Operating Systems (INFR09047)

2019/2020 Semester 2

Memory Management

abarbala@inf.ed.ac.uk

Chapter 9 (9.1, 9.2, 9.5)

Overview

- Background: basics
- Running without any memory abstraction
- Background: binding
- Running with compiler support
- Base + Limit Registers
- Address Space and Logical memory
- MMU, Relocation + Limit Registers
- Contiguous Memory Allocation
- Swapping
- Fragmentation

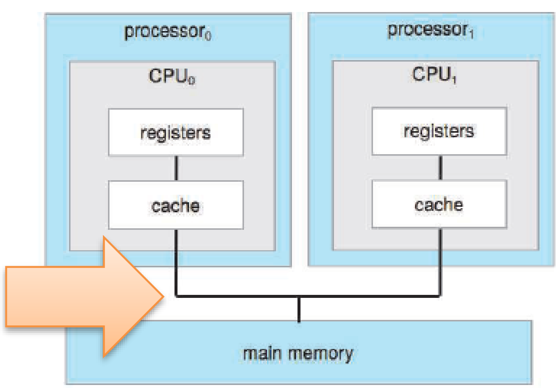
Goals of Memory Management

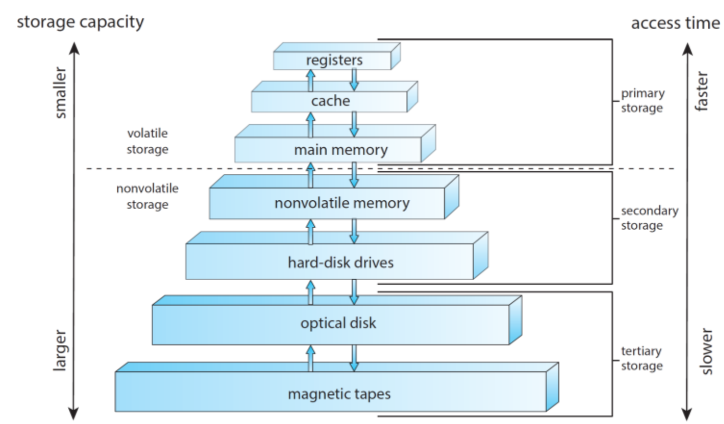
- **Allocate** memory resources among **processes and OS**
  - Maximizing memory utilization and system throughput
- Provide **convenient abstraction** for **processes (applications) and OS** programmers
  - Simplify memory utilization and addressability
- Provide **isolation** between **processes and OS**
  - Addressability and protection orthogonal problems

Background: Basics

- Program must be brought (from disk) **into memory** and **placed within a process context** to be run
- *Main memory* and *CPU registers* are the only storage **CPU can access directly** (e.g., with a CPU instruction)
  - Memory unit only sees
    - Physical memory address + read request, or
    - Physical memory address + data and write request
  - Register access in one CPU clock (or less)
  - Main memory takes multiple CPU cycles, causing **CPU to stall**
  - **Cache** sits between main memory and CPU registers
    - Reduces CPU cycles to access memory
    - Transparent to the (assembly) programmer

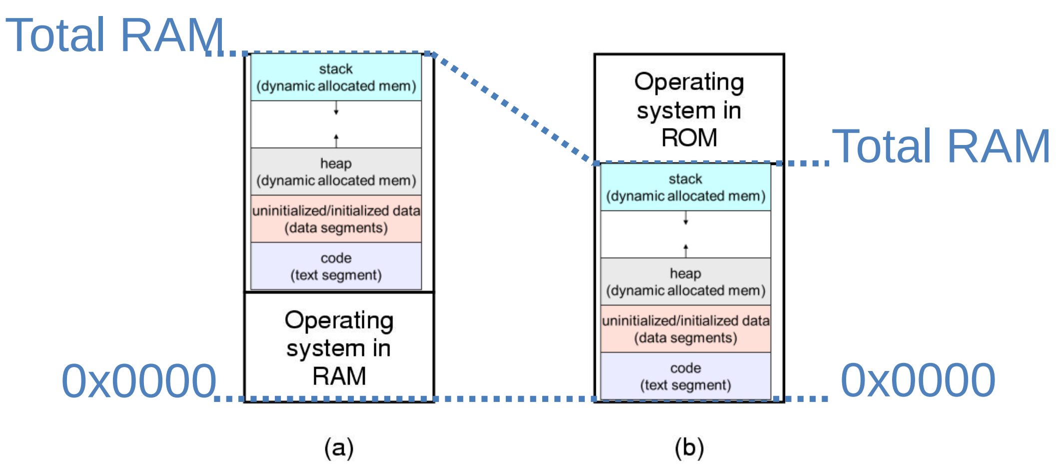
Physical Addr + Request type + Data (if W)





One Program with No Memory Abstraction

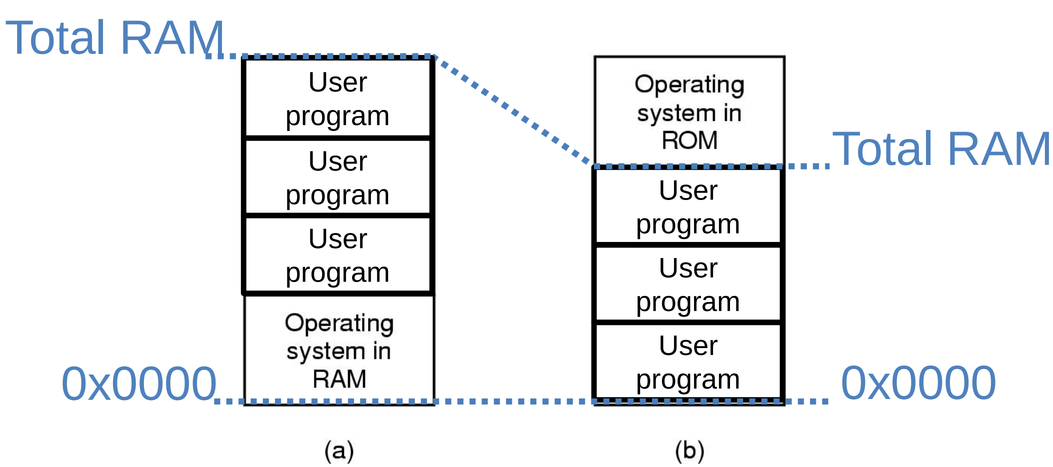
- Program sees (access) the physical memory
- Sharing physical memory with the OS (and even BIOS)
  - **Program can mess up with OS (and BIOS)**
  - Example, MSDOS



Two simple ways of organizing memory with an operating system and one user process.

Multiple Programs with No Memory Abstraction

- **Every** program sees physical memory
  - Can access each other memory
  - Total # of programs depends on their size and memory size
- Sharing physical memory with the OS and even BIOS
  - **Programs can mess up with OS and BIOS**

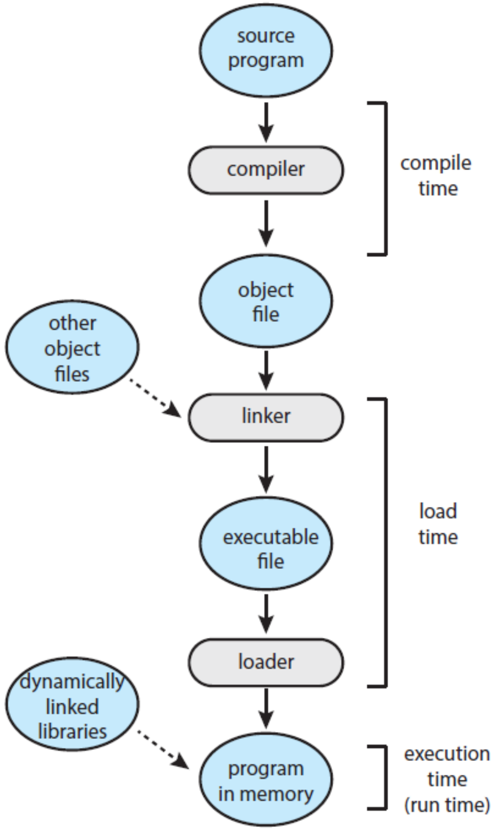


Two simple ways of organizing memory with an operating system and **multiple** user processes.

Background: Binding

It is all about (memory) addresses

- Addresses in a program are usually symbolic (e.g. variable *count*)
  - Binding is the process of mapping
- Compiler (From source to object file)
  - **Binds** symbolic addresses to *relocatable addresses*
    - **Relative addresses, e.g., from the beginning of …**
- Linker (From object file to executable file)
  - **Binds** relocatable to *absolute addresses*
    - If the **final** memory location is known
    - Cannot be moved without hardware support
- Loader (Executable file image moved to memory)
  - **Binds** relocatable to *absolute addresses*
    - Cannot be moved without hardware support
- Execution



Multiple Programs: Relocation Problem #1

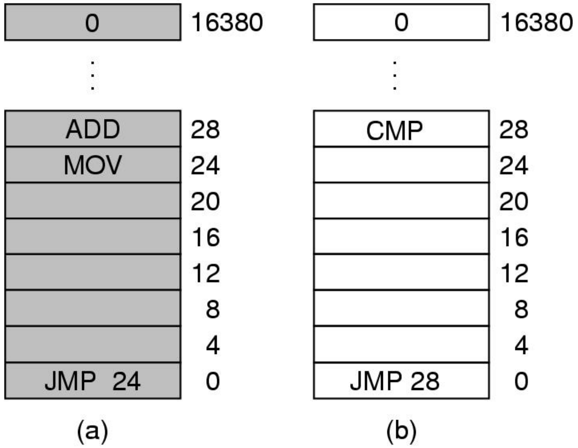


Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Multiple Programs: Relocation Problem #1

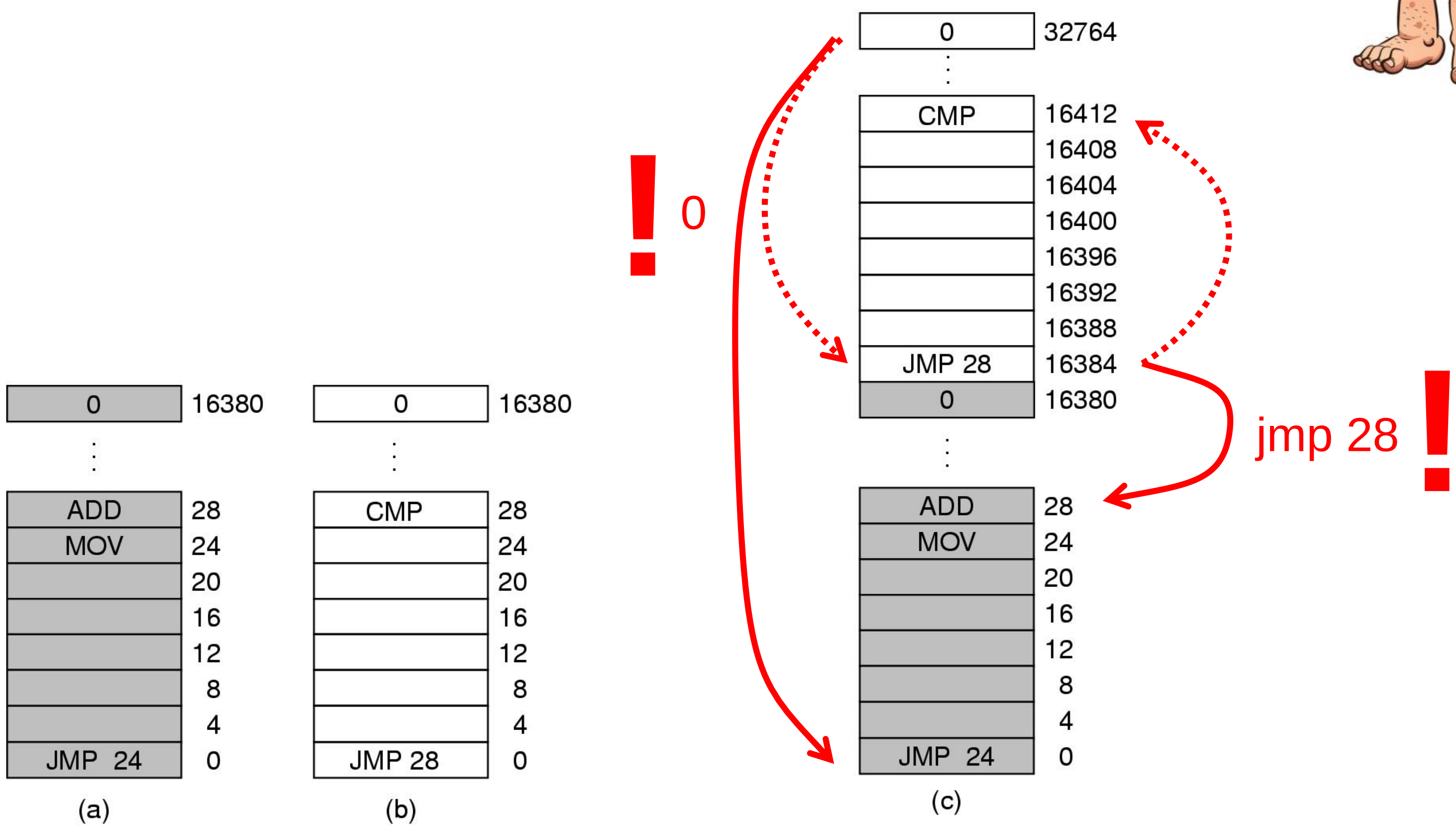
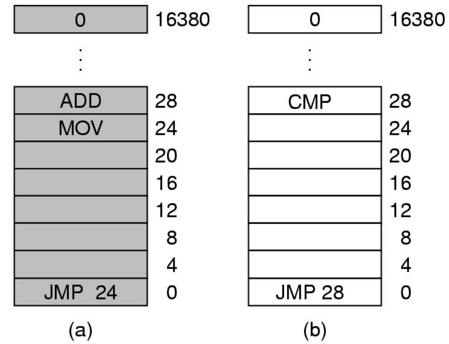
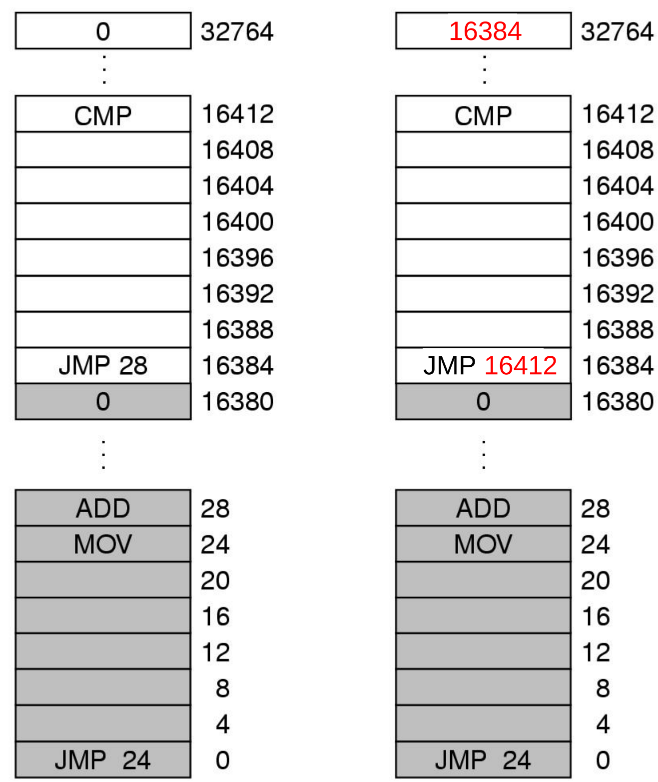


Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Multiple Programs: Relocation #2

- Programs must be **relocatable**
  - Written to be **placed and run** at any memory address
  - Extra information in executable
- **Loader** decides where to place them
  - Based on the available physical memory
  - It does relocation (increases load time)





Physical Memory (before relocation)

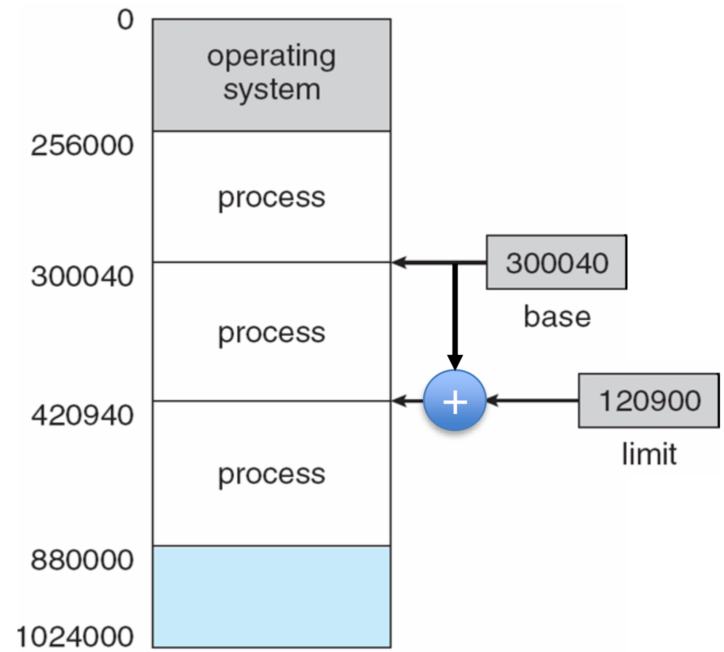
Physical Memory (after relocation)

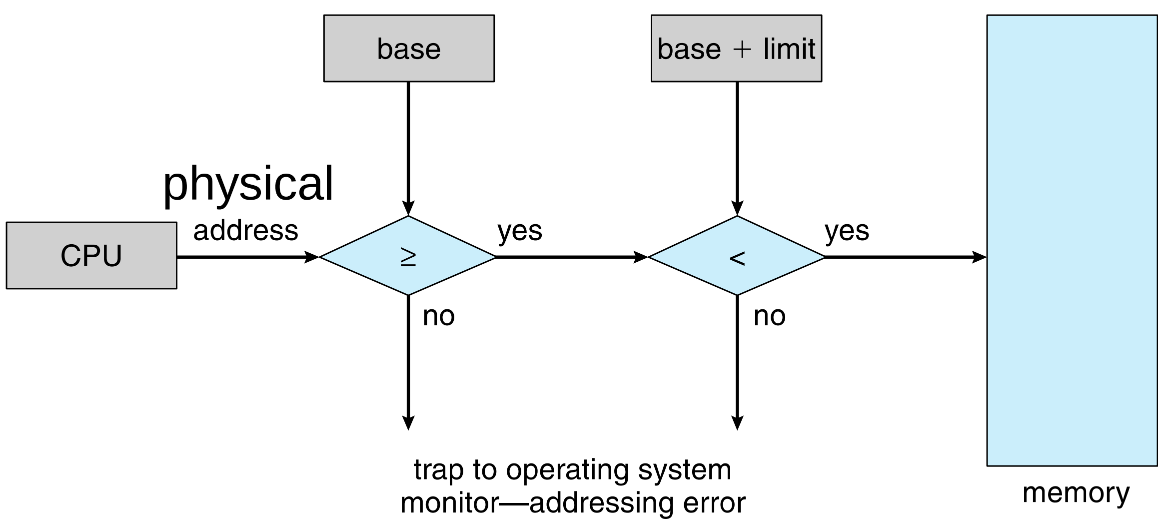
Multiple Programs on Physical Memory

- All physical memory exposed to processes
  - User processes may **interfere** with OS and each other
  - Processes may **access** OS's and each other **secrets**
  - Programs must be relocated when loaded
- **Problems**
  - **No protection**
  - **Expensive relocation**

Protection: Base and Limit Registers

- **Base** and **limit registers** define the **valid** addresses for a process
- CPU checks every memory access generated by the process
- If the address is not valid (outside the range) traps to OS

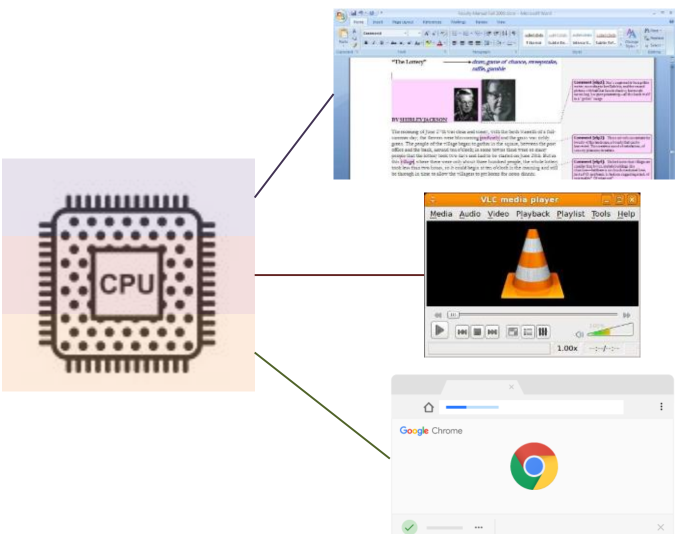




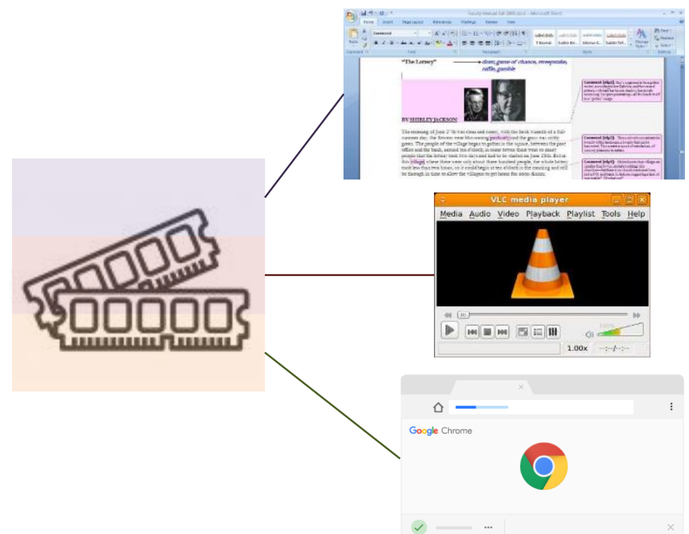
Memory Abstraction: Address Space

NOT just physical memory addresses!

- Address Space
  - Abstraction from physical memory space
  - Set of memory addresses that a process can use
    - Independently from other processes



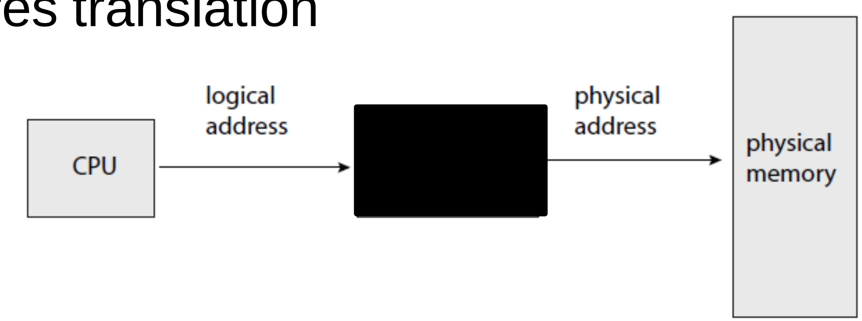
Process, abstracts physical CPU



Address space, abstracts physical memory

Logical Addresses #1

- To make it easier to manage memory of multiple processes
- Make processes use logical addresses
  - Logical addresses are independent of physical addresses
    - Data lives in physical addresses
    - OS manages data location in physical memory
- Instructions issued by CPU are logical addresses
  - Example: pointers, arguments to load/store instructions, PC, etc.
- Logical addresses are translated by hardware into physical addresses
  - OS configures translation



Logical Addresses #2

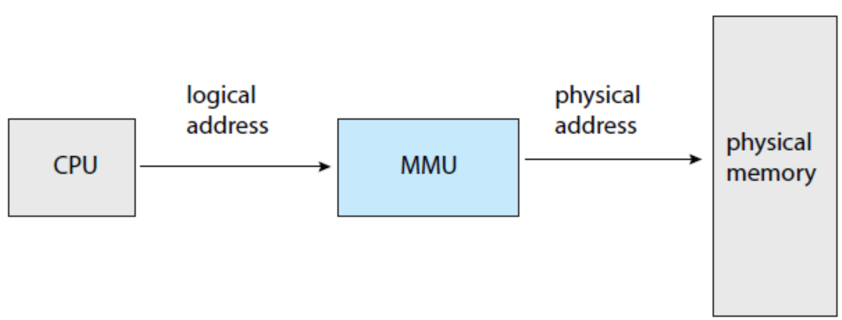
- Set of logical addresses a process can reference is its address space
- Program issues addresses in a logical address space
  - Must be translated to physical address space
  - Think of the program as having
    - A contiguous **logical address space** that starts at 0
    - A contiguous **physical address space** that starts somewhere

- **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 (after translation)

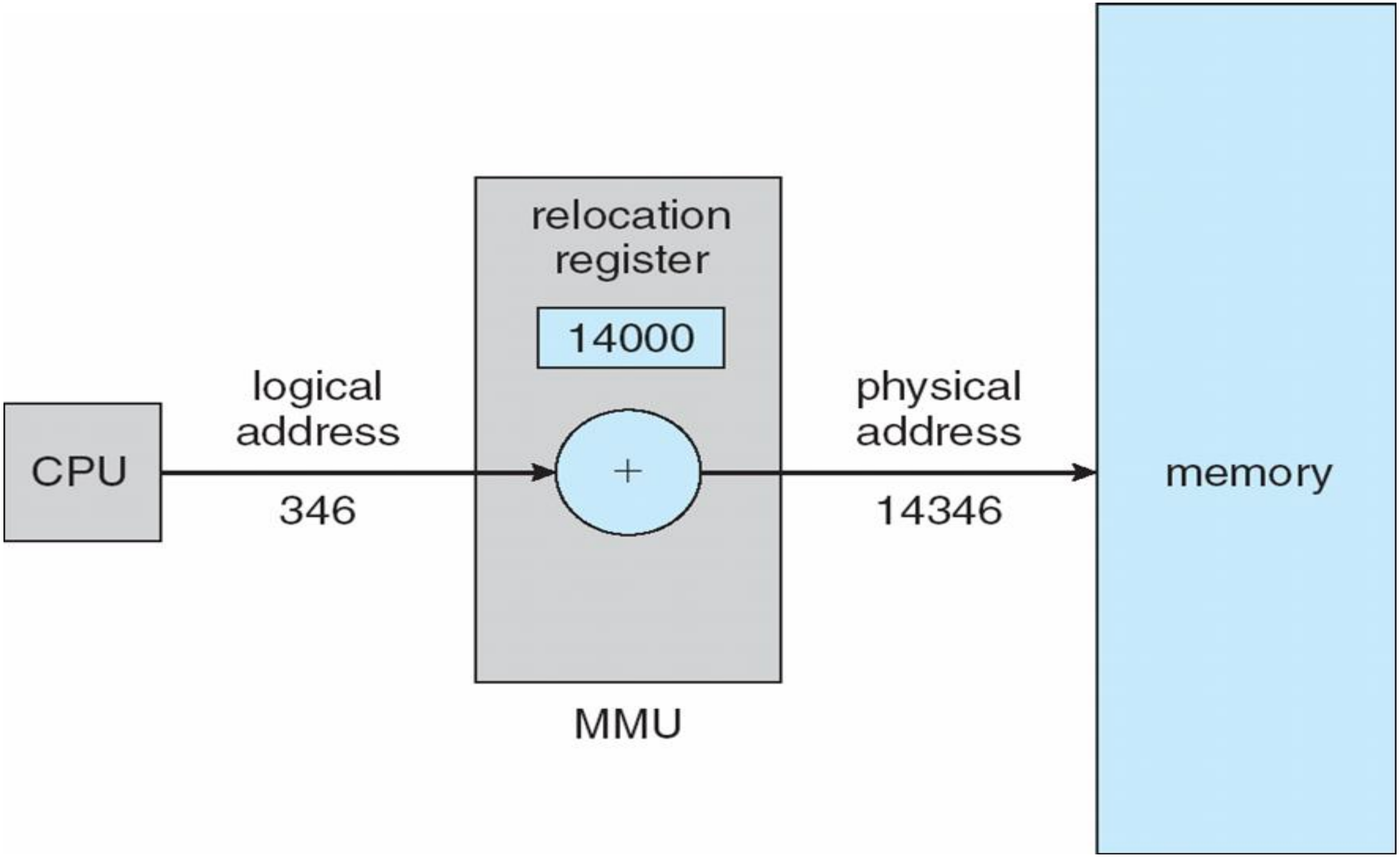
Memory-Management Unit (MMU)

- Hardware component
  - **Translates** CPU generated addresses to physical addresses
- Programs deal with *logical addresses*; never see physical *addresses*
  - Logical address *bound to* physical addresses

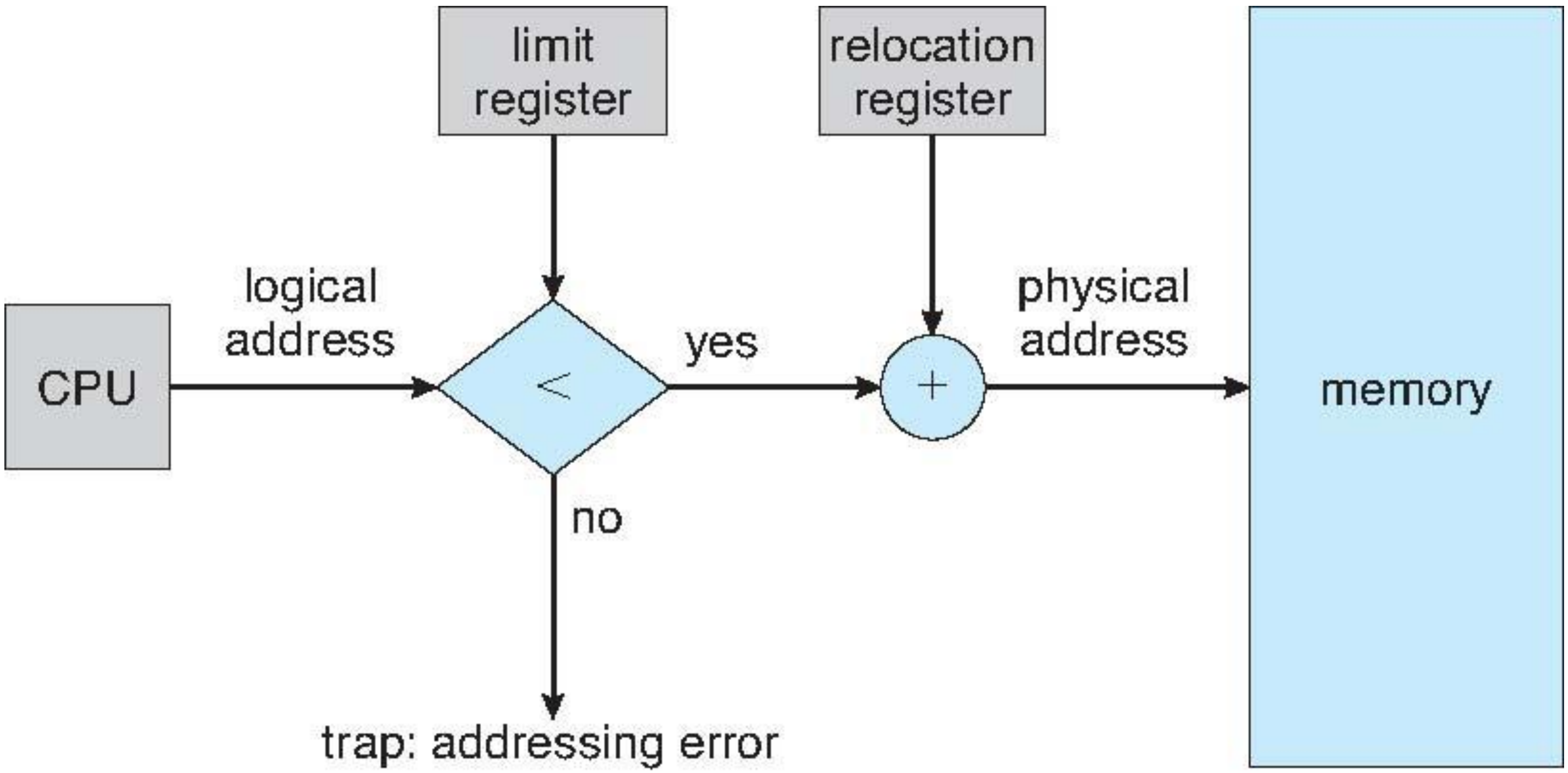
- Many implementations
  - Base+limit registers, segmentation, paging, etc.
- Many names, based on features
  - MMU, MPU, etc.



MMU as a Relocation Register

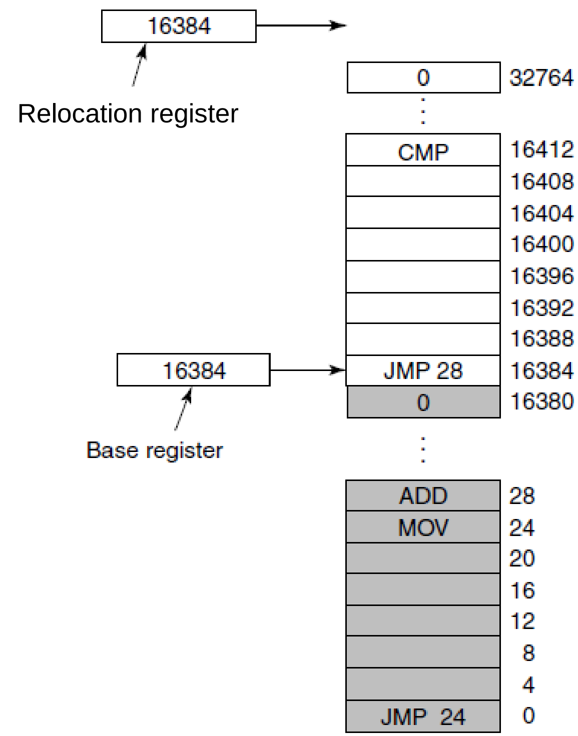


MMU as a Relocation and Limit Registers

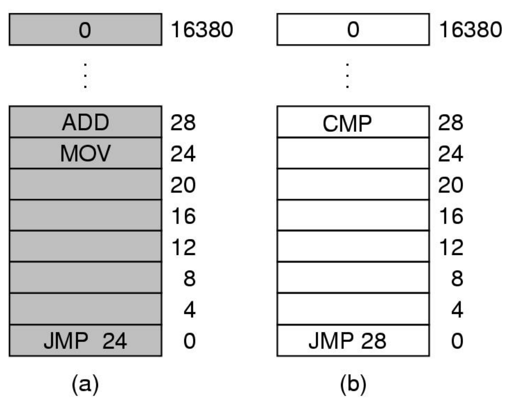


Multiple Programs: Relocation and Limit Registers #1

- **Hardware support** to ease relocation
  - Base register
  - Limit register
- **Program not relocatable**, simple loader
  - Faster load time
- Protection
  - Each process its own private address space

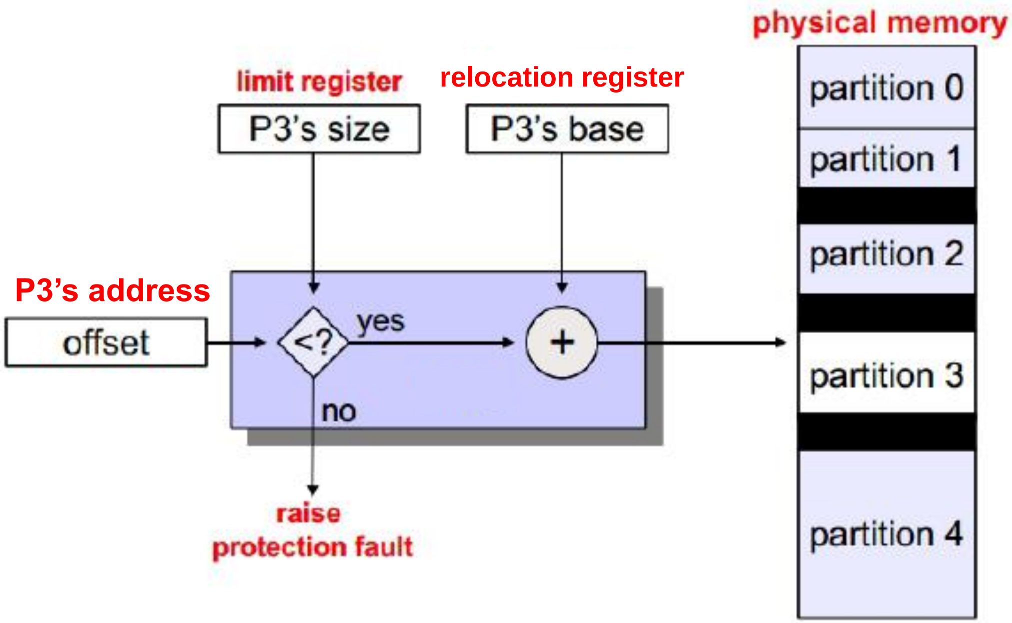


Base and limit registers can be used to give each process a separate address space.



Multiple Programs: Base and Limit Registers #2

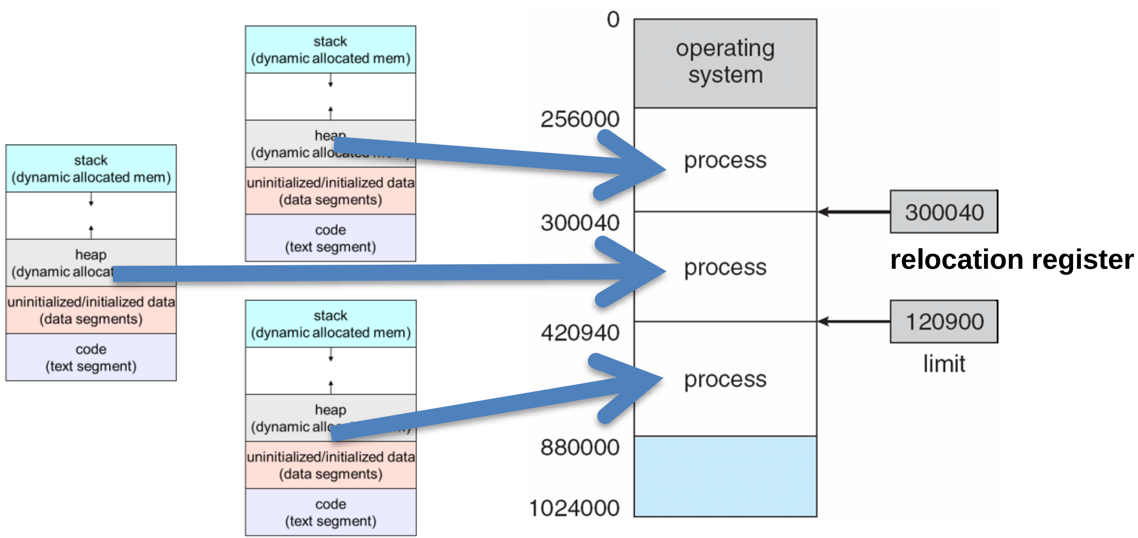
- Variable size partitions (program size)
  - Base register and a limit register arithmetic
  - Arithmetic performed for each memory access



Contiguous Allocation #1

- Main memory **must** support both OS and user processes
- Memory limited resource **must** allocate efficiently

- Contiguous allocation is an *early method*
- Main memory usually into two **partitions**
  – OS (usually) held in low memory with interrupt vector
  – Processes held in high memory
    - Each contained in single contiguous section of memory



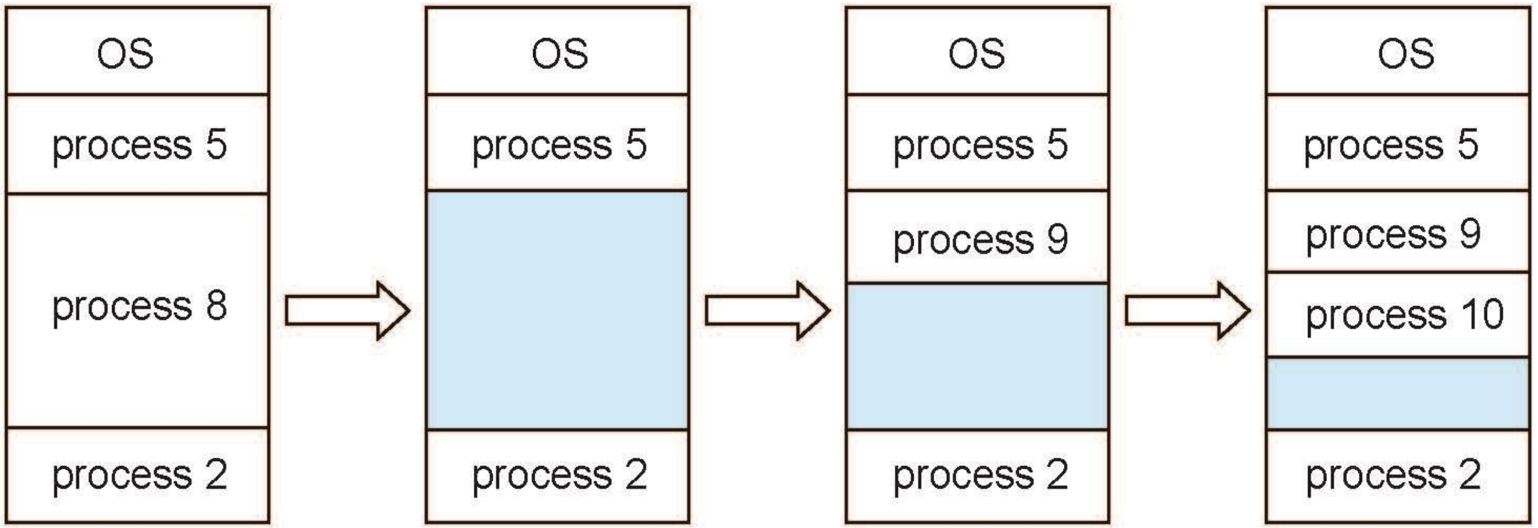
Contiguous Allocation #2

- Relocation and limit registers
  - To **protect user processes**
    - from each other
    - from changing OS code and data
  - Relocation 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 translates logical address
  - *transparently, during execution*

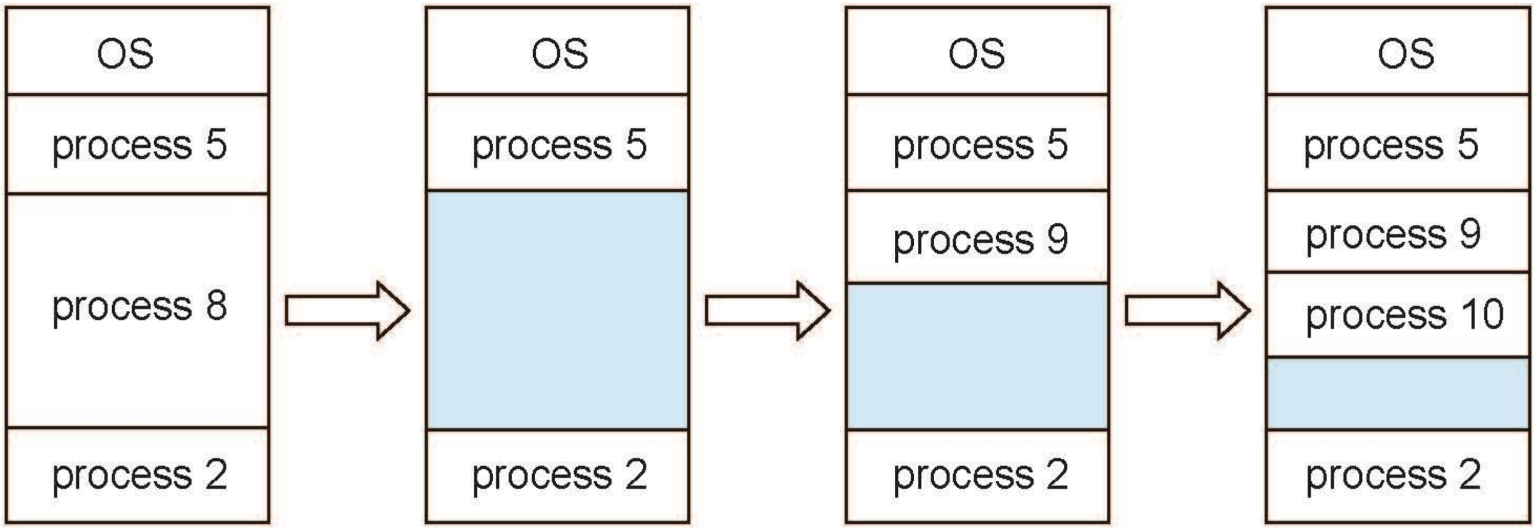
Multiple-partition Allocation #1

- Multiple-partition allocation
  - Variable partition size for efficiency (size of a program)
  - Degree of multiprogramming limited by number of partitions



Multiple-partition Allocation #2

- Multiple-partition allocation
  - **Hole:** block of available memory; holes of various sizes
  - When a process arrives, **OS allocates** memory from a hole large enough to accommodate it
  - Process exiting, **returns partition to OS** , adjacent free partitions combined
  - Operating system maintains information about
    - allocated partitions
    - free partitions (hole)



Dynamic Storage-Allocation Problem

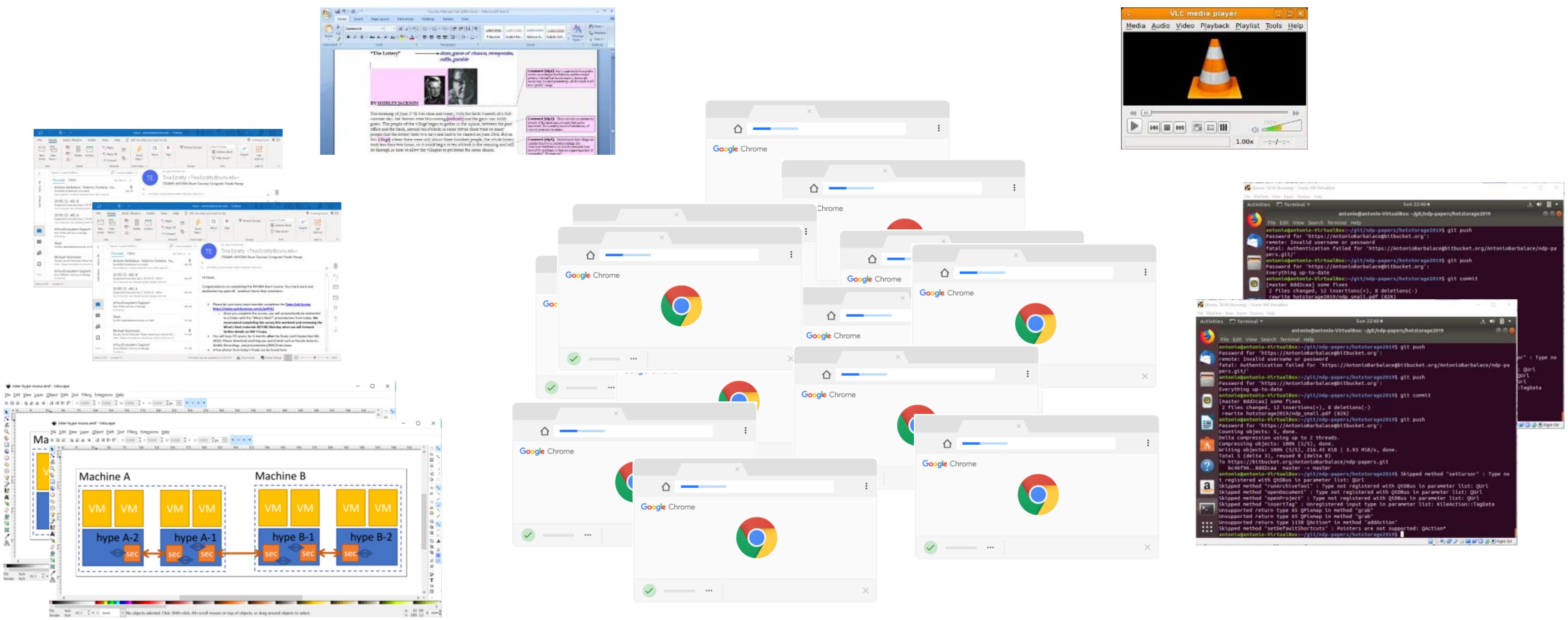
How to satisfy a request of size ***n*** from a list of free holes?

- **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

Multiple Programs: Swapping (1)

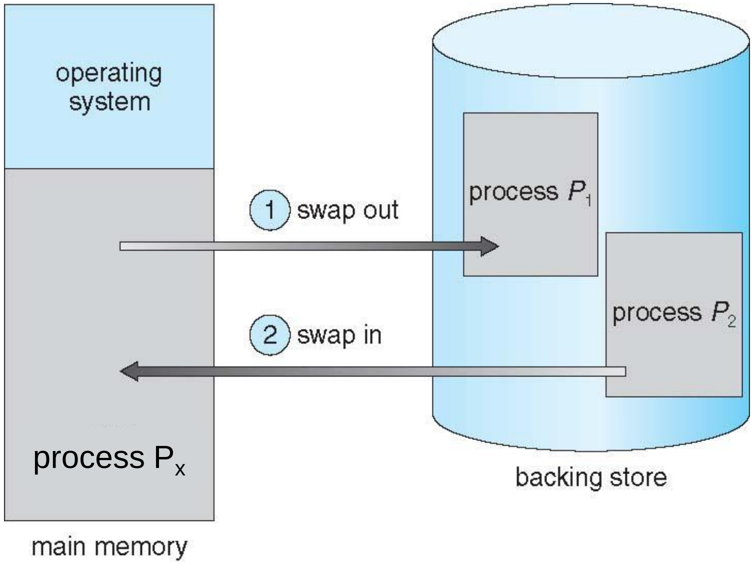
- How many programs are currently executed?
- Do they all fit in memory?



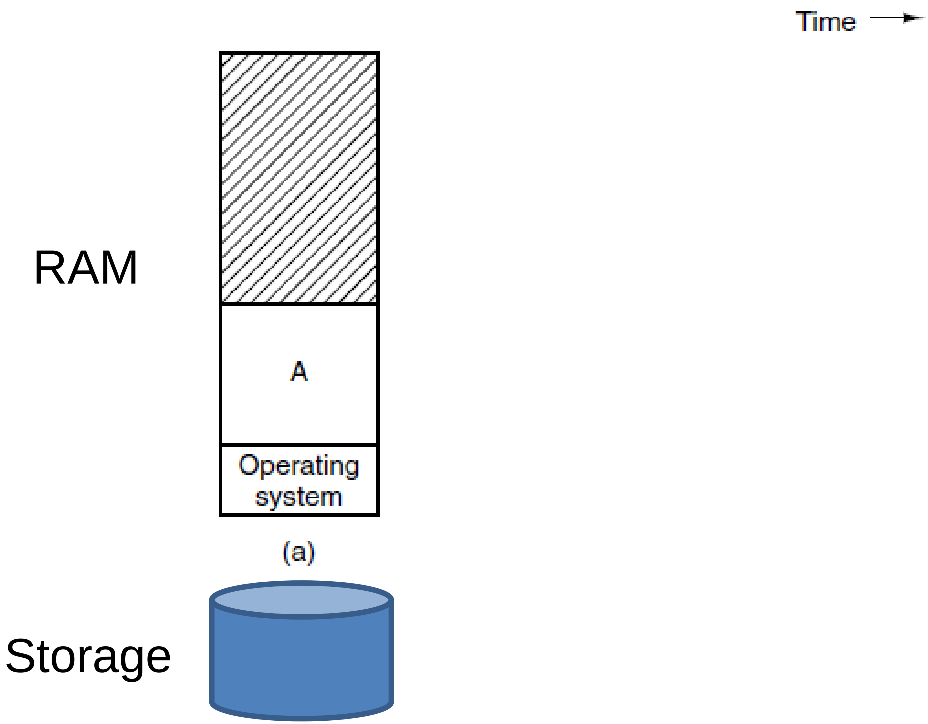
Multiple Programs: Swapping (2)

- How many programs are currently executed?
- Do they all fit in memory?

- Swapping
  a) **SWAP IN:** Bringing in memory a process in its entirety (from disk)
  b) **RUN:** Running it for a while
  c) **SWAP OUT:** Putting it back from memory (to disk)
  – About **running** processes, no programs!
  – Idle processes are stored on disk,
    - Do not take up any memory when they are not running

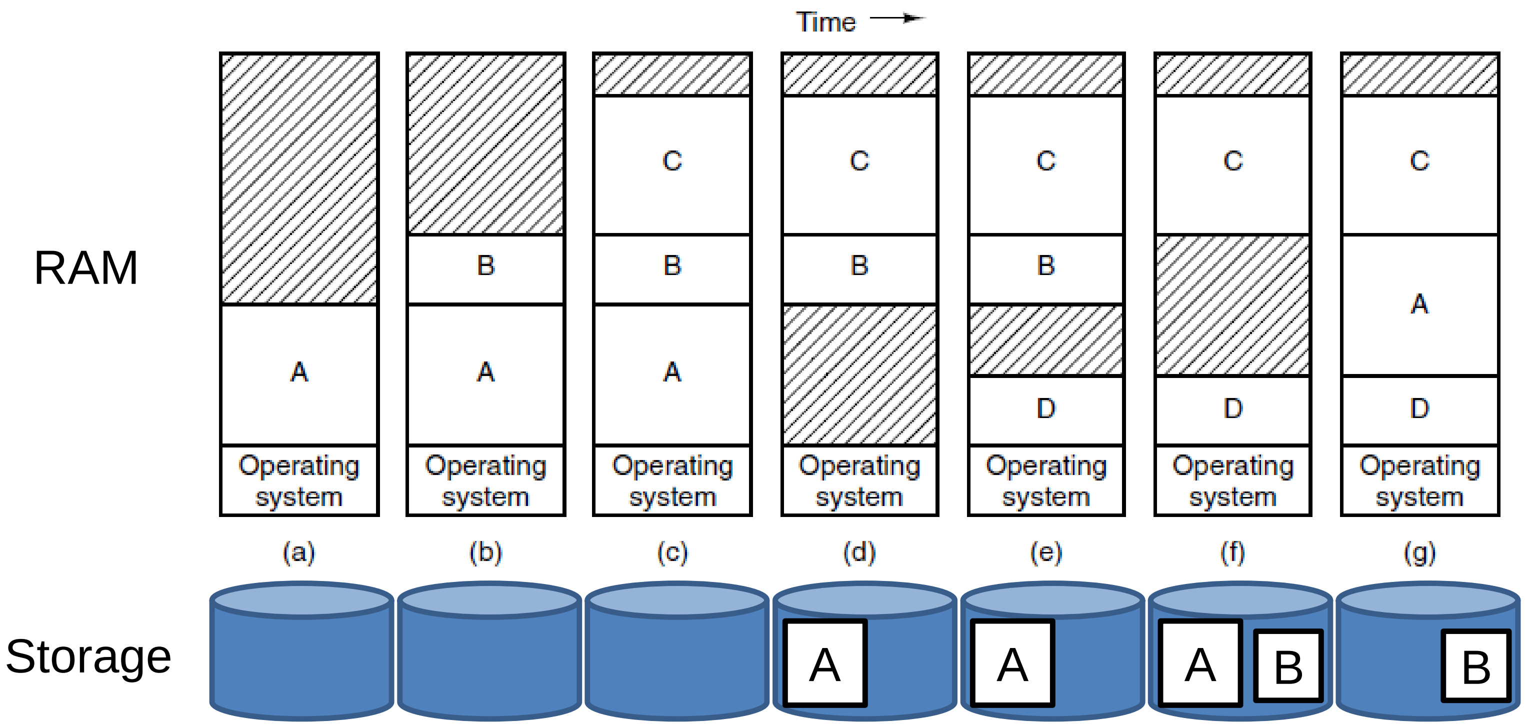


Multiple Programs: Swapping (3)



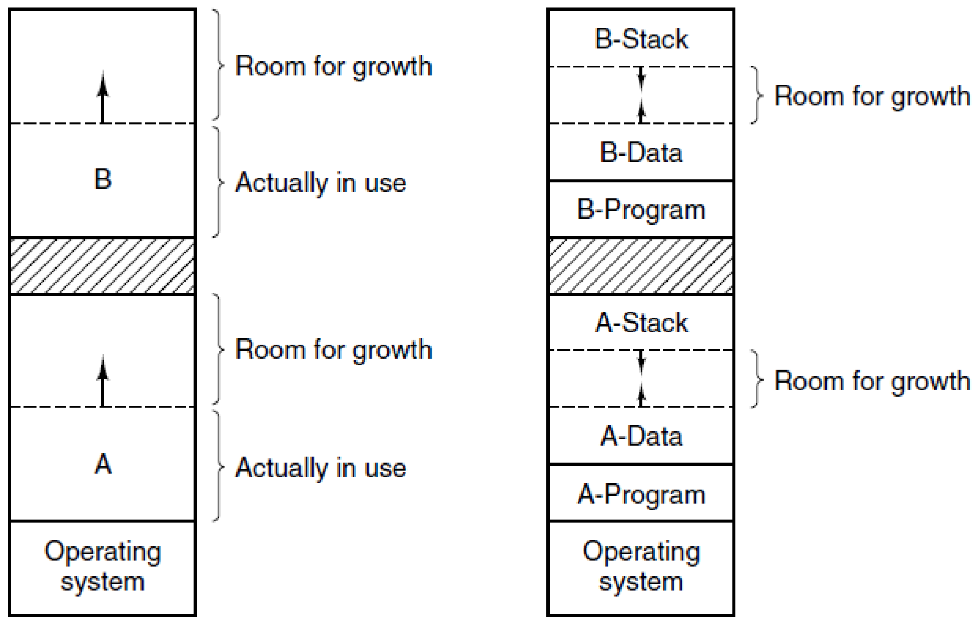
Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory (MOS Figure 3-4)

Multiple Programs: Swapping (3)



Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory (MOS Figure 3-4)

Multiple Programs: Growing Programs



(a) (b)

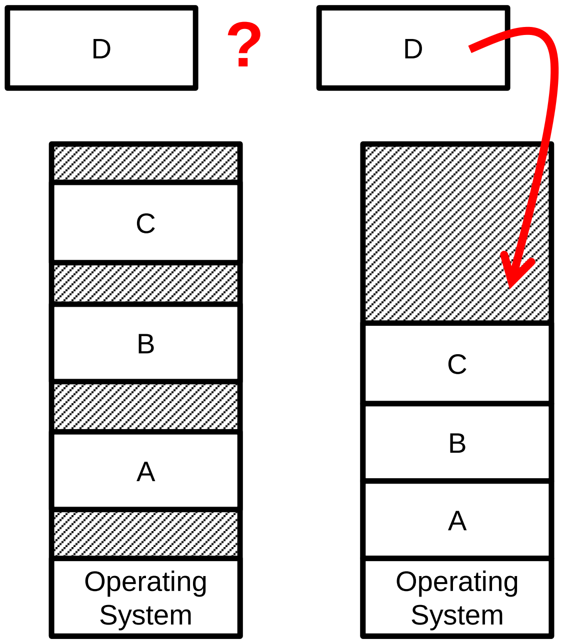
(a) Allocating space for a growing program at the end – if more space is needed, relocate

(b) Allocating space for a growing program in its address space – max amount of growth

(c) Both solutions are not ideal

Multiple Programs: Memory Fragmentation #1

- Process creation and exit, and swapping
- Create **memory holes**
  - New processes, memory is available, cannot be placed



- **Compaction**
  - Move all processes tightly together
    - By memory copy or
    - By swapping out and in
    - Computationally expensive
  - What if a process needs to grow its heap or stack?

Multiple Programs: Memory Fragmentation #2

- **External Fragmentation**
  - You allocate the exact amount of memory requested
  - Total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**
  - You allocate more than what required
  - Allocated memory may be slightly larger than requested memory
- First fit analysis reveals given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable

Concepts

- Relocatable binaries
- MMU
- Logical Address Space vs Physical Address Space
- Swapping
- Contiguous Memory Allocation
  - First fit, best fit, worst fit
- Fragmentation
  - Compaction