CSCI 460 Operating Systems

Processes (Part II)

Professor Travis Peters
Fall 2019

Some slides & figures adapted from Stallings instructor resources.
Some slides adapted from Adam Bates's F'18 CS423 course @ UIUC
https://courses.engr.illinois.edu/cs423/sp2018/schedule.html
Goals for Today

- **Learning Objectives**
  - Understand basic concept of process (control info, creation, termination, states, etc.)
  - Review some important UNIX syscalls and concepts for sys. programming

- **Announcements**
  - Grades should be fixed…
  - Rough schedule posted today
  - Coming Soon…
    - zyBook for OS (optional resource); link posted soon
    - 1st programming assignment
    - Details about project

https://www.traviswpeters.com/cs460/
Last Time… Creating & Managing Processes

![Diagram showing the process tree and state transitions for processes in a system.](image-url)
Creating a Process

DEMO

take a look at a process tree (already-created processes)

```
ps axjf
```

Some things to note:
- **names** (e.g., init),
- **relationships** (parents, children, grandchildren),
- **IDs** (PID, PPID)
Creating a Process

- **Q:** But how to create a process? What UNIX call creates a process?
Creating a Process - `fork()`

- **Q:** But *how* to create a process? What UNIX call creates a process?
  - `fork()` duplicates a process so that instead of *one* process, you get *two*!
    - *P1 and P2 continue in parallel from the statement that follows the `fork()`*
- **Q:** How can you tell P1 and P2 apart?
  - …the return value of `fork()`!
    - `rval == 0` //to child
    - `rval == child_pid` //to parent
    - `rval == -1` //`fork()` failed

- **Q:** Will *child* see changes to global variable made by the *parent*?
  - No! On `fork()`, child gets new PC, stack, heap, globals, PID!
Creating a (Different) Process

• **Q:** What if we want the child process to execute different code than the parent process?
Creating a (Different) Process - \texttt{exec()} \\

- **Q:** What if we want the child process to execute different code than the parent process? 
- \texttt{exec()} ... 
  - ...allows child to execute code that is different from the parent’s code 
  - ...has a family of functions (\texttt{execl()}, \texttt{execv()}, \texttt{execle()}, \texttt{execlp()}, \texttt{execlvp()}) that provide a facility for overlaying the process image of the calling process with a new image 
  - ...returns -1 and sets \texttt{errno} if unsuccessful
Example: `fork()` and `exec()`

```
/* Start a new process to do the job. */
cpid = fork();
// printf("process id is %d\n", cpid);

if(cpid < 0) {
    perror("fork");
    free(main_ptr); // clean-up
    return;
}

/* Check for who we are! */
if(cpid == 0) {
    /* We are the child! */
    execvp(main_ptr[0], main_ptr);
    perror("exec");
    free(main_ptr); // clean-up
    exit(127);
}

/* Have the parent wait for child to complete */
if(wait(&status) < 0)
    perror("wait");
// printf("wait result for process id %d is %d\n", cpid, status);
```


— fork() and exec() code snippet that Travis wrote back when he was an undergrad...
Terminating a Process

• There must *(at least should…)* be a means for a process to indicate its completion
• A batch job should include a HALT instruction (or something similar)
• An interactive process will terminate when a user, e.g., logs off, quits an app
• Again, *there is some discussion in the text on when/why processes terminate…*
  • …*normal completion*
  • …*timeout*
  • …*killed due to error/violation*
  • …*parent terminates*
  • …*explicitly requested (e.g., parent’s request)*
  • *etc.*
Terminating a Process

- **Q:** What is the mechanism in UNIX to explicitly terminate a process?
Terminating a Process - `kill()`

- **Q:** What is the mechanism in UNIX to explicitly terminate a process?
- **`kill()`**
  - …enables terminating a process by specifying the **PID** and a **signal**
  - …default at commandline is to send signal 15 (SIGTERM) — “please terminate…”
- **Read up on **signals** and different types of signals…**
  - …signals are basically just **software interrupts** that can be sent to a program to indicate that an important event has occurred.
- **Exercise:** run the following command in a terminal: `kill -l`
Suspended Processes & Swapping

• Swapping
  • Move parts of the process from main memory to disk to free up resources for other processes
  • OS will choose to swap blocked process(es) out to disk into a **suspended state**
  • Swapping is an I/O operation; has the potential to make the problem worse…
    • on-system I/O is pretty fast though (relative to, e.g., network I/O)
    • Swapping *usually* improves performance

• **Q:** Why swap?

  *OS needs to release resources, debugging, periodic task, etc.*

• **Q:** Why not make memory bigger?!

  *Cost; not more processes, usually bigger processes*
Suspended Processes & Swapping (Before)
Suspended Processes & Swapping (After)
Suspended Processes & Swapping *(After +Ready/Blocked)*

![Process State Transition Diagram with Suspend States](image)

- **New**
- **Ready/Suspend**
- **Blocked/Suspend**
- **Ready**
- **Blocked**
- **Running**
- **Exit**

Events and Transitions:
- Admit
- Suspend
- Activate
- Dispatch
- Release
- Exit
- Timeout
- Event Wait
- Event Occurs

**(a) With One Suspend State**

**(b) With Two Suspend States**
Summary: Process States

• It is **important for the OS to maintain information** about each process and its state **to manage system resources effectively**.

• Many, *many* **trade-offs** to consider…
  
  • More states (and more data structures for managing processes) are needed as we want to **reduce unnecessary operations**, such as searching through queues to identify ready processes, higher-priority processes, etc.
  
  • But more data structures means **more memory is used** to maintain this state information, and **more overhead operations** to move processes between different data structures when they are blocked, suspended, ready, etc.

**Suggestion:**

> know the differences between various process states, and understand the reasons for (and pros/cons of) each.
Fun: The **fork()** Bomb

- **WARNING:** Run at your own risk. I’m running on a VM…

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
    while(1) {
        fork();
    }
    return 0;
}
```

forkbomb() { forkbomb | forkbomb & } ; forkbomb

—https://en.wikipedia.org/wiki/Fork_bomb