

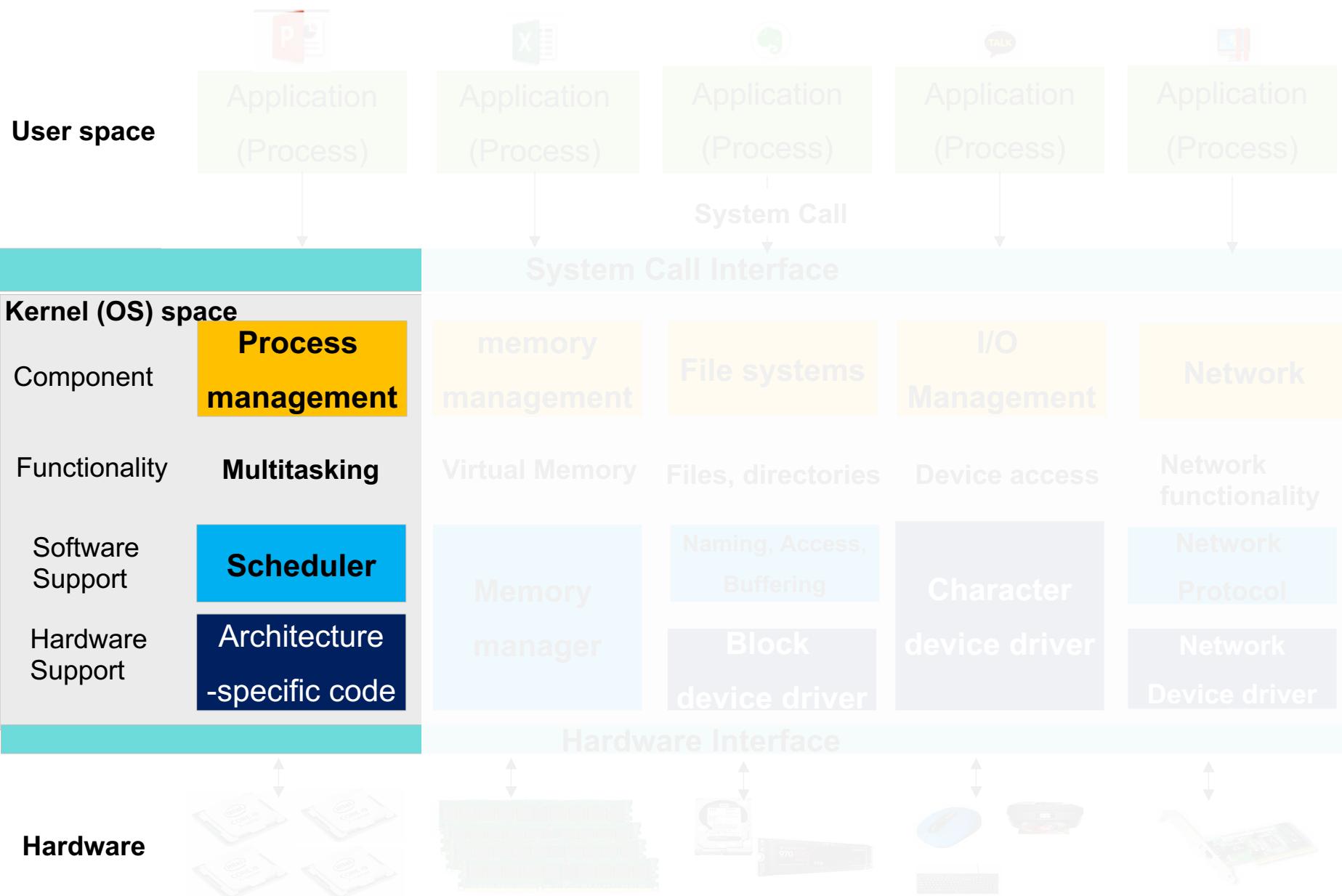
Ch8. Processes and Programs - Studying sh

Prof. Seokin Hong

Kyungpook National University

Fall 2019

What will we learn?



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

- fork, exec, wait, exit

- Commands

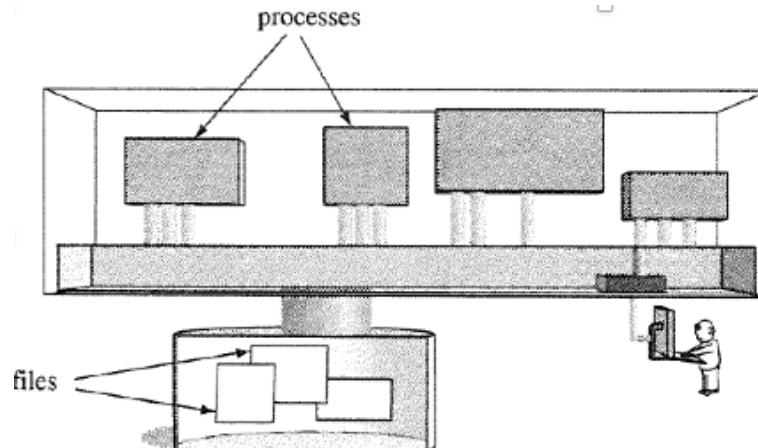
- sh, ps

Contents

- 8.1 Process
- 8.2 Learning about Processes with ps
- 8.3 The Shell:
 A Tool for Process and Program Control
- 8.4 How the Shell Runs Programs
- 8.5 Writing a Shell: psh2.c

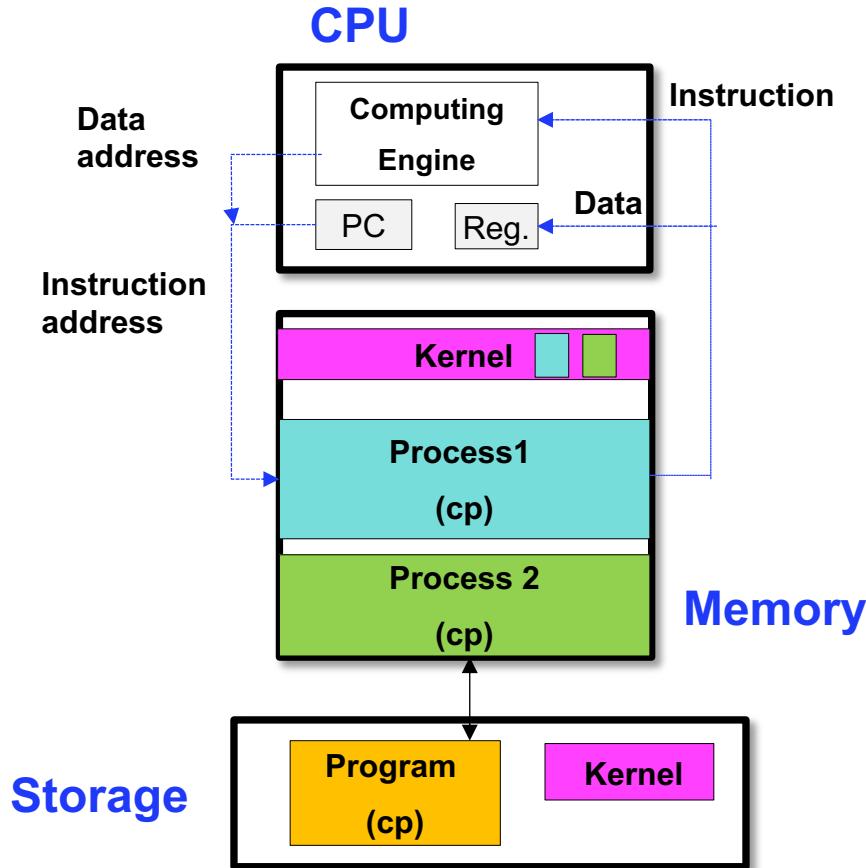
Program and Process

- Program
 - A sequence of machine-language instructions stored in a file
- Running a program means
 - Load that list of machine-language instructions into memory
 - The processor(CPU) execute the instructions one by one
- Process = Programs in action
 - A process is an **instance of a computer program that is being executed**. It is a piece of program that is loaded, being executing and running in the system.



Program and Process

- A **Process** is a program in execution
- We use the term **process**, **job** and **task** interchangeably

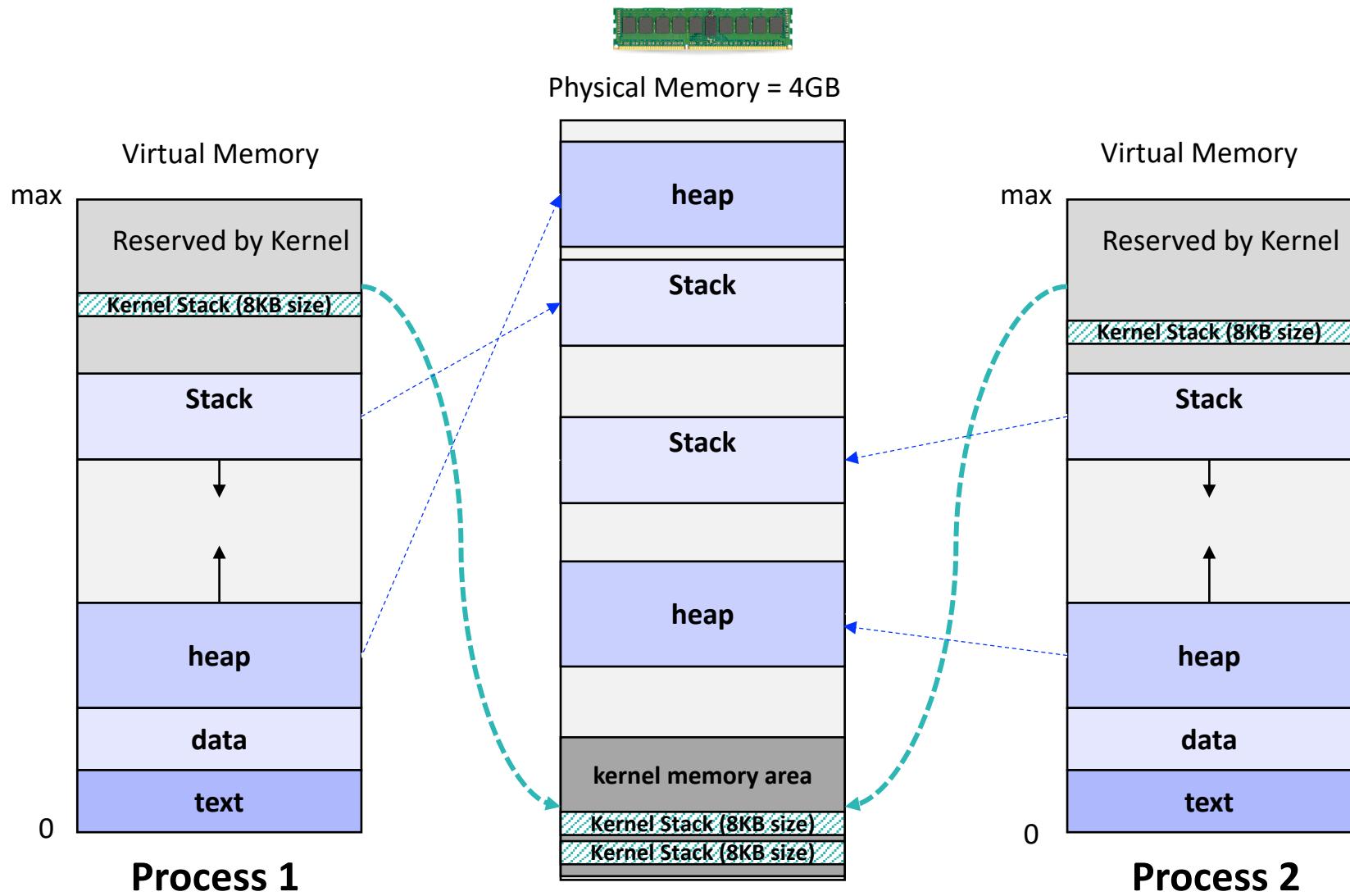


More than one process can exist from a program

```
top - 07:21:54 up 48 days, 17:12, 2 users, load average: 0.71, 1.92, 3.12
Tasks: 800 total, 2 running, 798 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 99.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 10559539+total, 41167129+free, 51635940 used, 59264672+buff/cache
KiB Swap: 10728099+total, 10728099+free, 0 used. 99940678+avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S %CPU	%MEM	TIME+	COMMAND
2527	seokin	20	0	15432	1648	1500	R 100.0	0.0	0:02.87	quickreader
2528	seokin	20	0	42488	4268	3048	R 11.8	0.0	0:00.02	top
43749	seokin	20	0	95996	5616	3456	S 11.8	0.0	30:48.33	sshd
1187	libvirt+	20	0	9428180	1.213g	24324	S 5.9	0.1	28:48.01	qemu-system-x86
14553	libvirt+	20	0	9313412	1.125g	24660	S 5.9	0.1	23:47.58	qemu-system-x86
15845	libvirt+	20	0	9390212	1.156g	24752	S 5.9	0.1	28:45.62	qemu-system-x86
43747	seokin	20	0	95968	5476	3348	S 5.9	0.0	32:30.77	sshd
43750	seokin	20	0	12884	2076	1848	S 5.9	0.0	7:44.60	sftp-server
43751	seokin	20	0	12884	1888	1668	S 5.9	0.0	7:46.85	sftp-server
1	root	20	0	120124	6320	4028	S 0.0	0.0	0:52.91	systemd
2	root	20	0	0	0	0	S 0.0	0.0	0:00.33	kthreadd
3	root	20	0	0	0	0	S 0.0	0.0	0:02.83	ksoftirqd/0
5	root	0	-20	0	0	0	S 0.0	0.0	0:00.00	wkworker/0:0H
6	root	20	0	0	0	0	S 0.0	0.0	0:00.00	lru睡着了

Process Memory Layout



Learning about Processes with “ps” command

- **ps** : list current processes

```
[seokin@compasslab1:~$ ps
  PID TTY          TIME CMD
23467 pts/0    00:00:00 bash
23476 pts/0    00:00:00 ps
```

- **PID**
 - process ID
- **TTY**
 - the name of the console or terminal that the user logged into
- **TIME**
 - the amount of CPU time in minutes and seconds that the process has been running
- **CMD**
 - the name of the command that launched the process

Learning about Processes with “ps” command

■ ps -a

```
[seokin@compasslab1:~$ ps -a
 PID TTY          TIME CMD
 973 tty1        00:00:00 gnome-session-b
 980 tty1        00:01:39 gnome-shell
 999 tty1        00:00:00 Xwayland
1373 tty1        00:00:00 ibus-daemon
1376 tty1        00:00:00 ibus-dconf
1379 tty1        00:00:00 ibus-x11
1416 tty1        00:00:00 gsd-xsettings
1424 tty1        00:00:00 gsd-a11y-settin
1432 tty1        00:00:00 gsd-clipboard
1433 tty1        00:04:04 gsd-color
1436 tty1        00:00:00 gsd-datetime
1438 tty1        00:00:00 gsd-housekeepin
1440 tty1        00:00:00 gsd-keyboard
1441 tty1        00:00:00 gsd-media-keys
1444 tty1        00:00:00 gsd-mouse
1446 tty1        00:00:01 gsd-power
```

:
:
:
:

-a option

- lists more processes, including ones being run by other users and at other terminals
- the output from –a does not include the shell

Learning about Processes with “ps” command

■ ps -al

```
$ 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

- **S** : status of each process.
 - S: sleeping, R : running
- **UID** : user ID
- **PID** : process ID
- **PPID** : parent process ID
- **PRI** : priority
 - higher numbers mean lower priority

- **NI : niceness level**
 - a process with a higher nice number will yield CPU time to other processes on the system.
 - The kernel uses these values, PRI & NI, to decide when to run processes.
- **SZ : process size (KB)**
 - the amount of memory the process is using

Learning about Processes with “ps” command

■ ps –af

```
[seokin@compasslab1:~$ ps -af
UID      PID  PPID  C STIME TTY          TIME CMD
gdm      973  969  0 10月 24 tty1    00:00:00 /usr/lib/gnome-session/gnome-session-b
gdm      980  973  0 10月 24 tty1    00:01:39 /usr/bin/gnome-shell
gdm      999  980  0 10月 24 tty1    00:00:00 /usr/bin/Xwayland :1024 -rootless -ter
gdm     1373  980  0 10月 24 tty1    00:00:00 ibus-daemon --xim --panel disable
gdm     1376  1373  0 10月 24 tty1    00:00:00 /usr/lib/ibus/ibus-dconf
gdm     1379      1  0 10月 24 tty1    00:00:00 /usr/lib/ibus/ibus-x11 --kill-daemon
gdm     1416  973  0 10月 24 tty1    00:00:00 /usr/lib/gnome-settings-daemon/gsd-xse
gdm     1424  973  0 10月 24 tty1    00:00:00 /usr/lib/gnome-settings-daemon/gsd-a11
.....
seokin  23530 23467  0 23:36 pts/0    00:00:00 ps -af
```

- The username is displayed instead of the UID number
- Completed command line is listed in the CMD column

Contents

- 8.1 Process
- 8.2 Learning about Processes with ps
- **8.3 The Shell: A Tool for Process and Program Control**
- 8.4 How the Shell Runs Programs
- 8.5 Writing a Shell: psh2.c

THE SHELL

- A shell is a program that manages processes and runs programs
- Main functions of shells
 - Shells interprets user's input
 - Shells run programs
 - Shells manage input and output
 - Shells can be programmed

THE SHELL

```
[seokin@compasslab1:~$ ls
'[2018-2nd-sy whole directory removed
[2018-2nd-sy whole directory removed
a
add_user.py
a.out
Desktop
[seokin@compasslab1:~$ ps
 PID TTY      TIME CMD
24041 pts/0    00:00:00 bash
24050 pts/0    00:00:00 ps
```

dir1	log	system_programming
Documents	Music	Templates
Downloads	Pictures	test
examples.desktop	play_again1.c	Videos
hw1	Public	workspace
lab_dir_creation	student_id	

■ Running Programs

- The shell loads programs into memory and runs them
- Shell = program launcher

THE SHELL

```
[seokin@compasslab1:~$ ls
'[2018-2nd-sysp-001]HW1CopyaDirectory'      dir1          log          student_id
'[2018-2nd-sysp-001]HW1CopyaDirectory.zip'    Documents      ls.txt        system_programming
a                                         Downloads      Music         Templates
add_user.py                                examples.desktop Pictures      test
a.out                                     hw1           play_again1.c Videos
Desktop                                  lab_dir_creation Public       workspace
[seokin@compasslab1:~$ ls > ls.txt
[seokin@compasslab1:~$ ls | grep P
Pictures
Public
[seokin@compasslab1:~$ ]
```

■ Managing Input and Output

- >, <, | are symbols for input/output redirection
- With the symbols, a user tells the shell to attach the input and output of processes to disk file or to other processes

THE SHELL

```
[seokin@compasslab1:~$ for ((num =0; num <3 ; num++))  
[> do  
[> echo "num:$num"  
[> done  
num:0  
num:1  
num:2
```

-

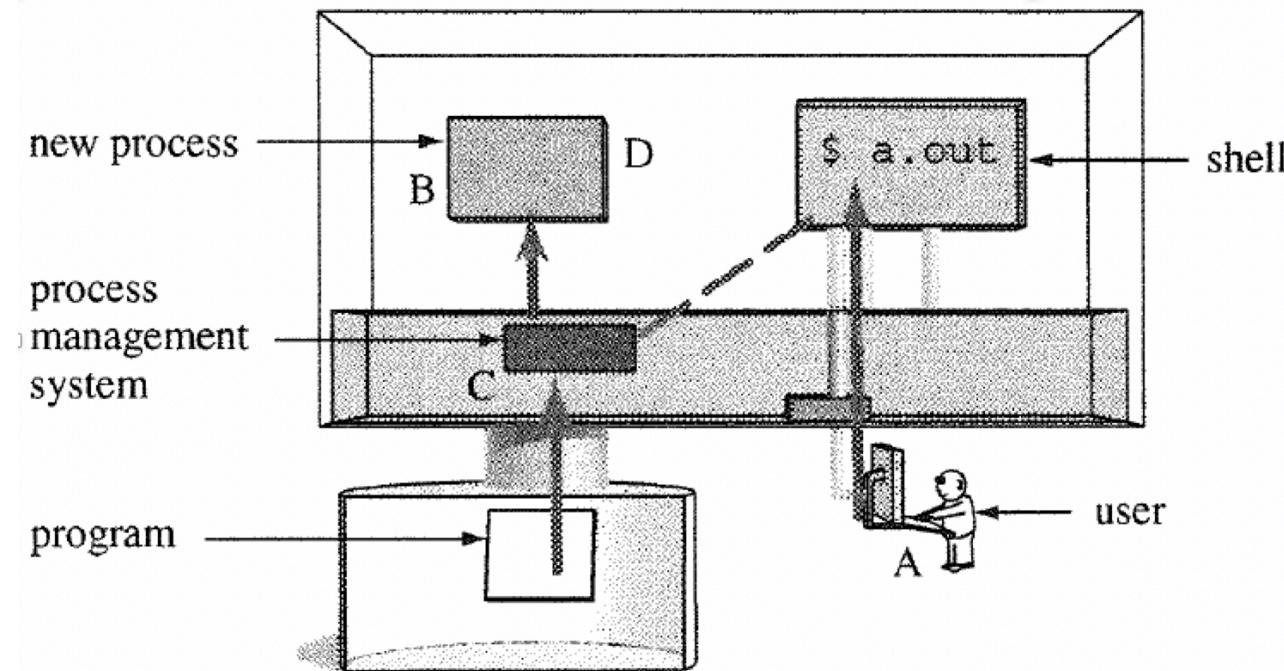
■ Programming

- The shell is also a programming language with variables and flow control (if, while, etc.)
- We can make an executable script (shell script)

Contents

- 8.1 Process
- 8.2 Learning about Processes with ps
- 8.3 The Shell: A Tool for Process and Program Control
- **8.4 How the Shell Runs Programs**
- 8.5 Writing a Shell: psh2.c

How the shell runs programs



- The user types “a.out”
- The shell creates a new process to run the program
- The shell loads the program from the disk into the process
- The program runs in its process until it is done

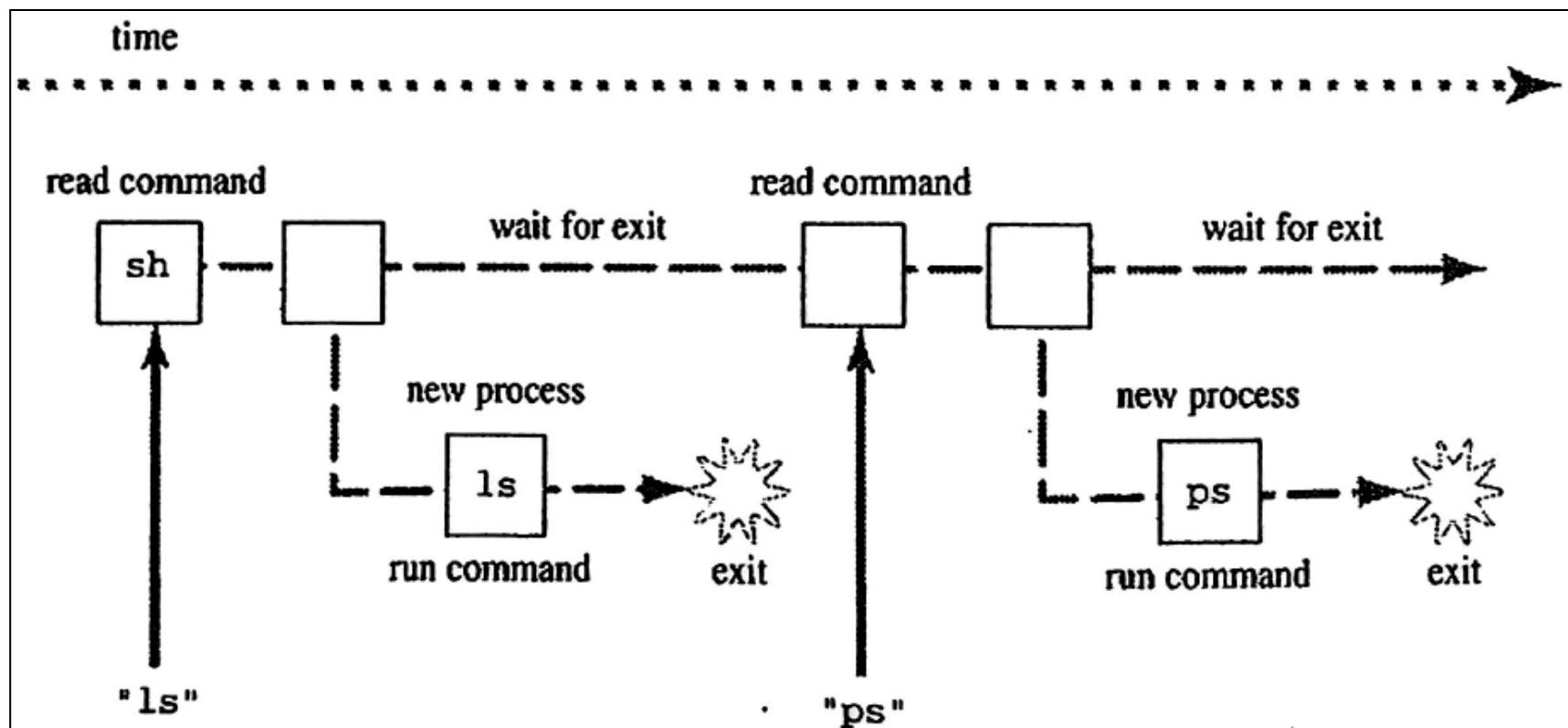
The Main Loop of a Shell

- Shell consists of following loop:

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

To write a shell, we need to learn how to

1. Run a program
2. Create a process
3. Wait for exit()



Q1: How Does a Program Run a Program?

- Answer: The program calls “execvp()” system call
-

execvp

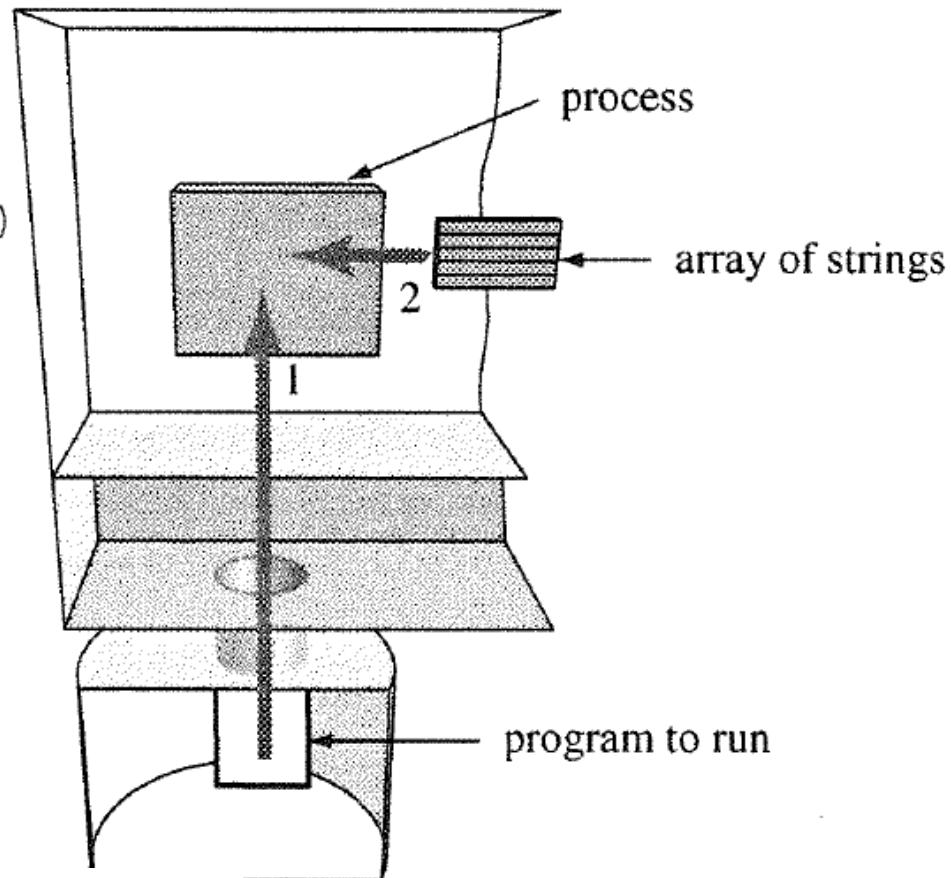
PURPOSE	Execute a file, with PATH searching
INCLUDE	#include <unistd.h>
USAGE	result = execvp(const char *file, const char *argv[])
ARGS	file name of file to execute argv array of strings
RETURNS	-1 if error

Q1: How Does a Program Run a Program?

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.



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

Ex 1-1

```
/* exec1.c - shows how easy it is for a program to run a program
 */
#include <stdio.h>
#include <unistd.h>

main()
{
    char      *arglist[3];
    arglist[0] = "ls";                  //First string is the name of the program
    arglist[1] = "-l";
    arglist[2] = 0 ;                   //Last string must have a null pointer
    printf("* * * About to exec ls -l\n");
    execvp( "ls" , arglist );
    printf("* * * ls is done. bye\n");
}
```

Ex 1-1

```
*** About to exec 'ls -l'
total 60
drwxrwxr-x 2 seokin seokin 4096 9月 12 01:50 lab1
drwxrwxr-x 2 seokin seokin 4096 9月 12 01:50 lab10
drwxrwxr-x 2 seokin seokin 4096 9月 12 01:50 lab11
drwxrwxr-x 3 seokin seokin 4096 9月 12 01:50 lab12
drwxrwxr-x 3 seokin seokin 4096 10月 17 02:30 lab2
drwxrwxr-x 2 seokin seokin 4096 10月 17 04:32 lab3
drwxrwxr-x 3 seokin seokin 4096 10月 16 23:26 lab4
drwxrwxr-x 2 seokin seokin 4096 10月 24 09:27 lab5
drwxrwxr-x 2 seokin seokin 4096 10月 24 07:55 lab6
drwxrwxr-x 3 seokin seokin 4096 10月 31 00:42 lab7
drwxrwxr-x 2 seokin seokin 4096 10月 31 00:40 lab8
drwxrwxr-x 2 seokin seokin 4096 9月 12 01:50 lab9
drwxrwxr-x 2 seokin seokin 4096 9月 12 01:50 lab_dir_create
drwxrwxr-x 2 seokin seokin 4096 10月 9 19:17 mylab5
-rw-rxr-x 1 seokin seokin 3222 9月 12 01:50 utmp.h
```

- Where is the second message?

Ex 1-2

- Compare pid before execvp and after
- pid_t getpid(void)
 - returns the process ID of the current process

```
/* after.c */

#include <unistd.h>
#include <stdio.h>

void main()
{
    pid_t pid = getpid();

    printf("After execvp(): %d\n", pid);

    return;
}
```

```
/* before.c */

#include <unistd.h>
#include <stdio.h>

void main()
{
    char *arg[2];
    arg[0] = "./after";
    arg[1] = 0;

    pid_t pid = getpid();

    printf("before execvp(): %d\n", pid);

    execvp(arg[0], arg);

    return;
}
```

Ex 1-2

■ result

```
[seokin@compasslab1:~/system_programming/labs/lab7$ ./before
Before execvp(): 24317
After execvp(): 24317
[seokin@compasslab1:~/system_programming/labs/lab7$ ./before
Before execvp(): 24318
After execvp(): 24318
[seokin@compasslab1:~/system_programming/labs/lab7$ ./before
Before execvp(): 24319
After execvp(): 24319
[seokin@compasslab1:~/system_programming/labs/lab7$ ./before
Before execvp(): 24320
After execvp(): 24320
[seokin@compasslab1:~/system_programming/labs/lab7$ ./before
Before execvp(): 24321
After execvp(): 24321
```

What does “execvp()” system call do

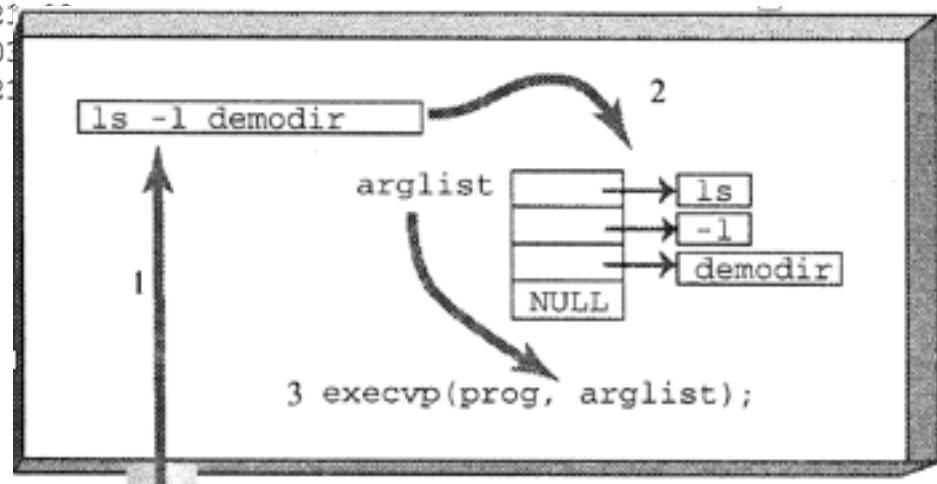
- execvp() system call clears out the machine-language code of the current program from the current process
- Puts the code of the program named in the execvp() system call
- Run that new program
- execvp() changes the memory allocation of the process to fit the space requirements of the new program.
- execvp() does not return if it succeeds
- **The process is the same, the contents are new!**

Ex 2: a prompting shell (psh1.c)

- The first version of a shell

- It will prompts the user for a program name and arguments and then run the program

```
Arg[0]? ls Enter
Arg[1]? -l Enter
Arg[2]? demodir Enter
Arg[3]? Enter
total 2
drwxr-x--- 2 bruce users 1024 Jul 14 21:00
drwxr-x--- 3 bruce users 1024 Jul 16 01:00
-rw-r--r-- 1 bruce users 0 Jul 14 21:00
$
```



1. Read command line into buffer,
2. split buffer into list of arguments,
3. pass argument list to execvp.

Ex 2: a prompting shell (psh1.c)

```
/*
 *      prompting shell version 1
 *          Prompts for the command and its arguments.
 *          Builds the argument vector for the call to execvp.
 *          Uses execvp(), and never returns.
 */

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>

#define MAXARGS          20           /* cmdline args */
#define ARGLEN           100          /* token length */

char* makestring(char*) ;
int execute(char* []);
```

Ex 2: a prompting shell (psh1.c)

```
int main()
{
    char    *arglist[MAXARGS+1];          /* an array of ptrs      */
    int     numargs = 0;                  /* index into array      */
    char    argbuf[ARGLEN];              /* read stuff here       */

    while ( numargs < MAXARGS )
    {
        printf("Arg[%d]? ", numargs);           /* 입력 첫 문자가
                                                '\n' 이 아닐 때 */
        if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
            arglist[numargs++] = makestring(argbuf);
        else //입력 첫 문자가 '\n' 일 때
        {
            if ( numargs > 0 ){                /* any args? */
                arglist[numargs]=NULL;         /* close list */
                execute( arglist );          /* do it */
                numargs = 0;                 /* and reset */
            }
        }
    }
    return 0;
}
```

Ex 2: a prompting shell (psh1.c)

```
int execute( char *arglist[] )
{
    /*
     *      use execvp to do it
     */
{
    execvp(arglist[0], arglist);          /* do it */
    perror("execvp failed");
    exit(1);
}

char * makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
    char      *cp ;
    buf[strlen(buf)-1] = '\0';           /* trim newline */
    cp = malloc( strlen(buf)+1 );         /* get memory */
    if ( cp == NULL ){
        fprintf(stderr,"no memory\n");
        exit(1);
    }
    strcpy(cp, buf);                    /* copy chars   */
    return cp;                          /* return ptr   */
}
```

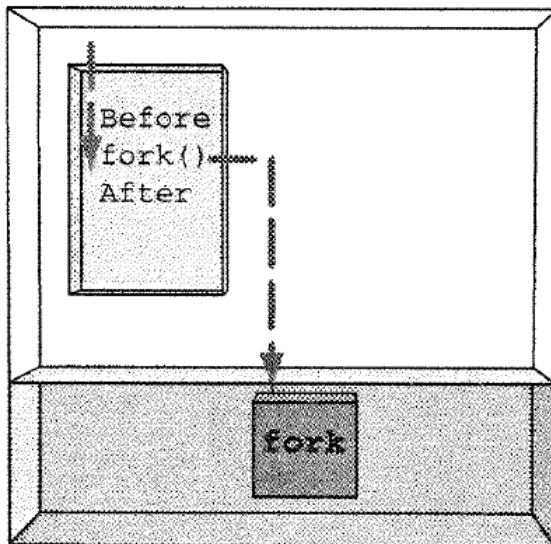
Ex 2: a prompting shell (psh1.c)

- How'd it Go?
 - The program works OK
 - but execvp replaces the code of the shell with the code of the command, then exits,
- A solution is to create a new process and have that new process execute the program.

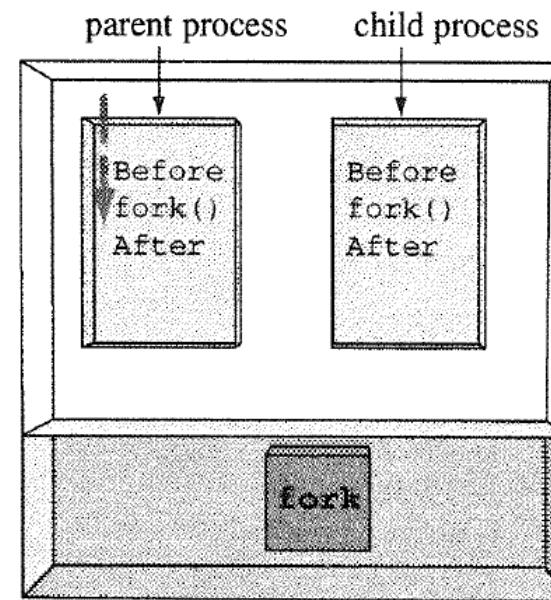
Q2: How Do We Get a New Process?

- Ans: A process calls fork() system call to replicate itself.

Before fork:



After fork:



The new process contains the same code and data as the parent process.

FIGURE 8.8

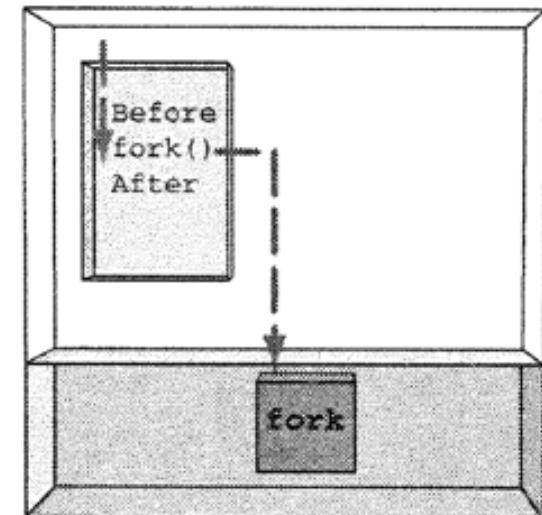
fork() makes a copy of a process.

fork() system call

- The process contains the program and a current location in the program.
- The process then calls fork().
- Control passes into the fork code inside the kernel.

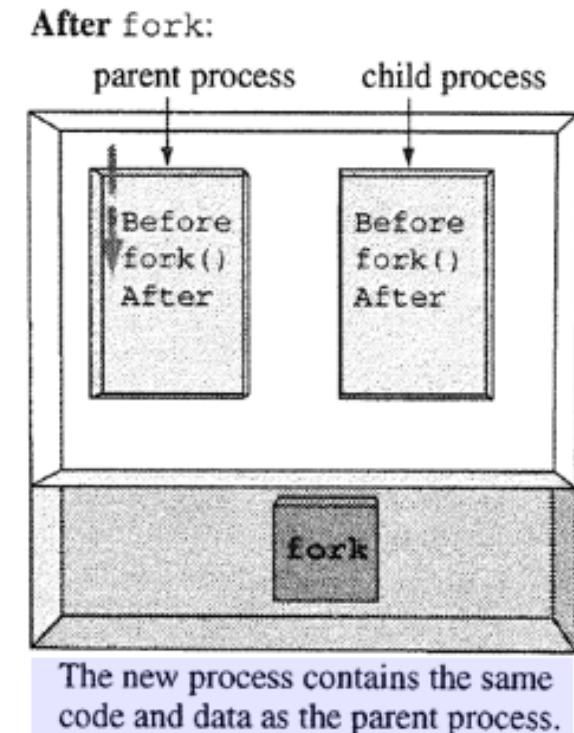


- The kernel does this:
 - (a) **Allocates** a new chunk of memory and kernel data structures.
 - (b) **Copies** the original process into the new process
 - (c) **Adds** the new process to the set of running processes
 - (d) **Returns** control back to both processes



fork() system call

- After a process calls fork, there are two separate process,
 - both digitally **identical**,
 - both in the middle of the same code,
 - but each a separate process can go its own way.



Ex3 - Creating a new process Example : forkdemo1.c

```
/* forkdemo1.c
 * shows how fork creates two processes, distinguishable
 * by the different return values from fork()
 */
#include <stdio.h>
#include <unistd.h>
main()
{
    int      ret_from_fork, mypid;

    mypid = getpid();                                /* who am i? */
    printf("Before: my pid is %d\n", mypid);        /* tell the world */

    ret_from_fork = fork();

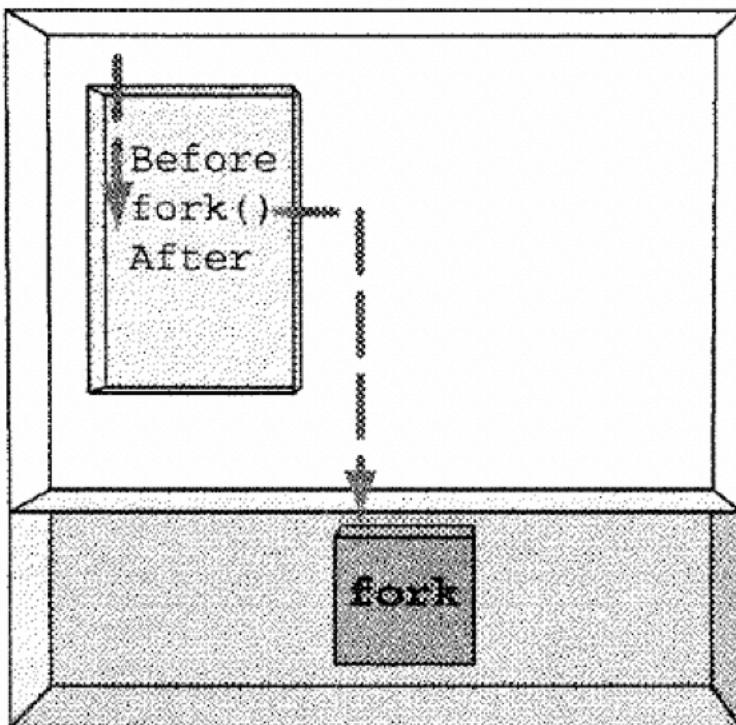
    sleep(1);
    printf("After: my pid is %d, fork() said %d\n",
           getpid(), ret_from_fork);
}
```

```
$ cc forkdemo1.c -o forkdemo1
$ ./forkdemo1
Before: my pid is 4170
After: my pid is 4170, fork() said 4171
$ After: my pid is 4171, fork() said 0
```

Three lines!... Why?

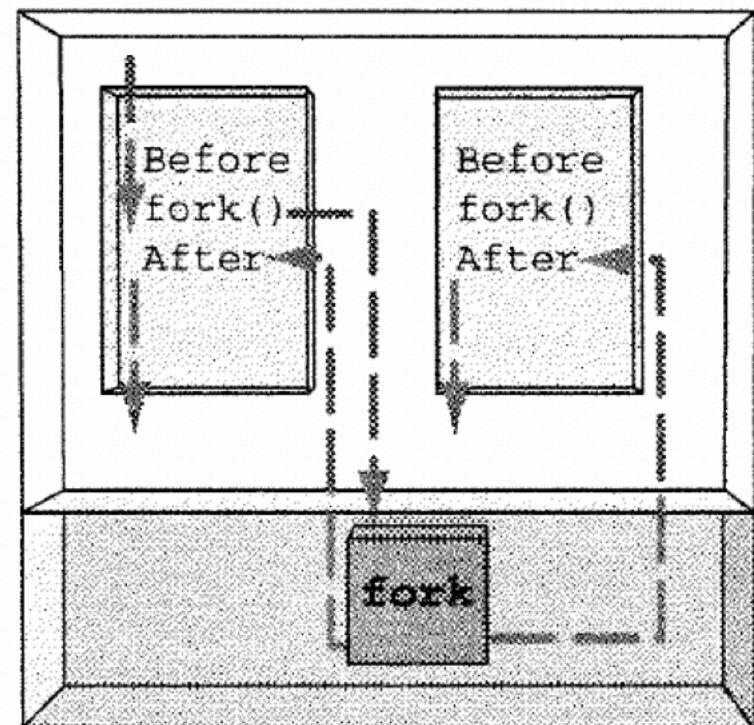
Ex3 - Creating a new process Example : forkdemo1.c

Before fork:



One flow of control enters the fork kernel code.

After fork:



Two flows of control return from fork kernel code.

Ex4 - Children creating process example : forkdemo2.c

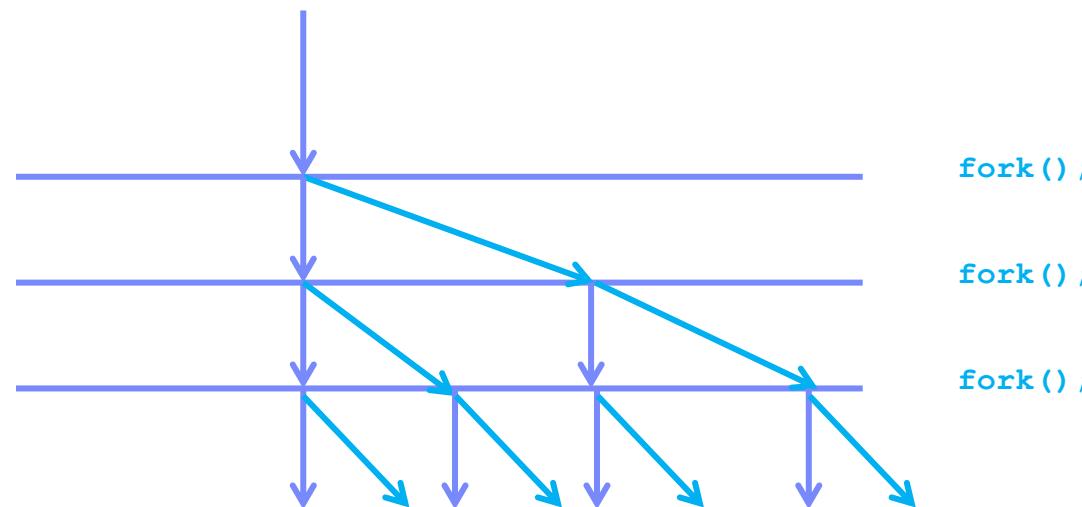
```
/* forkdemo2.c - shows how child processes pick up at the return
 *                  from fork() and can execute any code they like,
 *                  even fork().  Predict number of lines of output.
 */
#include      <stdio.h>
#include      <unistd.h>

main()
{
    printf("my pid is %d\n", getpid() );
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid() );
}
```

Ex4 - Children creating process example : forkdemo2.c

```
my pid is 24740  
my pid is 24740  
my pid is 24741  
my pid is 24742  
my pid is 24744  
my pid is 24745  
my pid is 24743  
my pid is 24746  
my pid is 24747
```

```
seokin@compasslab1:~/system_programming/labs/lab7$
```



fork() system call

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

Ex5 - Distinguishing parent from child Example : forkdemo3.c

```
/*  forkdemo3.c - shows how the return value from fork()
 *                  allows a process to determine whether
 *                  it is a child or process
 */
#include      <stdio.h>
#include      <unistd.h>
main()
{
    int      fork_rv;
    printf("Before: my pid is %d\n", getpid());
    fork_rv = fork();           /* create new process */
    if ( fork_rv == -1 )        /* check for error */
        perror("fork");
    else if ( fork_rv == 0 )
        printf("I am the child.  my pid=%d\n", getpid());
    else
        printf("I am the parent. my child is %d\n", fork_rv);
}
```

-
- Three skills needed to build a shell:
 - 1. Create a new process
 - 2. Run a program (execvp)
 - **3. Tell the parent wait until the child process finishes executing the command**

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

- Ans: A process calls wait() system call to wait for a child to finish.

```
pid = wait( &status );
```

- The wait system call does two things:
 - It pauses the calling process until a child process finishes running.
 - It retrieves the value the child process had passed to exit.

wait() system call

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)

wait() system call

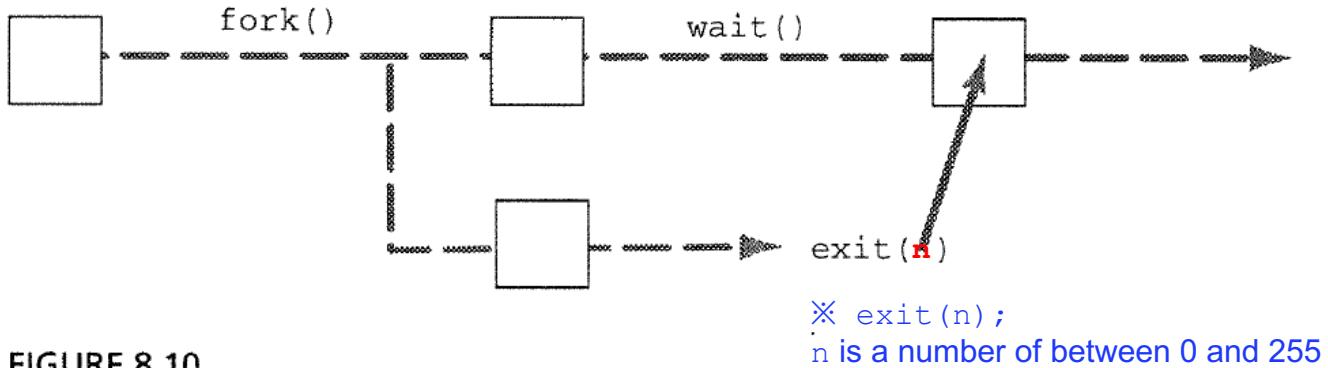


FIGURE 8.10

wait pauses the parent until the child finishes.

- When the child calls `exit()`,
 - the kernel wakes up the parent and
 - delivers the value the child passed to `exit()`.

- Thus, `wait()` performs two operations
 - notification and communication

Ex6 - Notification Example : waitdemo1.c

```
/* waitdemo1.c - shows how parent pauses until child finishes
 */
#include      <stdio.h>
#include      <stdlib.h>
#define DELAY    2

main()
{
    int newpid;
    void child_code(int), parent_code(int);
    printf("before: mypid is %d\n", getpid());

    if ( (newpid = fork()) == -1 )
        perror("fork");
    else if ( newpid == 0 )
        child_code(DELAY);
    else
        parent_code(newpid);
}
```

Ex6 - Notification Example : waitdemo1.c

```
/*
 * new process takes a nap and then exits
 */
void child_code(int delay)
{
    printf("child %d here. will sleep for %d seconds\n", getpid(),
           delay);
    sleep(delay);
    printf("child done. about to exit\n");
    exit(17);
}
```

Ex6 - Notification Example : waitdemo1.c

```
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
    int wait_rv;                  /* return value from wait() */
    wait_rv = wait(NULL);
    printf("done waiting for %d. Wait returned: %d\n", childpid,
           wait_rv);
}
```

```
$ ./waitdemo1
before: mypid is 10328
child 10329 here. will sleep for 2 seconds
child done. about to exit
done waiting for 10329. Wait returned: 10329
```

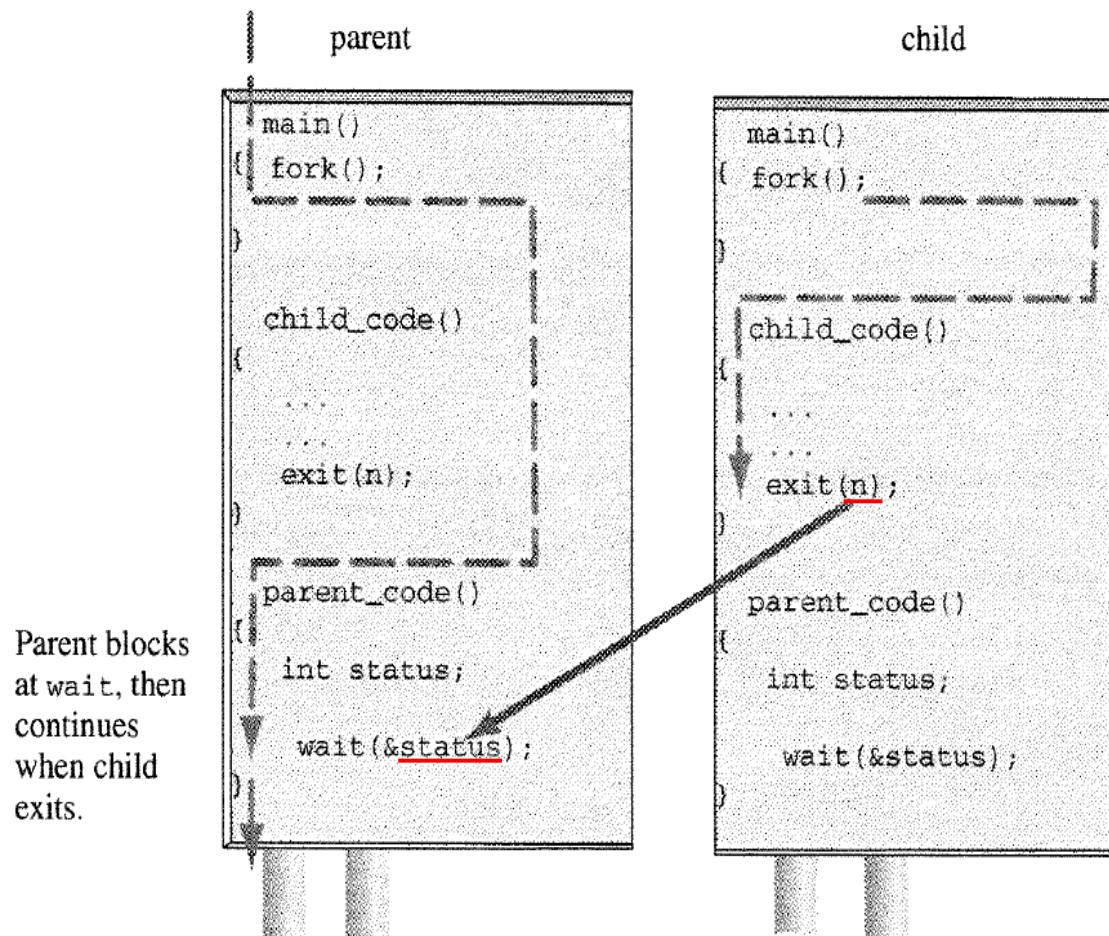


FIGURE 8.11

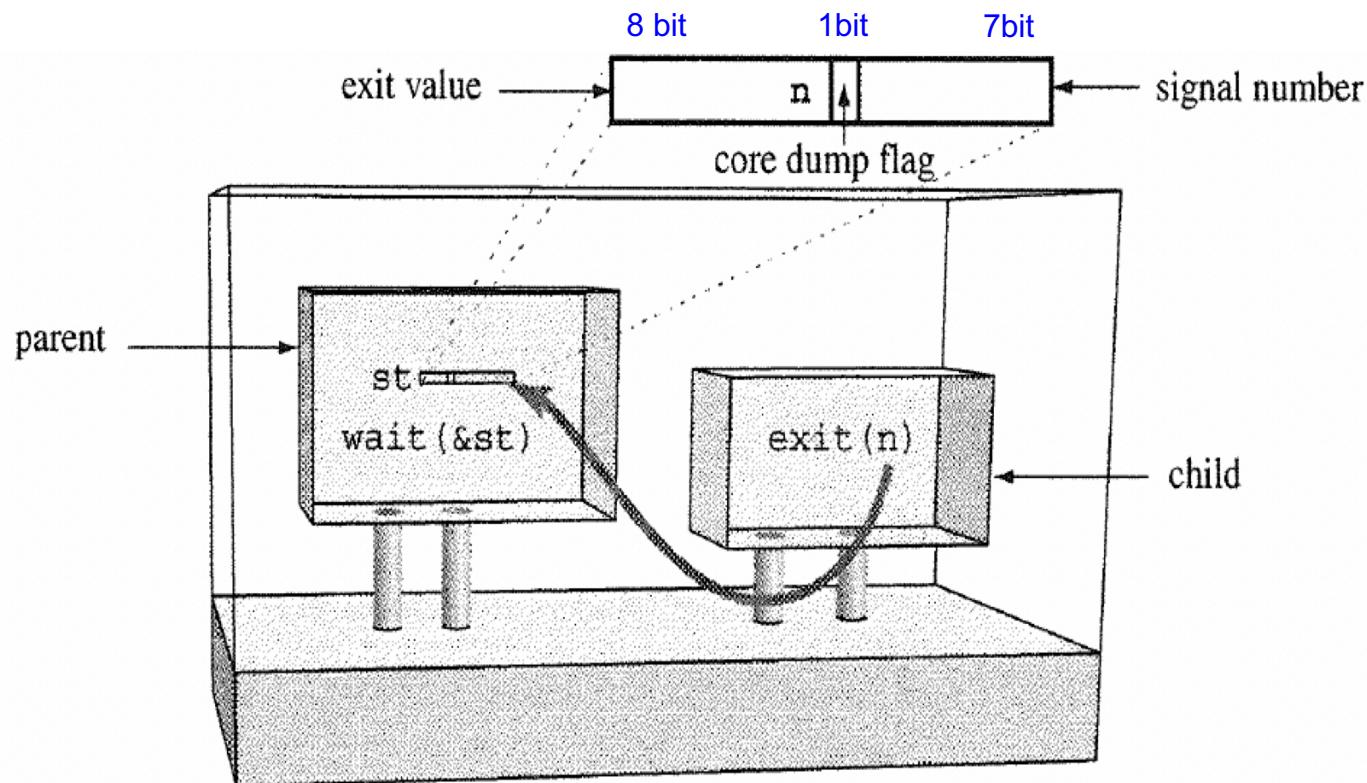
Control flow and communication with `wait()`.

Purpose of wait() system call

- To notify the parent that a child process finished running
- To tell the parent how a child process finished
 - A process ends in one of three ways
 - Success
 - Failure
 - Death
 - A process can succeed at its task:
exit(0) or return 0 from main
 - A process can fail at its task:
exit(nonzero value)
 - A process might be killed by a signal

How does the parent know if the child's exit status

- How does the parent know if the child succeeded, failed, or died?
 - Answer is in the argument (integer pointer) to wait() system call
 - Eight bit for exit value, seven bits for signal number, one bit to indicate a core dump



Ex7 - Communication Example: waitdemo2.c

```
/* waitdemo2.c - shows how parent gets child status
 */
#include <stdio.h>
#include <stdlib.h>
#define DELAY 5

main()
{
    int newpid;
    void child_code(), parent_code();

    printf("before: mypid is %d\n", getpid());
    if ( (newpid = fork()) == -1 )
        perror("fork");
    else if ( newpid == 0 )
        child_code(DELAY);
    else
        parent_code(newpid);
}
```

Ex7 - Communication Example: waitdemo2.c

```
/*
 * new process takes a nap and then exits
 */
void child_code(int delay)
{
    printf("child %d here. will sleep for %d seconds\n", getpid(),
           delay);
    sleep(delay);
    printf("child done. about to exit\n");
    exit(17);
}
```

Ex7 - Communication Example: waitdemo2.c

```
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
    int wait_rv;                  /* return value from wait() */
    int child_status;
    int high_8, low_7, bit_7;

    wait_rv = wait(&child_status);
    printf("done waiting for %d. Wait returned: %d\n", childpid,
           wait_rv);

    high_8 = child_status >> 8;      /* 1111 1111 0000 0000 */
    low_7  = child_status & 0x7F;    /* 0000 0000 0111 1111 */
    bit_7  = child_status & 0x80;    /* 0000 0000 1000 0000 */
    printf("status: exit=%d, sig=%d, core=%d\n", high_8, low_7,
           bit_7);
}
```

Ex7 - Communication Example: waitdemo2.c

- Run in the background and use kill to send SIGTERM to the child:

```
$ ./waitdemo2 &
$ before: mypid is 10857
child 10858 here. will sleep for 5 seconds
kill 10858
$ done waiting for 10858. Wait returned: 10858
status: exit=0, sig=15, core=0
          SIGTERM
```

Summary : How the Shell Runs Programs

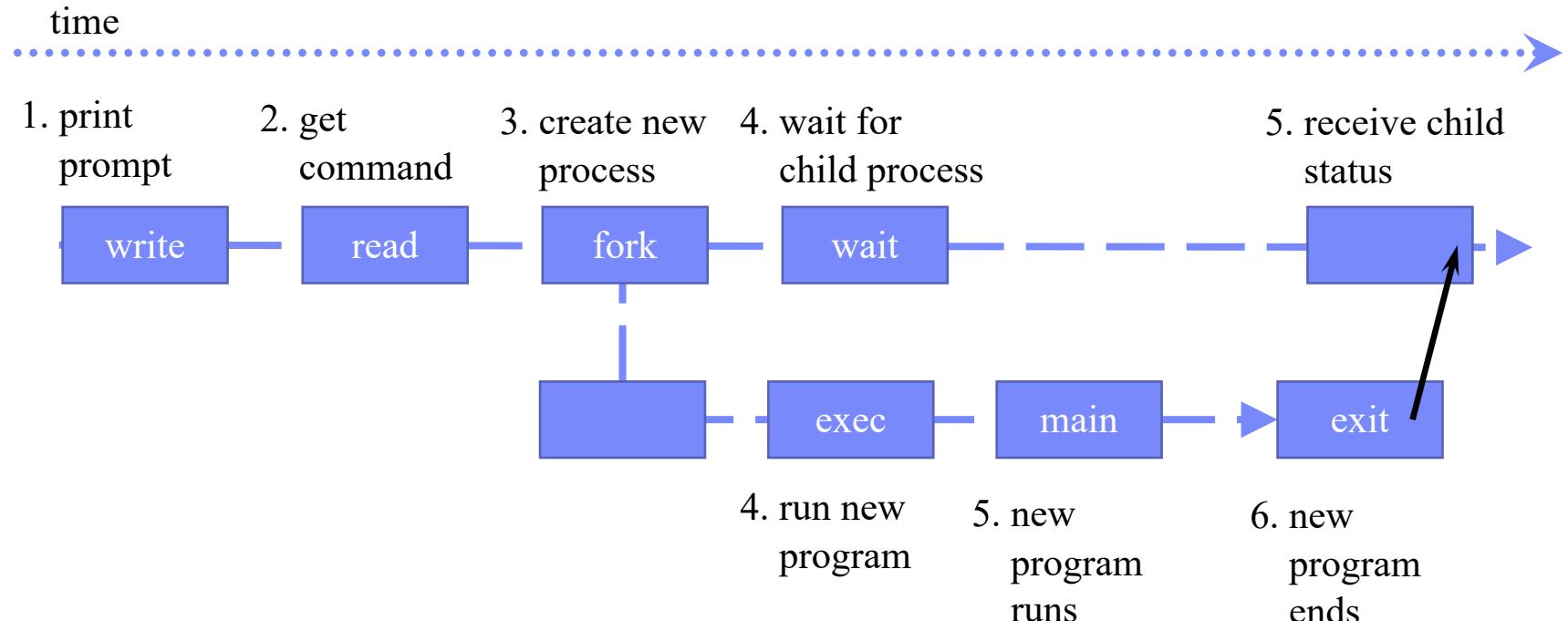


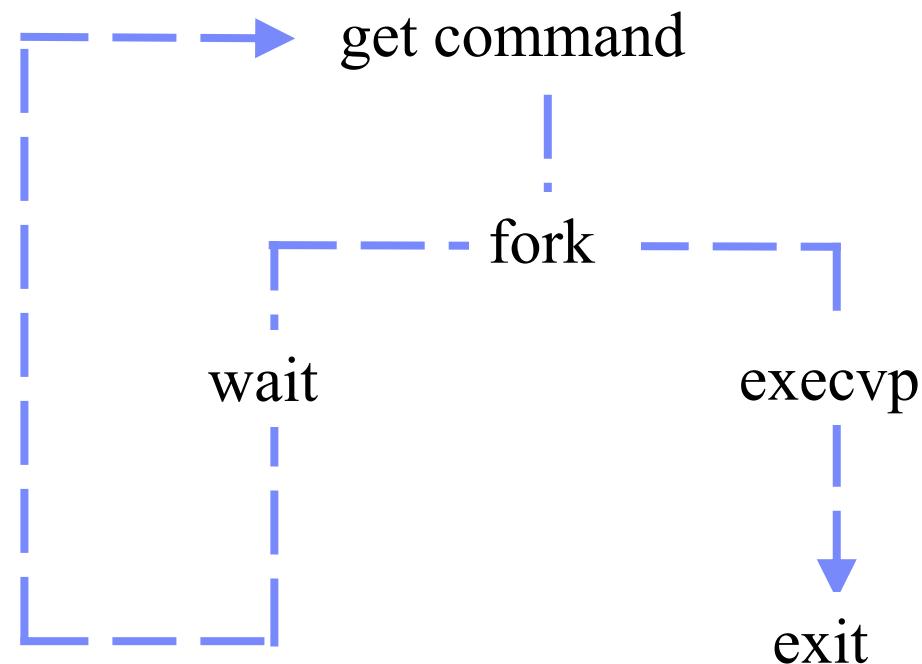
FIGURE 8.13 Shell loop with `fork()`, `exec()`, and `wait()`

Contents

- 8.1 Process
- 8.2 Learning about Processes with ps
- 8.3 The Shell: A Tool for Process and Program Control
- 8.4 How the Shell Runs Programs
- 8.5 Writing a Shell: psh2.c

Ex8 - Writing a shell : psh2.c

- Logic diagram



Ex8 - Writing a shell : psh2.c

```
/** prompting shell version 2 (psh2.c)
**
**           Solves the 'one-shot' problem of version 1
**           Uses execvp(), but fork()'s first so that the
**           shell waits around to perform another command
**           New problem: shell catches signals. Run vi, press ^c.
**/


#include      <stdio.h>
#include      <signal.h>
#include      <string.h>
#include      <stdlib.h>

#define MAXARGS      20          /* cmdline args */
#define ARGLEN       100         /* token length */

char* makestring(char*) ;
void execute(char* []);
```

Ex8 - Writing a shell : psh2.c

```
int main()
{
    char    *arglist[MAXARGS+1];          /* an array of ptrs      */
    int     numargs = 0;                  /* index into array      */
    char    argbuf[ARGLEN];              /* read stuff here      */

    while ( numargs < MAXARGS )
    {
        printf("Arg[%d] ? ", numargs);
        if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
            arglist[numargs++] = makestring(argbuf);
        else
        {
            if ( numargs > 0 ){           /* any args?      */
                arglist[numargs]=NULL;   /* close list      */
                execute( arglist );     /* do it          */
                numargs = 0;             /* and reset      */
            }
        }
    }
    return 0;
}
```

Ex8 - Writing a shell : psh2.c

```
- execute( char *arglist[] )
/*
 *      use fork and execvp and wait to do it
 */
{
    int      pid,exitstatus;                      /* of child      */
    pid = fork();                                /* make new process */
    switch( pid ){
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            execvp(arglist[0], arglist);           /* do it */
            perror("execvp failed");
            exit(1);
        default:
            while( wait(&exitstatus) != pid )
                ;
            printf("child exited with status %d,%d\n",
                   exitstatus>>8, exitstatus&0377);
    }
}
```

Ex8 - Writing a shell : psh2.c

```
char *makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
    char      *cp, *malloc();

    buf[strlen(buf)-1] = '\0';           /* trim newline */
    cp = malloc( strlen(buf)+1 );        /* get memory */
    if ( cp == NULL ){                  /* or die */
        fprintf(stderr, "no memory\n");
        exit(1);
    }
    strcpy(cp, buf);                   /* copy chars */
    return cp;                         /* return ptr */
}
```

```
$ ./psh2
Arg[0]? tr
Arg[1]? [a-z]
Arg[2]? [A-Z]
Arg[3]?
hello
HELLO
now to press
NOW TO PRESS
Ctrl-Cpress ^C here
$
```