Processes in linux

David Morgan

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

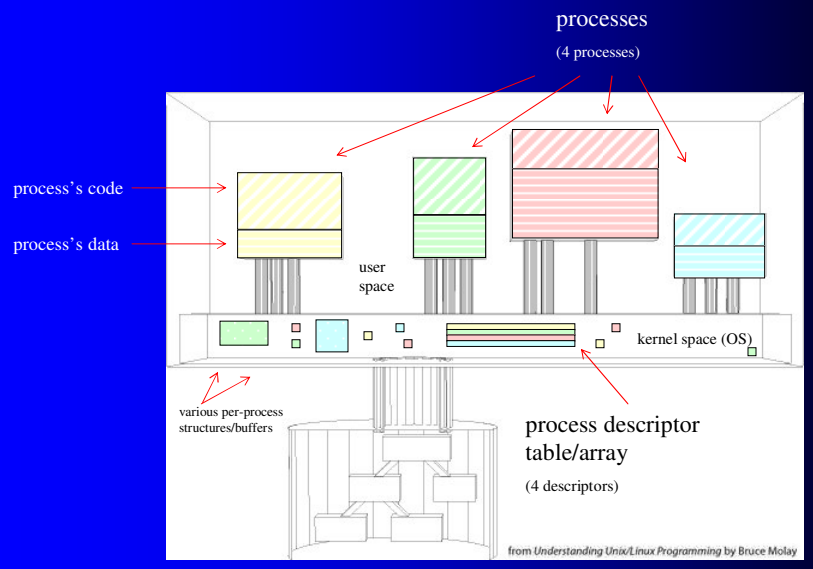
The associated data needed… (variables, work space, buffers, etc)

The execution context of the program

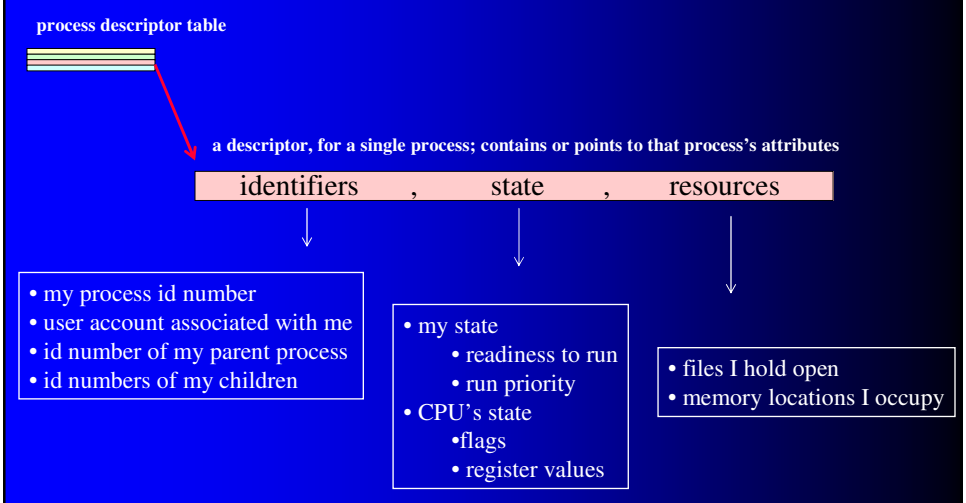
This last element is essential. The execution context, or process state, includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly…. Thus, the process is realized as a data structure [called the process control block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

Operating Systems, Internals and Design Principles, William Stallings

Process table tracks the processes



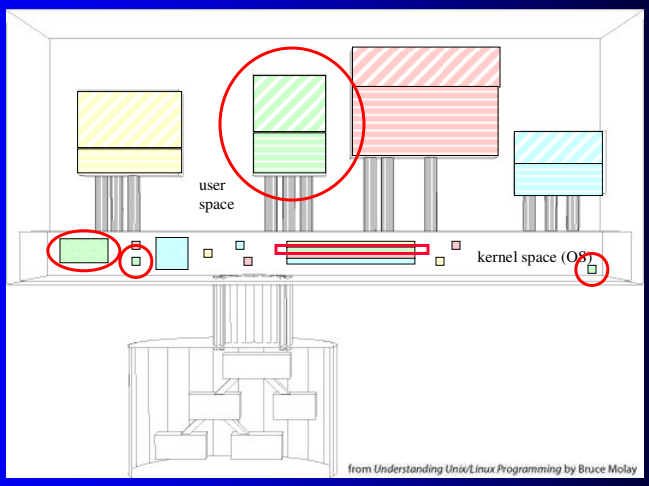
Process descriptor tracks a process



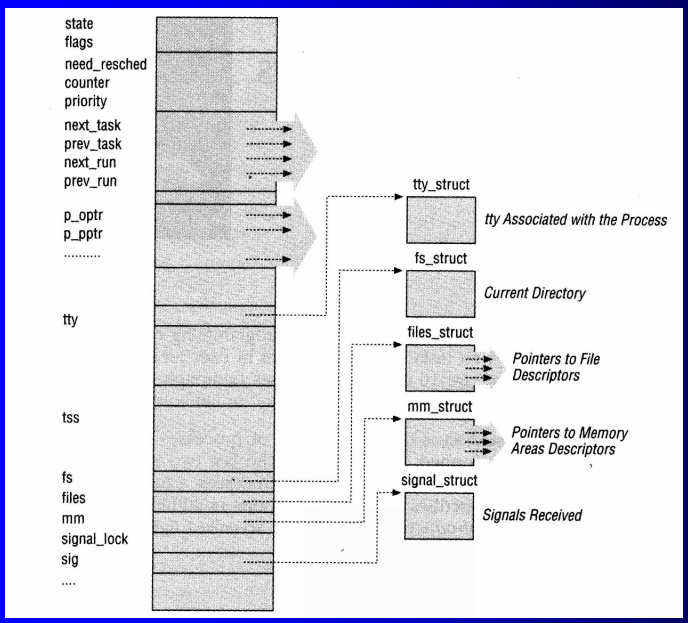
Per-process data structures

- 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



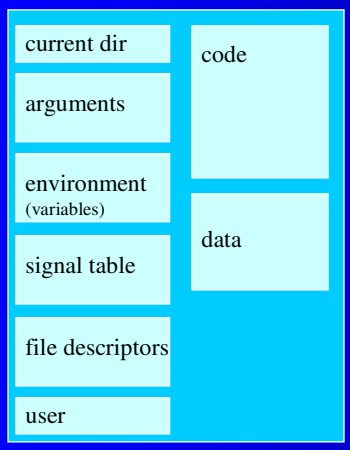
Understanding the Linux Kernel, Bovet & Cesati

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

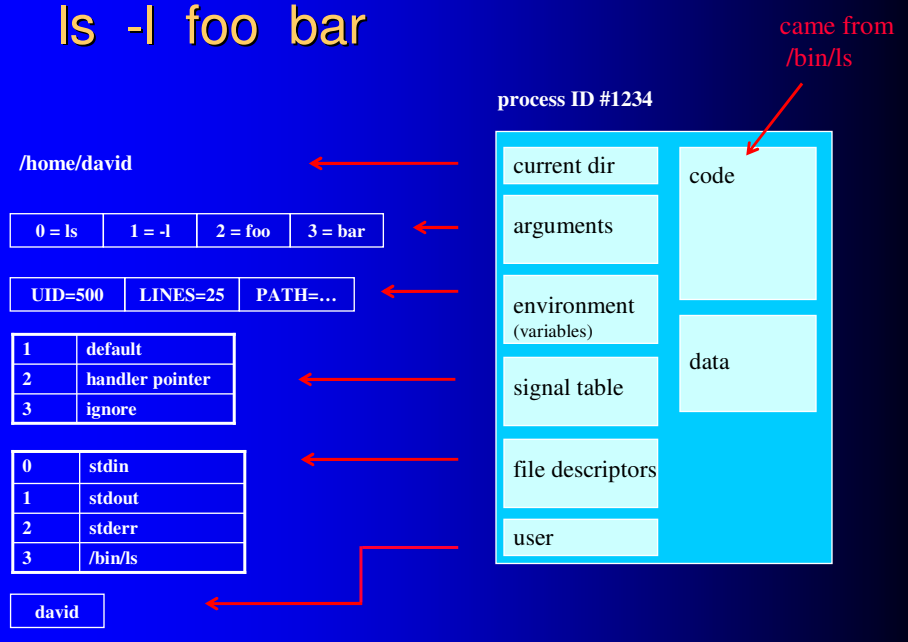
Single process in unix, consolidated view



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



Process creation

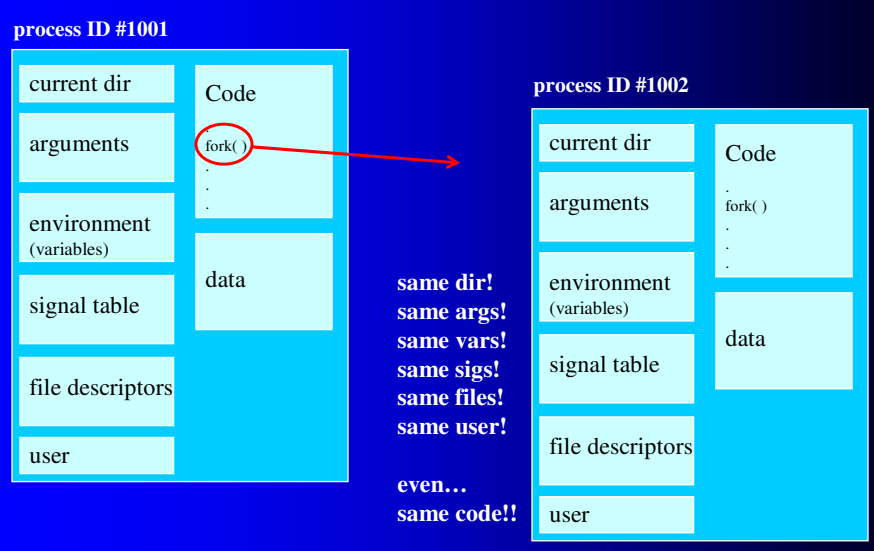
- Find empty slot in process table
- Write a process descriptor and put it there
- Read in program code from disk

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 - two, where there was one

```
root@EMACH1:~/class/books/molay/ch08/bookcode
File  Edit  View  Terminal  Tabs  Help
[root@EMACH1 bookcode]# cat fork1.c
#include        <stdio.h>
main() {
        printf("\nHow many times do you see this line?\n");
        fork();
        printf("How about this one?\n"); }

[root@EMACH1 bookcode]# gcc fork1.c -o fork1
[root@EMACH1 bookcode]# ./fork1

How many times do you see this line?
How about this one?
How about this one?
[root@EMACH1 bookcode]#
```

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

```
root@EMACH1:~/class/books/molay/ch08/
File  Edit  View  Terminal  Tabs  Help
[root@EMACH1 bookcode]# cat fork2.c
#include        <stdio.h>
main() {
        printf( "\n%i\n", getpid() );
        fork();
        printf( "%i\n", getpid() ); }

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

6749
6750
6749
[root@EMACH1 bookcode]#
```

process id # (respective)

double output (but non-identical)
6749 is parent, 6750 is child

fork - how to self-identify?

```
root@EMACH1:~/class/books/molay/ch08/bookcode
File  Edit  View  Terminal  Tabs  Help
[root@EMACH1 bookcode]# cat fork3.c
#include        <stdio.h>
main() {
        int result;
        printf( "\n%i\n", getpid() );
        result = fork();
        printf( "%i  - got %i\n", getpid(), result ); }

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

6765
6766  - got 0
6765  - got 6766
[root@EMACH1 bookcode]#
```

fork tells me

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

root@EMACH1:~/class/books/molay/ch08/bookcode

File Edit View Terminal Tabs Help

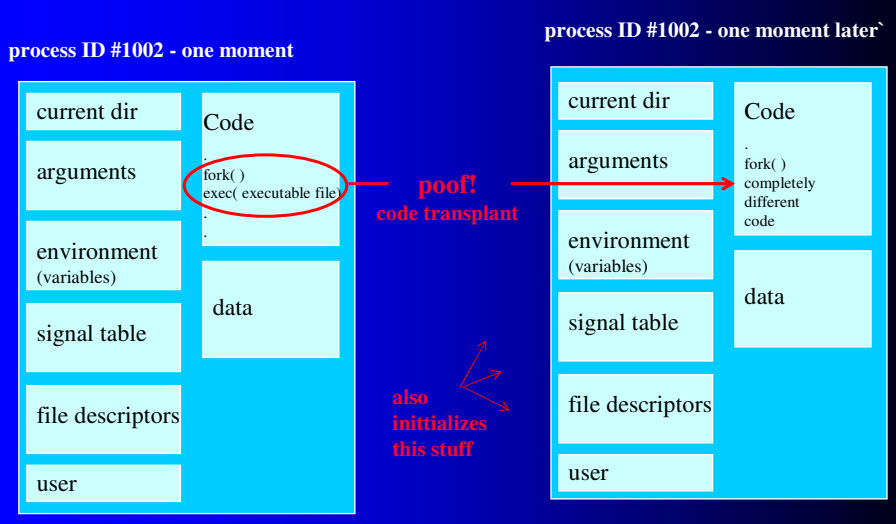
```
[root@EMACH1 bookcode]# cat fork4.c
#include        <stdio.h>
main() {
        int result;
        printf( "\nParent does stuff and then...\n\n" );
        result = fork();
        if ( result == 0 )
                printf("Child can do one thing...\n");
        else
                printf("...parent something completely different.\n\n"); }

[root@EMACH1 bookcode]# gcc fork4.c -o fork4
[root@EMACH1 bookcode]# ./fork4

Parent does stuff and then...

Child can do one thing...
...parent something completely different.

[root@EMACH1 bookcode]#
```

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( )



Provide new behavior from binary code

```
root@EMACH1:~/class/books/molay/ch08/bookcode
File  Edit  View  Terminal  Tabs  Help
[root@EMACH1 bookcode]# cat fork5.c
#include        <unistd.h>
#include        <stdio.h>
main() { int result;
        printf( "\nParent does stuff and then...\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...

total 60
-rw-r--r--  1 root root 32809 May 23 05:14 httpd.conf
-rw-r--r--  1 root root 12958 May 23 05:14 magic
...parent do something completely different.

[root@EMACH1 bookcode]#
```

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