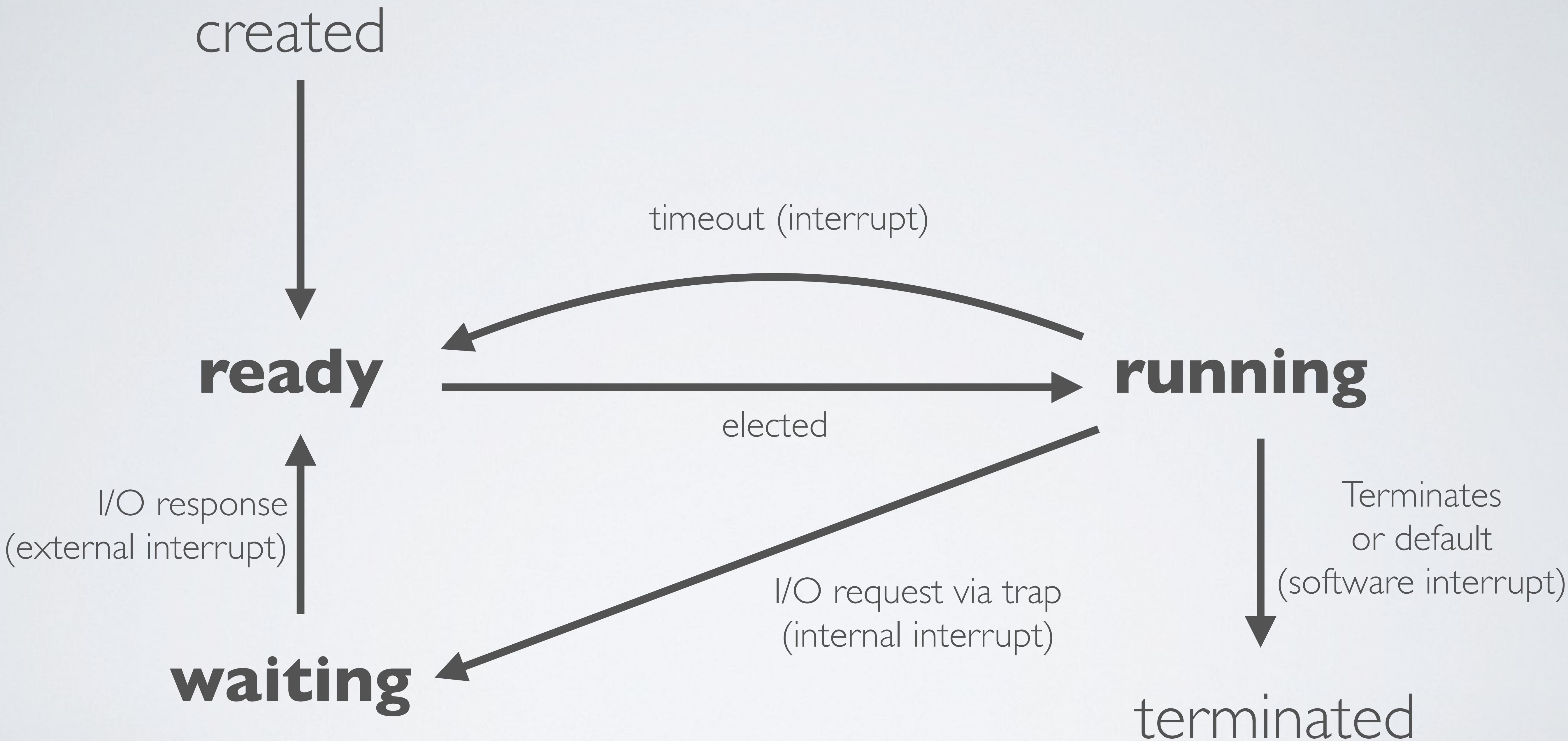
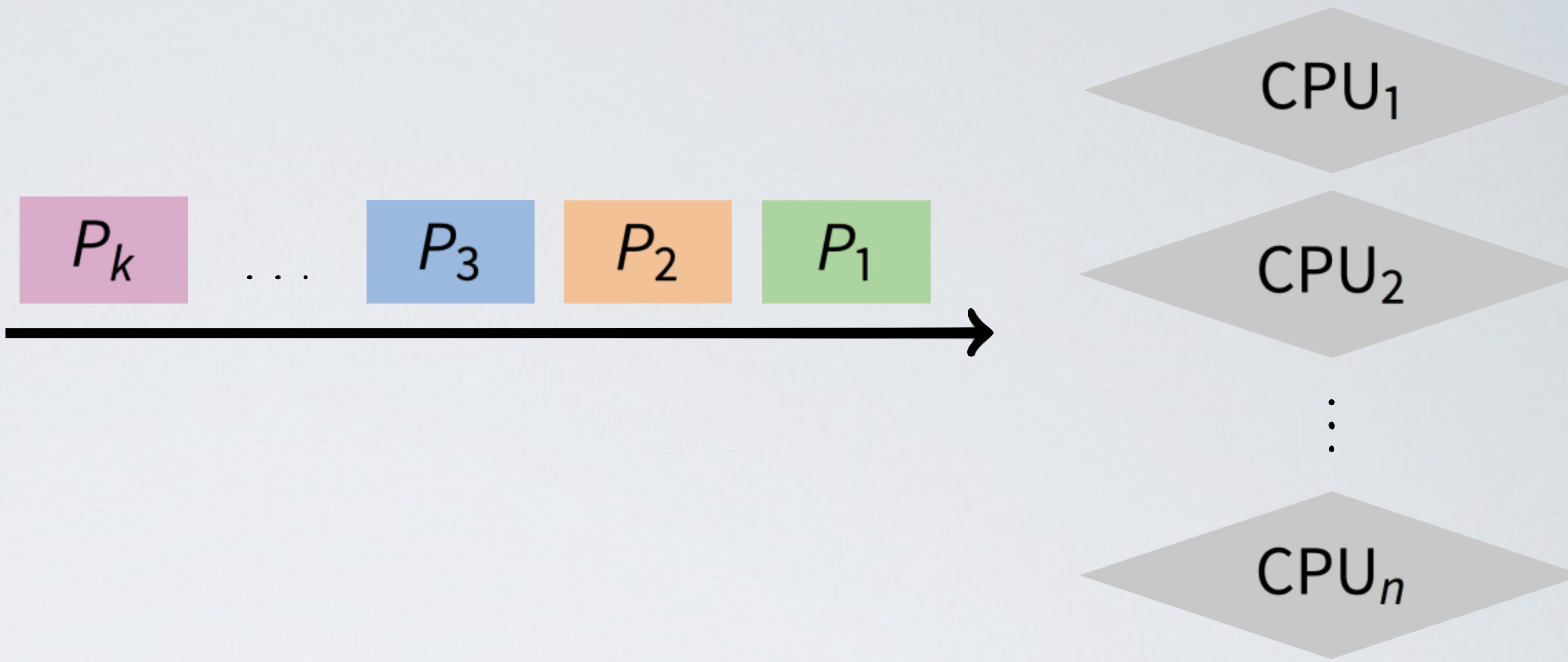
Scheduling

Thierry Sans

(recap) The different states of a thread



The scheduling problem



- n threads ready to run
- k ≥ 1 CPUs

➡ Scheduling Policy
which jobs should we assign to which CPU(s)?
and for how long?

Non Goals : Starvation

Starvation is when a thread is prevented from making progress because some other thread has the resource it requires (could be CPU or a lock)

➡ Starvation is usually a side effect of the scheduling algorithm

- e.g a high priority thread always prevents a low priority thread from running

➡ Starvation can be a side effect of synchronization (forthcoming lecture)

- e.g constant supply of readers always blocks out writers

Scheduling Criteria

- **Throughput** – # of threads that complete per unit time
  *# jobs/time (*Higher is better)
- **Turnaround time** – time for each thread to complete
  Tfinish – Tstart (Lower is better)
- **Response time** – time from request to first response ()
  i.e. time between waiting to ready transition and ready to running transition
  Tresponse – Trequest (Lower is better)

➡ Above criteria are affected by secondary criteria

- CPU utilization – %CPU fraction of time CPU doing productive work
- Waiting time – Avg(Twait) time each thread waits in the ready queue

How to balance criteria?

- **Batch systems** (supercomputers)
  strive for job throughput and turnaround time

- **Interactive systems** (personal computers)
  strive to minimize response time for interactive jobs

However, in practice, users prefer predictable response time over faster but highly variable response time

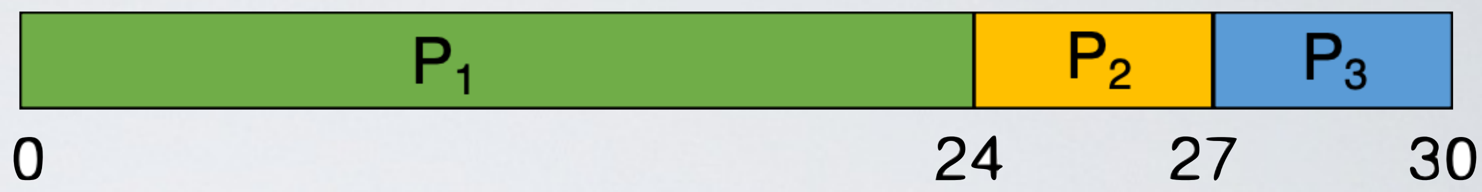
Often optimized for an average response time

Two kinds of scheduling algorithm

- **Non-preemptive scheduling** (good for batch systems) once the CPU has been allocated to a thread, it keeps the CPU until it terminates

- **Preemptive scheduling** (good for interactive systems) CPU can be taken from a running thread and allocated to another

FCFS - First Come First Serve (non-preemptive)



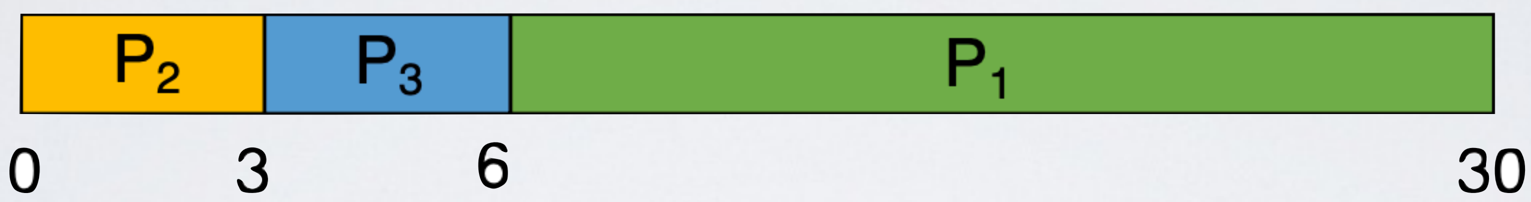
➡ Run jobs in order that they arrive (no interrupt)

Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(24 + 27 + 30) / 3 = 27 sec in average
Waiting Time	(0 + 24 + 27) / 3 = 17 sec in average

- **Problem : convoy effect**
  all other threads wait for the one big thread to release the CPU

SJF - Shortest-Job-First (non-preemptive)

➡ Choose the thread with the shortest processing time



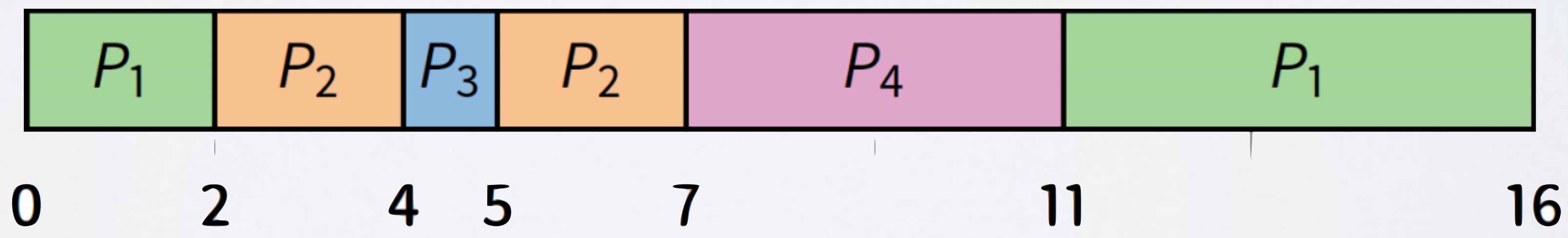
Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(30 + 3 + 6) / 3 = 13 sec in average
Waiting Time	(0 + 3 + 6) / 3 = 3 sec in average

◉ **Problem :** we need to know processing time in advance

SRTF - Shortest-Remaining-Time-First (preemptive)

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

➡ if a new thread arrives with CPU burst length less than remaining time of current executing thread, preempt current thread



✓ **Good :** optimize waiting time

◉ **Problem :** can lead to starvation

RR - Round Robin (preemptive)

➡ Each job is given a time slice called a quantum, preempt job after duration of quantum, move to back of FIFO queue

✓ **Good :** fair allocation of CPU, low waiting time (interactive)

◉ **Problem :** no priority between threads

Time Quantum

➡ Context switches are frequent and need to be very fast

- How to pick quantum?
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum - But not so large system reverts to FCFS

✓ Typical values: 1–100 ms

Why having priorities?

- ✓ Optimize job turnaround time for "batch" jobs
- ✓ Minimize response time for "interactive" jobs

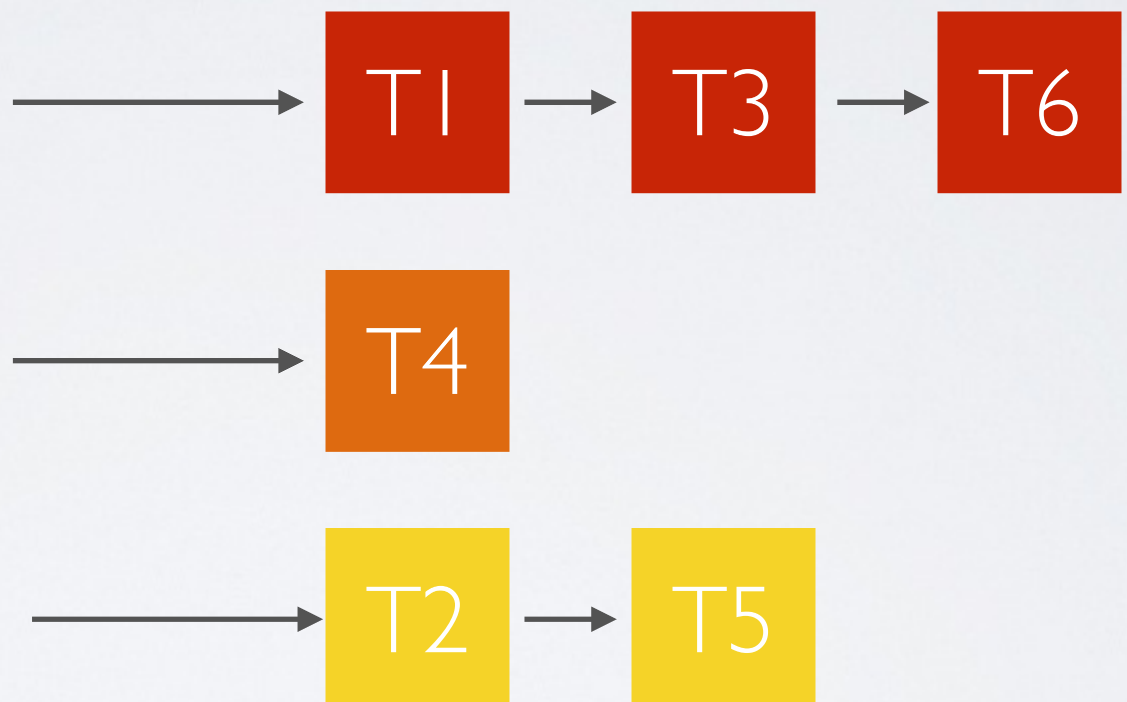
MLQ - Multilevel Queue Scheduling (preemptive)

➡ Associate a priority with each thread and execute highest priority thread first. If same priority, do round-robin.

high-priority queue
(e.g system thread)

medium priority
(eg. user thread)

low priority
(e.g background thread)



- **Problem 1 :** starvation of low priority thread
- **Problem 2 :** (possibly) starvation of high priority thread
- **Problem 3 :** how to decide on the priority?

MLQ - Starvation of high priority thread

1. T1 (low priority) starts, runs and acquires the lock `l`
2. T2 (medium priority) starts, preempts the CPU and runs
3. T3 (high priority) starts, preempts the CPU, runs
   but gets blocked while trying to acquire the lock `l`
4. T2 is elected to run (highest priority thread to be ready to run)

- **Problem** : starvation of a high priority thread
- ✓ **Solution** : priority donation

MLQ - Priority donation (simple example)

1. T1 (low priority) starts, runs and acquires the lock `l`
2. T2 (medium priority) starts, preempts the CPU and runs
3. T3 (high priority) starts, preempts the CPU, runs
   but gets blocked while trying to acquire the lock `l`
4. T3 gives its high priority to T1
5. T1 (now high priority) runs, releases the lock and returns to low priority immediately after
6. T3 (now unblocked) preempts the CPU and runs

Solutions to other MLQ problems

➡ **To prevent starvation of low priority thread**
change the priority over time by either

- increase priority as a function of waiting time
- or decrease priority as a function of CPU consumption

➡ **To decide on the priority**
by observing and keeping track of the thread CPU usage

MLFQ - Multilevel **Feedback** Queue Scheduling (preemptive)

➡ Same as MLQ but change the priority of the process based on *observations*

Rule 1	If Priority(A) > Priority(B), A runs
Rule 2	If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue
Rule 3	When a job enters the system, it is placed at the highest priority (the topmost queue)
Rule 4	Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)
Rule 5	After some time period S, move all the jobs in the system to the topmost queue

✓ **Good :** Turing-award winner algorithm

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

source: Wikipedia - Scheduling (Computing)
https://en.wikipedia.org/wiki/Scheduling_(computing)

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*