Computer systems architectures
CM12002

Russell Bradford

2023/24

### 1. Scheduling

#### Algorithms

**Highest Response Ratio Next**

A variant of SRT, where we take the time a process has been waiting since its last time slice into account

$Dynamic priority=\frac{time so far in
 system}{cpu used so far}$

* A process executes repeated time slices until its priority drops below that of another process
* Tries to avoid starvation
* Long jobs will eventually get a slice

### 2. Scheduling

#### Algorithms

**Highest Response Ratio Next**

* New jobs get immediate attention as CPU time is near 0
* But now critical shorter jobs might not finish in time as they could get scheduled after a long-waiting job
* This needs frequent re-evaluation of priorities to get good behaviour, which implies small timeslices, and so lots of scheduling overhead

### 3. Scheduling

#### Algorithms

**Multilevel Feedback Queueing**

Can be used when we have no estimates on run times

* There are multiple FIFO run queues, $RQ\_{0}$, $RQ\_{1}$, …$RQ\_{n}$. with $RQ\_{0}$ the highest priority, $RQ\_{n}$, the lowest
* Queues are processed in FIFO fashion, in priority order
* so $RQ\_{1}$ does not get a look-in until $RQ\_{0}$ has emptied
* and $RQ\_{2}$ does not get a look-in until $RQ\_{1}$ has emptied, and so on
* If a process appears in a higher queue, we go back to that higher queue

### 4. Scheduling

#### Algorithms

**Multilevel Feedback Queueing**

* Each process is allocated a *quantum* of time (a timeslice)
* A new process is admitted to the end (last) of $RQ\_{0}$
* When the running process has used its quantum of time, it is interrupted and placed at the end of the next lower queue: *demoted*

### 5. Scheduling

#### Algorithms

**Multilevel Feedback Queueing**

* If the running process relinquishes voluntarily before the end of the quantum, it gets placed back at the end of the *same* queue
* If it blocks for I/O, it will be *promoted* and placed at the end of the next higher queue (when ready to run)
* Demoted processes in $RQ\_{n}$ get placed back at the end of $RQ\_{n}$
* Similarly blocking processes in $RQ\_{0}$ get placed back at the end of $RQ\_{0}$

### 6. Scheduling

#### Algorithms

**Multilevel Feedback Queueing**



Multilevel Feedback Queueing

### 7. Scheduling

#### Algorithms

**Multilevel Feedback Queueing**

* This gives newer, shorter processes priority over older, longer ones
* I/O processes tend to rise, getting more priority
* Compute processes tend to sink, getting less

Old processes tend to starve with this, so a variant doubles the quantum for each level: $RQ\_{0}$ gets 1, $RQ\_{1}$ gets 2, $RQ\_{2}$ gets 4, and so on

So compute intensive processes get a big bite, whenever they get a chance, at the potential cost of responsiveness to a new process

### 8. Scheduling

#### Algorithms

Another advantage of MFQ is that it does not need to do any arithmetic: it just moves processes between queues

Remember, in early machines, arithmetic was a lot more time-consuming than it is now

This scheme was used by Windows NT and Unix derivatives, as we shall see next

### 9. Scheduling

#### Algorithms

**Traditional Unix scheduling**

As used in older Unix derivatives — modern scheduling is much more sophisticated

Everything is based on timer interrupts every $1/60^{
th}$ second

A priority is computed from the CPU use of each process

$$Priority=base priority+\frac{CPU time used}{2}$$

The 1/2 was a quirk of implementation and is not important

### 10. Scheduling

#### Algorithms

A process with the *smallest* priority value is chosen next. Thus — mostly — a process that has used less CPU

The base priority depends on whether this is a system process or a user process, with user priority being lower (i.e., with a larger value)

Processes of the same priority are treated round robin

Note that this is actually very similar in effect to Multilevel Feedback Queueing where a priority of $n$ corresponds to $RQ\_{n}$

### 11. Scheduling

#### Algorithms

The CPU use of a process is recorded and halved every second: this decays the influence of CPU usage over time and makes the priority based on *recent* behaviour

This algorithm gives more attention to processes that have used less CPU recently, e.g., interactive and I/O processes

$$Priority=base priority+\frac{decayed CPU time}{2}$$

### 12. Scheduling

#### Algorithms

Processes can choose to be *nice*

Generally, $−20\leq nice\leq 19$, but only certain users (administrators) can use negative nices

$$Priority=base priority+\frac{decayed CPU time}{2}+nice$$

A process that has nice $−20$ can really jam up the system

But nice also enables a *purchased* priority

### 13. Scheduling

#### Algorithms

There are a few problems with this technique

The priorities were recomputed once per second, all in a single pass, taking a significant chunk of time (on old machines)

It does not respond quickly enough to dynamic changes in the system

And does not scale to large numbers of processes

So this is not used in modern systems, where many 100s of processes is common

### 14. Scheduling

#### Algorithms

**Fair Share Scheduling**

And there are other problems that should be addressed

Modern machines can support many users simultaneously: what happens if user A has 9 processes and user B just 1?

Should A get 90% of the CPU time and B 10%?

*Fair share* scheduling is where each *user* (or group or other collective entity) gets a fair share, rather than each *process*

### 15. Scheduling

#### Algorithms

**Fair share Scheduling in Unix**

Recall Unix processes are collected in groups in a tree: a *process group*

$$\begin{matrix}Priority&=&base priority+\frac{CPU time used by
 process}{2}\\&&+\frac{CPU time used by process group}{2}+nice\end{matrix}$$

### 16. Scheduling

#### Algorithms

Modern Unix derivatives use much better, and much more complicated, scheduling algorithms than this

They can afford to be more complicated as CPUs are now much faster

Exercise. Read up on $O\left(1\right)$ scheduling and *The Completely Fair Scheduler*

Also have a look at scheduling for real-time systems: for when a process must *absolutely* get scheduled within a given time

### 17. Scheduling

Scheduling the CPU is clearly a difficult problem

It requires the collection and manipulation of many statistics about processes

Scheduling one resource (the CPU) is hard enough

We now look at a new problem that arises when we want to schedule *multiple* resources

### 18. Deadlock

Processes compete for resources like disks and network and the OS mediates this

To read from a disk, a process must call the OS kernel and wait for the kernel to reply

### 19. Terminology

When we say “a process waits for the kernel” we mean, of course, something entirely different

What actually happens is the kernel marks the process as blocked, and does not consider it for scheduling until the requested resource has arrived

There is no “waiting” happening: the process does not run when blocked

### 20. Deadlock

Processes compete for resources like disks and network and the OS mediates this

To read from a disk, a process must call the OS kernel and wait for the kernel to reply

Sometimes the wait is infinite!

### 21. Deadlock



Gridlock/Deadlock

### 22. Deadlock



Gridlock/Deadlock

### 23. Deadlock

This can happen in an OS

Process $P\_{1}$ wants to copy some data from disk $D\_{1}$ to disk $D\_{2}$, while process $P\_{2}$ wants to copy some data from disk $D\_{2}$ to disk $D\_{1}$

* Initially $P\_{1}$ is running and makes a request for access to $D\_{2}$
* The OS takes over and gives $P\_{1}$ exclusive access to $D\_{2}$
* The OS decides to run $P\_{2}$ (not a smart OS)
* $P\_{2}$ runs and makes a request for access to $D\_{1}$
* The OS takes over and gives $P\_{2}$ exclusive access to $D\_{1}$

### 24. Deadlock

* The OS decides to run $P\_{1}$
* $P\_{1}$ runs and makes a request for access to $D\_{1}$
* The OS takes over and notices $P\_{2}$ has locked $D\_{1}$, so $P\_{1}$ must wait until $P\_{2}$ has finished with it; $P\_{1}$ moves to state blocked
* The OS decides to run $P\_{2}$: it can’t run $P\_{1}$ as it is blocked
* $P\_{2}$ runs and makes a request for access to $D\_{2}$
* The OS takes over and notices $P\_{1}$ has locked $D\_{2}$, so $P\_{2}$ must wait until $P\_{1}$ has finished with it; $P\_{2}$ moves to state blocked
* Now both $P\_{1}$ and $P\_{2}$ are blocked and the OS can’t run either process!

### 25. Deadlock

$P\_{1}$ can’t run until $D\_{1}$ is free, but $D\_{1}$ won’t be free until $P\_{2}$ runs

$P\_{2}$ can’t run until $D\_{2}$ is free, but $D\_{2}$ won’t be free until $P\_{1}$ runs

This is called *deadlock*

Deadlock can happen on any kind of shared resources that require exclusive access

And with more than two processes: think of three or more processes in a circle

### 26. Deadlock

A formal definition of deadlock:

A set of processes $D$ is *deadlocked* if

1. each process $P\_{i}$ in $D$ is blocked on some event $e\_{i}$
2. event $e\_{i}$ can only be caused by some process in $D$

### 27. Deadlock

Note that you can only get deadlock if

* there is more than one resource
* there is more than one process[[1]](#footnote-74)[[2]](#footnote-75)
1. It could technically happen with just one process, but that would be quite dumb programming to request for a resource you already have [↑](#footnote-ref-74)
2. I’ve seen it happen [↑](#footnote-ref-75)