SIENAcollege
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2022

Topic Notes: CPU Scheduling Examples

Our text covers the basics of CPU scheduling well. These notes focus on some historical, academic, and current examples.

Traditional SysV Unix

SysV Unix and BSD were the two major early flavors of Unix.

Traditional SysV Unix systems used a multilevel feedback queue.

```
scheduler:
while (no process is picked to execute) {
  for (every process on run/ready queue) {
     pick highest priority process that is loaded in memory;
  }
  if (no process is eligible to execute) {
     idle until awakened by interrupt;
  }
}
remove chosen process from run/ready queue;
switch context to that process, resume its execution;
```

Ties in priority are broken by scheduling the process that has waited the longest (FIFO).

Each process has a priority field, function of its recent CPU usage.

Processes that have used the CPU a lot get lower priority.

Highest priorities are for low-level system processes. Then a class of kernel processes, then a class of user processes (n levels of priority within).

Processes that sleep in kernel mode are given high priority so they can continue immediately when they are awakened (I/O completes, for example).

A user-level process that gets preempted gets a "recent CPU usage" field that is set to the amount of time it just spent on the CPU.

Priority is calculated as:

```
priority=(``recent CPU usage''/2) + (base priority)
```

Once a second the system's clock handler recomputes priorities of all ready procsses by halving the recent CPU usage field.

This quickly "ages" processes which had a lot of usage but were then denied the CPU as other processes run.

This makes "interactive" processes get a higher priority, as they need little CPU. When they want the CPU, they will get it.

Add in the idea of "nice" to be able to modify process priorities:

```
priority=(``recent CPU usage''/const) + (base priority) + (nice value)
```

Fair-share Scheduling

What if the goal is to divide the CPU fairly among a group of users, regardless of the number of processes they start?

With a typical RR or feedback queue system, a user can launch lots of jobs.

It can be done exactly – keep track of the proportion of time each process/user has had access to the CPU and what proportion each was supposed to have. The processes most below their fair share are selected to run.

This was done in Unix SysV by Henry in 1984.

Lottery scheduling is one way to attempt to achieve a fair share scheduling discipline.

Each process gets a number of lottery tickets proportional to its fair share of the CPU.

The scheduler "holds a lottery" at each scheduling decision point and the process with the winning number get a prize – a trip on the CPU for up to one quantum!

This can be used to implement priorities – higher priority processes just get more tickets and hence a better chance to win each time.

If tickets are given to users to dole out among their processes, it can produce fair schedules even if some user tries to put many more processes on the system.

A new high-priority job can have a good chance for a good response time if it is given lots of tickets.

The Petrou paper linked from Canvas has all the details (you are not responsible for those details, but should just be able to reason about the general idea).

Multiple Processors/Cores

How have more than one processing core impact CPU scheduling? As you might expect, things become more complex!

For simplicity (and reflecting most realities) we assume symmetric multiprocessing: all CPUs/cores can run any task.

Options:

- separate queues for each CPU
- one set of queues shared among all CPUs

Having separate queues for each CPU makes it easy to choose the next process to run but we need to figure out in which CPU's queue to place a job and if/when to move jobs among the CPUs.

Having a shared single queue or set of queues means that any idle CPU can run any ready job.

But think about what this means if multiple CPUs need to choose a new process at exactly the same time.

Could they both choose the same job and either duplicate work (bad) or maybe corrupt kernel structures?

This is possible: they are truly concurrent – 2 independent processors.

If the CPUs have to wait and be sure only one is choosing a job next, that could be slow and would not scale well.

There is a potential advantage to schedule a process on the same CPU on which is was last executed. If we stay on the same CPU, there's a chance for cache reuse. Otherwise, we'll have misses for sure as the process ramps up on the new CPU.

This is called *affinity*.

But we don't necessarily want a process to be pinned to a CPU forever – processes come and go and this will not give a long-term load balance.

Solaris Scheduling

Solaris, the long-popular operating system from Sun Microsystems (now part of Oracle) includes these major classes for scheduling of threads:

- highest priority to real-time tasks
- next highest to system/kernel service threads
- fair share
- fixed priority
- interactive
- time-sharing

Solaris separates interactive and time-sharing to try to give GUIs and similar things a very fast response, even on a system with a good number of time-sharing type processes competing for the CPU.

Within a class, there are priorities. Threads with a given priority have a specific quantum (200-20) and depending on whether the thread leaves the CPU because the quantum expires or it blocks, it is given a different priority for its next time slice.

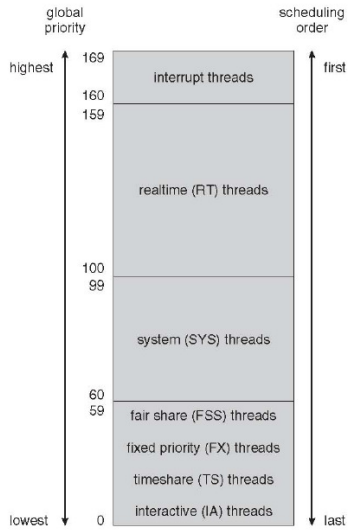
A subset of the table for time-sharing and interactive threads:

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

SGG Fig. 5.12

We can see that the higher the priority, the shorter the quantum. Any time a thread's quantum expires, its priority is reduced by 10. When it returns from sleep, its priority is increased. This is intended to provide fast response times for interactive processes.

The overall system is shown by:



SGG Fig. 5.13

Linux O(1) Scheduling

Linux CPU schedulers have evolved over the years.

- Kernel version 1.2 used a very simple round robin policy with a circular ready queue. It was not intended for advanced architectures.
- Kernel version 2.2 introduced scheduling classes (real-time, non-preemptive, non-real-time), and added SMP support.
- Kernel version 2.4 used a scheduler that operated in $O(N)$ time for a system with N tasks. It needed to iterate over the entire task list at each scheduling decision to find the task with the highest "goodness function".
  - Time is divided into *epochs*, where each task was allowed to execute until its time slice expired. If a task blocked before its quantum expired, half of its remaining slice would be added in the next epoch (which effectively increases in priority).
  - The scheduler was simple, but inefficient and not scalable to heavy loads or larger numbers of processors.

This led to the development of the $O(1)$ *scheduler*, which became the default in early 2.6 series kernels.

This scheduler is much more efficient – it selects a process for scheduling in constant time.

There are some details in Krohn 2003, linked from Canvas.

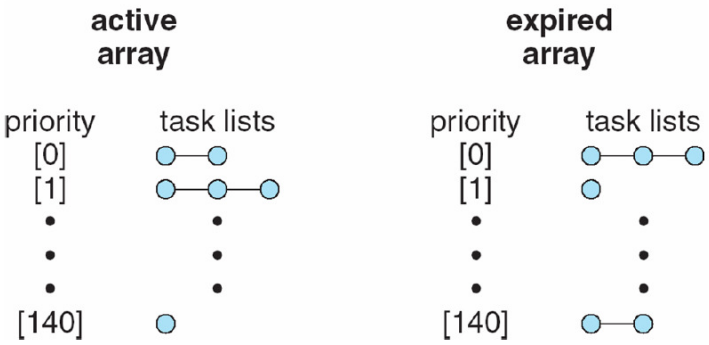
Priorities range from 0 to 140, with 0-99 reserved for real-time tasks, 100-140 for regular tasks.

Linux O(1) treats the time quantum differently than Solaris or Windows – low priority tasks get a short quantum (10 ms), ranging up to a 200 ms quantum for the highest-priority tasks.

It is fully preemptible – when a high priority task arrives, a lower priority task that is executing will be preempted for quick response times. Processes executing kernel code can be preempted - this was not the case in the previous Linux kernel.

For SMPs/CMPs, one "runqueue" is maintained for each processor. Claim: it scales well up to 64-processor systems.

Each runqueue is made up of two lists: the tasks that have not yet exhausted their time quantum and those that have:



SGG Fig. 5.16

Only tasks from the "active" array are chosen until all of those have exhausted their quantum and have then moved to the "expired" array. When the active array is empty, the expired and active arrays swap roles.

This ensures that all tasks have a chance to run and exhaust their quanta before others can come around to have another chance.

Each processing unit has its own runqueue set and always chooses the highest priority task from its own runqueue active arrays to execute next. Load balancing is achieved by having tasks move among runqueues when a CPU is idle, or periodically when not.

Dynamic priorities are computed and I/O bound jobs are given longer time slices.

Linux Completely Fair Scheduler

Starting in Kernel 2.6.23, the *Completely Fair Scheduler (CFS)* became the default.

Based on Kolivas' work on the *Rotating Staircase Deadline Scheduler*, CFS attempts to provide a fair or balanced access to the CPU(s) for all tasks in the system.

Details at: `https://developer.ibm.com/tutorials/l-completely-fair-scheduler/`

Of note: the run queues are replaced by a balanced binary tree!

FreeBSD

McKusick and Neville-Neil talk in detail about FreeBSD scheduling.

`http://portal.acm.org/citation.cfm?id=1035594.1035622&coll=portal&dl=ACM&idx=1035594&part=periodical&WantType=periodical&title=Queue&CFID=39231776&CFTOKEN=54087012`

FreeBSD up to 5.1 used the 4.4BSD scheduler. Its features include:

- multilevel feedback queues
- the next thread to run is always the one with the highest priority
- multiple processes at a priority are executed in round robin fashion
- immediate switch to newly-arrived higher priority job if the current job is in user mode only (interrupt is generated)

The article describes in detail how priorities are computed. They are based on two values associated with a thread:

1. `p_estcpu` – estimate of recent CPU utilization of the thread
2. `p_nice` – "nice" value between -20 (high priority) and 20 (low priority), by default is 0

This is similar to the Unix SysV approach mentioned earlier.

It does take into account the system "load average" when deciding on the decay rate of the CPU usage field.

Like SysV, it recomputes priorities once per second.

Blocked tasks do not need their priorities recomputed until they return to the system, so their recent CPU usage is computed as a function of system load and sleep time when they are returned to the ready state.

Even so, consider a heavily-loaded system. Once a second, lots of processes need to have their priorities recomputed and may be moved among the queues.

FreeBSD 5.2 and beyond use the *ULE scheduler*.

Like the Linux O(1) scheduler, it is intended to addresses SMP and multicore systems and is not dependent on the number of threads.

Why the name? It's implemented in `sched_ule.c` in the kernel code!

We can check it out on any FreeBSD 5 or higher machine, such as `noreaster.teresco.org`, which is running 13.1.

It includes per-processor queues to allow for affinity scheduling.

Threads can be migrated to another CPU when there is an idle processor to occupy (which a loaded processor can detect based on a bitfield). The system also periodically balances the loads of the most loaded and least loaded processors.

ULE also addresses multicore processors.

From the point of view of scheduling, these are multiple CPUs, but they are a little different in that there should be less of a penalty for migrating among processors on the same chip.

Each processor has three queues:

- idle queue – all "idle" (low priority) threads – these run only when there are no threads of higher priority to be executed
- current queue – set of jobs ready to run
- next queue – another set of jobs ready to run but only after the current queue empties

After all jobs from "current" are gone, the current and next queues are swapped.

Interactive jobs are inserted back into current after they have a turn on the CPU in order to maintain good response for those threads.

It decides which jobs are interactive based on the ratio of sleep time to run time.