Highest Response Ratio Next

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

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

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

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

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

- A process executes repeated time slices until its priority drops below that of another process

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

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

- A process executes repeated time slices until its priority drops below that of another process
- Tries to avoid starvation

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

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

- 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

Highest Response Ratio Next

Highest Response Ratio Next

- New jobs get immediate attention as CPU time is near 0

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

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

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

- There are multiple FIFO run queues, RQ_0, RQ_1, $\ldots RQ_n$. with RQ_0 the highest priority, RQ_n, the lowest

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

- There are multiple FIFO run queues, RQ_0, RQ_1, $\ldots \mathrm{RQ}_n$. with RQ_0 the highest priority, RQ_n, the lowest
- Queues are processed in FIFO fashion, in priority order

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

- There are multiple FIFO run queues, RQ_0, RQ_1, $\ldots$ 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

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

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

- There are multiple FIFO run queues, RQ_0, RQ_1, $\ldots RQ_n$. 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

Multilevel Feedback Queueing

- Each process is allocated a *quantum* of time (a timeslice)

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

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*

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

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)

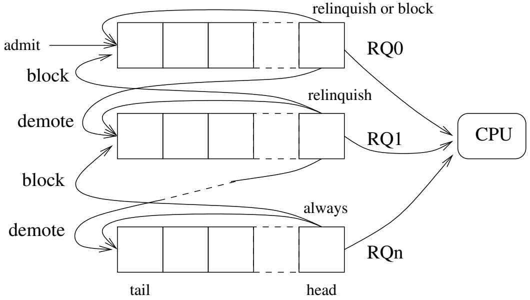
Multilevel Feedback Queueing

- 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

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

Multilevel Feedback Queueing



Multilevel Feedback Queueing

Multilevel Feedback Queueing

- This gives newer, shorter processes priority over older, longer ones

Multilevel Feedback Queueing

- This gives newer, shorter processes priority over older, longer ones
- I/O processes tend to rise, getting more priority

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

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

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

Scheduling

Algorithms

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

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

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

Scheduling
Algorithms

Traditional Unix scheduling

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

Traditional Unix scheduling

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

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

Traditional Unix scheduling

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

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

A priority is computed from the CPU use of each process

Scheduling

Algorithms

Traditional Unix scheduling

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

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

A priority is computed from the CPU use of each process

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

Traditional Unix scheduling

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

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

A priority is computed from the CPU use of each process

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

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

Scheduling

Algorithms

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

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)

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

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

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

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

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

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

Scheduling
Algorithms

Processes can choose to be *nice*

Scheduling

Algorithms

Processes can choose to be *nice*

Generally, $-20 \leq \text{nice} \leq 19$, but only certain users (administrators) can use negative nices

Scheduling
Algorithms

Processes can choose to be *nice*

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

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

Scheduling

Algorithms

Processes can choose to be *nice*

Generally, $-20 \leq \text{nice} \leq 19$, but only certain users (administrators) can use negative nices

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

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

Processes can choose to be *nice*

Generally, $-20 \leq \text{nice} \leq 19$, but only certain users (administrators) can use negative nices

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

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

But nice also enables a *purchased* priority

Scheduling
Algorithms

There are a few problems with this technique

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)

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

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

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

Fair Share Scheduling

And there are other problems that should be addressed

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?

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

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*

Fair share Scheduling in Unix

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

Fair share Scheduling in Unix

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

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

Scheduling

Algorithms

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

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

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 $O(1)$ scheduling and *The Completely Fair Scheduler*

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 $O(1)$ 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

Scheduling

Scheduling the CPU is clearly a difficult problem

Scheduling

Scheduling the CPU is clearly a difficult problem

It requires the collection and manipulation of many statistics about processes

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

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

Deadlock

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

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

Terminology

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

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

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

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

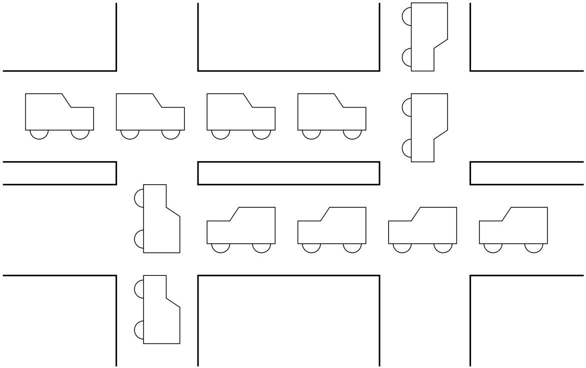
Deadlock

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

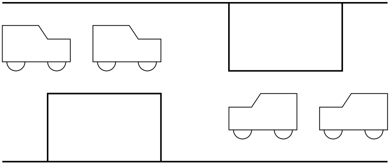
Sometimes the wait is infinite!

Deadlock



Gridlock/Deadlock

Deadlock



Gridlock/Deadlock

Deadlock

This can happen in an OS

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

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

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

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)

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

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

Deadlock

- The OS decides to run P_1

Deadlock

- The OS decides to run P_1
- P_1 runs and makes a request for access to D_1

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

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

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

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

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!

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

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

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

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

Deadlock

A formal definition of deadlock:

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

Deadlock

Note that you can only get deadlock if

Deadlock

Note that you can only get deadlock if

- there is more than one resource

Deadlock

Note that you can only get deadlock if

- there is more than one resource
- there is more than one process

Deadlock

Note that you can only get deadlock if

- there is more than one resource
- there is more than one process[1]

[1] It could technically happen with just one process, but that would be quite dumb programming to request for a resource you already have

Deadlock

Note that you can only get deadlock if

- there is more than one resource
- there is more than one process[1][2]

[1] It could technically happen with just one process, but that would be quite dumb programming to request for a resource you already have

[2] I've seen it happen