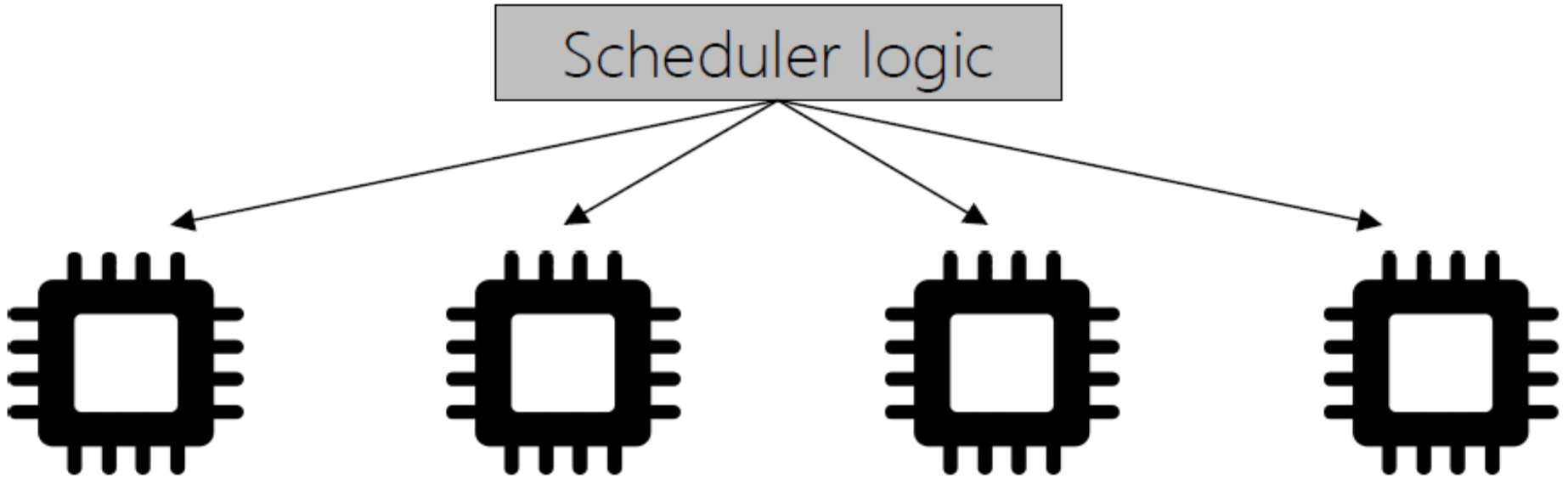
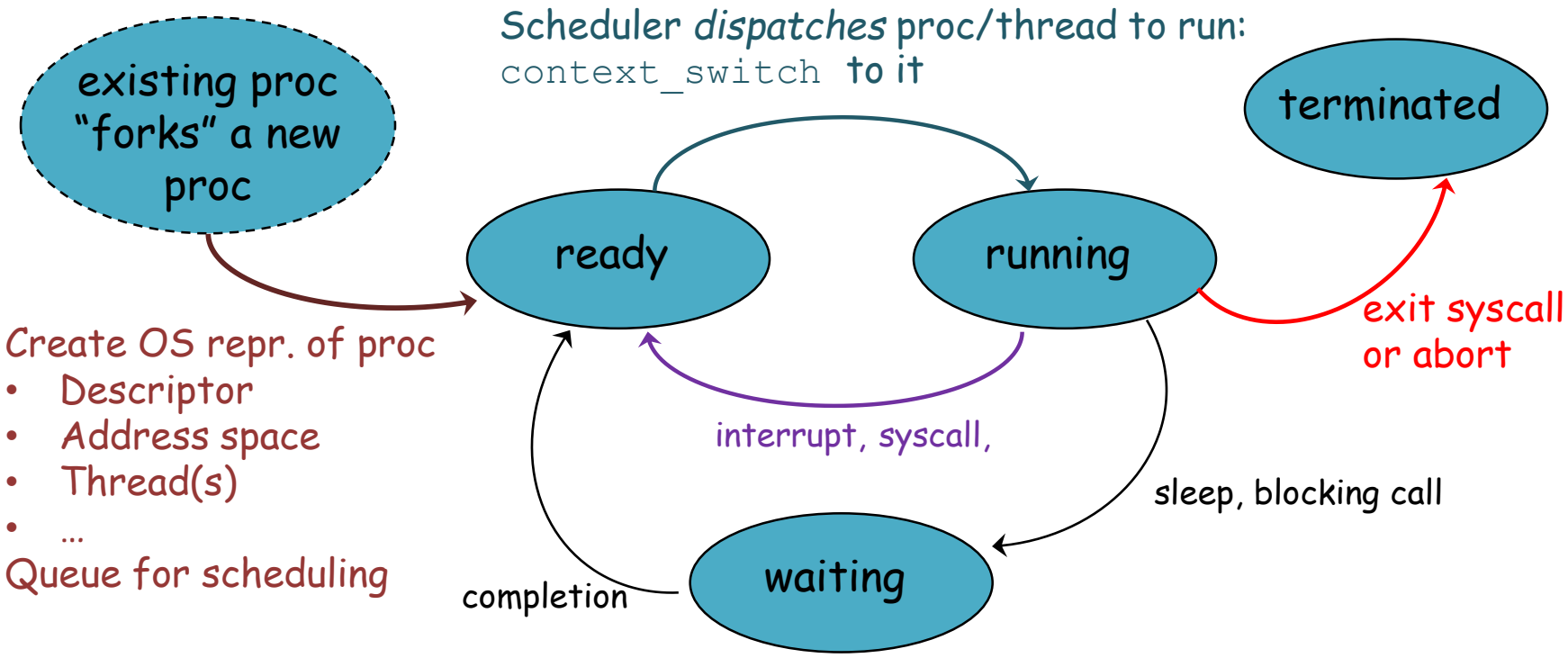
Scheduling Algorithms

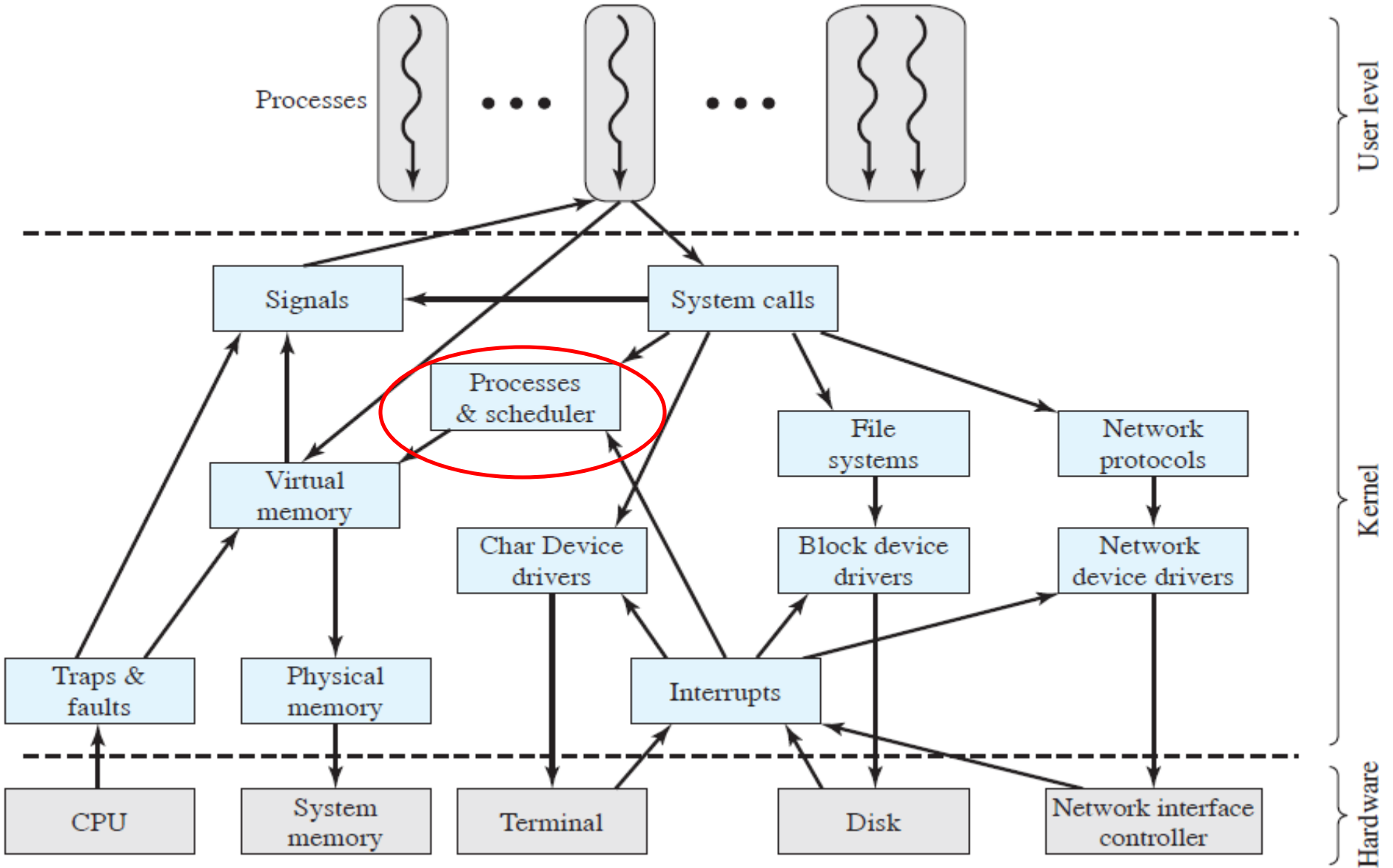


Lifecycle of a process



- OS juggles many process/threads using kernel data structures
- Proc's may create other process (fork/exec)
  - All starts with init process at boot

Linux Kernel Components



Basic Concepts

In a single-processor, only one process can run at a time. Others wait until CPU is free and can be rescheduled.

Multiprogramming aims at maximizing CPU utilization.

A process is executed until it must wait, e.g. for completion of I/O request. Rather than CPU staying idle, multiprogramming uses this time productively.

When one process has to wait, OS gives the CPU to another process residing in memory.

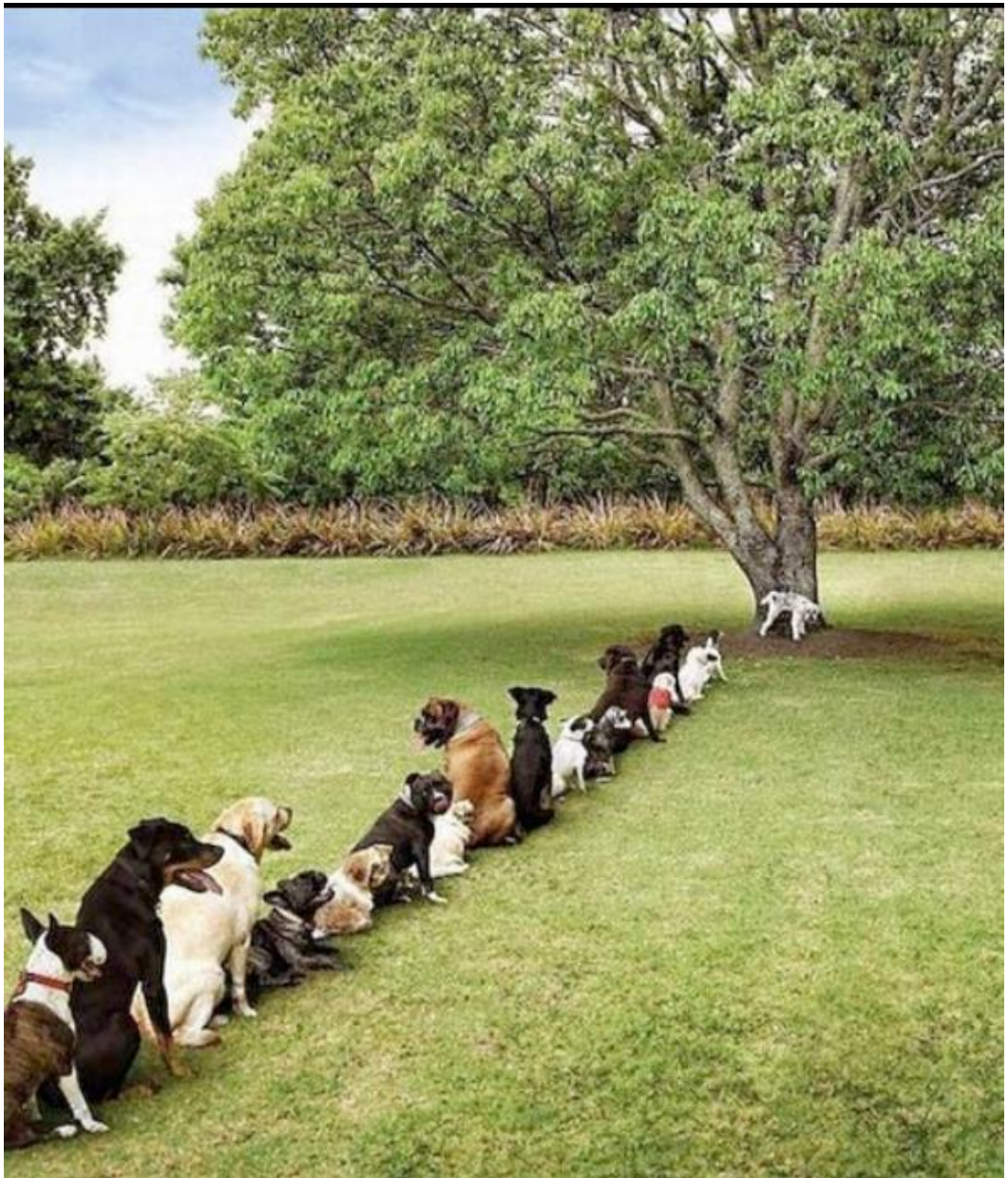
Scheduling is a fundamental OS function.

Scheduling Algorithms

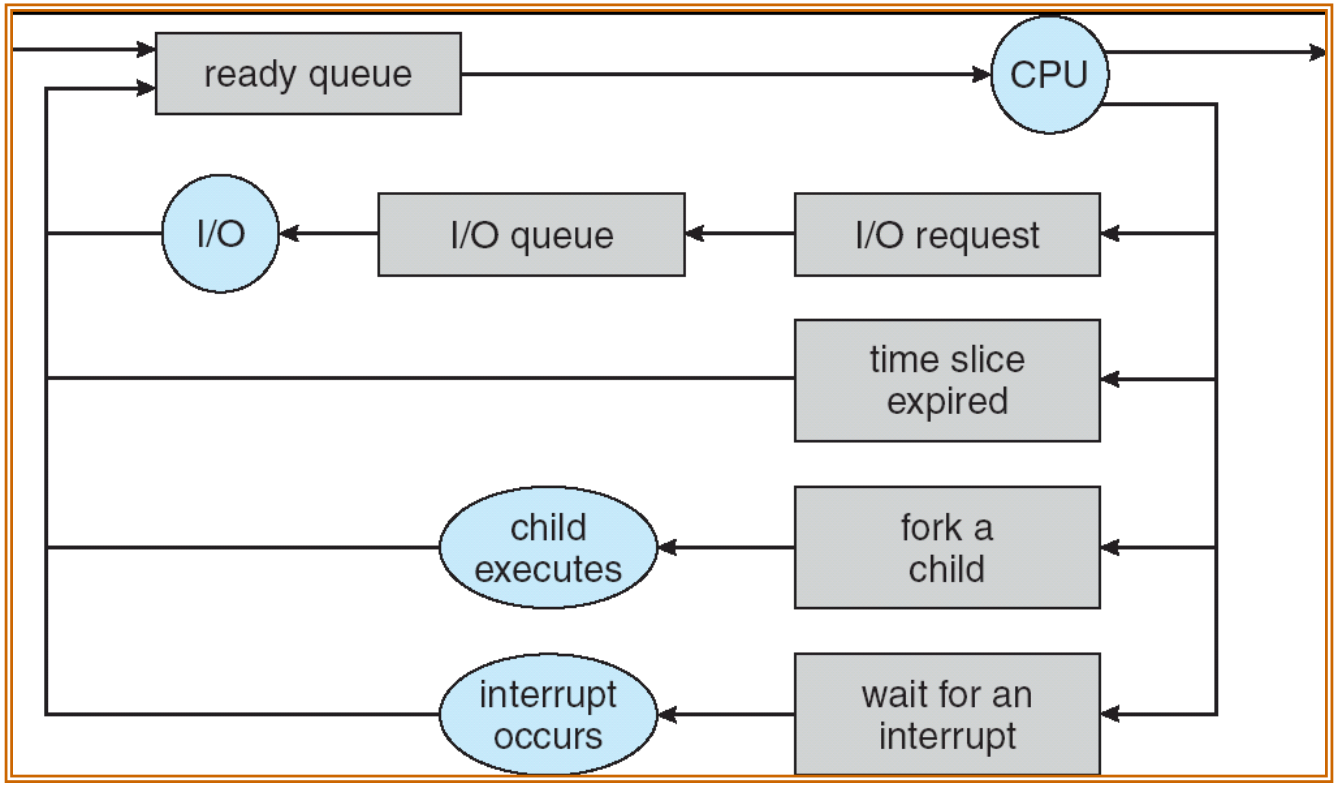
- Goals for scheduling
- FCFS & Round Robin
- SJF
- Multilevel Feedback Queues
- Lottery Scheduling

Scheduling: All About Queues

Scheduling: All About Queues



Scheduling: All About Queues

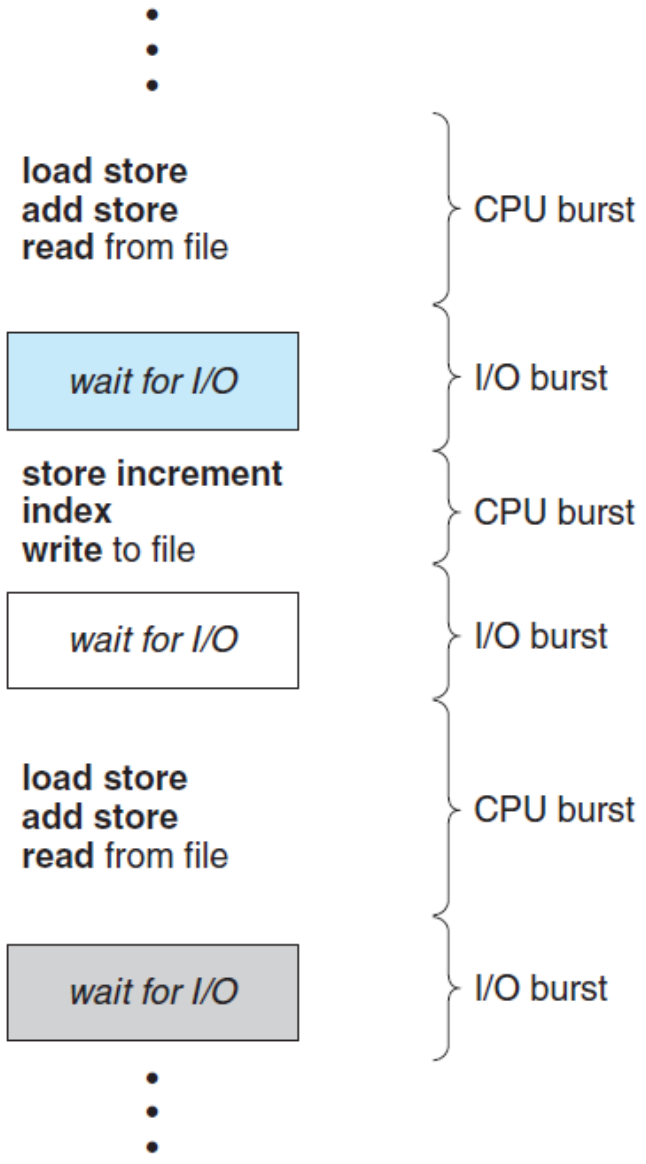


- PCBs move from queue to queue
- **Scheduling**: which order to remove from queue
  - Much more on this soon

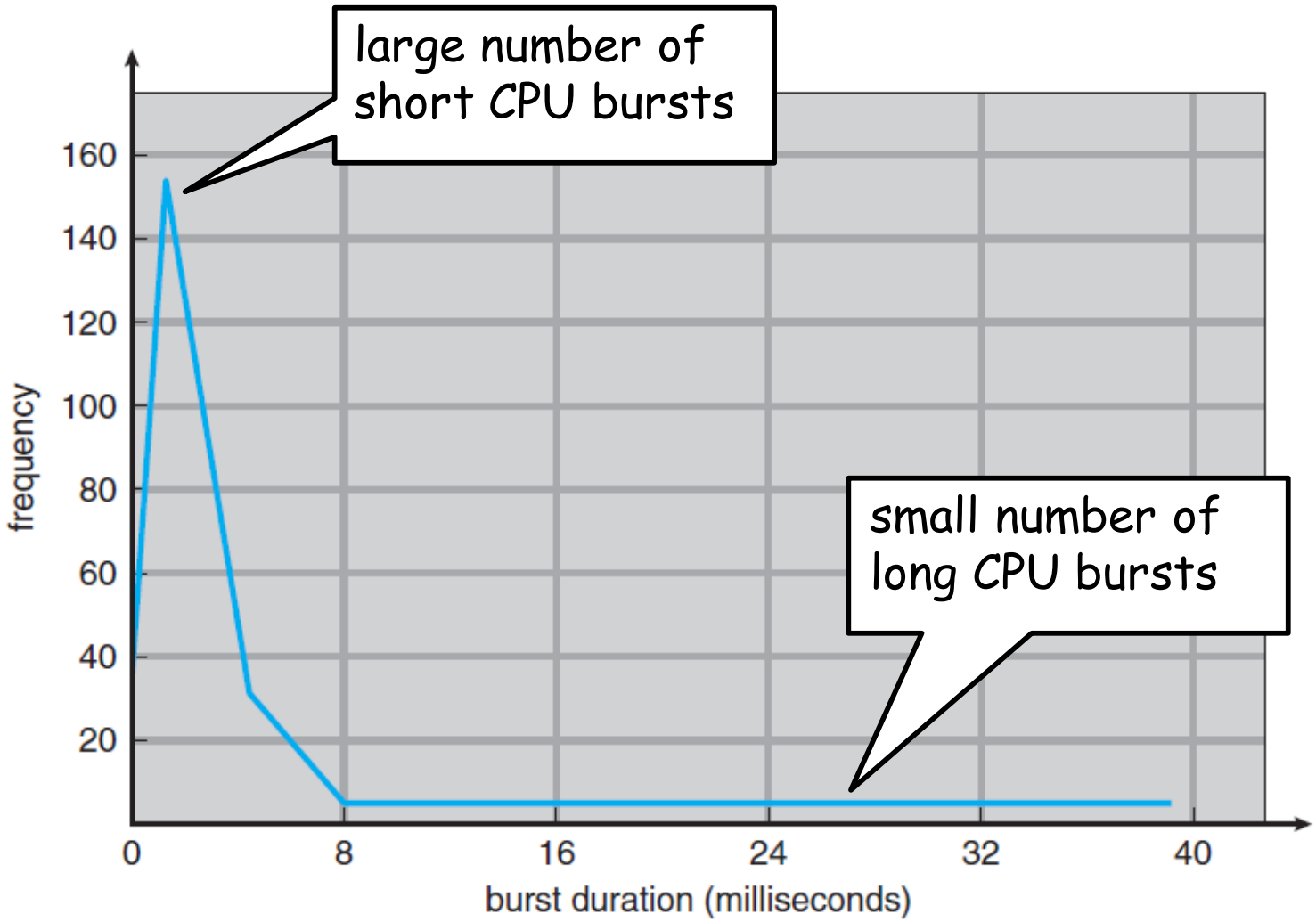
CPU–I/O Burst Cycle

Process execution consists of a **cycle** of CPU execution and I/O wait alternation, beginning with a **CPU burst**, followed by an **I/O burst**.

Eventually, the final CPU burst ends with a system request to terminate execution.



Histogram of CPU-burst durations.



An **I/O-bound** program has many short CPU bursts. A **CPU-bound** program have long CPU bursts.

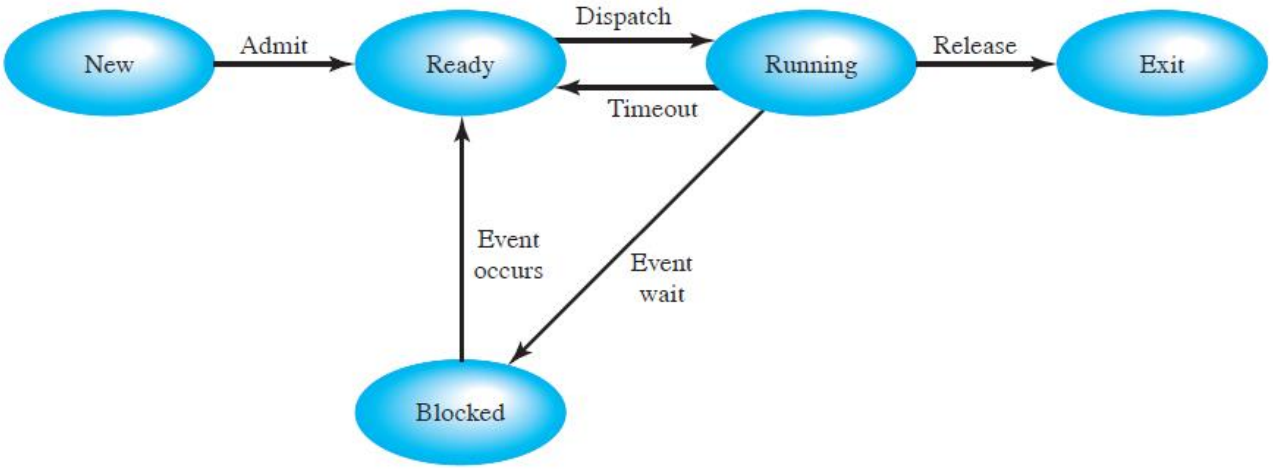
When CPU becomes idle, OS selects by the **short-term scheduler** one of the processes in the **ready queue** for execution.

The records in the queues are process control blocks (PCBs) of the processes.

Scheduling Processes

- Multiprogramming: running more than one process at a time enables the OS to increase system utilization and throughput by overlapping I/O and CPU activities.

Process Execution



All of the processes that the OS is currently managing reside in one and only one of these state queues.

Scheduling Processes

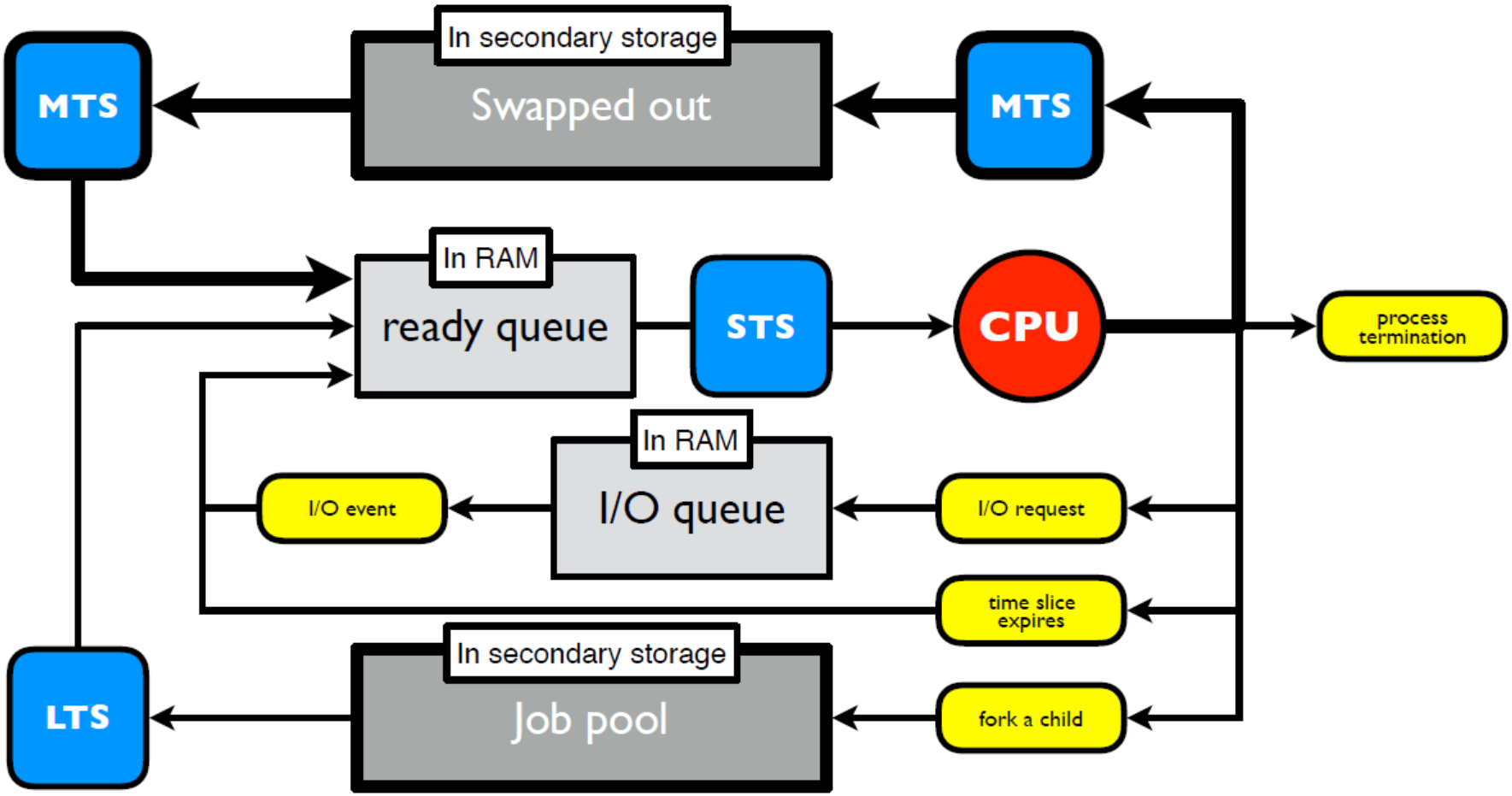
Long Term Scheduling: How does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory?

Medium-term scheduler can be added if degree of multiple programming needs to decrease

• **Short Term Scheduling**: How does (or should) the OS select a process from the ready queue to execute?

– Policy Goals
– Policy Options
– Implementation considerations

Scheduling Processes



Preemptive Scheduling

CPU-scheduling occurs when a process:

1. Switches from the running state to the waiting state (e.g. I/O request, **wait()** for termination of a child).
2. Switches from the running state to the ready state (e.g. by interrupt).
3. Switches from the waiting state to the ready state (e.g. at completion of I/O).
4. Terminates.

In situations 1 and 4 a new process (if ready queue nonempty) is selected for execution.

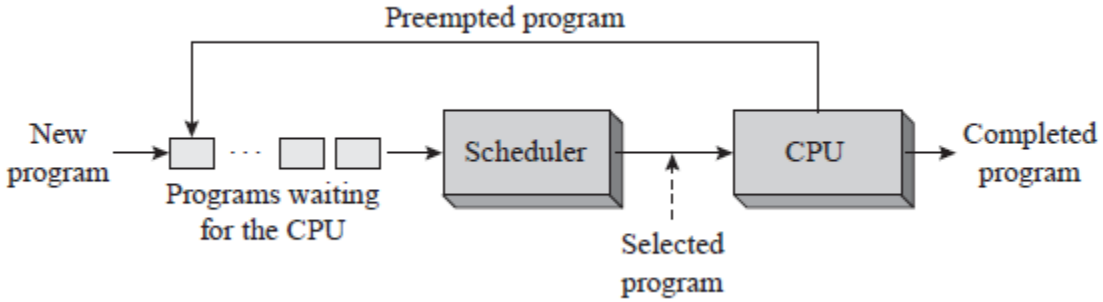
Situations 2 and 3 may or may not imply scheduling.

Scheduling under 1 and 4 only is called **non-preemptive** or **cooperative**. Otherwise, it is **preemptive**.

Non-preemptive scheduling keeps the CPU allocated to a process until it either terminates or switched to waiting state (Windows 3.x).

Most OS use preemptive scheduling, which can cause race conditions when data are shared among processes .

preemption


Preempted program
New program
Programs waiting for the CPU
Scheduler
CPU
Completed program
Selected program

Dispatcher

A **dispatcher** is the module giving control of the CPU to the process selected by the **short-term scheduler**.

Dispatching involves:

- Switching context.
- Switching to user mode.
- Jumping to the user's program PC location for restart.

The dispatcher should be as fast as possible, since it is invoked during every process switch.

The time it takes for the dispatcher to switch processes is called **dispatch latency**.

Criteria for Comparing Scheduling Algorithms

Criteria	Definition	Goal
CPU utilization	The % of time the CPU is executing user level process code.	Maximize
Throughput	Number of processes that complete their execution per time unit.	Maximize
Turnaround time	Amount of time to execute a particular process.	Minimize
Waiting time	Amount of time a process has been waiting in the ready queue.	Minimize
Response time	Amount of time it takes from when a request was submitted until the first response is produced.	Minimize

Scheduling Policies

Ideally, choose a CPU scheduler that optimizes all criteria simultaneously (utilization, throughput,..), but this is not generally possible

Instead, choose a scheduling algorithm based on its ability to satisfy a policy

- Minimize average response time - provide output to the user as quickly as possible and process their input as soon as it is received.
- Minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance.
- Maximize throughput - two components
  - minimize overhead (OS overhead, context switching)
  - efficient use of system resources (CPU, I/O devices)
- Minimize waiting time - give each process the same amount of time on the processor. This might actually increase average response time.

Scheduling Policies-Assumptions

Simplifying Assumptions

- **One process per user**
- **One thread per process**
- **Processes are independent**

Researchers developed these algorithms in the 70's when these assumptions were more realistic, and it is still an open problem how to relax these assumptions.

Scheduling Algorithms: A Snapshot

FCFS: First Come, First Served
Round Robin: Use a time slice and preemption to alternate jobs.
SJF: Shortest Job First
Multilevel Feedback Queues: Round robin on each priority queue.
Lottery Scheduling: Jobs get tickets and scheduler randomly picks winning ticket.

Scheduling Policies

FCFS: First-Come-First-Served (or FIFO: First-In-First-Out)

- The scheduler executes jobs to completion in arrival order.
- In early FCFS schedulers, the job did not relinquish the CPU even when it was doing I/O.
- We will assume a FCFS scheduler that runs when processes are blocked on I/O, but that is non-preemptive, i.e., the job keeps the CPU until it blocks (say on an I/O device).

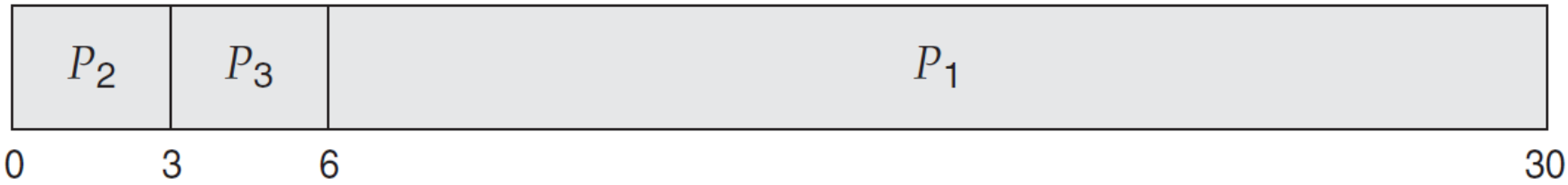
FCFS Scheduling Policy: Example

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

If the processes arrive in the order P_2, P_3, P_1, as shown



$$\text{average waiting time} = (6 + 0 + 3)/3 = 3.$$

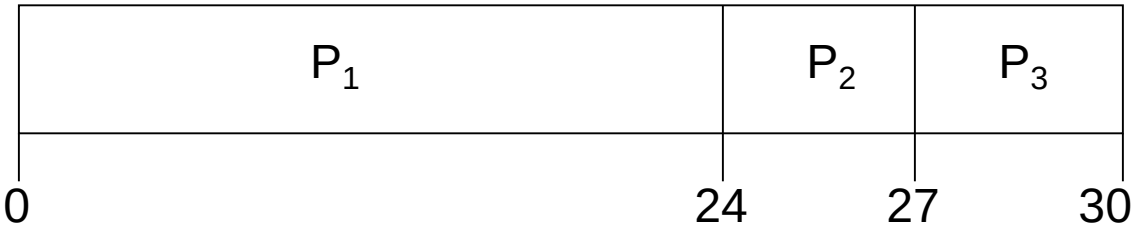
FCFS average waiting time is not minimal and vary substantially if processes' CPU burst times vary greatly.

Consider FCFS performance in a dynamic situation, with one CPU-bound and many I/O-bound processes.

The CPU-bound process will get and hold the CPU.

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

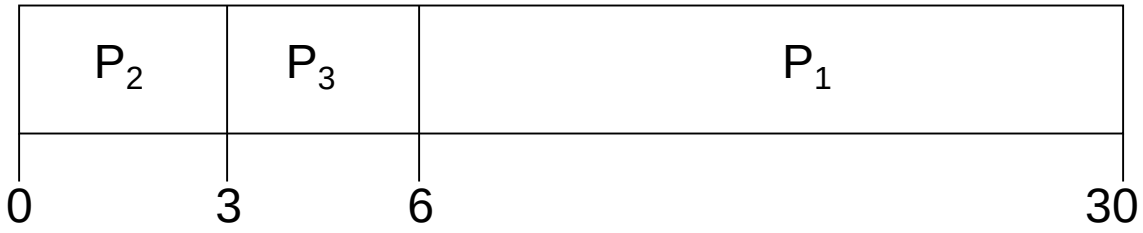


- **Throughput**: 3 processes/30 milliseconds = 1/10
- **Turnaround time** for P_1 = 24; P_2 = 27; P_3 = 30
- **Average turnaround time**: (24 + 27 + 30)/3 = 81/3 = 27
- **Waiting time** for P_1 = 0; P_2 = 24; P_3 = 27
- **Average waiting time**: (0 + 24 + 27)/3 = 51/3 = 17

FCFS Scheduling

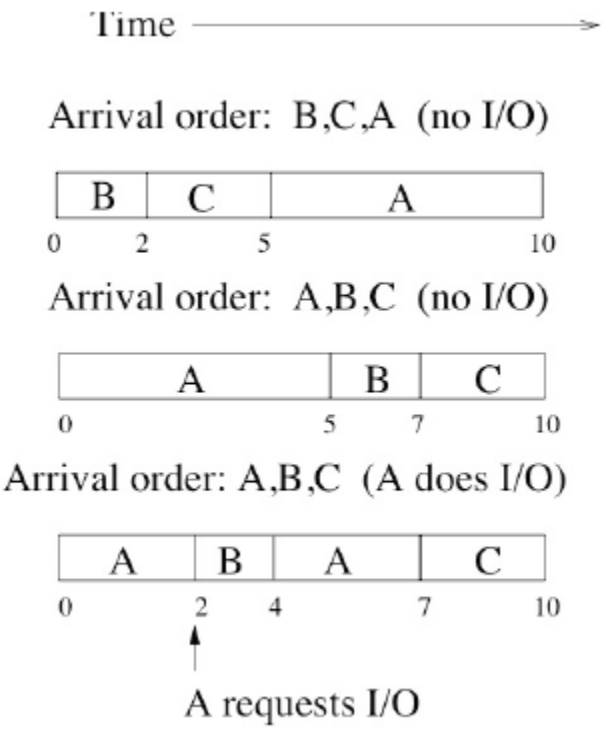
Suppose that the processes arrive in the order

P_2 , P_3 , P_1 all at time 0



- **Throughput**: 3 processes/30 milliseconds = 1/10
- **Turnaround time** for P_1 = 30; P_2 = 3; P_3 = 6
- **Average turnaround time**: (30 + 3 + 6)/3 = 39/3 = 13
- **Waiting time** for P_1 = 6; P_2 = 0; P_3 = 3
- **Average waiting time**: (6 + 0 + 3)/3 = 9/3 = 3

FCFS Scheduling Policy: Example



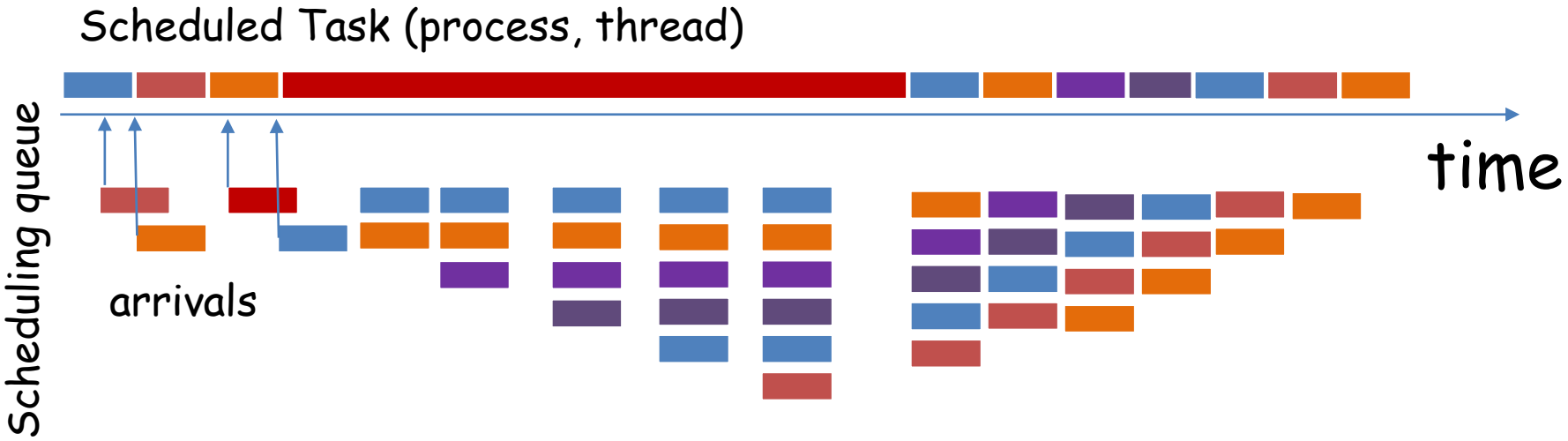
Again, all the I/O bound processes end up waiting in the ready queue until the CPU-bound process is done.

There is a **convoy effect**. All the other processes wait for the big to get off the CPU, yielding lower CPU and device utilization.

Allowing the shorter processes higher priority would be better.

FCFS algorithm is **non preemptive**. Process keeps CPU until releasing it, either terminating or requesting I/O.

Convoy effect



- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

FCFS: Advantages and Disadvantages

Advantage: simple

Disadvantages:

- average wait time is highly variable as short jobs may wait behind long jobs.

- may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle

Round Robin Scheduling

Variants of round robin are used in most time sharing systems

- Add a timer and use a preemptive policy.
- After each time slice, move the running thread to the back of the queue.

- Selecting a time slice:
  - Too large - waiting time suffers, degenerates to FCFS if processes are never preempted.
  - Too small - throughput suffers because too much time is spent context switching.

  => Balance these tradeoffs by selecting a time slice where context switching is roughly 1% of the time slice.

- Today: typical time slice= 10-100 ms, context switch time= 0.1-1ms
- **Advantage:** It's fair; each job gets an equal shot at the CPU.
- **Disadvantage:** Average waiting time can be bad.

The ready queue is treated as a **circular queue**. CPU is allocated to process for a time interval of 1quantum.

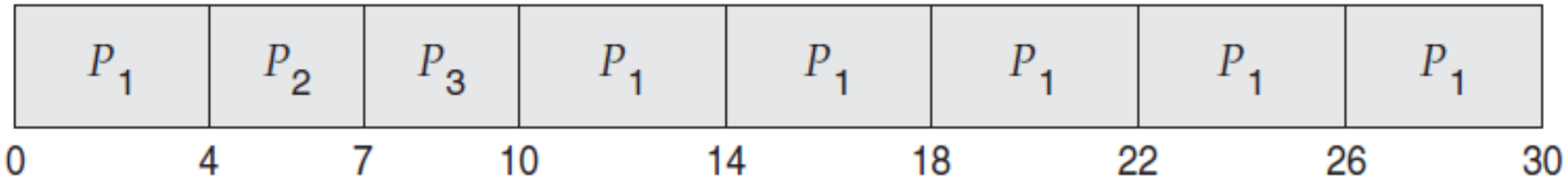
A new process is added to tail of ready queue

If CPU burst of current process is longer than 1 slice, the timer will interrupt, the context will be switched, and the process will be put at the queue tail.

The average waiting time under the RR is often long.

Example: Processes arrive at time 0, with the length of the CPU burst in mSec and time quantum of 4 mSec.

Process	Burst Time
P_1	24
P_2	3
P_3	3



average waiting time: $[(10-4)+4+7]/3 = 5.66$.

If a process's CPU burst exceeds 1 quantum, that process is preempted and is put back in the ready queue.

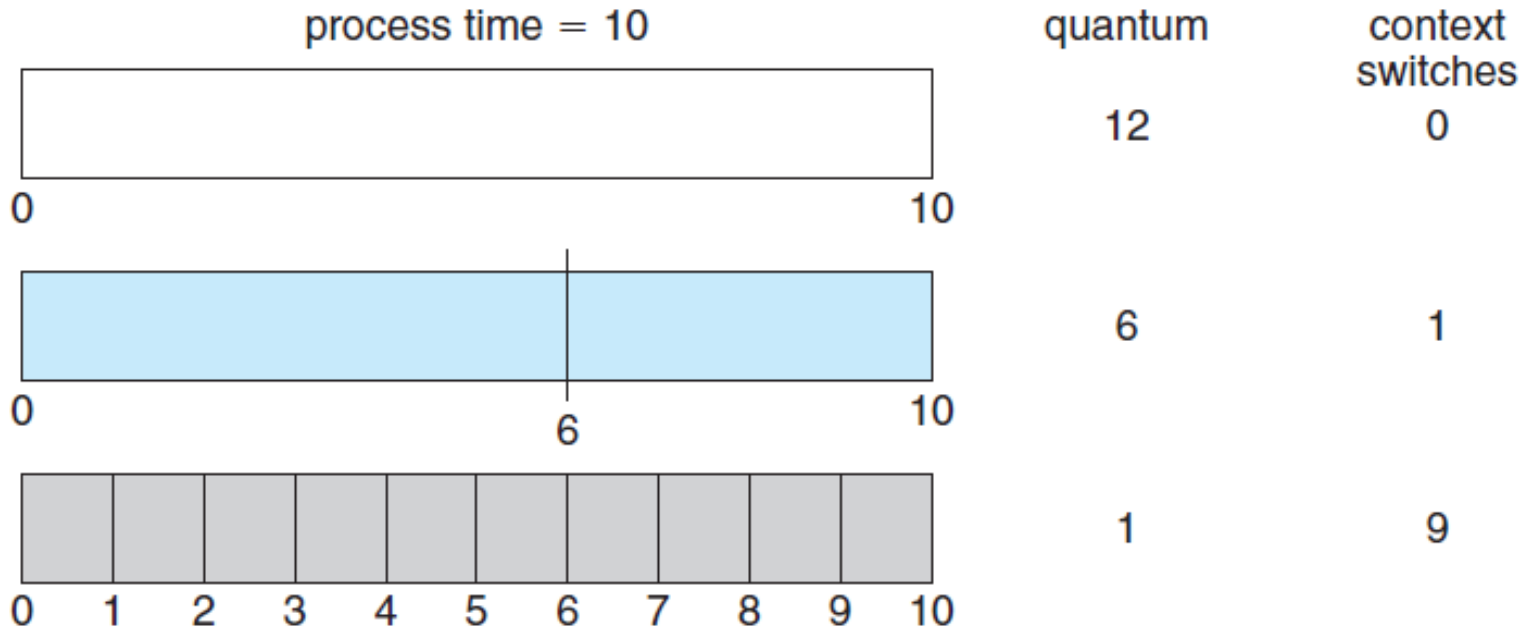
The RR scheduling algorithm is thus **preemptive**.

The performance of the RR algorithm depends heavily on the size of the time quantum.

If the time quantum is extremely large, RR turns FCFS.

Small quantum results in a large number of context switches overhead.

Example: One process of 10 time units.



Time quantum and context switches.

We want the time quantum to be large with respect to the context switch time.

If the context-switch time is 10% of the quantum, about 10% of the CPU time will be spent in context switching.

Modern systems have time quanta 10 to 100 mSec, while context switch is less than 10 µSec.

Turnaround time (from submission to completion) also depends on quantum.

The average turnaround time of a set of processes does not necessarily improve with quantum increase.

Round Robin Scheduling: Example 1

5 jobs, 100 seconds each, time slice 1 second, context switch time of 0

Job	Length	Completion Time		Wait Time	
		FCFS	Round Robin	FCFS	Round Robin
1	100				
2	100				
3	100				
4	100				
5	100				
Average					

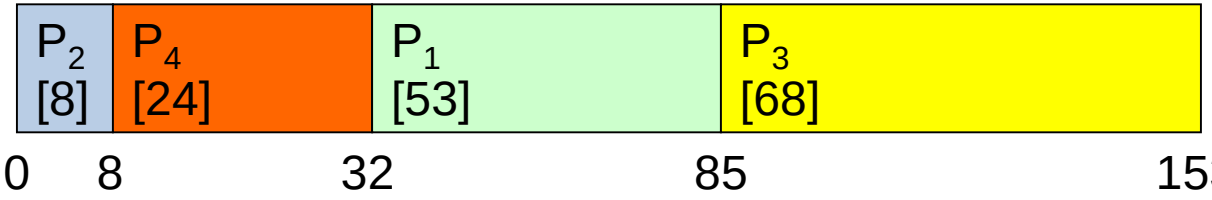
Round Robin Scheduling: Example 2

5 jobs, of length 50, 40, 30, 20, and 10 seconds each, time slice 1 second, context switch time of 0 seconds

Job	Length	Completion Time		Wait Time	
		FCFS	Round Robin	FCFS	Round Robin
1	50				
2	40				
3	30				
4	20				
5	10				
Average					

Example with Different Time Quantum

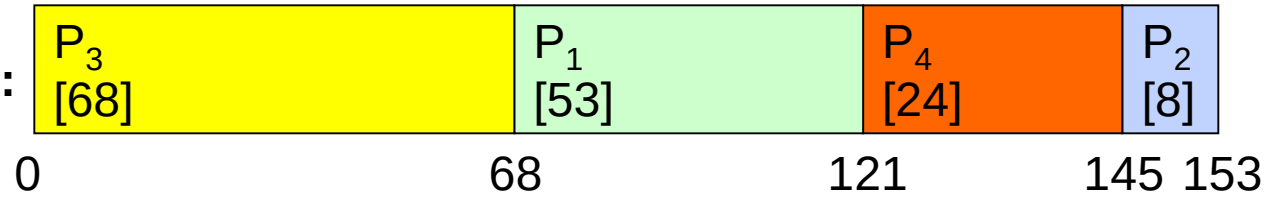
Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
Completion Time	Best FCFS	85	8	153	32	69½

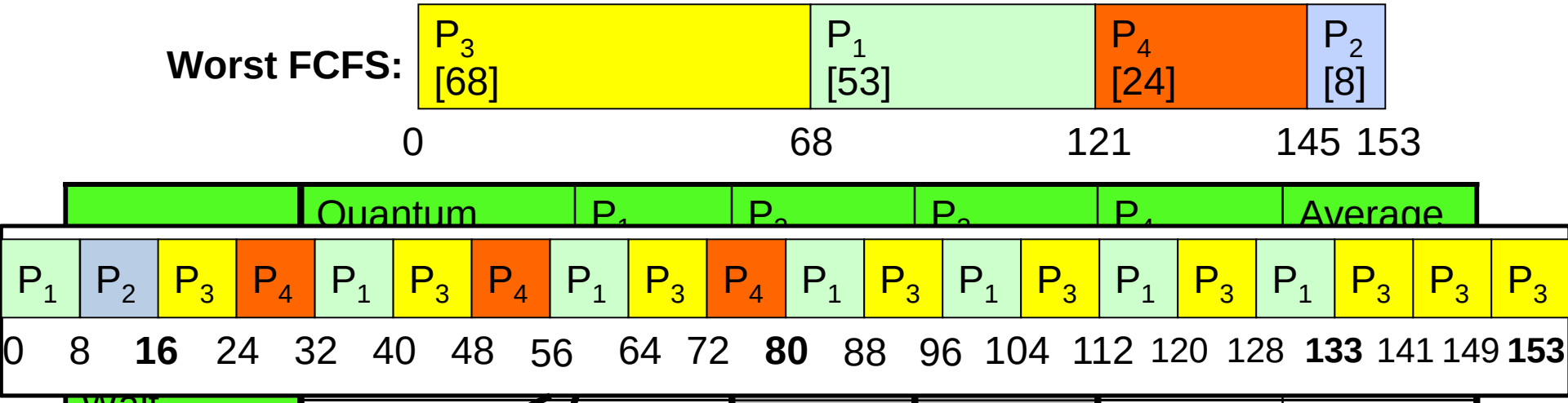
Example with Different Time Quantum

Worst FCFS:



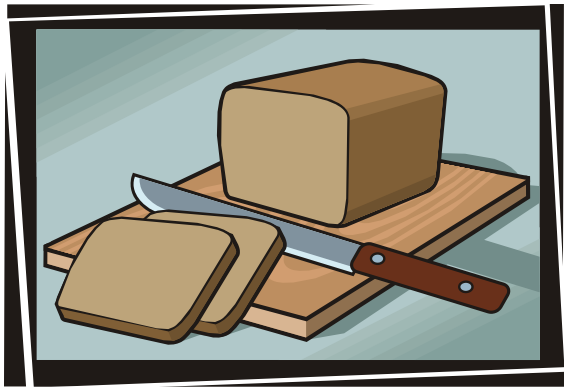
	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Worst FCFS	121	153	68	145	121¾

Example with Different Time Quantum



Wait Time	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Round-Robin -Summary



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Shortest-Job-First Scheduling

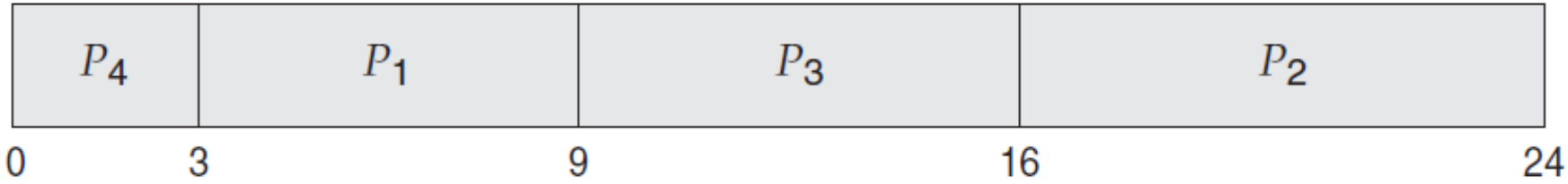
Shortest-job-first (SJF) associates with process the length of its **next CPU predicted burst**.

It assigns the CPU to the process having the **smallest** burst. In case of tie **FCFS** scheduling is used.

The following processes have length of next predicted CPU bursts in mSec.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

The SJF scheduling is:



$$\text{average waiting time} = (3 + 16 + 9 + 0)/4 = 7.$$

FCFS scheduling yields average waiting 10.25 mSec.

The SJF scheduling algorithm is provably optimal.

Problem: SJF requires knowing the length of the next CPU request.

Batch system (long-term) uses user specified time limit. SJF is used in **long-term scheduling**.

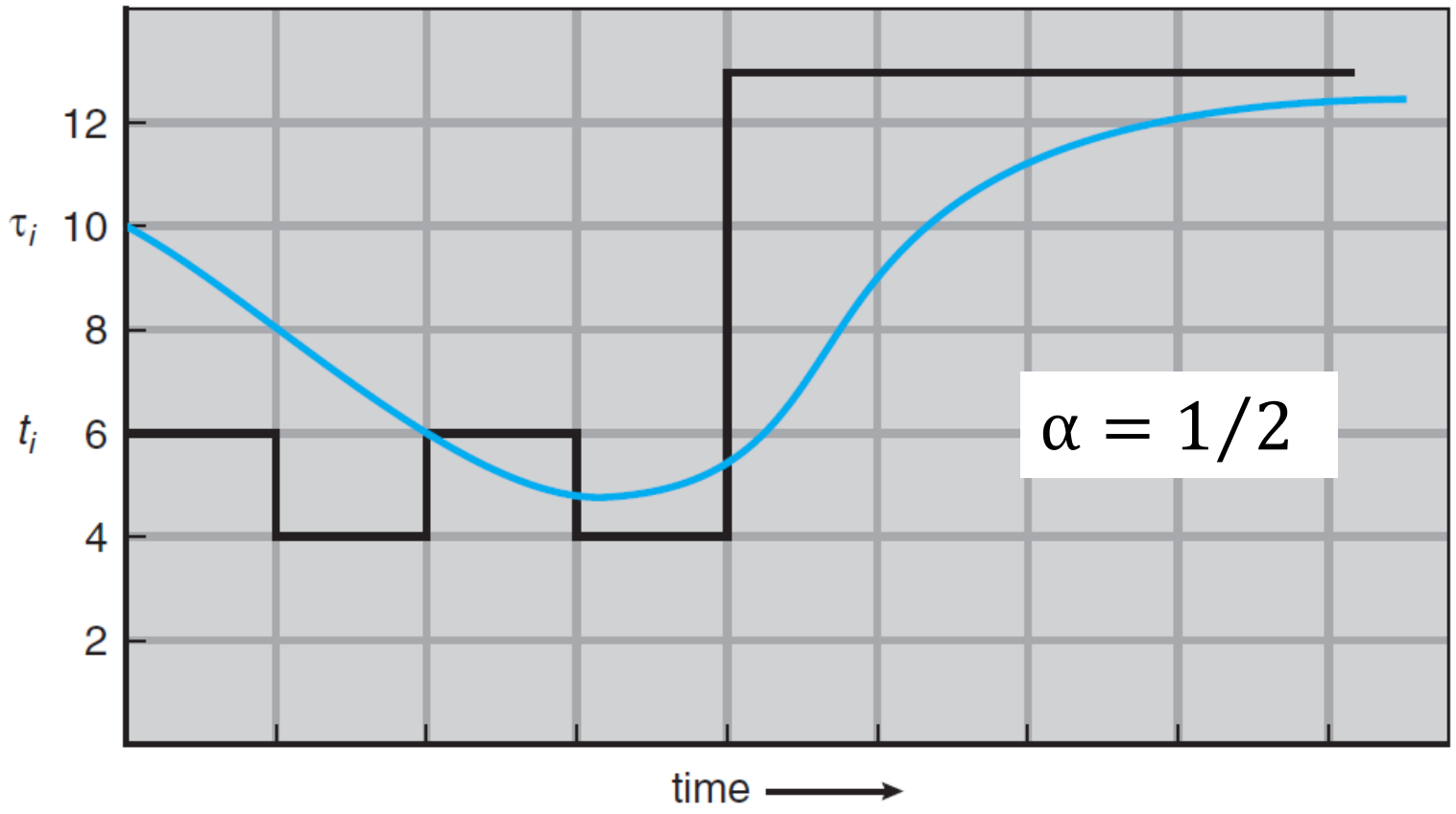
With **short-term scheduling**, there is no way to know the length of the next CPU burst and **prediction** is used.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts.

Let t_n be the length (measured) of the nth CPU burst and τ_{n+1} the prediction defined by averaging the most recent measured and predicted bursts, $0 \leq \alpha \leq 1$.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

commonly $\alpha = 1/2$.

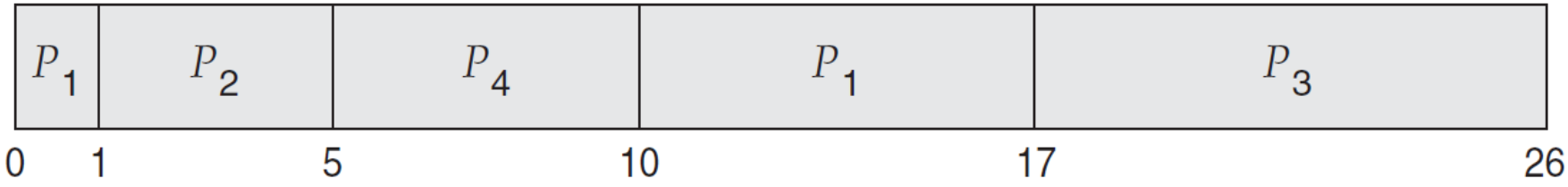


CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Example: The following processes have length of next predicted CPU bursts in mSec.

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

The **preemptive** SJF schedule is:



average waiting time:

$[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5.$

Non-preemptive SJF scheduling would result in an average waiting time of 7.75 mSec.

SJF/SRTF: Shortest Job First

Schedule the job that has the least (expected) amount of work (CPU time) to do until its next I/O request or termination.

- **Advantages:**

– Provably optimal with respect to minimizing the average waiting time

– Works for preemptive and non-preemptive schedulers

– Preemptive SJF is called SRTF - shortest remaining time first

=> I/O bound jobs get priority over CPU bound jobs

- **Disadvantages:**

– Impossible to predict the amount of CPU time a job has left

– Long running CPU bound jobs can starve

SJF: Comparison Example

5 jobs, of length 50, 40, 30, 20, and 10 seconds each, time slice 1 second, context switch time of 0 seconds

Job	Length	Completion Time			Wait Time		
		FCFS	RR	SJF	FCFS	RR	SJF
1	50	50	150		0	100	
2	40	90	140		50	100	
3	30	120	120		90	90	
4	20	140	90		120	70	
5	10	150	50		140	40	
Average		110	110		80	80	

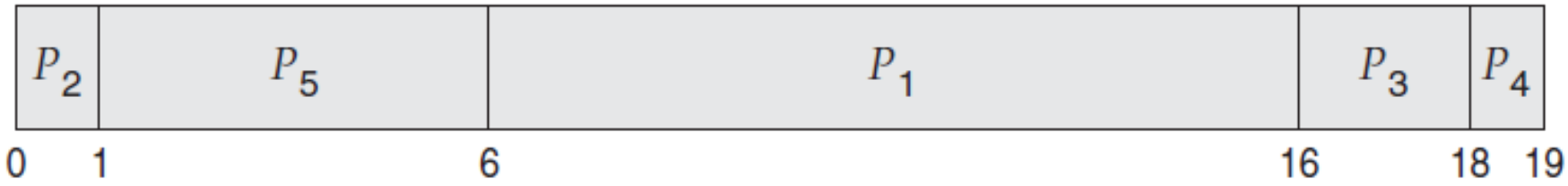
Priority Scheduling

A **priority**, e.g. $0, 1, \ldots, n$, is associated with processes. Smallest is usually the highest priority. CPU is allocated to highest priority. Ties are resolved by **FCFS**.

SJF is a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.

Example: Processes with burst time and priorities.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



Average waiting time is 8.2

Internal priorities are based on OS measures, such as time limits, memory requirements, number of open files, the ratio of average I/O burst to average CPU burst.

External priorities are set outside OS: importance, amount of funds paid, etc.

The priority of an arriving process is compared with that of the currently running.

A **preemptive** priority scheduling will preempt the CPU if the priority of the new is higher, whereas a **non-preemptive** put the new at the head of the ready queue.

Low priority processes can sometimes wait indefinitely, situation called **indefinite blocking**, or **starvation**.

A solution is by **aging**, gradually increasing the priority of processes that waiting in the system for a long time.

Example: Priorities from 127 (low) to 0 (high). Priority of waiting processes is increased by 1 every 15 minutes.

Initial priority of 127 would take 32 hours to age to 0.

Multilevel Queue Scheduling

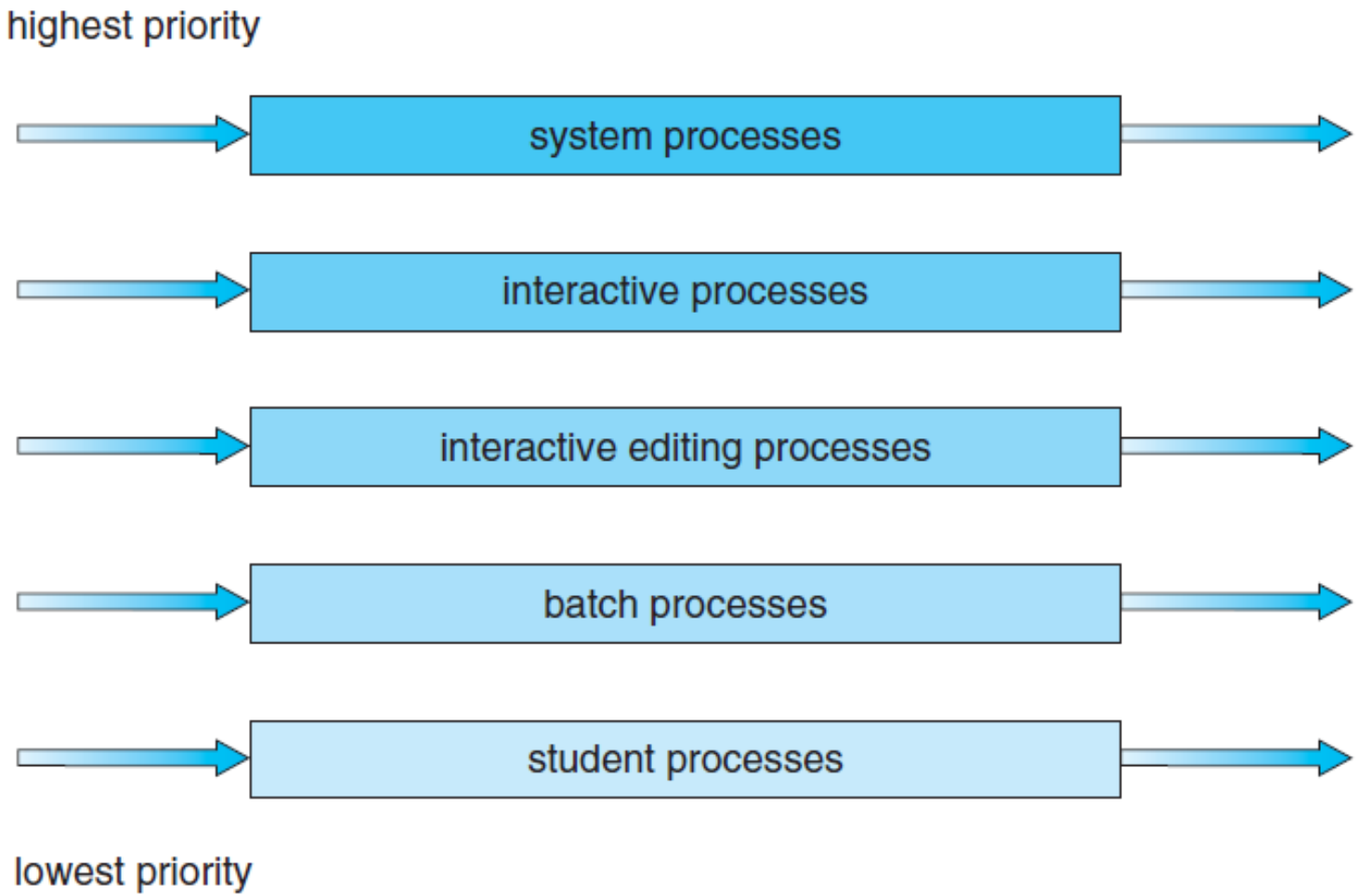
Foreground (interactive) processes and **background** (batch) processes have different response-time requirements, hence different scheduling needs.

Foreground may also have priority over **background**.

Multilevel queue scheduling partitions the ready queue into several separate queues.

Processes are assigned to queues, based on some memory size, priority, process type, etc.

Each queue has its own scheduling algorithm, e.g. **foreground** by **RR** and **background** by **FCFS**.



Multilevel queue scheduling.

Multilevel Feedback Queues (MLFQ)

- Multilevel feedback queues use past behavior to predict the future and assign job priorities

$\Rightarrow$ overcome the prediction problem in SJF

• If a process is I/O bound in the past, it is also likely to be I/O bound in the future (programs turn out not to be random.)
• To exploit this behavior, the scheduler can favor jobs that have used the least amount of CPU time, thus approximating SJF.

• This policy is **adaptive** because it relies on past behavior and changes in behavior result in changes to scheduling decisions.

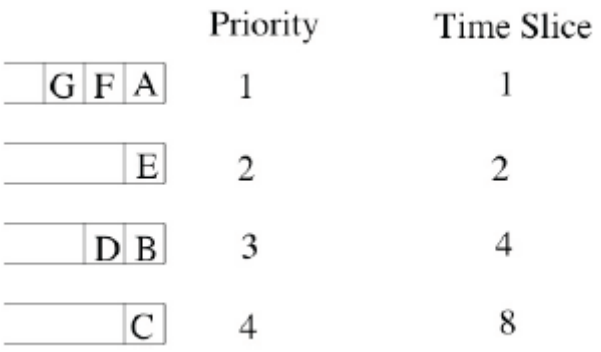
Approximating SJF: Multilevel Feedback Queues

Multiple queues with different priorities.

• Use Round Robin scheduling at each priority level, running the jobs in highest priority queue first.

• Once those finish, run jobs at the next highest priority queue, etc.
(Can lead to starvation.)

• Round robin time slice increases exponentially at lower priorities.



Adjusting Priorities in MLFQ

Job starts in highest priority queue.

- If job's time slices expires, drop its priority one level.

- If job's time slices does not expire (the context switch comes from an I/O request instead), then increase its priority one level, up to the top priority level.

⇒CPU bound jobs quickly drop in priority and I/O bound jobs stay at a high priority.

Multilevel Feedback Queues: Example 1

- 5 jobs, of length 30, 20, and 10 seconds each, initial time slice 1 second, context switch time of 0 seconds, all CPU bound (no I/O), 3 queues

Job	Length	Completion Time		Wait Time	
		RR	MLFQ	RR	MLFQ
1	30	60		30	
2	20	50		30	
3	10	30		20	
Average		46 2/3		26 2/3	

Queue	Time Slice	Job
1	1	
2	2	
3	4	

Lottery Scheduling

Give every job some number of lottery tickets.

- On each time slice, randomly pick a winning ticket.

- On average, CPU time is proportional to the number of tickets given to each job.

- Assign tickets by giving the most to short running jobs, and fewer to long running jobs (approximating SJF). To avoid starvation, every job gets at least one ticket.

- Degrades gracefully as load changes. Adding or deleting a job affects all jobs proportionately, independent of the number of tickets a job has.

Lottery Scheduling: Example

Short jobs get 10 tickets, long jobs get 1 ticket each.

# short jobs/ # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	91%	9%
0/2		
2/0		
10/1		
1/10		

Lottery Scheduling Example

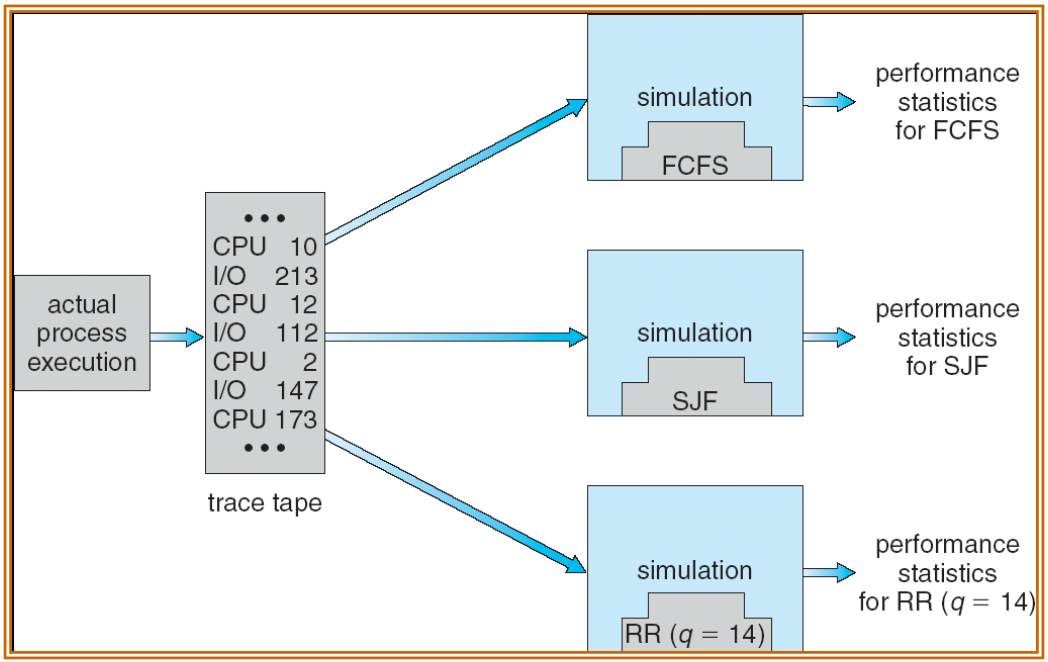
- Lottery Scheduling Example
  - short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable response time?
    - In UNIX, if load average is 100, hard to make progress
    - One approach: log some user out

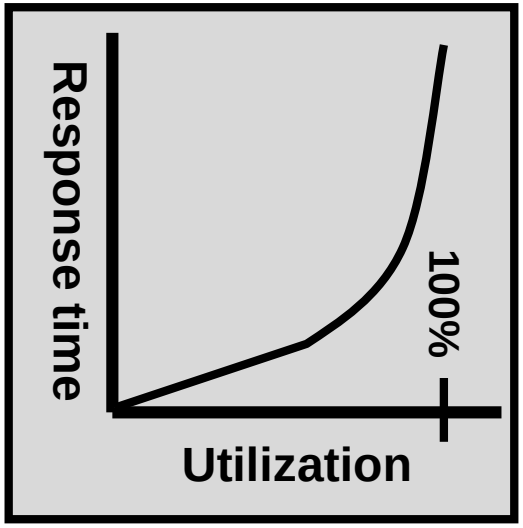
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - Takes a predetermined workload and compute the performance of each algorithm for that workload
- Queuing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.

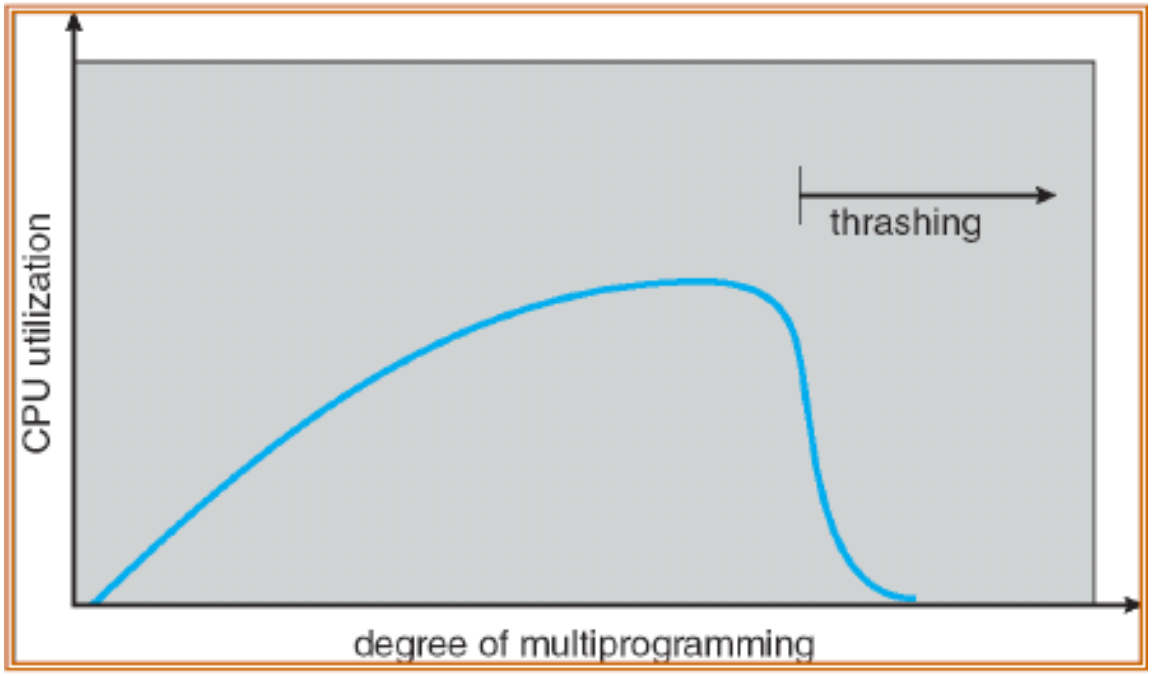


A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve



Thrashing



If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

–low CPU utilization

–operating system spends most of its time swapping to disk

Summary

- Scheduling: selecting a process from the ready queue and allocating the CPU to it

- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)

Cont.

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

- Lottery Scheduling:
  - Give each thread a number of tokens (short tasks $\Rightarrow$ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness