4. The Abstraction: The Process

Ryan Liu

July 8, 2025

A process is an instance of a program. The program itself is just text on the disk.

If a program is a fitness program, a process is a workout

The main problem to solve is providing the illusion of many CPUs, one to each process, with only a few physical CPUs available. The OS achieves this by **virtualizing** the CPU — by running one process, then stopping it and running another, etc. (**time sharing** the CPU).

The cost is performance; say n processes run round-robin, each is now slowed by a factor of n if they are all CPU bound.

To implement virtualization of the CPU, the OS utilizes low-level machinery (**mechanisms**, e.g. **context switch**) as well as high-level intelligence.

Policies are algorithms by which the CPU decides things,including how to time-share the CPU and space-share memory.

4.1 The Abstraction: A Process

To understand what constitutes a process, we have to understand its **machine state**: what a program can read or update when it is running. Much like how my sense of self extends not only to my body, but my place in the world.

One component is obviously memory. Instructions, data read/written to, sit in memory. The process' **address space** is the memory the process can address.

Some registers are special. The **program counter (PC) / instruction pointer (IP)** points to the next instruction to execute. The **stack pointer** and **frame pointer** manage the stack for function parameters, local variables, and return addresses. Programs often access persistent storage devices too; I/O info might include a list of files the process currently has open.

4.2 Process API

These APIs, in some form, are available on any modern OS.

- **Create:** Method to create new processes
- **Destroy:** Method to destroy new processes (many processes will exit by themselves when complete)
- **Wait:** uh yeah
- **Miscellaneous Control:** Sometimes other than killing or waiting for a process there are other controls possible (suspend/sleep)
- **Status:** there are usually interfaces to get some metainfo abt a process (runtime, state, etc.)

4.3 Process Creation: A Little More Detail

How are programs transformed into processes?

First, to run a program, the OS loads its coad and any static data (e.g. initializaed variables) into memory, into the process' address space. Programs initially sit on LTS in an **executable format**.

eager loading: load all at once before running (archaic)

lazy loading: load pieces of code only as demanded (think lazy seg tree pushdown operation)

Think **paging** and **swapping**, common theme in systems with cache hierarchy, covered in later chapters (like database buffer pool)

After loading into memory, the OS allocates the program's **run-time stack** or just **stack**; in C/C++, this is where local variables, function parameters, and return addresses are stored.

On UNIX based systems, the stack is typically 8MB by default; on Windows, 1MB.

C programs use the stack for local variables, function parameters, and return addresses. The OS allocates the memory, and initializes the stack with arguments (argc, argv).

The OS may also create some initial memory for the **heap**, whose size is limited by available disk space and virtual address space. Programs can allocate on the heap by calling malloc(), and free it with free. When the program rums out of heap memory, the OS must allocate more.

The OS also does some other initialization, particularly for I/O. In UNIX systems, each process, by default, has three open **file descriptors**, for stdio and stderr. This will be further explored in part 3.

After all this the OS starts the program running at the entry point, in main()

4.4