CS 111: Operating System Principles
Lecture 4

Processes
1.0.2

Jon Eyolfson
April 6, 2021

A Process is an Instance of a Running Program

A program (or application) is just a static definition, including:

- Instructions
- Data
- Memory allocations
- Symbols it uses

When you start executing a program, it turns into a process

Process is like a Combination of all the Virtual Resources

If we consider a “virtual CPU”, the OS needs to track all registers

It also contains all other resources it can access (memory and I/O)

Every execution runs the same code (part of the program)

An execution is running some specific code (part of the process)

A Process is More Flexible

A process contains both the program and information specific to its execution

It allows multiple executions of the same program

It even allows a process to run multiple copies of itself

A Process Control Block (PCB) Contains All Execution Information

Specifically, in Linux, this is the `task_struct` saw in Lab 0

It contains:

- Process state
- CPU registers
- Scheduling information
- Memory management information
- I/O status information
- Any other type of accounting information

Aside: Concurrency and Parallelism Aren't the Same

Concurrency

- Switching between two or more things (can you get interrupted)
  - Goal: make progress on multiple things

Parallelism

- Running two or more things at the same time (are they independent)
  - Goal: run as fast as possible

A Real Life Situation of Concurrency and Parallelism

You're sitting at a table for dinner, and you can:

- Eat
- Drink
- Talk
- Gesture

You're so hungry that if you start eating you won't stop until you finish

Which tasks can and can't be done concurrently, and in parallel?

Choose Any Two Tasks in the Real Life Example

You can't eat and talk (or drink) at the same time, and you can't switch

Not parallel and not concurrent

You could eat and gesture at the same time, but you can't switch

Parallel and not concurrent

You can't drink and talk at the same time, and you could switch

Not parallel and concurrent

You can talk (or drink) and gesture at the same time, and you could switch

Parallel and concurrent

Uniprogramming is for Old Batch Processing Operating Systems

Uniprogramming: only one process running at a time

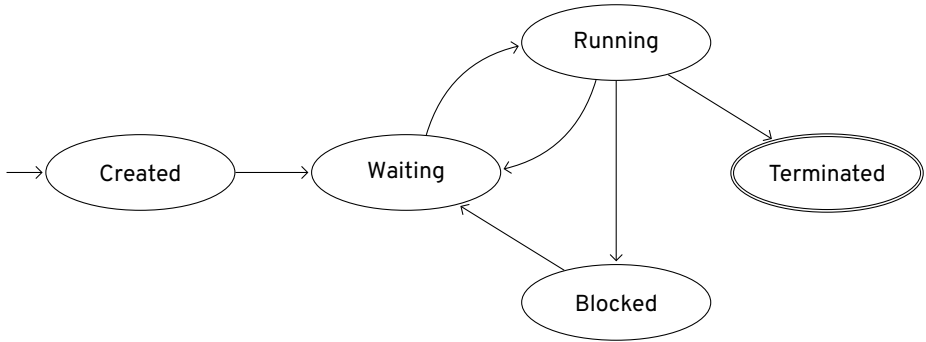
Two processes are not parallel and not concurrent, no matter what

Multiprogramming: allow multiple processes

Two processes can run in parallel or concurrently

Modern operating systems try to run everything in parallel and concurrently

Process State Diagram (You Could Rename Waiting to Ready)



The Scheduler Decides When To Switch

To create a process, the operating system has to at least load it into memory

When it's waiting, the scheduler (coming later) decides when it's running

We're going to first focus on the mechanics of switching processes

The Core Scheduling Loop Changes Running Processes

1. Pause the currently running process
2. Save its state, so you can restore it later
3. Get the next process to run from the scheduler
4. Load the next process' state and let that run

We Can Let Processes Themselves, or the Operating System Pause

Cooperative multitasking

The processes use a system call to tell the operating system to pause it

True multitasking

The operating system retains control and pauses processes

For true multitasking the operating system can:

- Give processes set time slices
- Wake up periodically using interrupts to do scheduling

Swapping Processes is called Context Switching

We've said that at minimum we'd have to save all of the current registers

We have to save all of the values, using the same CPU as we're trying to save

There's hardware support for saving state, however you may not want to save everything

Context switching is pure overhead, we want it to be as fast as possible

Usually there's a combination of hardware and software to save as little as possible

We Could Create Processes from Scratch

We load the program into memory and create the process control block

This is what Windows does

Could we decompose this into more flexible abstractions?

Instead of Creating a New Process, We Could Clone It

Pause the currently running process, and copy it's PCB into a new one

- This will reuse all of the information from the process, including variables!

Distinguish between the two processes with a parent and child relationship

- They could both execute different parts of the program together

We could then allow either process to load a new program and setup a new PCB

On Unix, the Kernel Launches A Single User Process

After the kernel initializes, it creates a single process from a program

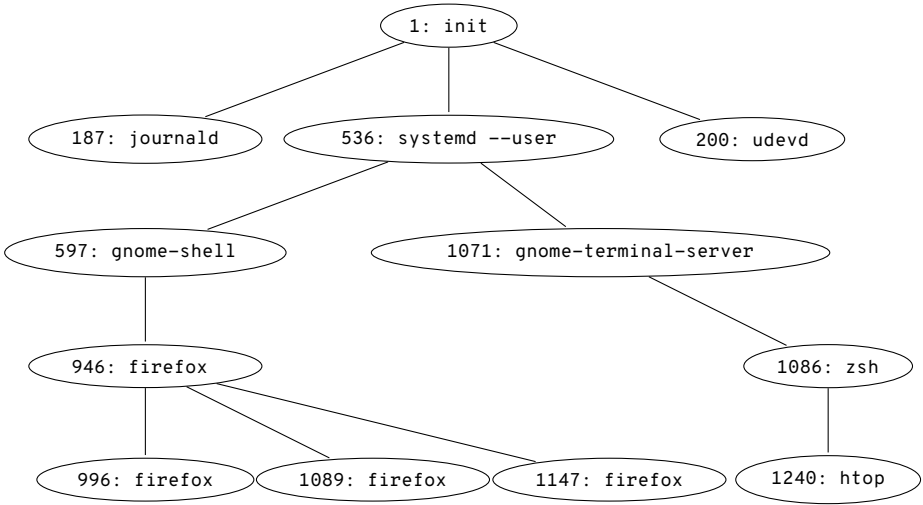
This process is called `init`, and it looks for it in `/sbin/init`

- Responsible for executing every other process on the machine
- Must always be active, if it exits the kernel thinks you're shutting down

For Linux, `init` will probably be `systemd` but there's other options

Aside: some operating systems create an "idle" process that the scheduler can run

A Typical Process Tree on the Virtual Machine



How You Can See Your Process Tree

Use `htop`

`sudo pacman -S htop` to install on the virtual machine

You can press F5 to switch between tree and list view

You may have to update all your packages first:

`sudo pacman -Syu` (Reboot if your kernel updates)

The Parent Process is Responsible for Its Child

The operating sets the exit status when a process terminates (by calling exit)

It can't remove its PCB yet

The minimum acknowledgment the parent has to do is read the child's exit status

There's two situations:

1. The child exits first (zombie process)
2. The parent exits first (orphan process)

A Zombie Process Waits for Its Parent to Read Its Exit Status

The process is terminated, but it hasn't been acknowledged

A process may have an error in it, where it never reads the child's exit status

The operating can interrupt the parent process to tell it to acknowledge the child

- This is a basic form of IPC called a signal (which may be ignored)

The operating system has to keep a zombie process until it's acknowledged

- If the parent ignores it, the zombie process needs to wait to be re-parented

An Orphan Process Needs a New Parent

The child process lost its parent process

- The child still needs a process to acknowledge its exit

The operating system re-parents the child process to `init`

- The `init` process is now responsible to acknowledge the child

The Operating System Creates and Runs Processes

The operating system has to:

- Loads a program, and create a process with context
- Maintain process control blocks, including state
- Switch between running processes using a context switch
- Unix kernels start an `init` process
- Unix processes have to maintain a parent and child relationship