CH08: Processes and Programs
Studying sh

Objectives

Ideas and Skills

- What a Unix shell does
- The Unix model of a process
- How to run a program
- How to create a process
- How parent and child processes communicate

System Calls, Functions, Commands

- fork
- exec
- wait
- exit
- sh
- ps

8.1: Processes = Programs In Action

Processes = Programs in Action

- A program is a sequence of machine-language instruction stored in a file.
- Running a program means loading that list of instructions into memory, then the CPU executes the instructions, one by one.
- In Unix, an ***executable program*** is a list of machine language and data.

Processes = Programs in Action

- And a ***process*** is the memory space and settings the program uses to run.

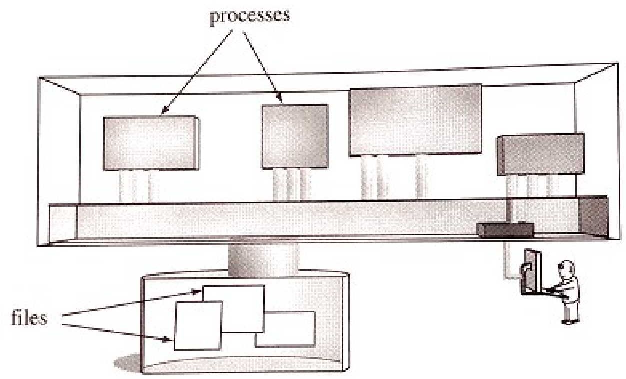


FIGURE 8.1
Processes are programs in action.

Processes = Programs in Action

- Data and programs are stored in files on disk.
- Programs run in processes.
- Let's learn more by playing with ps

8.2: Learning About Processes with *ps*

Learning about processes with ps

- Processes live in ***<u>user space</u>*** – the part of memory where running programs and their data live.

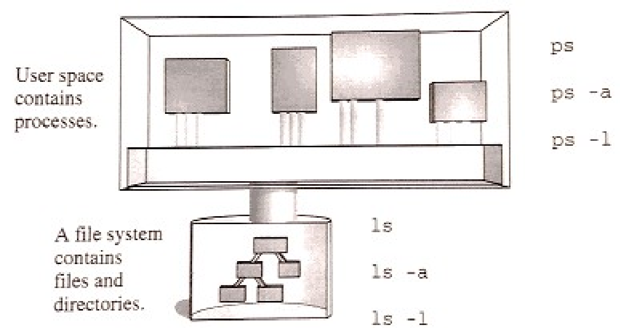


FIGURE 8.2
The ps command lists current processes

Learning about processes with ps

- Let's find out more using *ps*, which lists current processes:

```
hstalica@cat:~$ ps
  PID TTY          TIME CMD
 3037 pts/2    00:00:00 bash
 3168 pts/2    00:00:00 ps
hstalica@cat:~$ 
```

Learning about processes with ps

- Here, we see two processes running: bash (a shell) and ps.
- Each process has it's own id number, called a PID.
- Each process is connected to a terminal, here it's /dev/pts/2.
- Each has an elapsed time.

Learning about processes with ps

- ps has an -a option which shows more processes, including ones run by other users.
- -a doesn't show shells.

```
$ ps -a
  PID TTY          TIME CMD
 1779 pts/0    00:00:13 gv
 1780 pts/0    00:00:07 gs
 1781 pts/0    00:00:01 vi
 2013 pts/2    00:00:23 xpaint
 2017 pts/2    00:00:02 mail
 2018 pts/1    00:00:00 ps
```

Learning about processes with ps

- -l option prints more information:

```
$ ps -la
  F S   UID   PID  PPID  C PRI  NI ADDR   SZ WCHAN  TTY        TIME   CMD
000 S   504  1779  1731  0  69   0  -   1086 do_sel pts/0  00:00:13  gv
000 S   504  1780  1779  0  69   0  -   2309 do_sel pts/0  00:00:07  gs
000 S   504  1781  1731  0  72   0  -   1320 do_sel pts/0  00:00:01  vi
000 S   519  2013  1993  0  69  19  -   1300 do_sel pts/2  00:00:23  xpain
000 S   519  2017  1993  0  69   0  -    363 read_c pts/2  00:00:02  mail
000 R   500  2023  1755  0  79   0  -    750 -      pts/1  00:00:00  ps
```

Learning about processes with ps

- Column S shows status of each process.
- ps is running, indicated by the R.
- Each process belongs to a user, and their ID is listed.
- Each process has it's PID, but we also see the parent's process ID (PPID).

Learning about processes with ps

- PRI and NI indicate priority and niceness levels of a process. The kernel uses them to decide when to run the process.
- A process can increase it's niceness, which allows others to go before it.
- The superuser can decrease the niceness level, pushing ahead of it in line.

Learning about processes with ps

- Processes have a size, shown in SZ column which is the amount of memory taken up.
- The size can change as the process runs (dynamic memory allocation)
- WCHAN shows why a process is sleeping.
- do_sel and read_c indicate they are waiting for input – these are addresses in the kernel.

Learning about processes with ps

- ADDR and F are deprecated.
- These options can work differently from system to system. Check man pages.

System Processes

- Additionally, Unix runs it's own processes to perform system tasks:

```
hstalica@cat:~$ ps -ax | head -25
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:02 /sbin/init splash
    2 ?        S      0:00 [kthreadd]
    3 ?        S      0:00 [ksoftirqd/0]
    5 ?        S<     0:00 [kworker/0:0H]
    6 ?        S      0:00 [kworker/u2:0]
    7 ?        S      0:00 [rcu_sched]
    8 ?        S      0:00 [rcu_bh]
    9 ?        S      0:00 [rcuos/0]
   10 ?        S      0:00 [rcuob/0]
   11 ?        S      0:00 [migration/0]
   12 ?        S      0:00 [watchdog/0]
   13 ?        S<     0:00 [khelper]
   14 ?        S      0:00 [kdevtmpfs]
   15 ?        S<     0:00 [netns]
   16 ?        S<     0:00 [perf]
   17 ?        S      0:00 [khungtaskd]
   18 ?        S<     0:00 [writeback]
   19 ?        SN     0:00 [ksmd]
   20 ?        SN     0:00 [khugepaged]
   21 ?        S<     0:00 [crypto]
   22 ?        S<     0:00 [kintegrityd]
   23 ?        S<     0:00 [bioset]
   24 ?        S<     0:00 [kblockd]
   25 ?        S<     0:00 [ata_sff]
hstalica@cat:~$
```

System Processes

- This shows the first 25 of 163 processes running on my system.
- Some are system processes.
- Many don't have a terminal.
- Some manage memory, others manage system logfiles (klogd, syslogd), etc.

System Processes

- A lot can be learned about a system by reading ps – ax and man pages.
- Using ps allows you to see the number and diversity of processes living in a computer.

Process Management and File Management

- Processes have many attributes: belong to a user ID, have a size, a starting time, elapsed running time, priority, and niceness.
- Where are these properties stored?
- The kernel manages processes in memory and files on the disk.
- How similar are they?

Process Management and File Management

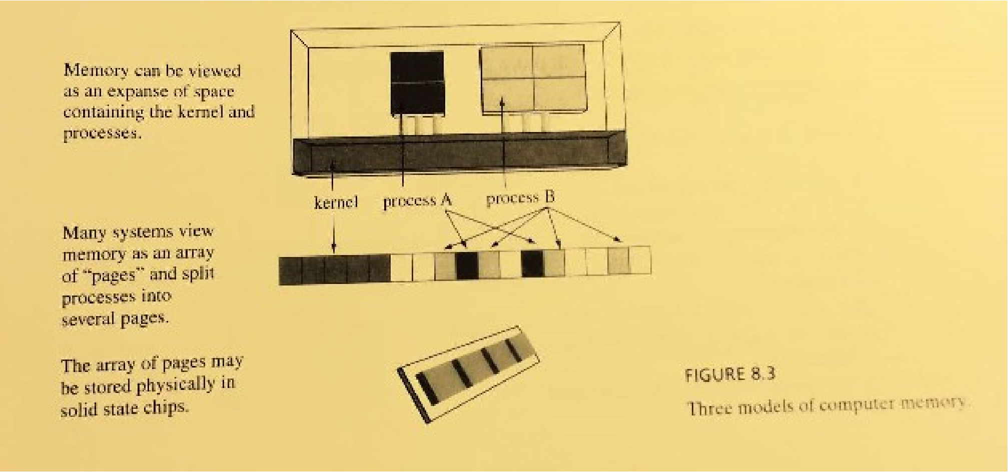
- Files contain data, processes contain executable code.
- Files have attributes, processes have attributes.
- Kernel creates and destroys files, same with processes.
- Kernel stores several processes in memory, and stores several files on disk.

Process Management and File Management

- The kernel has to allocate space and track which processes use which blocks of memory.
- Sound familiar?

Computer Memory and Computer Programs

- A process is an abstract idea, but represents the concrete: bytes in memory.



Computer Memory and Computer Programs

- Unix divides memory into kernel space and user space.
- Processes run in user space.
- Memory is just a big array (like disk blocks).
- A machine with as little as 64 megabytes of memory has an array of ~67 million memory locations.

Computer Memory and Computer Programs

- Some of that is instructions and data that make up the kernel.
- Some of that is instructions and data that compromise processes.
- Processes don't necessarily occupy a single chunk of memory, usually they are divided into smaller chunks. (just like disk files)

Computer Memory and Computer Programs

- Files have an allocation list of disk blocks, processes have a structure to hold the allocation list of ***memory pages***.
- Creating a process is similar to creating a disk file.
- The kernel finds free memory pages to store the instructions and data of the process.
- It then sets up data structures to store memory allocation info and attributes of the process.

8.3 The Shell: A Tool for Process and Program Control

The Shell

- A ***<u>shell</u>*** is a program that manages processes and runs programs.
- There are many shells available, but the popular ones provide three main functions:

  (a) Shells run programs
  (b) Shells manage input and output
  (c) Shells can be programmed

The Shell

- Consider:

```
hstalica@cat:~$ grep lp /etc/passwd
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
hstalica@cat:~$ TZ=PST8PDT ; export TZ ; date ; TZ=EST5EDT
Tue May 17 09:58:10 PDT 2016
hstalica@cat:~$ date
Tue May 17 12:58:20 EDT 2016
hstalica@cat:~$ ls -l /etc > etc.listing
hstalica@cat:~$ NAME=lp
hstalica@cat:~$ if grep $NAME /etc/passwd
> then
>     echo hello | mail $NAME
> fi
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```

Running Programs

- The commands grep, date, ls, echo, and mail are regular programs written in C.
- The shell loads them into memory and runs them.
- A shell can be thought of as a program launcher.

Managing Input and Output

- Using >, <, and |, a user tells the shell to attach input and output of processes to disks files or other processes.

Programming

- The shell is a programming language with variables and flow control.
- TZ is set to a string representing a time-zone.
- That value is used by the date command.
- An *if*...*then* statement is used.
- If the grep command succeeds in searching for "lp" in /etc/passwd, shell executes a command.

8.4: How The Shell Runs Programs

How the Shell Runs Programs

- You see a prompt, you type a command, the shell runs it, you see the prompt again. What's going on?

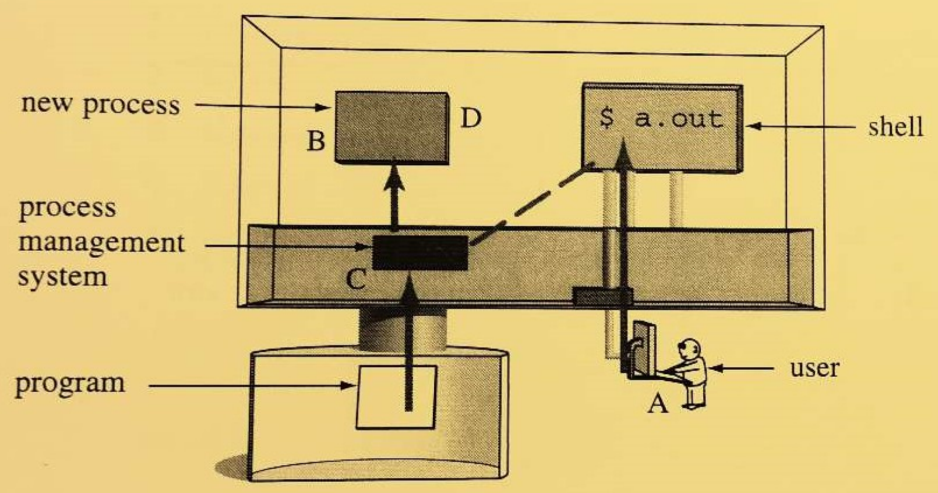


FIGURE 8.4

A user asks a shell to run a program.

How the Shell Runs Programs

- A shell follows the following steps: (fig 8.4)

  A. The user types a.out
  B. The shell creates a new process to run the program.
  C. The shell loads the program from disk into the process.
  D. The program runs in its process until done

- This is the main loop of the shell.

The Main Loop of a Shell

- The shell consists of the following loop:

```
while( ! end_of_input )
    get command
    execute command
    wait for command to finish
```

The Main Loop of a Shell

- Consider this typical shell interaction:

```
hstalica@cat:~$ ls
Desktop    etc.listing       Pictures  Public
Documents  examples.desktop  print.c   Templates
Downloads  Music             print.c~  Videos
hstalica@cat:~$ ps
  PID TTY          TIME CMD
 3037 pts/2    00:00:00 bash
 3220 pts/2    00:00:00 ps
hstalica@cat:~$
```

The Main Loop of a Shell

- Let's represent the events with the time line in fig 8.5:

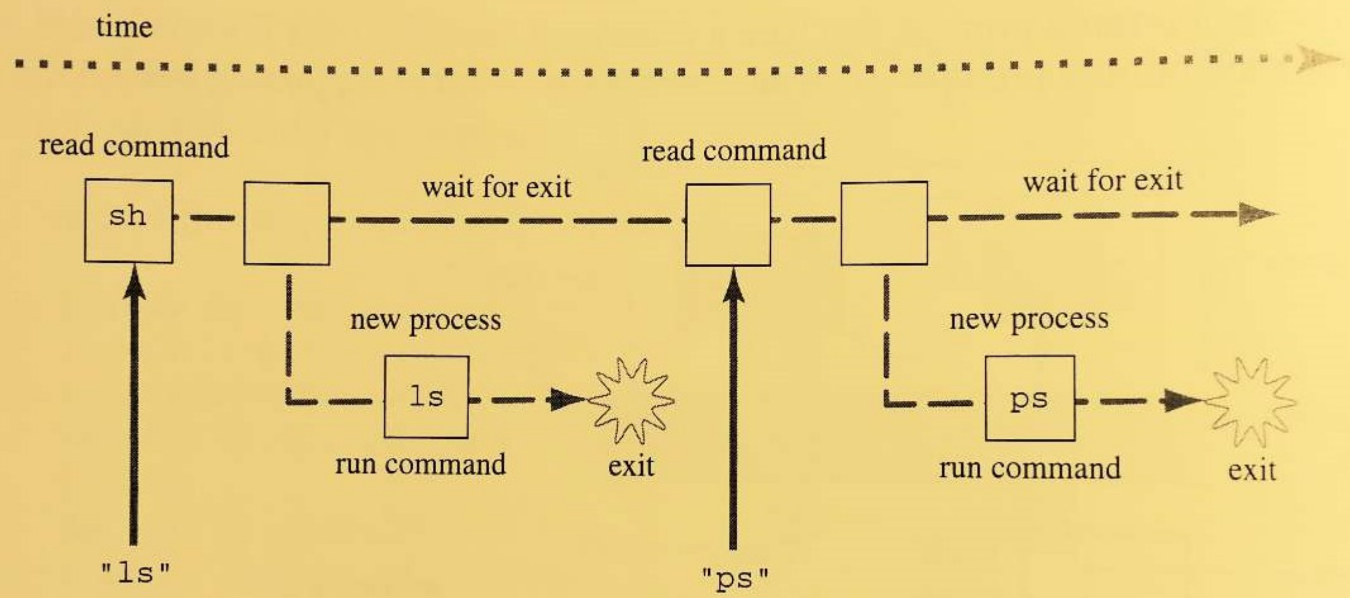


FIGURE 8.5
A time line of the main loop of the shell.

The Main Loop of a Shell

- Time's moving left to right. The shell is the box sh.
- Beginning on the left side, the shell moves right as time passes.
- The shell reads "ls" from the user, creates a new process, then runs the ls command in the process, and waits for it to finish.

The Main Loop of a Shell

- The shell then reads new input, creates a new process, runs the program in that process, then waits for that process to finish.
- When the shell detects end of input, it exits.
- To write a shell, we need to know how to:
  1. Run a program.
  2. Create a process.
  3. Wait for exit()

Q1: How Does a Program Run a Program?

- Answer: call *execvp*.
- Figure 8.6 shows how a program runs another:

How Unix runs programs:

`execvp(progname, arglist)`

1. copies the named program into the calling process,
2. passes the specified list of strings to the program as argv[], then
3. runs the program.

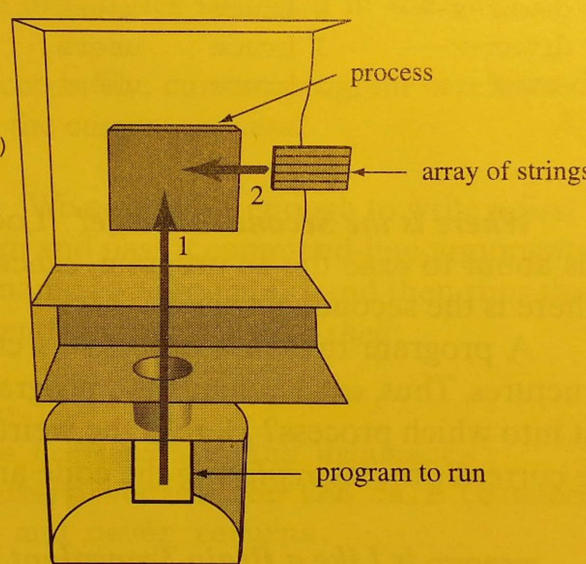


FIGURE 8.6

execvp copies into memory and runs a program.

Q1: How Does a Program Run a Program?

- To run *ls -la*, a program calls
  execvp("ls", arglist)
  where *arglist* is an array of command-line strings.
- The kernel loads the program from disk into memory.
- The command-line arguments *ls* and *-la* are passed to the program, and it runs.

Q1: How Does a Program Run a Program?

- In summary,

  1. Program calls *execvp*.
  2. Kernel loads program into the process.
  3. Kernel copies *arglist* into the process.
  4. Kernel calls *main(argc, argv)*

Q1: How Does a Program Run a Program?

- Let's look at a program that runs *ls -l*, exec1.c.
- Notice *execvp* takes two arguments: the name of the program to run, and the array of command-line arguments for that program.
- The array appears as argv[] when the program runs.

Q1: How Does a Program Run a Program?

- We set the first string to the program name.
- Also notice the array must have a null pointer in the last element.
- Compile and run:

```
hstalica@cat:~/Desktop/CS3560/CH08$ cc exec1.c -o exec1
hstalica@cat:~/Desktop/CS3560/CH08$ ./exec1
* * * About to exec ls -l
total 16
-rwxrwxr-x 1 hstalica hstalica 8632 May 17 10:05 exec1
-rw-rw-rw- 1 hstalica hstalica  426 May 17 10:04 exec1.c
hstalica@cat:~/Desktop/CS3560/CH08$
```

Where's the second message?

- Review the code. The program announced it will *exec ls*, then it *execs ls*, and then announces when *exec* finishes. But where's the second message?
- A program runs in a process, so *execvp* loads the program into a process, but which process?

Where's the second message?

- Well, the kernel loads the new program into the current process, ***replacing*** the code and data of that process!

execvp is Like a Brain Transplant

- The *exec* system call clears out the code of the current program from the current process, then, in the now empty process, puts the code of the program named in the *exec* call, then runs the new program.
- *exec* changes the memory allocation of the process to fit the new program requirements.
- The process is the same, the content is new.

Summary of execvp()

PURPOSE	Execute a file, with PATH searching	
INCLUDE	#include <unistd.h>	
USAGE	result = execvp(const char* file, const char* args[])	
ARGS	file	name of file to execute
	args	array of C-Strings
RETURNS	-1	if error

Summary of execvp()

- *execvp* loads program specified by *file* into the current process and attempts to execute.
- *execvp* passes a list of strings in a NULL-terminated array *argv* to the program.
- *execvp* searches for *file* in directories specified in PATH.
- *execvp* doesn't return if successful. The current program is removed, the new progam executes in the current process.

Q2: How Do We Get a New Process?

- Answer: Have the process call *fork* to replicate itself.
- Usage: *fork();* // takes no arguments

fork Explanation

- The solution is to clone the process that calls *execvp* and then have the clone call *execvp*, leaving the original process alone.
- This way, it's the clone shell that dies, and the original shell lives on.

fork Explanation

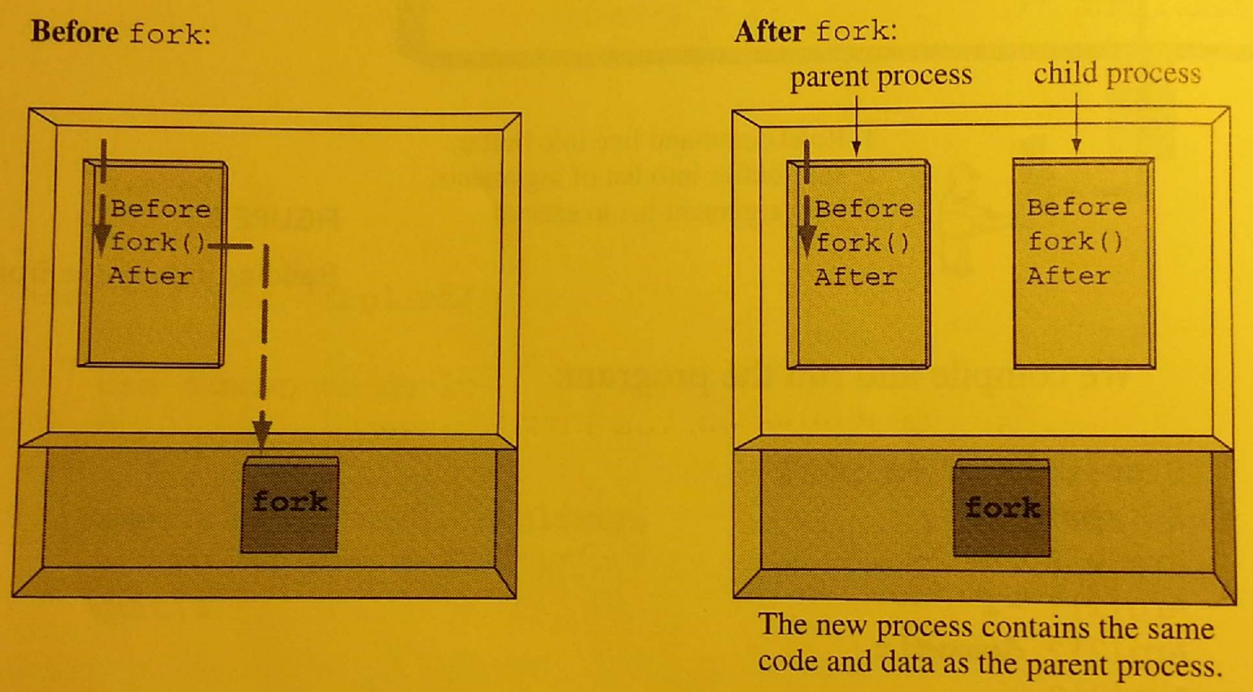


FIGURE 8.8

`fork()` makes a copy of a process.

fork Explanation

- Fig 8.8 shows the system before and after the *fork* call.
- The process contains the program and a current location in the program.
- The process calls *fork*.
- Control passes into the fork code inside the kernel.

fork Explanation

- The kernel does the following:

  (a) allocates a new chunk of memory and kernel data structures
  (b) copies original process into the new one
  (c) adds the new process to the set of running processes
  (d) returns control back to *both* processes

fork Explanation

- After cloning the process, there are two processes, both identical, both in the middle of the same instruction.
- However, each is a separate, new process, each able to continue execution independently.
- Let's consider *forkdemo1.c*

fork Explanation

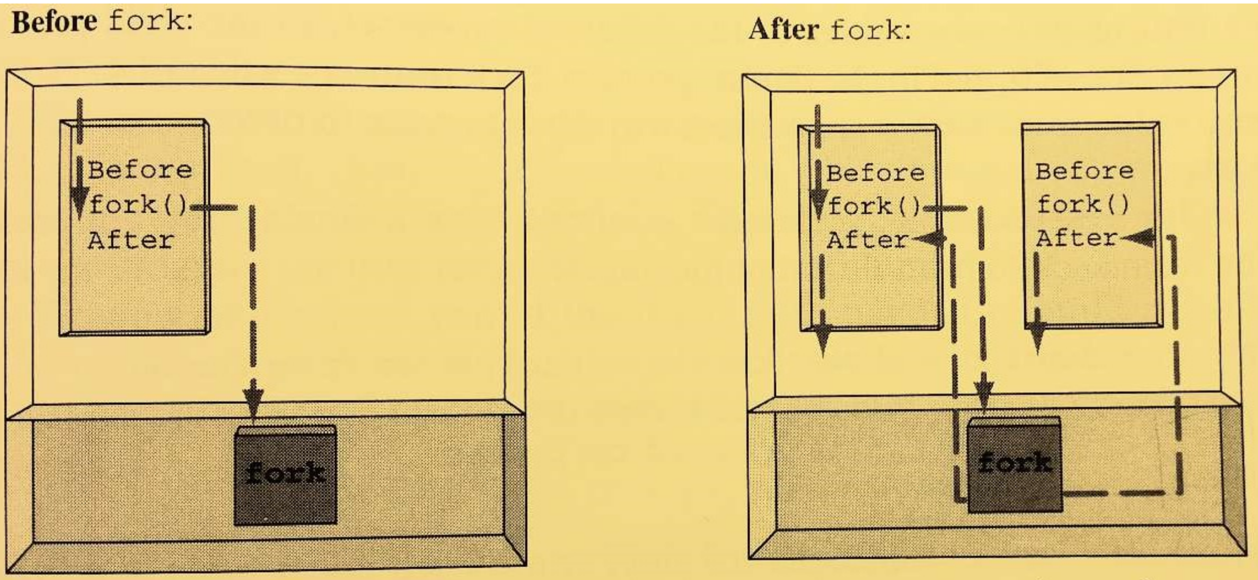
- Were this a normal program, we'd see two lines of output, one for each print statment. However, when run we see

```
hstalica@cat:~/Desktop/CS3560/CH08$ cc forkdemo1.c -o forkdemo1
hstalica@cat:~/Desktop/CS3560/CH08$ ./forkdemo1
Before:  my pid is 3355
After:  my pid is 3355, fork() said 3356
After:  my pid is 3356, fork() said 0
hstalica@cat:~/Desktop/CS3560/CH08$
```

fork Explanation

- We see 3 lines of output, one Before: message and two After: messages.
- Process 3335 prints a Before: and it prints an After:.
- Process 3356 prints the other After: message, but not Before: message. Why?

fork Explanation



One flow of control enters the fork kernel code.

Two flows of control return from fork kernel code.

FIGURE 8.9

The child executes the code after `fork()`.

fork Explanation

- The kernel creates process 3356 by cloning process 3355, copying the code and the *current line* in the code into the new process.
- The new process, 3356, picks up in the middle, and does not print a Before: message.

Example: forkdemo2.c – Children Creating Processes

- The child process begins its life, not at the start of *main* but at the return from fork.
- Can you predict how many lines of output are produced?

Example: forkdemo3.c – Distinguishing Parent from Child

- In *forkdemo1.c*, process 3355 called *fork*, creating child process 3356.
- Both processes ran the same code at the same line with all the same data and process attributes.
- How can a process tell if it's the parent or the child?

Example: forkdemo3.c – Distinguishing Parent from Child

- The two processes are not identical.
- Output from *forkdemo1.c* shows *fork* returns different values to the different processes.
- *fork* returns 0 in the child process.
- *fork* returns the PID of the child process in the parent process (3356 in this case).

Example: forkdemo3.c – Distinguishing Parent from Child

- Checking *fork*'s return value is the easiest way to figure out if you are the child or the parent.
- *forkdemo3.c* shows how a single program uses *fork*'s return value to print different messages.

Example: forkdemo3.c – Distinguishing Parent from Child

- A sample run:

```
hstalica@cat:~/Desktop/CS3560/CH08$ cc forkdemo3.c -o forkdemo3
hstalica@cat:~/Desktop/CS3560/CH08$ ./forkdemo3
Before:  my pid is 3372
I am the parent, my child is 3373
hstalica@cat:~/Desktop/CS3560/CH08$ I am the child.  my pid=3373
```

Summary of *fork*

PURPOSE :	Create a process	
INCLUDE :	#include <unistd.h>	
USAGE :	pid_t result = fork(void)	
ARGS :	None	
RETURNS :	-1	if error
	0	to child process
	pid	pid of child to parent process

Summary of *fork*

- Using *fork*, we create a new process, and can tell which is the new and which is the original process.
- The new process then calls *execvp*.

Q3: How Does the Parent Wait for the Child to Exit?

- Answer: A process calls *wait* to wait for a child to finish.
- Usage: *pid = wait(&status);*

wait Explanation

- *wait* does two things:

    1. it pauses the calling process until the child process finishes.
    2. it retrieves the value the child process passed to *exit*.

wait Explanation

- Fig 8.10 shows us how it works:

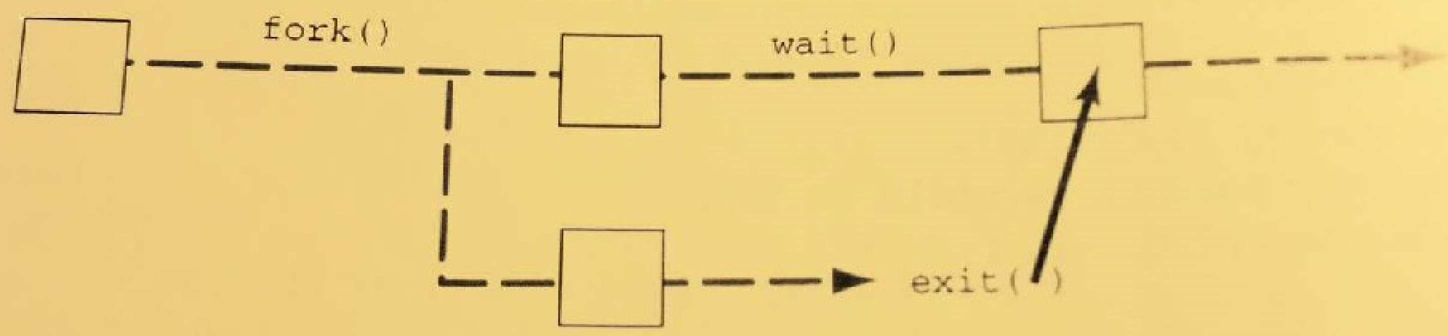


FIGURE 8.10

wait pauses the parent until the child finishes.

wait Explanation

- Time flows left to right.
- The parent begins at the left and calls *fork*.
- The kernel constructs the child process and starts running it concurrently with it's parent.
- The parent calls *wait* and the kernel suspends the parent until the child finishes
- The parent pauses for the part of the diagram marked *wait*.

wait Explanation

- Eventually, the child process finishes and calls *exit(n)*, passing a number between 0 and 255 as an argument.
- When the child calls *exit*, the kernel wakes up the parent and delivers the argument.
- This notification and transfer is represented by arrow leading from the *exit* parentheses to the parent process.

wait Explanation

- So, *wait* does two things: notification and communication.

Example: waitdemo1.c – notification

- *waitdemo1.c* shows how the *exit* call in the child process triggers a return from *wait* in its parent.
- A sample run:

```
hstalica@cat:~/Desktop/CS3560/CH08$ cc waitdemo1.c -owaitdemo1
hstalica@cat:~/Desktop/CS3560/CH08$ ./waitdemo1
before:  mypid is 3416
child 3417 here.  will sleep for 2 seconds
done waiting for 3417. Wait returned: 3417
4352
hstalica@cat:~/Desktop/CS3560/CH08$ 
```

Example: waitdemo1.c – notification

- Run the program and adjusting the time will show the parent always waits until the child calls *exit*.
- Fig 8.11 shows flow of control and data transfer between the two.
- In the parent, flow control starts in the middle of *main*, continues to the *child_code* function, and ends with the call to *exit*.

Example: waitdemo1.c – notification

- The *exit* call is like a wake up signal to the parent.

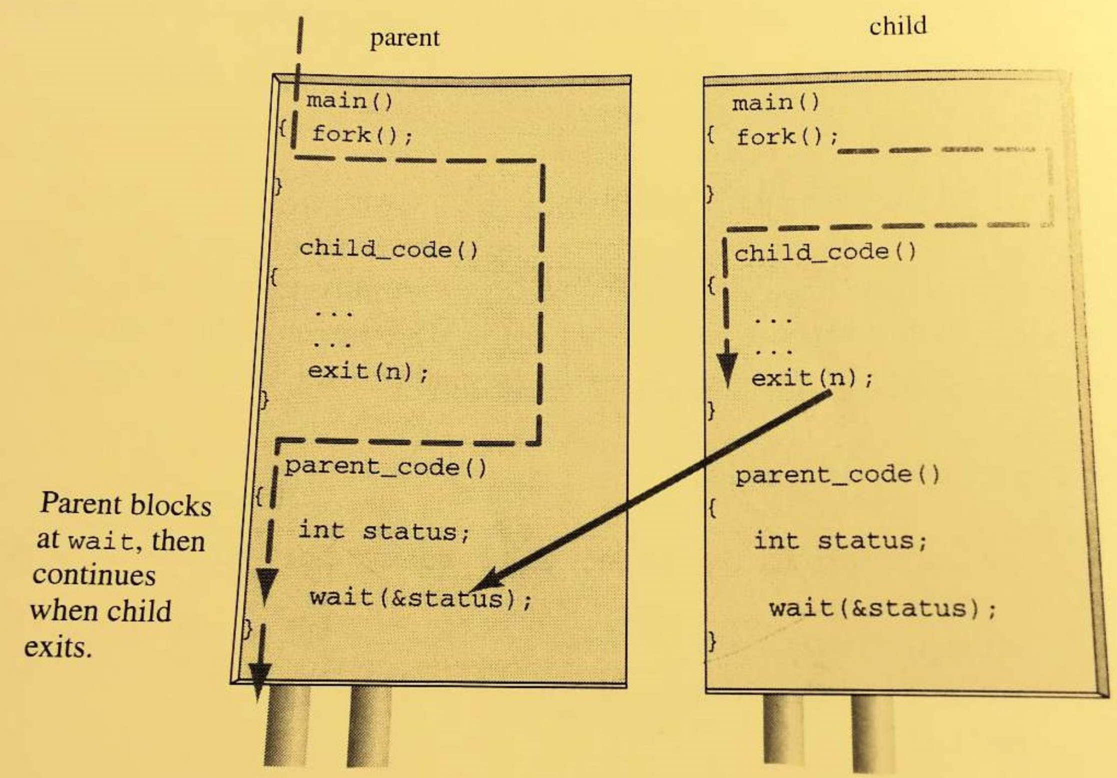


FIGURE 8.11

Control flow and communication with `wait()`.

Conclusions from *waitdemo1.c*

- *wait blocks the calling program until a child finishes.*
  A parent could *fork* a child process to mow the lawn, then wait until it's done.
  The pair of calls *exit* and *wait* is a way to synchronize these tasks.
- *wait returns the PID of the finishing process.*
  Return value from *wait* can tell which of several tasks is finished, so it can continue.

Example: *waitdemo2.c* - communication

- One use of *wait* is to notify the parent *that* a child is done. The other is to tell the parent *how* the child finished.
- A process ends in one of three ways: success (0), failure (nonzero value), death (by signal).
- *wait* returns to the parent the PID of the child process that finished. How does the parent know what happened?

Example: *waitdemo2.c* - communication

- Answer: the argument to *wait*.
- The parent calls *wait* with an integer variable address, the kernel stores in that integer the termination status of the child.
- If the child calls *exit* the kernel puts the exit value into the integer.
- If the child is killed, the kernel puts the signal number into the integer.

Example: *waitdemo2.c* - communication

- The integer consists of 3 regions: 8 bits for exit value, 7 bits for the signal number, and 1 bit to indicate a core dump.

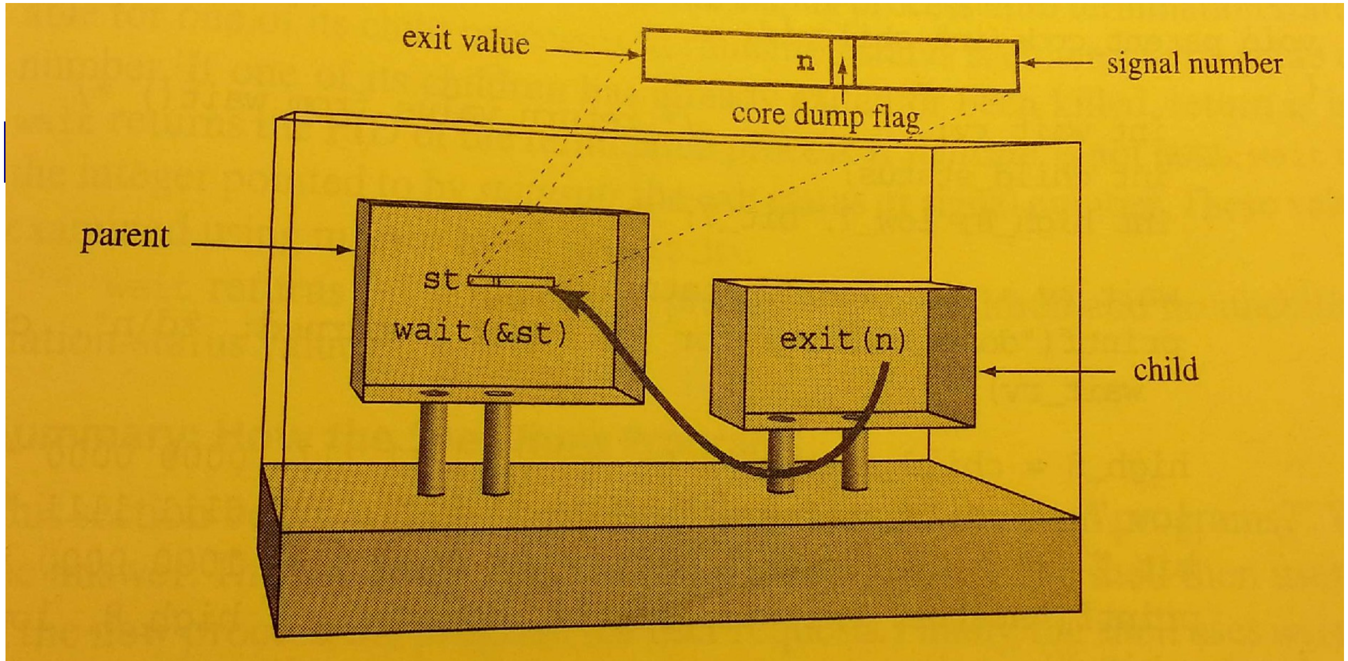


FIGURE 8.12
The child status value has three parts.

Example: *waitdemo2.c* - communication

- Let's look at waitdemo2.c.
- First, let's allow it to exit normally:

```
hstalica@cat:~/Desktop/CS3560/CH08$ cc waitdemo2.c -owaitdemo2
hstalica@cat:~/Desktop/CS3560/CH08$ ./waitdemo2
before:  mypid is 3439
child 3440 here.  will sleep for 5 seconds
child done, about to exit
done waiting for 3440.  Wait returned: 3440
status: exit=17, sig=0, core=0
hstalica@cat:~/Desktop/CS3560/CH08$
```

Example: *waitdemo2.c* - communication

- Then, run it in the background and then use the kill command to send SIGTERM to the child:

```
hstalica@cat:~/Desktop/CS3560/CH08$ ./waitdemo2 &
[1] 3453
hstalica@cat:~/Desktop/CS3560/CH08$ before:  mypid is 3453
child 3454 here.  will sleep for 5 seconds
kill 3454
hstalica@cat:~/Desktop/CS3560/CH08$ done waiting for 3454.  Wait re
turned: 3454
status: exit=0, sig=15, core=0
```

Summary of *wait()*

PURPOSE :	wait for process termination	
INCLUDE :	#include<sys/types.h> #include<sys/wait.h>	
USAGE :	pid_t result = wait(int* statusptr)	
ARGS :	statusptr child result	
RETURNS :	-1	if error
	pid	of terminated process
SEE ALSO :	waitpid (2), wait3 (2)	

Summary of *wait()*

- *wait()* suspends the calling process until termination status is available for one of its child processes.
- Termination status is either an exit value or a signal number.
- If a child has exited or been killed already, return is immediate.

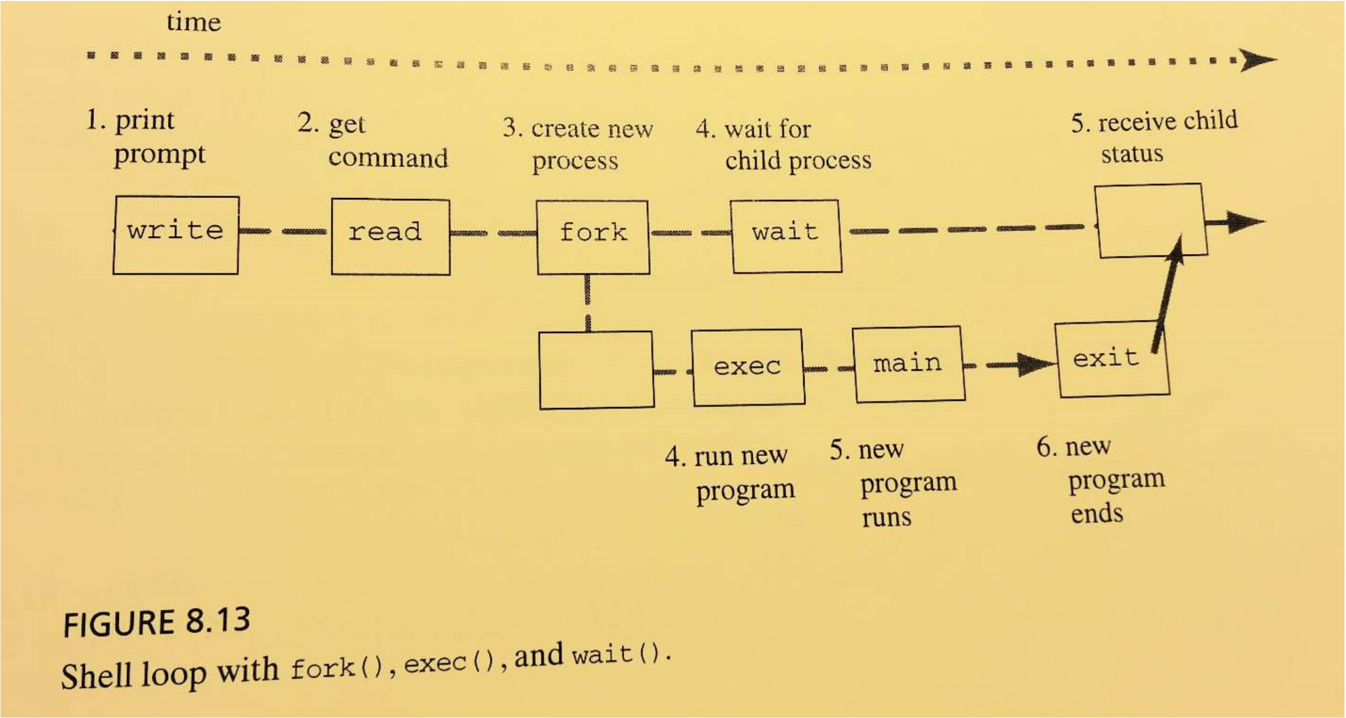
Summary of *wait()*

- *wait* returns the PID of the terminated process.
- If *statusptr* is not NULL, *wait* copies into the integer pointed to by *statusptr* the exit status or signal number.
- These values can be examined using macros in <sys/wait.h>
- *wait* returns -1 if the calling process has no children and no uncollected termination-status values.

Summary: How the Shell Runs Programs

- The shell uses *fork* to create a new process.
- The shell then uses *exec* to run, in the new process, the program the user requests.
- Finally, the shell uses *wait* to wait until the process finishes running the command.
- *wait* also obtains the exit status or signal number telling how the child died from the kernel.

Summary: How the Shell Runs Programs



- Every Unix shell uses this model. Let's combine the 3 calls to write a real shell.

Summary

Summary

- Unix runs a program by loading the executable code into a process and running that code. A process is memory space and other system resources required to run a program.
- Each running program runs in its own process. A process has a unique process ID number, and owner, a size, and other properties.

Summary

- The *fork* system call creates a new process by making an almost exact replica of the calling process. The new process is called a child process.
- A program loads and executes a new program in the current process by calling a function in the *exec* family. We studied *execvp*, but there are others.

Summary

- A program can wait for a child process to terminate by calling *wait*.
- A calling program can pass a list of strings to *main* in the new program. The new program can send back a small integer value by calling *exit*.
- A Unix shell runs programs by calling *fork*, *execvp*, and *wait*.

Free eye bleach