R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه دوازدهم – مدیریت حافظه (مقدمه)

جلسه‌ی گذشته

الگوریتم‌های زمان‌بندی پردازه‌ها

Scheduling Policies

- First-Come, First Served (FIFO)
- Shortest Job First (non-preemptive)
- Shortest Job First (with preemption)
- Round-Robin Scheduling
- Priority Scheduling
- Real-Time Scheduling

جلسه‌ی جدید

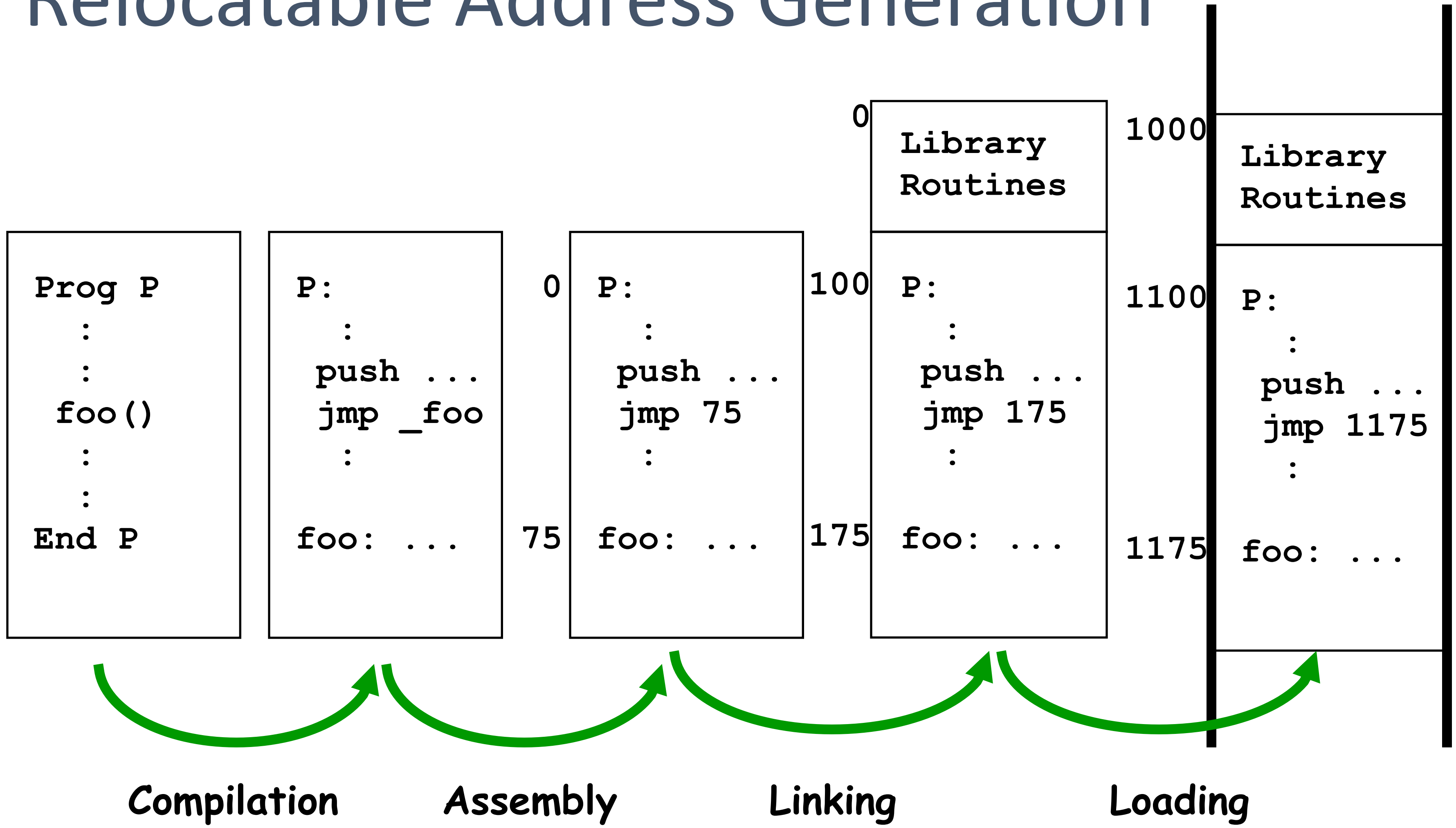
Memory Management

- Memory – a linear array of bytes
  - *Holds O.S. and programs (processes)*
  - *Each cell (byte) is named by a unique memory address*
- Recall, processes are defined by an *address space,* consisting of text, data, and stack regions
- Process execution
  - *CPU fetches instructions from the text region according to the value of the program counter (PC)*
  - *Each instruction may request additional operands from the data or stack region*

Addressing Memory

- Cannot know ahead of time where in memory a program will be loaded!
- Compiler produces code containing embedded addresses
  - *these addresses can't be absolute (physical addresses)*
- Linker combines pieces of the program
  - *Assumes the program will be loaded at address 0*
- We need to bind the compiler/linker generated addresses to the actual memory locations

Relocatable Address Generation



Address Binding

- Address binding
  - *fixing a physical address to the logical address of a process' address space*

Address Binding

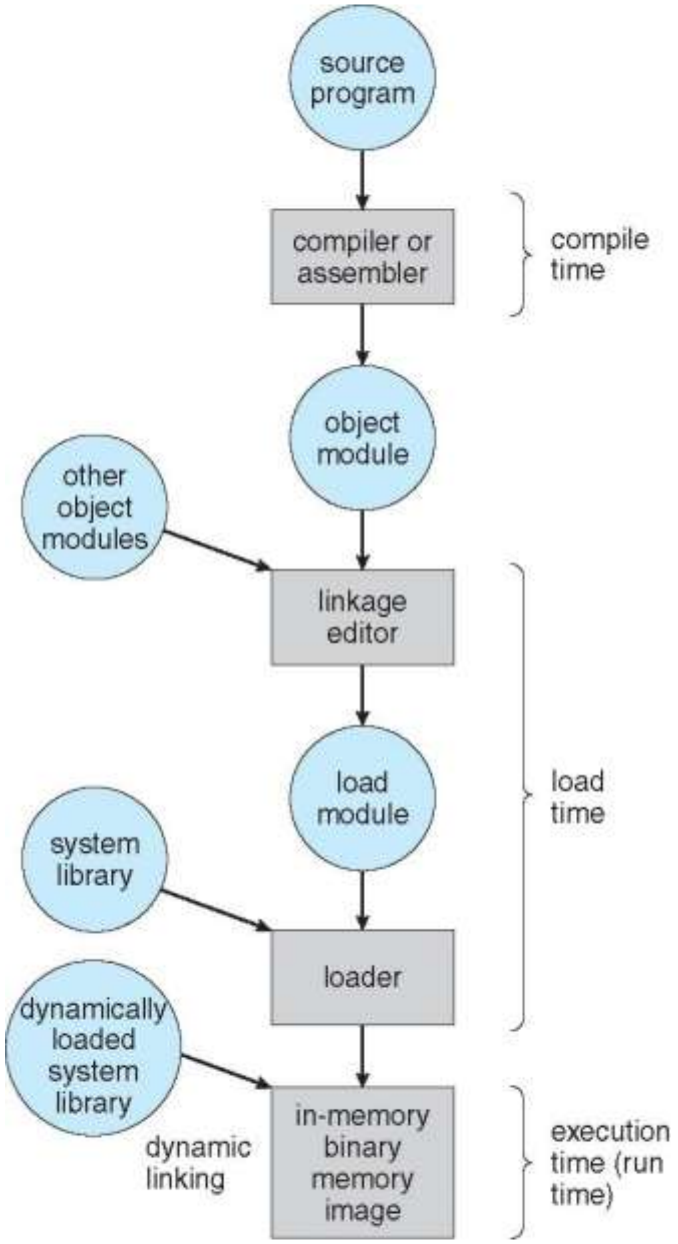
Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: 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)

Base and Limit Registers

- Simple runtime relocation scheme
  - *Use 2 registers to describe a partition*
- For every address generated, at runtime...
  - *Compare to the limit register (& abort if larger)*
  - *Add to the base register to give physical memory address*

Multistep Processing of a User Program

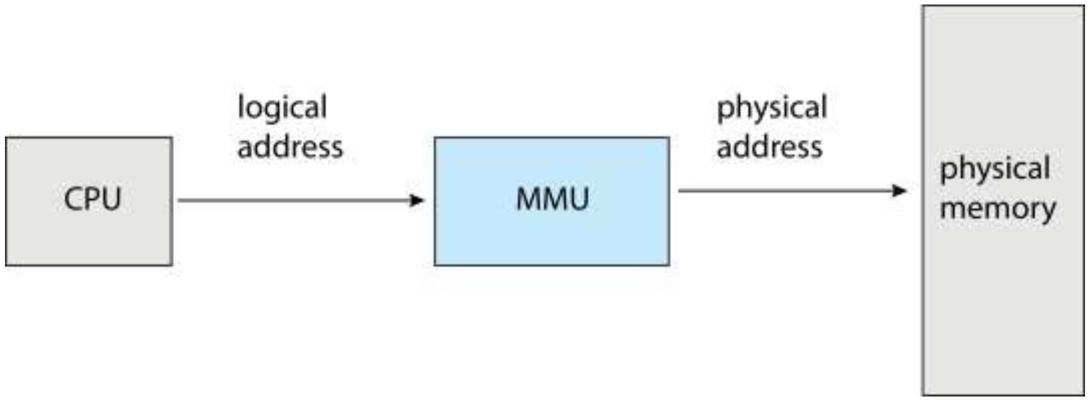


Logical vs. Physical Address Space

- 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
- **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)

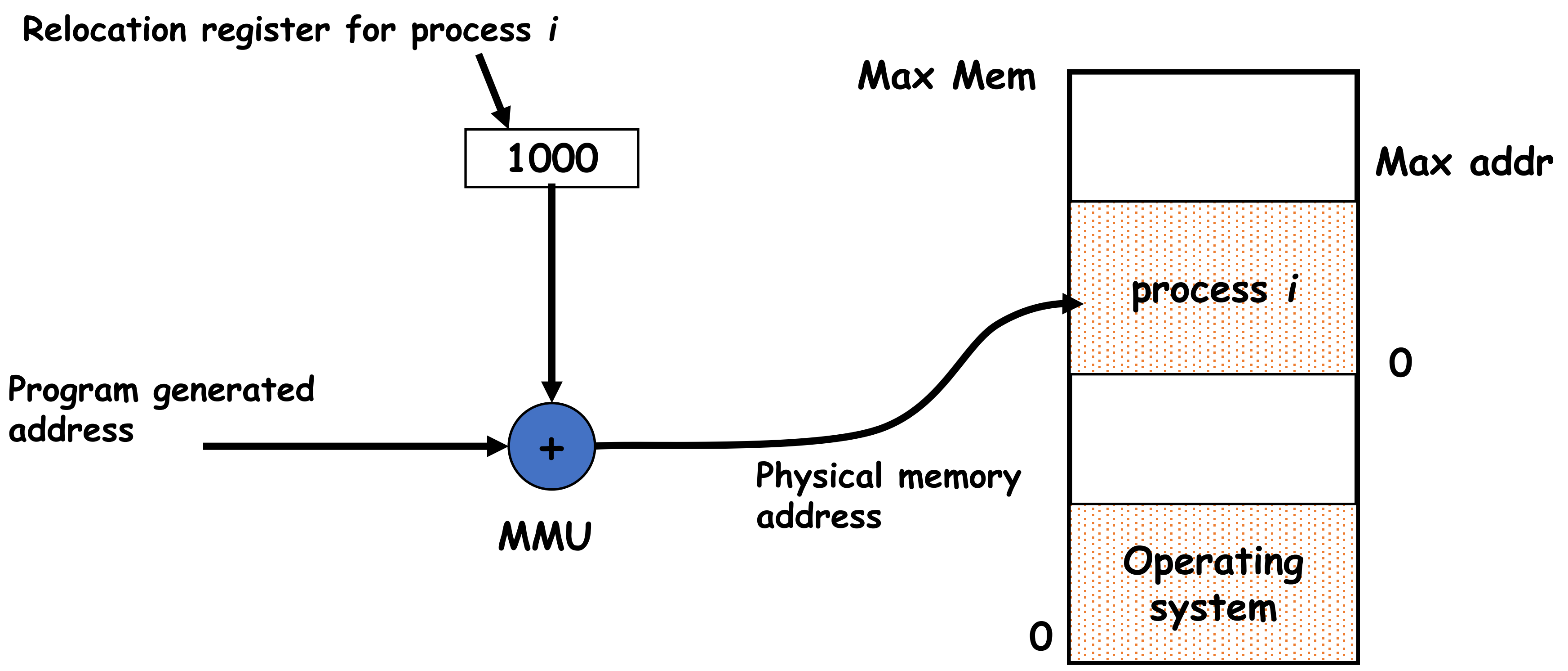
- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this lecture

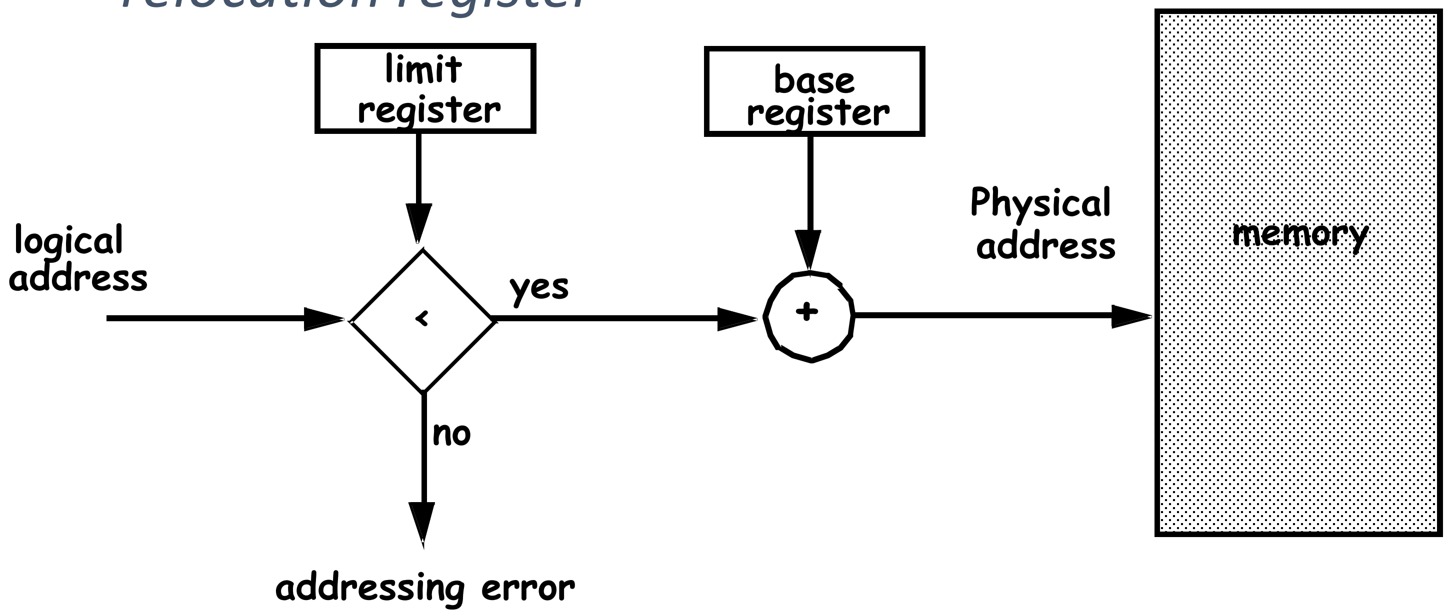
Dynamic Relocation

- **Memory Management Unit (MMU)**
- - Dynamically converts logical to physical address
- - Contains base address register for running process



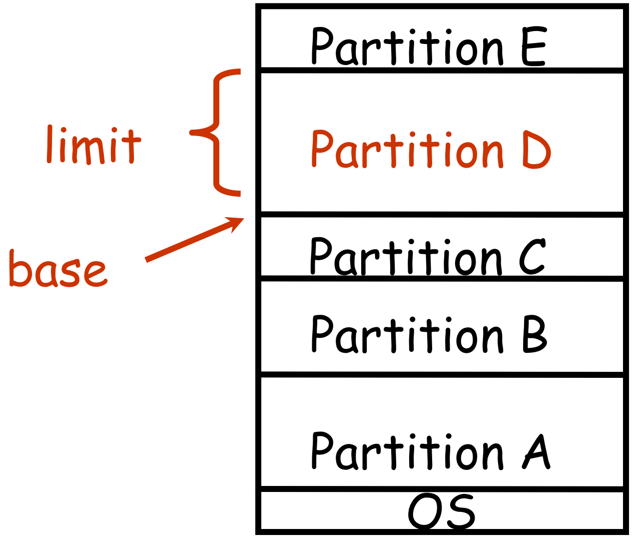
Protection

- Memory protection
  - *Base register gives starting address for process*
  - *Limit register limits the offset accessible from the relocation register*



Multiprogramming

- Multiprogramming: a separate partition per process
- What happens on a context switch?
  - *Store process base and limit register values*
  - *Load new values into base and limit registers*

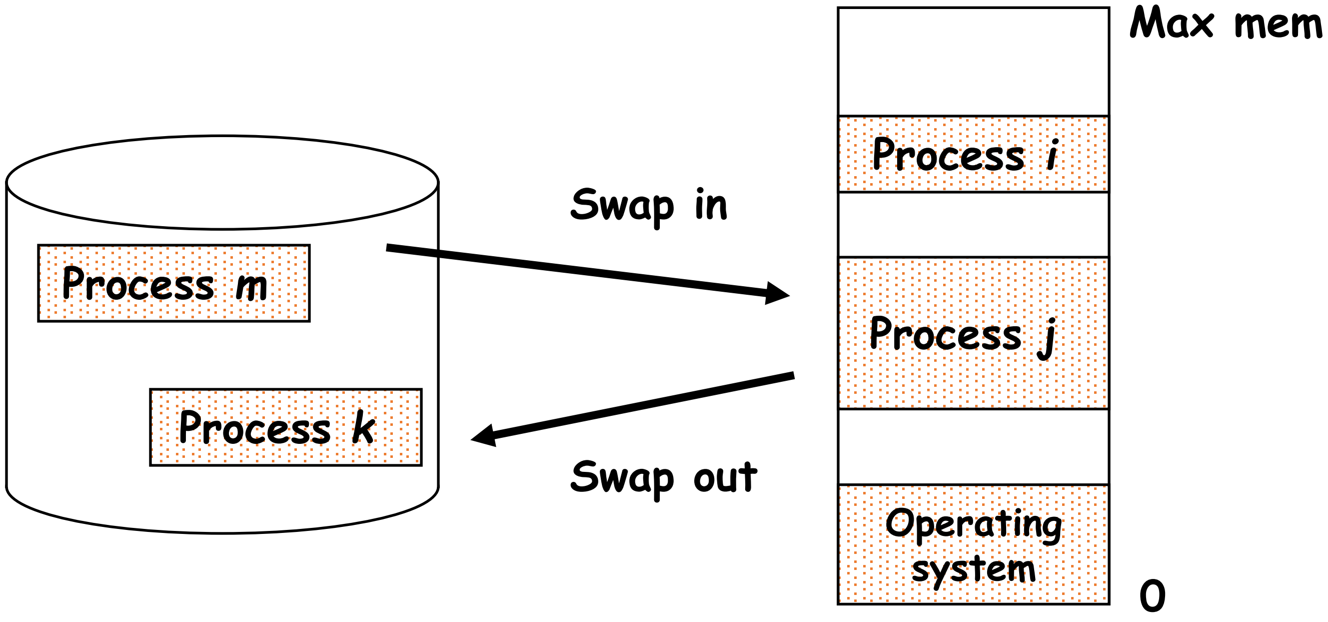


Swapping

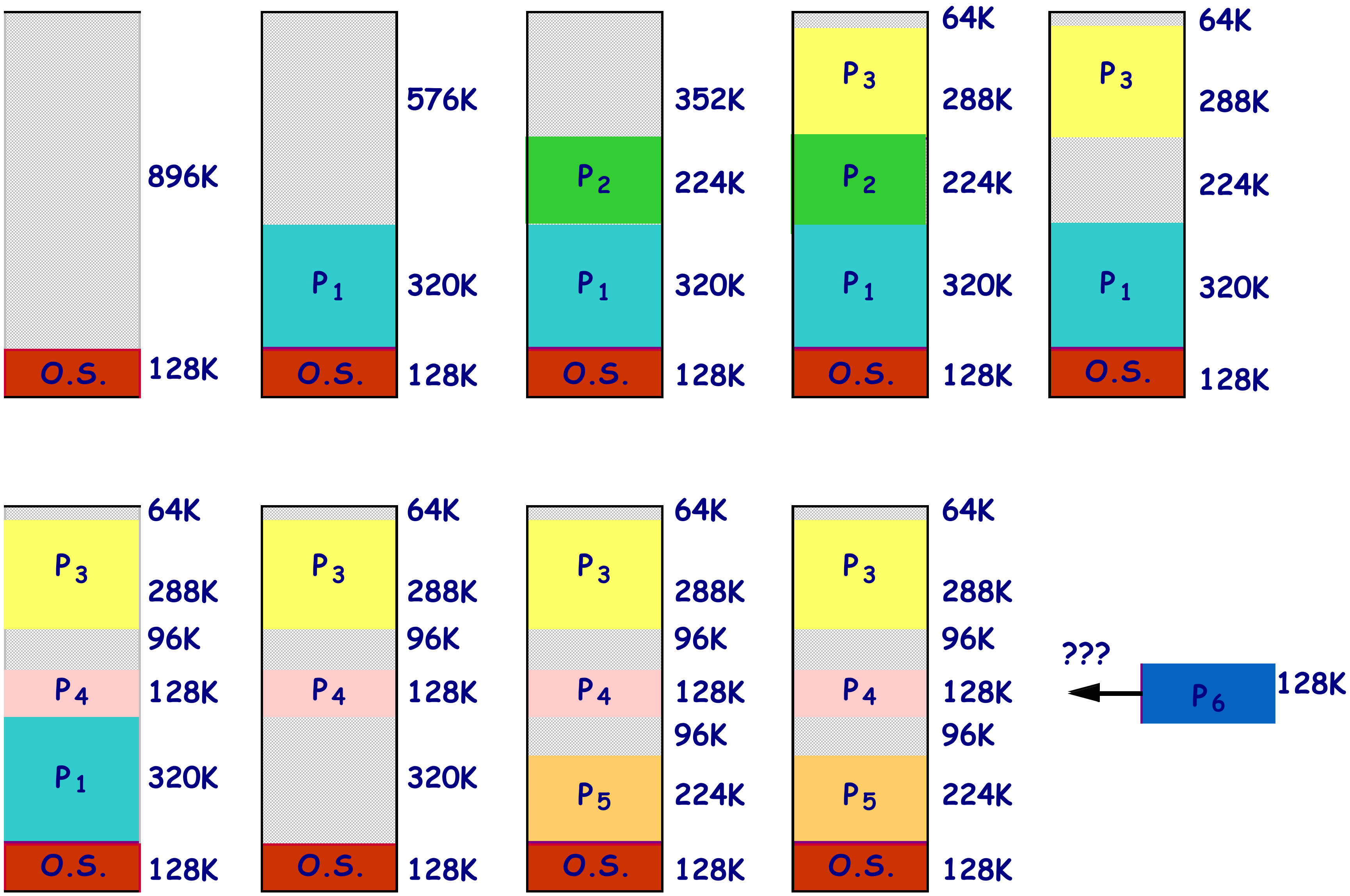
- When a program is running...
  - *The entire program must be in memory*
  - *Each program is put into a single partition*
- When the program is not running...
  - *May remain resident in memory*
  - *May get “swapped” out to disk*
- Over time...
  - *Programs come into memory when they get swapped in*
  - *Programs leave memory when they get swapped out*

Swapping

- Benefits of swapping:
  - *Allows multiple programs to be run concurrently*
  - *… more than will fit in memory at once*



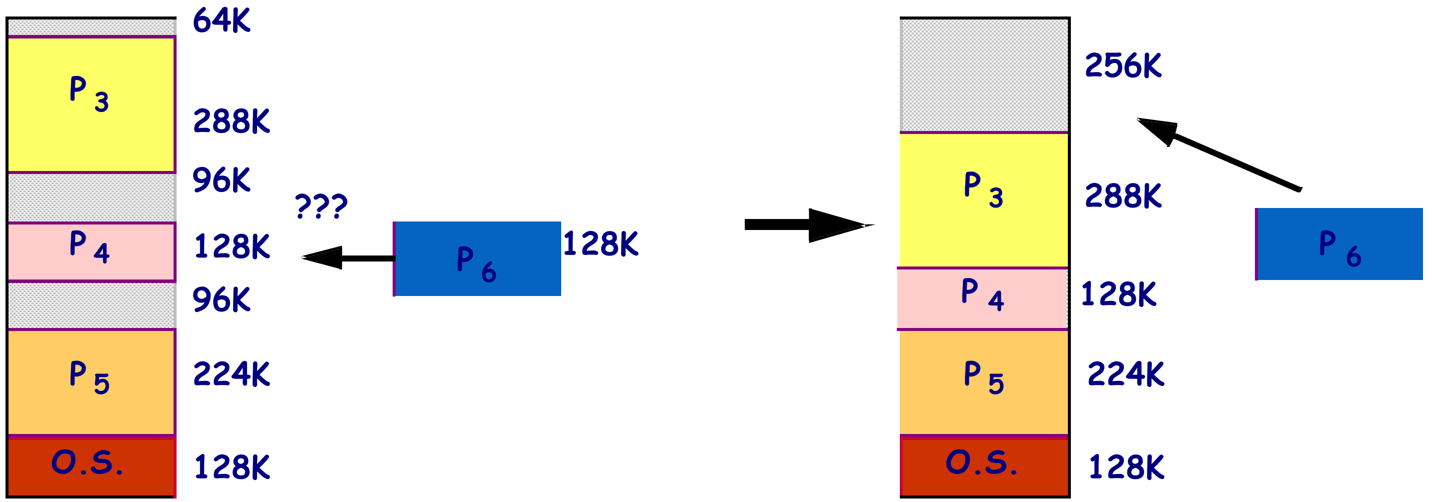
FRAGMENTATION



O.S.
896K
128K
P1
576K
320K
P2
352K
224K
P3
64K
288K
P4
96K
128K
P5
224K
???
P6

Dealing With Fragmentation

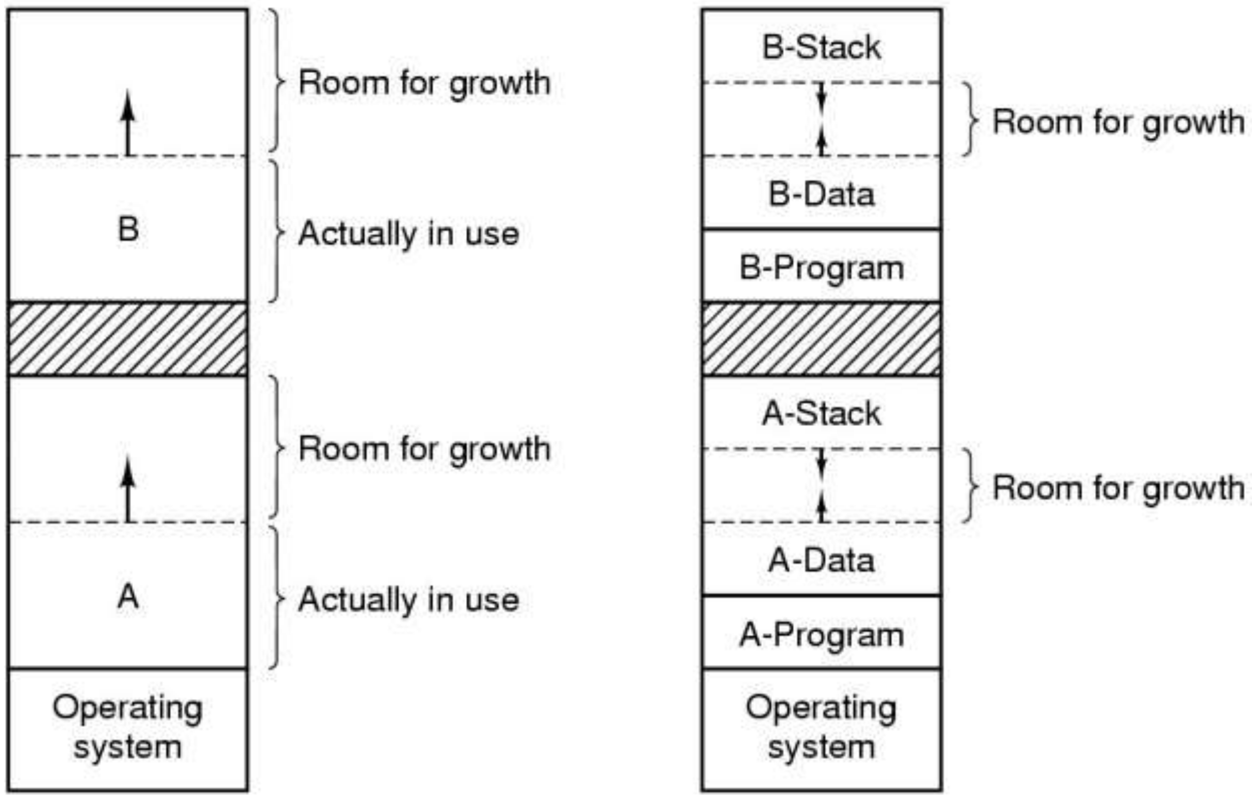
- *Compaction* – from time to time shift processes around to collect all free space into one contiguous block
  - *Memory to memory copying overhead*
  - *Memory to disk to memory for compaction via swapping!*



How Big Should Partitions Be?

- Programs may want to grow during execution
  - *More room for stack, heap allocation, etc*
- Problem:
  - *If the partition is too small, programs must be moved*
  - *Requires copying overhead*
  - *Why not make the partitions a little larger than necessary to accommodate "some" cheap growth?*

Allocating Extra Space Within



(a) (b)

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Management Data Structures

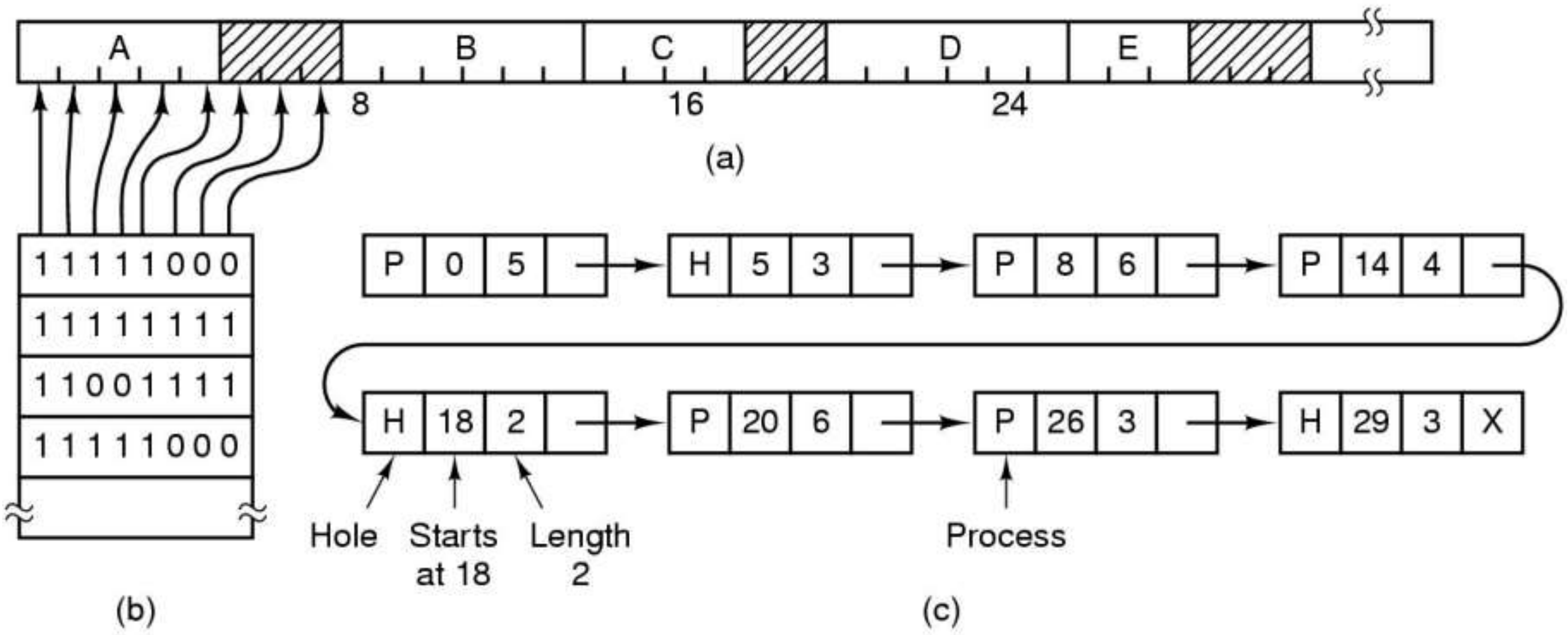
- Each chunk of memory is either
  - *Used by some process or unused (free)*

- Operations
  - *Allocate a chunk of unused memory big enough to hold a new process*
  - *Free a chunk of memory by returning it to the free pool after a process terminates or is swapped out*

Management With Bit Maps

- Problem - how to keep track of used and unused memory?
- **Technique 1** - Bit Maps
  - *A long bit string*
  - *One bit for every chunk of memory*
    - 1 = in use
    - 0 = free
  - *Size of allocation unit influences space required*
    - Example: unit size = 32 bits
      - *overhead for bit map: 1/33 = 3%*
    - Example: unit size = 4Kbytes
      - *overhead for bit map: 1/32,769*

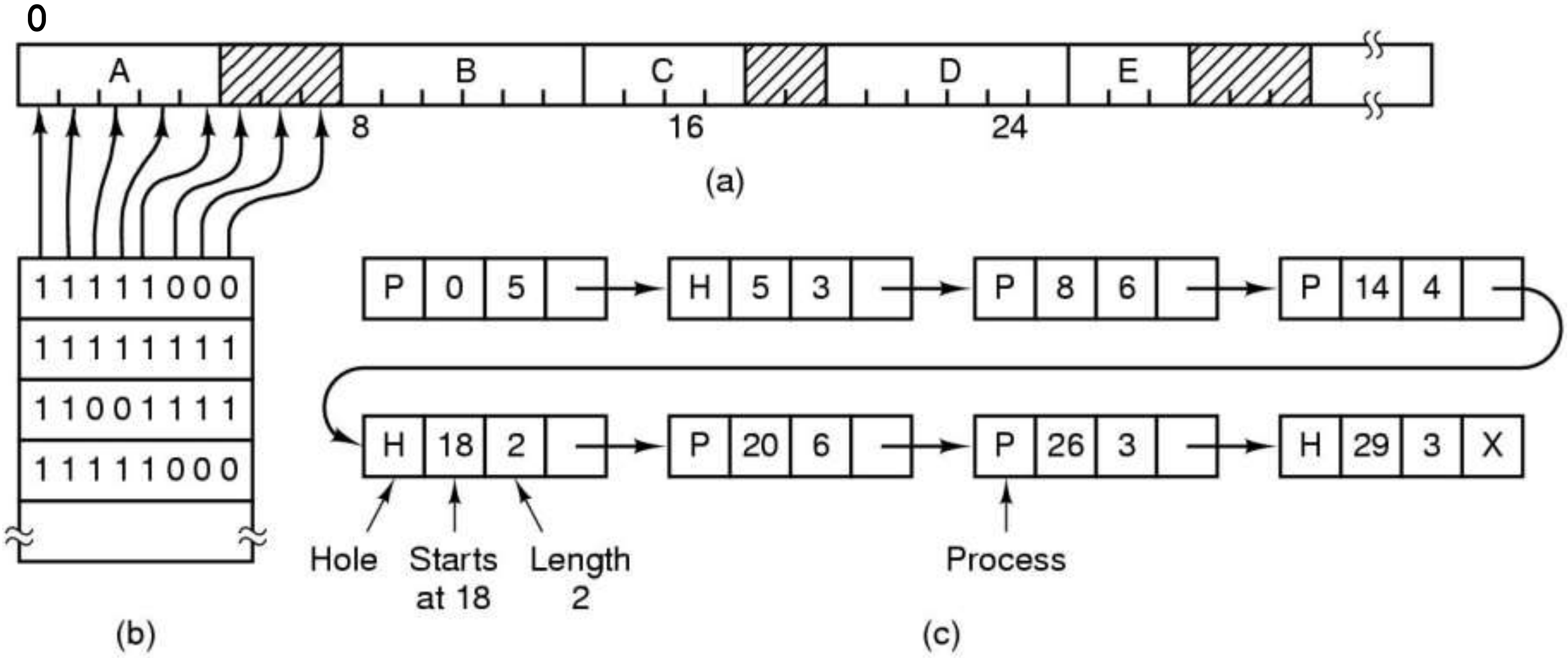
Management With Bit Maps

- 

Management With Linked Lists

- **Technique 2 -** Linked List
- Keep a list of elements
- Each element describes one unit of memory
  - *Free / in-use Bit ("P=process, H=hole")*
  - *Starting address*
  - *Length*
  - *Pointer to next element*

Management With Linked Lists

- 

Management With Linked Lists

Searching the list for space for a new process

- First Fit
- Next Fit
  - *Start from current location in the list*
- Best Fit
  - *Find the smallest hole that will work*
  - *Tends to create lots of really small holes*
- Worst Fit
  - *Find the largest hole*
  - *Remainder will be big*
- Quick Fit
  - *Keep separate lists for common sizes*

Fragmentation Revisited

- Memory is divided into partitions
- Each partition has a different size
- Processes are allocated space and later freed
- After a while memory will be full of small holes!
  - *No free space large enough for a new process even though there is enough free memory in total*
- If we allow free space within a partition we have fragmentation
  - *External fragmentation = unused space between partitions*
  - *Internal fragmentation = unused space within partitions*

Solutions to Fragmentation

- Compaction requires high copying overhead
- Why not allocate memory in non-contiguous equal fixed size units?
  - *No external fragmentation!*
  - *Internal fragmentation < 1 unit per process*
- How big should the units be?
  - *The smaller the better for internal fragmentation*
  - *The larger the better for management overhead*
- The key challenge for this approach

  How can we do secure dynamic address translation?