

CSE 120: Principles of Operating Systems

Lecture 7

Memory Management

October 16, 2003

Prof. Joe Pasquale

Department of Computer Science and Engineering

University of California, San Diego

Before We Begin ...

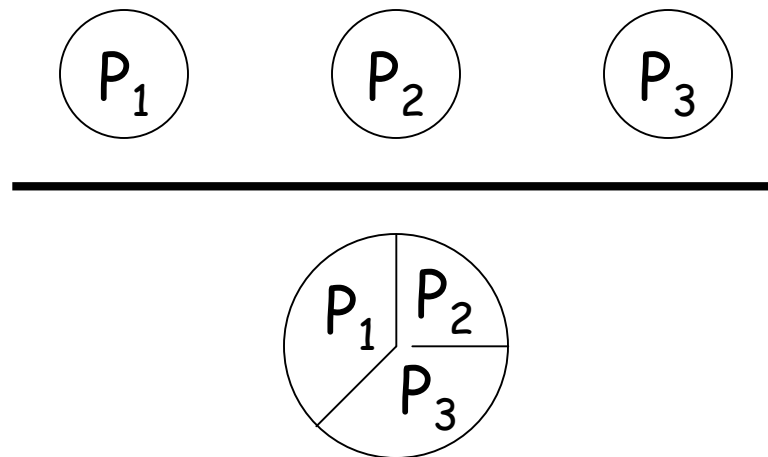
Read Chapter 9 (on Memory Management)

Midterm is on Tuesday of next week

What We Know So Far

Goal: run multiple programs "simultaneously"

- process: abstraction of program in execution
- programmer need not deal with multiplexing CPU



Process encapsulates more than CPU time, e.g., memory

Sharing the Physical Memory

When process is given CPU, must also be in memory

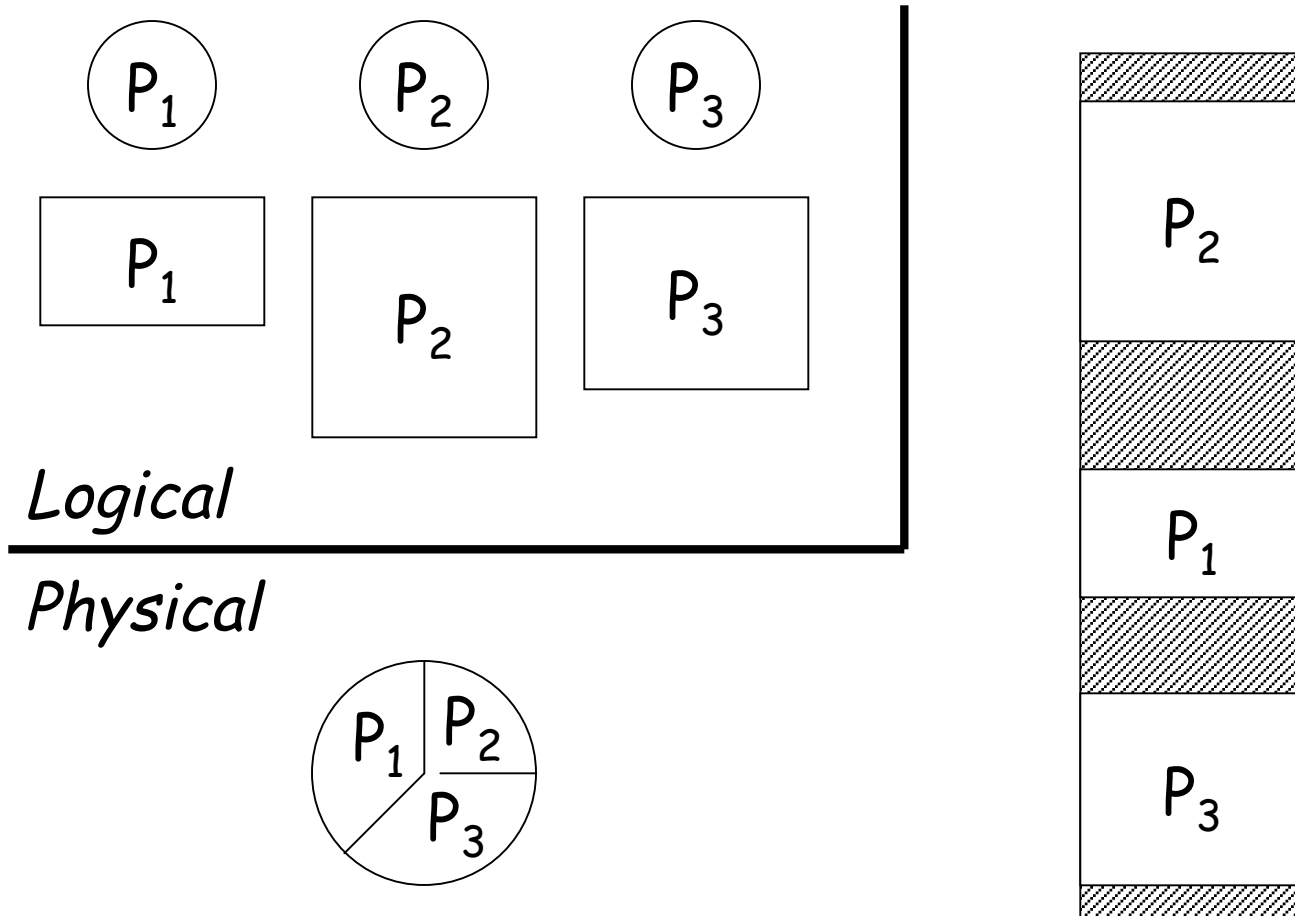
Problem

- context-switching time (CST): $10\ \mu\text{sec}$
- loading from disk: $10\ \text{MB/s}$
- to load 1 MB process: $100\ \text{msec} = 10,000 \times \text{CST!}$

Solution: keep multiple processes in memory

- Keep as many processes in memory as possible
- Context-switch: select only processes in me

Logical vs. Physical Views



Memory Management

How to manage space of (physical) memory

Allocate by partitioning into chunks

- one or more chunks per process

Free by deallocating chunks and coalescing if necessary

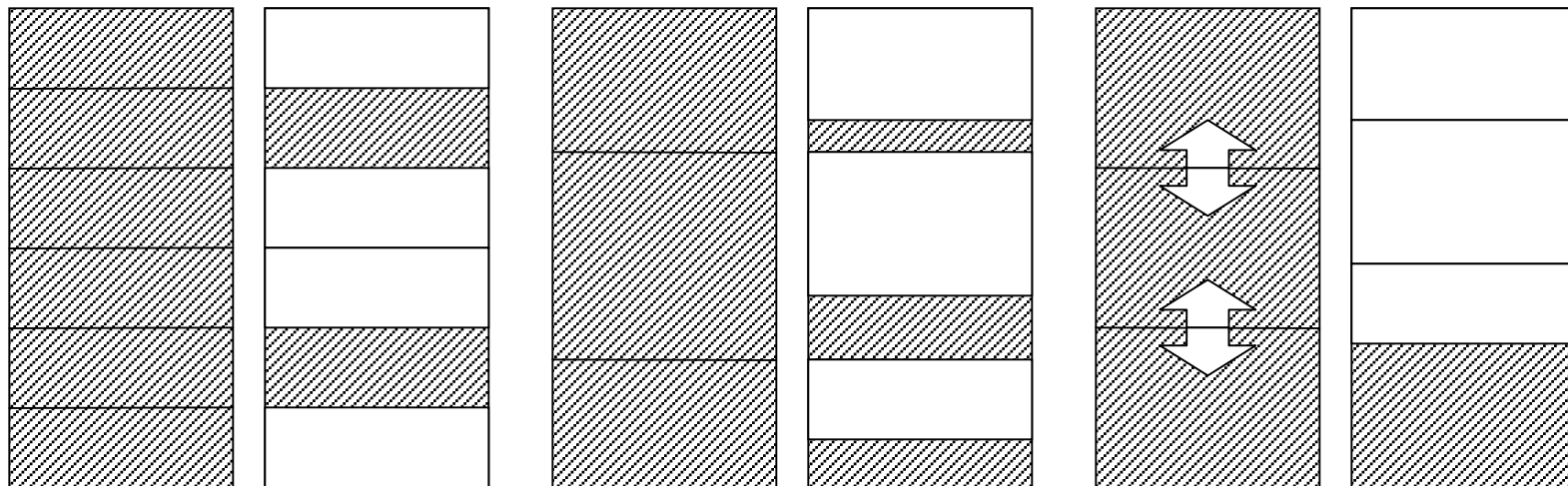
Rearrangement

- compaction: move chunks, leaving one free chunk
- spreading chunks, so each has more space to grow

Memory Partitioning

Fixed/static vs. variable/dynamic, equal vs. non-equal

Fragmentation: internal vs. external



Fixed
Equal

Fixed
Non-equal

Variable
Non-equal

Placement Algorithms

First-fit

Next-fit

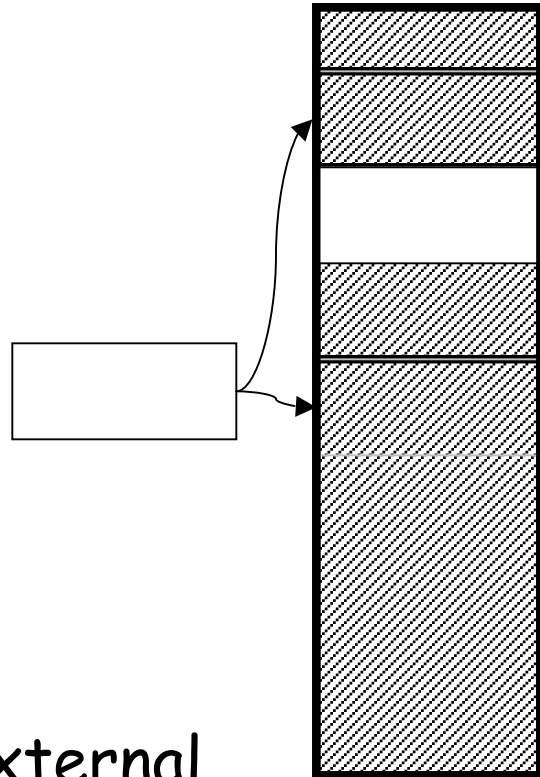
Best-fit

Worst-fit

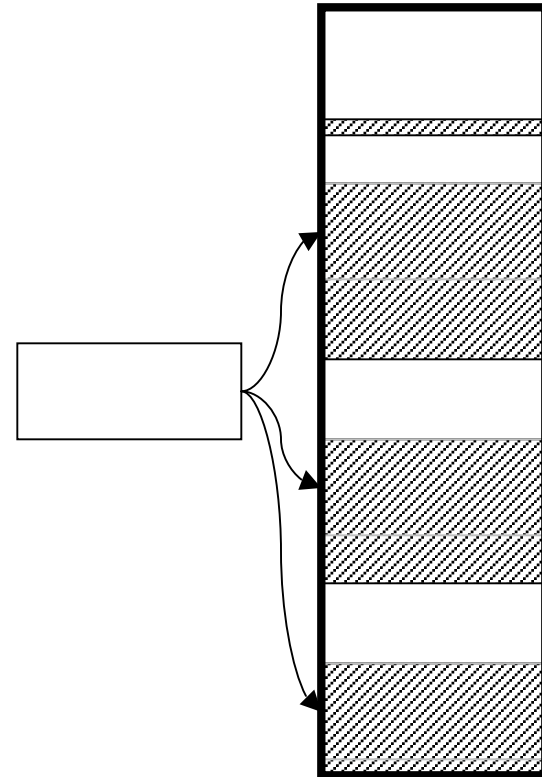
Compaction

Internal / external

fragmentation



Fixed
Non-equal



Variable
Non-equal

The Buddy System

Dynamically partition in powers-of-2 size chunks

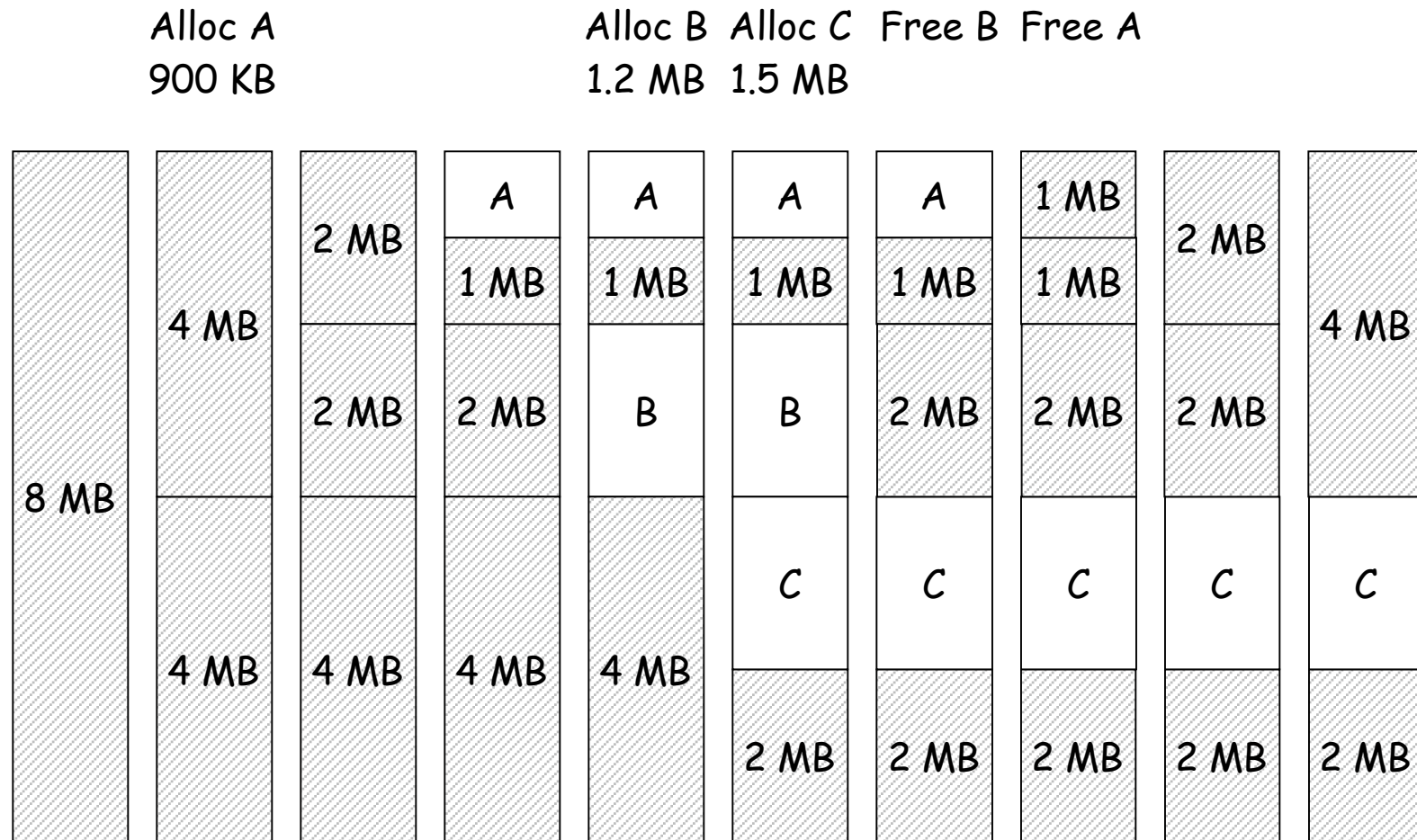
Allocation

- given request for size r , find smallest chunk
- while ($r < \text{sizeof}(\text{chunk})/2$)
 - divide chunk into 2 buddies (each of $1/2$ size), select one

Deallocation

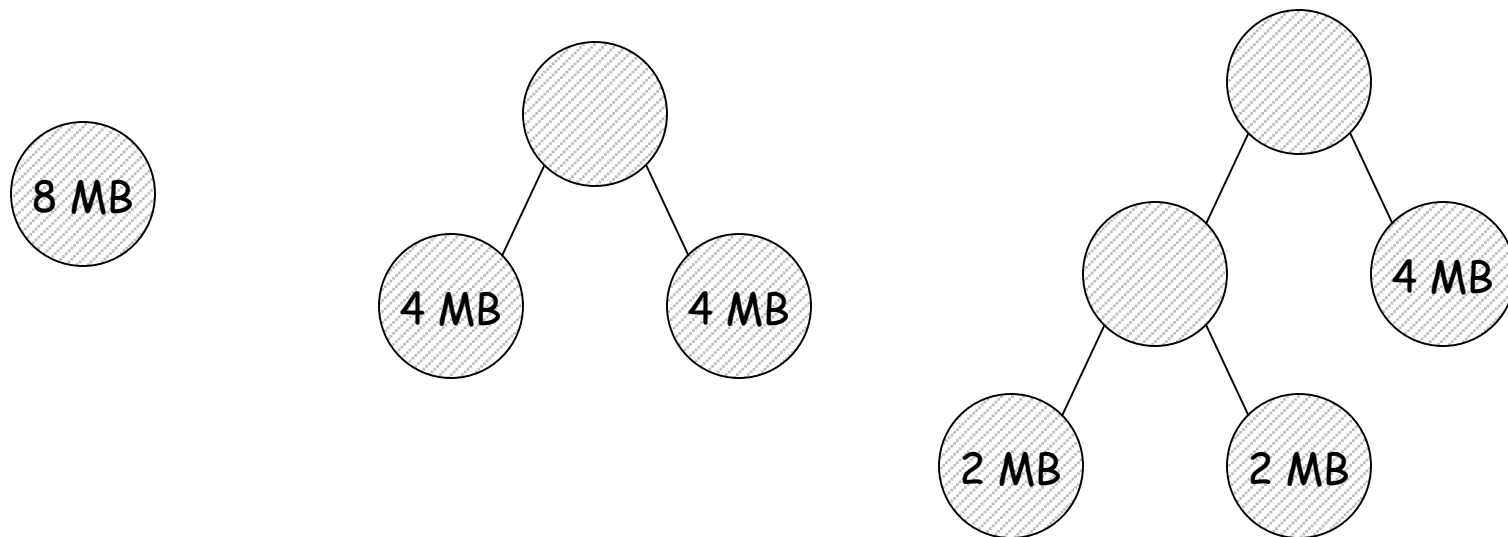
- free the chunk
- while (buddy is also free)
 - Coalesce

Example of Buddy System



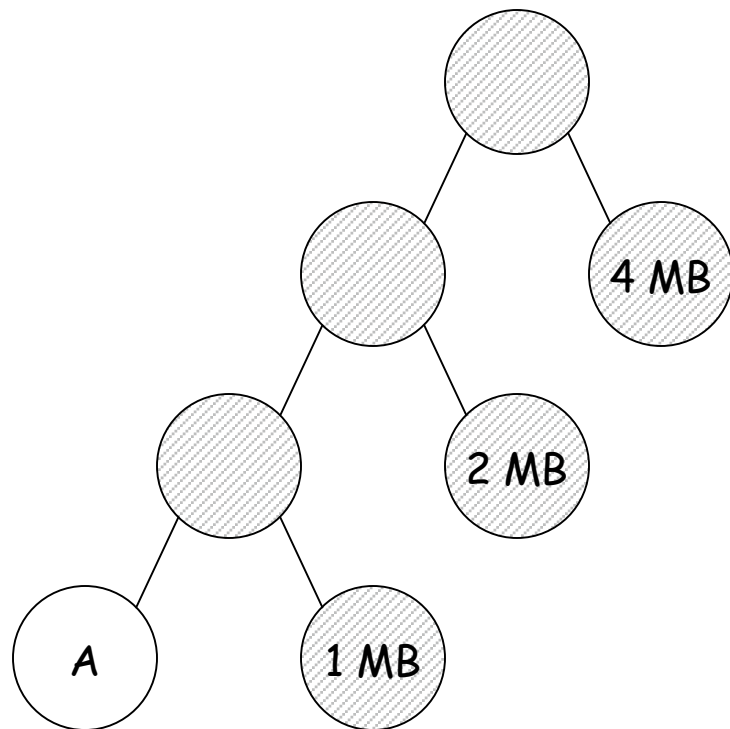
Data Structure for Buddy System

Alloc A: 900 KB

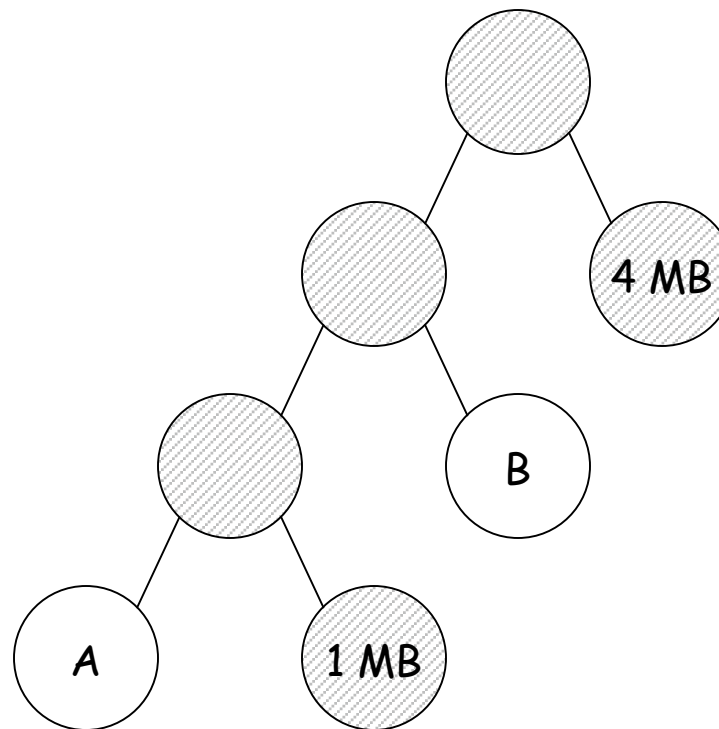


Data Structure for Buddy System

Alloc A: 900 KB

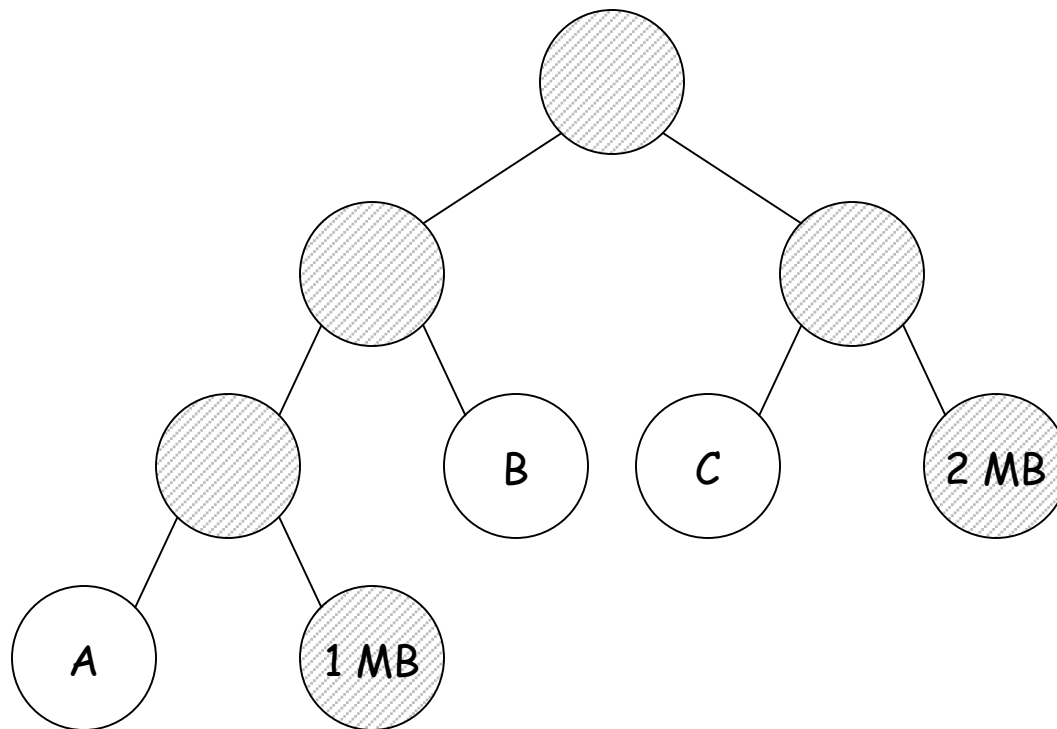


Alloc B: 1.2 MB



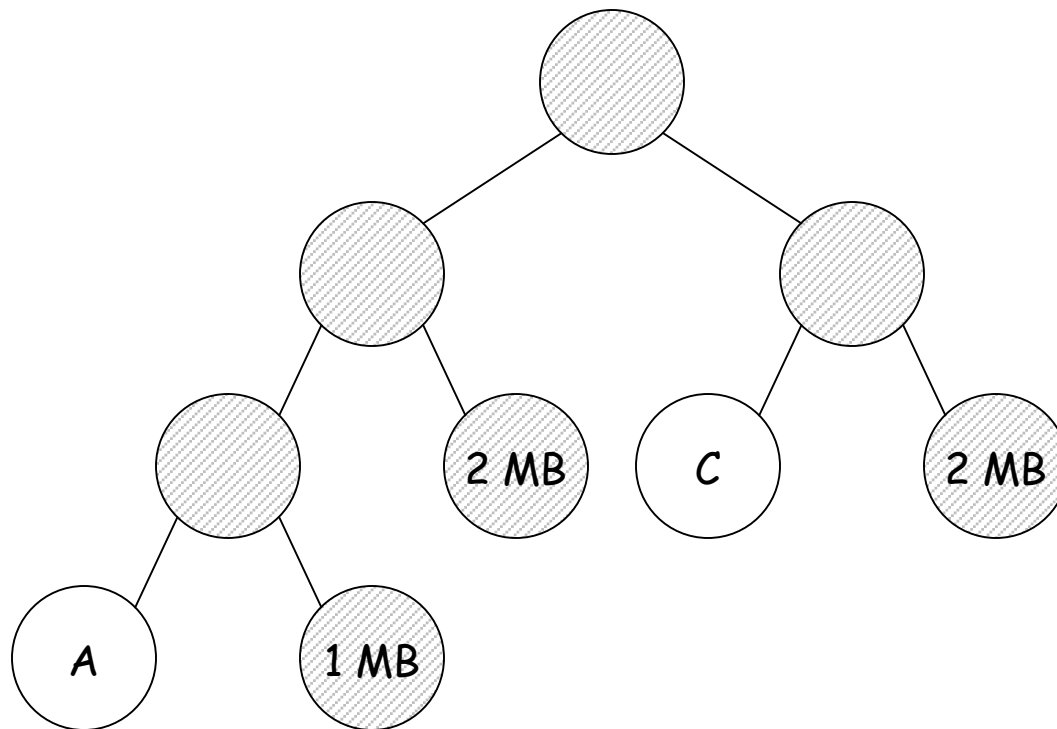
Data Structure for Buddy System

Alloc C: 1.5 MB



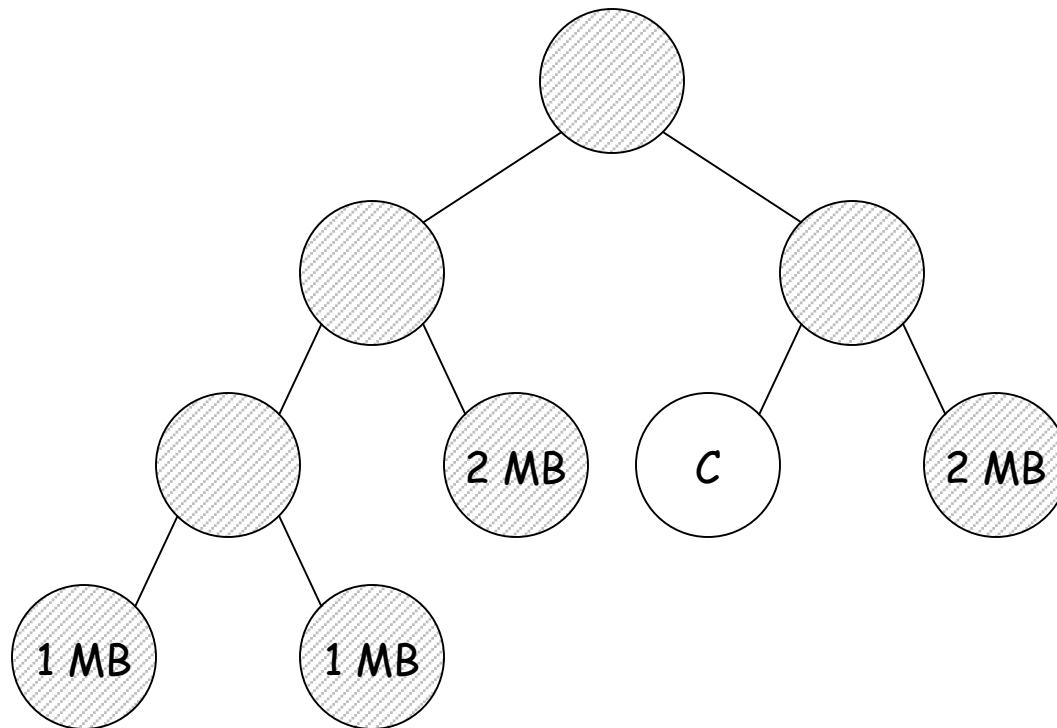
Data Structure for Buddy System

Free B



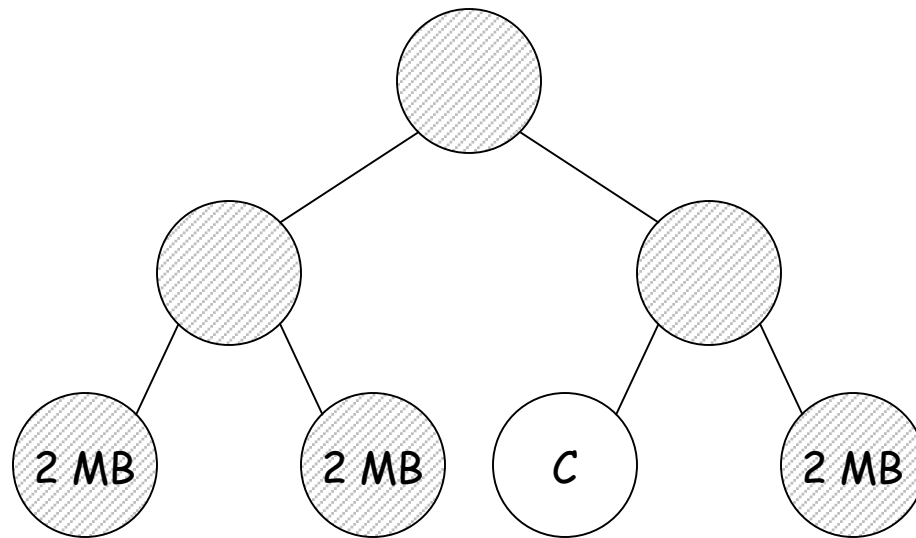
Data Structure for Buddy System

Free A

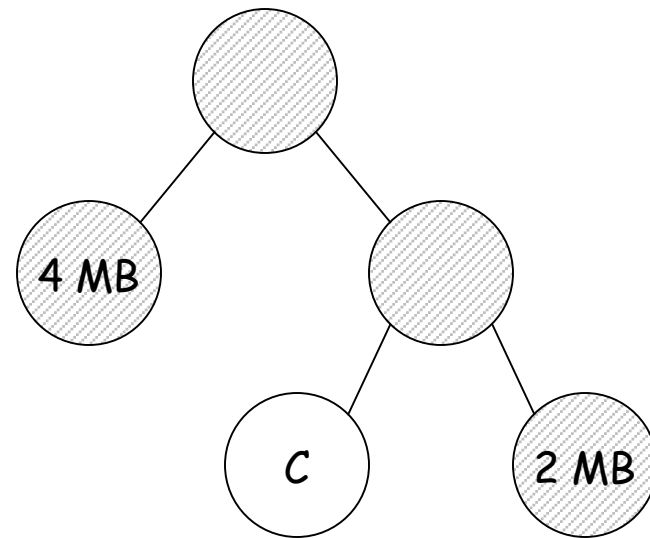


Data Structure for Buddy System

Coalesce



Coalesce



Problems with Sharing Memory

The Addressing Problem

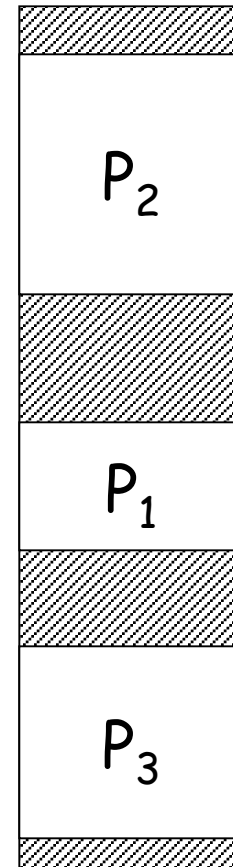
- compiler generates memory refs
- unknown where process will be in

The Protection Problem

- modifying another process's memory

The Space Problem

- the more processes there are, the less memory each have individually



Address Spaces

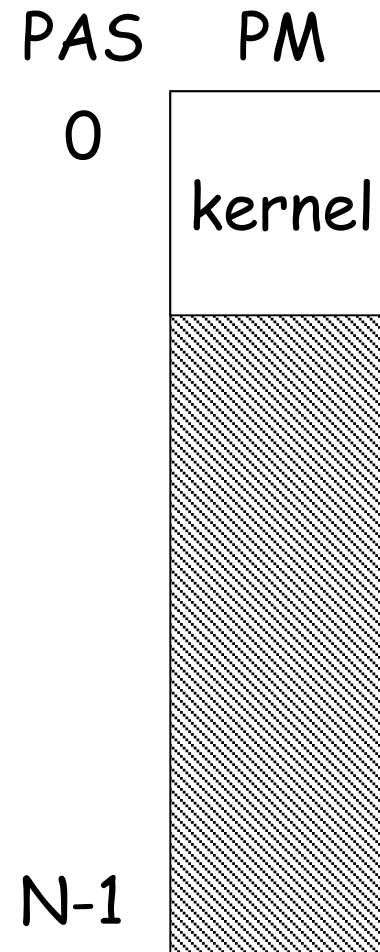
Address space

- set of addresses for memory

Usually linear: 0 to N-1 (size N)

Physical Address Space

- 0 to N-1, N = size
- kernel occupies lowest addresses (typically)



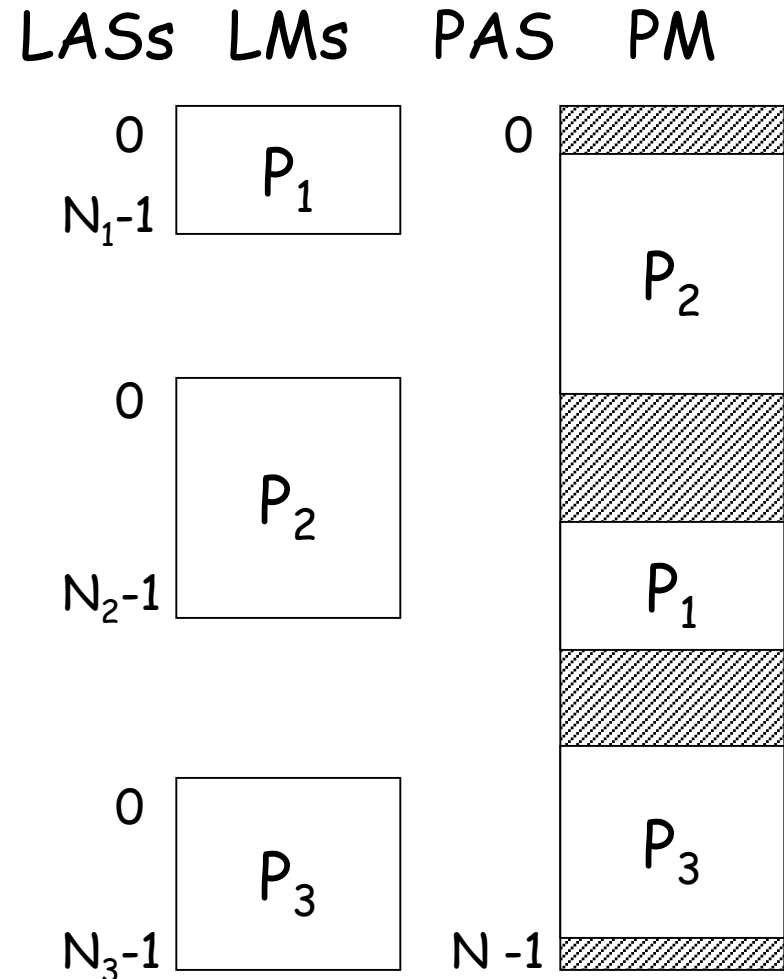
Logical vs. Physical Addressing

Logical addresses

- assumes separate memory starting at 0
- compiler generated
- independent of location in physical memory

Converting logical to physical

- S/W: at load time
- H/W: at access time



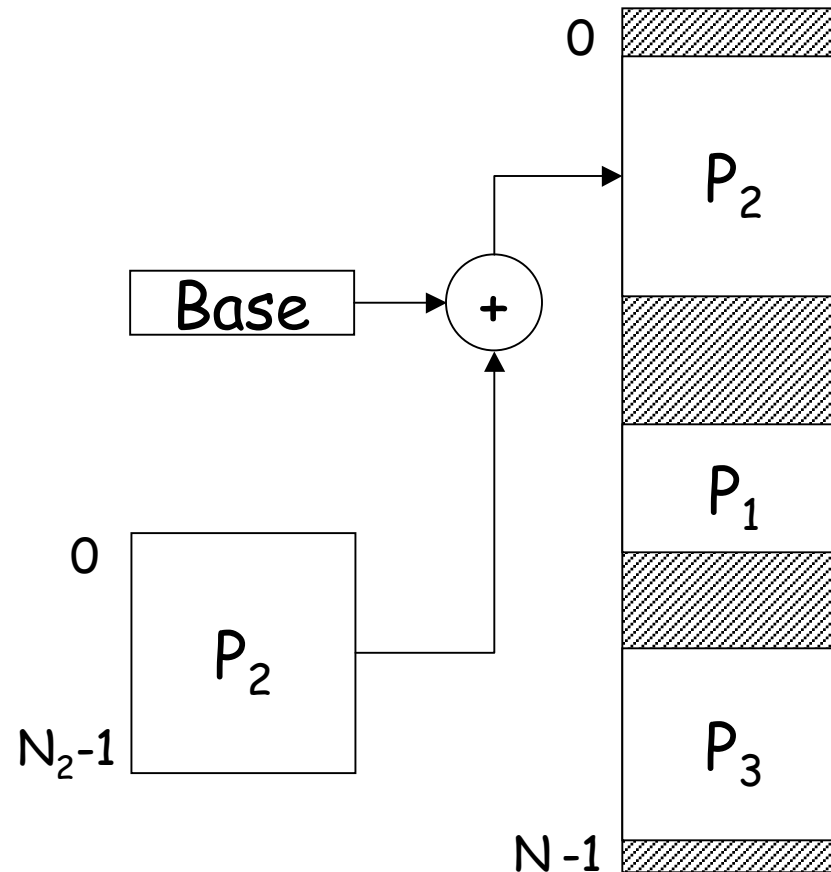
Hardware for Logical Addressing

Base register filled
with start address

To translate logical
address, add base

Achieves relocation

To move process:
change base



Protection

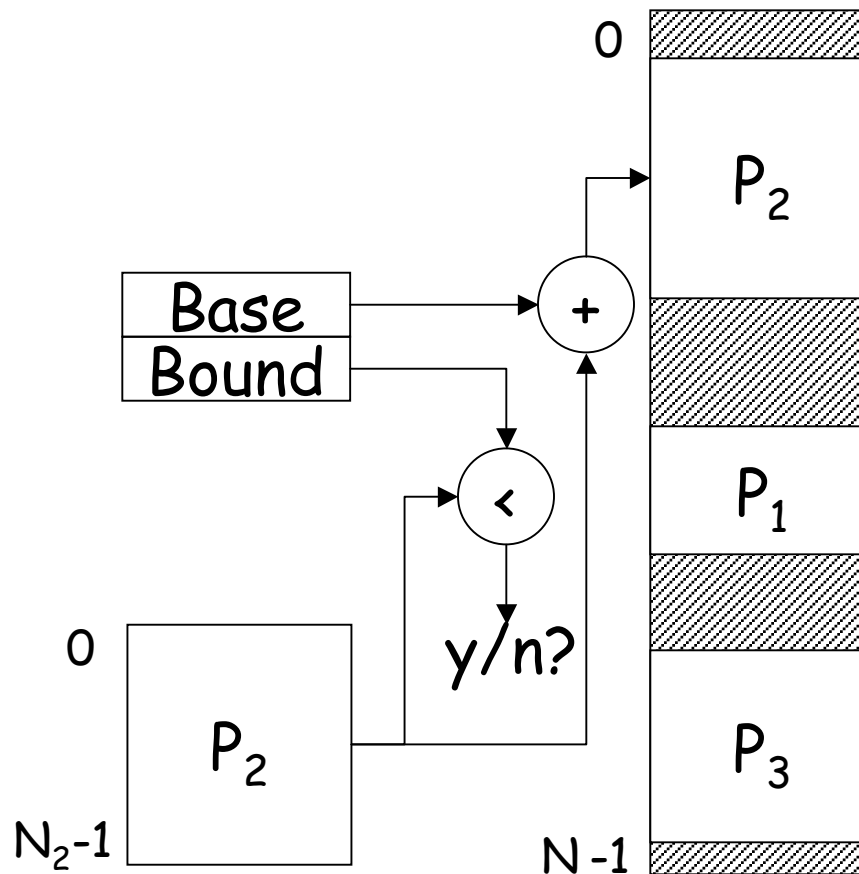
Bound register works

with base reg

Check if address less
than bound

If so, add to base

If not, invalid
address, trap



Memory Registers Part of Context

On Every Context Switch

- Load base/bound registers for selected process
- Only operating system does loading
- Kernel must be protected from all processes

Benefit

- Allows each process to be separately located
- Protects each process from all others

Loading

To create a process, must load it into memory

What to load, the load module, is based on program

- code
- data (initialized and uninitialized)
- stack (keeps track of pending calls, starts empty)

Absolute loading: load to a fixed location in memory

Relocatable loading: load to a variable location

- dynamic run-time loading: allow location to change

Linking

Take object modules, create load module

Linkage editor: resolves inter-object references

- Ex., modules A and B
- A: call $f(x)$, where $f(x)$ code is in B

Dynamic linker: defer linkages until

- load-time: resolve when load module is loaded
- run-time: resolve when referenced
 - Ex., wait until $f(x)$ called to resolve address of f