Process Scheduling

Wang Xiaolin
wx672ster@gmail.com

June 10, 2013

Contents

Textbooks:

- Chapter 4, *Process Scheduling*, [Lov10]
- Chapter 2, *Process Management and Scheduling*, [Mau08]

References

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Addison-Wesley, 2010.

[Mau08] W. Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008.

1 Multitasking

Multitasking

1. Cooperative multitasking

   Yielding a process voluntarily suspends itself

2. Preemptive multitasking

   Preemption involuntarily suspending a running process
   Timeslice the time a process runs before it's preempted
   - usually dynamically calculated
   - used as a configurable system policy

   But Linux's scheduler is different

2 Linux's Process Scheduler

Linux's Process Scheduler

up to 2.4: simple, scaled poorly

- $O(n)$
- non-preemptive
- single run queue (cache? SMP?)

from 2.5 on: $O(1)$ scheduler

- 140 priority lists — scaled well
- one run queue per CPU — true SMP support
- preemptive
- ideal for large server workloads
- showed latency on desktop systems

from 2.6.23 on: Completely Fair Scheduler (CFS)

- improved interactive performance

up to 2.4: (Sec 7.2, *The Scheduling Algorithm*, [BC05]) The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward: at every process switch the kernel scanned the list of runnable processes, computed their priorities, and selected the "best" process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly, that is, it spends too much time in high-end systems running thousands of processes.

No true SMP all processes share the same run-queue
Cold cache if a process is re-scheduled to another CPU

3 Scheduling Policy

Scheduling Policy

Must attempt to satisfy two conflicting goals:

1. fast process response time (low latency)
2. maximal system utilization (high throughput)

Linux tries

1. favoring I/O-bound processes over CPU-bound processes
2. doesn't neglect CPU-bound processes

———— ℹ ————

Process Priority

Usually,

- processes with a higher priority run before those with a lower priority
- processes with the same priority are scheduled *round-robin*
- processes with a higher priority receive a longer time-slice

———— ℹ ————

Linux implements two priority ranges:

1. **Nice value**: $-20 \sim +19$ (default 0)
   - large value $\Rightarrow$ lower priority
   - lower value $\Rightarrow$ higher priority $\Rightarrow$ get larger proportion of a CPU

   ```
   ~$ ps -el
   ```

   The *nice value* can be used as
   - a control over the *absolute* time-slice (e.g. MAC OS X), or
   - a control over the *proportion* of time-slice (Linux)
2. **Real-time**: $0 \sim 99$
   - higher value $\Rightarrow$ greater priority

   ```
   ~$ ps -eo state,uid,pid,rtprio,time,comm
   ```

———— ℹ ————

Time-slice

too long: poor interactive performance

too short: context switch overhead

I/O-bound processes: don't need longer time-slices (prefer short queuing time)

CPU-bound processes: prefer longer time-slices (to keep their caches hot)

Apparently, any long time-slice would result in poor interactive performance.

———— ℹ ————————

Problems With Nice Value

`if` two processes A and B

A: $NI = 0, t = 100ms$

B: $NI = 20, t = 5ms$

`then`, the CPU share

A: gets $\frac{100}{105} = 95\%$

B: gets $\frac{5}{105} = 5\%$

What if two $B_{ni=20}$ running?

Good news: Each gets 50%

Bad news: This '50%' is $\frac{5}{10}$, NOT $\frac{52.5}{105}$

Context switch twice every 10ms!

———— ℹ ————————

Comparing

$P_{ni=0}$ gets 100ms

$P_{ni=1}$ gets 95ms

With

$P_{ni=19}$ gets 10ms

$P_{ni=20}$ gets 5ms

This behavior means that “nicing down a process by one” has wildly different effects depending on the starting nice value.

3.1 Linux’s CFS

Completely Fair Scheduler (CFS)

For a perfect (unreal) multitasking CPU

- n runnable processes can run at the same time
- each process should receive $\frac{1}{n}$ of CPU power

For a real world CPU

- can run only a single task at once — unfair

  ☺ while one task is running

  ☹☹ the others have to wait
- `p->wait_runtime` is the amount of time the task should now run on the CPU for it becomes completely fair and balanced.

  ☺ on ideal CPU, the `p->wait_runtime` value would always be zero
- CFS always tries to run the task with the largest `p->wait_runtime` value

- kerneltrap: Linux: Discussing the Completely Fair Scheduler

CFS

In practice it works like this:

- While a task is using the CPU, its `wait_runtime` decreases

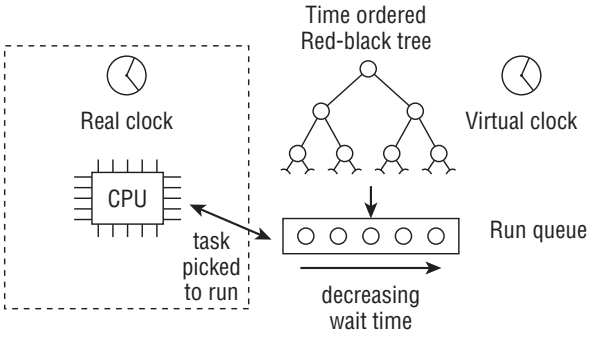
$$\texttt{wait\_runtime} = \texttt{wait\_runtime} - \texttt{time\_running}$$

if: its `wait_runtime` $\neq \texttt{MIN}_{wait\_runtime}$ (among all processes)

then: it gets preempted

- Newly woken tasks (`wait_runtime = 0`) are put into the tree more and more to the right

- slowly but surely giving a chance for every task to become the 'leftmost task' and thus get on the CPU within a deterministic amount of time



- the run queue is sorted by waiting time with a red-black tree
- the leftmost node in the tree is picked by the scheduler

- (p84, [Mau08]) An outstanding feature of the Linux scheduler is that it does not require the concept of time slices, at least not in the traditional way. Classical schedulers compute time slices for each process in the system and allow them to run until their time slice is used up. When all time slices of all processes have been used up, they need to be recalculated again. The current scheduler, in contrast, considers only the wait time of a process — that is, how long it has been sitting around in the run-queue and was ready to be executed. The task with the gravest need for CPU time is scheduled.

  ... but what do *fair* and *unfair* with respect to CPU time mean? Consider an ideal computer that can run an arbitrary number of tasks in parallel: If N processes are present on the system, then each one gets $\frac{1}{N}$ of the total computational power, and all tasks really execute physically parallel. Suppose that a task requires 10 minutes to complete its work. If 5 such tasks are simultaneously present on a perfect CPU, each will get 20 percent of the computational power, which means that it will be running for 50 instead of 10 minutes. However, all 5 tasks will finish their job after exactly this time span, and none of them will have ever been inactive!

  If multitasking is simulated by running one process after another, then the process that is currently running is favored over those waiting to be picked by the scheduler — the poor waiting processes are being treated unfairly. The unfairness is directly proportional to the waiting time.

  Every time the scheduler is called, it picks the task with the highest waiting time and gives the CPU to it. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system.

The virtual clock

Time passes slower on this clock than in real time

more processes waiting $\Rightarrow$ more slower

Example

if: 4 processes in run queue

then: the virtual clock speed is $\frac{1}{4}$ of the real clock

if: a process sitting in the queue for 20s in real time

then: resulting to 5s in virtual time

if: the 4 processes executing for 5s each

then: the CPU will be busy for 20s in real time

———— ℹ ————

- The virtual clock is a per process clock. Whenever the CPU comes, this clock starts ticking. And it stops ticking immediately when the CPU is going away.

To sort tasks on the red-black tree

```
fair_clock - wait_runtime
```

`fair_clock` The virtual time, e.g. 5s in the previous example

`wait_runtime` Fairness imbalance measure

To move a node rightward in the red-black tree

```
wait_runtime = wait_runtime - time_running
```

`time_running` When a task is allowed to run, the interval during which it has been running

———— ℹ ————

CFS

Example:
Assuming *targeted latency* is 20ms. If we have

2 processes: each gets 10ms

4 processes: each gets 5ms

20 processes: each gets 1ms

∞ **processes:** each gets 1ms (to avoid unacceptable context switching costs)

———— ℹ ————

- Sec *The Linux Scheduling Algorithm*, p49, [Lov10]

Example

A system with two processes running:

1. a text editor, say, Emacs (I/O-bound)
2. gcc is compiling the kernel source (CPU-bound)

if: they both have the same nice value

then: the proportion they get would be 50%-50%

Consequence:

- Emacs uses far less than 50% of CPU
- gcc can enjoy more than 50% of CPU freely

When Emacs wakes up

1. CFS notes that it has 50% of CPU, but uses very little of it (far less than gcc)
2. CFS preempts gcc and enables Emacs to run immediately

Thus, better interactive performance.

———— ℹ ————

4 Linux Scheduling Algorithm

Scheduler Classes

Different, pluggable algorithms coexist

- Each algorithm schedules its own type of processes
- Each scheduler class has a priority

`SCHED_FIFO`

`SCHED_RR`

`SCHED_NORMAL`

———— ℹ ————

Example: nice value difference

Assume:

1. nice value 5 pts up results in a $\frac{1}{3}$ penalty
2. targeted latency is again 20ms
3. 2 processes in the system

Then:

- $P_{ni=0}$ gets 15ms; $P_{ni=5}$ gets 5ms
- $P_{ni=10}$ gets 15ms; $P_{ni=15}$ gets 5ms

- Absolute nice values no longer affect scheduling decision
- Relative nice values does

———— ℹ ————

- p49, [Lov10]

5 The Linux Scheduling Implementation

Sec 4.5, *The Linux Scheduling Implementation*, [Lov10]

- Wikipedia: Completely Fair Scheduler
- IBM developerworks: Multiprocessing with the Completely Fair Scheduler
- Scheduling in the Linux kernel – RSDL/SD
- kerneltrap: Linux: The Completely Fair Scheduler
- Sec 7.2, [BC05].

Base Time Quantum

$O(1)$ **scheduler**

$$\begin{matrix}\text{base time quantum}\\(\text{ms})\end{matrix} = \begin{cases}(140 - \text{static priority}) \times 20 & if\ \text{static priority} < 120,\\(140 - \text{static priority}) \times 5 & if\ \text{static priority} \geq 120.\end{cases}$$

———— ℹ ————

Major Components of CFS

- Time Accounting
- Process Selection
- The Scheduler Entry Point
- Sleeping and Waking Up

———— ℹ ————

Time accounting

`sched_entity` keeps track of process accounting (`task_struct -> se`)

```
struct sched_entity {
  struct load_weight      load;           /* for load-balancing */
  struct rb_node          run_node;
  struct list_head        group_node;
  unsigned int            on_rq;

  u64                     exec_start;
  u64                     sum_exec_runtime;
  u64                     vruntime;
  u64                     prev_sum_exec_runtime;

  u64                     last_wakeup;
  u64                     avg_overlap;

  u64                     nr_migrations;

  u64                     start_runtime;
  u64                     avg_wakeup;

  /* many state variables elided, enabled only if CONFIG_SCHEDSTATS is set */
};
```

The Virtual Runtime

`vruntime` stores the *virtual runtime* of a process. On an ideal processor, all tasks' `vruntime` would be identical.

Accounting is done in `update_curr()` and `__update_curr()`

- `update_curr()` (line 518 in `kernel/sched_fair.c`)
- `__update_curr()` (line 503 in `kernel/sched_fair.c`)
- p51-p52, [Lov10]

Process Selection

The core of CFS algorithm Pick the process with the smallest `vruntime`

- run the process represented by the leftmost node in the `rbtree`

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
  struct rb_node *left = cfs_rq->rb_leftmost;
  if (!left)
    return NULL;
  return rb_entry(left, struct sched_entity, run_node);
}
```

Adding Processes to the Tree

- Happens when a process wakes up or is created
- `enqueue_entity()` and `__enqueue_entity()`

———— ℹ ————

- p54-55, [Lov10]

Removing Processes from the Tree

- Happens when a process blocks or terminates
- `dequeue_entity()` and `__dequeue_entity()`

———— ℹ ————

The Scheduler Entry Point

———— ℹ ————

Sleeping and Waking Up

———— ℹ ————