

# Processes

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

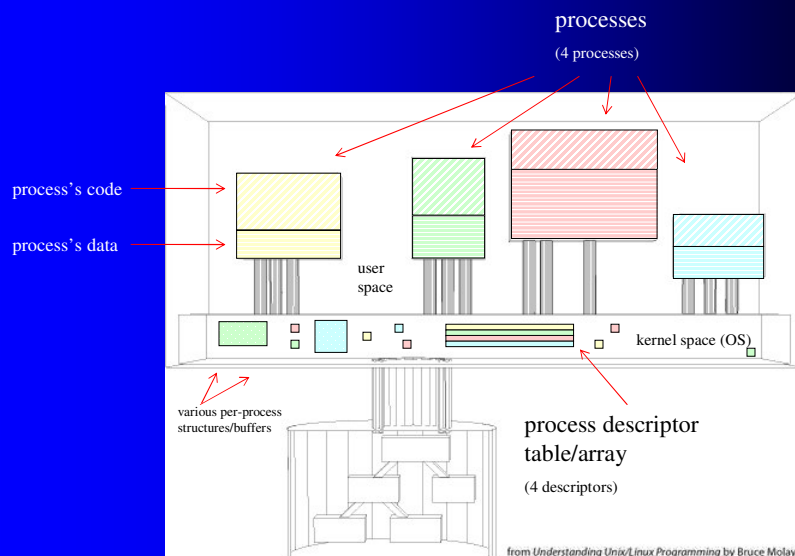
- The associated data needed... (variables, work space, buffers, etc)

- The execution context of the program

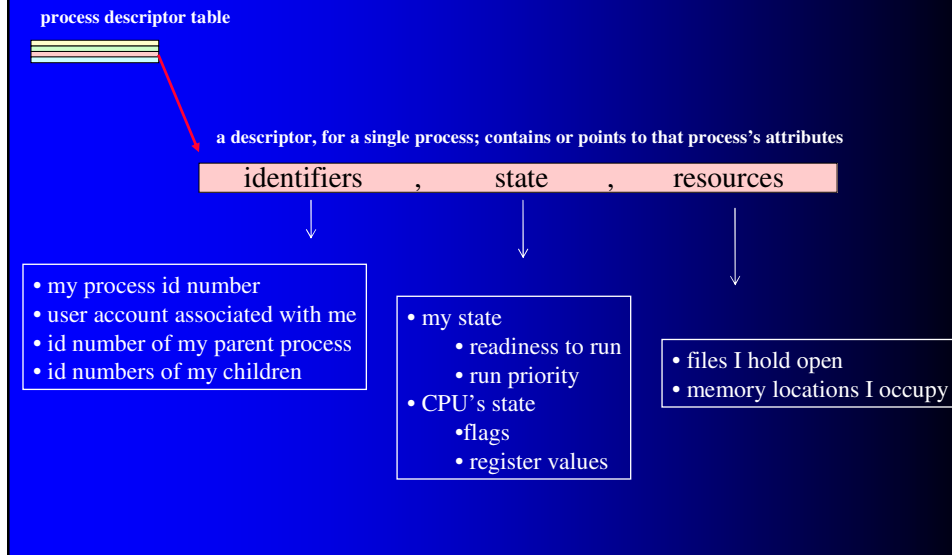
This last element is essential. The execution context, or process state, includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly.... Thus, the process is realized as a data structure [called the process control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

Operating Systems, Internals and Design Principles, William Stallings

## Process table tracks the processes



# Process descriptor tracks a process

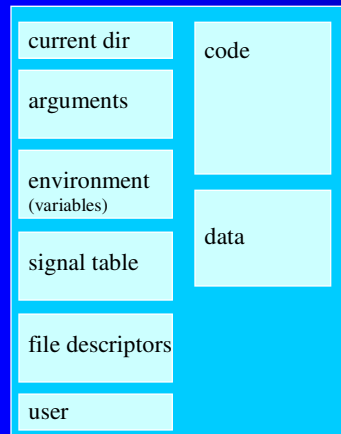


## Process descriptor's role

“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

/home/david

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

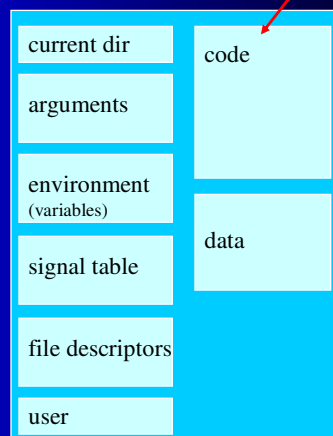
UID=500    LINES=25    PATH=...

1	default
2	handler pointer
3	ignore

0	stdin
1	stdout
2	stderr
3	/bin/ls

joe

process ID #1234



came from  
/bin/ls

# Windows process list

Image Name	User Name	CPU	Mem Usage
services.exe	SYSTEM	00	4,000 K
SMAgent.exe	SYSTEM	00	1,516 K
smss.exe	SYSTEM	00	372 K
SMTray.exe	Daniel	00	2,268 K
spoolsv.exe	SYSTEM	00	5,228 K
svchost.exe	SYSTEM	00	4,032 K
svchost.exe	NETWORK SERVICE	00	3,212 K
svchost.exe	LOCAL SERVICE	00	4,708 K
svchost.exe	SYSTEM	00	4,920 K
svchost.exe	NETWORK SERVICE	00	4,132 K
svchost.exe	SYSTEM	00	18,760 K
System	SYSTEM	00	276 K
System Idle Process	SYSTEM	98	16 K
taskmgr.exe	Daniel	00	2,508 K
TeaTimer.exe	Daniel	00	42,840 K
vmnat.exe	SYSTEM	00	1,828 K
vmnetdhcp.exe	SYSTEM	00	1,552 K
vmount2.exe	SYSTEM	00	4,400 K
vmserverdWin32...	SYSTEM	00	20,436 K

Processes: 47    CPU Usage: 4%    Commit Charge: 622M / 3434M

Process list as a whole

Individual descriptors (partial representation) of single processes, within the list

# Process creation

- create empty slot in process table
- write a process descriptor and put it there
- read in program code from disk

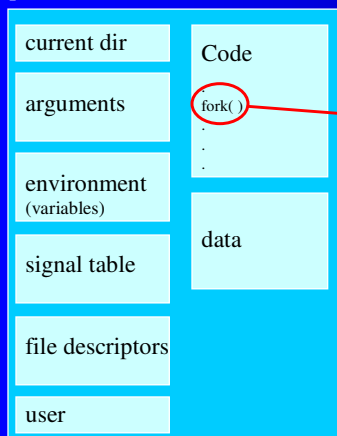
# Process creation in unix

--how can one process spawn another?

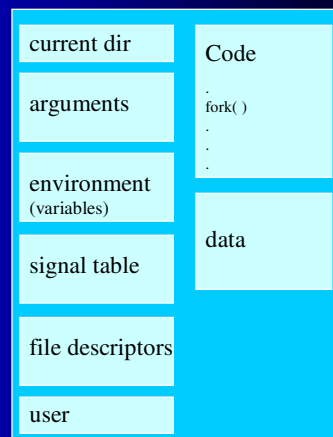
- performed by `fork( )` system call
- creates new process by copying old
- *both* copies then proceed running
  - old copy resumes (after “`fork( )`”)
  - so does new
- new copy is *not* functionally different

## New process creation - `fork( )`

process ID #1001



process ID #1002



same dir!  
same args!  
same vars!  
same sigs!  
same files!  
same user!  
even...  
same code!!

## 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"); } ← single print
                                     function
[root@EMACH1 bookcode]# gcc fork1.c -o fork1
[root@EMACH1 bookcode]# ./fork1 ← single run

How many times do you see this line?
How about this one? } ← but double (identical) output
How about this one? } ← because 2 (identical) processes
                        (the one we ran, the one it ran)
[root@EMACH1 bookcode]#
```

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

[root@EMACH1 bookcode]#
```

process id # (respective)

## 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(); ← conditional, on whether parent or child
    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.

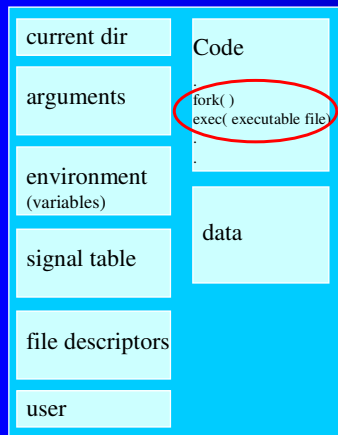
[root@EMACH1 bookcode]#
```

## 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”

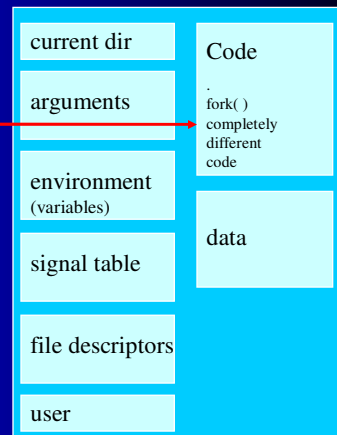
## Making it different - `exec( )`

process ID #1002 - one moment



**poof!**  
code transplant

process ID #1003 - one moment later`



also initializes this stuff

## 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

## Our own homemade shell

- extend our fork5.c program
- give it shell behavior
  - minimal
  - but full-fledged

## Capabilities of the “real” shell

- Command processing
  - parse
  - expand
  - execute
- I/O redirection
- Piping
- Environment control
- Background processing
- Shell scripts

## The dispensible ones (☒)

- Command processing
  - pa☒e
  - ex☒nd
  - execute ← the single essential, for a “command” shell
- I/☒redirection
- Pi☒ng
- En☒ironment control
- Ba☒kground processing
- Sh☒l scripts

## To make a “shell” out of it...

- make it orderly                    timing discipline
- make it interactive                user involvement
- make it repetitive                 as long as he wants

## Make it orderly

insert wait (parent) & exit (child)

```
root@EMACH1:~/class/books/molay/ch08/bookcode
[root@EMACH1 bookcode]# cat fork6.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main() { int result;
printf( "\nParent does stuff and then...\n\n" );
result = fork();
if ( result == 0 )
{
printf("child could run some time-consuming executable...\n\n");
sleep(10); ← waste 10 sec
exit(0);
}
else
printf("...while parent might plunge ahead in parallel...\n");
wait(NULL) ← stall till child is done
printf("...and/or wait for child termination to continue.\n\n");
}
[root@EMACH1 bookcode]# ./fork6
Parent does stuff and then...
Child could run some time-consuming executable...
...while parent might plunge ahead in parallel...
...and/or wait for child termination to continue. ← here, 10 secs pass
[root@EMACH1 bookcode]#
```

# Make it orderly

insert wait (parent) & exit (child)

```
root@EMACH1:~/class/books/molay/ch08/bookcode - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@EMACH1 bookcode]# cat fork6.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main() { int result;
  printf( "\nParent does stuff and then...\n\n" );
  result = fork();
  if ( result == 0 )
  {
    printf("Child could run some time-consuming executable...\n\n");
    execl("timewaster", "timewaster", NULL);
    exit(0);
  }
  else
  {
    printf("...while parent might plunge ahead in parallel...\n");
    wait(NULL);
    printf("...and/or wait for child termination to continue.\n\n");
  }
}

[root@EMACH1 bookcode]# ./fork6

Parent does stuff and then...

Child could run some time-consuming executable...

...while parent might plunge ahead in parallel...
...and/or wait for child termination to continue.

[root@EMACH1 bookcode]#
```

Annotations in the image:

- Red arrow pointing to `wait(NULL);` with text: "stall till child is done"
- Red arrow pointing to `execl("timewaster", "timewaster", NULL);` with text: "prog that wastes 10 sec"
- Red arrow pointing to `exit(0);` with text: "tell parent when done"
- Red arrow pointing to the output line `...and/or wait for child termination to continue.` with text: "here, 10 sec passes"

# Make it interactive

insert scanf( ) for program name

```
root@EMACH1:~/class/books/molay/ch08/bookcode - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@EMACH1 bookcode]# cat fork8.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main() { int result;
  char prog[100];
  printf( "\nParent does stuff and then...\n\n" );
  result = fork();
  if ( result == 0 )
  {
    printf("Child could run a user-specified executable:\n");
    printf("What executable should the child run? ");
    scanf( "%s", &prog);
    execl(prog, prog, NULL);
    exit(0);
  }
  else
  {
    wait(NULL);
    printf("\n...parent continues now, only when child is done.\n\n");
  }
}; }

[root@EMACH1 bookcode]# ./fork8

Parent does stuff and then...

Child could run a user-specified executable:
What executable should the child run? /bin/date
Tue Oct 25 22:32:25 PDT 2005

...parent continues now, only when child is done.

[root@EMACH1 bookcode]#
```

Annotations in the image:

- Red oval around the child's code block: `printf("Child could run a user-specified executable:\n"); printf("What executable should the child run? "); scanf( "%s", &prog); execl(prog, prog, NULL); exit(0);`
- Red arrow pointing to the user input `/bin/date` in the output.



## Make it repetitive

stick it in a while ( ) loop

```
root@EMACH1:~/class/books/molay/ch08/bookcode - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@EMACH1 bookcode]# cat fork9.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
    int result;
    char prog[100];

    printf( "\nWelcome to the Dshell...\n\n" );
    while (1)
    {
        printf("What executable should the child run? ");
        scanf( "%s", &prog);
        result = fork();
        if ( result == 0 )
        {
            execl(prog,prog,NULL);
            exit(0);
        }
        else
        {
            wait(NULL);
            printf("\n...done waiting. Welcome back!\n-----\n\n");
        }
    }
}

[root@EMACH1 bookcode]# ./fork9
Welcome to the Dshell...

What executable should the child run? /bin/date
Tue Oct 25 22:40:13 PDT 2005

...done waiting. Welcome back!
-----

What executable should the child run? /bin/pwd
/root/class/books/molay/ch08/bookcode

...done waiting. Welcome back!
-----

What executable should the child run? █
```

## Limitations of this 15-line shell

- full-path commands only
- single-token commands only (no args!)
- no frilly bash stuff
  - no filename globbing (wildcarding)
  - no history (commandline recall)
  - no redirection/pipes
  - no variables
  - no builtins (cd, exit, etc)