Processes and Threads

Written by Alan Ton.

When you're on a computer, you might have 13+ tabs of Google Chrome running simultaneously with Spotify and Slack. With limited hardware resources and capabilities, how does the operating system ensure that our computer doesn't explode with everything that's going on? To answer this, let's explore processes and threads.

Every process needs a program. Simply put, a program is a file resulting from compiling higher-level code into machine instructions. This file is called an *executable* or *binary* and is typically stored in Executable and Linkable Format (ELF). To run a program, we simply need to feed the instructions stored in the *executable* to the processor.

(Sidenote: The story above depicts programs written in compiled languages (e.g. C, Rust) processed in a UNIX environment. It is different for interpreted languages (e.g. Python, Ruby) and programs run on Windows. In addition, I left out the details on how libraries are linked and how the executable is loaded into memory.)

A process is a running instance of a program. Each process is only allowed to access to a subset of the computer's total memory for security purposes. The set of a process's accessible memory addresses is called the address space. Since each process has its own address space, other processes cannot tamper with another's memory. Therefore, processes are said to encapsulate isolation. Virtual memory and interprocess communication (IPC) are discussions for later. Depicted below is the layout of the virtual address space. It consists of the following components:

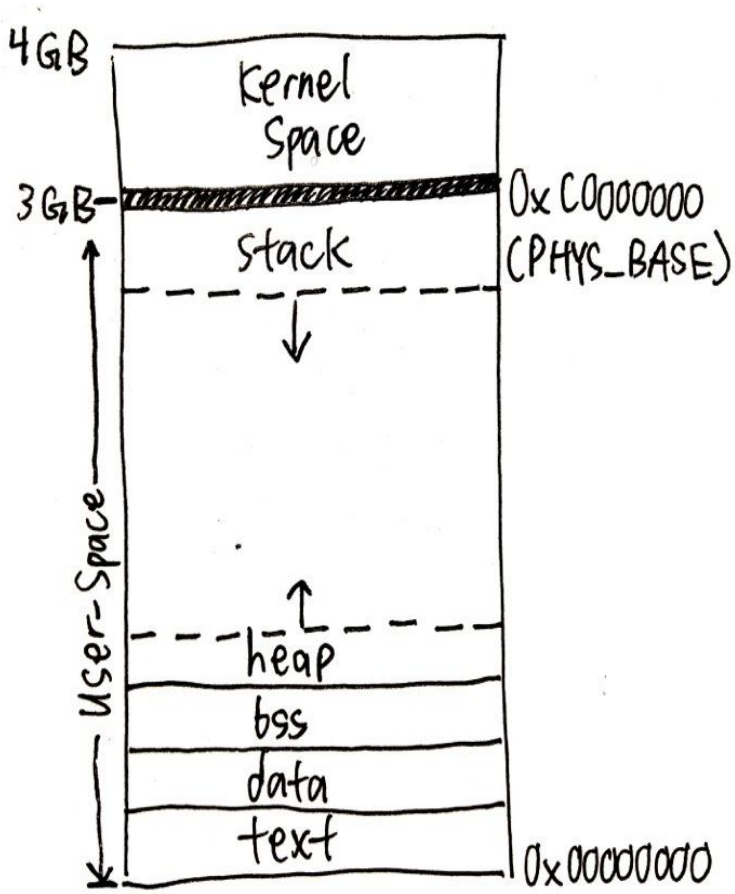
text: read-only region that contains machine instructions loaded from the code segment of the process's binary.

data: contains initialized global variables.

bss: contains uninitialized global variables.

heap: contains dynamically-allocated memory. It grows upwards starting from end of bss. This segment is managed by `malloc`, `calloc`, `realloc`, and `free`. In HW5, you'll be using the `sbrk` syscall to implement these library functions. When the heap collides with the stack, the process runs out of usable memory.

stack: last-in, first-out (LIFO) data structure that grows downwards from the top of user-space memory. A register called the stack pointer (`sp` in x86) tracks the end of the stack. With each function call, a *stack frame* is pushed to the stack. Each stack frame contains the function's local variables and return address. This is highly related to the environment diagrams in CS 61A. To learn more about the stack, you should take CS 164.



kernel-space: inaccessible to user programs, this space contains memory needed for kernel operations such as the process control block and the thread control block. In PintOS, this starts above `PHYS_BASE` which is `0xC000000`.

The kernel stores information about each process in the *process control block* (PCB). It typically stores:

1. Process ID (PID): a non-negative unique numeric identifier
2. Parent Process ID (PPID): PID of the process that created this process
3. List of Children PID: PIDs of all child processes
4. List of opened files: stores all of the file descriptors returned from the `open` syscall
5. Location of first-level page table:
    - Page tables translate virtual addresses to physical addresses.
    - All addresses in a process's address space are virtual. Anytime a process accesses memory, it will do so through virtual memory.
    - The exact mechanism behind address translation and page tables will be discussed later on. The first-level page table is also sometimes called the page directory.

Now that we know the purpose and structure of a process, let's dive into its lifecycle. In UNIX, starting a new process is simply calling `fork` and then calling `exec`.

`fork()`	Creates a new (child) process that is identical to the calling (parent) process EXCEPT for the following properties: (1) process ID, (2) parent process ID, (3) pending alarms/signals are cleared for the child. The child process gets a copy of the parent's address space. Though the entirety of both address spaces are the same, the child and the parent cannot write to the other's memory contents since they are separate processes (i.e. they each have their own copy to interact with). The child process also gets a copy of the parent's open file descriptors. Returns a value of 0 to the child process and returns the value of the new child's PID to the parent process. In both processes, execution resumes immediately after the call to `fork`. Run `man fork` in a terminal to learn more.
`exec()`	Refers to a family of functions that take different arguments but do the same thing. Given the path/name to an executable or binary, `exec` transforms the calling process into a new process by executing the new executable/binary file. This effectively "replaces" the current address space with an entirely new one. The process begins executing code from the entry point of the new program (e.g. the `main` function). However, there are a few things that are persisted: (1) PID, (2) PPID, (3) current working directory, (4) access permissions, (5) etc. Run `man 2 execve` in a terminal to learn more.

Imagine that each call to `fork` meant copying *all* of the data in the parent address space to the child address space (in the diagram earlier, this is a total 3 GB). That would be a real waste of effort if we were to call `exec` shortly after, which is almost always the case. Thankfully, this is not the case in real life. Instead, data in the

child address space is marked as *copy-on-write* (COW). That is, data is *only* copied over when the child process writes (i.e. modifies) to it.

At any given point in time, there are several applications opened on our computer. Depending on the amount of cores on its processor, the computer can only actually run a fixed amount of processes simultaneously. (On a single-core processor, we can only run one process at a time, yikes!). The operating system provides the illusion that *all* of our processes are running *at the same time* via scheduling algorithms. Though scheduling is a topic we'll explore later, here is an important consequence from the kernel's scheduler:

> You never know when the kernel will kick a process off the CPU and replace it with another one.

Given a parent process P and a child process C, there are no guarantees to the processing order. Some scenarios include:

A. P runs to completion, then C runs to completion.
B. C runs to completion, then P runs to completion.
C. P runs half-way to completion, then C runs half-way to completion, then P runs to completion, then C runs to completion.
D. C runs half-way to completion, then P runs half-way to completion, then C runs to completion, then P runs to completion.
E. P runs X lines of code, then C runs Y lines of code, then P runs half-way to completion, then C runs to completion, then P runs to completion.

There are several other scenarios, but this is how the processor is seemingly able to "multitask" -- by switching constantly between all the processes in the operating system. As programmers, how do we write code with guarantees to execution order?

We do so by leveraging the parent-child relationship between processes. Whenever a parent process creates a child process (by calling `fork`), it can use the `wait` function to suspend execution until the child process terminates.

`wait()`	Suspends execution of the calling process until a child process terminates. Use the `waitpid` function to wait on a specific child process. Both of these functions also return the exit status of the terminated child process. The exit status of a process is typically 0 if successful. A non-zero exit status is usually indicative of an error. `man 2 wait` for additional info.
`exit()`	Terminates a process. Before termination, this function calls all exit handlers (functions registered with `atexit`) and flushes all I/O streams to disk. `man 3 exit` for more info.

All processes must die. A process can terminate normally in the following ways:
(1) Calling `return` from the `main` function.
(2) Calling the `exit`, `_exit`, or `_Exit` functions.

Processes abnormally terminate by calling the `abort` function or by receiving a signal that terminates the process (see `man 3 signal`). Regardless of how a process terminates, the kernel will always close all open

descriptors and release memory that is was using.

Since execution order isn't guaranteed, the kernel accounts for the following scenarios:

(1) Parent terminates before children. An orphan process is a child process that continues to be alive after its parent has terminated. Its new Parent PID becomes 1 since it is adopted by the `init` process. The `init` process (PID 1) is the first process started during booting and is the direct or indirect ancestor of all processes. The `init` process adopts all orphaned processes and now every process is guaranteed to have a parent (and stable family structure).

(2) Child terminates but the parent doesn't `wait` for it. The child becomes a zombie process. The kernel keeps a small amount of info for every terminating process, so parent processes can access termination info (via `wait` or `waitpid`) long after a child has been terminated. At the bare minimum, this info includes: (1) the process ID, (2) the termination status of the process, and (3) the amount of CPU time used by the process.

We have now discussed what processes are and what they seek to achieve, but how do they actually execute the instructions that we want them to? Let's now introduce the concept of *threads of execution*. A process may consist of one or more threads. If a process consists of one thread, then it is called *single-threaded*. Similarly, a *multi-threaded* process is one that contains more than one thread. When a new process is created, it starts off with a single thread.

Threads belonging to the same process share the same virtual address space and system resources (defined in the PCB). Therefore, all threads within the same process:

1. execute the same program, though the threads may be using different regions of the text segment
2. have access to the same global variables
3. share data created by `malloc`, `calloc`, and `realloc`
4. possess the same PID

A thread embodies an execution context of a process. Thus, threads have their own execution environment. This means that each thread has its own program counter (PC), register state, and stack. These properties, along with others are stored in the *thread control block* (TCB). Altogether, this includes:

1. Thread ID (TID): a non-negative unique numeric identifier
2. Program Counter: points to the address of the next instruction to be executed
3. Register State
4. Stack pointer
5. Pointer to the process it belongs to
6. CPU Scheduling Info: thread priority, burst times, thread status, etc.

`pthread_create()`	Creates a new thread. The new thread starts running at the start routine (specified as an argument). The new thread terminates when it calls `return` from the start routine or when it call `pthread_exit`. The start routine function takes in a single typeless pointer called `arg`. If you need to pass in more than one argument, you'll need to store them in a struct and pass in a pointer to the structure into `arg`. When a new thread is created, there is no guarantee which thread will run first (the creator thread or the created thread). If any thread executes the `main` function's `return` statement or calls the `exit` function, the entire process (and all of its threads) is terminated.

	Use `pthread_join` after calling `pthread_create` to ensure an order of execution.
`pthread_join()`	Suspends execution of the calling thread until the targeted thread (specified as a parameter to `pthread_join`) terminates. Also reveals the return value of the targeted thread.
`pthread_exit()`	Terminates the calling thread. The `rval_ptr` parameter is made available to other threads in the process by calling the `pthread_join` function.

Since threads share the same address space, concurrent access to shared variables combined with the kernel's scheduler lead to nondeterminism. Synchronization primitives (e.g. locks, semaphores, condition variables) help programmers synchronize access to shared memory. A program that produces deterministic results in a multi-threaded environment is said to be thread-safe. More on these later.

All in all, processes encapsulate isolation and threads encapsulate concurrency. A process embodies a running computer program and threads embody the execution of a process.

==========
========== Frequently Asked Questions ==========
==========

Q: If threads belonging to the same process share the same virtual address space, then don't they share the same stack? Why are you saying that each thread has its own stack?

A: Since each thread executes a function independently from others, the stack of each thread must also be independent. Recall from above that a stack frame is pushed to the stack with each function call. Upon thread creation, the stack pointer of the new thread is set equal to the stack pointer of the creator thread. Contents of the stacks of the new thread and the creator thread begin to differ once the new thread begins executing code. However, both threads have access to the same contents of the stack *prior* to the creation of the new thread.

==========

Q: Can an orphan process become a zombie?

A: No. The `init` process is written to *always* call `wait` to fetch the termination status. By doing so, it prevents the system from being overrun by zombies.

==========