

# Operating Systems 2020

## Scheduling

October 30, 2020

# Scheduling

- There are multiple things to do - how do we choose, which one is done first?
- We have
  - some threads that are running
  - some threads that are ready to run
  - some threads that are blocked
- More runnable threads than processors - we need to choose

# Scheduling

- Which one is first?
- Easy - just do it in the order they arrived. Seems fair.



# Scheduling

- Is it important, how we do scheduling?
- Processors are soooo faaast anyhow....
- Actually it is. Processors can become overloaded
- Maybe not on your notebook .... but on Servers

# Scheduling

- Does not solve all problems:
  - if there are not enough resources, best scheduling wont help
- Scheduling policy important! Influences
  - **Response Time:** User perceived time to do some task
  - **Fairness:** Equality in number and timeliness of resources given to a task
  - **Throughput:** Rate at which tasks are completed

# Scheduling

- No “right” answer
- always trade-off
- We will look at some of them and discuss selecting a policy

# Example Scenario

- Running a web site for a company
- publicity hits
- suddenly you have twice as many users as the day before
- your site may appear terribly slow....

# Example Scenario

## Response time

Google, Amazon etc. estimate they lose 5%-10% of their customers if their response time increases by 100 milliseconds

(From Operating Systems, Principles & Practice, Anderson T. and Dahlin M., Recursive Books)



# Example Scenario

- scheduler may not be the main bottleneck
- good scheduling certainly important
- should you turn away users to improve performance?
- if you add a new server, how much better will performance get?

# Goals

- Users want
  - **Tasks** to be done quickly (low response time)
  - Task to be completed **predictably**: Low variance in response time
  - Tasks to be dealt with **fairly**
- Overall goals
  - High **Throughput**
  - Low **Scheduling Overhead**: The time to switch from one task to another
  - Avoid **Starvation**: Lack of progress for one task, due to resources given to a higher priority task

# Scheduling algorithm

- takes a **workload** as input

## Definition

Set of tasks for some system to perform along with when each task arrives and how long each task takes to complete.

- decides which tasks to do first
- Performance metric (throughput, latency) as output

# Road map

- Uniprocessor Scheduling
- Multiprocessor Scheduling

# Uniprocessor Scheduling

- One processor only
- three simple policies
  - first-in first-out
  - shortest-job-first
  - round-robin
- Look at strengths and weaknesses
- combine into a practical scheduler

# Workloads and Task Types

- Scheduling algorithms should work well across a variety of environments
- workloads varies from system to system and user to user
- Tasks can be
  - **compute-bound**: only use the processor
  - **I/O-bound**: most of the time wait for I/O-bound
  - **mixed**

## More terms

- Only **preemptive, work-conserving** schedulers to be considered
- Only **work-conserving** policies: never leave the processor idle when there is work to do.
- Scheduler can **preempt** the processor and give it to some other task

# First-In-First-Out

- Simplest algorithm
- also called FCFS - first-come-first-served

## FIFO

- Schedule tasks in the order they arrive
- Continue running them until they complete or give up the processor



# FIFO

## Pros

- Minimizes overhead - switch only when task complete
- best throughput when fixed number of cpu-bound tasks
- fair: every tasks waits its turn

# FIFO

## Cons

- Task with little work is behind tasks that takes a long time: has to wait
- makes system inefficient

# FIFO



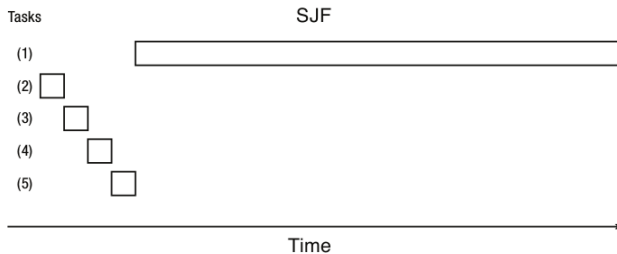
# FIFO

- simple, but maybe useful:
- Web-services store data in database
- cache in front of database: memcached (e.g Facebook)
- look up data in cache before accessing database
- requests are for small amount of data - memcached replies to requests in FIFO order
- FIFO is simple, minimizes average response time, maximizes throughput

# SJF - Shortest Job First

- Optimal policy for minimizing average response time?
- Yes: Schedule the shortest job first (SJF)
- If we know the time each task needs, we can select the task with the least work to do

# SJF



# SJF

- Downsides:
  - impossible to implement
  - worst in variance in response time
    - long tasks are done as slowly as possible
  - starvation and frequent context switches
    - enough short tasks arrive - long tasks may never complete
    - shorter task arrives: switch to it!

## Example - Supermarket

- No more Express-Kassen!
- If anyone has only a few items - go to the front!
  - current customer interrupted
  - immediate service
- full basket - you have to wait...



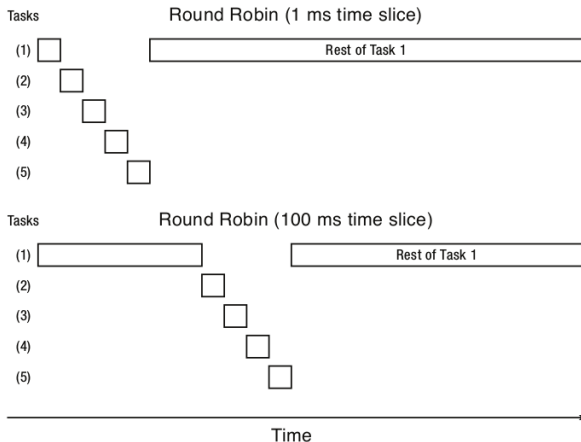
# Round Robin

- fighting starvation: schedule tasks in a round robin fashion
- Compromise between FIFO and SJF
- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line

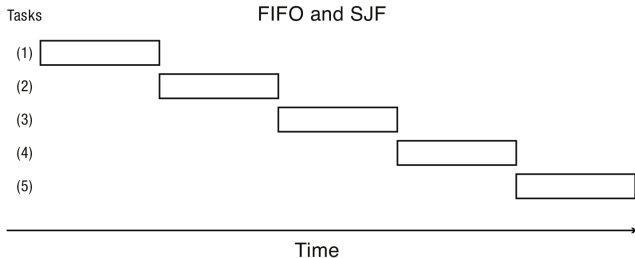
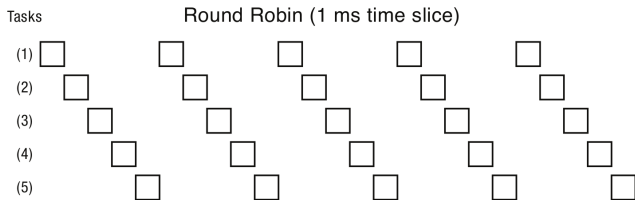
# Round Robin

- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

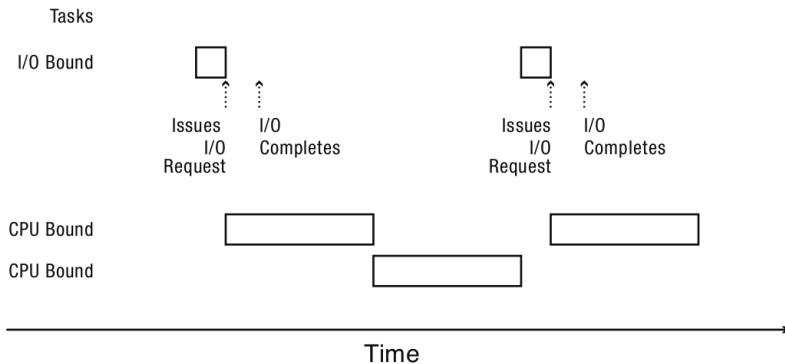
# Round Robin



# Round-Robin - equal length tasks



# RR - I/O and compute tasks



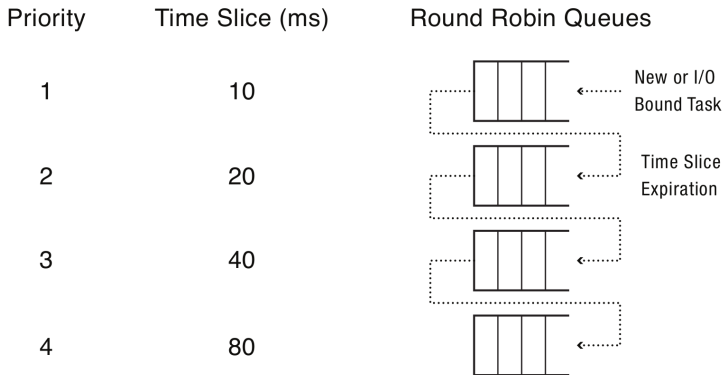
# Multi-Level Feedback

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Support priorities
  - Fairness (among equal priority tasks)
  
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

# MFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

# MFQ





# Uniprocessor Summary

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

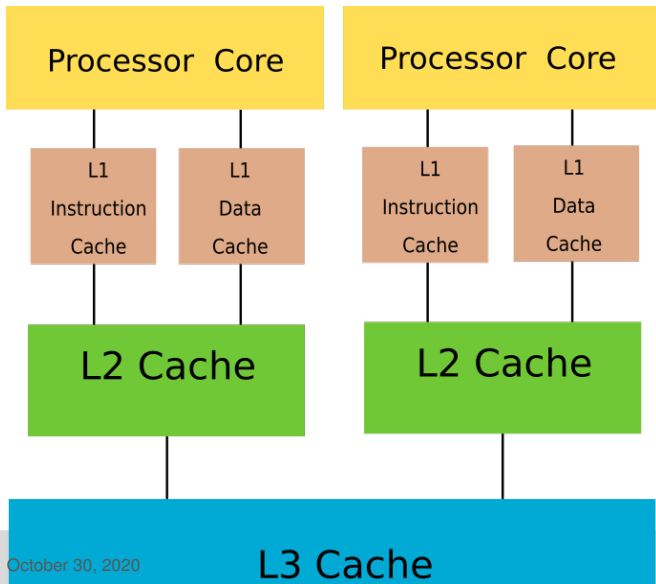
## Uniprocessor Summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

# Multi-Processor Scheduling

- Today: Most computers multi-processors
- questions:
  - How do we make effective use of multiple cores for running sequential tasks?
  - How do we adapt scheduling algorithms for parallel applications?

# Multi-Processor Architectures



# Multiprocessor Scheduling

Take into account system state and policies

- balance load across cores
- aggregate threads on a few cores and put remaining cores to sleep

if core becomes available - migrate the thread to new core

# Multi-Processor Scheduling

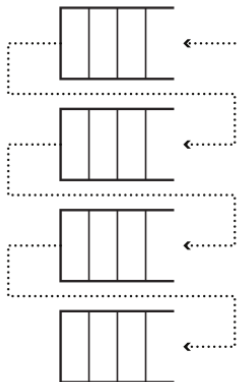
- Uniprocessor scheduling: MFQ
- What would happen if we used centralized MFQ on a multiprocessor?
  - Needs centralized MFQ lock - Contention, bottleneck
  - Cache access slowdown due to ready list data structure pinging from one CPU to another (cache coherence overhead)
  - Limited cache reuse: thread's data from last time it ran is often still in its old cache (on a different processor maybe)

# Per-Processor Affinity Scheduling

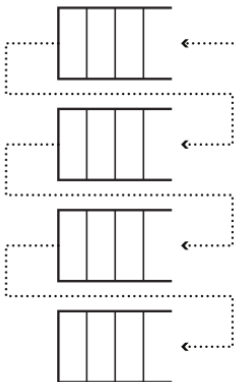
- Each processor has its own ready list
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run
  - Ex: when I/O completes, or on Condition->signal
  - maximizes cache reuse
- Idle processors can steal work from other processors
- per-processor data structures must still be locked

# Per-Processor Multi-level Feedback

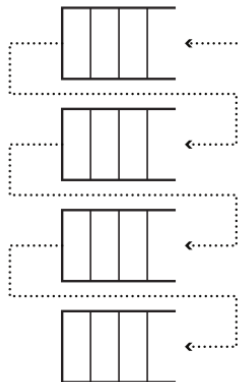
Processor 1



Processor 2



Processor 3





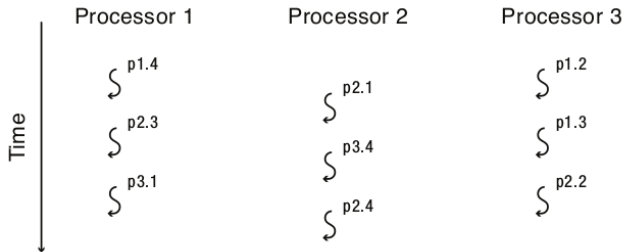
# Scheduling Parallel Programs

- Scheduling parallel application poses different challenges
- often natural decomposition of a parallel app onto a set of processors
- Example: image processing
  - divide image into chunks
  - assign one to each processor

## Scheduling Parallel Programs (2)

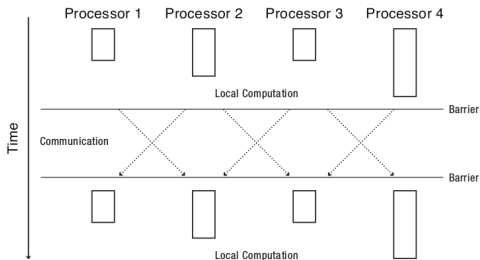
- Issues
  - no relationship between number of threads running and number of processors available
    - other processes or OS also need resources
- Use scheduling algorithm as discussed (oblivious scheduling)
  - multi-level feedback ensures all get fair share
  - OS schedules threads as independent entities
  - scheduler operates without knowledge of the intent of the application

# Oblivious Scheduling



px.y = Thread y in process x

# Bulk Synchronous Delay

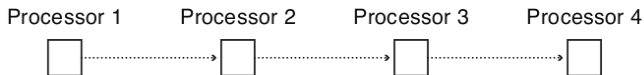


Computation limited by slowest processor involved

# Bulk Synchronous Delay

- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
  
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP
    - Sacrificing a small constant factor in performance

# Producer-Consumer-Delay



- preempting one thread stalls all others following in the chain



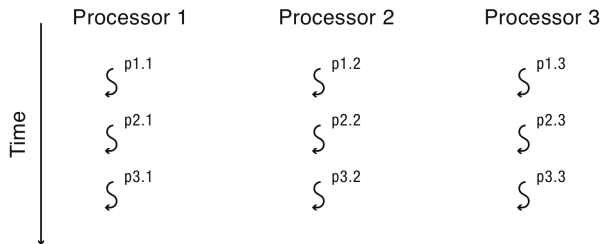
## Other issues II

- spin-then-wait strategy: short spin-waits, if lock unavailable block and look for other work to do
  - reduces overhead when lock is held for short period of time
  - problem when lock holder is preempted
  
- I/O



# Gang Scheduling

- Application picks decomposition of work into some number of threads
- threads run either together or not at all



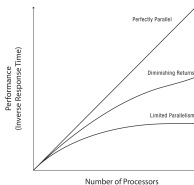
px.y = Thread y in process x

# Gang Scheduling

- Linux, Windows, MacOS: mechanisms for dedicating a set of processors to an application
- good on server with single primary use (e.g. database)
- application can pin threads to specific processor
- system reserves subset of processors to other applications

# Gang Scheduling I

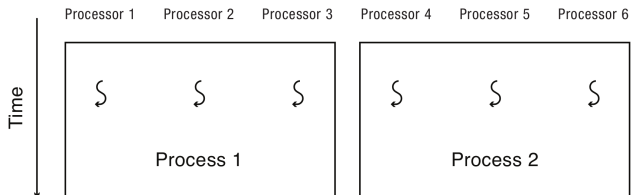
- Effect of adding processors to an application depends on application
  - some make efficient use of many processors
  - some have diminishing return



# Space Sharing

- Usually more efficient to run two parallel programs with half the number of processors than assigning all processors to one program
- different processors to different tasks: space sharing
- single processor to multiple tasks: time sharing, time slicing
- minimizes processor context switches

# Space Sharing



# Scheduler Activations

- Application can be informed by an “upcall” from the OS when a processor is added/taken away or a thread blocks
- Application can implement its own scheduling mechanism
- Open question: how many processors should we assign?

# Overload Management I

- Most systems: no direct control on workload.
- Whatever scheduling policy: if there is more load than the system can handle: response time will drop
- Key idea: do less work when overloaded
  - question: can we choose what to disable or do we let events choose?
- Obvious solution:
  - reject requests

# Overload Management II

- allows to preserve reasonable response time for remaining ones
- harsh but pragmatic

## Video Streaming

Reject to start new streams to be able to continue providing good streaming service to users that already have started

- Alternative: reduce service time per request



# Overload Management III

- Replace dynamic web page creation by static content
- reduce bit rate for movies
- Amazon has designed its web page to always return a result quickly
  - better to give a wrong answer quickly and later apologize than let user wait