

Processes

1 Process Status and States

A process is the OS's abstraction for a running program. The OS needs to be able to suspend, resume, and swap out processes at will. So each process has a status record associated with it that informs the OS of its current state. A simplified view of process states is depicted in the following diagram:

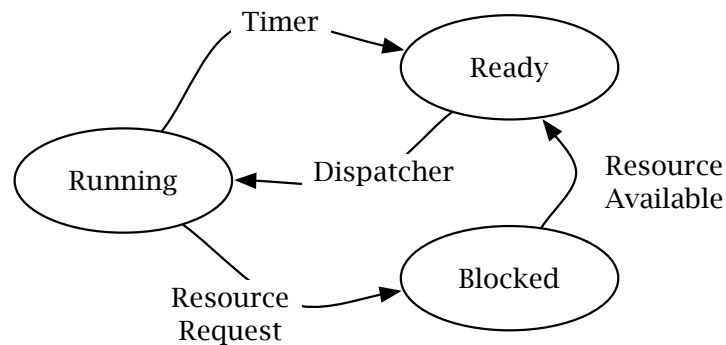


Figure 1: Simplified Process State Transitions

As you can see, there are basically three states a process can be in. Either it's actually **running** on the CPU, it's **ready** to run and just waiting to be scheduled, or it's **blocked** and suspended until the needed resource is available. A running process is one that is actually running on the CPU. If the running process makes a blocking request (for instance, to read a file from the disk), then it is suspended and transitioned into the blocking state (and a ready process is swapped in to run on the processor). If a running process exhausts its time-slice (the timer interrupt fires and activates the OS), then the OS suspends the process and puts it in the ready state (because the process can still run, it just ran out of time). Ready processes can only become running processes, and this happens when the OS scheduler needs to swap out the currently running process. That process is suspended and a ready process is selected, loaded onto the CPU and started up again. This selection process is another instance of scheduling. And just like disk request scheduling, process scheduling is a tricky balance between efficiency, throughput and fairness. A blocked process can only become ready, and this happens when the action that the process was blocked on completes. This normally means that the requested resource has finally become available.

1.1 Why Swap?

There are several reasons for swapping. As we've seen, it is often more efficient to be able to swap out a blocking process for one that is ready to run. Additionally, on timesharing systems,

to maintain the illusion of exclusive access the OS needs to regularly service all attached users. Lastly, sometimes the OS needs to do some housekeeping and regularly swapping out processes when the timer goes off gives the OS regular intervals in which to perform system maintenance tasks.

1.2 Process Creation

How are processes created? In general, a process is created by another process. When you type in the command `emacs foo.c` at the command line, your shell process **spawns** another process that executes the `emacs` binary with the argument `foo.c`. In this case, `emacs` is the **child** process and your shell is the **parent** process. Similarly, when you double-click on an icon, the windowing system would spawn a process to run the program associated with that icon. So now we see how your GUI or shell can create new processes. But who created your shell process? The **login** process that was responsible for authenticating you as a valid user on the system. But who spawned `login`? **init** did. Chances are, unless you've been in the UNIX sysadmin trenches, you've never heard of `init`. `init` is basically the primordial process, the first one spawned by the system and the process responsible for spawning all of the system daemon processes. These are also often called system services or servers, they're things like web servers, remote login servers, printer servers, etc. The UNIX kernel bootstraps the whole process creation cycle by instantiating `init` right off the bat.

1.2.1 Parent and Child Processes

Because it takes a process to make a process (unless you're `init`), there is a parent-child relationship between processes (similar to the parent child relationship between nodes in a binary tree). Parent processes can use system services to wait for their child processes to terminate, or they can continue on their merry way. Because UNIX allows parents to wait for their children to exit, there is the concept of a **zombie** process. Consider the following situation: a process spawns another process. The parent process continues executing (i.e. it isn't waiting, yet), and the child process does its thing and then exits. If the OS cleans up the child process right away, what happens if the parent process issues a `wait` call sometime in the future? Its child has exited, but since it wasn't waiting at the time, the parent will miss the signal and may block indefinitely. This is bad. So, on Linux systems, when a child process dies it is turned into a zombie. That way, if the parent issues a `wait` call before it exits it can tell if the child has finished. When a process exits, its zombie children are attached to its parent. This means that eventually all zombies become children of `init`. Zombies aren't inherently bad, they don't run on the processor, but they do consume some memory and fill up kernel data structures. So too many zombies can gum up the works. Therefore, `init` routinely purges its zombie children to keep the system relatively zombie free (like a good zombie movie).

1.2.2 fork and exec

In UNIX, process creation is divided into two parts. First, the parent process creates a copy of itself using `fork`. This is useful if you want to create multiple copies of a process in order to split

up work (e.g. apache spawning copies of itself to share server load). But, what if you want to run a completely different program? Then you use `exec` to start a completely different program. Note that this copy-then-execute model is peculiar to UNIX. On Windows systems, process creation always runs a new program (if you want a copy, you hand the same path to the process creation function). An advantage of the fork-and-exec model is that it makes it easy to run multiple copies of the same program, and in the days before multi-threading, that was how you parallelized server code. Additionally, with a virtual memory trick known as copy-on-write, forking can be made fairly cheap (by avoiding unnecessary copying of the parent's address space).

1.3 Seeing Processes in UNIX

Linux (and UNIX in general) is very process-centric. To see a list of currently running processes, you use the `ps` command. Each process is identified by a unique process ID (or **PID**) assigned by the OS at process creation. `ps` can list the processes and display their resource consumption (how much memory, CPU, etc. is being used). `top` is a tool for displaying and sorting the top resource-using processes. Depending on the size of your terminal `top` will display 20 or so processes. `pstree` lists the processes, but organizes them into a relationship tree that lets you see which processes created other processes.

2 Interrupts

The kernel is invoked by interrupts, and there are two kinds. Hardware interrupts come from devices (like the timer or the harddrive) or from error conditions (divide-by-zero), while software interrupts are triggered by code executing on the CPU. Hardware interrupts are communicated from the devices to the CPU via the bus. Software interrupts are caused by a specific machine instruction (`INT` on intel). From an assembly-programming point of view, software interrupts (also known as system calls/syscalls or traps) are very similar to function calls. The calling program is responsible for setting up the syscall arguments in registers or on the stack and then invoking the system call instruction.

2.1 Interrupt Handling

Interrupts are handled in two phases. The first phase is purely in **hardware** and involves saving some processor context and switching into supervisor mode. The second phase is in **software**, and involves executing the kernel code specific to the interrupt that was triggered. The hardware phase begins by switching to some pre-designated storage. Some processors have special registers set aside for this, but on x86 machines the interrupt causes the processor to switch to an alternate stack (specified ahead of time by the kernel). Once a special storage area is available, the processor can backup key registers (PC, SP, etc.) that are in danger of being overwritten. Then control is passed to the interrupt handler code (i.e. the PC is set to point to the interrupt code). Now the interrupt is in the software phase. Usually, the first step is to back up additional registers that will be overwritten (this is especially important on the register-starved x86). Then, some book-keeping has to occur

so that the system knows what state things are in. For instance, a flag can be updated so that the scheduler knows that this process was interrupted. Then this generic start-up code will jump to the handling code for this specific interrupt (the code identifying the interrupt will be either read off the bus or from one of the registers).

There are some tricky details. For instance, what do you do when another interrupt arrives while handling another interrupt? A simple solution is just disabling interrupts. Most processors have a supervisor-level instruction that lets you turn off the CPU's acknowledgment of interrupts. Turning that off lets you ensure that no interrupts will be triggered during interrupt processing. This is bad. On a uniprocessor system, this can lead to devices waiting longer. Another trick is to have a priority associated with the interrupt, lower priority interrupts have to wait while higher priority interrupts can preempt the currently executing interrupt. This has been available in the x86 line of processors since the Pentium. Because of this, Linux requires that handlers and device drivers be **reentrant**. This is fancy language for code that can be run several times simultaneously. Essentially, this is thread-safe code. Reentrancy means that the handlers themselves can't interfere with other invocations, this usually boils down to not modifying and depending on global data (that may get corrupted by other executions, e.g. race conditions). If your handlers are reentrant, then if a handler gets preempted mid-way, you can be certain that your code will act consistently.

3 Scheduling Theory

Whenever you have a scarce resource with multiple demands on it, you have a situation where you may need to schedule. With Operating Systems, the hardware resources are scarce and the processes wanting to use them must be scheduled to make fair and effective use of the hardware. We've already examined some scheduling policies for hard drives and now we're going to focus on CPU scheduling. As an analogy, consider a water fountain line for preschoolers. There's only one water fountain (CPU) and the teacher (OS) has to make sure that each kid (process) gets a drink of water during recess. Some kids will want to drink more than others, so the teacher may have to stop the kid and let other children drink (preemption).

3.1 Criteria

Any scheduling policy is going to have to make trade-offs to maximize some criteria. Here's a (by no means exhaustive) list.

- **Fairness** The OS should share resources equally among processes
- **Efficiency** The OS should maximize CPU utilization
- **Response Time** The OS should ensure low-latency for interactive processes (users)
- **Turnaround** The OS should ensure low-latency for batch processes
- **Throughput** The OS should maximize the number of processes per second

- **Priority** The OS should dedicate more resources to ‘important’ processes
- **Realtime** The OS should meet specified deadlines

Not all these criteria are mutually compatible, in general: real-time schedulers emphasize deadlines above all else; batch schedulers are primarily concerned with throughput; and schedulers for interactive systems tend to focus on response time.

4 Scheduling Policies

An OS designer chooses a scheduling policy based upon the expected use of the system. There are two big classes of policies: non-preemptive policies cannot interrupt a running process, while preemptive policies can.

4.1 Non-preemptive policies

- **FCFS** The bank-queue policy, jobs are processed in the order they arrive
- **SJN** Shortest Job Next, the scheduler runs the jobs with the shortest expected time to completion. This can starve long-running jobs (and is therefore not fair)
- **Priority** Jobs have a numerical priority attached that indicates their relative importance. A priority scheduler executes higher priority processes before low priority ones. This can starve low priority processes (and is therefore not fair)
- **Deadline** Popular in real-time applications, a deadline scheduler guarantees an execution rate by a deadline (e.g. process X needs 10ms of processing time in the next 100ms). This requires foreknowledge of the processing requirements for all processes in the system.

4.2 Preemptive Policies

- **Round Robin** Simple and fair, all ready processes run in sequence. The OS allocates time-slices equally (e.g. if there are N processes in the system and M time slices per second, each process will get $\frac{M}{N}$ time-slices per second).
- **Preemptive Priority** Similar to non-preemptive priority, but the OS periodically interrupts processes to check for higher priority ready processes.
- **Lottery** ‘Tickets’ are issues to processes according to their priority (higher priority = more tickets). Each quantum, the OS generates a random number (the winning number), and the process holding that ticket gets to run for the quantum. On average, this will distribute CPU time according to priority. However, now the kernel has to generate all kinds of random numbers, which increases scheduling overhead.

- **Multiple Queues** Each priority/rank has its own queue, the dispatcher selects the highest priority queue with runnable processes and then selects a process in that queue. Multiple Queues get interesting when you add feedback, which is using the processes' behavior to assign it to a different queue (maybe higher, maybe lower) when it is being swapped out (this variation is often called **multilevel feedback queues**). This is usually considered the 'standard' UNIX scheduling policy.

5 Linux Scheduler

The Linux $O(1)$ scheduler is a highly-engineered version of multilevel feedback queues. The Linux scheduler has two sets 140 queues! One set is the **active** set and the other is the **expired** set. The 140 queues are divided into 100 “realtime” and 40 user levels. When you start a process on the desktop or at the command line, you're running in one of those bottom 40 user levels. The user levels correspond to `nice` priority levels. When you start a process, it is by default at `nice` level 0 (which means that it's in queue number 120). Users can increase the nice level (up to 19) which actually lowers the priority. Only root can decrease the nice level.

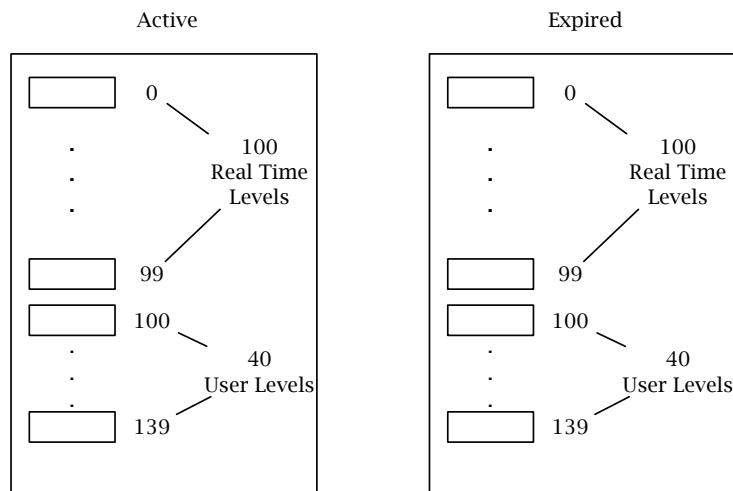


Figure 2: Linux O(1) Scheduler

Linux organizes CPU time into “epochs”. For each epoch, processes have pre-allocated timeslices (10-200ms). Of course, processes can under-use their timeslices, but the basic idea is that the task of allocating CPU time fairly occurs only during epoch switches, rather than every time the timer interrupt fires.

The process of actually scheduling and dispatching a process is roughly:

1. Highest active runqueue with ready processes is selected. A task is chosen from that queue (currently running tasks are favored).

2. When the process' timeslice is expired, it is moved to the expired list. It may not occupy the same level queue in the expired list as it did in the active list. The process' new priority is an adjustment from its static priority based on its behavior during this last epoch.
3. When the active queue empties, the active and expired queues are swapped.
4. On multiprocessing systems, every so often (200ms or so), the OS also checks the processing load across all processors and shifts ready processes around to help balance the load.

So, the Linux scheduler is similar to multi-level feedback queues. However the epoch system allows Linux to allocate variable-sized timeslices to processes, and to make that scheduling decision less frequently. Process priority is also variable. Each process has a static priority (the nice level for user processes). But the queue to which a process is assigned is based on dynamic priority, which is the static priority plus-or-minus some adjustments based on the process's interactive behavior. Essentially, when a process sleeps (i.e. makes a blocking call) its dynamic priority is boosted. When a process exhausts its timeslice, its dynamic priority is lowered. This means that processes like an interactive shell, will have very high priority (mostly they just wait for user input, or for child processes to terminate), so that they will get lots of little time-slices quite frequently. This is good for interactivity. However, this also means that any process that is I/O intensive will also have a high dynamic priority.

6 Windows Scheduling

Windows 2000 uses a simpler version of multilevel feedback queues. Windows does not use an epoch like system, so all scheduling decisions occur during process switching. Windows has 32 priority levels, 16 variable and 16 "real time". It is fully prioritized, so that all the realtime queues will be emptied before any variable queues will be visited. This can result in starvation. Windows time quanta can vary between 20 and 200ms (although server versions can have much longer quanta). Similarly to Linux, variable priorities can be dynamically adjusted based on past behavior. I/O interrupts boost variable priority, while timer interrupts decrease priority.

A big difference between windows and Linux is the integration of the GUI and the OS. Because the windows team and Microsoft can tightly control the GUI system, they use various user actions to supply the OS with 'hints'. For example, when you minimize an application on windows, that is treated as a 'hint' to lower that program's priority (additionally, that also 'hints' to the OS memory manager to swap out that application's memory to make room for higher priority processes).