Operating Systems

Process Scheduling

By Paul Krzyzanowski
February 13, 2012 [origiinal: September 27, 2010]

> *The finest eloquence is that which gets things done; the worst is that which delays them.*
> *— David Lloyd George (speech, Jan. 1919)*

Introduction

If you look at any process, you'll notice that it spends some time executing instructions (computing) and then makes some I/O request, for example to read or write data to a file or to get input from a user. After that, it executes more instructions and then, again, waits on I/O. The period of computation between I/O requests is called the **CPU burst**.

Interactive processes spend more time performing and waiting for I/O and generally experience short CPU bursts:

Compute-intensive processes, conversely, spend more time running instructions and less time on I/O. They exhibit long CPU bursts:

Most interactive processes, in fact, spend the vast bulk of their existence doing nothing but waiting on data. As I write this on my Mac, I have 44 processes running just under my user account. This includes a few browser windows, a word processor, spreadsheet, several shell windows, Photoshop, iTunes, and various monitors and utilities. Most of the time, these processes collectively are using less than 3% of the CPU. This is not surprising since most of these programs are waiting for user input, a network message, or sleeping and waking up periodically to check some state.

Consider a quad-core 3.4 GHz Intel Core i7 processor. It can execute a peak rate of 13.6 billion instructions per second (four hyperthreaded cores). It can run 136 billion instructions in the ten seconds it might take you to skim a web page before you click on a link — or 3.4 billion instructions in the quarter second it might take you to hit the next key as you're typing. The big idea in increasing overall throughput was the realization that we could switch the processor to running another process when a process has to do I/O. This is **multiprogramming**. The next big idea was realizing that you could preempt a process and let another process run and do this quickly enough to give the illusion that many processes are running at the same time. This is **multitasking**.

Scheduler

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests and a small number of processes with long CPU bursts. To provide good time-sharing performance, we may preempt a running process to let another one run. The **ready list**, also known as a **run queue**, in the operating system keeps a list of all processes that are ready to run and not blocked on some I/O or other system request, such as a semaphore. The entries in this list are pointers to the process control block, which stores all information and state about a process. When an I/O request for a process is complete, the process moves from the *waiting* state to the *ready* state and gets placed on the ready list.

The **process scheduler** is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which

process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

1. The current process goes from the *running* to the *waiting* state because it issues an I/O request or some operating system request that cannot be satisfied immediately.
2. The current process terminates.
3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the *running* to the *ready* state.
4. An I/O operation is complete for a process that requested it and the process now moves from the *waiting* to the *ready* state. The scheduler may then decide to move this *ready* process into the *running* state.

A scheduler is a **preemptive** scheduler if it has the ability to get invoked by an interrupt and move a process out of a *running* state and let another process run. The last two events above may cause this to happen. If a scheduler cannot take the CPU away from a process then it is a **cooperative**, or **non-preemptive** scheduler. Old operating systems such as Windows 3.1 or MacOS before OS X are examples of such schedulers. Even older batch processing systems had **run-to-completion** schedulers where a process ran to termination before any other process would be allowed to run.

The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the **scheduling algorithm** (or **scheduling policy**). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

- Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
- Be efficient – keep the CPU busy all the time.
- Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
- Minimize response time – interactive users should see good performance.
- Be predictable – a given job should take about the same amount of time to run when run multiple times. This keeps users sane.
- Minimize overhead – don't waste too many resources. Keep scheduling time and context switch time at a minimum.
- Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We'll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more important, processes cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.
- Avoid indefinite postponement – every process should get a chance to run eventually.
- Enforce priorities – if the scheduler allows a process to be assigned a priority, it should be meaningful and enforced.
- Degrade gracefully – as the system becomes more heavily loaded, performance should deteriorate gradually, not abruptly.

It is clear that some of these goals are contradictory. For example, minimizing overhead means that jobs should run longer, thus hurting interactive performance. Enforcing priorities means that high-priority processes will always be favored over low-priority ones, causing indefinite postponement. These factors make scheduling a task for which there can be no perfect algorithm.

To make matters even more complex, the scheduler does not know much about the behavior of each process. As we saw earlier, some processes perform a lot of input/output

operations but use little CPU time (examples are web browsers, shells and editors). They spend much of their time in the blocked state in between little bursts of computation. The overall performance of these **I/O bound processes** is constrained by the speed of the I/O devices. **CPU-bound processes** and spend most of their time computing (examples are ray-tracing programs and circuit simulators). Their execution time is largely determined by the speed of the CPU and the amount of CPU time they can get.

To help the scheduler monitor processes and the amount of CPU time that they use, a programmable interval timer interrupts the processor periodically (typically 50 or 60 times per second). This timer is programmed when the operating system initializes itself. At each interrupt, the operating system's scheduler gets to run and decide whether the currently running process should be allowed to continue running or whether it should be suspended and another ready process allowed to run. This is the mechanism used for preemptive scheduling.

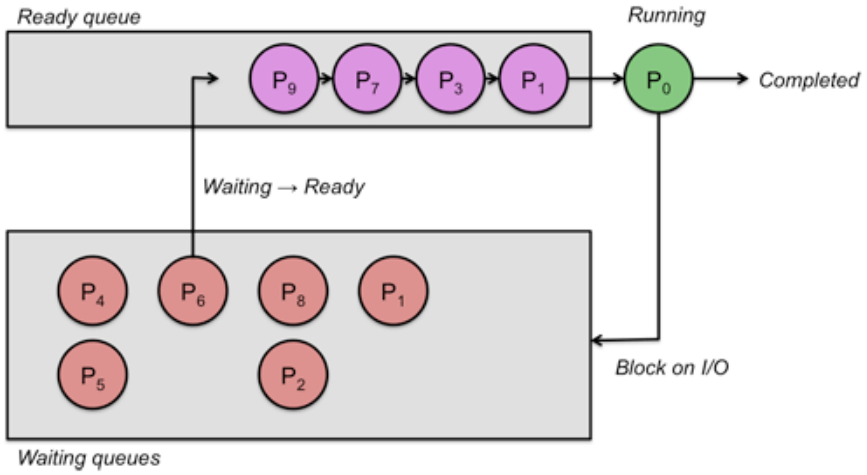
Preemptive scheduling allows the scheduler to control response times by taking the CPU away from a process that it decided has been running too long. It has more overhead than nonpreemptive scheduling since it has to deal with the overhead of context switching processes instead of allowing a process to run to completion or until the next I/O operation or other system call. However, it allows for higher degrees of concurrency and better interactive performance.

The scheduling algorithm has the task of figuring out whether a process should be switched out for another process and which process should get to run next. The **dispatcher** is the component of the scheduler that handles the mechanism of actually getting that process to run on the processor. This requires loading the saved context of the selected process, which is stored in the process control block and comprises the set of registers, stack pointer, flags (status word), and a pointer to the memory mapping (typically a pointer to the page table). Once this context is loaded, the dispatcher switches to user mode via a *return from interrupt* operation that causes the process to execute from the location that was saved on the stack at the time that the program stopped running — either via an interrupt or a system call.

In the following sections, we will cover a number of scheduling algorithms.

First-Come, First-Served Scheduling

Possibly the most straightforward approach to scheduling processes is to maintain a FIFO (first-in, first-out) ready queue. New processes go to the end of the queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. The scheduler is non-preemptive. If the process has to block on I/O, it enters the *waiting* state and the scheduler picks the process from the head of the queue. When I/O is complete and the process is ready to run again, it gets put at the end of the queue.



With first-come, first-served scheduling, a process with a long CPU burst will hold up other processes. Moreover, it can hurt overall throughput since I/O on processes in the *waiting* state may complete while the CPU bound process is still running. Now devices are not being used effectively. For increasing throughput, it would have been great if the

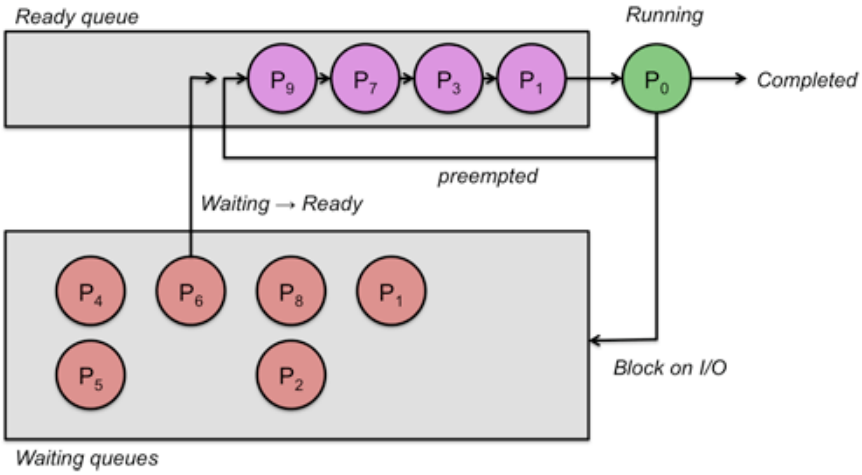
scheduler instead could have briefly run some I/O bound process that could request some I/O and wait. Because CPU bound processes don't get preempted, they hurt interactive performance because the interactive process won't get scheduled until the CPU bound one has completed.

Advantage: FIFO scheduling is simple to implement. It is also intuitively fair (the first one in line gets to run first).

Disadvantage: The greatest drawback of first-come, first-served scheduling is that it is not preemptive. Because of this, it is not suitable for interactive jobs. Another drawback is that a long-running process will delay all jobs behind it.

Round robin scheduling

Round robin scheduling is a preemptive version of first-come, first-served scheduling. Processes are dispatched in a first-in-first-out sequence but each process is allowed to run for only a limited amount of time. This time interval is known as a **time-slice** or **quantum**. If a process does not complete or get blocked because of an I/O operation within the time slice, the time slice expires and the process is preempted. process gets blocked because of an I/O operation), it is then preempted. This preempted process is placed at the back of the ready list where it must wait for the processes that were already on the list to cycle through the CPU.



With round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the ready list (run queue). A very long quantum makes the algorithm behave very much like first come, first served scheduling since it's very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

Advantage: Round robin scheduling is fair in that every process gets an equal share of the CPU. It is easy to implement and, if we know the number of processes on the ready list, we can know the worst-case response time for a process.

Disadvantage: Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

Setting the quantum size

What should the length of a quantum be to get "good" performance? A short quantum is good because it allows many processes to circulate through the processor quickly, each getting a brief chance to run. This way, highly interactive jobs that usually do not use up their quantum will not have to wait as long before they get the CPU again, hence improving interactive performance. On the other hand, a short quantum is bad because the operating system must perform a context switch whenever a process gets preempted. This is overhead: anything that the CPU does other than executing user code is essentially overhead. A short

quantum implies many such context switches per unit time, taking the CPU away from performing useful work (i.e., work on behalf of a process).

The overhead associated with a context switch can be expressed as:

$$\textit{context switch overhead} = C / (Q+C)$$

where Q is the length of the time-slice and C is the context switch time. An increase in Q increases efficiency but reduces average response time. As an example, suppose that there are ten processes ready to run, Q = 100 msec, and C = 5 msec. Process 0 (at the head of the ready queue) gets to run immediately. Process 1 can run only after Process 0's quantum expires (100 msec) and the context switch takes place (5 msec), so it starts to run at 105 msec. Likewise, process 2 can run only after another 105 msec. We can compute the amount of time that each process will be delayed and compare the delays between a small quantum (10 msec) and a long quantum (100 msec.):

Proc #	*Q = 100 msec. delay (msec.)*	*Q = 10 msec. delay (msec.)*
0	0	0
1	105	15
2	210	30
3	315	45
4	420	60
5	525	75
6	630	90
7	735	105
8	840	120
9	945	135

We can see that with a quantum of 100 msec and ten processes, a process at the end of the queue will have to wait almost a second before it gets a chance to run. This is much too slow for interactive tasks. When the quantum is reduced to 10 msec, the last process has to wait less than 1/7 second before it gets the CPU. The downside of this is that with a quantum that small, the context switch overhead (5/(10+5)) is $33\frac{1}{3}\%$. This means that we are wasting over a third of the CPU just switching processes! With a quantum of 100 msec, the context switch overhead is just 4%.

Shortest remaining time first scheduling

The **shortest remaining time first** scheduling algorithm is a preemptive version to an older non-preemptive algorithm known as **shortest job first** scheduling. Shortest job first scheduling runs a process to completion before running the next one. The queue of jobs is sorted by estimated job length, so that short programs get to run first and not be held up by long ones. This minimizes average response time.

Here's an extreme example. It's the 1950s and three users submit jobs (a deck of punched cards) to an operator. Two of the jobs are estimated to run for 3 minutes each while the third job while the third user estimates that it will take about 48 hours to run the program. With a shortest job first approach, the operator will run the three-minute jobs first and then let the computer spend time on the 48-hour job.

With the shortest remaining time first algorithm, we take into account the fact that a process runs as a series of CPU bursts: processes may leave the *running* state because they need to wait on I/O or because their quantum expired. The algorithm sorts the ready list by the the process' anticipated CPU burst time, picking the shortest burst time. Doing so will optimize the average response time of processes.

Here's an example of five processes in the ready list. If we process them in a FIFO

manner, we see that all the CPU bursts add up to 25 (pick your favorite time unit; this is just an example). The mean run time for a process, however, is the mean of all the run times, where the run time is *the time spent waiting to run + the CPU burst time of the process*. In this example, our mean run time is (8 + 11 + 21 + 23 + 25)/5, or 17.6.

Burst time	2	2	10	3	8	Total time = 25
Process	*E*	*D*	*C*	*B*	*A*	
Total run time	25	23	21	11	8	Mean time = 17.6

If we reorder the processes in the queue by the estimated CPU burst time, we still have the same overall total (the processes take the same amount of time to run) but the mean run time changes. It is now (2 + 4 + 7 + 15 + 25), or 10.6. We reduced the average run time for our processes by 40%!

Burst time	2	2	10	3	8	Total time = 25
Process	*E*	*D*	*C*	*B*	*A*	
Total run time	25	23	21	11	8	Mean time = 17.6

Estimating future CPU burst time

The biggest problem with sorting processes this way is that we're trying to optimize our schedule using data that we don't even have! We don't know what the CPU burst time will be for a process when it's next run. It might immediately request I/O or it might continue running for minutes (or until the expiration of its time slice).

The best that we can do is guess and try to predict the next CPU burst time by assuming that it will be related to past CPU bursts for that process. All interactive processes follow the following sequence of operations:

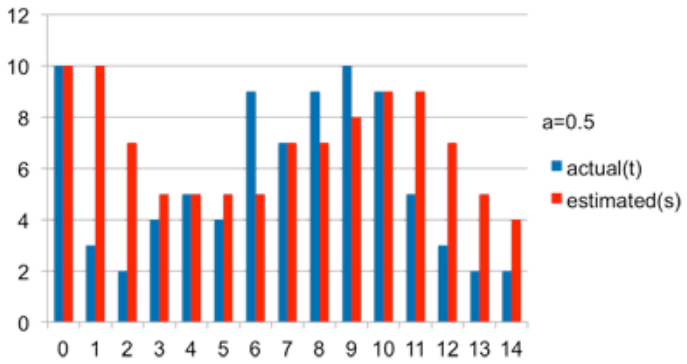
compute — I/O — compute — I/O — compute — I/O

Suppose that a CPU burst (compute) period is measured as T_0. The next compute period is measured as T_1, and so on. The common approach to estimate the length of the next CPU burst is by using a time-decayed **exponential average** of previous CPU bursts for the process. We will examine one such function, although there are variations on the theme. Let T_n be the measured time of the n^{th} burst; s_n be the predicted size of the n^{th} CPU burst; and a be a weighing factor, $0 \le a \le 1$. Define s_0 as some default system average burst time. The estimate of the next CPU burst period is:

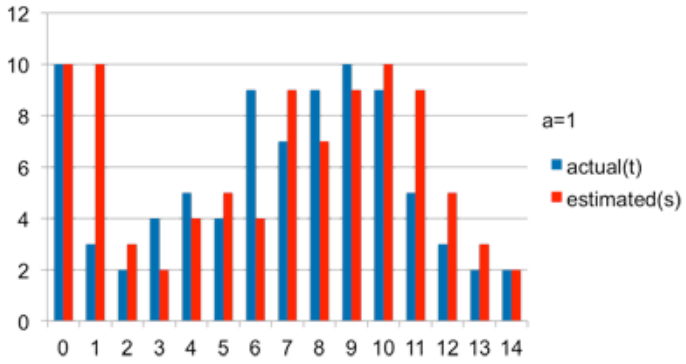
$$s_{n+1} = aT_n + (1 - a)s_n$$

The weighing factor, a, can be adjusted how much to weigh past history versus considering the last observation. If $a = 1$, then only the last observation of the CPU burst period counts. If $a = ½$, then the last observation has as much weight as the historical weight. As a gets smaller than ½, the historical weight counts more than the recent weight.

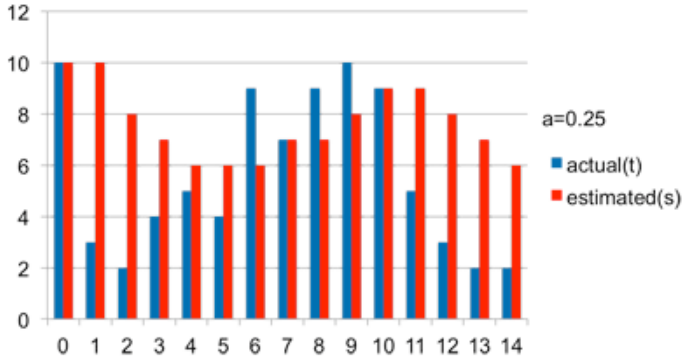
Here is an example with $a = 0.5$. The blue bars represent the actual CPU burst over time. The red bars represent the estimated value. With a weighting value of ½, we can see how the red bars are strongly influenced by the previous actual value but factor in the older values.



Now let's see what happens when we set $a = 1$. This ignores history and only looks at the previous CPU burst. We can see that each red bar (current estimate) has exactly the same value as the previous blue bar (latest actual CPU burst).



For a final example, let's set *a= 0.25*. Here, the last measured value only counts for 25% of the estimated CPU burst, with 75% being dictated by history. We can see how immediate changes in CPU burst have less impact on the estimate when compared with the above graph of *a = 0.5*. Note how the estimates at 2, 3, 13, and 14 still remain relatively high despite the rapid plunge of actual CPU burst values.



Advantage of shortest remaining time first scheduling: This scheduling always produces the lowest mean response time Processes with short CPU bursts run quickly (are scheduled frequently).

Disadvantages: Long-burst (CPU-intensive) processes are hurt with a long mean waiting time. In fact, if short-burst processes are always available to run, the long-burst ones may never get scheduled. Moreover, the effectiveness of meeting the scheduling criteria relies on our ability to estimate the CPU burst time.

Priority scheduling

Round robin scheduling assumes that all processes are equally important. This generally is not the case. We would sometimes like to see long CPU-intensive (non-interactive) processes get a lower priority than interactive processes. These processes, in turn, should get a lower priority than jobs that are critical to the operating system. In addition, different users may have different status. A system administrator's processes may rank above those of a student's. These goals led to the introduction of **priority scheduling**. The idea here is that each process is assigned a priority (just a number). Of all processes ready to run, the one with the highest priority gets to run next (there is no general agreement across operating systems whether a high number represents a high or low priority; UNIX-derived systems tend to use smaller numbers for high priorities while Microsoft systems tend to use higher numbers for high priorities).

With a priority scheduler, the scheduler simply picks the highest priority process to run. If the system uses preemptive scheduling, a process is preempted whenever a higher priority process is available in the ready list.

Priorities may be **internal** or **external**. **Internal priorities** are determined by the system using factors such as time limits, a process' memory requirements, its anticipated I/O to CPU ratio, and any other system-related factors. **External priorities** are assigned by administrators.

Priorities may also be **static** or **dynamic**. A process with a **static priority** keeps that

priority for the entire life of the process.

A process with a **dynamic priority** will have that priority changed by the scheduler during its course of execution. The scheduler would do this to achieve system goals. For example, the scheduler may decide to decrease a process' priority to allow a lower-priority job to run. If a process is I/O bound (spending most if its time waiting on I/O), the scheduler may give it a higher priority so hat it can get off the ready queue quickly and schedule another I/O operation.

Static and dynamic priorities can coexist. A scheduler would know that a process with a static priority cannot have its priority adjusted throughout the course of its execution.

Ignoring dynamic priorities, the priority scheduling algorithm is straightforward: each process has a priority number assigned to it and the scheduler simply picks the process with the highest priority.

Advantage: priority scheduling provides a good mechanism where the relative importance of each process may be precisely defined.

Disadvantage: If high priority processes use up a lot of CPU time, lower priority processes may starve and be postponed indefinitely. The situation where a process never gets scheduled to run is called **starvation**. Another problem with priority scheduling is deciding which process gets which priority level assigned to it.

Dealing with starvation

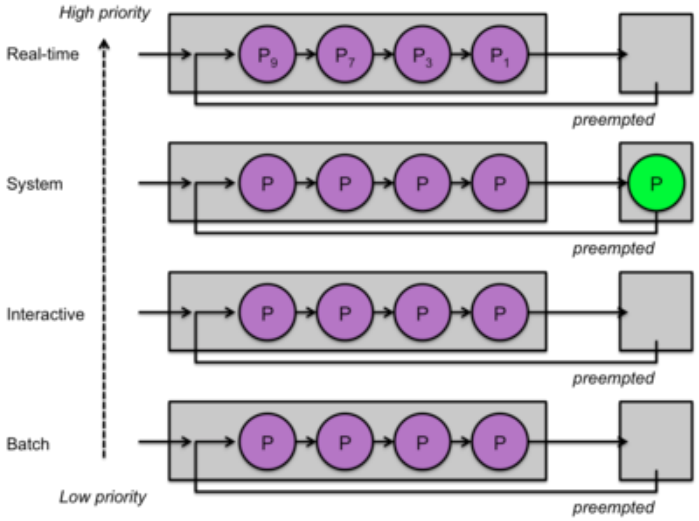
One approach to the problem of indefinite postponement is to use dynamic priorities. At the expiration of each quantum, the scheduler can decrease the priority of the current running process (thereby penalizing it for taking that much CPU time). Eventually its priority will fall below that of the next highest process and that process will be allowed to run.

Another approach is to have the scheduler keep track of low priority processes that do not get a chance to run and gradually increase their priority so that eventually the priority will be high enough so that the processes will get scheduled to run. Once it runs for its quantum, the priority can be brought back to the previous low level. This approach is called **process aging**.

Multilevel queues

What happens if several processes get assigned the same priority? This is a realistic possibility since picking a unique priority level for each of possibly hundreds or thousands of processes on a system may not be feasible.

We can group processes into **priority classes** and assign a separate run queue for each class. This allows us to categorize and separate system processes, interactive processes, low-priority interactive processes, and background non-interactive processes. The scheduler picks the highest-priority queue (class) that has at least one process in it. In this sense, it behaves like a priority scheduler. It then picks a process from the queue. Each queue may use a different scheduling algorithm, if desired (e.g., typically round robin) and a different quantum. For example, low-priority non-interactive processes could be given a longer quantum. They won't get to run as often since they are in a low priority queue but, when they do, the scheduler will let them run longer.



One problem with feedback queues is that the process needs to be assigned to the most suitable priority queue *a priori*. If a CPU-bound process is assigned to a short-quantum, high-priority queue, that's not optimal for either the process nor for overall throughput.

Multi-level queues are generally used as a top-level scheduling discipline to separate broad classes of processes, such as real-time, kernel threads, interactive, and background processes. Specific schedulers within each class determine which process gets to run within that class.

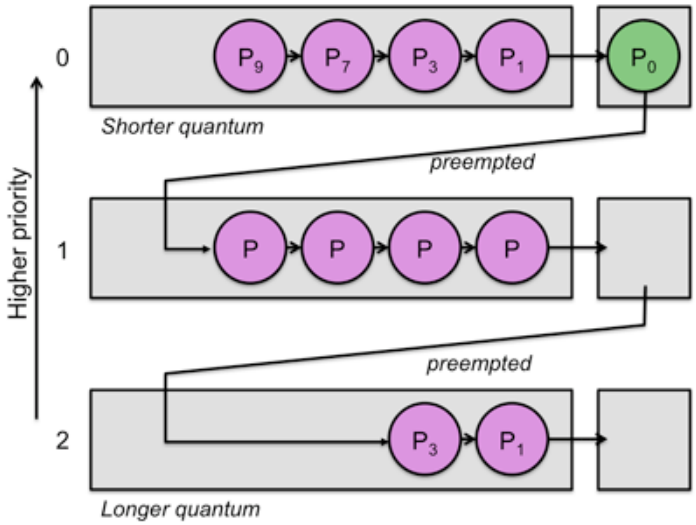
Multilevel feedback queues

A variation on multilevel queues is to allow the scheduler to adjust the priority of a process: to relocate it from one queue to another based on the behavior of the process. The goal of **multilevel feedback queues** is to automatically place processes into priority levels based on their CPU burst behavior. I/O-intensive processes will end up on higher priority queues and CPU-intensive processes will end up on low priority queues.

The key to the algorithm is that high priority queues have short time slices. The quantum length increases with each decreasing priority queue. A process initially starts in the highest priority queue. Once it gets scheduled to run, it will either run and then block on some event or it will run until the quantum expires. If the process blocks then the scheduler will move the process onto the same priority queue when it is ready to run again. If, however, the process runs to expiration (i.e., the scheduler sees that the process is still running when its quantum expires) then the scheduler preempts the process and places it in the queue at the next lower priority level.

As long as a process uses the full quantum at each level, it continues to drop to lower priority levels (eventually reaching the lowest level, where it circulates round robin). At each lower priority level, the quantum is increased (so the process does not get the CPU often, but gets to use more of it when it does get it). The rationale for this is that a processes at lower levels have demonstrated that they are compute bound rather than I/O bound. By giving them a fatter chunk of the CPU, but only when there are no higher priority processes to run, we can minimize the context switch overhead and get more efficient use of the CPU.

A process that waits too long to be scheduled may be moved up to a higher-priority queue (process aging). While it's common for each queue to circulate processes round-robin, as with multilevel queues, each queue may have its own scheduling algorithm to determine the order in which the scheduler removes a process from the queue.



Multi-level feedback queues work well in favoring I/O bound processes with frequent scheduling by keeping them at high priorities. What would happen if an I/O intensive process such as a text editor first needed to perform some CPU-intensive work (such as initializing a lot of structures and tables)? Although it would get scheduled at a high priority initially, it would find itself using up its quantum and demoted to a lower level. There too, it may not be done with its tasks and be demoted to yet lower levels before it blocks in I/O. Now it will be doomed to run at a low priority level, providing inadequate interactive performance. Some systems, such as MIT's Compatible Time-Sharing System (CTSS), dealt with this problem by raising the priority of a process whenever it blocked on I/O. Process aging can also rescue an interactive process that had a period of CPU-intensive activity and was therefore "punished" down to a lower priority queue. A drawback with the multi-level feedback queue approach is the users could game the system by inserting useless I/O operations into their programs just to fool the scheduler.

Advantages: Multi-level feedback queues are good for separating processes into categories based on their need for a CPU. They favor I/O bound processes by letting them run often. Versions of this scheduling policy that increase the quantum at lower priority levels also favor CPU bound processes by giving them a larger chunk of CPU time when they are allowed to run.

Disadvantages: The priority scheme here is one that is controlled by the system rather than by the administrator or users. A process is deemed important not because it is, but because it happens to do a lot of I/O. This scheduler also has the drawback that I/O bound processes that become CPU bound or CPU bound processes that become I/O bound will not get scheduled well.

Guaranteed scheduling

Guaranteed scheduling is a scheduling algorithm that attempts to be fair to the users, *not* the processes. In its simplest form, if *n* users are logged in, it will try to give each user (*not process*) *1/n* of the CPU time. What the scheduler does is keep track of how much CPU time a user has had for all processes since login and how long the user has been logged in. It then computes the user's share of the CPU as:

User's share of CPU = (time since login / n)

The actual share of the CPU is also known. The scheduler compares that (the actual) to the ideal (computed) share and picks the process with the lowest ratio. As it continues to be scheduled, its the actual:ideal ratio will increase and eventually move ahead of its nearest competitor, causing the next process to get scheduling priority.

Another form of guaranteed scheduling is to assign groups of users into a several classes and give each class a predefined share of the CPU. For example, suppose three departments jointly own a computer. They may set up the scheduler such that any processes from users in department A will get no more than 20% of the CPU, users in department B will get no more

than 50% and users in department C will get no more than 30% of the CPU time. A similar mechanism was used under VM (IBM's virtual machine system) in which each operating system running under VM would be guaranteed a certain percentage of the CPU — no more and no less.

Advantage: Guaranteed scheduling is fair to the user. It penalizes those users who wish to run many processes and favors those who run only a few. The shares of CPU time are allocated by the administrator and fit well with other bureaucratic processes, such as the purchase and upkeep of the computer.

Disadvantage: Guaranteed scheduling requires quite a bit of bookkeeping. While it's fair to the user, it's not fair to processes or the CPU. A user that is getting a large share of the CPU may never need it, but will get it anyway. Meanwhile, other processes may be hungry to use the CPU.

Multiprocessors

Our discussions thus far assumed an environment where a single process gets to run at a time. Other ready processes wait until the scheduler switches their context in and gives them a chance to run. With multiple CPUs, multiple cores on one CPU, hyperthreaded processors, more than once thread of execution can be scheduled at a time. The same scheduling algorithms apply; the scheduler simply allows more than one process to be in the running state at one time.

The envionment we will consider here is the common symmetric multiprocessing (SMP) one, where each processor has access to the same memory and devices. Each processor is running its own process and may, at any time, invoke a system call, terminate, or be interrupted with a timer interrupt. The scheduler executes on that processor and decides which process should be context switched to run. It is common for the operating system to maintain one run queue per processor.

Processors are designed with cache memory that holds frequently-used regions of memory that processes accessed. This avoids the time of going out to the external memory bus to access memory and is a huge boon to performance. As we will see in our forthcoming discussion on memory management, processors also contain a translation lookaside buffer, or TLB, that stores recent virtual to physical address translations. This also speeds up memory access dramatically.

When a scheduler reschedules a process onto the same processor, there's a chance that some of the cached memory and TLB lines are still present. This allows the process to run faster since it will make less references to main memory. If a scheduler reschedules a process onto a different processor then no part of the process will be present in that processor's cache and the program will start slowly as it populates its cache.

Processor affinity is the aspect of scheduling on a multiprocessor system where the scheduler keeps track of what processor the process ran on previously and attempts to reschedule the process onto that same processor. There are two forms of processor affinity. **Hard affinity** ensures that a process always gets scheduled onto the same processor. **Soft affinity** is a best-effort approach. A scheduler will attempt to schedule a process onto the same CPU but in some cases may move the process onto a different processor. The reason for doing this is that, even though there may be an initial performance penalty to start a process on another CPU, it's probably better than having the CPU sit idle with no process to run. The scheduler tries to **load balance** the CPUs to make sure that they have sufficient tasks in their run queue. There are two approaches to load balancing among processors:

1. **Push migration** is where the operating system checks the load (number of processes in the run queue) on each processor periodically. If there's an imbalance, some processes will be moved from one processor onto another.
2. **Pull migration** is where a scheduler finds that there are no more processes in the run queue for the processor. In this case, it raids another processor's run queue and transfers a process onto its own queue so it will have something to run.

It's common to combine both push and pull migration (Linux does it, for example).

References

- Inside the Linux scheduler (http://www.ibm.com/developerworks/linux/library/l-scheduler/), M. Tim Jones, IBM developerWorks Technical Library, June 30, 2006
- Understanding the Linux Kernel (http://oreilly.com/catalog/linuxkernel/chapter/ch10.html), Daniel P. Bovet & Marco Cesati, October 2000, Chapter 10: Process Scheduling
- Inside the Linux 2.6 Completely Fair Scheduler: Providing fair access to CPUs since 2.6.23 (http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/), M. Tim Jones, IBM developerWorks, December 15, 2009
- Multiprocessing with the Completely Fair Scheduler (http://www.ibm.com/developerworks/linux/library/l-cfs/), Introducing the CFS for Linux. Avinesh Kumar, IBM developerWorks Technical Library, June 8 2008
- Completely Fair Scheduler (http://en.wikipedia.org/wiki/Completely_Fair_Scheduler), Wikipedia article
- Linux: The Completely Fair Scheduler (http://kerneltrap.org/node/8059), Kernel Trap, April 2007.
