CPU Scheduling

Prof. Sirer
(dr. Willem de Bruijn)
CS 4410
Cornell University

Problem

- You are the cook at the state st. diner
  - customers continually enter and place their orders
  - Dishes take varying amounts of time to prepare
- What is your goal?
- Which strategy achieves this goal?

Multitasking Operating Systems

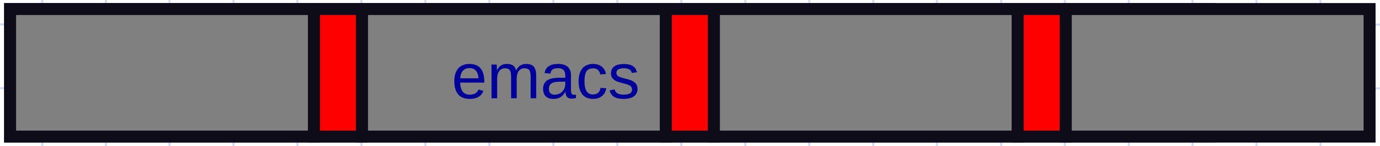
time sharing

the cpu among processes

Process Model

- Process alternates between CPU and I/O bursts
  - CPU-bound jobs: Long CPU bursts

  - I/O-bound: Short CPU bursts

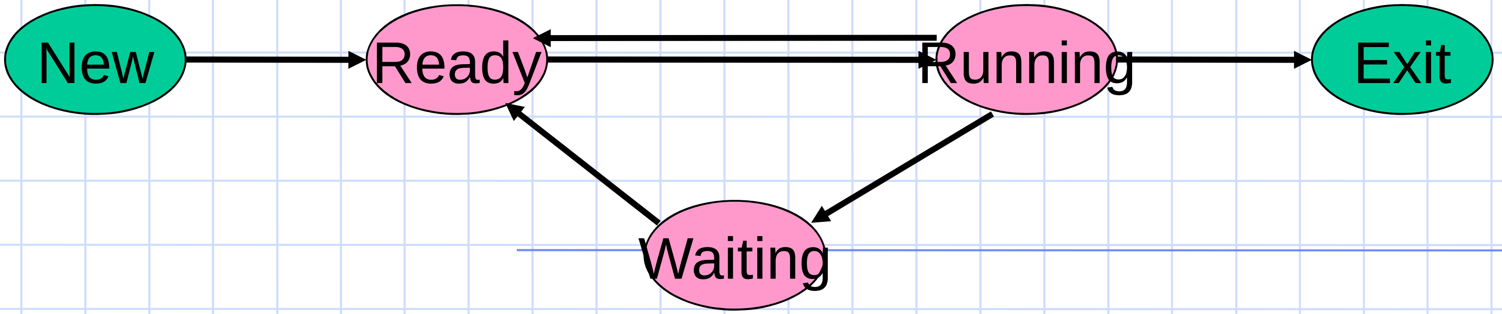


  - I/O burst = process idle, switch to another “for free”

CPU Scheduler

- Processes and threads migrate among queues
  - Ready queue, device wait queues, …
- Scheduler selects one from ready queue to run

- Which one?
  - 0 ready processes: run idle loop
  - 1 ready process: easy!
  - > 1 ready process: what to do?



multitasking

design decisions

algorithms

advanced topics

Goals

- What are metrics that schedulers should optimize for ?
  - There are many, the right choice depends on the context
- Suppose:
  - You own an (expensive) container ship and have cargo across the world
  - You own a sweatshop, and need to evaluate workers
  - You own a diner and have customers queuing
  - You are a nurse and have to combine care and administration

Scheduling Metrics

- Many quantitative criteria for evaluating scheduler algorithm:
  - CPU utilization: percentage of time the CPU is not idle
  - Throughput: completed processes per time unit
  - Turnaround time: submission to completion
  - Waiting time: time spent on the ready queue
  - Response time: response latency
  - Predictability: variance in any of these measures
- The right metric depends on the context

“The perfect scheduler”

- Minimize latency: response or job completion time
- Maximize throughput: Maximize jobs / time
- Maximize utilization: keep all devices busy
- Fairness: everyone makes progress, no one starves

Task Preemption

- Non-preemptive
  - Process runs until voluntarily relinquish CPU
    - process blocks on an event (e.g., I/O or synchronization)
    - process terminates
    - Process periodically calls the yield() system call to give up the CPU
  - Only suitable for domains where processes can be trusted to relinquish the CPU
- Preemptive
  - The scheduler actively interrupts and deschedules an executing process
  - Required when applications cannot be trusted to yield
  - Incurs some overhead

Other considerations

Uni vs. Multiprocessor

- Uniprocessor scheduling concerns itself with the selection of processes on a single processor or core
- Multiprocessor scheduling concerns itself with the partitioning of jobs across multiple CPUs or multiple cores

Best Effort vs. Real Time

Predictability:

- ABS in your car vs. navigation software
- heart rate monitor

Real time schedulers give strong predictability guarantee

Best effort schedulers provide no such guarantees

Most desktop Oses use best effort schedulers multitasking

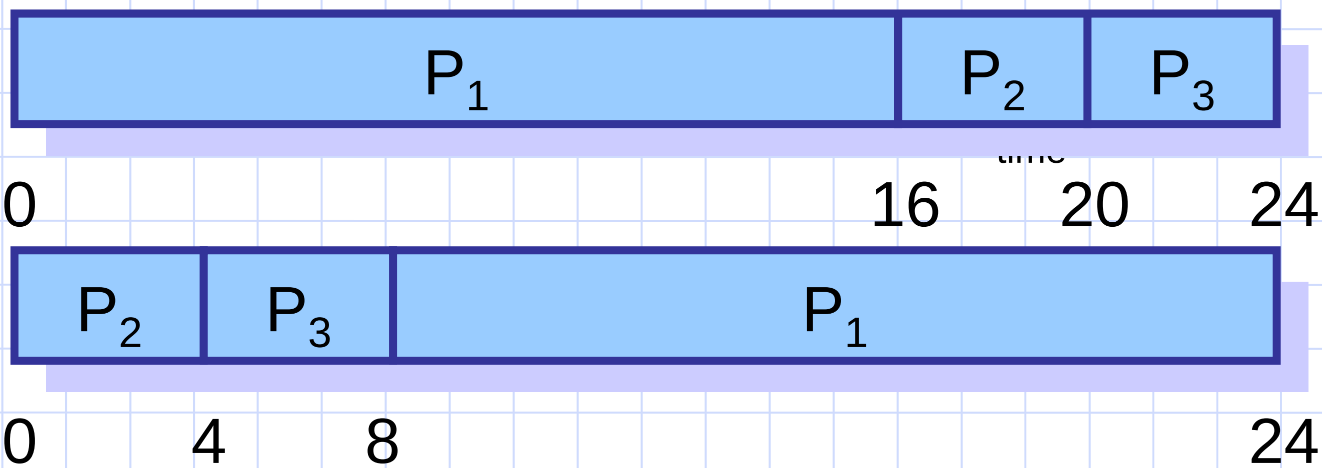
design decisions

algorithms

advanced topics

Scheduling Algorithms FCFS

- **First-come First-served (FCFS) (FIFO)**
  - Jobs are scheduled in order of arrival
  - Non-preemptive
- **Problem:**
  - Average waiting time depends on arrival order



- **Advantage:** really simple!

Scheduling Algorithms LIFO

- **Last-In First-out (LIFO)**
  - Newly arrived jobs are placed at head of ready queue
  - Improves response time for newly created threads
- **Problem:**
  - May lead to starvation – early processes may never get CPU

Round Robin

FCFS with preemption

- Often used for timesharing
- Ready queue is treated as a circular queue (FIFO)
- Each process is given a time slice called a *quantum*
- It is run for the quantum or until it blocks
- RR allocates the CPU uniformly (fairly) across participants.
- If average queue length is n, each participant gets 1/n

RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

RR: Choice of Time Quantum

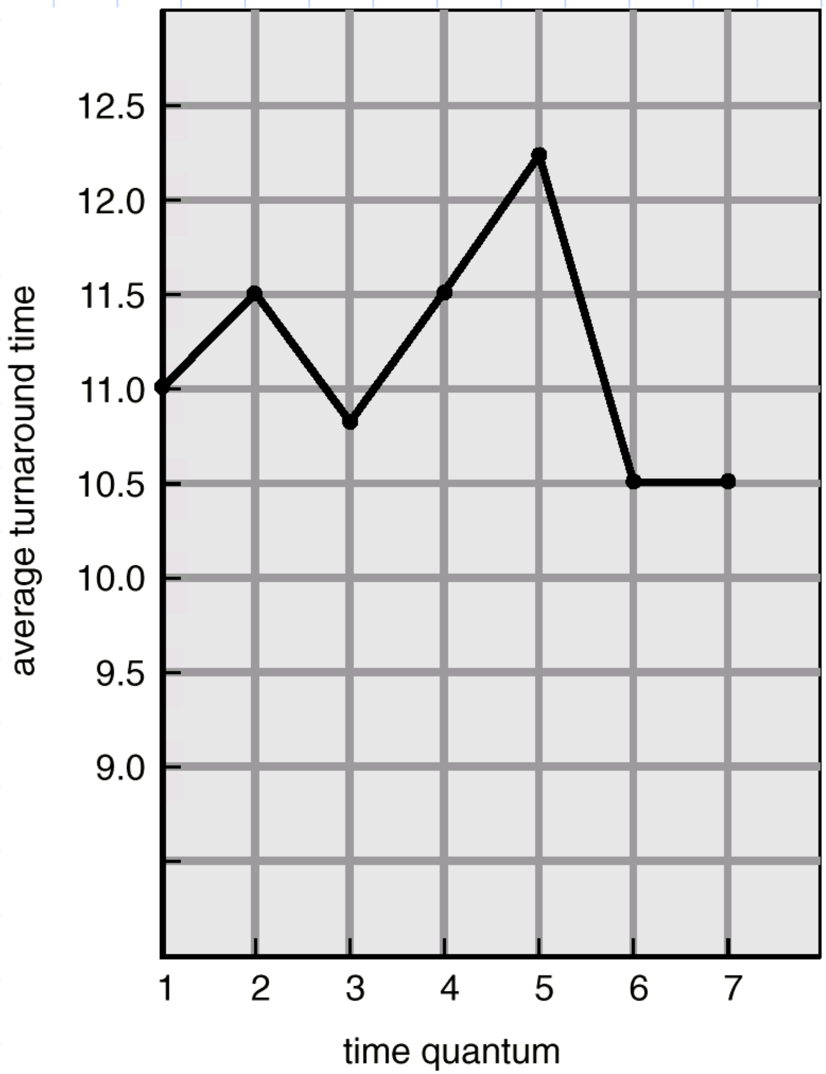
- Performance depends on length of the timeslice
  - Context switching isn't a free operation.
  - If timeslice time is set too high
    - attempting to amortize context switch cost, you get FCFS.
    - i.e. processes will finish or block before their slice is up anyway
  - If it's set too low
    - you're spending all of your time context switching between threads.
  - Timeslice frequently set to ~100 milliseconds
  - Context switches typically cost < 1 millisecond

Moral:

Context switch is usually negligible (< 1% per timeslice)

unless you context switch too frequently and lose all productivity

Turnaround Time w/ Time Quanta



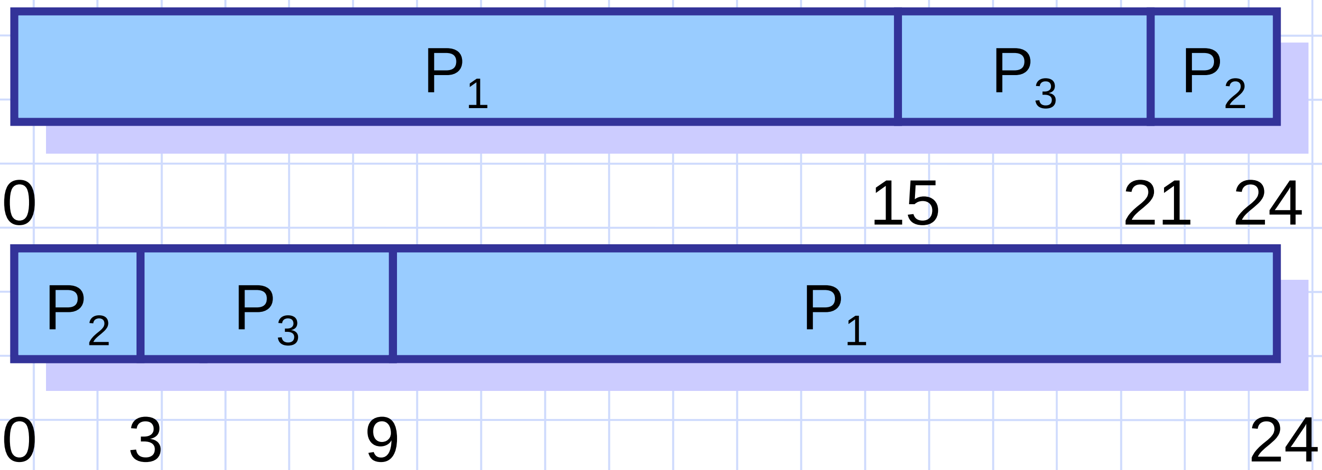
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Problem Revisited

- You work as a short-order cook
  - Customers come in and specify which dish they want
  - Each dish takes a different amount of time to prepare
- Your goal:
  - minimize average time customers wait for their food
- What strategy would you use ?
  - Note: most restaurants use FCFS.

Scheduling Algorithms: SJF

- **Shortest Job First (SJF)**
  - Choose the job with the shortest next CPU burst
  - Provably optimal for minimizing average waiting time



- **Problem:**
  - Impossible to know the length of the next CPU burst

Shortest Job First Prediction

- Approximate next CPU-burst duration
  - from the durations of the previous bursts
    - The past can be a good predictor of the future
- No need to remember entire past history
- Use exponential average:

t_n duration of the n^{th} CPU burst

τ_{n+1} predicted duration of the $(n+1)^{st}$ CPU burst

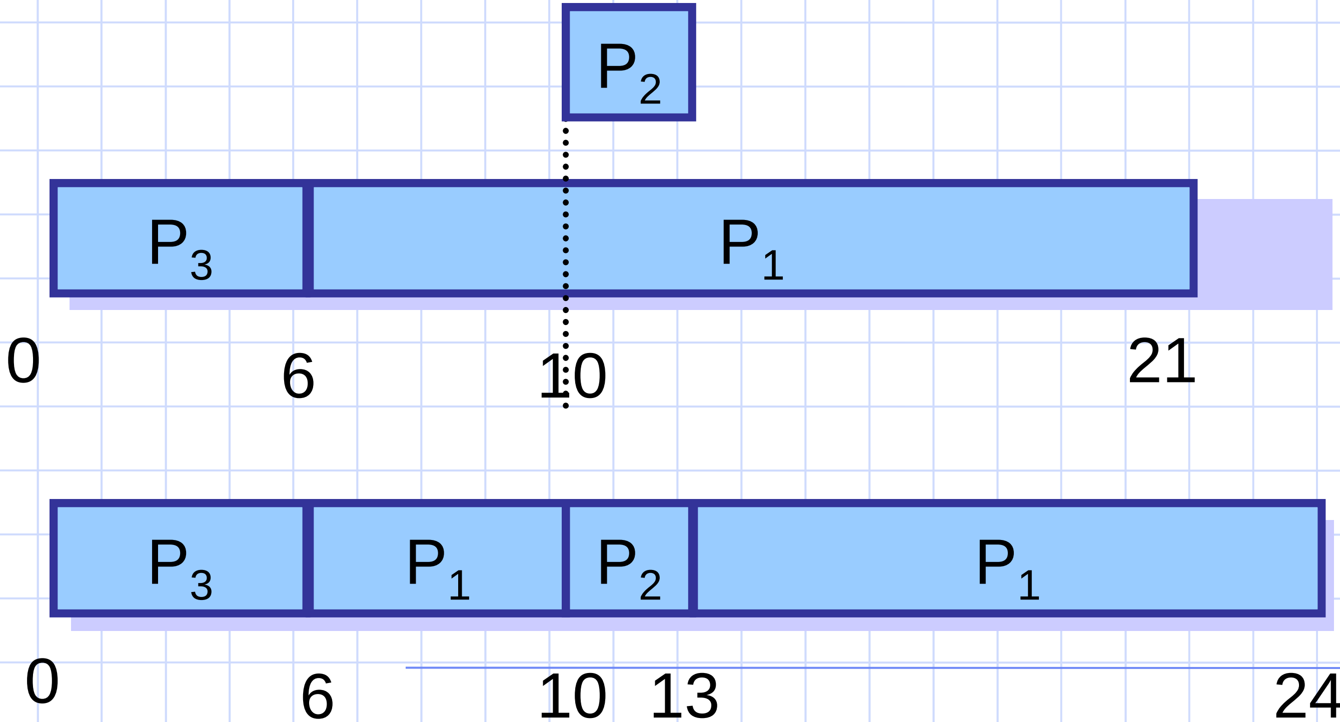
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$$

where $0 \leq \alpha \leq 1$

α determines the weight placed on past behavior

Scheduling Algorithms SRTF

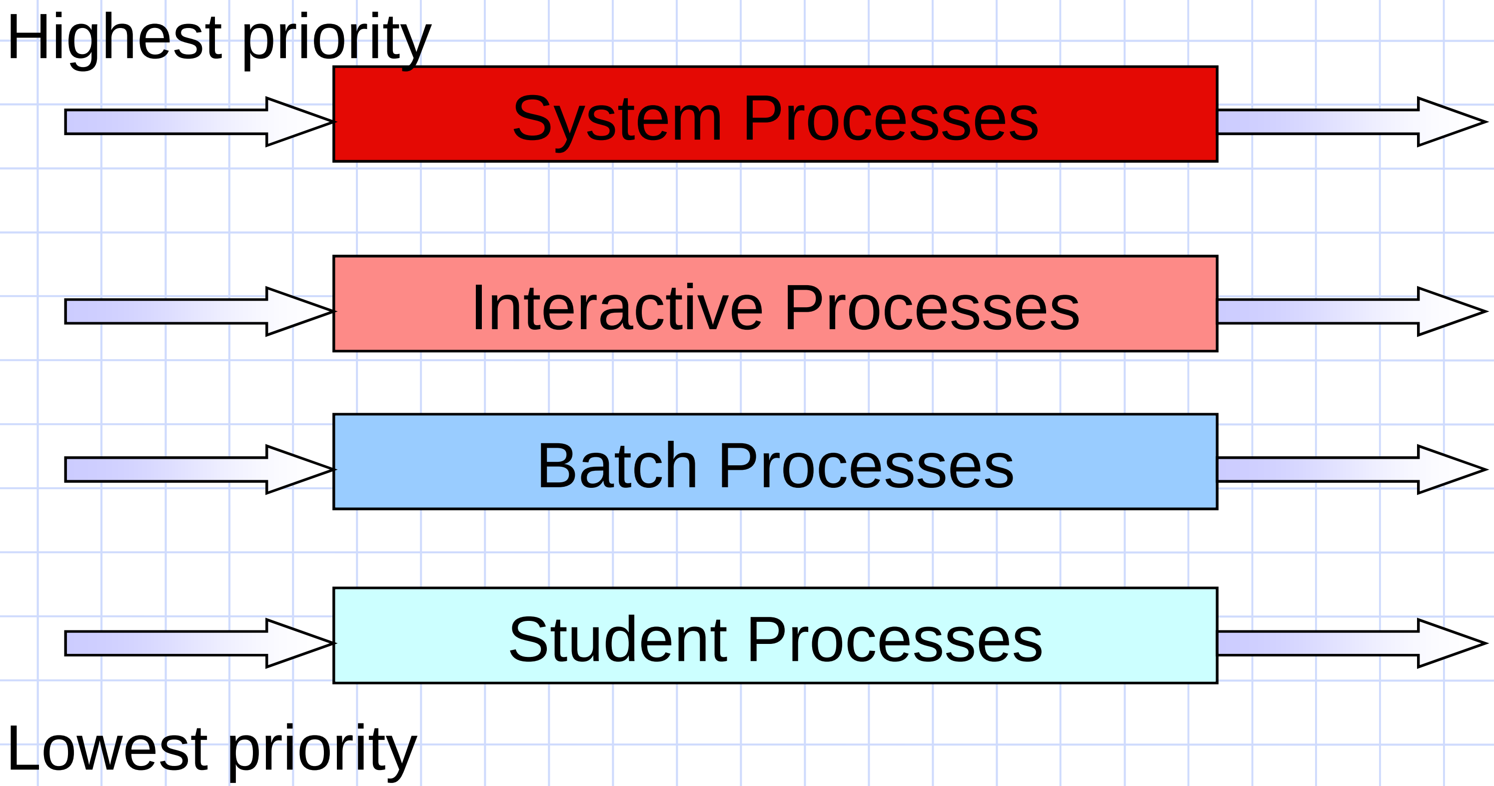
- SJF can be either preemptive or non-preemptive
  - New, short job arrives; current process has long time to execute
- Preemptive SJF is called *shortest remaining time first*



Priority Scheduling

- **Priority Scheduling**
  - Choose next job based on priority
  - For SJF, priority = expected CPU burst
  - Can be either preemptive or non-preemptive
- **Priority schedulers can approximate other scheduling algorithms**
  - **P = arrival time => FIFO**
  - **P = now - arrival time =>LIFO**
  - **P = job length => SJF**
- **Problem:**
  - Starvation: jobs can wait indefinitely
- **Solution to starvation**
  - Age processes: increase priority as a function of waiting time

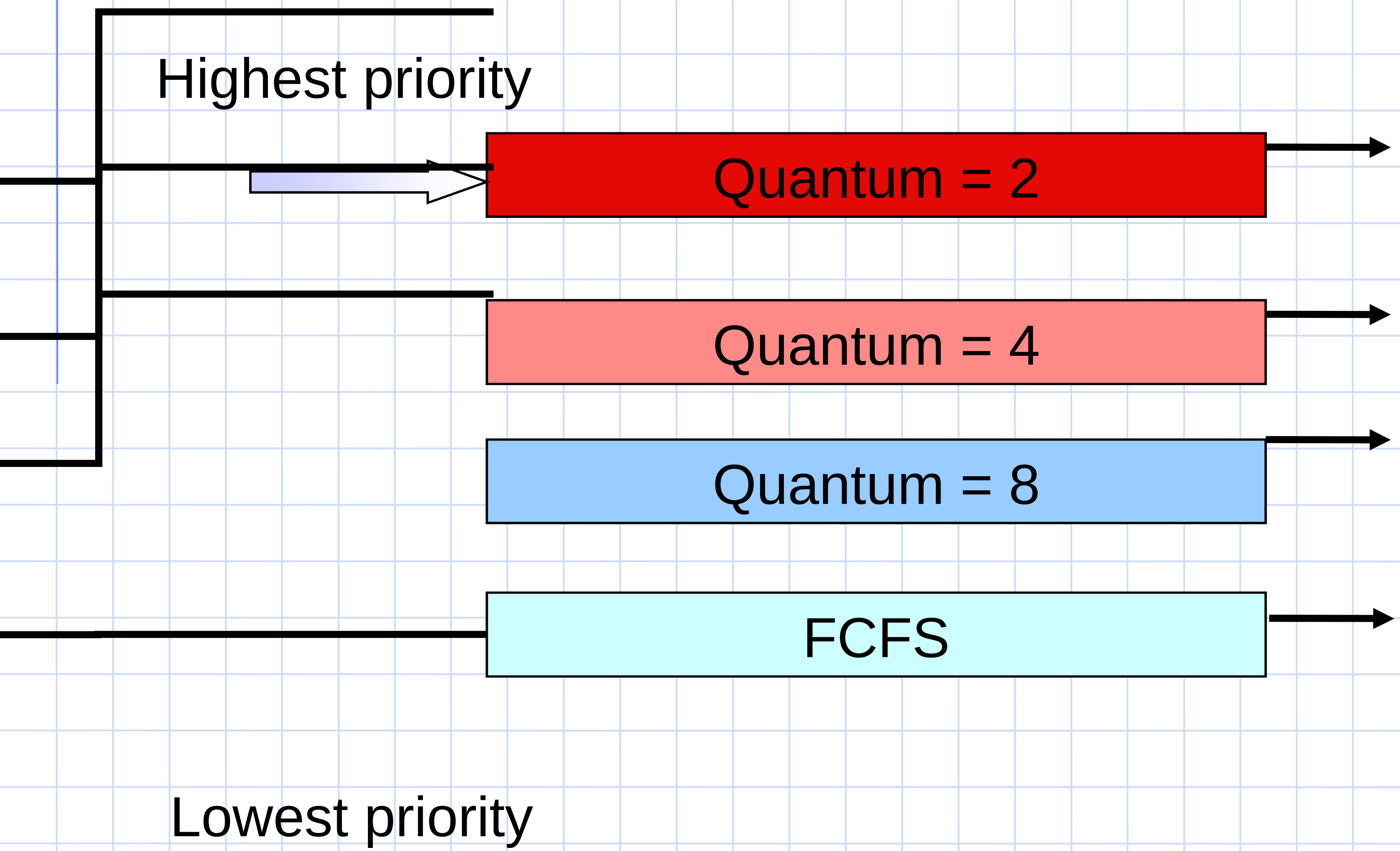
Multilevel Queue Scheduling



Multilevel 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

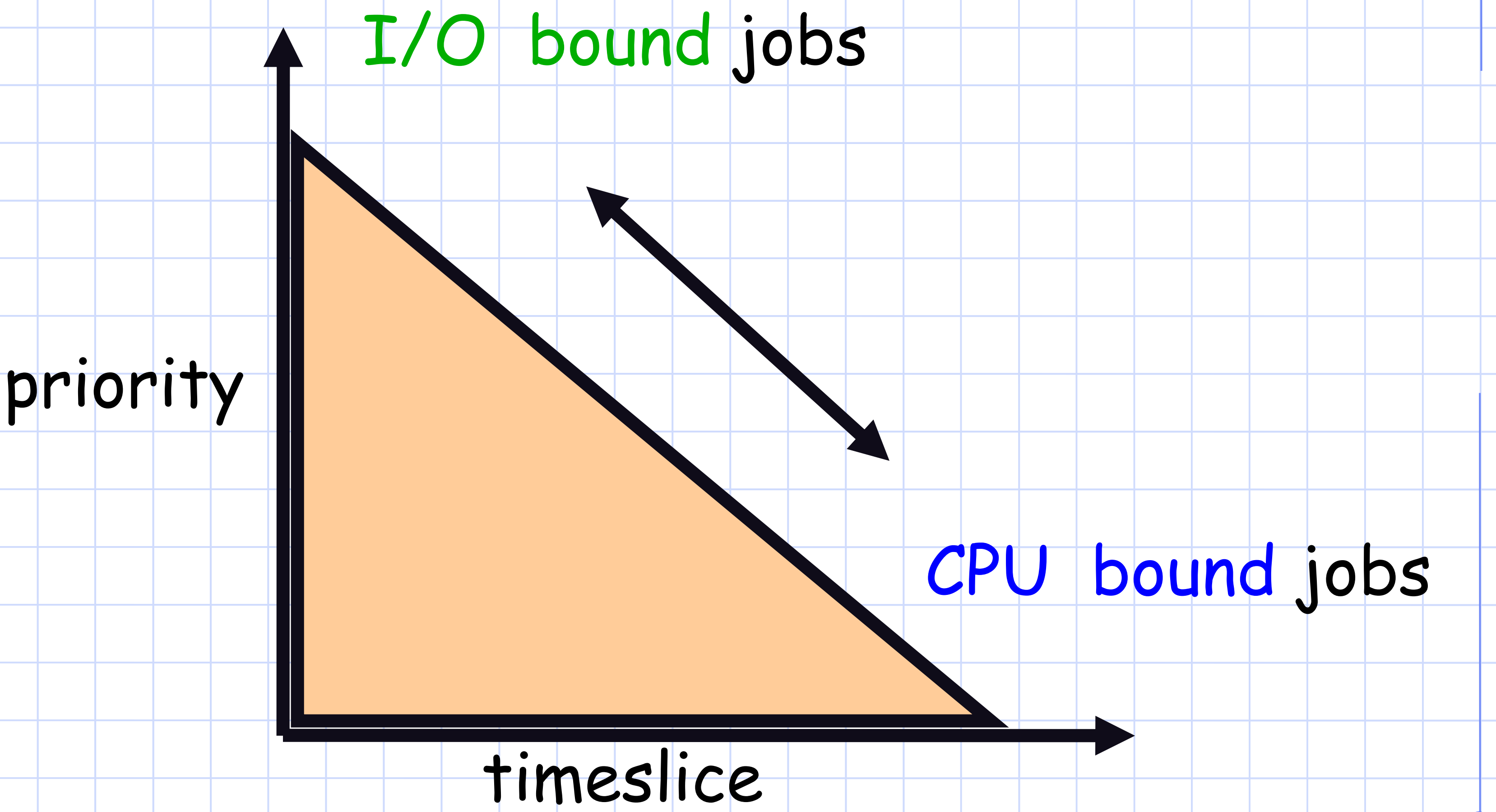
Multilevel Feedback Queues



Multilevel 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

A Multi-level System



Algorithm Summary

- FIFO
- LIFO
- RR
- SJF
- SRTF
- Priority-based
- Multilevel queue
- Multilevel *feedback* queue

multitasking

design decisions

algorithms

advanced topics

real-time, threads, multiprocessor

Real-time Scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
- Common RT scheduling policies
  - Rate monotonic
    - Just one scalar priority related to the periodicity of the job
    - Priority = 1/rate
    - Static
  - Earliest deadline first (EDF)
    - Dynamic but more complex
    - Priority = deadline
- Both require admission control to provide guarantees

Common misconception: real time does not mean fast!

Thread Scheduling

all threads share code & data segments

Option 1: Ignore this fact

Option 2: Two-level scheduling:

- user-level scheduler
- schedule processes, and within each process, schedule threads
- reduce context switching overhead and improve cache hit ratio

Multiprocessor Scheduling

Option 1. Ignore this fact

- random in time and space

Option 2. Space-based affinity:

- assign threads to processors

+ control resource sharing: sharing/interference

- possibly poor load balancing

Option 3. Gang scheduling

- run all threads belonging to a process <u>at the same time</u>

+ low-latency communication

- greater distance (cache)

Postscript

- To do absolutely best we'd have to predict the future.
  - Most current algorithms give highest priority to those that need the least!
- Scheduling has become increasingly ad hoc over the years.
  - 1960s papers very math heavy, now mostly "tweak and see"

Convoy Effect

- A CPU bound job will hold CPU until done,
  - or it causes an I/O burst
    - rare occurrence, since the thread is CPU-bound

  $\Rightarrow$ long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- Example: one CPU bound job, many I/O bound
    - CPU bound runs (I/O devices idle)
    - CPU bound blocks
    - I/O bound job(s) run, quickly block on I/O
    - CPU bound runs again
    - I/O completes
    - CPU bound still runs while I/O devices idle (continues…)
  - Simple hack: run process whose I/O completed?
    - What is a potential problem?

Problem Cases

- Blindness about job types
  - I/O goes idle
- Optimization involves favoring jobs of type “A” over “B”.
  - Lots of A’s? B’s starve
- Interactive process trapped behind others.
  - Response time suffers for no reason
- Priorities: A depends on B. A’s priority > B’s.
  - B never runs