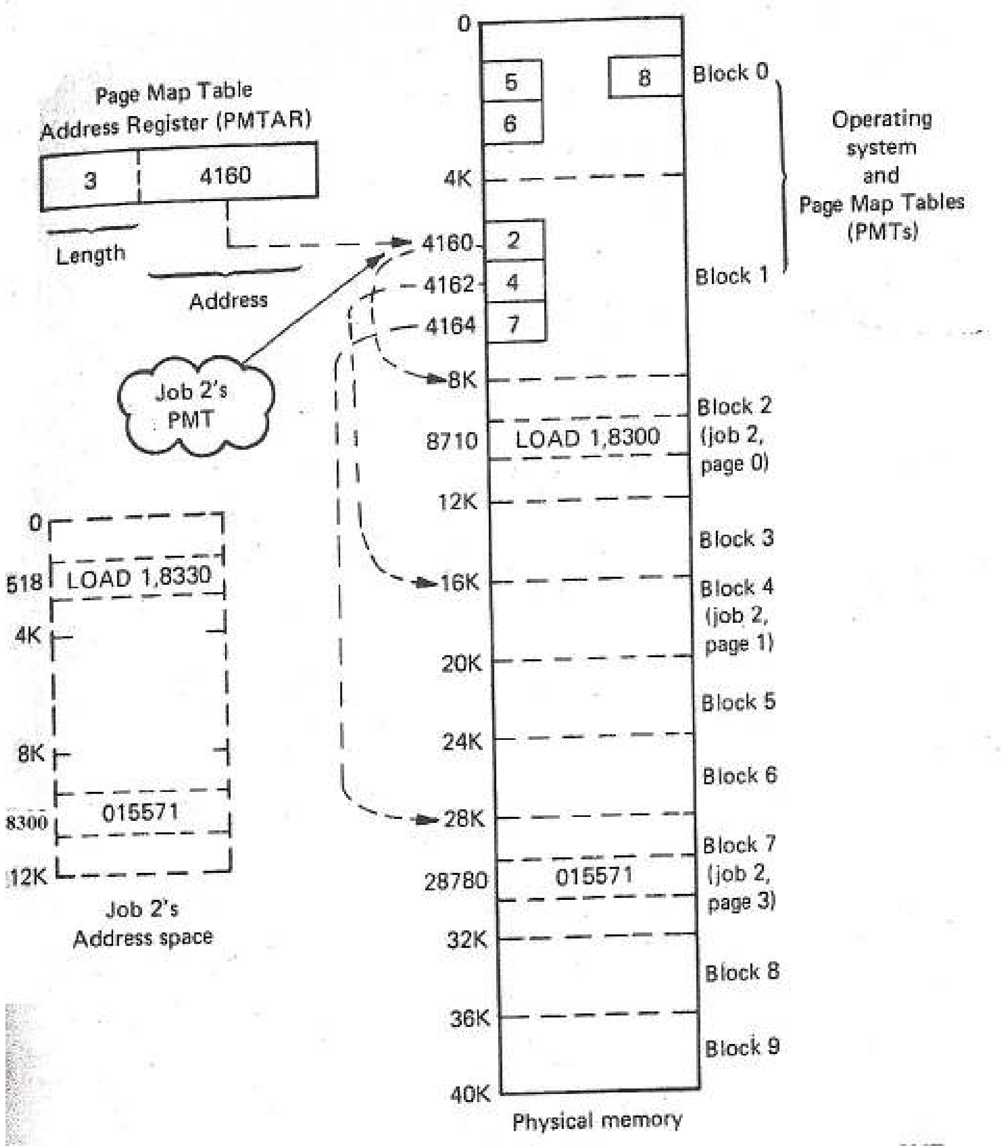
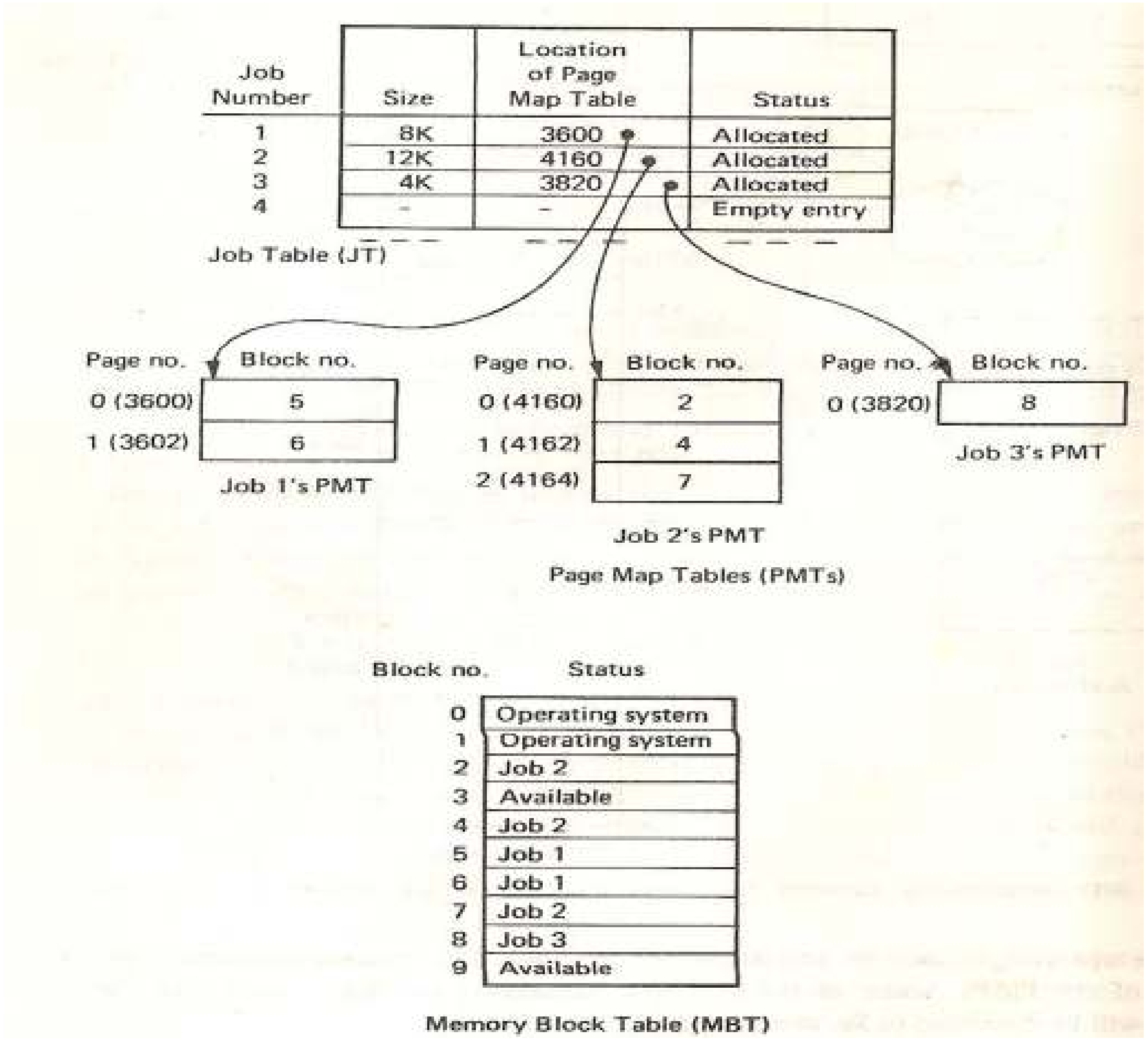


Figure 3-17 Relationship between the pages, the blocks, the PMTAR, and the PMTs

Paged Memory Management



Job Number	Size	Location of Page Map Table	Status
1	8K	3600	Allocated
2	12K	4160	Allocated
3	4K	3820	Allocated
4	-	-	Empty entry

Job Table (JT)

Page no.	Block no.
0 (3600)	5
1 (3602)	6

Job 1's PMT

Page no.	Block no.
0 (4160)	2
1 (4162)	4
2 (4164)	7

Job 2's PMT

Page no.	Block no.
0 (3820)	8

Job 3's PMT

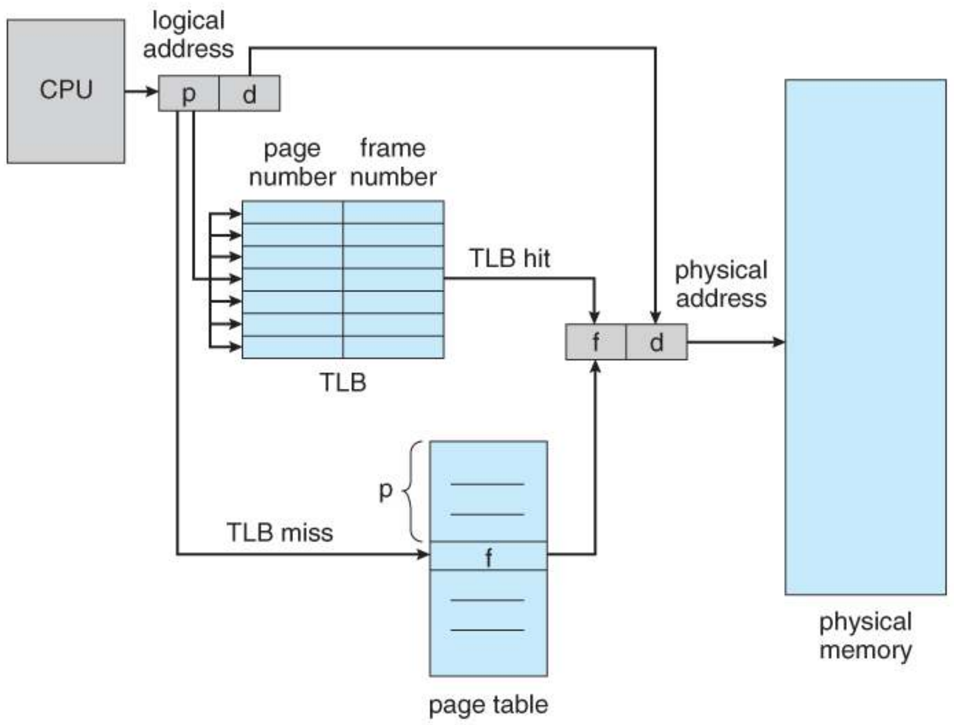
Page Map Tables (PMTs)

Block no.	Status
0	Operating system
1	Operating system
2	Job 2
3	Available
4	Job 2
5	Job 1
6	Job 1
7	Job 2
8	Job 3
9	Available

Memory Block Table (MBT)

Translation Look-aside Buffer

- However memory access just got half as fast, because every memory access now requires ***two*** memory accesses - One to fetch the block number from PMT and then another one to access the desired memory location.
- The solution - a very special high-speed memory device called the ***translation look-aside buffer, TLB.***
- The TLB is on the cache - very expensive, however, and therefore very small. (Not large enough to hold the entire page table.)
- Addresses are first checked against the TLB, and if the info is not there (a ***TLB miss***), then the frame is looked up from main memory and the TLB is updated.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.
- If the TLB is full, then replacement strategies (eg ***LRU***, random) are applied.



Paging : Adv : Disadv

- **Advantages**:
  - Avoids fragmentation
  - Provide more memory that can be used for more jobs
  - Higher degree of multiprogramming results in increased processor and memory utilization
  - Compaction overhead in relocatable partition schemes is eliminated
- **Disadvantages**:
  - Page address mapping hardware usually increases cost of computer and also slows down processor
  - Memory is used to store PMT; processor time (overhead) must be expended to maintain and update these tables
  - Though fragmentation is eliminated, Internal Fragmentation / Page Breakage does occur.

	What is the swap space in the disk used for?
A	Saving temporary html pages
B	Saving process data
C	Storing the super-block
D	Storing device drivers

	Increasing the RAM of a computer typically improves performance because:
A	Virtual memory increases
B	Larger RAMs are faster
C	Fewer page faults occur
D	Fewer Segmentation Faults

What are the types of memory in hierarchy?

What is compaction?
Compaction is a process in which the free space is collected in a large memory chunk to make some space available for processes.

Runtime mapping from logical to physical memory is done by
MMU, CPU, PCI, none of above

Swapping

moving out some blocked process from the main memory to the secondary memory(hard disk) and moving in a ready process

Example

- Disk Storage: best access speed from a solid state drive is about 600 MB/second
- Main Memory: Best access speed is around 10 GB/second
- Cache: Best access speed is around 100 GB/second

Try:

- A user process occupies 10 MB
- Transfer rate for the backing store is 40 MB / second
- Then it how much time would it take to transfer data?
- 1/4 second (250 milliseconds) for data transfer.
- Recognize swapping involves moving old data out as well as new data in
- ~half a second (500milliseconds)

Example Cont'd

- Do modern OS use swapping

  No, more sophisticated paging

- Is swapping supported on mobile systems?

  No, mobile typically use flash memory, which is small, low bandwidth, and Flash memory can only be written to a limited number of times before it becomes unreliable

- Where is the OS stored?

  OS resides in Low Memory and user processes are held in high memory.

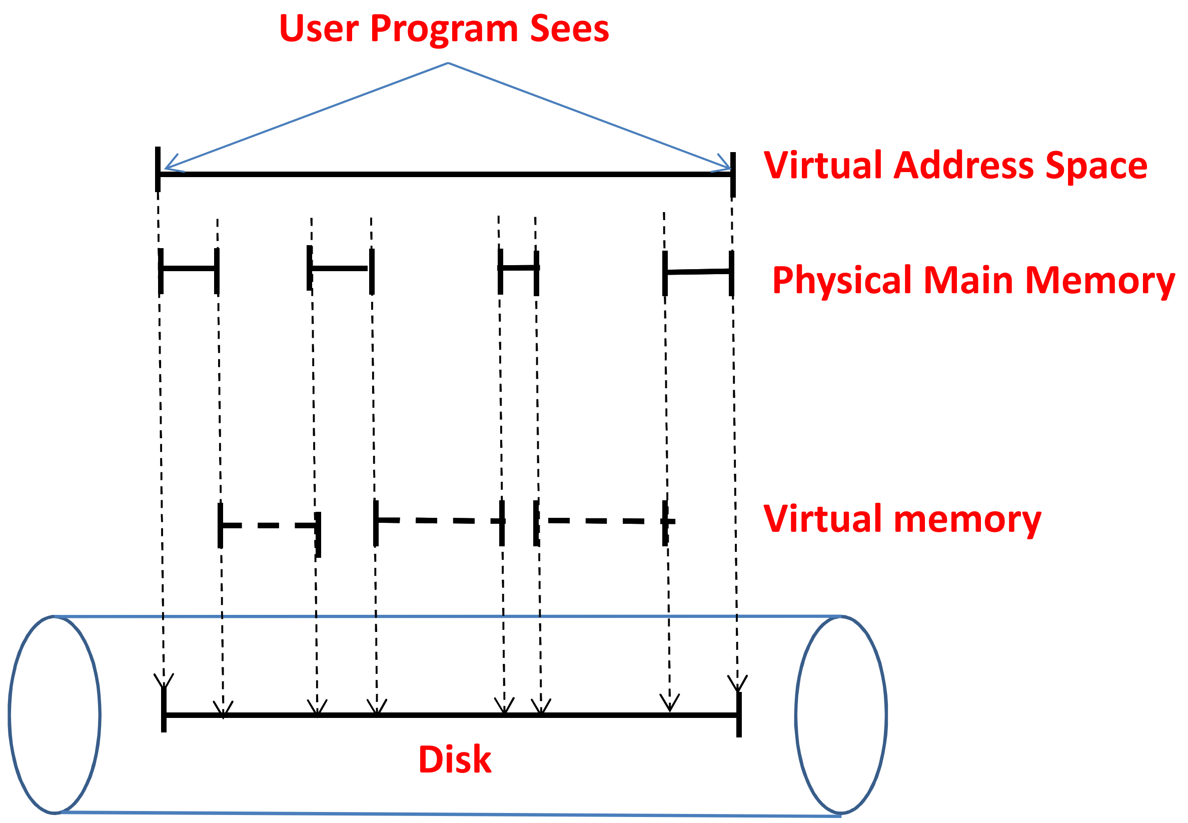
Try

- Consider we have 500K Bytes of Memory
- Requests for Memory are as – 70K, 200K, 150K, 51K, and 50K
- Write a allocator based on Worst Fit Algorithm

Virtual Memory

Virtual Memory

- Memory that appears to exist as main storage (VV high capacity)
- although most of it is supported by data held in secondary storage
- Maintained by transfer between the two being made automatically as required.
- Benefit is that programs could be written for a much larger address space , higher degree of multi programming



Software Mechanism

- The OS will reserve an area on the disk to hold the image of the virtual address space of each process.
- This is called ***swap area***.
- Some (or all)of the pages in the swap area will be in memory at any one time.

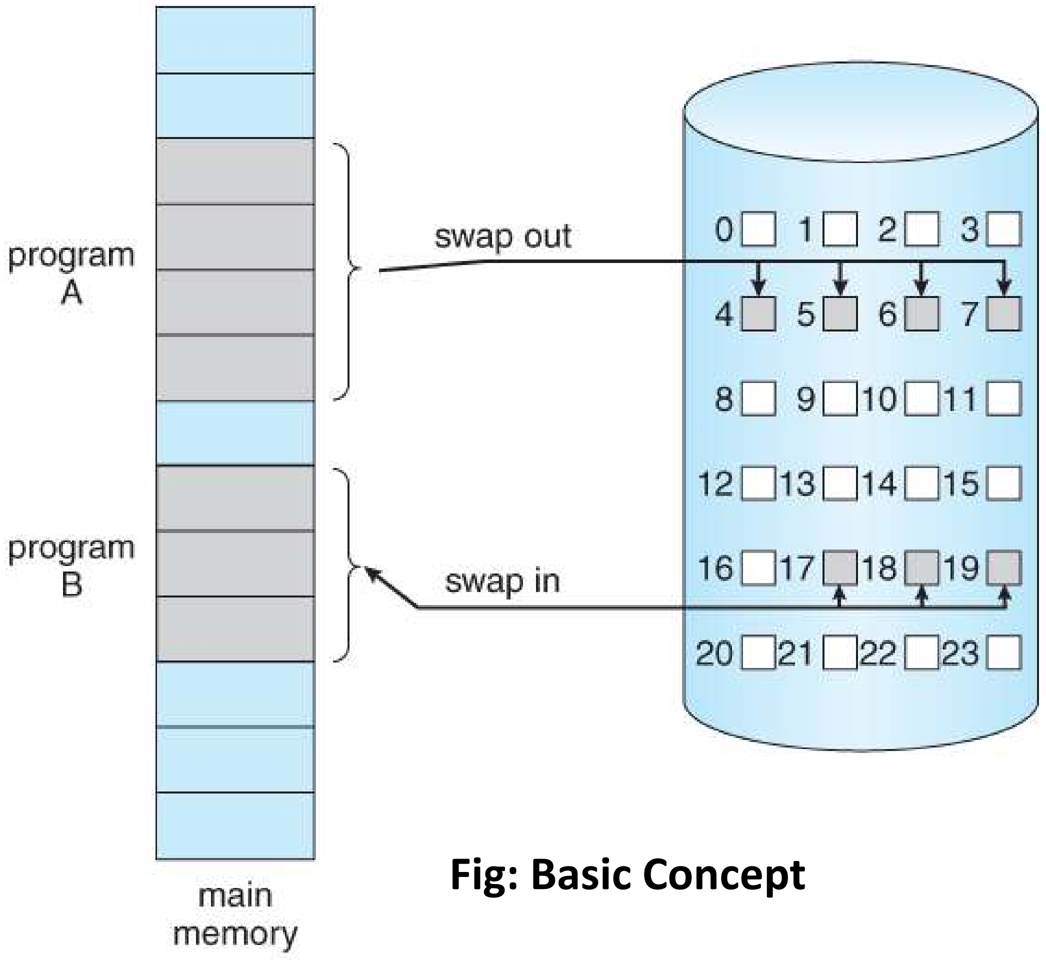


Fig: Basic Concept

Memory Management

1. Keeping track of status

    Page Map – Memory Block Tables

2. Determining who gets memory – largely decided by job scheduler

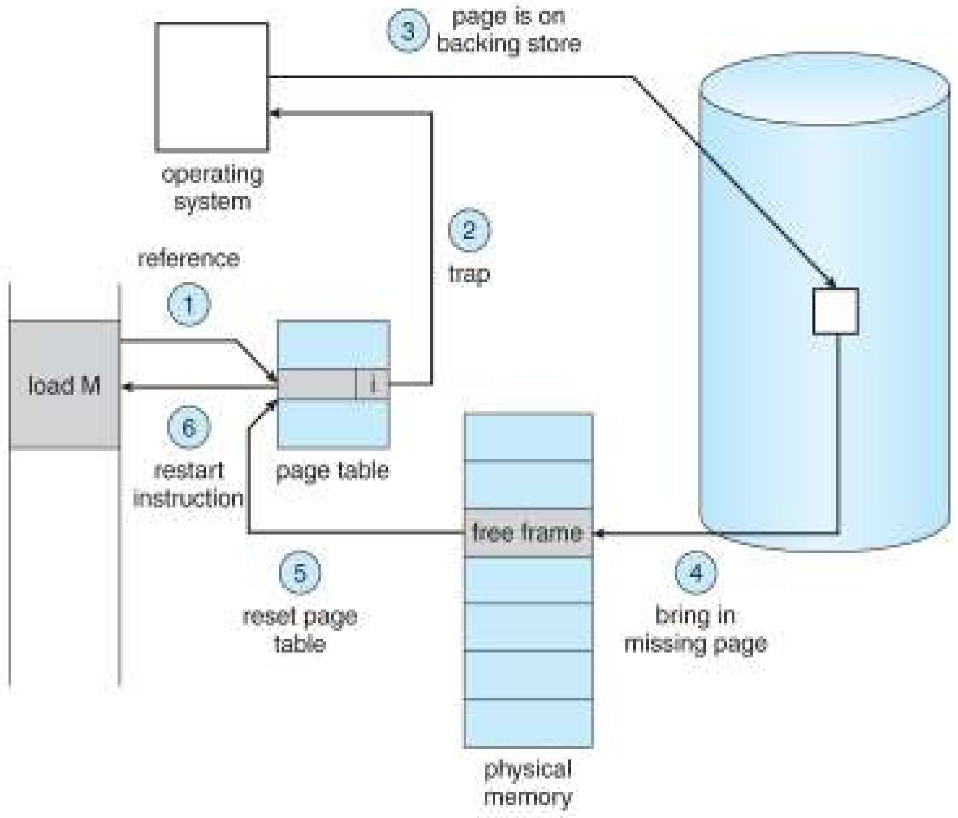
3. Allocation – all pages of the job must be loaded into assigned blocks and appropriate entries made in Page Map Table and Memory Block Table

4. Deallocation – when job is done, blocks must be returned to free status by adjusting entries in block table

Algorithm

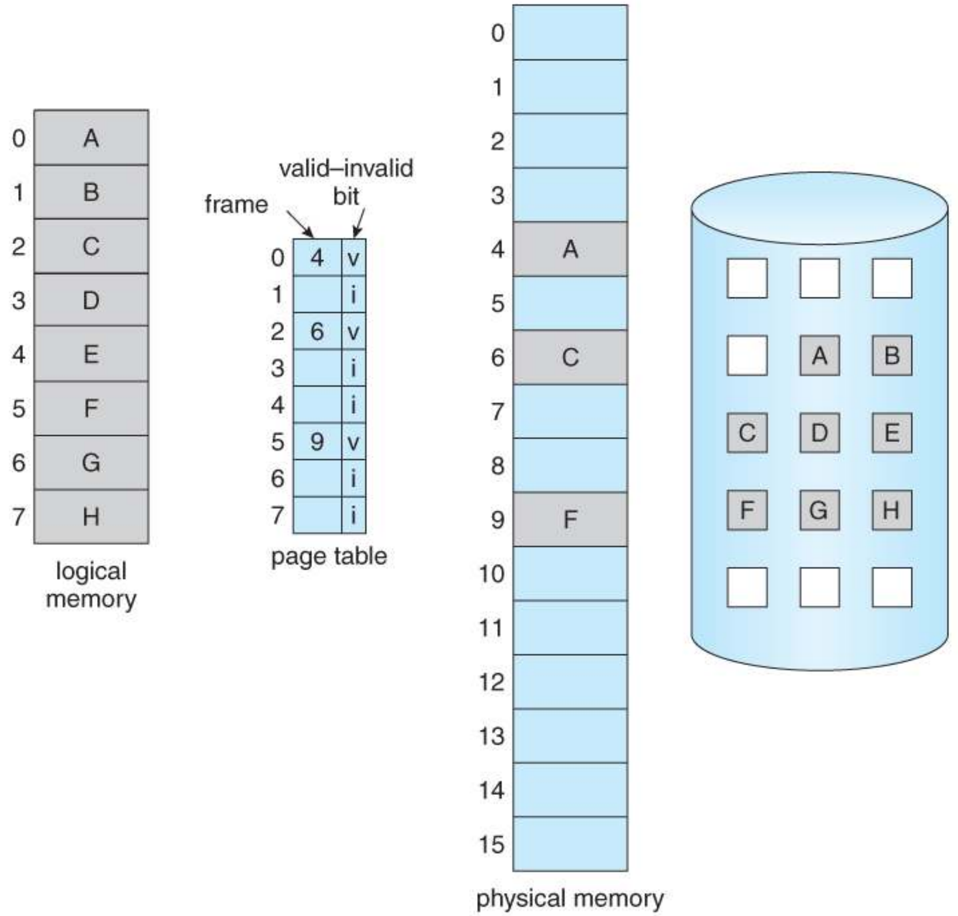
When a process tries to access a page

1. We check an internal table to determine whether the reference was valid or invalid
2. If reference invalid, then terminate
   If valid, but, not yet brought in that page, we now page it in
3. We find a free frame (memory block)
4. We schedule a disk operation and read the desired page into the newly allocated frame (memory block).
5. When disk read is complete, modify internal table
6. Restart the instruction that was interrupted by illegal address trap



Hardware support for virtual memory

- To support Virtual Memory we will add a new field to each page map table (single bit)
- This bit is called ***present bit***
- If it is **1**, then the page is present **in memory**, and the base address field is valid
- If it is **0**, the page reference is legal but the page is **not in memory**
- When a page is accessed, the hardware checks the **present bit**, and if the present bit is **0**, it generates an interrupt (**page fault interrupt**)
- The page fault interrupt handler – reads the page and fixes up the page table



Demand Paging

- **Demand paging** (as opposed to **anticipatory** paging) is a method of virtual memory management.
- In this method, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (*i.e.*, if a **page fault** occurs).
- It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of a **lazy loading** technique.

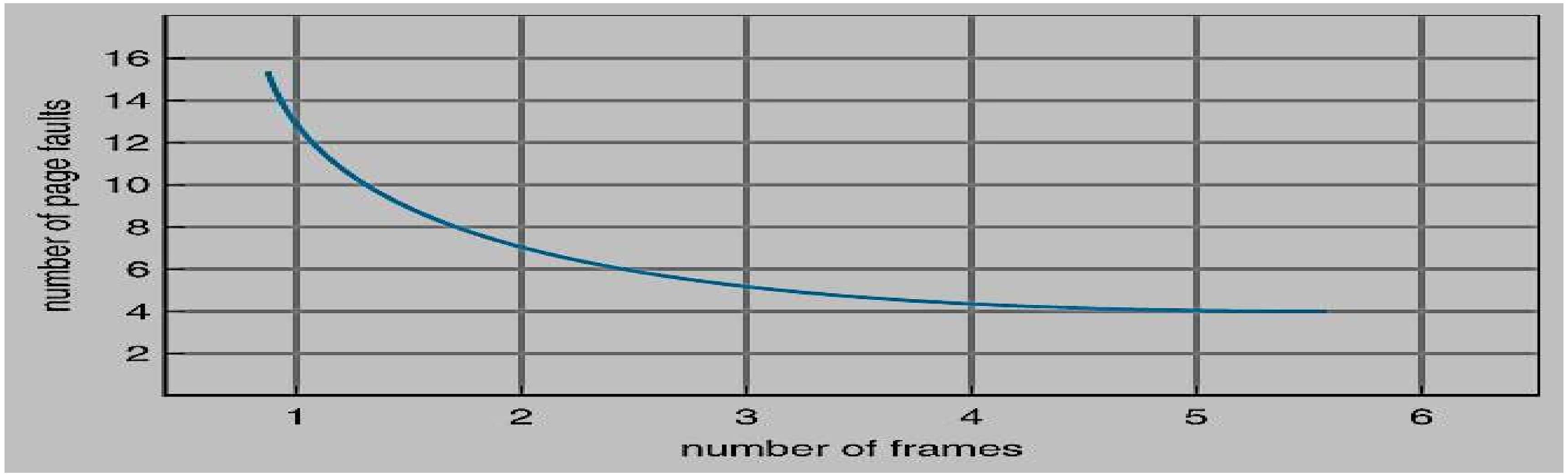
- To implement Demand paging we must develop
  - Frame allocation algorithm
  - Page replacement algorithm

Page Replacement

Basic Scheme

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

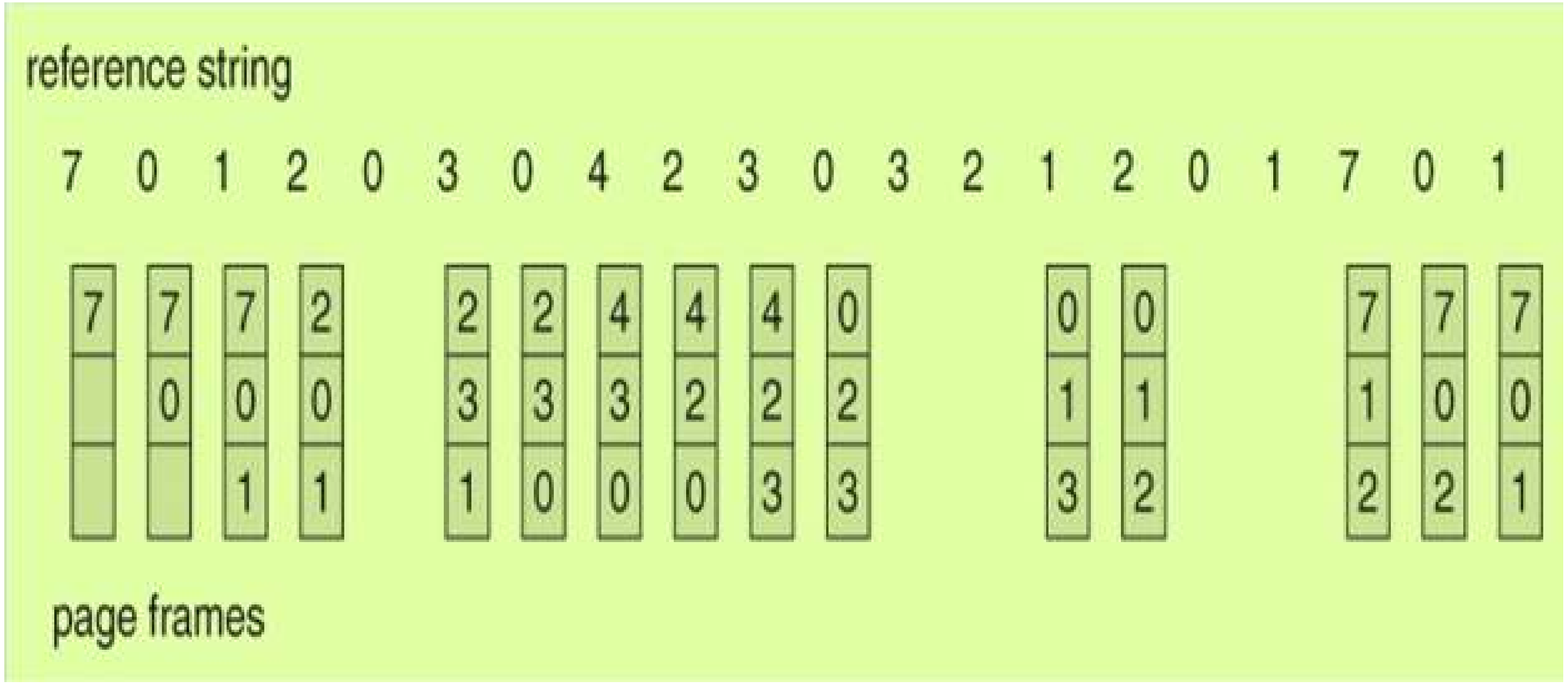
Terminologies



- The string of memory references is called ***Reference String***
- The page size is generally fixed (say 4K), so we need to consider only the page number (***p***)
- If we have a ***reference to page p***, then any **immediately** following references to page ***p*** will **not** cause a page fault.
- To determine the ***number of page faults***, for a given ***reference string,*** we need to know the number of ***page frames*** (memory blocks) available

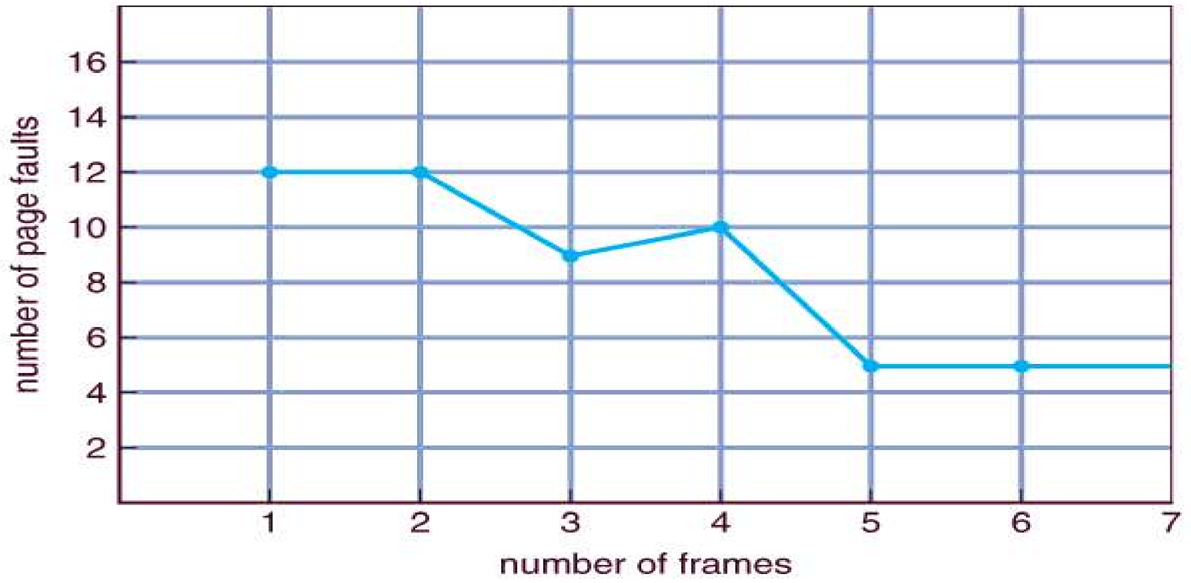
FIFO Page Replacement

- **First In First Out**
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim



Belady's Anomaly

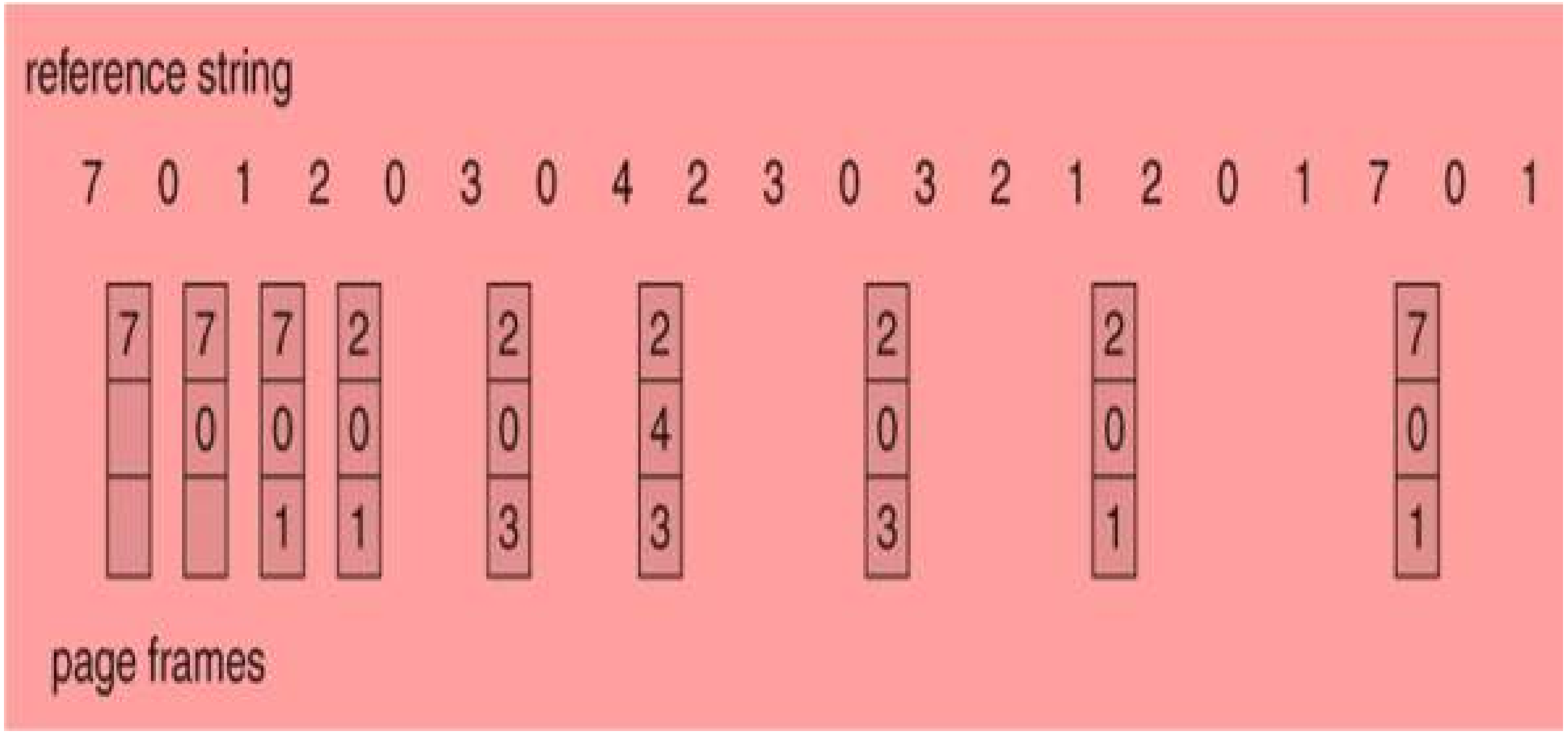
- Consider a page sequence 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 and a varying number of available frames



- Normally Increasing Number of frames should reduce page faults
- But the number of page faults for four frames is 10 is greater than the number of faults for three frames (9)!
- ***Belady's anomaly***: is unexpected result - in which for some page replacement algorithms, the page fault rate may ***increase*** as the number of allocated frames increases!
- FIFO exhibits Belady’s Anomaly

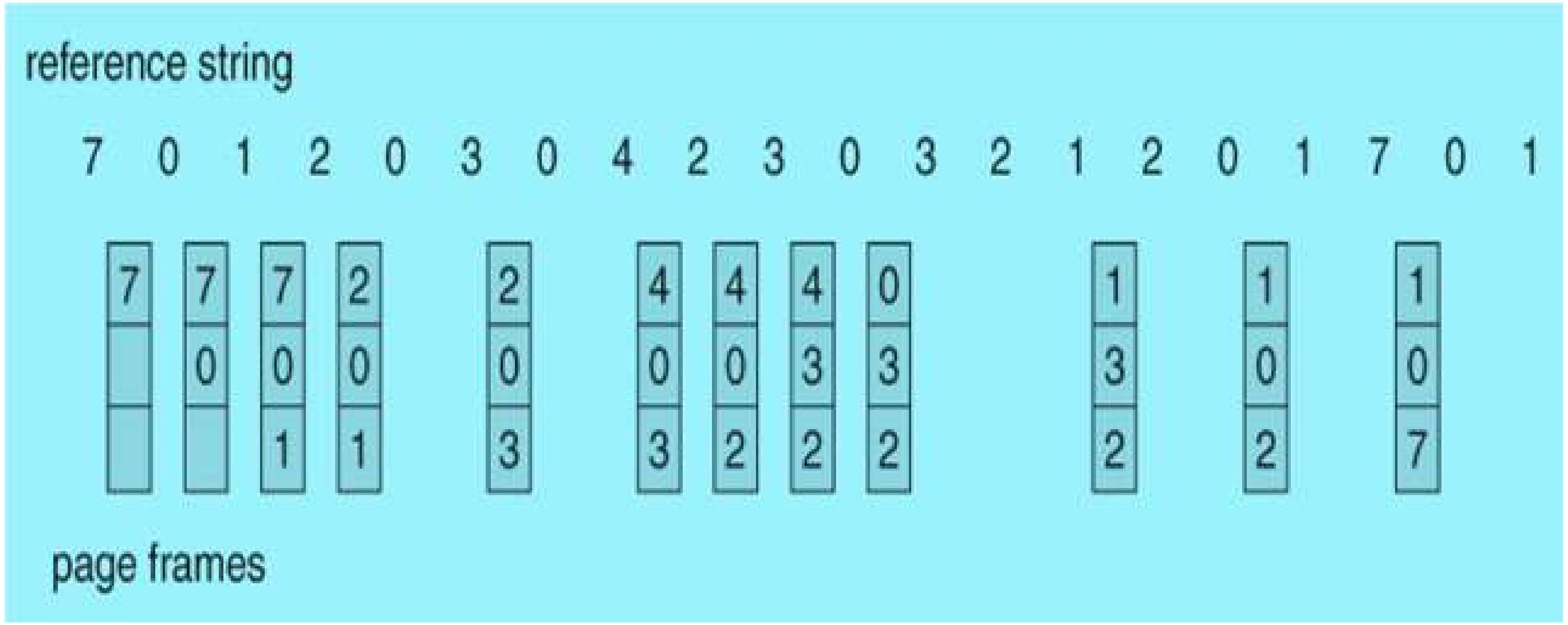
Optimal Page Replacement

- Replace the page that will not be used for the longest period of time!



LRU page Replacement

- Replace the page that has ***not been used*** for the longest period of time
- ***LRU – Least Recently Used***



Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***

Avoiding Thrashing

- Increased multi processing results in thrashing.
- **Avoidance**:
- **Local Replacement Algorithm**
  - If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also.
- **Page fault Frequency**

  We established upper and lower bounds on the desired page-fault-rate; if the page-fault-rate exceeds the upper limit, we allocate that process another frame; if the page-fault-rate falls below the lower limit, we remove frame from that process .

Try

- Consider the following page-reference string:

  1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
- Assume Four Frames
- How many page faults would occur if you use
  - FIFO
  - Optimal
  - LRU

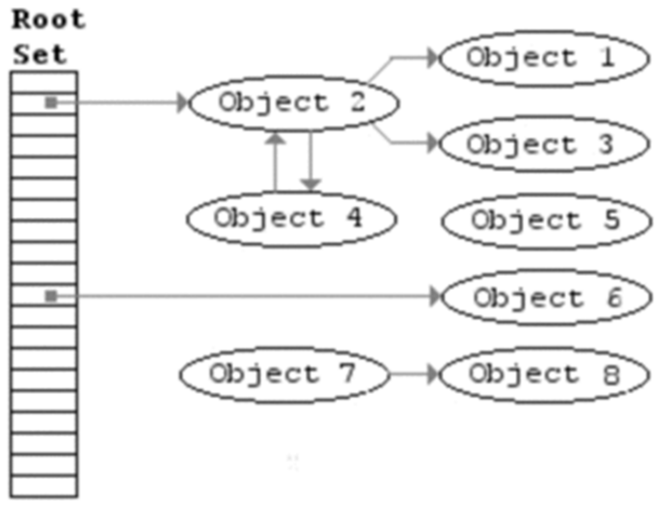
LRU Implementation Issues

- The problem is to determine an order for the frames by the last time of their use
- Two methods
  - **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
  - **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- However, both implementations of LRU require *hardware support*, either for incrementing the counter or for managing the stack, as these operations must be performed for every memory access.

Garbage Collection

- **Garbage collection** (**GC**) is a form of automatic memory management.
- The *garbage collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program
- each object in memory has a flag (a single bit) to indicate whether “in-use”.
- memory is scanned from start to finish, examining all free or used blocks; those with the in-use flag still cleared are not reachable by any program or data, and their memory is freed

Naïve mark-and-sweep



Objects #1, #2, #3, #4, and #6 are strongly referenced
Objects #5, #7, and #8 are not referenced directly or indirectly

Performance of Demand Paging

Effective Access Time = (1-p) X ma + p X page fault time

p – probability of page fault

ma – memory access time (10 to 200 nanoseconds)

page fault time – 500 milliseconds

Let p = 20%

Quiz/Zzzz?

- Which one of the following is the address generated by CPU?
  logical address

- Run time mapping from virtual to physical address is done by
  memory management unit

- The address of a page table in memory is pointed by
  page table base register

- What is compaction?
  a technique for overcoming external fragmentation

- **Explain Belady's Anomaly?**
- Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is Belady's Anomaly. (Also called FIFO anomaly). This is true for certain page reference patterns.

- **What is thrashing?**
- It is a phenomenon in virtual memory schemes when the processor spends most of its time swapping pages, rather than executing instructions. This is due to very high number of page faults.
- **What is the Translation Lookaside Buffer (TLB)?**
- In a cached system, the base addresses of the last few referenced pages is maintained in registers called the TLB that aids in faster lookup. TLB contains those page-table entries that have been most recently used.
- **In the context of memory management, what are placement and replacement algorithms?**
- Placement algorithms determine where in available real-memory to load a program. Common methods are first-fit, next-fit, best-fit. Replacement algorithms are used when memory is full, and one process (or part of a process) needs to be swapped out to accommodate a new program. The replacement algorithm determines which are the partitions to be swapped out.
- **In loading programs into memory, what is the difference between load-time dynamic linking and run-time dynamic linking?**
- *For **load-time dynamic linking***: Load module to be loaded is read into memory. Any reference to a target external module causes that module to be loaded and the references are updated to a relative address from the start base address of the application module.
- *With **run-time dynamic loading***: Some of the linking is postponed until actual reference during execution. Then the correct module is loaded and linked.

Thank You