CS 537
Lecture 3
Scheduling

Michael Swift

Scheduling

- In discussing process management, we talked about context switching between processes/thread on the ready queue
  - but, we glossed over the details of which process 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

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

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, …

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

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place 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
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

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

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)

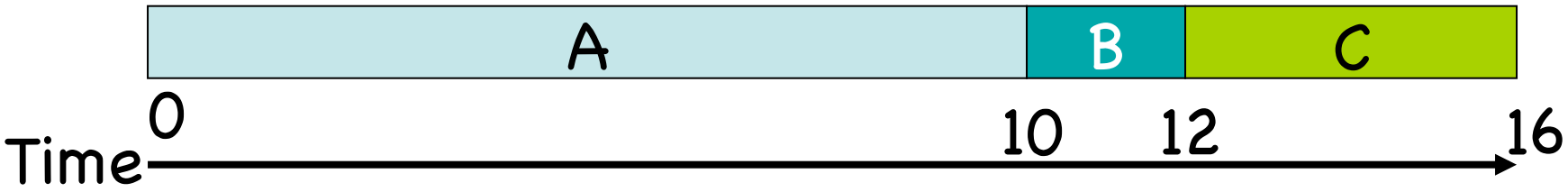
Scheduler Non-goals

- 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

Gantt Chart

- Illustrates how jobs are scheduled over time on CPU

  Example:



First-Come-First-Served (FCFS)

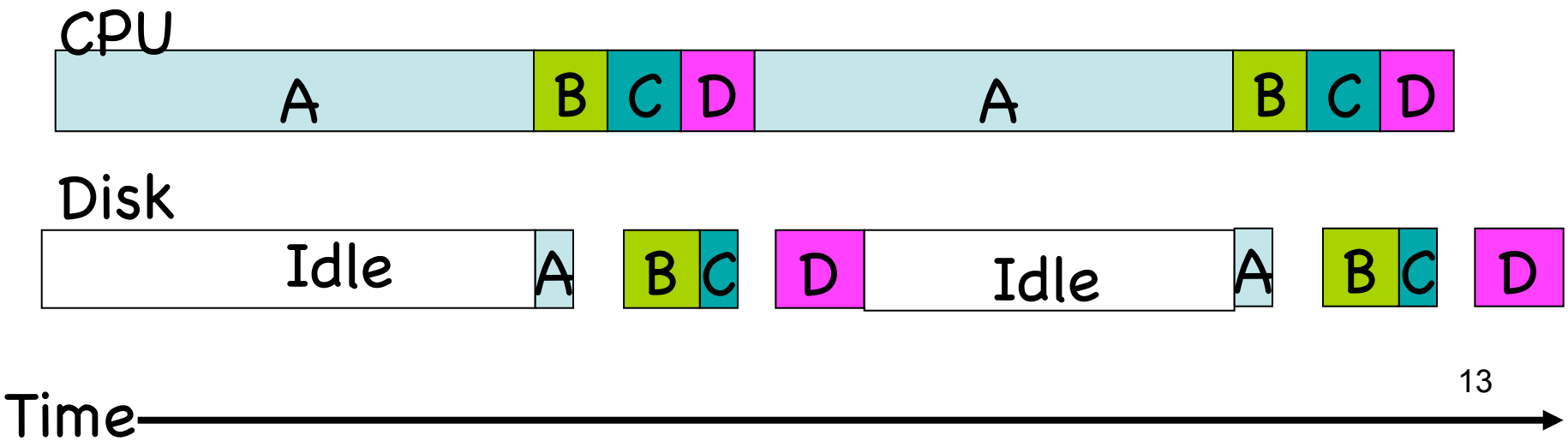
- Idea: Maintain FIFO list of jobs as they arrive
  - Non-preemptive policy
  - Allocate CPU to job at head of list

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

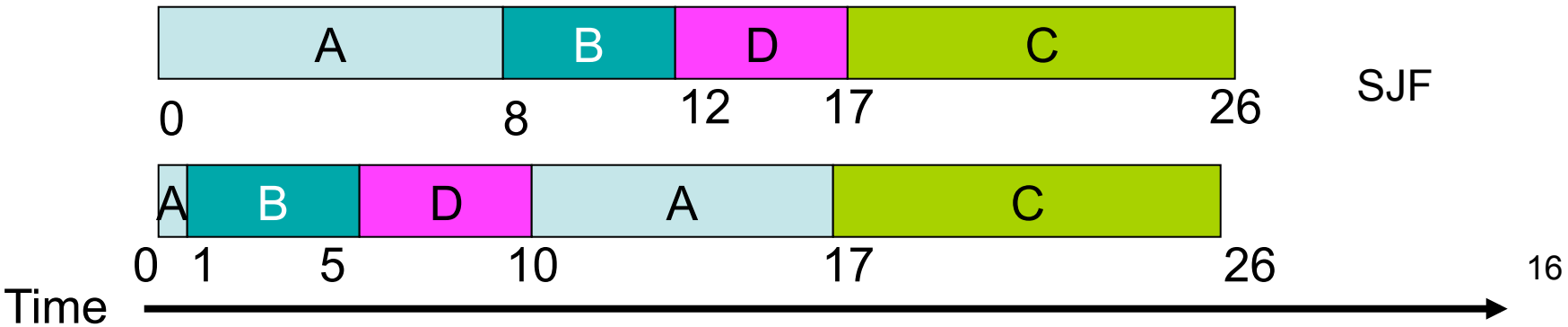
Shortest-Time-to-Completion-First (STCF or SCTF)

- 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:

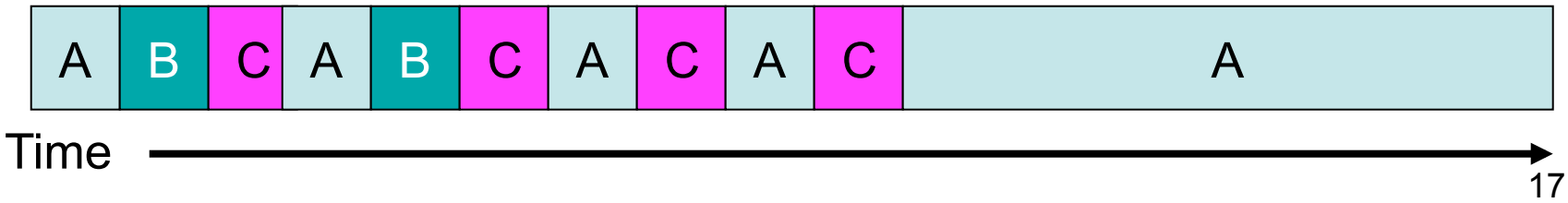


Round-Robin (RR)

- 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:

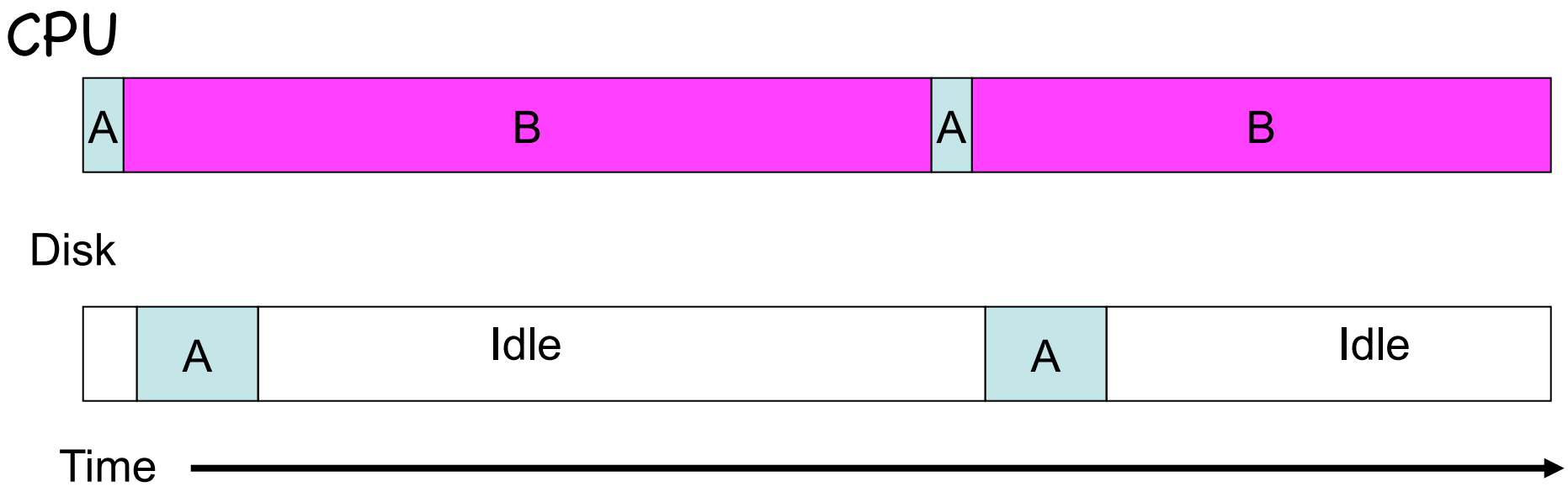


RR Discussion

- 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

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



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

Priority-Based

- Idea: Each job is assigned a priority
  - **Schedule highest priority ready job**
    - Low priority jobs wait if **any** high priority job ready
  - May be preemptive or non-preemptive
  - Priority may be static or dynamic
    - static = set once by program / user / admin
      - e.g. make the video player high priority to avoid skips
    - dynamic = adjust priorities as you run
      - foreground window is higher priority
      - Jobs that just waited for I/O get higher priority

Priority Advantages/Disadvantages

- Advantages
  - Static priorities work well for real time systems
    - guarantee jobs get serviced
    - Known set of jobs, so can assign priorities (e.g. radio in a cellphone is high priority, browser is low priority)
  - Dynamic priorities work well for general workloads
    - can react to changes in programs and 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

Priority Example

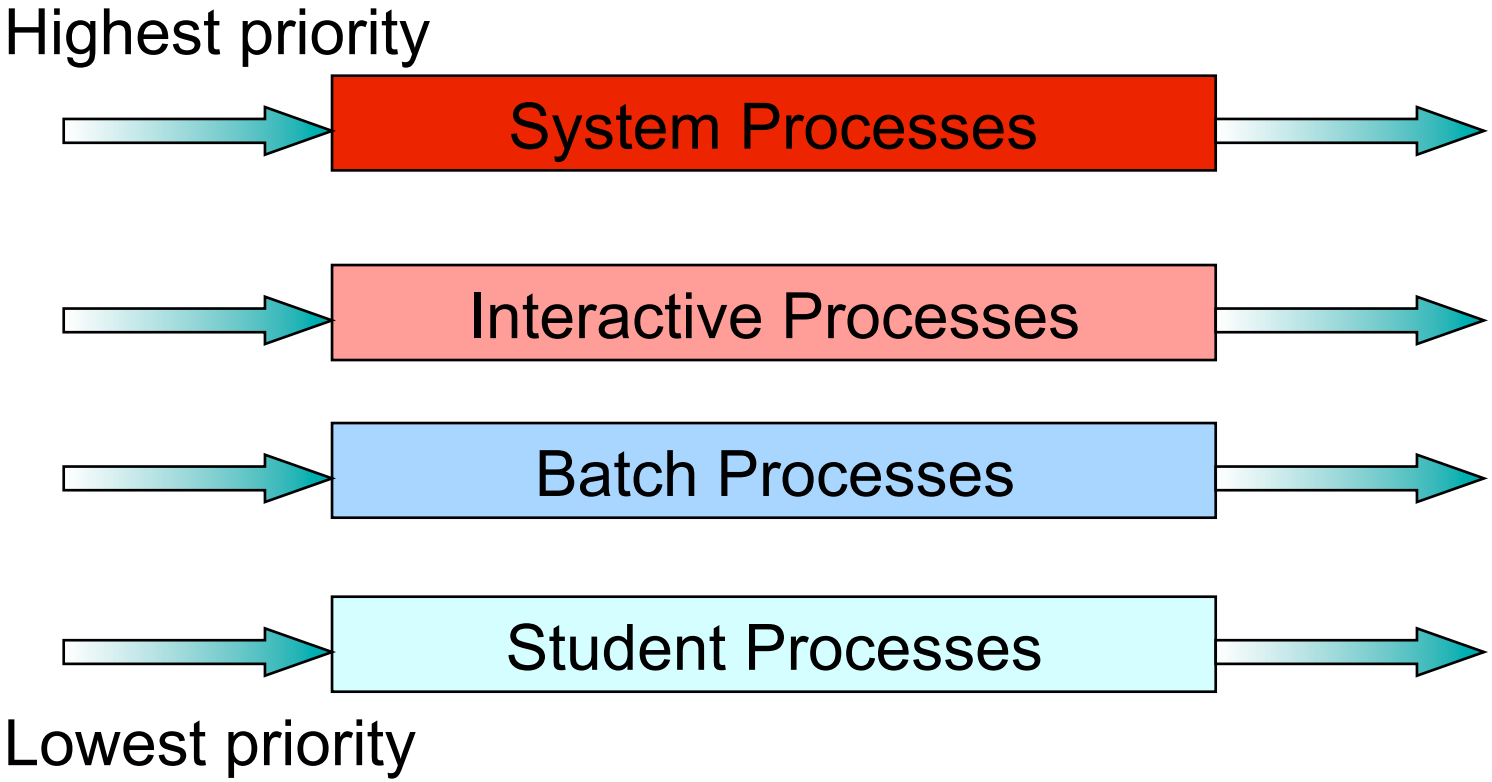
- Mechanism: sort ready queue by priority
- Higher priority is better

Job	Arrival	CPU burst	Prio
A	0	10	5
B	1	2	10
C	2	4	3

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

Multiple Priority Queues



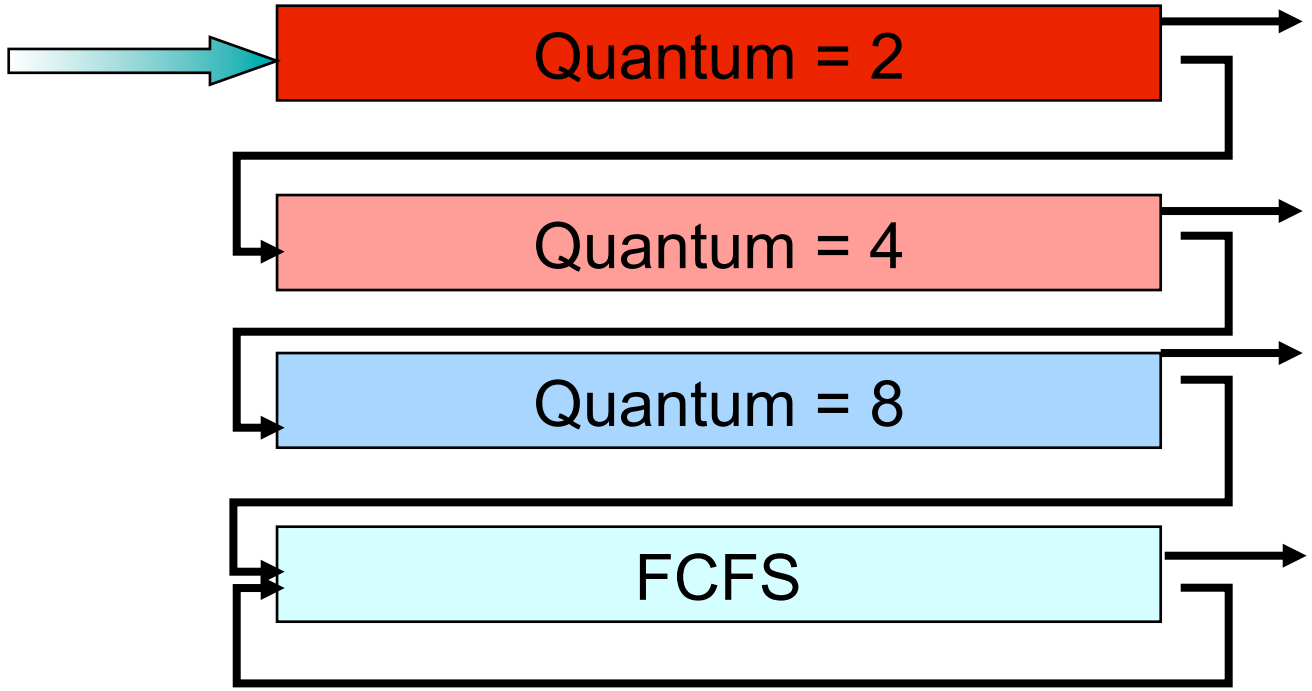
Scheduling Algorithms

- 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

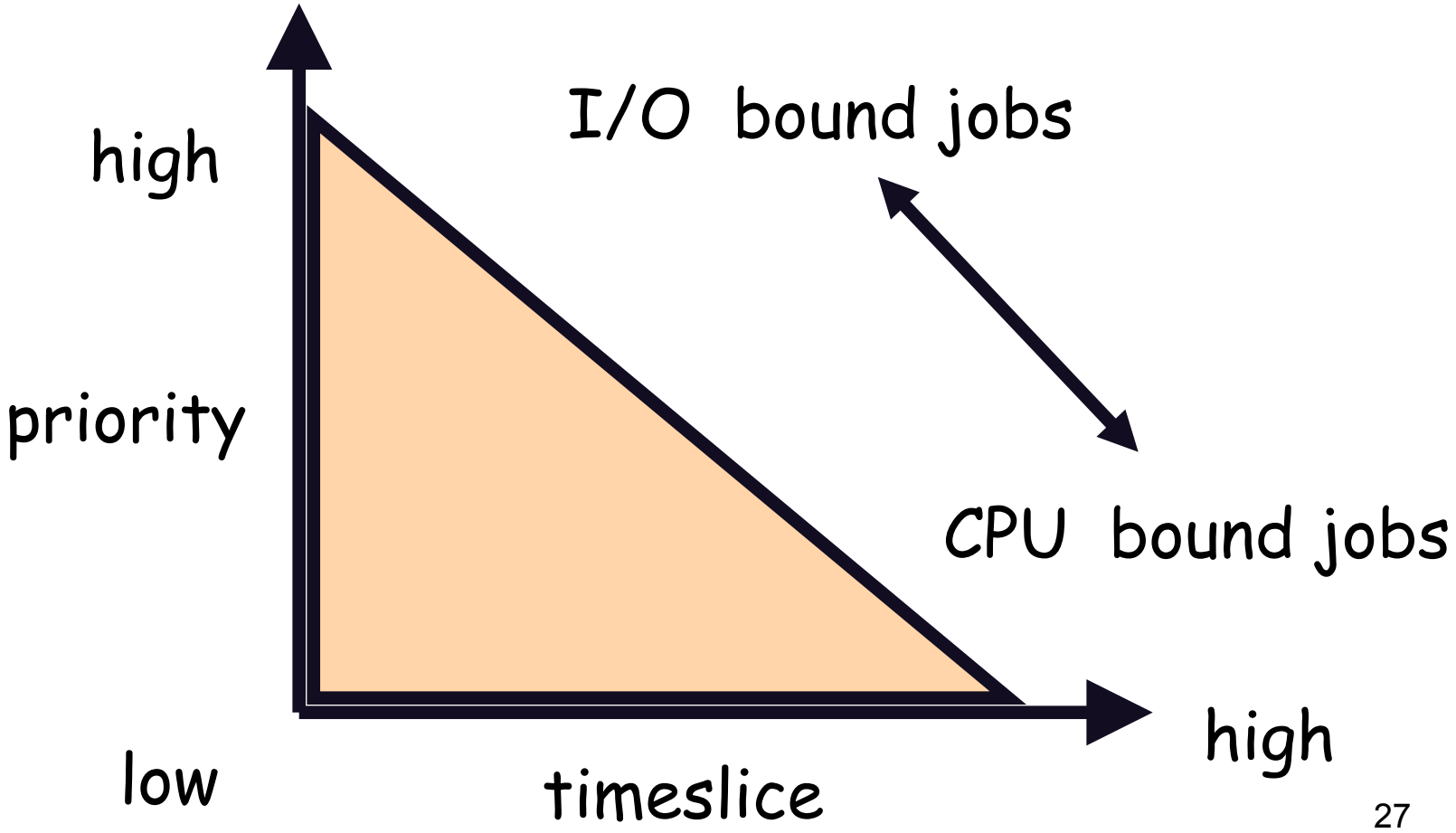
Multilevel Feedback Queues

Highest priority



Lowest priority

A Multi-level System



Solaris Schedulers

- Scheduling classes
  - Time sharing – dynamically alters priorities and timeslices – higher priority indicates lower timeslice, more responsive but less time to respond.
  - Fixed priority – priorities don't change, good for real time. Is preemptive
  - Fair share – doesn't assign priority, but shares CPU equally among processes at this level
- Preemption: will preempt lower priority thread when higher becomes able to run
- Table driven MLFQ. Priority 0 is lowest, priority 59 is highest
  - If quantum expires, priority is lowered
  - If wake up from sleep / IO, priority gets a boost
  - If waits too long without executing, gets priority boost

Table Example

Priority	Quantum (ms)	Quantum expired Prio	Return from sleep prio	MaxWait	Wait Level
0	200	0	50	10000	5
5	200	0	50	…	10
10	160	0	51		15
15	160	5	51		20
20	120	10	52		25
25	120	15	52		30
30	80	20	53		35
35	80	25	54		40
40	40	30	55		45
45	40	35	56		50
50	40	40	58		55
55	40	45	58		59
59	20	49	59		59