Scheduling

The datapath in project 1 can only run one assembly program at one time

- Most processors need to be able to multitask and perform serveral operations concurently
- CPUs are often idling during I/O operations, so how do we optimize this?

How to save state of a process?

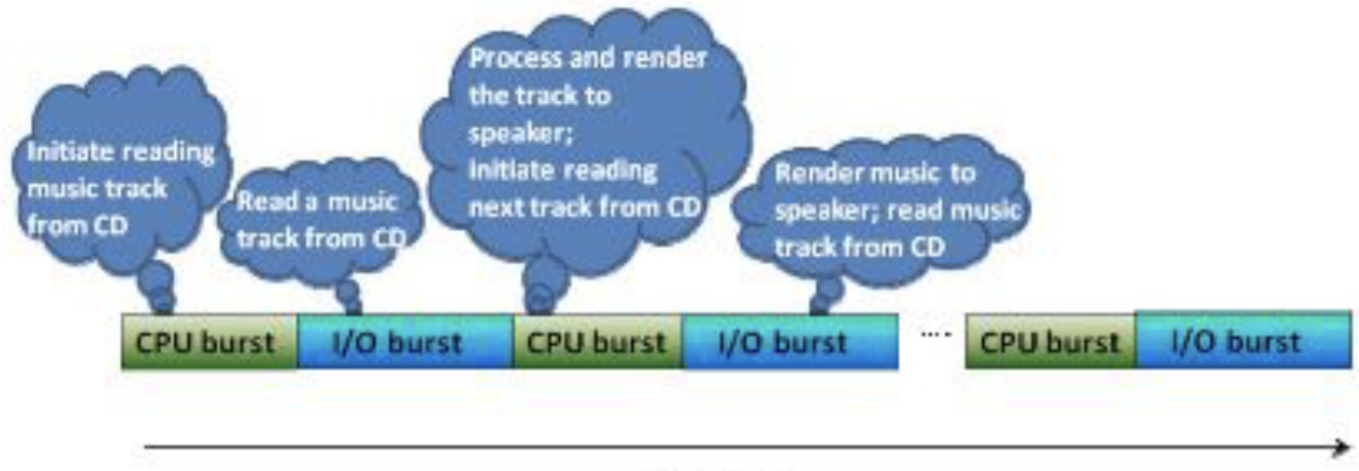
```
enum  state_type {new, ready, running, waiting, halted};

typedef struct control_block_type {
   enum state_type state;           /* current state */
   address PC;                      /* where to resume */
   int reg_file[NUMREGS];           /* contents of GPRs */
   struct control_block *next_pcb; /* list ptr */
   int priority;                    /* extrinsic property */
   address address_space;           /* where in memory */
          ….
          ….
} control_block;
```

Figure 6.8: Process Control Block (PCB)

Siri's New Beatbox

- We need a queue to hold the order in which processes should execute
- We also need bursts, which will hold the process being executed at any given time stamp

CPU Burst = stretch of time that a process runs without making an I/O call
I/O Burst = stretch of time that a process needs to complete an I/O operation



Ready Queue = list of PCBs from different processes that are ready to run on the CPU
I/O Queue = list of PCBs that are waiting for some I/O operation to be initiated

Important Concepts

A program is ***static***, while a process is a program ***currently in execution***

Scheduler = a software that runs some algorithm to determine which processes to run

- Long term = balances the job mix in memory, rare in modern OSes
- Medium-term = monitors the dynamic memory usage to determine the number of processes in memory. mainly used to control thrashing
- **Short-term** = responsible for selecting the process in memory to run

Other terms

- **Loader** = software that loads the user program from disk to memory
- **Dispatcher** = populate the CPU registers with the state of the process selected for running by the short-term scheduler
- **Thrashing** = situation where memory requirements of the processes combined exceed the system capability
  - Takes places when the process spends too much time swapping and reduces throughput

Types of Schedulers

1. Non-preemptive
2. Preemptive

Non-preemptive = processes either executes to completion or gives up the CPU resource on its own

- Examples: FCFS, SJF, Priority

Preemptive = scheduler yanks the processor away from the current process to give it to another process

- Examples: Preemptive FCFS, SRTF, Round Robin

Steps

1. Notify - get the processor's attention
2. Save - state of the running process
3. Select - new process to run
4. Dispatch - newly selected process to run on the processor (done by dispatcher)

Non-Preemptive Scheduling Algorithms

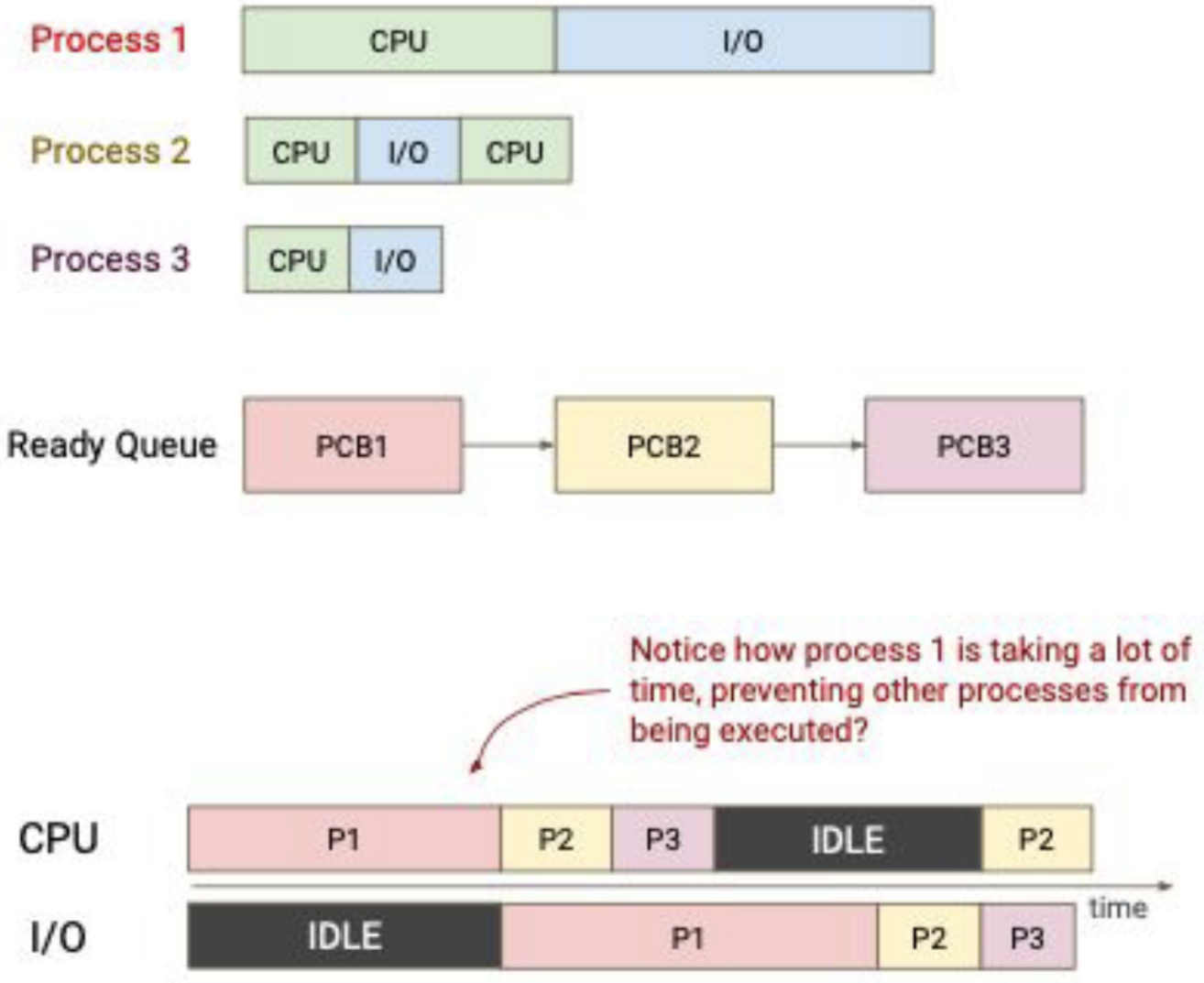
First Come First Serve

Sorts ready queue by arrival time and always execute the element closer to the front of the queue

- Arrival time = when you launch

We have a problem: convoy effect

- One process monopolizes resources and keeps other processes from being executed in a timely manner

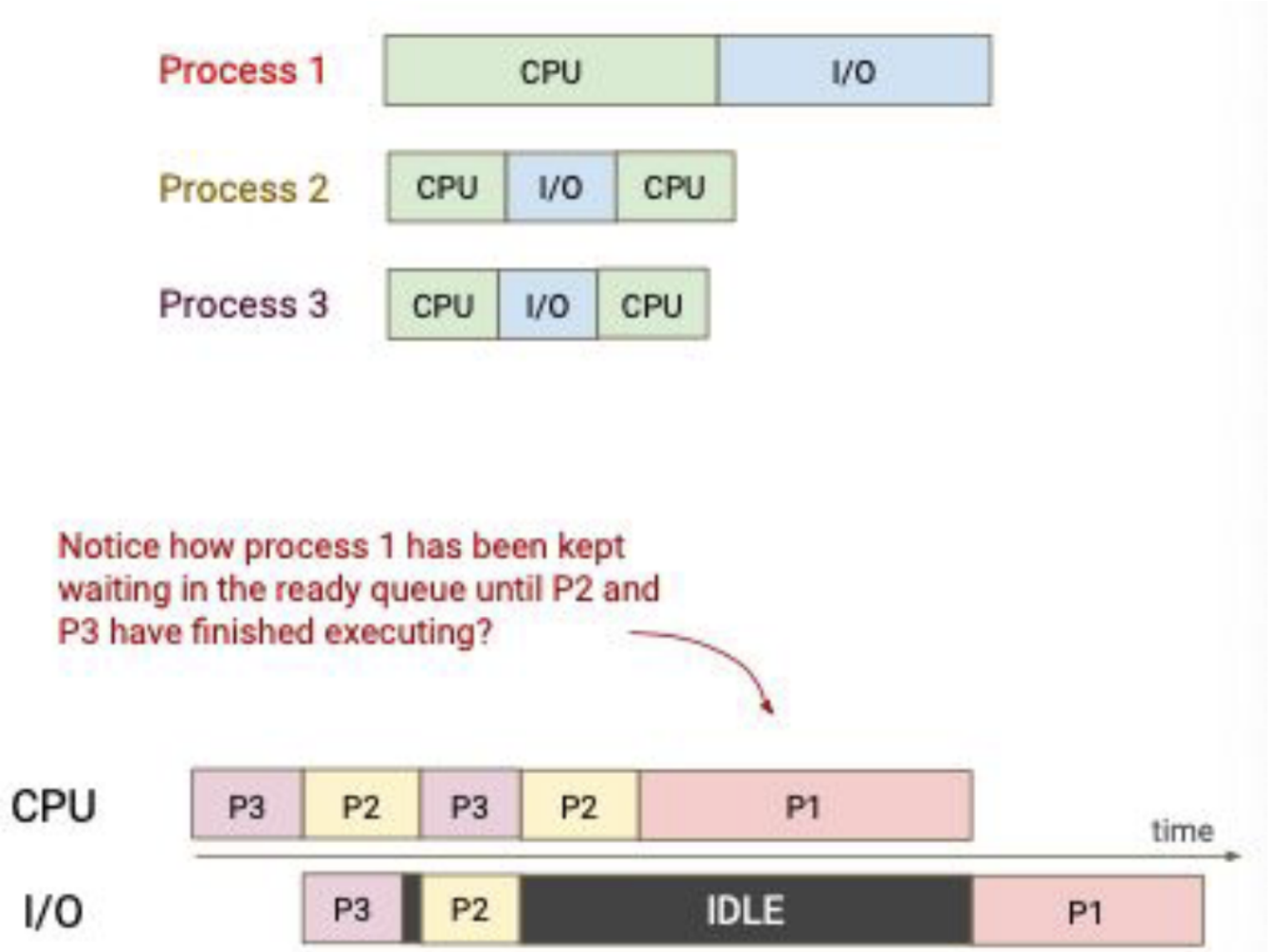


Shortest Job First (SJF)

To prevent the convoy effect, we can also select and execute programs with the shortest CPU bursts first

Problem: starvation

- What happens to longer (and potentially more important) processes when we keep queueing shorter processes first?

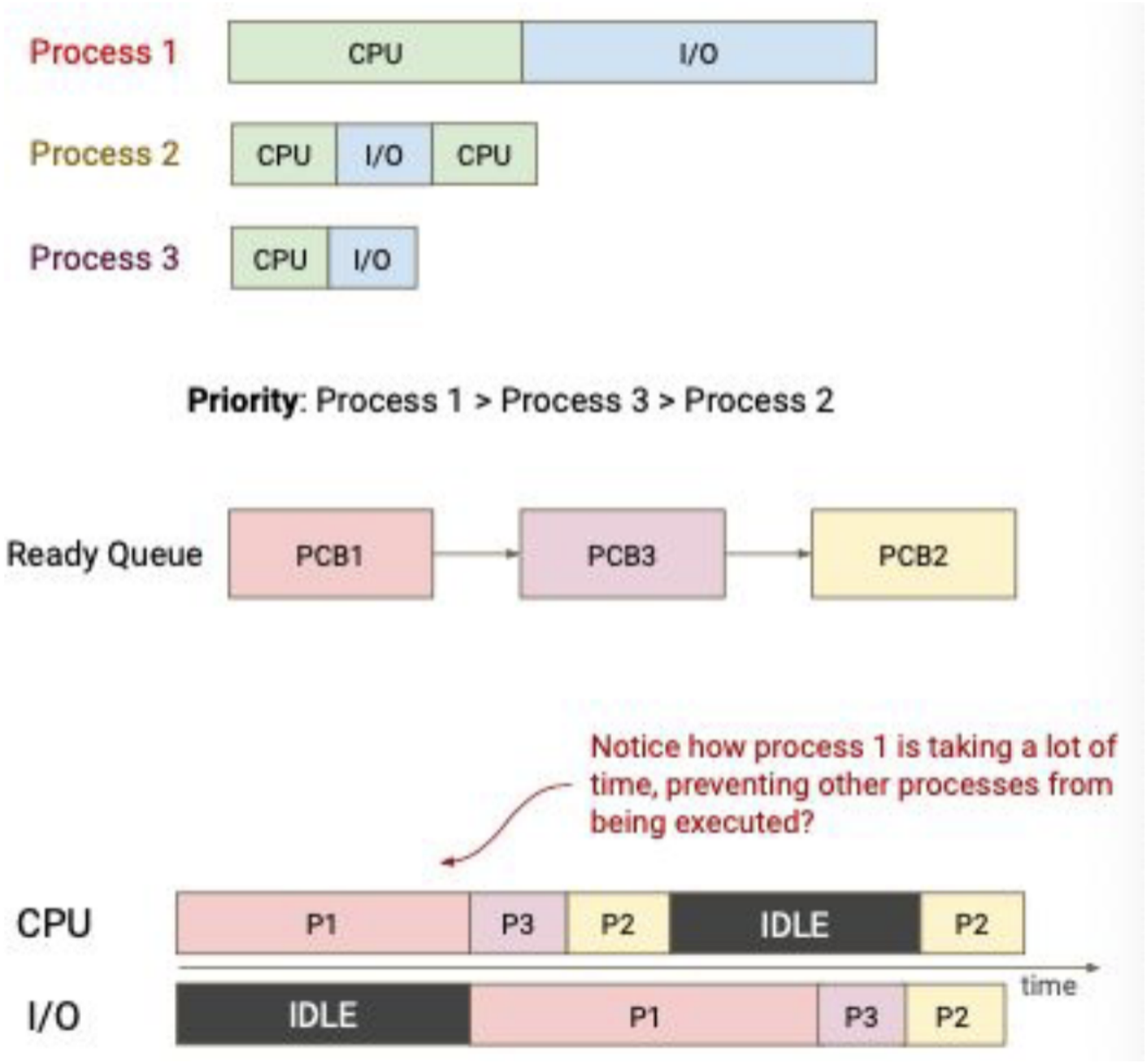


Priority

To favor processes that are more important (rather than shorter or longer processes), we can sort the ready queue by priority

- Extrinsic property that represents of the importance relative to other processes
- Works almost exactly like FCFS after the ready queue has been sorted

The problem is we can still run into the convoy and starvation effect



Non-Preemptive Example

Suppose we have the following proceses:

PCB #1:
- state = ready to run
- PC = 0x1000
- priority = 2

PCB #2:
- state = ready to run
- PC = 0x1000
- priority = 1

PCB #3:
- state = ready to run
- PC = 0x1000
- priority = 3

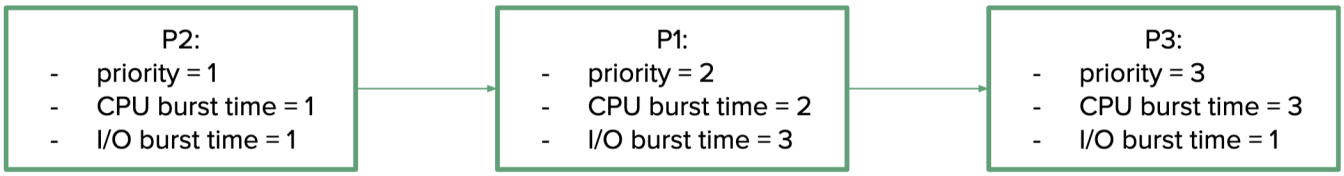
PCB #1 Burst time:
CPU burst time = 2 units
I/O burst time = 3 units

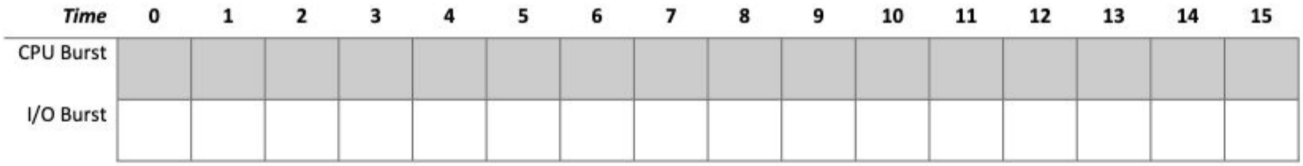
PCB #2 Burst time:
CPU burst time = 1 units
I/O burst time = 1 units

PCB #3 Burst time:
CPU burst time = 3 units
I/O burst time = 1 units

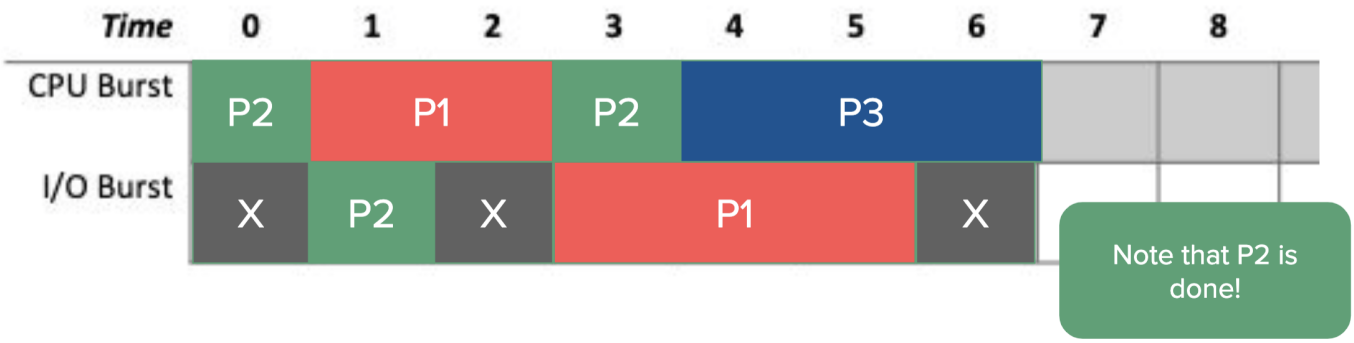
Which one will have the highest priority?

- PCB #2
- Highest priority = lowest priority number lol





Ready Queue: { ?, ?, ? }



What observations can we make from this example?

- We didn't fully finish the execution of P1 and P3 – the idea of the example was to give you an idea as to what we prioritize depending on the kind of algorithm we're working with
- What happened to P3?
  - It just starved lmao
  - Just because P3 is a low priority process doesn't mean it's not important :/

Preemptive Scheduling Algorithms

Shortest Remaining Time First (SRTF)

- Favors shorter processes
- Preempts the currently running process if anothoer process with a shorter remaining time enters a ready queue

- Still starves processes with longer execution times
- How do we calculate the remaining time?

Process	Arrival Time	Execution Time
Process 1	T + 0ms	3ms
Process 2	T + 1ms	1ms
Process 3	T + 2ms	3ms

	Remaining time for		
Current time	Process 1	Process 2	Process 3
T + 0	**3**	*N/A*	*N/A*
T + 1	2 (preempted)	**1**	*N/A*
T + 2	**2**	0	3
T + 3	**1**	0	3
T + 4	0	0	**3**
T + 5	0	0	**2**
T + 6	0	0	**1**
T + 7	0	0	0

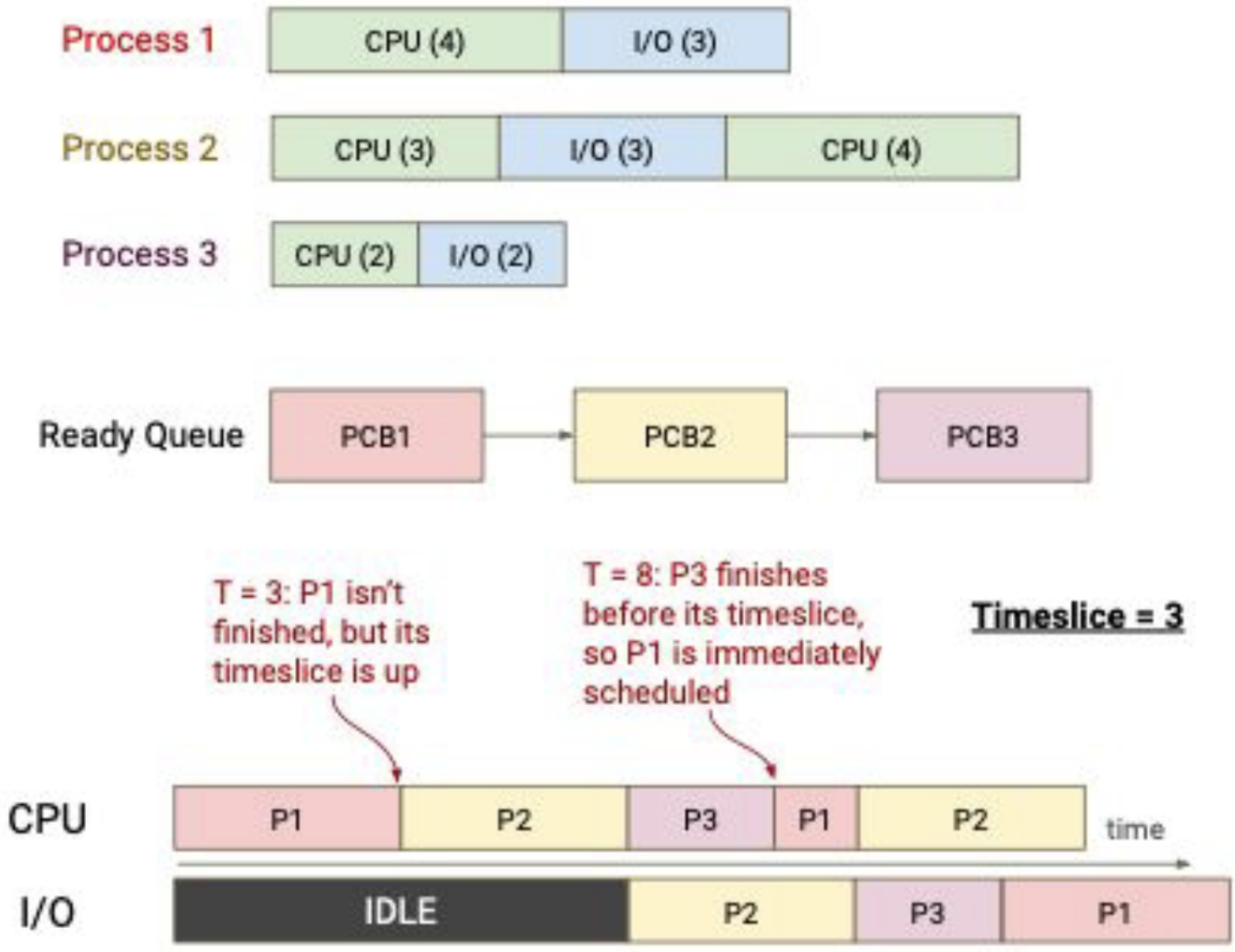
Round Robin

- Each process gets a certain amount of time (a time slice or time quantum) to run on the process
- The process executing on the processor changes if
  - the currently executing process finishes executing

- the currently executing process is not finished executing but its time slice has completed => gets kicked off CPU :((

Timeslices

- Too short = thrashing (context switching too much)
- Too long = lose benefit of preemption



Preemptive Example

In what order do these processes execute?

PCB #1:
- state = ready to run
- PC = 0x1000
- priority = 2

PCB #2:
- state = ready to run
- PC = 0x1000
- priority = 1

PCB #3:
- state = ready to run
- PC = 0x1000
- priority = 3

PCB #1 Burst time:
CPU burst time = 2 units
I/O burst time = 3 units

PCB #2 Burst time:
CPU burst time = 1 units
I/O burst time = 1 units

PCB #3 Burst time:
CPU burst time = 3 units
I/O burst time = 1 units

timeslice = 2

Priority doesn't matter, in round robin we order them by arrival time

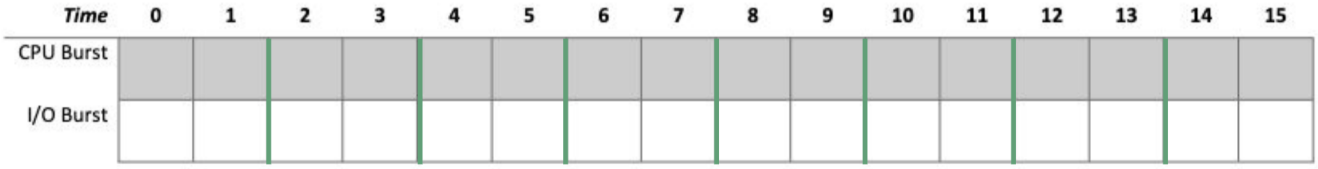
P1:
- CPU burst time = 2
- I/O burst time = 3

→

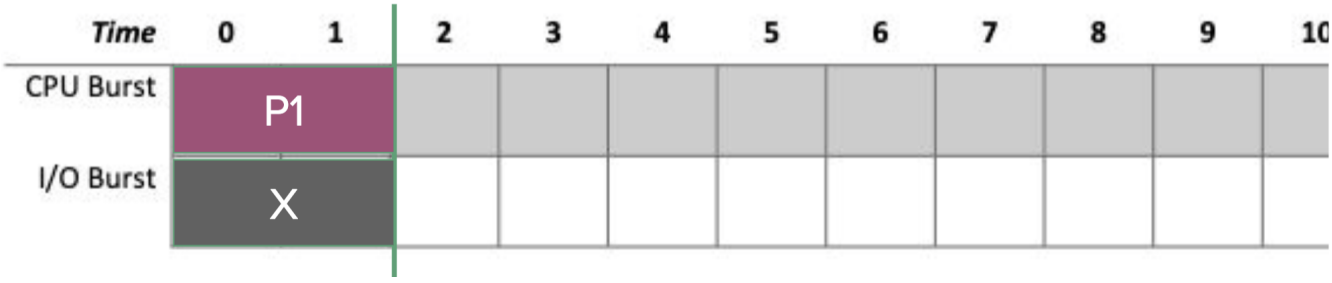
P2:
- CPU burst time = 1
- I/O burst time = 1

→

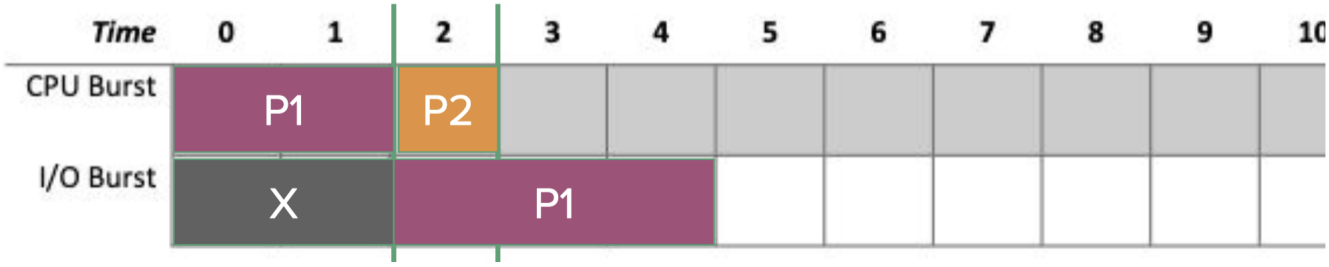
P3:
- CPU burst time = 3
- I/O burst time = 1



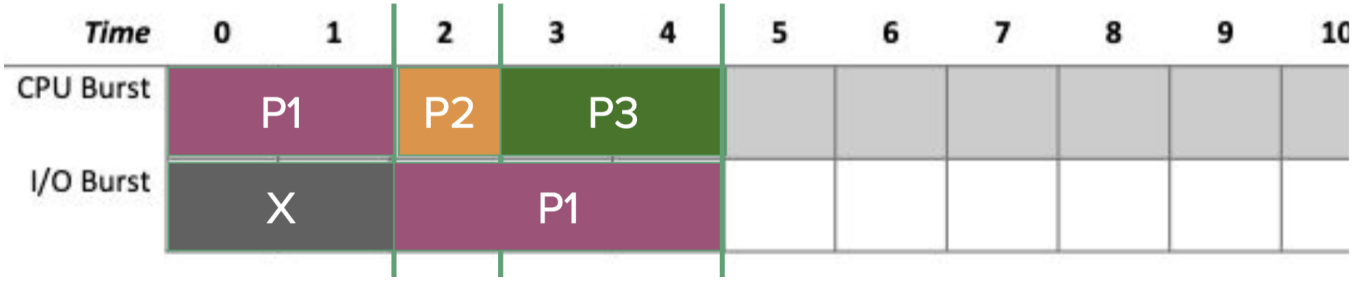
Ready Queue: { (P1, 2), (P2, 1), (P3, 3) }



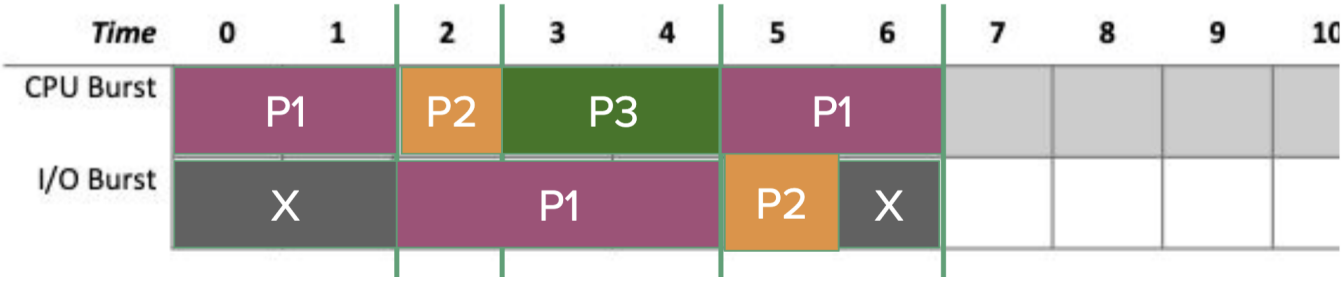
Ready Queue: { (P2, 1), (P3, 3), (P1, 2) }



Ready Queue: { (P3, 3), (P1, 2), (P2, 1) }



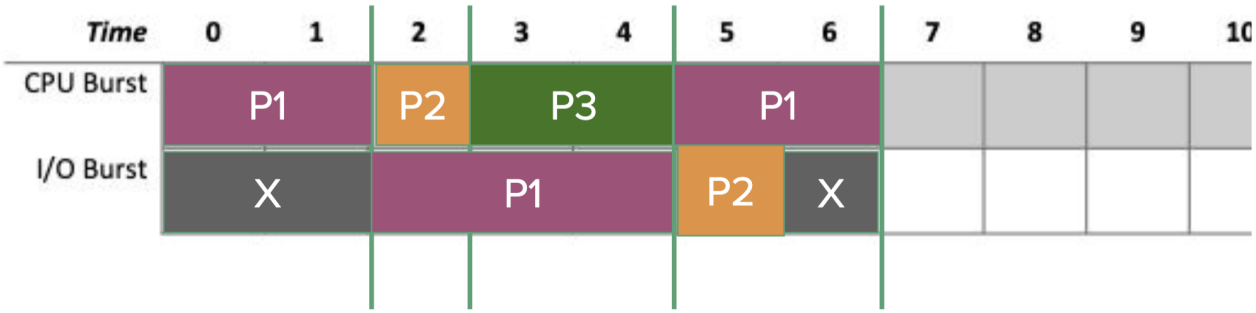
Ready Queue: { (P1, 2), (P2, 1), (P3, 1) }



What do we queue next?

Option #1:
Queue P3! This is the process we kicked off, and there's still 1 CPU burst left

Option #2:
Queue P2! This process is ready to go back in on the CPU now that it has completed its I/O



Answer: Either!

- Both approaches are correct!

Option #1 is correct based on the argument that we shouldn't move P3 to the end of the ready queue if it hasn't completed its first CPU burst – that is, **processes should only be moved to the end of the queue if they've moved to the I/O burst.**

Option #2 is correct based on the argument that **P2 is now ready to use CPU resources for its second burst, and P2 is favored over P3**, so we should move P2 to the CPU rather than P3. This is common in Linux schedulers especially :)

Key Takeaway: Any non-preemptive scheduling algorithm can be made preemptive!

SRTF:

- What algorithm do we get if we remove preemption?
- Shortest job first (SJF)

Round Robin:

- What happens if our time slice is infinitely long?
- First come first serve

Metrics

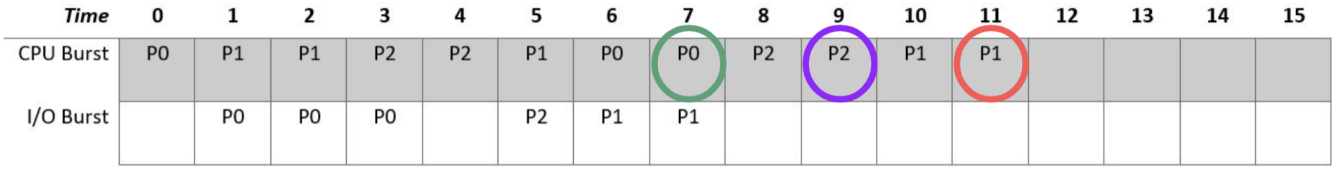
Two types

1. User centric
2. System centric

Metric	Explanation	How to find	Centric
Throughput	Number of jobs executed in a period of time	numJobs/time	System
Average Turnaround time	Average elapsed time between when jobs enter the system and when they leave	sum(leave - enter)/numJobs	System
Average Waiting Time	Average time a job is in the ready queue and not execution	sum(waitTime)/numJobs	System
Response Time	Turnaround time of a specific process	completionTime	User
CPU Utilization	what % of time CPU is doing something	busyTime/totalTime	System

Metrics Example

Turnaround Time Calculations



Turnaround for P0 = time left - time entered = 8 - 0 = 8
Turnaround for P1 = time left - time entered = 12 - 0 = 12
Turnaround for P2 = time left - time entered = 10 - 0 = 10

** note that all processes enter the system at time 0 because they're all ready to run when they enter **

Average turnaround time = (8 + 12 + 10)/3 = 10

Waiting Time Calculations

We have established our average turnaround time to be 10.

Average waiting time = average turnaround time - average execution time

Process #0:
- CPU burst #1 = 1
- I/O burst #1 = 3
- CPU burst #2 = 2

Process #1:
- CPU burst #1 = 3
- I/O burst #1 = 2
- CPU burst #2 = 2

Process #2:
- CPU burst #1 = 2
- I/O burst = 1
- CPU burst #2 = 2

Execution time: 1 + 3 + 2 = 6

Execution time: 3 + 2 + 2 = 7

Execution time: 2 + 1 + 2 = 5

Average execution time = (6 + 7+ 5)/3 = 6

Average waiting time = 10 - 6 = 4

Intro to Memory Management

What else do we need to do to ensure that the processor can execute these processes accurately?

For instance – what would happen if all these processes could write to each other's data?

Goals of Memory Management

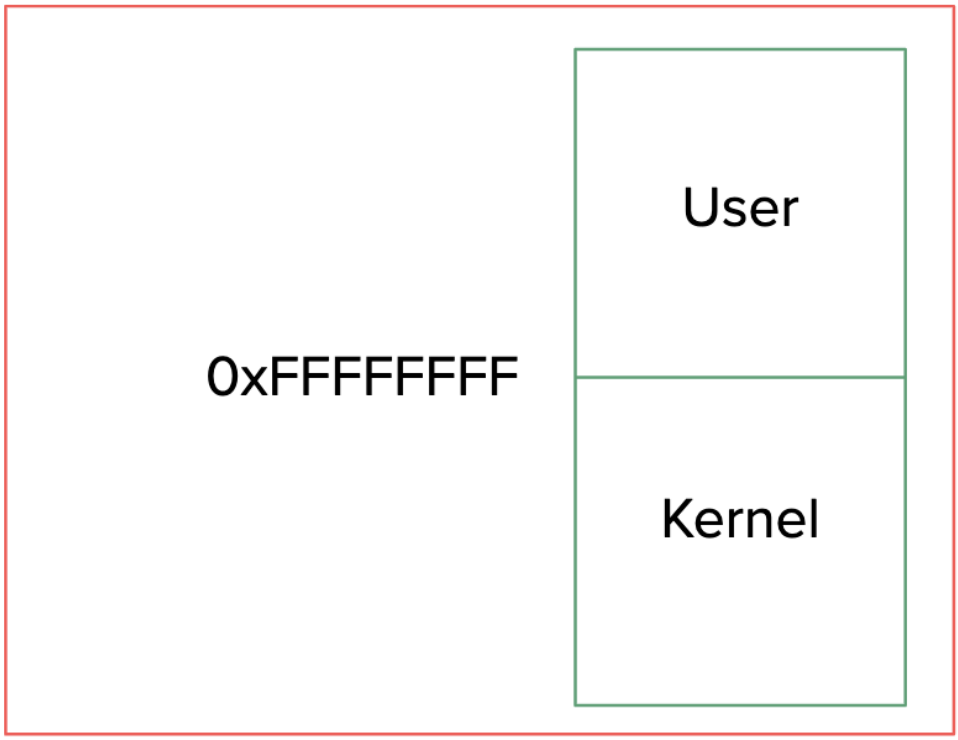
- Maximize resource utilization
- Isolate processes from one another
  - We don't want process 1's data to overwrite process 2's data in memory
  - We can't trust processes to be well behaved
- Liberation from resource limitations
  - We don't want processes to worry about the scarcity of memory. From their perspective, they think they have access to the entire system's memory
- Faciliate memory sharing
  - Some processes may want to share memory either implicitly or explicitly

In short, we need memory management techniques to ensure ***isolation*** and ***protection***.

Simple Management Schemes

Fence Register

- Ensures separation of user and kernel
- Fence register checks the memory addresses
- Broker traps if the user tries to access an invalid memory location



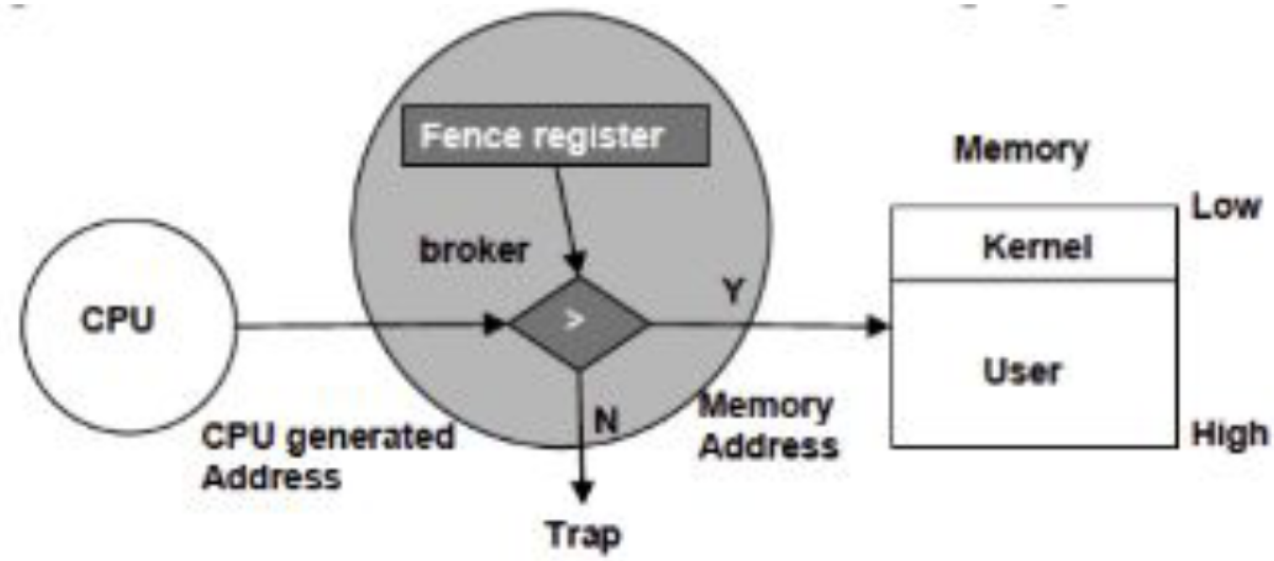


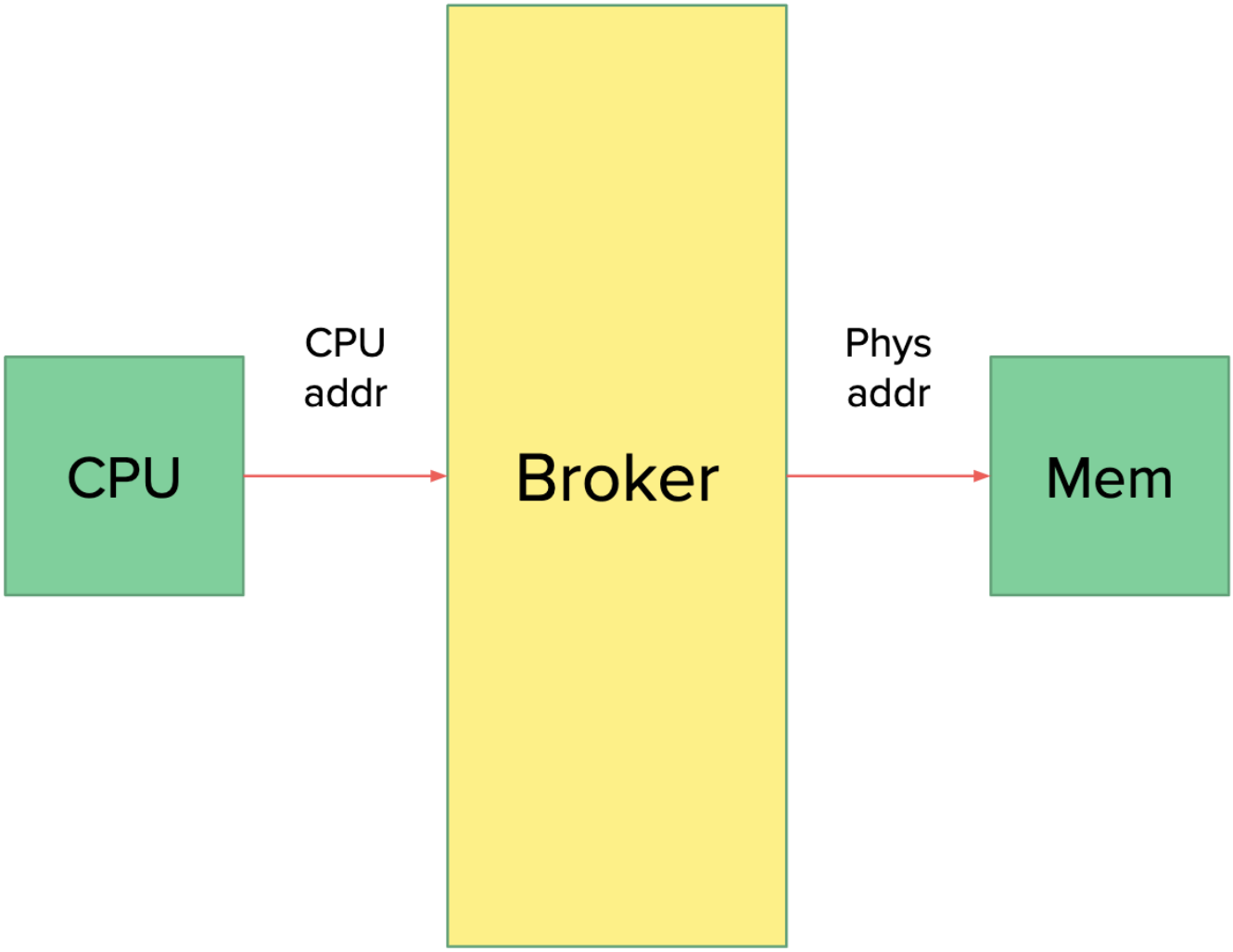
Figure 7.2: Fence Register

Memory Broker

To facilitate the goals of memory management, we introduce a hardware broker between the CPU and memory

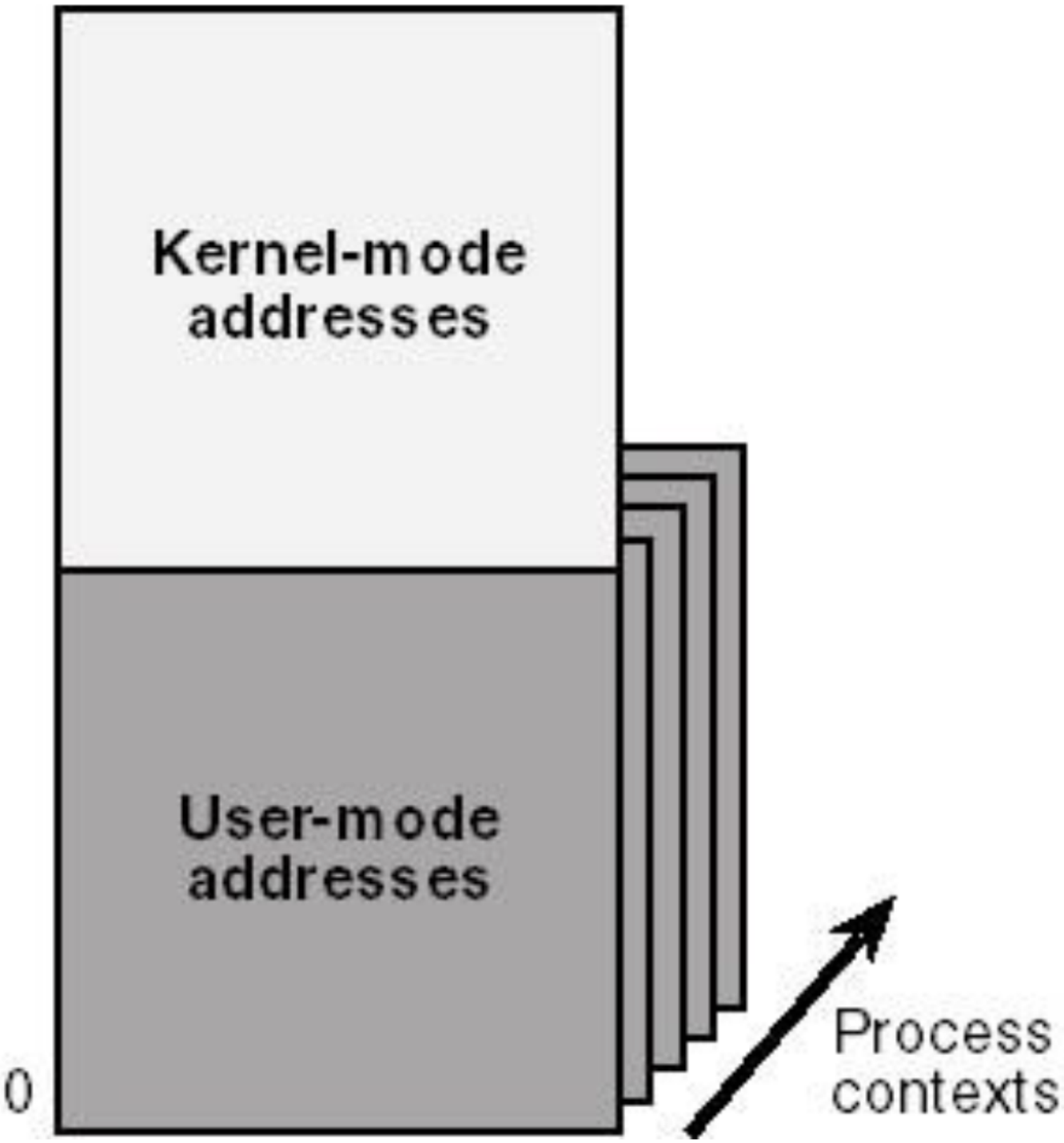
- The broker provides the hardware mechanisms to implement management policies

- Good memory manager only requires minimal change to hardware



Multiple Processes

Fence registers only take care of user-kernel separation, but we don't want processes to write into other processes' memory



Static Relocation

- Lower and upper bounds specified and then stored in the PCB for each process
- Allows for swapping process data in and out of memory
  - Can't be mapped to any other location other than their static assignment
- Let's say Process A is swapped out of memory and B is swapped into the same or overlapping spot. A can't be scheduled till B is completed or swapped out
- Poor memory utilization

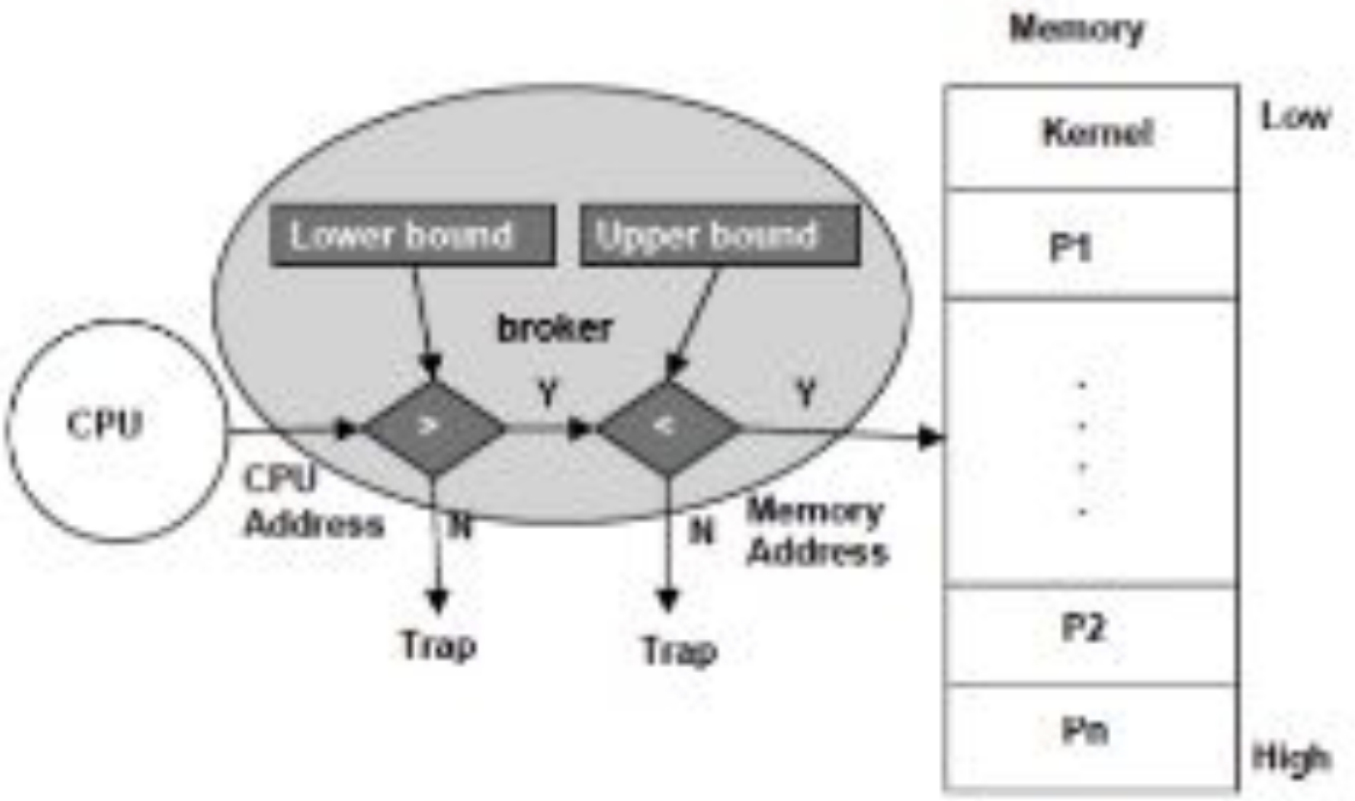


Figure 7.3: Bounds Registers

Dynamic Relocation

- Can place process in any region of memory
- Base + limit registers store the bounds of the process
- Solves poor memory utilization problem because each process is given as much space as it needs

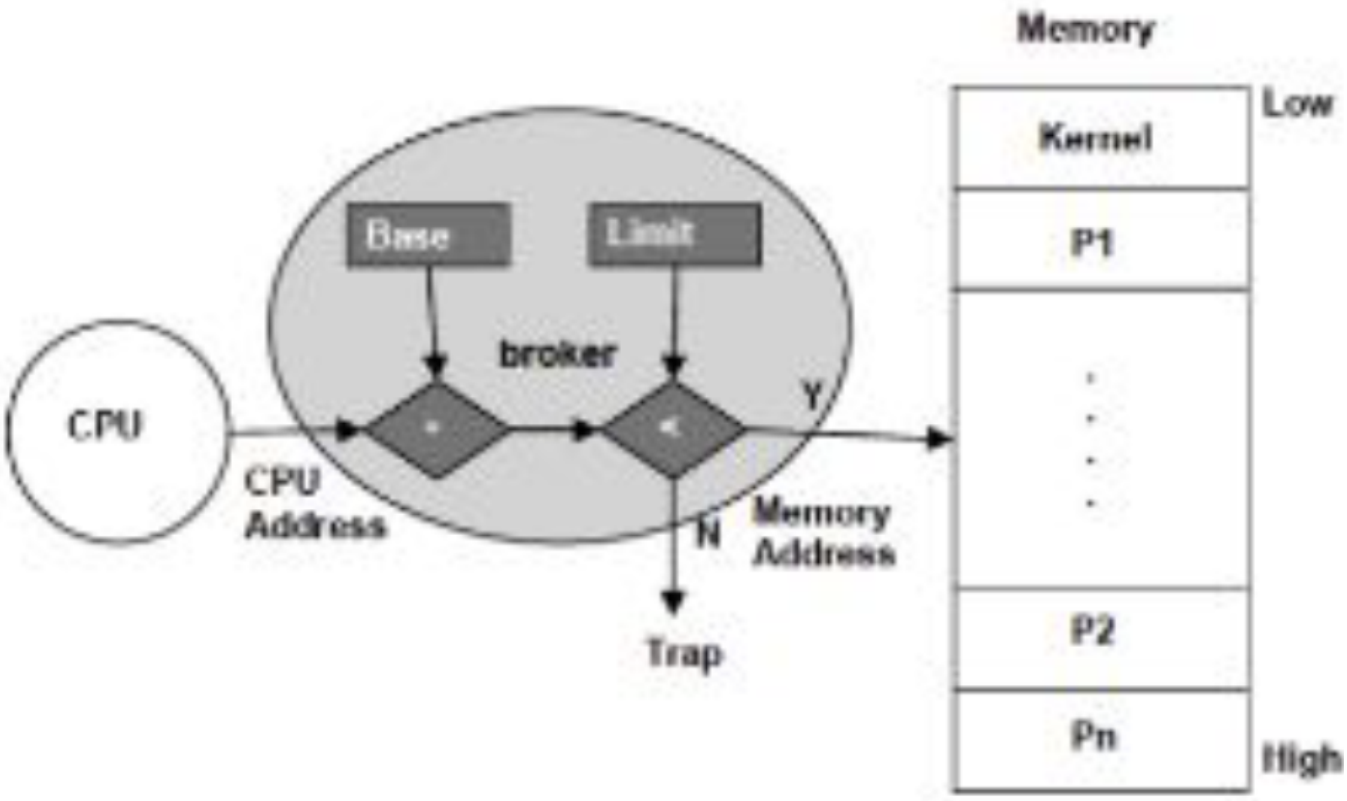


Figure 7.4: Base and Limit Registers

Memory Allocation

Option #1: Fixed size partitions = when we divide memory up into fixed size partitions
Option #2: Variable size partitions = same as fixed size, but the partition is determined by how much the process asks for

Figure 4: Fixed Partitioning

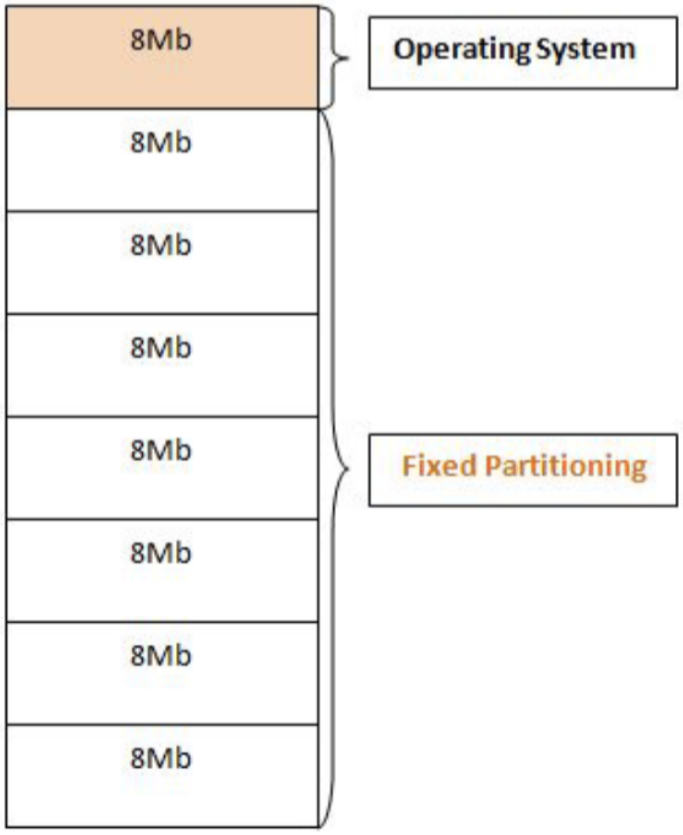
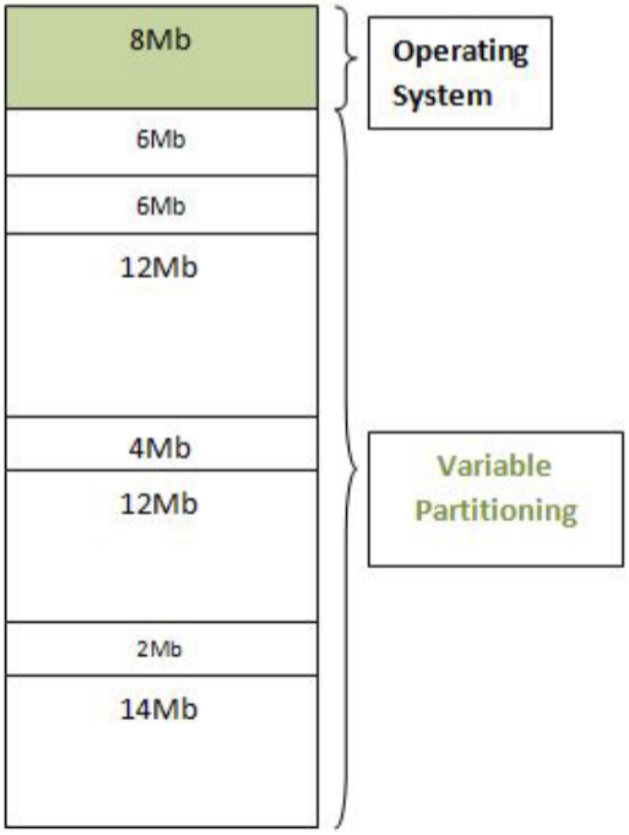


Figure 5: Variable Partitioning

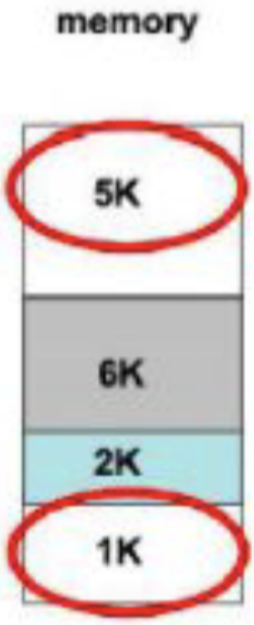


Fixed Size Partition

What happens when we pre-allocate memory without knowing how much our processes need?

Allocation table

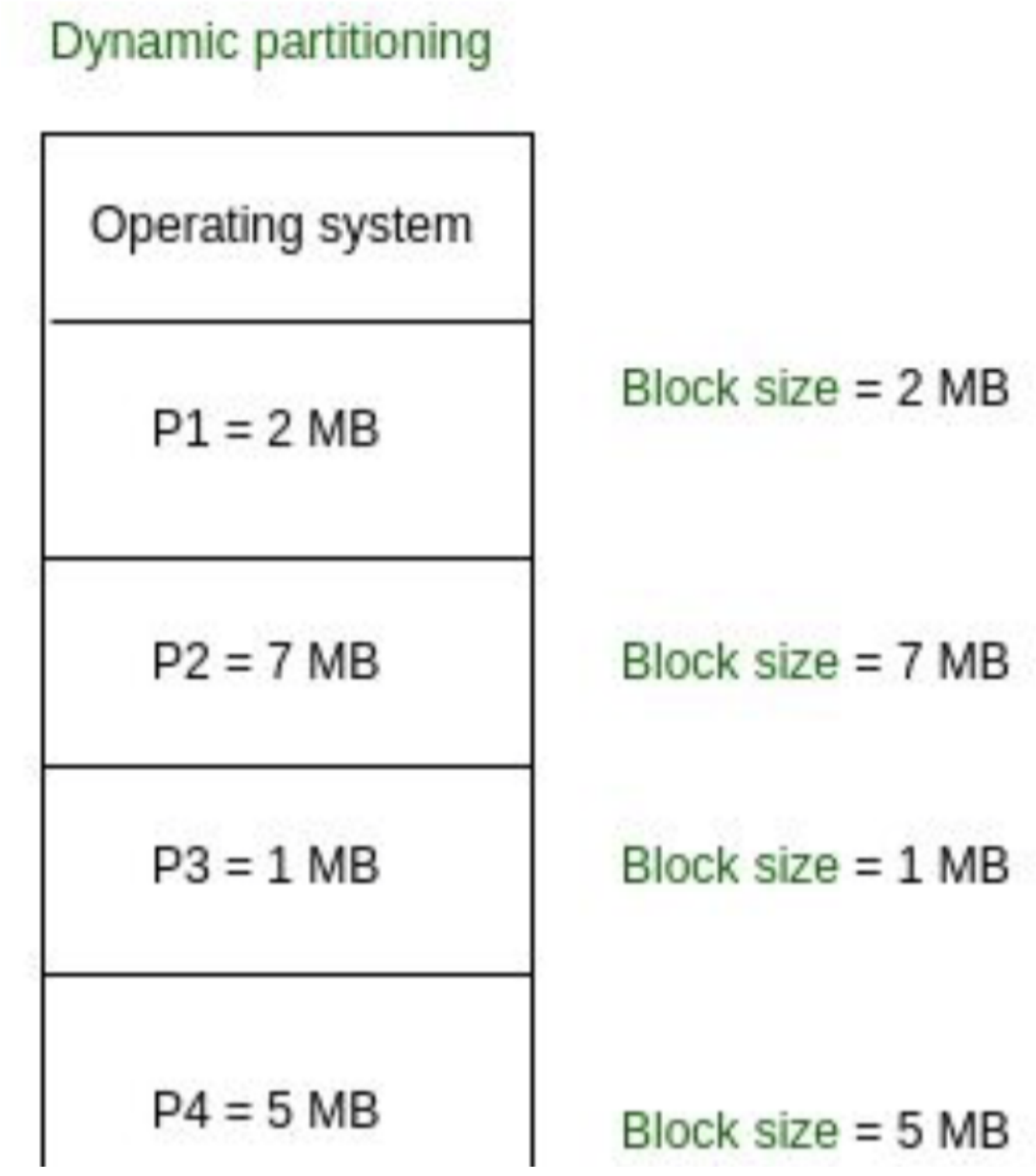
Occupied bit	Partition Size	Process
0	5K	XXX
1	8K	P1
0	1K	XXX

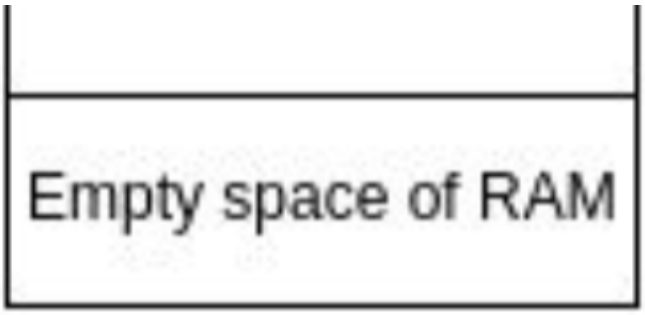


If P2 comes along and asks for 6K of memory, we can't provide it memory since memory has to be contiguous

Variable Size Partition

- "Holes" = free space are kept in the memory layout between processes so memory sizes can be allocated per process more easily
- Still fixed in that after memory has been allocated to a process, it will not change unless the process is removed or swapped out





Partition size = process size
So, no internal Fragmentation

Two More Definitions

Internal Fragmentation = the amount of space that is unused within a memory partition

- Memory is allocated but might not necessary be used
- Doesn't allow for you to use that unused memory for anything else, basically holding memory for no reason

External Fragmentation = the amount of free memory cannot be allocated because it isn't contiguous

- Really bad and very expensive to deal with
- Kills performance
- Compacting processes can help us deal with this