CMPT 300
Operating System I
Memory Management – Chapter 9

Dr. Hazra Imran

Summer 2022

Outline

- Introduction
- Virtualization
- **Registers**
- Segmentation
- Swapping
- Fragmentation
- Free Space Management
- Paging

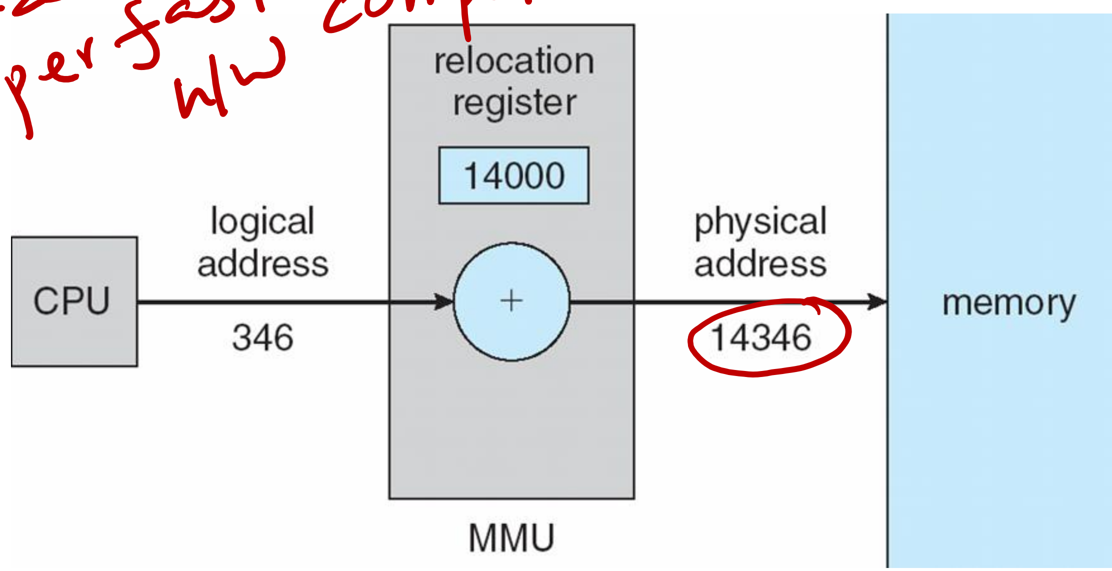
Base and Limit Registers

virt addr — phy address

A **base** and a **limit** register define the logical address space

- Physical address = Base + Virtual Address
  - Base: starting address in memory (physical)
  - Virtual address: offset into the address space

1 specialized
2 super fast h/w components

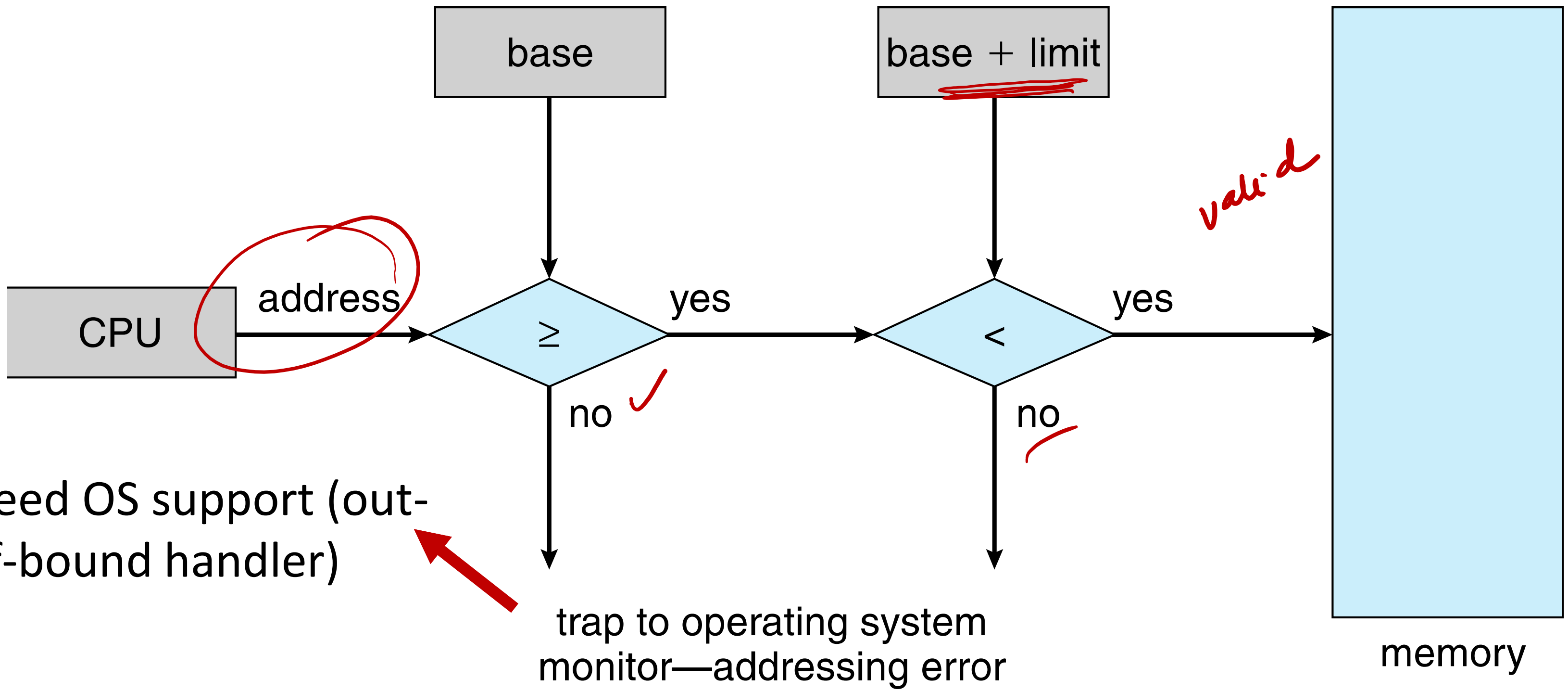


- Supported by memory management unit (MMU)
- Privileged instructions for modifying these registers

Base and Limit Registers

A **base** and a **limit** register define the logical address space

- Physical address = **Base + Virtual Address**
  - Base: starting address in memory (physical)
  - Virtual address: offset into the address space



Need OS support (out-of-bound handler)

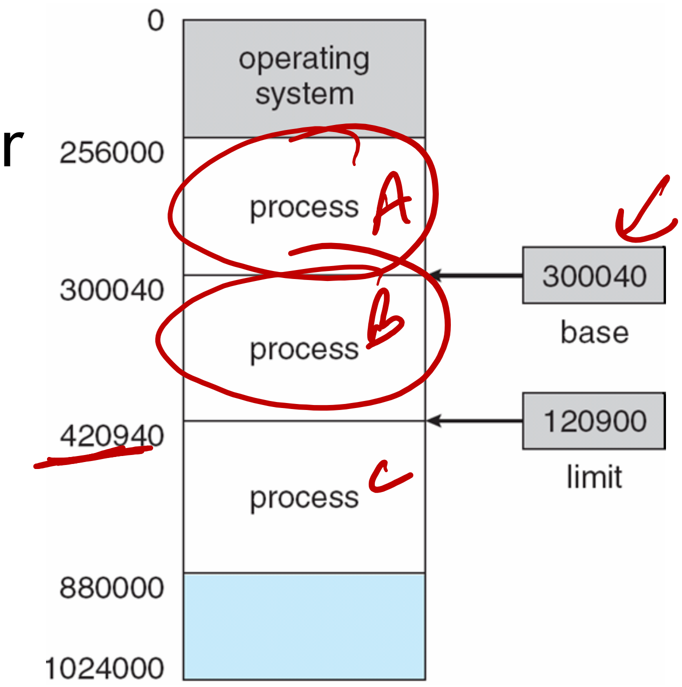
Base and Limit Registers

A **base** and a **limit** register define the logical address space

- Physical address = Base + Virtual Address

Limitations:

- Logical address space cannot be larger than physical address space
  - I.e., entire process fits in memory
- Logical address space must be stored **contiguously**
- May waste space
  - Space between heap and stack



Better alternative: segmentation

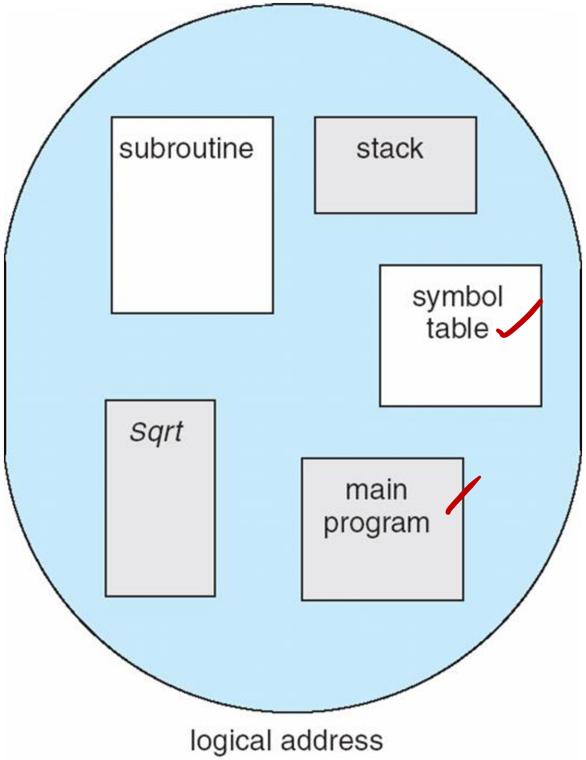
Outline

- Introduction
- Virtualization
- Registers
- **Segmentation**
- Swapping
- Fragmentation
- Free Space Management
- Paging

Segmentation: Generalized Base+Limit

Programmer's view of memory:

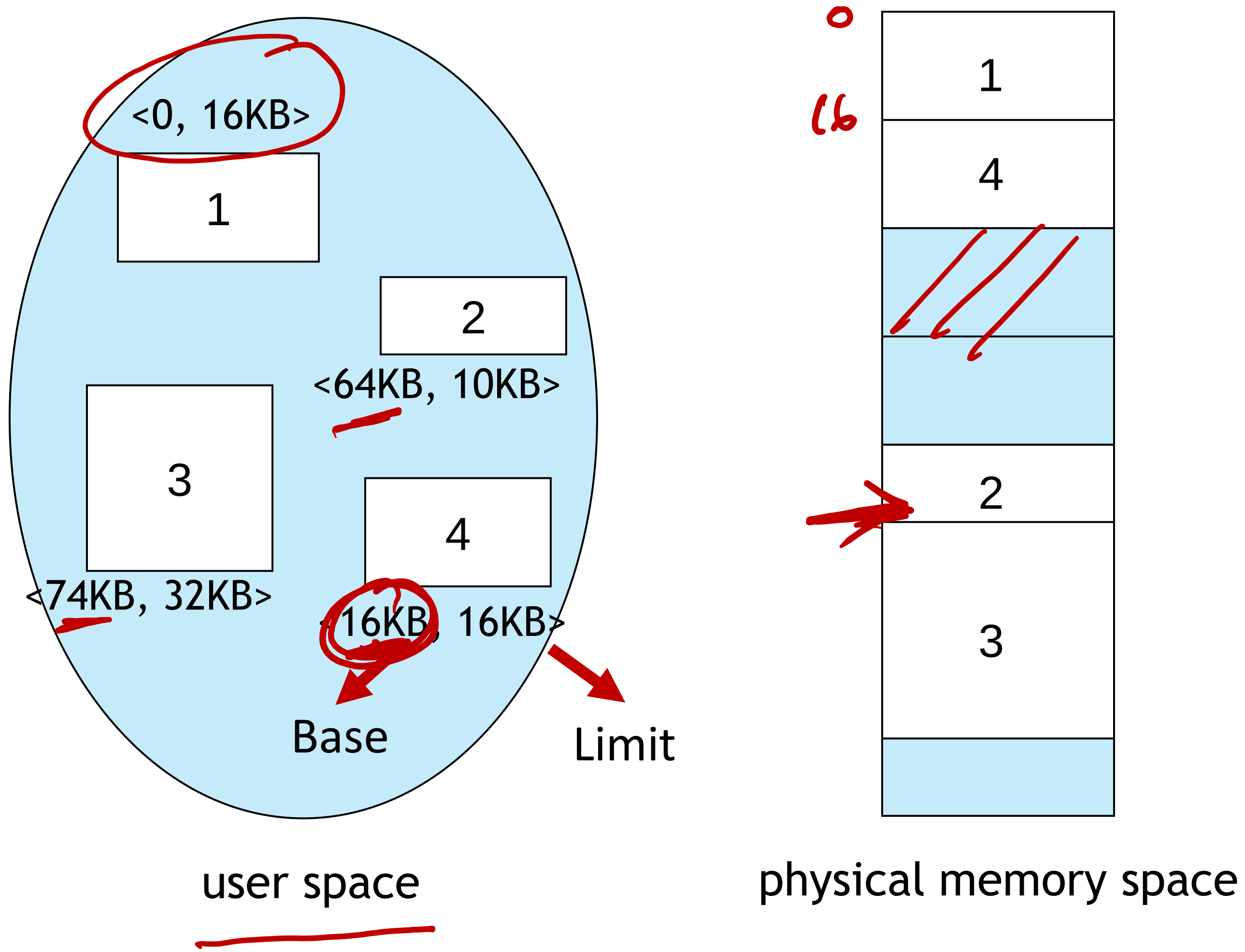
- A program is a collection of segments
- A segment is a logical unit such as:
  - Main program
  - Functions, procedures
  - Method
  - Object
  - Local variables, global variables
  - Stack
  - Symbol tables
  - Arrays



Segmentation: Have a base + limit pair per logical segment

Segmentation: Generalized Base+Limit

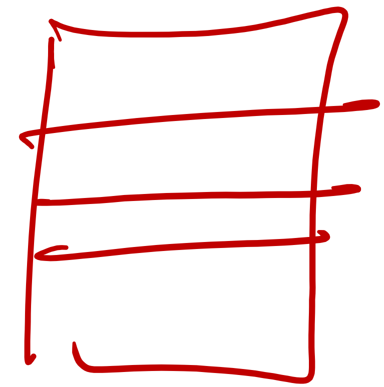
Have one base + limit pair per logical segment



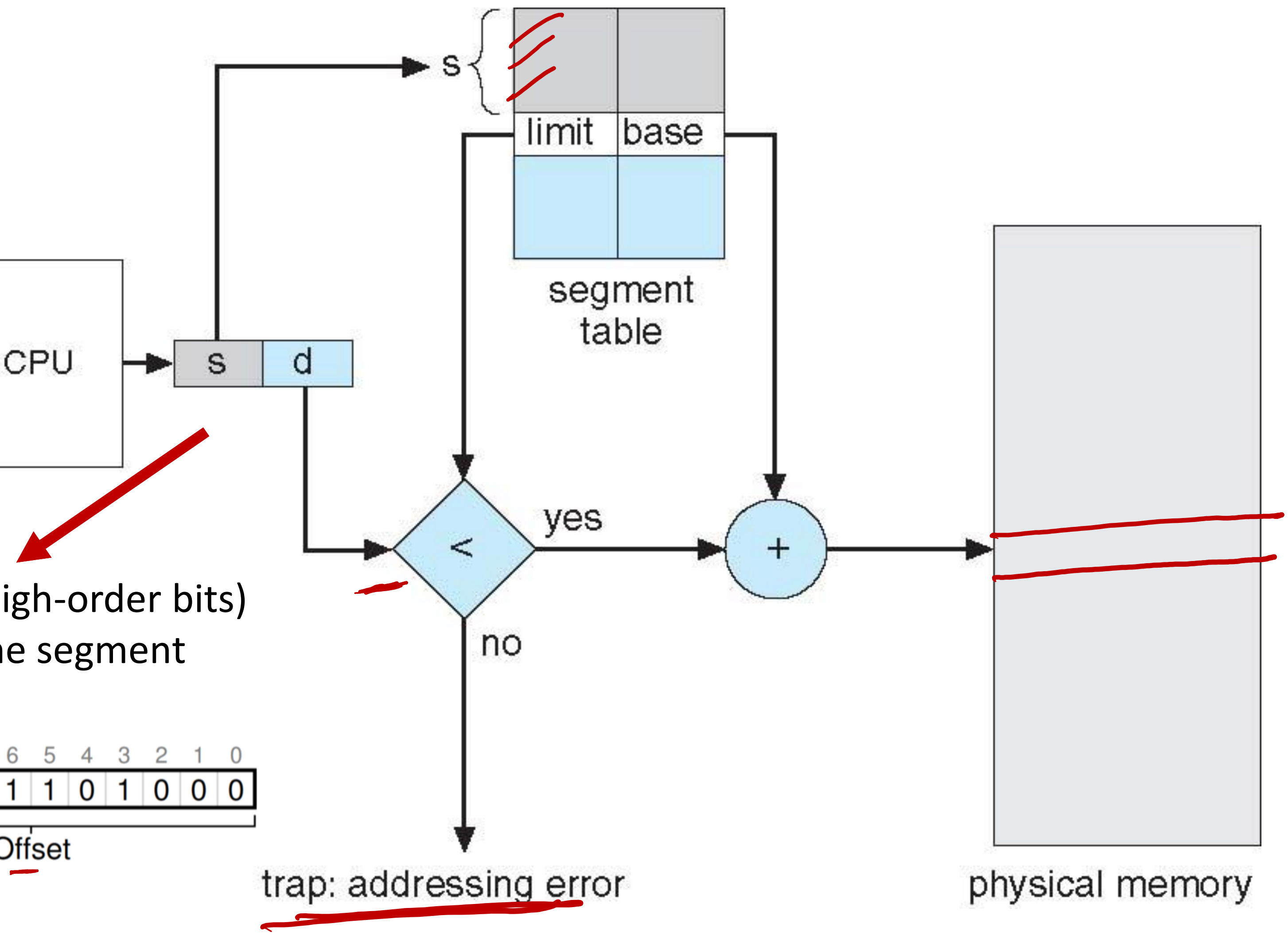
Segmentation Architecture

- Logical address has the form: <segment-number, offset>

- Segment table
  - Maps logical address to physical address
  - Each entry has
    - Base: starting physical address of segment
    - Limit: length of segment (offset)



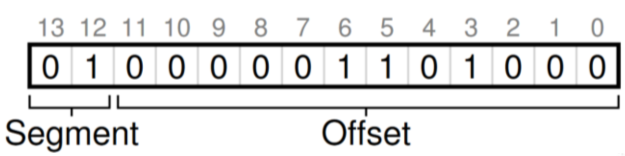
Segmentation Hardware



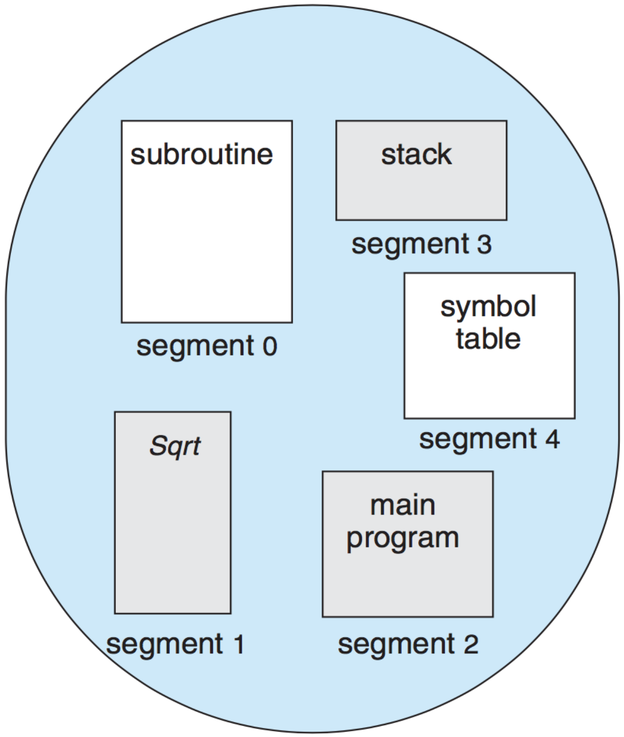
s: segment # (high-order bits)

d: offset into the segment

Example:



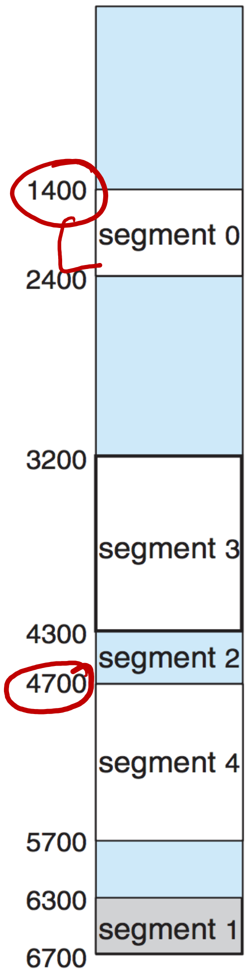
Segmentation Example



logical address space

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

segment table



physical memory

Issues in Segmentation

Issue 1: Need to save and restore segment registers upon context switch

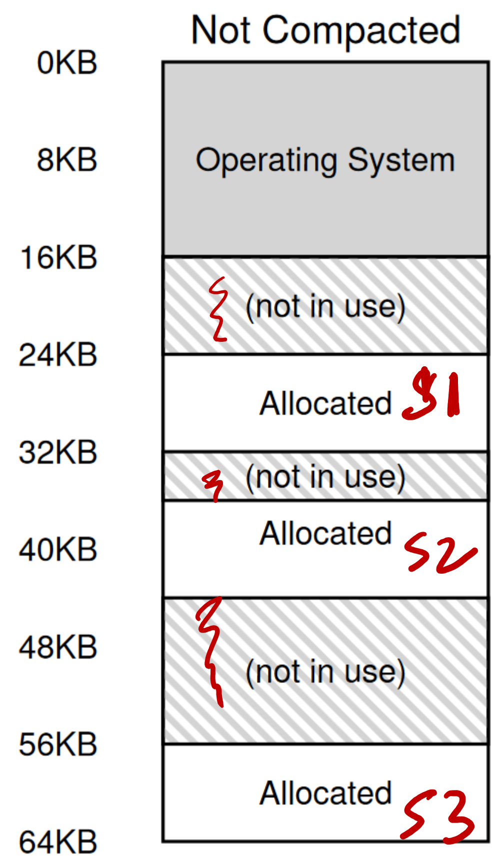
Issue 2: Managing free spaces:

- Segments vary in length, OS needs to find free space in memory for new address space

- This can lead to **external fragmentation**
  - Physical memory becomes full of small holes of free spaces
  - hard to allocate new segments or grow existing ones

- **Solutions:** free space management schemes

Fragmentation example:



Summary

- Address space
  - Easy-to-use abstraction of memory
  - Gives processes the illusion that it owns the entire system
  - Provides isolation and protection between processes

- Basic approach: base and limit registers
  - Base: starting physical address
  - Limit: size of segment
  - Need privileged instructions to modify base and limit registers
  - Various limitations

- Segmentation: generalized base+limit registers approach
  - One pair of base+limit registers per segment
  - Main issue: external fragmentation

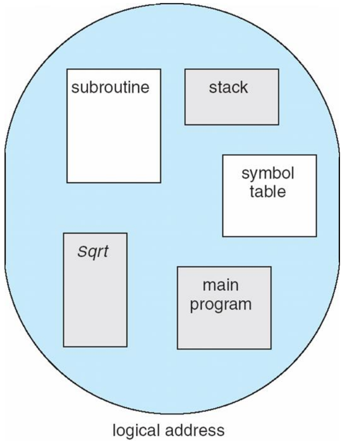
Segmentation

- The process address space is a set of disjoint pieces (text; data; stack; heap), i.e. no overlapping of the pieces

- Some pieces dynamically grow and/or shrink (stack; heap; text if loaded on demand)

- The programmer does not care how pieces are organized in the address space

- These pieces are **Segments**



Segmentation

- The logical address space is a collection of segments. A logical address is decomposed as:
  - A **segment number**
  - An **offset in the segment**

- The compiler/language interpreter handles the segments and the logical addresses are built appropriately

- Typical segments used by a C compiler
  - Text, data, heap ,stacks. standard C library

Segmentation

- Logical address = Segment number + Offset
- Protection mechanisms need to be set up.
- A **segment table** with one entry per segment number
    - Base: Starting address of the segment
    - Limit: Length of the segment
- The segment table is stored in memory
    - A Segment-Table Base Register (STBR): Points to the segment table address
    - A Segment-Table Length Register (STLR): Gives the length of the segment table; Makes it easy to detect an invalid segment offset

MMU and Segmentation

- Segmentation is easy
- Reserve bits (e.g., the left-most ones) in the logical address to reference a segment (the segment bits)
- Lookup the segment table to find out the segment’s base/limit value segmentation

Demand on memory space

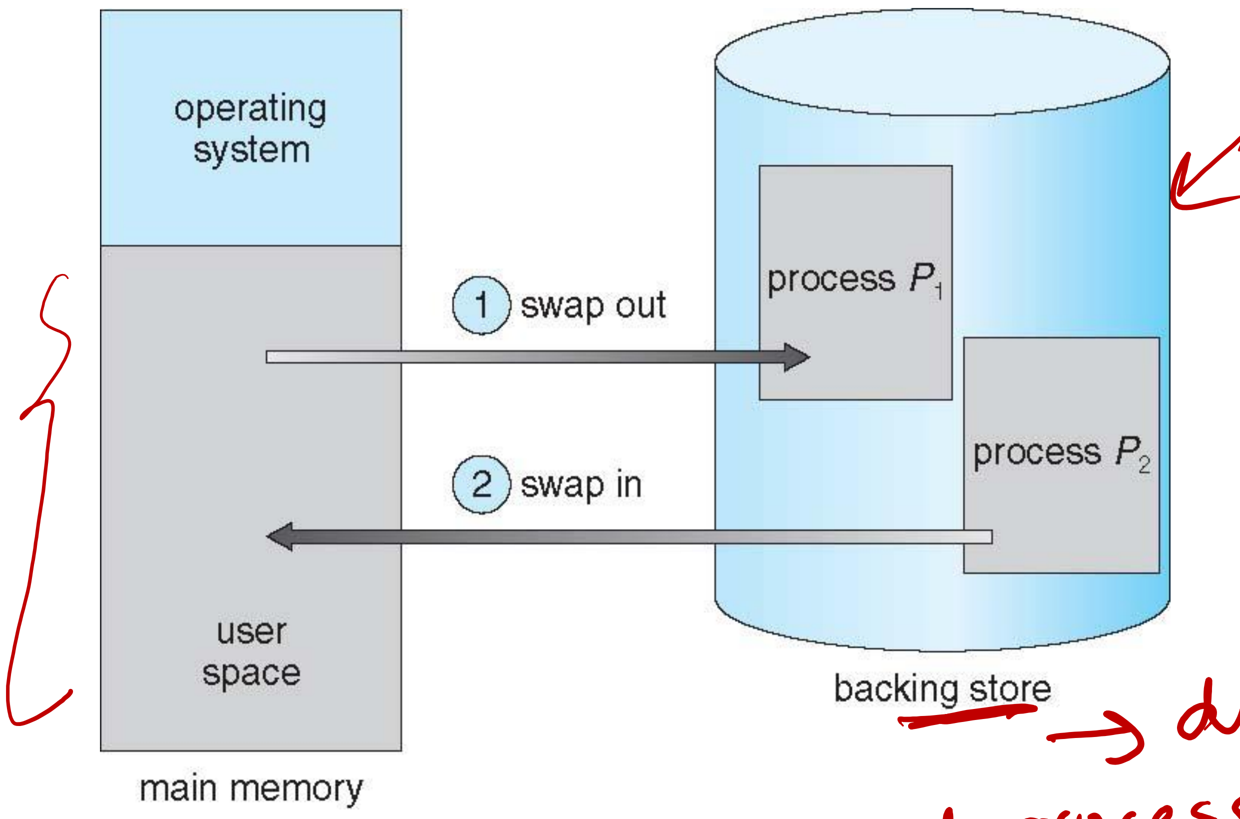
- Considering that multiprogramming enables our computer to simultaneously serve many processes, and memory size is much smaller than disk drive.

- Then what if jointly all processes ask for more memory space than is available?

Outline

- Introduction
- Virtualization
- Registers
- Segmentation
- **Swapping**
- Fragmentation
  - Free Space Management
- Paging

Schematic View of Swapping



→ disk drive

Ready process

memory

b.store

Swapping

- Does the swapped-out process need to swap back into same physical addresses?

- Depends on address binding method
  - Plus consider pending I/O to / from process memory space

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Swapping

- What if we want to load another process that would not fit in memory?
- We must save the address space of one (or more) processes from RAM to a "backing store" (the disk)
- Moving processes back and forth between main memory and the disk is called **swapping**
- When a process is swapped back in, it may be put into the same physical memory space or not
- (But this is no problem....thanks to address virtualization)
- With swapping a process can "be in RAM" or "be on Disk". Therefore, a context-switch can involve the disk!!

Swapping and DMA

- With swapping, at any time the OS could kick a process out of RAM and save it to disk
  - This raises a concern with Direct Memory Access (DMA)

- Consider a process that has initiated a DMA operation and is swapped to disk
  - The DMA controller may have no idea and happily continues to write data (into some other process' address space, which has replaced that of the one that was swapped out!)

  - Operating systems must deal with this (because DMA is so useful we can't leave without it)
  - One option could be: never swap a process engaged in DMA
  - In fact, OSes do something else ("paging", next topic)

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But the modified version is common
    - Swap only when free memory is extremely low

The Bad News about Swapping

- **The disk is slooooooooow** (even if it's an SSD)

- Several ways to cope with slow disks have been used:
  - An OS could swap in/out only processes with small address space (rather than processes with large address space)

  - One can dedicate a disk/partition to swapping (so as to minimize disk seeks on a hard drive)

The Bad News about Swapping

- One approach is to just **not swap.**

- Swapping should be an exceptional occurrence
  - In older OSes swapping was user-directed (e.g., Windows 3.1)

  - Swapping is now often disabled (e.g., on laptops)

  - If the normal mode of operation of the system requires frequent swapping, the system is in trouble (buy more RAM!)
    - But perhaps it's just a temporary rare load spike?

- A key solution is to not swap whole address spaces ("paging")

Where are we?

We now have a whole bunch of **mechanisms:**

- We know how to allocate memory “portion” to each process
- We know how to reduce address spaces
- We know how to swap processes in and out of memory

We now need a policy to decide how to place each portion in memory:

- We want to have as many process address spaces in memory as possible
- We want to minimize swapping

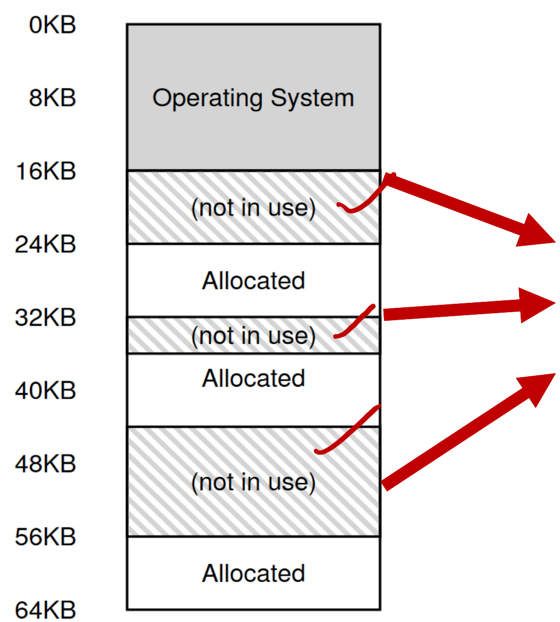
    - What is a good policy?

Outline

- Introduction
- Virtualization
- Registers
- Segmentation
- Swapping
- **Fragmentation**
  - Free Space Management
- Paging

Contiguous Memory Allocation

- Segments vary in length; OS needs to find free space in memory for new address space
- As processes/segments are created and destroyed, memory becomes full of holes of free spaces
- Holes have different sizes and scattered in the memory



Which hole to choose for a process?

Contiguous Allocation Policies

Which hole to choose for a process?

- OS maintains information (lists) on **allocated space** and **free space** (holes)

First-fit: Allocate first hole that is big enough

Best-fit: Allocate 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

Pros and cons of contiguous allocation:

- Pros: simple to implement
- Cons: → Memory fragmentazion

↓
NC