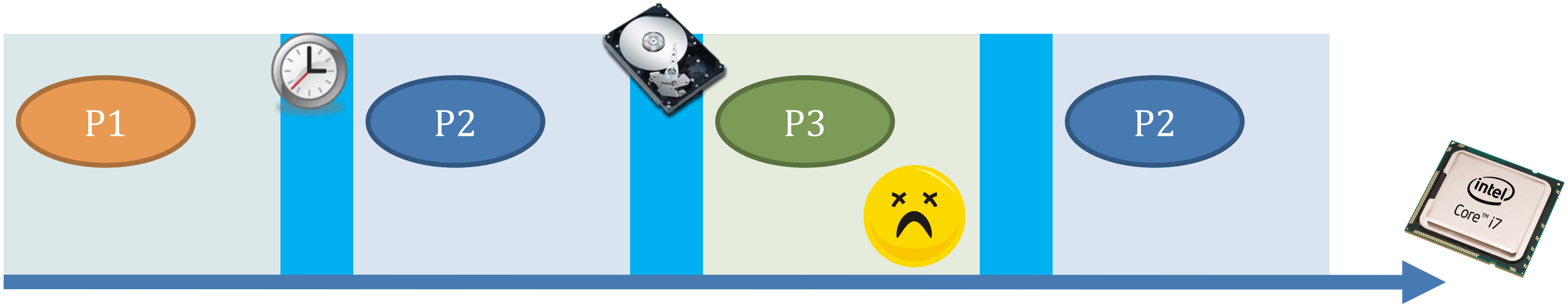
Lecture 5: Job Schedule

Bo Tang @ 2021, Spring

What is context switching?

Scheduling is the procedure that decides which process to run next.

Context switching is the actual switching procedure, from one process to another.



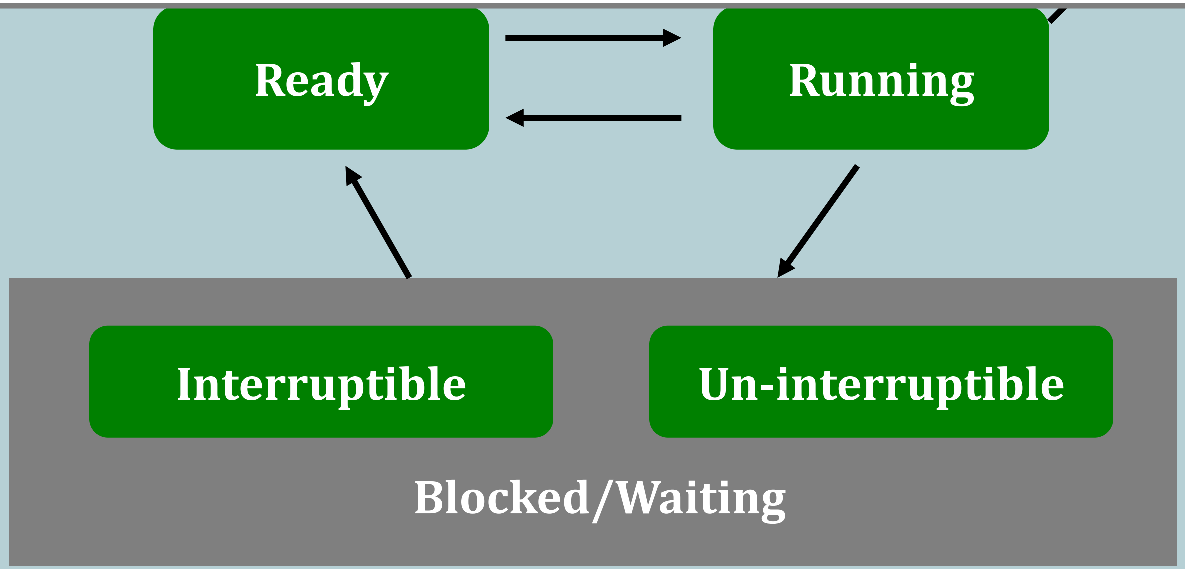
Timer interrupt.

Hardware interrupt.

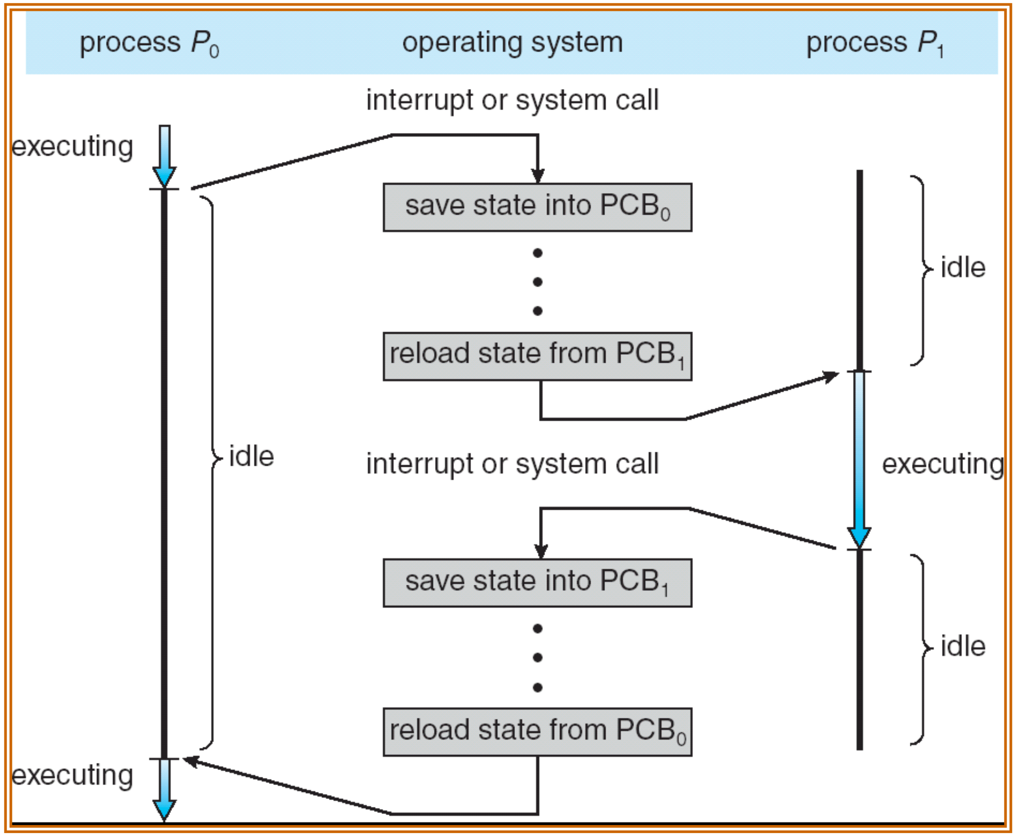
What is context switching?

- Whenever a process goes to blocking / waiting state; e.g., `wait()/sleep()` is called
- A POSIX signal arrives (e.g., SIGCHLD)
- An interrupt arrives (e.g., keystroke)
- When the OS scheduler says “time’s up!” (e.g., round-robin)
    - Put it back to “ready”
- When the OS scheduler says “hey, I know you haven’t finished, but the PRESIDENT just arrives, please hold on” (e.g., preemptive, round-robin <u>with priority</u>)
    - Put it back to “ready”
- ...

Why?
- For multi-tasking
- For fully utilize the CPU



CPU Switch From Process A to Process B



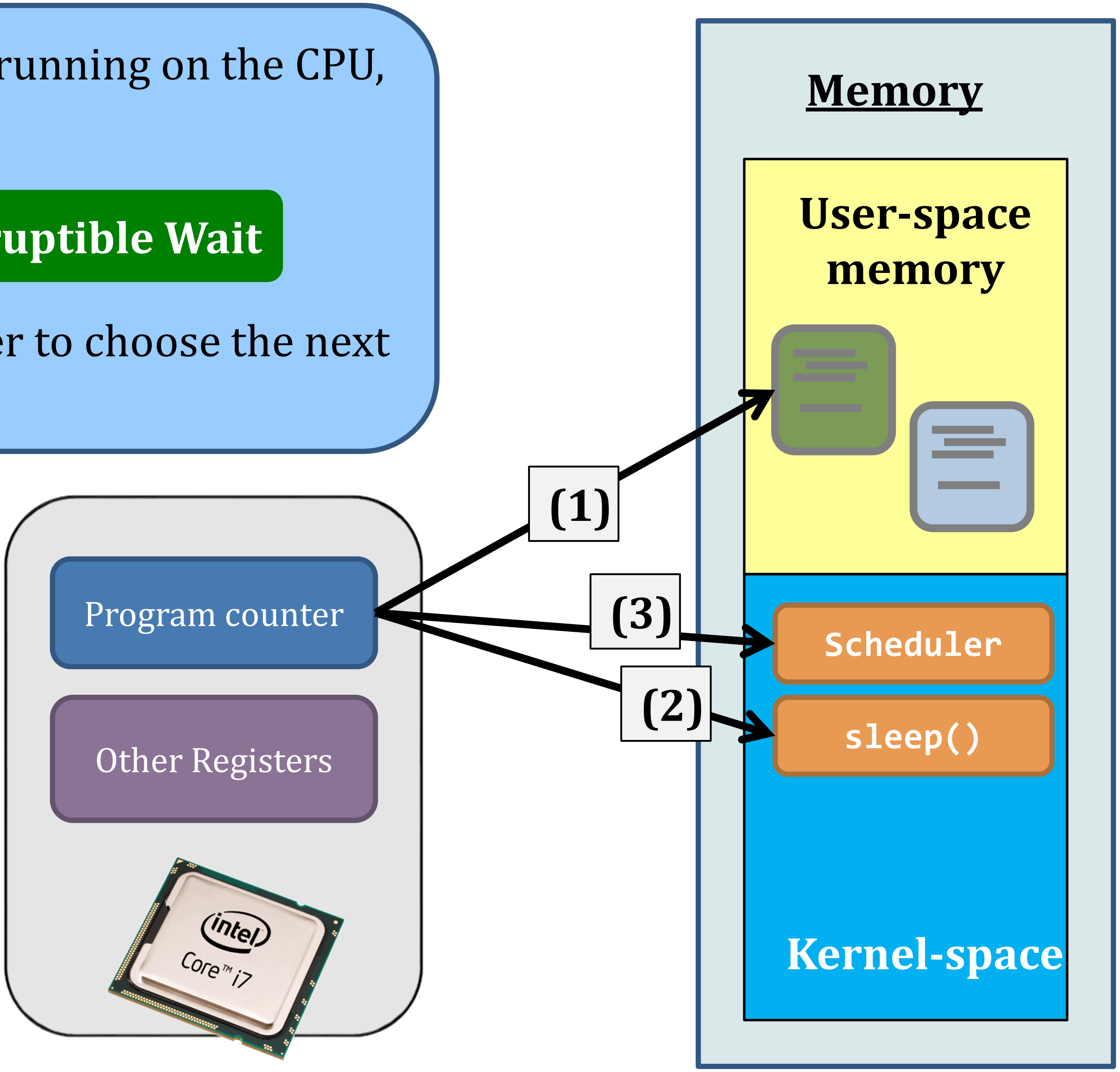
- This is also called a "context switch"
- Code executed in kernel above is *overhead*

Context switching

Suppose this process gives up running on the CPU, e.g., calling **`sleep()`**. Then:

Running → **Interruptible Wait**

Now, it is time for the scheduler to choose the next process to run.

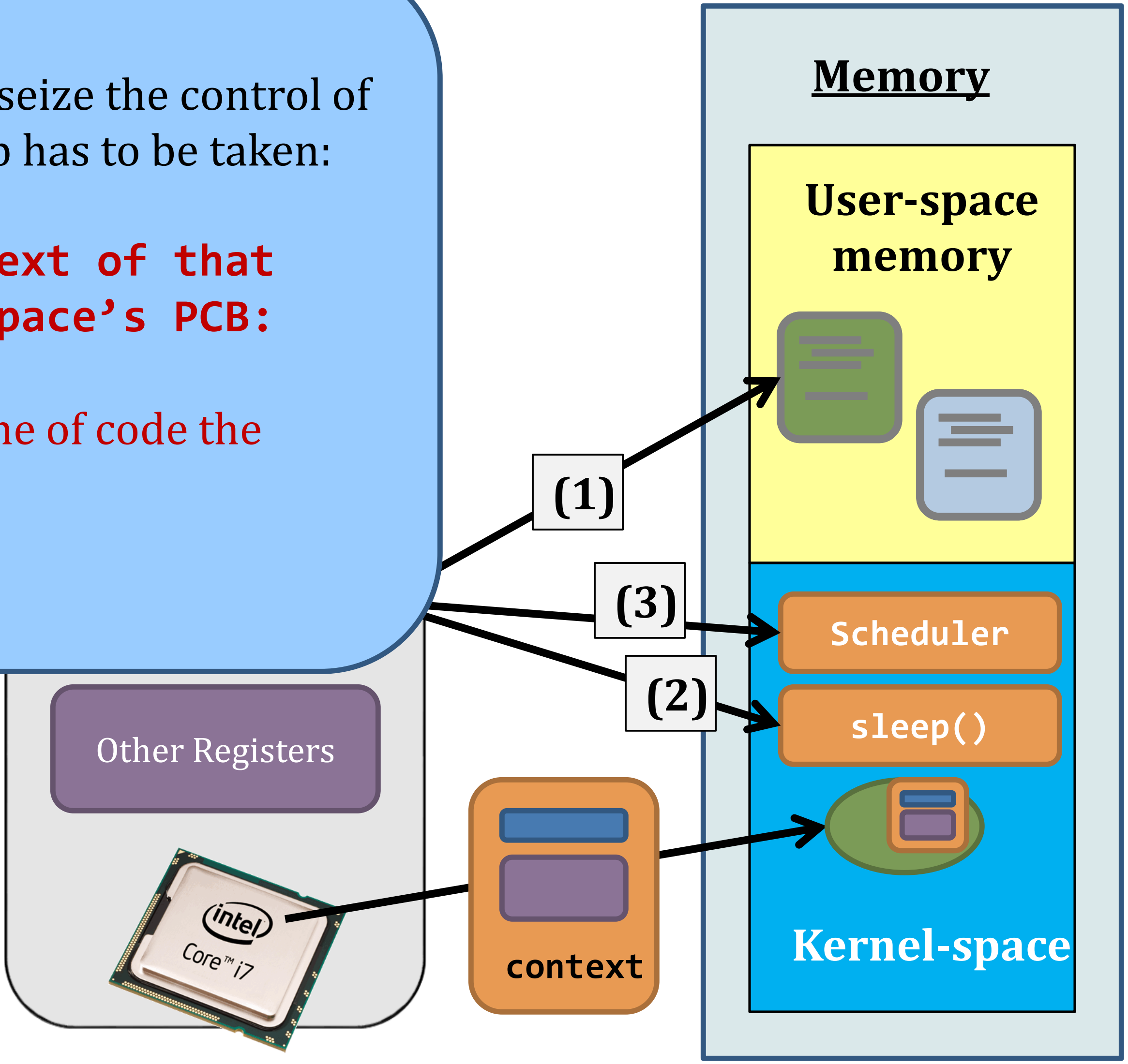


Context switching

But, before the scheduler can seize the control of the CPU, a very important step has to be taken:

`Backup all current context of that process to the kernel-space's PCB:`

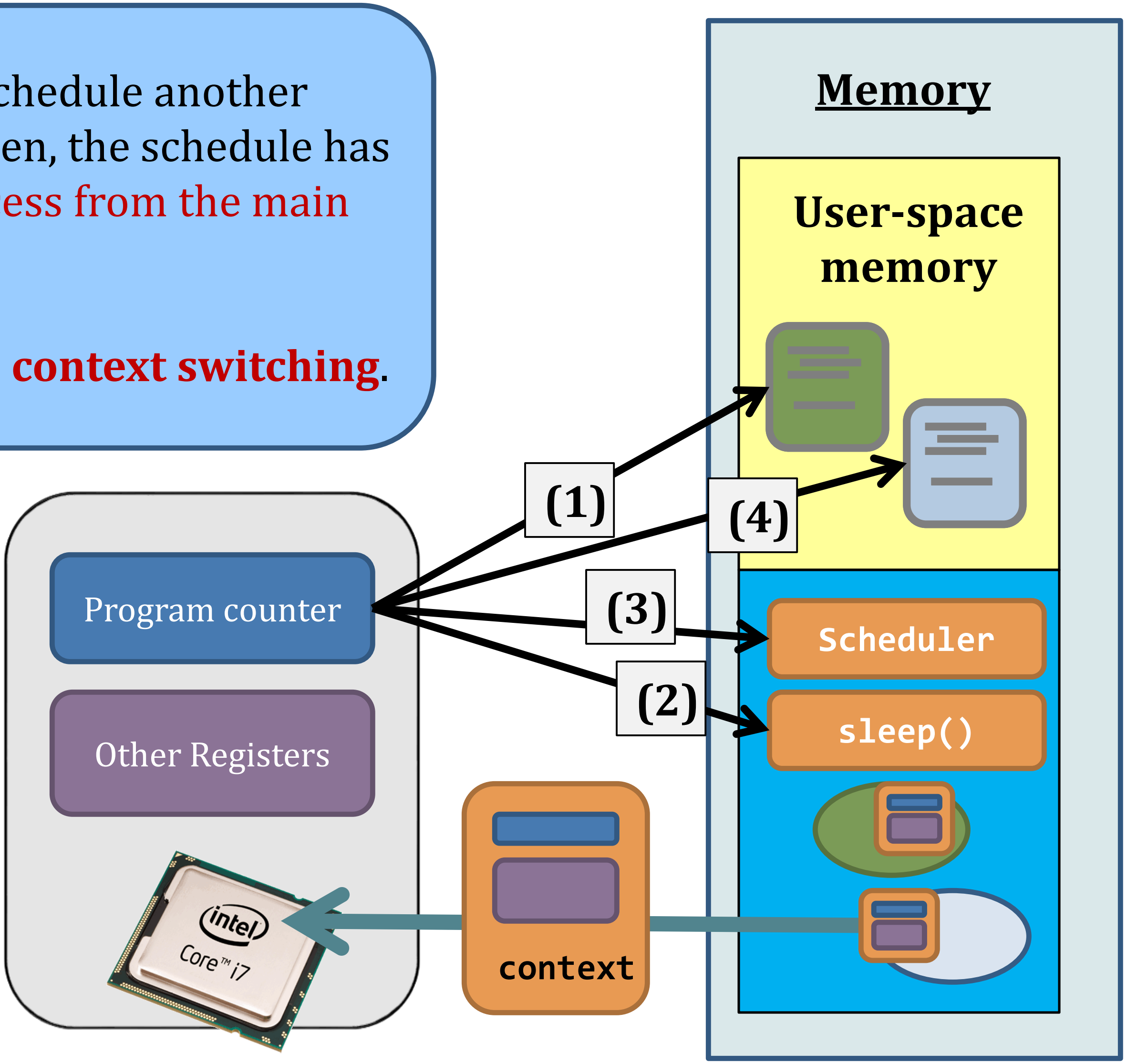
- current register values
- program counter (which line of code the current program is at)
- ...



Context switching

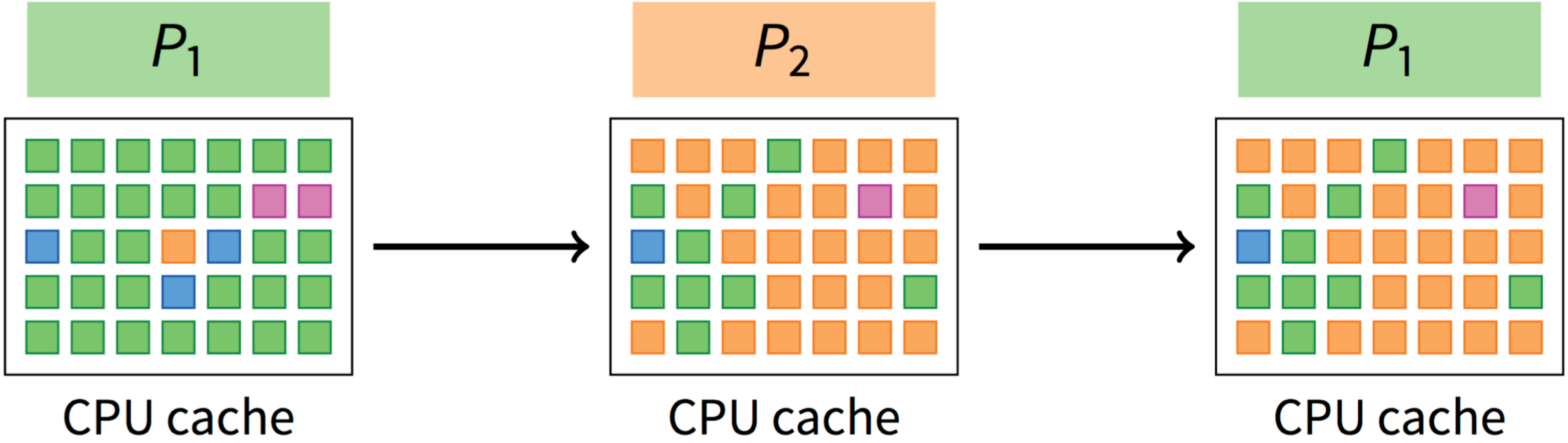
Say, the scheduler decides to schedule another process in the ready queue. Then, the schedule has to load the context of that process from the main memory to the CPU.

We call the entire operation: context switching.



Context switch is expensive

- Direct costs in kernel:
  - Save and restore registers, etc.
  - Switch address space (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses



Ack: Stanford cs140

Topics

- Context switching;
- Scheduling.
  - some basics.

What is process scheduling?

- Scheduling is an important topic in the research of the operating system.
  - Related theoretical topics are covered in computer system performance evaluation.
- Scheduling is required because the number of computing resource – the CPU – is **limited**.

CPU-bound Process	**I/O-bound process**
Spends most of its running time on the CPU, i.e., **user-time > sys-time**	Spends most of its running time on I/O, i.e., **sys-time > user-time**
Examples - AI course assignments.	**Examples** - **/bin/ls**, networking programs.

Classes of process scheduling

- Preemptive scheduling (Non-preemptive is out)

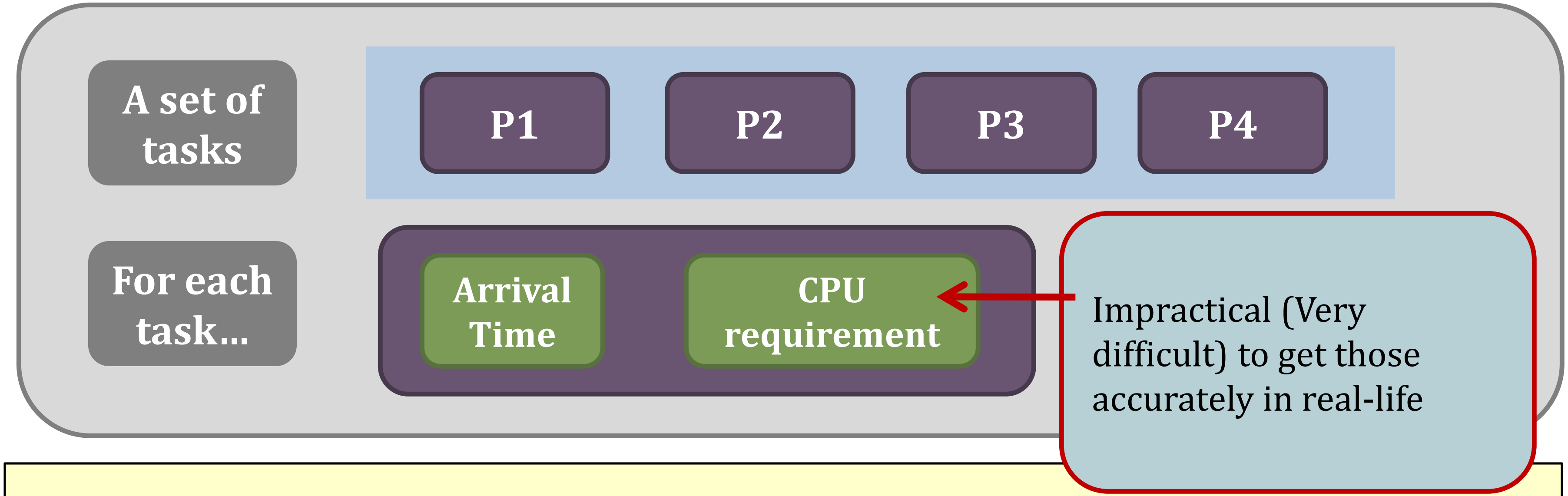
What is it?	When a process is chosen by the scheduler, the process would have the CPU until… -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit()`**. -**particular kinds of interrupts (e.g., periodic clock interrupt, a new process steps in) are detected**.
History	In old days, it was called "time-sharing" Nowadays, all systems are time-sharing
Pros	Good for systems that emphasize **interactiveness**. - Because every task will receive attentions from the CPU.
Cons	Bad for systems that emphasize the time in finishing tasks.

Topics

- Context switching;
- Process scheduling.
  - some basics.
  - different algorithms.

Scheduling algorithms

Inputs to the algorithms.



A set of tasks

P1

P2

P3

P4

For each task…

Arrival Time

CPU requirement

Impractical (Very difficult) to get those accurately in real-life

Online VS Offline

An **offline scheduling algorithm** assumes that you know the sequence of processes that a scheduler will face
- Theoretical baseline

An **online scheduling algorithm** does not have such an assumption.
- Practical use

Algorithm evaluation

Number of context switches

Individual & average turnaround time

Individual & average waiting time

Turnaround time

The time between the arrival of the task and the termination of the task.

Waiting time

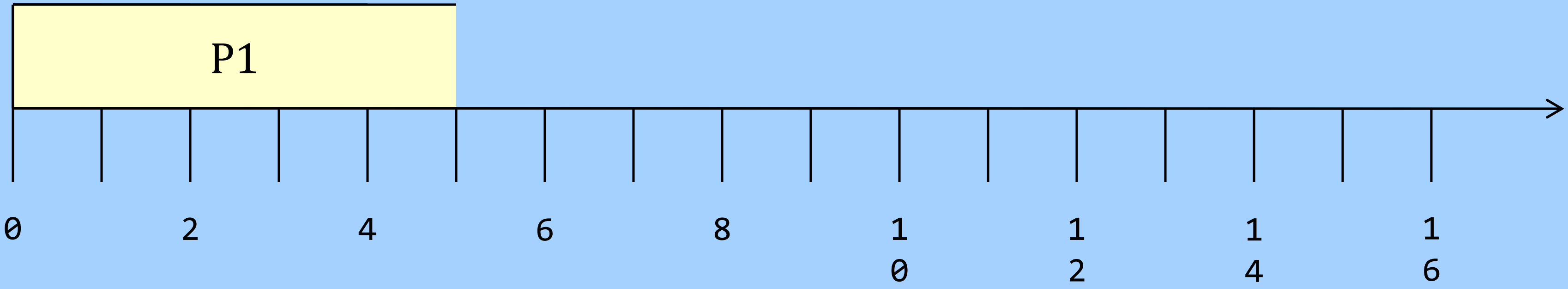
The accumulated time that a task has waited for the CPU.

Different algorithms

Algorithms
Shortest-job-first (SJF)
Round-robin (RR)
Priority scheduling with multiple queues.
…… (lab session)

Assumption: context switch is free (in practice, it is expensive)

Non-preemptive SJF



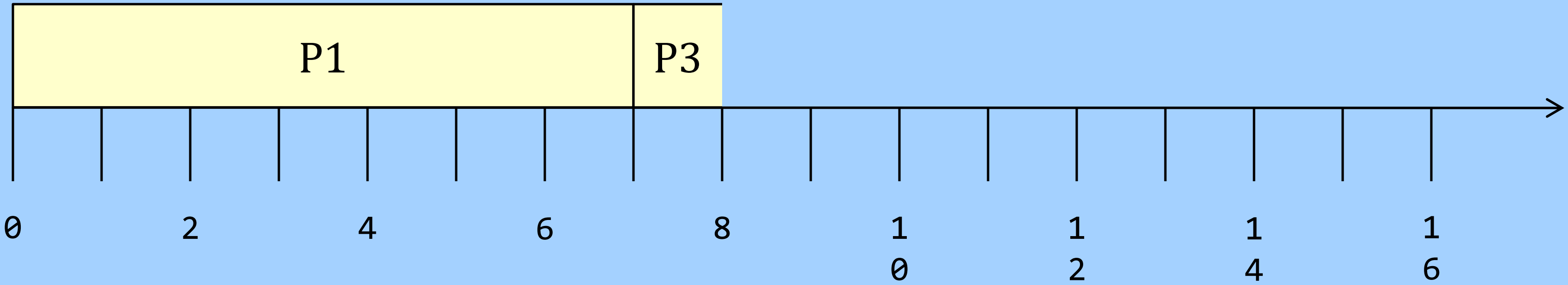
Time = 5

Set of processes

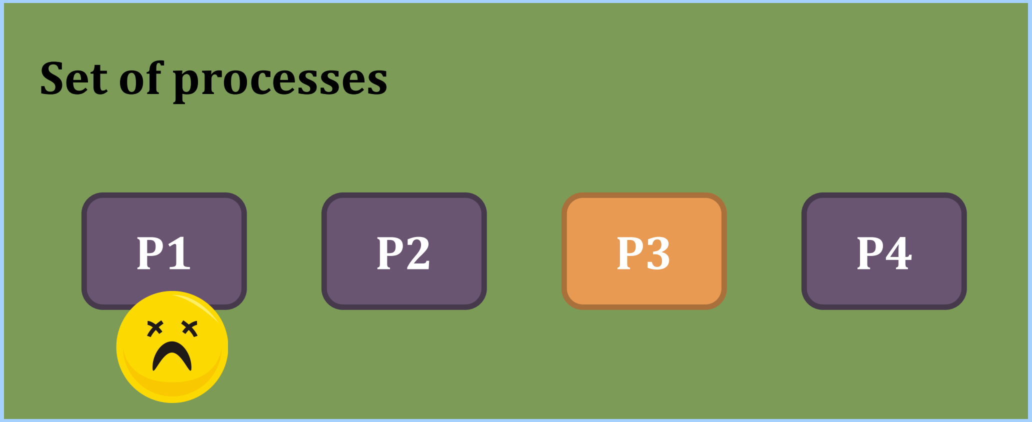
P1 **P2** **P3** **P4**

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



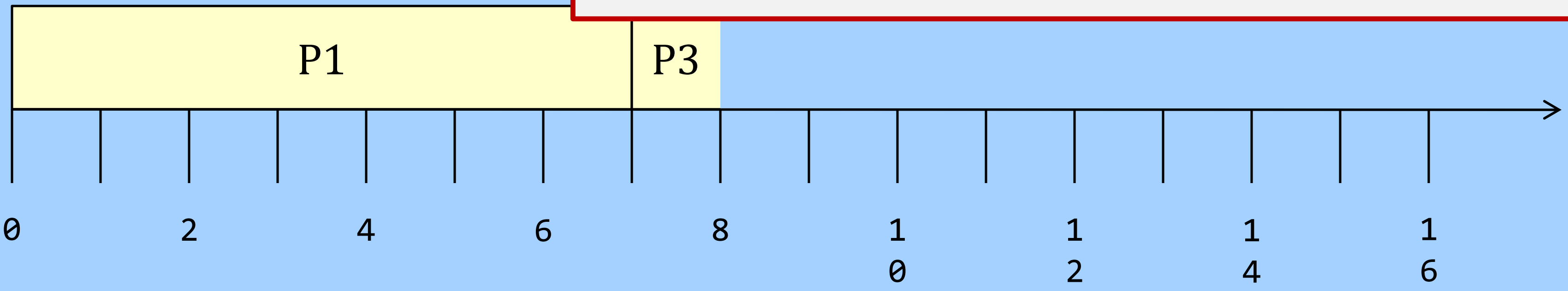
Time = 7



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

In this example, we use **FIFO** to break the tie.



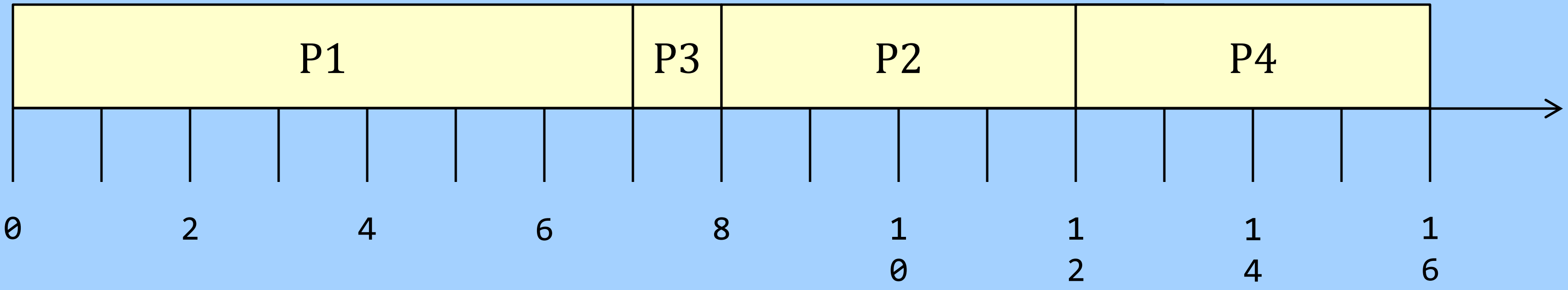
Time = 7

Set of processes

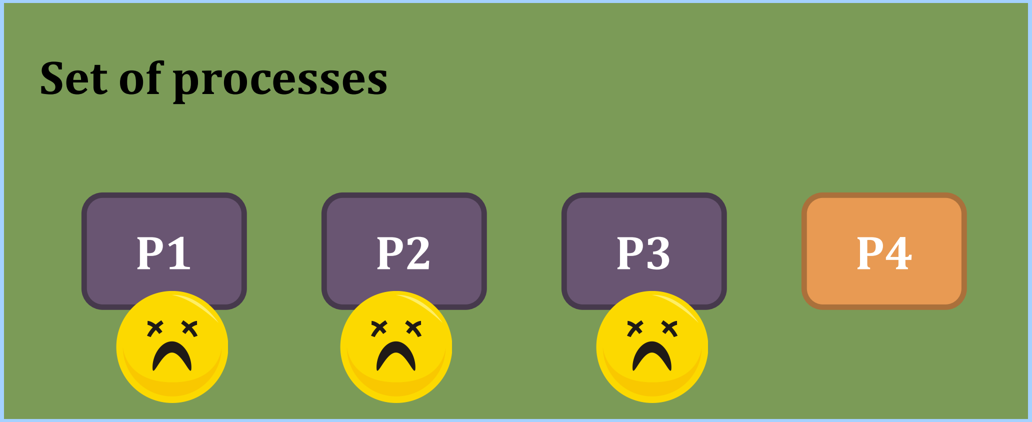
P1 **P2** **P3** **P4**

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

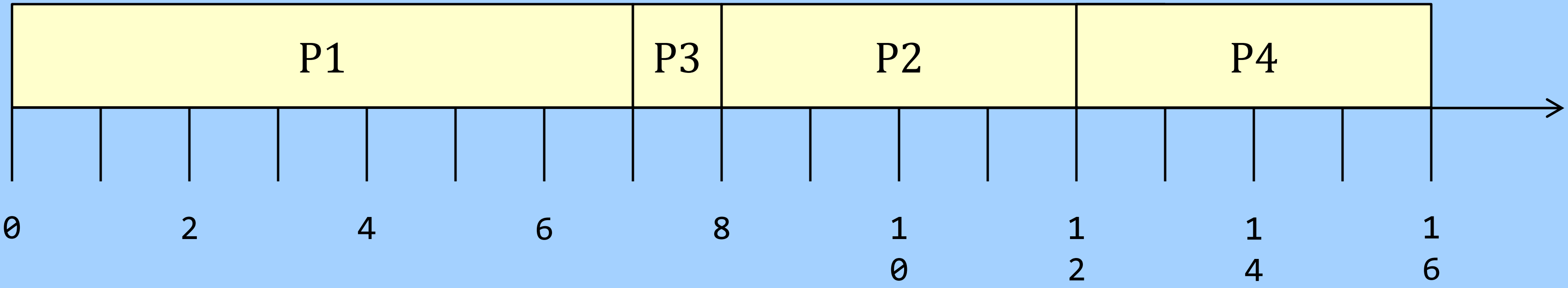


Time = 16



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



Waiting time:

P1 = 0; P2 = 6; P3 = 3; P4 = 7;

Average = (0 + 6 + 3 + 7) / 4 = 4.

Turnaround time:

P1 = 7; P2 = 10; P3 = 4; P4 = 11;

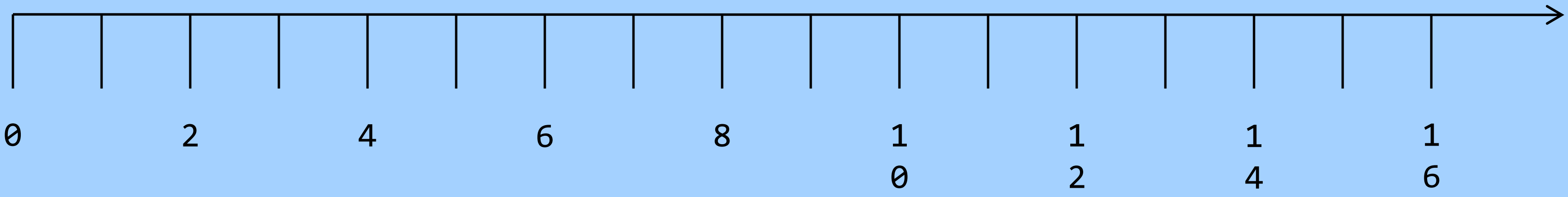
Average = (7 + 10 + 4 + 11) / 4 = 8.

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

SJF

- Problem:
  - What if tasks arrive after P2 all have CPU requirement < 3?
  - Problem persists even for its preemptive version

Preemptive SJF

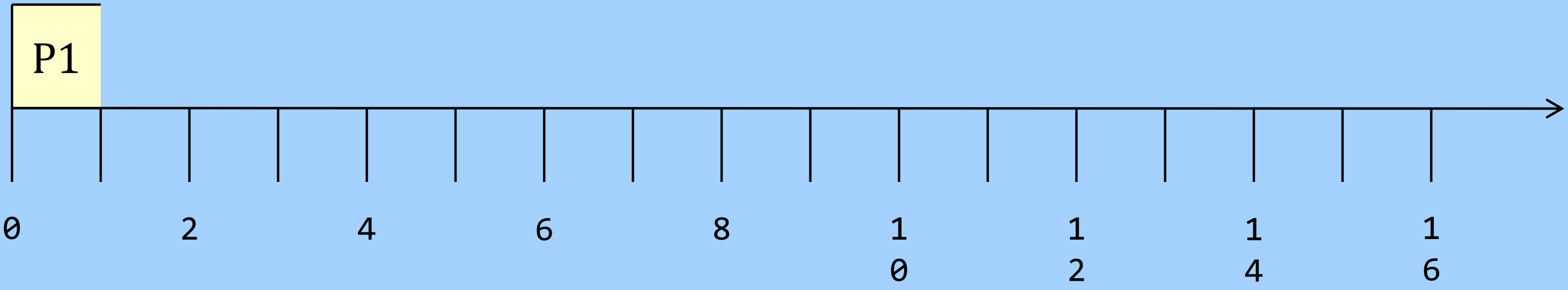


-Whenever a new process arrives at the system, the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Animation; don’t print

Preemptive SJF



Time = 0

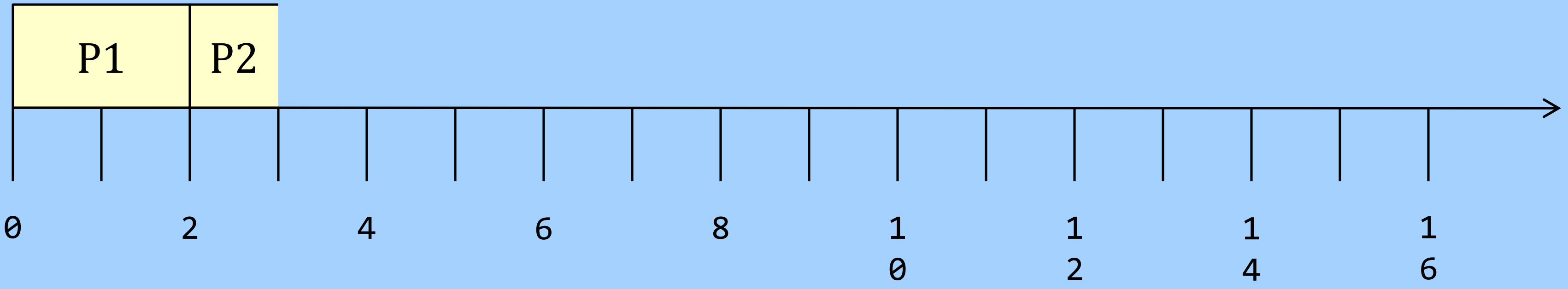
Set of processes

P1

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Animation; don’t print

Preemptive SJF



Time = 2

P2 is selected!

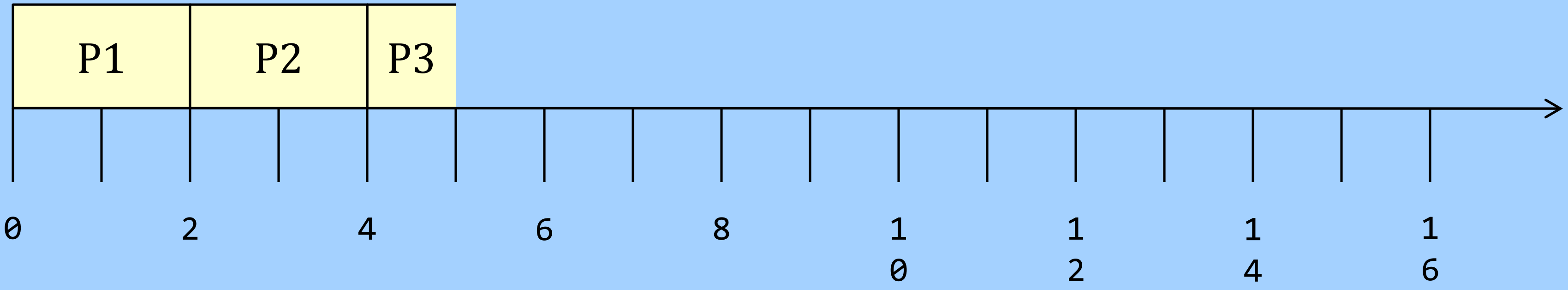
Set of processes

P1 **P2**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Animation; don’t print

Preemptive SJF



Time = 4

P3 is selected!

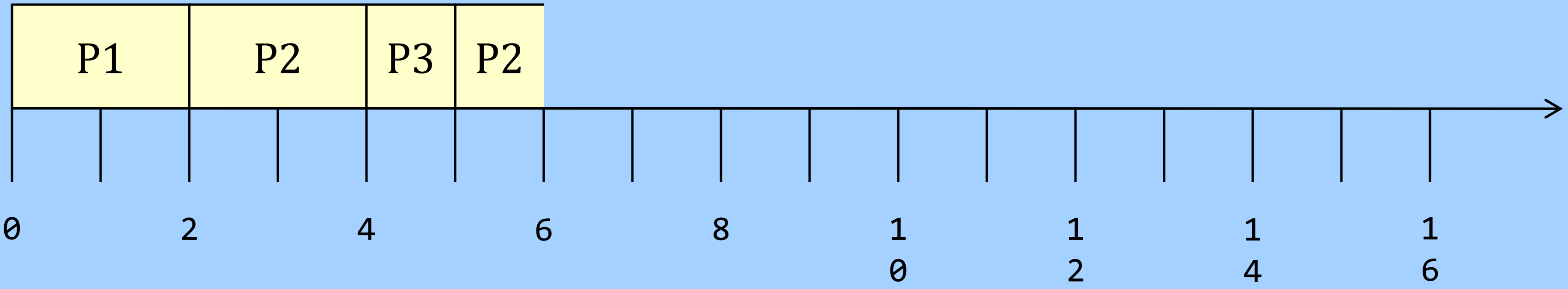
Set of processes

P1 **P2** **P3**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

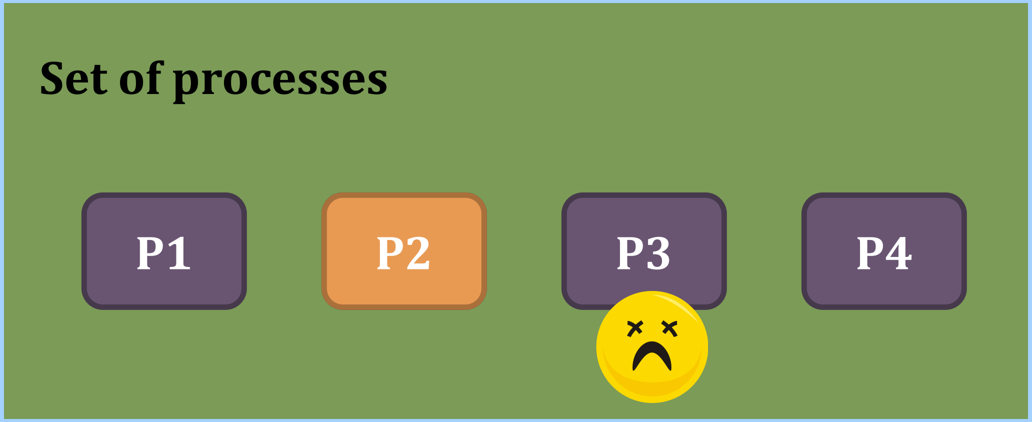
Animation; don’t print

Preemptive SJF



Time = 5

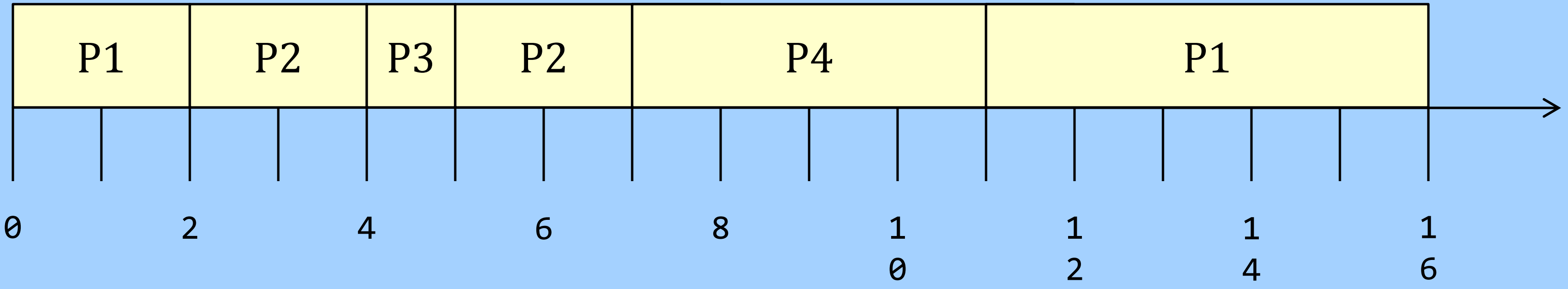
P2 is selected!



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	4

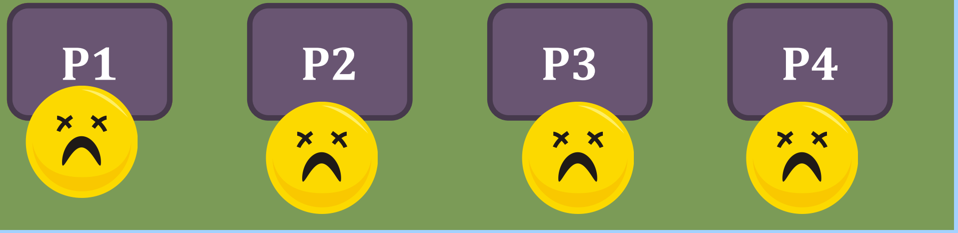
Animation; don’t print

Preemptive SJF



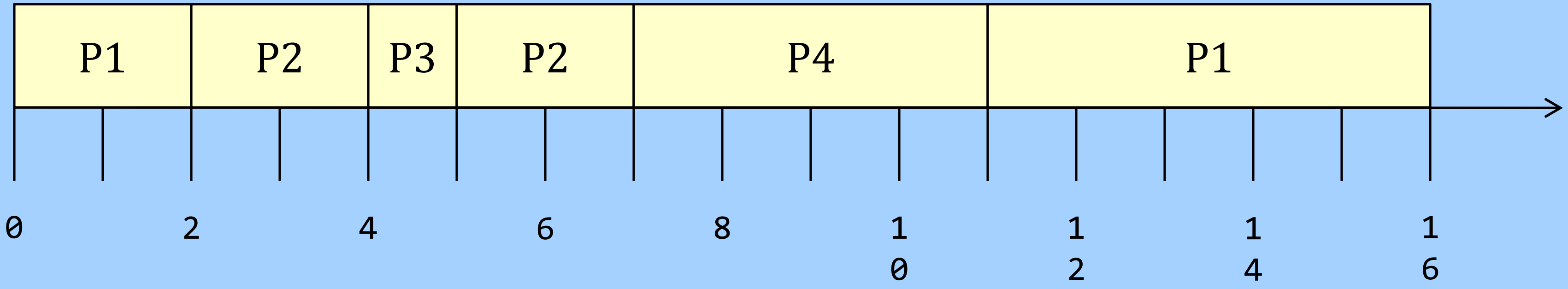
Time = 16

Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

Preemptive SJF



```
Waiting time:

   P1 = 9; P2 = 1; P3 = 0; P4 = 2;

   Average = (9 + 1 + 0 + 2) / 4 = 3.
```

```
Turnaround time:

   P1 = 16; P2 = 5; P3 = 1; P4 = 6;

   Average = (16 + 5 + 1 + 6) / 4 = 7.
```

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

SJF: Preemptive or not?

	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	**3 (smallest)**
Average turnaround time	8	**7 (smallest)**
# of context switching	3	**5 (largest)**

The waiting time and the turnaround time decrease at the expense of the **increased number of context switches**.

Context switch is expensive. (That’s why we shall minimize the # of sys calls as well; on a syscall, the program switch from user-process to kernel-”process”.)

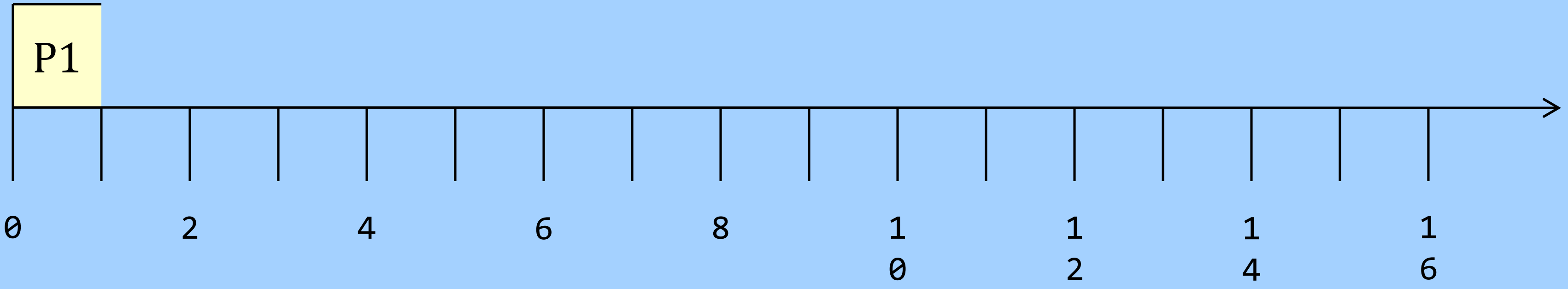
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Round-robin

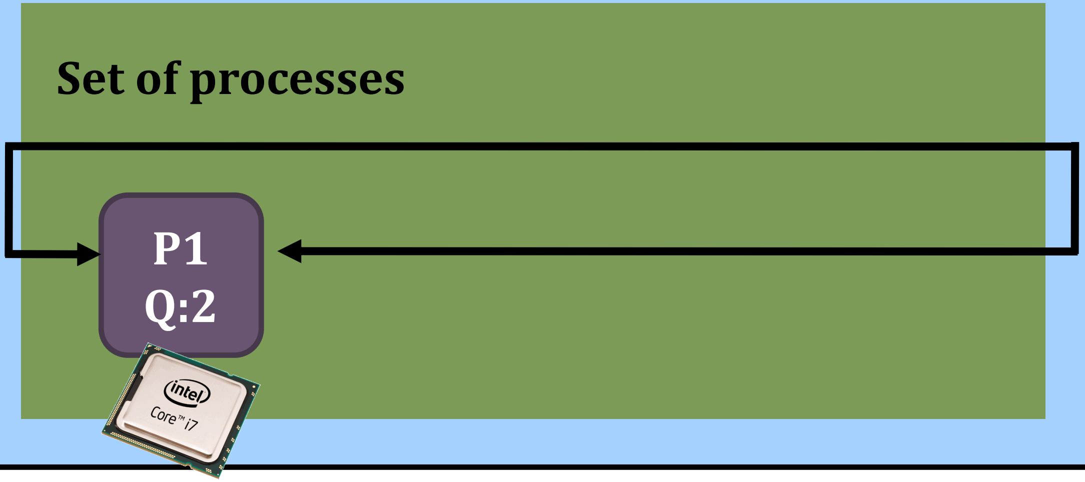
- Round-Robin (RR) scheduling is preemptive.
  - Every process is given a **quantum**, or the amount of time allowed to execute.
  - Whenever the quantum of a process is **used up** (i.e., 0), the process releases the CPU and **this is the preemption**.
  - Then, the scheduler steps in and it chooses **the next process which has a non-zero quantum** to run.
  - If all processes in the system have used up the quantum, they will be re-charged to their initial values.
  - Processes are therefore running one-by-one as a **circular queue**, for the basic version (i.e., no priority)
    - New processes are added to the tail of the ready queue
    - New process arrives won't trigger a new selection decision

Animation; don't print

Round-robin (quantum =2)



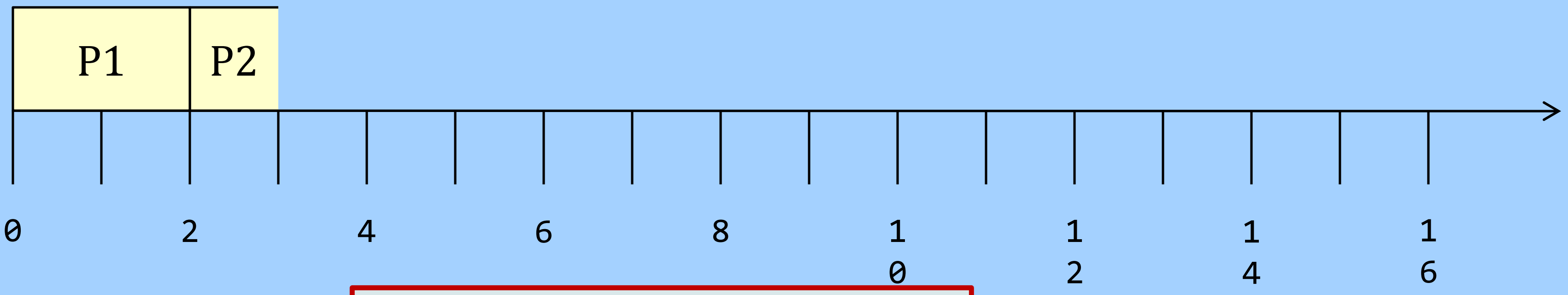
Time = 0



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

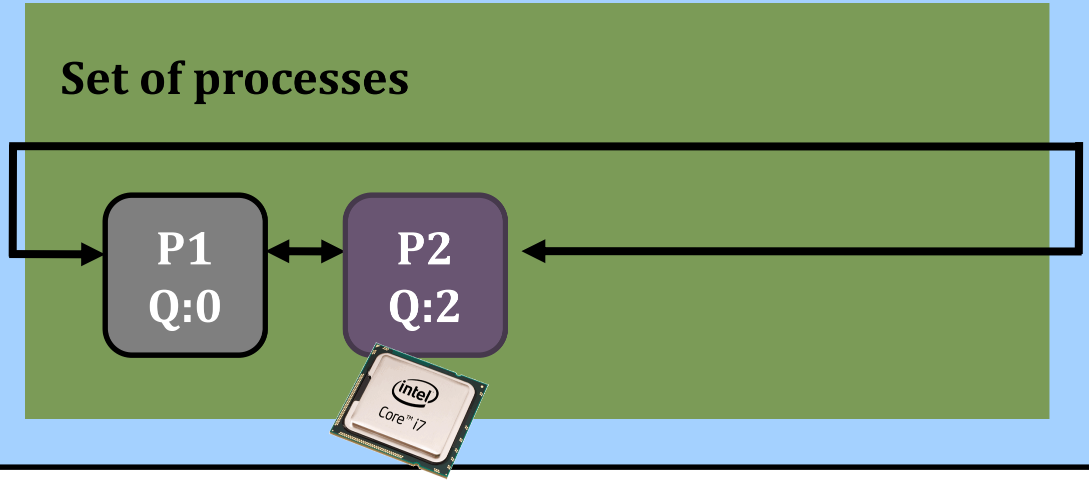
Animation; don’t print

Round-robin



Time = 2

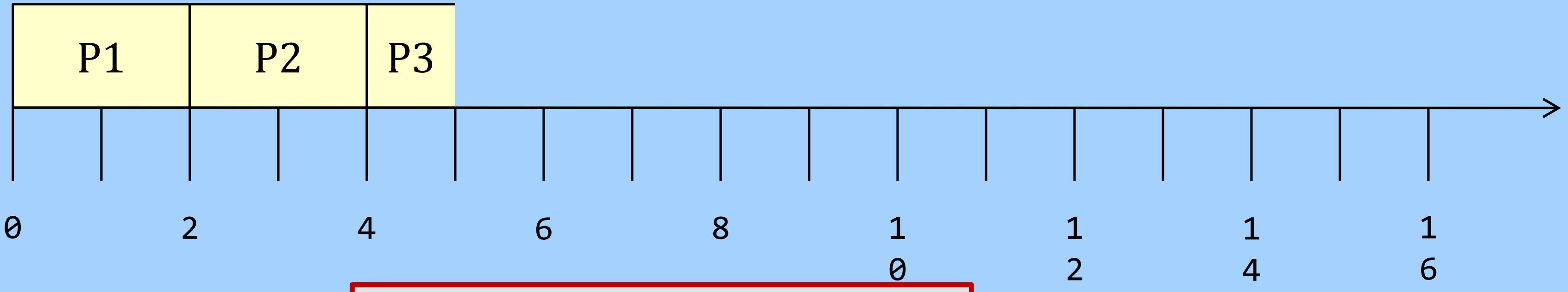
P1’s quantum is 0;
P2 is selected!



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Animation; don’t print

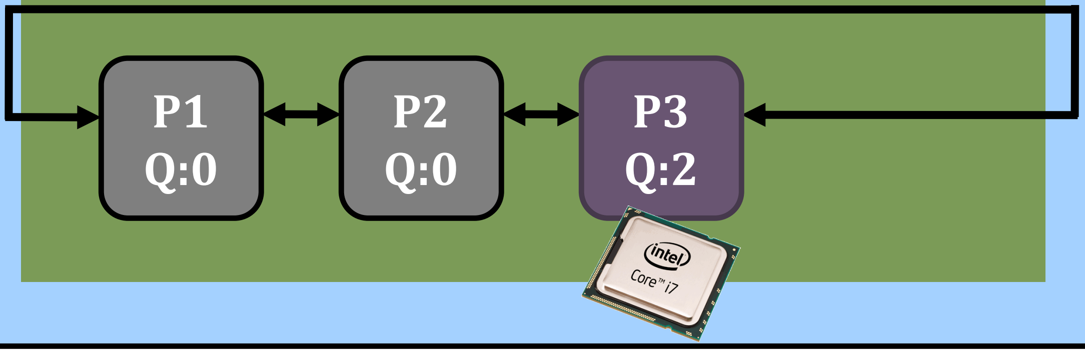
Round-robin



Time = 4

P1’s & P2’s quanta are 0;
P3 is selected!

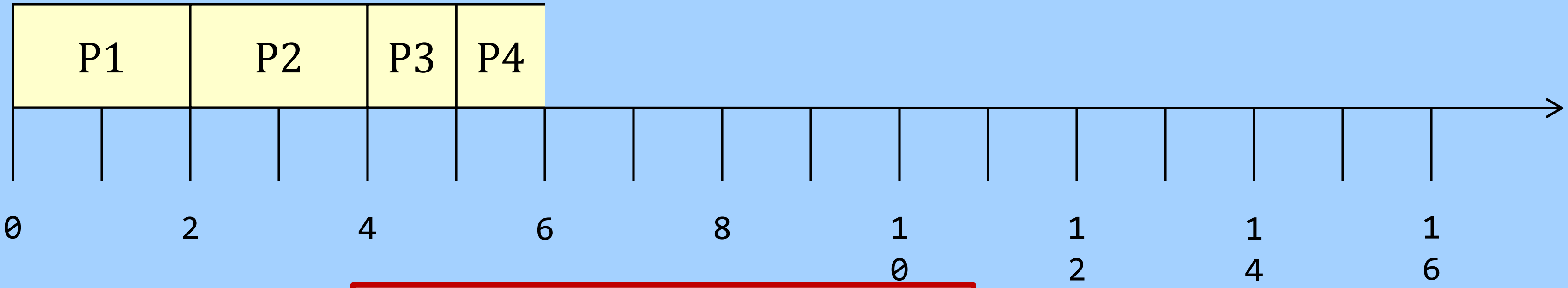
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

Animation; don't print

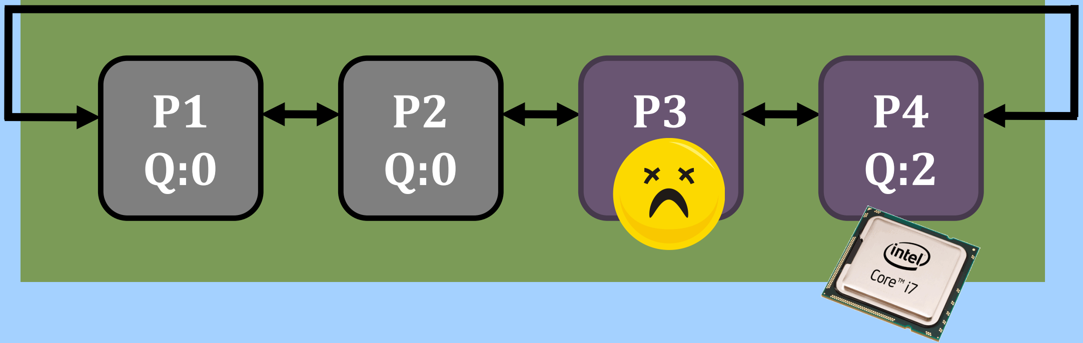
Round-robin



Time = 5

P1's & P2's quanta are 0;
P4 is selected!

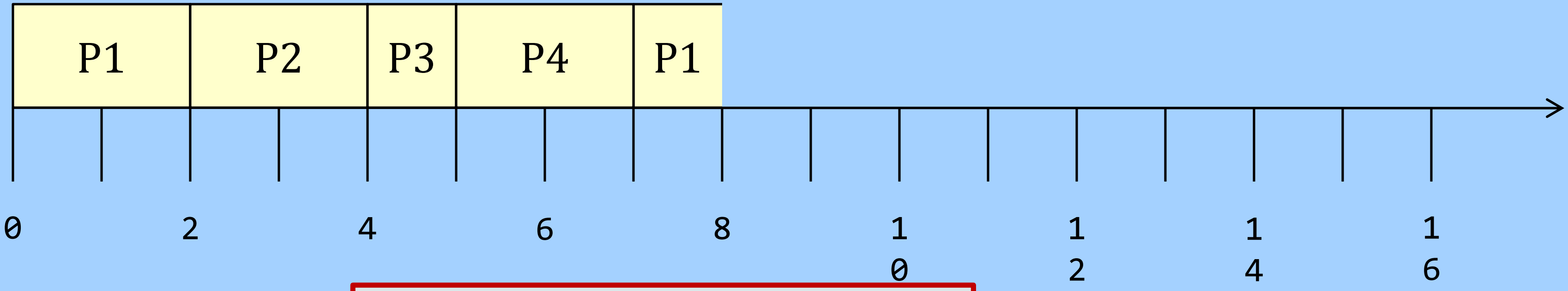
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

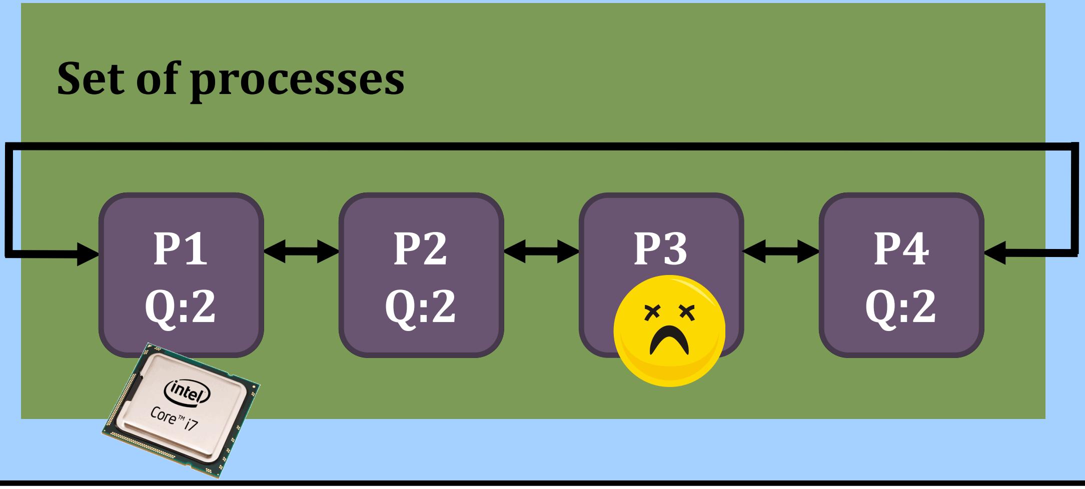
Animation; don’t print

Round-robin



Time = 7

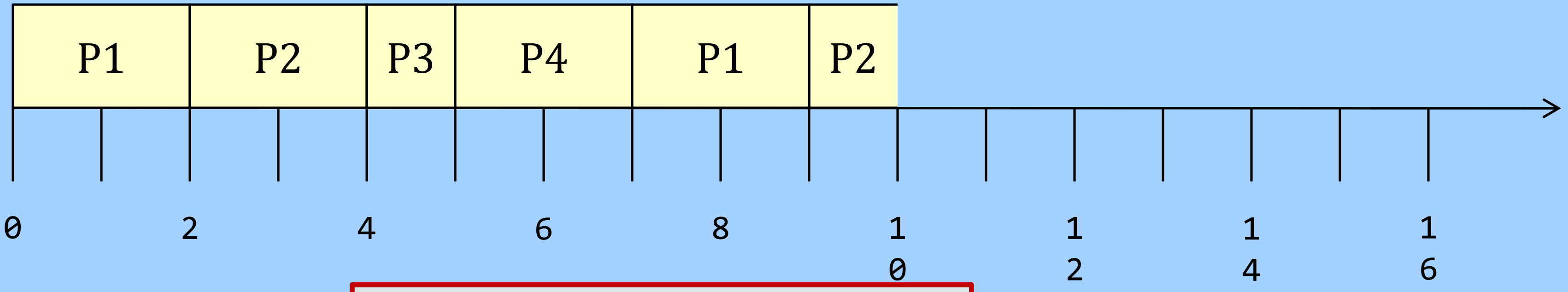
Now, recharge is needed.
P1 is selected.


Set of processes

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

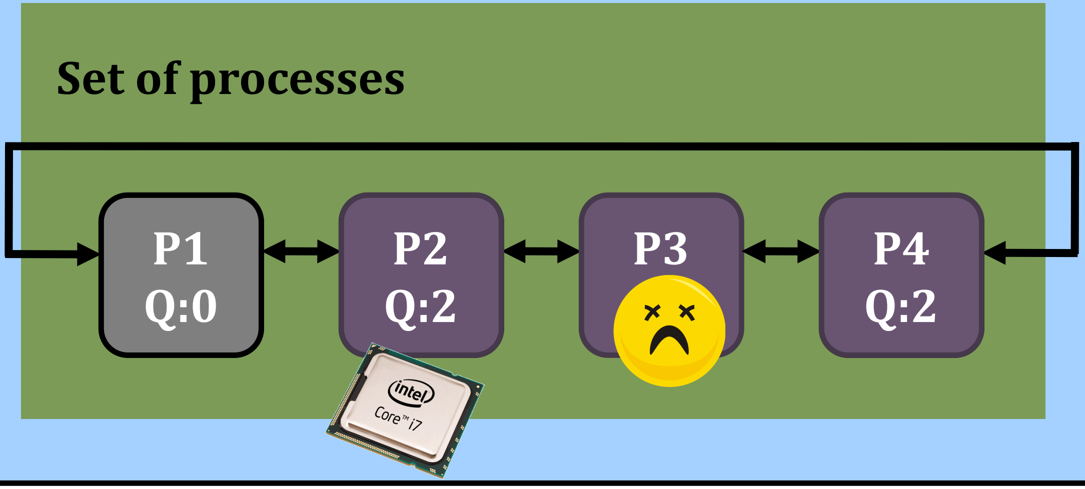
Animation; don’t print

Round-robin



Time = 9

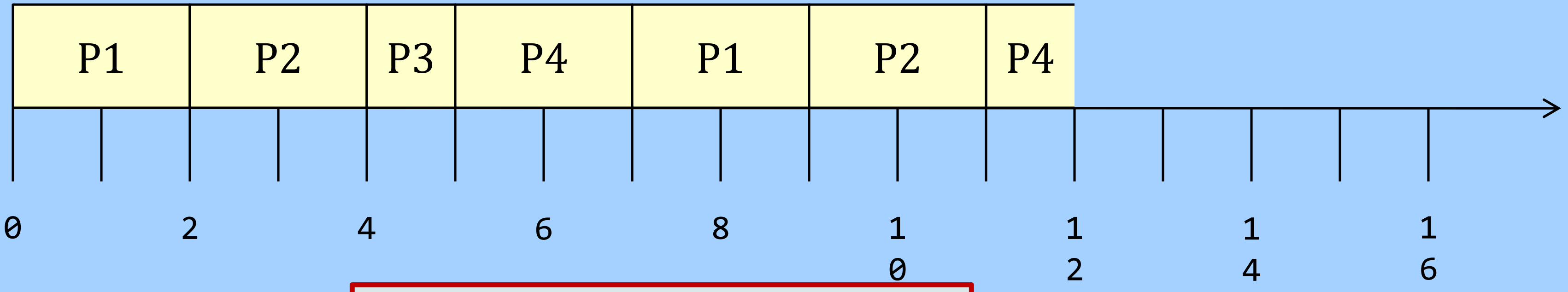
P1’s quantum is 0;
P2 is selected!



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	2
P3	4	1	0
P4	5	4	2

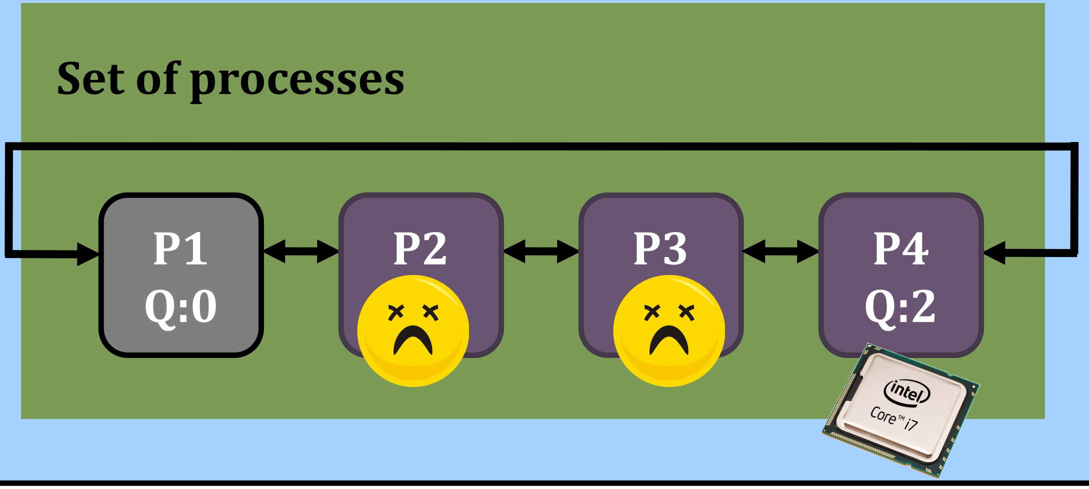
Animation; don't print

Round-robin



Time = 11

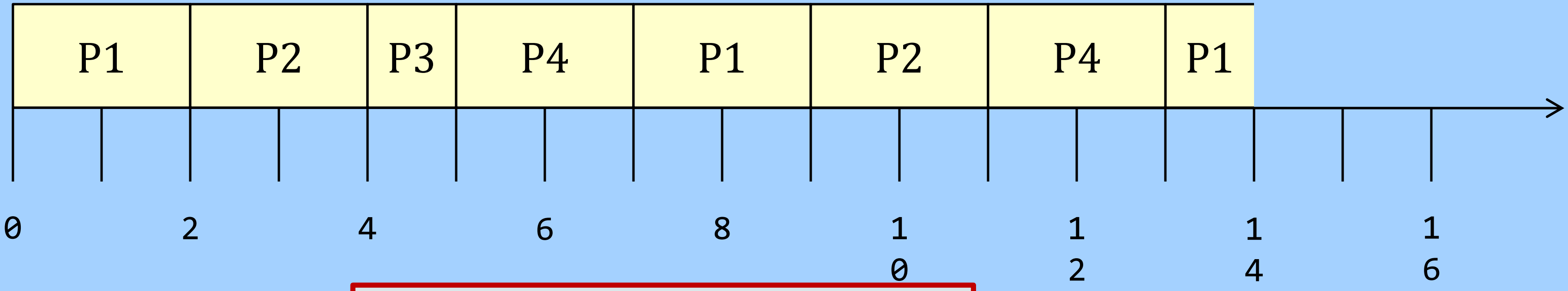
P1's quantum is 0;
P4 is selected!



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	2

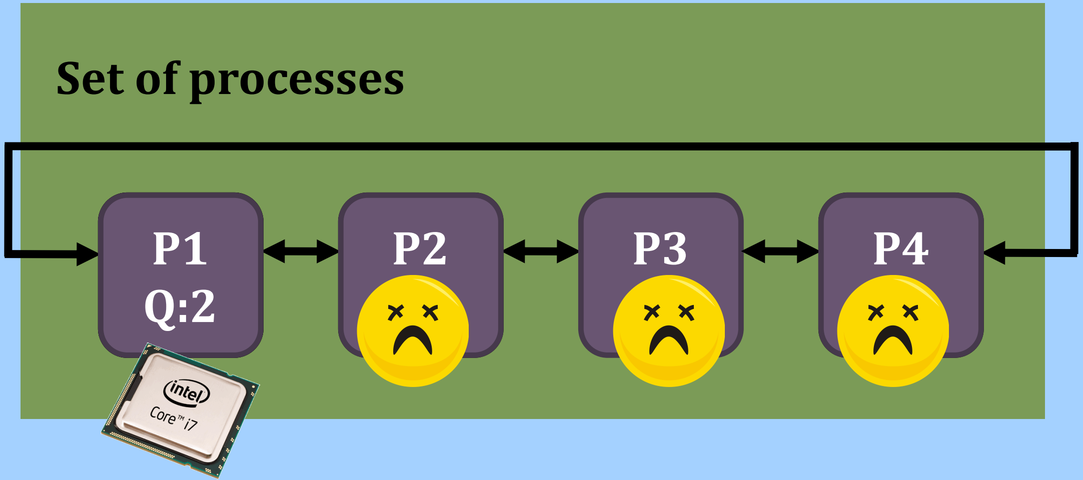
Animation; don't print

Round-robin



Time = 13

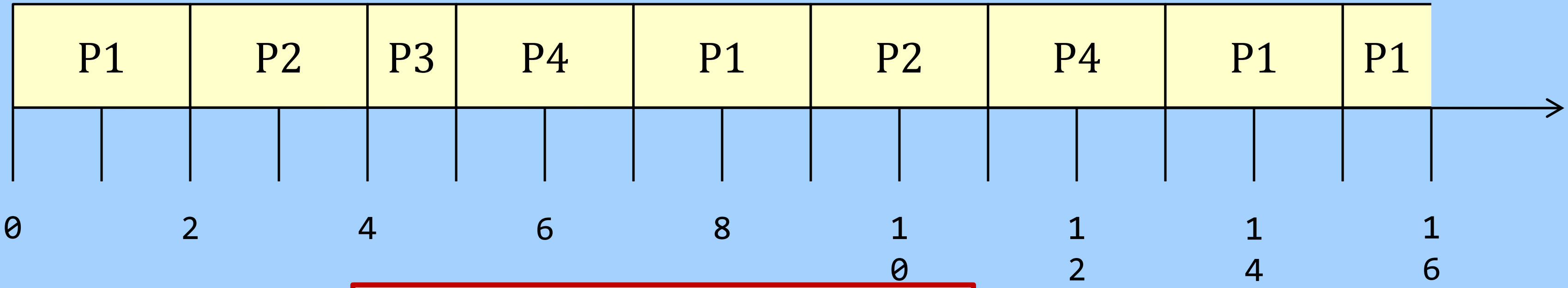
Now, recharge is needed.
P1 is selected.



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	0

Animation; don't print

Round-robin

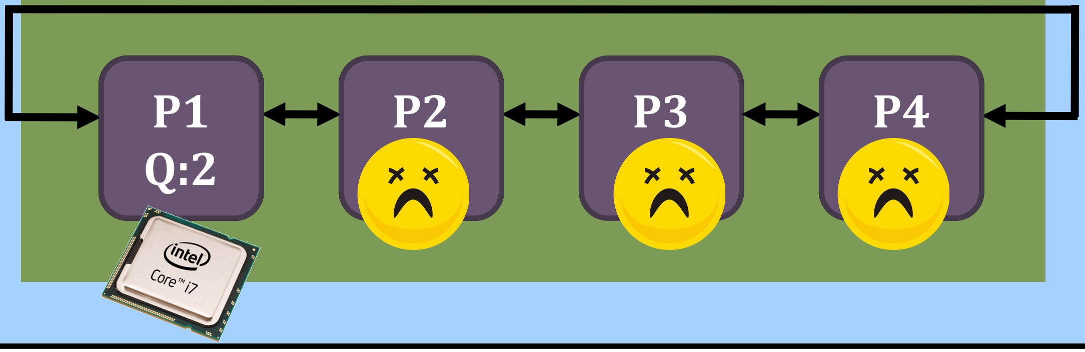

P1 P2 P3 P4 P1 P2 P4 P1 P1

0 2 4 6 8 10 12 14 16

Time = 15

Now, recharge is needed.
P1 is selected.

Set of processes


P1
Q:2

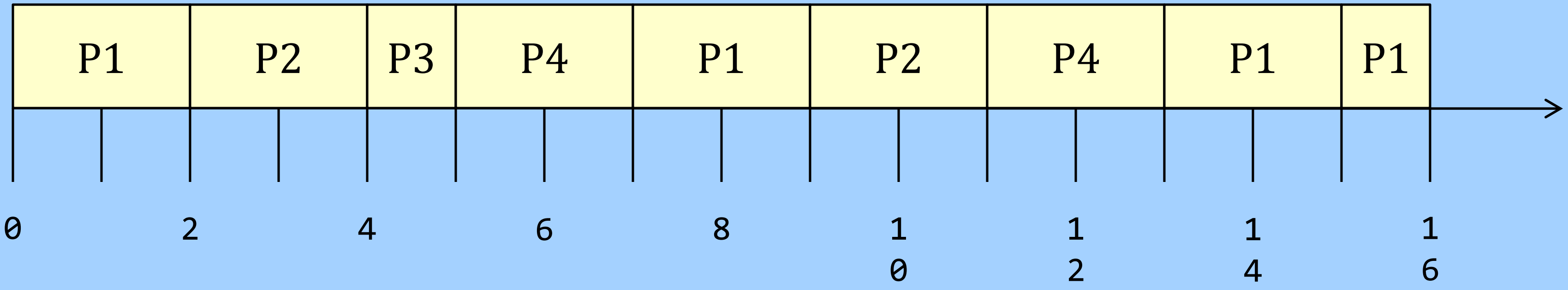
P2

P3

P4

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	0
P3	4	1	0
P4	5	4	0

Round-robin



P1 P2 P3 P4 P1 P2 P4 P1 P1

0 2 4 6 8 10 12 14 16

Waiting time:

P1 = 9; P2 = 5; P3 = 0; P4 = 4;

Average = (9 + 5 + 0 + 4) / 4 = 4.5

Turnaround time:

P1 = 16; P2 = 9; P3 = 1; P4 = 8;

Average = (16 + 9 + 1 + 8) / 4 = 8.5

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

RR VS SJF

	Non-preemptive SJF	Preemptive SJF	RR
Average waiting time	4	3	**4.5 (largest)**
Average turnaround time	8	7	**8.5 (largest)**
# of context switching	3	5	**8 (largest)**

So, the RR algorithm gets all the bad! Why do we still need it?

The responsiveness of the processes is great under the RR algorithm. E.g., you won't feel a job is "frozen" because every job gets the CPU from time to time!

Priority Scheduling

- A task is given a priority (and is usually an integer).
- A scheduler selects the next process based on the priority
- A higher priority process + RR = priority queue + new process arrival triggers a new selection

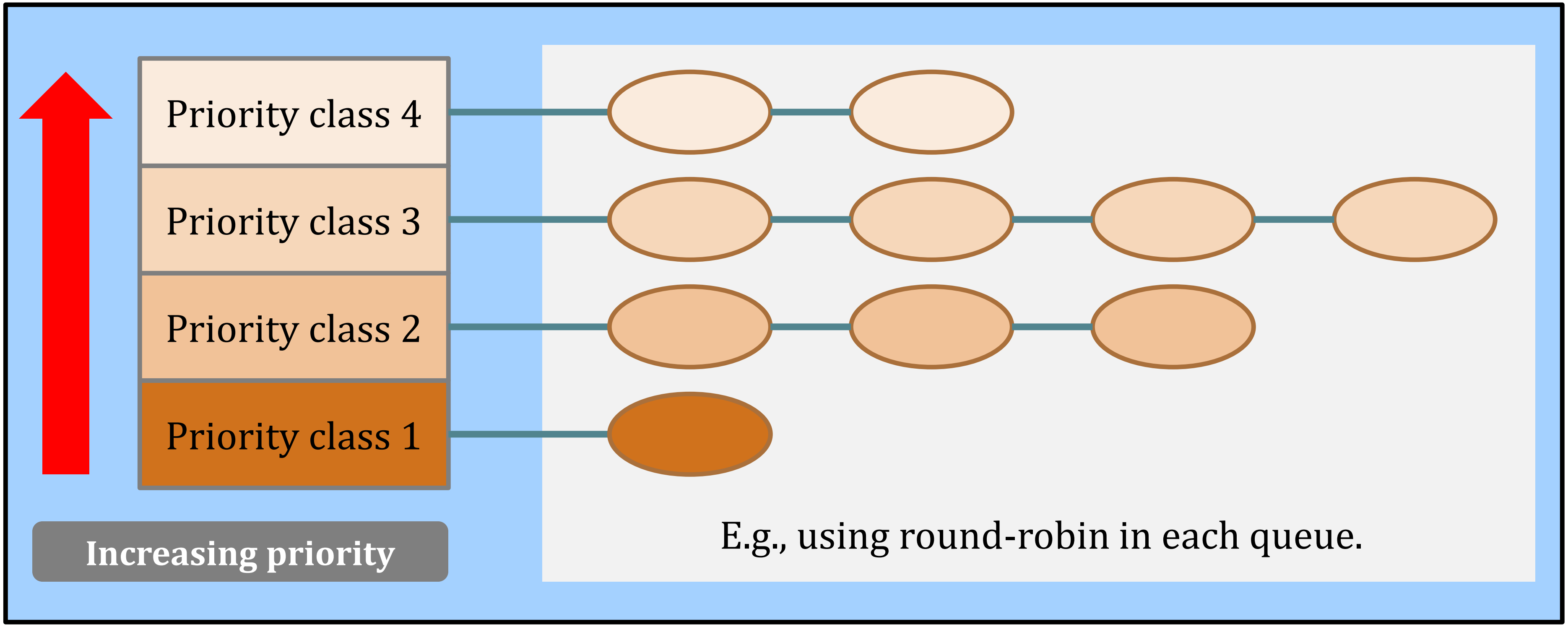
2 Classes	
Static priority	**Dynamic priority**
Every task is given a fixed priority.	Every task is given an initial priority.
The priority is **fixed** throughout the life of the task.	The priority is **changing** throughout the life of the task.

If a task is preempted in the middle

- Note:
  - it has been dequeued
  - Re-enqueue back to the queue
  - Quantum preserved / recharge?
    - Depends
      - Preserved: need more book keeping
      - Recharge: easy (assumed in this course)

Static priority scheduling – an example

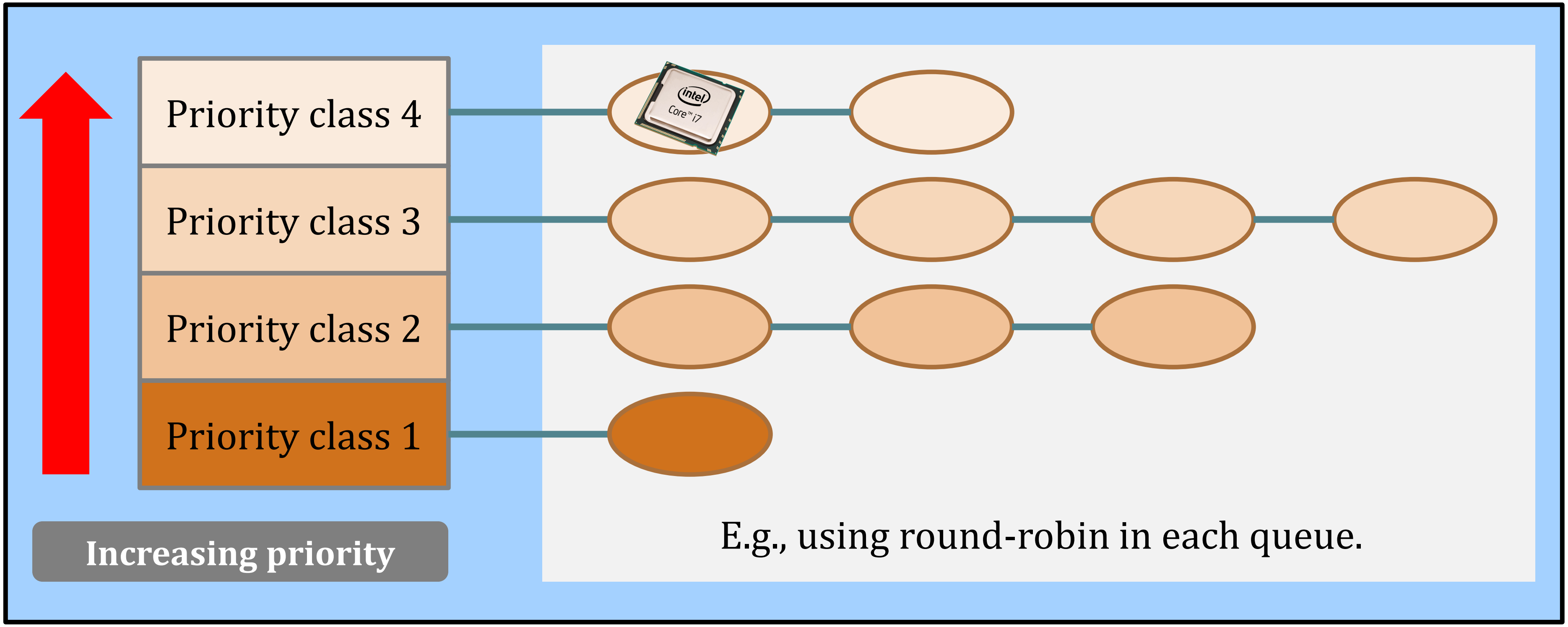
- **Properties**: process is assigned a fix priority when they are submitted to the system.
  - E.g., Linux kernel 2.6 has 100 priority classes, [0-99].



E.g., using round-robin in each queue.

Static priority scheduling – an example

- The highest priority class will be selected.
  - The tasks are usually short-lived, but important;
    - To prevent high-priority tasks from running indefinitely.



Priority class 4

Priority class 3

Priority class 2

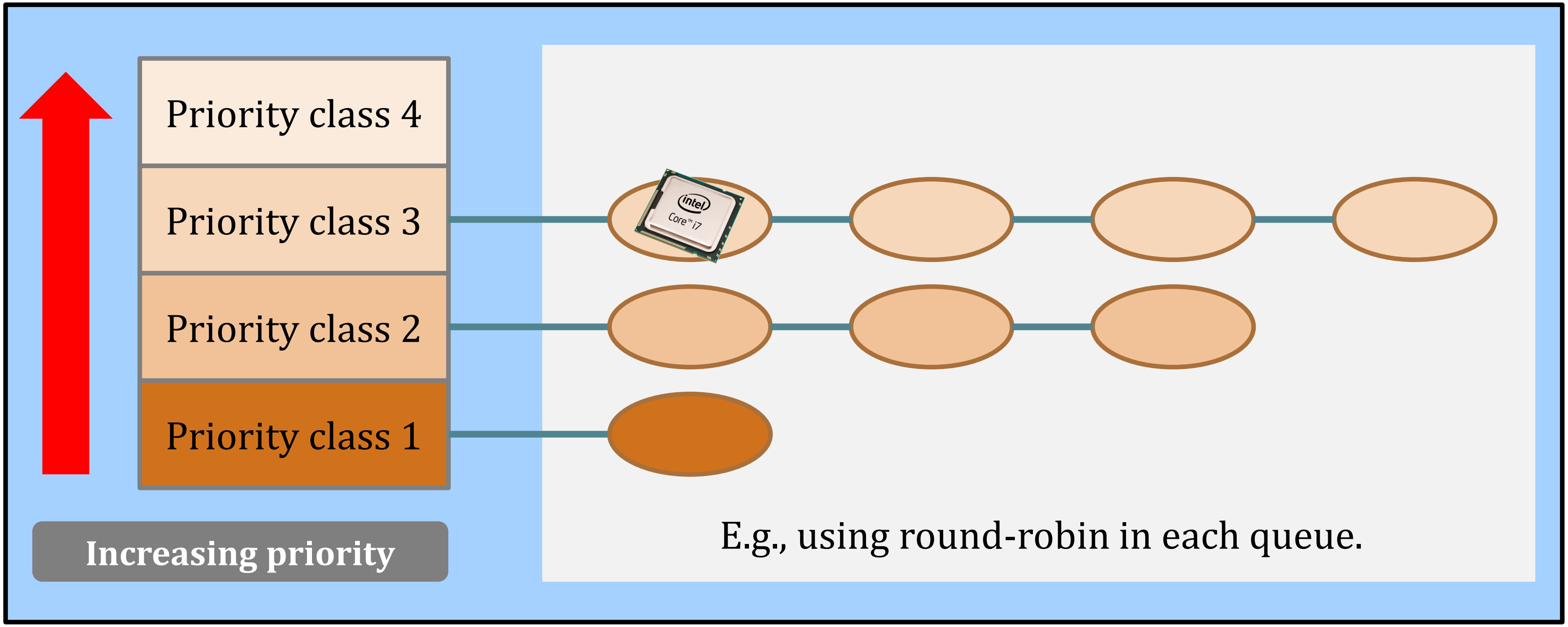
Priority class 1

Increasing priority

E.g., using round-robin in each queue.

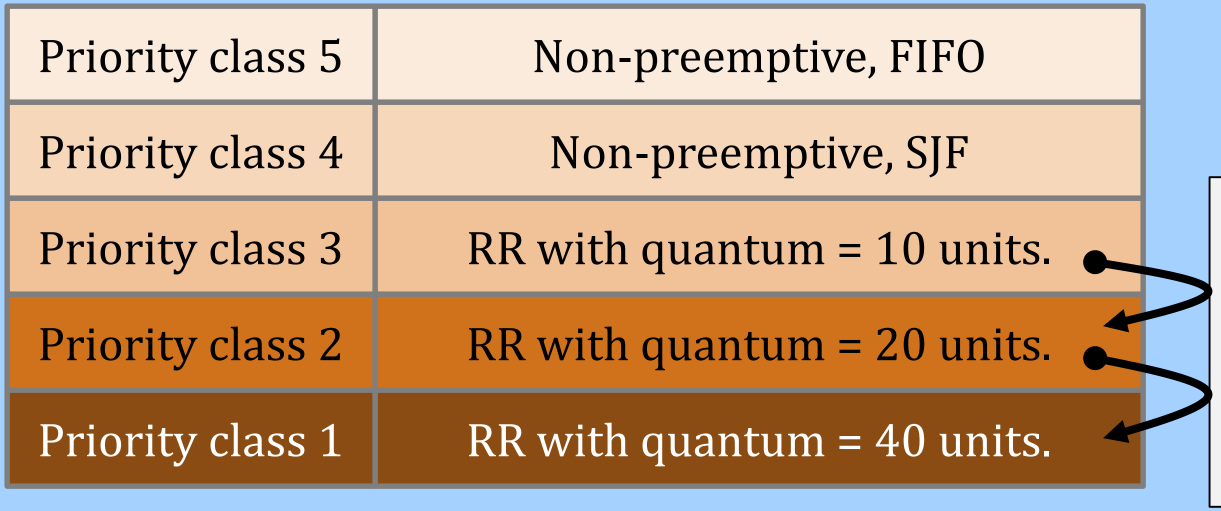
Static priority scheduling – an example

- Lower priority classes will be scheduled only when the upper priority classes has no tasks.



Multiple queue priority scheduling

- **Definitions.**
  - It is still a priority scheduler.
  - But, at each priority class, **different schedulers** may be deployed.
  - The priority can be a mix of static and dynamic.



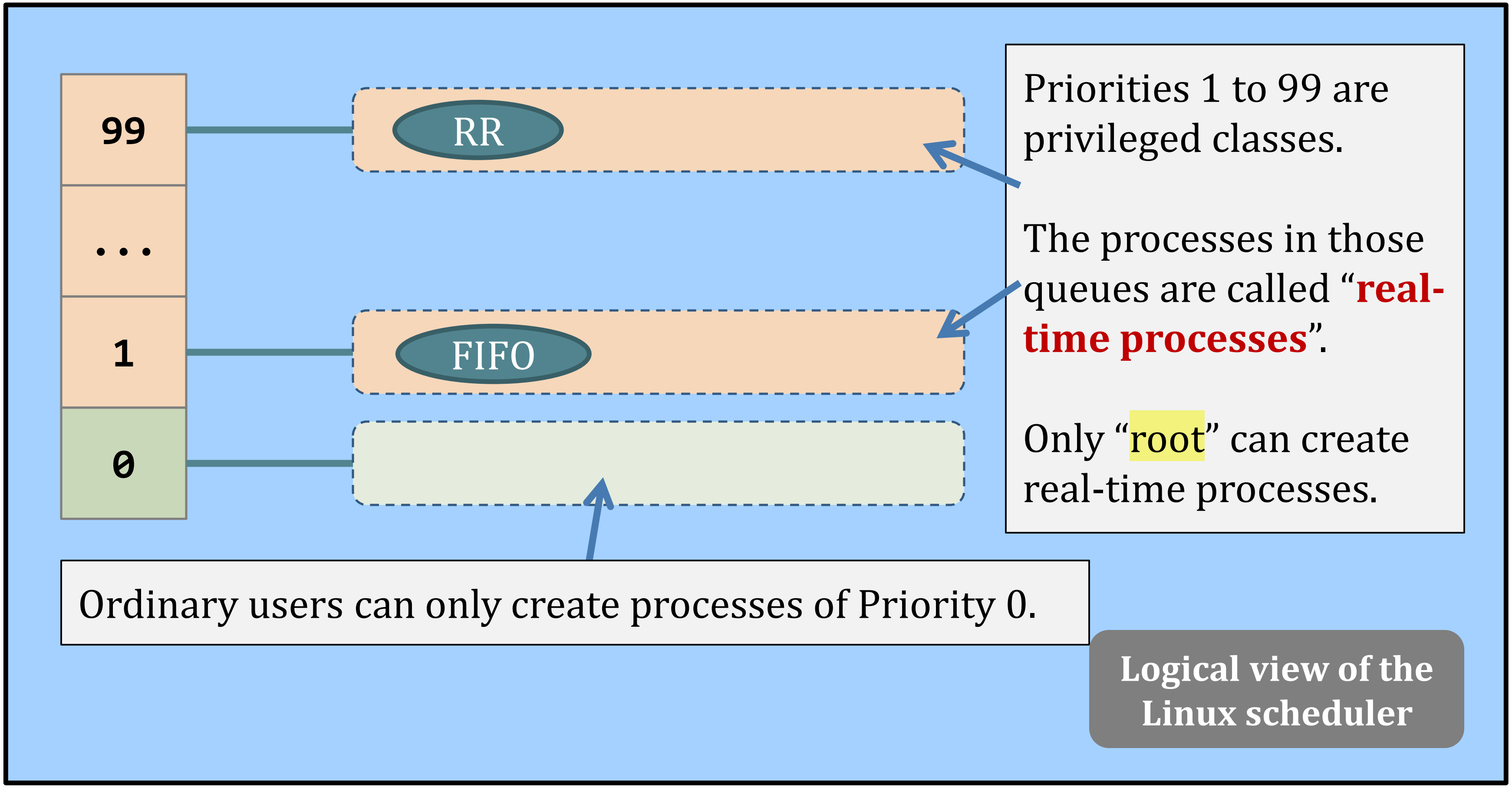
Priority class 5	Non-preemptive, FIFO
Priority class 4	Non-preemptive, SJF
Priority class 3	RR with quantum = 10 units.
Priority class 2	RR with quantum = 20 units.
Priority class 1	RR with quantum = 40 units.

Just an example.

A process drops to a lower priority class after it has used up its quantum and has the quantum recharged.

Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



Priorities 1 to 99 are privileged classes.

The processes in those queues are called "**real-time processes**".

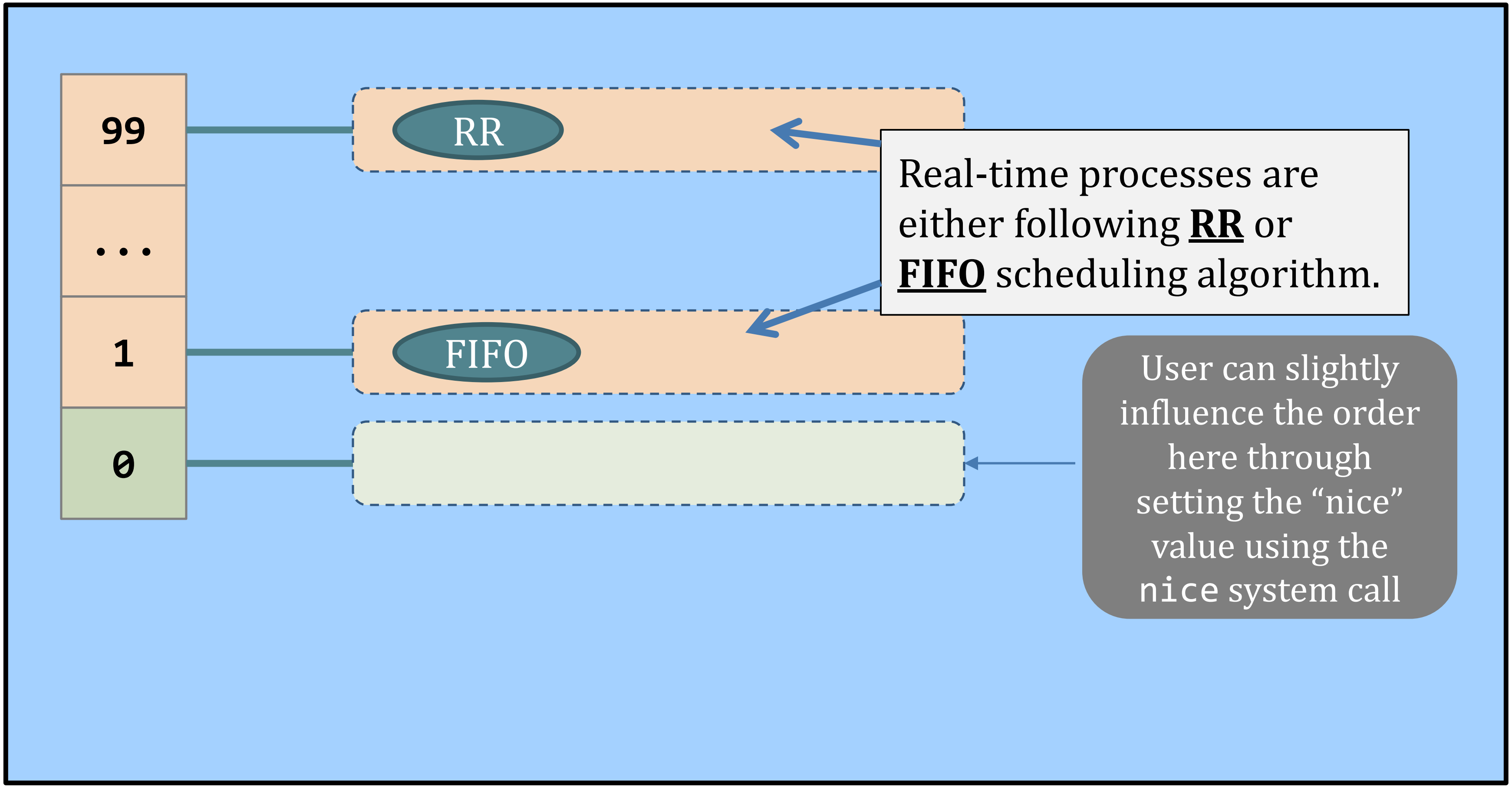
Only "root" can create real-time processes.

Ordinary users can only create processes of Priority 0.

Logical view of the Linux scheduler

Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.

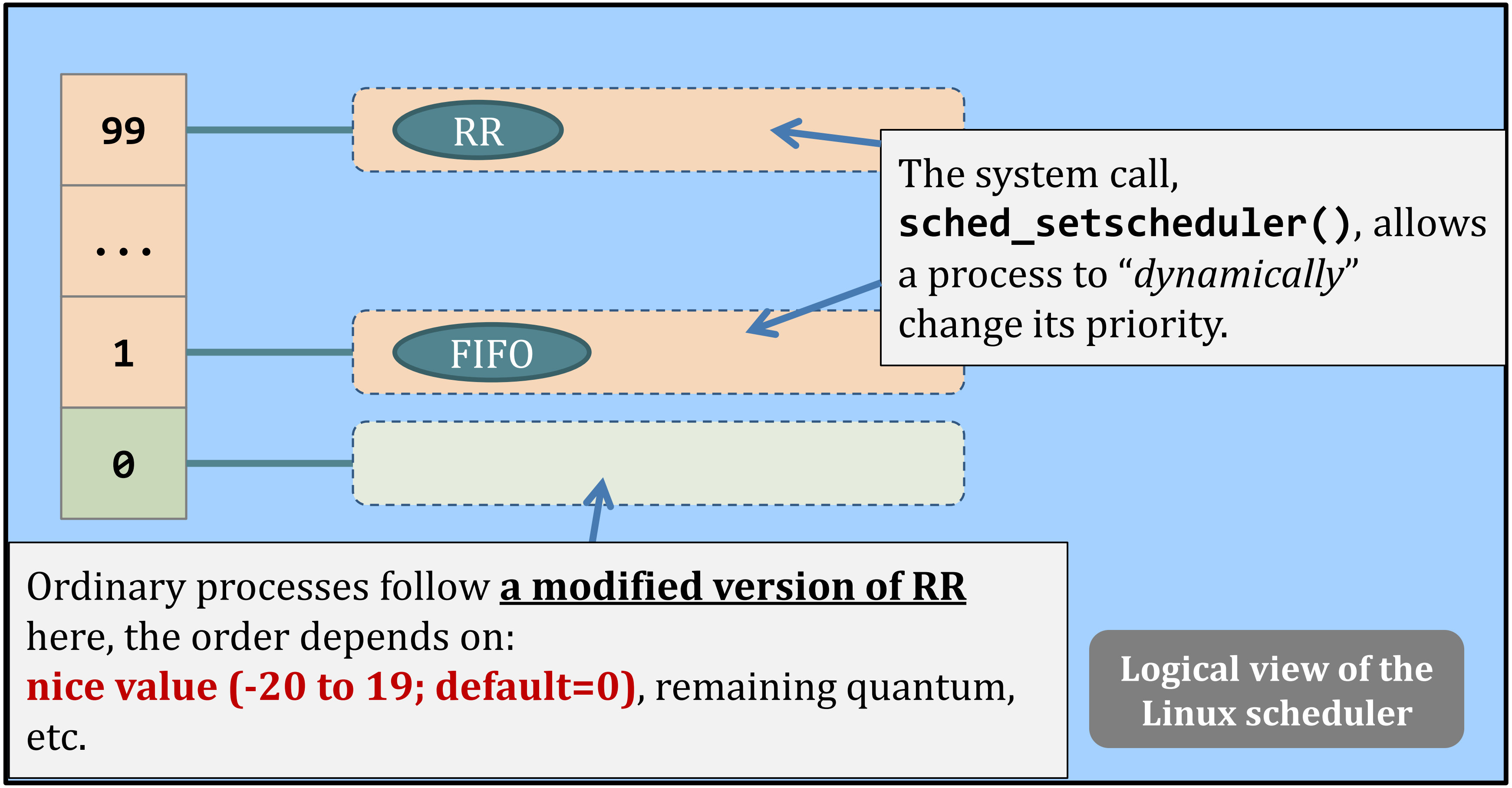


Real-time processes are either following **RR** or **FIFO** scheduling algorithm.

User can slightly influence the order here through setting the “nice” value using the `nice` system call

Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



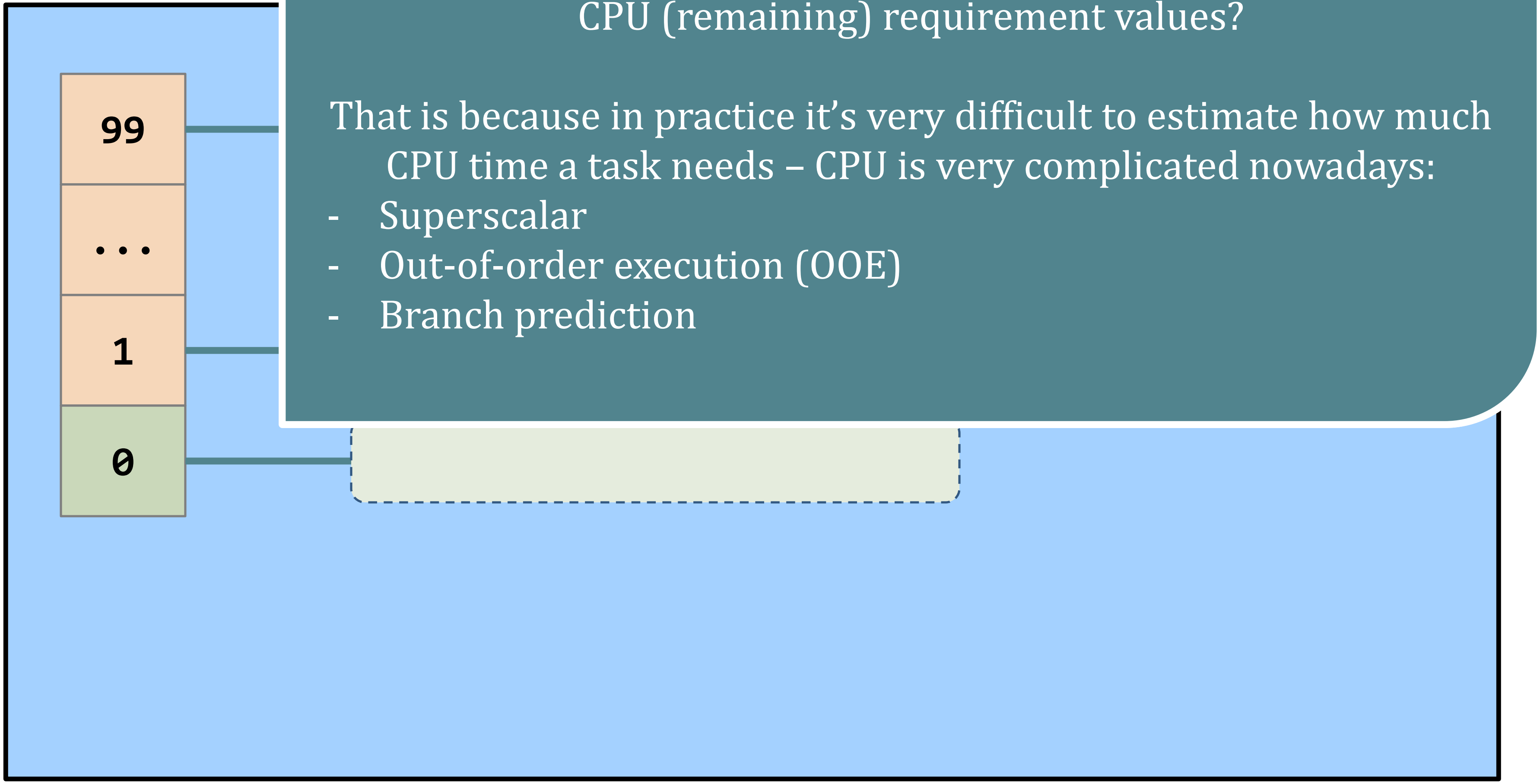
The system call, **`sched_setscheduler()`**, allows a process to "*dynamically*" change its priority.

Ordinary processes follow **<u>a modified version of RR</u>** here, the order depends on: **nice value (-20 to 19; default=0)**, remaining quantum, etc.

Logical view of the Linux scheduler

Multiple queue priority scheduling

- **Real example, the Linux Scheduler**
  - A multi



Do you notice that real (Linux) scheduler does not rely on task's CPU (remaining) requirement values?

That is because in practice it's very difficult to estimate how much CPU time a task needs – CPU is very complicated nowadays:

- Superscalar
- Out-of-order execution (OOE)
- Branch prediction

Recall: Four Fundamental OS Concepts

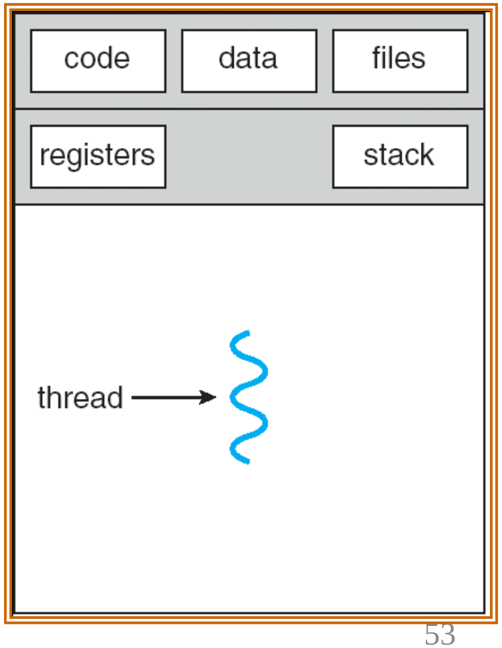
- Thread
  - Single unique execution context: fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with translation)
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Dual mode operation / Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

Recall: Process (What we knew so far)

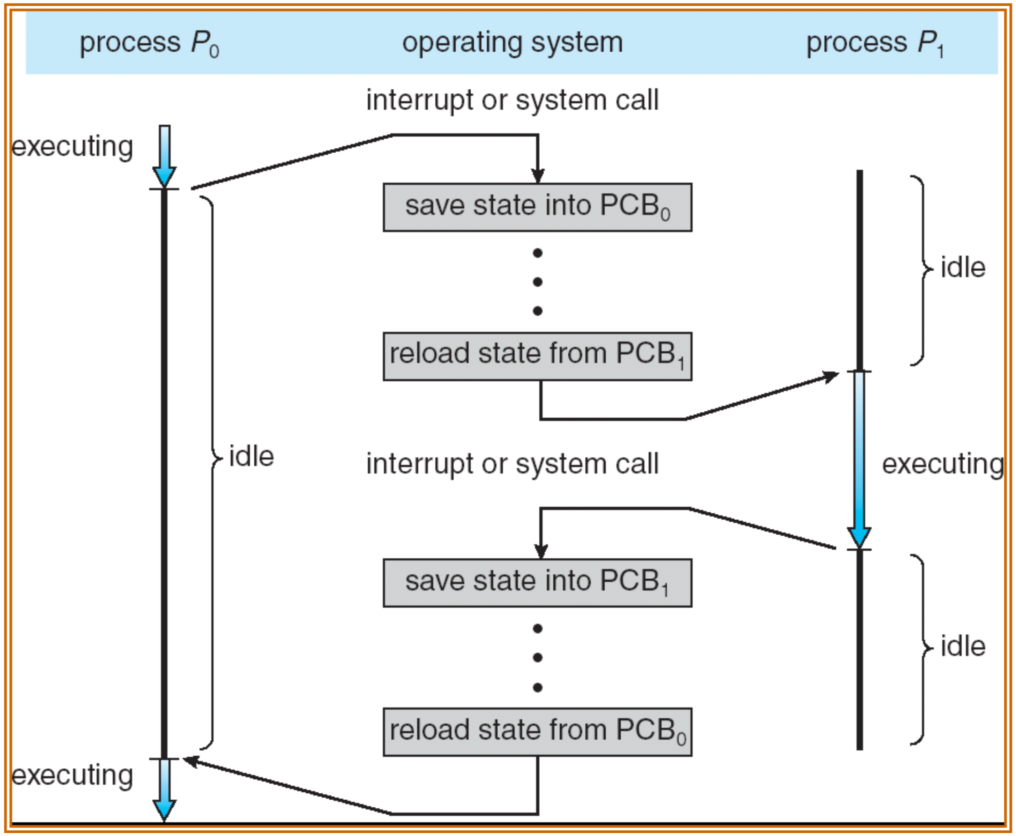
- Process: An instance of an executing program
  - An address space,
  - One or more threads of control

- ***Heavyweight*** process: a process has a single thread of control

- Two properties of heavyweight process
  - Sequential program execution stream
    - Active part
  - Protected resource
    - Passive part



Recall: CPU Switch From Process A to Process B



- This is also called a "context switch"
- Code executed in kernel above is *overhead*

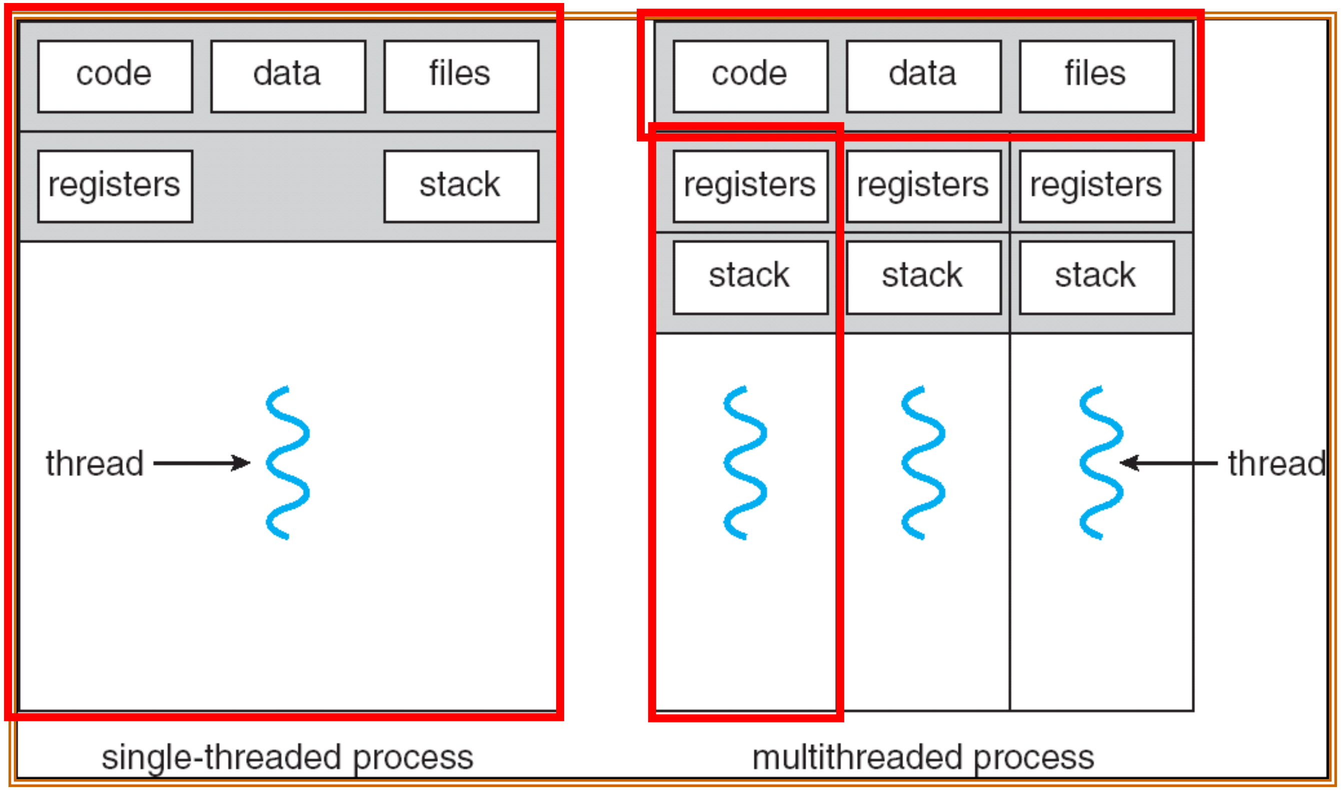
Modern Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads

- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada …

- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (*concurrency, parallelism*)
  - Separate from the "address space" (protection)
  - Heavyweight Process ≡ Process with one thread

Single and Multithreaded Processes



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system

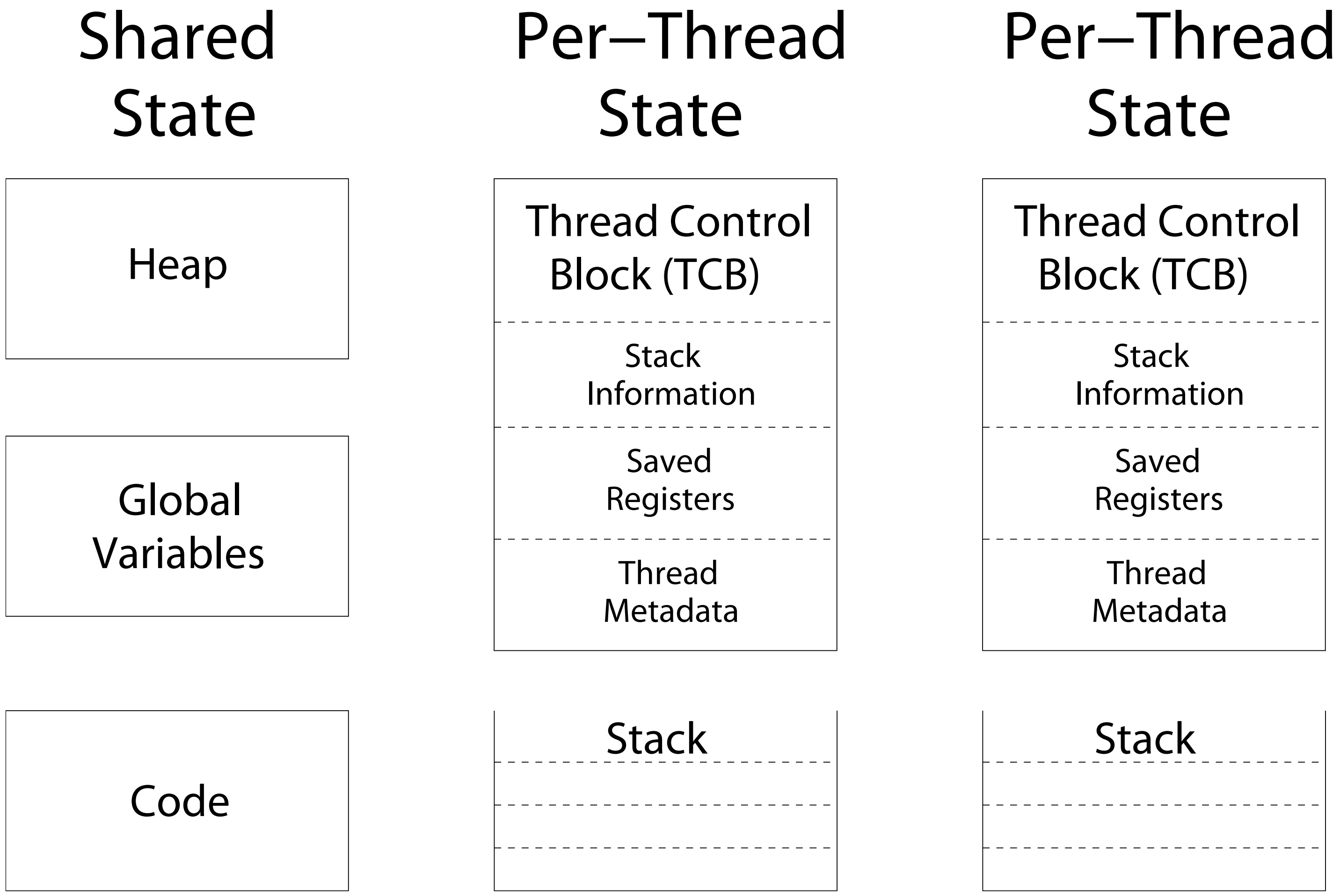
Threads in a Process

- Threads are useful at user-level: parallelism, hide I/O latency, interactivity (support for threads)
- Option A (early Java): user-level library, one multi-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Linux/Unix variants): many single-threaded processes
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processes to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode

Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

Shared vs. Per-Thread State



Shared State

Heap

Global Variables

Code

Per–Thread State

Thread Control Block (TCB)

Stack Information

Saved Registers

Thread Metadata

Stack

Per–Thread State

Thread Control Block (TCB)

Stack Information

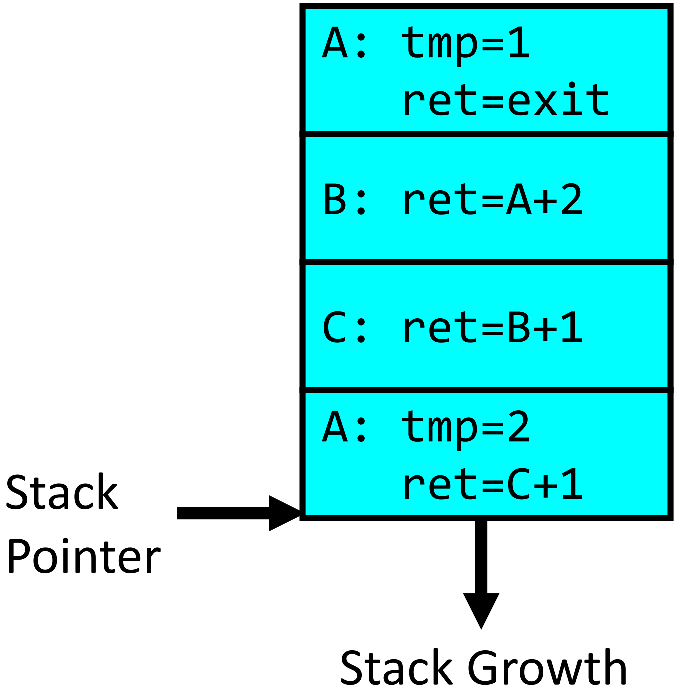
Saved Registers

Thread Metadata

Stack

Execution Stack Example

```
       A(int tmp) {
  A:     if (tmp<2)
A+1:       B();
A+2:     printf(tmp);
       }
       B() {
  B:     C();
B+1:   }
       C() {
  C:     A(2);
C+1:   }
       A(1);
exit:
```



A: tmp=1
ret=exit

B: ret=A+2

C: ret=B+1

A: tmp=2
ret=C+1

Stack Pointer

Stack Growth

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

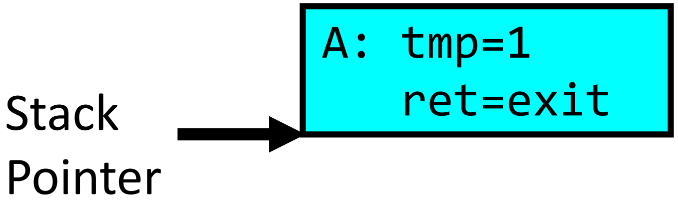
Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
   A:   if (tmp<2)
 A+1:     B();
 A+2:   printf(tmp);
      }
      B() {
   B:   C();
 B+1: }
      C() {
   C:   A(2);
 C+1: }
      A(1);
exit:
```



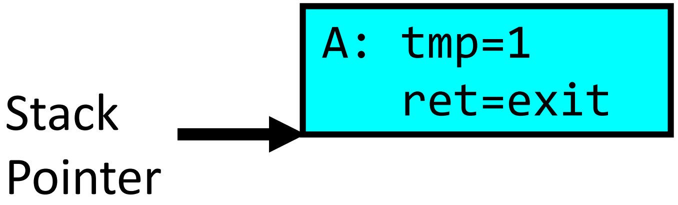
Stack Pointer

```
A:  tmp=1
    ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
       A(int tmp) {
A:       if (tmp<2)
A+1:       B();
A+2:     printf(tmp);
       }
       B() {
B:       C();
B+1:   }
       C() {
C:       A(2);
C+1:   }
       A(1);
exit:
```



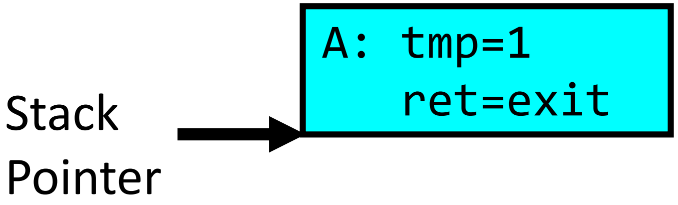
Stack Pointer

A: tmp=1
ret=exit

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

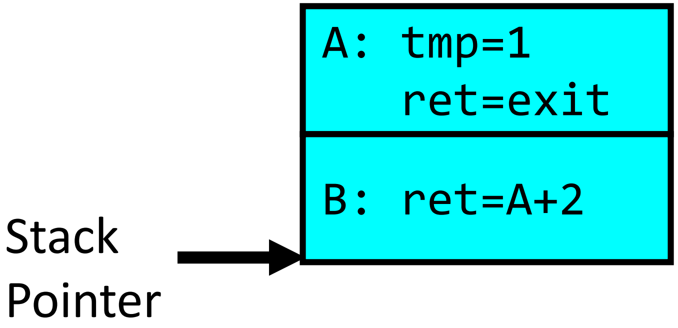
```
      A(int tmp) {
  A:    if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
  B:    C();
B+1:  }
      C() {
  C:    A(2);
C+1:  }
      A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

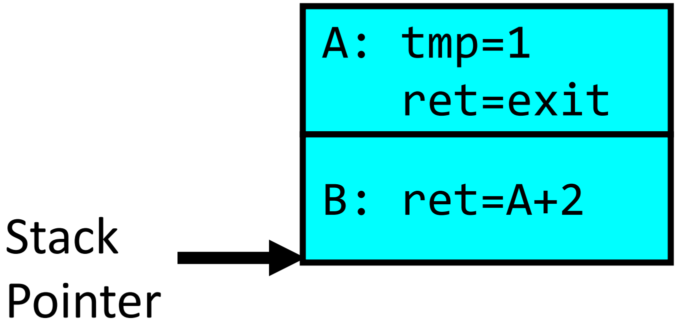
```
      A(int tmp) {
   A:   if (tmp<2)
 A+1:     B();
 A+2:   printf(tmp);
      }
      B() {
   B:   C();
 B+1: }
      C() {
   C:   A(2);
 C+1: }
      A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
   A:   if (tmp<2)
 A+1:     B();
 A+2:   printf(tmp);
      }
      B() {
   B:   C();
 B+1: }
      C() {
   C:   A(2);
 C+1: }
      A(1);
exit:
```



A: tmp=1
ret=exit

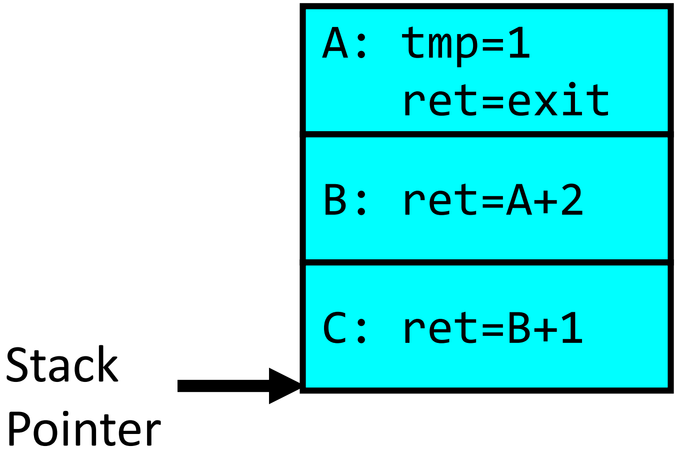
B: ret=A+2

Stack Pointer

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



A: tmp=1
ret=exit

B: ret=A+2

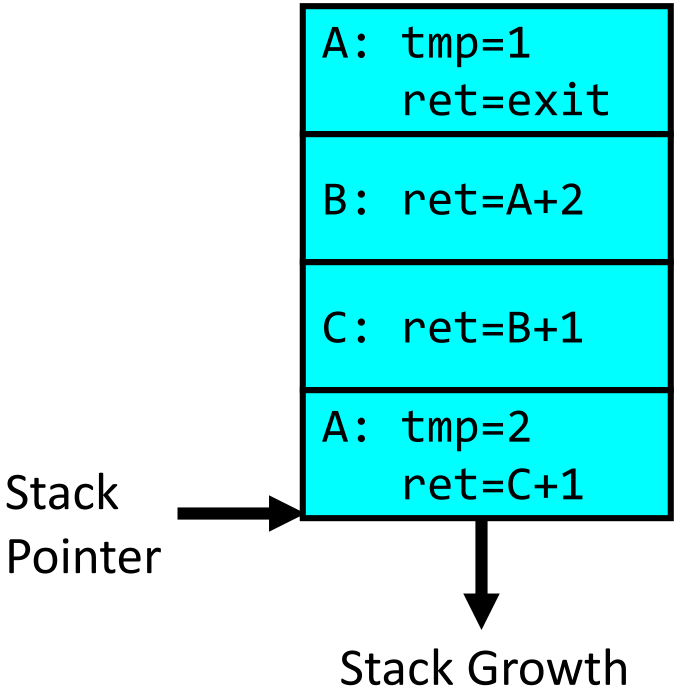
C: ret=B+1

Stack Pointer

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

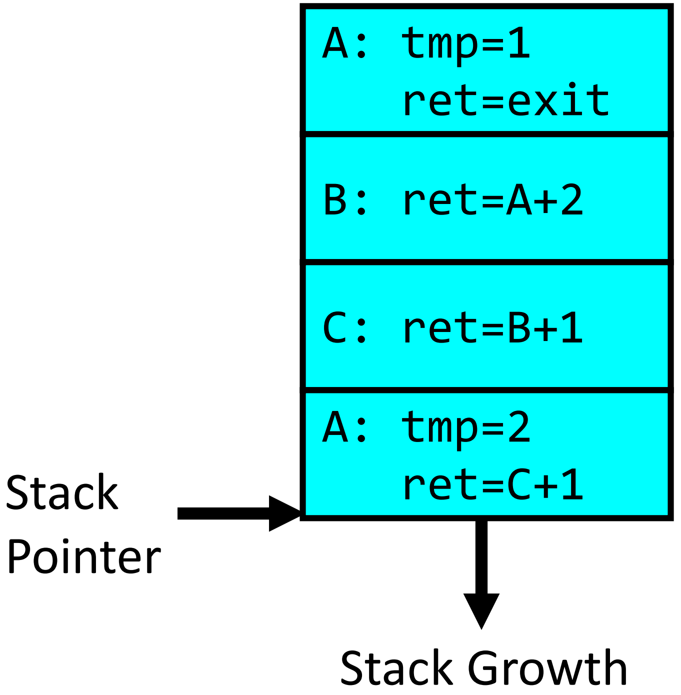
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
       A(int tmp) {
   A:    if (tmp<2)
 A+1:      B();
 A+2:    printf(tmp);
       }
       B() {
   B:    C();
 B+1:  }
       C() {
   C:    A(2);
 C+1:  }
       A(1);
exit:
```



A: tmp=1
ret=exit

B: ret=A+2

C: ret=B+1

A: tmp=2
ret=C+1

Stack Pointer

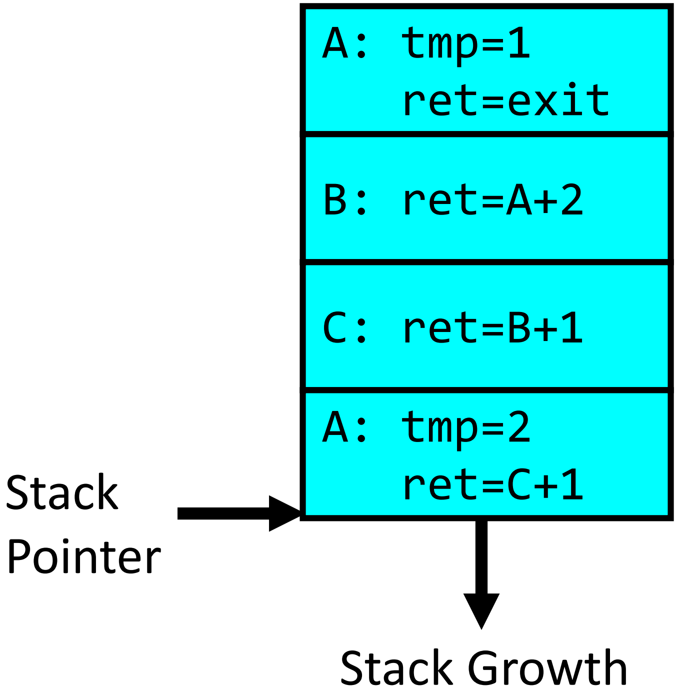
Stack Growth

Output: 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



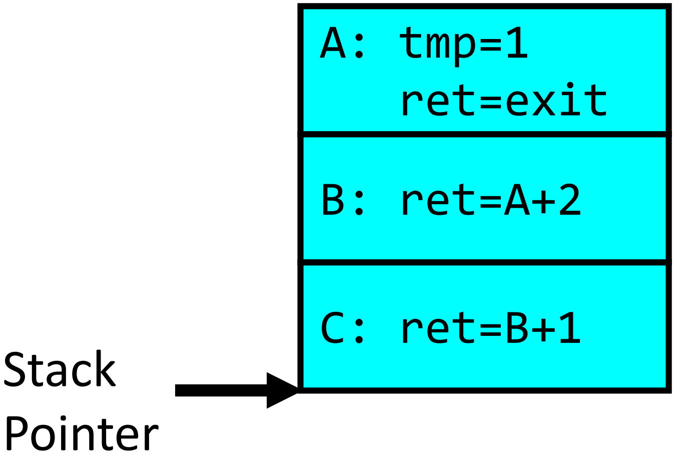
```
A:  tmp=1
    ret=exit
B:  ret=A+2
C:  ret=B+1
A:  tmp=2
    ret=C+1
```

Output: 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

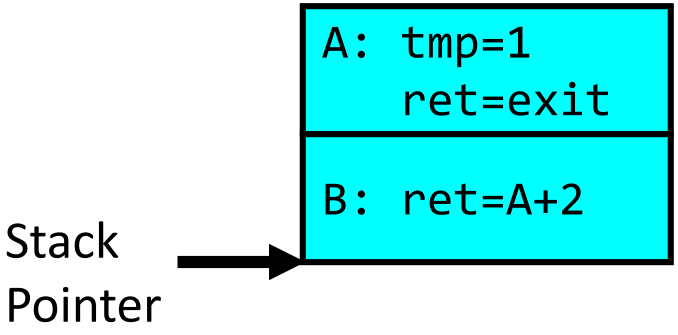


```
Output: 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
   A:   if (tmp<2)
 A+1:     B();
 A+2:   printf(tmp);
      }
      B() {
   B:   C();
 B+1: }
      C() {
   C:   A(2);
 C+1: }
      A(1);
exit:
```



A: tmp=1
ret=exit

B: ret=A+2

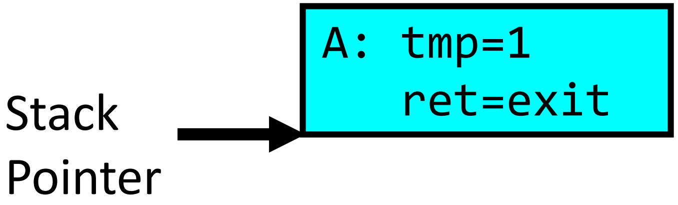
Stack Pointer

```
Output: 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```

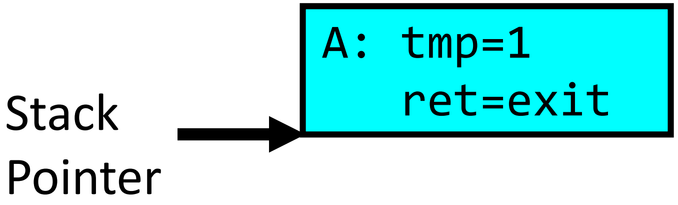


```
Output: 2 1
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



```
Output: 2 1
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

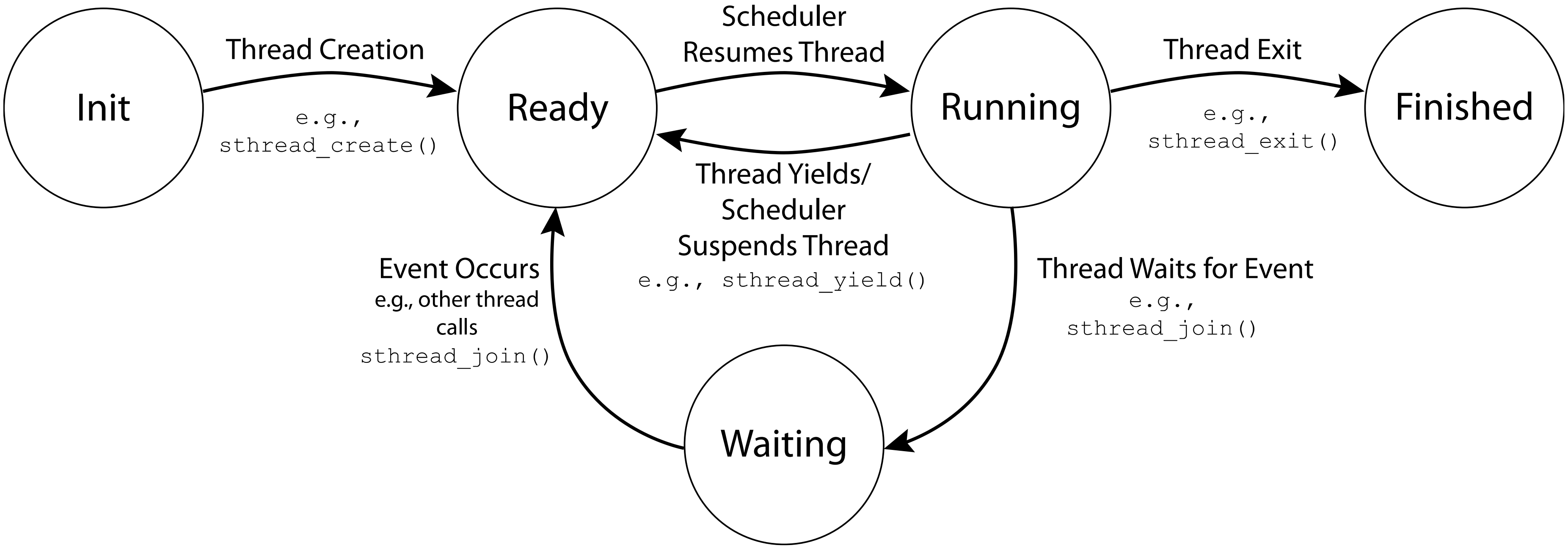
Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

```
Output: 2 1
```

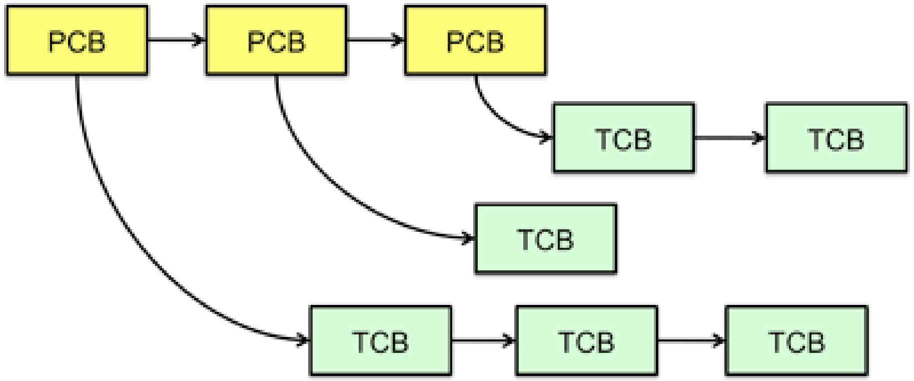
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Thread Lifecycle



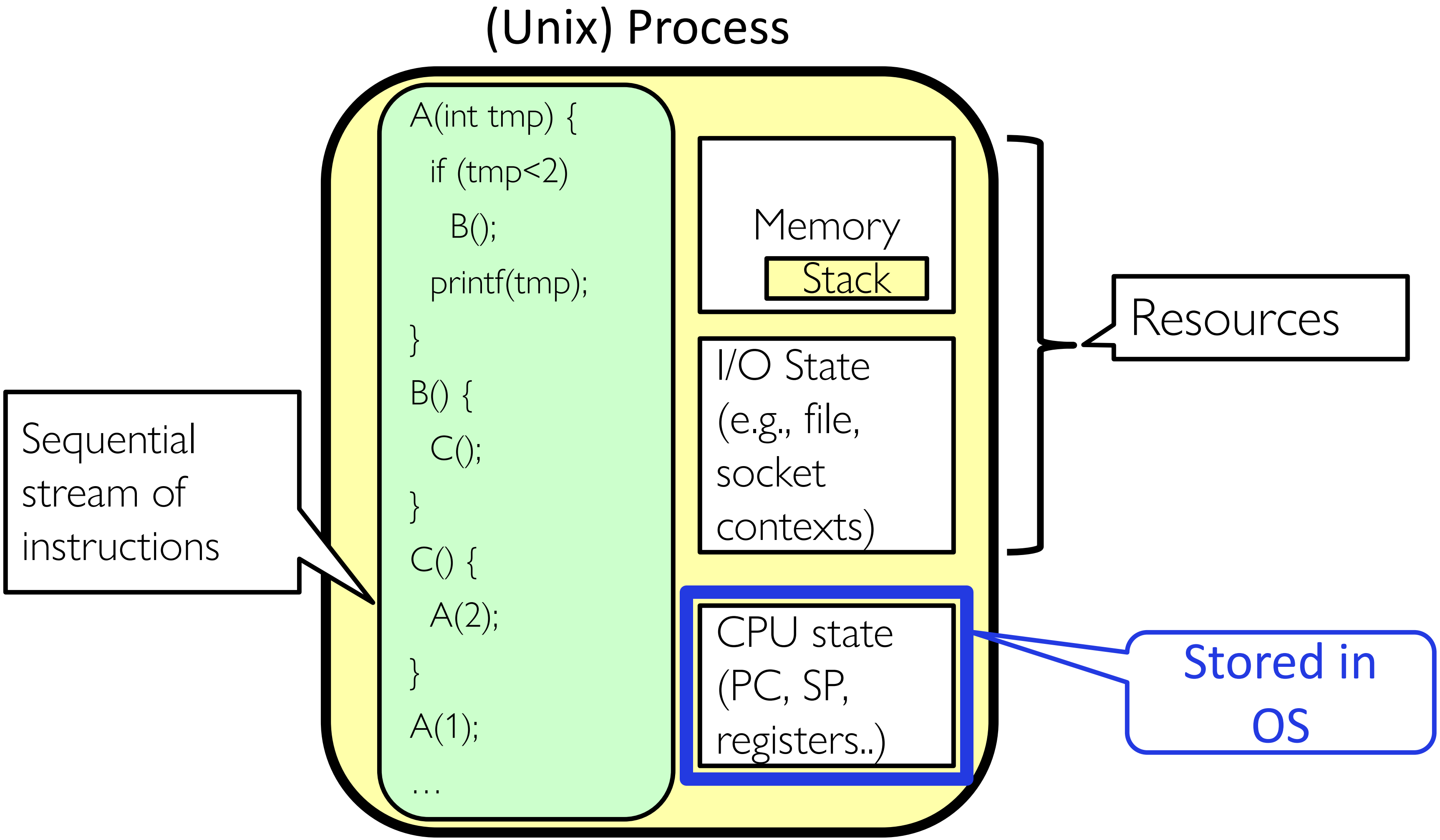
Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):



- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

Putting it Together: Process



(Unix) Process

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
…
```

Memory

Stack

I/O State (e.g., file, socket contexts)

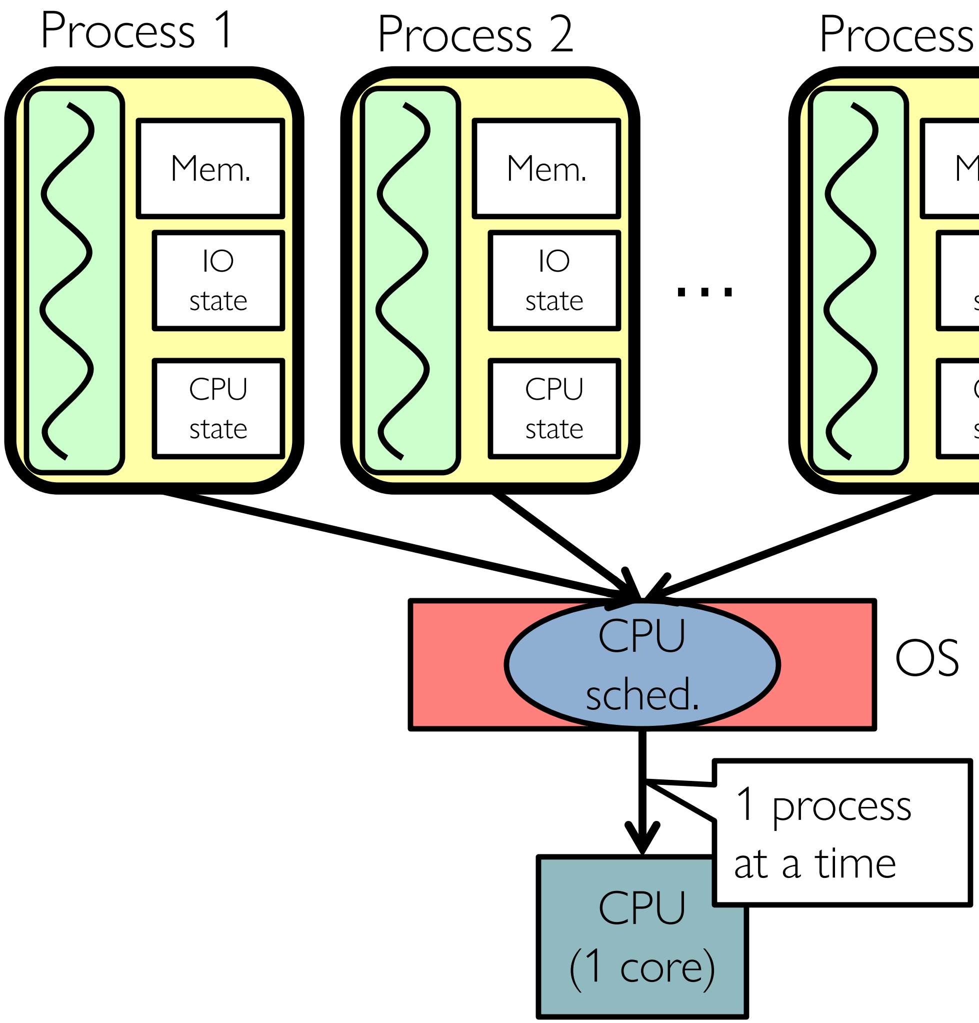
CPU state (PC, SP, registers..)

Resources

Sequential stream of instructions

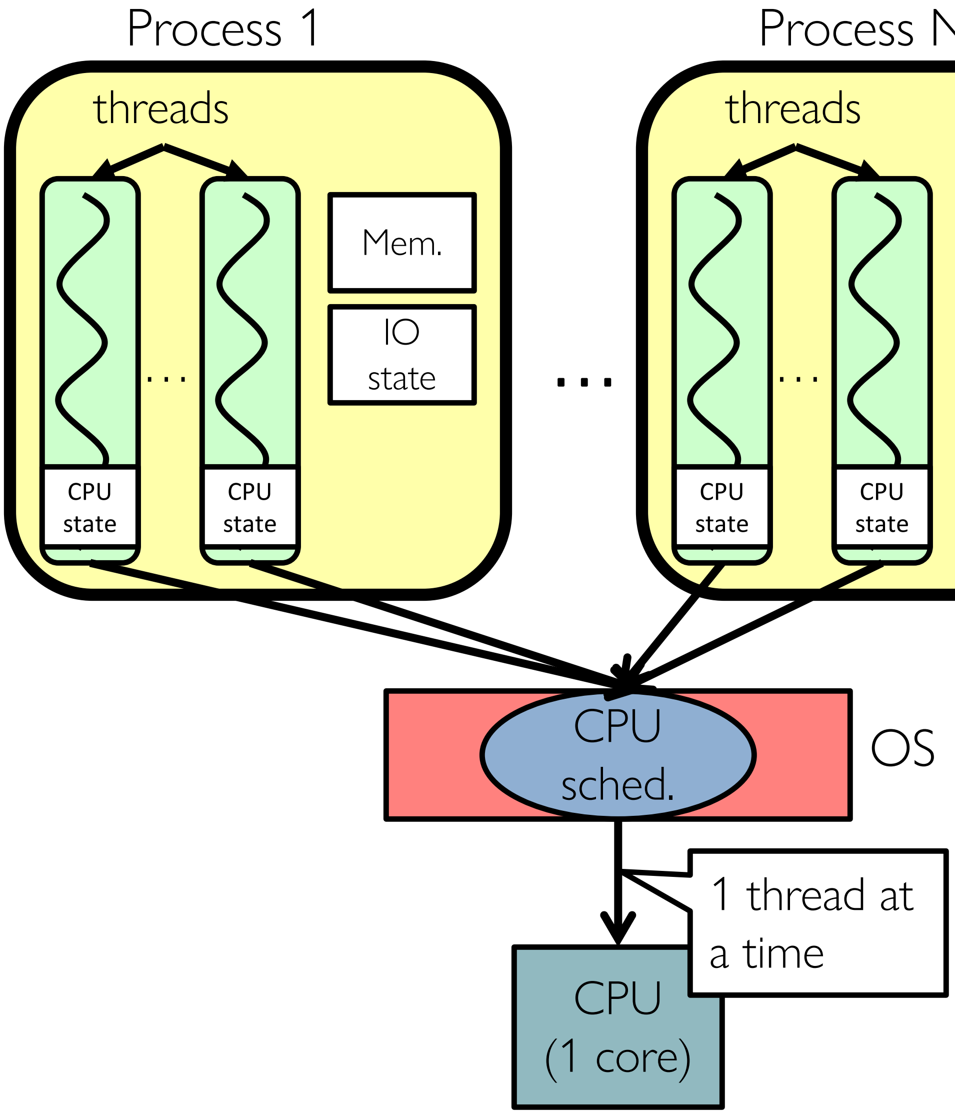
Stored in OS

Putting it Together: Processes



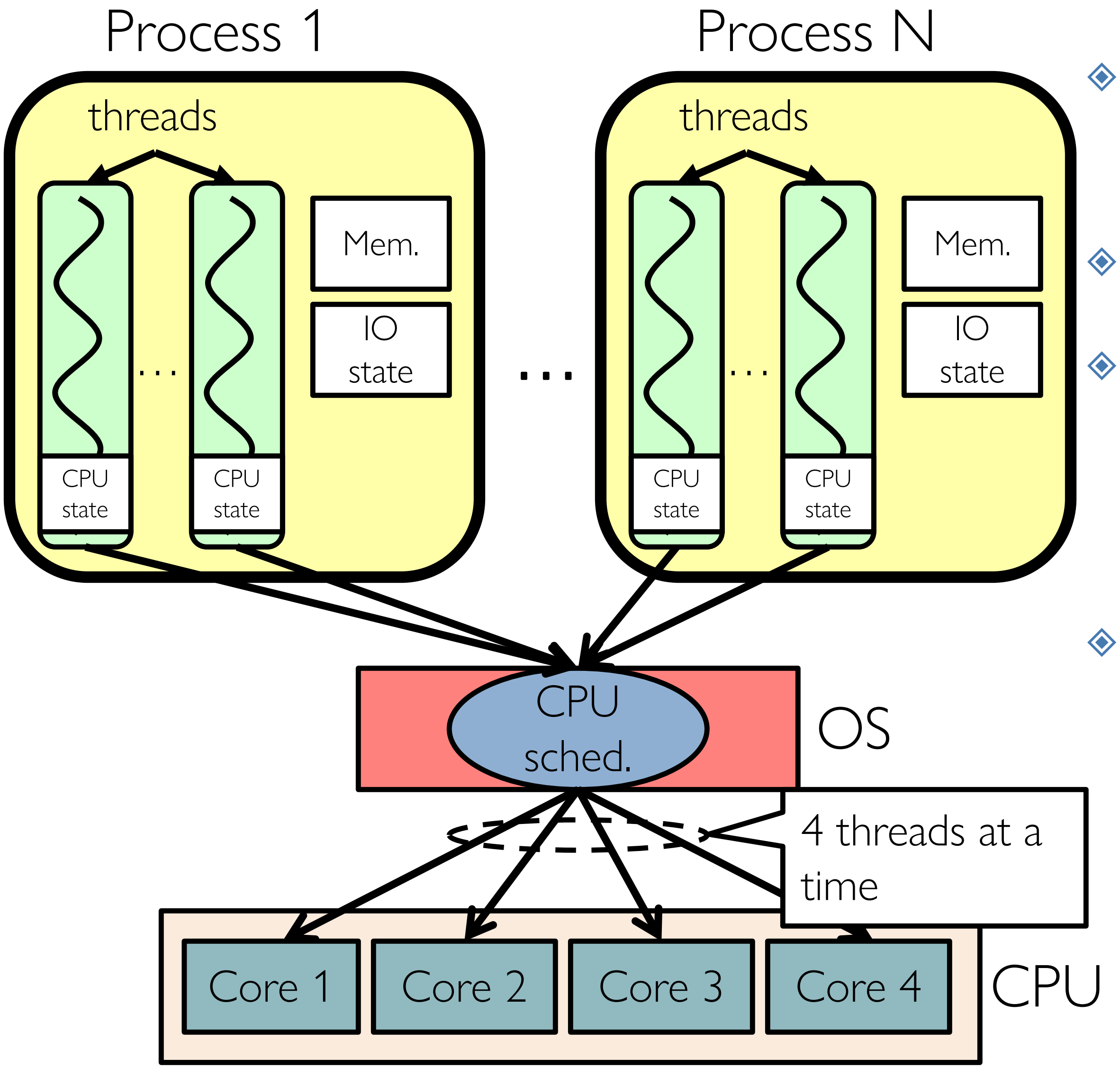
- Switch overhead: high
  - CPU state: *low*
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: *yes*
  - Memory/IO: *yes*
- Sharing overhead: high (involves at least a context switch)

Putting it Together: Threads



- Switch overhead: low *(as only CPU state)*
- Thread creation: low
- Protection
  - CPU: *yes*
  - Memory/IO: no
- Sharing overhead: *low* (thread switch overhead low)

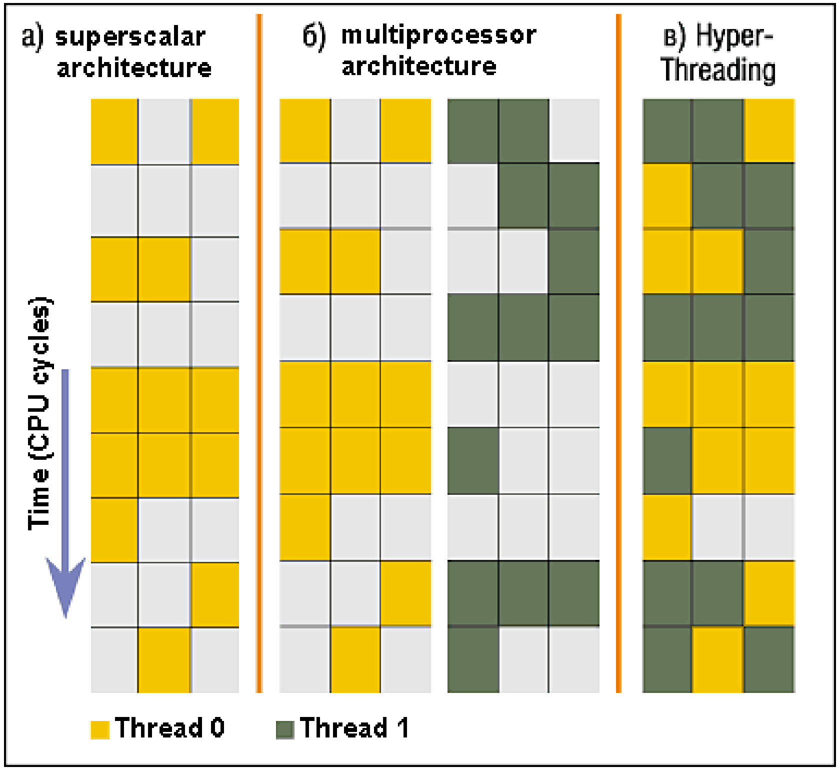
Putting it Together: Multi-Cores



- Switch overhead: *low* (only CPU state)
- Thread creation: *low*
- Protection
  - CPU: *yes*
  - Memory/IO: No
- Sharing overhead: *low* (thread switch overhead low, may not need to switch at all!)

Simultaneous MultiThreading/Hyperthreading

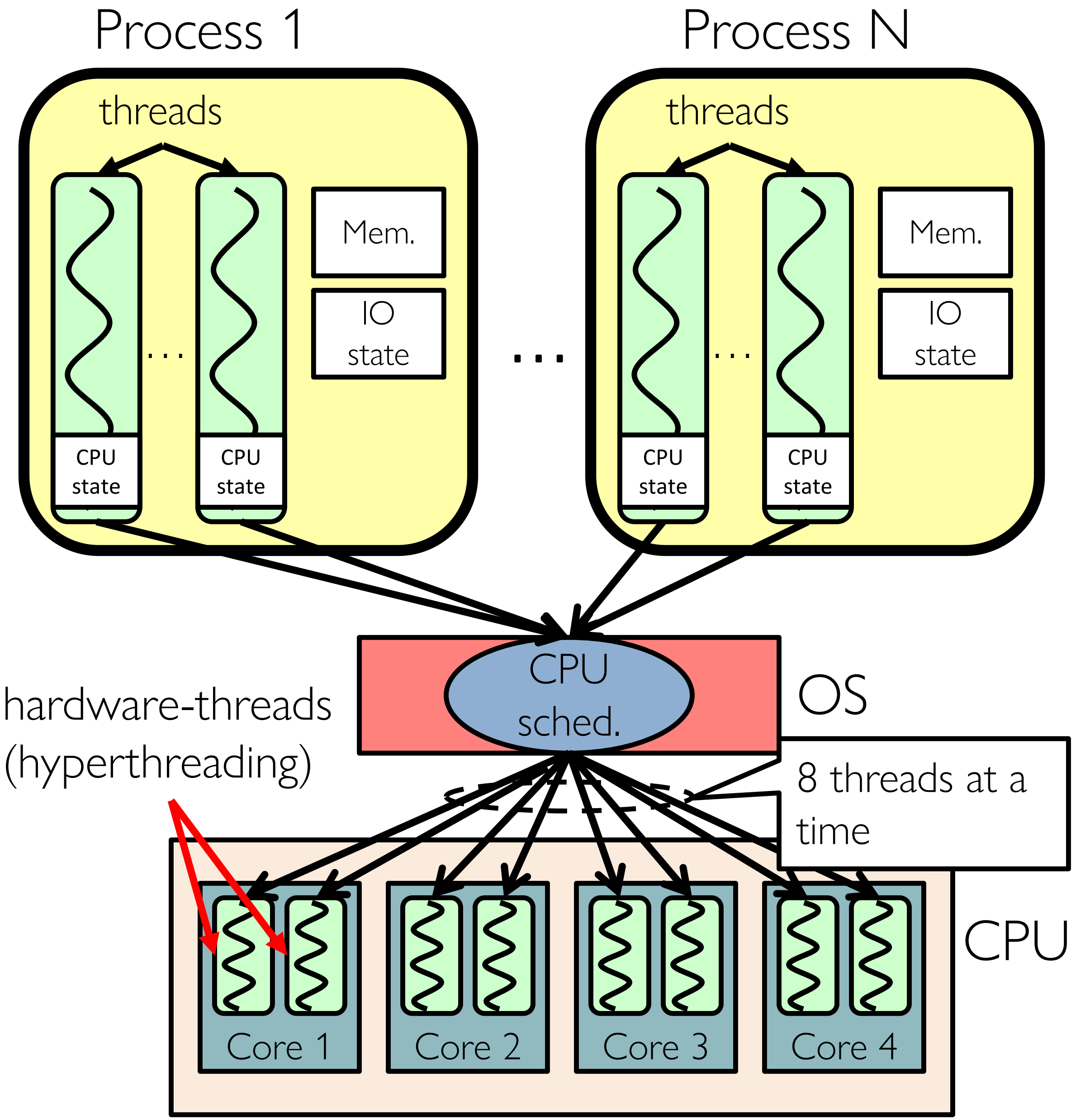
- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



Colored blocks show instructions executed

- Original called "Simultaneous Multithreading"
  - http://www.cs.washington.edu/research/smt/index.html
  - Intel, SPARC, Power (IBM)
  - A virtual core on AWS' EC2 is basically a hyperthread

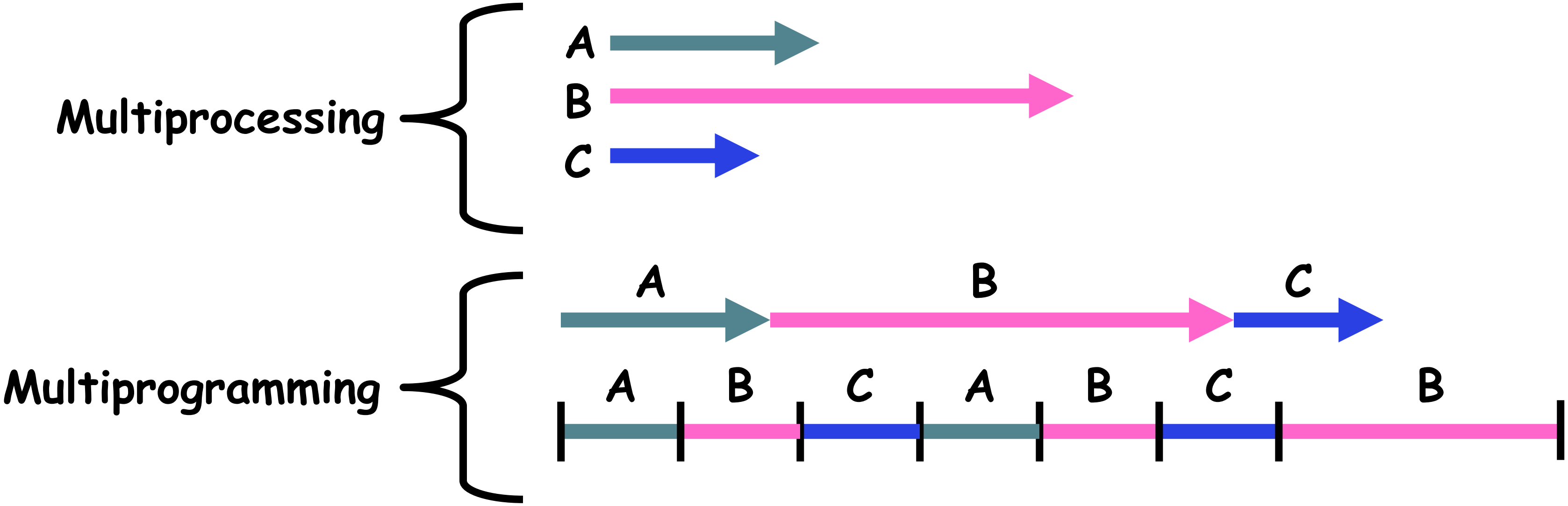
Putting it Together: Hyper-Threading



- Switch overhead between hardware-threads: *very-low* (done in hardware)
- Contention for ALUs/FPUs may hurt performance

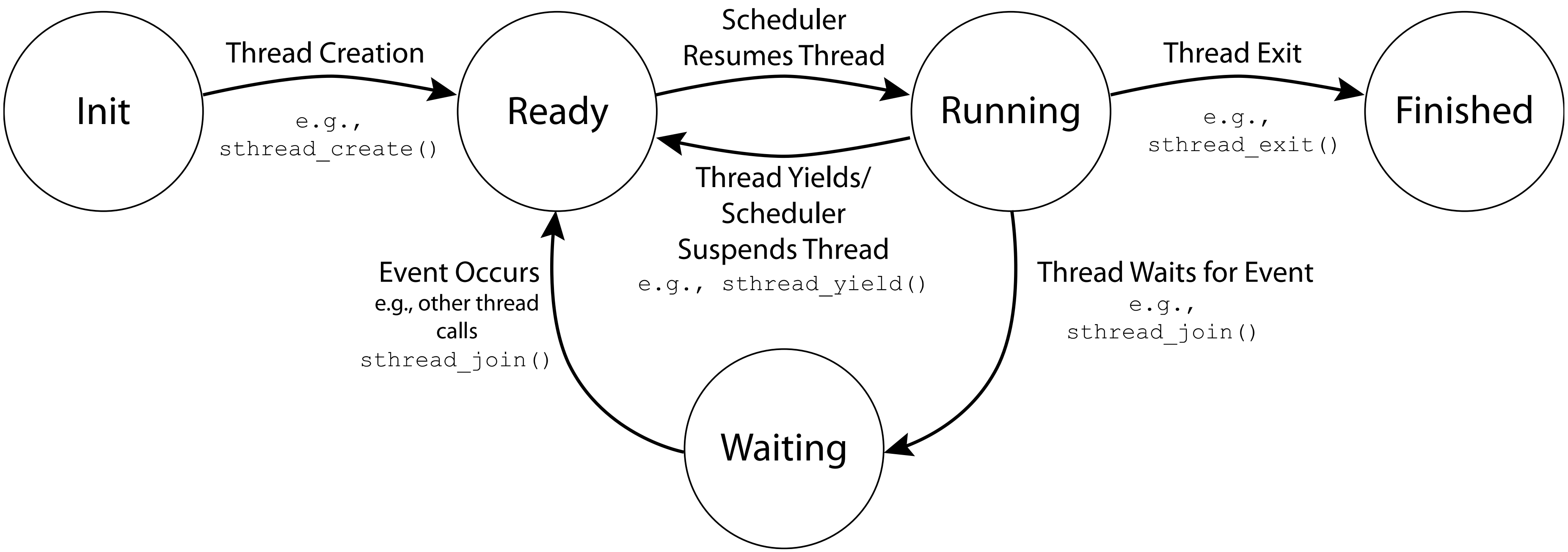
Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

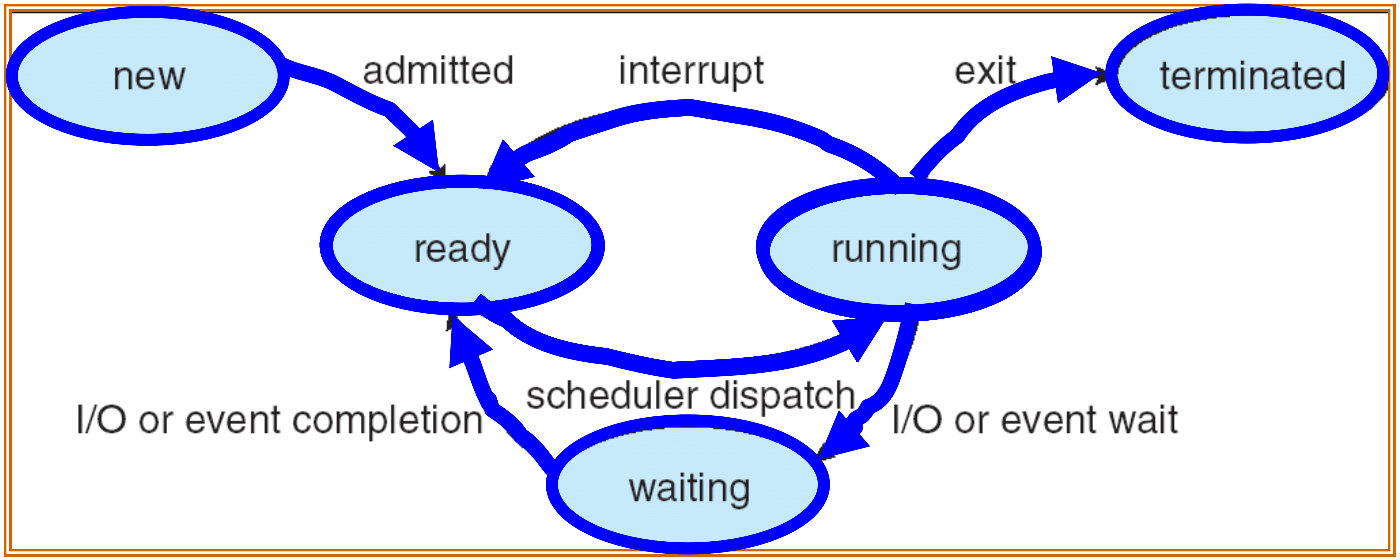


Thank You!

Thread Lifecycle



Lifecycle of a Process



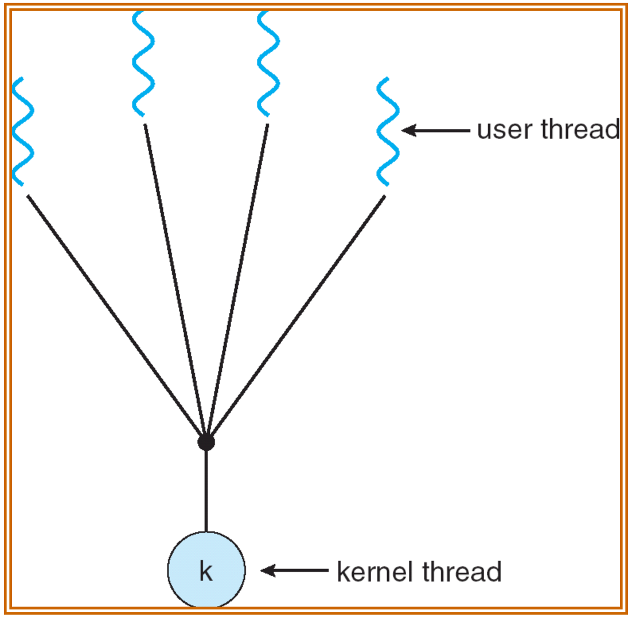
- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Kernel versus User-Mode Threads

- We have been talking about kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Lighter weight option: User level Threads

User-Mode Threads

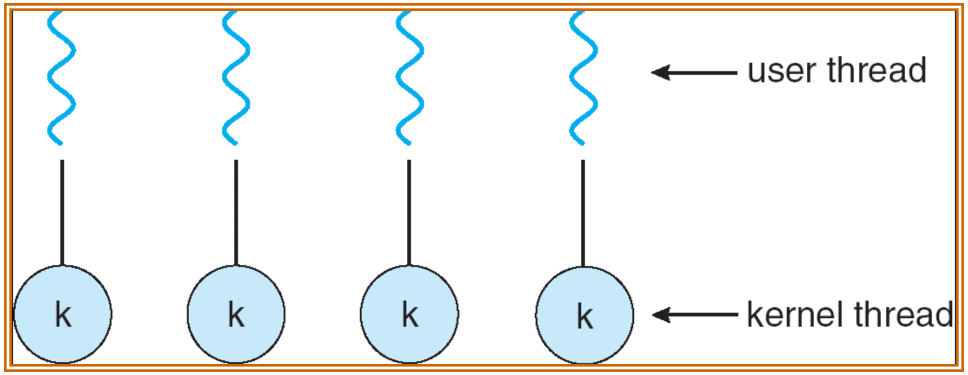
- Lighter weight option:
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap

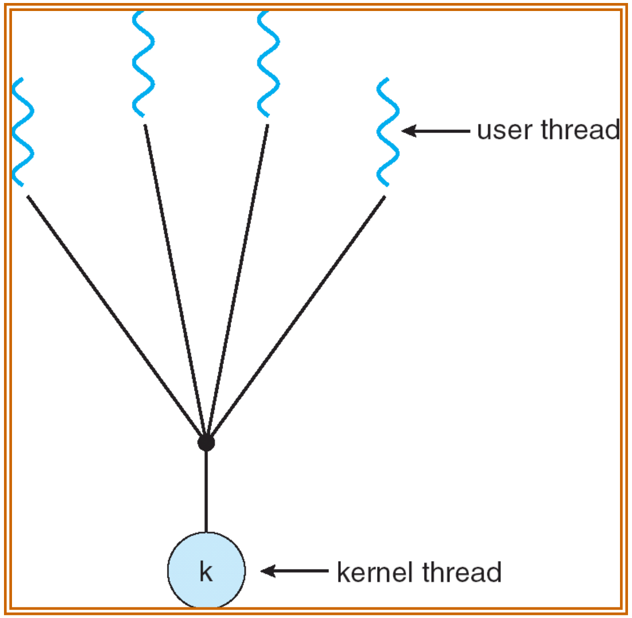


- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    - Have kernel inform user level when thread blocks…

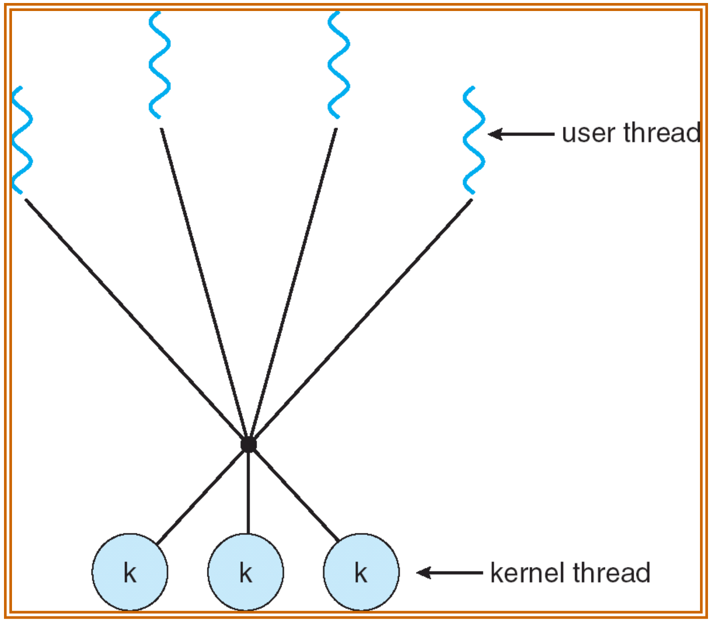
Some Threading Models

Simple One-to-One Threading Model





Many-to-One



Many-to-Many