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 table tracks the processes
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

© David Morgan 2003-19
**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*

---

**Single process in unix, consolidated view**

Some important components

- code
- data
- current directory
- argument list
  - tokens from command line
- environment (variable) list
  - name=value pairs
- responses to signals
- list of open files
- user “as whom” process operates
ls -l foo bar

came from /bin/ls

/home/david

0 = ls  1 = -l  2 = foo  3 = bar

UID=500  LINES=25  PATH=…

1 default
2 handler pointer
3 ignore

0 stdin
1 stdout
2 stderr
3 /bin/ls

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()

- fork() is used to create a new process.
- Code is the same in both processes.
- Arguments, environment, signal table, file descriptors, and user are identical.
- Current directory is the same.

fork - two, where there was one

- Output is double (identical) because two identical processes were run.
- Process ID #1001
  - Code: fork()
  - Arguments
  - Environment (variables)
  - Signal table
  - File descriptors
  - User
- Process ID #1002
  - Code: fork()
  - Arguments
  - Environment (variables)
  - Signal table
  - File descriptors
  - User

Single print function
Single run

but double (identical) output
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

code:
```c
#include <stdio.h>

main() {
    printf( "\n\n", getpid() );
    fork();
    printf( "%i\n", getpid() ); }```

```
[root@EMACH1 bookcode]# cat fork2.c
#include <stdio.h>

main() {
    printf( "\n\n", getpid() );
    fork();
    printf( "%i\n", getpid() ); }```

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

6749
6750
6749

6749 is parent, 6750 is child

double output (but non-identical)

process id # (respective)
fork - how to self-identify?

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

int main() {
    int result;
    result = fork();
    if (result == 0) {
        printf("Child can do one thing\n");
    } else {
        printf("...parent something completely different\n");
    }
}
```

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( )

- fork()
  - exec( executable file)

process ID #1002 - one moment

current dir
arguments
environment (variables)
signal table
file descriptors
user

Code

proof!

Also initializes this stuff

code transplant

process ID #1002 - one moment later

Code

completely different code

data

user

file descriptors
signal table
environment (variables)
arguments
current dir

Making it different - exec( )

- fork()
  - exec( executable file)

process ID #1002 - one moment

current dir
arguments
environment (variables)
signal table
file descriptors
user

Code

proof!

Also initializes this stuff

code transplant

process ID #1002 - one moment later

Code

completely different code

data

user

file descriptors
signal table
environment (variables)
arguments
current dir

Provide new behavior from binary code

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