Processes in Linux

What’s a “process”?

A dynamically executing instance of a program.
Constituents of a “process”

- its code
- data
- various attributes OS needs to manage it

OS keeps track of all processes

- Process table/array/list
- Elements are process descriptors (aka control blocks)
- Descriptors reference code & data
Process state as data structure

“We can think of a process as consisting of three components:
   An executable program
   The associated data needed… (variables, work space, buffers, etc)
   The execution context of the program

This last element is essential. The execution context, or process state, includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly…. Thus, the process is realized as a data structure [called the process control block or process descriptor].”

*Operating Systems, Internals and Design Principles*, William Stallings
Process descriptor tracks a process

- process descriptor table
  - a descriptor, for a single process; contains or points to that process’s attributes
  - identifiers, state, resources
    - • my process id number
    - • user account associated with me
    - • id number of my parent process
    - • id numbers of my children
    - • my state
      - • readiness to run
      - • run priority
      - • CPU’s state
      - • flags
      - • register values
    - • files I hold open
    - • memory locations I occupy

Per-process data structures

- Process “image”
  - all constituents collectively
    - • code
    - • data
    - • attributes
- Process descriptor (aka control block)
  - attribute-holding data structure
Process “image”: all memory components together

Process descriptor in Linux

Understanding the Linux Kernel
Bovet & Cesati
Process descriptor’s role

“The process control block [or process descriptor] is the most important data structure in an operating system. Each process control block contains all of the information about a process that is needed by the operating system. The blocks are read and/or modified by virtually every module in the operating system, including those involved with scheduling, resource allocation, interrupt processing, and performance monitoring and analysis. One can say that the set of process control blocks defines the state of the operating system.”

Operating Systems, Internals and Design Principles, William Stallings
Process creation in unix

--how can one process spawn another?

- performed by `fork()` system call
- creates new process by copying old
- *both* copies then proceed running
  - old copy resumes (after "fork()")
  - so does new
- new copy is *not* functionally different
New process creation - `fork()`

- **process ID #1001**
  - current dir
  - arguments
  - environment (variables)
  - signal table
  - file descriptors
  - user
  - Code: `fork()`

- **process ID #1002**
  - current dir
  - arguments
  - environment (variables)
  - signal table
  - file descriptors
  - user
  - Code: `fork()`

**same dir!**
**same args!**
**same vars!**
**same sigs!**
**same files!**
**same user!**

even...
**same code!!**

---

fork - two, where there was one

```c
#include <stdio.h>

int main() {
    printf("How many times do you see this line?\n");
    fork();
    printf("How about this one?\n");
}
```

- **single print function**
- **single run**

How many times do you see this line?
How about this one?
How about this one?

because 2 (identical) processes
(the one we ran, the one it ran)
Process differentiation in unix

- identical? not what we had in mind!
- more useful if child does different stuff
- can we give it different behavior?

fork - same code, different output

```
#include <stdio.h>
main() {  
    printf("\n\n", getpid());  
    fork();  
    printf("\n", getpid());  
}
```

```bash
[root@EMACH1 bookcode]# cat fork2.c
#include <stdio.h>
main() {  
    printf("\n\n", getpid());  
    fork();  
    printf("\n", getpid());  
}
```

```
[root@EMACH1 bookcode]# gcc fork2.c -o fork2
[root@EMACH1 bookcode]# ./fork2
6749
6750
6749
```

process id #
(respective)

double output (but non-identical)
6749 is parent, 6750 is child
fork - how to self-identify?

```c
#include <stdio.h>

int main() {
    int result;
    printf( "\n%u\n", getpid() );
    result = fork();
    printf( "%d - got %d\n", getpid(), result );
}
```

```
[root@EMACH1 bookcode]# cat fork3.c
#include <stdio.h>

int main() {
    int result;
    printf( "\n%u\n", getpid() );
    result = fork();
    printf( "%d - got %d\n", getpid(), result );
}
```

```
[root@EMACH1 bookcode]# gcc fork3.c -o fork3
[root@EMACH1 bookcode]# ./fork3
```

- if 0, I must be the child copy
- if not, I must be the parent copy

---

Now provide different behavior

- in the form of source code or
- in the form of an existing binary executable
Provide new behavior from source code

conditional, on whether parent or child

Process differentiation in unix

- performed by `exec()` system call
- guts code and replaces it
- copy now does/is something “else”
- complete strategy is “selfcopy-and-alter” not just “create”
Making it different - exec()

process ID #1002 - one moment

- Code
  - fork()
- environment (variables)
- signal table
- file descriptors
- user

process ID #1002 - one moment later

- Code
  - exec() completely different code
- environment (variables)
- signal table
- file descriptors
- user

Making it different - exec()

- exec() code transplant
- also initializes this stuff

Process ID #1002 - one moment

Poof! code transplant

Process ID #1002 - one moment later

Poof! code transplant

Provide new behavior from binary code

```bash
[root@EMACH1 bookcode]# cat fork5.c
#include <unistd.h>
#include <stdio.h>

main() { int result;
  printf( "\nParent does stuff and then...\n\n\n" );
  result = fork();
  if ( result == 0 ) {
    printf("Child could run some executable...\n\n");
    execl("/bin/ls", /bin/ls","-l","/etc/httpd/conf",NULL); }
  else
    printf("...parent do something completely different.\n\n"); }

[root@EMACH1 bookcode]# ./fork5

Parent does stuff and then...
Child could run some executable...

ls -l /etc/httpd/conf (the real thing)
```
Some system function calls

- **fork** - creates a child process that differs from the parent process only in its PID and PPID
- **exec** - replaces the current process image with a new process image
- **wait** - suspends execution of the current process until its child has exited
- **exit** - causes normal program termination and a return value sent to the parent

For example...

- Shell is running
- You type “ls” and Enter
- Shell is parent, spawns ls as child