Week 7: Process Lifecycle

Now we're finally talking about one of the key responsibilities of any OS: managing processes. There are two aspects to this: scheduling, and process lifecycle.

Scheduling deals with when to run different processes in the limited resources you have on your system. This includes figuring out what to run first, for how long, and on what processor. It's a complex decision process that requires a lot of trade-offs be made.

Process lifecycle is a related, but separate endeavour. It involves figuring out when a process should be running, whether to be scheduling it, whether it's about to die and what states can it switch between. Let's jump into this first.

The process lifecycle diagram

A diagram you'll often see describing this process is one like this:

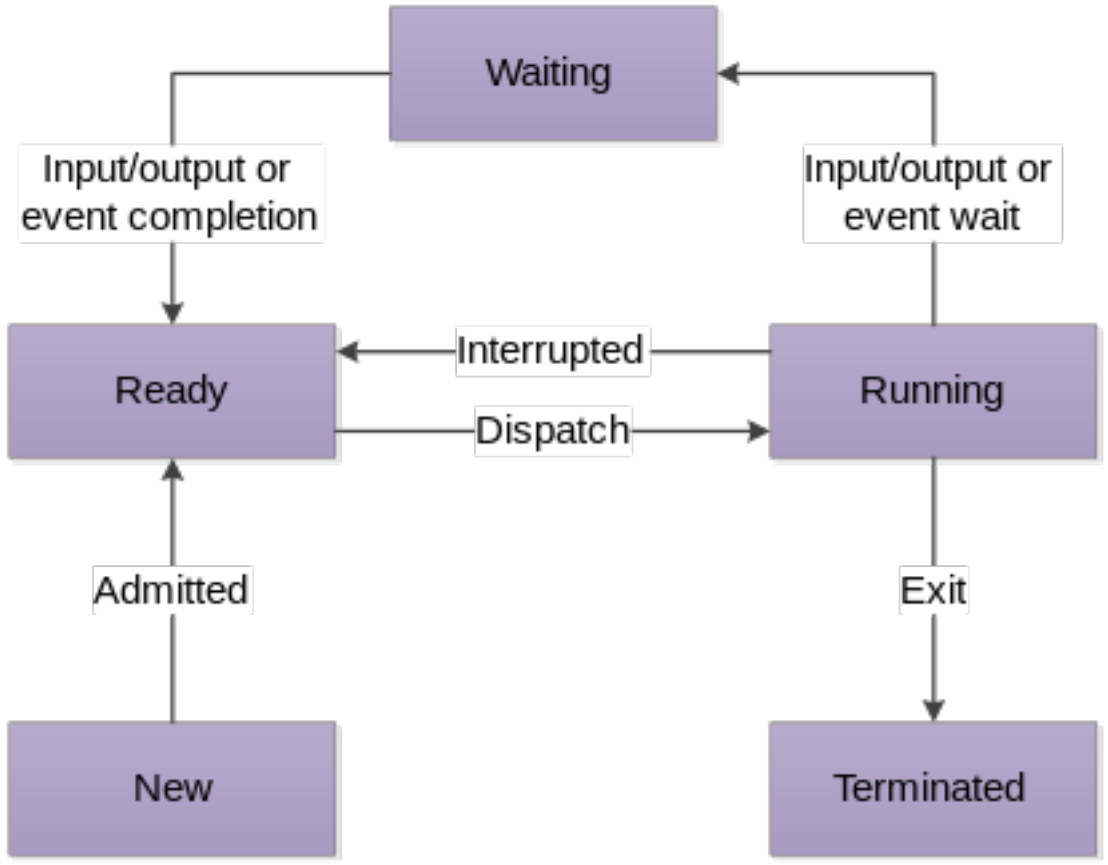

Figure 1: The process life cycle diagram, by StanleyWood

On the most part, you'll encounter these 5 states. When a new process is created, it starts without a state. Once Linux finishes setting up the object, it enters a "runnable" (here ready) state. Here it waits for the OS to schedule it, and when it does, it gets moved to a "running" state, where it stays, until either it gets pre-empted by the kernel to run something else (in

which case it gets moved back to the “runnable” state), or until it executes something that requires it to wait for a result. This will send it to the “waiting” state. An example of why you might get sent here is when you open a file. Opening a file often requires the OS to go to disk and read off a result from there. However, it takes a long time to tell the disk to fetch some piece of data, so while it waits, the process is sent to the “waiting” state. It then sits there until the kernel gets the response back (in the form of a hardware interrupt in this case), and then it moves it back to the “runnable” state, where it waits for it to continue execution before returning the result.

The final thing that can happen during execution is the process requests the OS to stop executing it, usually by the program ending, or by calling the `exit()` system call. This will let the kernel know that the process should not get scheduled again, and it’s sent to the STOPPED state.

Now, this covers a very basic overview of what happens to processes, but the actual diagram is a little bit more complex, and you can read more about it here. One interesting thing to note here is that there are a few extra states. Notably, instead of “waiting” we have interruptible and non-interruptible sleep, there’s a distinction between running in user vs kernel mode, and, most interestingly, there’s a “zombie” state. We’ll talk about that one for a bit. However, the diagram above mostly covers the behaviour you’ll observe in Linux.

Creating your own processes

Now, an interesting topic here is how you might go about creating new processes. There’s a handful of syscalls here that are extremely useful for this: `fork()` (nowadays implemented instead with the syscall `clone()`), `exec()` and `wait()`. These three syscalls serve three vital purposes:

- **`fork()` or `clone()`**: This syscall creates a new process, identical to the current one. If using `fork()`, it will create a copy of all the memory used by the process, and everything else about the process, but will assign it a new PID. The `clone()` syscall lets you pick what things to clone and what not to. You can read more details in `man 2 fork`, but it’s important to note that the way you know whether you’re in the child or parent depends on the return value of fork.
- **`exec()`**: In reality, these are a family of syscalls, all which replace the currently running process with whatever program you provide in. They also let you provide other relevant parameters, like the command-line arguments, and the environment, depending on which “flavour” of `exec*()` you use.
- **`wait()`**: When a process creates a child process, it is expected to read back the return value (often called exit code) of its child. There are two variations of this syscall, `wait()` which will wait on the first child process to die, and `waitpid()`, which waits on a child process with a given PID and can be configured to not return immediately if the child isn’t finished.

Each one of these should have left you with nagging questions, which we’ll address here.

Fork() and memory

One thing that should jump out is the statement ”it will create a copy of all the memory of the memory. As we discussed last week, this memory is virtual, paged memory so, in theory, you’d have to copy every single used virtual page over to a new physical page, so the two processes don’t start modifying the memory. However, this can be very expensive, so instead, we pull up a little trick out of our sleave: **copy-on-write**, abbreviated CoW, is a technique by which we

only copy the memory in pages if a write happens to that memory. That way, we only copy the pages we have to.

The implementation is quite ingenious. If you look back at the intel IA32/64 manual, on Table 4-6 in page 3A 4-13, you'll notice that one of the bits defines whether a given page can be written to. If you use that bit to mark the page as read-only, you'll get a page fault with every attempt to write to the page. You can use this so that, the moment a page that the kernel knows is copy-on-write is written to, the kernel creates a new copy, which it points one of the two processes to, and marks both page table entries as read+write again.

Why is this important? Well, let's remember, everything lives in memory. This includes your program's stack, the "heap" memory, static resources, and, most interestingly, the executable you're running. Since code loaded into memory generally doesn't get changed (though there are exceptions, that means that, unless you call `exec()`, you'll only have to keep one in-memory copy of the code that's executing for two binaries. Virtual memory lets you do tons of crazy memory-saving tricks and techniques.

The exec() family

What we've been referring to as the `exec()` syscall is, in reality, a family of syscalls and glibc wrappers that provide the same functionality in different ways. When you're replacing a program, you often want to specify the arguments you're passing in, the environment variables, and do all this in different forms. Notable forms include:

- `execl()`, which lets you pass in command-line arguments as arguments to the function, a "variadic function"
- `execle()`, which is like `execl()` but with an argument for the environment array
- `execv()`, which lets you pass in command-line arguments as a null-terminate array of null-terminated strings
- `execvp()`, like `execv()`, but it implements the search for the binary provided by looking at the directories in the `$PATH` environment variable, much like bash or any other shell would.

All these variations are just the result of selecting from 3 different things:

- If you want to provide an array of arguments, use the `execv*()` variation, and if you'd rather provide arguments as arguments to the function, use the `execl*()` variety
- If you want to have the function search for the binary in `$PATH` for you, append a `p`
- If you want to specify the environment (aka the environment variables), append an `e`

The wait() obligation

One thing we mentioned in regards to the `wait()` system call is that whenever we fork off a child process the parent must call the `wait()` system call on that PID. There's more than one way of doing this. The first is to simply call the `wait()` syscall the instant you fork and letting it stay waiting for the child process to die. However, oftentimes you want to be able to do other things while the child process is off doing its thing. For these cases, Linux provides us with its own guarantee: it'll send a `SIGCHLD` signal to the parent every time a child dies. You can then call `waitpid()` in the signal handler of `SIGCHLD` (using a PID of -1 which will wait on any dead child and the flag that prevents hangup so that it exists if there's no child to wait on), and fulfil your duty!

So, why is this duty important, and what happens if you don't call `wait()` or `waitpid()` for every process you've killed? Well, calling `wait()` or `waitpid()` returns the result for the

process, and since the kernel doesn't know if the former parent will eventually call `wait()`, and it needs to guarantee that said syscall always results in the expected output, regardless of the order between the death of the child and when `wait()` is called, it has to keep around all the metadata for the process (specifically, an entry in the "process table", a table containing all running processes) until either the parent dies or `wait()` is called. That said, everything else about the process can be freed, the memory, the open files, network sockets, etc. These are often called "zombie processes" since they're neither dead nor alive. If you take a look at the status of the process (either with something like `top` or the `/proc/<PID>/status` file), it'll be classified as "Zombie". The only issue that zombie processes cause is that, since Linux has a maximum PID number, they can take up precious PIDs that could have been freed up already.

Scheduling

Finally, we can move over to scheduling. I'll do my best to keep this section brief. One of the core functions of any OS kernel is to decide when to run what. One of the most important concepts to consider here is how we decide when to switch what's running. You could imagine a system where you schedule jobs (or processes), and they all execute on the machine's single-core, one by one, until each one finishes. This can be a useful scheduling mechanism, but it means that if a job is further down the queue it's not possible to bump it up and finish it earlier. You might be tempted to then switch this queue mechanism with something more akin to a priority queue, where you always run the most important jobs first. But then a question arises, what counts as a more important job? Should you run the highest priority job first always, even if we have a slightly lower priority job that will take a tiny of a fraction of that original job's execution time? What's the value of finishing early vs finishing first?

This is a very tough trade-off to make, and you could come up with one and a million ways to solve it. How do modern operating systems solve this? Simple, they sidestep it. Instead of having to decide what processes need to run first and letting them run to completion, the OS instead schedules every process on the system for a short period, often referred to as the *quantum* or *time slice*. It schedules a given process for a given period and, once that period is up, it schedules a different program based on the criteria of the scheduler.

The way it manages to get programs to run for a given period is quite interesting. Most modern CPUs provide instructions to let the kernel set up timers. When the timer ends, it'll fire off an interrupt, and in the interrupt handler for these timers, the kernel will run the scheduler, figuring out what was just running, what to run next, and updating the state of things.

As mentioned already, there's a lot of potential implementations of schedulers that could be used in Linux. There are round-robin schedulers, multilevel queue scheduler, amongst others, but the one used today by the Linux kernel is the completely fair scheduler. The algorithm for it is fairly complex, so feel free to give the Wikipedia article a read, but we won't dig into the details of the algorithm here. One of the interesting things to note is it doesn't use fixed-size time slices.

The context switch

One of the key concepts that we implied with the previous discussion is that of the context switch. Every time our time slice runs out, an interrupt gets sent to the processor, which stop whatever was running on the processor. However, we want to be able to continue execution wherever we left off, so for that, we perform a context switch. What this involves is saving all the state of the processor that would be overridden (aka all the registers) into memory, in a place we can recover the data once we're ready to resume execution. Then on the other end, we would load up the context of a different piece of code and continue executing that.

While the concept is quite simple, the effects are very important. Interrupts and the corresponding context switches are not cheap. They can take on the order of ~10ms. This might seem extremely short, but remember processors are executing billions of instructions per second. This can result in a significant cost, which is why the time slice is usually picked to be at a point where the overhead of context switches is not too significant.

Now, you don't usually have a lot of control over how often the kernel forces a context switch for scheduling another job (on the most part), but you do have control over a different kind of context switch: system calls. Every time you perform a system call, you incur the cost of a context switch, since the kernel works on a different stack, and you kick off a syscall using a context switch. This is why, if you can avoid performing a lot of syscalls and instead just do a handful, you can save a lot of time. An example of this is picking how many bytes to read at a given time when using the `read()` system call. The more you can read the better, as you reduce the number of context switches. You'll likely not have to go to disk with every call, as the kernel will probably cache a portion of the file into memory, but the context switches can rack up a lot of time.