Operating Systems

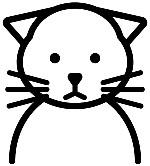
Scheduling and Deadlocks

Daniel Gruss

2023-10-08

Scheduling

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
07.00							
08.00							
09.00							
10.00							
11.00							
12.00							
13.00							
14.00							
15.00							
16.00							

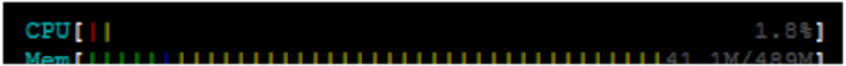


- No “right” answer
- always a trade-off

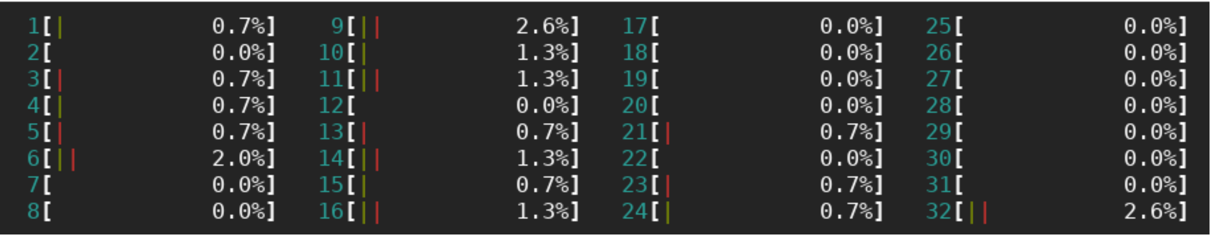
Some Scheduling Ideas Already

- Oldest homework first: **First In First Out (FIFO)**
- Homework with earliest deadline next: **Earliest Deadline First (EDF)**
- 1 hour of this, then 1 hour of that, ... until everything is done: **Round Robin (RR)**
- Short homework first: **Shortest Job First (SJF)**

... which are used in the real world!


CPU[1.8%]

VS


1[0.7%] 9[2.6%] 17[0.0%] 25[0.0%]
2[0.0%] 10[1.3%] 18[0.0%] 26[0.0%]
3[0.7%] 11[1.3%] 19[0.0%] 27[0.0%]
4[0.7%] 12[0.0%] 20[0.0%] 28[0.0%]
5[0.7%] 13[0.7%] 21[0.7%] 29[0.0%]
6[2.0%] 14[1.3%] 22[0.0%] 30[0.0%]
7[0.0%] 15[0.7%] 23[0.7%] 31[0.0%]
8[0.0%] 16[1.3%] 24[0.7%] 32[2.6%]

Scheduling

Similar design challenges as with PRAs:

- latency
- throughput
- fairness


HAVING A TIRE BLOWOUT A WEEK AFTER HITTING A SIDECURB
PRINCIPLE OF LOCALITY

Common Terms around Scheduling

- **Task**: anything that consumes time
- **Latency**: time until a task is resolved
- **Predictability** of runtime
- **Throughput**: how much do we get done over time?
- **Scheduling Overhead**: time to switch from one task to another
- **Fairness**: above properties wrt. different tasks
- **Starvation**: task doesn't make any progress due to other tasks

Scheduling algorithm

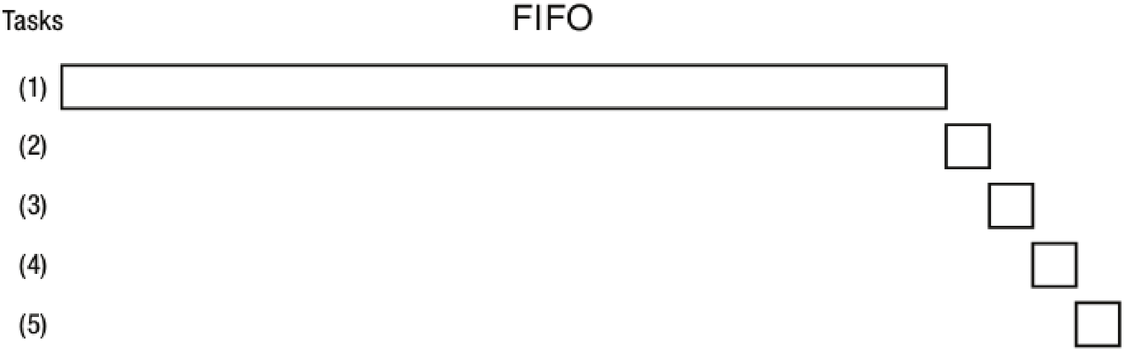
- takes a workload as input
- decides which tasks to do first
- Performance metric (throughput, latency) as output
- Only **preemptive**, work-conserving schedulers to be considered

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 CPU
  - **I/O-bound**: most of the time wait for I/O-bound
  - **mixed**

First-In-First-Out (FIFO)

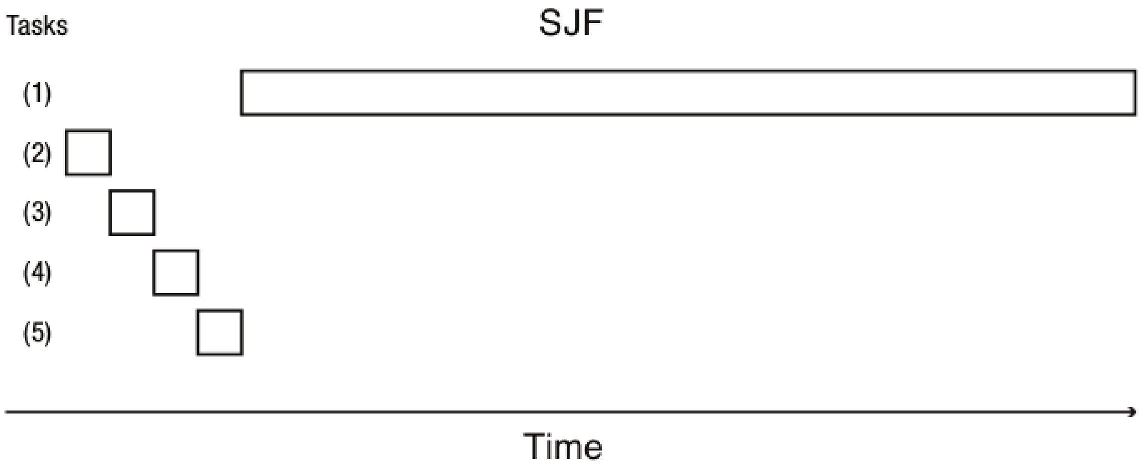
- aka first-come-first-serve
- Run tasks in order of arrival until they complete or yield


Tasks
FIFO
(1)
(2)
(3)
(4)
(5)

SJF - Shortest Job First

- FIFO optimized for throughput - other extreme: optimize for latency

$\rightarrow$ schedule the shortest job first (SJF)


SJF
Tasks
(1)
(2)
(3)
(4)
(5)
Time

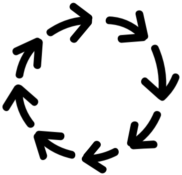
Example - Supermarket

- No more Express-Kassen!
- Skip ahead in the waiting line until everybody in front of you has the same or fewer items

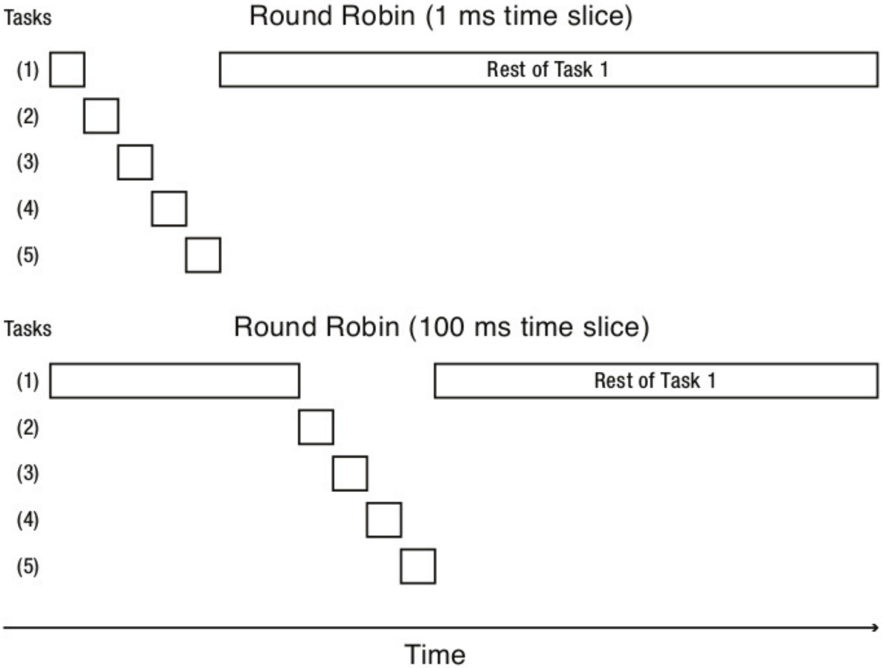
→ current customer interrupted

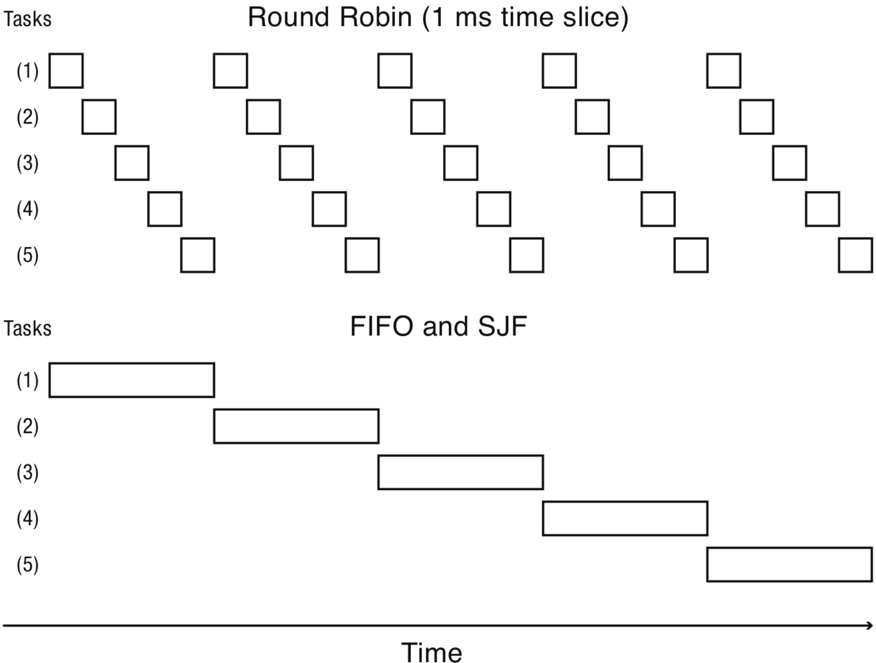
→ full basket - you have to wait...

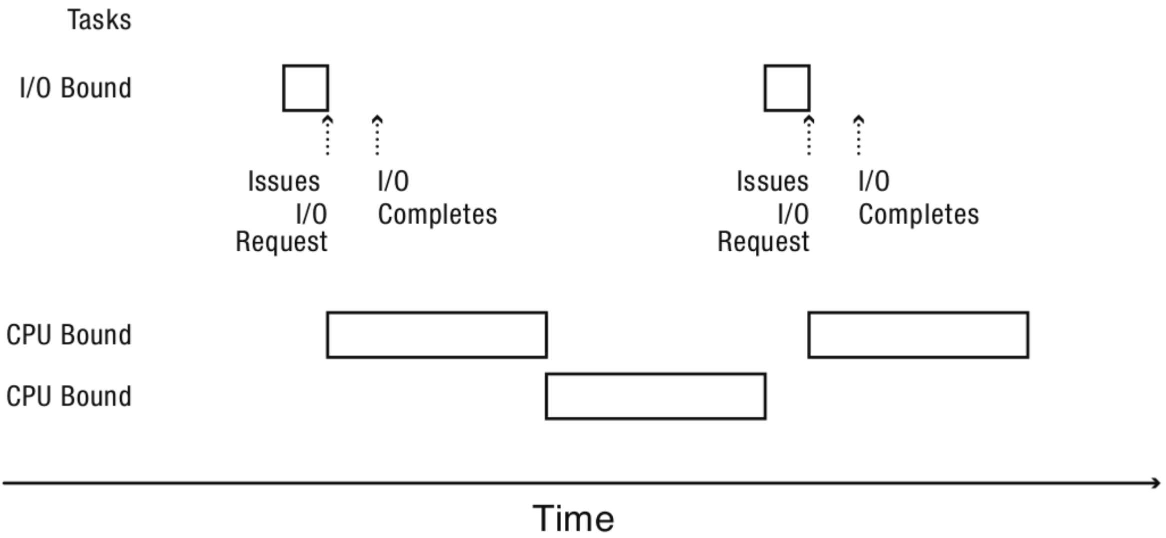
Round Robin



- fighting starvation: schedule tasks in a round robin fashion
- compromise between FIFO and SJF
- each task: fixed period of time (time quantum)
- not finished? → back in line


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

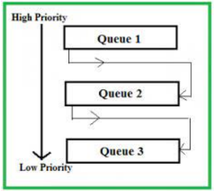

Round Robin (1 ms time slice)
Tasks
(1)
(2)
(3)
(4)
(5)
FIFO and SJF
Tasks
(1)
(2)
(3)
(4)
(5)
Time


Tasks
I/O Bound
Issues I/O Request
I/O Completes
Issues I/O Request
I/O Completes
CPU Bound
CPU Bound
Time

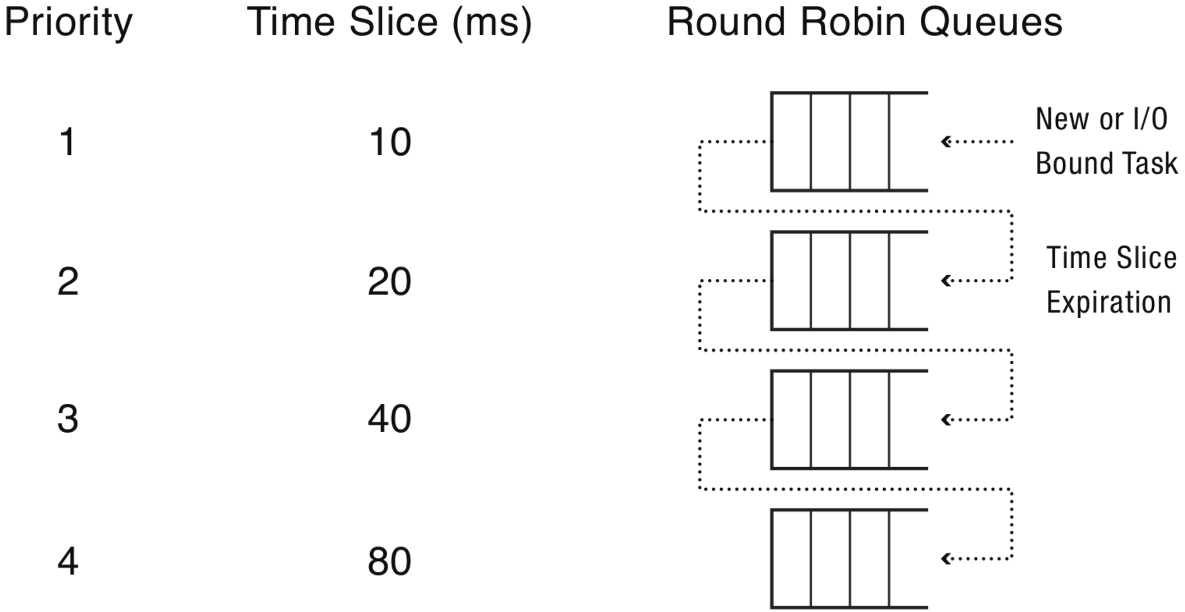
Multi-level Feedback Queue (MFQ)

- Goals:
  - Latency
  - 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



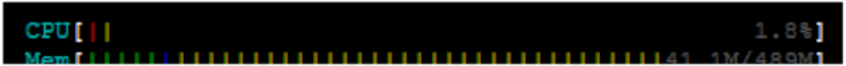
Priority
Time Slice (ms)
Round Robin Queues
1
10
New or I/O
Bound Task
2
20
Time Slice
Expiration
3
40
4
80

Single Core Summary

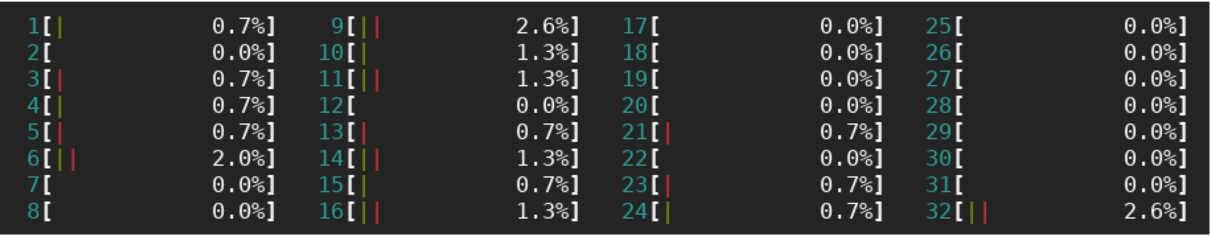
- FIFO: simple + high throughput. but variable size tasks $\rightarrow$ bad latency
- SJF: often impossible(?) + latency variance
- RR: variable size tasks $\rightarrow \approx$ SJF.
- RR: equal size tasks $\rightarrow$ bad

$\rightarrow$ CPU and I/O mixed $\rightarrow$ SJF $>$ RR

- MFQ balances latency, overhead and fairness


CPU[|| 1.8%]

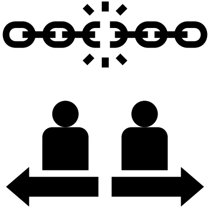
VS


1[| 0.7%]
2[0.0%]
3[| 0.7%]
4[| 0.7%]
5[| 0.7%]
6[|| 2.0%]
7[0.0%]
8[0.0%]
9[|| 2.6%]
10[| 1.3%]
11[|| 1.3%]
12[0.0%]
13[| 0.7%]
14[|| 1.3%]
15[| 0.7%]
16[|| 1.3%]
17[0.0%]
18[0.0%]
19[0.0%]
20[0.0%]
21[| 0.7%]
22[0.0%]
23[| 0.7%]
24[| 0.7%]
25[0.0%]
26[0.0%]
27[0.0%]
28[0.0%]
29[0.0%]
30[0.0%]
31[0.0%]
32[|| 2.6%]

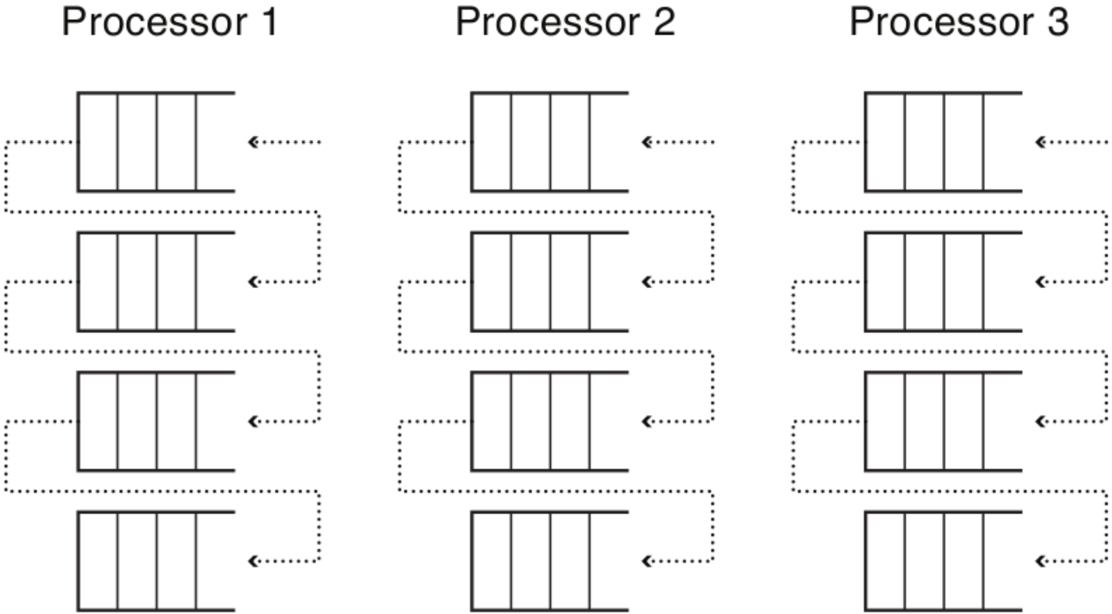
Multi-Core Scheduling

- What would happen if we used MFQ on a multi-core CPU?
  - Lock for global MFQ lists $\rightarrow$ bad performance
  - Cache slowdown (write access on one core $\rightarrow$ slower read on other core)
  - Limited cache reuse: thread's data from last time could still be in the cache (of another core!)

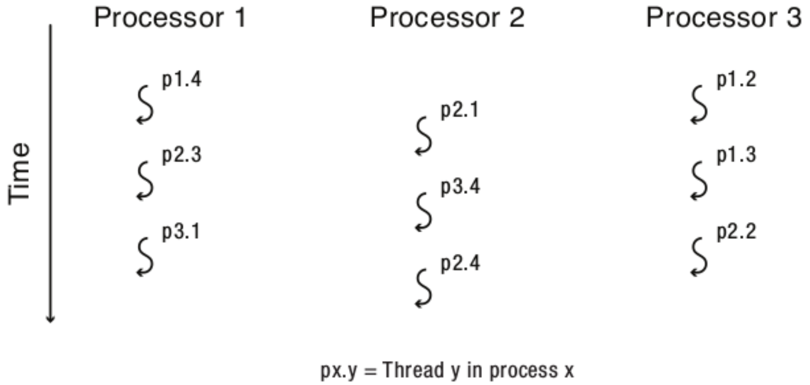
Core Affinity



- Each core has its own thread list
- Protected by a per-core lock
- Idle cores can "steal" threads from other cores

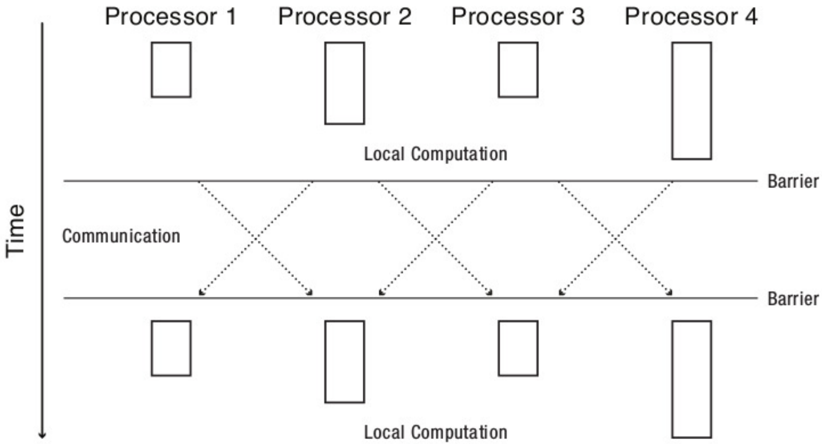

Processor 1
Processor 2
Processor 3

Scheduling Multi-threaded Programs



"Just schedule threads" – yes, but ...

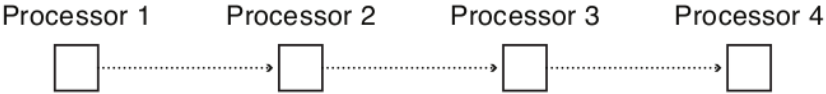
Bulk Synchronous Parallelism



Bulk Synchronous Parallelism

- Loop on each core:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other cores (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

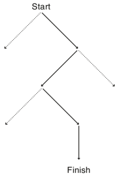
Producer-Consumer-Delay



- preempting one thread stalls all others

Other issues

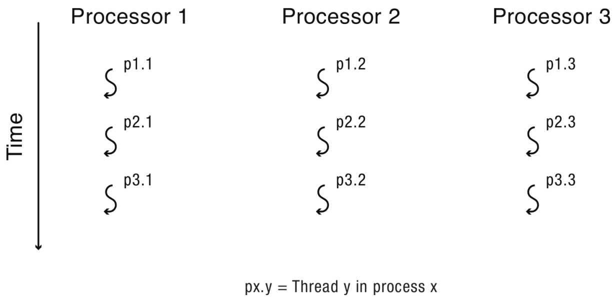
- Critical Path Delay
  - Preempting a thread on a critical path will slow down end result



- Preemption of lock holder

Gang Scheduling

- Application splits work into threads
- threads always run together (if possible)

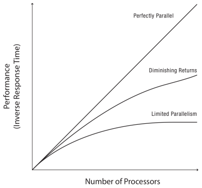


Gang Scheduling

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

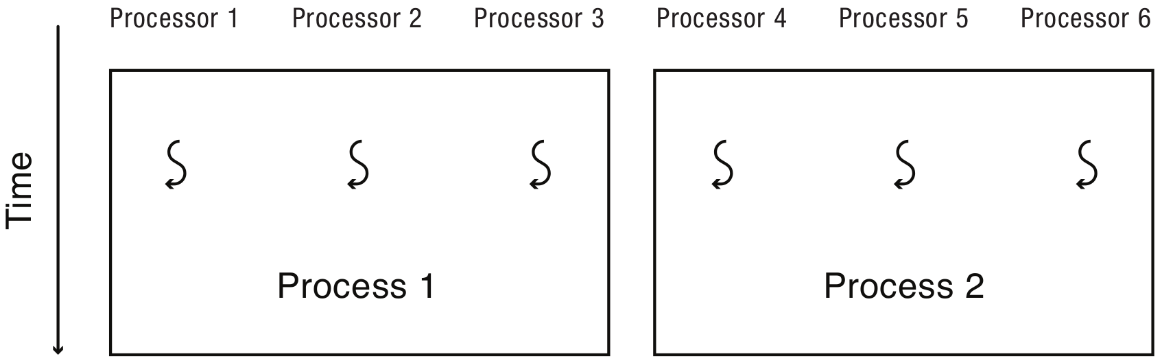
Gang Scheduling

- Some make efficient use of many cores
- some have diminishing return



Space Sharing

- give two parallel programs each half of the cores $\rightarrow$ **space sharing**
- minimizes context switches for each core
- what we discussed before was: **time sharing, time slicing** (single core to multiple tasks)


Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6
Time
Process 1
Process 2

Deadlocks

Classic Deadlock Example

```
wait(Resource_1);
wait(Resource_2);
use_Resource();
signal(Resource_2);
signal(Resource_1);
```

```
wait(Resource_2);
wait(Resource_1);
use_Resource();
signal(Resource_2);
signal(Resource_1);
```

Formal definition

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Assumptions: processes, threads - both may be deadlocked. Number of threads, types of resources relevant.

Conditions for deadlocks (1)

Mutual Exclusion condition

Each resource is either currently assigned to exactly one process or is available.

Conditions for deadlocks (2)

Hold-and-wait condition

Processes currently holding resources that were granted earlier can request new resources

Conditions for deadlocks (3)

No-preemption condition

Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them

Conditions for deadlocks (4)

Circular wait condition

There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain

Conditions for deadlocks

All four conditions must be present for a deadlock to occur

- Mutual Exclusion condition
- Hold-and-wait condition
- No-preemption condition
- Circular wait condition

Strategies to deal with deadlocks

- Ignore it (maybe it ignores us too...)
- Detection and Recovery
- Avoidance
- Prevention

Ignoring - the “Ostrich Algorithm”

Mathematical Approach

We MUST prevent deadlocks!

Engineering Approach

- How often does the problem occur?
- How expensive is it to solve?
- Let’s do a cost-benefit analysis!

Ignoring as Best Practice

- Unix, Windows: the problem is ignored
- Cost to prevent deadlocks too high
- Prevention may not be possible at all
- Even detection is too expensive
- Weigh “comfort” versus “correctness”

Example

- Resources in OS are limited
- limited number of processes or open files at any time
- assume: all active process need to do another fork or open one more file
- None are available $\rightarrow$ deadlock!
- Now how likely is that?

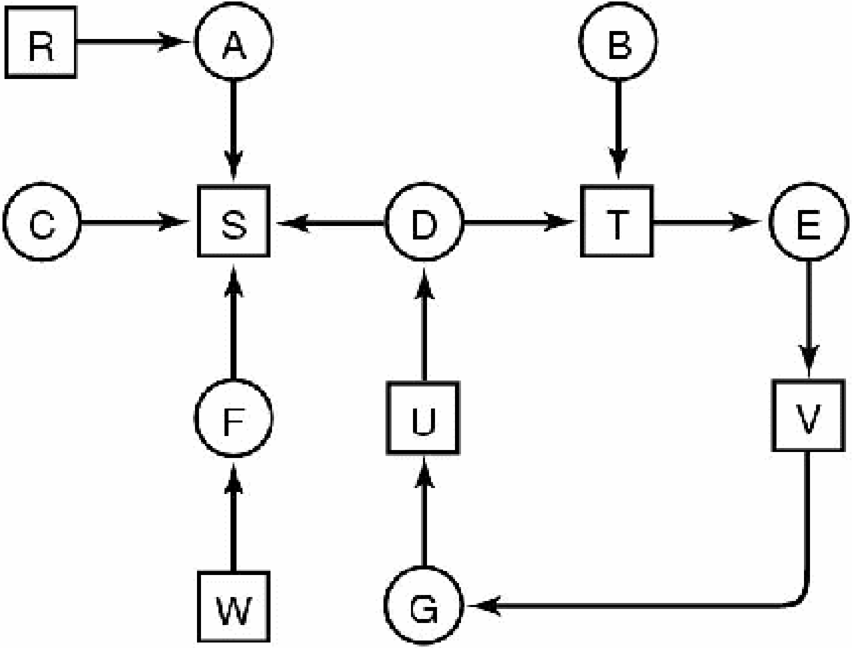
Detection and Recovery

- Don't prevent occurrence
- try to detect occurrence and deal with it when it happens
- how can we do that?
- e.g.: "draw" resource-graphs and detect circles

Detection

- Example: is the following system deadlocked?

```
Process A holds R and wants S
Process B holds nothing but wants T
Process C holds nothing but wants S
Process D holds U but wants S and T
Process E holds T but wants V
Process F holds W but wants S
Process G holds V but wants U
```


R
A
B
C
S
D
T
E
F
U
V
W
G

Detection

- easy - visually
- but there is an algorithm too
- many algorithms for detecting cycles in directed graphs

Simple Algorithm (useful for `pthread_join`)

Depth-first search in a tree

- take each node as the root of a tree
- do a depth-first search
- if we ever come back to a node we have already been to: cycle found
- when we have visited all arcs from a node: backtrack one level up
- back to start: no deadlock found
- need to try for all nodes as roots

not quite optimal

Detection

- When do we check for deadlock?
  - each request? (earliest detection, expensive)
  - every x minutes?
  - nothing else to do (or low CPU workload)?
- And what do we do?? Preemption, roll back, kill processes?

Recovery/Preemption

- Take resource away from process
- may be possible with some resources
- side-effects?
- difficult to impossible
- manual intervention may be required

Rollback

- Assume deadlocks are likely
- set checkpoints all the time (memory, registers, everything...)
- When deadlock occurs, select process and set it back checkpoint before deadlocked resource was assigned

Kill Processes

- Simple and effective
- just kill a process involved in deadlock
- if this resolves deadlock: fine
- if not: kill one more
- best to kill one which can easily start again (like a compiler)
- killing processes that e.g. changed databases not a great idea

Avoidance

- Processes ask for resources on at a time
- avoidance would check if resource can be assigned safely and assign resource only when safe
- is there an algorithm that can do this?
- Yes, if certain information is available in advance

Prevention

- Avoidance rarely practical
- Recovery after detection difficult
- What can we do?
- Prevent - by excluding one of the requirements

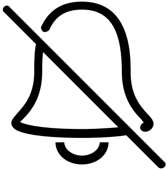
Mutual Exclusion

- no mutual exclusion - no deadlock
- since mutual exclusion is a requirement, this is practically impossible
    - avoid using (and thus locking) a resource unless absolutely necessary
    - try to make sure that as few processes as possible may actually claim the resource

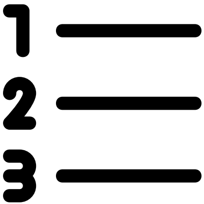
Hold and Wait

- prevent processes that hold resources form waiting for more resources
- e.g. require all processes to request all resources before starting execution
  - processes don't necessarily know that
  - very defensive tactics - and very very bad for effective resource utilization
- Alternative:
  - release all resources first whenever acquiring a new one
  - then try to get all of them again

No preemption



- Very difficult. Rarely possible.

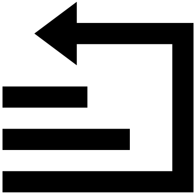


- Easy way: allow only one resource to be held. Not very practical though.
- Better way: **Provide global numbering of all resources.**
- Processes can request resources, but only in numerical order.
- No cycle can exist.
- Problem: It can be difficult to find a working numbering scheme. What to do if resources are dynamic?

Two-Phase-Locking

- Avoidance and Prevention not very promising in the general case
- for specific applications excellent algorithms are known
- one example: database systems
  - frequently need locks on several records
  - then update all of them
- multiple processes: real danger of a deadlock

Two-Phase-Locking



- Phase 1: try to get locks for all records
- successful:
  - Phase 2: update records and release locks
- unsuccessful:
  - release locks and start again with Phase 1

Starvation

- Closely related to deadlocks
- policies decide who gets which resource when
- may lead to the situation that some process never gets service even if they are not deadlocked
- can e.g. be avoided by a first-come-first-served basis