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. We'll start with this

The process lifecycle diagram

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

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 out 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. Openning 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 here 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.

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 things to note here, is that there's a few extra states. Notably, instead of "waiting" we have interruptable and non-interruptable sleep, there's a distinction between running in user vs kernel mode, and, most interestingly, theres 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()` (now a days 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 basically 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()`**: These are actually a family of syscalls, all which replace the current 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’s 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,