Process Scheduling I

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Outline

- Introduction to scheduling
- Scheduling algorithms
- Real time Scheduling
- Evaluation

Direction within course

- Until now: interrupts, processes, address spaces, threads, synchronization
  - Mostly mechanisms

- From now on: resources
  - Resources: things processes operate upon
    - E.g., CPU time, memory, disk space
  - Policies play a more important role

Types of resources

- Preemptible
  - – OS can take resource away, use it for something else, and give it back later
    - E.g., CPU
- Non-preemptible
  - – OS cannot easily take resource away; have to wait after the resource is voluntarily relinquished
    - E.g., disk space
- Type of resource determines how to manage

Decisions about resource

- Allocation: which process gets which resources
  - Which resources should each process receive?
  - Space sharing: Controlled access to resource through indirection
  - Implication: resources are not easily preemptible

- Scheduling: how long process keeps resource
  - In which order should requests be serviced?
  - Time sharing: more resources requested than can be granted
  - Implication: resource is preemptible

Role of Dispatcher vs. Scheduler

- Dispatcher
  - Low-level mechanism
  - Responsibility: context switch

- Scheduler
  - High-level policy
  - Responsibility: deciding which process to run

- Could have an allocator for CPU as well
  - Early job-based systems (before timesharing)
  - Parallel and distributed systems

When to schedule?

- When does scheduler make decisions?

  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

- Minimal: nonpreemptive
  - ?

- Additional circumstances: preemptive
  - ?

Outline

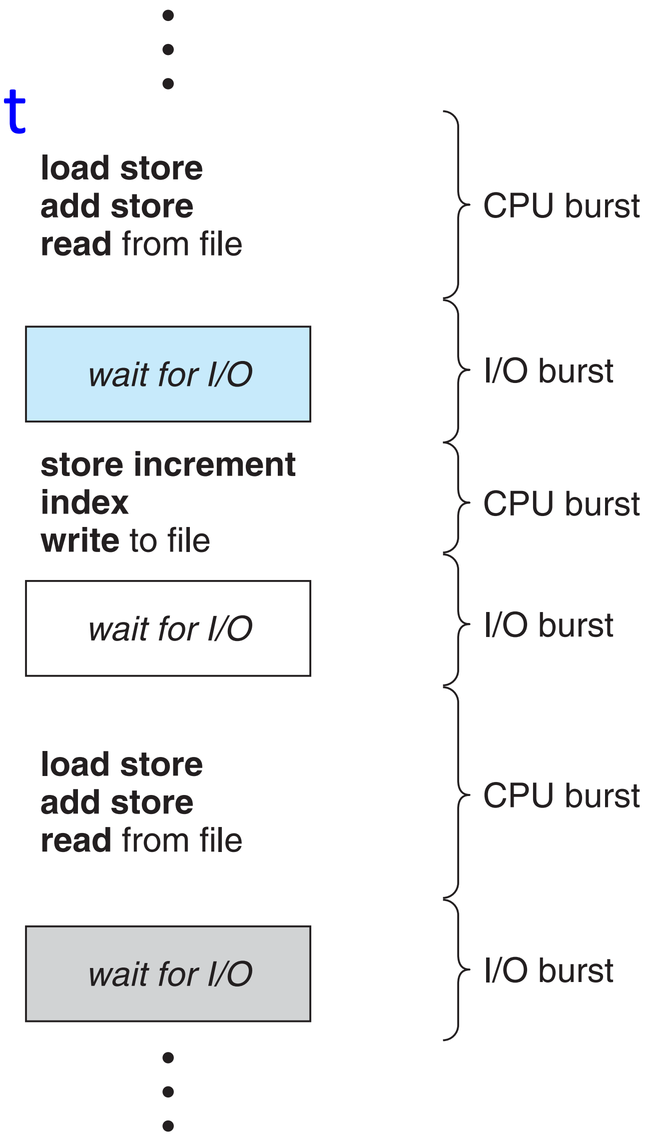
- Introduction to scheduling
- Scheduling algorithms
- Real Time Scheduling
- Evaluation

Overview of scheduling algorithms

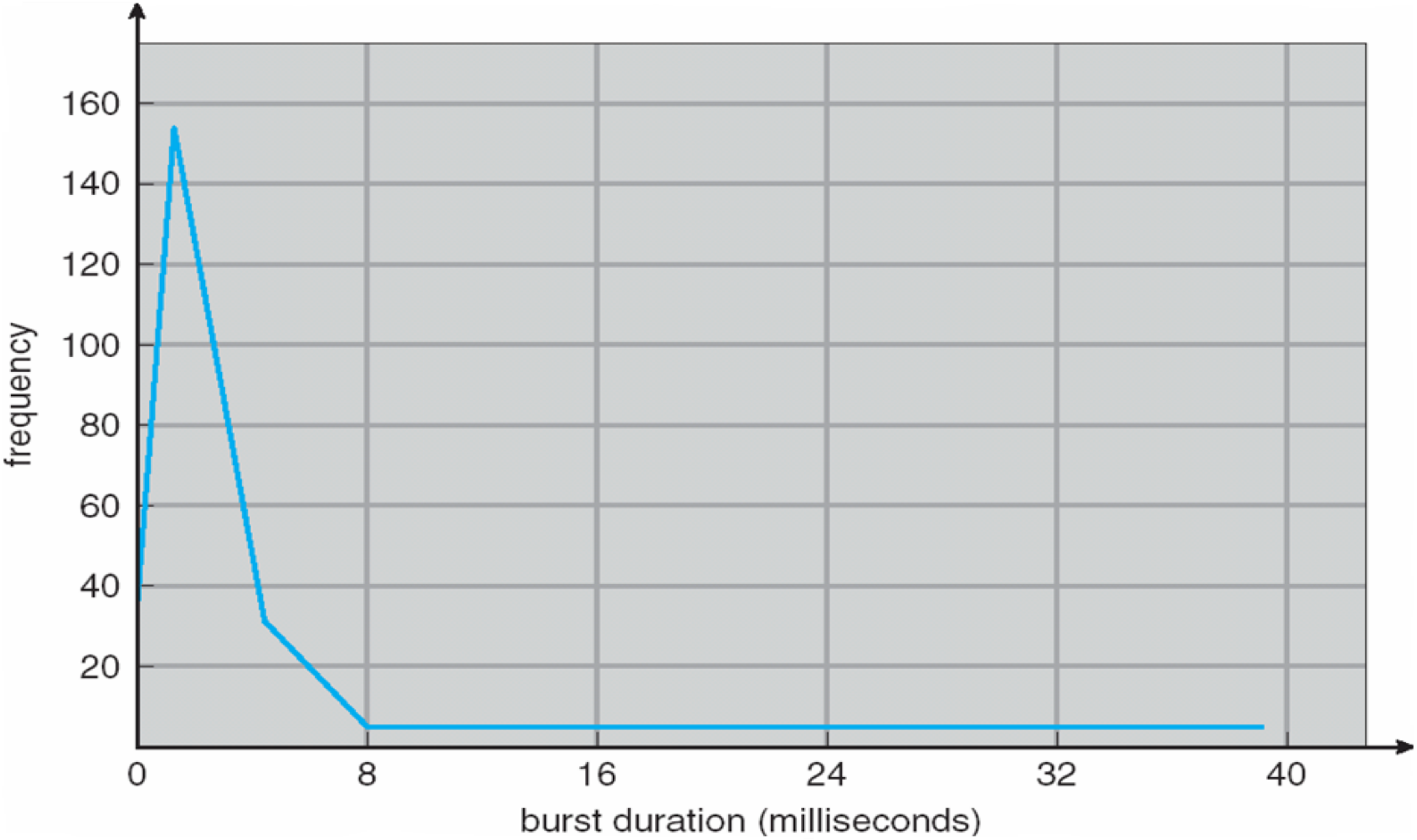
- Criteria: workload and environment

- Workload
  - Process behavior: alternating sequence of CPU and I/O bursts
  - CPU bound v.s. I/O bound

- Environment
  - Batch v.s. interactive?
  - Specialized v.s. general?



Typical Burst Times



Scheduling performance metrics

- Min waiting time: time spent waiting in queue for service
  - don't have process wait long in ready queue
- Max CPU utilization: % of time CPU is busy
  - keep CPU busy
- Max throughput: processes completed/time
  - complete as many processes as possible per unit time
- Min response time: submission to beginning of response
  - respond immediately
- Fairness: give each process/user same percentage of CPU

First-Come, First-Served (FCFS)

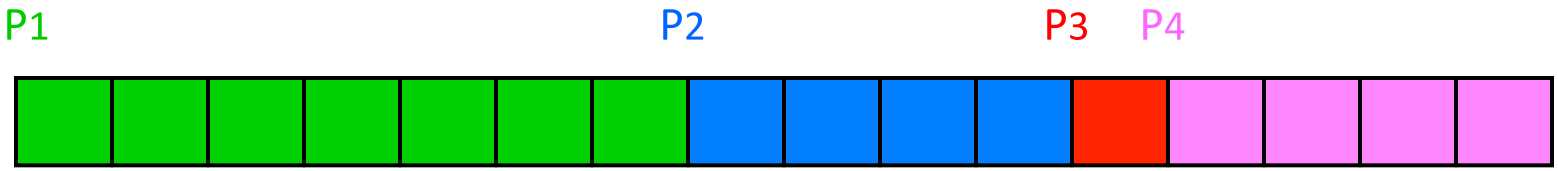
- Simplest CPU scheduling algorithm
  - First job that requests the CPU gets the CPU
  - Nonpreemptive
- Implementation: FIFO queue

Example of FCFS

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

- Gantt chart

Schedule:

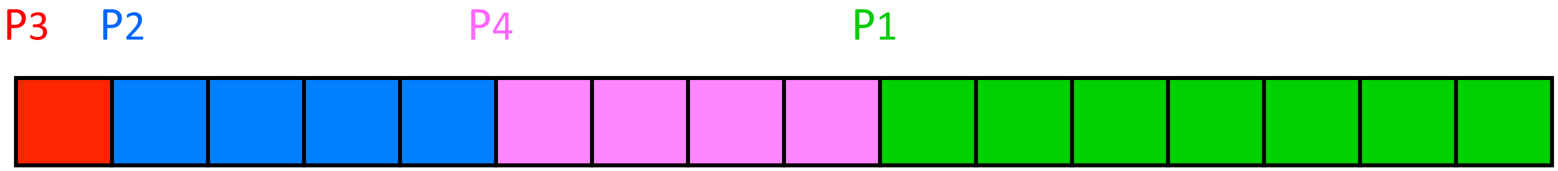


- Average waiting time: (0 + 7 + 11 + 12)/4 = 7.5

Example of FCFS: different arrival order

Process	Arrival Time	Burst Time
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)

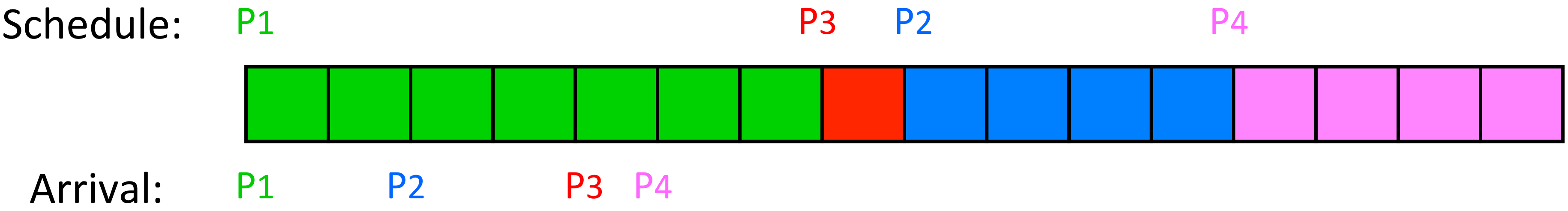
- Schedule the process with the shortest time

- FCFS if same time

Example of SJF (w/o preemption)

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

- Gantt chart



Schedule: P1 P3 P2 P4

Arrival: P1 P2 P3 P4

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

Shortest Job First (SJF)

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

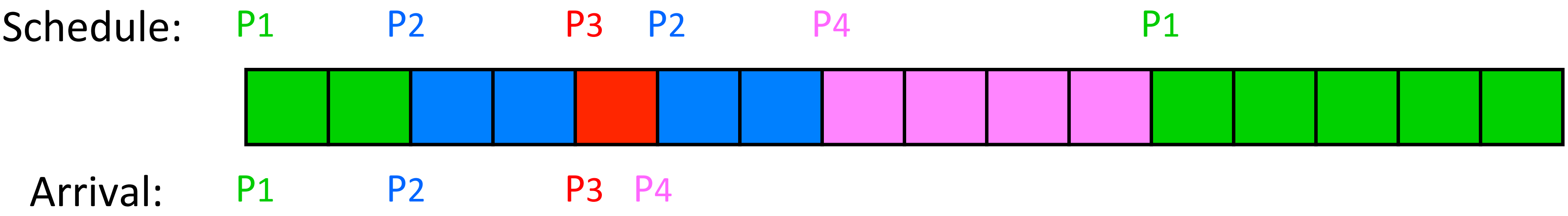
- If new process arrives w/ shorter CPU burst than the remaining for current process, schedule new process
  - SJF with preemption

- Advantage: reduces average waiting time
  - Provably optimal

Example of SRTF

Process	Arrival Time	Burst Time
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

Length of Next CPU Burst?

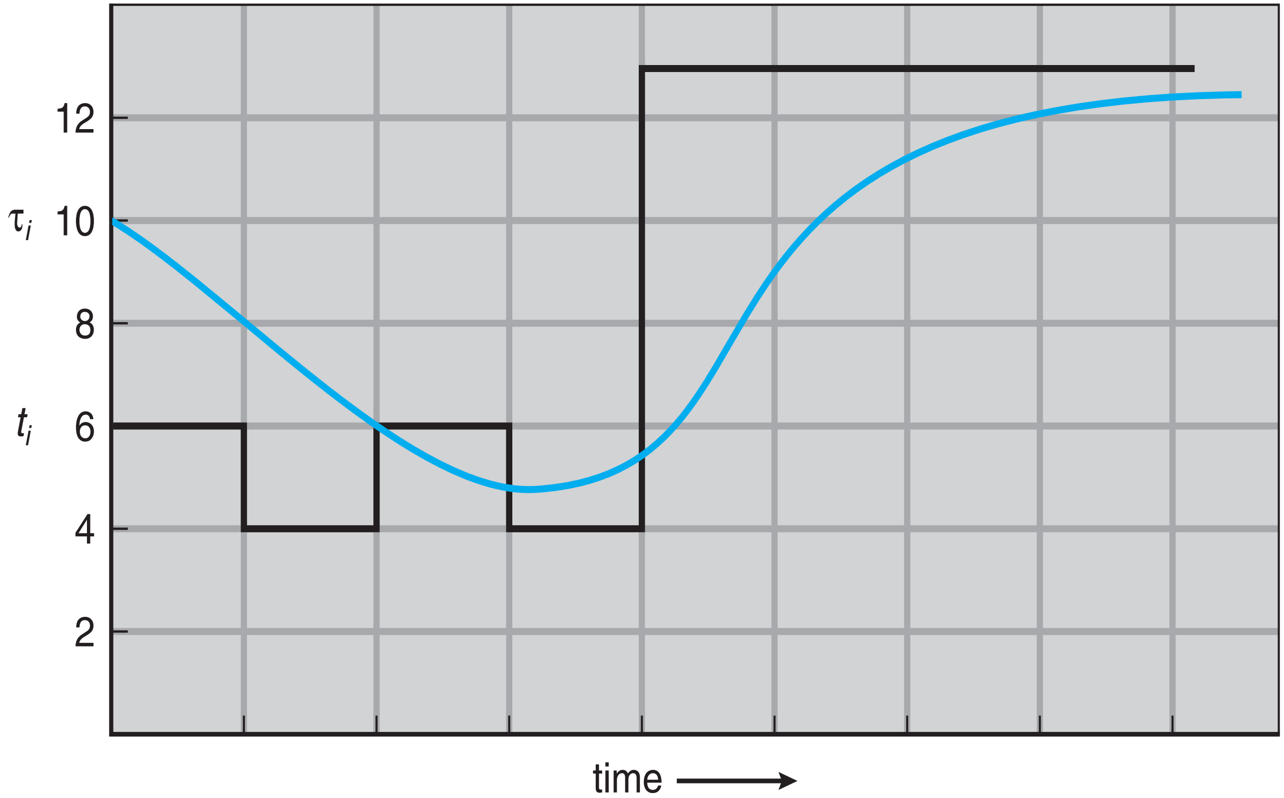
- Estimate the length: similar to the previous bursts
  - Pick process with shortest predicted next CPU burst
- Combine predictions and measured bursts using exponential averaging (or smoothing)

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

- Commonly, α set to ½
- “Exponential averaging” because expanding recursion gives:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_n - 1 + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

Exponential Smoothing



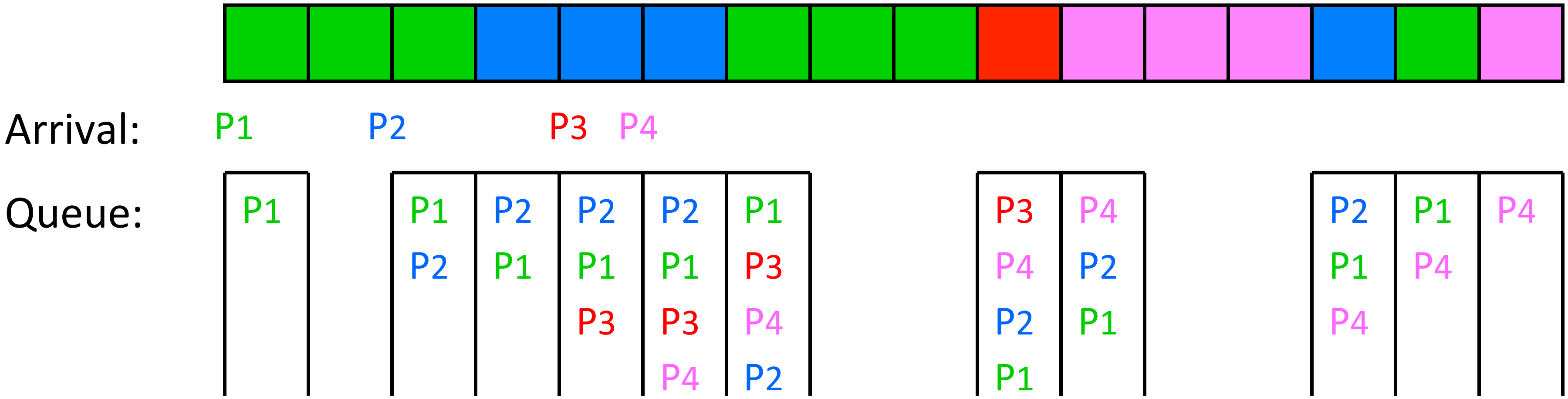
CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Round-Robin (RR)

- Practical approach to support time-sharing
- Run process for a time slice, then move to back of FIFO queue
- Preempted if still running at end of time-slice
- How to determine time slice?

Example of RR: time slice = 3

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

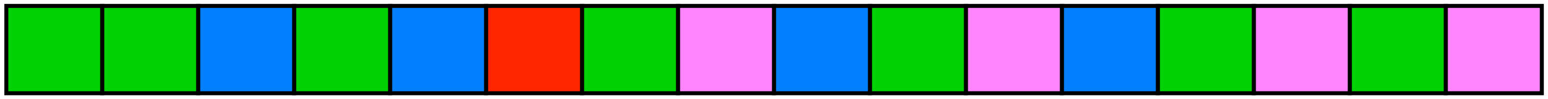
Queue:

P1	P1	P2	P2	P2	P1	P3	P4	P2	P1	P4
	P2	P1	P1	P1	P3	P4	P2	P1	P4	
			P3	P3	P4	P2	P1	P4		
				P4	P2	P1				

- 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

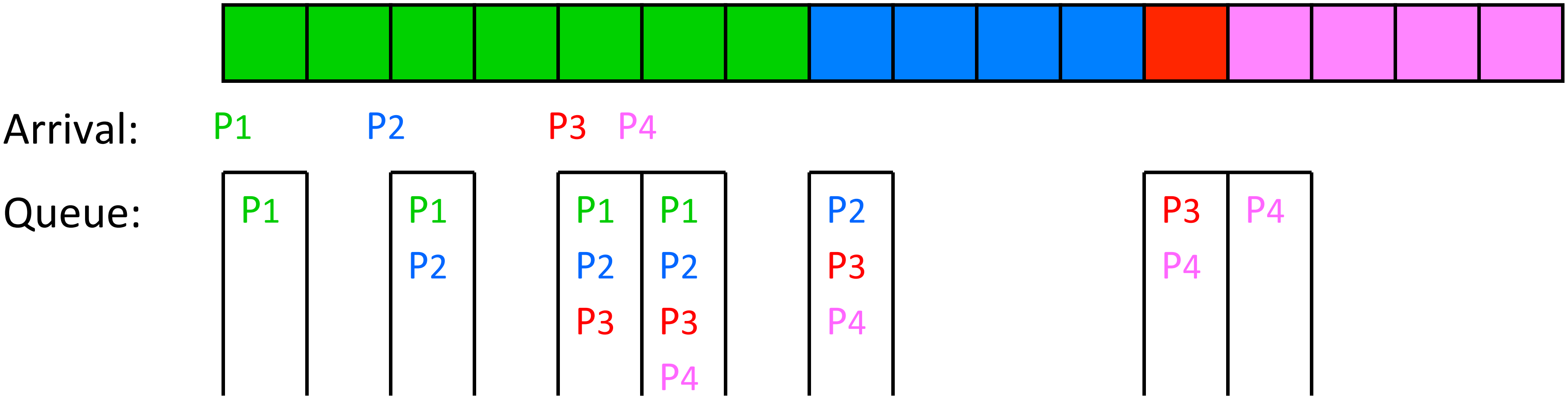


	🟩	🟩	🟦	🟩	🟦	🟥	🟩	🟪	🟦	🟩	🟪	🟦	🟩	🟪	🟩	🟪
Arrival:	P1		P2		P3	P4										
Queue:	P1	P1	P2 P1	P1 P2	P2 P3 P1	P3 P1 P4 P2	P1 P4 P2	P4 P2 P1	P2 P1 P4	P1 P4 P2	P4 P2 P1	P2 P1 P4	P1 P4	P4 P1	P1 P4	P4

- 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

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



- 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

Outline

- Introduction to scheduling
- Scheduling algorithms
- Real Time Scheduling
- Evaluation

Real-time scheduling

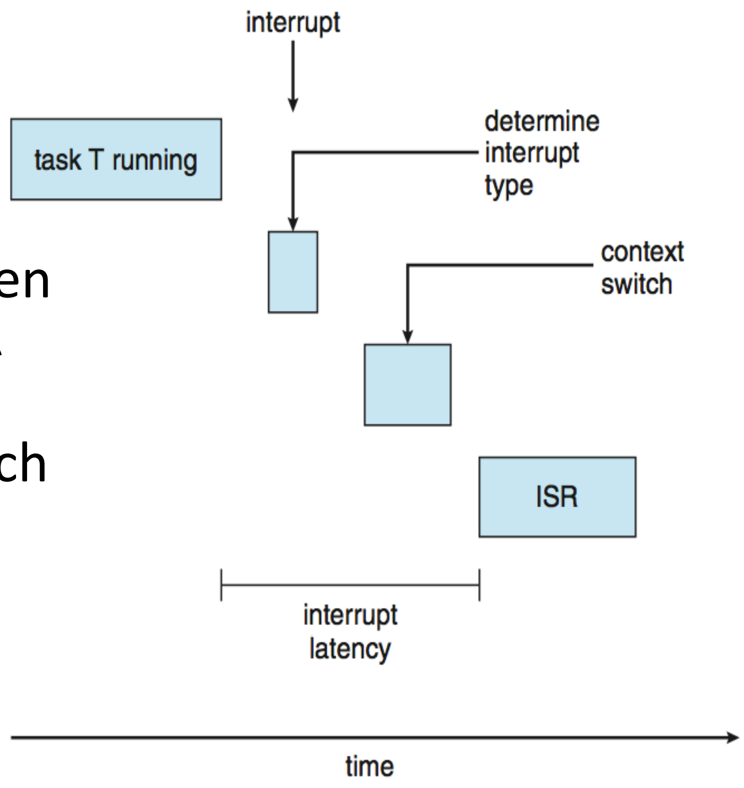
- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
  - E.g. gaming, video/music player, autopilot, medical devices…

- Hard real-time systems – required to complete a critical task within a guaranteed amount of time
- Soft real-time computing – requires that critical processes receive priority over less fortunate ones

- Linux supports soft real-time

Real-Time Scheduling

- Mechanism Challenges
  - Latencies can affect guarantees
    1. Interrupt latency: time between interrupt arrival to start of ISR (don't disable interrupts!)
    2. Dispatch latency: time to switch processes
- Policy Challenges
  - Ensure that soft real-time processes get priority
  - Ensure that hard real-time processes can finish within deadline
    - Admission Control is key



Priorities

- A priority is associated with each process
  - Run highest priority ready job (some may be blocked)
  - Round-robin among processes of equal priority
  - Can be preemptive or nonpreemptive

- Representing priorities
  - Typically an integer
  - The larger the higher or the lower?

Setting priorities

- Priority can be statically assigned
  - Some always have higher priority than others
  - Problem: starvation
- 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();
}
```

This code is taken almost verbatim 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?

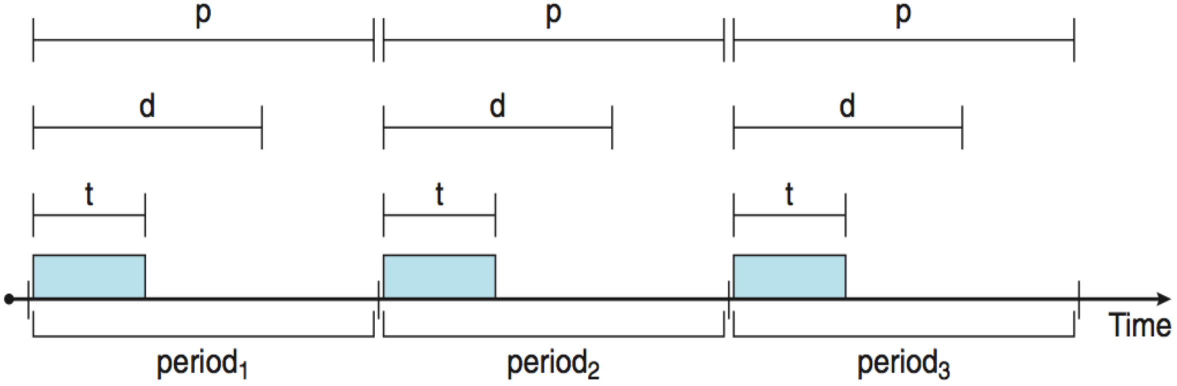
P1 (low): lock(my_lock) (gets my_lock)
P2(high): lock(my_lock)
P2 waits, P1 completes, P2 is scheduled

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
- Critical for real time systems
  - Example: Mars rover (http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/)

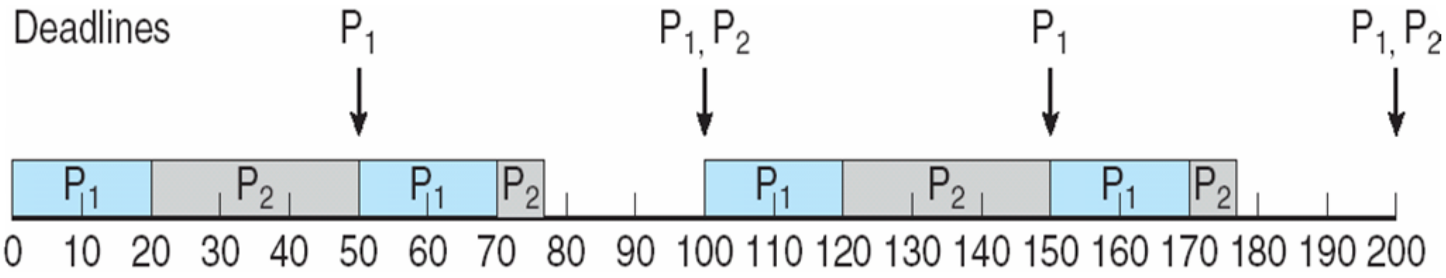
Hard Real-time Scheduling

- Priority scheduling only guarantees soft real-time
- Hard real-time: must also meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



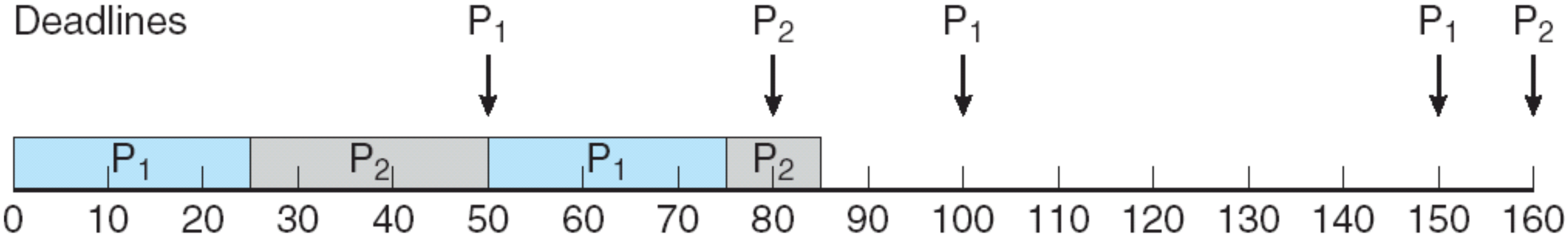
Rate Montonic Scheduling

- Applicable only to periodic processes
- Static priority based on period
- Don't need to know burst length
- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
- E.g., P1: p=50, t=20   P2: p=100, t=35
- P1 higher than P2
- CPU Utilization U = 20/50 + 35/100 = 0.75, so good…



Optimality of Rate Monotonic Scheduling

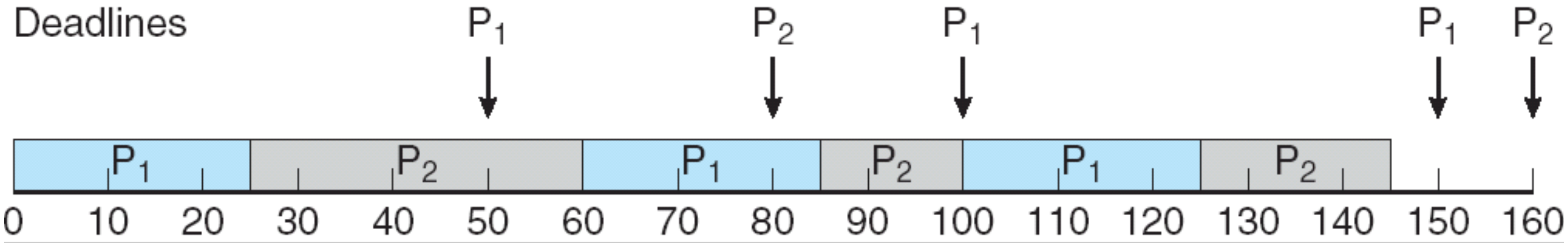
- Optimal static scheduling policy
- But not optimal dynamic one
- E.g., P1: p=50, t=25 P2: p=80, t=35
- Utilization = 25/50 + 35/80 = 0.9375, but…



- P2 misses deadline
- In general, Rate monotonic can't guarantee if
  - Utilization > $N(2^{1/N}-1)$ (or > 83%)
  - Admission control must deny to ensure schedulability

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines
  - Earlier deadline, higher priority, later deadline, lower the priority
- Dynamic priorities
  - Process can have higher/lower priority at different times
  - Doesn't require periodicity
  - Doesn't require knowledge of burst length
  - Provably optimal, but need to know deadlines
- Earlier ex. P1: t=25, d=50,100,150… P2: t=35 d=80, 160, 240,…



- Dynamic EDF order: P1, P2, P1, P1, P2, …

Outline

- Introduction to scheduling
- Scheduling algorithms
- Real Time Scheduling
- Evaluation

Evaluating Scheduling Algorithms

- Difficult: scheduling dependent on complex inputs
  - Workloads are non-deterministic even in tightly controlled environments
  - Timer interrupts can occur asynchronously
  - Hard to reproduce the same environment

- How to test?
  - How the system "feels": responsive? sluggish?
  - Analytical: Gantt charts, queuing models
  - Simulation

Analytical Evaluation

- Deterministic (Gantt charts)
  - Like what we've done in this lecture
  - Construct deterministic workload
  - For each algorithm, calculate minimum average waiting time
  - Simple and fast, but requires exact numbers for input, applies only to those inputs

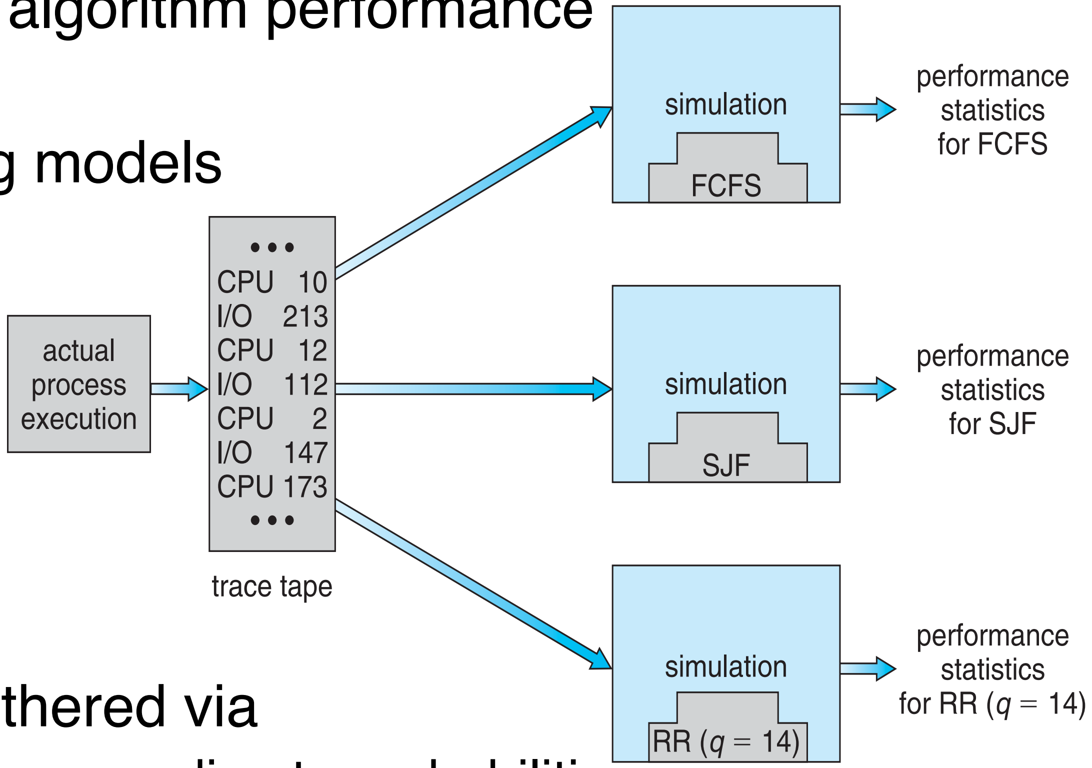
- Probabilistic (Queuing models)
  - Describe the arrival of processes, CPU, I/O bursts probabilistically
  - Simple distributions (e.g., exponential)
  - Compute average throughput, utilization, waiting time
  - Limited in kinds of policies that can be modeled
  - Generally out of scope of this class, except…

Little's Law

- Valid for any scheduling algorithm and arrival distribution
  - *n* = average queue length
  - *W* = average waiting time in queue (sec)
  - *λ* = average arrival rate into queue (processes/sec)
  - **Little's law: n = λ x W**
- Why? Complex proof, but intuitively…
  1. Let N = total number of jobs over some large time T
  2. n = Avg. # of queue length = Sum_T(# jobs in queue at time T)/T
  3. Sum_T(# jobs in queue at time T) = Sum_jobs(time of job j in queue)
  4. n = Sum_jobs(wait time of job j)/T = Sum_jobs(wait time of job j)/N*N/T
  5. n = Avg. wait time * Arrival rate = W * *λ*
- E.g.: if on average 7 processes arrive per sec, and normally 14 processes in queue, then average wait time per process = 2 sec

Simulation

- Programmed model of computer system
- Gather statistics indicating algorithm performance
- Clock is a variable
- More detailed than queuing models



actual process execution

•••
CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173
•••

trace tape

simulation
FCFS
performance statistics for FCFS

simulation
SJF
performance statistics for SJF

simulation
RR ($q = 14$)
performance statistics for RR ($q = 14$)

- Data to drive simulation gathered via
  - Random number generator according to probabilities
  - Distributions defined mathematically or empirically
  - Traces: recorded sequences of real events in real systems

Time slice and Context Switch Time