Operating Systems (INFR09047)

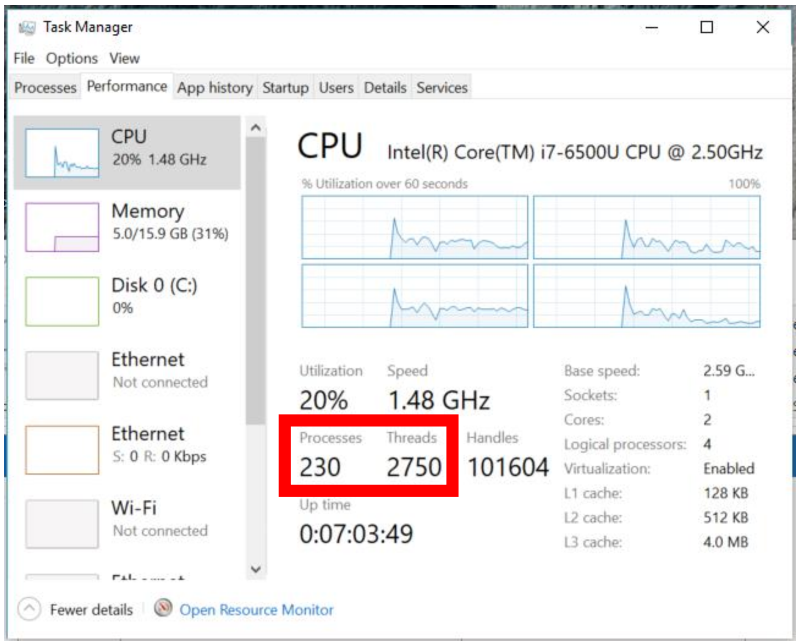
2019/2020 Semester 2

Scheduling

abarbala@inf.ed.ac.uk

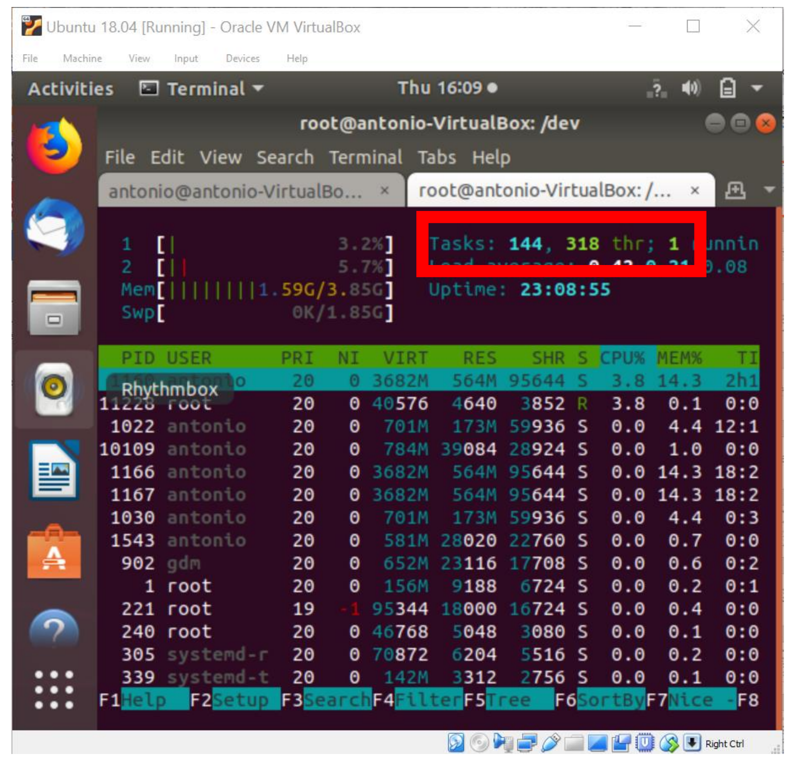
Chapter 5

How Many Processes or Threads Are Running On My Computer?



Window’s Task Manager

UNIX’s htop (or top)



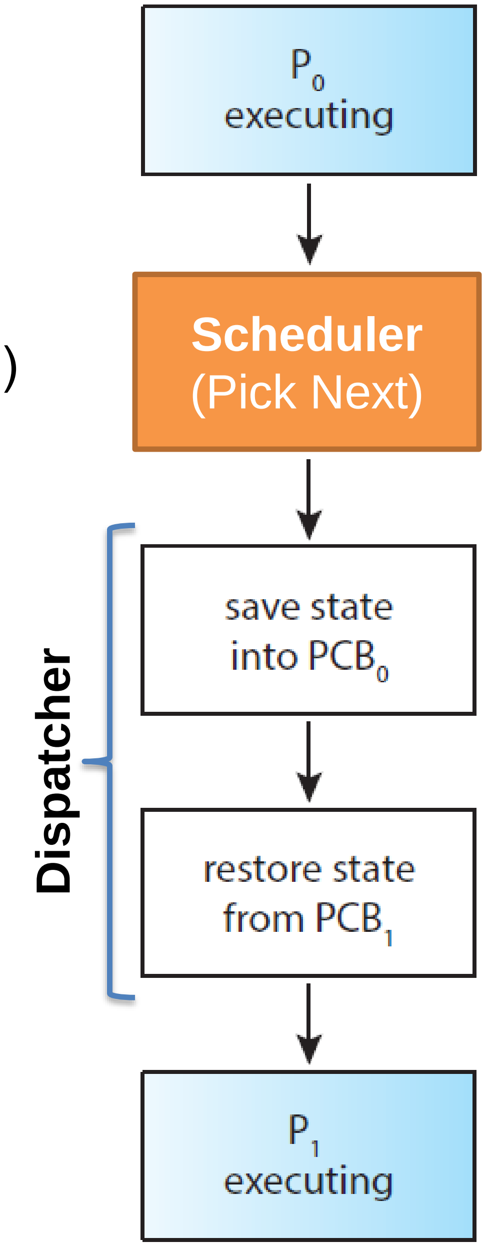
Who will run next?

Scheduling

Single CPU/core

- Decision of who will r**un next**
  - Process or threads
  - **In kernel for kernel-level threading** (slideset' focus)
- **Pick among** the ones in ready queue(s)

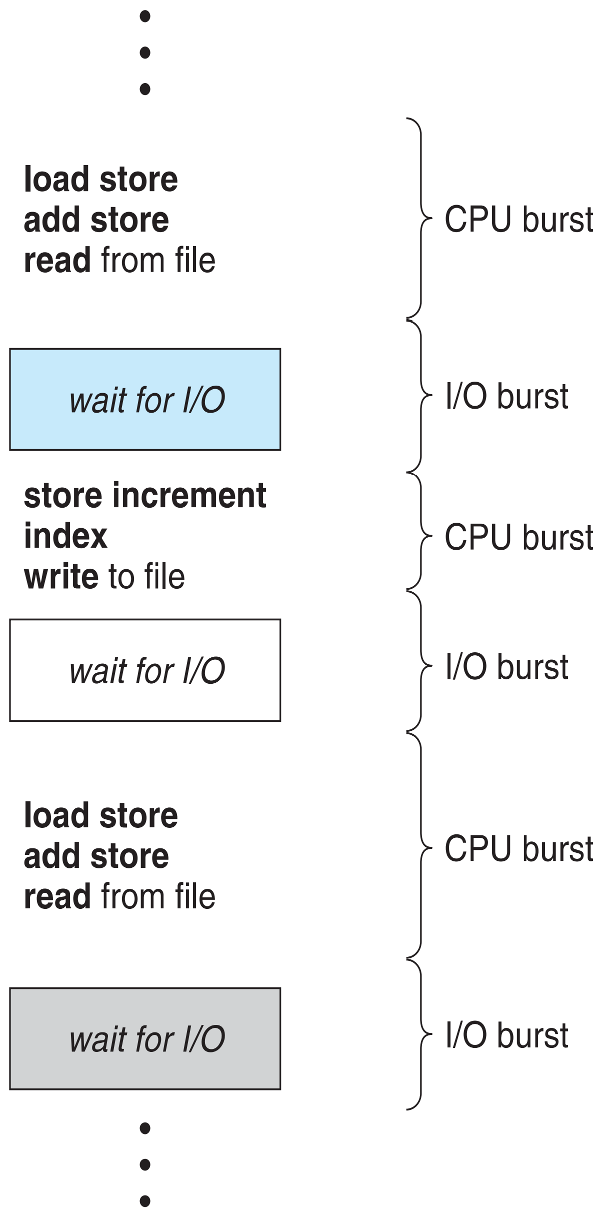
- *When?*
  - Potentially, **all times we switch** to the OS
    - Interrupt (device completion, timer interrupt, etc.)
    - Syscall (including voluntary process/thread termination, or yield)
    - Exception (including involuntary termination)
- *How?*
  - **Scheduler** decides
    - Policy (implemented by an algorithm)
  - Task switched by **dispatcher**
    - Mechanism (kernel or user code)



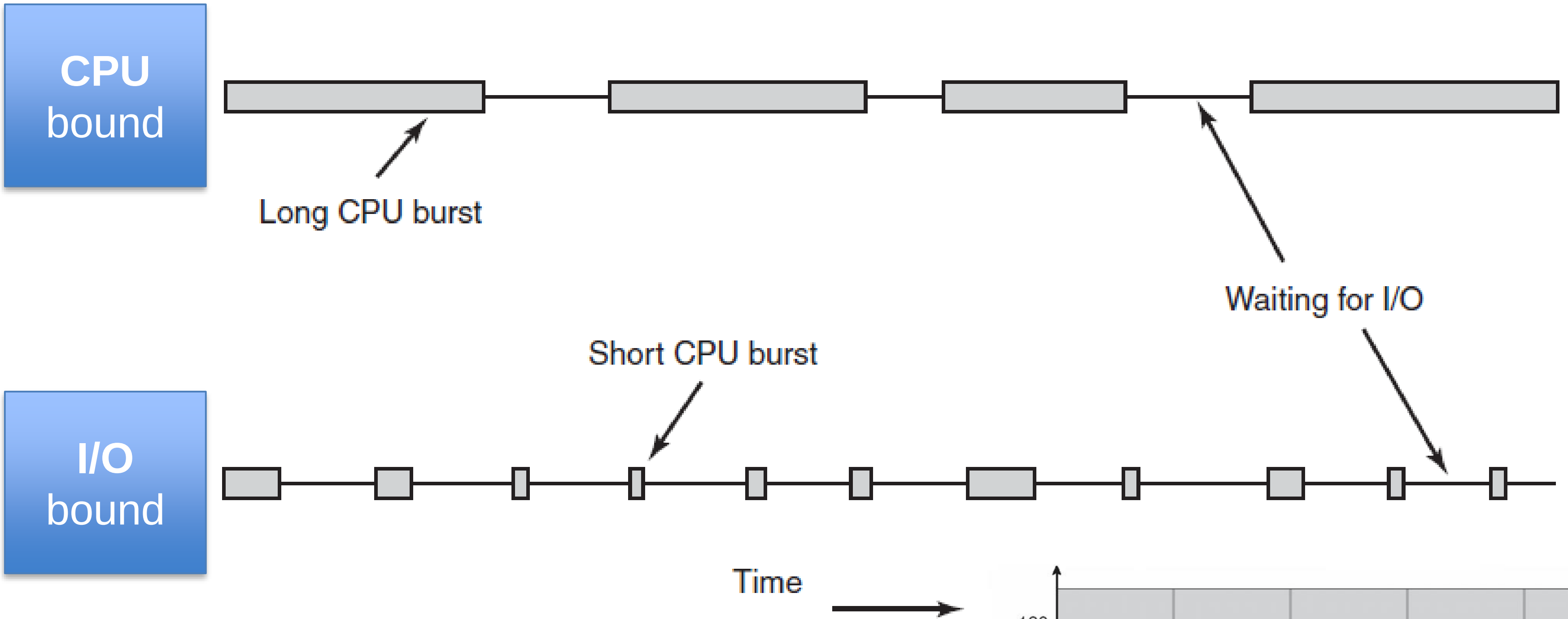
Process or Thread Behavior #1

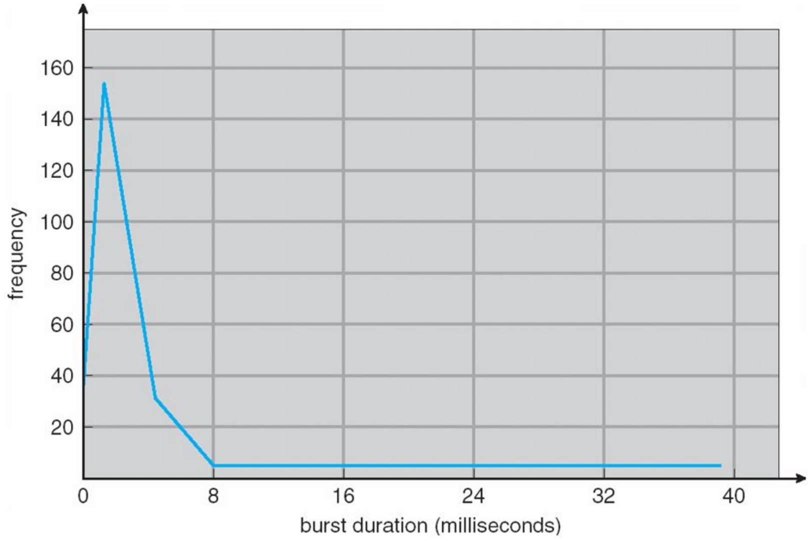
- Process or thread execution consists of cycles of
  - CPU execution: **CPU burst**
  - I/O wait: **IO burst**
- **CPU bursts** distribution is **application dependent**

- **Maximum CPU utilization** with multiprogramming
  - Don't leave CPU idle
- While a process or thread is waiting for IO **another can run** on the CPU
  - Focus on **single** CPU/core



Process or Thread Behavior #2





While a process or thread is waiting for IO **another can run** on the CPU

Scheduling Goals: Performance

- Many performance goals (which may conflict)
  - Maximize CPU utilization
  - Maximize throughput (`processes completed per time unit`)
  - Minimize turnaround time (`time from submission of task to completion`)
  - Minimize waiting time (`all periods spent waiting in the ready queue from submission`)
  - Minimize response time (`time from submission of request to response is produced`)
  - Minimize energy (`joules per instruction`) subject to some constraint (`e.g., frames/second`)
- In most cases we optimize the **average metric**
  - But sometimes minimize the worst case (e.g., response time)

Scheduling Goals: 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?

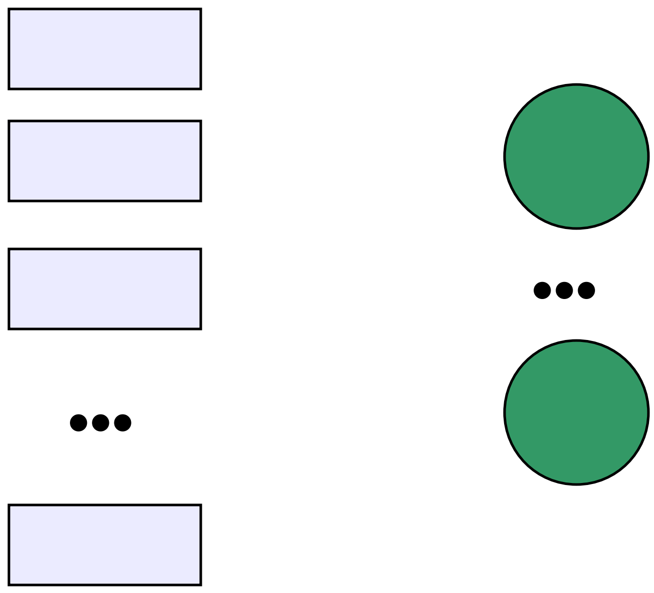
- Sometimes the goal is to be **unfair**
  - Explicitly favor some particular class of requests
    - **Priority system**
  - Avoid starvation
    - Be sure everyone gets at least some service

Classes of Schedulers

- Batch
  - Throughput / utilization oriented
  - *Example:* audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs
- Interactive
  - Response time oriented
  - *Example:* window-based operating system
- Real time
  - Deadline driven
  - *Example:* embedded systems (cars, airplanes, etc.)

We will **not talk** about real-time scheduling

General Scheduling Problem



Schedulable units
(Processes or Threads)

Resources
(CPU, …)

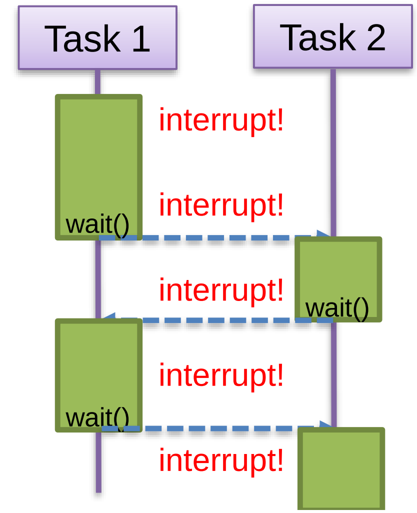
Scheduling:

- **Who** to *assign* each resource to
- **When** to re-evaluate your decisions

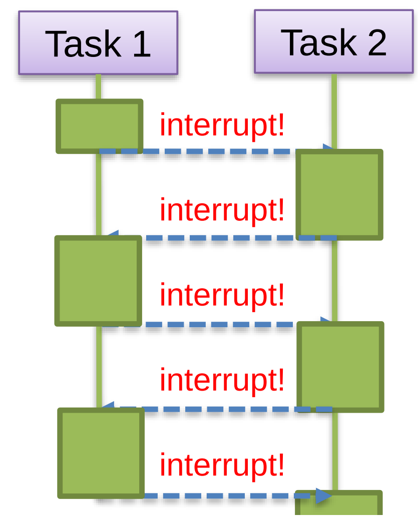
Scheduling **is not** just about assigning *processes and threads* to the CPU, but to **any other HW/SW resource** of the computer

When to Re-evaluate the Decision?

- **Non-preemptive** scheduling
  - Processes/threads execute until completion or until they want
    - **Voluntary process switch**
    - **Process/thread switch on blocking calls**
  - The scheduler gets involved only at exit or on request
    - For every clock interrupt, running process keeps going



- **Preemptive** scheduling
  - While a process/thread executes, its execution may be paused, and another process/thread resumes its execution, etc.
    - **Involuntary process switch**
    - For every clock interrupt, running process may be suspended and switched with another process (if there is any)



wait()s removed to ease the draw

Algorithm #1

First-come First-served (FCFS)

- Processes are assigned to the CPU in the **order they request it** (or they **arrive**)
- **Non-**preemptive

- Real-world scheduling of people in (single) lines
  - Supermarkets

Algorithm #1

FCFS Example #1

- Arrival **order** for the processes
  - P1, P2, P3
- Turnaround time
  - P1 = 24
  - P2 = 27
  - P3 = 30
- Average turnaround time
  - (24+27+30)/3 = 27
- Short process delayed by long process
  - **Convoy effect**

Process	CPUTime
P1	24
P2	3
P3	3

Turnaround Time – time taken by a job to complete after submission

Algorithm #1

FCFS Example #2

- Arrival **order** for the processes
  - **P2, P3, P1**
- Turnaround time
  - P1 = 30
  - P2 = 3
  - P3 = 6
- Average turnaround time
  - (30+3+6)/3 = 13
- Much better than the previous case

Process	CPUTime
P1	24
P2	3
P3	3

Algorithm #1

FCFS Drawbacks

- Average response time can be **poor**
  - Short tasks wait behind big ones (convoy effect)

- May lead to poor utilization of **other resources**
  - Poor overlap of CPU and I/O activity
  - Example
    - 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

Algorithm #2

Shortest Job First (SJF)

- Associate with each process the **length of its CPU time**
- Use the CPU time length to schedule the process with the **shortest CPU time first**

- Two variations
  - *Non-preemptive* – once CPU is given to the process, it cannot be taken away until completion (or blocking)
  - *Preemptive* – if a new process arrives with CPU time less than the remaining time of current executing process, preempt.
    - ***Shortest Remaining Time Next, SRTN***

Algorithm #2

Non-Preemptive SJF Example

- Arrival time for the processes
  - P1 at 0, P2 at 2, P3 at 4, P4 at 5
- Turnaround time
  - P1 = 7
  - P2 = 12 – 2 = 10
  - P3 = 8 – 4 = 4
  - P4 = 16 – 5 = 11
- Average turnaround time
  - (7+10+4+11)/4 = 8
- 3 ctx switches

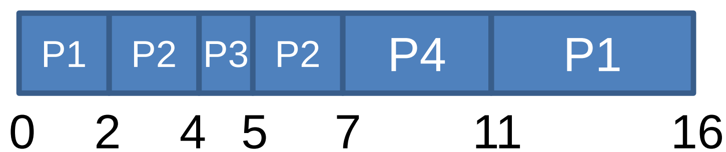
Process	ArrivalTime	CPUTime
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Algorithm #2

Preemptive SJF Example

- Arrival time for the processes
  - P1 at 0, P2 at 2, P3 at 4, P4 at 5
- Turnaround time
  - P1 = 16
  - P2 = 7 – 2 = 5
  - P3 = 5 – 4 = 1
  - P4 = 11 – 5 = 6
- Average turnaround time
  - (16+5+1+6)/4 = 7
- 5 ctx switches

Process	ArrivalTime	CPUTime
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Algorithm #2

SJF Drawbacks

- Preemptive SJF is **optimal**
- But it can only be approximated
  - **Too complex** to be implemented in practice
  - **Not always possible** to determine the CPU/IO burst

Algorithm #3

Round-robin (RR)

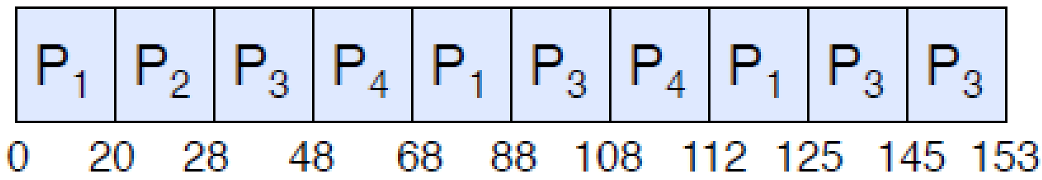
- Each process is allowed to run for a specified time interval
  - Called **quantum**
- After this time **has elapsed**
  1. The process is preempted
  2. And added to the end of the ready queue
  3. The next process is scheduled
- If the process **terminates or blocks for IO** before this time
  1. It is added to a wait queue
  2. The next process is scheduled

Algorithm #3

Round-robin Example

Process	CPU time
P1	53
P2	8
P3	68
P4	24

Time Quantum = 20



- Waiting time for
  - P_1=(68-20)+(112-88)=72
  - P_2=(20-0)=20
  - P_3=(28-0)+(88-48)+(125-108)=85
  - P_4=(48-0)+(108-68)=88
- Average waiting time (72+20+85+88)/4=66.25
- Average turnaround time (125+28+153+112)/4=104.5

Algorithm #3

What About the Time Quantum?

- **Context switching** may impact the choice of the time quantum
- Example
  - Context switch is 1ms
  - Time quantum is 4ms
  - 20% of the time is thrown away context switching
- Typical numbers
  - Context switch is in the order of tens of us
  - Timeslice/quantum is 1KHz (every 1ms)
- But when there are a lot of processes a long time quantum causes a **poor response time**

Algorithm #3

What About Round-robin?

- Advantages
  - Solution to fairness and starvation
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness (interactivity) if small number of jobs

- Disadvantages
  - Context-switching time may add up for long jobs

FCFS vs RR

- Assuming zero-cost context-switching time, is RR better than FCFS?
- Simple example
  - 10 jobs, each takes 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs start at the same time

Job #	FIFO	RR
1	100	991
2	200	992
...	...	...
9	900	999
10	1000	1000

- Turnaround Times
  - Both RR and FCFS finish at the same time
  - Average turnaround time is much worse under RR
  - Bad when all jobs same length
- Cache state must be shared between all jobs with (may slow down RR execution)
  - Total time for RR longer even for zero-cost switch

FCFS vs RR with Different Quantum

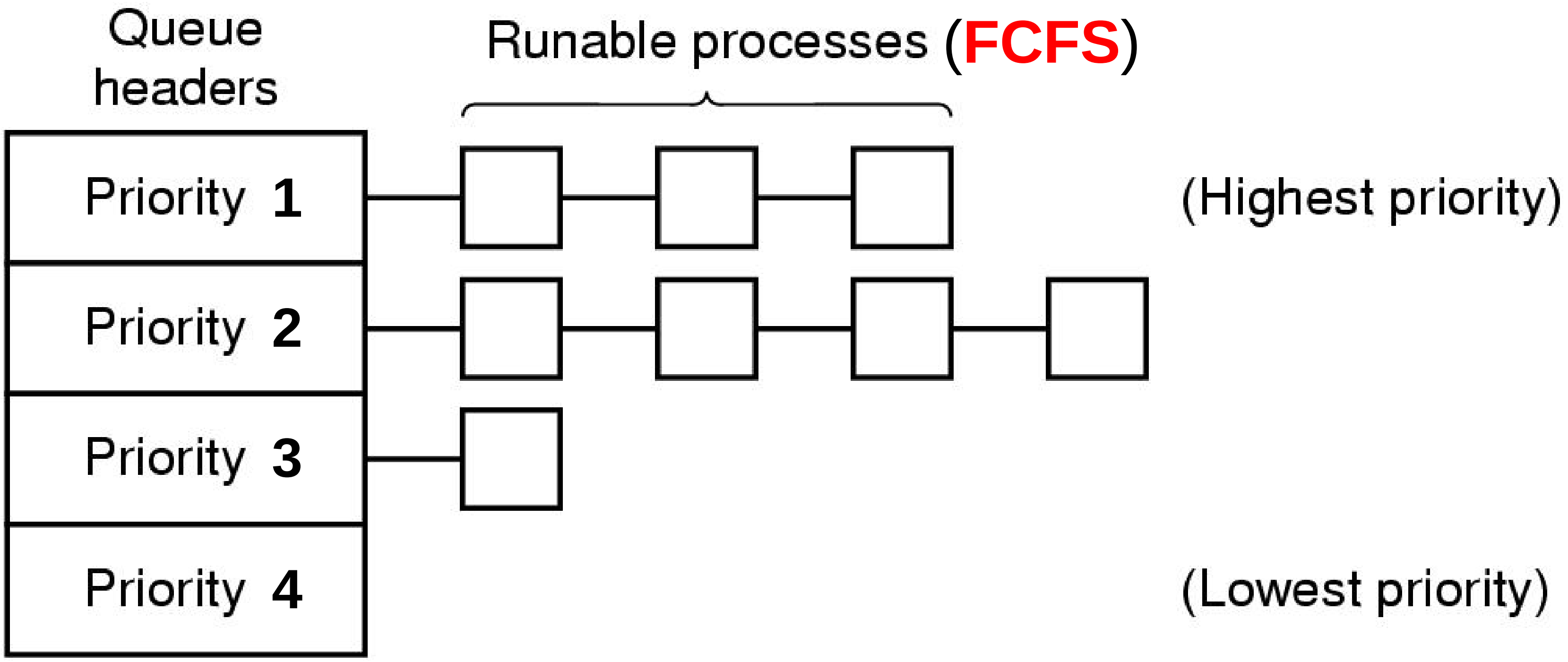
Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]
0–8	8–32	32–85	85–153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Turnaround Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Algorithm #4

Priority (PRIO) #1



A scheduling algorithm with four priority classes

Algorithm #4

Priority (PRIO) #2

- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue processed in FCFS
- Problems
  - Starvation
    - Lower priority jobs don’t get to run because higher priority tasks always running
  - Deadlock
    - Priority Inversion
      - Not strictly a problem with priority scheduling
      - Happens when low priority task has lock needed by high-priority task (busy waiting)

Algorithm #4

Priority Example

- Turnaround time
  - P1 = 16
  - P2 = 1
  - P3 = 18
  - P4 = 19
  - P5 = 6
- Average turnaround time (16+1+18+19+6)/5 = 12

Process	CPUTime	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

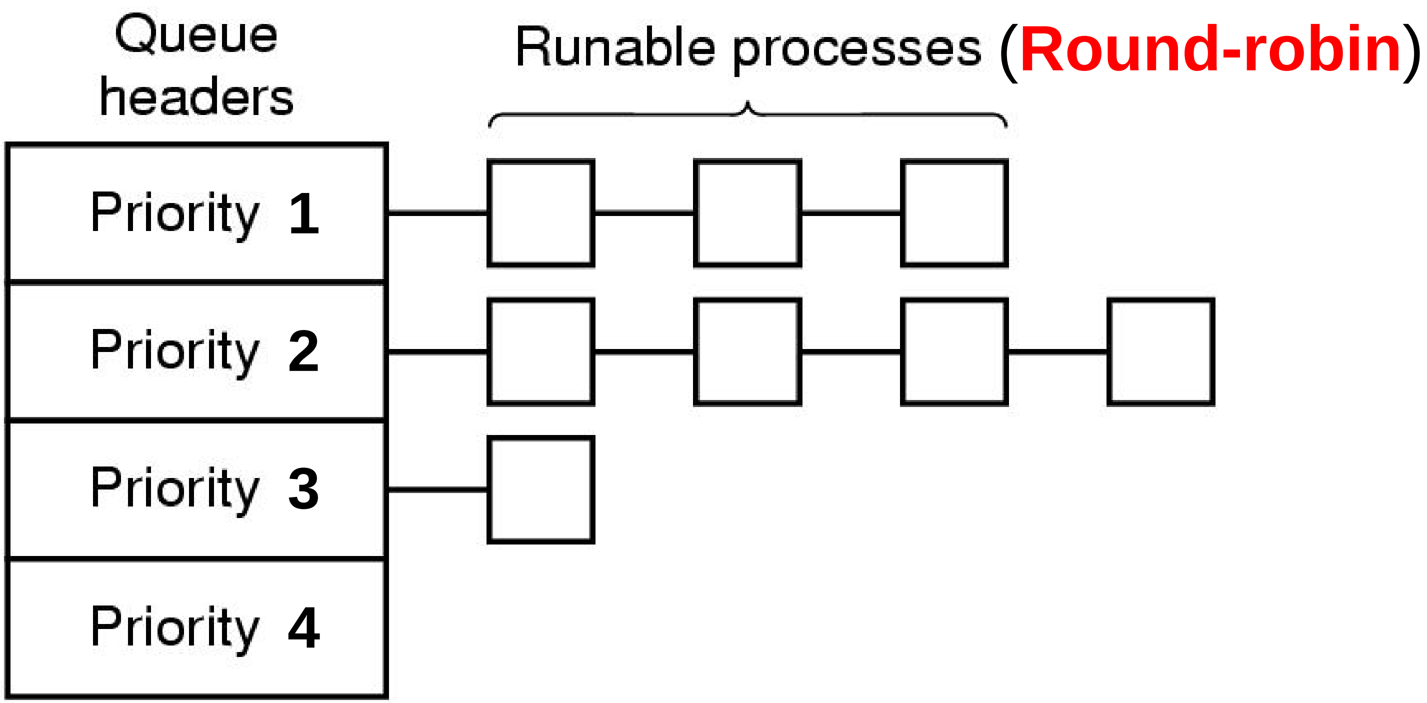
Algorithm #4

How to Assign Priorities?

- Statically, based on
  - Process type
  - User
  - How much the user paid
- Dynamically, based on how much they run vs IO
  - Priority = 1/f
  - f = size of quantum used last
    - The longer a process ran, the lower its priority
    - The process that runs the shortest gets highest priority to run next

Algorithm #5

Multiple Queues (MQ)

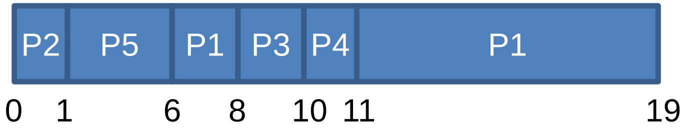


Algorithm #5

Multiple Queues Example

- Turnaround time
  - P1 = 19
  - P2 = 1
  - P3 = 10
  - P4 = 11
  - P5 = 6
- Average turnaround time (19+1+10+11+6)/5 = 9.4

Process	**CPU time**	**Priority**
P1	10	3
P2	1	1
P3	2	3
P4	1	3
P5	5	2



Quantum = 2

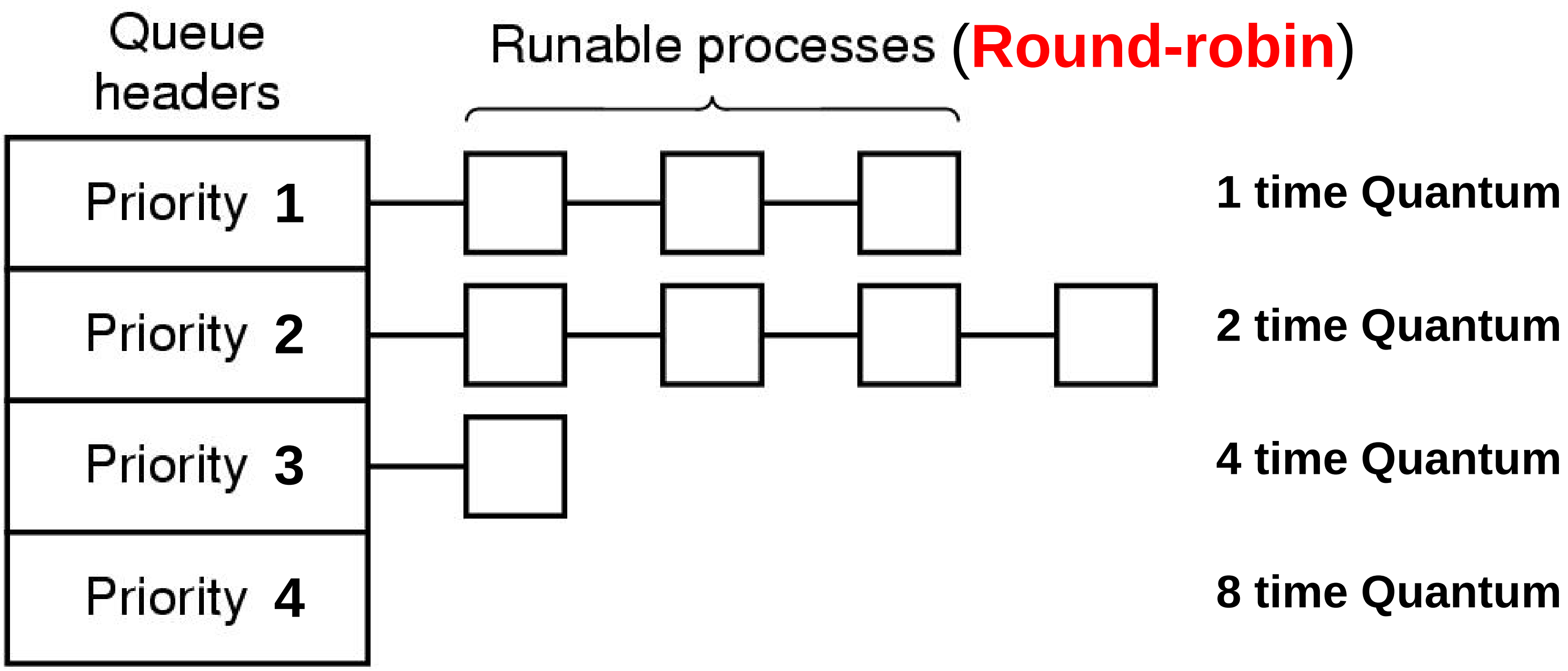
Algorithm #6

Multilevel Feedback Queue (MLFQ) #1

- Execution Plan
  - Same as MQ scheduling
  - But each queue has a different time quanta
    - Shortest for high-prio
    - Longer for low-prio
  - Processes start at the highest priority
  - When a process **exceeds** its quanta is moved to the lower priority
  - When a process **becomes interactive** is moved to higher priority
- Problem
  - If the user discovers how make his/her tasks interactive he/she can play the system

Algorithm #6

Multilevel Feedback Queue (MLFQ) #2

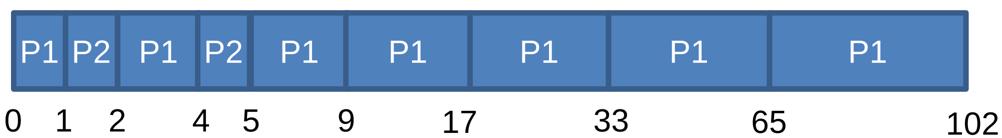


Algorithm #6

MLFQ Example

- Turnaround time
  - P1 = 102
  - P2 = 3
- Average turnaround time (102+3)/2 = 52.5
- 8 Context Switches
- vs 101 context switches with **fixed quantum**

Process	**CPU time**
P1	100
P2	2



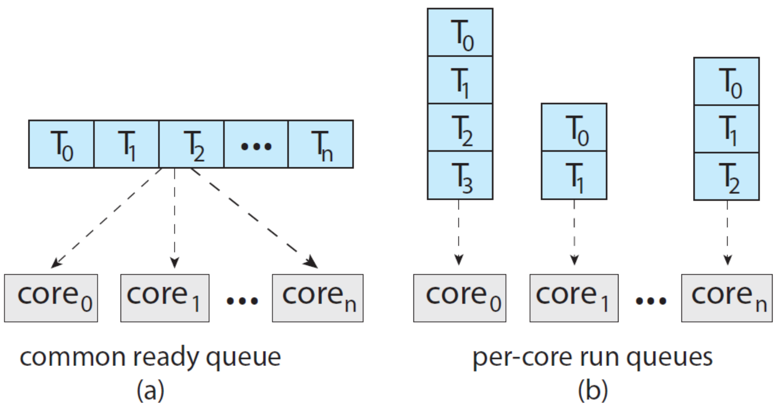
Min Quantum = 1
(1, 2, 4, 8, 16, 32)

Algorithm Evaluation

- How to select a CPU-scheduling algorithm for a system?
  - Based on **criteria/goal**
    - CPU utilization, response time, throughput, etc.
- Evaluation **methods**
  - Deterministic modeling (analytical evaluation)
    - Given algorithm and the (known) system workload, evaluate the performance
  - Queueing models (mathematical model)
  - Simulations (programming a model)
  - Actual-implementation (real-world testing)

Multicore/Multiprocessor Scheduling

- Multiple CPUs are available
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing
- **Load-sharing**
  - Processes/threads can run in parallel
- **Asymmetric** multiprocessing (AMP)
  - Common in embedded systems
- **Symmetric** multiprocessing (SMP)
  - **Widely** adopted (Linux, Windows, etc.)
    - ❑ All procs\thrs in **common** ready queue
    - ❑ Each processor **its own** ready queue
      - Work sharing/stealing
  - But, can specify **CPU affinity**



Linux Standard Scheduling, and API

- Different scheduling **classes**
  - Each class different policy (sort of MQ, but each “queue” different algorithm)
  - Implements POSIX policies
    - SCHED_FIFO, SCHED_RR – real-time (high prio)
    - SCHED_OTHER/SCHED_NORMAL, SCHED_BATCH – fair scheduling, **CFS algorithm**
    - SCHED_IDLE – idle scheduling
- To change **scheduling policy**
  - Process: `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)`
  - Thread: `int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);`
- To change **scheduling priority**
  - Process: `int setpriority(int which, id_t who, int prio);`
  - Thread: `int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);`
- To change **CPU affinity** with *multiple CPUs* (non-POSIX)
  - `int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);`

Summary

- Scheduling is a fundamental feature of OS
- Multiple goals, sometimes conflicting
- It can make a huge difference in performance
  - Difference increases with the variability in service requirements
- Many (single-CPU) algorithms
  - FCFS, SJF, RR, Priority, MQ, MLFQ
- Same algorithms adapted for multiple CPUs
- Evaluation of what algorithm is best is complex
- Real systems use hybrids