Scheduling

Algorithms

Traditional Unix scheduling

Scheduling

Algorithms

Traditional Unix scheduling

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

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

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

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

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

Scheduling

Algorithms

Traditional Unix scheduling

A process with the *smallest* priority value is chosen next (thus – mostly — a process that has used less CPU)

Scheduling

Algorithms

Traditional Unix scheduling

A process with the *smallest* priority value is chosen next (thus – mostly — a process that has used less CPU)

Processes of the same priority are treated round robin

Scheduling

Algorithms

Traditional Unix scheduling

A process with the *smallest* priority value is chosen next (thus – mostly — a process that has used less CPU)

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

Traditional Unix scheduling

A process with the *smallest* priority value is chosen next (thus – mostly — a process that has used less CPU)

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

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

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

Traditional Unix scheduling

Processes can choose to be *nice*

Scheduling

Algorithms

Traditional Unix scheduling

Processes can choose to be *nice*

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

Scheduling

Algorithms

Traditional Unix scheduling

Processes can choose to be *nice*

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

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

Scheduling

Algorithms

Traditional Unix scheduling

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

Scheduling

Algorithms

Traditional Unix scheduling

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

Traditional Unix scheduling

There are a few problems with the traditional technique

Scheduling

Algorithms

Traditional Unix scheduling

There are a few problems with the traditional technique

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

Scheduling

Algorithms

Traditional Unix scheduling

There are a few problems with the traditional 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

Traditional Unix scheduling

There are a few problems with the traditional 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

Traditional Unix scheduling

There are a few problems with the traditional 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

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?

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%?

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*

Scheduling

Algorithms

Fair share Scheduling in Unix

Recall processes are collected in groups in a tree

Scheduling

Algorithms

Fair share Scheduling in Unix

Recall processes are collected in groups in a tree

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

Scheduling

Algorithms

Fair share Scheduling in Unix

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

Scheduling

Algorithms

Fair share Scheduling in Unix

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

Fair share Scheduling in Unix

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

Fair share Scheduling in Unix

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

□