Chapter 4

Processes

The process is a fundamental concept in operating systems. In this chapter we look at what a process is, the concept of an execution environment, how processes are created and destroyed, and how they interact with threads. In the next chapter we’ll look at scheduling, which is the process of picking which process to run.

4.1 Basic definitions

Definition 4.1 (Process). A process is the execution of a program on a computer with restricted rights.

Remarks:

- A process can be thought of as an instance of a program – there can be multiple processes in a computer executing different instances of the same program.

- A process combines execution – running the program – and protection. An operating system protects processes (and their data and resources) from other processes in the computer.

- A process is a resource principal: In terms of implementation, a process bundles a set of hardware and software resources together: some memory, some CPU cycles, file descriptors, etc.

- A process can be thought of as a virtualized machine executing a program, albeit a machine very different from the underlying hardware.

- Processes are typically named in the system using process identifiers, or PIDs.

- The complete software of a running computer system can be thought as the set of running processes plus the kernel.
4.2 Execution environment

Definition 4.2 (Execution environment). *The execution environment of a process is the (virtual) platform on which it executes: the virtual address space, available system calls, etc.*

Remarks:

- The execution environment of the kernel is purely defined by the machine hardware (in the absence of virtualization). In contrast, in a process it is defined by the user-mode processor architecture plus whatever the kernel chooses to present to the process, for example the virtual address space.

- The set of system calls constitutes the API to the kernel, but can also be thought of as *extensions* to the set of machine instructions the process can execute. A process can be thought of (and older texts on operating systems talk about this much more) as a *virtual machine* for executing the user’s program. This machine has a processor (or more than one processor, if the program is multi-threaded), a lot of memory (due to paging), etc.

- The process’s *virtual processor* (or processors) doesn’t have a simple relationship to the real processors that the OS kernel is managing. However, in UNIX and Windows, the process appears to have one or more virtual processors exclusively to itself. In practice, the OS is constantly preempting the process, running other processes, handling interrupts, etc. Because the process is always resumed exactly where it left off when the kernel was entered, the program running in the process behaves as if it had the processor entirely to itself and nothing else happened.

- Some OS designs go beyond this “resume” model. UNIX also has *signals* (which we will see in a later chapter) that are analogous to hardware interrupts, except that they are generated by the kernel (sometimes on behalf of another process) and delivered to the process in user space as an *upcall*.

- Other OSes are more radical: instead of resuming a process after the kernel has been entered, they always jump back into the process at a single, fixed address, and let the process’ own code at that address figure out how to resume from where it was. This mechanism is called *scheduler activations*, and they even available in new versions of Windows.

4.3 Process creation

How is a process created? There are a number of ways operating systems create processes, but they boil down to one of two models.
4.3. PROCESS CREATION

Definition 4.3 (Child, Parent). When a process creates (via the OS) another new process, the creating process is called the **parent** and the newly created process the **child**.

This creates a **process tree**: every process in the system has a parent (except one, the root of the tree).

Definition 4.4 (Spawn). An OS **spawns** a child process by creating it from scratch in a single operation, with a program specified by the parent.

Remarks:

- Unless you’re familiar with **fork()** (below), this is the obvious way to create a process.

- Windows creates processes by spawning using the **CreateProcess()** system call.

- The spawn operation has to explicitly specify everything about a process that needs to be fixed before it starts: what program it will run, with which arguments, what protection rights it will have, etc. This can be quite complex; on Windows **CreateProcess()** takes 10 arguments, two of which are pointers to more (optional) arguments.

Definition 4.5 (Fork). In UNIX, a **fork** operation creates a new child process as an exact copy of the calling parent.

Remarks:

- Since the child is an exact copy of the parent, one process calls **fork()**, but both parent and child return from it.

- The only difference between the parent and child processes is the return value from **fork()**: the parent gets the PID of the child it just created (or -1 upon error). In the child, the same invocation of **fork()** returns 0.

- In contrast to spawn, fork doesn’t need any arguments at all: the OS doesn’t need to know anything about the child except that it’s an exact copy of the parent.

- On the face of it, **fork()** is expensive: it involves copying the entire address space of the parent. This is, indeed, what it used to do, but in the chapter on virtual memory we will encounter a technique called **copy-on-write** which makes **fork()** much cheaper (though not free).

Definition 4.6 (Exec). In UNIX, an **exec** operation replaces the contents of the calling process with a new program, specified as a set of command-line arguments.
Remarks:

- `exec()` does not create a new process, instead it is the complement to `fork()`—without it, you could not run any new programs.

- `exec()` never returns (except if it fails), instead the new program starts where you might expect in `main()`.

- Splitting process creation in UNIX into `fork` and `exec` allows the programmer to change whatever features of the child process (e.g., protection rights, open files, etc.) themselves in the code between `fork()` returning and `exec()` being called, rather than having to specify all of this to a spawn call.

**Definition 4.7** (The initial process). *The initial process, often called init, is the first process to run as a program when a machine boots.*

Remarks:

- On UNIX, the first process naturally cannot be forked. Instead, it is constructed by the kernel, and given the PID of 0. PID 0 never runs a user program; in older UNIX versions it was called swapper. Instead, PID 0 calls the kernel entry point to `fork` to create `init`, which therefore has a PID of 1.

### 4.4 Process life cycle

What happens to a process after it has been created?

**Definition 4.8** (Process states). *Each process is said to be one of a set of states at any point in time. Running processes are actually executing code, either in kernel mode or in user space. Runnable (also called waiting or ready) processes can execute, but are not currently doing so. Blocked (also called asleep) processes are waiting for an event to occur (such as an I/O operation to finish, or a page fault to be serviced) before they can run.***

**Example 4.10** (UNIX process lifecycle). *Figure 4.9 shows a slightly simplified process state machine for UNIX. A running process moves between kernel and user space, but can stop running due to either blocking (being put to sleep) or some other process running instead (preemption).***

Remarks:

- A process which executes a blocking system call (like `read()` or `recv()`) enters the asleep state until the operation is ready to complete.

- A process running in user space must always first enter the kernel (via a system call, or an asynchronous trap) before it can change to another state.

- A process which exits (either voluntarily or otherwise) becomes a zombie.
Definition 4.11 (Zombie). A process which exits is generally not removed completely from the system, but enters a state between being alive and being deleted from the OS. A process in this state is a zombie.

Remarks:

- Zombies solve a problem. If a process which exited is completely deleted from the system, there is no record of it left. This means that the process’ parent cannot determine anything about it (such as its exit code). Instead, the exiting process hangs around until its parent asks for its exit status, and can then be safely deleted.
- In Unix, the operation for reading a process’ exit status (and thus deleting it) is called wait().

Definition 4.12 (Orphan). A process which is alive, but whose parent has exited, is an orphan.

Remarks:

- In Unix, orphans are adopted not by the parent’s parent, but by init, i.e. process ID 1.
- A key role of the init process is to “reap” orphan processes by calling wait.

4.5 Coroutines

Before we get into threads, it’s worth pointing out the variety of different ways of expressing concurrency and parallelism.
Definition 4.13 (Coroutines). A coroutine is a generalization of the concept of a subroutine. A coroutine can be entered at multiple times, at multiple points, and return multiple times. Programming with coroutines is sometimes referred to as cooperative multitasking.

Example 4.14. The classic simple use-case of coroutines is a pair of activities, one of which processes data created by the other:

Algorithm 4.15 Test-And-Set
1: inputs
2: q {A queue data structure}
3: coroutine producer:
4: loop
5: while q not full do
6:  i ← new item
7:  q.insert(i)
8: end while
9: yield to consume
10: end loop
11: end coroutine
12: coroutine consume:
13: loop
14: while q not empty do
15:  i ← q.remove()
16:  process(i)
17: end while
18: yield to produce
19: end loop
20: end coroutine

This requires no threads, merely a different form of control flow to what you may be used to. Coroutines are an old idea, and express concurrency (more than one thing going on at the same time) without parallelism (actual simultaneous execution of multiple tasks). They are the basis not only for subroutines (which are strictly nested coroutines) but also iterators, which you might have seen in C++ or Python, for example. However, coroutines are more general than these.

4.6 Threads

Early multiprocessing operating systems (like the original Unix) did not provide threads: each process was single-threaded. This was not a problem when most computers were single-processor machines and programming languages were relatively low-level. However, the model prevents a programmer from using threads as language abstractions to express concurrency in her program (as coroutines can), and also prevents her from exploiting parallel hardware such as multiple cores.

Definition 4.16 (User threads). User threads are implemented entirely within a user process (as a library or as part of the language runtime). They
are sometimes known as **lightweight processes**, but this latter term is a bit ambiguous.

**Remarks:**

- On a multiprocessor, user threads can be multiplex across multiple kernel threads (see below) to give a process true parallelism.

- Since user threads are context switched entirely in user space, they can be very fast.

- User threads can implement a “directed yield” to another thread, providing most of the functionality of coroutines.

- If a user thread is about to block (for example, when it is about to execute a system call which might block the whole process), it is typical for the thread library to intercept the call and turn it into a **non-blocking** variant so another user thread in the same process can run while the system call is serviced – otherwise, performance is severely impacted. Indeed, one use for user-level threads is as a convenient programming abstraction above non-blocking I/O primitives.

- This trick of intercepting blocking calls does not work, however, for unintentional synchronous processor exceptions, in particular page faults. A page fault on one user-level thread will block the entire process, since the kernel has no way of scheduling one of the other threads (or indeed being aware of them).

Where the OS has no support for threads whatsoever, user threads are an attractive option. However, most mainstream OSes today provide threads in the kernel.

**Definition 4.17** (Kernel threads). **Kernel threads** are implemented by the OS kernel directly, and appear as different virtual processors to the user process.

**Remarks:**

- This is the default model in, for example, Linux.

- Each thread is now scheduled by the kernel itself, which keeps track of which threads are part of which process.

- Whereas a process used to be a thread of execution in a virtual address space, in this model it becomes a non-empty set of threads which share a virtual address space and which might also be scheduled intelligently together.

- Each kernel thread can now block (including on page faults) without stopping other threads in the same process.

- The kernel is now more complicated, since it has to track the relation between threads, address spaces, and processes.

- Thread creation, and context-switching between threads, is slower since it requires the kernel to be entered.
Chapter Notes

Processes have been around for a long time; they were a well-established idea when Unix was created [RT73].

Scheduler activations were described as such by Anderson (he of the textbook) and others [ABLL92], though the Psyche operating system (presented in the same session of the same conference!) also used upcall-based dispatch [MSLM91], and the basic idea may have appeared earlier in IBM mainframes. While microkernel-based operating systems [Lie93, Lie95] tend to provide just threads and address spaces, exokernel-based systems [EKO95, LMB96], tend to use upcalls since thread-management is pushed into user space anyway, and the basic philosophy is to expose as much as possible to the user process (or its libraries, at least) [EK95]. This also is a natural fit for virtual machine monitors [BDGR97, BDR12], since the upcall is delivered as a “hardware” interrupt to the guest operating system.

Bibliography


