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**
  - `fork, exec, wait, exit`
- **Commands**
  - `sh, ps`

PROCESSES = PROGRAMS IN ACTION

- **Program : …**
- **Running a program : …**
- **Process**
  - The memory space and settings with which the program runs

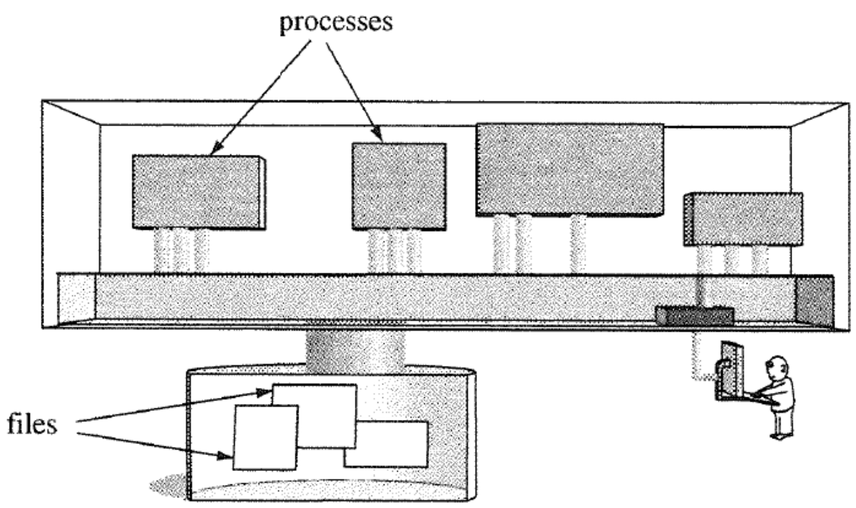


FIGURE 8.1

Processes are programs in action.

Learning about Processes with ps

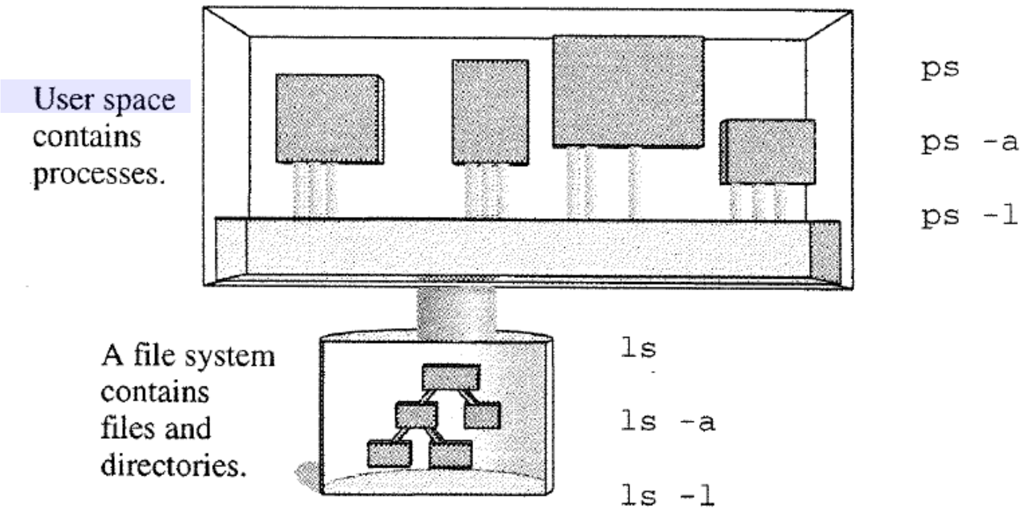


FIGURE 8.2

The ps command lists current processes.

-a: 터미널에 연결된 모든 사용자의 프로세스 출력
-l: long format

```
$ ps
  PID TTY          TIME CMD
 1755 pts/1    00:00:17 bash
 1981 pts/1    00:00:00 ps
```

```
$ 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
```

```
$ 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**: Process state code (e.g., R for running, S for sleeping).
- **UID**: User ID of the process owner.
- **PID**: Process ID.
- **PPID**: Parent process ID.
- **C**: Processor utilization in terms of scheduling priority.
- **PRI**: Process priority.
- **NI**: Nice value, affecting priority.
- **SZ**: Size of the process in memory.
- **WCHAN**: Waiting channel, indicating what the process is waiting on.
- TTY: Terminal type.
- **TIME**: CPU time used by the process.
- **CMD**: Command name.

```
$ ps -fa
UID        PID   PPID  C STIME TTY          TIME CMD
betsy      1779  1731  0 19:53 pts/0    00:00:01 gv dinner.ps
betsy      1780  1779  0 19:53 pts/0    00:00:07 gs -dNOPLATFONTS
betsy      1781  1731  0 19:54 pts/0    00:00:02 vi dinner
yuriko     2013  1993  0 20:15 pts/2    00:00:00 xpaint
yuriko     2017  1993  0 20:16 pts/2    00:00:00 mail bruce
bruce      2401  1755  0 20:36 pts/1    00:00:00 ps -af
```

Username

The complete command line

System Processes (1/2)

- **Processes run by users and Unix system**

```
$ ps -ax|head -25
PID TTY        STAT    TIME COMMAND
  1 ?          S       0:05 init
  2 ?          SW      3:54 [kflushd]
  3 ?          SW      0:38 [kupdate]
  4 ?          SW      0:00 [kpiod]
  5 ?          SW      2:13 [kswapd]
 35 ?          SW      0:00 [uhci-control]
 36 ?          SW      0:00 [khubd]
420 ?          S       0:25 syslogd
423 ?          S       0:36 klogd -k /boot/System.map-2.2.14
  …

563 tty3       SW      0:00 [getty]
$ ps -ax| wc -l
     82
```

※ print the newline count

D Uninterruptible sleep (usually IO)
R Running or runnable (on run queue)
S Interruptible sleep (waiting for an event to complete)
T Stopped, either by a job control signal or because it is being traced.
W paging (not valid since the 2.6.xx kernel)
X dead (should never be seen)
Z Defunct ("zombie") process, terminated but not reaped by its parent

System Processes (2/2)

- What do all these **system processes do:**
  - Manage different parts of **memory** : kernel buffers, virtual memory pages
  - Manage system **logfiles** (`klogd, syslogd`)
  - **Schedule** batch jobs (`cron, atd`)
  - **Watch** for potential intruders (`portsentry`)
  - Allow regular users to log in (`sshd, getty`)

Process Management and File Management

- **The kernel manages processes** in memory and **files** on the disk.

- **How similar** is memory management to disk management:

  - 파일은 데이터를, 프로세스는 실행 코드를 포함
  - 파일과 프로세스 모두 속성을 가짐
  - 커널이 파일을 생성/삭제하는 것처럼 프로세스도 생성/삭제
  - 커널은 메모리에 여러 프로세스를, 디스크에 여러 파일을 저장
  - 커널은 메모리와 디스크 블록 할당 및 추적 필요

Computer Memory and Computer Programs (1/2)

Memory can be viewed as an expanse of space containing the kernel and processes.

Many systems view memory as an array of “pages” and split processes into several pages.

The array of pages may be stored physically in solid state chips.

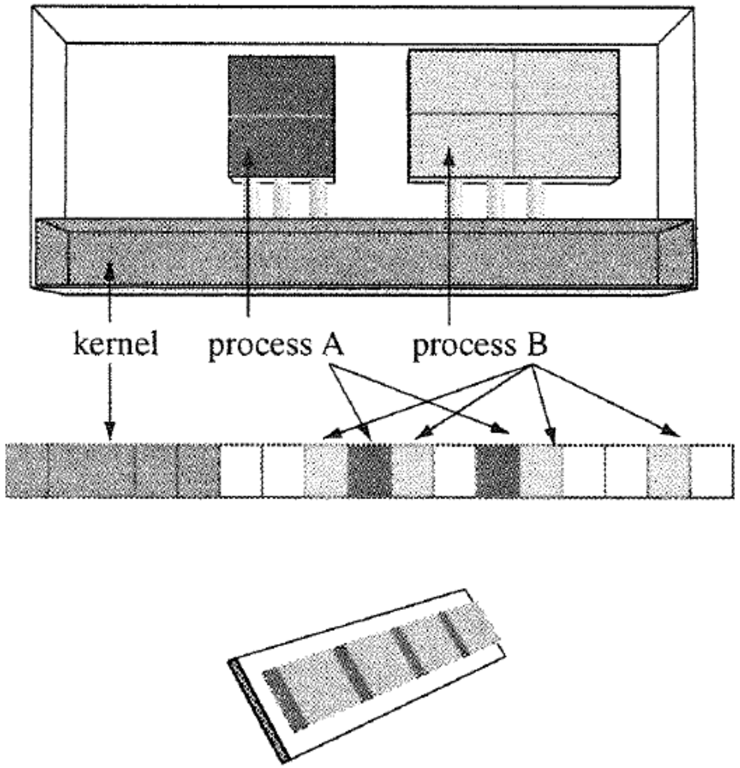


FIGURE 8.3

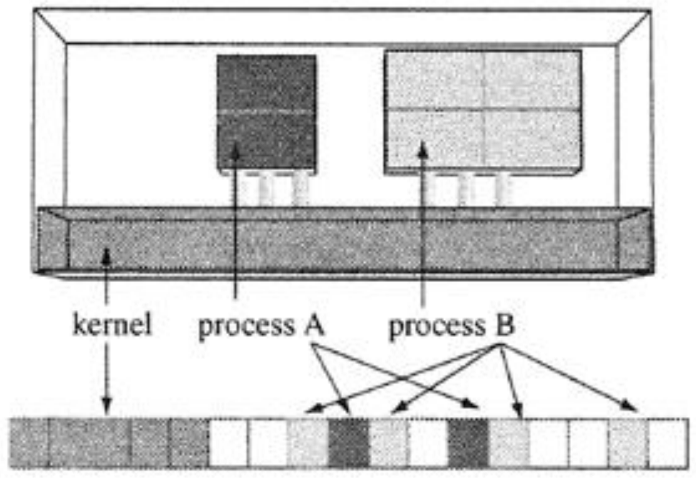
Three models of computer memory.

To display size of a page in bytes:

$ getconf PAGESIZE

Computer Memory and Computer Programs (2/2)

- **Creating a process** is similar to creating a disk file
  - Kernel has to find some free pages of memory
    - to hold the machine-language codes and data bytes for the program
  - Kernel sets up some data structures
    - to store memory allocation information and the attributes of the process



Shell

- **A program that manages processes and runs programs**
  - There are many shells, all with different styles and strengths

- **Three main functions:**
  - (a) Shells run programs : …
  - (b) Shells manage input and output : …
  - (c) Shells can be programmed

How the Shell Runs Programs

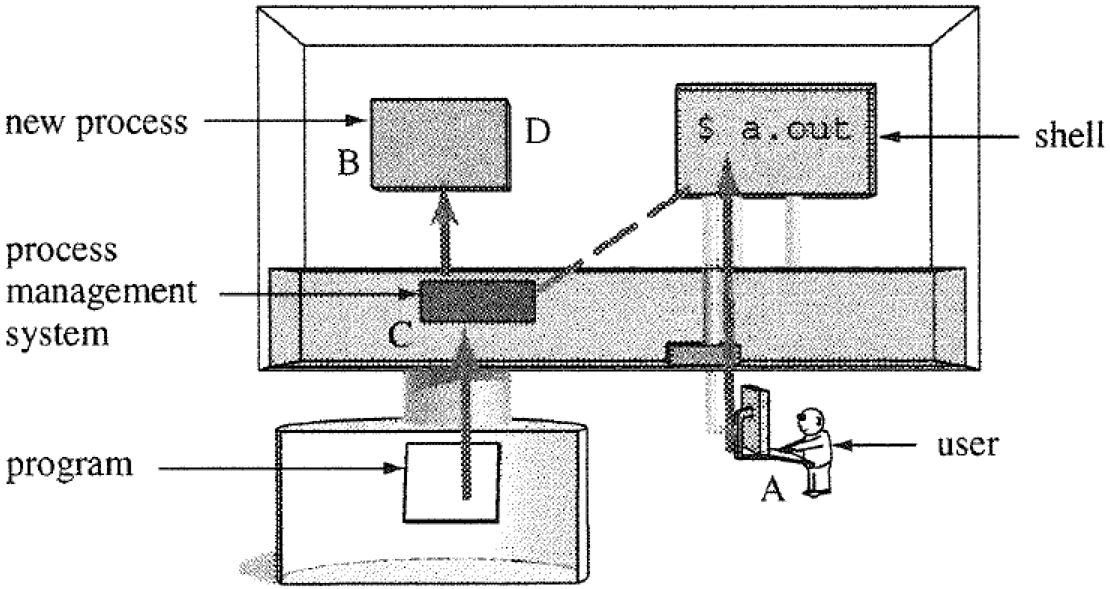


FIGURE 8.4

A user asks a shell to run a program.

A. The user types `a.out`.
B. The shell creates a new process to run the program.
C. The shell loads the program from the disk into the process.
D. 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
```

- **Consider this typical interaction with the shell:**

```
$ ls
Chap.bak   Story08.tr   chap08.ps      chap08.tr   outline.08
Makefile   chap08       chap08.short   code        pix
$ ps
  PID TTY          TIME CMD
29182 pts/5    00:00:00 bash
29183 pts/5    00:00:00 ps
$
```

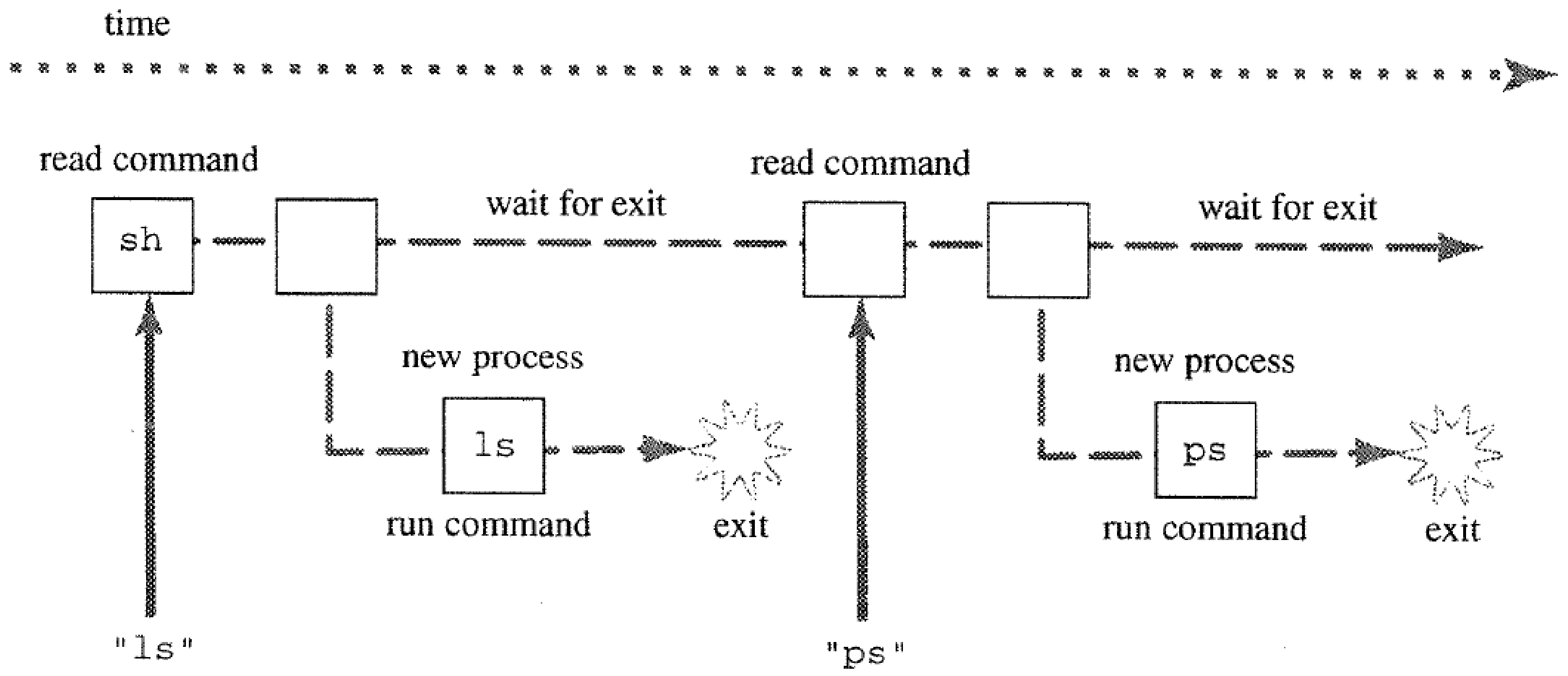


FIGURE 8.5

A time line of the main loop of the shell.

- **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?

- **Ans: The program calls `execvp`.**

	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	

```
/* 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";
        arglist[1] = "-l";
        arglist[2] = 0 ;
        printf("* * * About to exec ls -l\n");
        execvp( "ls" , arglist );
        printf("* * * ls is done. bye\n");
}
```

```
$ cc exec1.c -o exec1
$ ./exec1
* * * About to exec ls -l
total 28
drwxr-x---   2 bruce   users       1024 Jul 14 21:02 a
drwxr-x---   3 bruce   users       1024 Jul 16 03:16 c
-rw-r--r--   1 bruce   users          0 Jul 14 21:03 y
$
```

How Unix runs programs:

```
$ ls -l
```

```
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.

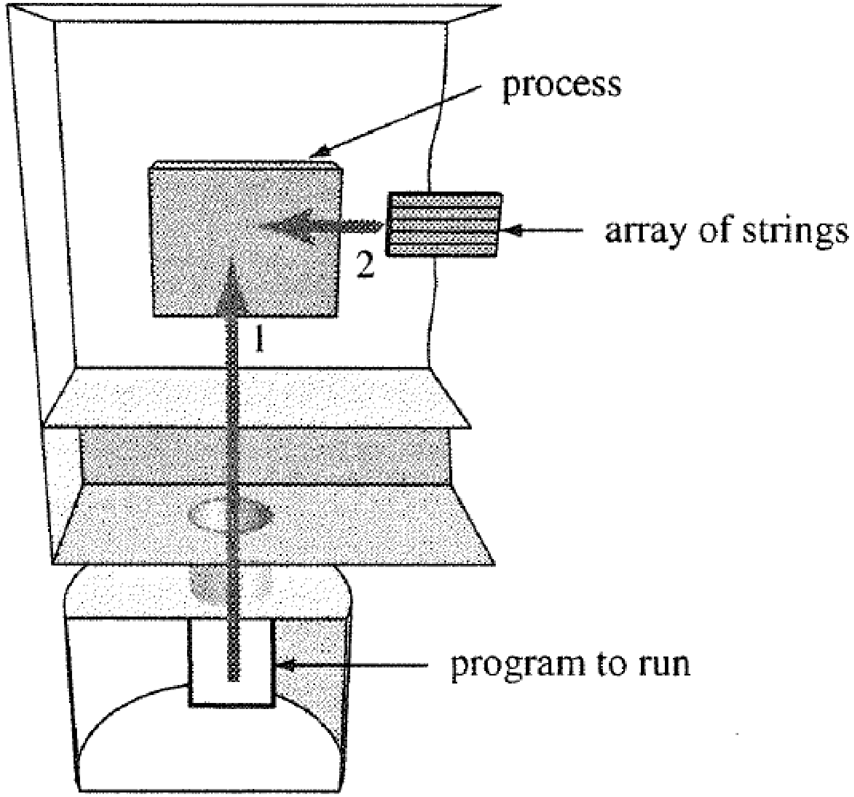


FIGURE 8.6

execvp copies into memory and runs a program.

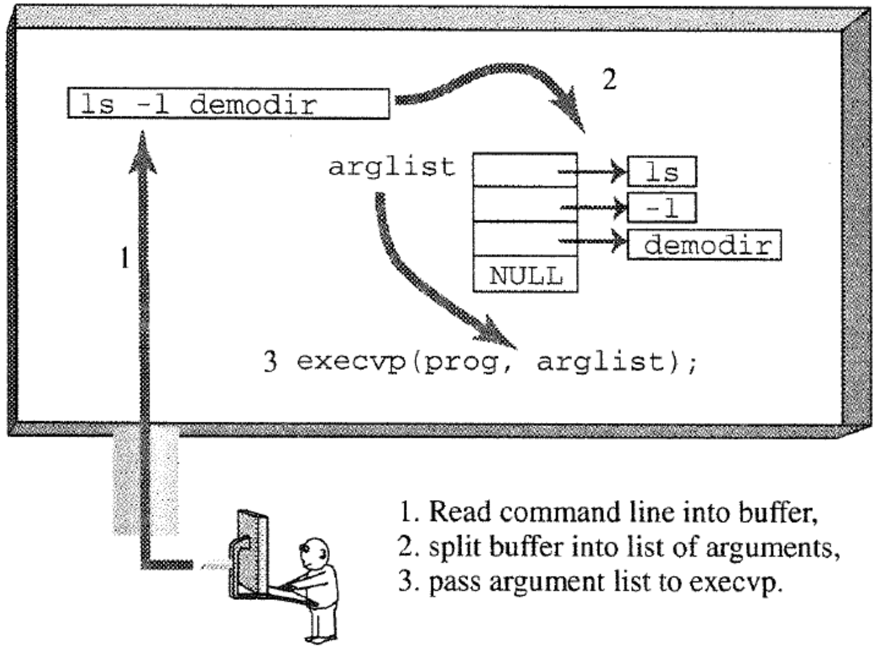
```
main()
{
        char    *arglist[3];
        arglist[0] = "ls";
        arglist[1] = "-l";
        arglist[2] = 0 ;
        printf("* * * About to exec ls -l\n");
        execvp( "ls" , arglist );
        printf("* * * ls is done. bye\n");
}
```

```
$ cc exec1.c -o exec1
$ ./exec1
* * * About to exec ls -l
total 28
drwxr-x---    2 bruce    users        1024 Jul 14 21:02 a
drwxr-x---    3 bruce    users        1024 Jul 16 03:16 c
-rw-r--r--    1 bruce    users           0 Jul 14 21:03 y
$
```

- **The kernel loads the new program** into the current process:
  - **replacing** the code and data of that process

	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

Ex2: A Prompting Shell




```
$ cc psh1.c -o psh1
$ ./psh1
Arg[0]? ls
Arg[1]? -l
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---   2 bruce    users        1024 Jul 14 21:02 a
drwxr-x---   3 bruce    users        1024 Jul 16 03:16 c
-rw-r--r--   1 bruce    users           0 Jul 14 21:03 y
$
```

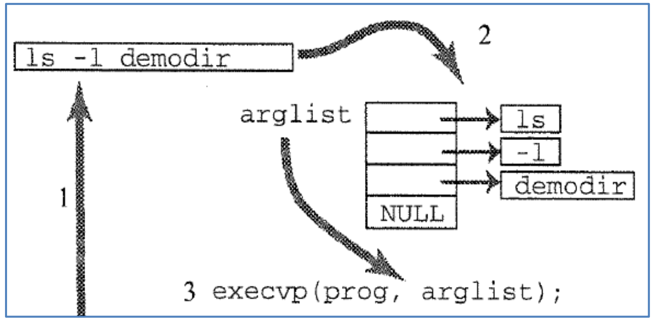
FIGURE 8.7

Building an `arglist` from one string.

```c
/*      prompting shell version 1   (psh1.c)
 *              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*[]);
```

```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 )
        {                                                  ※입력 첫 문자가
                                                           '\n'이 아닐 때 *
                printf("Arg[%d]? ", numargs);
                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;
}
```

```
$ ./psh1
Arg[0]? ls
Arg[1]? -l
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---    2 bruce
drwxr-x---    3 bruce
```



```c
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 ){                      /* or die       */
                fprintf(stderr,"no memory\n");
                exit(1);
        }
        strcpy(cp, buf);                /* copy chars   */
        return cp;                      /* return ptr   */
}

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

- **The program works OK, but..**
  - `execvp` replaces the code of the shell with the code of the command, then exits.

```
$ cc psh1.c -o psh1
$ ./psh1
Arg[0]? ls
Arg[1]? -l
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---   2 bruce    users        1024 Jul 14 21:02 a
drwxr-x---   3 bruce    users        1024 Jul 16 03:16 c
-rw-r--r--   1 bruce    users           0 Jul 14 21:03 y
$
```

- 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: Process calls `fork` to replicate itself.**

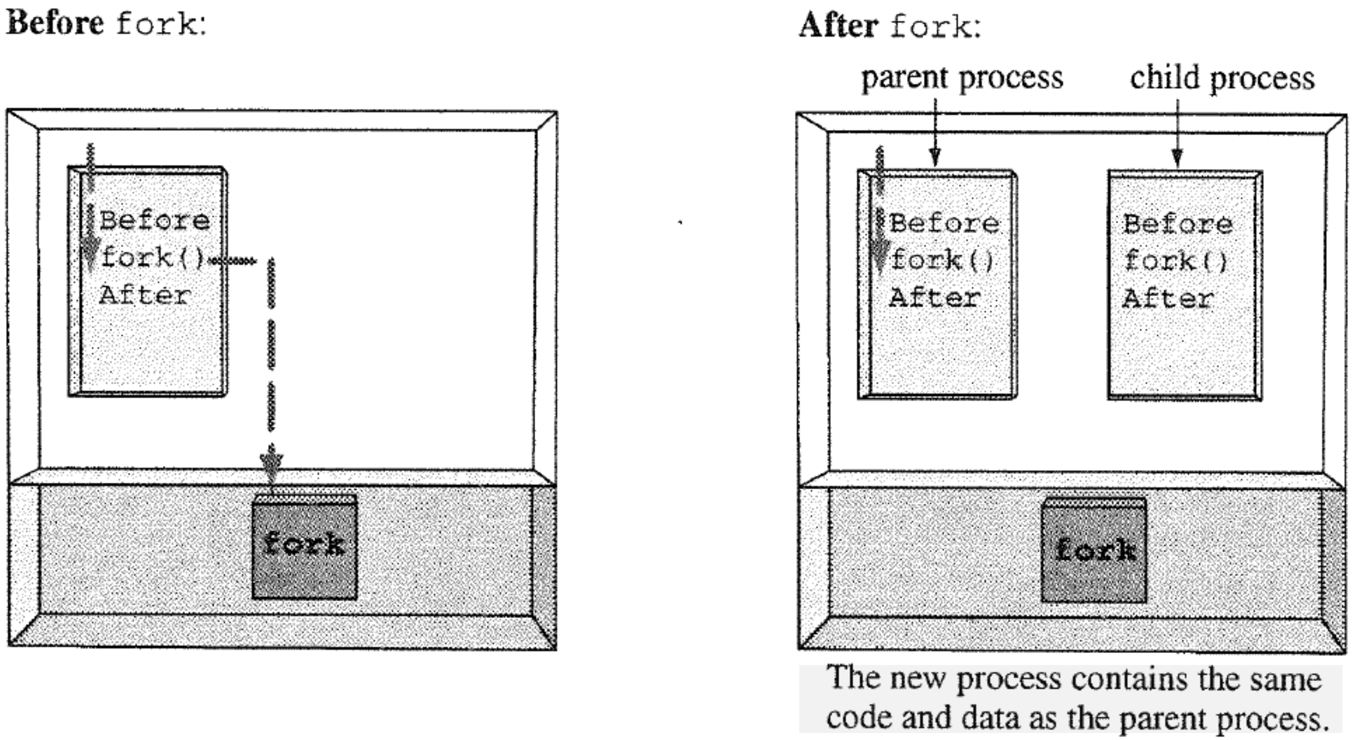


FIGURE 8.8

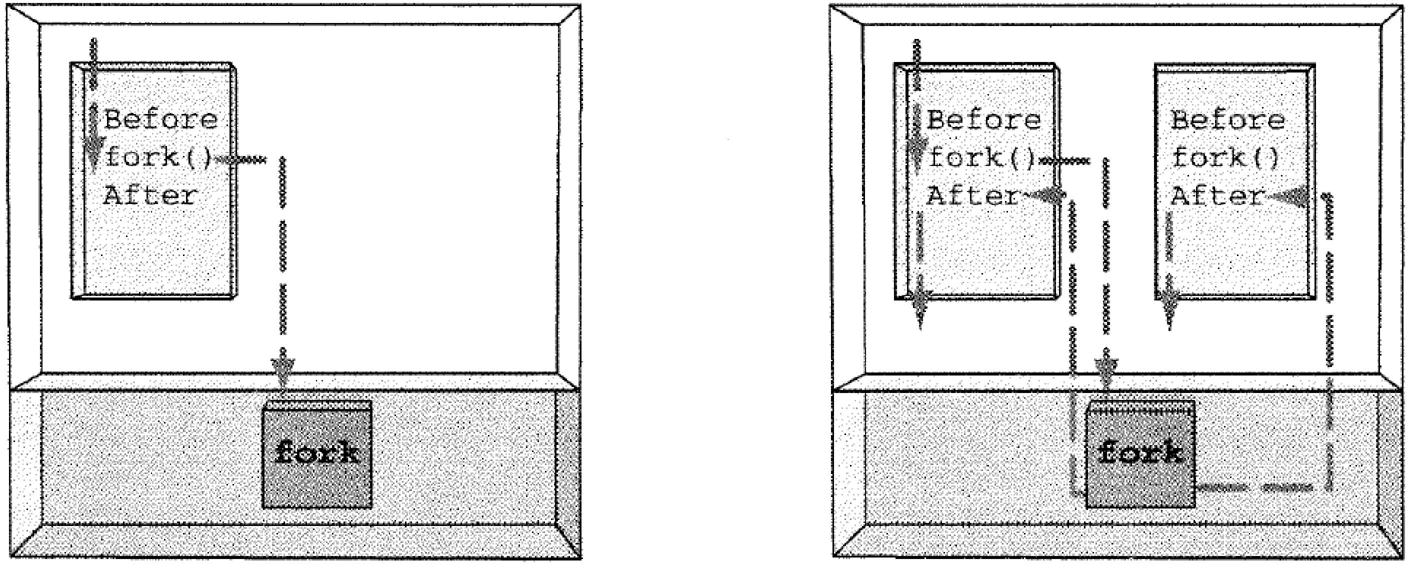
`fork()` makes a copy of a process.

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

```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
```

Before `fork`: **After** `fork`:



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()`.

Process contains
program + current location
in the program

- **What the kernel `by fork()` does:**
  - (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

```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() );
}
```

```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);
}
```

```
$ ./forkdemo3
Before: my pid is 5931
I am the parent. my child is 5932
I am the child.  my pid=5932
$
```

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

- **Ans: Process calls** `wait` to wait for a child to finish

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

	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)

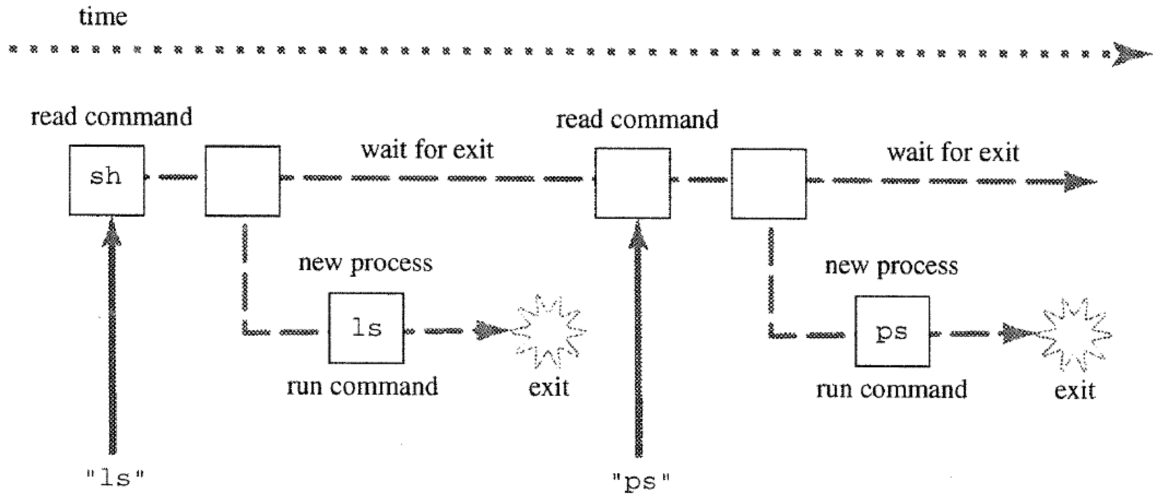


FIGURE 8.5
A time line of the main loop of the shell.

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

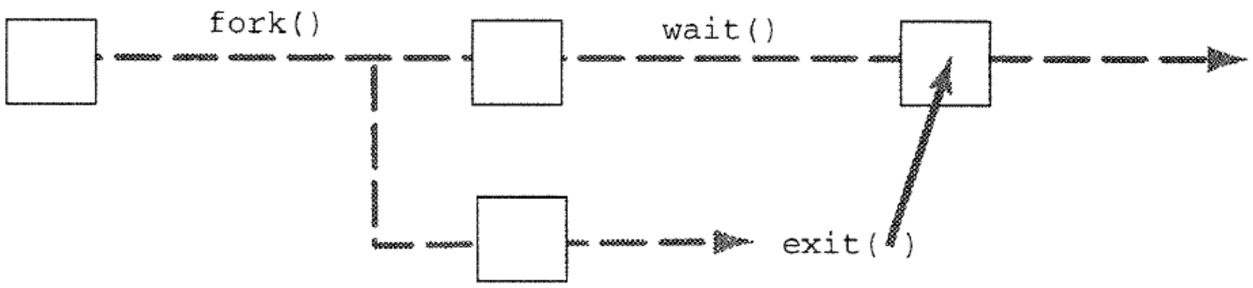
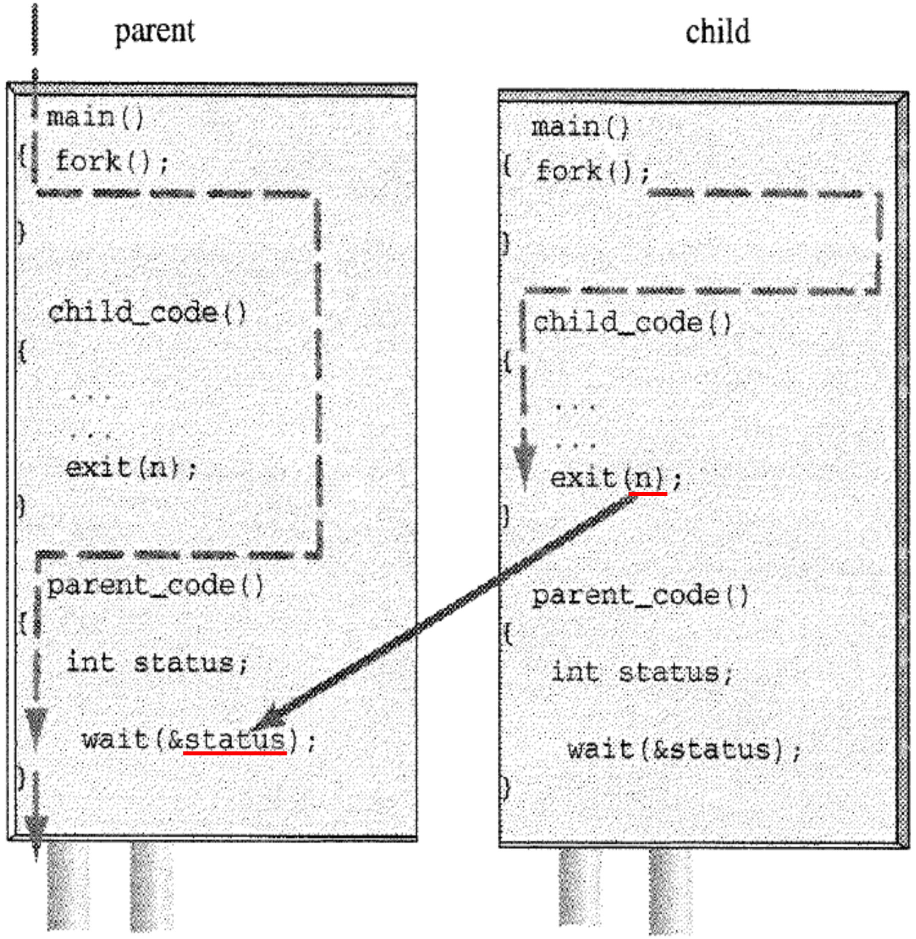


FIGURE 8.10

wait pauses the parent until the child finishes.



Parent blocks at `wait`, then continues when child exits.

FIGURE 8.11

Control flow and communication with `wait()`.

```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);
}
```

```
$ ./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
```

```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);
}

/*
 * 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);
}
```

It blocks the calling program until a child finishes
It returns the PID of the finishing process

- **Purpose of `wait`**
  - To **notifiy** 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, and Death**
  1. A process can succeed at its task:
     `exit(0)` or return 0 from `main`
  2. A process can fail at its task:
     `exit(nonzerovalue)`
  3. A process might be killed by a ***signal***

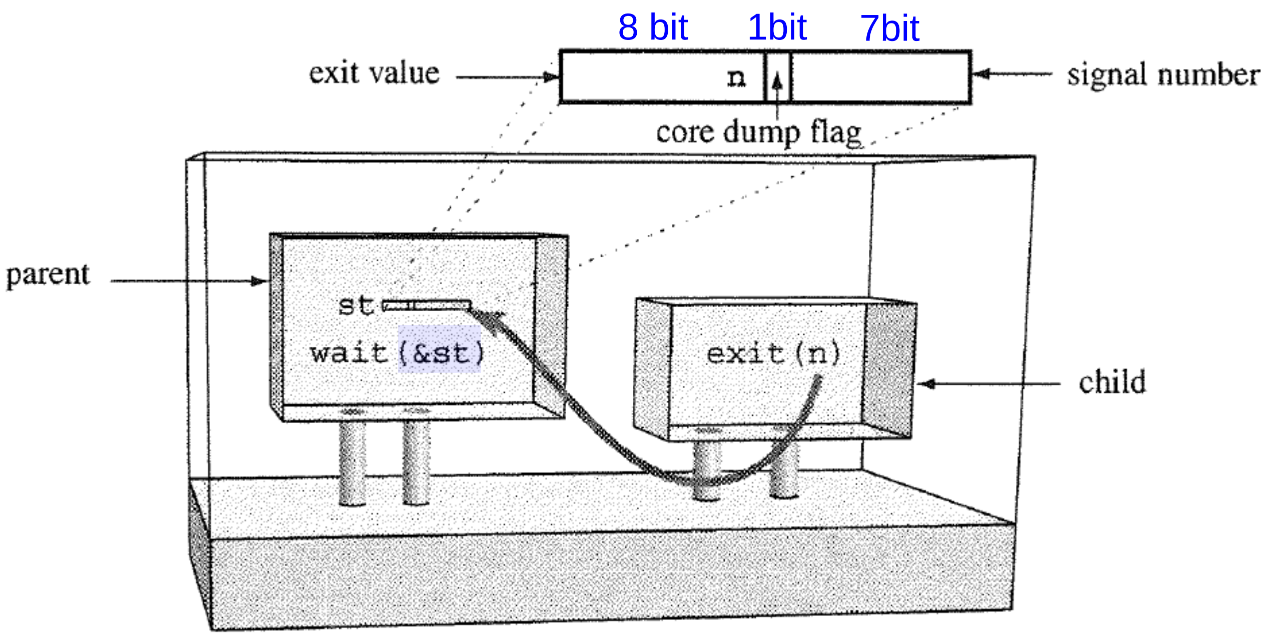


FIGURE 8.12

The child status value has three parts.

```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);
}
/*
 * 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);
}
```

```
$ ./waitdemo2
before: mypid is 10855
child 10856 here. will sleep for 5 seconds
child done. about to exit
done waiting for 10856. Wait returned: 10856
status: exit=17, sig=0, core=0
$
```

```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);
}
```

- **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 ※ Input rapidly!
$ done waiting for 10858. Wait returned: 10858
status: exit=0, sig=15, core=0
                  SIGTERM
```

Summary

- **How the Shell Runs Programs:**

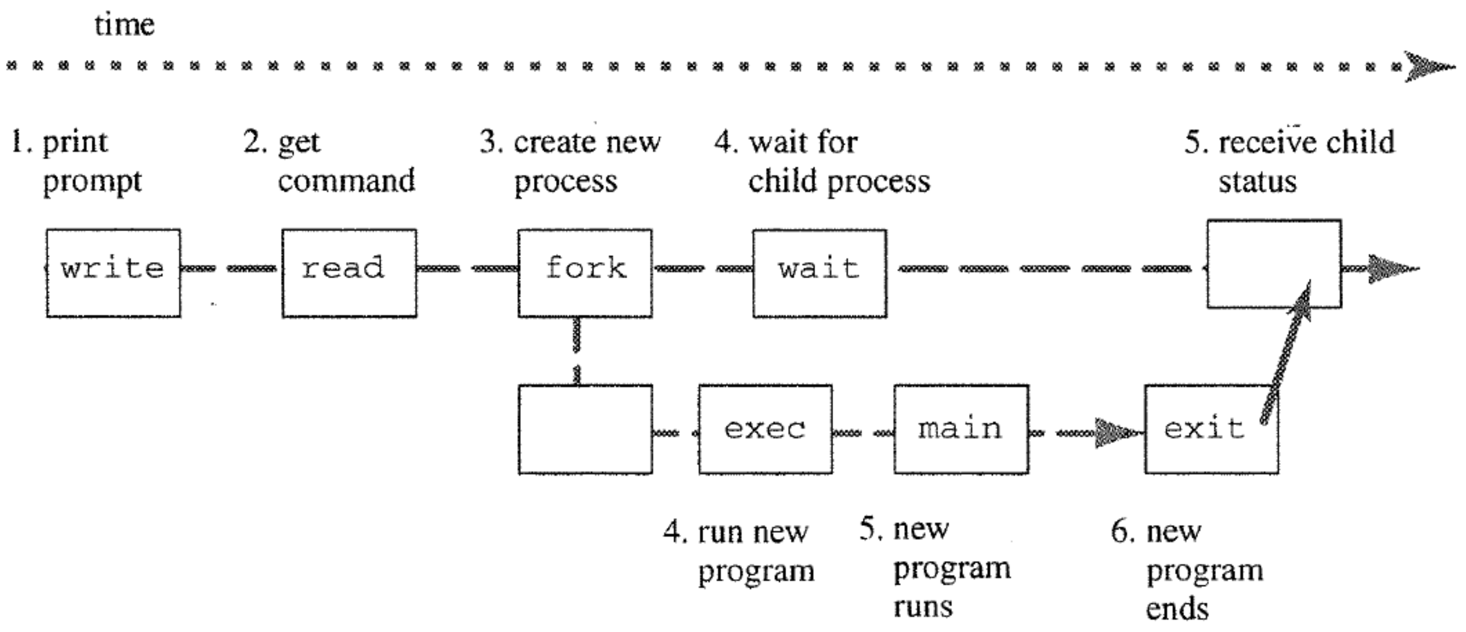


FIGURE 8.13
Shell loop with fork(), exec(), and wait().

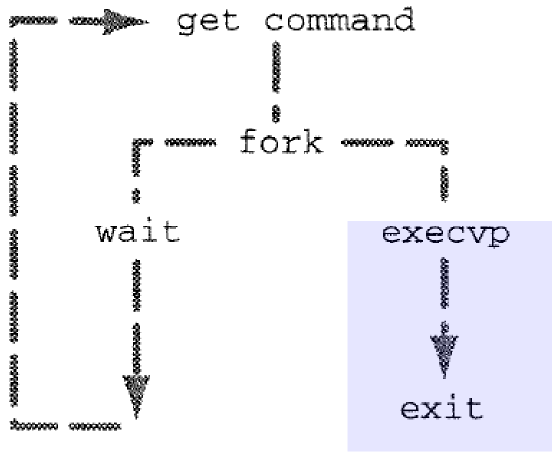


FIGURE 8.14

Logic of a basic Unix shell.

```
$ ./psh2
Arg[0]? ls
Arg[1]? -l
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---   2 bruce    users        1024 Jul 14 21:02 a
drwxr-x---   3 bruce    users        1024 Jul 16 03:16 c
-rw-r--r--   1 bruce    users           0 Jul 14 21:03 y
child exited with status 0,0
Arg[0]? ps
Arg[1]?
  PID TTY          TIME CMD
11616 pts/4    00:00:00 bash
11648 pts/4    00:00:00 psh2
11664 pts/4    00:00:00 ps
child exited with status 0,0
Arg[0]? ./psh1  Look! We can run psh1
Arg[1]?
Arg[0]? ps      This is the prompt from psh1!
Arg[1]?
  PID TTY          TIME CMD
11616 pts/4    00:00:00 bash
11648 pts/4    00:00:00 psh2
11683 pts/4    00:00:00 ps
child exited with status 0,0
Arg[0]? grep
Arg[1]? fred
Arg[2]? /etc/passwd
Arg[3]?
child exited with status 1,0
```

```
/**  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*[]);
```

```
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;
}
```

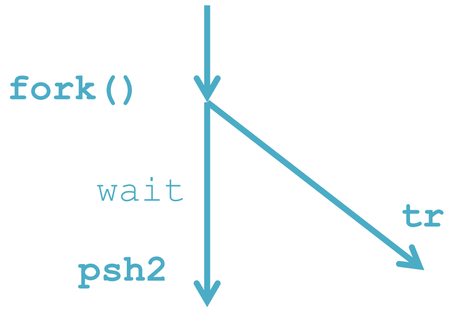
```
char *makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
        char    *cp, *malloc();

        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   */
}
```

```
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);
        }
}
```

Signals and `psh2.c`

- **What happen if** we press **Ctrl-C**
  - when `psh2` is waiting for the child process to finish…

```
$ ./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
$
```



- **Keyboard signals go to ALL** attached processes
  - How can we prevent this? …

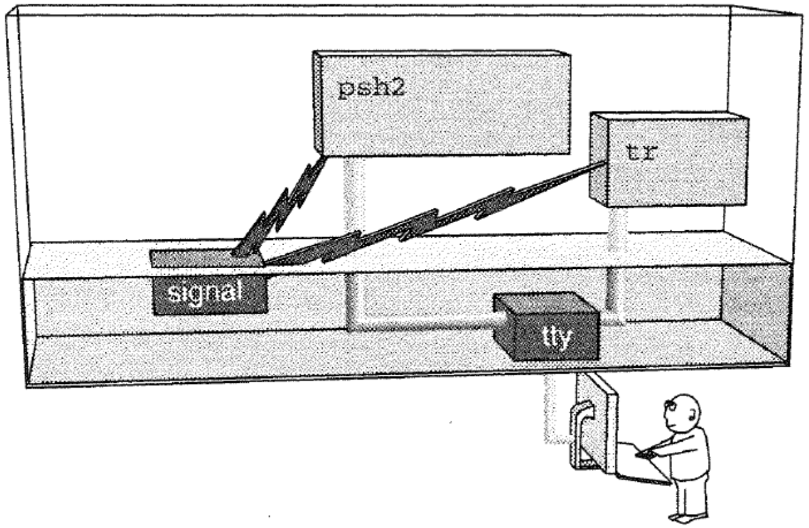


FIGURE 8.15

Keyboard signals go to all attached processes.

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`