ECE 344: Operating Systems

Lecture 3

Process Life Cycle

1.1.0

Jon Eyolfson

September 13/14, 2022

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

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

A *process* executes a program, and the kernel manages processes

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` you can browse on GitHub

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

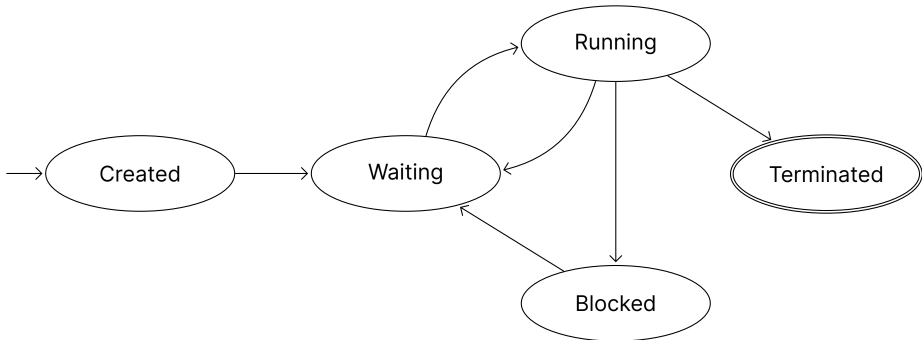
- Two processes are not parallel and not concurrent, no matter what

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

System Calls are Rare in C

Mostly you'll be using functions from the C standard library instead

Most system calls have corresponding function calls in C, but may:

- Set `errno`
- Buffer reads and writes (reduce the number of system calls)
- Simplify interfaces (function combines two system calls)
- Add new features

C `exit` Has Additional Features

System call `exit` (or `exit_group`): the program stops at that point

C `exit`: there's a feature to register functions to call on program exit (`atexit`)

```
#include <stdio.h>
#include <stdlib.h>

void fini(void) {
  puts("Do fini");
}

int main(int argc, char **argv) {
  atexit(fini);
  puts("Do main");
  return 0;
}
```

`lecture-03/atexit-example`

`execve` Replaces the Process with Another Program, and Resets

`execve` has the following API:

- `pathname`: Full path of the program to load
- `argv`: Array of strings (array of characters), terminated by a null pointer
  - Represents arguments to the process
- `envp`: Same as `argv`
  - Represents the environment of the process
- Returns an error on failure, does not return if successful

execve-example.c Turns the Process into ls

```
int main(int argc, char *argv[]) {
  printf("I'm going to become another process\n");
  char *exec_argv[] = {"ls", NULL};
  char *exec_envp[] = {NULL};
  int exec_return = execve("/usr/bin/ls", exec_argv, exec_envp);
  if (exec_return == -1) {
    exec_return = errno;
    perror("execve failed");
    return exec_return;
  }
  printf("If execve worked, this will never print\n");
  return 0;
}
```

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
- Replace programs running in processes (Unix)