TDIU11: Operating Systems

II. Processes, Threads and Scheduling

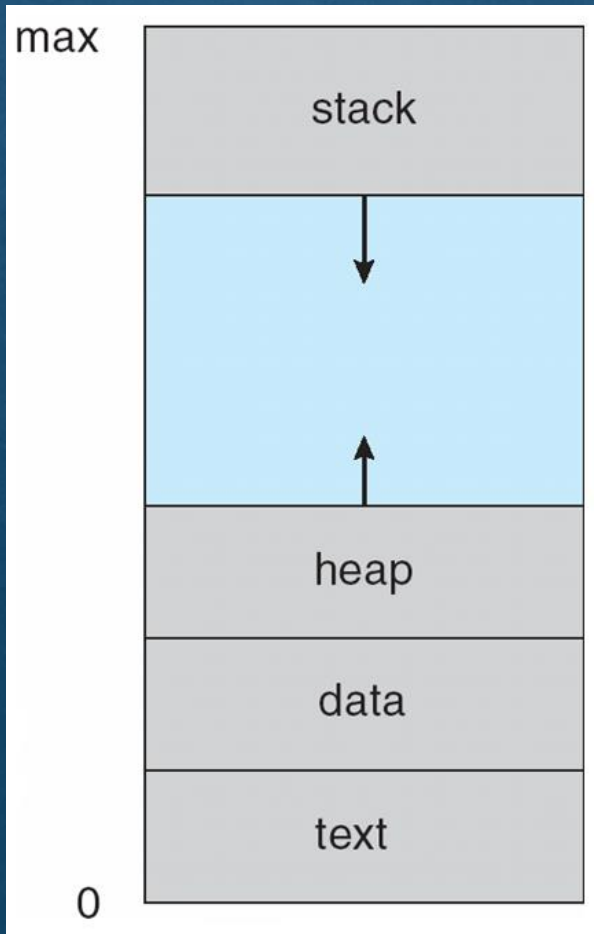
- Process concepts: context switch, scheduling queues, creation
- Multithreaded programming
- Process scheduling

SGG9: 3.1-3.3, 4.1-4.3, 5.1-5.4
SGG10: 3.1-3.3, 4.1-4.3, 5.1-5.4

Ahmed Rezine, Linköping University

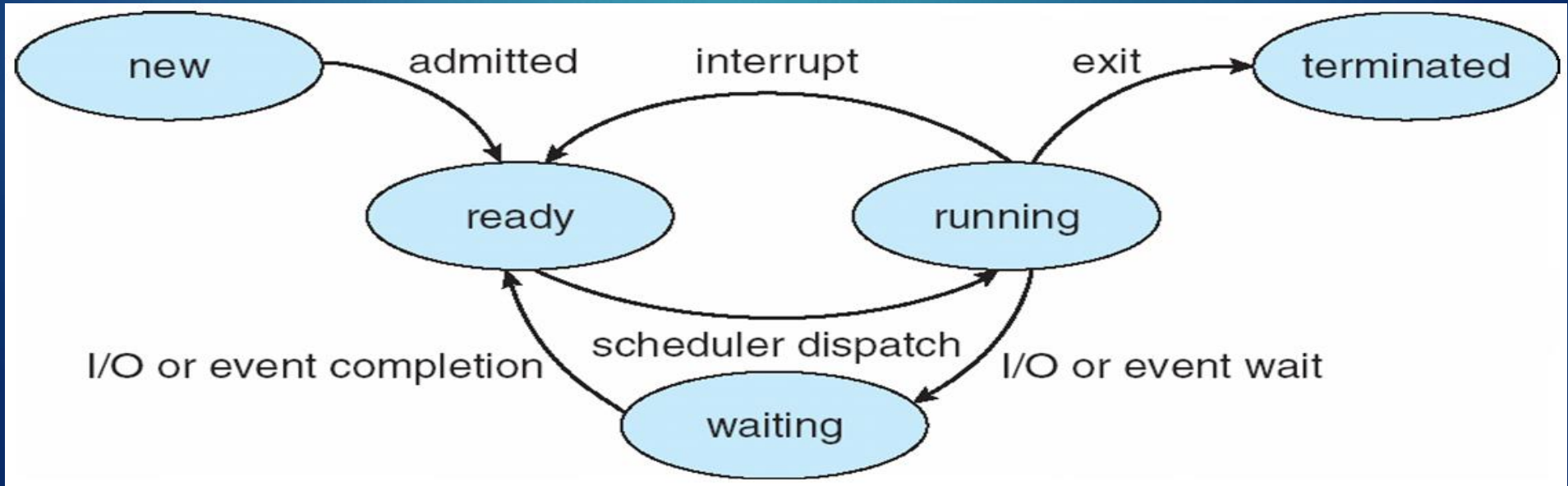
Process: a program in execution

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** with temporary data: function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

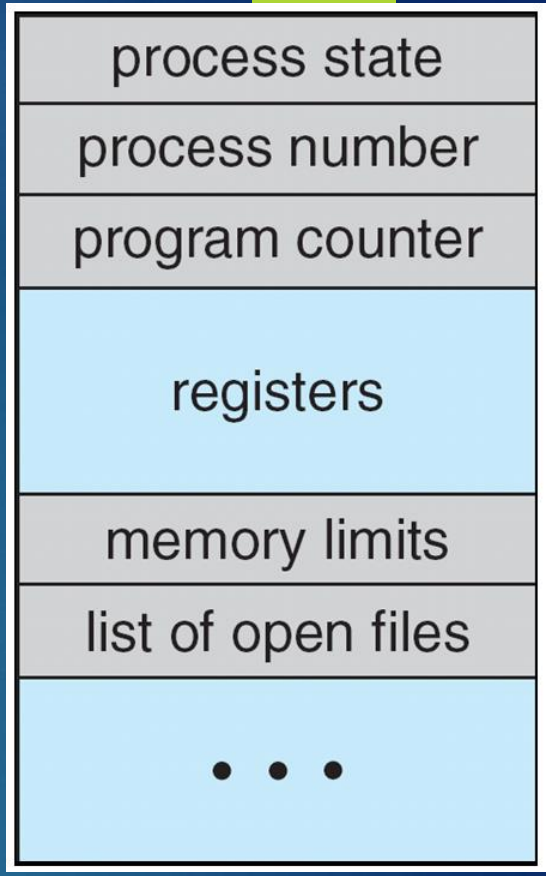


Process states as it executes

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



Process Control Block (PCB)



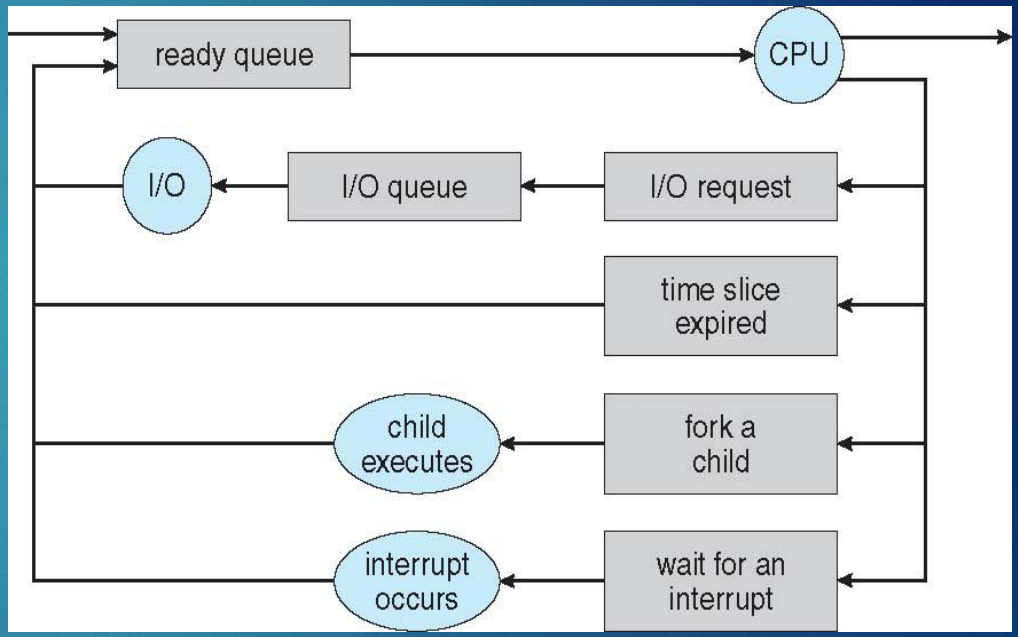
Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for execution on CPU
- **Scheduling queues** of processes
  - **Ready queue** –processes in main memory, ready and waiting to execute
  - **Job queue** – set of all processes in the system
  - **Device queues** – set of processes waiting for an I/O device



- **I/O-bound process:** many short CPU bursts
- **CPU-bound process:** few very long CPU bursts

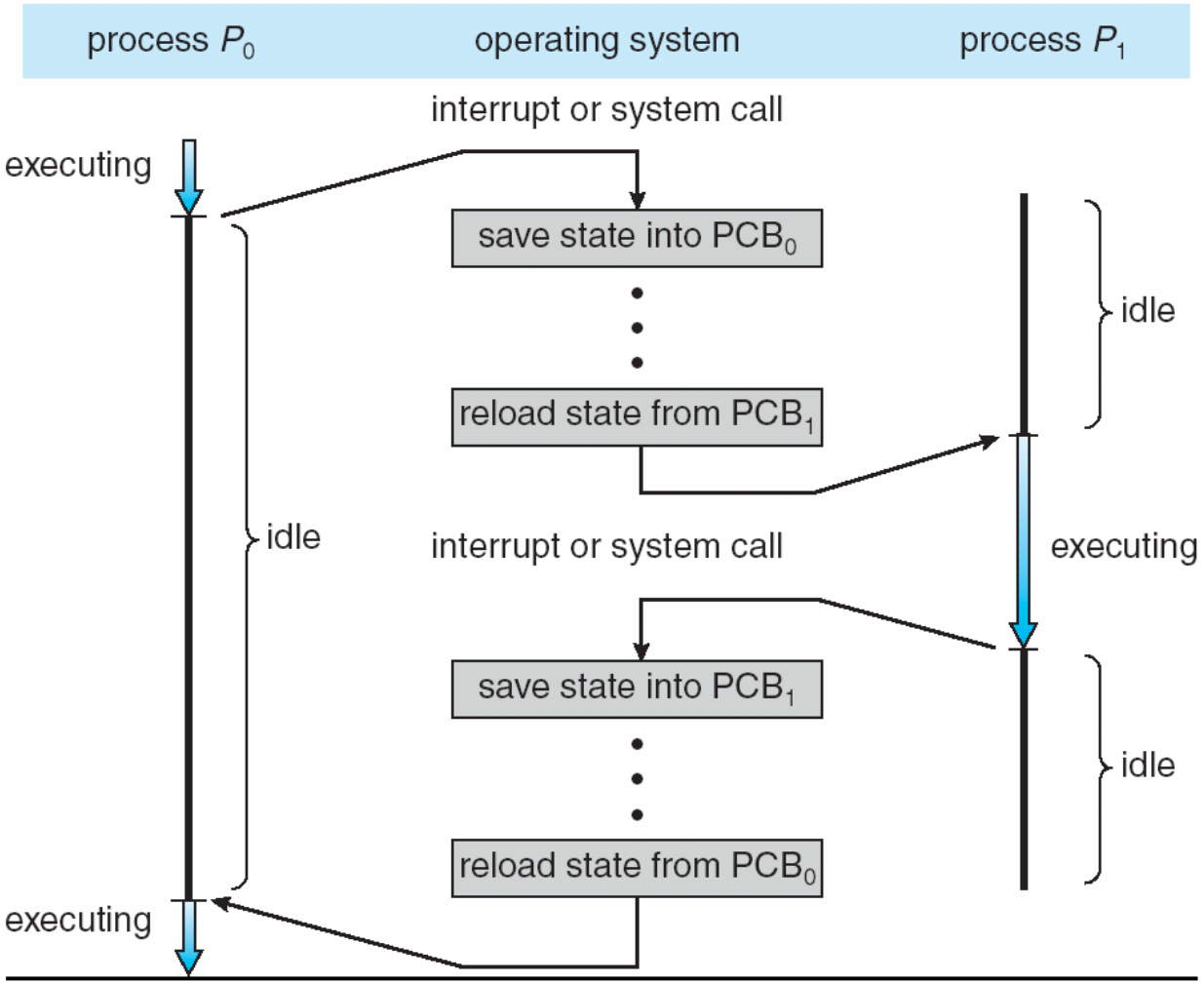
CPU and Job schedulers

- **Short-term scheduler** (or **CPU scheduler**) :
  - Selects process for execution, allocates CPU, sometimes the only scheduler
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**):
  - Selects processes to be brought to ready queue
  - Invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - Controls **degree of multiprogramming**
- Long-term scheduler strives for good *process mix of CPU bound and I/O bound processes*

Context Switch

- When CPU switches to another process, system **saves state** of old process and loads **saved state** for new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch

CPU Switch From Process to Process

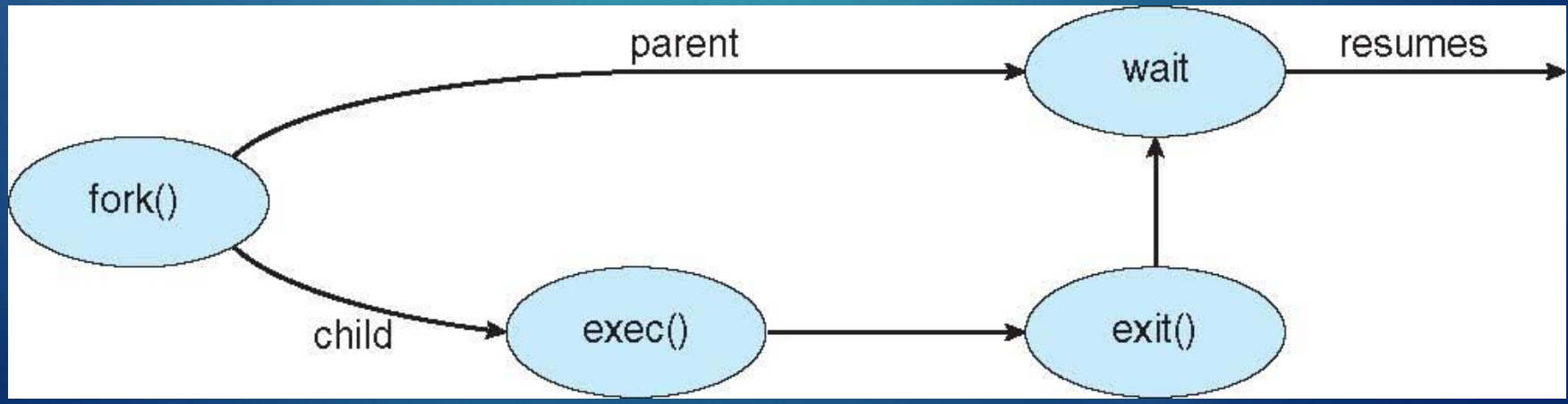


Operations on Processes: Creation

- **Parents** create **children** forming a **tree** of processes
- Process identified and managed via **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all / subset / no resources
- Execution options
  - Parent executes concurrently with / waits for children

Operations on Processes: Creation

- Address space: Child duplicate of parent, can have a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork():**
    - **R**eplace process' memory space with a new program



Operations on Processes: Termination

- Process executes last statement, asks OS to delete it using **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate children processes using **abort()** system call:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and OS does not allow a child to continue if parent terminates

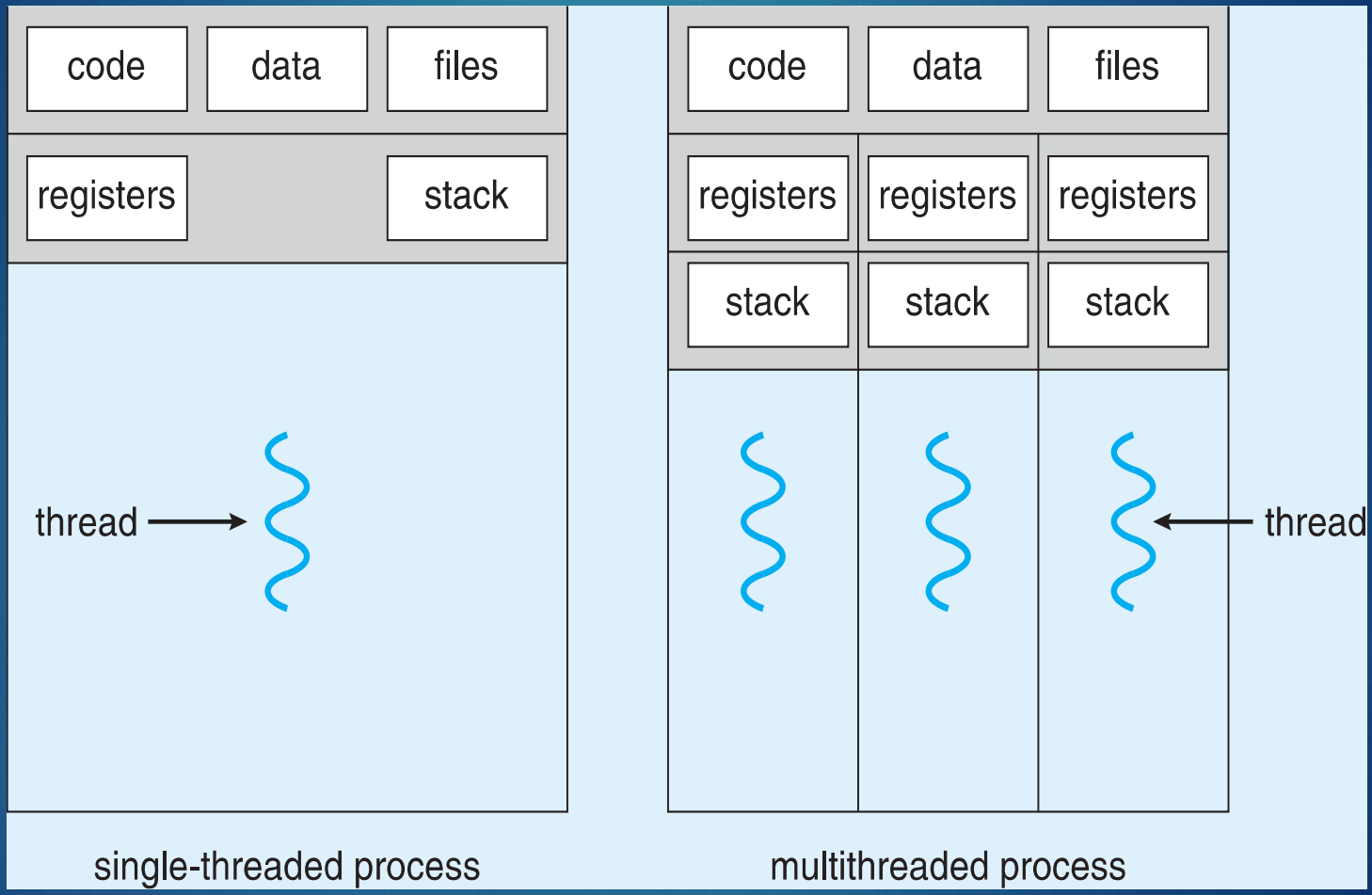
Operations on Processes: Inter-process Communication

- Processes may be ***independent*** or ***cooperating***
- Reasons for cooperating processes:
  - Information sharing, speedup, modularity, convenience
- Cooperating processes need **inter-process communication** (**IPC**). Two models of IPC: **Shared memory, Message passing**

Threads: Motivation

- Most modern applications are multithreaded
- Tasks within application can be implemented by separate threads
  - Update display, fetch data, spell checking, answer a network request
- Process creation is heavy-weight, thread creation is light-weight
- Kernels are generally multithreaded

Single and Multithreaded Processes

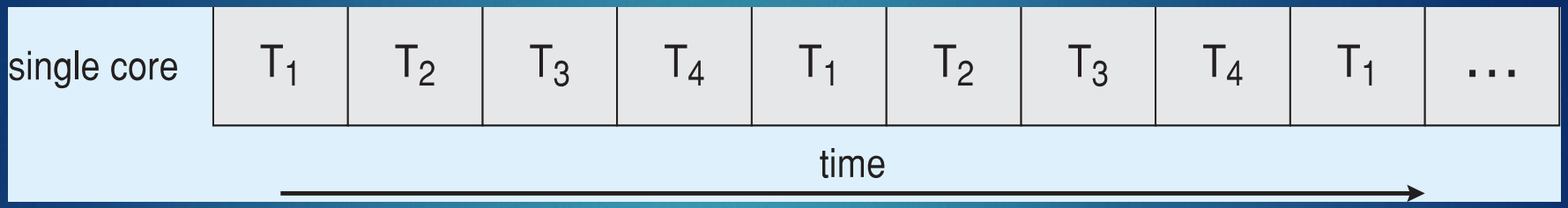


Threads: Benefits

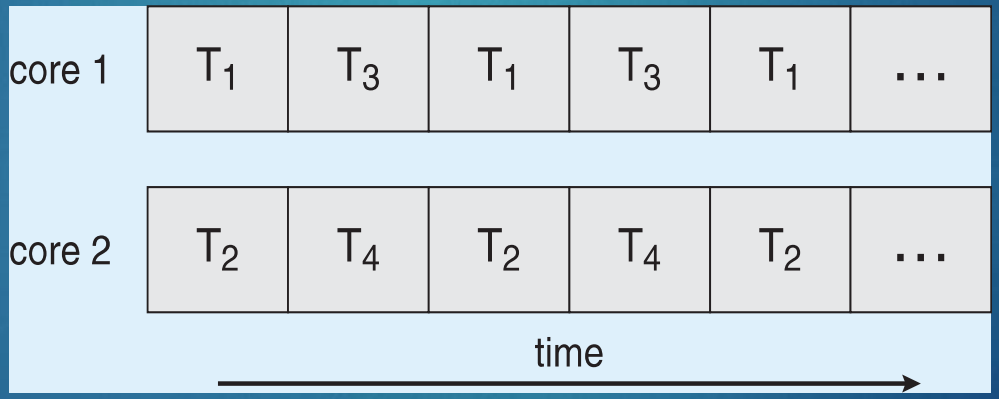
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching for processes
- **Scalability –** single threaded process can take advantage of only a processor in a multiprocessor architectures

Multicore Programming

- ***Concurrency*** supports more than one task making progress



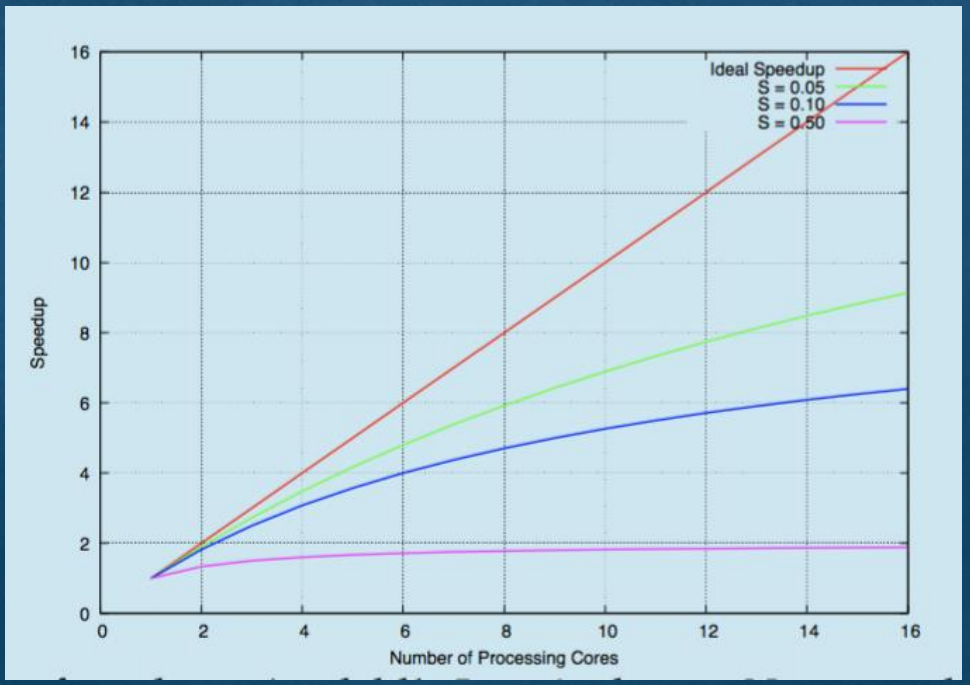
- ***Parallelism*** implies a system can perform more than one task simultaneously



- Multicore or multiprocessor challenges: dividing activities, balance, data splitting, data dependency, testing and debugging

Amdahl's Law

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

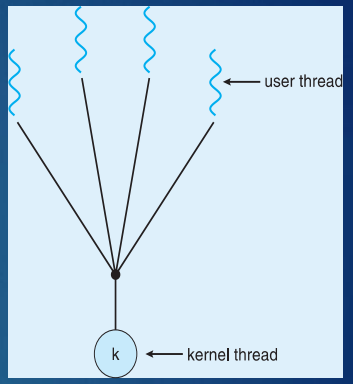


- Identifies performance gains from adding additional cores to an application
- *S* is serial portion, *N* processing cores
- If application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

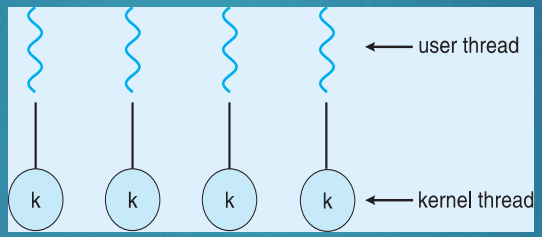
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

User Threads and Kernel Threads

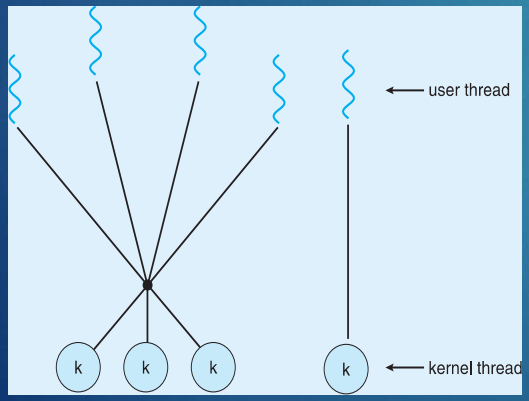
- **User threads** - management done by user-level threads library: POSIX **Pthreads,** Windows threads, Java threads
- **Kernel threads** - Supported by the Kernel



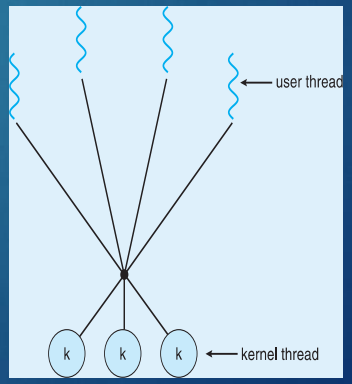
many-to-one



one-to-one



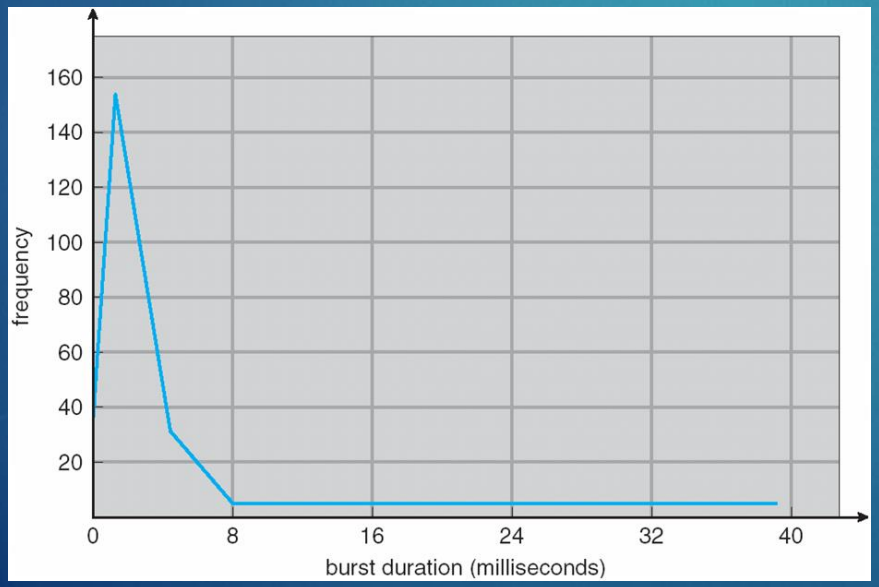
two-level-model

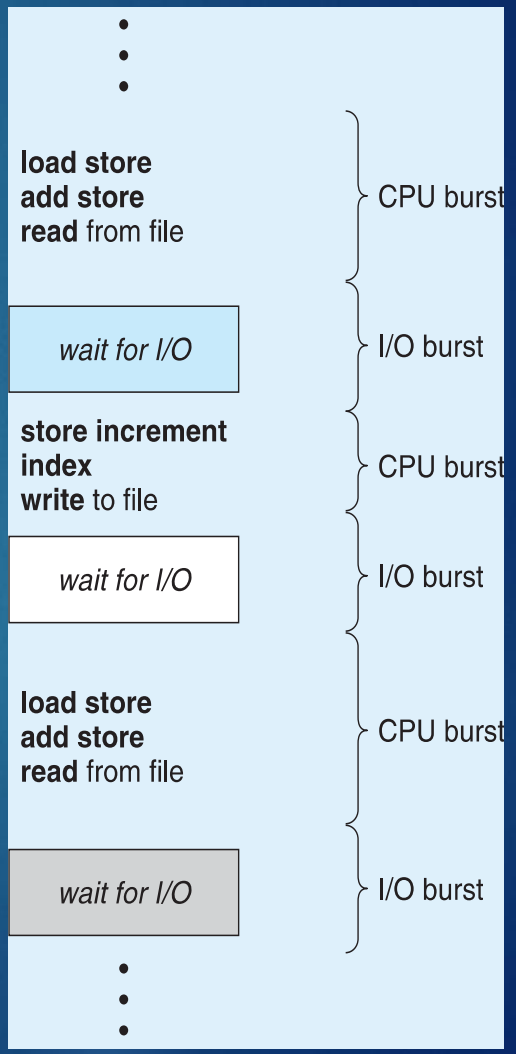


many-to-many

Scheduling: Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



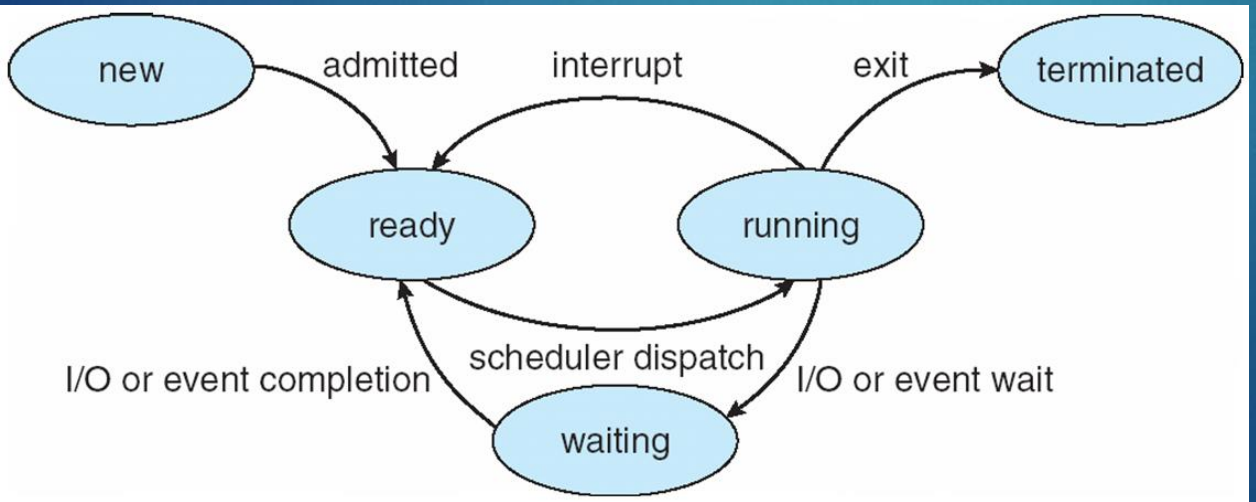


CPU Scheduler

- **Short-term scheduler** selects among processes in ready queue
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

- Scheduling under 1 and 4 is **non-preemptive**
- Any other scheduling is **preemptive.**



Shared data, preemption, during crucial OS activities

Scheduling Criteria

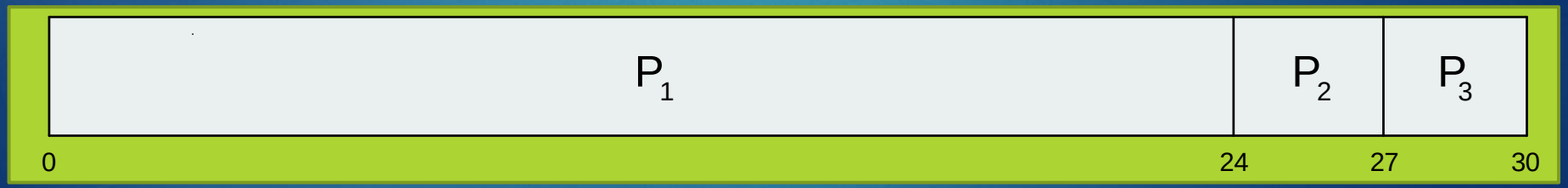
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process: from submission to completion, including waiting to get to memory, ready queue, executing on CPU, I/O,…
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

  The Gantt Chart for the schedule is:

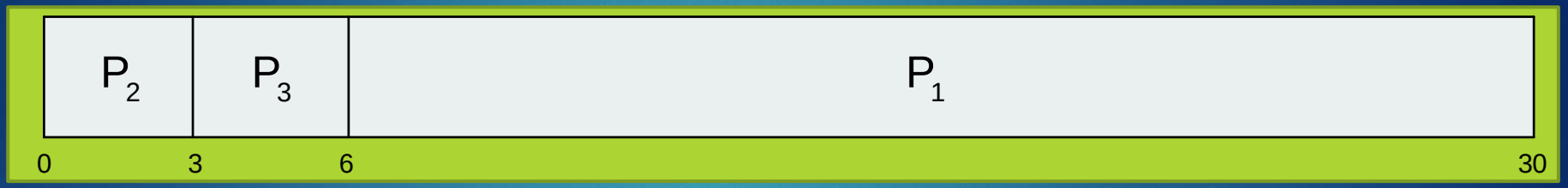


- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order: P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

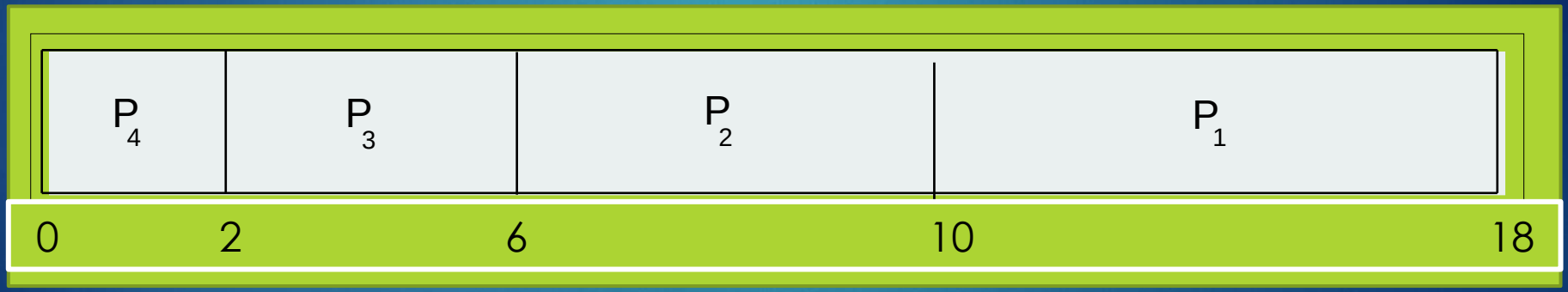
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Arrival Time	Burst Time
P_1	0	8
P_2	0	6
P_3	0	4
P_4	0	2

- SJF scheduling chart



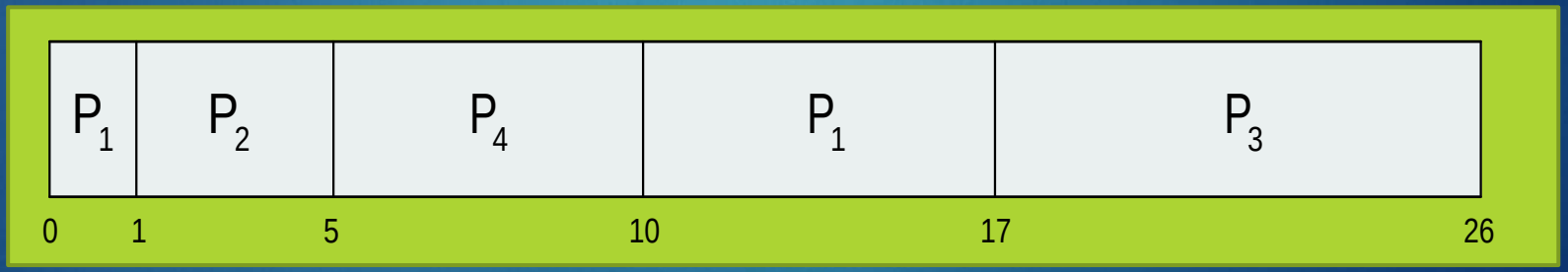
- Average waiting time = (10 + 6 + 2 + 0) / 4 = 4.5

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

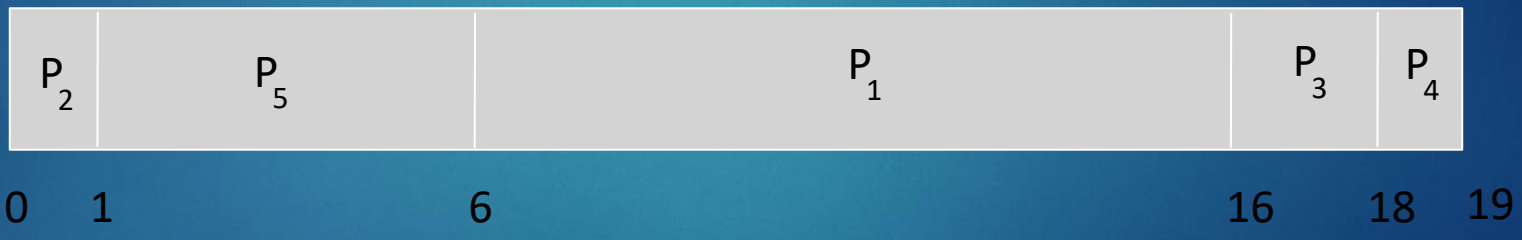
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - It can be Preemptive or Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Possible solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority (smaller number is higher priority)
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2

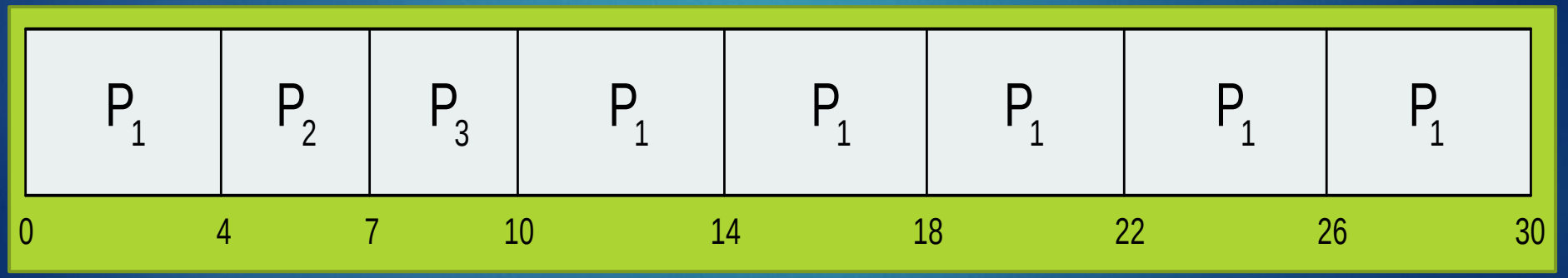
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Timer interrupts every quantum to schedule next process
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

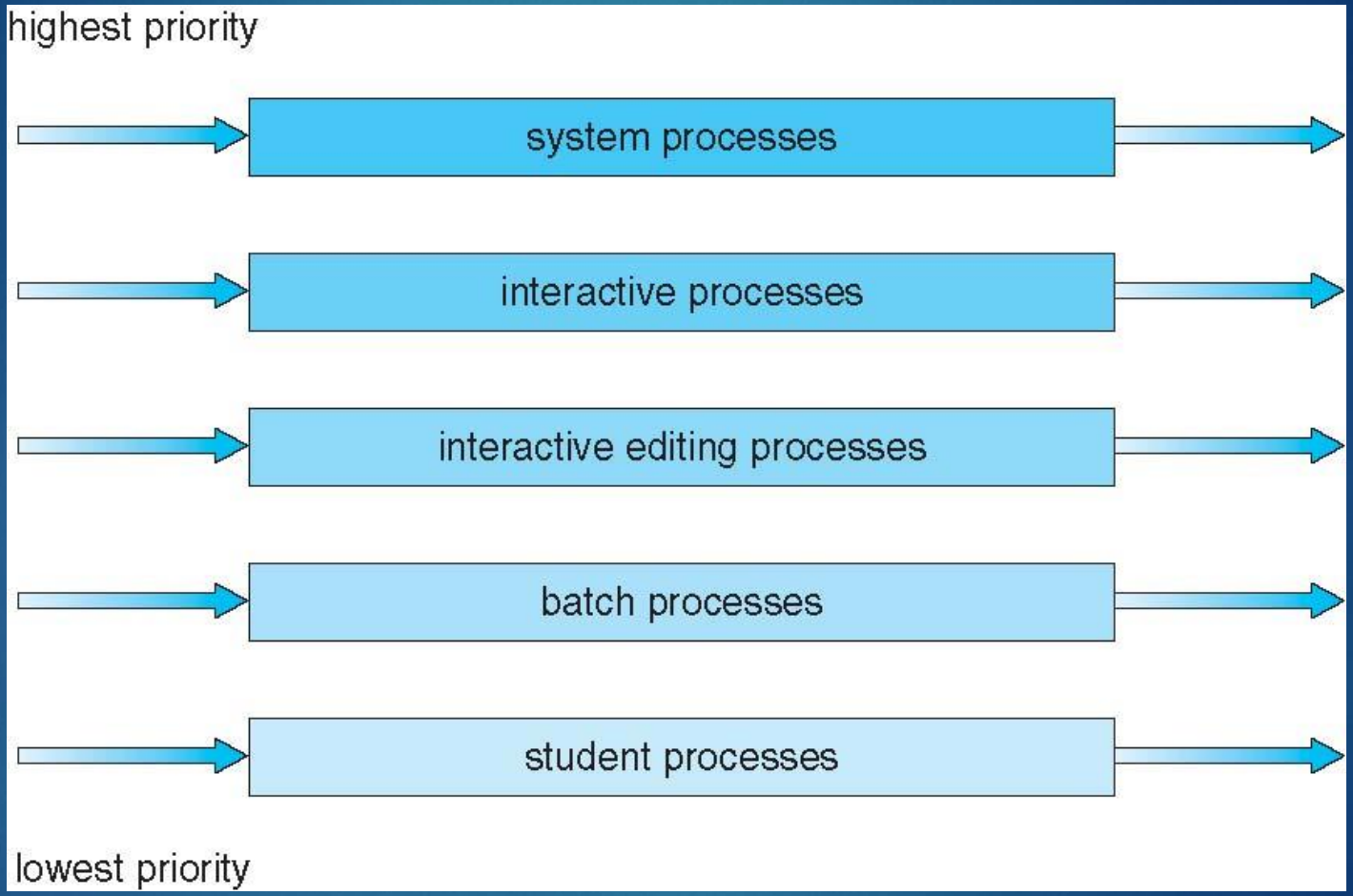


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS