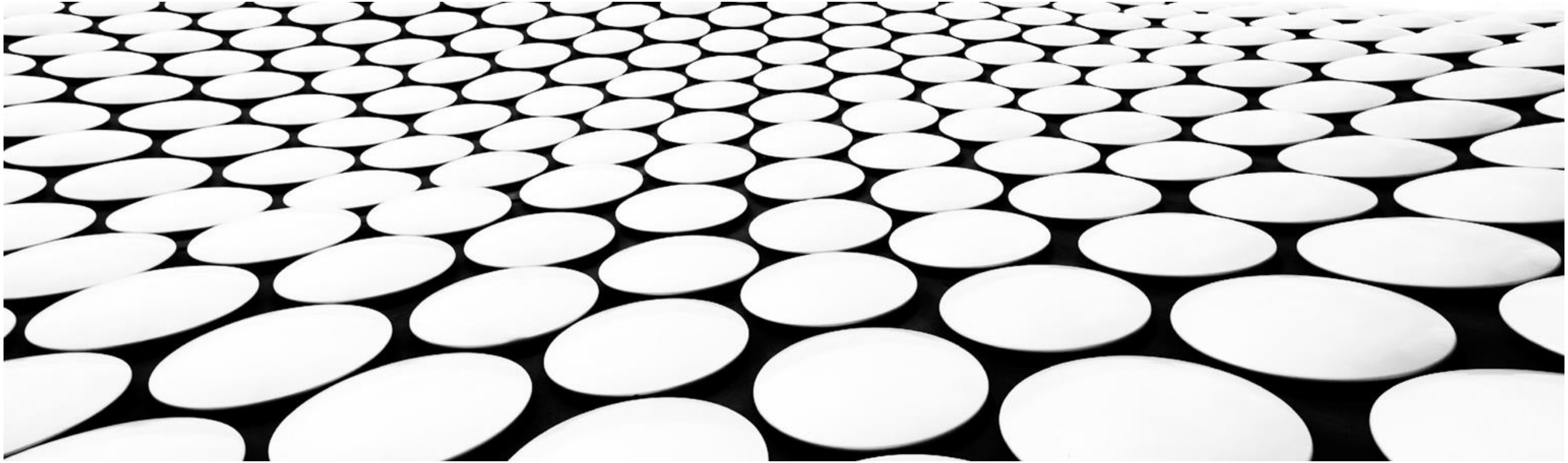
LECTURE 17: **MEMORY MANAGEMENT**

DR. ARIJIT ROY

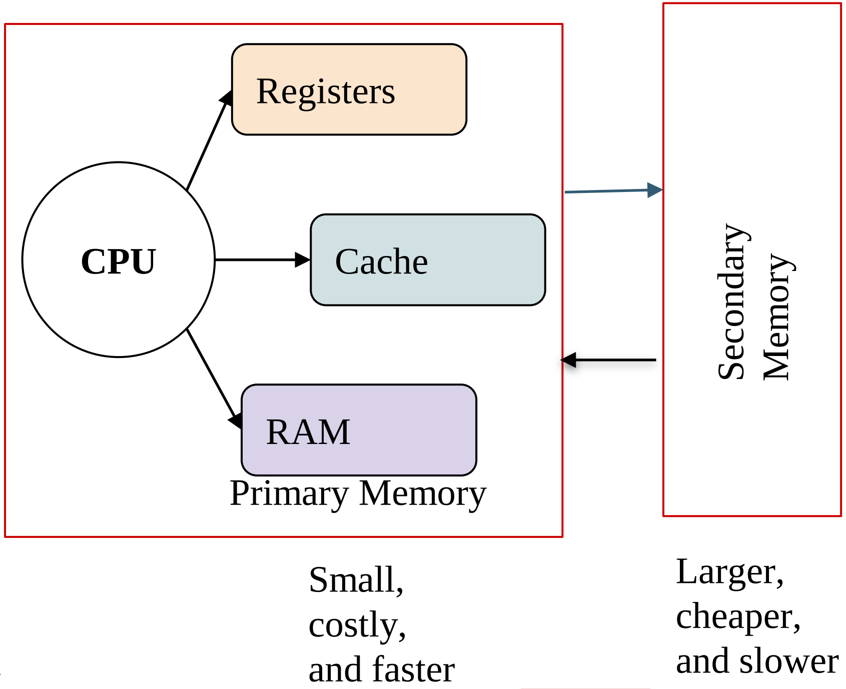
COMPUTER SCIENCE AND ENGINEERING GROUP

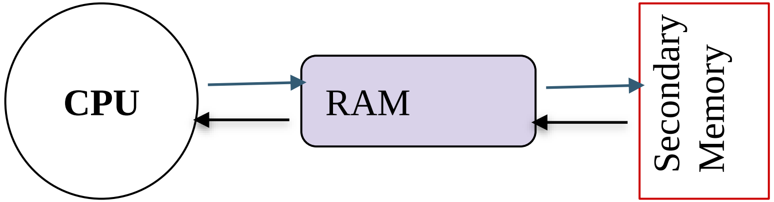
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY



BACKGROUND

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of **addresses + read requests**, or **address + data and write requests**
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

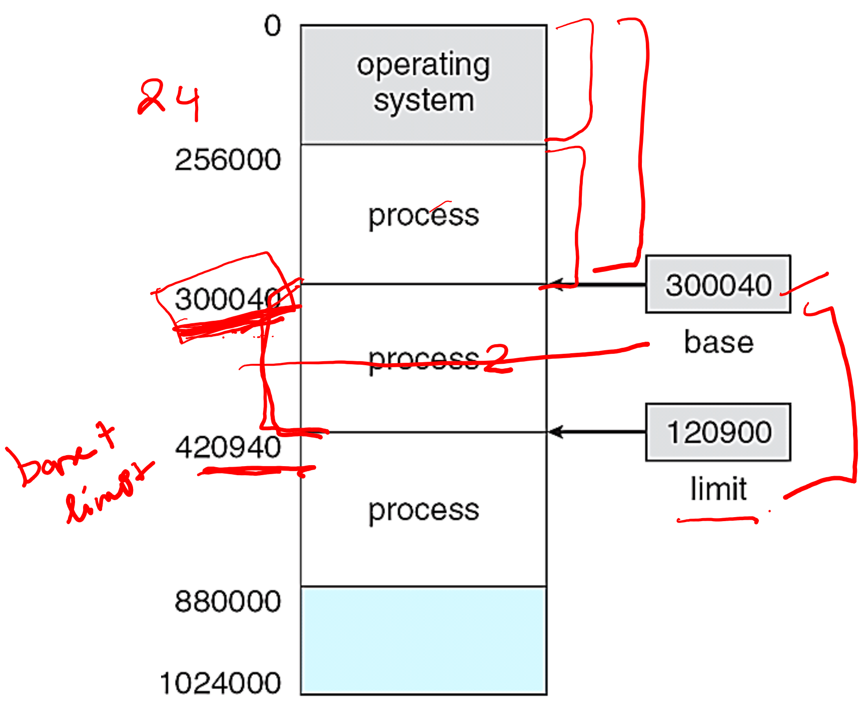




BASE AND LIMIT REGISTERS

Protect unauthorized access

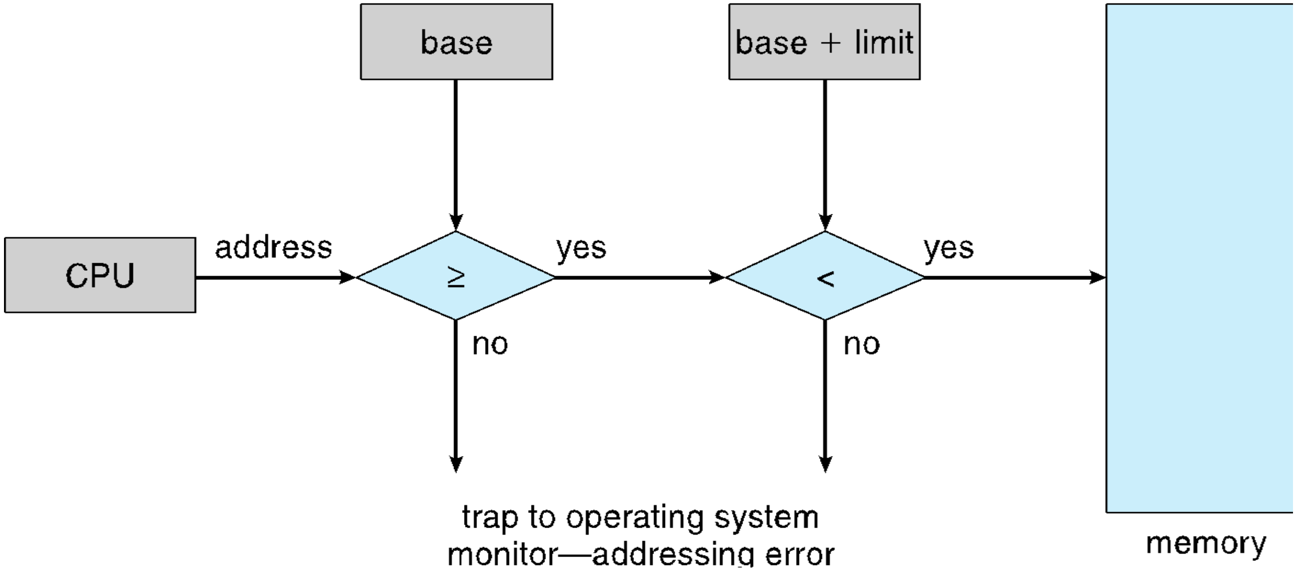
- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Base register: Hold smallest legal physical memory address

Limit register: Specifies the size of the range

HARDWARE ADDRESS PROTECTION



Base and limit registers loaded only by the OS, which uses a special privileged instruction (typically runs in kernel mode)

ADDRESS BINDING

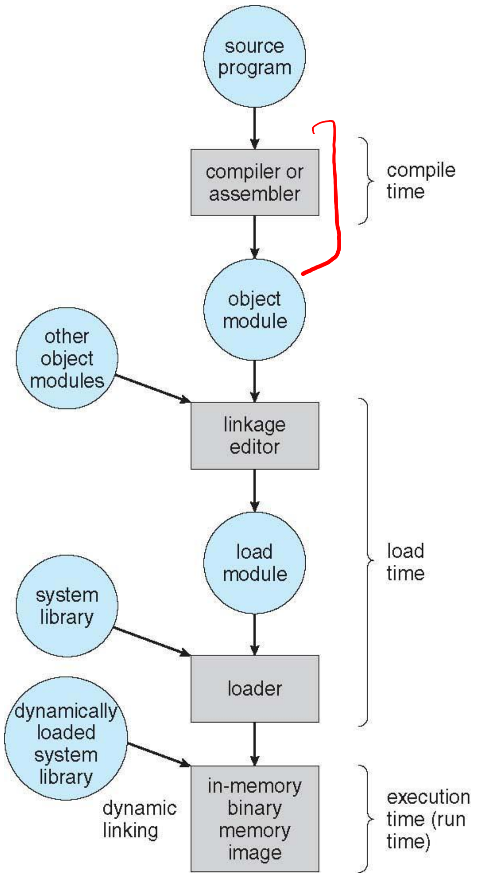
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, not necesassry to be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic int a 70x473
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

BINDING OF INSTRUCTIONS AND DATA TO MEMORY

7 4 3 2 1

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: If not known at compile time where the process will reside in memory, then compiler Must generate **relocatable code**
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

MULTISTEP PROCESSING OF A USER PROGRAM



LOGICAL VS. PHYSICAL ADDRESS SPACE

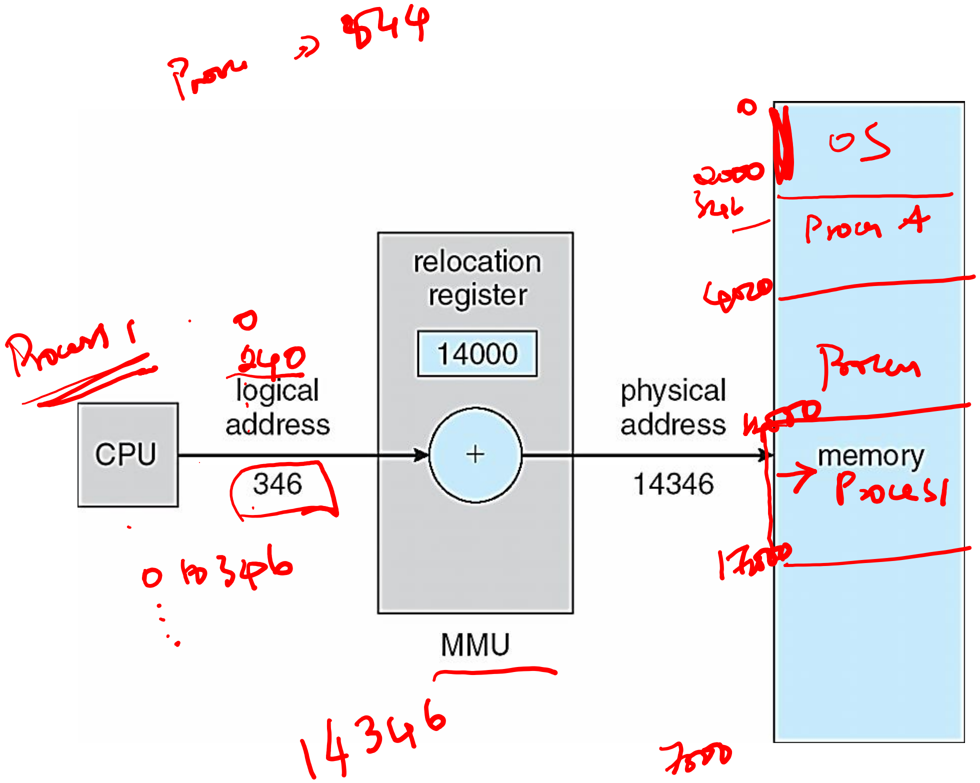
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit- that is the one loaded into the memory-address register of the memory
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

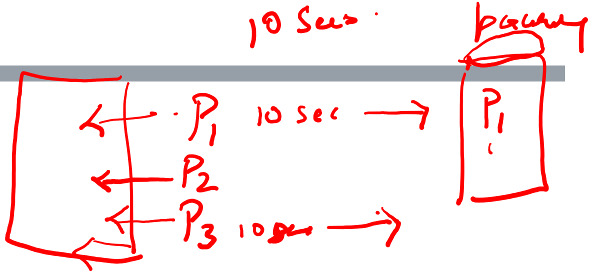
DYNAMIC RELOCATION USING A RELOCATION REGISTER

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



DYNAMIC LINKING

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed



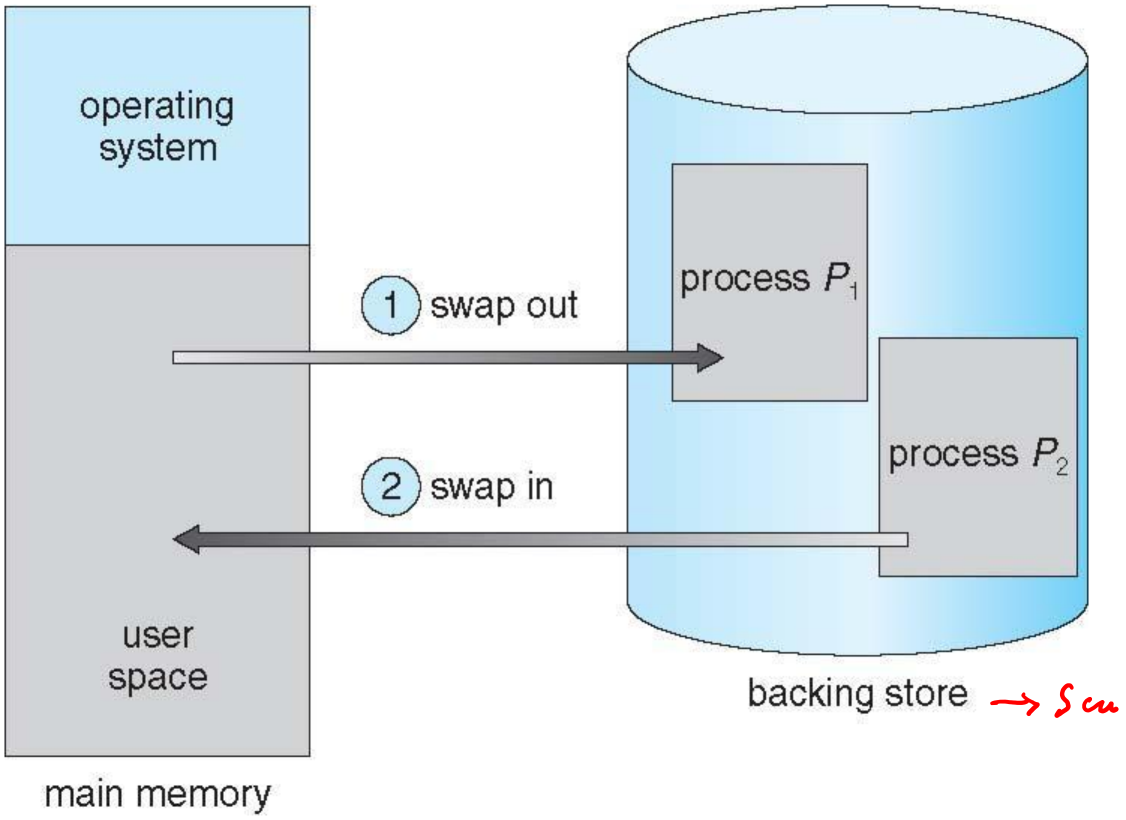
SWAPPING

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- The dispatcher checks whether the next process is in memory. If it is not and no free space than it will swap out a process currently in memory and swap in desired process.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

SWAPPING (CONT.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method -assembly/ loading time and execution time
  - 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

SCHEMATIC VIEW OF SWAPPING

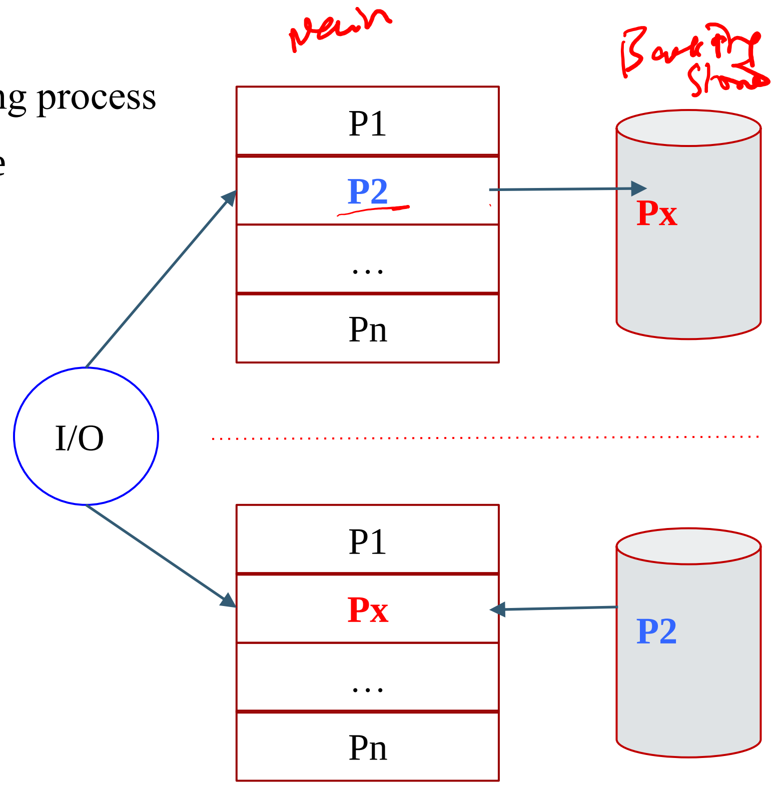


CONTEXT SWITCH TIME INCLUDING SWAPPING

- 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 modified version common
    - Swap only when free memory extremely low



SWAPPING ON MOBILE SYSTEMS

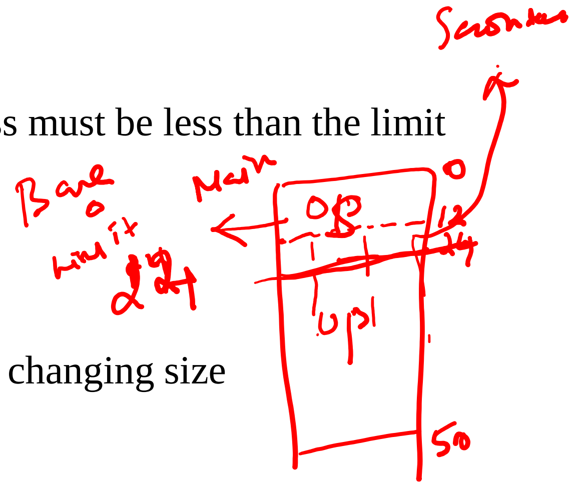
- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates process if low free memory, but first writes **application state** to flash for fast restart

CONTIGUOUS ALLOCATION

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

CONTIGUOUS ALLOCATION (CONT.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - The dispatcher loads the value of base and limit during context switch
  - Can then allow actions such as kernel code being **transient** and kernel changing size



Example:

- An operating systems contains code and buffer space for device drivers.
- If a device driver is not commonly used, we do not want to keep the code and data in the memory
- We may use this space for other purposes
- Such code is known as transient OS code – it comes and goes

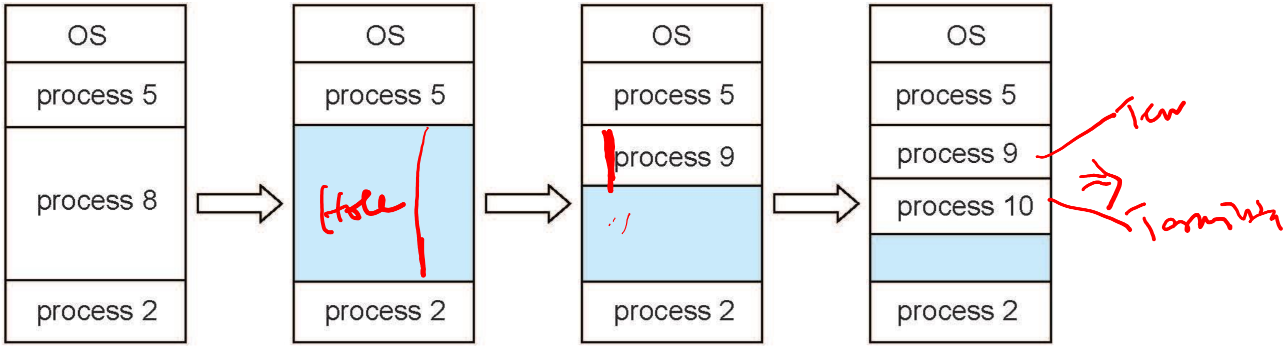
MULTIPLE-PARTITION ALLOCATION

Table

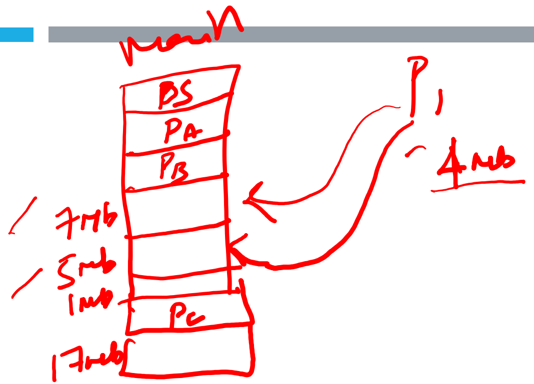
- Multiple-partition allocation
  - **Fixed-sized partitioned-** Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
    - **Hole** – block of available memory; holes of various size are scattered throughout memory
    - When a process arrives, it is allocated memory from a hole large enough to accommodate it
    - Process exiting frees its partition, adjacent free partitions combined
    - Operating system maintains information about:
      a) allocated partitions b) free partitions (hole)

1Mb
2Mb
7mb

$P_1 \Rightarrow$ 3Mb



DYNAMIC STORAGE-ALLOCATION PROBLEM



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

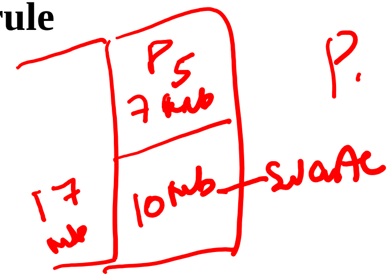
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list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

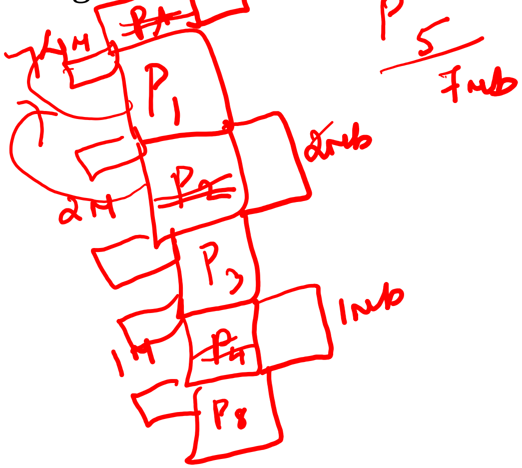
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

FRAGMENTATION

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**





FRAGMENTATION (CONT.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

SEGMENTATION

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

    local variables, global variables

    common block

    stack

    symbol table

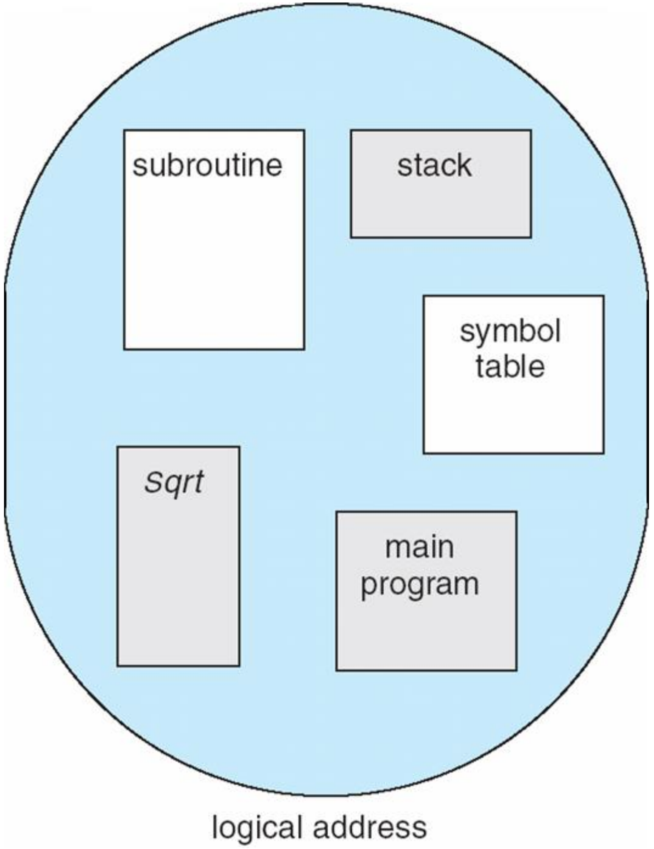
    arrays

USER'S VIEW OF A PROGRAM

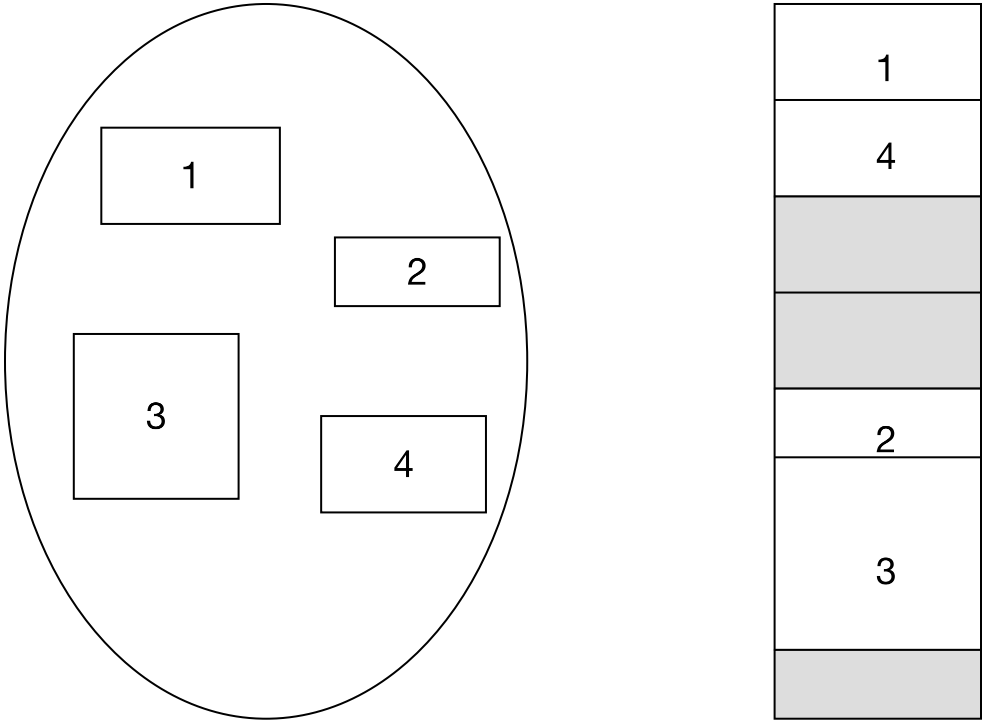
$\boxed{\text{main program}, 5}$

$\boxed{0, 5}$

0, 122



LOGICAL VIEW OF SEGMENTATION



user space

physical memory space

SEGMENTATION ARCHITECTURE

- Logical address consists of a two tuple:

  <segment-number, offset>,

$\boxed{1,\ 330}$

Segment table

Segment	Limit	Base
0	100	2000
1	400	3000
2	300	7000
3	400	1000

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment

2000 – 2100

- **Segment-table base register (STBR)** points to the segment table's location in memory

- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

SEGMENTATION ARCHITECTURE (CONT.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

THANK YOU!