# OPERATING SYSTEMS

## Memory Management -2

**Nitin V Pujari**
**Faculty, Computer Science**
**Dean - IQAC, PES University**

# OPERATING SYSTEMS

## Course Syllabus - Unit 3

Unit-3:Unit 3: Memory Management: Main Memory

Hardware and control structures, OS support, Address translation, Swapping, Memory Allocation (Partitioning, relocation), Fragmentation, Segmentation, Paging, TLBs context switches

Virtual Memory - Demand Paging, Copy-on-Write, Page replacement policy - LRU (in comparison with FIFO & Optimal), Thrashing, design alternatives - inverted page tables, bigger pages.

Case Study: Linux/Windows Memory

## Course Outline

| | | | |
|---|---|---|---|
| 25 | Main Memory: Hardware and control structures, OS support, Address translation | 8.1 | 64.2 |
| 26 | Dynamic linking, Swapping | 8.2 | |
| 27 | Memory Allocation (Partitioning, relocation), Fragmentation | 8.3 | |
| 28 | Segmentation | 8.4 | |
| 29 | Paging: OS Support, TLBs, Address Translation | 8.5 | |
| 30 | Structure of the Page Table | 8.6 | |
| 31 | Design Alternatives - Inverted Page Tables, Bigger Pages | 8.7-8.8 | |
| 32 | Virtual Memory: Demand Paging, Copy-OnWrite | 9.1-9.3 | |
| 33 | Page replacement policy - LRU | 9.4 | |
| 34 | FIFO & Optimal | 9.5 | |
| 35 | Thrashing | 9.6 | |
| 36 | Case Study: Linux/ Windows Memory Management | 9.10 | |

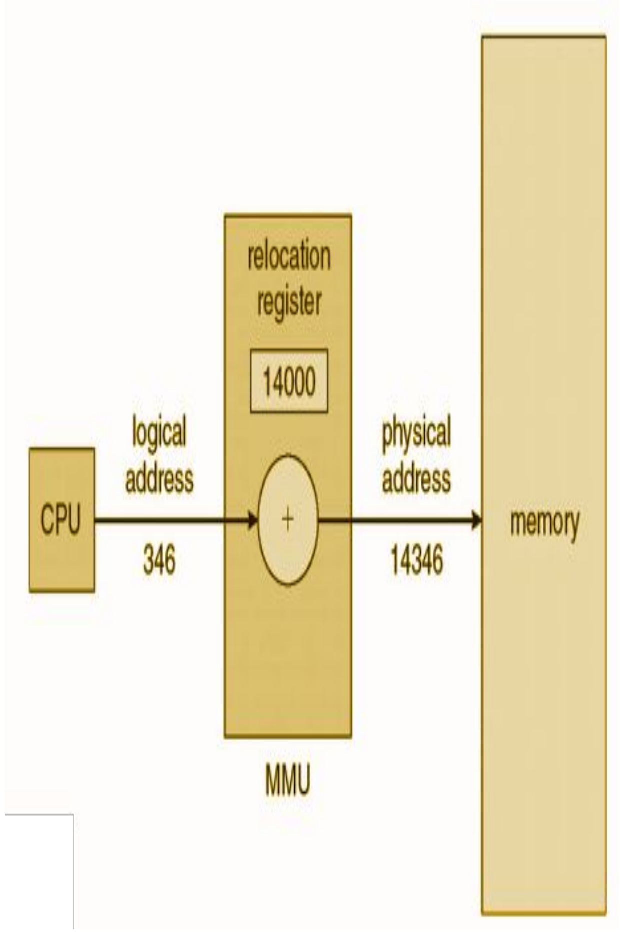
PES UNIVERSITY ONLINE

## Topic Outline

- Dynamic Allocation using Relocation Register
- Dynamic Linking
- Process Swapping
- Schematic view of Swapping
- Context Switching time including swapping
- Swapping on Mobile Systems

# Dynamic Allocation using 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



## Program Development in Unix - Additional Input

- When a process is running, what does its memory look like ? A collection of regions called sections. Basic memory layout for Linux and other Unix systems:

    - Code (or "text" in Unix terminology): starts at location 0
    - Data: starts immediately above code, grows upward
    - Stack: starts at highest address, grows downward

- System components that take part in managing a process's memory:
- **Compiler and assembler:**
    - Generate one object file for each source code file containing information for that source file.
    - Information is incomplete, since each source file generally references some things defined in other source files.

## Program Development in Unix - Additional Input

- **Linker:**
  - Combines all of the object files for one program into a single object file.
  - Linker output is complete and self-sufficient.

- **Operating system:**
  - Loads object files into memory.
  - Allows several different processes to share memory at once.
  - Provides facilities for processes to get more memory after they've started running.

- **Run-time library:**
  - Works together with OS to provide dynamic allocation routines, such as malloc and free in C.

# 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 process's memory address

## Dynamic Linking

- Since late 1980's most systems have supported shared libraries and dynamic linking
- For common library packages, only keep a single copy in memory, shared by all processes.
- Don't know where library is loaded until runtime; must resolve references dynamically, when program runs.

## Process 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 is fast disk large enough to accommodate copies of all memory images for all users; providing direct access to these memory images

## Process Swapping

- 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

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

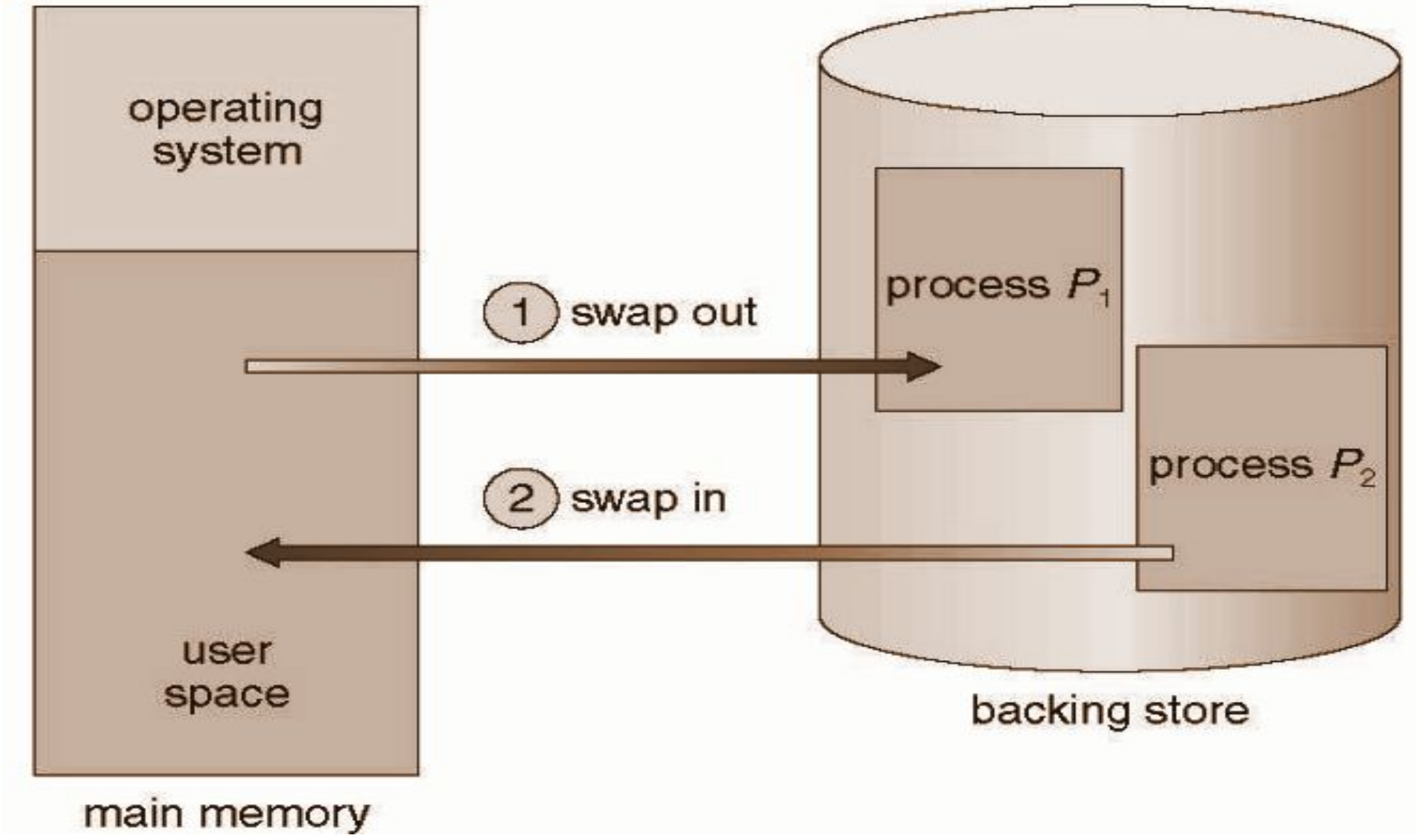
- System maintains a ready queue of ready-to-run processes which have memory images on disk

## Process 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

- 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 Process Swapping



## Context Switch Time including Process 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 => 50MB / 1000 ms => .05 ms per Mb

    - Swap out time of 2000 ms
    - Plus swap in case of same sized process
    - Total context switch swapping component time of 4000 ms (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 including Process Swapping

- 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

## Process Swapping on Mobile System

- 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

## Process Swapping on Mobile System

- 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 apps if low free memory, but first writes application state to flash for fast restart

# THANK YOU

**Nitin V Pujari**
**Faculty, Computer Science**
**Dean - IQAC, PES University**

**nitin.pujari@pes.edu**

**For Course Deliverables by the Anchor Faculty click on [www.pesuacademy.com](www.pesuacademy.com) and complete reading assignments provided by the Anchor on Edmodo**