

Virtualization

Each peach-eater can eat their own *virtual* peach, even though there's only one *physical* peach.

The Base Abstraction: Processes

A *process* is a running program. As users, we often want to run multiple processes at the same time. To make this happen, the operating system must provide the illusion that there are many CPUs, so that each process can run on one.

This is done through *virtualization* – processes are stopped and started (*context switching*), sharing the CPU time at the cost of taking longer to run. Operating systems also have *policies* that determine exactly how decisions like scheduling are made.

Because a process is by definition a running program, we can understand what processes are by taking account of everything that happens during execution.

The *machine state* of a process is everything that the program can read or update while running – all the parts we're interested in when thinking about the process. Key components of the machine state are:

- *Memory*, which includes the instructions and all the data the program touches (its *address space*)
- *Registers*, part of the processor that are read from and written to over the course of program execution (including the *program counter*, *stack pointer*, and *frame pointer*)
- Various IO devices, like persistent storage

The Process API

In any implementation, the OS needs to provide a basic interface for interacting with processes, supporting operations:

- *Create*
- *Destroy*
- *Status*
- *Wait*, which can allow one process to wait for another to terminate
- Miscellaneous control, including support for *suspending* and *resuming* processes

Process Creation

Process creation involves taking bytes on disk that specify instructions and turning them into a running program. First, data is *loaded* into memory,

including all of its static data. On modern operating systems, this is done *lazily*. Next, memory is allocated for the *stack*, possibly initialized with values (eg. C's `argc` and `argv`). *Heap* data might also be allocated, and depending on the OS there might be additional initialization tasks (like opening file descriptors). Finally, execution begins from `main`, and control is transferred to the CPU to begin execution.

Process States

Processes have three high-level states:

- Running – executing instructions
- Ready – ready to run but not currently being executed
- Blocked – not ready to run until some other event takes place (for example, waiting for an IO request to resolve)

Transitions between ready and running (*scheduled* vs *descheduled*) happen at the discretion of the OS based on decisions of the *scheduler*.

Data Structures

A few key data structures are used to keep track of state in typical operating systems. For example, a *process list* tracks process state. This includes the *register context* (register contents) of each stopped process, process states, etc.

Aside: The Unix Process API

The core of the Unix process API is three functions: `fork()`, `exec()`, and `wait()`.

Fork

`fork()` can be called by a process to create a clone (exact copy) of the calling process. The function returns twice, with value 0 in the child process and a non-zero positive process id of the child in the calling parent. This process is *nondeterministic* – the parent might run and terminate before the child, or vice-versa.

Wait

`wait()` can be used to have the parent block until the child finishes executing.

Exec

`exec()` is used to run a program that is different from the calling program (which we want to do basically always). The call accepts as arguments the name of an executable (like `wc`) and some arguments, and then *transforms* the currently running process into the one specified by the arguments. Thus, successful calls to `exec` never return, so code after `exec` isn't run.

Motivation

Motivation for the API can be understood by considering how it is more useful than alternatives in practice. For example, having distinct `fork()` and `exec()` calls makes it possible for a shell to modify the process environment before running a program. For example, shell redirection might involve setting up some file descriptors before transferring control to the running process.

Process Control and Users

Beyond the main three functions, Unix also provides interfaces for interacting with processes. For example, `kill()` is used to send *signals* to processes including `SIGINT` (interrupt) which generally terminates process and `SIGSTOP` which typically suspends them. Process can also “catch” signals with `signal()`, allowing arbitrary code to be run in response to them.

To remain sound alongside core OS goals like security and isolation, signals can't be sent arbitrarily. If this were the case, a rogue process could terminate all others on the system. Instead, the OS maintains a notion of a user, who must log in, and can then send signals to processes they have started, but not those started by other users on the machine.

Mechanism: Limited Direct Execution

CPU virtualization is achieved through *time sharing* – a process is run for a bit, then swapped out, etc. The OS must do this while maintaining high *performance* and full *control*.

Limited Direct Execution

To avoid paying a high performance penalty, user programs are run directly on the CPU. However, this poses two challenges. First, the OS must prevent the process from doing bad things. Second, in order to implement *time sharing* the OS must have some way to take back control.

Problem 1: Restricted Operations

If operations weren't restricted, a user process could do all sorts of malicious things like reading or writing anywhere on disk. To prevent this, a new processor mode called *user mode* is introduced which isn't allowed to do things like make IO requests without raising an exception (as distinct from *kernel mode* which is unrestricted).

When user processes need to perform privileged operations, they issue a *system call*. With this system call, the user includes a system call number (among the values supported by the OS). In response, the OS executes a *trap* instruction, jumping into the kernel and raising the privilege level and conditionally executing the call. The kernel refers to a *trap table* to tell which code to execute (i.e. which *trap handler*) based on the exception that was raised. This table is initialized with the OS boots up and remains constant until the program is rebooted.

After, a *return from trap* instruction is executed, reducing the privilege level and returning control to the user process. Logistically, this involves saving the caller's registers so that the program can return to normal execution after completion.

Restricted operations are thus supported by setup at boot-time along with having two privilege levels.

Problem 2: Switching Between Processes

Because the OS cedes control in the direct execution model, it isn't obvious how it will be able to take it back. Early systems attempted a *cooperative approach*, where the OS trusts processes to give up control. This obviously is problematic because it can't deal with malicious or problematic programs that don't cede control.

Again, hardware support is used in the form of a *timer interrupt*. At configured (at boot-time) intervals, a timer will go off, transferring control to an interrupt handler and thus granting control to the OS.

After a timer interrupt occurs, the OS *scheduler* decides whether to cede control back to that running process to switch to a new one. *Context switching* requires saving registers and restoring those from the newly-running process.

All this is complicated by the fact that an interrupt could occur during the processing of a system call, for example. Solutions to this sort of problem might involve locking, or temporarily disabling interrupts.