Scheduling

COMPSCI 2SD3

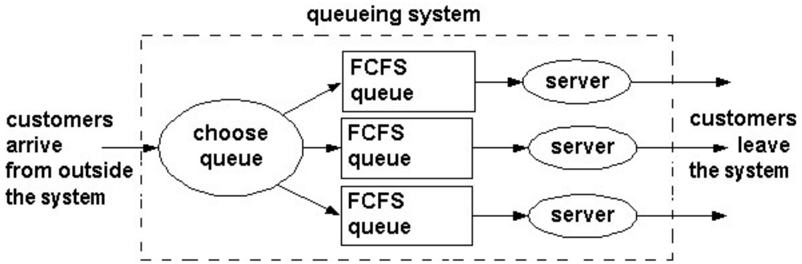
Concurrent Systems

Term II, 2021/22

Outline

Scheduling

FCFS: First-Come First-Served scheduling
(aka FIFO, supermarket)



SJF: Shortest-Job-First scheduling

Though FCFS is fair, it may not optimize the average waiting time:

- job1 (10 minutes), job2 (1 min), job3 (1 min): job1 waits 0 minutes, job2 waits 10 minutes, job3 waits 11 minutes, average waiting time is (0+10+11)/3 = 21/3 = 7 minutes.
- If we do job2, job3, job1: job2 waits 0 minutes, job3 waits 1 minute, job1 waits 2 minutes, average waiting time is (0+1+2)/3 = 3/3 = 1 minute.

Highest-Response-Ratio-Next scheduling

A modification as not to discriminate too much against long jobs in SJF scheduling:

$$\text{Response Ratio} = \frac{\text{Wait Time} + \text{Job Time}}{\text{Job Time}}$$

The job with the highest *ResponseRatio* is chosen to go next. This scheduling system is a compromise between FCFS and SJF systems.

Priority scheduling

The job with the highest priority is scheduled to go next.

The problems:

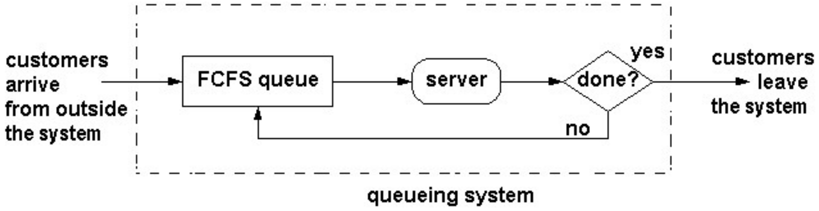
- assigning priorities
- conflict resolution (two jobs with the same priority?)

Deadline scheduling

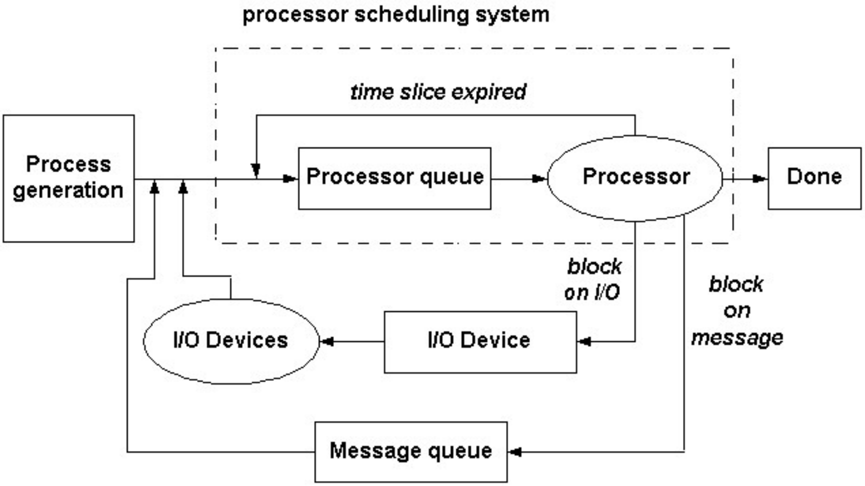
A job is scheduled first if otherwise it would not meet a deadline.

RR: Round-Robin scheduling

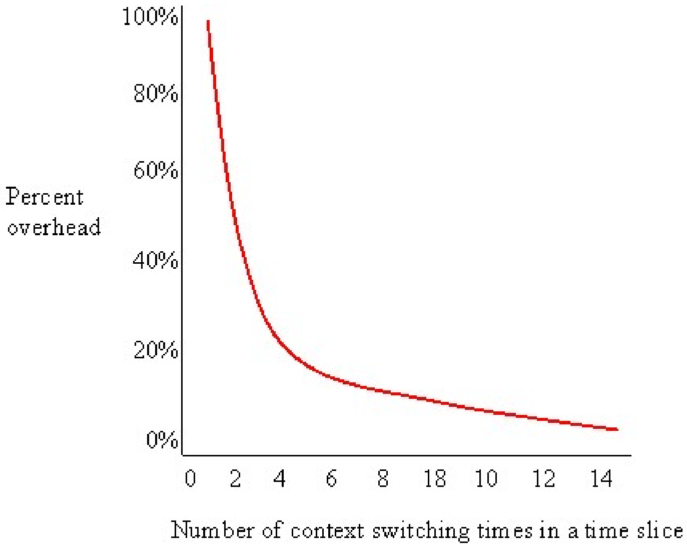
A ***preemptable*** process is such that it can be stopped and then after some time restarted. Preemtable processes can be scheduled in Round-Robin when every one is scheduled for a while in a circular fashion. It is a fair method of sharing something that can be preempted.


customers arrive from outside the system
FCFS queue
server
done?
yes
no
customers leave the system
queueing system

There are several ways to a preemptive scheduling. For the following discussion we are assuming that processes enter the scheduling system and remain there (sometimes executing and sometimes waiting to execute) until they have finished execution or in *blocking wait*.


processor scheduling system
time slice expired
Process generation
Processor queue
Processor
Done
block on I/O
block on message
I/O Devices
I/O Device
Message queue

- Thus a process will enter and leave the processor scheduling system many times during its entire execution, maybe hundred of thousands of times (typically it runs on the processor for about 50 milliseconds = $50x10^{-3}$ seconds).
- Still, each one is a separate transaction as far as the processor scheduling system is concerned.
- Round robin scheduling is “easy", the hard part is to determine the length of the ***time slice*** (also called ***quantum***).



100%
80%
60%
Percent overhead
40%
20%
0%
0
2
4
6
8
18
10
12
14
Number of context switching times in a time slice

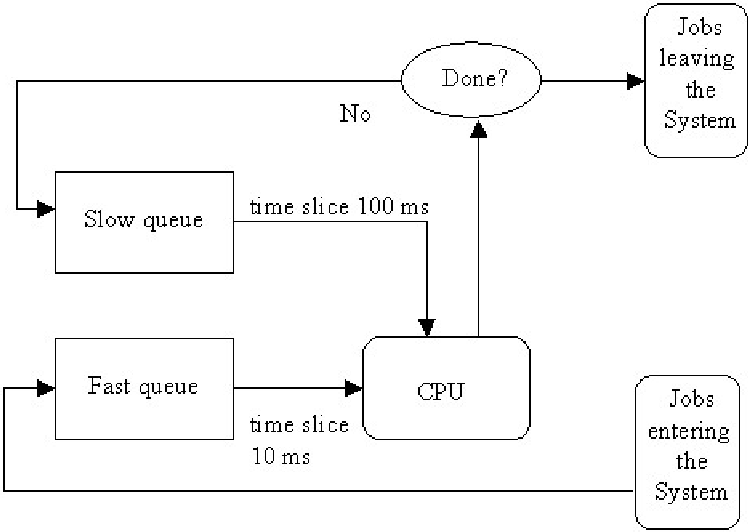
The duration of context switching (preemption) is x. Let us express the quantum as a multiple of x, e.g. $q = Nx$

If $N = 1$, then the overhead for context switching is 100%. The previous graph shows how for an increasing N, the overhead decreases (simply put, the bigger the quantum, the smaller the overhead, but the longer the interactive response time). So, q should be at least $5x$, after that the decrease in overhead is not significant, while the increase in the response time becomes significant.

What to do when a round-robin system is

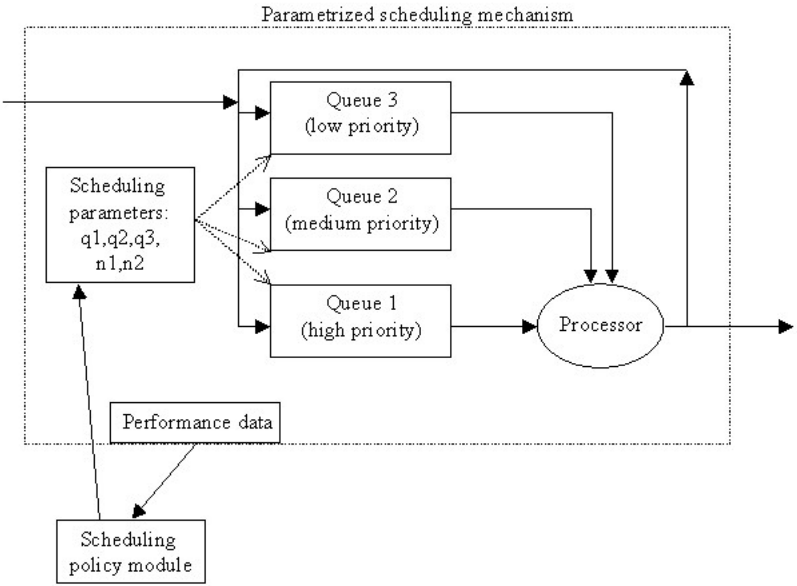
heavily loaded, i.e. there are so many jobs that the response time (even with the minimum quantum) is unacceptably high?

RR systems don't ***degrade gracefully*** with respect to load. In order to maintain good response time, we may need two queues.


Jobs leaving the System
Done?
No
Slow queue
time slice 100 ms
Fast queue
CPU
time slice 10 ms
Jobs entering the System

Two-queue scheduling divides processes to 2 classes, improves the response time for very short jobs (finishing within one time slice).

What to do to improve response time for medium jobs?
We may solve this by a multiple-queue scheduling, separating the mechanism of scheduling from the policy of scheduling:


Parametrized scheduling mechanism
Queue 3
(low priority)
Queue 2
(medium priority)
Queue 1
(high priority)
Scheduling
parameters:
q1,q2,q3,
n1,n2
Processor
Performance data
Scheduling
policy module

qi - time slice for the queue *Qi*
ni - number of times a process can go through the queue *Qi*

- At any time we chose a job from *Q*1,
- if empty, we chose a job from *Q*2,
- if empty we chose a job from *Q*3.
- The job from *Qi* is given *qi* time slice.

When it completes, it may go to the end of the queue it came from, unless it exceeds its *ni*, in which case it must go to the queue with the next lower priority.

This scheduler works as

- FCFS, if $n1 = q1 = \infty$
- RR, if $q1 < \infty$
- 2-queue system, if $n2 = \infty, n1 = 1, q1 \leq q2 < \infty$

This is an example of policy/mechanism split: the dispatcher is in the kernel and fixed (mechanism), while the policy (how to set the parameters) module can run once a while, examine the system's performance, and change scheduling policy (the parameters) accordingly.

There is a remarkable uniformity in the scheduling algorithms in various OS:

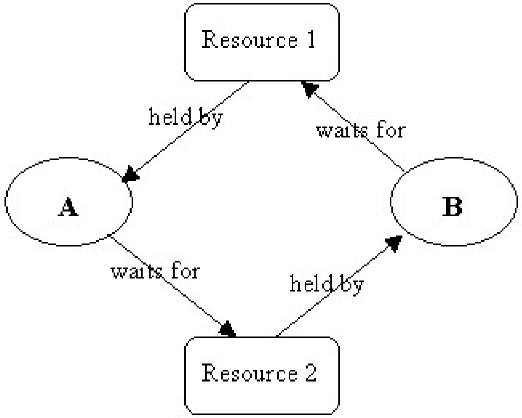
UNIX (and Linux) scheduling:
preemptive priority scheduler with 160 priority levels. Processes at levels in 0-59 are in the time-sharing class. Processes at levels 60-99 are for system priorities (kernel-mode processes run in this class), processes at levels 100-159 are in the real-time class. The process with the highest priority is always running. Processes with the same priority are scheduled round-robin.

SOLARIS[1] scheduling:
SOLARIS had additional levels 160-169 that are for interrupt handling. Time slices range from 4 to 20 milliseconds. SOLARIS used priority inheritance to avoid the problem of priority inversion, i.e. when a process waits for a lock, the process holding the lock will inherit its priority if it is higher than its own (otherwise the lower priority holder of the lock will never get a chance to run while the higher priority process trying to get the lock will never have a chance to get the lock).

[1]SOLARIS was a UNIX based operating system of Sun Microsystems. When Sun was bought in 2010 by ORACLE, it was renamed to ORACLE SOLARIS, and it was effectively discontinued in 2017.

Deadlocks

Two or more processes are waiting for a resource that another process in the group holds:



Processes can deadlock on consumables, software resources, as well as hardware resources:

Process A	Process B
receive(B,msg)	receive(A,msg)
send(B,msg)	send(A,msg)

will immediately deadlock. Such a deadlock may involve more than just two processes:

Process A	Process B	Process C	Process D

receive(D,msg)
send(B,msg)
	receive(A,msg)
	send(C,msg)
		receive(B,msg)
		send(D,msg)
			receive(C,msg)
			send(A,msg)

The following is a problem of “bad programming", but it could be an error in the system as well:

Process	Process B
while(1) {	while(1) {
send(B,msg)	send(A,msg)
receive(B,msg)	receive(A,msg)
}	}

If no message is lost, it works fine, but consider:

- B waits for a message from A
- A sends message to B
- The message from A to B is lost in transit
- A waits for a return message from B

The processes will deadlock.

Another example - airline reservation:

Agent A	Agent B
Lock(X)	Lock(Y)
see if space avail	see if space avail
Lock(Y)	Lock(X)
see if space avail	see if space avail
Unlock(Y)	Unlock(X)
Unlock(X)	Unlock(Y)

Again, we get a deadlock.

Conditions for deadlock to occur:

- resources are not preemptable
- resources are not shared
- a process can hold one resource and request another
- circular wait is possible

How to deal with deadlocks:

- ***Prevention*** - by placing restrictions on the resource requests so that deadlock cannot occur.
- ***Avoidance*** - plan ahead so that you never get into a situation where deadlock is inevitable.
- ***Recovery*** - detect when deadlock has occurred and recover from it

PREVENTION

- Allowing Preemption (it can always be done, but cost may be prohibitive)
- Avoiding Mutual Exclusion (*by virtualization - good for a hardware that you do not need in "real-time"*)
- Avoiding hold and wait (*get all the resources you need at one time, prevents deadlocks but can lead to inefficient utilization of resources*)

PREVENTION con’t

- Avoid circular wait (*assign each process a unique positive integer id and only acquire resources in ascending, numerical order of the id’s - good, often used, also can lead to inefficient use of resources*)

DEADLOCK RECOVERY

- First, you must ***detect*** a possible deadlock (essentially finding a cycle in a graph of resource requests, if in run time, this graph and its checking must be periodically updated, say every few seconds or when a resource requests blocks and then after a few seconds - e.g. VMS OS).
- Second, you must ***break*** the deadlock. This involves preempting a resource, which might mean canceling a process and starting it over.

How to approach resource requests in order to prevent deadlocks:

- Never grant resource request - (*not very practical*)
- Serialization - *one process at a time - a very slow solution*
- One-shot allocation - *each process must request all resources it will require in one time* (*practical in some cases, may lead to inefficient use of resources*)
- Hierarchical - *resources organized in a hierarchy and must be requested in a specific order* (*can also be inefficient*)

How to approach resource requests in order to prevent deadlocks con't:

- Advance claim - *each process must declare in advance how much of each resource it will need, the system uses deadlock avoidance algorithm* (*sure-proof, but not very practical and hence not often used*)
- Always allocate (*if you have the resource*) - *you detect and recover from deadlocks if and when they occur*

Two-phase locking

- Two-phase locking (lock - access - unlock) is often used to avoid deadlocks (in databases in particular).
- This can lead to ***starvation*** when a process never gets a “shot" at the resource (*for instance when a process waiting for a resource may be chosen at random*).

The solution may be FCFS queuing, but it only works if you are waiting for a single resource. So it usually is resolved by some kind of more complex queuing.

The problem is that a deadlock resolution can lead to starvation, while a starvation resolution can lead to a deadlock!