Computer Science 330
Operating Systems
Siena College
Spring 2012

Topic Notes: CPU Scheduling

The CPU is the first of several scarce resources that we will consider. A primary function of an operating system is to determine which processes (and, in turn, users) get to utilize the CPU(s).

CPU scheduling can be done at three different levels:

1. *Long-term Scheduling* – also known as *batch scheduling*. This scheduler decides which jobs/processes are allowed into the system to compete for CPU (and other) resources.
2. *Short-term Scheduling* – or *interactive scheduling*. This scheduler decides which from a collection of ready processes gets to access the CPU next.
3. *Medium-term Scheduling* – or *memory scheduling*. This scheduler decides if/when a process should be "swapped out" or back in based on memory available.

We will discuss primarily short-term scheduling here, though we will discuss all of the algorithms that may also be applied to medium-term and long-term scheduling.

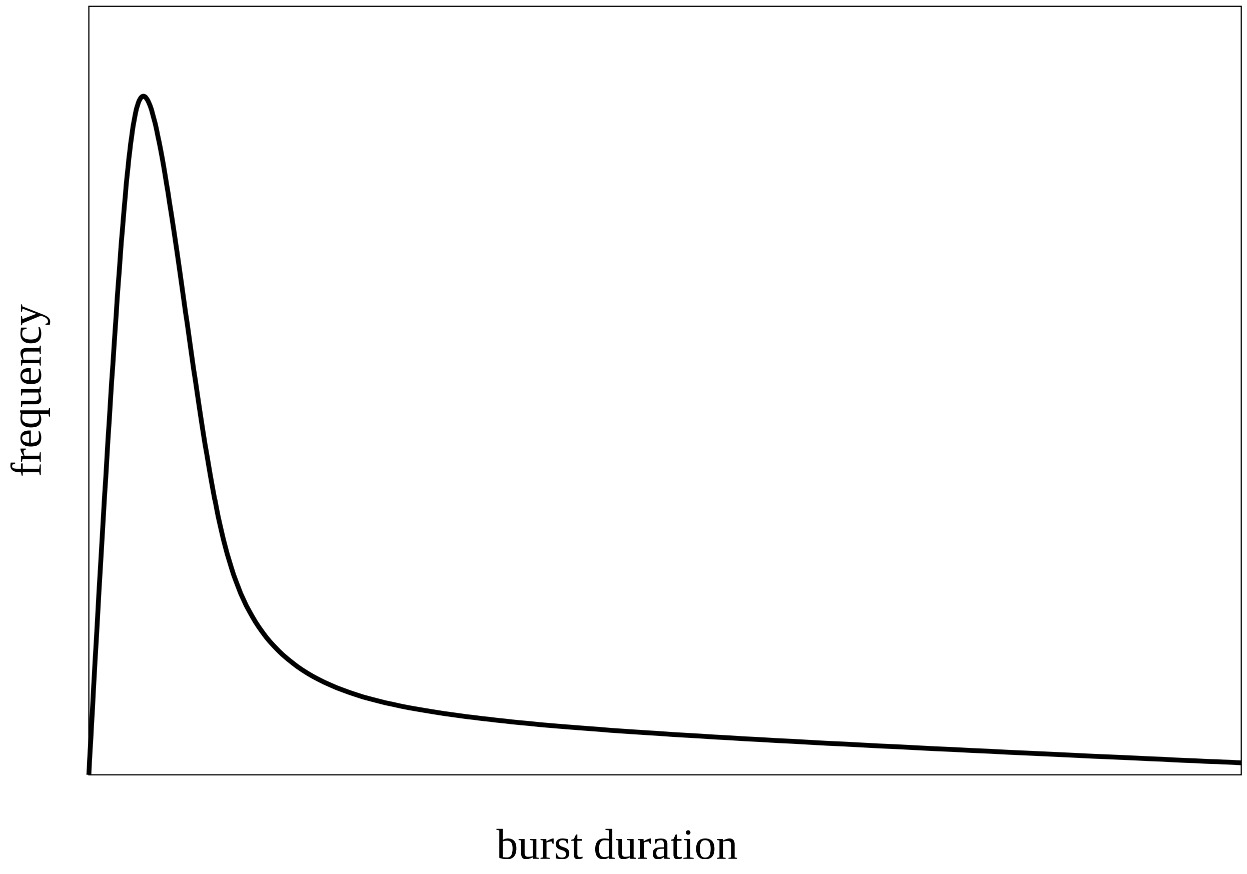
For the purposes of this discussion of general scheduling algorithms, I will use the term "process" but it could also apply to threads in many cases.

A typical process alternates between the need for CPU and the need for I/O service throughout its lifespan. This is called the *CPU-I/O burst cycle*.

It is this fact that makes multiprogramming essential. When a process needs I/O, it's good to have another process ready to move in and take advantage of the available CPU resource.

The amount of time that it can make use of the CPU is known as its *CPU burst time*.

A typical distribution of CPU burst times looks like this:



Processes may be categorized as:

- *CPU-bound* – process does not need much I/O service, almost always want the CPU
- *I/O-bound* – short CPU burst times, needs lots of I/O service
- *Interactive* – short CPU burst times, lots of time waiting for user input (keyboard, mouse)

The type of processes in the system will affect the performance of scheduling algorithms.

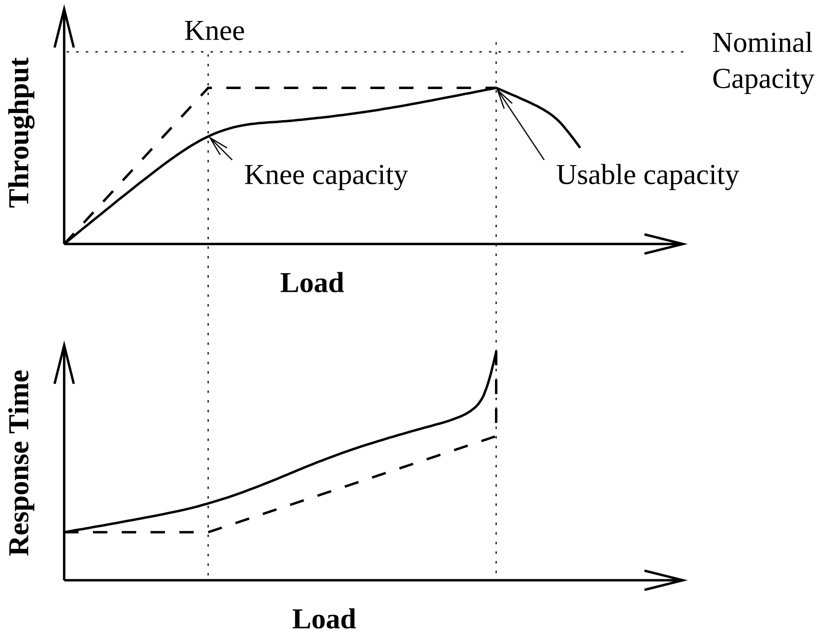
A short-term CPU scheduling decision is needed when a process:

1. switches from a running to a waiting state (non-preemptive)
2. switches from a running to a ready state (preemptive)
3. switches from a waiting to a ready state (preemptive)
4. terminates (non-preemptive)

Goals of a CPU scheduler

- maximize *CPU Utilization* – keep the CPU busy doing useful work (owner)
- maximize *Throughput* – rate of process completion (owner)
- minimize *Turnaround Time* – amount of time to execute a particular process (user)
- minimize *Waiting Time* – amount of time that a process is waiting in the ready queue (user)
- minimize *Response Time* – amount of time it takes from a process' arrival until its first turn on the CPU (user)

Response Time's Relationship to Throughput



CPU Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

As its name suggests, the first process to arrive gets to go first. It is a non-preemptive FIFO system.

Example:

Consider 4 processes P_1, P_2, P_3, and P_4 that have burst times of 18, 3, 5, and 4, respectively.

If the processes arrive in the order P_1, P_2, P_3, P_4, FCFS will service them in that order. We visualize this with a *Gantt Chart*:

```
|--------P1--------|-P2|--P3-|-P4-|
0                 18  21    26   30
```

Waiting times: $P_1 = 0$; $P_2 = 18$; $P_3 = 21$; $P_4 = 26$, Avg: $\frac{0+18+21+26}{4} = 16.25$.

But what if the processes arrive P_2, P_4, P_3, P_1?

```
|-P2|-P4-|--P3-|--------P1--------|
0   3    7    12                 30
```

Waiting times: $P_1 = 12$; $P_2 = 0$; $P_3 = 7$; $P_4 = 3$, Avg: $\frac{12+0+7+3}{4} = 5.5$.

FCFS characteristics:

- Penalizes short jobs
- Rewards long jobs
- Large variance in throughput
- Sensitive to arrival order
- Is starvation free – every job gets its turn
- Easy to implement

Shortest-Job-First (SJF) Scheduling

a.k.a. Shortest Process Next (SPN)

Choose the process with the smallest next CPU burst.

Non-premptive – process does not leave until its burst is complete

Preemptive – if a new process arrives with CPU burst length less than the remaining time of the currently executing process, we preempt. This is also known as *Shortest Remaining Time First (SRTF)*.

Example:

Consider these processes

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Non-preemptive:

```
|---P1---|P3|--P2--|--P4--|
0        7  8     12     16
```

Average waiting time: $\frac{0+6+3+7}{4} = 4$.

Preemptive (SRTF):

```
|-P1-|-P2-|P3|-P2-|--P4--|--P1---|
0    2    4  5    7      11      16
```

Average waiting time: $\frac{9+1+0+2}{4} = 3$.

SJF characteristics:

- It is *optimal* in minimizing waiting time
- Penalizes long jobs
- Rewards short jobs
- Somewhat sensitive to arrival order
- Gives optimal nonpreemptive throughput
- Permits starvation
- Difficult to predict burst/service times. We can try to estimate it based on previous bursts, or by averaging (text discusses this). However, this leads to an approximate SJF, which would not necessarily provide the optimal schedule.

Priority Scheduling

Associate a priority with each process and always choose the one with the highest priority.

SJF/SRTF are examples of this, with the priority assigned as the burst times.

Biggest problem: potential starvation of low-priority jobs.

One technique to avoid starvation is *aging* – increasing the priority of processes that are not getting any CPU time.

Round Robin (RR) Scheduling

Each process gets a small unit of time on the CPU (the *time quantum*), typically 10-100 ms. After this time, the job is preempted and added to the end of the ready queue.

For n processes and quantum q, each process gets $\frac{1}{n}$ of the CPU time. No process waits more than $(n-1)q$ for its next turn on the CPU.

RR characteristics:

- Preemptive (at quantum q)

- Less sensitive to arrival order
- Quantum should not be too small relative to context switch time
- At overly large q, approximates FCFS
- Low overhead
- Starvation impossible

Example, $q = 20$:

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

```
|-P1-|-P2-|-P3-|-P4-|-P1-|-P3-|-P4-|-P1-|-P3-|-P3-|
0   20   37   57   77   97   117  121  134  154  162
```

What if we changed q to 5? Lots more context switching - more overhead. You can see this in the queueing system lab.

Real values for a quantum tend to be in the 20-200 ms range, usually about 100 ms. This, amazingly, has been consistent for decades as hardware and operating systems have evolved.

Perhaps this is more a function of the delay that a human is willing to accept than something related to hardware speeds.

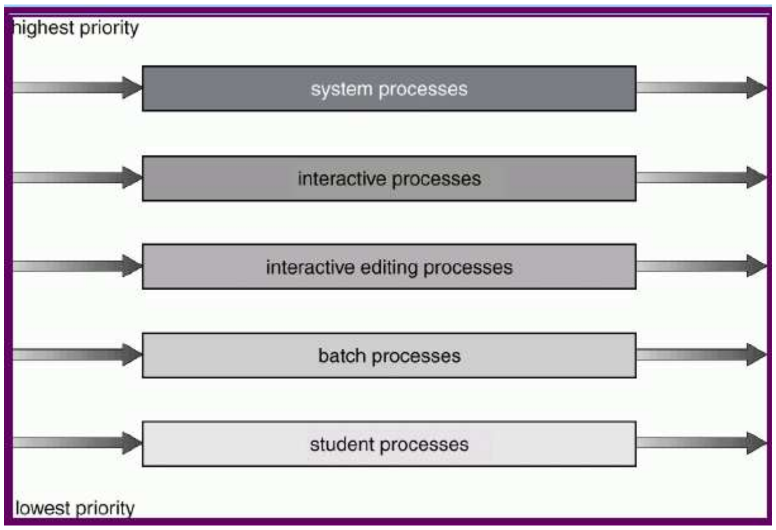
Multilevel queues

We can partition the ready queue into a number of queues.

Perhaps with 2 queues, one can have an RR discipline for interactive jobs and the other can have a FCFS discipline for batch jobs.

Need to schedule among the queues as well, then.

Perhaps, a priority system – take jobs from high priority queues:



Multilevel Feedback Queues

This is a method used in real systems.

Processes may move among queues; aging may be implemented this way.

Parameters:

- number of queues
- when to move a process to a different queue
- where new processes will enter
- scheduling among queues

A simple example:

Three queues, Q_0 has $q = 50ms$, Q_1 has $q = 100ms$, Q_2 is FCFS.

Processes enter at Q_0, if not finished after 50 ms, move to Q_1, if not finished after an additional 100 ms, move to Q_2.

Q_2 is served only when Q_1 and Q_0 are empty. Q_1 is served only when Q_0 is empty.

Alternately, if there are a large number of queues, the time given to jobs in each state can be allocated among the queues to continue to make progress on low-priority jobs while still paying most attention to the high-priority ones.

How does this benefit interactive jobs? How does this benefit CPU-bound jobs? I/O-bound jobs?

See the Unix `nice(1)` command for ways to change the priority of a process in Unix.

Traditional SysV Unix systems use this kind of scheduling:

```
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 do this.

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.

See the Petrou paper linked from the lecture page for details (though you are not responsible for those details, just the idea).

Comparing Algorithms

How can we compare the algorithms and approaches?

- Deterministic modeling – consider a set of processes and see what happens with various algorithms. Our Gantt charts were an example.
- Queueing models – take a mathematical approach. Given mathematical descriptions of the kind of processes and the underlying system, determine expected performance. This would likely be a major topic in a graduate level operating systems course.
- Implementation – try it out on a real system. May be impractical, but it's the best test, of course.
- Simulation – devise a model and simulate it. You'll do this!

Multiple Processors/Cores

How do we do CPU scheduling when we have more than one processing core? 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.

Some architectures also feature a *non-uniform memory access (NUMA)*, where each CPU has access to all of the system's memory, but has faster access to some subset of the memory (e.g., the memory on the same board as the processor). In such a case, there is an advantage to allocating the memory for and scheduling the process to take advantage of the faster memory access when possible.

Examples

Linux 2.6 and FreeBSD 5.x and up have introduced relatively new schedulers. More on these later.

The book says some things about a few systems.

Solaris

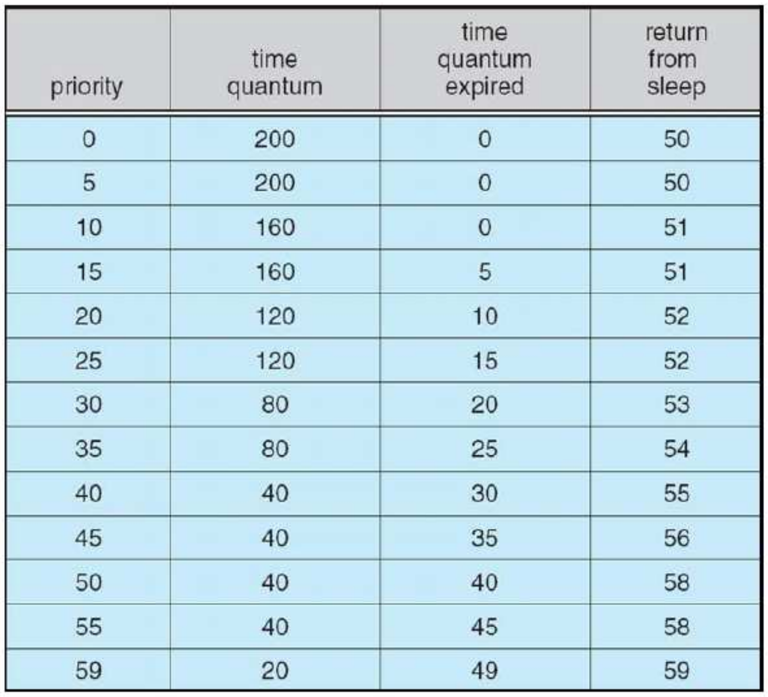
Solaris, the long-popular operating system from Sun Microsystems (now part of Oracle) includes 4 major classes for scheduling of threads:

- 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:

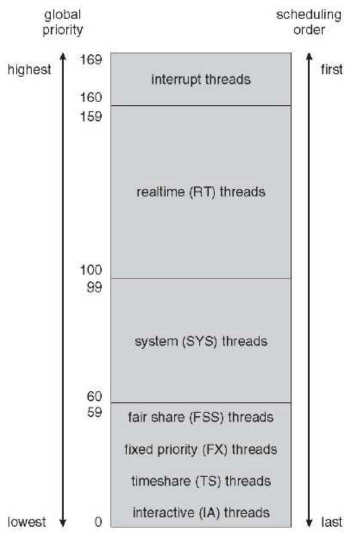


priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

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

Windows XP

Windows uses a strict priority scheme. The highest-priority task always runs next. There are several classes of threads and priorities within a class. Each thread is assigned a single number that represents its priority.

As a primarily single-user operating system, Windows also attempts to give the program running in the "foreground" of the user interface a better chance to run (and hence, be responsive). It accomplishes this by giving threads for that program a larger time quantum.

Linux O(1)

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 developmenf of the $O(1)$ *scheduler*, which was 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 the text and in the supplemental reading (Krohn 2003).

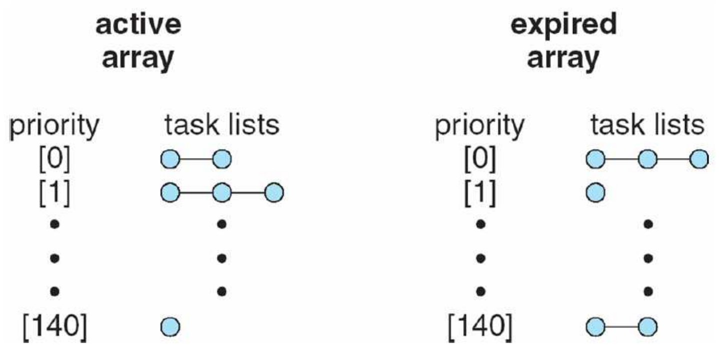
Priorities range from 0 to 140, with 0-99 reserved for real-time tasks, 100-140 for regular tasks.

Linux 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)* has been 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: `http://www.ibm.com/developerworks/linux/library/l-completely-fair-sc`

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 uses 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)
- see methods: `resetpriority()` (note that the code matches the formula in the article), `setrunnable()`, `wakeup()`, `roundrobin()`, and `schedcpu()`, many defined in `/sys/kern/kern_synch.c` on a FreeBSD 4.x system.

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 discussed 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 `winterstorm.teresco.org`, which is running 8.2.

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.

Final Thoughts

Modern schedulers are very concerned with SMP.

A good scheduler will have the capability to be tuned to the specifics of a system.

A scheduler must be able to select a task quickly even when there are a lot of jobs in the system.

CPU scheduling remains an active area of research and development, in part because the underlying computer architectures continue to evolve, rendering some old assumptions invalid.

For thought: when choosing the time slice in a priority system, we see some systems that give high-priority jobs a longer quantum, others give high-priority jobs a shorter quantum. Why?