COMP124

Computer Systems

13 | Linux Processes | Pipes | Sockets

Dr Stuart Thomason

Process Creation

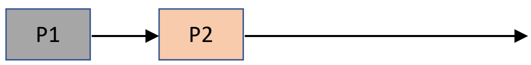
- Processes are created (spawned) by other processes
- Original process is the parent
- New process is the child
- All running processes form a tree structure
- Linux pstree command
- Everything has systemd as the top-level ancestor

Linux Process Creation

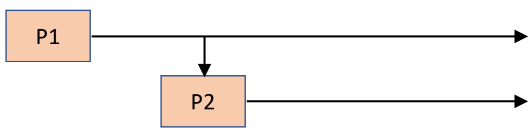
- Linux provides several system calls that allow a process to spawn a child
- exec()
  - Allows the process to execute another process
  - Child replaces (overwrites) parent in memory and PCB
- fork()
  - Spawns a new clone of the process
  - Both parent and child continue to run
- wait()
  - Called by parent process
  - Blocks until child process terminates

Process Execution Flow

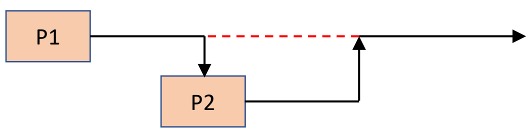
- P1 uses exec() to spawn P2



- P1 uses fork() to spawn P2



- P1 uses fork() to spawn P2 and then wait() to block



fork() – Return Value

- The fork() system call returns one of three possible values
  - < 0 (negative) – If the child could not be created (failure)
  - = 0 (zero) – In the child process
  - > 0 (positive) – In the parent process (child process ID)
- Typical fragment of C code to spawn a new process…

```
int pid = fork();
// Call returns in each of the two processes
if(pid == 0) {
    printf("I'm the child process");
    // Will usually call exec() to load its own code
} else {
    printf("I'm the parent and my child's ID is %d", pid);
}
```

The First Process

- The ROM stores a small program that runs a bootloader when a system is first turned on (ie. booted)
  - Linux systems use GRUB – GNU Grand Unified Bootloader
  - Loads kernel image from disk and starts it executing
- The first process to run is called systemd
  - Its process ID (PID) is 1
  - Spawns all the other processes required by the kernel
  - Can be configured for various targets (eg. server or desktop)
- Continues to run as a background process (daemon)
  - Offers on-demand spawning of other services
  - Maintains logfiles to record system activity
  - Keeps track of other processes and kernel settings

Shell (Terminal) Login

- The sshd daemon runs in the background, waiting for incoming connections (spawned by systemd when the system boots up)
  - We use a ssh client to connect to a Linux server
  - The sshd daemon uses fork() to spawn a child process
  - The child uses exec() to run a login process
  - The login process checks our credentials are correct
  - Then it uses exec() to run our preferred shell process
- Processes are being created all the time to handle whatever the system needs to do, and whatever users are trying to do
- Everything is done via the fork() and exec() system calls

Running a Shell Command

- When we type a command into the shell…
  - The shell uses fork() to create a child process
  - That child uses exec() to run the command we typed in
- You can see this with the ps command

```
[scap21@lxfarm03 ~]$ ps
  PID TTY          TIME CMD
  730 pts/0    00:00:00 bash
 4737 pts/0    00:00:00 ps
[scap21@lxfarm03 ~]$
```

- Both the shell (bash, PID 730) and the ps command (PID 4737) are running at the same time

Zombies and Orphans

- Parent processes usually wait for their children to die (!)

- If the death (termination) of a child is not acknowledged by the parent via the wait() system call…
  - The child becomes a zombie
  - It has finished but is still present in the process table (because clean-up doesn't happen until parent is ready)

- If a parent terminates before its children…
  - The children become orphans
  - They are adopted by the systemd process

- The systemd process periodically calls wait() on its children to clean up any zombies and orphans that are in the process table

Daemon Processes

- Already mentioned some daemon processes such as systemd and sshd, but there are many others
  - Usually the process name ends with 'd' to signify daemon
  - Not associated with any shell or any user
  - Run permanently in the background

- Perform the background operations of the operating system
  - Subsystem managers are daemon processes
  - Need their own time on the CPU, so must be scheduled!
  - Usually run with a higher priority

- Perform tasks requested by other processes
  (ie. they act as servers in a client-server relationship)

Processes in the Linux File System

- Kernel stores housekeeping information in the /proc directory
  - Dynamic details about the current state of the kernel
  - Virtual file system called procfs (ie. not all real files on disk)
  - Subdirectory for each running process

Subdirectory	Purpose
/proc/PID/	Stores all the details and status of process PID
/proc/PID/cmdline	The text that was typed to start the process
/proc/PID/fdinfo	The status of any open files used by the process
/proc/PID/status	The overall status of the process (lots of detail)
/proc/cpuinfo	Stores details about the physical CPU(s)
/proc/modules	Stores info about currently loaded kernel modules

(Note: There are many other subdirectories – these are just a few examples)

Looking at Linux Processes

- We can use the top command to see dynamic process details

```
top - 09:25:03 up 916 days, 18:14,  2 users,  load average: 0.08, 0.06, 0.05
Tasks: 240 total,   1 running, 239 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.0 us,  0.0 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16304004 total,  1259880 free,  3400064 used, 11644060 buff/cache
KiB Swap: 16777212 total, 10266540 free,  6510672 used. 11910512 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1717 gdm       20   0 1054492 263320   1276 S   7.0  1.6  65753:59 gsd-color
  898 scap21    20   0  172960   2484   1636 R   0.3  0.0   0:00.33 top
11123 root      20   0  863580  12420   5512 S   0.3  0.1  25:18.90 docker-containe
15768 root      20   0  153536  19860   8028 S   0.3  0.1 945:52.83 gitlab-runner
31438 root      20   0 1218932  47472  33304 S   0.3  0.3 391:22.48 fail2ban-server
    1 root      20   0  128584   7180   4088 S   0.0  0.0 192:31.17 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   2:14.13 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   3:28.69 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S   0.0  0.0   1:35.39 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.02 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0 443:45.73 rcu_sched
   10 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 lru-add-drain
```

Linux Signals

- A process (running from the shell) can usually be terminated by typing ^C (CTRL + C)
  - This sends an interrupt signal to the process
  - Process intercepts signal and responds by terminating
- Signals can be sent between processes with signal() system call
- We can send signals with the kill command at the shell prompt
  - There are various signals denoted by numbers and codes
  - For example, to terminate process 438

    kill -s SIGKILL 438

Responding to Signals

- The process that receives a signal can respond in three ways
  - Perform the action requested
  - Ignore the signal completely
  - Catch the signal and run some other arbitrary code
- The only signal that cannot be ignored or caught is SIGKILL (9)

Code	Number	Meaning
SIGINT	2	Interrupted from keyboard (via CTRL+C)
SIGKILL	9	Request to terminate process (cannot be ignored)
SIGTERM	15	Request to terminate process (might be ignored)
SIGCHLD	17	Indicates that a child process has terminated
SIGIO	29	Indicates that input or output is ready

(Note: There are many other signals – these are just a few examples)

Terminating Zombie Processes

- Imagine a parent process is badly coded…
  - Spawns a child via fork()
  - But does not call wait() to clean up after its child terminates
- As we saw, the child becomes a zombie
  - Hangs around in the process table doing nothing
  - Will take up some (minimal) system resources
- We can try to send the SIGCHLD signal to the parent, but if it's badly coded it will probably just ignore it anyway
- So we will have to send the SIGKILL signal to the parent to kill it
  - Parent will terminate and systemd will adopt the zombie
  - Zombie will be cleaned up via the periodic call to wait()

Inter-Process Communication (IPC)

- Processes very often need to communicate with each other
  - To share, send and receive data
  - To provide services (servers) to other processes (clients)
- Types of IPC
  - Shared memory
  - Shared files
  - Pipes
  - Sockets
- Shared memory and shared files allow two processes to access the same memory location or file at the same time
  - Introduces synchronisation issues
  - Needs to be coordinated by semaphores and locks (which will be covered in a future bundle of videos)

Pipes

- A pipe is a form of IPC between two children of the same parent

- Usually triggered by typing a command at the prompt
  - Join two processes with the | (pipe) symbol
  - Output from the first becomes input to the second

- Example…
  - We can list all the kernel modules with cat /proc/modules
  - And we can count the lines in a file with wc --lines filename (but if we miss out the filename it counts lines of input)

```
[scap21@lxfarm03 ~]$ cat /proc/modules | wc --lines
149
[scap21@lxfarm03 ~]$
```

  - So we can find out how many kernel modules are running

Sockets

- A socket is a form of IPC that can span multiple systems
  - One process is the server (daemon) listening for clients
  - Other process is the client that connects to the server
  - Communication is bidirectional (ie. both can send/receive)
  - Implemented as special files in the file system

- The processes don't need to be on the same machine
  - Don't need the same parent process (unlike pipes)
  - Typically provide internet services to the rest of the world
  - Server (eg. httpd, maild, sshd) is running as a daemon
  - Clients connect when they need to get/send data

- Server process uses the listen() system call to wait for clients
  - When a client connects, usually spawns child via fork()
  - Child process handles the communication then terminates

Summary

- Process creation via system calls
- Chain of processes involved in logging in and running commands
- Zombies and orphans
- Daemon processes
- Process housekeeping stored in the file system
- Signals
- Inter-process communication – pipes and sockets