CS343: Operating System

Memory Management

Lect26 : 06th Oct 2023

Dr. A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Motivations for Memory management
- Memory Management

Protection Issue

- Long term scheduler may put many process in to Ready at a time
  - Where to put ?
  - How to access them ?
  - Is there any efficient way to put and access?
  - How ensure safety and protection?

Memory Allocation: Top Down

Assume 4GB of RAM,
Avg Process Size 200MB
4 Segments/Process

OS
P5
P9
P8
P2

Finding Hole
First Fit
Worst fit
Best Fit

Very low complexity

Very High Mem Wastage

Process Continuous Allocation
10 process and 20 holes

Segment continuous Allocation
40 segment and 80 holes

low complexity

High Memory Wastage

Paging: 200MB process, 4KB page
Two Issue: pages and frames

Very Low Mem Wastage

High Complexity: 51200 pages/Process

Loading 51K pages in 10^6 frames

High Complexity: $4GB/4KB=2^{20}=10^6$ frames

4GB paged

SOL

Demand Paging Load the required 50-100 pages and load as an when necessary

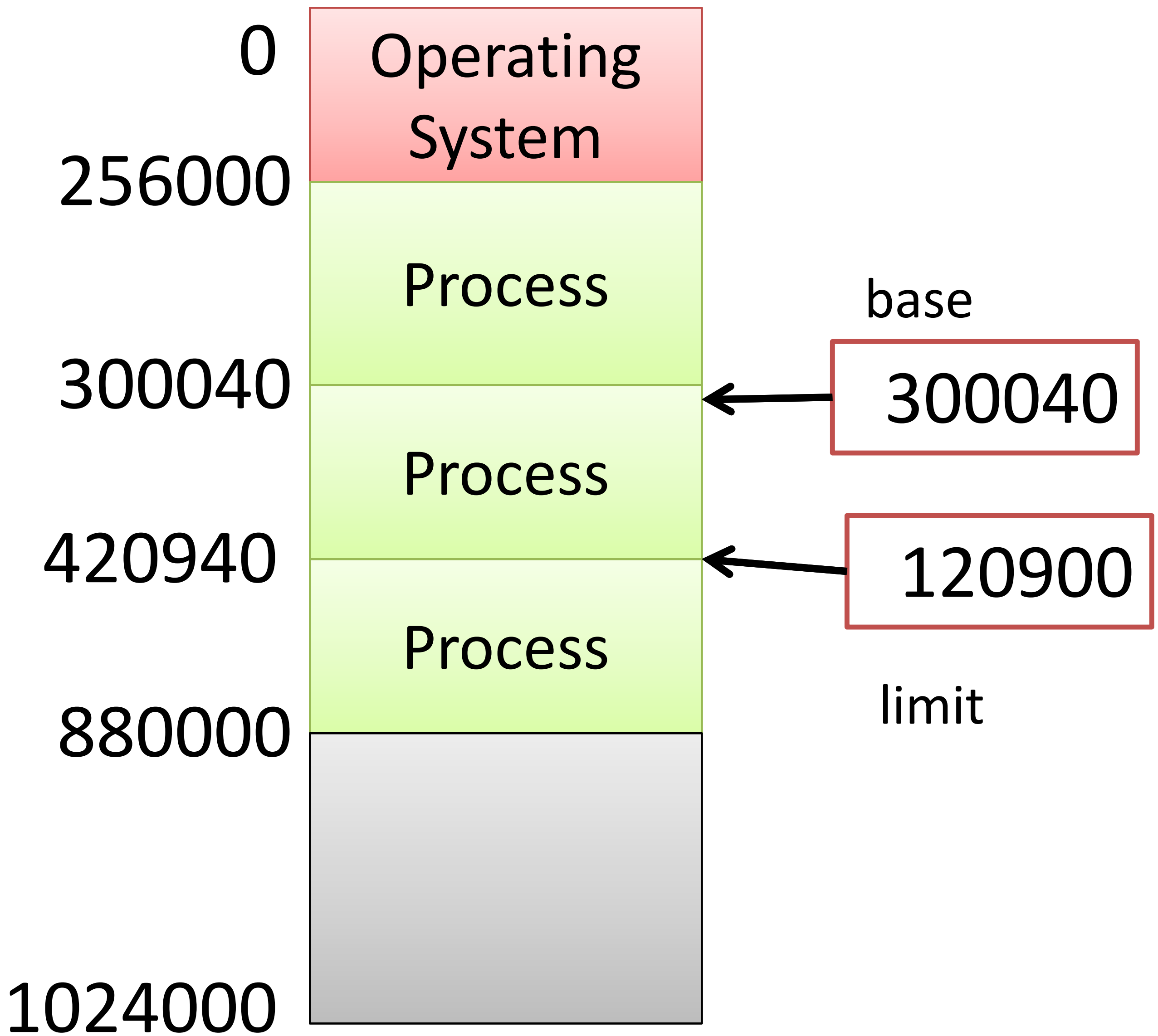
SOL

Allocate 100-200 frames per process: reduce search Space

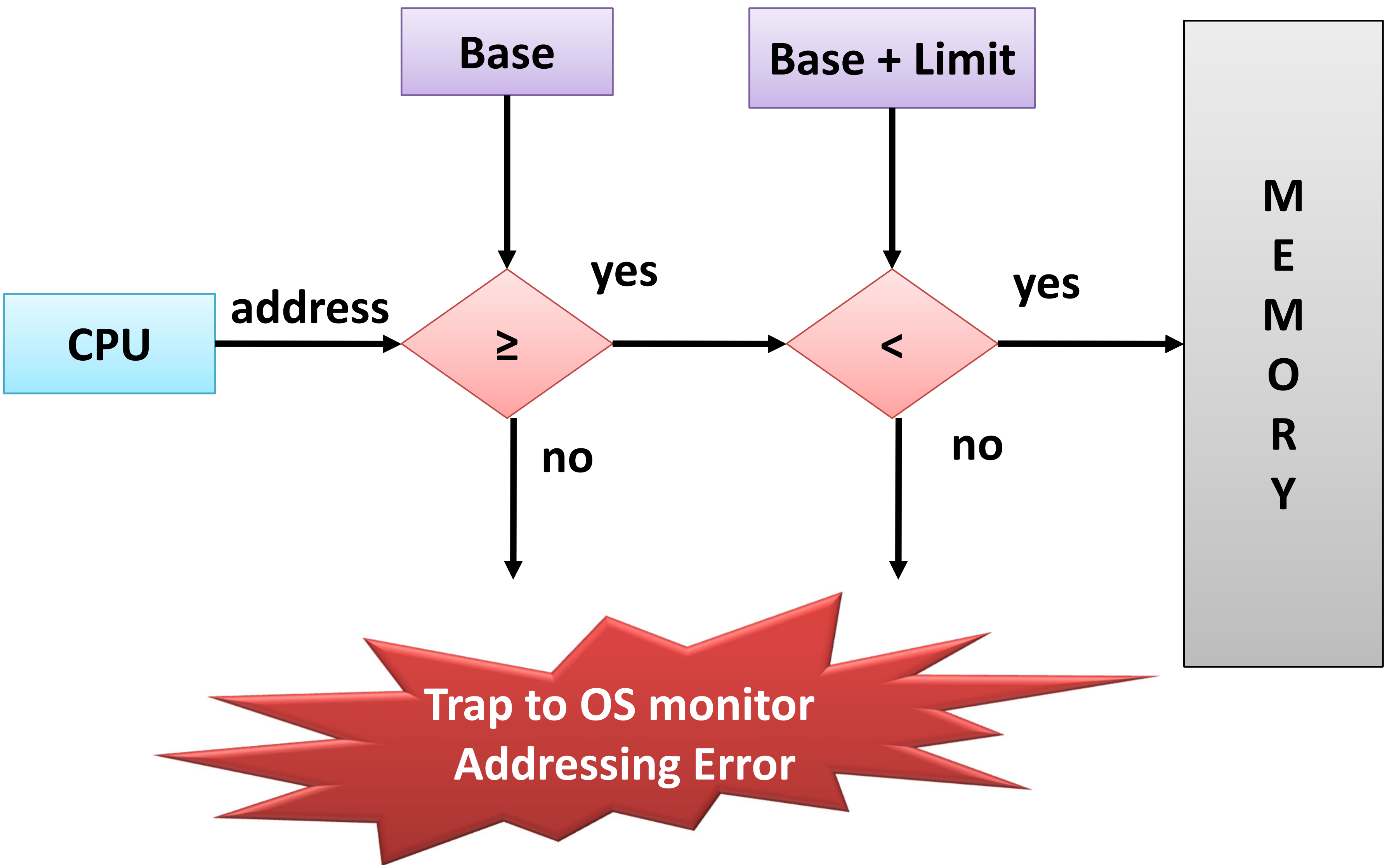
Base and Limit Registers

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

Base and Limit Registers



Hardware Address Protection



Memory for Process

- Single Partition
  - OS + One more process can run at a time
- Same amount to all processes
  - Allow only N processes to be in Memory
  - Size of process is limited by MemSize/N
- Variable amount of memory
  - Allow some process to be fit in the memory at a time
  - Paging +Segmentation: Improve share and reduce fragmentation
- Virtually : Allow a process to take infinite amount of memory

Logical vs. Physical Address Space

- 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

Logical vs. Physical Address Space

- **Logical (Virtual) and physical addresses**
  - Are same in compile-time and load-time address-binding schemes
  - Differ in execution-time address-binding scheme
- **Logical address space**
  - Set of all logical addresses generated by a program
- **Physical address space**
  - set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- MMU: Hardware device
  - At run time maps virtual Address to physical address
- A Simple Scheme
  - 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

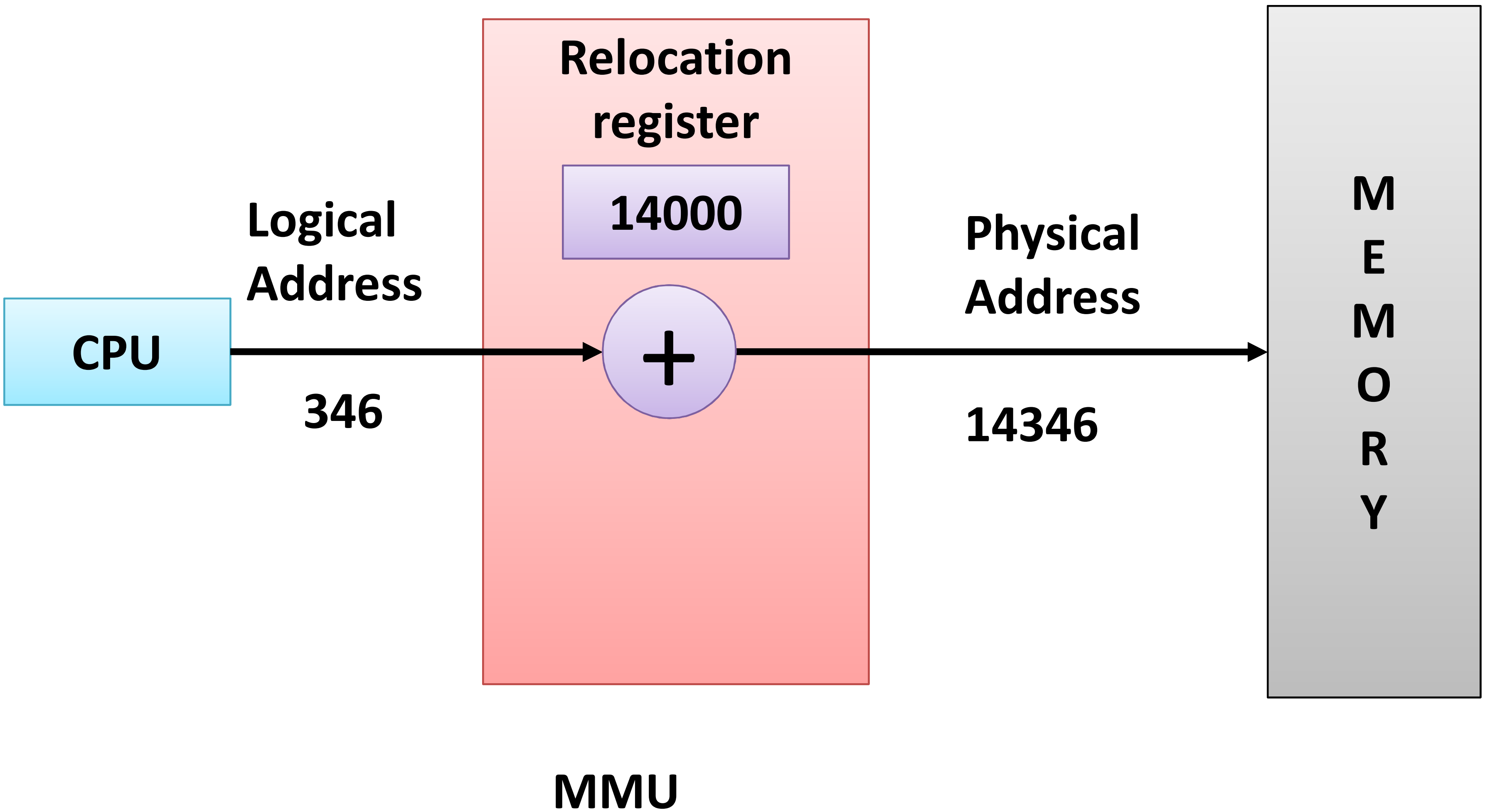
Memory-Management Unit (MMU)

- 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
    - Both for Instruction and Data
  - 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
  - Useful when large amounts of code are needed to handle infrequently occurring cases
- All routines kept on disk: Relocatable load format
- No special support from OS is required
  - Implemented through program design
  - OS provide libraries to implement dynamic loading

Dynamic relocation using a relocation register



Dynamic Linking

- Static linking : **lib****.a**
  - System libraries and program code combined by the loader into the binary program image
- Dynamic linking : **lib****.so**
  - Linking postponed until execution time
- Stub (a small piece of code)
  - Used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine

Dynamic Linking

- OS 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

Dynamic/Static Linking

$gcc test.c *//stdio.h → libc.so load dynamically*

$./a.out

$size a.out *//size is small as only stubs are loaded*

$gcc test.c -static /usr/lib/x86_64-linux-gnu/libc.a

$size a.out //libc.a embedded to a.out

$./a.out

//check out contents of libc.a

$nm /usr/lib/x86_64-linux-gnu/libc.a | grep printf

$objdump -a /usr/lib/x86_64-linux-gnu/libc.a | grep printf

Swapping

- 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 (Ex : HDD/SDD)**
  - fast disk large enough to accommodate copies of all memory images for all users;
  - must provide direct access to these memory images

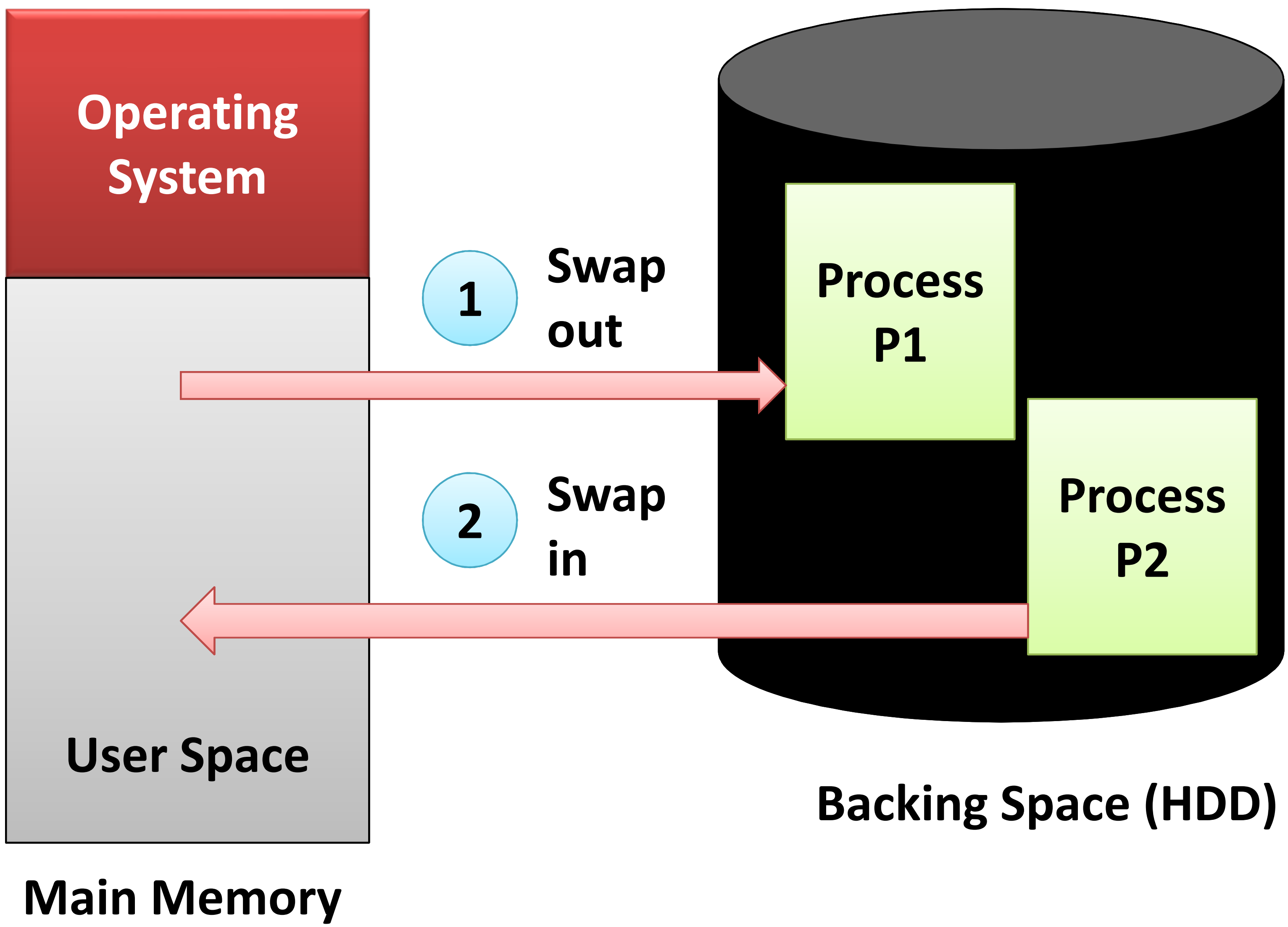
Swapping

- **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 **queue**
  - 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
  - Plus consider pending I/O to / from process memory space

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 2000 ms + Swap in of same sized process = 4 seconds
  - Swapping component of Context Switch time is 4 seconds

Context Switch Time including Swapping

- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - Read only memory Discard
    - Code and Declared Read only Data, Constant Area
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Real System :Memory SwapIn/SwapOut

- Server : Utilization is high, always get a crunch…..Supposed to utilized all resources
- PC (4GB RAM, 500GB HDD)
  - Most of the time RAM utilization is bellow 40%
  - Most of the time we get a space for our process to Run….. :) :)
- Mobile (256MB RAM, 8GB SSD)
  - SSD is faster, Read takes less time but write take time, Number write to SSD is limited by a number in SSD life time
  - SSD can n't be used as RAM & System have low RAM

Real System :Memory SwapIn/SwapOut

- In PC
- 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 on Mobile Systems

- Not typically supported : Flash memory based
  - Small amount of space
  - Limited number of write cycles (NVRAM, STT RAM, PCM RAM)
  - Poor throughput between flash memory and CPU on mobile platform

Swapping on Mobile Systems

- 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
  - Both OSes support paging (will be discussed)

Thanks