OPERATING SYSTEMS

Memory Management

Likitha P

Department of Computer Science

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Prof. Venkatesh Prasad, Department of CSE, PES University.
2. Prof. Chandravva Hebbi, Department of CSE, PES University.
3. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018.
4. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018.
5. Some presentation transcripts from A. Frank – P. Weisberg.
6. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau.

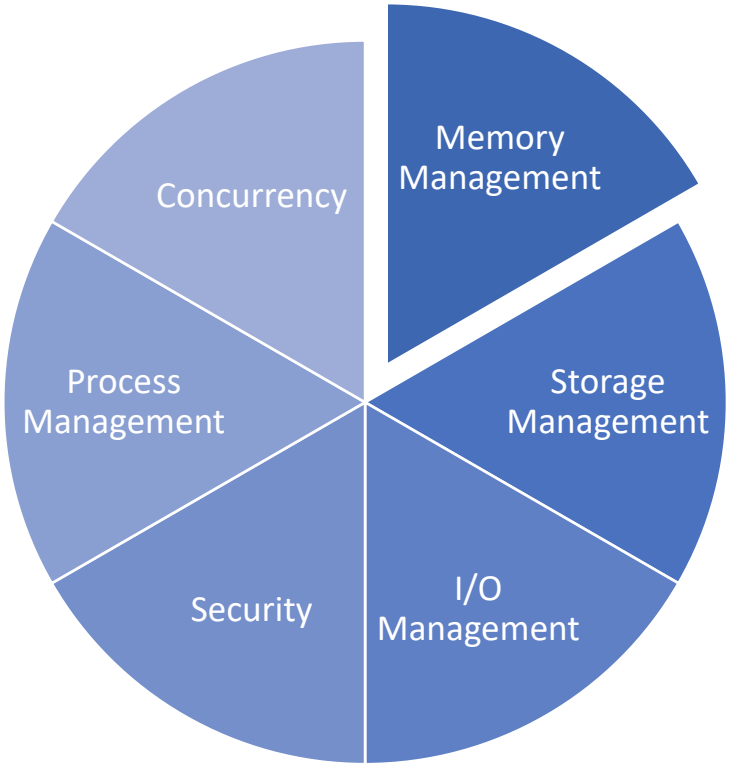
OPERATING SYSTEMS

Introduction

- CPU is shared by a number of processes and CPU scheduling allows improved utilization of the CPU and speed of computer.
- To realize this, we may need several processes in memory i.e. share memory.

Objectives

- Describe various techniques of organizing memory hardware.
- Explore memory allocation techniques - paging, segmentation.
- Paging in contemporary computer systems.

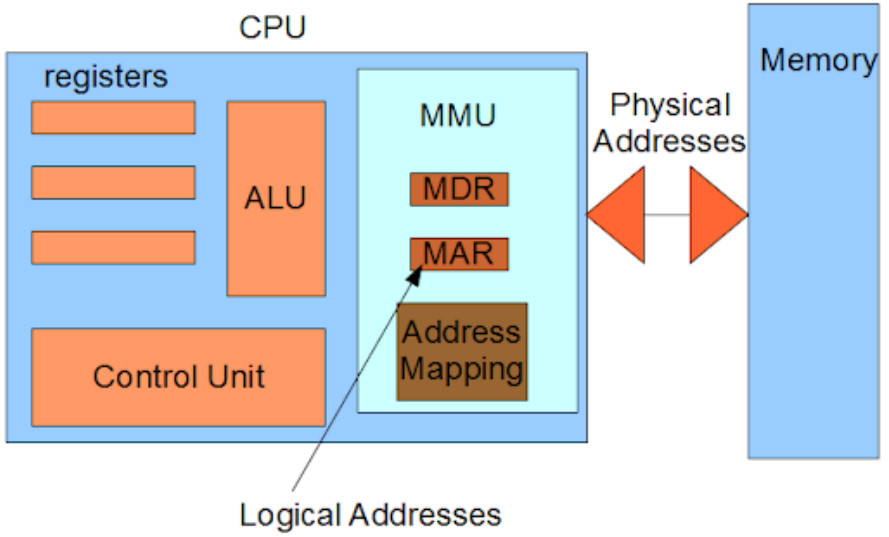


Components of an Operating System

Background

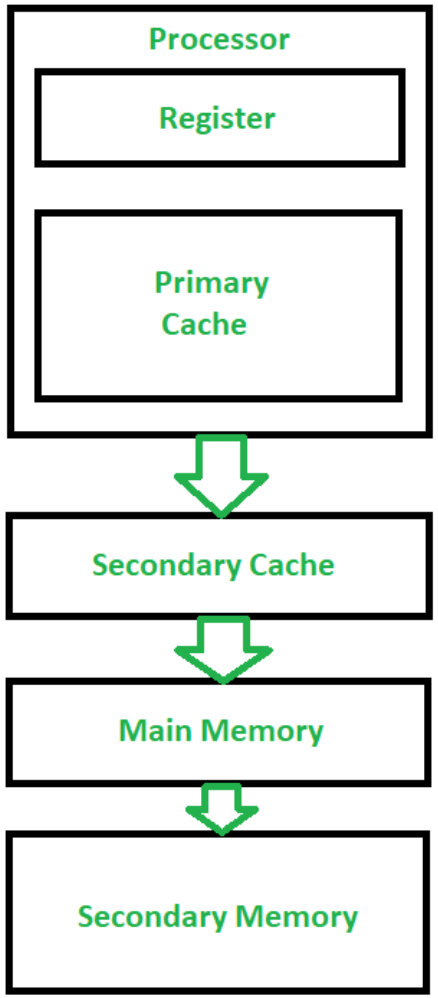
- To run a program, it must be brought from disk into memory and placed within a process.
- This is because the CPU can directly access only the Main memory and registers.
- Memory is a large array of bytes, each with its own address.
- Memory unit sees:
  - stream of addresses + read requests, or
  - address + data and write requests.

  Does not see the code/instructions.



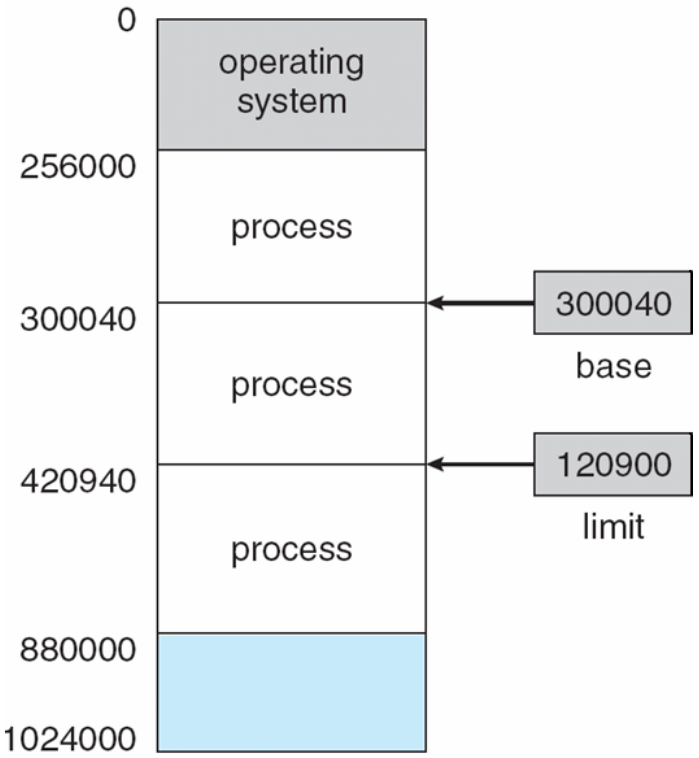
Background

- Registers are built into the CPU and are generally accessible in one CPU clock (or less).
- Main memory may require many cycles, causing the processor to **stall** as it is waiting for the necessary data from main memory.
- Thus, a fast memory called **Cache** sits between main memory and registers to match the speeds and reduce waiting.
- Not only speed, we must look at protection of memory to ensure correct and legal access operations.
- For this, provide each process a separate memory space. It is also fundamental to loading multiple processes into memory for concurrent execution.



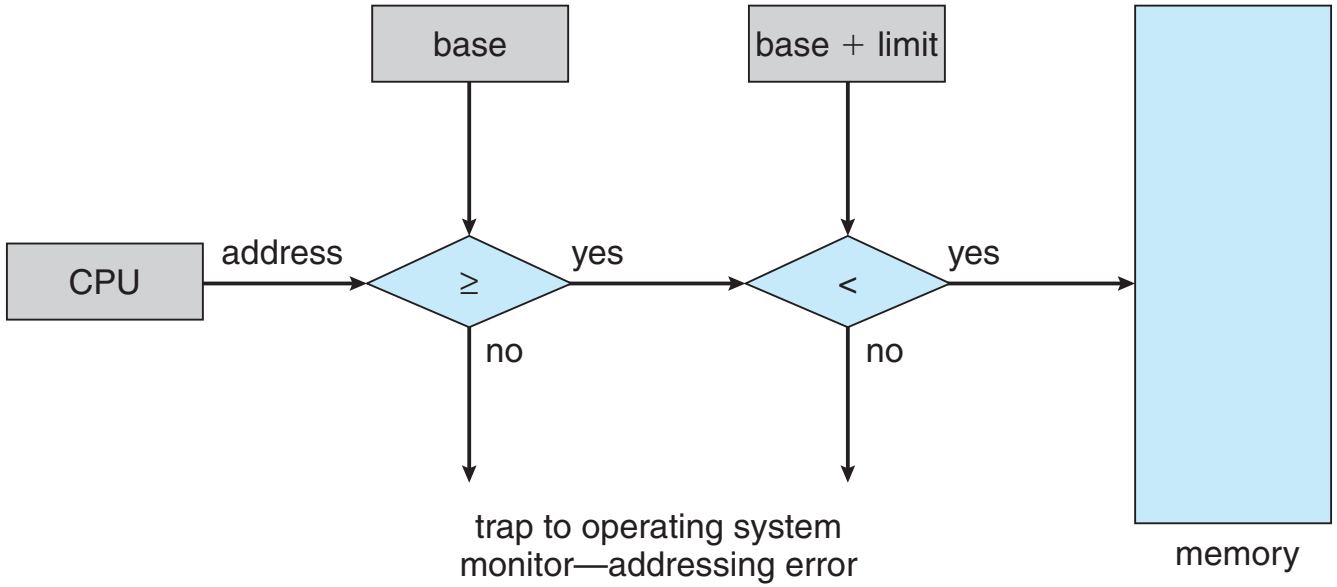
Base and Limit Registers

- To provide separate per-process memory spaces, we can use two registers. These two will define the range of legal addresses for a process.
- This is called the logical address space of the process.
- Base register – Holds the smallest legal physical memory address.
- Limit register – Defines the size of the address space and gives the range of the legal addresses.
- Base register with address 300040 and limit of 120900 denotes the space in 300040 – 420940.



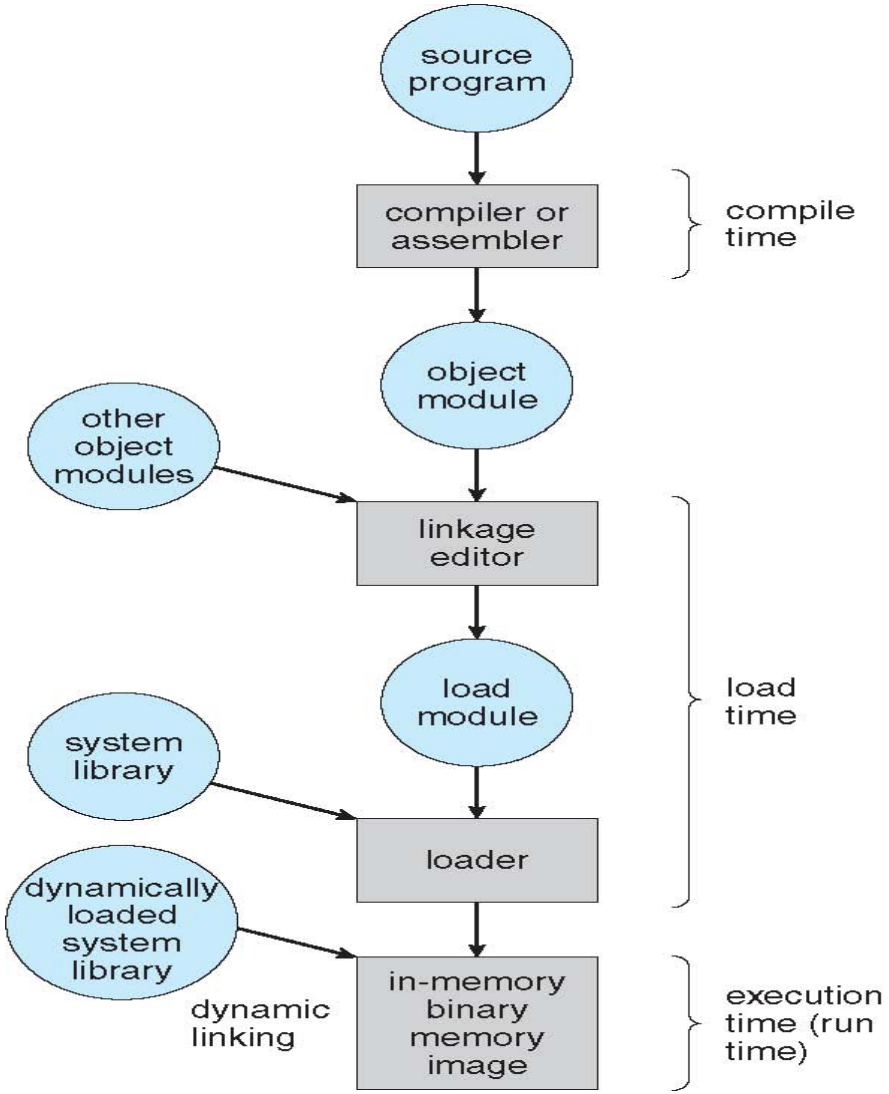
Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that use.
- The instructions to load the base and limit registers are privileged and hence executed in the kernel mode by the operating system. Also ensures user does not alter the register's contents.



OPERATING SYSTEMS

Multistep Processing of a User Program



Address Binding

- Addresses are represented in different ways at different stages of a program's life:
  - Source code addresses are usually symbolic e.g. int x – x is a symbolic address.
  - Compiled code addresses are **bound** to relocatable addresses e.g. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses e.g. 74014.
  - Each binding maps one address space to another.
- Without this, inconvenient to always have first user process physical address at 00000.

Binding of Instructions and Data to Memory

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

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 known as **virtual address.**
  - **Physical address** – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; they 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.

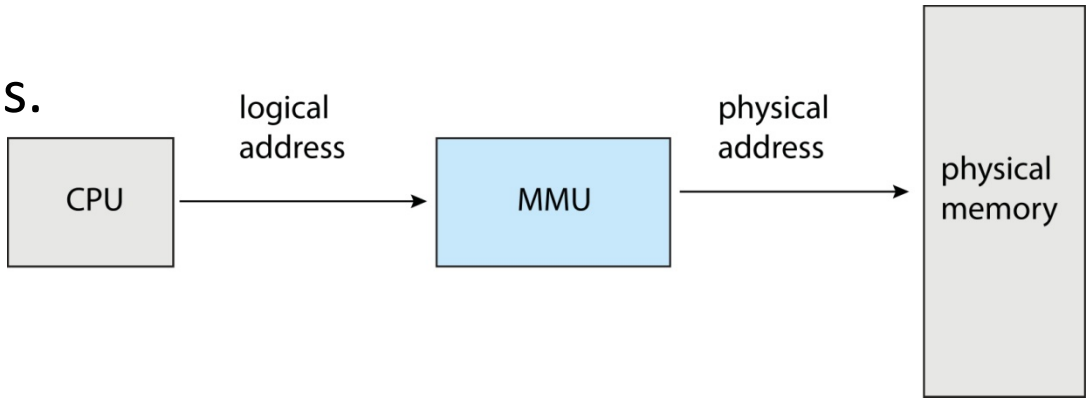
Memory-Management Unit (MMU)

Hardware device that at run time maps virtual to physical address

Many methods possible to accomplish this mapping, will be discussed in the next few lectures.

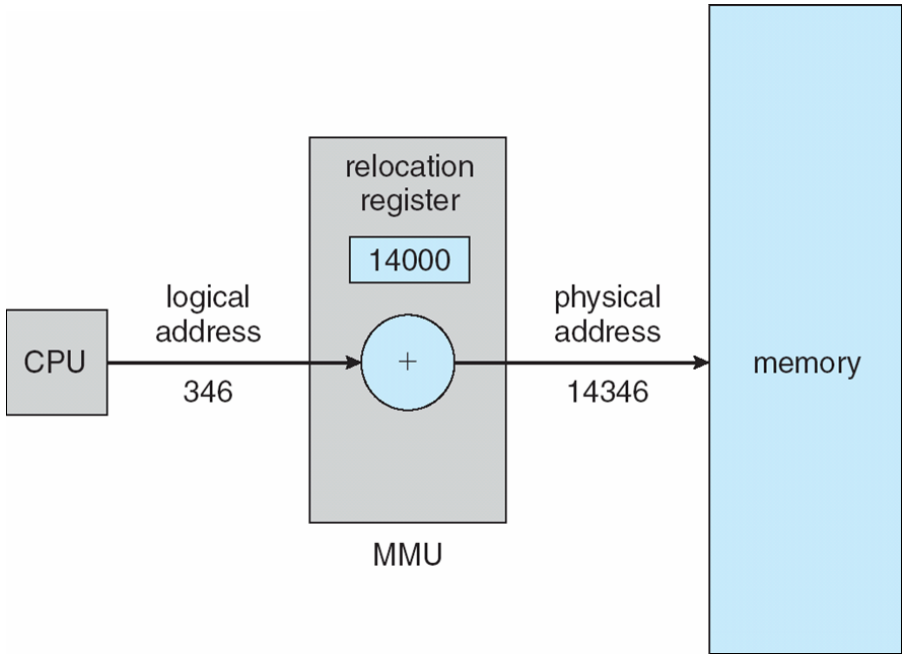
The user program deals with *logical* addresses; it never sees the *real* physical addresses.

- Execution-time binding occurs when reference is made to location in memory.
- Logical address is bound to physical address.



Memory-Management Unit (Cont.)

- To start, consider a 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 is now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers



Dynamic Loading

- The entire program needs to be in memory to execute.
- Routines are not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded.
- All routines are kept on disk in relocatable address 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.

Linking

- **Static linking** – The loader combines the system libraries and program code to form the binary program image.
- Dynamic linking –linking is postponed to execution time.
- Small piece of code, **stub**, is used to locate the appropriate memory-resident library routine.
- Address of the routine replaces stub, and then routine is executed.
- Operating system checks if routine is in process' 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

Static and Dynamic Linking

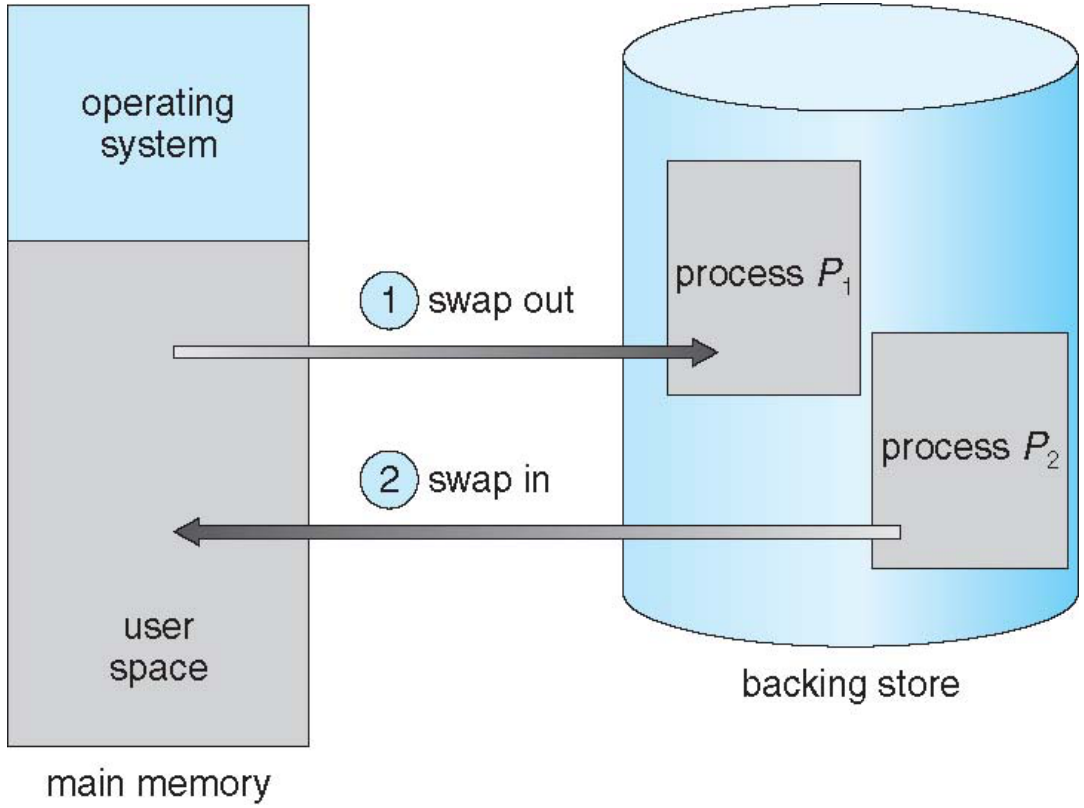
- A program whose necessary library functions are embedded directly in the program's executable binary file is ***statically*** linked to its libraries.
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

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.
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is rolled out so higher-priority process can be rolled in and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

OPERATING SYSTEMS

Schematic View of Swapping



Context Switch Time

- If next processes to be put in 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)
- How can we reduce the time required? Reduce size of memory swapped – know how much memory really being used.
  - System calls are used to inform OS of memory use via request_memory() and release_memory()

Swapping

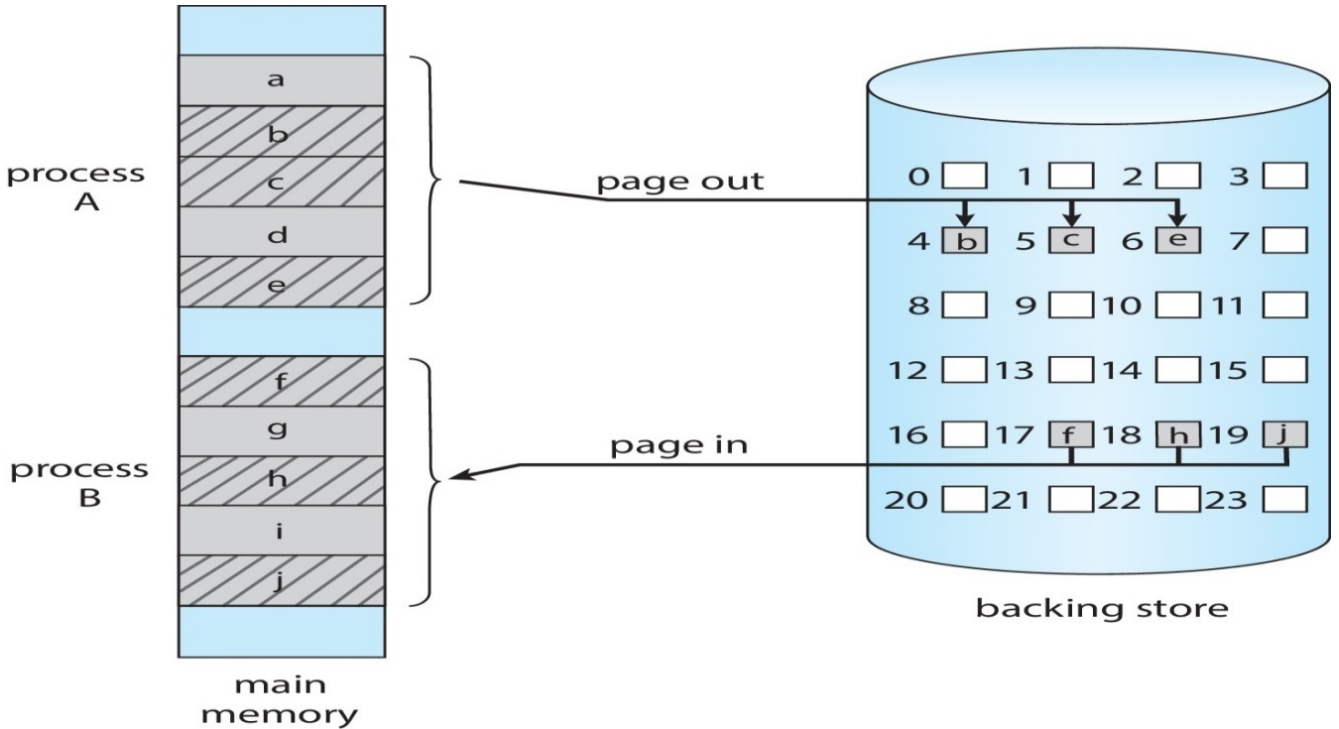
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to/from process memory space
- Other constraints on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device or process
    - Known as **double buffering**, adds overhead

Swapping on Computer Systems

- Standard swapping is not used in modern operating systems
  - But modified versions are common
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping is normally disabled but started if more than threshold amount of memory is allocated. Disabled again once memory demand reduces below threshold.
  - Swap only portions of process instead of entire process. Reduces the swap time as well.
  - Swapping with paging.

Swapping with Paging

- **Standard swapping** is generally no longer used .
- A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**.
- A subset of pages for processes *A* and *B* are being paged-out and paged-in respectively.
- **Swapping with paging** works well in conjunction with virtual memory.



OPERATING SYSTEMS
Swapping on Mobile Systems

- Not typically supported as system is Flash memory based.
  - Small amount of space,
  - Limited number of write cycles,
  - Poor throughput between flash memory and CPU on mobile platform.
- Instead, if memory is low use other methods to free it:
  - 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 apps if low free memory, but first writes **application state** to flash memory for fast restart

THANK YOU

Likitha P
Department of Computer Science Engineering

likithap@pes.edu