

Scheduling I

- Scheduling algorithms

Dispatcher vs. Scheduler

□ Dispatcher

- Low-level mechanism
- Responsibility: context switch
 - `context_switch()` in Linux kernel

□ Scheduler

- High-level policy
- Responsibility: deciding which process to run
 - `pick_next_task()` in Linux kernel

Scheduling performance metrics

- ❑ **Min waiting time:** don't have process wait long in ready queue
- ❑ **Max CPU utilization:** keep CPU busy
- ❑ **Max throughput:** complete as many processes as possible per unit time
- ❑ **Min response time:** respond immediately
- ❑ **Fairness:** give each process (or user) same percentage of CPU

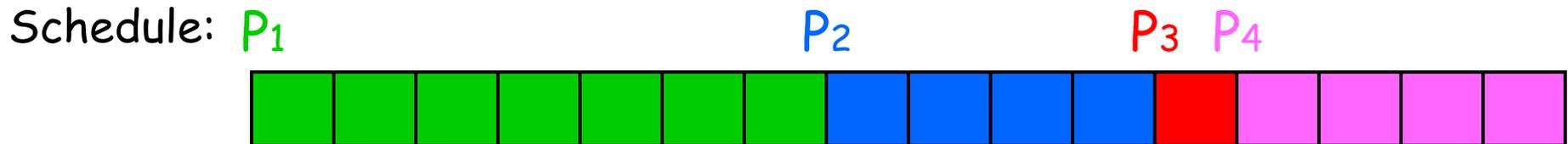
First-Come, First-Served (FCFS)

- Simplest CPU scheduling algorithm
 - First job that requests the CPU gets the CPU
 - Nonpreemptive
- Implementation: FIFO queue

Example of FCFS

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	7
P_2	0	4
P_3	0	1
P_4	0	4

□ Gantt chart

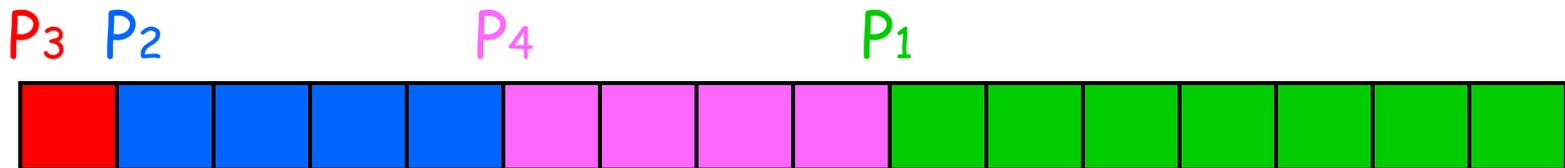


□ Average waiting time: $(0 + 7 + 11 + 12)/4 = 7.5$

Example of FCFS: different arrival order

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	7
P_2	0	4
P_3	0	1
P_4	0	4

Arrival order: P_3 P_2 P_4 P_1



□ Average waiting time: $(9 + 1 + 0 + 5)/4 = 3.75$

FCFS advantages and disadvantages

□ Advantages

- Simple
- Fair

□ Disadvantages

- waiting time depends on arrival order
- **Convoy effect**
 - Short process stuck waiting for long process
 - Also called **head of the line blocking**

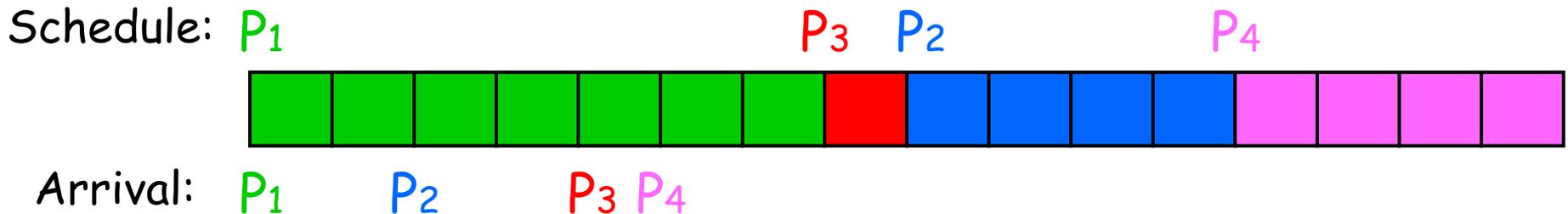
Shortest Job First (SJF)

- Schedule the process with the shortest time
- FCFS if same time

Example of SJF (w/o preemption)

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

□ Gantt chart



□ Average waiting time: $(0 + 6 + 3 + 7)/4 = 4$

SJF Advantages and Disadvantages

□ Advantages

- Minimizes average wait time.
- Provably optimal if no preemption allowed

□ Disadvantages

- Not practical: difficult to predict burst time
 - Possible: past predicts future
- May starve long jobs

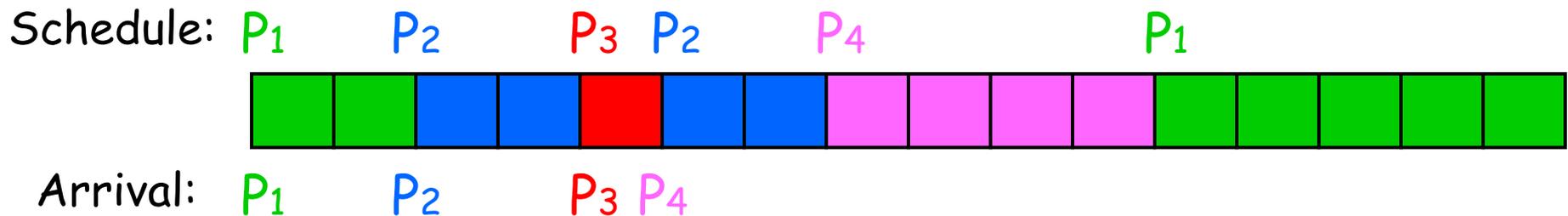
Shortest Remaining Time First (SRTF)

- If new process arrives w/ shorter CPU burst than the remaining for current process, schedule new process
- Also known as:
 - SJF with preemption
 - Shortest Time-to-Completion First (STCF)
- **Advantage:** reduces average waiting time
 - **Provably optimal**

Example of SRTF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

□ Gantt chart



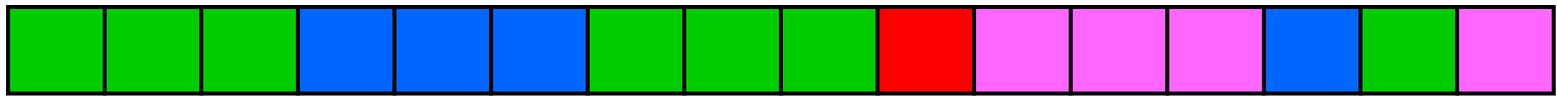
□ Average waiting time: $(9 + 1 + 0 + 2)/4 = 3$

Round-Robin (RR)

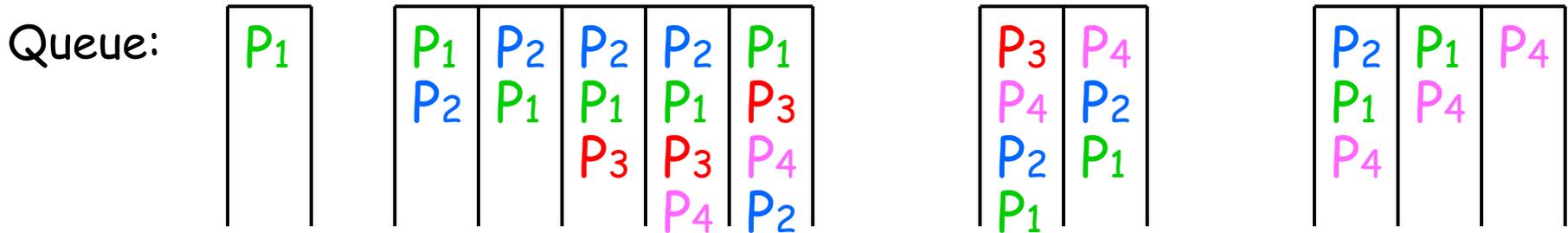
- ❑ Process runs for a predetermined time slice, and then moves to back of queue
- ❑ Process gets preempted at the end of time slice
- ❑ How long should the time slice be?

Example of RR: time slice = 3

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



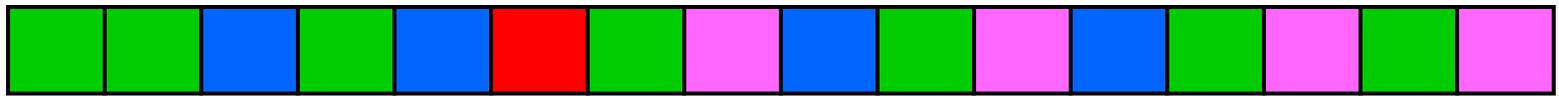
Arrival: P₁ P₂ P₃ P₄



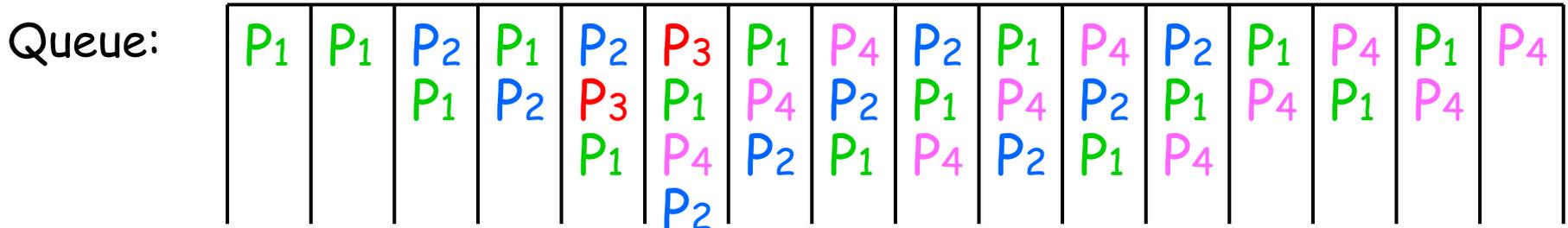
- Average waiting time: $(8 + 8 + 5 + 7)/4 = 7$
- Average response time: $(0 + 1 + 5 + 5)/4 = 2.75$
- # of context switches: 7

Smaller time slice = 1

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



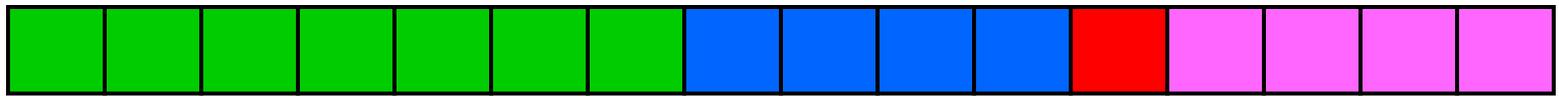
Arrival: P₁ P₂ P₃ P₄



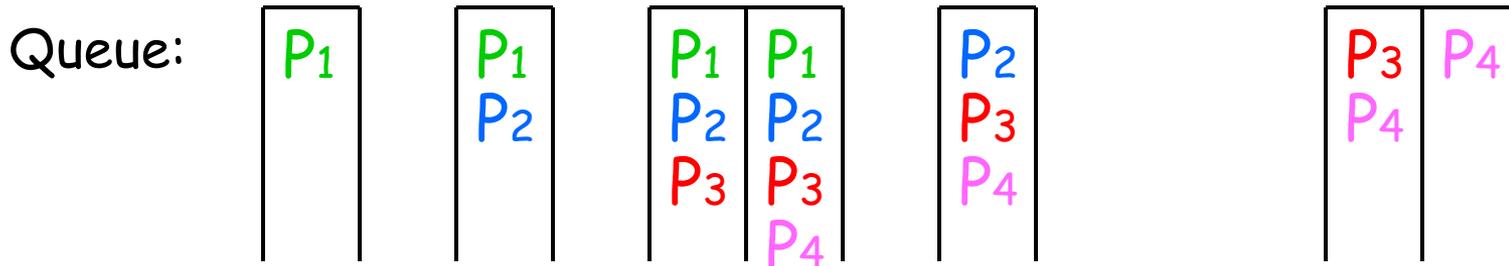
- Average waiting time: $(8 + 6 + 1 + 7)/4 = 5.5$
- Average response time: $(0 + 0 + 1 + 2)/4 = 0.75$
- # of context switches: 14

Larger time slice = 10

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



Arrival: P₁ P₂ P₃ P₄



- ❑ Average waiting time: $(0 + 5 + 7 + 7)/4 = 4.75$
- ❑ Average response time: same
- ❑ # of context switches: 3 (minimum)

RR advantages and disadvantages

□ Advantages

- Low response time, good interactivity
- Fair allocation of CPU across processes
- Low average waiting time **when job lengths vary widely**

□ Disadvantages

- Poor average waiting time when jobs have similar lengths
 - **Average waiting time is even worse than FCFS!**
- Performance depends on **length of time slice**
 - Too high → degenerate to FCFS
 - Too low → too many context switches, costly

Priorities

- Priority is associated with each process
 - Run highest priority process that is ready
 - Round-robin among processes of equal priority
- Priority can be statically assigned
 - Some always have higher priority than others
- Priority can be dynamically changed by OS
 - Aging: increase the priority of processes that wait in the ready queue for a long time

```
for (pp = proc; pp < proc+NPROC; pp++) {  
    if (pp->prio != MAX)  
        pp->prio++;  
    if (pp->prio > curproc->prio)  
        reschedule();  
}
```

Code from
6th Edition UNIX
circa 1976

Priority inversion

- High priority process depends on low priority process (e.g. to release a lock)
 - Another process with in-between priority arrives?

P1 (low): lock(my_lock) (gets my_lock)

P2(high): lock(my_lock)

P3(medium): while (...) {}

P2 waits, P3 runs, P1 waits

P2's effective priority less than P3!

- Solution: **priority inheritance**
 - Inherit highest priority of waiting process
 - Must be able to chain multiple inheritances
 - Must ensure that priority reverts to original value
- Google for "mars pathfinder priority inversion"

Multi-Level Feedback Queue (MLFQ)

- Processes move between queues
 - Queues have different priority levels
 - Priority of process changes based on observed behavior
- MLFQ scheduler parameters:
 - number of queues
 - scheduling algorithms for each queue
 - when to upgrade a process
 - when to demote a process
 - which queue a process will start in

MLFQ example from OSTEP book

- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't)
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR using the time slice of the queue
- **Rule 3:** When a job enters the system, it starts in the topmost queue (of the highest priority)
- **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