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

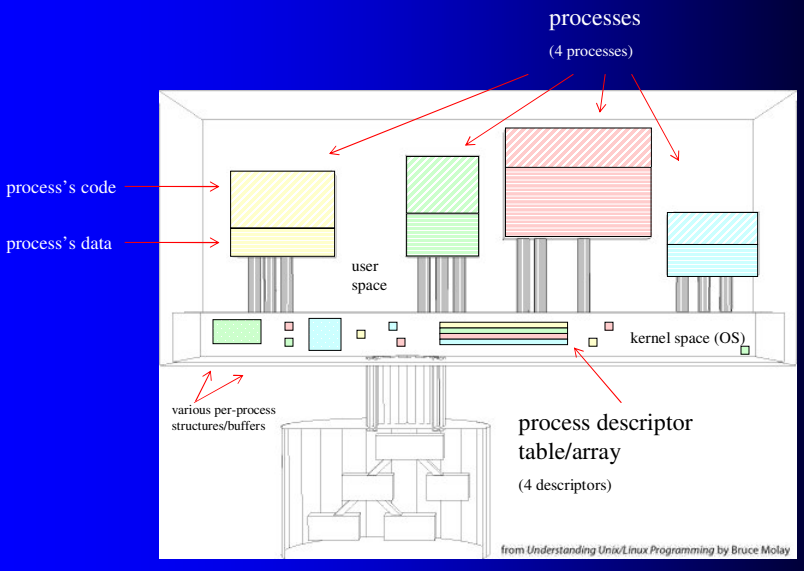
The associated data needed… (variables, work space, buffers, etc)

The execution context of the program

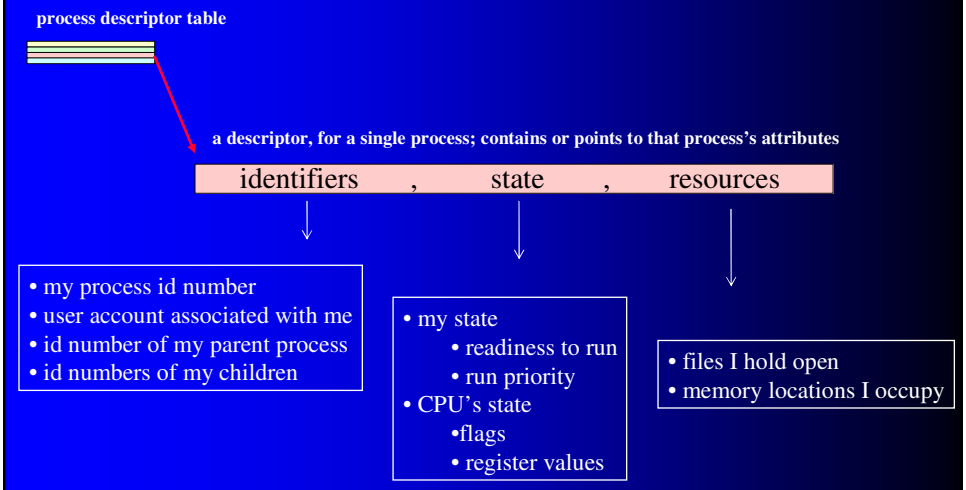
This last element is essential. The execution context, or process state, includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly…. Thus, the process is realized as a data structure [called the process control block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

Operating Systems, Internals and Design Principles, William Stallings

Process table tracks the processes



Process descriptor tracks a process

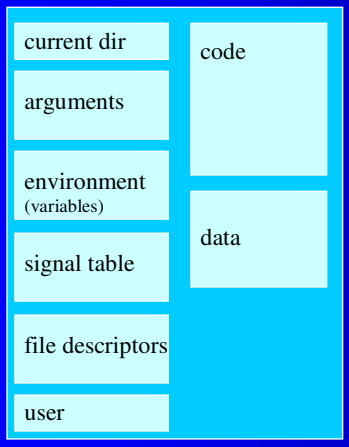


Process descriptor's role

"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

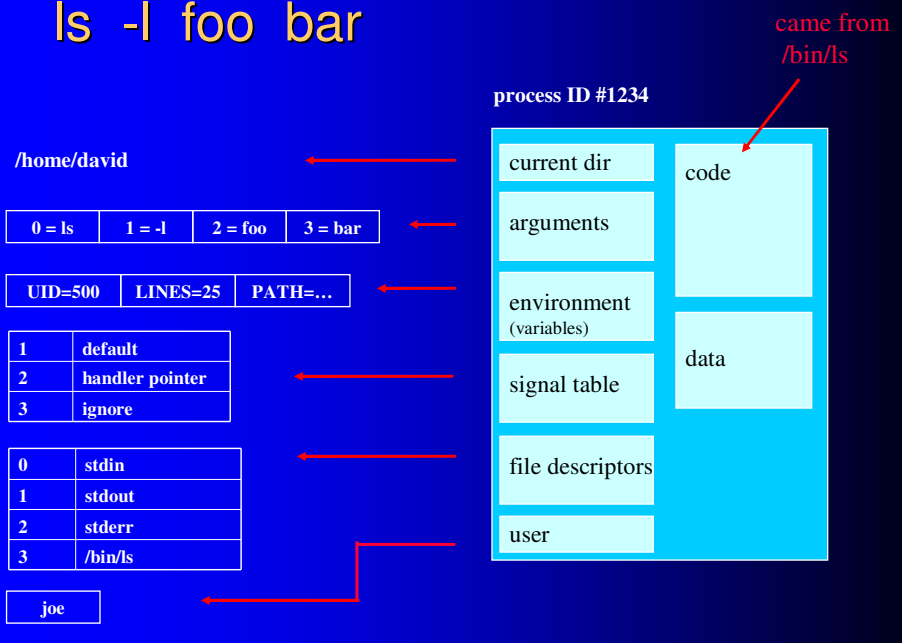
Single process in unix, consolidated view



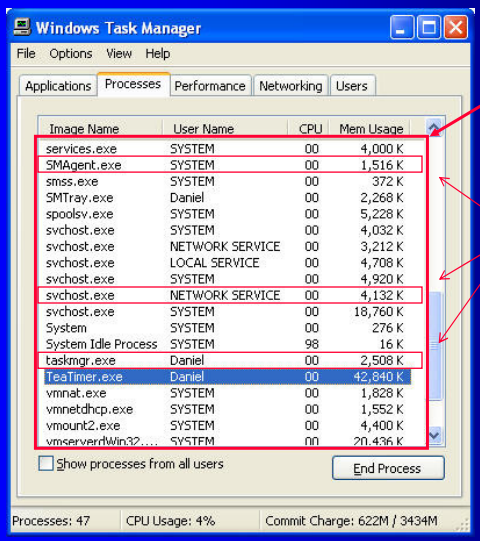
Some important components

- 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



Windows process list



Process list as a whole

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

Process creation

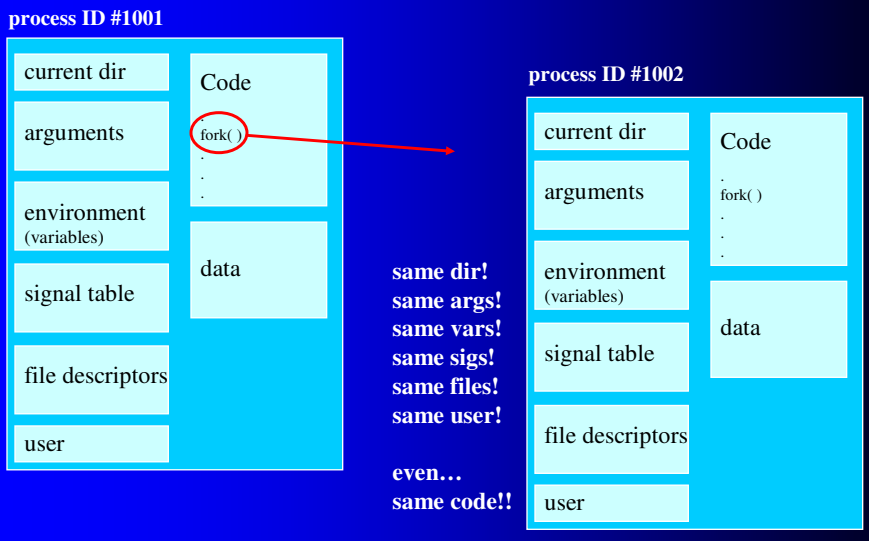
- create empty slot in process table
- write a process descriptor and put it there
- read in program code from disk

Process creation in unix

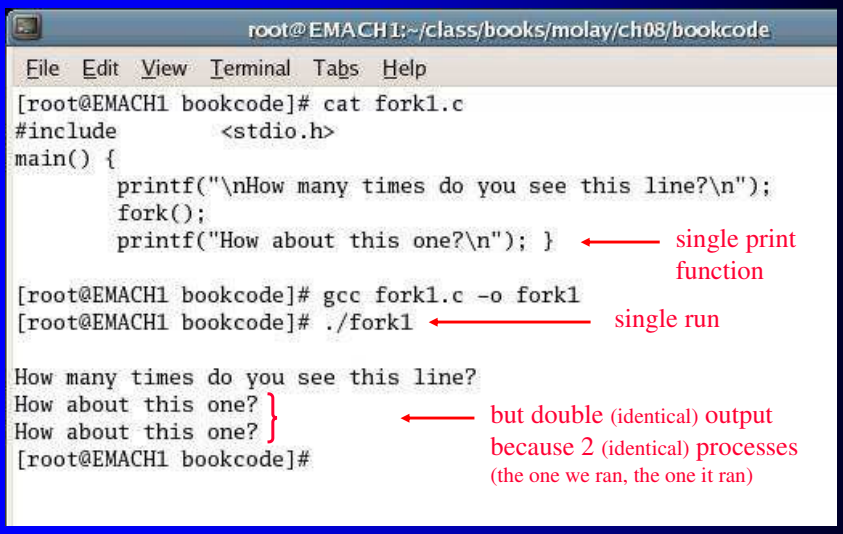
--how can one process spawn another?

- 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( )



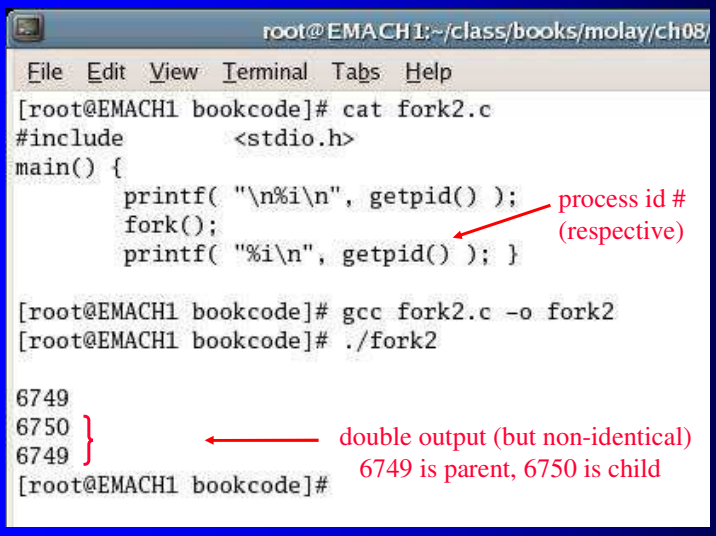
fork - two, where there was one



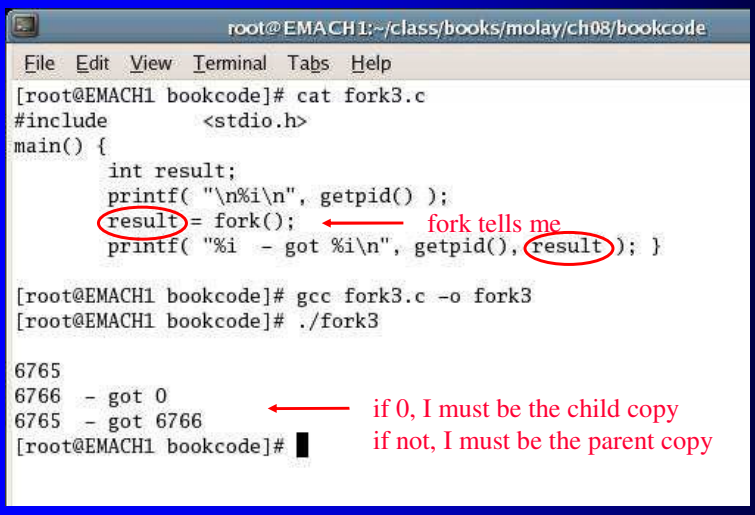
Process differentiation in unix

- 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



fork - how to self-identify?

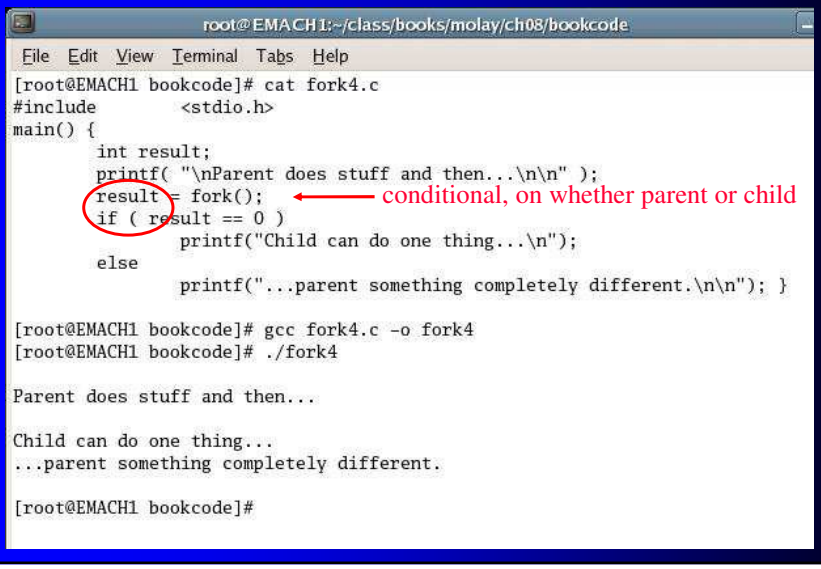


Now provide different behavior

- in the form of source code or
- in the form of an existing binary executable

Provide new behavior

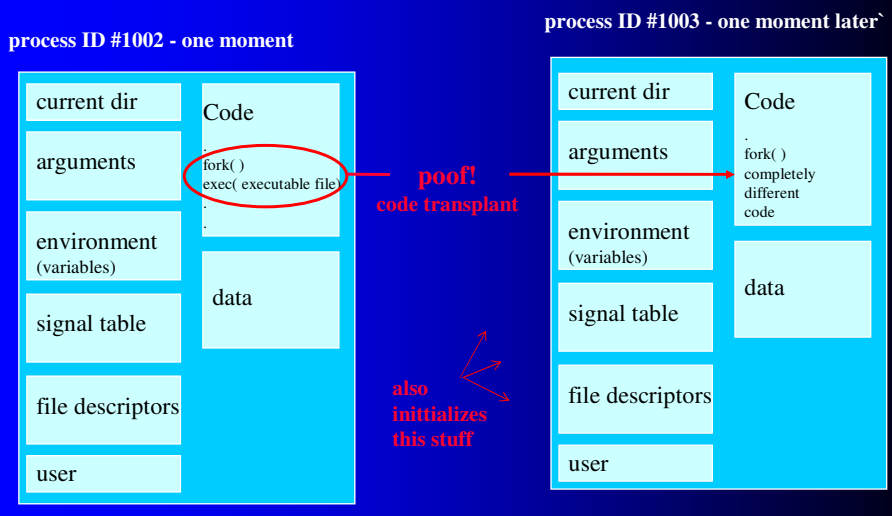
from source code



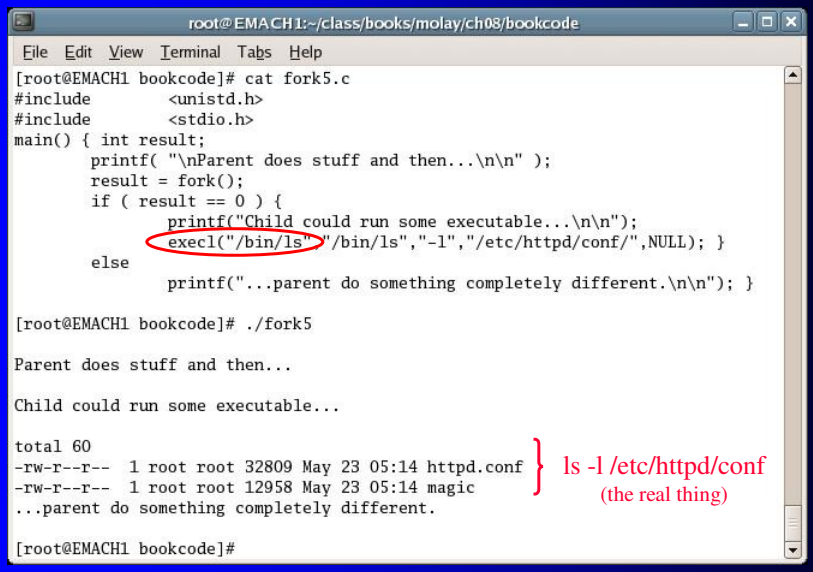
Process differentiation in unix

- 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



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

Capabilitites 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);
                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]#
```

tell parent when done

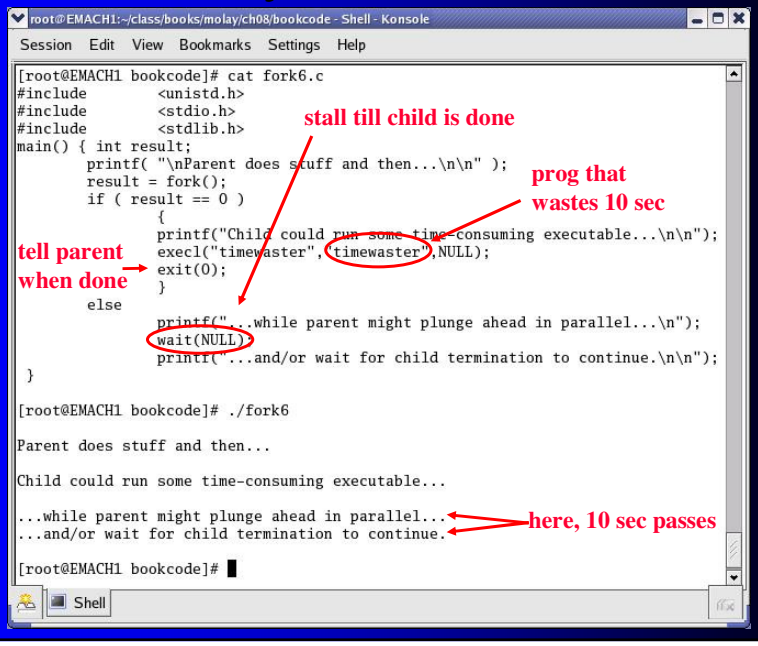
waste 10 sec

stall till child is done

here, 10 secs pass

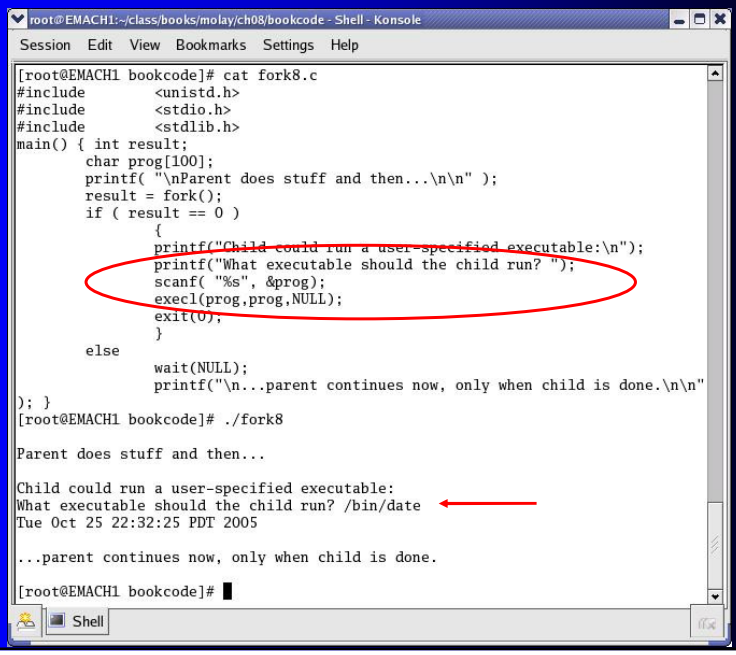
Make it orderly

insert wait (parent) & exit (child)



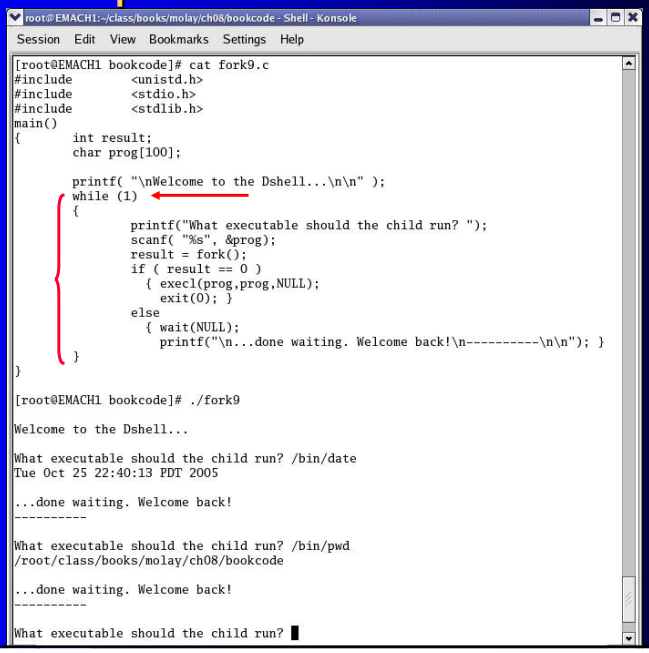
Make it interactive

insert scanf( ) for program name



Make it repetitive

stick it in a while ( ) loop



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)