ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 29 — Scheduling in UNIX, Linux, & Windows

Jeff Zarnett

In this lecture we will examine how real commercial operating systems schedule their processes and threads. We will examine UNIX, Linux, and finally Windows scheduling. We will see what approaches are used and what is interesting or novel about them.

Traditional UNIX

The traditional UNIX scheduling is really ancient; as in System V R3 and BSD 4.3. It was replaced in SVR4 (which had some real-time support). The information about the traditional UNIX scheduler comes primarily from [Sta14].

The routine is a multilevel feedback system using Round Robin within each of the priority queues. Time slicing is implemented and the default time slice is a (very long) 1 second. So if a process does not block or complete within 1 s, then it will be preempted. Priority is based on the process type as well as the execution history.

Processor utilization for a process j is calculated for an interval i by the following formula:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

And the priority is for process j at interval i is calculated by the formula:

$$P_j(i) = B_j + \frac{CPU_j}{2} + N_j$$

where B_j is the base priority of process j and N_j is the "nice" value of process j. The "nice" value is a UNIX way to allow a user to voluntarily reduce the priority of a process to be "nice" to other users (but honestly, who uses this?) [Tan08]. Actually, the answer to that question is: system administrators. An admin can "re-nice" a process and make it somewhat nicer than it would otherwise be.

The CPU and N components of the equation are restricted to prevent a process from migrating outside of its assigned category. A process is assigned to a given category based on what kind of process it is. To put it in simple terms, the OS puts its own needs first and tries to make the best use of resources it can. From highest to lowest priority, the categories are:

1. Swapper (move processes to and from disk)
2. Block I/O device control (e.g., disk)
3. File manipulation
4. Character I/O device control (e.g., keyboard)
5. User processes

Yes, unfortunately, user processes get piled at the bottom of the list. The use of the hierarchy should provide for efficient use of I/O devices and tends to penalize processor-bound processes at the expense of I/O bound processes. This is beneficial, because CPU-bound processes should be able to carry on executing when an I/O-bound process waits for the I/O operation to complete, but when an I/O operation is finished, we would like to start the next I/O operation so the I/O device is not waiting. This strategy is reasonably effective for a general-purpose, time-sharing operating system.

Linux

Linux has two scheduling modes: Real-Time and Non-Real-Time (or perhaps we should call that the "normal" one). It is not necessary to use the real-time scheduler, strictly speaking, and if the real-time scheduler is used, the system can still have non-real-time threads which will be scheduled according to the normal scheduler routine.

Linux Real-Time Scheduler

The Linux scheduler operates based on *scheduling classes*, which are very much like the categories above. There are three classes into which priorities can be assigned [Sta14]:

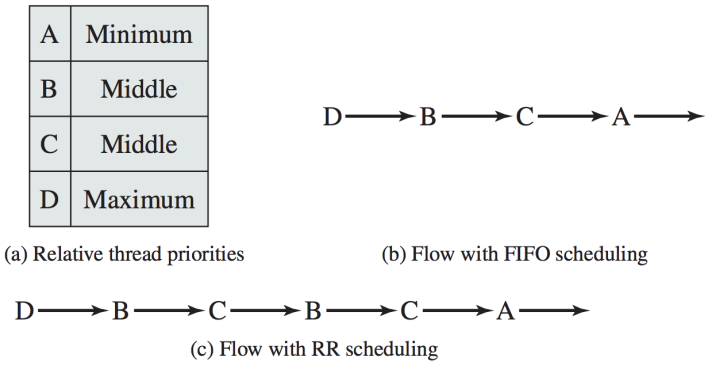
- `SCHED_FIFO`: First-In, First-Out Real-Time threads
- `SCHED_RR`: Round-Robin Real-Time threads
- `SCHED_OTHER`: Other (non-real-time) threads.

In each class, threads may have different priorities relative to one another. Lower numbers indicate higher priorities. Real-time priorities are in the range [0-99] and the other priorities are [100-139].

The rules for `SCHED_FIFO` are as follows [Sta14]:

1. The system will only interrupt a FIFO thread if one of the following is true:
   (a) Another FIFO thread of higher priority becomes read.
   (b) The current FIFO thread gets blocked (e.g., on I/O).
   (c) The current FIFO thread yields the CPU with `sched_yield`.
2. If a FIFO thread is interrupted, it is placed in the queue associated with its priority.
3. If a FIFO thread becomes ready and that thread has higher priority than the currently-executing thread, the currently-executing thread is preempted in favour of the highest priority ready FIFO thread. If two or more threads are at the highest priority, the one that has been waiting the longest is chosen.

The policy is the same for Round-Robin real-time scheduling, except time slicing is implemented. So if a Round-Robin thread has executed for a full time slice it is suspended and the scheduler will select a real-time thread of equal or higher priority (which could certainly be the same thread, but is not necessarily). The difference is illustrated in the diagram below:



(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Real-Time scheduling in Linux comparing FIFO to Round-Robin (RR) [Sta14].

One of the threads in the `SCHED_OTHER` category can execute only if there are no threads in the Round-Robin or FIFO queues that are ready at the moment.

Linux Non-Real-Time Schedulers

In Linux 2.4 and earlier (shockingly late, now that I think of it), the Linux kernel used something like the traditional algorithm. Then they introduced a scheduling algorithm that was commonly called the $O(1)$ scheduler, because it executed in constant time ($O(1)$) under all circumstances. This was a big improvement over the previous scheduling algorithm which ran in $O(n)$ time. It also worked a lot better for SMP systems, because it introduced processor affinity and load balancing. Since version 2.6.23 of the kernel, however, a new scheduling algorithm has replaced the $O(1)$ scheduler; it is called the *Completely Fair Scheduler* (CFS).

Let us start by looking at the $O(1)$ scheduler. The traditional UNIX scheduler fell down on a couple of fronts: it was not very good at handling very large numbers of processes; it was an $O(n)$ algorithm, so its performance got worse as more processes appeared in the system. It also had significant difficulty with SMP systems due to its design, notably [Sta14]:

1. A single run queue;
2. A single run queue lock; and
3. An inability to pre-empt running processes.

The single run queue means a task can and will be scheduled on any processor (good for load balancing), but there is no implementation of processor affinity. Thus, a task running on CPU-0 could be easily reassigned to CPU-1 resulting in lots of cache misses.

The single run queue lock means there is one mutual exclusion construct protecting manipulation of the run queue. Thus, when one processor wants to modify it (enqueueing or dequeueing a task, for example), all other processors have to wait until it is unlocked (which can take non-trivial time as an $O(n)$ operation for sufficiently large values of n). Thus, processors may be waiting for something to do.

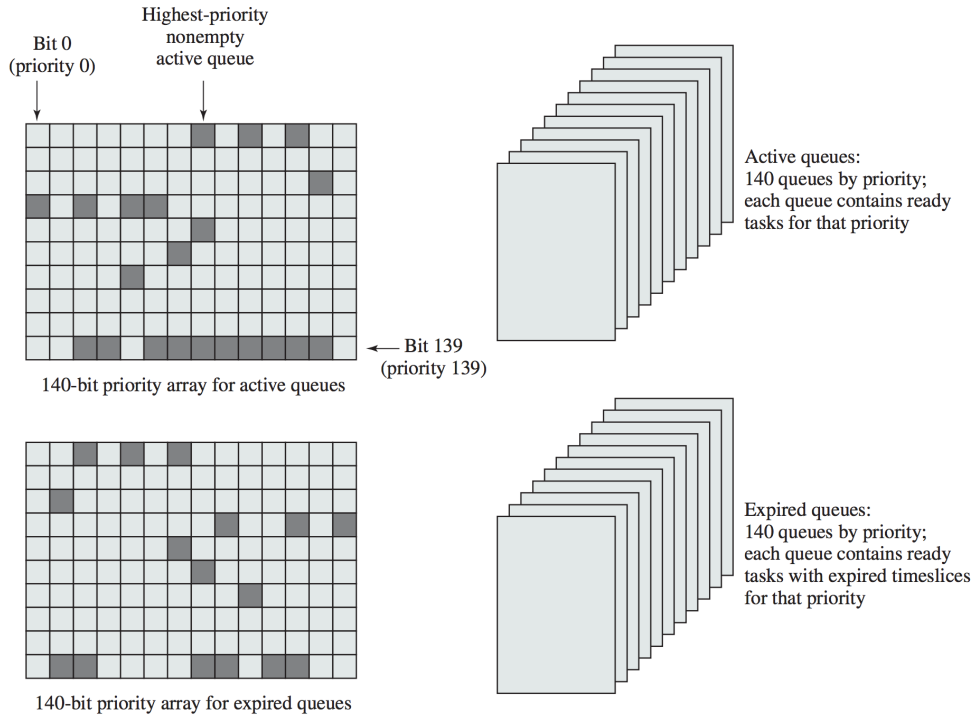
Finally, pre-emption was not possible; lower priority tasks would continue to execute while higher priority tasks were waiting. Only something getting blocked, a time slice expiration, or an interrupt might cause the scheduler to re-evaluate what process should run next.

The kernel would maintain two data structures for the processor in the system [Sta14]:

```
struct prio_array {
  int nr_active; /* number of tasks in this array */
  unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
  struct list_head queue[MAX_PRIO]; /* priority queues */
}
```

There is one queue for each priority level, and `MAX_PRIO` (140) is both the highest priority and the number of queues. The bitmap array is of a size to provide one bit per priority level, so with 140 levels and 32 bit words, `BITMAP_SIZE` is 5. The purpose of the bitmap is to indicate which queues are empty. There is an active queue structure as well as an expired queue structure.

Initially, there are no tasks in any queues and all the bits in the bitmap are zero. If a process is created and enters the ready queue, it is put in the queue corresponding to its priority value. If that queue was previously empty, then its bit in the bitmap is set to 1 to indicate that queue is no longer empty. See the diagram below:

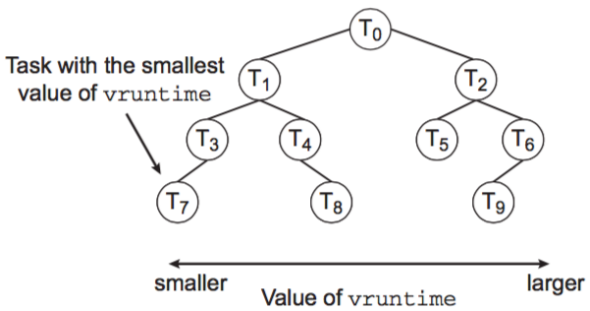


Linux $O(1)$ scheduler internal structures [Sta14].

If a process does not complete its full time slice before it is preempted, then it goes back in the ready queue. If it does run to the end of the time slice, it is placed in the expired queue instead. All scheduling takes place from the active queues. The highest priority queue is chosen; if there are multiple tasks in that queue, they are scheduled in Round-Robin fashion. This continues until the active queue structure is empty. When that happens, the active and expired queues change places, and execution (scheduling) continues [Sta14].

Part of the difficulty with the $O(1)$ scheduler is that it does not provide very good performance for interactive processes, notably the ones you work with on your desktop computer. Given that the Linux folks always claim that this year or next year is "the year of the Linux desktop" (... still waiting for that) a new scheduler was needed. Hence, the relatively rapid replacement with the Completely Fair Scheduler.

The CFS, written by Ingo Molnár, is not $O(1)$, unfortunately. It uses a red-black tree to model the ready queue, where processes are inserted based on a linear ordering of execution time. The leftmost node in this tree is therefore the task that has spent the least amount of time executing and that is what will be scheduled next. Because a red-black tree remains balanced, the time to access the leftmost root node will be $O(\ln(n))$, though caching could be used to make access to the next task faster. If a task gets blocked it will not end up in the queue again, but if it reaches the end of a timeslice or gets preempted, then it will be inserted into the tree with its updated execution time, which is very likely not the same place it was taken from (which might require rebalancing the tree (a $O(\ln(n))$ operation) [HZM14].



The Completely Fair Scheduler's red-black tree structure of ready tasks [SGG13].

Rather than using a strict rule, the CFS scheduler assigns a proportion of CPU processing time to each task based on the nice value. A nice value may be in the range -20 to +19 (lower priority is still higher priority). The CFS does not use a particular length of time slice, but instead has a *target latency* which is an interval of time in which all ready tasks should get to run at least once. The CPU time is then doled out based on the targeted latency. There are usually default and minimum values, but targeted latency can increase if there is a big increase in the number of tasks to be executed [SGG13].

The linear ordering of execution time, called `vruntime` in the earlier diagram, is also called the *virtual run time*. This is a way of keeping track of how much time a task has been executing. As with a lot of history keeping, there is a decay factor so that more recent history is more highly weighed in the calculation. Higher priority processes' history decays faster; lower priority processes' history decays more slowly. For tasks at a normal priority (nice value of zero), the virtual run time equals the physical run time. If the physical run time is, say, 50 ms, a process with a nice value of 0 will have a virtual runtime of 50. If the process has a positive nice value, the virtual runtime will be larger than 50; if a negative nice value, the virtual runtime will be less than 50 [SGG13].

Tasks that spend a lot of time using the CPU will, under this system, normally get a lower priority than a task that spends a lot of time waiting for I/O (e.g., sleeping). So a process that is user-interactive and waiting for user input will get to execute fairly quickly, making the system seem responsive to the user. Which users, of course, like.

Another thing that is noteworthy in the CFS is the addition of group scheduling: we may designate a number of processes as belonging to a group. This is useful when a process spawns lots of threads or lots of new processes. Instead of treating every thread or process totally equally, a multithreaded program's threads can all be pooled together so that the group is equal to other processes. Within the group, the scheduler will try to treat the threads or processes fairly, too.

Windows

Windows schedules threads using a priority-based, preemptive scheduling algorithm, ensuring that the highest priority thread runs. The official name for the selection routine is the *dispatcher*. A thread will run until it is preempted, blocks, terminates, or until its time slice expires. If a higher priority thread is unblocked, it will preempt a lower priority thread. Windows has 32 different priority levels, the regular (priority 1 to 15) and real-time classes (16 to 31). A memory management task runs at priority 0. The dispatcher maintains a queue for each of the scheduling priorities and goes through them from highest to lowest (yes, in Windows, higher numbers are higher priorities) until it finds something to do (and if there is nothing else currently ready, the System Idle Process will "run") [SGG13].

There are six priority classes you can set a process to via Task Manager, from highest to lowest:

1. Realtime
2. High
3. Above Normal
4. Normal
5. Below Normal
6. Low

A process is usually in the Normal class. Within that class there are some more relative priorities. Unless it is in the real-time category, the relative priority can change. The details are summarized in the table below:

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows thread priorities [SGG13].

If a process reaches the end of a time slice, the thread is interrupted, and unless it is in the real-time category, its priority will be lowered, to a minimum of a of the base priority of each class. When a process that was blocked on something (e.g., a wait or I/O), its priority is temporarily boosted (unless it is real-time, in which case it cannot be boosted). The amount of the boost depends on what the event was: a process that was waiting for keyboard input gets a bigger boost than one that was waiting for a disk operation, for example [SGG13].

The operating system also gives a priority to whatever process is running in the selected foreground window. This is different from foreground vs. background processes as discussed earlier, because the definition of a foreground process was one that was user-interactive. Here, the distinction is which of the user-interactive processes is currently "on top" in the UI. Not only does this process get a priority boost, but it also gets longer time slices. This can be disabled, if so desired, but it really highlights the different heritages of Windows and UNIX. UNIX was originally a time-sharing system with multiple users and lots of processes; Windows was originally a single-user desktop operating system doing one or maybe a few things at a time.

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.