Operating Systems 2019
Scheduling

October 16, 2019 —

Scheduling

- there are multiple things to do - how do we chose, 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 do 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 are overloaded
- Maybe not on your notebook .... but on Servers
- Scheduling policy influences performance

Scheduling

- Does not solve all problems:
  - if there are not enough resources, best scheduling wont help
- Policy influences
  - response time
  - fairness
  - throughput

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

- quickly implement a different scheduling policy - does this help?
- how much worse will the performance get if the number of users doubles again?
- should you turn away users to improve performance?
- does it matter which users you turn away?
- if you buy a new server, how much better will performance get?
- are there different answers if you are under a denial-of-service attack?

Terms

- **Task**: A user request
- **Response** Time: User perceived time to do some task
- **Predictability**: Low variance in response time
- **Throughput**: Rate at which tasks are completed
- **Scheduling Overhead**: The time to switch from one task to another
- **Fairness**: Equality in number and timeliness of resources given to a task
- **Starvation**: Lack of progress for one task, due to resources given to a higher priority task

Scheduling algorithm

- takes a workload as input
- decides which tasks to do first
- Performance metric (throughput, latency) as output
- Only preemptive, work-conserving schedulers to be considered

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

More terms

Definition

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

Input to scheduling algorithm

More terms

- 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

- Some policies are best for a workload
- ... and some are worst
- we only compare policies that are **work-conserving**:

Definition

A scheduler is work-conserving, if it never leaves the processor idle when there is work to do.

More terms

- We assume a scheduler has the ability to **preempt** the processor and give it to some other task
- happens after an interrupt
- there are non-preemptive schedulers, but they are not important

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

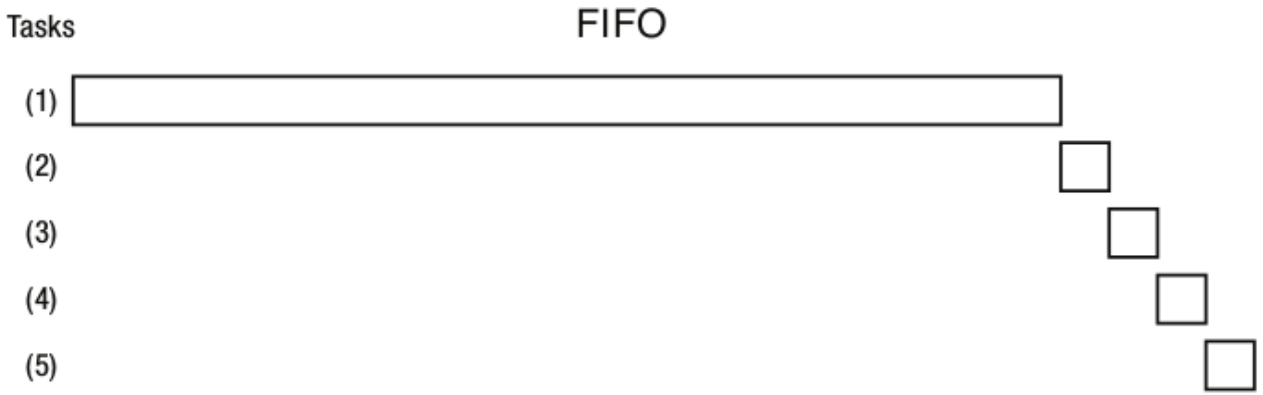
- Minimizes overhead - switch only when task complete
- best throughput when fixed number of cpu-bound tasks
- fair: every tasks waits its turn

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...

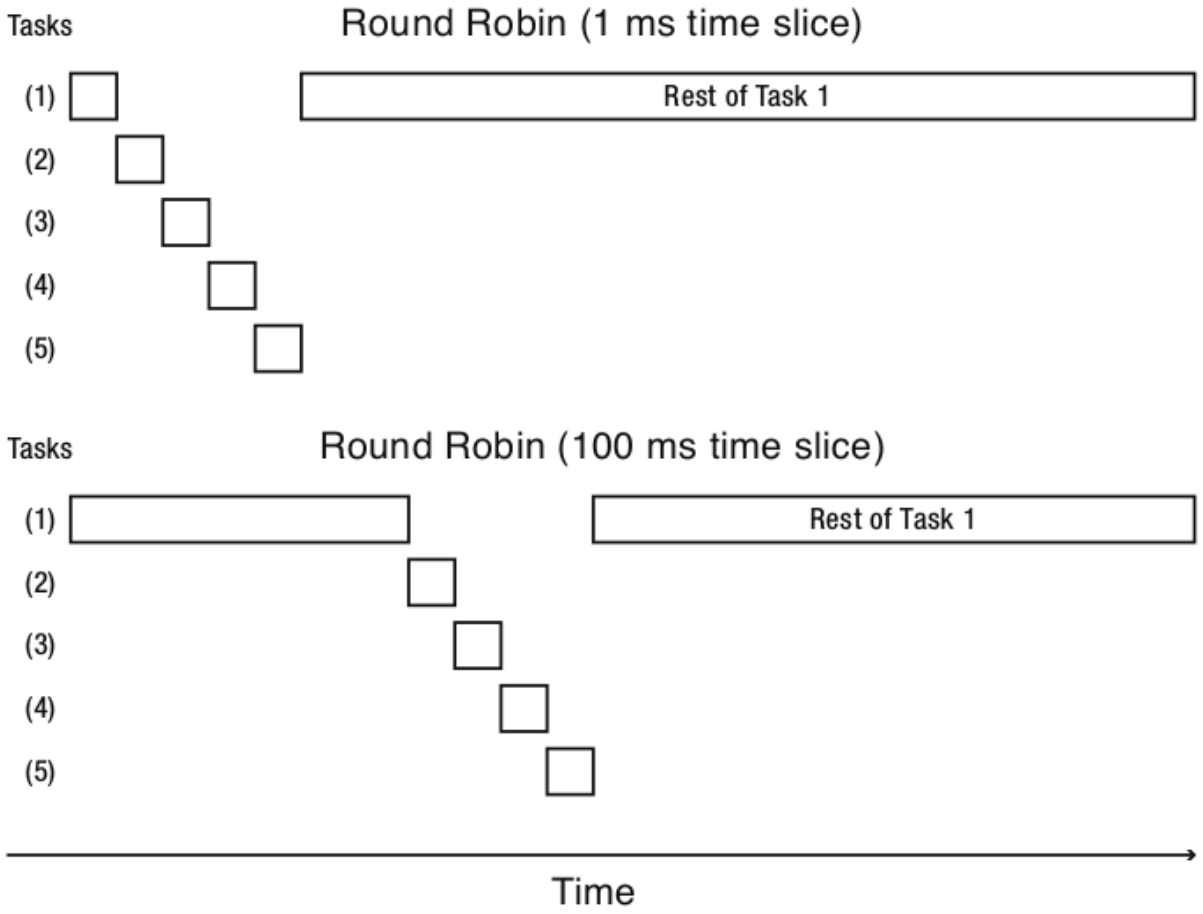
Round Robin

- 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

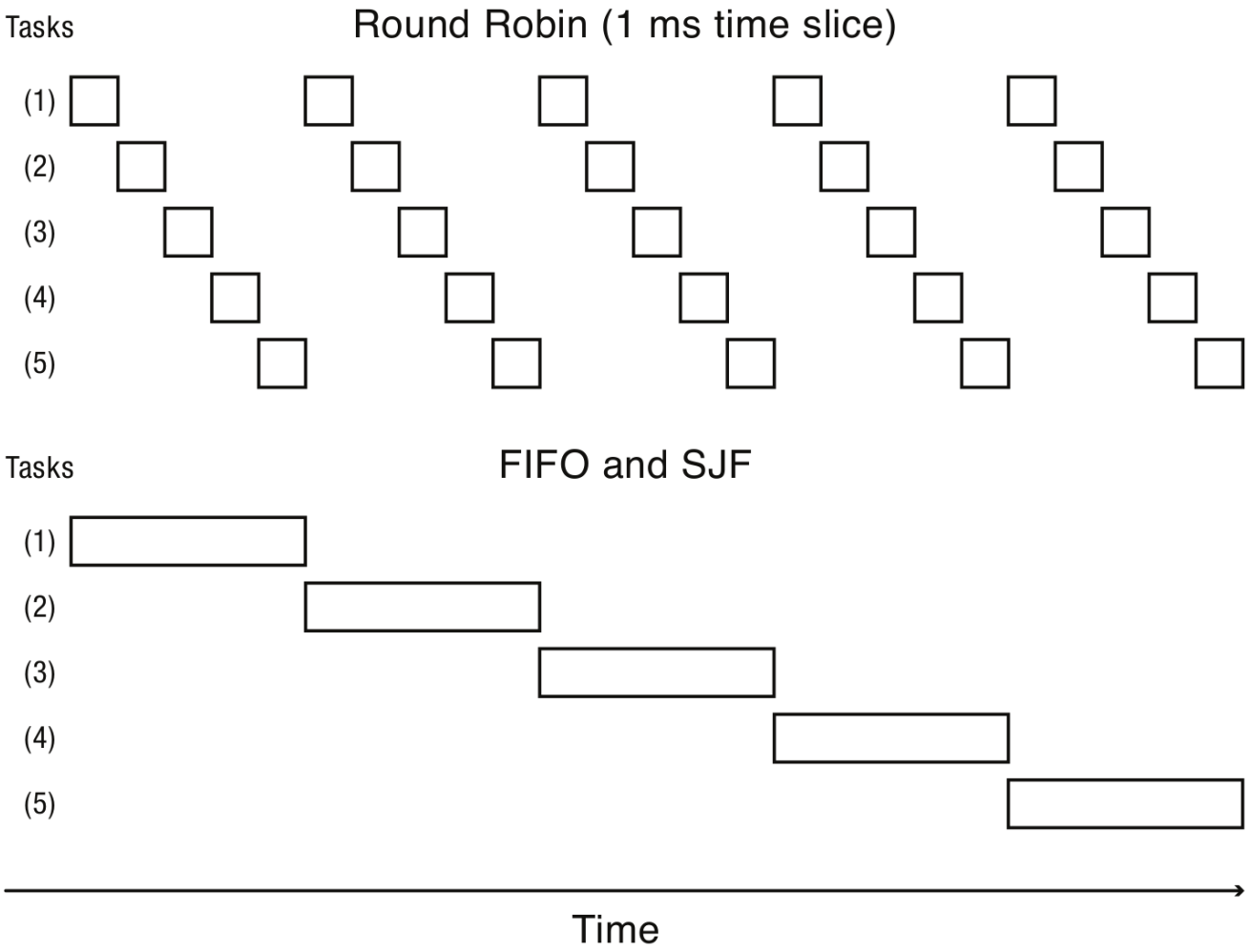
Round Robin

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

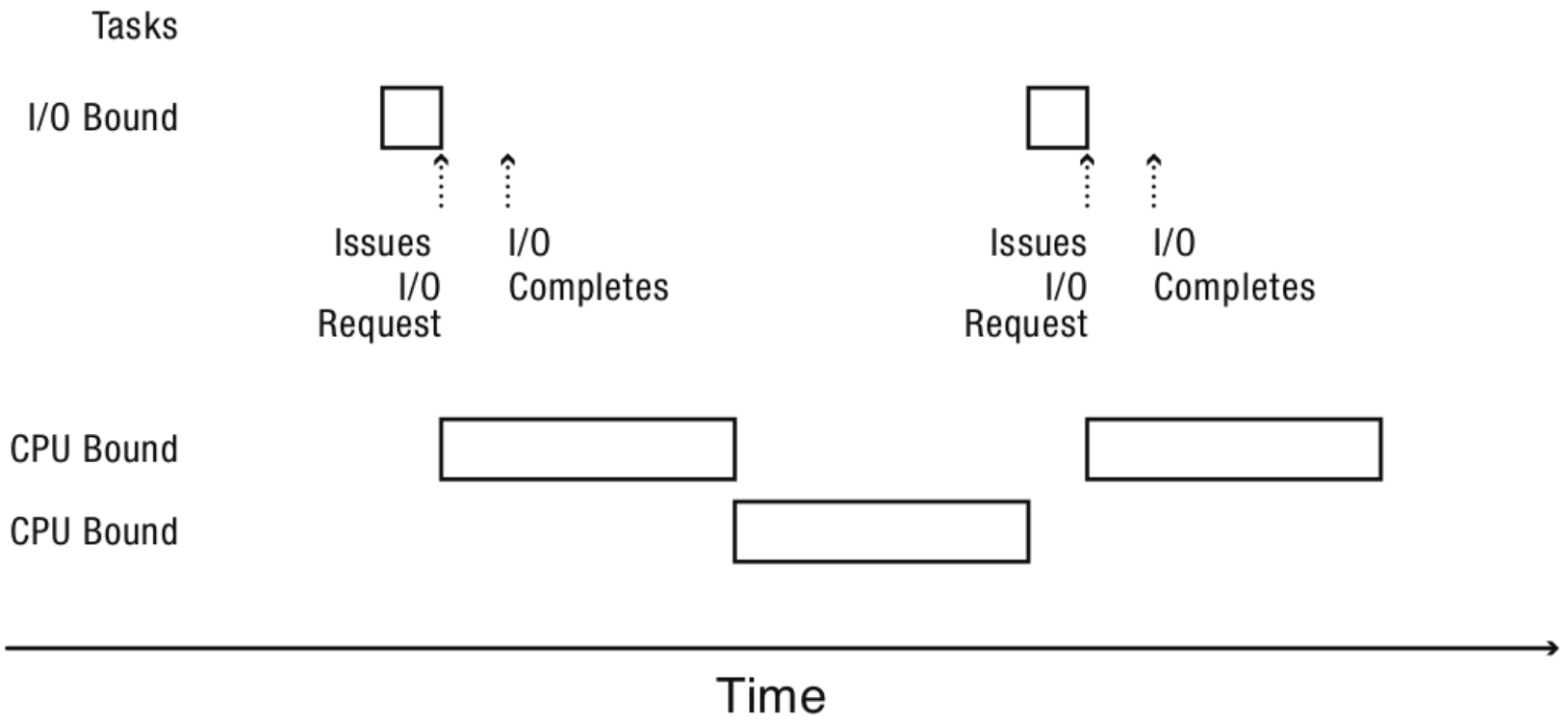
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Time

Round-Robin - equal length tasks



RR - I/O and compute tasks



Multi-Level Feedback

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

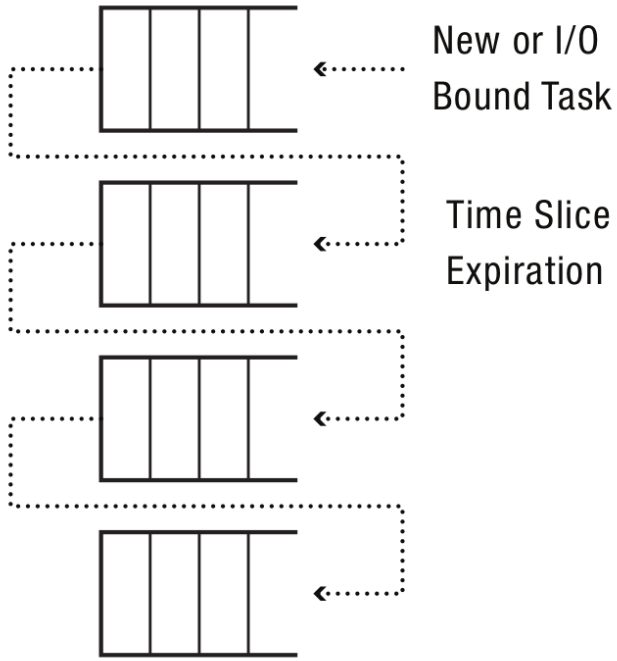
MFQ

- 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

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



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?

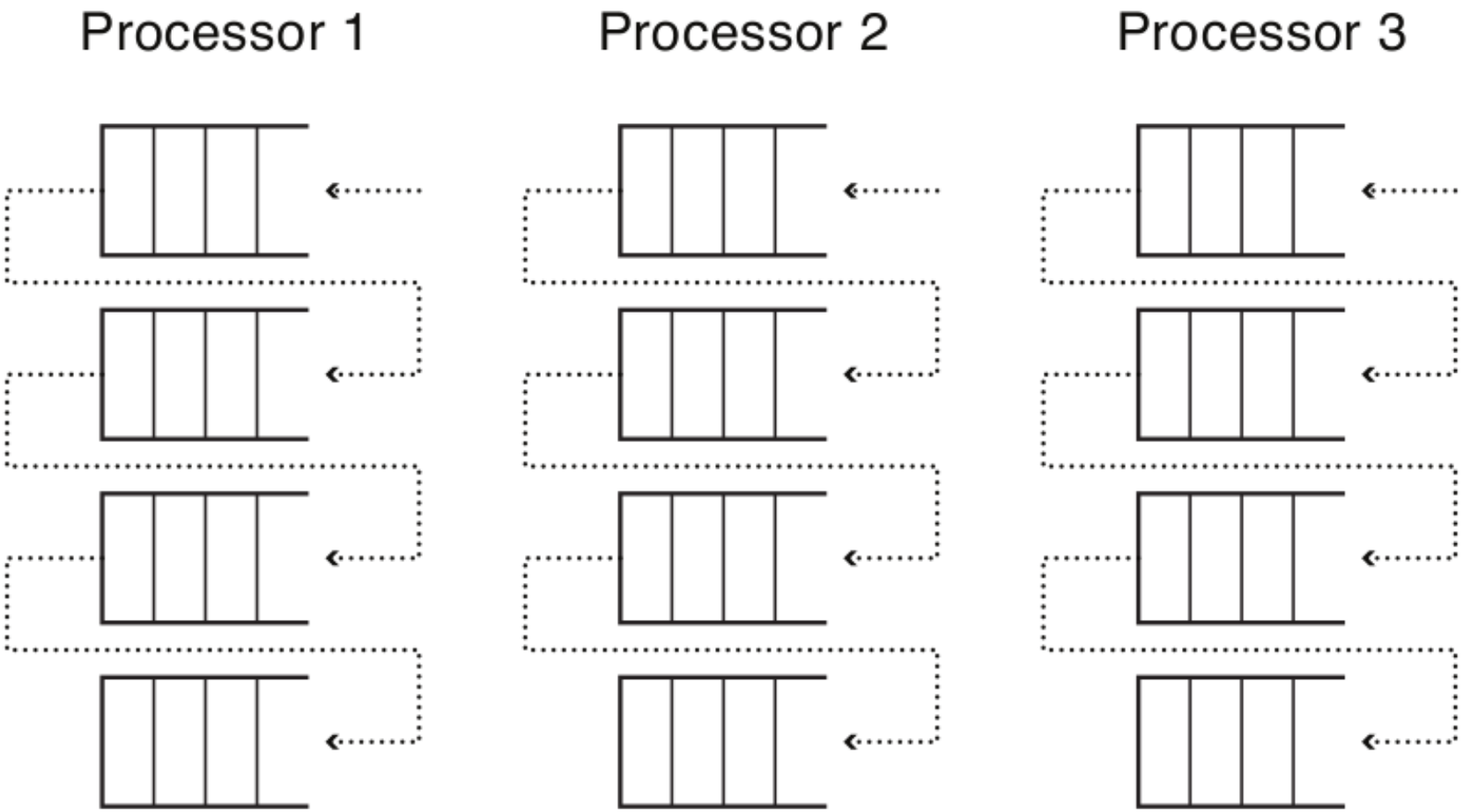
Multi-Processor Scheduling

- What would happen if we used MFQ on a multiprocessor?
  - Contention for centralized MFQ lock
  - Cache 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

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
- Idle processors can steal work from other processors

Per-Processor Multi-level Feedback



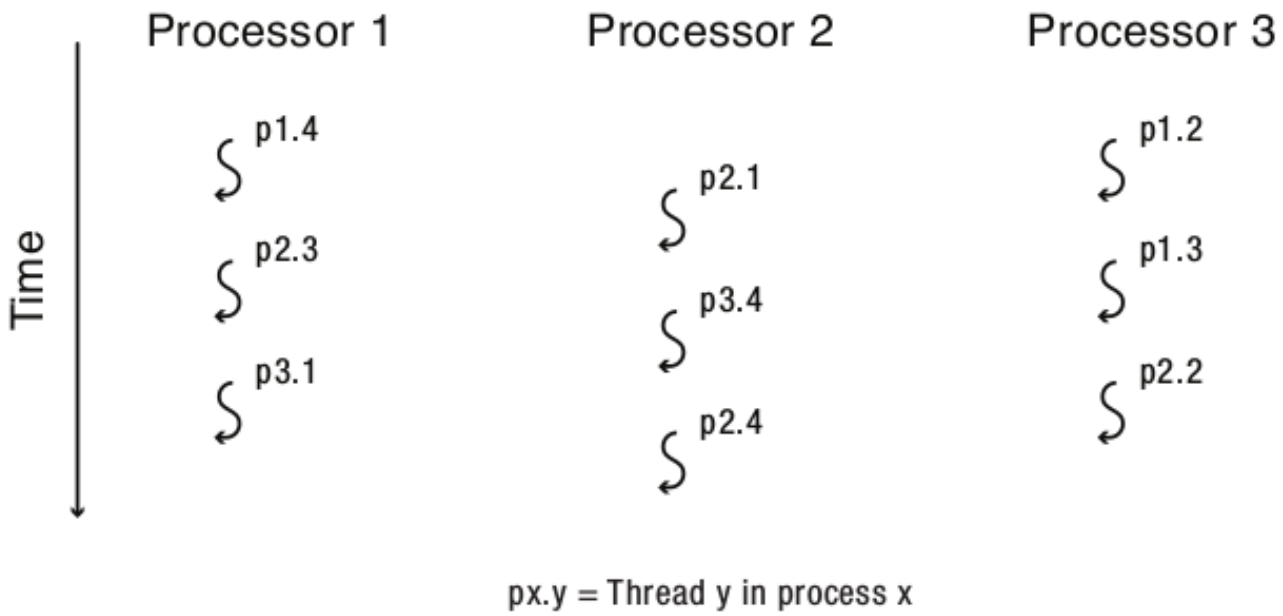
Scheduling Parallel Programs

- 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

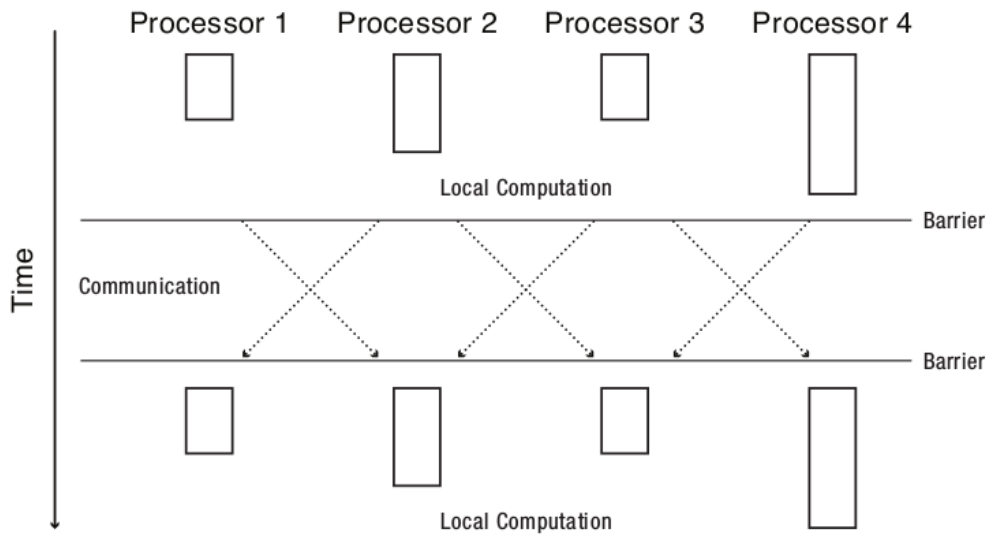
Scheduling Parallel Programs (2)

- 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



Bulk Synchronous Delay

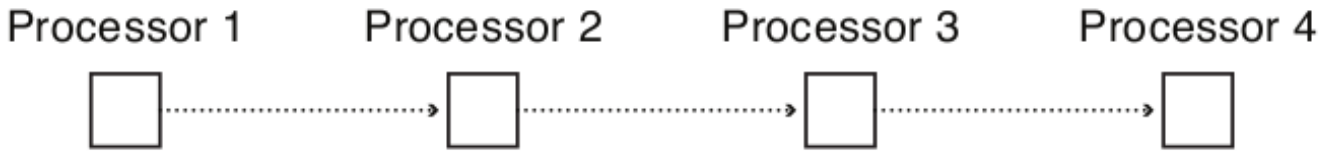


Computation limited by slowest processor involved

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

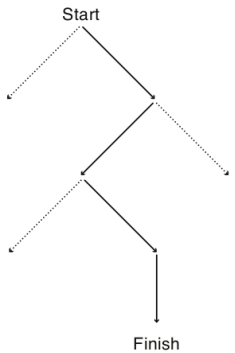
Producer-Consumer-Delay



- preempting one thread stalls all others

Other issues

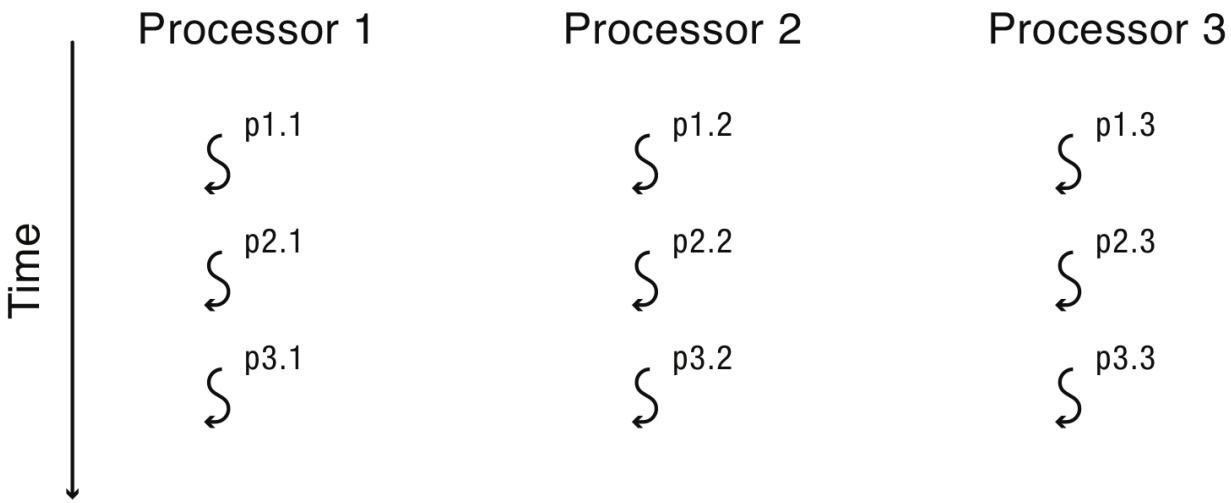
- Critical Path Delay
  - Preempting a thread on a critical path will slow down end result



- Preemption of lock holder

Gang Scheduling

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

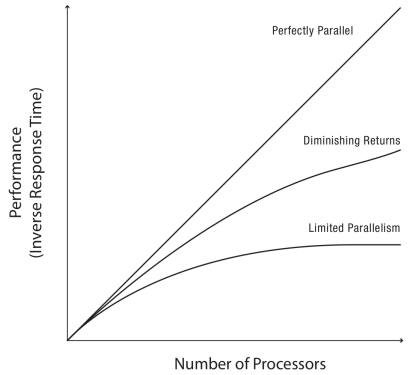


Gang Scheduling

- 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

- 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