

# CS 537

## Lecture 10

### Scheduling

Michael Swift

1

## Scheduling

- In discussion process management, we talked about context switching between threads/process on the ready queue
  - but, we glossed over the details of which process or thread is chosen next
  - making this decision is called **scheduling**
    - scheduling is **policy**
    - context switching is **mechanism**
- Today, we'll look at:
  - the goals of scheduling
    - starvation
  - well-known scheduling algorithms
    - standard UNIX scheduling

2

## Types of Resources

- Resources can be classified into one of two groups
- Type of resource determines how the OS manages it
- 1) Non-preemptible resources
  - Once given resource, cannot be reused until voluntarily relinquished
  - Resource has complex or costly state associated with it
  - Need many instances of this resource
  - Example: Blocks on disk
  - OS management: **allocation**
    - Decide which process gets which resource
- 2) Preemptible resources
  - Can take resource away, give it back later
  - Resource has little state associated with it
  - May only have one of this resource
  - Example: CPU
  - OS management: **scheduling**
    - Decide order in which requests are serviced
    - Decide how long process keeps resource

3

## Multiprogramming and Scheduling

- Multiprogramming increases resource utilization and job throughput by overlapping I/O and CPU
  - We look at scheduling policies
    - which process/thread to run, and for how long
  - schedulable entities are usually called **jobs**
    - processes, threads, people, disk arm movements, ...
- There are two time scales of scheduling the CPU:
  - long term: determining the multiprogramming level
    - how many jobs are loaded into primary memory
    - act of loading in a new job (or loading one out) is **swapping**
  - short-term: which job to run next to result in "good service"
    - happens frequently, want to minimize context-switch overhead
    - good service could mean many things

4

## Scheduling

- The **scheduler** is the module that moves jobs from queue to queue
  - the **scheduling algorithm** determines which job(s) are chosen to run next, and which queues they should wait on
  - the scheduler is typically run when:
    - a job switches from running to waiting
    - when an interrupt occurs
      - especially a timer interrupt
    - when a job is created or terminated
- There are two major classes of scheduling systems
  - in **preemptive** systems, the scheduler can interrupt a job and force a context switch
  - in **non-preemptive** systems, the scheduler waits for the running job to explicitly (voluntarily) block


5

## Levels of CPU Management

- Dispatcher
  - Low-level mechanism
  - Performs context-switch
    - Save execution state of old thread in TCB
    - Add TCB to appropriate queue (ready or blocked)
    - Load state of next thread from TCB to registers
    - Switch from kernel to user mode
    - Jump to instruction in user thread
- Scheduler
  - Policy to determine which thread gets CPU when
- Allocator
  - Policy to determine which threads compete for which CPU
  - Needed for multiprocessor, parallel, and distributed systems

6

## Process Model

- Workload contains collection of jobs (processes)
  - Process alternates between CPU and I/O bursts
    - CPU-bound jobs: Long CPU bursts
- 
- I/O-bound: Short CPU bursts
- I/O burst = process idle, switch to another "for free"
  - Problem: don't know job's type before running
    - Need job scheduling for each **ready** job
    - Schedule each CPU burst

7

## Scheduling Goals

- Scheduling algorithms can have many different goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize job throughput ( $\#jobs/s$ )
  - minimize job turnaround time ( $T_{finish} - T_{start}$ )
  - minimize job waiting time ( $Avg(T_{wait})$ ): average time spent on wait queue
  - minimize response time ( $Avg(T_{resp})$ ): average time spent on ready queue
  - Maximize resource utilization
    - Keep expensive devices busy
  - Minimize overhead
    - Reduce number of context switches
  - Maximize fairness
    - All jobs get same amount of CPU over some time interval
- Goals may depend on type of system
  - batch system: strive to maximize job throughput and minimize turnaround time
  - interactive systems: minimize response time of interactive jobs (such as editors or web browsers)

8

## Scheduler Non-goals

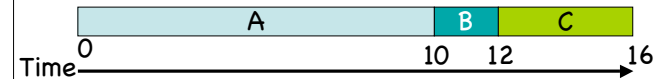
- Schedulers typically try to prevent starvation
  - **starvation** occurs when a process is prevented from making progress, because another process has a resource it needs
- A poor scheduling policy can cause starvation
  - e.g., if a high-priority process always prevents a low-priority process from running on the CPU

9

## Gantt Chart

- Illustrates how jobs are scheduled over time on CPU

Example:



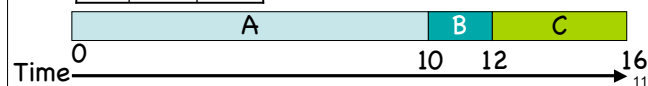
10

## First-Come-First-Served (FCFS)

- Idea: Maintain FIFO list of jobs as they arrive
  - Non-preemptive policy
  - Allocate CPU to job at head of list

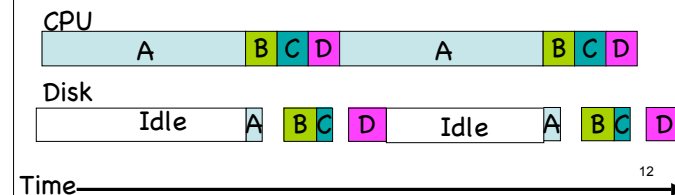
Job	Arrival	CPU burst
A	0	10
B	1	2
C	2	4

Average wait time:  
 $(0 + (10-1) + (12-2))/3 = 6.33$   
 Average turnaround time:  
 $(10 + (12-1) + (16-2))/3 = 11.67$



## FCFS Discussion

- Advantage: Very simple implementation
- Disadvantage
  - Waiting time depends on arrival order
  - Potentially long wait for jobs that arrive later
  - Convoy effect: Short jobs stuck waiting for long jobs
    - Hurts waiting time of short jobs
    - Reduces utilization of I/O devices
    - Example: 1 mostly CPU-bound job, 3 mostly I/O-bound jobs



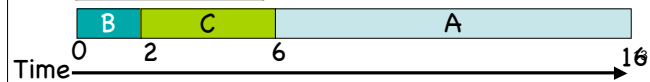
## Shortest-Job-First (SJF)

- Idea: Minimize average wait time by running shortest CPU-burst next
  - Non-preemptive
  - Use FCFS if jobs are of same length

Job	Arrival	CPU burst
A	0	10
B	0	2
C	0	4

Average wait:

Average turnaround:



## SJF Discussion

- Advantages
  - Provably optimal for minimizing average wait time (with no preemption)
    - Moving shorter job before longer job improves waiting time of short job more than it harms waiting time of long job
  - Helps keep I/O devices busy
- Disadvantages
  - Not practical: Cannot predict future CPU burst time
    - OS solution: Use past behavior to predict future behavior
  - Starvation: Long jobs may never be scheduled

14

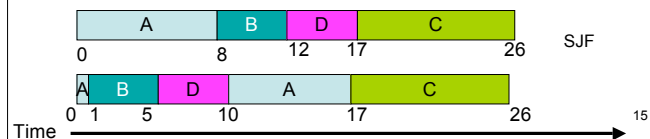
## Shortest-Time-to-Completion-First (STCF or SCTF)

- Idea: Add preemption to SJF
  - Schedule newly ready job if shorter than remaining burst for running job

Job	Arrival	CPU burst
A	0	8
B	1	4
C	2	9
D	3	5

SJF Average wait:

STCF Average wait:

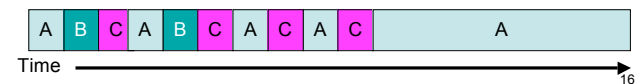


## Round-Robin (RR)

- Idea: Run each job for a time-slice and then move to back of FIFO queue
  - Preempt job if still running at end of time-slice

Job	Arrival	CPU burst
A	0	10
B	1	2
C	2	4

Average wait:



## RR Discussion

- Advantages
  - Jobs get fair share of CPU
  - Shortest jobs finish relatively quickly
- Disadvantages
  - Poor average waiting time with similar job lengths
    - Example: 10 jobs each requiring 10 time slices
    - RR: All complete after about 100 time slices
    - FCFS performs better!
  - Performance depends on length of time-slice
    - If time-slice too short, **pay overhead of context switch**
    - If time-slice too long, degenerate to FCFS

17

## RR Time-Slice

- IF time-slice too long, degenerate to FCFS
  - Example:
    - Job A w/ 1 ms compute and 10ms I/O
    - Job B always computes
    - Time-slice is 50 ms

CPU



Disk



Time →

Goal: Adjust length of time-slice to match CPU burst

18

## Priority-Based

- Idea: Each job is assigned a priority
  - Schedule highest priority ready job
  - May be preemptive or non-preemptive
  - Priority may be static or dynamic
- Advantages
  - Static priorities work well for real time systems
  - Dynamic priorities work well for general workloads
- Disadvantages
  - Low priority jobs can starve
  - How to choose priority of each job?
- Goal: Adjust priority of job to match CPU burst
  - Approximate SCTF by giving short jobs high priority

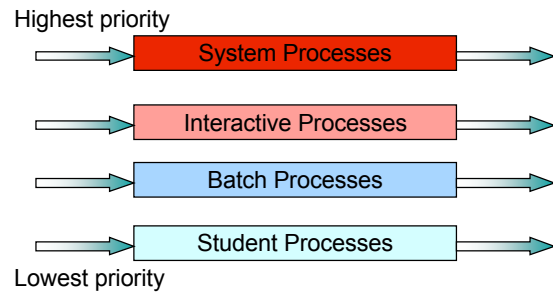
19

## Scheduling Algorithms

- **Multi-level Queue Scheduling**
- Implement multiple ready queues based on job “type”
  - interactive processes
  - CPU-bound processes
  - batch jobs
  - system processes
  - student programs
- Different queues may be scheduled using different algorithms
- Intra-queue CPU allocation is either strict or proportional
- Problem: Classifying jobs into queues is difficult
  - A process may have CPU-bound phases as well as interactive ones

20

## Multilevel Queue Scheduling



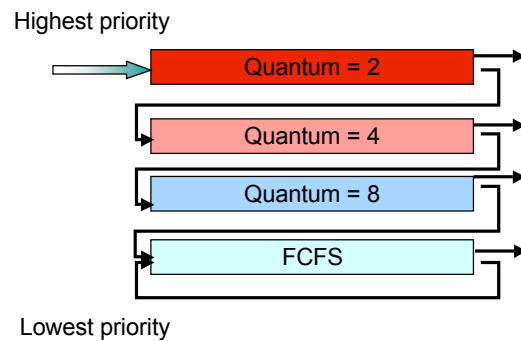
21

## Scheduling Algorithms

- **Multi-level Feedback Queues**
  - Implement multiple ready queues
    - Different queues may be scheduled using different algorithms
    - Just like multilevel queue scheduling, but assignments are not static
  - Jobs move from queue to queue based on feedback
    - Feedback = The behavior of the job,
      - e.g. does it require the full quantum for computation, or
      - does it perform frequent I/O ?
- Very general algorithm
- Need to select parameters for:
  - Number of queues
  - Scheduling algorithm within each queue
  - When to upgrade and downgrade a job

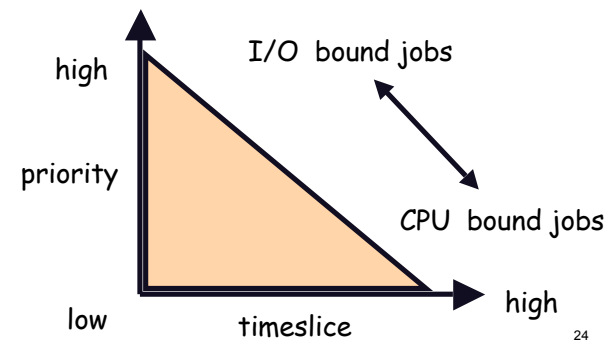
22

## Multilevel Feedback Queues



23

## A Multi-level System



24

## UNIX Scheduling

- Canonical scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - timesharing: first 60 priorities
    - system: next 40 priorities
    - real-time: next 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

25