Memory Management

CMSC 125

Memory Manager

- Problem of determining which ready processes should be allocated some amount of memory
- CPU and memory manager must “complement” each other

Address Binding

- Crucial to know that code of a process can be moved around in memory during execution
- Process must be loaded in memory during execution
- Executable code may be loaded in different parts of the memory
- Compiler binds instructions and data to memory addresses

Address Binding

- Binding at compile time
  - If it is known where the process will reside in memory before it is executed, compiler may generate *absolute* code (code will be loaded at a fixed memory address)

Address Binding

- Binding at load time
  - If it is not known at compile time where the code will be loaded, compiler may produce *relocatable* code
  - Binding of instructions and data to memory locations is delayed until load time
  - If starting address changes, just re-load the code

Address Binding

- Binding at execution time
  - – A requirement of memory management schemes that allows code to move from one location to another during the execution of a process

Address Binding

- Most modern OS requires binding occur at execution time, to allow the code to be moved around in memory
- Memory managers will tend to move code from memory to secondary storage and back to memory again
- When code is moved back in memory it may be loaded in a different location

Logical vs Physical Address

Logical address

- address generated by the CPU
- virtual address

Physical address

- address seen by the memory unit (the one loaded into the memory address register of the memory)

Logical vs Physical Address

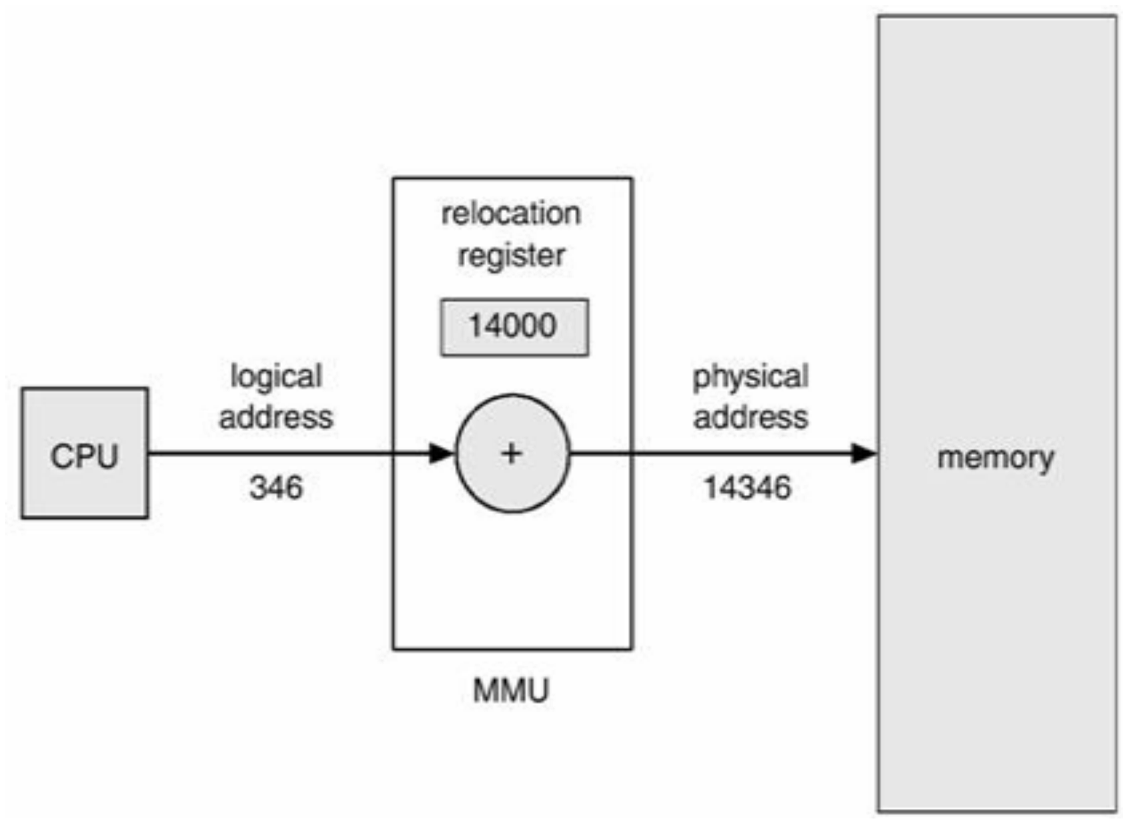
- Compile-time and load-time address-binding schemes results in an environment where logical and physical addresses are the same.
- Execution-time address-binding schemes, logical and physical addresses differ.

Logical vs Physical Address

Memory Management unit (MMU)

- run-time mapping from virtual to physical address
- this is a hardware device; relocation register
- value in the relocation register is added to every address generated by a user process at the time it is sent to memory

Logical vs Physical Address



Dynamic relocation using a relocation register

Memory Management in Early Systems

Single-user Systems

- Memory management was simple
  - Most single-user memory managers tries to allocate all the memory required by a user process
  - Allocates a contiguous piece of memory to the single-user process

Single-user Systems

- Problems:
  - Single processors does not fully occupy the allotted memory for user process
  - Creates an unused portion of the memory
  - Some user process try to access (worse modify) accidentally or intentionally, area occupied by the OS

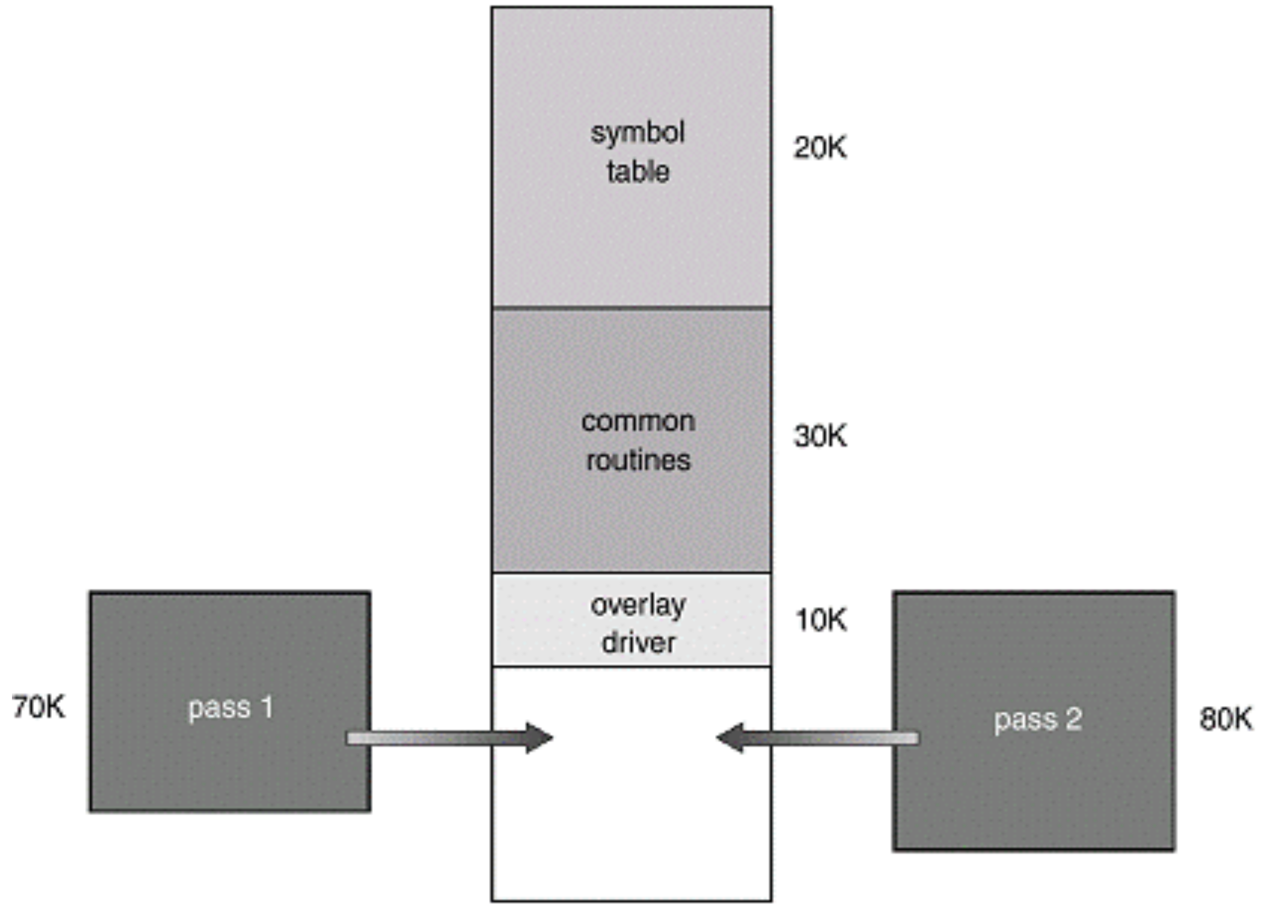
Single-user Systems

- Sizes of the program are limited to the amount of main storage reserved for the user processes
- If memory requirement of a user process is greater than the available memory, then it cannot be executed anymore

Single-user Systems

- Solved by **Overlays**
  - When a section of a program is not needed anymore, another section may be brought in from secondary storage to occupy the storage used to be occupied by the section not needed for execution anymore
  - Allows programmers to extend the limited main storage for user processes
  - Controlled by programmers; requires careful planning

Single-user Systems



Overlays for a two-pass assembler

Multiprogramming System

- There is a need to enhance memory manager to one where it allows several processes to be occupying memory at the same time, to allow continued usage of the CPU when a switch from one job to another is being made

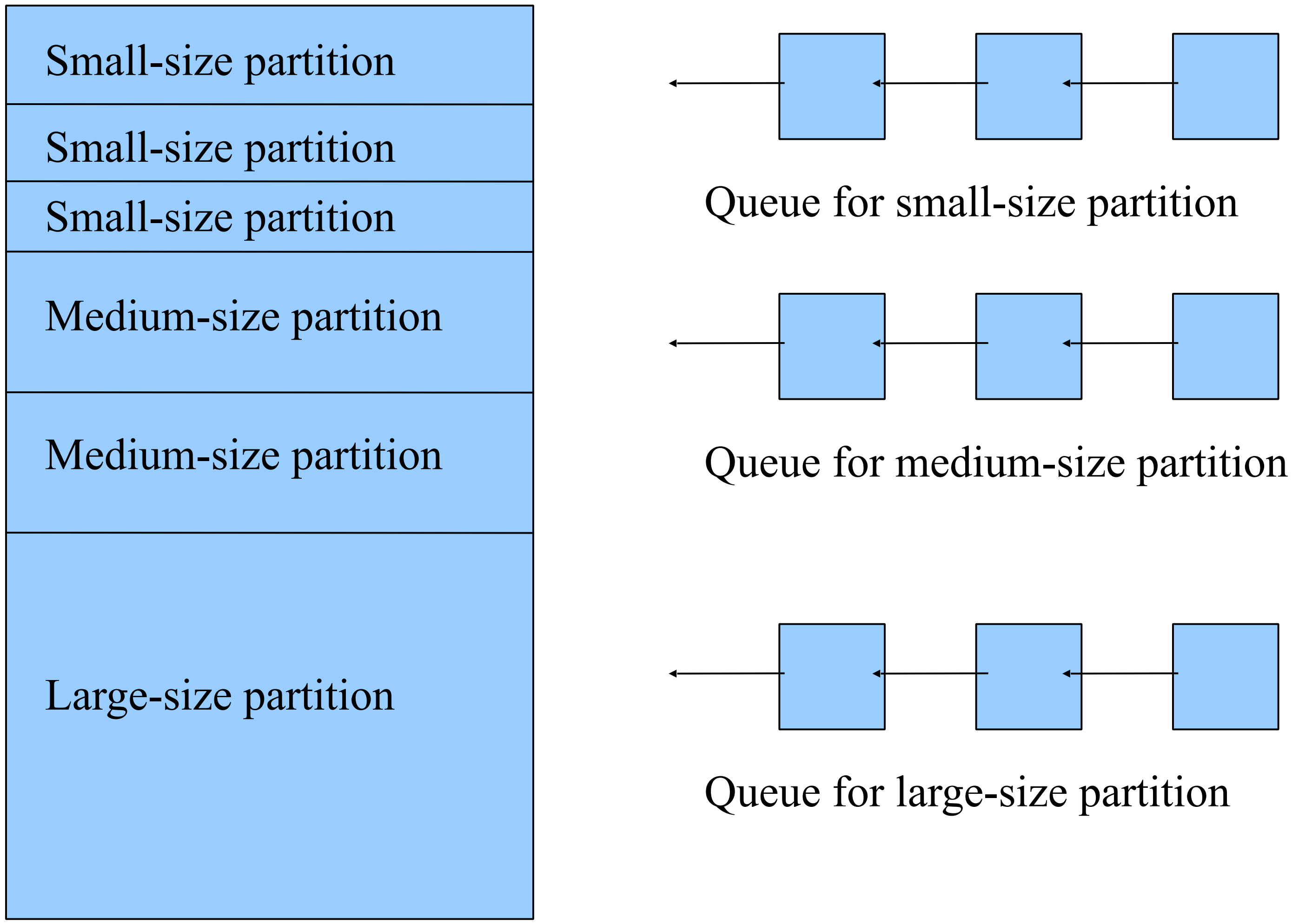
Multiprogramming System

- Memory manager needs to provide a scheme that minimizes the number of processes being blocked because their codes are not loaded yet

Multiprogramming System

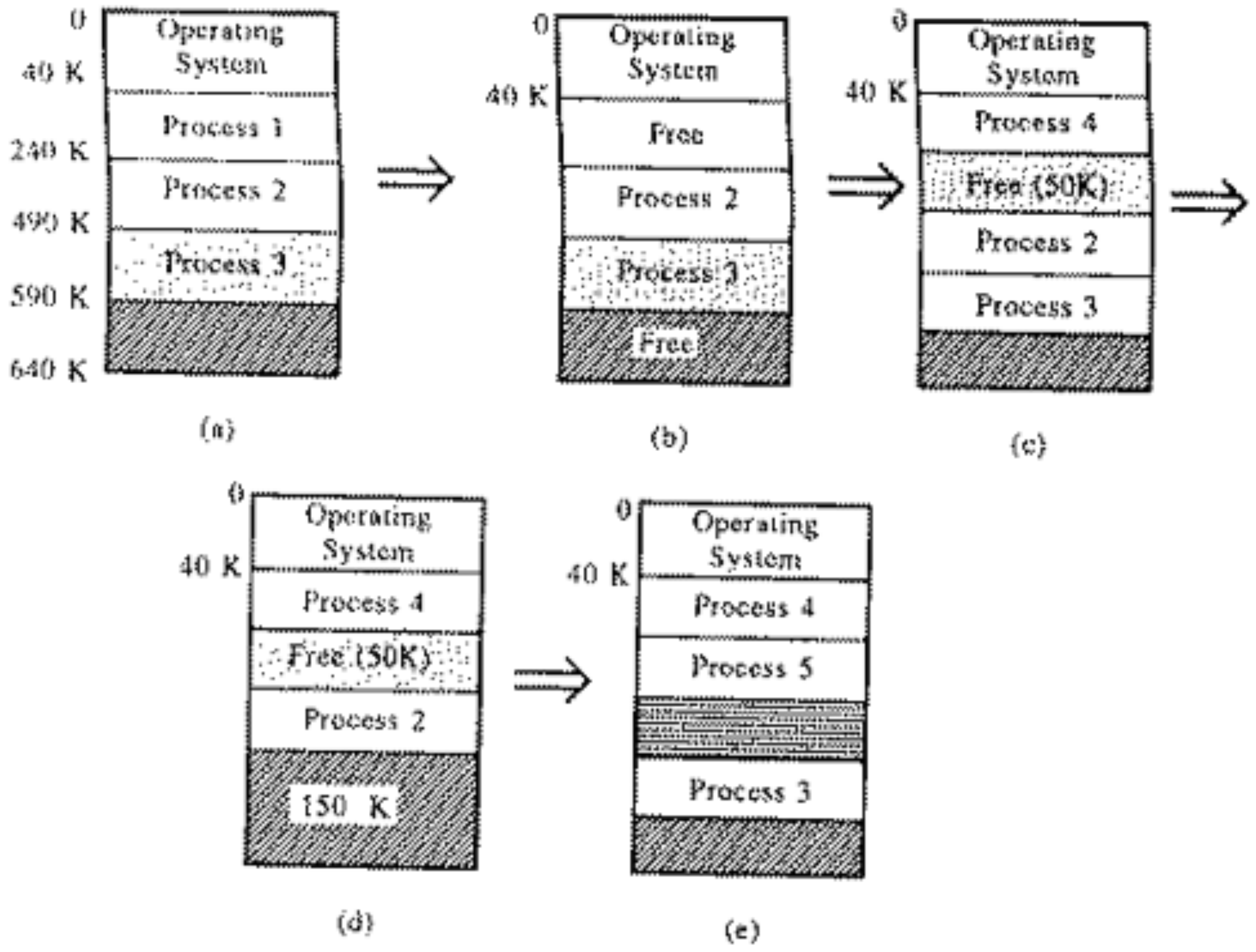
- For batched systems
- Load *fixed number of tasks* in memory
- Divide memory into *small-size*, *medium-size* and *large-size* partitions
- Example:
  - 640k user memory area may be partitioned into ten 20k partitions (small-size), four 50k partitions (medium-size) and one 240k partition (large-size)

Multiprogramming System



Multiprogramming with fixed number of tasks

Multiprogramming System



Multiprogramming System

- Each partition is associated with a queue
- Each partition could hold one job and CPU is switched rapidly between jobs in the partition to create the illusion of parallelism

Multiprogramming System

- Partitions are set up by the operator in the morning until the next day, or until the operator decides to change the sizes during the day
- Requires reconfiguration of the system, usually done in the morning when the load is not that much yet

Multiprogramming System

- This scheme was extensively used by OS/360 on large IBM mainframe for many years
- MFT (Multiprogramming with Fixed number of Tasks or OS/MFT)
- When a job arrives, in can be placed into a queue for the smallest partition large enough to hold it

Multiprogramming System

- Best-fit algorithm
  - Place a job in partition that will minimize the amount of memory unused
- Since partitions are fixed,any unoccupied space is lost
- Internal fragmentation
  - When a user job does not completely fill their designated partition
  - Lost memory caused by memory deemed allocation but is unused

Multiprogramming System

- External fragmentation
  - Lost memory caused by memory that can be allocated but no taker is available to occupy it
  - Empty partitions because no job is queued on them
  - There is enough total memory space to allocate a request, but not contiguous, storage is fragmented into different holes

Multiprogramming System

- Scheme that allows a variable number of tasks
- MVT (Multiprogramming with Variable number of Tasks or OS/MVT) arised to solve the problems of MFT

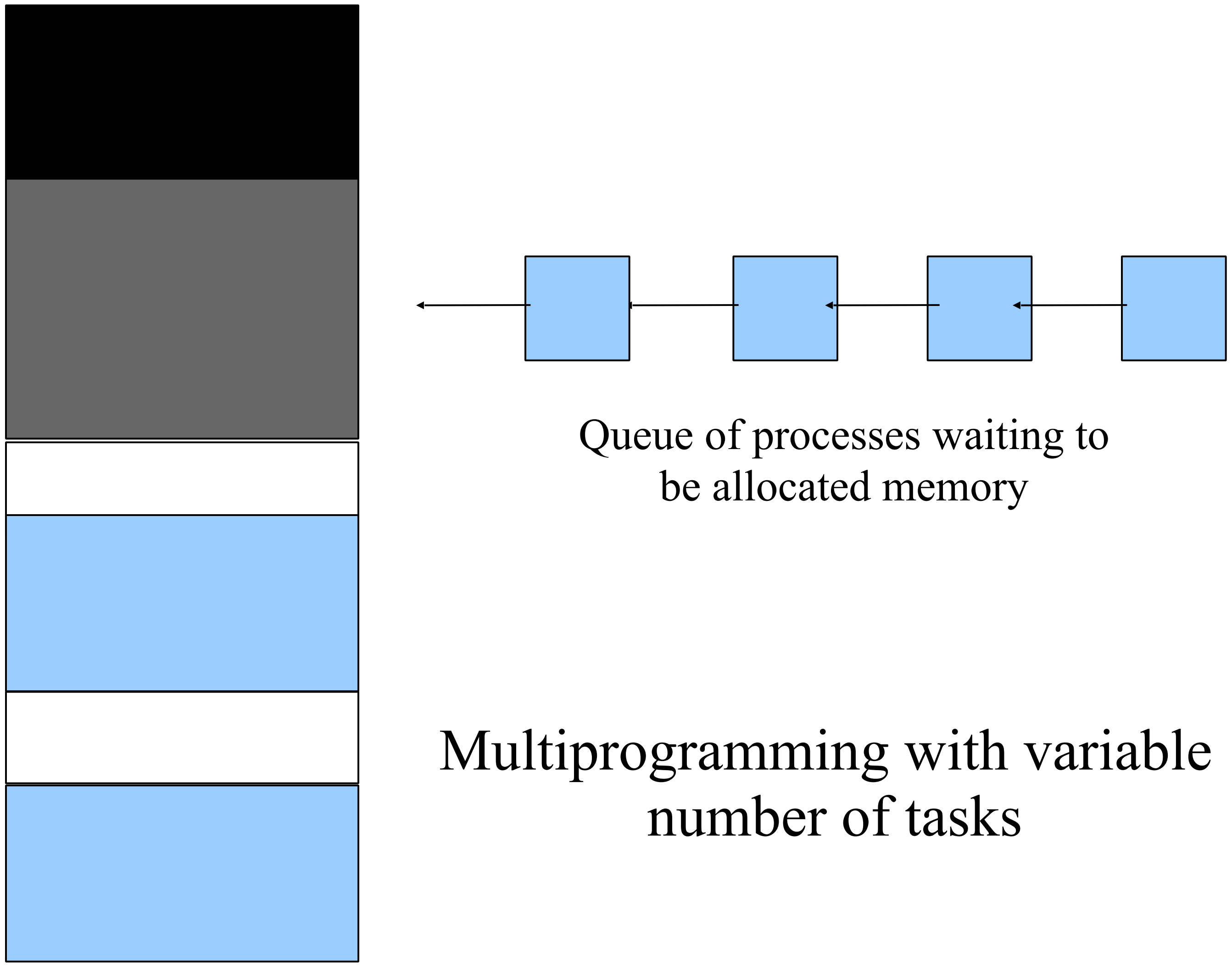
Multiprogramming System

- Number of task is variable because its number is dependent on the number that is allocated memory
- Only one queue of job
- Allocate memory whose size is equal to the size of the code for the process
- Problems will occur once the jobs starts to terminate and will leave holes in memory

Multiprogramming System

- External fragmentation
- One may attempt to compact the memory in order to make a hole big enough to accommodate an incoming job
- Compaction is expensive

Multiprogramming System



Queue of processes waiting to be allocated memory

Multiprogramming with variable number of tasks

Multiprogramming System

- External fragmentation may become serious, i.e., there are so many free memories (not contiguous) to accommodate an incoming job
- Solutions:
  - Compaction
  - Relocation

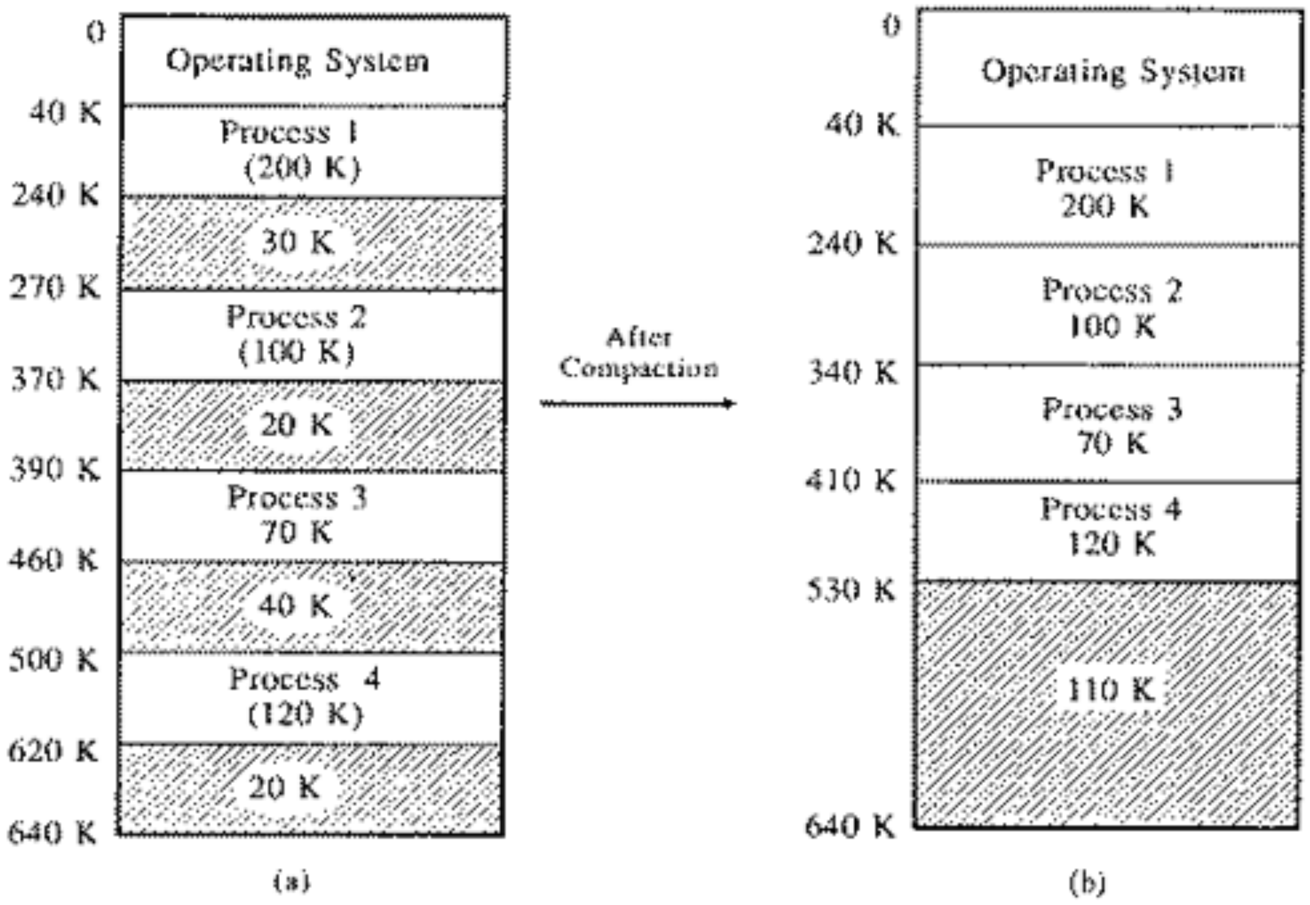
Multiprogramming System

- Compaction
  - Moving all programs in memory to one side of memory so that they are side by side to each other, leaving a large space at the other side of the memory
  - Usually used as a last resort, expensive
  - Consumes CPU time that could otherwise be used by user processes
  - Must stop everything while it is being done
  - Requires relocating codes of user process

Multiprogramming System

- Compaction
  - Possible only if the program in memory can be relocated
  - Programs that can be relocated are programs that can be located/loaded in different parts of the memory at different times during its execution

Multiprogramming System



Multiprogramming System

- Relocation
  - Normally, a program can grow in size especially if it involves operation on the heap part of the data
  - A growing program will then have no place to grow/expand (heap)
  - Relocate it to another part of the memory

Contiguous and Non-contiguous Memory Allocation

- Early operating systems like MFT and MVT allocates memory contiguously
- If no memory is big enough to accommodate a program, then it stays in secondary storage
- If program is larger than available memory, it is not possible to execute the program

Contiguous and Non-contiguous Memory Allocation

- Advantage
  - simple to implement
- Disadvantage
  - low level of multiprogramming
  - rampant fragmentation
  - Expensive if compaction/ relocation is allowed

Contiguous and Non-contiguous Memory Allocation

- Modern OS use non-contiguous memory allocation
- Program is divided into several blocks or segments, blocks may be placed anywhere in memory in pieces (not necessarily contiguously and not necessarily all the blocks)

Contiguous and Non-contiguous Memory Allocation

- Advantage
  - blocks are smaller in size and could easily fit in holes found in the memory
  - fragmentation is minimized
  - All that is needed to be loaded are only the blocks that are involved in the current execution
  - High level of multiprogramming since many programs can be loaded in memory at the same time

Contiguous and Non-contiguous Memory Allocation

- Disadvantage
  - Very complicated to implement

Memory Management in Time-Sharing Systems

- Allocation of memory to ready processes in a time-sharing system (considered as one of the major problems for the designers of the OS)
- Several user processes are running and not all of them will fit in memory
- Problem: how to manage memory to handle several processes without too much degradation in response time

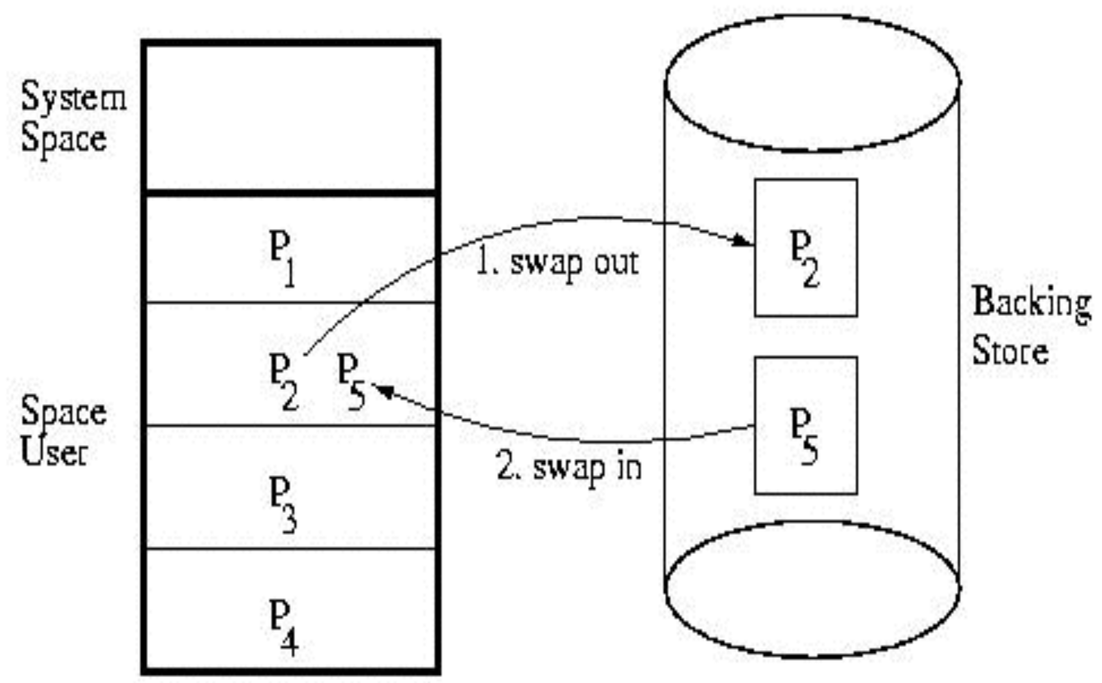
Swapping

- Sometimes, a ready process (in a time-shared system) will have to be allocated the CPU even when its code is still in the backing store
- One of those currently occupying the memory will have to give up its space to make way for the higher priority process

Swapping

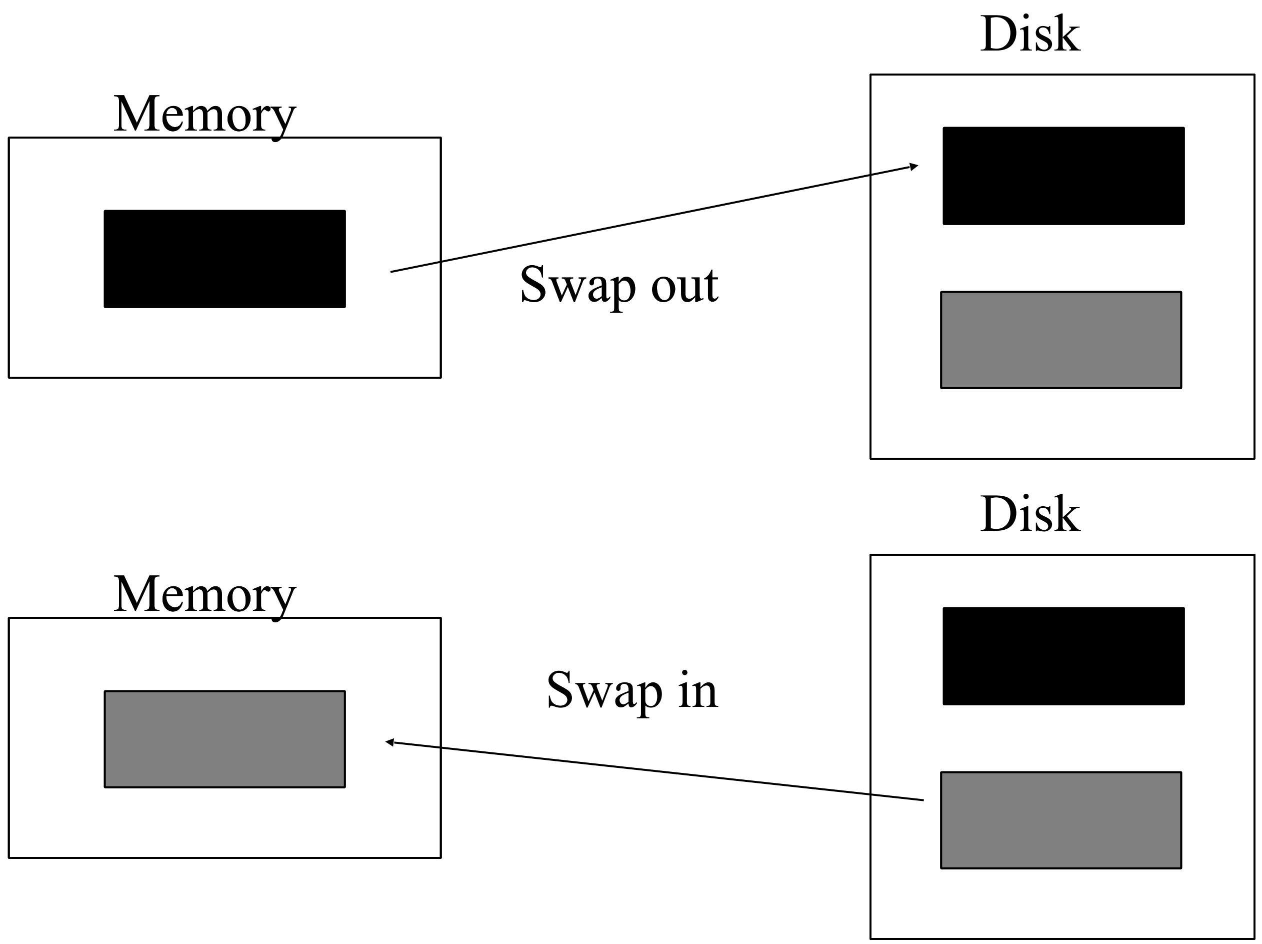
- Process that is uselessly occupying memory (waiting for an I/O operation to complete or has consumed its time-share) is selected to give up its space in memory
- Write the core image of this process to the backing store and the core image of the higher priority process is read in

Swapping



Swapping of two processes using a disk as a backing store

Swapping



Swapping

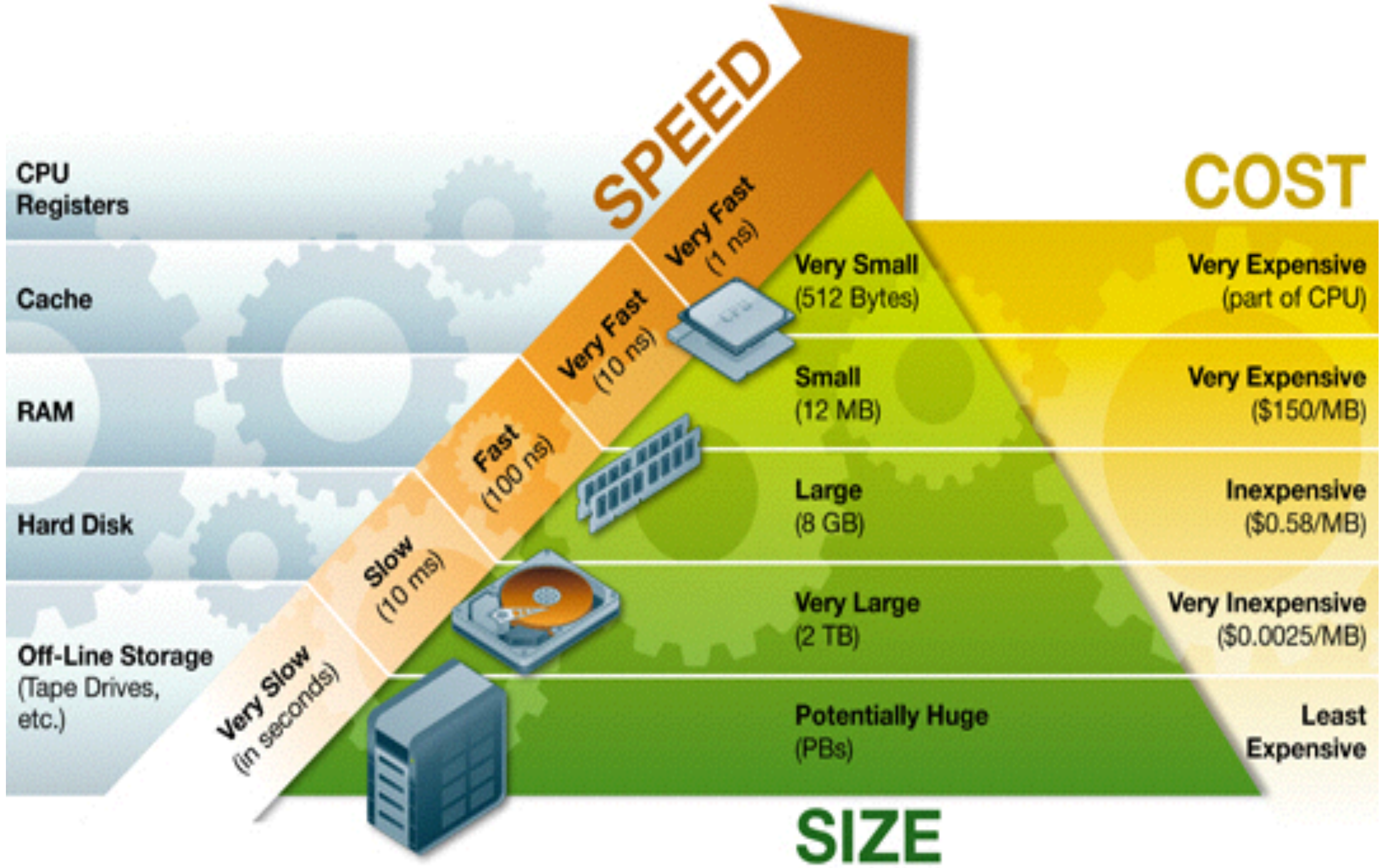
- Swapping device (storage device used to store the core images of swapped out process) may consist of a hierarchy of storage devices, varying in capacity and speed from comparatively small, fast fixed head disk, to slow, but larger exchangeable disks
- Aside from secondary storage, may employ a fast memory, but expensive, to help speedup swapping

Swapping

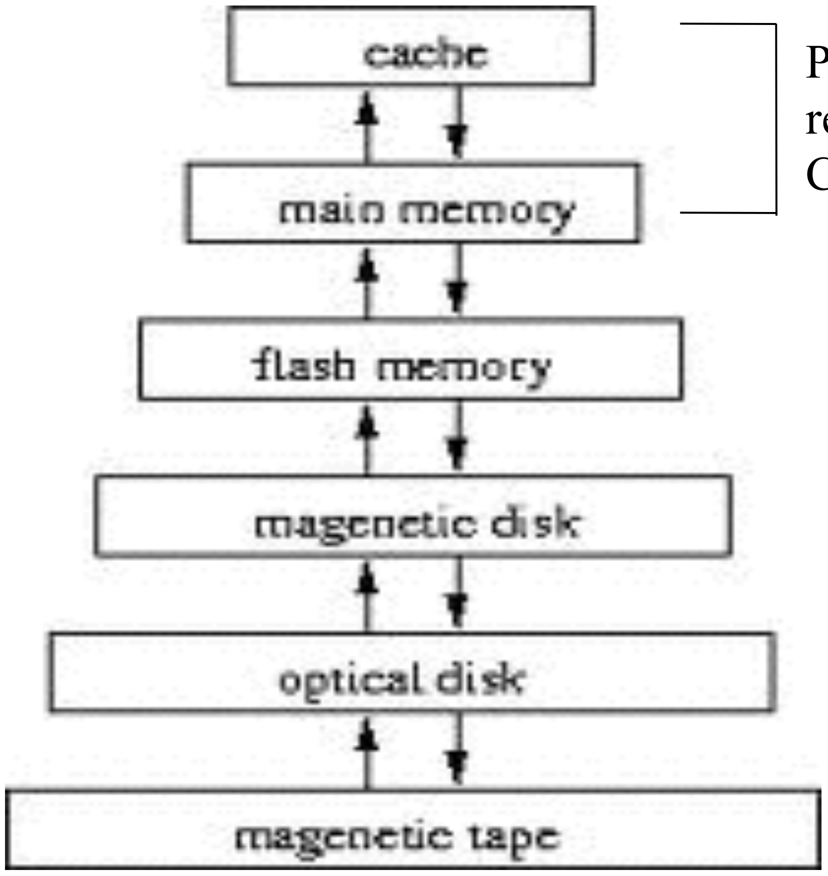
- Cache memory
  - Creating hierarchical access techniques that takes advantage of smaller but faster memories

Memory manager is responsible for organizing the swapping of programs between the devices in the hierarchy, although management of the cache is in general a function of the hardware.

Swapping


SPEED
COST
SIZE
CPU Registers
Cache
RAM
Hard Disk
Off-Line Storage (Tape Drives, etc.)
Very Fast (1 ns)
Very Fast (10 ns)
Fast (100 ns)
Slow (10 ms)
Very Slow (in seconds)
Very Small (512 Bytes)
Small (12 MB)
Large (8 GB)
Very Large (2 TB)
Potentially Huge (PBs)
Very Expensive (part of CPU)
Very Expensive ($150/MB)
Inexpensive ($0.58/MB)
Very Inexpensive ($0.0025/MB)
Least Expensive

Swapping



Program and data referenced by the CPU directly

Programs and data must be moved to memory before they can be referenced by the CPU

Hierarchy of storage devices

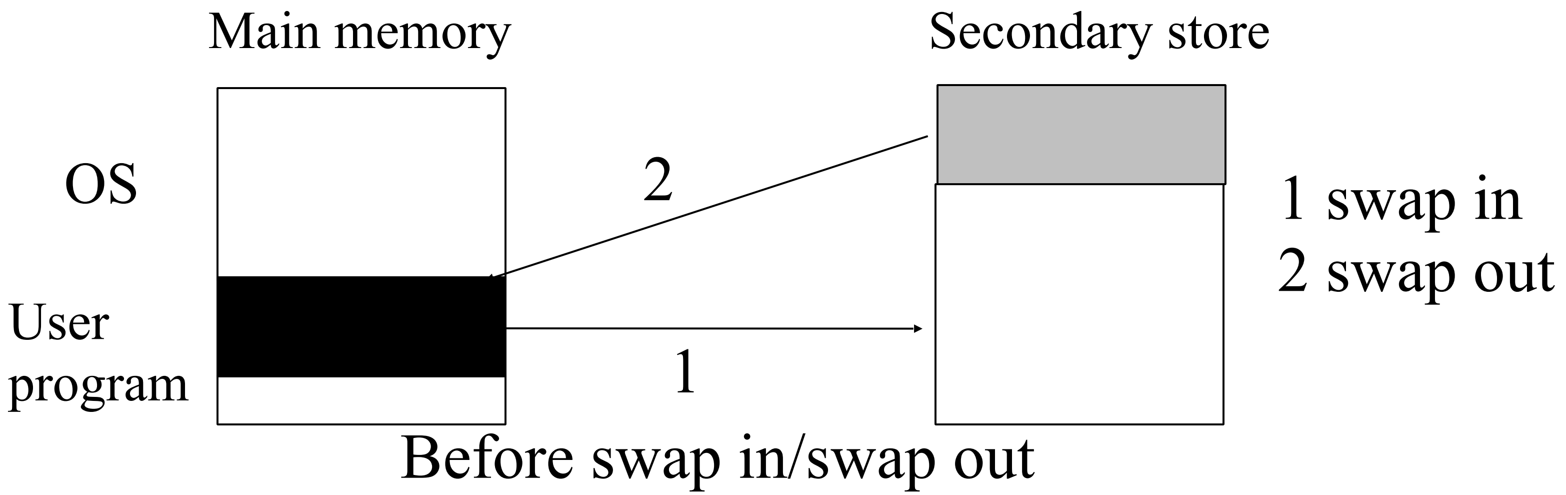
Swapping Strategies

- Several variations in the technique of swapping programs
  - Total amount of store required;
  - Time waster as a result of swapping

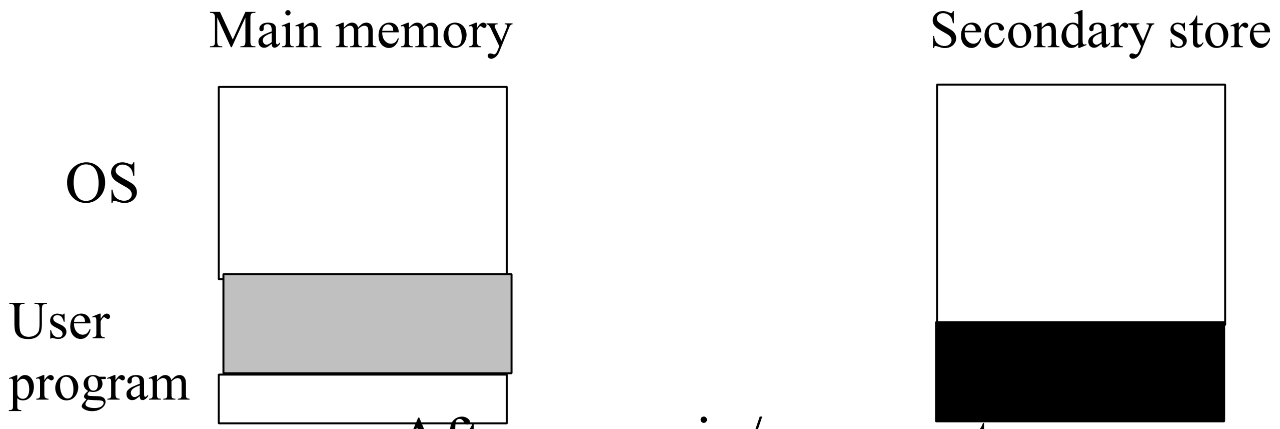
Swapping strategies

1. Simple swapping system
2. A more complex swapping system

Swapping Strategies



Before swap in/swap out



After swap in/swap out

Swapping Strategies

- Let *sin* swap-in time

  sout swap-out time

  slice time slice

We can arrange the system so that memory is large enough for just the OS and one user process, and all processes have to be swapped into the single space when they are allocated their time slice

Swapping Strategies

Store size = 1 process + OS (in memory)

time for each interaction = *sin* + *sout* + CPU time

** CPU time will vary accdg. to what the process is doing, up to a maximum value given by the time slice period.

Swapping Strategies

CPU utilization = (useful time) / (total time) x 100%

= *slice* / (*sin*+*sout*+*slice*) x 100%

= 1 / (1+1+1) x 100%

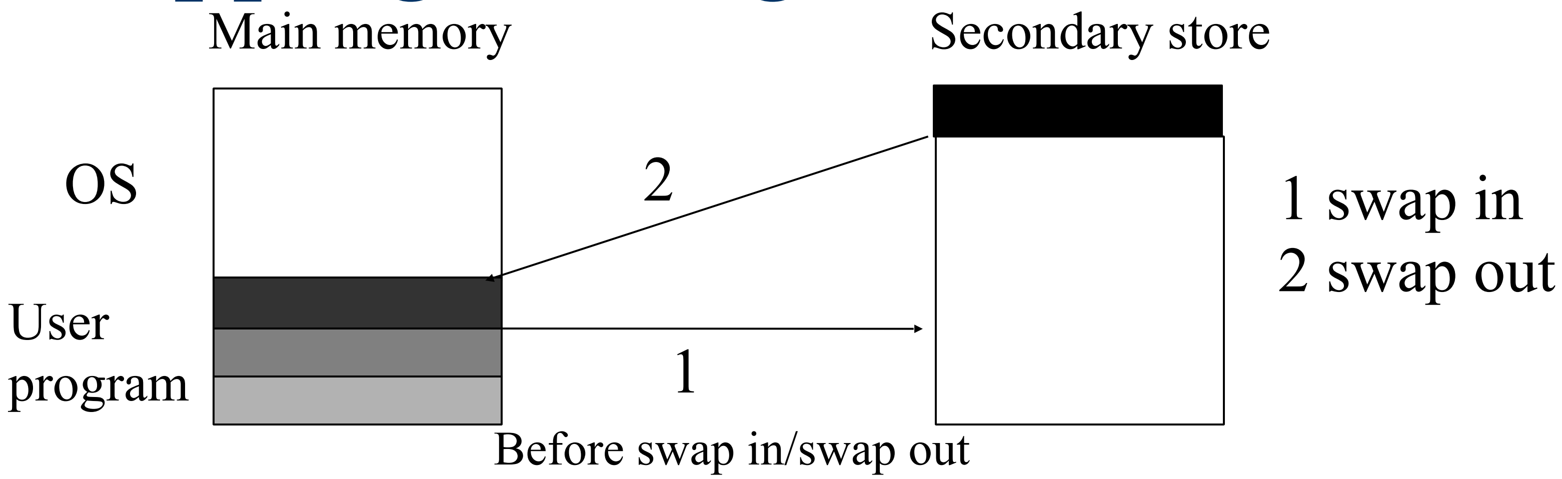
= 1 / 3 x 100%

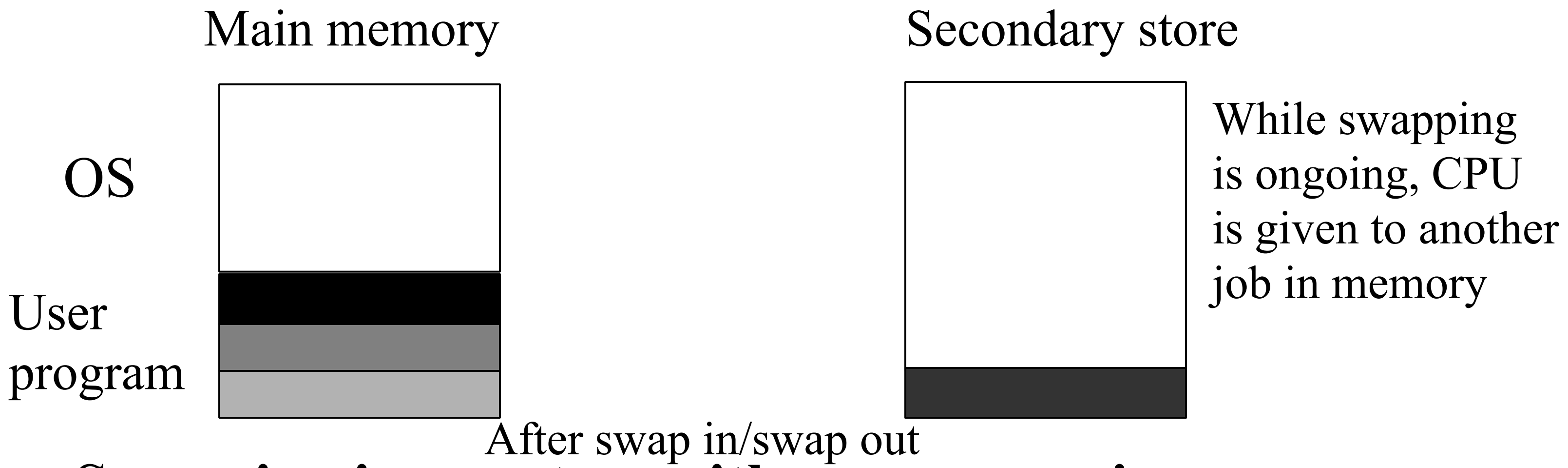
= 33.33%

Swapping Strategies

- Disadvantage: CPU is idle while the process is swapped in and out of the memory.
- We can arrange for another process to take control of the CPU while process swapping is going on.
- So…we obviously need more than one process in memory.

Swapping Strategies





While swapping is ongoing, CPU is given to another job in memory

Swapping in a system with n processes in memory

Swapping Strategies

With this scheme, possible that when process i-1 is swapped out, process i+1 is swapped in and while swapping is going on, process i is executing.

Swapping Strategies

Store size = n processes (n >= 1) + OS

time for each interaction = max(*slice*,*sin*+*sout*)

CPU utilization = (useful time) / (total time) x 100%

= *slice* / max(*slice*,*sin*+*sout*) x 100%

= 1 / max(1,1+1) x 100%

= 1 / 2 x 100%

= 50%

Swapping Strategies

- Time spent for swapping is an important factor for determining time slice value.
- If time slice is very small, system might be spending most of its time swapping (overhead, remember?) than doing some useful work.
- Example: increasing time slice to 2 secs increases CPU utilization from 50% to 100%

Memory Protection

- Memory belonging to a process (kernel or user) must be protected from unauthorized access.
- Usual approach: separate OS from user process, allowing only a user process to access its own address space while OS process is allowed to access any memory location.

Memory Protection

- Methods of protection:
  - Automatic Address Translation
  - Page 0 - Page 1 addressing
  - Use of base and limit registers

Memory Protection

- Automatic Address Translation
  - OS is placed in either the low or high end of the memory
  - User references are modified to prevent them from accessing the address space of the OS
  - Example: if OS is loaded at the lower end of the memory (i.e., loaded at address *0* to *S*), then a user reference to address *u* is automatically modified by accessing (*u*+*S*)

Memory Protection

- Automatic Address Translation
  - Example: if OS is loaded at higher end of memory (i.e., upper *S* location of memory), then a user reference to address *u* is modified by accessing (*u mod* (*M*-*S*)), where *M*=total size of memory
  - So it wraps around the user area when a user reference is made.

Memory Protection

- Page 0 - Page 1 addressing
  - Used only when the address space is equally divided between the OS and the user process (i.e., half the address space is for OS and other half for user)
  - Suppose OS is loaded at the lower end of the memory. Every time a user reference to address *u* is made, this address is concatinated with a 1 at the beginning of its binary representation (in effect, adds *M/2* to *u*, where *M*=size of memory)

Memory Protection

- Using base and limit registers
  - A user process can be prevented from accessing the OS area by providing two CPU registers; base and limit registers.
  - Let *base* and *limit* be the values stored in the base and limit registers, respectively. Value of *base* is the starting address where the process is loaded. Value of *limit* is the size of the code for the said process.

Memory Protection

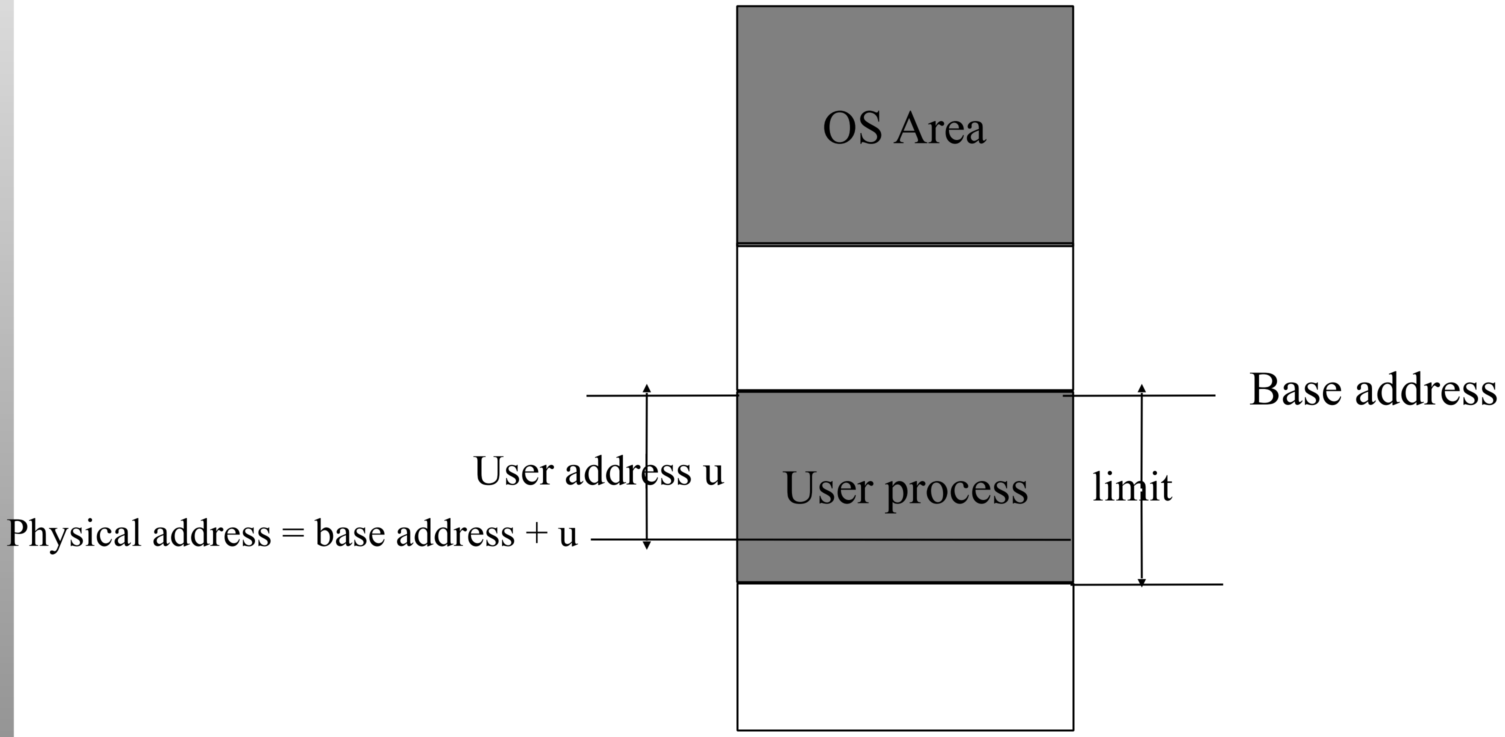
- Using base and limit registers
  - User reference to address u, will have to be modified as follows:

```
if (u > limit)
    "address error"
else
    physical add = u + base;
```

Memory Protection

- Among the 3 methods of protection, the use of base limits and registers is the one that can be easily extended to protect a multi-user OS.
- Idea is to add 2 entries in the Process Control Block (PCB) of a process; base and limit.

Memory Protection



Base and limit registers protection

Memory Protection: Example

- Windows NT
  - OS runs in kernel mode
  - Applications in user mode
  - OS area can be accessed only when the code running is in kernel mode
  - User applications that are running in user mode will never be able to access the area allocated to the OS

Memory Protection: Example

- Windows NT
  - User app may request access to the OS area by requesting an appropriate OS code to do it
  - Windows app is protected from another application by making each app run its own private address space.
  - When an app tries to access outside this private address space, it is prevented from doing so. Accidental access of one app to another app is not possible.

Design Considerations of Memory Management

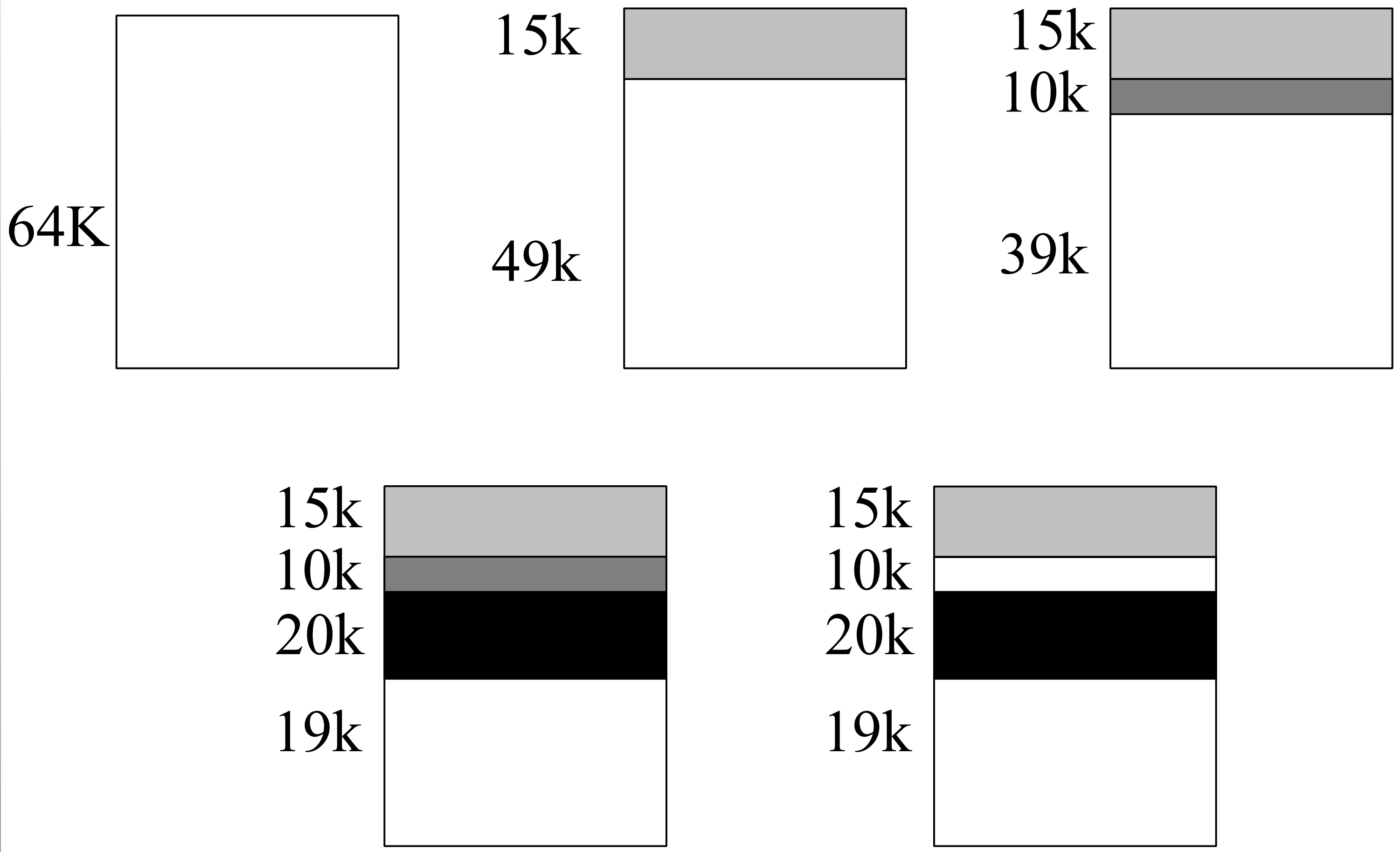
To implement an efficient memory manager, need to consider the ff:

1. Fragmentation
2. Locality of programs
3. Sharing of code

Memory Fragmentation

- Holes in memory might be created in the process of swapping in and out of programs in memory
- It is not possible to allocate these holes to existing jobs that need memory since the size of the holes are too small

Memory Fragmentation



Memory Fragmentation

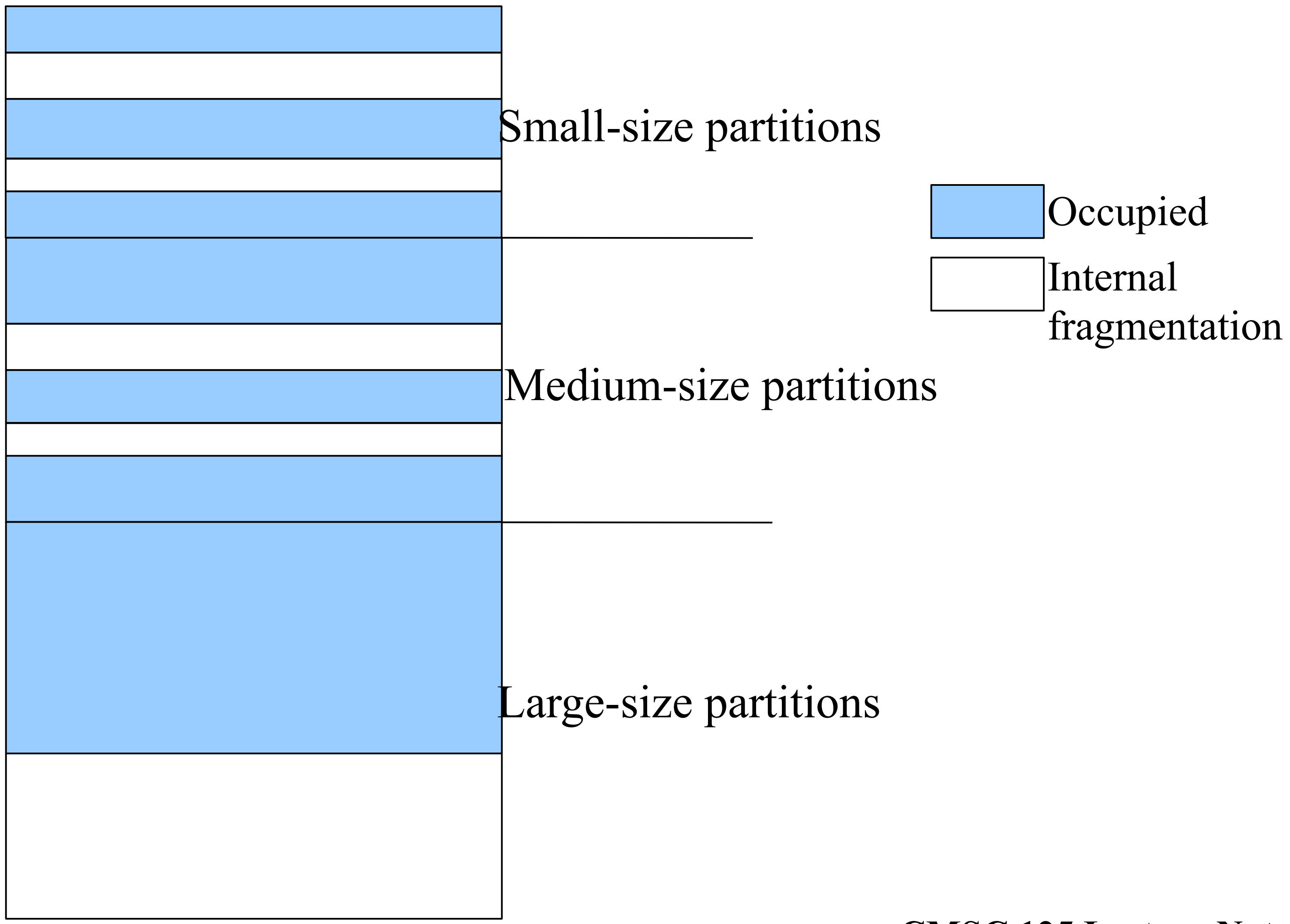
Initially, memory is free.

Let jobs whose sizes are *15k*, *10k* and *20k* are loaded in that order.

After a while, *10k* job terminates, which creates two chunks of memory that are free (*10k* and *19k* chunks).

Suppose next job needs *25k* for memory. Although total space available is *29k*, it's not possible to allocate the free memory, since it's not contiguous.

Memory Fragmentation



Memory Fragmentation

When a job needs *M* words of memory and is allocated *N* words, where $N>M$, $N-M$ is the amount of internal fragmentation in the system.

Memory Fragmentation

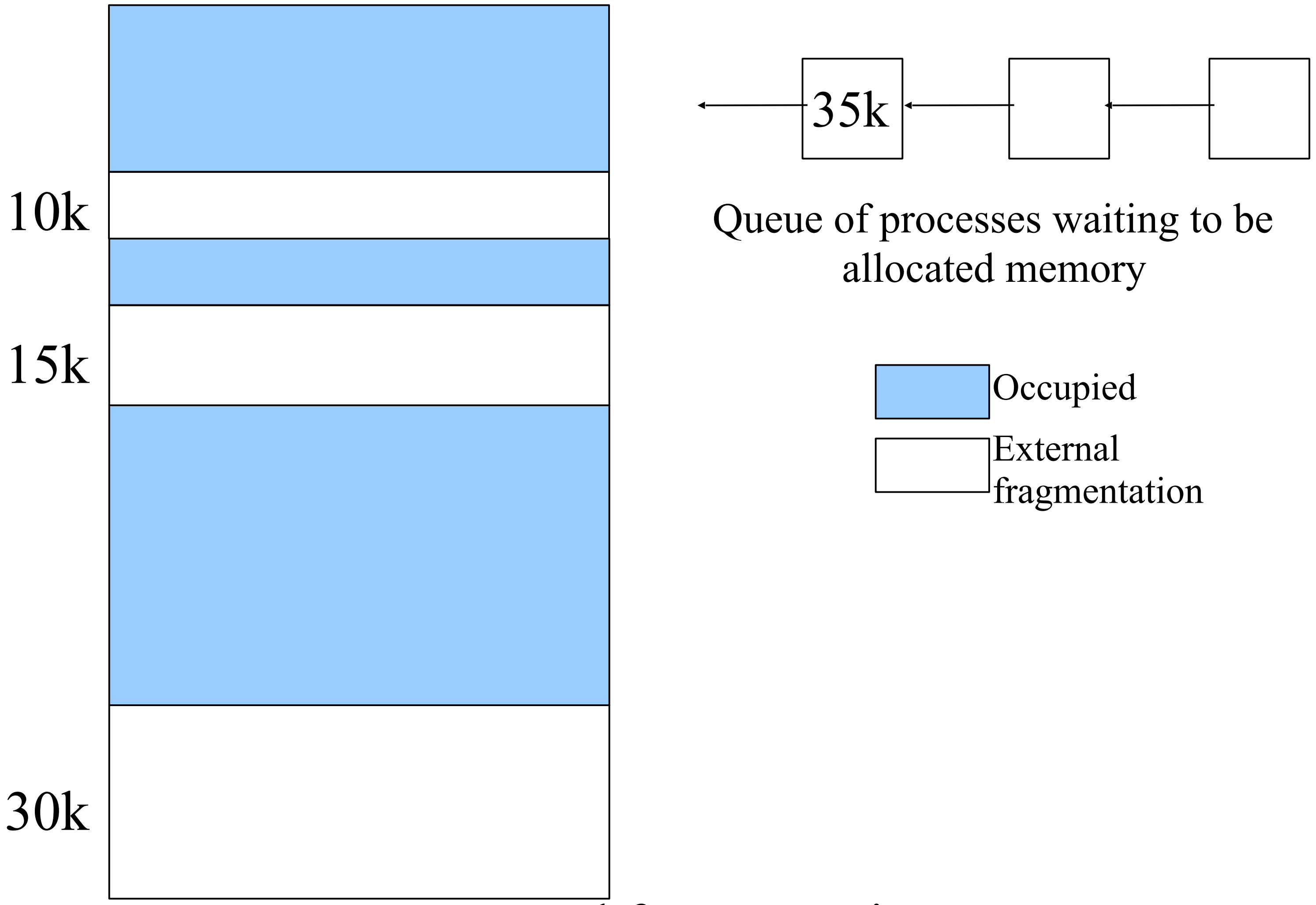
As long as the OS allows the allocation of memory whose size is greater that what is required by a process, internal fragmentation can never be eliminated.

To solve external fragmentation, move up all jobs to “squeeze” out the holes and leave all unused memory as one contiguous block in one end of memory.

Memory Fragmentation

- Compaction – very high overhead, consider the binding of user variables to memory location
  - Worse case, needs recompilation of user programs in memory.
  - Okay, since code in most systems is relocatable. No need to load whole program to memory, only the necessary parts.

Memory Fragmentation



External fragmentation

Locality of Program Execution

- Concerned with the way programs use their virtual address space
- In paging systems, it is observed that processes tend to access certain subsets of pages and that these pages are often adjacent to one another

Locality of Program Execution

- Can be observed in accesses to the memory by computer systems
- Temporal locality can be observed when the computer system execute code that contains loops in them
- Spatial locality is observed when the computer system execute code that is sequential.

Sharing of Code

- In time-sharing systems, several users might need the same utilities (compiler, assembler, text editor etc)
- No need to provide each process a copy of the code
- No need to swap in a copy of the compiler with each request from a process
- Memory manager just checks if the said compiler is loaded

Sharing of Code

In order to be able to share codes in the system, the processes should:

1. not be self-modifying
2. store their data separate from the programs

Virtual Memory Management

- One problem of earlier systems: Impossible to run a program if its size is greater than the available memory allocated to users.
- Overlays were introduced to solve this problem. Overlays are stored in secondary memory and are swapped in and out of the memory.
- User has to make user the size of one overlay does not exceed the size of available memory

Virtual Memory Management

A process maintains a virtual memory that is part main memory and part secondary storage.

Virtual Memory Management

Techniques:

1. Segmentation
2. Paging
3. Paged Segmentation

Segmentation

- One segment may contain set of instructions (program part) and another segment for data (data part)
- Program part may compose of several procedures (some user-defined, some library)
- Virtual address space is divided into segments (procedures, modules, data, etc.)

Segmentation

- Each segment is loaded in memory when needed in the execution of an instruction
- Segments are of different sizes
- Quite similar to MVT, but whole program is not loaded in memory
- Programmer has the control how the program is divided into segments

Segmentation

- Virtual address (*s*,*d*)
  - *s* – segment number
  - *d* – address within the segment (offset from the start of the segment)
- Segment number is used as index for locating an entry in the segment table
- Each entry in the segment table contains the base location (*b*) and the length of the segment (*l*)

Segmentation

- Offset (*d*) of the logical address must be between 0 and the segment limit (*l*).
  - If *d>l*, error interrupt (address error) occurs
  - Otherwise, d is added to the segment base to produce the physical address

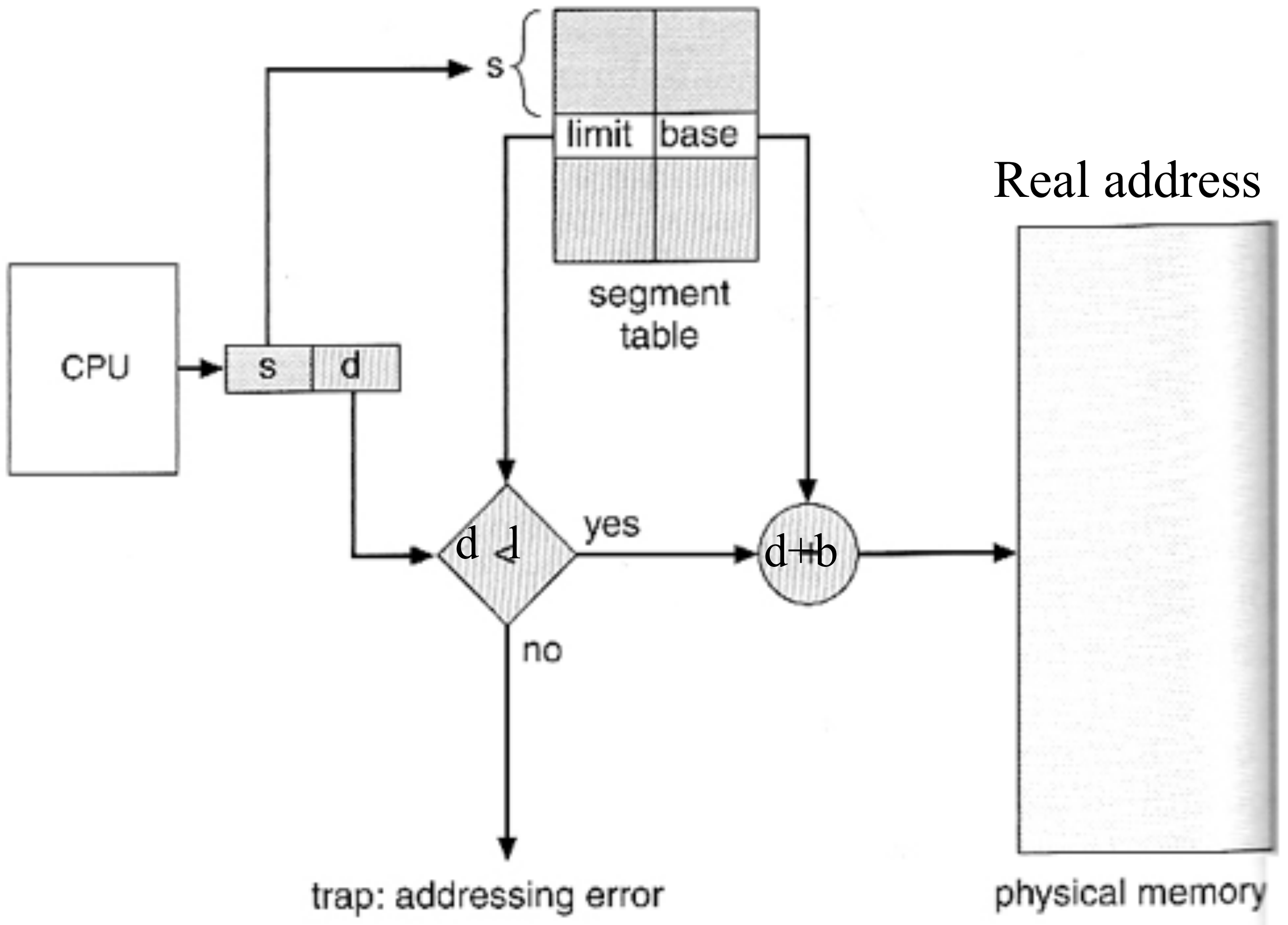
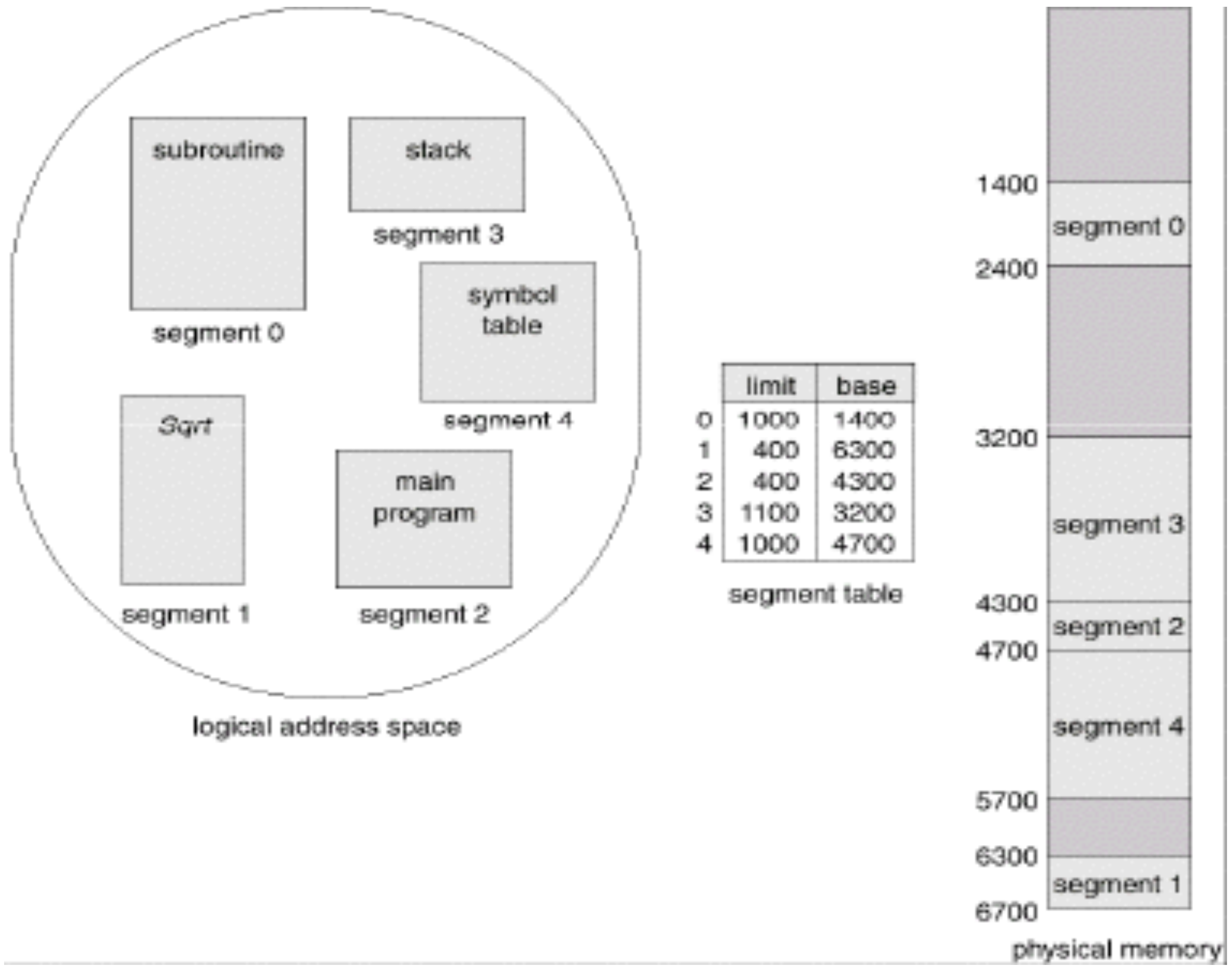


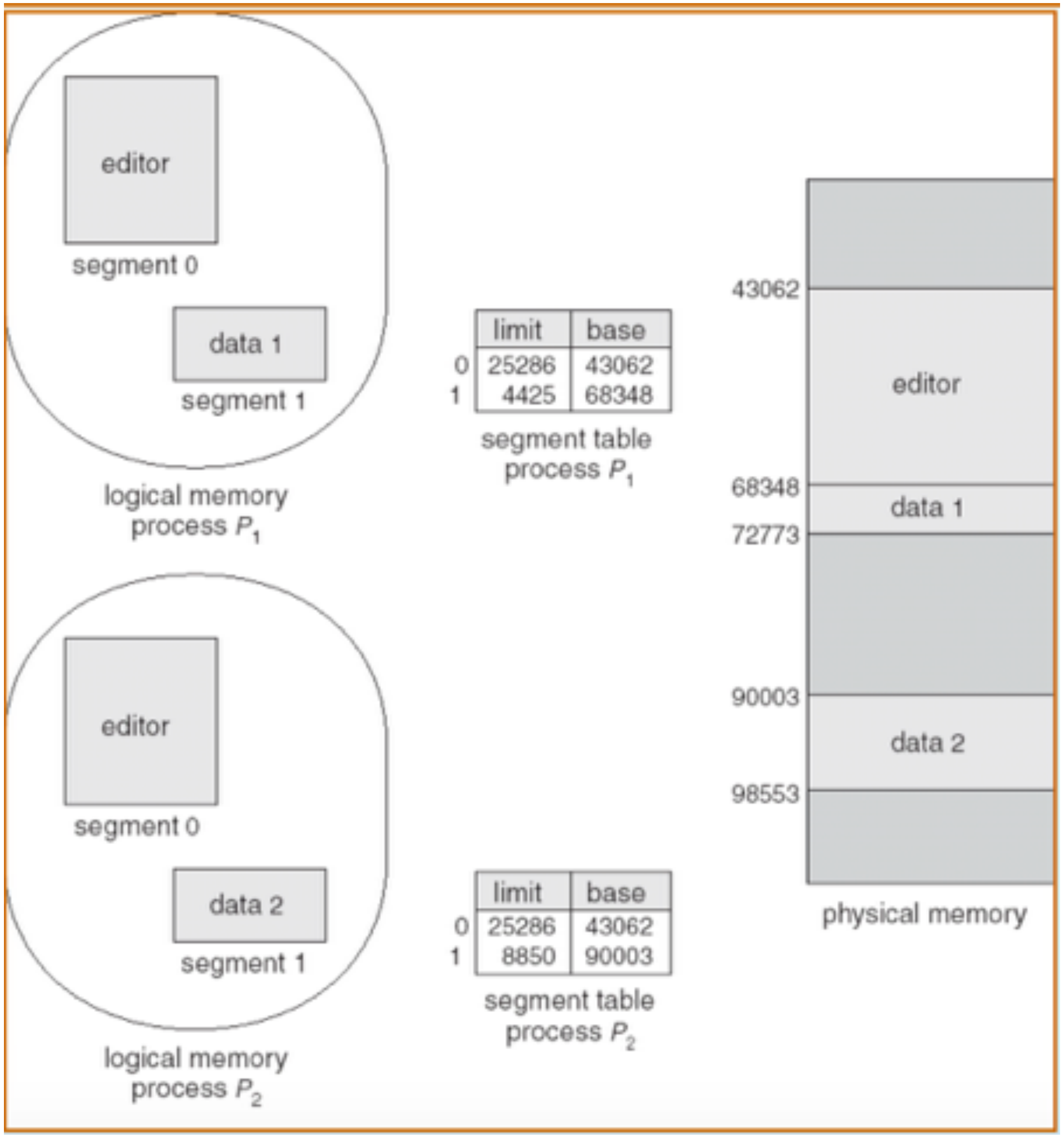
Figure 8.19 Segmentation hardware.

Address translation in segmentation



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

An example of how segments are allocated memory



Sharing of segments

Segmentation

Problem:

1. External fragmentation

   Consequence of segments being of different sizes...

2. Deadlock

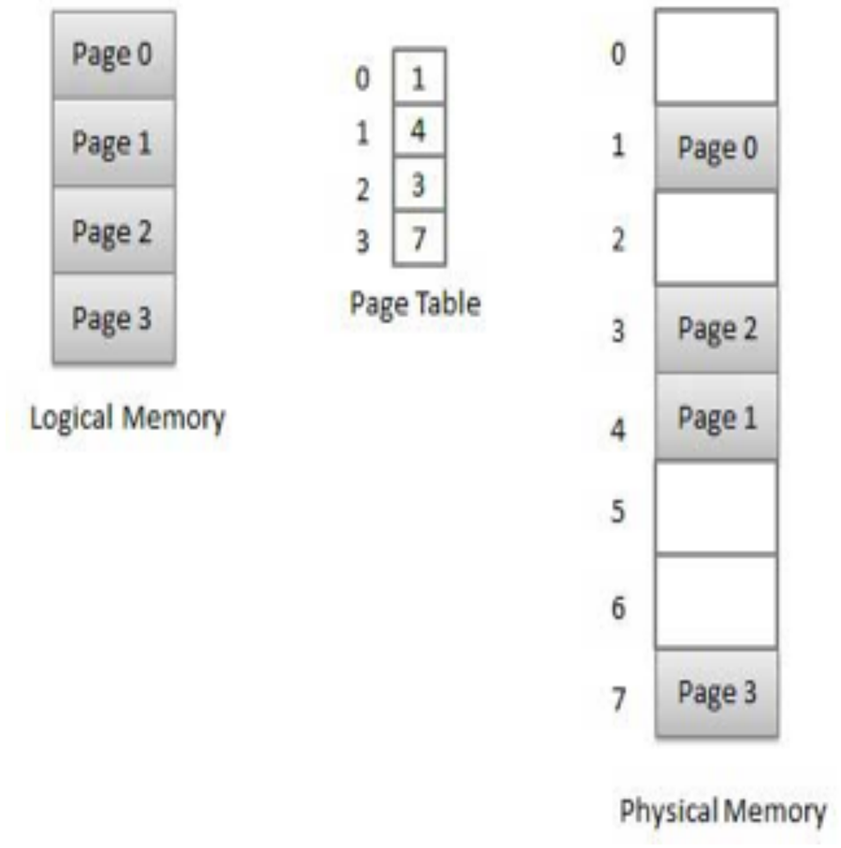
   Although all processes have some of their segments in memory, the number of segments may not be enough for a process to continue

Paging

- Virtual address space is divided into pages of equal sizes
- Main memory (or physical address space) is divided into *page frames* of the same size as the pages.
- Page frames are shared between the processes currently in the system (READY and RUNNING processes)

Paging

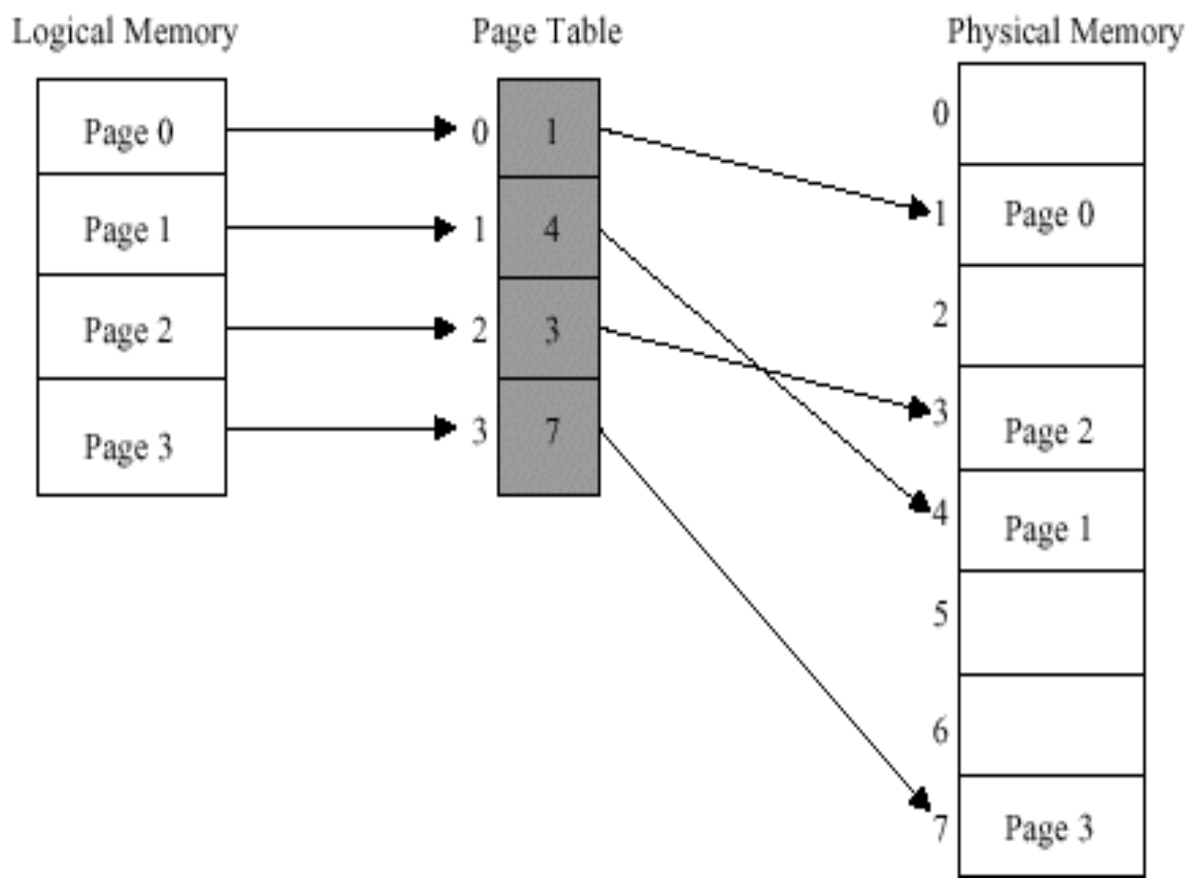
In paging, it is the OS that's responsible for dividing the programs into pages.



Paging

- Virtual address (*p*,*d*)
  - *p* – page number
  - *d* – page offset
- Page number (*p*) is used as an index to locate an entry in a page table that contains the base addresses of the page frames being occupied by the pages.

Paging


Logical Memory
Page Table
Physical Memory
Page 0
Page 1
Page 2
Page 3
0
1
2
3
1
4
3
7
0
1
Page 0
2
3
Page 2
4
Page 1
5
6
7
Page 3

Paging

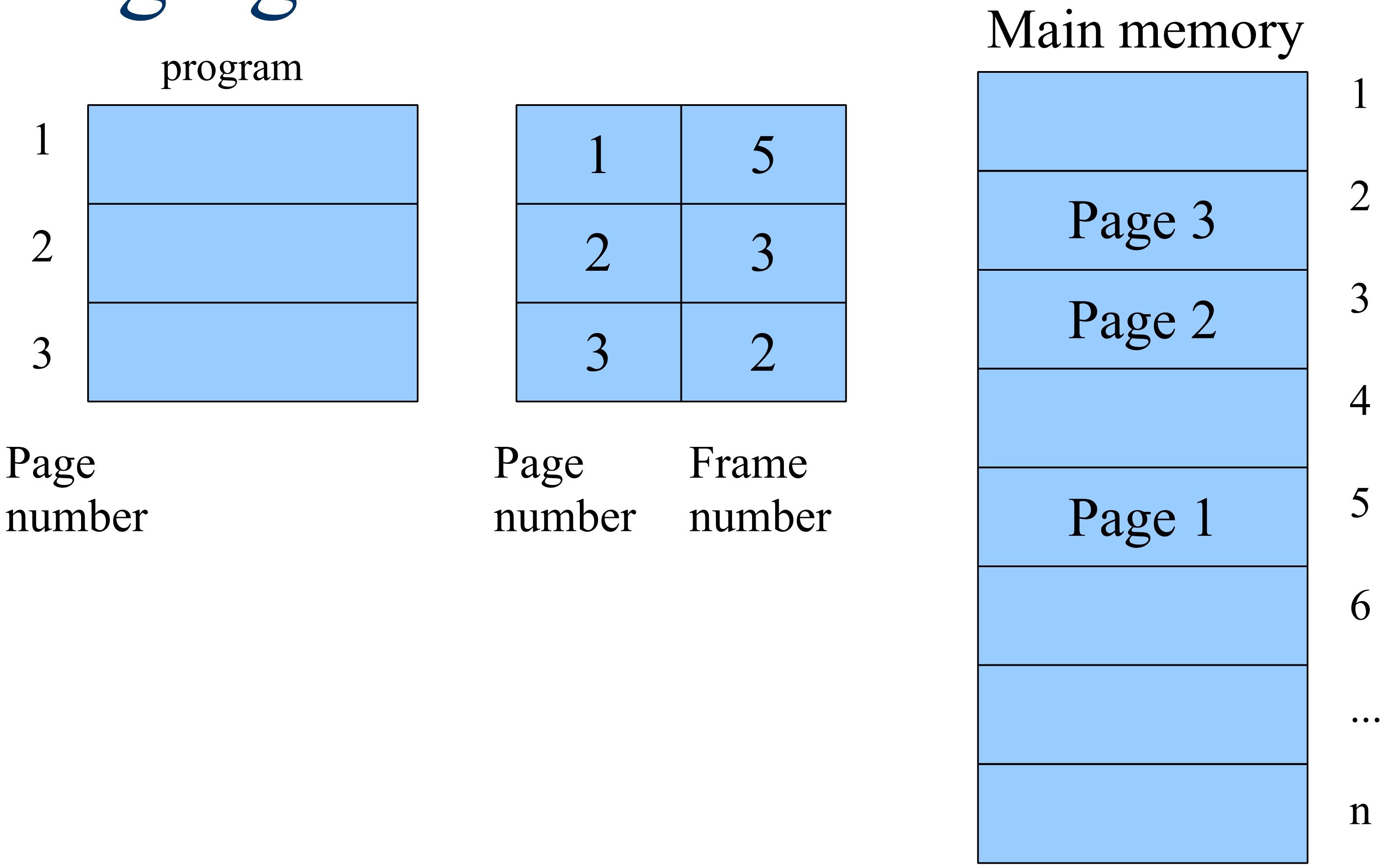
Functions of paging mechanisms:

1. To perform the address mapping operation; to determine which page a program address refers to, and to find which page frame, if any, the page currently occupies

Paging

2. To transfer pages from secondary memory to main memory when required, and to transfer them back to secondary memory when they are no longer being used

Paging



Page number	Frame number
1	5
2	3
3	2

An example of how pages are allocated to page frames

Paging

- Address translation in paging occurs every time access to the memory is made.
- Most systems opt for a hardware implementation of the page table.
- Use a set of dedicated registers to store the entries of the page table, but use of register is acceptable when page table is relatively small

Paging advantages

1.) Completely avoids the problem of external fragmentation (since memory is allocated in fixed-sized blocks & there is no need to place them in memory contiguously). But there is still some form of internal fragmentation (minimal, though)

Paging advantages

2.) Increases potential for multiprogramming; no need to load the whole user program, just pages that are needed. Deadlock is also possible in paging, occurs when pages from a program are shuttled back and forth to and from memory

Paging advantages

3.) A page can be loaded anywhere in memory, thus, easier to implement swapping in and out of pages in memory

Paging

Issue: Page Size

- Too small – minimize amount of internal fragmentation, program will need many pages in memory before it can execute
- prone to page fault errors or deadlock
- large page table will be needed by the system

Paging vs Segmentation

Differences:

1. Segmentation – division into segments of a program involves the programmer

   Paging – division into pages is transparent to the programmer

2. Separate compilation is allowed in segmentation, not allowed in paging

Paging vs Segmentation

Evaluate the virtual addressing schemes according to:

1. memory utilization
2. memory allocation
3. sharing of code

Memory Utilization

Paging achieves very good memory utilization. May have internal fragmentation but wastage not that much.

Memory Utilization

Efficient memory utilization is more difficult in segmented systems. Although, one can share segments to avoid loading identical copies of some programs. But since segments may vary in size, it's difficult to efficiently manage memory.

Memory Allocation

Good in paging; simple, since in paging you have a pool of identical page frames. Problem when new page frames comes much faster than the voluntary release of page frames; preemption of page frames has to be done to allocate new page frames coming in.

Memory Allocation

More difficult in segmentation; best-fit, first-fit, worst-fit are all well as long as there as “holes” big enough to take a new segment. If external fragmentation is rampant, may resort to compaction, which is very expensive.

Sharing of Code

Difficult in paged systems.

Perfect for segmentations. Can specify allowed usage on it, highly sharable.

Paged Segmentation

- Advantages of the ff. virtual memory techniques:
  - Paging – eliminated external fragmentation
  - Segmentation – logical division of programs and sharing of code readily available

Paged Segmentation

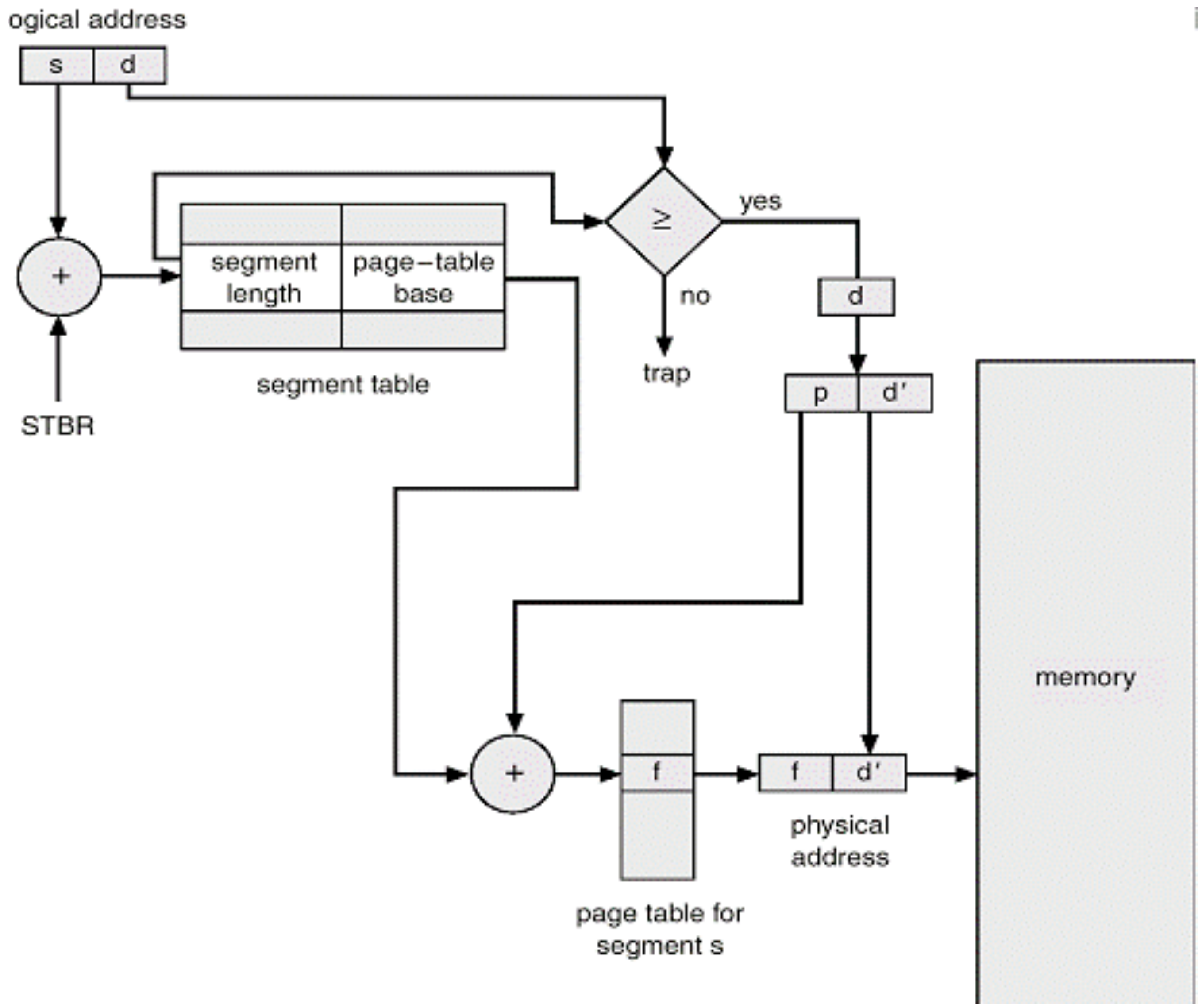
A virtual memory technique wherein we can divide segments into pages, to combine the advantages of the paging and segmentation.

Paged Segmentation

- Virtual address (*s*, *p*,*d'*)
  - *s* – segment number
  - *p* – page number
  - *d'* – displacement within the page
- 2 maps for each user process – 2 stages in the mapping procedure
  - 1st map describes the segments, detailing their allowed usage, size, etc.

Paged Segmentation

- cont...
  - Where in the system memory their page-map tables have been placed
  - For each segment there is a page map defining the page frames on the backing store and, if allocated, in main memory that corresponds to each page



Paged Segmentation

Memory Allocation Strategies

1. Placement Policies

   - where in main memory should incoming processes be placed

2. Replacement Policies

   - which process should be taken out of memory to make room for incoming process

3. Fetch Policies

   - where to obtain the code of the next process for insertion to the main memory

Placement Policies

- In non-paged systems, placement policy must maintain a list of the locations and sizes of all unallocated parts of memory (free list).
- Decide which free memory should be used, and update the free list after each insertion

Placement Policies

Placement Algorithms:

1. Best-fit – chunk of memory in the free list which produce the smallest left-over
2. Worst-fit – largest left-over
3. First-fit – first chunk of memory in the free list that is big enough to accommodate the requesting process

Placement Policies

Example:

- Free list contains list of free memory sizes: 10k, 25k, 17k, 20k, 50k, 13k, 5k
- Suppose an incoming process requires 18k of memory. The ff. will be the free memory that will be allocated:
- Best-fit – 20k
- Worst-fit – 50k
- First-fit - 25k

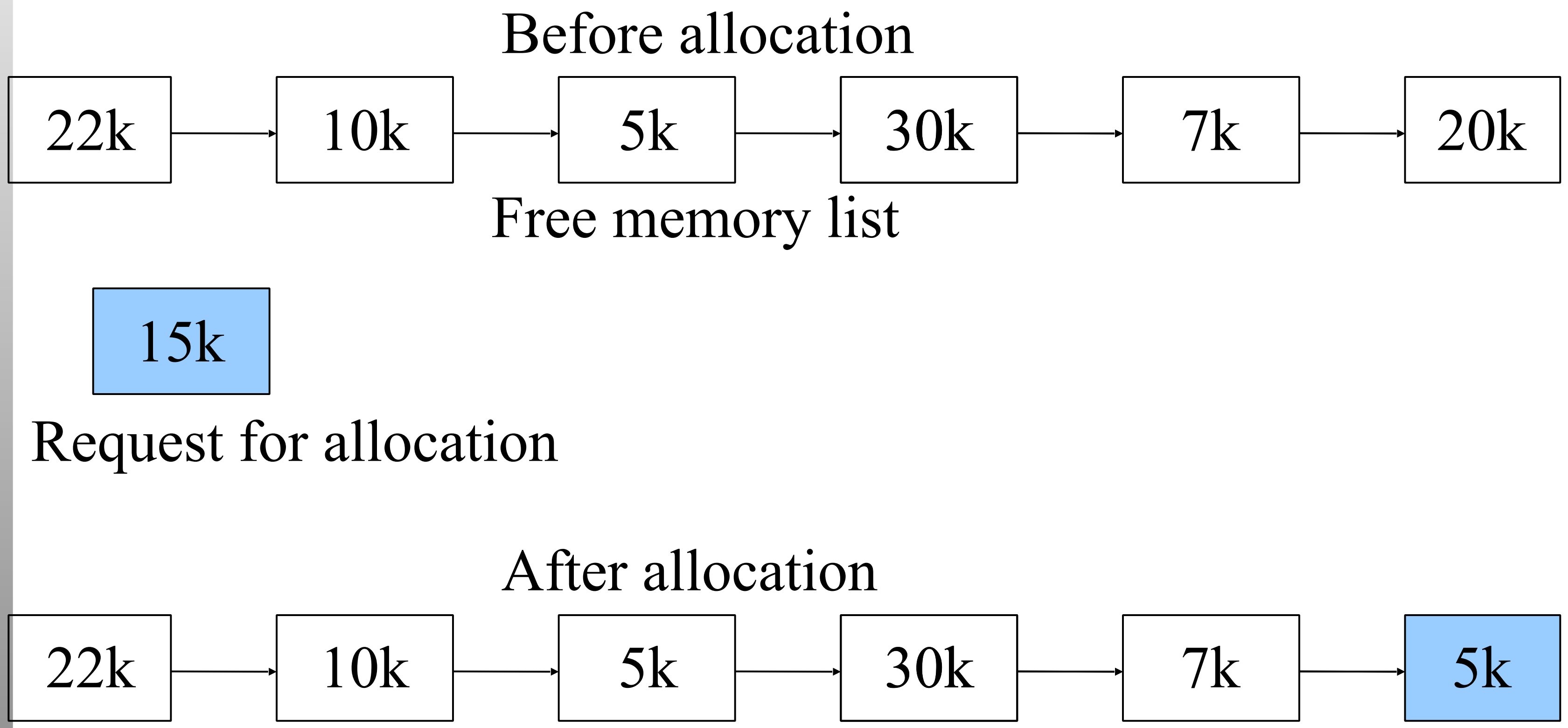
Placement Policies

It is possible for best-fit and first-fit to be one and the same if we have a sorted free list.

If free list sorted in increasing memory size, first-fit will also be the best-fit.

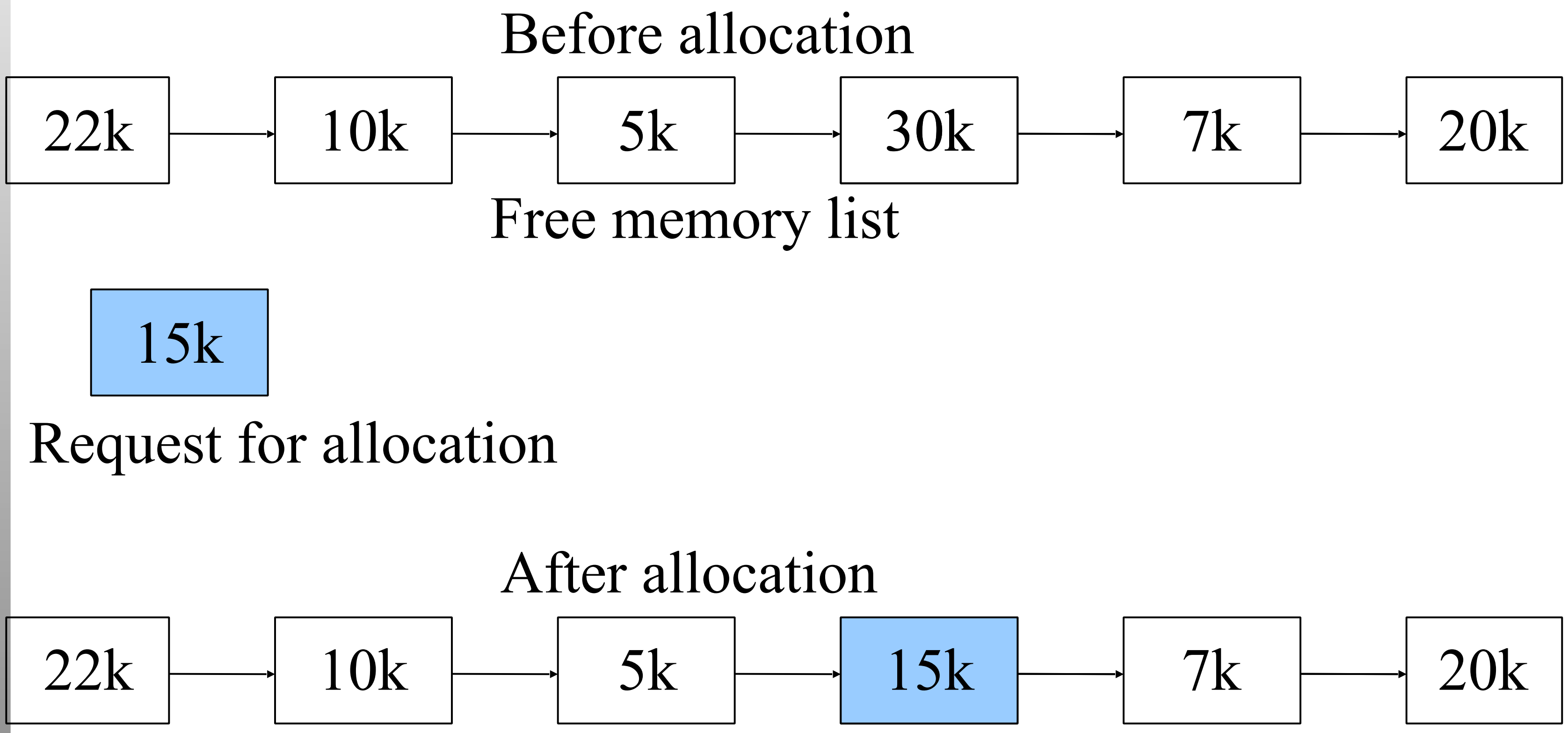
If free list sorted in decreasing memory size, first-fit will also be the worst-fit.

Placement Policies



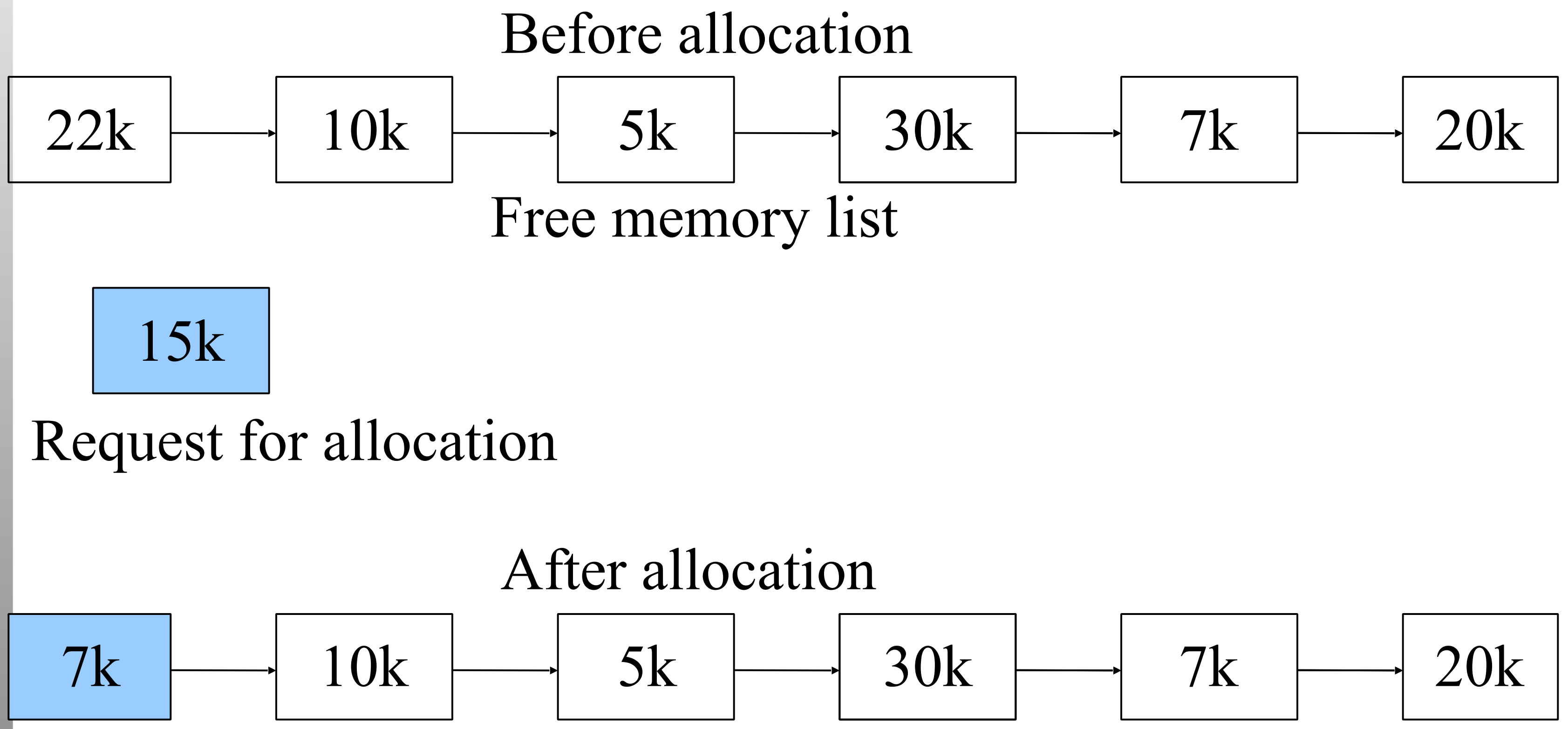
Best-fit algorithm

Placement Policies



Worst-fit algorithm

Placement Policies



First-fit algorithm

Replacement Policies

In non-paged systems...

Replace a segment that is least likely to be referenced in the immediate future; consider that segments vary in sizes, so sizes should be considered.

Replacement Policies

- In paged systems...
  - Replace pages that are not going to be referenced for the longest time in the future; just infer from past behavior what future behavior is likely to be

Least Recently Used (LRU)

Replace the page that has not been recently been used.

Every time a page is referenced, it is stamped with the time it was referenced. Selecting next page to be replaced is the page with the oldest time stamp.

Least Recently Used (LRU)

Simple, but expensive to implement properly. Maintain a sorted linked list of all pages in memory.

- Newer time stamps are placed at the tail of the list
- Older time stamps placed at the head of the list
- Every time a page comes in memory and a victime page is to be replaces, the tip of the head of the queue is selected

Least Frequently Used (LFU)

Replace the page that has been used less frequently during some immediate preceding time interval.

A *use counter* is associated with each page in memory. *Use counter* is incremented when a reference to it is made. Page with lowest *use counter* is replaced when a request for memory is made.

Least Frequently Used (LFU)

Maintain a sorted linked list of all pages in memory.

- Higher use count are placed at the tail of the list
- Lower use count placed at the head of the list
- Every time a page comes in memory and a victim page is to be replaces, the tip of the head of the queue is selected

Not Recently Used (NRU)

Each page is associated with a reference bit and modified bit (dirty bit).

- Initially the reference and modified bits of all pages are 0
- when a page is referenced, reference bit set to 1
- when a page is modified, modified bit set to 1
- when a page is about to be replaced, find a page that has not been referenced, otherwise replace a referenced page

Not Recently Used (NRU)

- If a page has been referenced, check if it has been modified
- if not modified, replace it; otherwise replace a modified page

Not Recently Used (NRU)

Four classes of pages:

Class 1: Not referenced; not modified

Class 2: Not referenced; modified

Class 3: Referenced; not modified

Class 1: Referenced; modified

Not Recently Used (NRU)

Last priority: Class 4 pages

Why? When they are selected as victims, there is a need to write them on secondary storage because their contents were modified. Unmodified page victims need not be written back in secondary storage, so they can be handled faster.

First-in-first-out (FIFO)

- Replace the “oldest” page
- To lessen overhead cost (which is apparent in previous methods discussed).
- When a page is loaded, it is time stamped, then replace oldest page in memory.
- Maintain a queue.

First-in-first-out (FIFO)

- Shortcomings: this policy disregards usage of a particular page
- Resolution: Implement FCFS with another method (example NRU)

Random

Simply replace a page at random.

Good, since it has no overhead cost.

So each page in memory has an equal probability to be replaced.

Page frame replacement policies

Global replacement

- Allows selection of a victim from all the set of page frames even if the frame is currently allocated to another process
- It is allowed for one process to take the page frame allocated to another process

Page frame replacement policies

Local replacement

- Allows only that each process selects a victim from out of the frames allocated to that process
- # of frames allocated to a process does not change
- Lesser flexible than local replacement

Fetch Policies

- Fetch policies determine when to move a block of information from secondary memory to main memory
- Policies for choosing a block depends on whether the blocks are pages or segments
- *Anticipatory* or *demand*

Thrashing

- A situation where the system spends more time shuttling the parts of a program (pages or segments) from main memory to backing store than in executing the program.
- One cause: multiprogramming of too many jobs, leaving an insufficient number of main memory page frames being allocated to each individual job

Thrashing

Often occurs in system where local replacement is used.

Can be minimized if the replacement policy used is global. If one process starts to thrash, it can steal frames from one process causing them to avoid thrashing.