# CS202 (003): Operating Systems
# Scheduling

Instructor: Jocelyn Chen

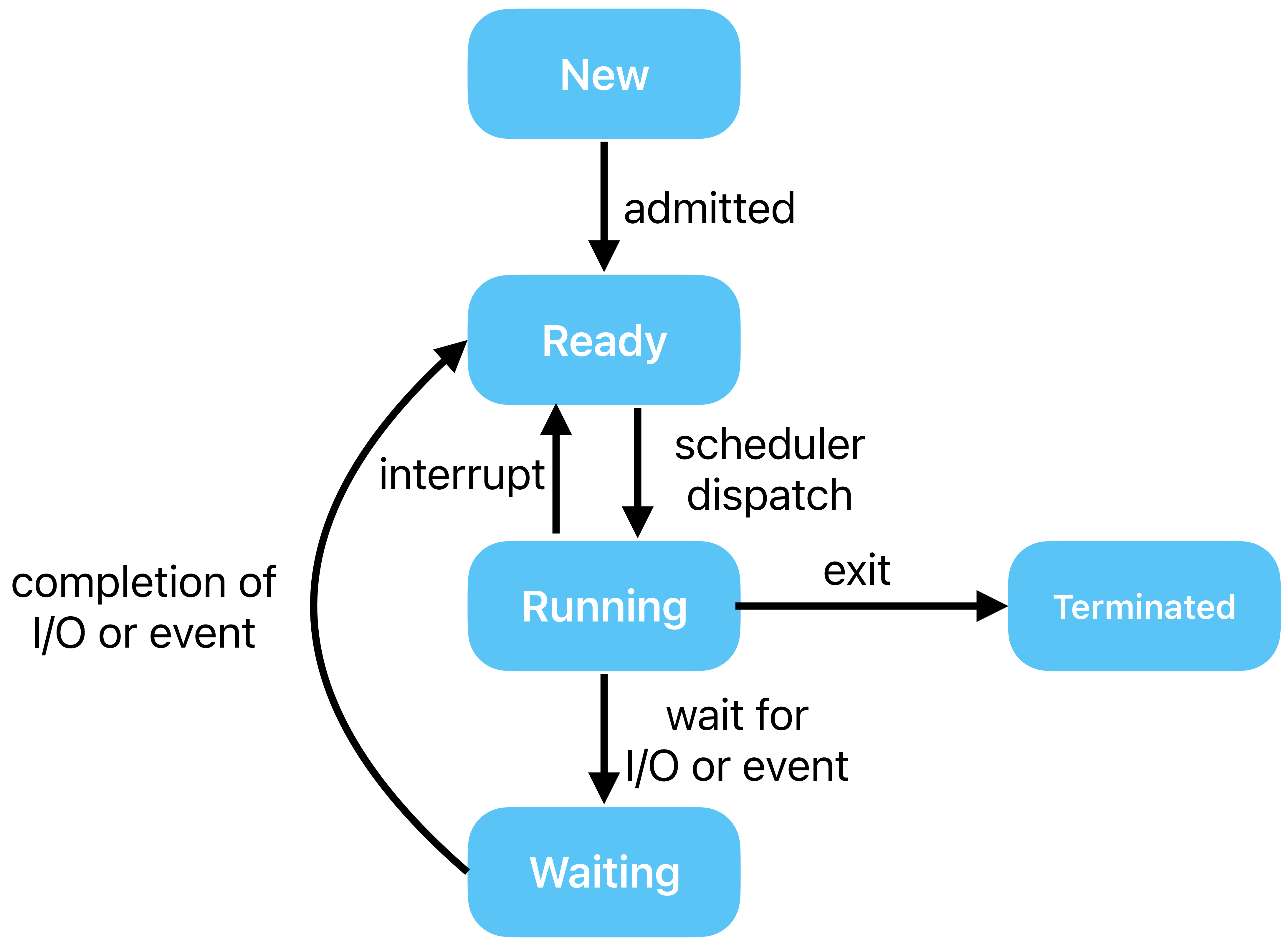
Most of the materials covered in this slide come from the lecture notes of Mike Walfish’s CS202

# Last Time

# Have you ever wondered how we decide what next process/thread to run?

Operating system has to decide on this!

# When scheduling decisions happen




(i) switches from running to waiting state
(ii) switches from running to ready state
(iii) switches from waiting to ready
(iv) exits

**Preemptive scheduling**
willing to stop one process from running in order to run another

**(i), (ii), (iii), (iv)**

**Non-preemptive scheduling**
run each job to completion before considering whether to run a new job

**(i), (iv)**

# What are the metrics and criteria for making decisions?

**Turnaround time**
Time for each process to complete (from arrival)

**Waiting/Response/Output time**
Time spent waiting for something to happen

**Response time:** time between when jobs enters system and starts executing

**Output time:** time from request to first response

**System throughout**
# of processes that complete per unit time

**Fairness**
(different possible definitions )

Free from starvation

All users get equal time on CPU

Highest priority jobs get most of CPU

……

# We call …

Stopping one running process temporality and resuming (or starting) another process

Context Switch

Context switching has a **cost**!

**CPU time in kernel:** save/restore registers, switch address spaces

**Indirect cost:** TLB shootdown, processor caches, OS caches

More frequent context switches will lead to worse throughput (higher overhead)

# Scheduling disciplines (without I/O)

FCFS/FIFO

SJF and STCF

Round-robin (RR)

# FCFS/FIFO

Run each job until it's done

| Job | Time Needed (s) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |


**Throughput** = $\frac{3 \text{ jobs}}{30 \text{ seconds}} = 0.1 \text{ jobs/second}$

**Avg Turnaround Time** = $\frac{24 + 27 + 30}{3} = 27$

How can we lower avg turnaround time?


Advantages

- simple
- no starvation
- few context switches

Disadvantages

- short jobs get stuck behind long ones!

# SJF and STCF

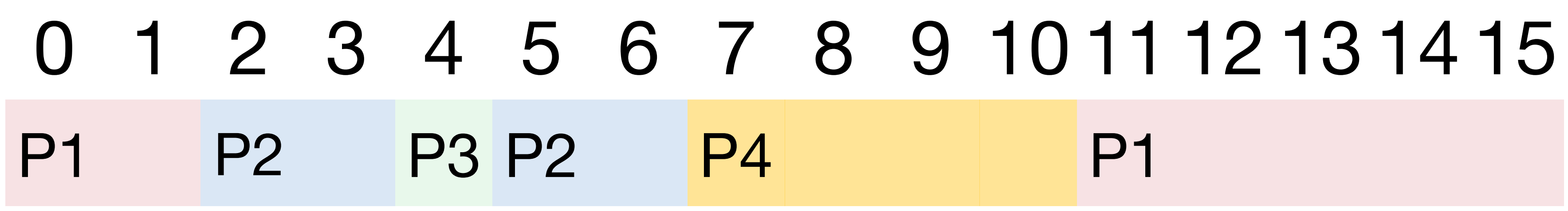
**SJF**

Schedule the job whose next CPU burst is the shortest

**STCF**

Preemptive version of SJF: if the new job arrived has a shorter time to completion than the remaining time on the current job, immediately preempt CPU to give to new job

| Job | Arrival Time (s) | Burst Time (s) |
|---|---|---|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |




**Advantages**

- Discuss later!

**Disadvantages**

- Discuss later!

# Round Robin

Let's start considering response time
(i.e., we are adding a timer our scheduler)

**Preempt CPU from long-running jobs (per time slice/quantum)**
=> if a job hasn't finished by the end of a time slice,
put it to the back of the ready queue

Advantages

- Fair allocation of CPU across jobs
- Low average **response time** when job length vary
- Good for output time if small number of jobs

Disadvantages

- RR does not care about turnaround time!

| Job | Time Needed (in time unit) |
|---|---|
| P1 | 50 |
| P2 | 50 |

What is the average turnaround time if we have quantum of 1?

**100.5**

What happens if we use FIFO?

**75**

# Round Robin

Let's start considering response time
(i.e., we are adding a timer our scheduler)

**Preempt CPU from long-running jobs (per time slice/quantum)**
=> if a job hasn't finished by the end of a time slice,
put it to the back of the ready queue

Advantages

- Fair allocation of CPU across jobs
- Low average **response time** when job length vary
- Good for output time if small number of jobs

Disadvantages

- RR does not care about turnaround time!

How do we choose quantum size?

- Want much larger than context switch cost (amortization)
- Majority of bursts should be less than quantum
- If too small -> spend too much time context switching
- If too large -> response time suffers (and reverts to FIFO)

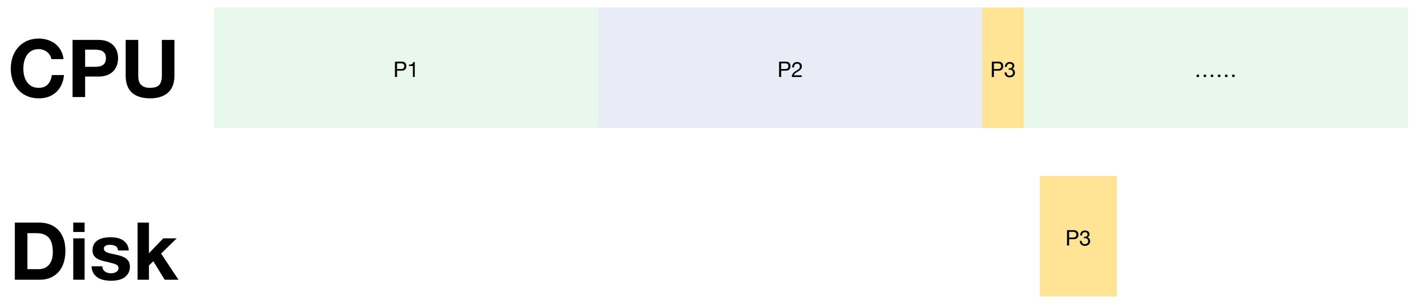
# Scheduling disciplines (**with** I/O)

| Job | Time Needed |
|---|---|
| P1 | CPU-bound, 1 week |
| P2 | CPU-bound, 1 week |
| P3 | I/O bound, loop: 1ms CPU, 10ms Disk I/O |

**FCFS/FIFO**

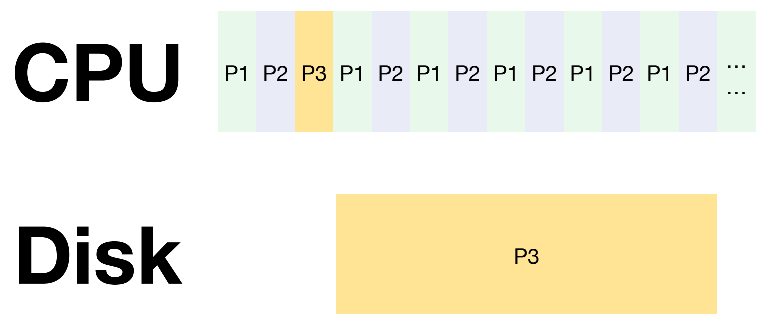
P1+P2 will take 2 weeks

**RR (100ms quantum)**



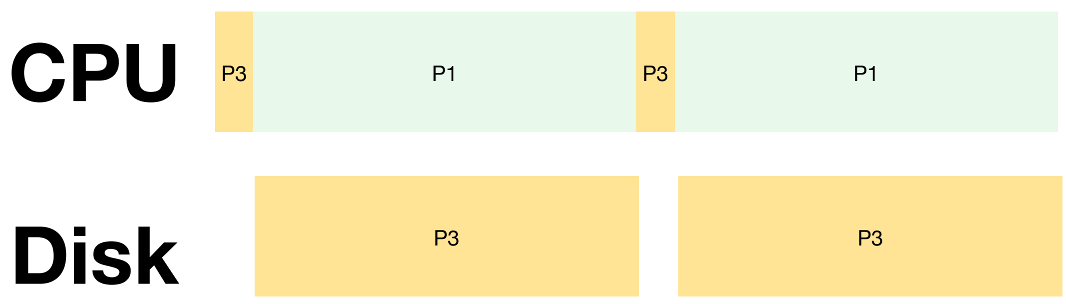

**Disk Utilization** = $\frac{10\text{ms}}{201\text{ms}} \approx 5\,\%$

**RR (1ms quantum)**




**Disk Utilization** = $\frac{10\text{ms}}{11\text{ms}} \approx 91\,\%$

**STCF**




Good disk utilization

Optimal average turnaround time

Low overhead

# SJF and STCF

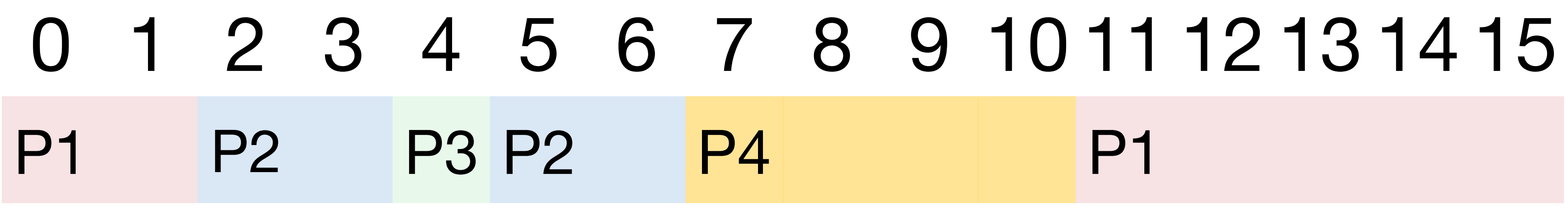
**SJF**

Schedule the job whose next CPU burst is the shortest

**STCF**

Preemptive version of SJF: if the new job arrived has a shorter time to completion than the remaining time on the current job, immediately preempt CPU to give to new job

| Job | Arrival Time (s) | Burst Time (s) |
|---|---|---|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |




**Advantages**

- Good disk utilization
- Optimal (minimum) average turnaround time
- Low overhead (no needless preemption)

**Disadvantages**

- Long-running jobs get starved
- Does not optimize response time
- Requires predicting the future

# Predicting CPU burst: EWMA

## (exponentially weighted average)

Attempt to estimate future based on the past

$t_n$ :**(time) length of proc's** $n^{\text{th}}$ **burst**

$\tau_{n+1}$: **estimate for** $n+1$ **burst**

$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$ **where** $0 < \alpha \leq 1$

Favor jobs that have been using CPU the least amount of time

# Key idea in scheduling: Priority

Give every process a number, and give the CPU to the process with highest priority (which is either the highest/lowest numbers)

We don't want to use strict priority (that leads to starvation on low priority tasks)

To reduce starvation, we can increase a process's priority as it waits

# Optimizing turnaround + response time: MLFQ

## (multi-level feedback queue)

Multiple queues, each with different priority

RR within each queue

Processes priority changes overtime

Advantages

- Approximate SRTCF (shortest remaining time first)
- It overall gives higher priority that use less CPU time
- Helps reduce average turnaround time and response time for short jobs

Disadvantages

- Cannot donate priority
- Not very flexible
- Not good for real-time and multimedia
- Can be gameable

# Another way of optimization: fair-share scheduler

Try to guarantee that each job obtain a certain percentage of CPU time

# Lottery scheduling

Tickets: the share of a resource that a process should receive
*The percent of tickles that a process has represents its share of the system resources*

Hold a lottery to determine which process should get to run next, every now and then

**Let $p_i$ has $t_i$ tickets**

**Let $T$ be total # of tickets,** $T = \sum_i t_i$

**Chance of winning the next quantum** $= \frac{t_i}{T}$

Control long-term average proportion of CPU for each process!

# Lottery scheduling

Hold a lottery to determine which process should get to run next, every now and then

Advantages

- Deals with starvation (if you have ticket, you will make progress)
- Don't worry that adding one high priority job will starve all others
- Adding/deleting jobs affects all jobs proportionally
- Can transfer tickets between processes
- Flexible by using ticket as a currency

Disadvantages

- Latency is unpredictable
- Expected error somewhat high

Follow-up work to reduce randomness -> Stride Scheduling (see textbook for details)

# What Linux does: completely fair scheduler (CFS)

It aims to distribute CPU time fairly among all runnable processes using a virtual runtime metric.

CFS organizes processes in a red-black tree and selects the one with the lowest virtual runtime to run next. This approach balances fairness, efficiency, and interactivity.

See the textbook for more details

# Scheduling, lesson learned

Write down your goals (**policy**) before picking the scheduling algorithm (**mechanism**)

Start from/Compare with the optimal solution, even though it cannot be built

Many schedulers in the system that interact:
mutex, interrupt, disk, network, …