Operating Systems

Scheduling

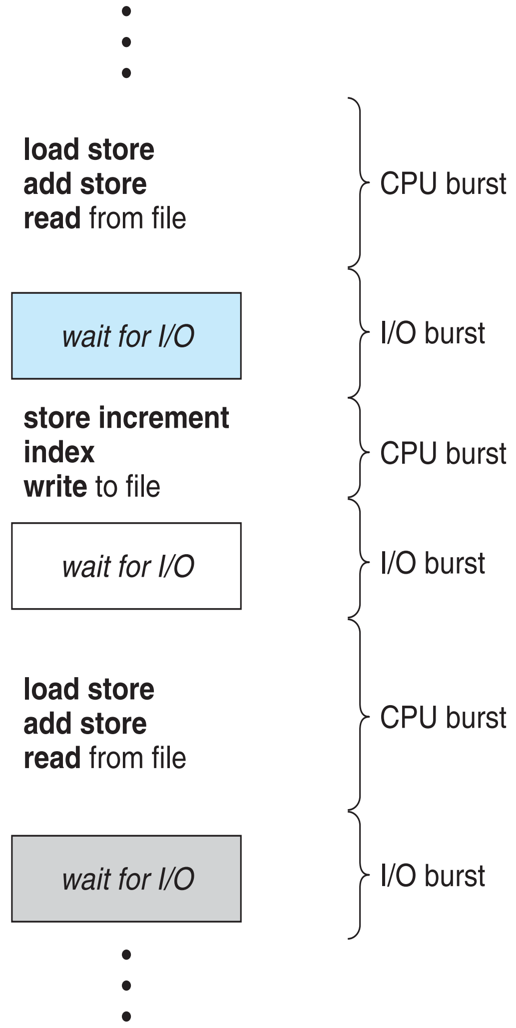
Lecture 8
Michael O’Boyle

Scheduling

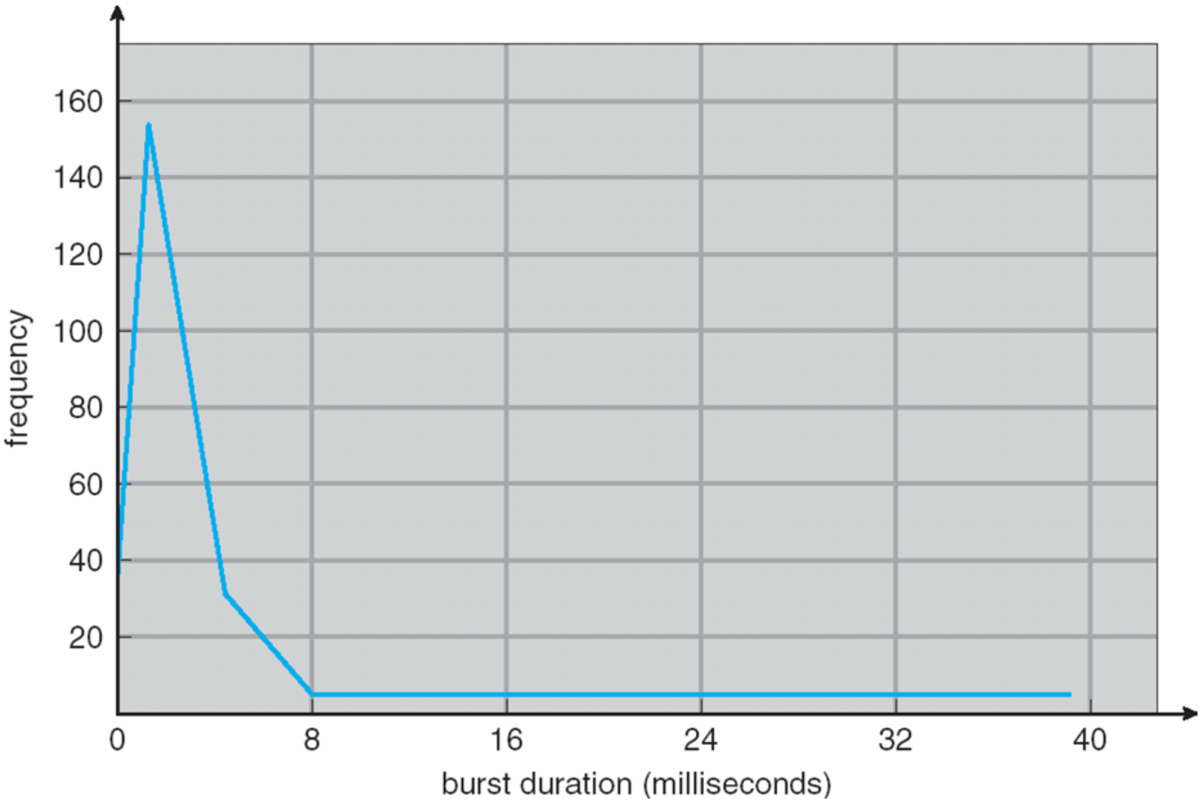
- We have talked about context switching
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes a trap or exception
  - may need to choose a different thread/process to run
- Glossed over which process or thread to run next
  - "some thread from the ready queue"
- This decision is called scheduling
  - scheduling is a policy
  - context switching is a mechanism

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



Exploit this : let another job use CPU

Classes of Schedulers

- Batch
  - Throughput / utilization oriented
  - Example: audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs
- Interactive
  - Response time oriented
- Real time
  - Deadline driven
  - Example: embedded systems (cars, airplanes, etc.)
- Parallel
  - Speedup-driven
  - Example: “space-shared” use of a 1000-processor machine for large simulations

We’ll be talking primarily about interactive schedulers

Multiple levels of scheduling decisions

- Long term
  - Should a new “job” be “initiated,” or should it be held?
    - typical of batch systems
- Medium term
  - Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
- Short term
  - Which thread should be given the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - should we attempt to coordinate the running of threads from the same address space in some way?
    - should we worry about cache state (processor affinity)?

Scheduling Goals I: Performance

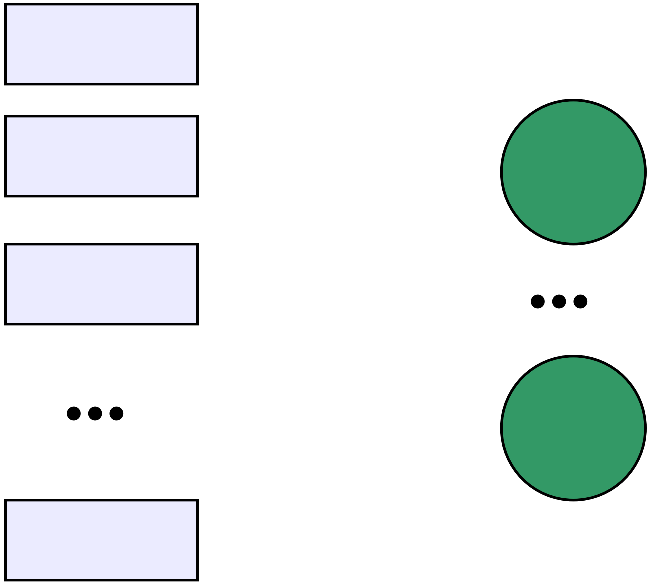
- Many possible metrics / performance goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize throughput (`requests completed / s`)
  - minimize average response time (`average time from submission of request to completion of response`)
  - minimize average waiting time (`average time from submission of request to start of execution`)
  - minimize energy (`joules per instruction`) subject to some constraint (`e.g., frames/second`)

Scheduling Goals II: Fairness

- No single, compelling definition of “fair”
  - How to measure fairness?
    - Equal CPU consumption? (over what time scale?)
  - Fair per-user? per-process? per-thread?
  - What if one process is CPU bound and one is I/O bound?

- Sometimes the goal is to be unfair:
  - Explicitly favor some particular class of requests (priority system), but…
  - avoid starvation (be sure everyone gets at least some service)

The basic situation



Schedulable units

Resources

Scheduling:
- Who to assign each resource to
- When to re-evaluate your decisions

When to assign?

- Pre-emptive vs. non-preemptive schedulers
  - Non-preemptive
    - once you give somebody the green light, they've got it until they relinquish it
      - an I/O operation
      - allocation of memory in a system without swapping
  - Preemptive
    - you can re-visit a decision
      - setting the timer allows you to preempt the CPU from a thread even if it doesn't relinquish it voluntarily

    - Re-assignment always involves some overhead
      - Overhead doesn't contribute to the goal of any scheduler

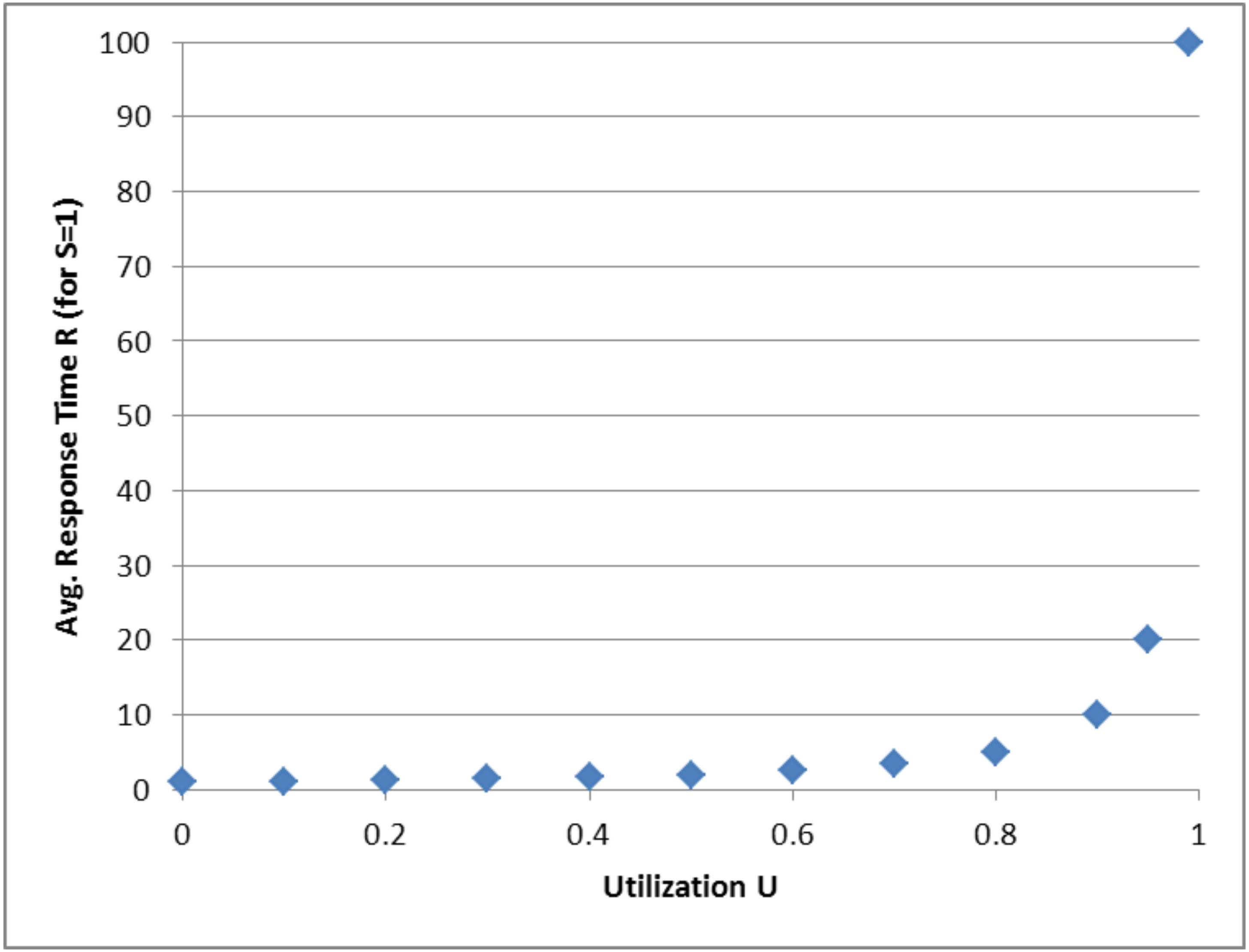
- We'll assume "work conserving" policies
  - Never leave a resource idle when someone wants it
    - Why even mention this? When might it be useful to do something else?

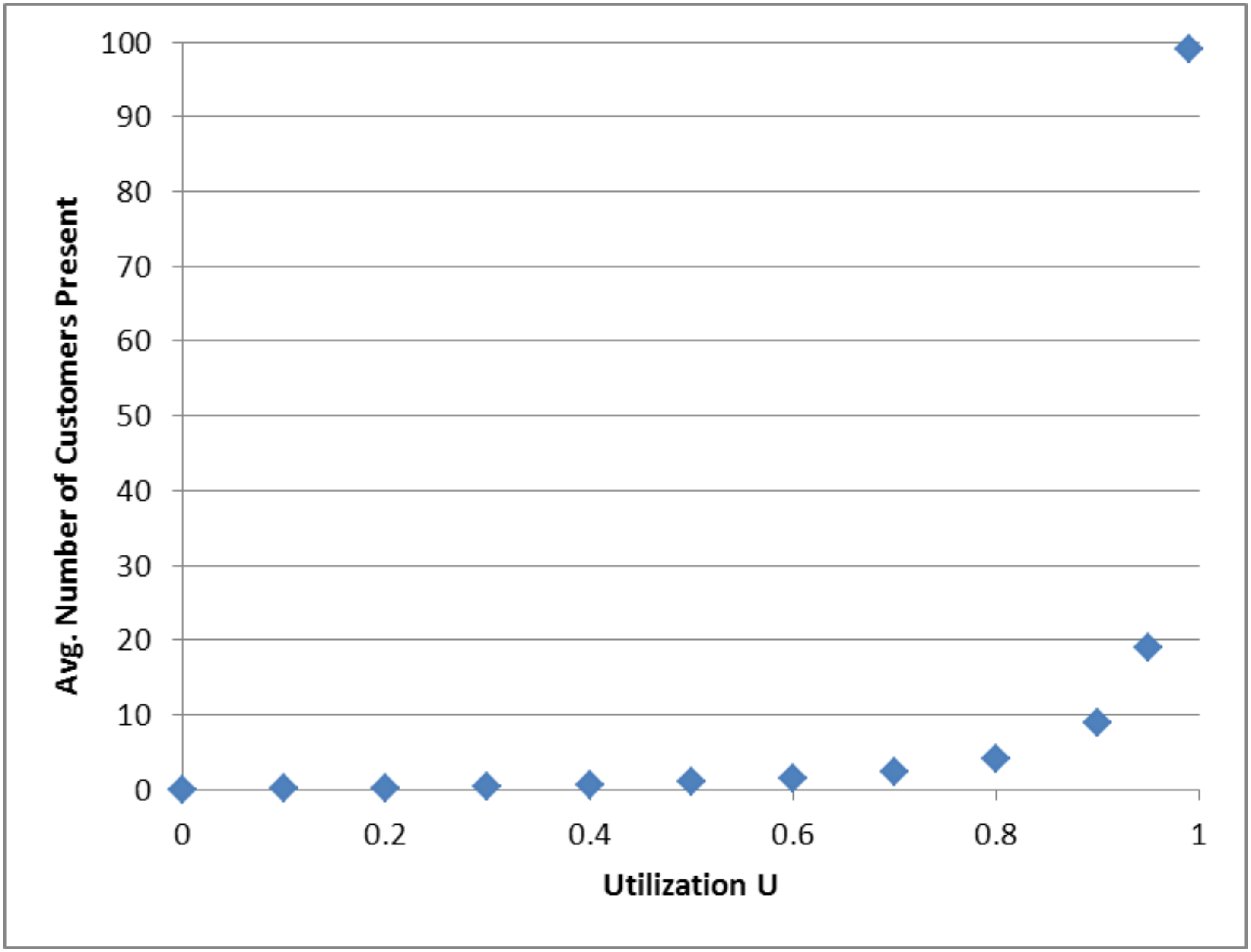
Laws and Properties

- The Utilization Law: $U = X * S$
  - U is utilization,
  - X is throughput (requests per second)
  - S is average service time
  - This means that utilization is constant, independent of the schedule, so long as the workload can be processed

- Little's Law: $N = X * R$
  - Where N is average number in system, X is throughput, and R is average response time (average time in system)
    - This means that better average response time implies fewer in system, and vice versa

- Response Time R at a single server under FCFS scheduling:
  - $R = S / (1-U)$ and
  - $N = U / (1-U)$


Avg. Response Time R (for S=1)
100
90
80
70
60
50
40
30
20
10
0
0
0.2
0.4
0.6
0.8
1
Utilization U


Avg. Number of Customers Present
100
90
80
70
60
50
40
30
20
10
0
0
0.2
0.4
0.6
0.8
1
Utilization U

Algorithm #1: FCFS/FIFO

- First-come first-served / First-in first-out (FCFS/FIFO)
  - schedule in the order that they arrive
  - “real-world” scheduling of people in (single) lines
    - supermarkets
  - jobs treated equally, no starvation
    - In what sense is this “fair”?
- Sounds perfect!
  - in the real world, does FCFS/FIFO work well?

First- Come, First-Served (FCFS) Scheduling

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: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

FCFS/FIFO drawbacks

- Average response time can be poor: small requests wait behind big ones
- May lead to poor utilization of other resources
  - if you send me on my way, I can go keep another resource busy
  - FCFS may result in poor overlap of CPU and I/O activity
    - E.g., a CPU-intensive job prevents an I/O-intensive job from a small bit of computation, preventing it from going back and keeping the I/O subsystem busy
- The more copies of the resource there are to be scheduled
  - the less dramatic the impact of occasional very large jobs (so long as there is a single waiting line)
  - E.g., many cores vs. one core

Algorithm #2: Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

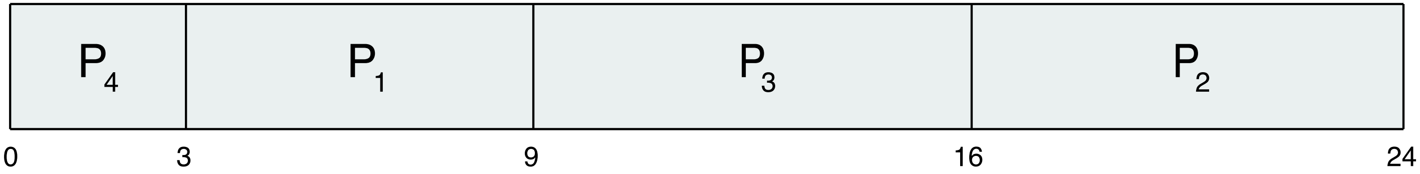
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Algorithm #2:

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

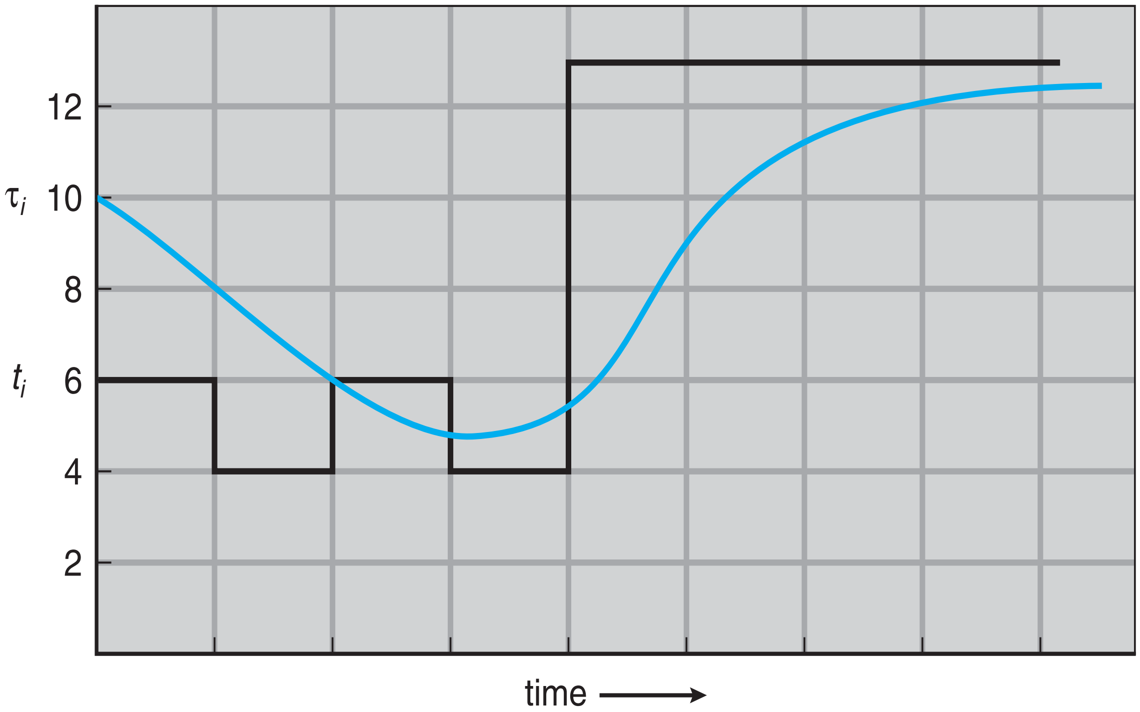
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

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
- Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



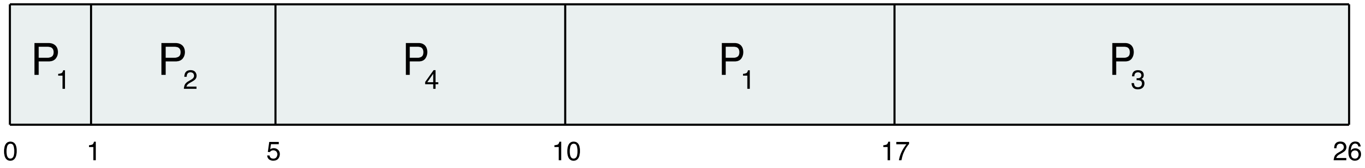
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Algorithm #3: Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are *n* processes in the ready queue and the time quantum is *q*,
  - then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
  - No process waits more than $(n-1)q$ time units.

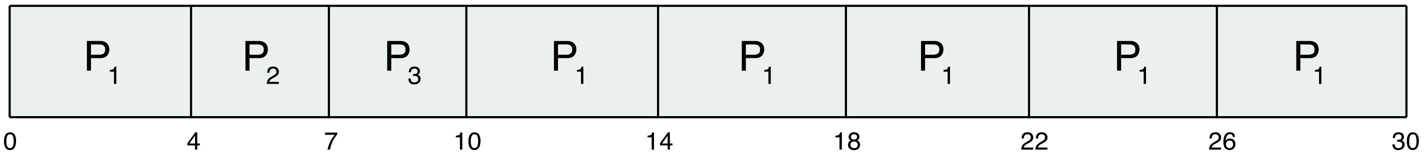
- Timer interrupts every quantum to schedule next process

- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

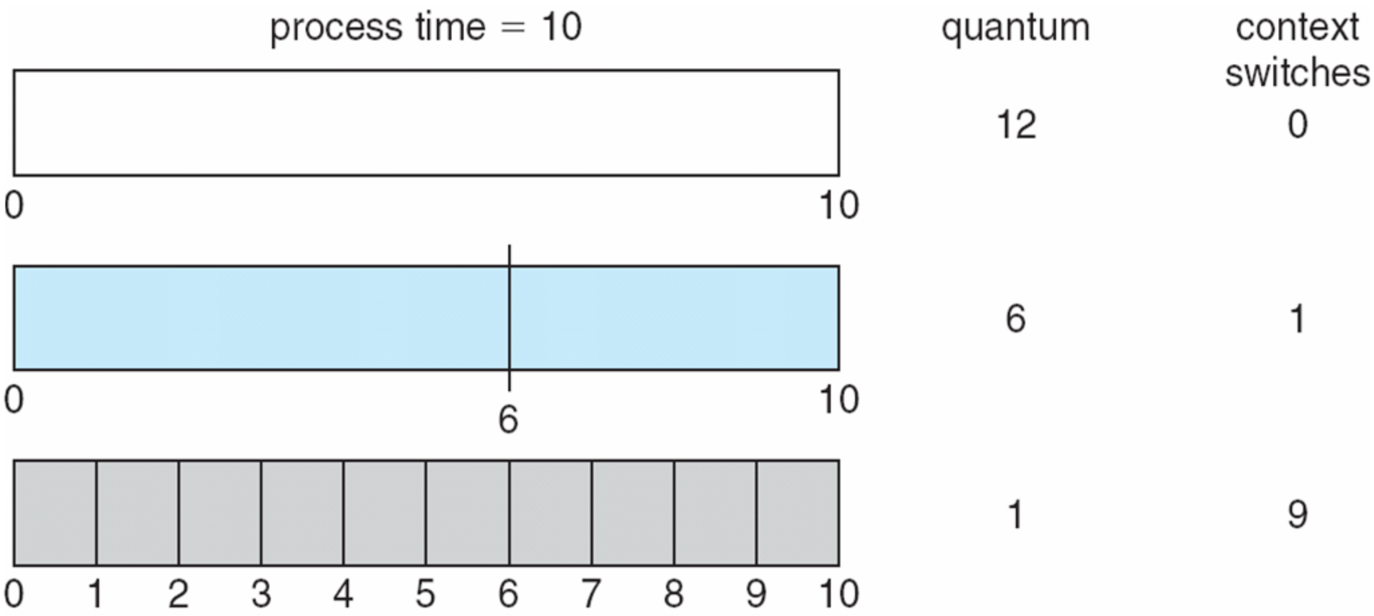
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

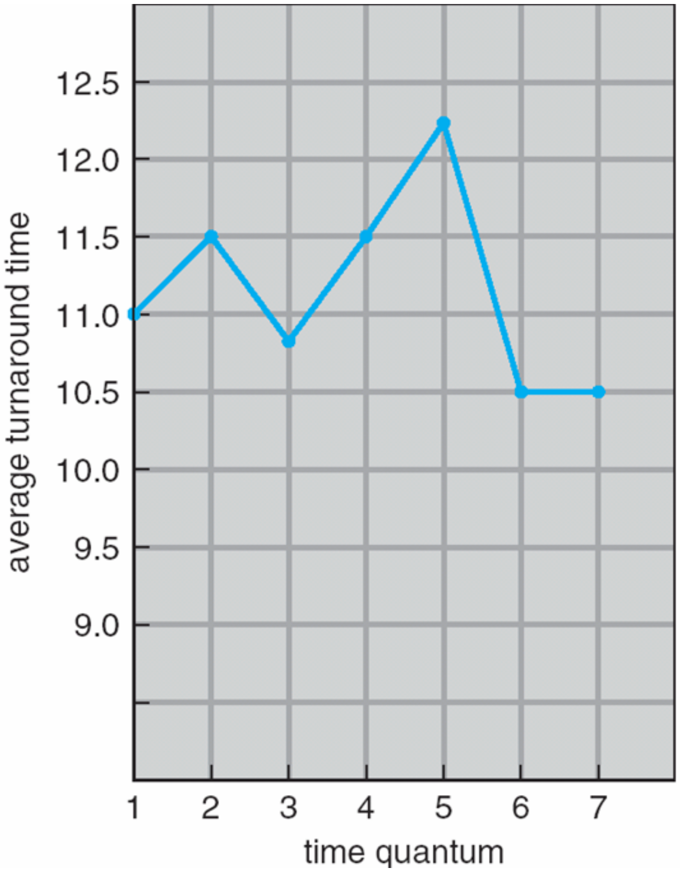


- Typically, higher average turnaround than SJF,
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

RR drawbacks

- What if all jobs are exactly the same length?
  - What would the pessimal schedule be (with average response time as the measure)?
- What do you set the quantum to be?
  - no value is “correct”
    - if small, then context switch often, incurring high overhead
    - if large, then response time degrades
- Treats all jobs equally
  - What about CPU vs I/O bound?

Algorithm #4: Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

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

- Priority scheduling Gantt Chart

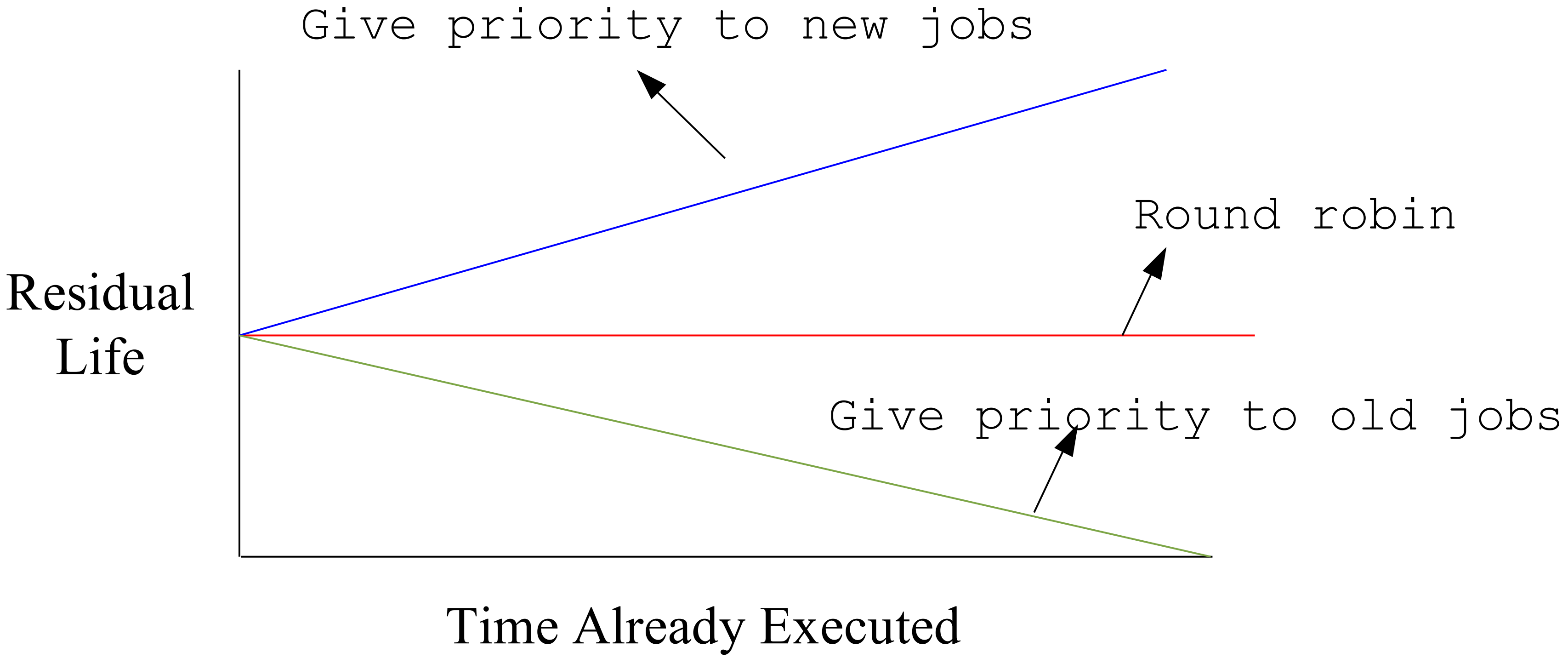
- Average waiting time = 8.2 msec
- Error in Gantt: P2, P5, P1, P3, P4

Program behavior and scheduling

- An analogy:
  - Say you're at the airport waiting for a flight
  - There are two identical ATMs:
    - ATM 1 has 3 people in line
    - ATM 2 has 6 people in line
  - You get into the line for ATM 1
  - ATM 2's line shrinks to 4 people
  - Why might you now switch lines, preferring 5th in line for ATM 2 over 4th in line for ATM 1?

Residual Life

- Given that a job has already executed for X seconds, how much longer will it execute, on average, before completing?



Multi-level Feedback Queues (MLFQ)

- It’s been observed that workloads tend to have increasing residual life – “if you don’t finish quickly, you’re probably a lifer”
- This is exploited in practice by using a policy that discriminates against the old
- MLFQ:
  - there is a hierarchy of queues
  - there is a priority ordering among the queues
  - new requests enter the highest priority queue
  - each queue is scheduled RR
  - requests move between queues based on execution history

UNIX scheduling

- Canonical scheduler is pretty much MLFQ
  - 3-4 classes spanning ~170 priority levels
    - timesharing: lowest 60 priorities
    - system: middle 40 priorities
    - real-time: highest 60 priorities
  - priority scheduling across queues, RR within
    - process with highest priority always run first
    - processes with same priority scheduled RR
  - processes dynamically change priority
    - increases over time if process blocks before end of quantum
    - decreases if process uses entire quantum
- Goals:
  - reward interactive behavior over CPU hogs
    - interactive jobs typically have short bursts of CPU

Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many "pure" algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids that exploit observed program behavior
- Scheduling is important
  - Look at muticore/GPU systems in later research lecture