Linux Scheduler

Descending to Reality...
Philosophies
Processor Scheduling
Processor Affinity
Basic Scheduling Algorithm
The Run Queue
The Highest Priority Process
Calculating Timeslices
Typical Quanta
Dynamic Priority
Interactive Processes
Using Quanta
Avoiding Indefinite Overtaking
The Priority Arrays
Swapping Arrays
Why Two Arrays?
The Traditional Algorithm
Linux is More Efficient
Locking Runqueues
Real-Time Scheduling
Sleeping and Waking
Timers
The Linux scheduler tries to be very efficient.

To do that, it uses some complex data structures.

Some of what it does actually contradicts the schemes we’ve been discussing...
Philosophies

- Use large quanta for important processes
- Modify quanta based on CPU use
- Bind processes to CPUs
- Do everything in $O(1)$ time
Processor Scheduling

- Have a separate run queue for each processor
- Each processor only selects processes from its own queue to run
- Yes, it’s possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, the queues are rebalanced: if one processor’s run queue is too long, some processes are moved from it to another processor’s queue
Each process has a bitmask saying what CPUs it can run on

Normally, of course, all CPUs are listed

Processes can change the mask

The mask is inherited by child processes (and threads), thus tending to keep them on the same CPU

Rebalancing does not override affinity
Basic Scheduling Algorithm

- Find the highest-priority queue with a runnable process
- Find the first process on that queue
- Calculate its quantum size
- Let it run
- When its time is up, put it on the expires list
- Repeat
The Run Queue

- 140 separate queues, one for each priority level
- Actually, that number can be changed at a given site
- Actually, two sets, active and expired
- Priorities 0-99 for real-time processes
- Priorities 100-139 for normal processes; value set via `nice()` system call
The Highest Priority Process

- There is a bit map indicating which queues have processes that are ready to run
- Find the first bit that’s set:
  - 140 queues $\Rightarrow$ 5 integers
  - Only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
  - Time depends on the number of priority levels, *not* the number of processes
Calculating Timeslices

- Calculate

\[
\text{Quantum} = \begin{cases} 
(140 - \text{SP}) \times 20 & \text{if } \text{SP} < 120 \\
(140 - \text{SP}) \times 5 & \text{if } \text{SP} \geq 120 
\end{cases}
\]

where SP is the *static priority*

- Higher priority process get *longer* quanta
- Basic idea: important processes should run longer
- Other mechanisms used for quick interactive response

- Typical Quanta
- Dynamic Priority
- Interactive Processes
- Using Quanta
- Avoiding Indefinite Overtaking
- The Priority Arrays
- Swapping Arrays
- Why Two Arrays?
- The Traditional Algorithm
- Linux is More Efficient
- Locking Runqueues
- Real-Time Scheduling
- Sleeping and Waking
- Timers
### Typical Quanta

<table>
<thead>
<tr>
<th>Static Pri</th>
<th>Niceeness</th>
<th>Quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest Static Pri</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>High Static Pri</td>
<td>110</td>
<td>-10</td>
</tr>
<tr>
<td>Normal</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>Low Static Pri</td>
<td>130</td>
<td>+10</td>
</tr>
<tr>
<td>Lowest Static Pri</td>
<td>139</td>
<td>+20</td>
</tr>
</tbody>
</table>
### Dynamic Priority

- Dynamic priority is calculated from static priority and *average sleep time*
- When process wakes up, record how long it was sleeping, up to some maximum value
- When the process is running, decrease that value each timer tick
- Roughly speaking, the bonus is a number in \([0, 10]\) that measures what percentage of the time the process was sleeping recently; 5 is neutral, 10 helps priority by 5, 0 hurts priority by 5

\[
DP = \max(100, \min(SP - \text{bonus} + 5, 139))
\]
A process is *interactive* if

\[ \text{bonus} - 5 \geq \frac{S}{4} - 28 \]

Low-priority processes have a hard time becoming interactive

A default priority process becomes interactive when its sleep time is greater than 700 ms
Using Quanta

- At every time tick, decrease the quantum of the current running process
- If the time goes to zero, the process is done
- If the process is non-interactive, put it aside on the expired list
- If the process is interactive, put it at the end of the current priority queue
- If there’s nothing else at that priority, it will run again immediately
- Of course, by running so much is bonus will go down, and so will its priority and its interactive status
Avoiding Indefinite Overtaking

- There are two sets of 140 queues, active and expired.
- The system only runs processes from active queues, and puts them on expired queues when they use up their quanta.
- When a priority level of the active queue is empty, the scheduler looks for the next-highest priority queue.
- After running all of the active queues, the active and expired queues are swapped.
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched.
The Priority Arrays

```c
struct runqueue {
    struct prioarray *active;
    struct prioarray *expired;
    struct prioarray arrays[2];
};

struct prioarray {
    int nr_active;  /* # Runnable */
    unsigned long bitmap[5];
    struct list_head queue[140];
};
```
Swapping Arrays

struct prioarray *array = rq->active;

if (array->nr_active == 0) {
    rq->active = rq->expired;
    rq->expired = array;
}
Why Two Arrays?

- Why is it done this way?
- It avoids the need for traditional *aging*
- Why is aging bad?
- It’s $O(n)$ at each clock tick
for (pp = proc; pp < proc+NPROC; pp++) {
    if (pp->prio != MAX)
        pp->prio++;
    if (pp->prio > curproc->prio)
        reschedule();
}

Every process is examined, quite frequently
(This code is taken almost verbatim from 6th Edition Unix, circa 1976.)
Processes are touched only when they start or stop running

That’s when we recalculate priorities, bonuses, quanta, and interactive status

There are no loops over all processes or even over all runnable processes
To rebalance, the kernel sometimes needs to move processes from one runqueue to another. This is actually done by special kernel threads. Naturally, the runqueue must be locked before this happens. The kernel always locks runqueues in order of increasing address. Why? Deadlock prevention! (It is good for something...
Real-Time Scheduling

- Linux has soft real-time scheduling
- Processes with priorities $[0, 99]$ are real-time
- All real-time processes are higher priority than any conventional processes
- Two real-time scheduling systems, FCFS and round-robin
- First-come, first-served: process is only preempted for a higher-priority process; no time quanta
- Round-robin: real-time processes at a given level take turns running for their time quantum
Sleeping and Waking

Scheduler-Related System Calls
Major Kernel Functions
Fair Share Scheduling
Timers
Sleeping and Waking

- Processes need to wait for events
- Waiting is done by putting the process on a wait queue
- Wakeups can happen too soon; the process must check its condition and perhaps go back to sleep
DECLARE_WAIT_QUEUE(wait, current);

/* Sleep on queue ’q’ */
add_wait_queue(q, &wait);
while (!condition) {
    set_current_state(TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        /* handle signal */
        schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
Waking Up a Process

- You don’t wake a process, you wake a wait queue
- There may be multiple processes waiting for the event, i.e., several processes trying to read a single disk block
- The condition may not, in fact, have been satisfied
- That’s why the sleep routine has a loop
nice()  Lower a process’ static priority

getpriority() / setpriority()  Change priorities of a process group

sched_getscheduler() / sched_setscheduler()  Set scheduling policy and parameters. (Many more starting with sched_; use man -k to learn their names.)
Major Kernel Functions

- `scheduler_tick()`: Called each timer tick to update quanta.
- `try_to_wakeup()`: Attempts to wake a process, put in on a run queue, rebalance loads, etc.
- `re.calc_task.prio()`: Update average sleep time and dynamic priority.
- `schedule()`: Pick the next process to run.
- `rebalance_tick()`: Check if load-balancing is needed.
Suppose we wanted to add a fair share scheduler to Linux

What should be done?

Add a new scheduler type for `sched_setscheduler()`

Calculate process priority, interactivity, bonus, etc., based on all processes owned by that user

How can that be done efficiently? What sorts of new data structures are needed?
Timers

Why Does the Kernel Need Timers?
Two Basic Functions
Timer Types
Timer Ticks
Jiffies
Potent and Evil
Magic
Time of Day
Kernel Timers
Dynamic Timers
Delay Functions
System Calls
Why Does the Kernel Need Timers?

- Animated applications
- Screen-savers
- Time of day for file timestamps
- Quanta!
Two Basic Functions

- Time of day (especially as a service to applications)
- Interval timers — something should happen $n$ ms from now
Timer Types

- **Real-Time Clock**: tracks time and date, even if the computer is off; can interrupt at a certain rate or at a certain time. Use by Linux only at boot time to get time of day.
- **Time Stamp Counter**: ticks once per CPU clock; provides very accurate interval timing.
- **Programmable Interval Timer**: generates periodic interrupts. On Linux, the rate, called HZ, is usually 1000 Hz (100 Hz on slow CPUs).
- **A variety of special, less common (and less used) timers**.
Timer Ticks

- Linux programs a timer to interrupt at a certain rate
- Each tick, a number of operations are carried out
- Three most important
  - Keeping track of time
  - Invoking dynamic timer routines
  - Calling `scheduler_tick()`
- The system uses the best timer available
Each timer tick, a variable called jiffies is incremented

It is thus (roughly) the number of HZ since system boot

A 32-bit counter incremented at 1000 Hz wraps around in about 50 days

We need 64 bits — but there’s a problem
A 64-bit value cannot be accessed atomically on a 32-bit machine

A spin-lock is used to synchronize access to jiffies_64; kernel routines call get_jiffies_64()

But we don’t want to have to increment two variables each tick

Linker magic is used to make jiffies the low-order 32 bits of jiffies_64

Ugly!
Time of Day

- The time of day is stored in `xtime`, which is a `struct timespec`
- It’s incremented once per tick
- Again, a spin-lock is used to synchronize access to it
- The apparent tick rate can be adjusted slightly, via the `adjtimex()` system call
Kernel Timers

- Two types of timers use by kernel routines
- Dynamic timer — call some routine after a particular interval
- Delay loops — tight spin loops for very short delays
- User-mode interval timers are similar to kernel dynamic timers
Dynamic Timers

- Specify an interval, a subroutine to call, and a parameter to pass to that subroutine
- Parameter used to differentiate different instantiations of the same timer — if you have 4 Ethernet cards creating dynamic timers, the parameter is typically the address of the per-card data structure
- Timers can be cancelled; there is (as usual) a delicate synchronization dance on multiprocessors. See the text for details
Delay Functions

- Spin in a tight loop for a short time — microseconds or nanoseconds
- Nothing else can use that CPU during that time, except via interrupt
- Used only when the overhead of creating a dynamic timer is too great for a very short delay
time() and gettimeofday()

- adjtimex() — tweaks apparent clock rate (even the best crystals drift; see the Remote Physical Device Fingerprinting paper from my COMS E6184 class)

- setitimer() and alarm() — interval timers for applications