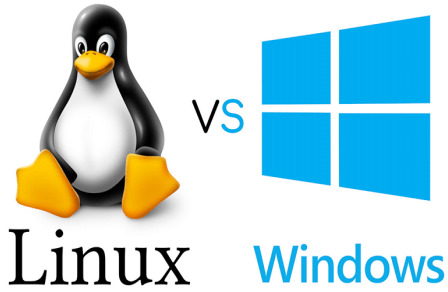
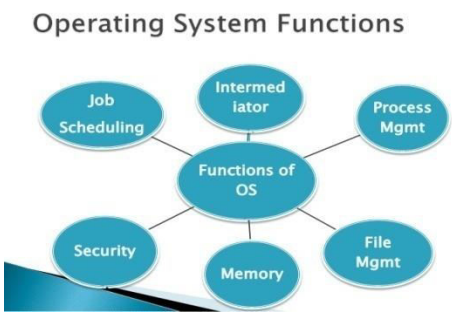
CIT210 OP. SYS. MANAG.

CSUN
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

Week5Day2 : Memory Management, Memory Hierarchy, Physical Organization, Logical Organization, Logical vs Physical address, Address Binding, Swapping and Scheduling





Objectives (Memory Management)

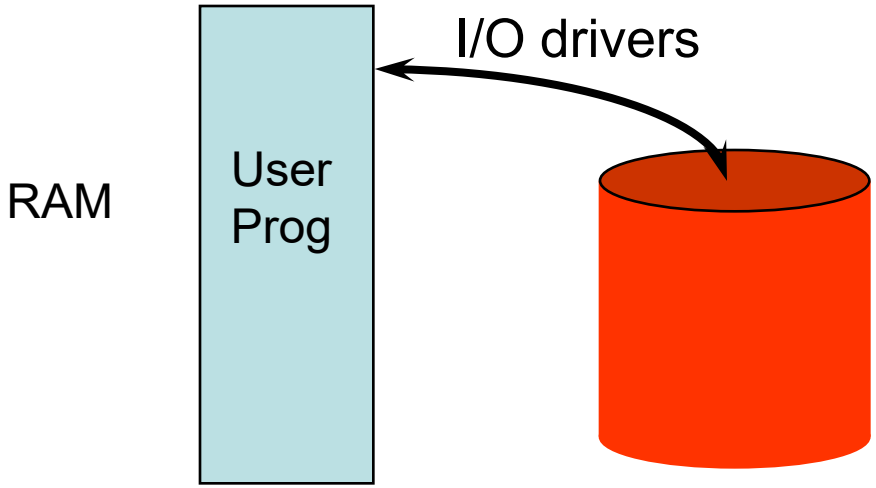
- Background
- Types of Memory
- Logical vs Physical Address
- Memory Management Requirements
- Swapping
- Contiguous Memory Allocation

In the Beginning (prehistory)…

- **Batch systems**
  - **One program loaded in physical memory at a time**
  - **Runs to completion**
- **If job larger than physical memory, use *overlays***
  - **Identify sections of program that**
    - Can run to a result
    - Can fit into the available memory
  - **Add statement after result to load a new section**
  - **Example: passes of a compiler**
  - **Example: SAGE – *North American Air Defense System***

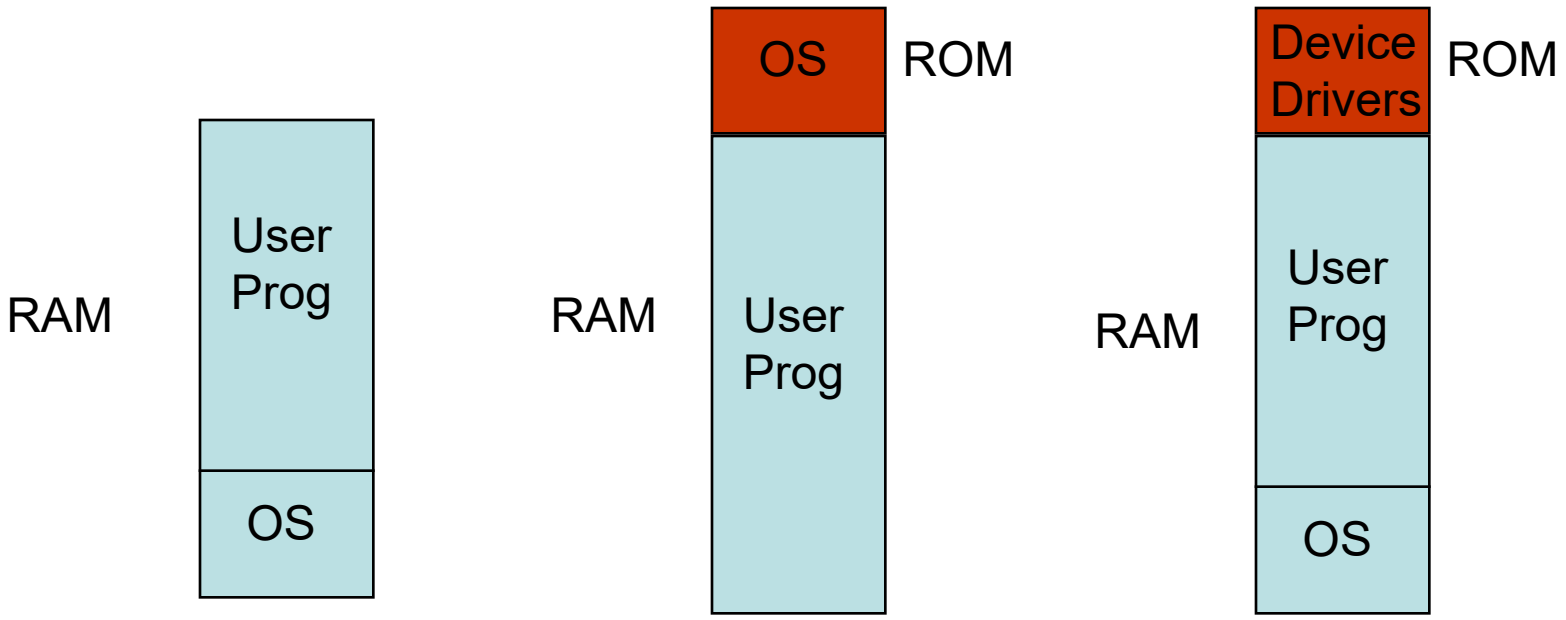
Simple Memory Management

- One process in memory, using it all
  - each program needs I/O drivers
  - until 1960



Simple Memory Management

- **Small, protected OS, drivers**
  - **DOS**



- "Mono-programming" -- No multiprocessing!

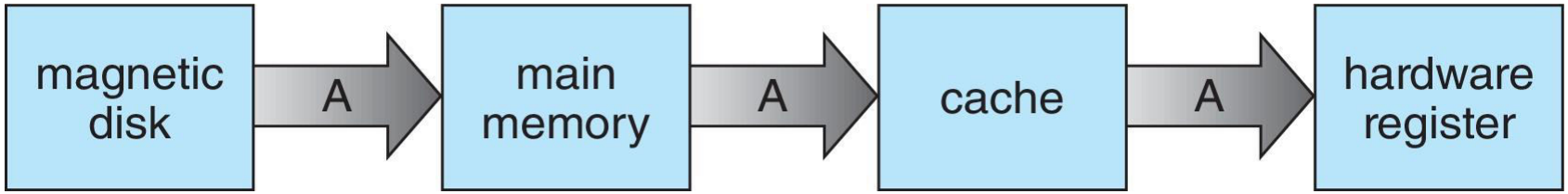
  - Early efforts used "Swapping", but slooooow

Multiprogramming and Multitasking

- Multiprogramming (Batch system) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via job scheduling
  - When it has to wait (for I/O for example), OS switches to another job
- Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing
  - Response time should be < 1 second
  - Each user has at least one program executing in memory ⇨process
  - If several jobs ready to run at the same time ⇨ CPU scheduling
  - If processes don't fit in memory, swapping moves them in and out to run
  - Virtual memory allows execution of processes not completely in memory

Migration of data “A” from Disk to Register

- **Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy**



- **Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache**

Memory Management Concept

- **Memory management**
  - Is the task carried out by the OS and hardware to accommodate multiple processes in main memory
  - If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle
  - Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible
  - In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes

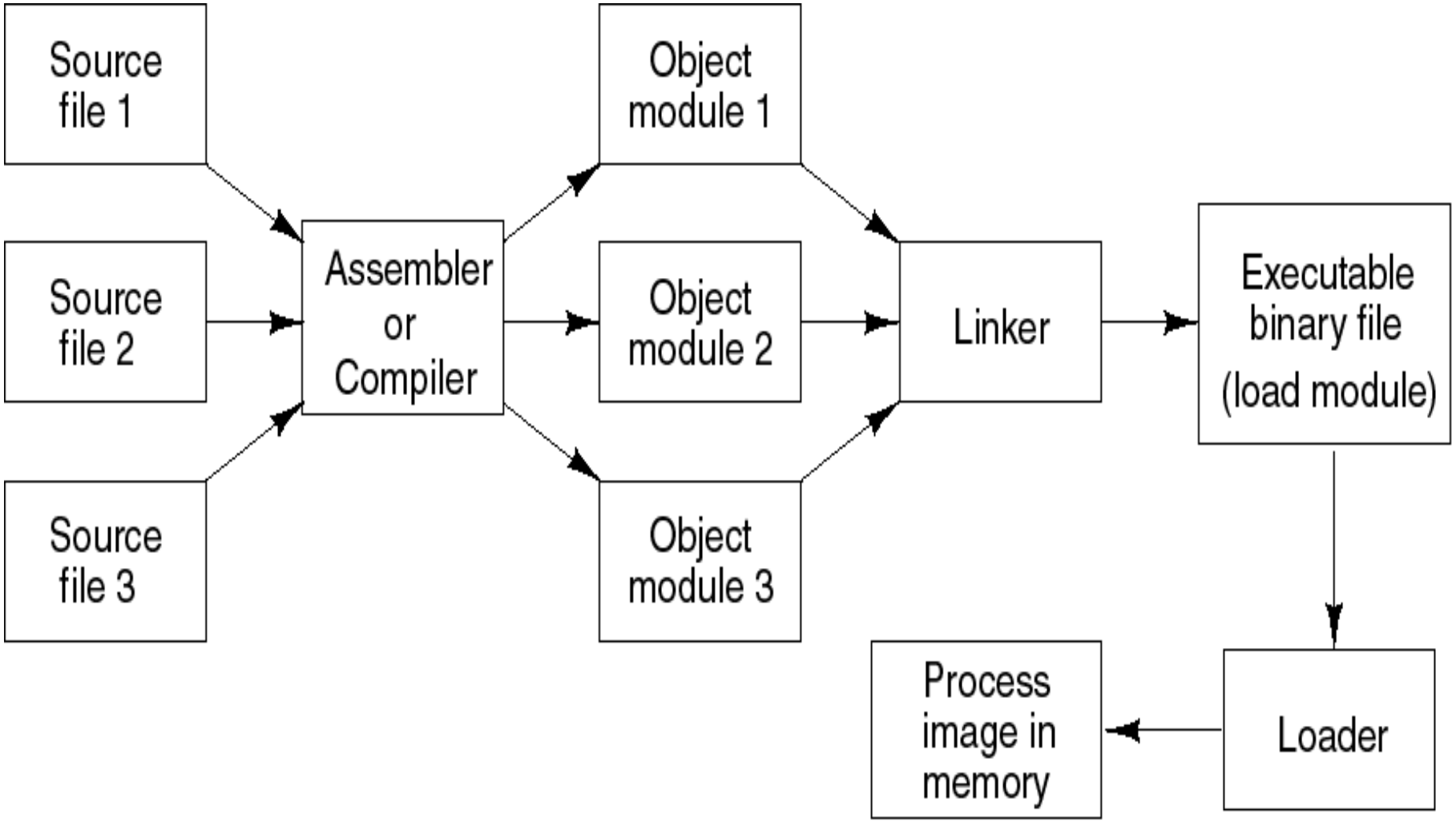
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Memory Management(multi-tasking) …

- Multiple processes in physical memory at the same time
  - allows fast switching to a ready process
  - Divide physical memory into multiple pieces – *partitioning*
  - Modern *real-time* operating systems
- Partition requirements
  - *Protection* – keep processes from smashing each other
  - *Fast execution* – memory accesses can't be slowed by protection mechanisms
  - *Fast context switch* – can't take forever to setup mapping of addresses

Multi-step processing of user program



Address Binding

- A process must be tied to a physical address at some point (bound)
- Binding can take place at 3 times
  - Compile time
    - Always loaded to same memory address
  - Load time
    - relocatable code
    - stays in same spot once loaded
  - Execution time
    - may be moved during execution
    - special hardware needed

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.

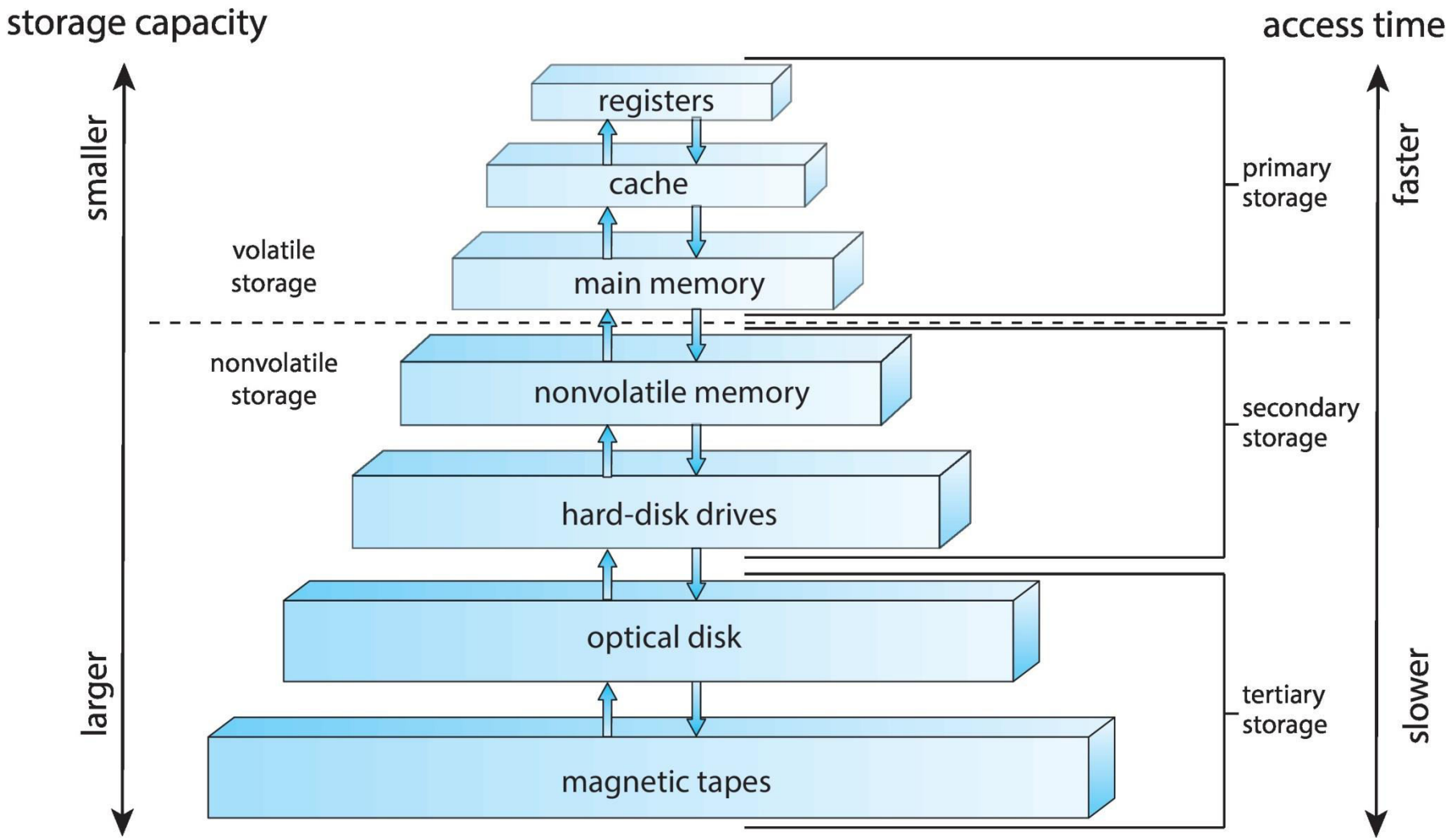
Background

- Memory management is the task carried out by the OS and hardware to accommodate multiple processes in main memory.
- User programs go through several steps before being able to run.
- This multi-step processing of the program invokes the appropriate utility and generates the required module at each step.

Memory Hierarchy

- **What is the memory hierarchy?**
  - **Different levels of memory**
  - **Some are small & fast**
  - **Others are large & slow**
- **What levels are usually included?**
  - **Cache: small amount of fast, expensive memory**
    - L1 (level 1) cache: usually on the CPU chip
    - L2 & L3 cache: off-chip, made of SRAM
  - **Main memory: medium-speed, medium price memory (DRAM)**
  - **Disk: many gigabytes of slow, cheap, non-volatile storage**
- **Memory manager handles the memory hierarchy**

Storage-Device Hierarchy



The need for memory management

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU's efficiency

Memory Management

Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time

Goals of memory management

- Allocate scarce memory resources among competing processes, maximizing memory utilization and system throughput
- Provide a convenient abstraction for programming (and for compilers, etc.)
  - Hide sharing
  - Hide scarcity
- Provide isolation between processes
  - we have come to view "addressability" and "protection" as inextricably linked, even though they're really orthogonal

Definition

- Memory management is the process of
  - allocating primary memory to user programs
  - reclaiming that memory when it is no longer needed
  - protecting each user's memory area from other user programs; i.e., ensuring that each program only references memory locations that have been allocated to it.

Types of Memory

- Real memory
  - Main memory
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

Today's desktop and server systems

- **The basic abstraction that the OS provides for memory management is virtual memory (VM)**
  - **VM enables programs to execute without requiring their entire address space to be resident in physical memory**
    - program can also execute on machines with less RAM than it "needs"
  - **many programs don't need all of their code or data at once (or ever)**
    - e.g., branches they never take, or data they never read/write
    - no need to allocate memory for it, OS should adjust amount allocated based on run-time behavior
  - **virtual memory isolates processes from each other**
    - one process cannot name addresses visible to others; each process has its own isolated address space

Memory Management Terms

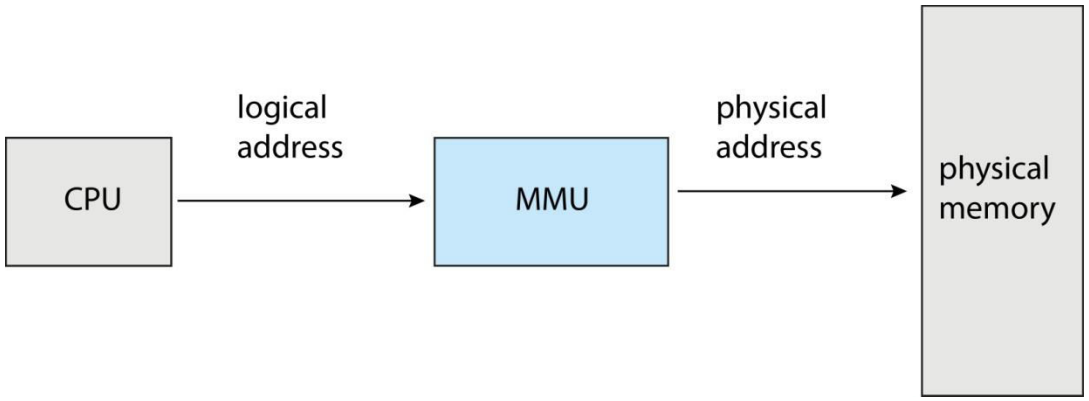
Term	Description
Frame	***Fixed***-length block of main memory.
Page	***Fixed***-length block of data in secondary memory (e.g. on disk).
Segment	***Variable-length*** block of data that resides in secondary memory.

Memory Management Requirements

- **Physical Organization**
  - Main memory verses secondary memory
  - Overlaying
- **Logical Organization**
  - Support modules, shared subroutines
- **Relocation**
  - Users generally don't know where they will be placed in main memory
  - May swap in at a different place (pointers???)
  - Generally handled by hardware
- **Protection**
  - Prevent processes from interfering with the O.S. or other processes
  - Often integrated with relocation
- **Sharing**
  - Allow processes to share data/programs

Memory-Management Unit (MMU)

- **Hardware device that at run time maps virtual to physical address**



Physical Organization

- **Two-level memory for program storage:**
  - **Disk (slow and cheap) & RAM (fast and more expensive)**
  - **Main memory is volatile, disk isn't**
- **User should not have to be responsible for organizing movement of code/data between the two levels.**
  - **main memory for program and data currently in use, secondary memory is the long term store for swapping and paging**
  - **moving information between levels of memory is a major concern of memory and file management (OS)**
    - should be transparent to the application programmer

Physical Organization

Cannot leave the programmer with the responsibility to manage memory

Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

overlaying allows various modules to be assigned the same region of memory but is time consuming to program

Logical Organization

Programs are written in modules

- code is usually execute-only (reentrant)
- data modules can be read-only or read/write
- References between modules must be resolved at run time
- Segmentation is one useful approach for all this
  - but not the only approach

Logical Organization

- **Main memory is organized as a linear address space consisting of a sequence of bytes or words.**
- **Programs aren't necessarily organized this way**

Programs are written in modules

- modules can be written and compiled independently
- different degrees of protection given to modules (read-only, execute-only)
- sharing on a module level corresponds to the user's way of viewing the problem

Relocation

- Why is relocation needed?
  - Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
  - Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
- Consequently it must be possible to adjust addresses whenever a program is loaded.
  - A program may be *relocated* (often) in main memory due to swapping
  - Memory references in code (for both instructions and data) must be translated to actual physical memory addresses

Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection
  - Cannot do this at compile time, because we do not know where the program will be loaded in memory
  - address references must be checked at run time by hardware

Sharing

- **Advantageous to allow each process access to the same copy of the program rather than have their own separate copy**
- **Memory management must allow controlled access to shared areas of memory without compromising protection**
- **Mechanisms used to support relocation support sharing capabilities**
  - Cooperating processes may need to share access to the same data structure
  - Program text can be shared by several processes executing the same program for a different user
    - Saves memory over separate copy for each
    - also applies to dynamic (sharable) libraries

Addresses

Logical

- reference to a memory location independent of the current assignment of data to memory
- a translation must be made to a physical address before the memory access can be achieved

Relative

- A **relative address** is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register

Physical or Absolute

- actual location in main memory

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
- 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

Virtual addresses for multiprogramming

- **To make it easier to manage memory of multiple processes, make processes use virtual addresses**
  - **virtual addresses are independent of location in physical memory (RAM) where referenced data lives**
    - OS determines location in physical memory
  - **instructions issued by CPU reference virtual addresses**
    - e.g., pointers, arguments to load/store instructions, PC …
  - **virtual addresses are translated by hardware into physical addresses (with some setup from OS)**

Virtual addresses for multiprogramming

- The set of virtual addresses a process can reference is its address space
  - many different possible mechanisms for translating virtual addresses to physical addresses
- Note: We are not talking about paging, or virtual memory – only that the program issues addresses in a virtual address space, and these must be "adjusted" to reference memory (the physical address space)
  - for now, think of the program as having a contiguous virtual address space that starts at 0, and a contiguous physical address space that starts somewhere else

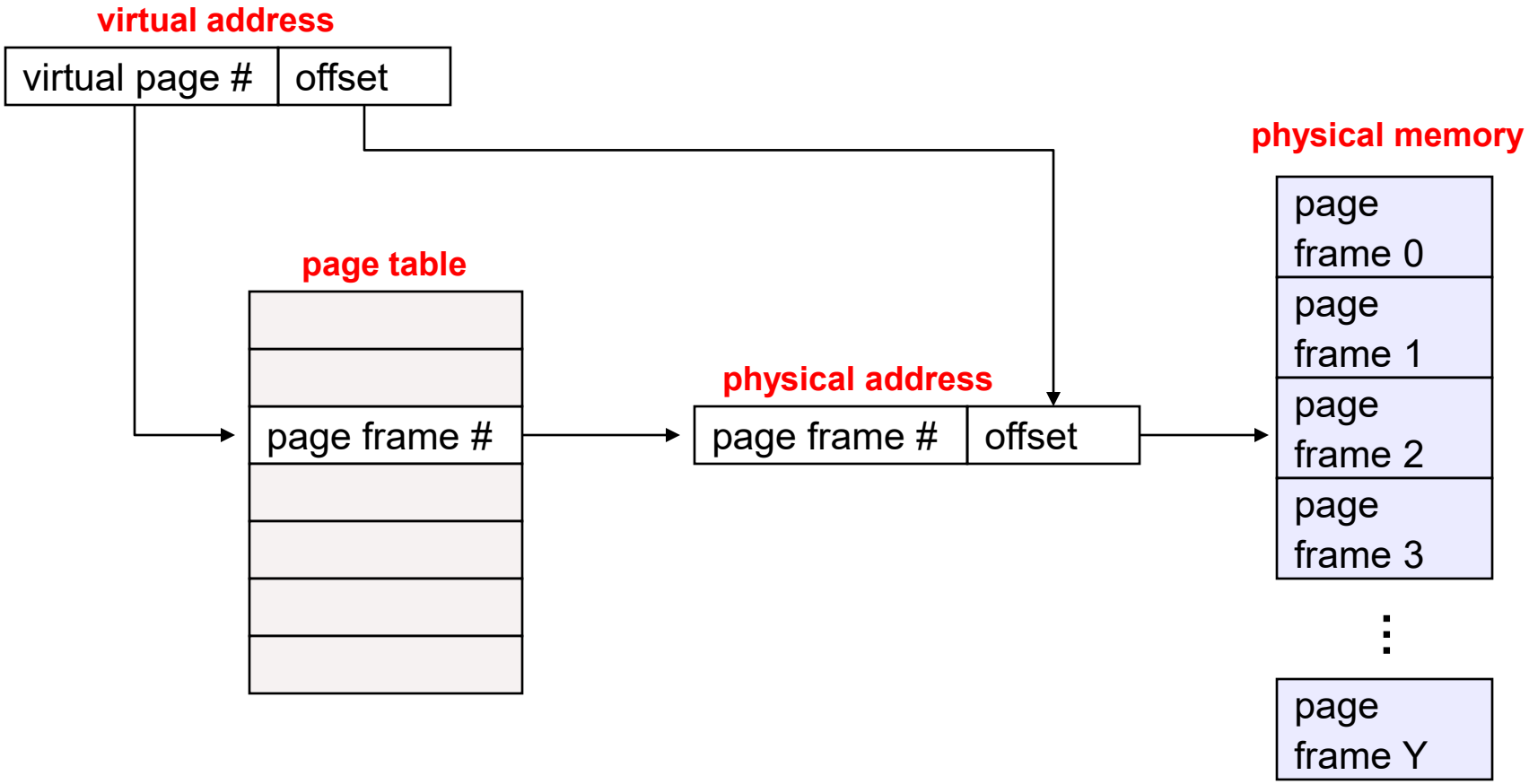
User's perspective

- Processes view memory as a contiguous address space from bytes 0 through N
  - virtual address space (VAS)
- In reality, virtual pages are scattered across physical memory frames – not contiguous as earlier
  - virtual-to-physical mapping
  - this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - the virtual address 0xDEADBEEF maps to different physical addresses for different processes

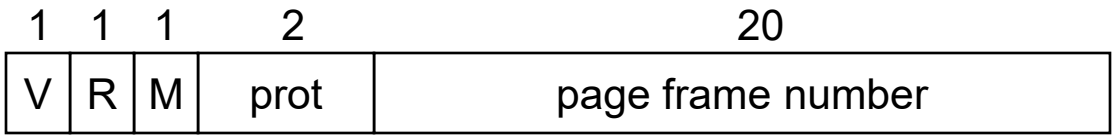
Address translation

- **Translating virtual addresses**
  - **a virtual address has two parts: virtual page number & offset**
  - **virtual page number (VPN) is index into a page table**
  - **page table entry contains page frame number (PFN)**
  - **physical address is PFN::offset**
- **Page tables**
  - **managed by the OS**
  - **map virtual page number (VPN) to page frame number (PFN)**
    - VPN is simply an index into the page table
  - **one page table entry (PTE) per page in virtual address space**
    - i.e., one PTE per VPN

Mechanics of address translation



Page Table Entries (PTEs)



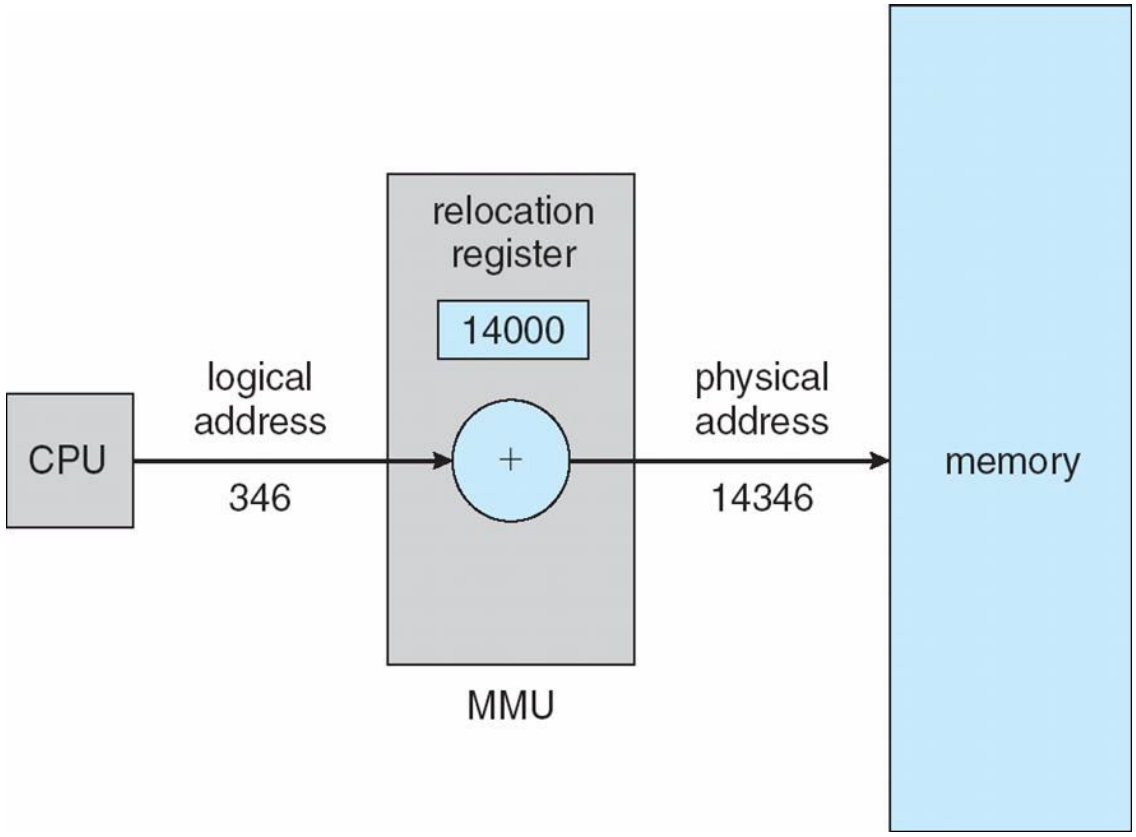
- **PTE's control address mappings**
  - **The page frame number indicates the physical page**
  - **The valid bit says whether or not the PTE can be used**
    - says whether or not a virtual address is valid
  - **The referenced bit says whether the page has been accessed recently**
  - **The modified bit says whether or not the page is dirty**
    - it is set when a write to the page has occurred
  - **The protection bits control which operations are allowed**
    - read, write, execute

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible
- 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

Logical vs. Physical Address Space

- **Physical address: The actual hardware memory address.**
  - **32-bit CPU's physical address 0 ~ 2^{32}-1 (00000000 – FFFFFFFF)**
  - **1GB's memory address 0 ~ 2^{30}-1 (00000000 – 4FFFFFFF)**
- **Logical address: Each (relocatable) program assumes the starting location is always 0 and the memory space is much larger than actual memory**



Dynamics of hardware translation of addresses

- When a process is assigned to the running state, a relocation/base register gets loaded with the starting physical address of the process.
- A limit/bounds register gets loaded with the process's ending physical address.
- When a relative addresses is encountered, it is added with the content of the base register to obtain the physical address which is compared with the content of the limit/bounds register.
- This provides hardware protection: each process can only access memory within its process image.

Linking and Loading

- Linking. *Bind* together the different parts of a program in order to make them into a runnable entity - Linkage editor
  - static (before execution)
  - dynamic (on demand during execution)
- Loading. Program is loaded into main memory, ready to execute. May involve address translation.
  - Absolute, relocatable
  - static, dynamic

Static Linking

- **Static linking –**
  - **System libraries and program code combined by the loader into the binary program image**
  - **The linker uses tables in object modules to link modules into a single linear addressable space.**
  - **The new addresses are addresses relative to the beginning of the load module.**

Dynamic Linking

- **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

Program vs. Memory sizes

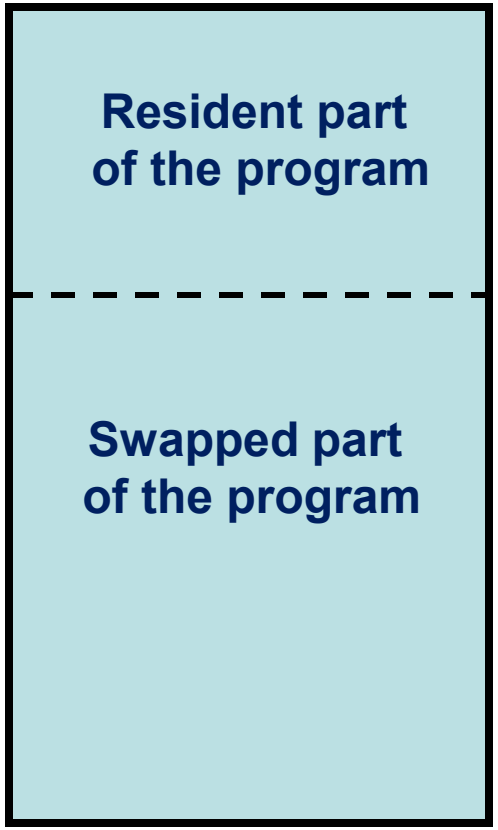
- What to do when program size is larger than the amount of memory/partition (that exists or can be) allocated to it?
- There are two basic solutions within real memory management:
  1. Overlays
  2. Dynamic Linking (Libraries – DLLs)

Overlays

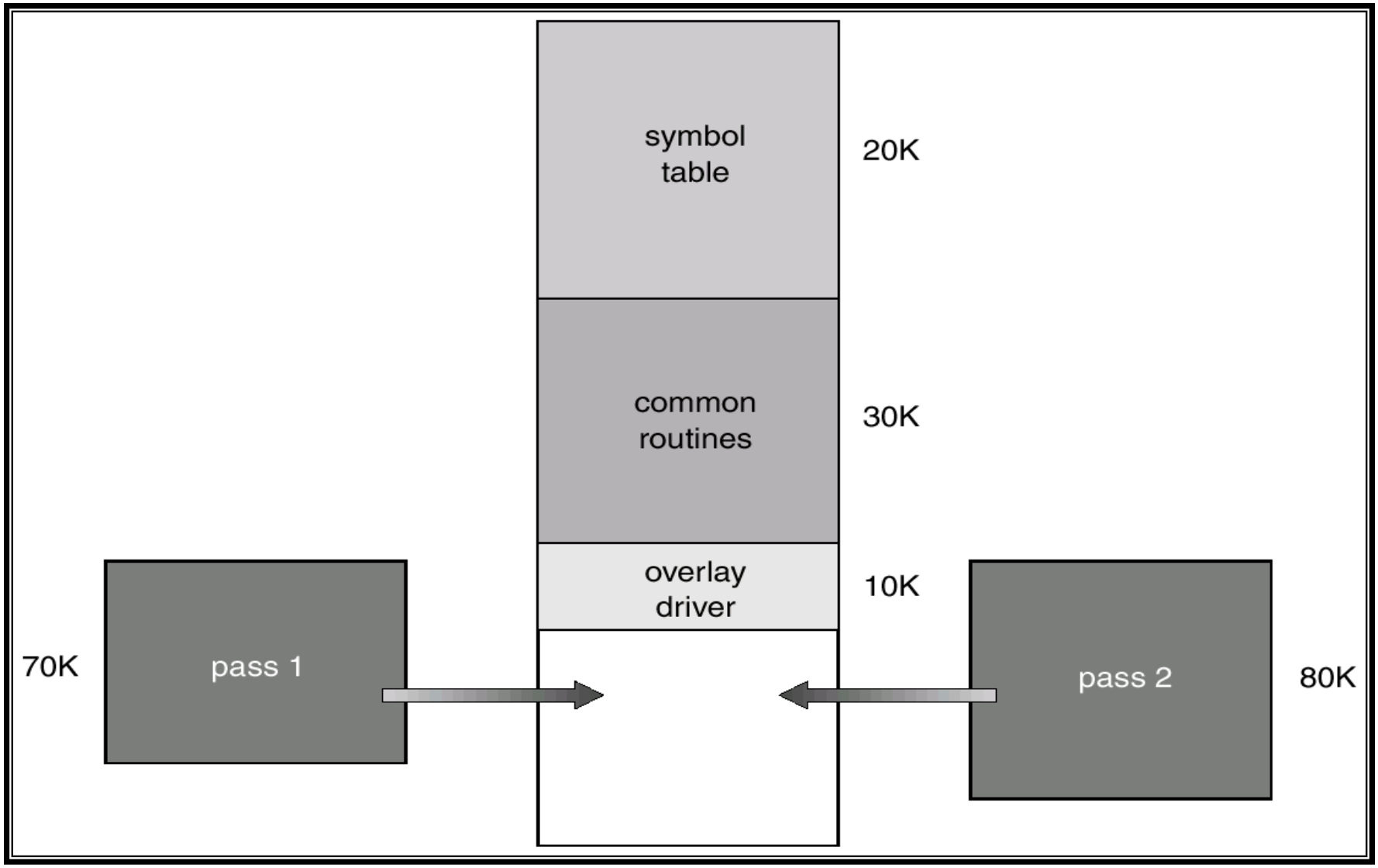
- any program smaller than a partition can be loaded into the partition
- if all partitions are occupied, the operating system can swap a process out of a partition to make room
- a program may be too large to fit in a partition. The programmer would then use *overlays*:
  - when a module needed is not present the *user program* must load that module into a part of the program's partition, overlaying whatever program or data were there before

Overlays

Partition



Overlays for a Two-Pass Assembler



Overlays

- Keep in memory only the overlay (those instructions and data that are) needed at any given phase/time.
- Overlays can be used only for programs that fit this model, i.e., multi-pass programs like compilers.
- Overlays are designed/implemented by programmer. Needs an overlay driver.
- No special support needed from operating system, but program design of overlays structure is complex.

Dynamic Linking

- Dynamic linking is useful when large amounts of code are needed to handle infrequently occurring cases.
- Routine is not loaded unless/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.
- Not much support from OS is required – implemented through program design.

Dynamics of 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.
- OS needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for shared/common libraries – here full OS support is needed.

Advantages of Dynamic Linking

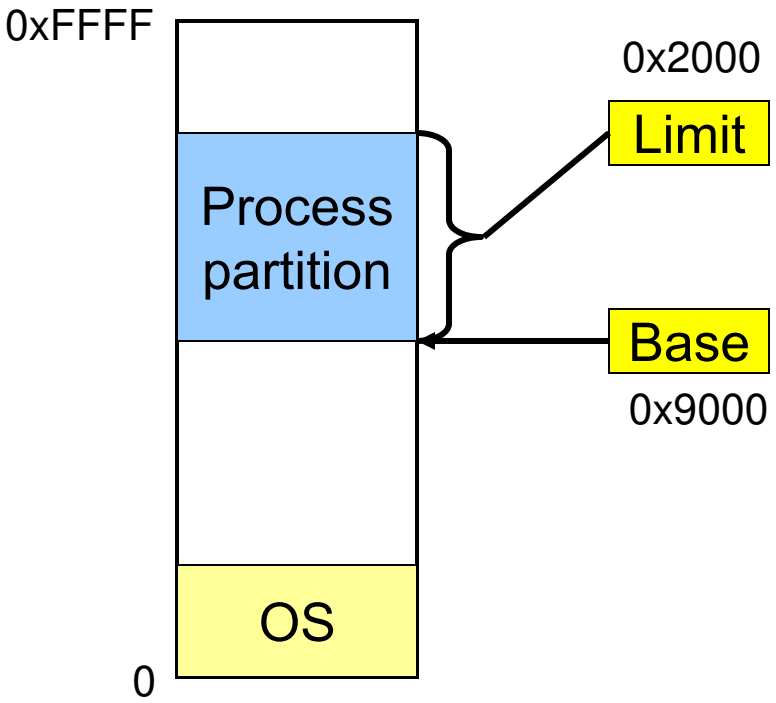
- Executable files can use another version of the external module without the need of being modified.
- Each process is linked to the same external module.
  - Saves disk space.
- The same external module needs to be loaded in main memory only once.
  - Processes can share code and save memory.
- Examples:
  - Windows: external modules are .DLL files.
  - Unix: external modules are .SO files (shared library).

Calculating Absolute Address

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system

Base and limit registers

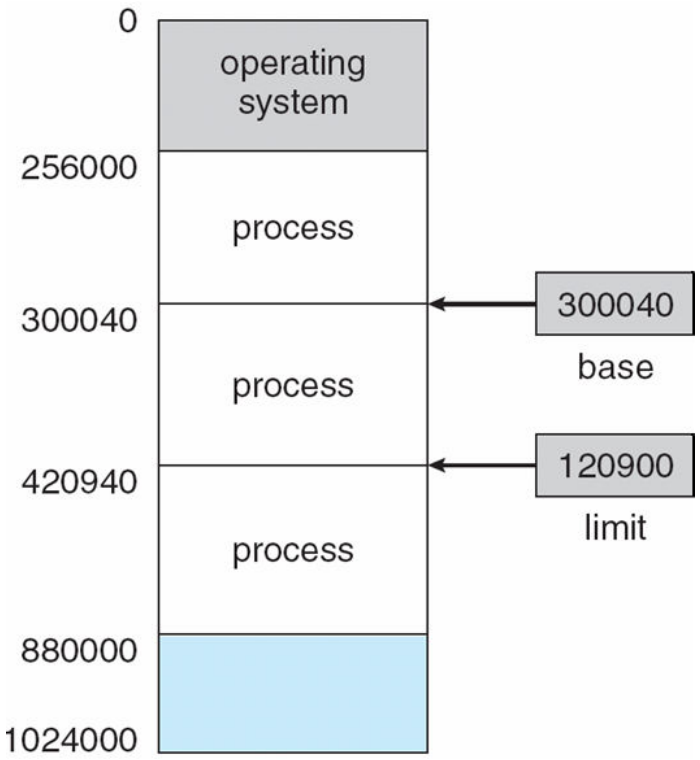
- **Special CPU registers: base & limit**
  - **Access to the registers limited to system mode**
  - **Registers contain**
    - Base: start of the process's memory partition
    - Limit: length of the process's memory partition
- **Address generation**
  - **Physical address: location in actual memory**
  - **Logical address: location from the process's point of view**
  - **Physical address = base + logical address**
  - **Logical address larger than limit => error**



Logical address: 0x1204
Physical address:
0x1204+0x9000 = 0xa204

Base and Limit Registers

- 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



Address Binding

- Programs on disk, ready to be brought into memory to execute form an input queue
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - 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

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time:
    - Code is fixed to an absolute address.
    - must recompile code if starting location changes

Binding of Instructions and Data to Memory

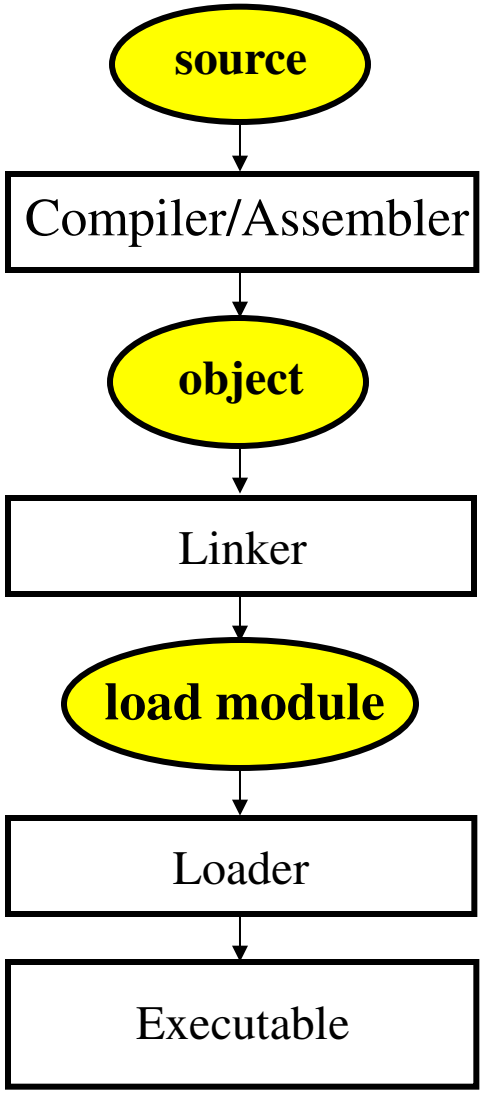
- Address binding of instructions and data to memory addresses can happen at three different stages
  - Load time:
    - Code can be loaded to any portion of memory. (Relocatable code)
    - If memory location is not known at compile time, compiler must generate relocatable code.
    - Loader knows final location and binds addresses for that location

Binding of Instructions and Data to Memory

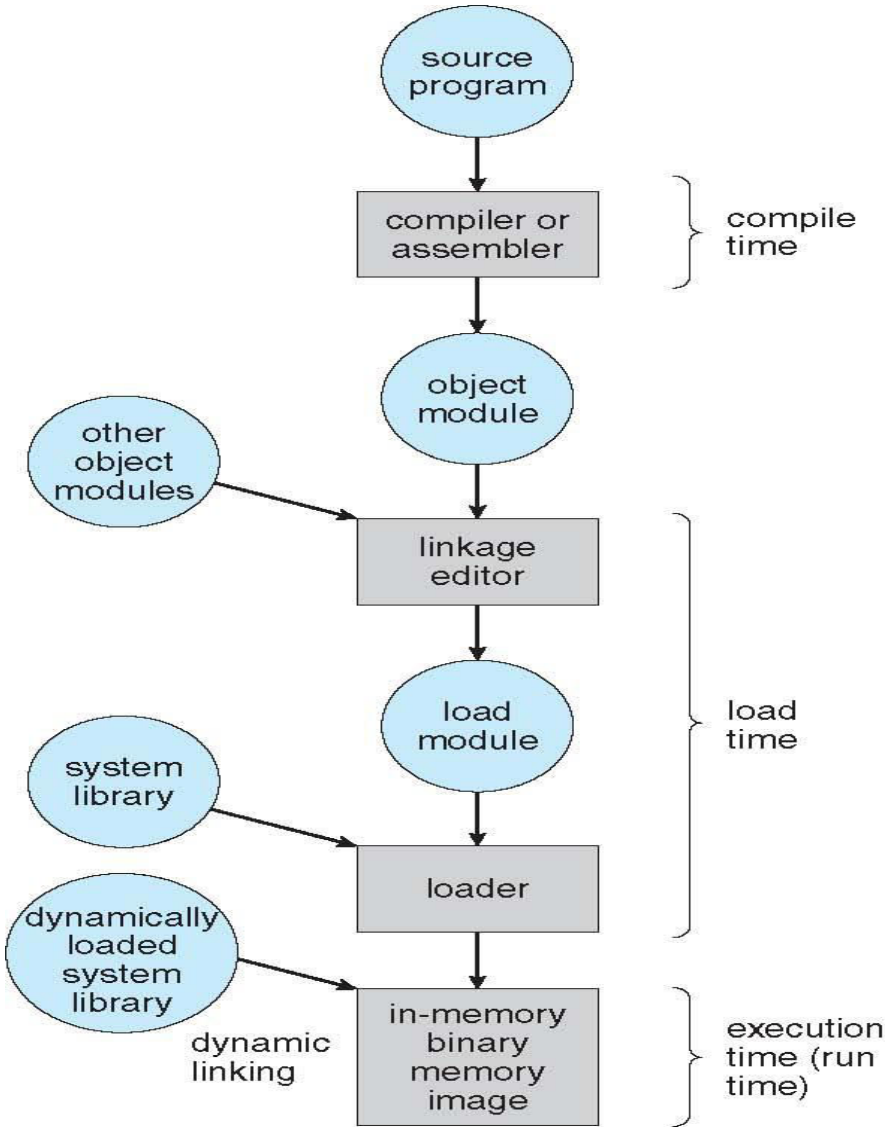
- Address binding of instructions and data to memory addresses can happen at three different stages
  - Execution (Run) time:
    - Code can be move to any portion of memory during its execution.
    - Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Address Binding

- **A process must be tied to a physical address at some point (bound)**
- **Binding can take place at 3 times**
  - **Compile time**
    - Always loaded to same memory address
  - **Load time**
    - relocatable code
    - stays in same spot once loaded
  - **Execution time**
    - may be moved during execution
    - special hardware needed



Multistep Processing of a User Program



Memory Management Requirements

- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle.
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible. Need additional support for:
  1. Swapping (Relocation)
  2. Protection
  3. Sharing
  4. Logical Organization
  5. Physical Organization

Swapping and Scheduling

- **Swapping**
  - **Move process from memory to disk (swap space)**
    - Process is blocked or suspended
  - **Move process from swap space to big enough partition**
    - Process is ready
    - Set up Base and Limit registers
  - **Memory Manager (MM) and Process scheduler work together**
    - Scheduler keeps track of all processes
    - MM keeps track of memory
    - Scheduler marks processes as swap-able and notifies MM to swap in processes
    - Scheduler policy must account for swapping overhead
    - MM policy must account for need to have memory space for ready processes

Swapping (Relocation)

- Swapping (Relocation)
  - Programmer does not know where the program will be placed in memory when it is executed
  - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  - Memory references in the code must be translated to actual physical memory address

Swapping enables the OS to have a larger pool of ready-to-execute processes.

Compaction enables the OS to have a larger contiguous memory to place programs in.

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

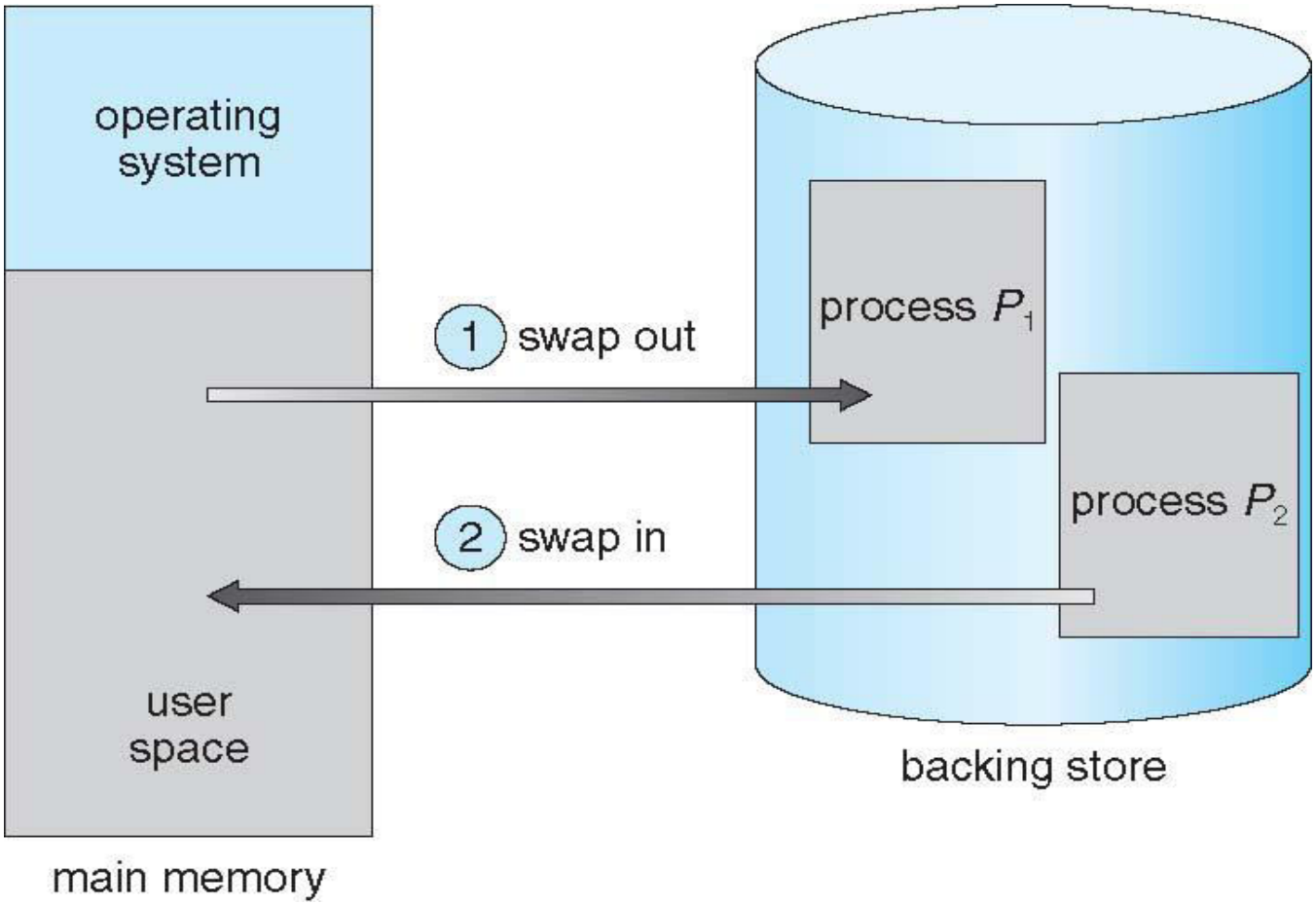
Swapping

- 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

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
- 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

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**

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- An executing process must be loaded entirely in main memory (if overlays are not used).
- 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*
  - Can then allow actions such as kernel code being transient and kernel changing size