Kernel hackers' favorite spot:

The Scheduling Algorithm

Daniel P. Bovet, Marco Cesati

Talk outline

- What is the scheduler
- The scheduler of Linux 2.4
- The scheduler of Linux 2.5
- Conclusions

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems

- The effect is achieved by switching from a process to another in a very short time frame

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems

- The effect is achieved by switching from a process to another in a very short time frame

- The kernel switches to another process when:

  – a more important process becomes runnable

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems

- The effect is achieved by switching from a process to another in a very short time frame

- The kernel switches to another process when:

  - a more important process becomes runnable

  - the current process must block waiting for some event

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems
- The effect is achieved by switching from a process to another in a very short time frame
- The kernel switches to another process when:
  - a more important process becomes runnable
  - the current process must block waiting for some event
  - the current process has exhausted a predefined time quantum

Scheduling in Linux

- Linux is time-shared: multiple processes appear to be simultaneously executed, even in uniprocessor systems

- The effect is achieved by switching from a process to another in a very short time frame

- The kernel switches to another process when:

  - a more important process becomes runnable

  - the current process must block waiting for some event

  - the current process has exhausted a predefined time quantum

- The kernel program that selects the new process to run is named scheduler

Goals of the Scheduler

A good scheduler must fulfill several conflicting objectives:

- Fast process response time for interactive processes

Goals of the Scheduler

A good scheduler must fulfill several conflicting objectives:

- Fast process response time for interactive processes

- Good throughput for background jobs

Goals of the Scheduler

A good scheduler must fulfill several conflicting objectives:

- Fast process response time for interactive processes
- Good throughput for background jobs
- Avoidance of process starvation

Goals of the Scheduler

A good scheduler must fulfill several conflicting objectives:

- Fast process response time for interactive processes
- Good throughput for background jobs
- Avoidance of process starvation
- Hardware-cache awareness

Goals of the Scheduler

A good scheduler must fulfill several conflicting objectives:

- Fast process response time for interactive processes
- Good throughput for background jobs
- Avoidance of process starvation
- Hardware-cache awareness
- Fast execution time of the scheduler itself

Scheduling Policy

- Scheduling policy: the set of rules that determine when and how selecting a new process to run

Scheduling Policy

- Scheduling policy: the set of rules that determine when and how selecting a new process to run

- It is usually based on ranking the processes according to a given priority

Scheduling Policy

- Scheduling policy: the set of rules that determine when and how selecting a new process to run

- It is usually based on ranking the processes according to a given priority

- Static priority: a well-defined value assigned to a process, which specifies how “important” is the computation issued by the process itself

Scheduling Policy

- Scheduling policy: the set of rules that determine when and how selecting a new process to run

- It is usually based on ranking the processes according to a given priority

- Static priority: a well-defined value assigned to a process, which specifies how “important” is the computation issued by the process itself

- Dynamic priority: a continuously changing value that denotes the amount of system resources (mainly, CPU time) used by the process

A Process Classification

- I/O-Bound processes: make heavy use of the I/O devices (disks, networks, keyboards, . . .), and spend much time while waiting for I/O operations to complete

- CPU-Bound processes: make heavy use of the CPU (“number-crunching” applications)

A Process Classification

- I/O-Bound processes: make heavy use of the I/O devices (disks, networks, keyboards, . . .), and spend much time while waiting for I/O operations to complete

- CPU-Bound processes: make heavy use of the CPU ("number-crunching" applications)

Examples:

I/O-bound applications: word processors, database search engines, . . .
CPU-bound applications: compilers, image rendendering engines, . . .

An Alternative Process Classification

- Interactive processes: interact continuously with the user, and spend a lot of time waiting for keypresses and mouse events

- Batch processes: do not need user interaction, and often run “in background” (without terminal focus)

- Real-time processes: have very strong requirements for their response times to external events

An Alternative Process Classification

- Interactive processes: interact continuously with the user, and spend a lot of time waiting for keypresses and mouse events

- Batch processes: do not need user interaction, and often run “in background” (without terminal focus)

- Real-time processes: have very strong requirements for their response times to external events

Examples:

interactive applications: word processors, WWW browsers, . . .
batch applications: compilers, database search engines, . . .
real-time applications: video streamers, robot controllers, . . .

Process Classification in Linux 2.4

- The `policy` field of the process descriptor (**`struct task_struct`**, *include/linux/sched.h*) contains:
  - `SCHED_FIFO`: First-In First-Out real-time
  - `SCHED_RR`: Round-Robin real-time
  - `SCHED_OTHER`: non real-time

Process Classification in Linux 2.4

- The `policy` field of the process descriptor (`struct task_struct`, *include/linux/sched.h*) contains:
  - `SCHED_FIFO`: First-In First-Out real-time
  - `SCHED_RR`: Round-Robin real-time
  - `SCHED_OTHER`: non real-time
- Other type of processes are not explicitly recognized

Non Real-Time Process Preemption

Rule 1: Kernel is never preemptible

Any process running in Kernel Mode cannot be replaced
unless it voluntarily relinquishes the CPU

Non Real-Time Process Preemption

Rule 1: Kernel is never preemptible

Any process running in Kernel Mode cannot be replaced unless it voluntarily relinquishes the CPU

Rule 2: User Mode processes are always preemptible

Any User Mode process can be replaced when either

- a higher priority process becomes runnable, or
- the process has exhausted a predefined time quantum

Epoch

- The scheduler divides the CPU time in epochs

Epoch

- The scheduler divides the CPU time in epochs

- When starting a new epoch, the scheduler assigns a new “personal” time quantum to every process

Epoch

- The scheduler divides the CPU time in epochs

- When starting a new epoch, the scheduler assigns a new “personal” time quantum to every process

- When a process has exhausted its time quantum, it cannot run anymore until the epoch terminates

Epoch

- The scheduler divides the CPU time in epochs
- When starting a new epoch, the scheduler assigns a new “personal” time quantum to every process
- When a process has exhausted its time quantum, it cannot run anymore until the epoch terminates
- The epoch terminates when all *runnable* processes have exhausted their time quantum

Time Quantum — How Long?

- if it is too long, non-running processes might be freezed for long time

Time Quantum — How Long?

- if it is too long, non-running processes might be freezed for long time

- if it is too short, too many CPU cycles are “wasted” in Kernel Mode

Time Quantum — How Long?

- if it is too long, non-running processes might be freezed for long time
- if it is too short, too many CPU cycles are “wasted” in Kernel Mode

A false statement: The shorter the tq, the better the response time of interactive applications

Time Quantum — How Long?

- if it is too long, non-running processes might be freezed for long time

- if it is too short, too many CPU cycles are “wasted” in Kernel Mode

A false statement: The shorter the tq, the better the response time of interactive applications

A true statement: A too long tq degrades the responsiveness of the system

Time Quantum — How Long?

- if it is too long, non-running processes might be freezed for long time

- if it is too short, too many CPU cycles are “wasted” in Kernel Mode

A false statement: The shorter the tq, the better the response time of interactive applications

A true statement: A too long tq degrades the responsiveness of the system

Linux’s tq ranges between 10 ms and 300 ms

Base Time Quantum

- The Base Time Quantum is the default time quantum assigned to a new process

Base Time Quantum

- The Base Time Quantum is the default time quantum assigned to a new process

- On all architectures, it is roughly equal to 50 ms

Base Time Quantum

- The Base Time Quantum is the default time quantum assigned to a new process

- On all architectures, it is roughly equal to 50 ms

- The `nice()` system call can raise or lower the process' base time quantum

Base Time Quantum

- The Base Time Quantum is the default time quantum assigned to a new process

- On all architectures, it is roughly equal to 50 ms

- The `nice()` system call can raise or lower the process' base time quantum

- On a Intel-based architecture, the base time quantum is:

$$6 - \texttt{nice}/4 \quad \text{ticks}$$

where 1 ticks is about 10 ms and $-20 \leq \texttt{nice} \leq +19$

How many ticks have been spent in this epoch?

- The `counter` field of the process descriptor contains the number of ticks left to the process before its time quantum expires

How many ticks have been spent in this epoch?

- The `counter` field of the process descriptor contains the number of ticks left to the process before its time quantum expires

- A periodic timer interrupt decrements `current->counter` once every tick

How many ticks have been spent in this epoch?

- The `counter` field of the process descriptor contains the number of ticks left to the process before its time quantum expires
- A periodic timer interrupt decrements `current->counter` once every tick
- When `current->counter` becomes 0, the scheduler is invoked

How many ticks have been spent in this epoch?

- The `counter` field of the process descriptor contains the number of ticks left to the process before its time quantum expires
- A periodic timer interrupt decrements `current->counter` once every tick
- When `current->counter` becomes 0, the scheduler is invoked
- When `counter` of all *runnable* processes is 0, a new epoch starts

Starting a new epoch

- When starting a new epoch, the scheduler (`schedule()`, *kernel/sched.c*) updates the time quantum of *all* processes:

```
for_each_task(p)
    p->counter = (p->counter / 2) + (6 - p->nice/4);
```

Starting a new epoch

- When starting a new epoch, the scheduler (`schedule()`, *kernel/sched.c*) updates the time quantum of *all* processes:

```
for_each_task(p)
    p->counter = (p->counter / 2) + (6 - p->nice/4);
```

- If a process had exhausted its time quantum in the previous epoch, it got a fresh base time quantum

Starting a new epoch

- When starting a new epoch, the scheduler (`schedule()`, *kernel/sched.c*) updates the time quantum of *all* processes:

```
for_each_task(p)
    p->counter = (p->counter / 2) + (6 - p->nice/4);
```

- If a process had exhausted its time quantum in the previous epoch, it got a fresh base time quantum

- A *suspended* process gets a larger time quantum than before (half of the number of ticks left plus a base time quantum)

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

- The scheduler privileges the I/O-bound processes:

$$\text{new time quantum} = 6 - \texttt{nice}/4 + \texttt{counter}/2$$

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

- The scheduler privileges the I/O-bound processes:

$$\text{new time quantum} = \underbrace{6 - \texttt{nice}/4}_{\text{base time quantum}} + \texttt{counter}/2$$

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

- The scheduler privileges the I/O-bound processes:

$$\text{new time quantum} = \underbrace{6 - \texttt{nice}/4}_{\text{base time quantum}} + \underbrace{\texttt{counter}/2}_{\text{premium for I/O-bound}}$$

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

- The scheduler privileges the I/O-bound processes:

$$\text{new time quantum} = \underbrace{6 - \texttt{nice}/4}_{\text{base time quantum}} + \underbrace{\texttt{counter}/2}_{\text{premium for I/O-bound}}$$

- I/O-bound processes have larger time quantum duration

Catching I/O-bound processes

- The number of ticks left in a time quantum (`counter`) determines also the dynamic priority of a process

- The scheduler privileges the I/O-bound processes:

$$\text{new time quantum} = \underbrace{6 - \texttt{nice}/4}_{\text{base time quantum}} + \underbrace{\texttt{counter}/2}_{\text{premium for I/O-bound}}$$

- I/O-bound processes have larger time quantum duration
  $\Rightarrow$ have higher dynamic priority than CPU-bound processes

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	**I/O-bound**	**CPU-bound**
1	counter=6, running	counter=6, runnable

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	**I/O-bound**	**CPU-bound**
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	
7	counter=9, sleeping	counter=6, running

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	
7	counter=9, sleeping	counter=6, running
8	counter=9, sleeping	counter=5, running

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	**I/O-bound**	**CPU-bound**
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	
7	counter=9, sleeping	counter=6, running
8	counter=9, sleeping	counter=5, running
	keypress in text-editor	

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	
7	counter=9, sleeping	counter=6, running
8	counter=9, sleeping	counter=5, running
	keypress in text-editor	
8	counter=9, running	counter=5, runnable

A simple example

A text-editor and a number-crunching application are running on a single-processor machine (nice=0, base time quantum=6 ticks)

Tick	I/O-bound	CPU-bound
1	counter=6, running	counter=6, runnable
	text-editor starts sleeping	
1	counter=6, sleeping	counter=6, running
2	counter=6, sleeping	counter=5, running
	. . .	
7	counter=6, sleeping	counter=0
	new epoch starts	
7	counter=9, sleeping	counter=6, running
8	counter=9, sleeping	counter=5, running
	keypress in text-editor	
8	counter=9, running	counter=5, runnable
	text-editor starts sleeping	
8	counter=9, sleeping	counter=5, running

Dynamic priority

The dynamic priority of a process is computed by the goodness() function (*kernel/sched.c*):

Dynamic priority

The dynamic priority of a process is computed by the goodness() function (*kernel/sched.c*):

goodness($\mathtt{p}$) $= \;\; 20 - \texttt{p->nice}$ (base time quantum)

Dynamic priority

The dynamic priority of a process is computed by the goodness() function (*kernel/sched.c*):

$$\text{goodness(p)} = 20 - \texttt{p->nice} \quad \text{(base time quantum)}$$
$$+\texttt{p->current} \quad \text{(ticks left in the time quantum)}$$

Dynamic priority

The dynamic priority of a process is computed by the goodness() function (*kernel/sched.c*):

goodness(p) =	$20 -$ `p->nice`	(base time quantum)
	$+$`p->current`	(ticks left in the time quantum)
	$+1$	(if `p` uses the page tables of the previous process)

Dynamic priority

The dynamic priority of a process is computed by the goodness() function (*kernel/sched.c*):

$$\begin{aligned}
\text{goodness(p)} = \;& 20 - \texttt{p->nice} && \text{(base time quantum)} \\
& + \texttt{p->current} && \text{(ticks left in the time quantum)} \\
& + 1 && \text{(if } \texttt{p} \text{ uses the page tables of the previous process)} \\
& + 15 && \text{(in SMP, if } \texttt{p} \text{ was last running on the same CPU)}
\end{aligned}$$

Selecting the new process to run

`schedule()` scans the whole list of *runnable* processes and finds the process having highest dynamic priority:

```
c = -1000;
for (each runnable process p) do {
    w = goodness(p);
    if (w > c)
         c = w, next = p;
}
```

Selecting the new process to run

`schedule()` scans the whole list of *runnable* processes and finds the process having highest dynamic priority:

```
c = -1000;
for (each runnable process p) do {
    w = goodness(p);
    if (w > c)
         c = w, next = p;
}
```

At least one runnable process exists: the so-called *swapper* kernel thread, having PID=0

Hardware caches — 1

Hardware caches are crucial for system performance: processes with “hot cache” can run up to 100 times faster than processes with “cold cache”

Hardware caches — 1

Hardware caches are crucial for system performance: processes with “hot cache” can run up to 100 times faster than processes with “cold cache”

Two types of caches:

- Translation Lookaside Buffers: keeps the physical address associated with a linear address, as computed by looking at the current page table entries

Hardware caches — 1

Hardware caches are crucial for system performance: processes with “hot cache” can run up to 100 times faster than processes with “cold cache”

Two types of caches:

- Translation Lookaside Buffers: keeps the physical address associated with a linear address, as computed by looking at the current page table entries

- Hardware Memory Cache: keeps the contents of physical memory cells, so as to avoid a costly access to the RAM chips

Hardware caches — 2

In general, a process switching:

- Does not force the kernel to invalidate the HMC; however, the new process might find a “cold cache”

Hardware caches — 2

In general, a process switching:

- Does not force the kernel to invalidate the HMC; however, the new process might find a “cold cache”
- Forces the kernel to invalidate the TLBs, because the new process uses a different set of page tables

Hardware caches — 2

In general, a process switching:

- Does not force the kernel to invalidate the HMC; however, the new process might find a “cold cache”

- Forces the kernel to invalidate the TLBs, because the new process uses a different set of page tables

Any scheduler should be aware of the hardware caches!

Preserving the TLBs

The scheduler gives a small goodness bonus to processes that make use of the same set of page tables as the previously running process

Preserving the TLBs

The scheduler gives a small goodness bonus to processes that make use of the same set of page tables as the previously running process

Two fortunate cases:

- The new process is a clone of the previous process

Preserving the TLBs

The scheduler gives a small goodness bonus to processes that make use of the same set of page tables as the previously running process

Two fortunate cases:

- The new process is a clone of the previous process

- The new process is a kernel thread, which makes use only of linear addresses in the fourth gigabyte (identical for all processes)

Preserving the TLBs

The scheduler gives a small goodness bonus to processes that make use of the same set of page tables as the previously running process

Two fortunate cases:

- The new process is a clone of the previous process

- The new process is a kernel thread, which makes use only of linear addresses in the fourth gigabyte (identical for all processes)

In both cases, no change of the set of page table entries is required

Preserving the HMC

- Modern architectures have several levels of HMC

- The lowest cache level is integrated in the CPU chip (in multiprocessor systems, the hardware must take care of hardware synchronization)

Preserving the HMC

- Modern architectures have several levels of HMC

- The lowest cache level is integrated in the CPU chip (in multiprocessor systems, the hardware must take care of hardware synchronization)

- When a process migrates from a CPU to another, it likely finds a “cold cache”

Preserving the HMC

- Modern architectures have several levels of HMC
- The lowest cache level is integrated in the CPU chip (in multiprocessor systems, the hardware must take care of hardware synchronization)
- When a process migrates from a CPU to another, it likely finds a “cold cache”
- The scheduler attempts to “bind” any process to the CPU that has lastly executed it (+15 bonus in `goodness()`)

Rescheduling processes

When a process `p` becomes runnable, the `reschedule_idle()` function (*kernel/sched.c*) checks whether any one of the currently running processes should be preempted.

Rescheduling processes

When a process `p` becomes runnable, the `reschedule_idle()` function (*kernel/sched.c*) checks whether any one of the currently running processes should be preempted.

In order of preference:

- `p` was lastly executing on an idle CPU

Rescheduling processes

When a process `p` becomes runnable, the `reschedule_idle()` function (*kernel/sched.c*) checks whether any one of the currently running processes should be preempted.

In order of preference:

- `p` was lastly executing on an idle CPU
- There is an idle CPU that might execute `p` (least recently active CPU is choosen, because it likely has the largest number of invalid HMC lines)

Rescheduling processes

When a process `p` becomes runnable, the `reschedule_idle()` function (*kernel/sched.c*) checks whether any one of the currently running processes should be preempted.

In order of preference:

- `p` was lastly executing on an idle CPU
- There is an idle CPU that might execute `p` (least recently active CPU is choosen, because it likely has the largest number of invalid HMC lines)
- There is a CPU that might execute `p` and whose currently executing process has smaller dynamic priority than `p` (the preempted process is the one that maximes the difference)

Pitfalls of the Linux 2.4 scheduler

- The scheduler scans the whole list of runnable processes every time it must perform a process switching

Pitfalls of the Linux 2.4 scheduler

- The scheduler scans the whole list of runnable processes every time it must perform a process switching

- Starting a new epoch is expensive

Pitfalls of the Linux 2.4 scheduler

- The scheduler scans the whole list of runnable processes every time it must perform a process switching

- Starting a new epoch is expensive

- I/O-bound processes are not boosted when the number of runnable processes is high (any epoch is quite long)

Pitfalls of the Linux 2.4 scheduler

- The scheduler scans the whole list of runnable processes every time it must perform a process switching

- Starting a new epoch is expensive

- I/O-bound processes are not boosted when the number of runnable processes is high (any epoch is quite long)

- No distinction between interactive processes and batch I/O bound processes

Pitfalls of the Linux 2.4 scheduler

- The scheduler scans the whole list of runnable processes every time it must perform a process switching

- Starting a new epoch is expensive

- I/O-bound processes are not boosted when the number of runnable processes is high (any epoch is quite long)

- No distinction between interactive processes and batch I/O bound processes

- Very roughly distinction between I/O-bound and CPU-bound processes

The Linux 2.5 $\mathcal{O}(1)$ scheduler

- Linux 2.5's scheduler has been rewritten from scratch

The Linux 2.5 $\mathcal{O}(1)$ scheduler

- Linux 2.5's scheduler has been rewritten from scratch
- Runs in constant time

The Linux 2.5 $\mathcal{O}(1)$ scheduler

- Linux 2.5's scheduler has been rewritten from scratch
- Runs in constant time
- Explicitly recognizes processes as being I/O-bound or CPU-bound

The Linux 2.5 $\mathcal{O}(1)$ scheduler

- Linux 2.5's scheduler has been rewritten from scratch
- Runs in constant time
- Explicitly recognizes processes as being I/O-bound or CPU-bound
- Is still "work in progress" . . .

Constant time scheduling — 1

- Any CPU has its own runqueue of runnable processes

Constant time scheduling — 1

- Any CPU has its own runqueue of runnable processes
- Runnable processes migrate from a runqueue to another when the runqueue lengths are unbalanced

Constant time scheduling — 1

- Any CPU has its own runqueue of runnable processes

- Runnable processes migrate from a runqueue to another when the runqueue lengths are unbalanced

- Process migration is HMC aware: least recently active processes are migrated first

Constant time scheduling — 2

- Any runqueue consists of several round-robin lists including processes having the same priority

Constant time scheduling — 2

- Any runqueue consists of several round-robin lists including processes having the same priority

- At any timer tick, each CPU decrements the number of tick lefts to the current process before the time quantum expires

Constant time scheduling — 2

- Any runqueue consists of several round-robin lists including processes having the same priority

- At any timer tick, each CPU decrements the number of tick lefts to the current process before the time quantum expires

- The scheduler is invoked whenever the process has exhausted its time quantum

Constant time scheduling — 2

- Any runqueue consists of several round-robin lists including processes having the same priority

- At any timer tick, each CPU decrements the number of tick lefts to the current process before the time quantum expires

- The scheduler is invoked whenever the process has exhausted its time quantum

- The scheduler always selects the first process in the highest-priority list of the runqueue

Recognizing CPU-bound and I/O-bound processes

- The process priority does not depend on the number of ticks left in the time quantum

Recognizing CPU-bound and I/O-bound processes

- The process priority does not depend on the number of ticks left in the time quantum

- If a process goes to sleep, it is rewarded by increasing its priority

Recognizing CPU-bound and I/O-bound processes

- The process priority does not depend on the number of ticks left in the time quantum

- If a process goes to sleep, it is rewarded by increasing its priority

- Any process whose priority is higher than a given threshold is recognized as I/O bound

Recognizing CPU-bound and I/O-bound processes

- The process priority does not depend on the number of ticks left in the time quantum

- If a process goes to sleep, it is rewarded by increasing its priority

- Any process whose priority is higher than a given threshold is recognized as I/O bound

- If a CPU-bound process has exhausted its time quantum, it is inserted in a expired list, and it is never executed again until the epoch terminates

Recognizing CPU-bound and I/O-bound processes

- The process priority does not depend on the number of ticks left in the time quantum

- If a process goes to sleep, it is rewarded by increasing its priority

- Any process whose priority is higher than a given threshold is recognized as I/O bound

- If a CPU-bound process has exhausted its time quantum, it is inserted in a expired list, and it is never executed again until the epoch terminates

- If a I/O-bound process has exhausted its time quantum, it receives a fresh time quantum and it is inserted in the last position of the list associated with its priority

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler

- Add support for measuring scheduler performances and statistics

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler
- Add support for measuring scheduler performances and statistics
- Implement support for interactive processes:
  - Add a flag to the process descriptor

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler

- Add support for measuring scheduler performances and statistics

- Implement support for interactive processes:
  - Add a flag to the process descriptor
  - Implement a system call to let a process declare *I'm interactive!*

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler
- Add support for measuring scheduler performances and statistics
- Implement support for interactive processes:
  - Add a flag to the process descriptor
  - Implement a system call to let a process declare *I'm interactive!*
  - Modify the scheduler to properly handle interactive processes

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler

- Add support for measuring scheduler performances and statistics

- Implement support for interactive processes:
  - Add a flag to the process descriptor
  - Implement a system call to let a process declare *I'm interactive!*
  - Modify the scheduler to properly handle interactive processes
  - Compare performances with respect to the vanilla scheduler

Hints to beginner kernel hackers

- Start playing with the Linux 2.4 scheduler: the code is much easier to understand than the 2.5 scheduler

- Add support for measuring scheduler performances and statistics

- Implement support for interactive processes:

  – Add a flag to the process descriptor

  – Implement a system call to let a process declare *I'm interactive!*

  – Modify the scheduler to properly handle interactive processes

  – Compare performances with respect to the vanilla scheduler

- Share your changes with others! Who knows . . .