

CPU Scheduling

(Chapters 7-11)


The Problem

- You are the cook at the State Street Diner
 - Customers enter and place orders 24 hours a day
 - Dishes take varying amounts of time to prepare
- What are your goals?
 - Minimize **average latency**
 - Minimize **maximum latency**
 - Maximize **throughput**
- Which strategy achieves your goal?

Context matters!

- What if instead you are:
 - the owner of an expensive container ship, and have cargo across the world
 - the head nurse managing the waiting room of an emergency room
 - a student who has to do homework in various classes, hang out with other students and (occasionally) sleep

Kernel Operation (conceptual, simplified)



```
Initialize devices
Initialize "first process"
while (TRUE) {
  □ while device interrupts pending
    - handle device interrupts
  □ while system calls pending
    - handle system calls
  □ if run queue is non-empty
    - select a runnable process and switch to it
  □ otherwise
    - wait for device interrupt
}
```

CPU Scheduling

Schedulers in the OS

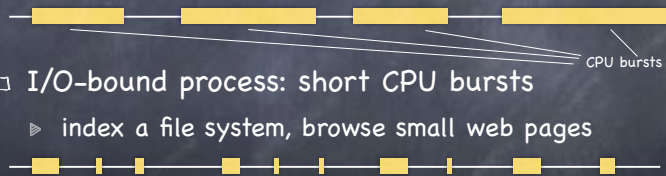
- ④ CPU scheduler selects next process to run from the ready queue
- ④ Disk scheduler selects next read/write operation
- ④ Network scheduler selects next packet to send or process
- ④ Page Replacement scheduler selects page to evict

Scheduling processes

- ④ OS keeps PCBs on different queues
 - Ready processes are on ready queue - OS chooses one to dispatch
 - Processes waiting for I/O are on appropriate device queue
 - Processes waiting on a condition are on an appropriate condition variable queue
- ④ OS regulates PCB migration during life cycle of corresponding process

Why scheduling is challenging

- ④ Processes are not created equal!
 - CPU-bound process: long CPU bursts
 - ▶ mp3 encoding, compilation, scientific applications
 - I/O-bound process: short CPU bursts
 - ▶ index a file system, browse small web pages
- ④ Problem
 - don't know jobs type before running it
 - jobs behavior can change over time

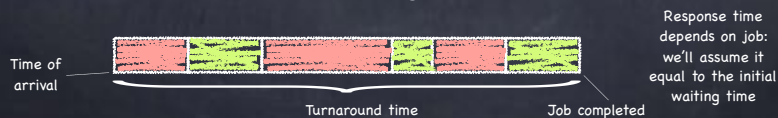


Terminology and Metrics

- ④ Job/Task
 - A user request: e.g., mouse click, web request, shell command...
- ④ Turnaround time
 - Time elapsed between a job's arrival and its completion
- ④ Throughput
 - Number of tasks completed per unit of time

More Metrics

- **Response time**
 - Time between job's arrival and first response produced
- **Initial waiting time**
 - Time between job's arrival and first time job runs
- **Total waiting time**
 - Time on the ready queue but not running
 - ▶ sum of "red" intervals below
- **Execution time:** sum of "green" intervals



Other Concerns

- **Fairness**
 - Equitable division of resources
- **Starvation**
 - Lack of progress by some job
- **Overhead**
 - Time wasted switching between jobs
- **Predictability**
 - Low variance in response time for repeated requests

The Perfect Scheduler

- Minimizes response and turnaround time
- Maximizes throughput
- Maximizes resource utilization ("work conserving")
- Meets deadlines
 - think watching a video, operating car brakes, etc
- Guarantees fairness
- Is envy-free
 - no job wants to switch its schedule with another

Alas, no such scheduler exists...

When Does the Scheduler Run?

- **Non-preemptive**
 - job runs until its actions cause it to yield CPU
 - ▶ job blocks on an event (e.g., I/O or P(sem))
 - ▶ job explicitly yields
 - ▶ job terminates
- **Preemptive**
 - all of the above, plus timer and other interrupts
 - incurs some context switching overhead

Workload assumptions

- Jobs arrive at the same time
 - but can still be ordered w.r.t. one another
- Once started, jobs run to completion
 - unless preempted
- Run-time of each job is known

Basic Scheduling Algorithms

- FIFO (First In First Out)
- SJF (Shortest Job First)
- STCF (Shortest Time-to-Completion First)
 - preemptive
- Round Robin
 - preemptive

FIFO

- Jobs J_1, J_2, J_3 with compute time 12, 3, 3
 - Job arrival J_1, J_2, J_3



FIFO

- Jobs J_1, J_2, J_3 with compute time 12, 3, 3
 - Job arrival J_1, J_2, J_3



- Job arrival J_2, J_3, J_1



Average turnaround time very sensitive to arrival time!

FIFO



The Good

Simple
Low overhead
No starvation
Optimal average turnaround time (with same-sized jobs)



The Bad

Poor average turnaround time when jobs have variable size
Average turnaround time very sensitive to arrival time



The Ugly

Not responsive to interactive tasks

SJF: Shortest Job First

- Schedule jobs in order of estimated completion time
- Optimal* average turnaround time (*att*)

*when jobs are available simultaneously

SJF: Shortest Job First

- Schedule jobs in order of estimated completion time
- Optimal* average turnaround time (*att*)
- Intuition $att = (r_1 + r_2 + r_3 + r_4 + r_5 + r_6) / 6$



- Can switching execution order reduce response time?



$$att = (r_1 + r_2 + (r_4 - c_3) + (r_5 - c_3) + r_3 + c_4 + c_5 + r_6) / 6$$

$$= (r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + (c_4 + c_5 - 2c_3)) / 6$$

*when jobs are available simultaneously

SJF



The Good

Optimal average turnaround time (when jobs are available simultaneously)



The Bad

Pessimal in how turnaround times can get far apart (see under "starvation")

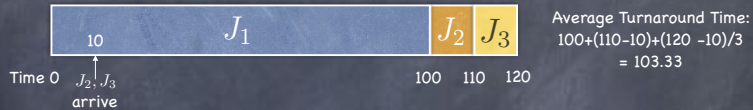


The Ugly

Needs estimate of execution times
Can starve long jobs

Relaxing "Same Arrival Time"

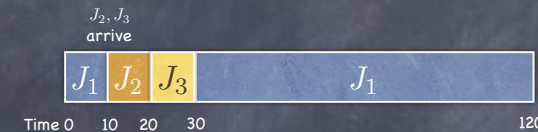
- J_1 arrives at time 0; J_2, J_3 arrive at time 10



- To retain benefits of SJF, we relax "Jobs run to completion"
 - use a preemptive scheduler

STCF: Shortest Time-to-Completion First

- On job arrival, scheduler schedules job with shortest remaining time



But what if the completion time is unknown?

Shortest Process Next (SJF for interactive jobs)

- Enqueue in order of **estimated** completion time
 - Use recent history as indicator of near future
- Let t_n = duration of n^{th} CPU burst
 τ_n = estimated duration of n^{th} CPU burst
 τ_{n+1} = estimated duration of next CPU burst

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

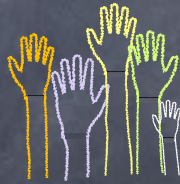
$0 \leq \alpha \leq 1$ determines weight placed on past behavior

Round Robin

- Each process is allowed to run for a **quantum**
- Context is switched (at the latest) at the end of the quantum
- What is a good quantum size?
 - Too long, and it morphs into FIFO
 - Too short, and much time lost context switching
 - Typical quantum: about 100X cost of context switch (~100ms vs. << 1ms)

Round Robin vs FIFO

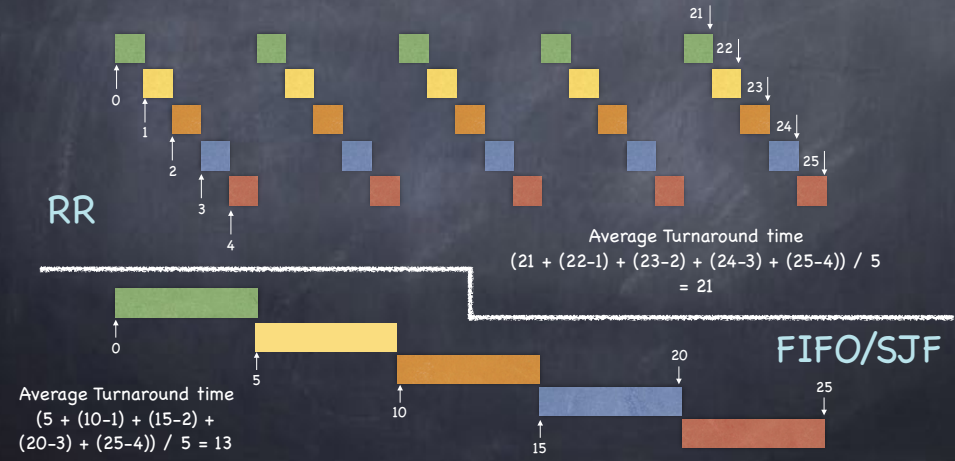
- Assuming no overhead to time slice, is Round Robin always better than FIFO?



- What is the least efficient way you could get work done this semester using RR?

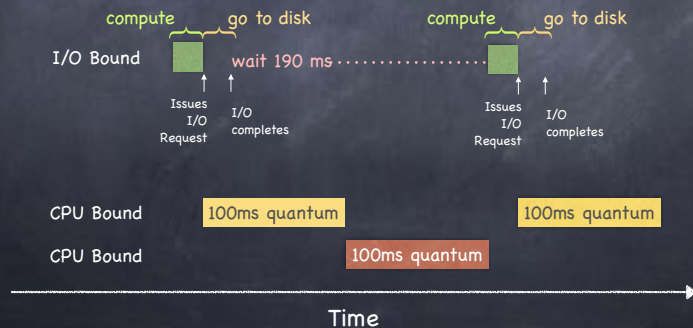
Round Robin vs FIFO

Jobs of about equal length start at about the same time



At least it is fair...?

- Mix of one I/O-bound and two CPU-bound jobs
 - I/O-bound: compute; go to disk; repeat



Round Robin



No starvation
Can reduce response time



Overhead of context switching
Mix of I/O and CPU bound



Particularly bad average turnaround for simultaneous, equal length jobs

Taking stock

- STCF has great average turnaround time, but very uneven response time
- Round Robin can reduce response time, but can have terrible att
- FIFO works well if jobs are short, but otherwise is problematic for both turnaround (unless jobs have equal size) and response time...

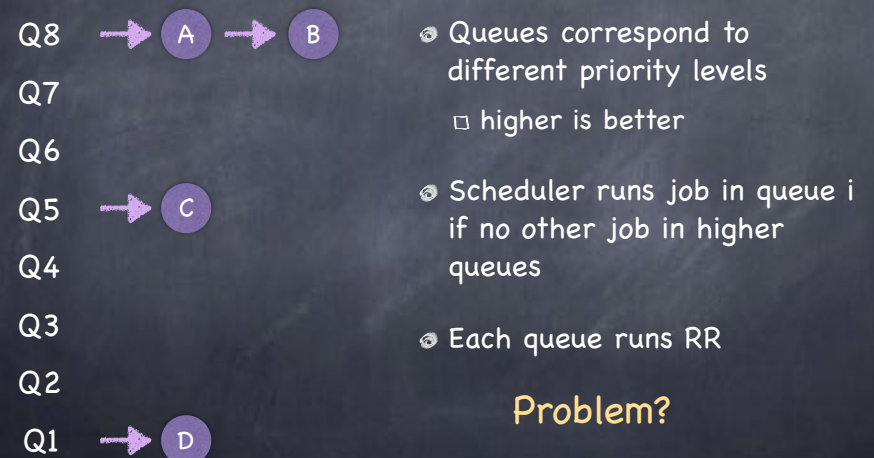
Priority Scheduling

- Assign a number (priority) to each job and schedule jobs in priority order
- Reduces to STCF if τ_{n+1} is used as priority
- To avoid starvation, change job's priority with time (aging)

Multi-level Feedback Queue (MFQ)

- Scheduler learns characteristics of the jobs it is managing
 - Uses the past to predict the future
- Favors jobs that used little CPU...
 - ...but can adapt when the job changes its pattern of CPU usage

The Basic Structure




Mobility

- Q8 → A → B
- Q7
- Q6
- Q5 → C
- Q4
- Q3
- Q2
- Q1 → D
- Job starts at the top level
 - If it uses full quantum before giving up CPU, moves down
 - Otherwise, stays where it is
 - What about I/O?
 - Job with frequent I/O will not finish its quantum and stay at the same level
- Problem?**


Movin'On Up

- Q8 → A
- Q7
- Q6
- Q5 → C
- Q4
- Q3
- Q2
- Q1 → D → B
- A job's behavior can change
 - After a CPU-bound interval, process may become I/O bound
 - Must allow jobs to climb up the priority ladder...
 - As simple as periodically placing all jobs in the top queue, until they percolate down again
- Problem?**

Sneeeeakyyy...

- Q8 → A → B
- Q7
- Q6
- Q5 → C
- Q4
- Q3
- Q2
- Q1 → D
- Say that I have a job that requires a lot of CPU
 - Start at the top queue
 - If I finish my quantum, I'll be demoted...
 - 
 - ...just give up the CPU before my quantum expires!
 - **Better accounting**
 - fix a job's time budget at each level, no matter how it is used

Proportional Share Scheduling

- Each job receives a set fraction of CPU time
 - Several approaches (see your readings)
 - Lottery scheduling
 - give jobs a number of lottery tickets in proportion to the target share
 - leverages randomness
 - Stride scheduling (deterministic)
 - stride $\propto 1/\text{tickets}$
 - when scheduled, the job "takes a stride"; strides add up to constitute the job's **pass**
 - scheduler chooses job with lowest pass
- 

Linux's CFS (Completely Fair Scheduler)

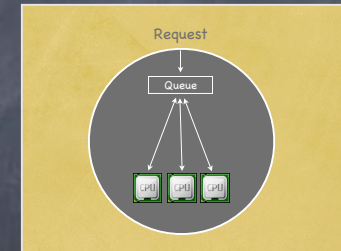
- ⑥ Tracks processes' v(irtual)runtime
 - Picks next process with lowest *vruntime*
- ⑥ All processes are equal when accounting for *vruntime*
 - but some processes are more equal than others!
- ⑥ When translating runtime to *vruntime*, p_i receives a "discount" proportional to its weight

$$vruntime_i = vruntime_i + \frac{weight_o}{weight_i} \times runtime_i$$

More
in the readings!
(and you are responsible
for them...)

Multiprocessor Scheduling: Sequential Applications

- ⑥ A web server
 - A thread per user connection
 - Threads are I/O bound (access disk/network)
 - ▶ favor short jobs!



An MFQ, right?

- Idle processors take task off MFQ
- Only one processor at a time gets access to MFQ
- If thread blocks, back on the MFQ

Single MFQ Considered Harmful

- ⑥ Contention on MFQ lock
- ⑥ Limited cache reuse
 - since threads hop from processor to processor
- ⑥ Cache coherence overhead
 - processor need to fetch current MFQ state
 - on a uniprocessor, likely to be in the cache
 - on a multiprocessor, likely to be in the cache of another processor
 - ▶ 2-3 orders of magnitude more expensive to fetch

Multiprocessor
Scheduling:
Sequential
Applications

To Each (Process), its Own (MFQ)

- ⑥ Processors use **affinity scheduling**
 - each thread is run repeatedly on the same processors
 - ▶ maximizes cache reuse
 - more complex to achieve on a single MFQ
- ⑥ Idle processors can **steal work** from other processors
 - only if it is worth the time of rewarming the cache!

Multiprocessor
Scheduling:
Sequential
Applications

Multiprocessor Scheduling: Parallel Applications

- Application is decomposed in parallel tasks

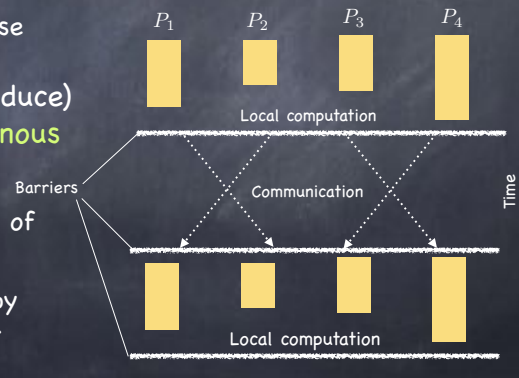
- granularity roughly equal to available processors

- or poor cache reuse

- Often (e.g., MapReduce) using **bulk synchronous parallelism (BSP)**

- tasks are **roughly** of equal length

- progress limited by slowest processor

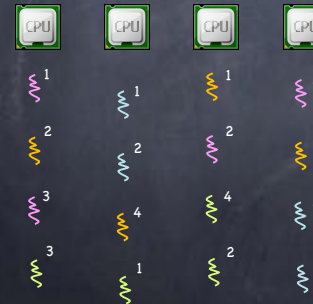


Scheduling Bulk Synchronous Applications

Oblivious Scheduling

Each process time-slices its ready list independently

Four applications, $\color{magenta}\bullet$ $\color{cyan}\bullet$ $\color{green}\bullet$ $\color{orange}\bullet$, each with four threads



Length of BSP step determined by last scheduled thread!



Gang Scheduling

Schedule all tasks from the same program together

Four applications, $\color{magenta}\bullet$ $\color{cyan}\bullet$ $\color{green}\bullet$ $\color{orange}\bullet$, each with four threads

