Spring 2008 Handout 2

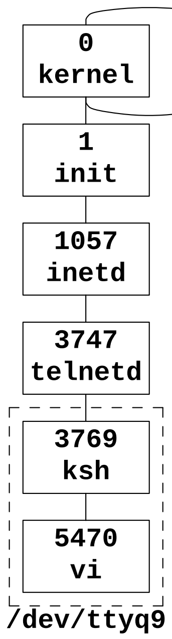
The difference between a program and a process: Curry pp. 283; Bach pp. 146−151; KP pp. 33−35

A *program* is what **`ls`** lists; a *process* is what **`ps`** lists. The sticky bit **`t`** is in Curry p. 125, Bach pp. 225−226.

```
1$ cd /usr/bin
2$ ls -l
-r-xr-xr-x   5 root     bin       240316 Jun  8  2006 vi
```

```
#!/bin/ksh
#Find all the files whose sticky bit is on.

find / -type f -perm -1000 -print 2> /dev/null
```

```
3$ ps -Af | more
USER       PID  PPID %CPU STARTED  TTY             TIME COMMAND
root         0     0  0.0   Dec 12 ??          09:32:11 [kernel idle]
root         1     0  0.0   Dec 12 ??           9:58.93 /sbin/init -a
root      1057     1  0.0   Dec 12 ??           6:16.06 /usr/sbin/inetd
root      3747  1057  0.0 21:29:16 ??           0:03.35 telnetd
mm64      3769  3747  0.0 21:29:17 ttyq9        0:02.46 -ksh (ksh)
mm64      5470  3769  0.0 22:03:44 ttyq9        0:00.19 vi process.ms
```



Give birth to a child: Curry pp. 292−295; KP pp. 184−185; K&R pp. 167, 253

The string that you give to **`system`** must use the Bourne shell syntax. The standard output of the **`cal`** in line 7 will become part of the standard output of the following C program:

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/system.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>  /* for system */
 3
 4 int main()
 5 {
 6     printf("The current month is\n");
 7     fflush(stdout);
 8
 9     system("cal");   /* Don’t need newline. */
10     system("cal 12 2000");
11     system("cal 12 2000 > $HOME/cal.out");    /* Bourne shell won’t take tilde. */
12
13     system("who | wc -l");
14     system("grep can\\’t /lyrics/stones/satisfaction");
15     system("grep \"can’t\" /lyrics/stones/satisfaction");
16
17     return EXIT_SUCCESS;
18 }
```

```
1$ grep can\’t /lyrics/stones/satisfaction
2$ grep "can’t" /lyrics/stones/satisfaction
```

KP p. 75

To get the exit status of the program run by **`system`**, store the return value in an **`int`** variable and examine it with the **`W`** macros in **`wait`**(2). Not every process returns an exit status: some are terminated or stopped by a signal first.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/systemexit.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
    /* blank and tab within the [] */
    int status = system("who | grep -q ’^abc1234[ \t]’");

    if (WIFEXITED(status)) {
        printf("My child’s exit status was %d.\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
        printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
    } else {
        fprintf(stderr, "%s: couldn’t find out how child ended up.\n", argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
22 }

    My child’s exit status was 1.
```

system in perl

http://i5.nyu.edu/~mm64/x52.9544/src/systemexit

```
#!/bin/perl
use POSIX;

$status = system(’who | grep -q \’^abc1234[ \t]\’’);

if (WIFEXITED($status)) {
    print "My child’s exit status was ", WEXITSTATUS($status), ".\n";
} elsif (WIFSIGNALED($status)) {
    print "My child was terminated by signal number ", WTERMSIG($status), ".\n";
} elsif (WIFSTOPPED($status)) {
    print "My child was stopped by signal number ", WSTOPSIG($status), ".\n";
} else {
    die "$0: couldn’t find out how child ended up.";
}

exit 0;
```

▼ **Homework 2.1: get the child’s exit status**

Call `system` to give birth to a child that produces an exit status but no output. Then print a message determined by the exit status of the child. Let the child be one of the following programs, or a pipeline ending with one of the following programs. Or write your own child in C, C++, Perl, or the shell language.

`1$ mail -e`	*exit status is 0 if you have mail*
`2$ grep -q word file`	*exit status is 0 if* `file` *contains* `word`
`3$ cmp -s file1 file2`	*exit status is 0 if* `file1` *and* `file2` *are identical*
`4$ sort -c file 2> /dev/null`	*exit status is 0 if* `file` *is already sorted*
`5$ gcc -o /dev/null prog.c`	*exit status is 0 if* `prog.c` *has no compilation errors*
`6$ test -f file`	*exit status is 0 if* `file` *exists*
`7$ test -f file -a -w file`	*exit status is 0 if* `file` *exists and is writable*
`8$ test -d directory`	*exit status is 0 if* `directory` *exists*
`9$ mkdir directory`	
`10$ test ‘who | awk ’{print $1}’ | sort | uniq | wc -l‘ -gt 20`	
`11$ true`	*exit status always 0*
`12$ false`	*exit status always 1*
`13$ /usr/sbin/ping -c 1 acf5.nyu.edu > /dev/null 2>&1`	*exit status is 0 if* `is` *is online*

Here are some machines you can `ping`:

`andrew.cmu.edu`	*Carnegie Mellon University*
`www.uquebec.ca`	*Universite du Quebec*
`www.unipi.it`	*Università degli Studi di Pisa*

▲

Give birth to a child with a pipe from the parent to the child

popen runs another program and lets you send output to its standard input. **pclose** sends an EOF through the pipe and returns the exit status of the program run by **popen**. You can store it in an **int** variable and examine it with the **W** macros:

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/popen_to_child.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4
 5 int main(int argc, char **argv)
 6 {
 7     FILE *lpr = popen("lpr", "w");
 8     int status;
 9
10     if (lpr == NULL) {
11         perror(argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     fprintf(lpr, "hello\n");
16     fprintf(lpr, "goodbye\n");
17
18     status = pclose(lpr);
19
20     if (WIFEXITED(status)) {
21         printf("My child’s exit status was %d.\n", WEXITSTATUS(status));
22     } else if (WIFSIGNALED(status)) {
23         printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
24     } else if (WIFSTOPPED(status)) {
25         printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
26     } else {
27         fprintf(stderr, "%s: couldn’t find out how child ended up.\n", argv[0]);
28         return EXIT_FAILURE;
29     }
30
31     return EXIT_SUCCESS;
32 }
```

The string that you give to **popen** is not limited to one program. You can change line 7 to

```
 7     FILE *lpr = popen("sort | cat -n | pr -l60 | lpr", "w");
```

Use Bourne shell syntax.

Give birth to a child with a pipe from the child to the parent

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/popen_to_parent.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4
 5 int main(int argc, char **argv)
 6 {
    FILE *wc = popen("who | awk ’{print $1}’ | sort | uniq | wc -l", "r");
    int n;  /* number of people logged in */
    int status;

    if (wc == NULL) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    fscanf(wc, "%d", &n);
    printf("There are %d people logged in.\n", n);

    status = pclose(wc);
    if (!WIFEXITED(status) || WEXITSTATUS(status) != EXIT_SUCCESS) {
        fprintf(stderr, "%s: child came to grief somehow.\n", argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

You can call **popen** several times in the same C program. This allows you to have more than one pipe coming into and/or going out of a C program (or a Perl program), which you can’t have in a shellscript.

popen in Perl

http://i5.nyu.edu/~mm64/x52.9544/src/popen

```
#!/bin/perl
use POSIX;

open(WC, ’who | awk \’{print $1}\’ | sort | uniq | wc -l |’) || die "$0: $!";
open(LPR, ’| lpr’) || die "$0: $!";

$_ = <WC>;
chomp;
print LPR "There are $_ people logged in.\n";

close WC;
#Should have checked WIFEXITED before calling WEXITSTATUS.
print ’The exit status of the wc -l was ’, WEXITSTATUS($?), ".\n";

close LPR;
print ’The exit status of the lpr was ’, WEXITSTATUS($?), ".\n";

exit 0;
```

▼ **Homework 2.2: pipe data to sort**

Make Homework 13.4 list everything in alphabetical order. Simply use **popen**, **fprintf**, and **pclose** to pipe your C program’s output to **sort +8**.

```
int main(int argc, char **argv)
{
    opendir;
    popen("sort", "w");

    fprintf all the output into the pipe;

    pclose;
    closedir;
}
```

If you have done the extra credit parts of Homework 1.8, some lines of output will not have nine fields, so **`sort +8`** won't work. Use

```
    awk '{print $NF, $0}' | sort | sed 's/^[^ ][^ ]* //'
```

instead. (The **`sed`** remove everything up to and including the first blank on each line.)
▲

The hidden price of the system and popen functions: Curry pp. 292−295, 355−357; *Oedipus Rex*

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/hidden.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  /* for getpid */

int main()
{
    printf("My PID is %d.\n\n", getpid());
    system("ps -Af");
    return EXIT_SUCCESS;
}
```

selected output shown below

```
1$ a.out | more
My PID is 14473.

USER       PID  PPID %CPU STARTED  TTY             TIME COMMAND
mm64     12577 11823  0.0 19:58:50 ttyq2        0:04.72 -ksh (ksh)
mm64     14473 12577  0.0 21:37:07 ttyq2        0:00.02 a.out
mm64     14478 14473  0.0 21:37:07 ttyq2        0:00.02 sh -c ps -Af
root     19059 14478  0.0 21:37:07 ttyq2        0:00.37 ps -Af
```

fork without exec

See Bach pp. 148, 192−200; Curry pp. 295−298; KP pp. 222−223; *Men Without Women* by Ernest Hemingway. Einstein said that space is what you measure with a rule, time is what you measure with a clock. To see the processes on a Windows system, righht-click the task bar at the botton of the screen and select **`Process Manager`**.

Why does the following program output three words instead of two?

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/fork.c`

```
#include <stdio.h>    /* for printf and perror */
#include <stdlib.h>   /* for EXIT_SUCCESS */
#include <unistd.h>   /* for fork */

int main(int argc, char **argv)
{
 7     printf("hello\n");
 8
 9     if (fork() < 0) {
10         perror(argv[0]);
11         return EXIT_FAILURE;
12     }
13
14     printf("goodbye\n");
15     return EXIT_SUCCESS;
16 }
```

```
1$ gcc -o ~/bin/fork fork.c
2$ ls -l ~/bin/fork

3$ fork
hello
goodbye
goodbye
```

Perl doesn't require the empty parentheses in line 4. But C does, and I'm a C programmer.

http://i5.nyu.edu/~mm64/x52.9547/src/fork.pl

```
#!/bin/perl

print "hello\n";
defined fork() or die "$0: $!";
print "goodbye\n";

exit 0;
```

Put the Perl program in your **~/bin** subdirectory and say

```
4$ cd ~/bin
5$ pwd

6$ chmod 755 fork.pl                    Make it executable: change mode to rwxr-xr-x
7$ ls -l fork.pl

8$ fork.pl
hello
goodbye
goodbye
```

▼ **Homework 2.3: always flush before forking**

Remove the **\n** from line 7 of the above program. Why does it now output **hello** twice, as well as **goodbye** twice?

```
hellogoodbye
hellogoodbye
```

See **_IOLBF** in **setvbuf**(3); Bach p. 239 ex. 1; Curry pp. 98−99.

In C, always do an **fflush(stdout);** (or better yet, an **fflush(NULL);**) immediately before a **fork**. In Perl, always do an **autoflush**.
▲

▼ Homework 2.4: how many times will it print "hello"?

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/fork3.c`

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

int main(int argc, char **argv)
{
    if (fork() < 0) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    if (fork() < 0) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    if (fork() < 0) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    printf("hello\n");
    return EXIT_SUCCESS;
}
```

```
1$ gcc -o ~/bin/fork3 fork3.c
2$ ls -l ~/bin/fork3
3$ fork3 | cat -n
```

`http://i5.nyu.edu/~mm64/x52.9547/src/fork3.pl`

```
#!/bin/perl

defined fork() or die "$0: $!";
defined fork() or die "$0: $!";
defined fork() or die "$0: $!";

print "hello\n";
exit 0;
```

Put the Perl program in your **`~/bin`** subdirectory and say

```
4$ cd ~/bin
5$ pwd

6$ chmod 755 fork3.pl
7$ ls -l fork3.pl
8$ fork3.pl | cat -n
```

Make it executable: change mode to **`rwxr-xr-x`**

How not to use fork

See error **`EAGAIN`** in **`fork`**(2) and **`intro`**(2).

```
/* For pedagogical purposes only.  Do not try this! */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    for (;;) {
        if (fork() < 0) {
            perror(argv[0]);
            return EXIT_FAILURE;
        }
    }
}
```

```
#!/bin/perl
#For pedagogical purposes only.  Do not try this!

for (;;) {
    defined fork() or die "$0: $!";
}
```

Parent and child

See Curry pp. 284−285, 295−298. To see the PID number of each process,

```
1$ ps -Af | more                        every process ("all")
2$ ps -f | more                         just your own
     UID   PID  PPID  C    STIME TTY      TIME CMD
    mm64  1637  1635  0 09:00:52 pts/33   0:01 -ksh
etc.
```

The process in which **`pid > 0`** is called the *parent;* the one in which **`pid == 0`** is called the *child.* The standard output of the child is automatically directed to the same destination as the standard output of the parent.

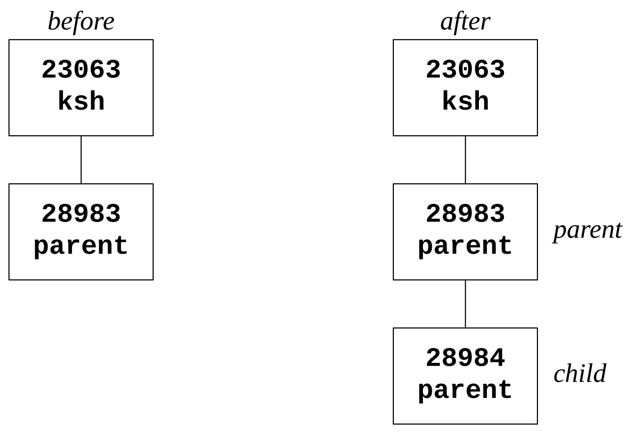
```
/* Excerpt from /usr/include/sys/types.h.
typedef int pid_t;
```

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/parent.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>   /* for pid_t */
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid;

    printf("My PID is %d and my parent's PID is %d.\n", getpid(), getppid());
    fflush(NULL);

13      pid = fork();
14      if (pid < 0) {
15          perror(argv[0]);
16          return EXIT_FAILURE;
17      }
18
19      printf("My PID is %d and my parent’s PID is %d.  fork returned %d.\n",
20          getpid(), getppid(), pid);
21
22      return EXIT_SUCCESS;
23  }
```

```
3$ gcc -o ˜/bin/parent parent.c
4$ ls -l ˜/bin/parent
5$ parent
My PID is 28983 and my parent’s PID is 23063.                        before the fork
My PID is 28983 and my parent’s PID is 23063.  fork returned 28984.  parent
My PID is 28984 and my parent’s PID is 28983.  fork returned 0.      child
```

The last two lines above will not always come out in this order.



You can combine lines 13−14 to

```
13      if ((pid = fork()) < 0) {
```

http://i5.nyu.edu/~mm64/x52.9547/src/parent.pl

```
#!/bin/perl
use FileHandle;   #for autoflush
STDOUT->autoflush(1);

$ppid = ‘ps -o ppid= -p $$‘;
chomp $ppid;
print "My PID is $$ and my parent’s PID is $ppid.\n";

$pid = fork();
defined $pid or die "$0: $!";

$ppid = ‘ps -o ppid= -p $$‘;
chomp $ppid;
print "My PID is $$ and my parent’s PID is $ppid.  fork returned $pid.\n";

exit 0;
```

Make the parent and child do different things

See Curry pp. 296−298.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/different1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid = fork();
    if (pid < 0) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    if (pid == 0) {
        printf("I am the child.\n");
    } else {
        printf("I am the parent.\n");
    }

    return EXIT_SUCCESS;
}
```

```
1$ gcc -o ~/bin/different1 different1.c
2$ ls -l ~/bin/different1
3$ different1
I am the parent.
I am the child.
```

The last two lines above will not always come out in this order.

http://i5.nyu.edu/~mm64/x52.9547/src/different1.pl

```
#!/bin/perl
use FileHandle;   #for autoflush
STDOUT->autoflush(1);

$pid = fork();
defined $pid or die "$0: $!";

if ($pid == 0) {
    print "I am the child.\n";
} else {
    print "I am the parent.\n";
}

exit 0;
```

The above program appears to be a classic opportunity to use **`if-then-else`**. But write it the following way instead, because the child's code will be short while the parent's code will go on and on.

Write the child's code before the parent's. The child's code must *always* end with a **`return`** from **`main`** or with an **`exit`** (line 16):

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/different2.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/types.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     pid_t pid = fork();
 9     if (pid < 0) {
10         perror(argv[0]);
11         return EXIT_FAILURE;
12     }
13
14     if (pid == 0) {
15         printf("I am the child.\n");
16         return EXIT_SUCCESS;
17     }
18
19     printf("I am the parent.\n");
20     return EXIT_SUCCESS;
21 }
```

```
4$ gcc -o ~/bin/different2 different2.c
5$ ls -l ~/bin/different2
6$ different2
I am the parent.
I am the child.
```

http://i5.nyu.edu/~mm64/x52.9547/src/different2.pl

```
#!/bin/perl
use FileHandle;   #for autoflush
STDOUT->autoflush(1);

$pid = fork();
defined $pid or die "$0: $!";

if ($pid == 0) {
    print "I am the child.\n";
    exit 0;
}

print "I am the parent.\n";
exit 0;
```

exec without fork

See Bach pp. 217−227; Curry pp. 298−301; KP p. 220−222. The following program transforms itself into **`cal`** by calling **`execl`**. It retains no trace of its previous identity, so there is no way to undo the transformation. If the transformation succeeded, the statement(s) after the **`execl`** (lines 11−12) will therefore be destroyed before they have a chance to execute.

Always follow **`execl`** with a **`perror`**. Why is there no need to write lines 11−12 in an **`if`**? What does the **`fflush`** prevent?

`fork` creates a new process and adds it to the tree of processes; **`execl`** doesn't.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/execl.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4
 5 int main(int argc, char **argv)
 6 {
 7     printf("I am about to transform myself into the cal program.\n");
 8     fflush(NULL);
 9
10     execl("/bin/cal", "cal", "9", "1752", (char *)0);
11     perror(argv[0]);
12     return EXIT_FAILURE;
13 }
```

```
1$ gcc -o ~/bin/execl execl.c
2$ ls -l ~/bin/execl
3$ execl
I am about to transform myself into the cal program.
      September 1752
Sun Mon Tue Wed Thu Fri Sat
         1   2  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
```

`die` gives us no control of the exit status number, so we use **`warn`** and exit instead.

http://i5.nyu.edu/~mm64/x52.9547/src/exec.pl

```
#!/bin/perl

use FileHandle;
STDOUT->autoflush(1);
print "I am about to transform myself into the cal program.\n";

exec {'/bin/cal'} 'cal', '9', '1752';   #first arg has braces, not comma
warn "$0: $!";
exit 2;       #a number different from any that could be retunned by cal
```

▼ Homework 2.5: what can go wrong with exec

(1) What error message do you get if you misspell the first **`cal`**? To verify that the error message comes from **`perror`** (or from the **`die`** in Perl), remove the **`perror`** and try it again.

(2) What error message do you get if you try to turn yourself into a file whose **`x`** bits are off?

(3) What error message do you get if you try to turn yourself into a shellscript whose **`#!/bin/ksh`** line is misspelled or absent? See **`#!`** in **`execve`**(2).

(4) Change line 10 to

```
execl("/bin/ls", "ls", "-l", "*.c", (char *)0);
```

Why doesn't this list everything in the current directory whose name ends with **`.c`**? What does it try to list instead? See KP pp. 220−221.
▲

What the process retains after the exec

In the above example, **`cal`** inherits the following features (and more) from your C or C++ program. See Bach pp. 149−151, 221; Curry pp. 299−300; **`fork`**(2).

(1) **`PID`** and **`PPID`** numbers
(2) owner and group
(3) current directory
(4) control terminal
(5) environment variables
(6) the **`umask`**
(7) the right to use all the currently open file descriptors, but the new program should exercise this right only for file descriptors 0, 1, and 2.

▼ Homework 2.6: verify that the exec'ed process retains the right to use all the currently open file descriptors

Direct the above program's standard output into a file:

```
1$ execl > ~/outfile
```

Observe that even after **`prog`** transforms itself into **`cal`**, its standard output is still directed to **`~/outfile`**.
▲

▼ **Homework 2.7: the four flavors of exec**

	fixed number of arguments	*variable number of arguments*
don't use **$PATH**	**execl**	**execv**
use **$PATH**	**execlp**	**execvp**

Each of these four functions ultimately calls the system call **execve** to perform the transformation.

See Bach pp. 217, 245 ex. 35; Curry p. 299; **execl**(2); **execve**(2).

Make the following changes in the above C program:

(1) Change line 10 to

```
execlp("cal", "cal", "9", "1752", (char *)0);
```

Does it still work? **execlp** calls **getenv("PATH")**.

(2) Add the following array to the program

```
char *new_argv[] = {"cal", "9", "1752", (char *)0};
```

and change line 10 to

```
execv("/bin/cal", new_argv);
```

Does it still work? Where else have we seen an array of strings that holds the command line of a program?

(3) Change line 10 to

```
execvp("cal", new_argv);
```

Does it still work?

▲

Deceive a process about its own name

Write a program named **realname** that prints out its own name:

```
#!/bin/ksh

echo My name is $0.
exit 0
```

Run it like this:

```
execl("/home1/a/abc1234/bin/realname",
    "othername", "arg1", "arg2", "arg3", (char *)0);
```

realname will output

```
My name is othername.
```

ps will also display **othername** instead of **realname**. Is there an argument of **ps** that will display the real name?

ps shows that the Korn shell thinks that its name is **-ksh** instead of **ksh**. The Korn shell's parent must therefore have run it like this:

```
execl("/bin/ksh", "-ksh", ...
```

The first thing that the Korn shell does is to look at its own name. If the name starts with a dash, the Korn shell will execute the commands in its owner's **.profile** file. See **login**(1), **ksh93**(1) p. 33. For other programs that decide what to do by looking at their own names, see KP pp. 85−86.

fork, exec, and wait

In peace sons bury fathers, but in war fathers bury sons.
—Herodotus, *The Histories*, I, 87

See Bach pp. 213−227; Curry pp. 301−309; KP pp. 222−225.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/forkexecwait.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid = fork();
    int status;

    if (pid < 0) {
        perror(argv[0]);
        return 1;
    }

    if (pid == 0) {
        /* Arrive here if I am the child. */
        execl("/usr/xpg4/bin/grep",
            "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
        perror(argv[0]);
        return 3;       /* different from grep's exit status */
    }

    /* Arrive here if I am the parent. */
    pid = wait(&status);
    if (pid < 0) {
        perror(argv[0]);
        return 2;
    }
    printf("My child's PID number was %d.\n", pid);

    if (WIFEXITED(status)) {
        printf("My child's exit status was %d.\n", WEXITSTATUS(status));
    }

    else if (WIFSIGNALED(status)) {
        printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
    }

    else if (WIFSTOPPED(status)) {
        printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
    }

    else {
        fprintf(stderr, "%s: couldn't find out how child ended up.\n", argv[0]);
        return 3;
    }

    return EXIT_SUCCESS;
}
```

```
    1$ gcc -o ~/bin/forkexecwait forkexecwait.c
    2$ ls -l ~/bin/forkexecwait
    3$ forkexecwait
    My child’s PID number was 2759.
    My child’s exit status was 1.
```

See **wait**(2) for the various flavors of **wait**. See **signal**(3head) for a list of the signal numbers, or

```
    4$ awk '$1 == "#define" && $2 ~ /^SIG/' /usr/include/sys/iso/signal_iso.h | more
    #define SIGHUP   1   /* hangup */
    #define SIGINT   2   /* interrupt (rubout) */
    #define SIGQUIT  3   /* quit (ASCII FS) */
```

Bach p. 226: “Would it not be more natural to combine the two system calls [**fork** and **execl**] into one...? Ritchie surmises that **fork** and **exec** exist as separate system calls, because when designing the UNIX system, he and Thompson were able to add the **fork** system call without having to change much code in the existing kernel.”

The above child calls **execl** immediately after the **fork**. But later children will have alot of work to do between the **fork** and the **execl**. That’s the real reason why **fork** and **execl** are separate system calls.

▼ **Homework 2.8: examine the child’s exit status**

Run the above program. Give **/usr/xpg4/bin/grep** different arguments to verify that the parent can get three different exit status numbers from the child. 0 means that **grep** found what it was looking for; 1 means that **grep** didn’t find what it was looking for; 2 means that you gave **grep** an incorrect regular expression (e.g., **[abc[**) or a misspelled or read-protected filename. See **grep**(1).

Now misspell **/usr/xpg4/bin/grep** and verify that the child is unable to **execl** it and returns 3.

▲

▼ **Homework 2.9: rewrite Homework 14.1**

Rewrite Homework 14.1 using **fork**, **exec**, and **wait** instead of **system**.

▲

▼ **Homework 2.10: which is faster, system or fork-exec-wait?**

Write the smallest possible child:

```
#include <stdlib.h>
#include <unistd.h>

int main()
{
    _exit(EXIT_SUCCESS);
}
```

Write a C program that gives birth to this child 1000 times using **system** within a **for** loop. Time the C program with **/bin/time**:

```
1$ /bin/time cal 4 2008 > /dev/null

real        0.0
user        0.0
sys         0.0
```

Then write another C program that gives birth to this child 1000 times using **`fork`**, **`execl`**, and **`wait`** within a **`for`** loop. Which is faster?

▲

fork-exec-wait in Perl

`http://i5.nyu.edu/~mm64/x52.9544/src/forkexecwait.pl`

```
#!/bin/perl
use POSIX;

$pid = fork();
defined $pid or die "$0: $!";

if ($pid == 0) {
    #Arrive here if I am the child.
    exec {'/usr/xpg4/bin/grep'} 'grep', '-q', '^abc1234:', '/etc/passwd';
    warn "$0: $!";
    exit 3;
}

#Arrive here if I am the parent.
$pid = wait();
if ($pid < 0) {
    die "$0: $!";
}
print "My child's PID number was $pid.\n";

if (WIFEXITED($?)) {
    print "My child's exit status was ", WEXITSTATUS($?), ".\n";
}

elsif (WIFSIGNALED($?)) {
    print "My child was terminated by signal number ", WTERMSIG($?), ".\n";
}

elsif (WIFSTOPPED($?)) {
    print "My child was stopped by signal number ", WSTOPSIG($?), ".\n";
}

else {
    die "$0: couldn't find out how child ended up.";
}

exit 0;
```

```
1$ forkexecwait
My child's PID number was 1701.
My child's exit status was 1.
```

If we changed lines 4−5 to

```
$pid = fork() or die "$0: $!"
```

then we would **`die`** whenever the **`$pid`** was zero. But a zero **`$pid`** is not a cause for death—it just means that I'm the child.

wait for a specific child

See Curry p. 304.

	any child	*child whose PID is* **`pid`**
wait till child dies	**`wait(&status);`**	**`waitpid(pid, &status, 0);`**
return immediately	**`waitpid(-1, &status, WNOHANG);`**	**`waitpid(pid, &status, WNOHANG);`**

A process can give birth to a second child without first **`wait`**'ing for the elder child to die. Thus a process can be the parent of more than one child simultaneously.

`wait` returns the exit status of whichever child dies first. Since different children run at different speeds (as in life itself), this makes it impossible to predict which child will be harvested by a given call to **`wait`**. That's why **`wait`** returns the **`PID`** of the harvested child.

To wait for a specific child, use **`waitpid`** instead of **`wait`**:

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/waitpid.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid1 = fork();
    pid_t pid2;
    int status;

    if (pid1 < 0) {
        perror(argv[0]);
        return 1;
    }

    if (pid1 == 0) {
        /* Arrive here if I am the first child. */
        execl("/usr/xpg4/bin/grep",
            "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
        perror(argv[0]);
        return 3;   /* different from grep's exit status */
    }

    /* Arrive here if I am the parent. */
    pid2 = fork();
    if (pid2 < 0) {
        perror(argv[0]);
        return 2;
    }

    if (pid2 == 0) {
        /* Arrive here if I am the second child.  mailx -e returns
        exit status 0 if there is mail waiting for you, 1 otherwise. */
        execl("/bin/mailx", "mailx", "-e", (char *)0);
        perror(argv[0]);
        return 2;   /* different from mail’s exit status */
    }

    /* Arrive here if I am the parent.  I don’t necessarily have to wait
    for my children in the order in which they were born. */

    waitpid(pid2, &status, 0);
    if (WIFEXITED(status)) {
        if (WEXITSTATUS(status) == 0) {
            printf("There is mail waiting for you.\n");
        } else if (WEXITSTATUS(status) == 1) {
            printf("There is no mail waiting for you.\n");
        } else {
            printf("My second child couldn’t turn into the mailx program.\n");
        }
    } else if (WIFSIGNALED(status)) {
        printf("My second child (mailx) was terminated by signal number %d.\n",
            WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
        printf("My second child (mailx) was stopped by signal number %d.\n",
            WSTOPSIG(status));
    }

    waitpid(pid1, &status, 0);
    if (WIFEXITED(status)) {
        if (WEXITSTATUS(status) == 0) {
            printf("abc1234 has an account.\n");
        } else if (WEXITSTATUS(status) == 1) {
            printf("abc1234 has no account.\n");
        } else if (WEXITSTATUS(status) == 2) {
            printf("My first child (grep) couldn’t search /etc/passwd.\n");
        } else {
            printf("My first child couldn’t turn into the grep program.\n");
        }
    } else if (WIFSIGNALED(status)) {
        printf("My first child (grep) was terminated by signal number %d.\n",
            WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
        printf("My first child (grep) was stopped by signal number %d.\n",
            WSTOPSIG(status));
    }

    return EXIT_SUCCESS;
}
```

```
1$ gcc -o ~/bin/waitpid waitpid.c
2$ ls -l ~/bin/waitpid
3$ waitpid
abc1234 has no account.
There is mail waiting for you.
```

Non-blocking wait

See Curry p. 305.

In all of the above examples, **`wait`** and **`waitpid`** cause the process to block (i.e., wait and do nothing) until a child dies. To always return immediately from **`waitpid`**, use **`WNOHANG`**:

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/wnohang.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid = fork();
    int status;

    if (pid < 0) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    if (pid == 0) {
        /* Arrive here if I am the child. */
        execl("/usr/xpg4/bin/grep",
            "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
        perror(argv[0]);
        return 3;   /* different from grep’s exit status */
    }

    /* Arrive here if I am the parent. */

    if (waitpid(pid, &status, WNOHANG) == 0) {
        printf("The child isn’t ready for harvesting yet, which is\n");
        printf("okay because I have plenty of other work to do.\n");
    } else if (WIFEXITED(status)) {
        if (WEXITSTATUS(status) == 0) {
            printf("abc1234 has an account.\n");
        } else if (WEXITSTATUS(status) == 1) {
            printf("abc1234 has no account.\n");
        } else {
            printf("My child (grep) couldn’t search /etc/passwd.\n");
        }
    } else if (WIFSIGNALED(status)) {
        printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
        printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
    }

    pid = waitpid(-1, &status, WNOHANG);
    if (pid == 0) {
        printf("I have no children which are ready to be harvested right now.\n");
    } else {
        printf("My child’s PID number was %d and his exit status was %d.\n",
                pid, WEXITSTATUS(status));
    }

    pid = wait(&status);
    if (pid < 0) {
        printf("I have no other children at all.\n");
    } else {
        printf("My child’s PID number was %d and his exit status was %d.\n",
            pid, WEXITSTATUS(status));
    }

    return EXIT_SUCCESS;
}
```

```
1$ gcc -o ˜/bin/wnohang wnohang.c
2$ ls -l ˜/bin/wnohang
3$ wnohang
The child isn’t ready for harvesting yet, which is
okay because I have plenty of other work to do.
I have no children which are ready to be harvested right now.
My child’s PID number was 17109 and his exit status was 1.
```

http://i5.nyu.edu/~mm64/x52.9544/src/wnohang.pl

```
#!/bin/perl
use POSIX;

$pid = fork();
defined $pid or die "$0: $!";

if ($pid == 0) {
    #Arrive here if I am the child.
    exec {’/usr/xpg4/bin/grep’} ’grep’, ’-q’, ’ˆabc1234:’, ’/etc/passwd’;
    warn "$0: $!";
    exit 3;
}

#Arrive here if I am the parent.
if (waitpid($pid, WNOHANG)) {
    if (WIFEXITED($?)) {
        print "The child’s exit status is ", WEXITSTATUS($?), "\n";
    }
    exit 0;
}

print "The child isn’t ready for harvesting yet.\n";
wait();
if (WIFEXITED($?)) {
    print "The child’s exit status is ", WEXITSTATUS($?), "\n";
}

exit 0;
```

```
4$ wnohang.pl
The child isn’t ready for harvesting yet.
The child’s exit status is 1
```

▼ Homework 2.11: write a shell: Curry pp. 103−105

Hand in only the last version of this program. You get no credit if you hand in more than one version.

(1) Write a program named **`mysh`** (“my shell”) that will take no command line arguments. It will **`printf`** the word **`m1$`** (followed by one blank) as a prompt, and then **`gets`** one line from the standard input. Assume that line of input contains exactly one word. Here are three examples of input that **`mysh`** might receive:

```
date
cal
who
```

Give birth to a child that will **`execlp`** itself into the program whose name was read from the standard input. The parent, meanwhile, should **`wait`** for the child and then call **`exit`**.

```
 1 /* Excerpt from the file /usr/include/sys/syslimits.h. */
 2
 3 /* Maximum number of bytes in a command line.  See E2BIG in man intro(2). */
 4 #define ARG_MAX 38912

 5     declare an array of ARG_MAX characters named line;
 6
 7     printf the prompt;
 8     flush to make sure that the prompt is output immediately;
 9     if (gets(line) == NULL) {
10         fprintf an error message and exit;
11     }
12
13     fork;
14     if (I’m the child) {
15         /* Arrive here if I’m the child. */
16         execlp the line;
17         fprintf an error message and exit: the execlp must’ve failed;
18     }
19
20     /* Arrive here if I am the parent */
21     waitpid for the child to die without the WNOHANG;
22     exit or return from main;
```

(2) Remove the **`exit`** from the end of the above program, and wrap an infinite **`for`** loop around what remains. **`mysh`** will now read in many lines of input and give birth to the program named in each one. **`break`** out of the main loop when you encounter the end of the standard input or the word **`exit`**:

```
1 #include <string.h>     /* for strcmp */
2
3     for (;;) {          /* main loop */
4         printf the prompt;
5         if (gets(line) == NULL || strcmp(line, "exit") == 0) {
6             break out of the main loop;
7         }
8         do the fork, exec, wait in lines 13-20 of the previous paragraph;
```

```
    }

    /* Arrive here when at end of input. */
    exit;
```

Use **for (;;) {** for the infinite loop (K&R p. 60). You get no credit if you say **while (1) {**.

(3) Allow each line of input to have more than one word:

```
date
ls -l
grep ism$ /usr/dict/words
```

mysh *does not take single quotes.*

Assume that the words are separated from each other by one or more blanks and/or tabs.

Here's an example of a C program that splits a line into words, and stores a pointer to each word in an array of pointers named **new_argv**. Don't forget to store a **(char *)0** after the last word:

The second argument of **strtok** (K&R p. 250) contains one blank and one tab. **strtok** is easier to use than the **strpbrk** in Curry p. 317.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/strtok.c

```
/* Store the words in the line of input into an array of strings. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXWORDS 16

int main(int argc, char **argv)
{
    char line[ARG_MAX];
    char *new_argv[MAXWORDS];    /* an array of strings */
    int i;
    int new_argc;    /* the number of words in the line */

    if (gets(line) == NULL) {
        fprintf(stderr, "%s: requires one line of input.\n", argv[0]);
        return 1;
    }

    for (i = 0; i < MAXWORDS; ++i) {
        new_argv[i] = strtok(i == 0 ? line : NULL, " \t");
        if (new_argv[i] == NULL) {
            goto done;
        }
    }

    fprintf(stderr, "%s: more than %d words\n", argv[0], MAXWORDS);
    return 2;

    done:;
    new_argc = i;
    new_argv[new_argc] = (char *)0;

    /* Demonstrate that strtok works. */
    for (i = 0; i < new_argc; ++i) {
        printf("%s\n", new_argv[i]);
    }

    return EXIT_SUCCESS;
}
```

```
2$ prog
ls -l moe          You type this.
ls                 It outputs this.
-l                 It outputs this.
moe                It outputs this.
3$
```

In Perl, the regular expression **`\s+`** means **`[	][	]*`**, where each pair of square brackets enclose one blank and one tab. For documentation,

```
4$ man -M /usr/local/lib/perl5/man 3 Text::ParseWords
```

http://i5.nyu.edu/~mm64/x52.9544/src/quotewords

```
#!/bin/perl
use POSIX;

$_ = <STDIN>;    #Input one line from the standard input.
chomp;

@new_argv = quotewords('\s+', 1, $_);
$new_argc = @new_argv;
print "\$new_argc == $new_argc.\n";

for ($i = 0; $i < $new_argc; ++$i) {
    print "$new_argv[$i]\n";
}

exit 0;
```

The parent should parse the command line into separate words before calling **`fork`**. Between the parse and the **`fork`**, print out words in **`new_argv`**. In other words, **`mysh`** should operate like the Korn shell with **`set -x`** or the C shell with **`set echo`**. If **`new_argv`** contains no words (i.e., if **`new_argc`** equals zero), do not bother to **`fork`**. Otherwise, if the first word in **`new_argv`** is **`exit`**, **`break`** out of the main loop. (Remove the cruder test for **`exit`** shown earlier.)

After the **`fork`**, the child now has to call **`execvp`** instead of **`execlp`**, and should use **`new_argv`** as the second argument of **`execvp`**.

(4) Before calling **`fork`**, the parent should see if the last word of the line is an ampersand. (Call **`strcmp`** only once per line to do this: you do not have to examine any word except the last.) If so, the parent should remove the ampersand from the **`new_argv`** array (and subtract one from **`new_argc`**). The parent should then call **`fork`** only if **`new_argv`** still contains at least one word. (In other words, don't give birth to a child if some joker types in a command line consisting only of an ampersand.) And after **`fork`**'ing, the parent should merely print the child's PID number instead of **`wait`**'ing for the child.

If you end many command lines with ampersands, it will cause you to give birth to many children without **`wait`**'ing for them to die. This will cause the machine to fill up with children waiting to be harvested. At the end of the main **`for (;;) {`** loop (just before you go back to the top and print the prompt again), harvest all the background children that are ready to be harvested (if any):

```
            /* Arrive here if I am the parent. */
```

```

        if (the last word of the line of input to mysh was "&") {
            print("Running PID %d in the background.\n", pid);
        } else {
            waitpid for that pid;
        }

        /* Harvest any children that are ripe (i.e., zombies). */
        while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
            if (WIFEXITED
                printf("Background PID %d terminated with exit status %d.\n",
                    pid, WEXITSTATUS(status));
            }
        }
    }                              /* end of main for (;;) loop */

    /* Don't bother to wait for last crop of children, if any. */
    return EXIT_SUCCESS;
}   /* end of main */
```

There must be one or more blanks and/or tabs before the ampersand:

```
who
find /home1/a/abc1234 -type f -name core -print &
date
```

(5) If one of the words in **new_argv** starts with a dollar sign, replace the entire word (including the leading dollar sign) with the value of the corresponding environment variable. Call **getenv**.

(6) If one of the words in **new_argv** is an asterisk all by itself, remove that word from **new_argv**. Insert in its place the names of everything in the current directory, each as a separate element in the **new_argv** array. Also add the correct amount to **new_argc**. If the current directory is empty, leave the asterisk untouched.

(7) Before parsing the command line into separate words, call **strchr** (Curry pp. 21−22) to see if the command line contains a **'#'**. If so, remove the **'#'** and all the characters after it by changing the **'#'** to a **'\0'**.

(8) I wish you could create a file:

```
#!/home1/a/abc1234/bin/mysh
#This file is named shelly.
date
cal
who
```

and execute it like this:

```
5$ shelly
```

Thus far **mysh** has taken no command line arguments and reads from the standard input. Change **mysh** to read from the standard input only if there are no command line arguments. If there is an argument, assume it's a filename and read from it instead. If there is more than one argument, output an error message and die. Output the prompt only if **fp** is coming from a terminal.

You must remove the **'\n'** after a **fgets** but not after a **gets**. See K&R pp. 164, 247.

fileno simply returns the **_file** field of the structure whose address is given as an argument.

```
    FILE *fp;

    if (argc == 1) {
        fp = stdin;
    } else if (argc == 2) {
        fp = fopen argv[1] as an input file;
        check for fopen error
    } else {
        too many arguments: exit or return from main;
    }

    /* start of main loop */
    for (;;) {
        if (isatty(fileno(fp)) {
            print the prompt;
        }
        if (fgets(line, sizeof line, fp) == NULL) {
            end of input: break out of the main for loop
        }
        remove the '\n' from the end of the line;
        parse the line, store the individual words in new_argv;
```

In Perl, the filehandle **ARGV** inputs from the files named as command line arguments, or from the standard input if there were no arguments. **-t** stands for "terminal". See pp. 140 and 85 respectively in the O'Reilly *Programming Perl, 2nd. ed.*

```
#!/bin/perl
if (@ARGV > 1) {
    too many arguments: exit;
}

#start of main loop
for (;;) {
    if (-t ARGV) {
        print the prompt;
    }
    $_ = <ARGV>;
    etc.
```

(9) ✎ For space cadets only. Instead of reading the standard input with a **gets** loop, use **yacc**. Allow the input to contain expressions such as

```
prog1 ; prog2
prog1 && prog2
prog1 || prog2
prog1 && prog2 || prog3
prog1 && ( prog2 || prog3 )
prog1 && { prog2 || prog3 }
prog1 arg1 arg2 arg3 && prog2
```

What precedences and associativities should these operators have?
▲

▼ Homework 2.12: photograph a zombie: Bach pp. 148−149, 212−213; Curry p. 286

A process calls the **exit** function when it has finished its work. Even after calling **exit**, however, an empty husk of the process continues to exist. This ghoulish remnant, called a *zombie*, serves only to hold the exit status of the process until the parent harvests it by calling **wait** (or **waitpid** or **wait3**). **wait** then removes the zombie.

Thus the real effect of **`exit`** is to turn a process into a zombie. Zombies are rarely seen because the parent of each process usually calls **`wait`** promptly. To verify that zombies exist, write a program that **`fork`**'s into two processes. The child should **`exit`** immediately, thus turning itself into a zombie. The parent should **`sleep`** for 10 seconds (to give the child time to **`exit`**) and should then call **`execl`** to transform itself into **`ps -Aj`** (for "job control"). Hand in your program and its output, showing a process whose name is **`<defunct>`**. (In other versions of Unix, the **`S`**tatus of this process will be **`Z`**).

changes into **`ps -Aj`** *before the following output appears.*

```
1$ a.out
USER       PID  PPID  PGID  SESS JOBC S    TTY              TIME COMMAND
mm64     19224   847 19224 19224    0 S    ttyp7         0:01.20 -csh (csh)
root     21005 19224 21005 19224    1 R  + ttyp7         0:00.37 ps -Aj
mm64     20903 21005 21005 19224    1   <+ ttyp7         0:00.00 <defunct>
```

▲

▼ Homework 2.13: who adopts an orphaned child? Bach p. 213; Curry p. 287

Write a program that **`fork`**'s. The parent should **`exit`** immediately. The child should **`sleep`** for 10 seconds (to give the parent time to **`exit`**) and should then print its parent's PID number by calling **`getppid`**.

▲

dup2 in the Korn shell language: KP pp. 93, 141−142

The standard output (file descriptor 1) and standard error output (file descriptor 2) of a process are usually directed to two different destinations. The **`1>&2`** in the following shellscript, however, tells **`echo`** to direct all the data that it outputs through file descriptor 1 to the same destination used by file descriptor 2. In other words, all the **`print`**'s and **`fprintf(stderr`**'s in **`echo`** will now go to the same destination. This common destination is the same as the destination that was originally used by the **`fprintf(stderr`**'s.

```
#!/bin/ksh

if [[ $# -ne 1 ]]
then
    echo $0: requires 1 command line argument 1>&2
    exit 1
fi

echo I received the argument $1.
exit 0
```

Standard output to **`outfile`**, *standard error output to terminal:*

```
1$ shelly arg1 arg2 arg3 > outfile
```

A World Wide Web gateway is another example of a program whose standard output and standard error output are directed to the same destination.

dup vs. dup2: Bach pp. 117−119, 144; Curry pp. 68−69, 312; KP pp. 223−225

The single system call

```
dup2(2, 1);
```

does the work which used to be done by the two system calls

```
close(1);
dup(2);
```

Redirect the standard output of a C program with dup2 and close: Bach pp. 117−119; Curry pp. 24−25, 105−106; KP pp. 223−224

The destination of output that you **`write`** to file descriptor 1 (the standard output) is usually determined by the command line that ran the program:

```
1$ prog > outfile
2$ prog | subsequent_prog
3$ prog
```

The following program, however, will always take send its standard output to the file **`outfile2`** even if you run it like this:

```
4$ prog > outfile1
```

(1) At line 9, we're allowed to say

```
write(1, "hello", 5);
```

(2) At line 15, we're allowed to say both of the following

```
write(1, "hello", 5);
write(fd, "hello", 5);
```

to send output to two different destinations.

(3) At line 20, we're still allowed to say both of the following

```
write(1, "hello", 5);
write(fd, "hello", 5);
```

but they now send their output to the same destination, namely **`outfile2`**.

(4) At line 25, we're allowed to say only

```
write(1, "hello", 5);
```

which continues to send its output to **`outfile2`**. The **`printf`** in line 26 ultimately calls **`write(1,`**. In other words, we can still perform standard output, but it now goes to **`outfile2`**.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/dup2.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <fcntl.h>   /* for O_CREAT */
 4 #include <sys/stat.h>    /* for S_IRUSR */
 5 #include <sys/types.h>
 6 #include <unistd.h>  /* for dup2 */
 7
 8 int main(int argc, char **argv)
 9 {
10     int fd = open("outfile2", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR);
11     if (fd < 0) {
12         perror(argv[0]);
13         return 1;
14     }
15
16     if (dup2(fd, 1) < 0) {
17         perror(argv[0]);
18         return 2;
19     }
20
```

```
    if (close(fd) != 0) {
        perror(argv[0]);
        return 3;
    }

    printf("hello\n");
    return EXIT_SUCCESS;
}
```

See **freopen**(3) for another way to do this.

▼ Homework 2.14: redirect the standard input

Write a program that calls **open**, **dup2**, and **close** to override the source of the standard input specified on the command line. Take the standard input from the file **/dev/tty** instead. Verify that the input comes from the terminal even if you say

```
1$ prog < infile
```

▲

Give the child a different source of standard input: Bach pp. 117−119; Curry pp. 311−318; KP pp. 223−224

A child automatically inherits the source of its parent’s standard input. The child can redirect its standard input, however, before it transforms itself into a different program. The redirection survives the transformation.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/dup22.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid = fork();
    int fd;
    int status;

    if (pid < 0) {
        perror(argv[0]);
        return 1;
    }

    if (pid == 0) {
        /* Arrive here if I am the child. */
        fd = open("letter", O_RDONLY, 0);
        if (fd < 0) {
            perror(argv[0]);
            return 1;
        }

        if (dup2(fd, 0) < 0) {
            perror(argv[0]);
            return 2;
        }

        if (close(fd) != 0) {
            perror(argv[0]);
            return 3;
        }

        execl("/usr/ucb/mailx", "mail", "def5678", (char *)0);
        perror(argv[0]);
        return 4;
    }

    /* Arrive here if I am the parent. */
    wait(&status);
    if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
        printf("The letter was successfully mailed.\n");
        return EXIT_SUCCESS;
    } else {
        printf ("The letter was not mailed.\n");
        return EXIT_FAILURE;
    }
}
```

▼ Homework 2.15: give the child a different destination for its standard output

Give birth to a child. Have the child change the destination of its standard output. Then have the child transform itself into another program. The parent, meanwhile, will **wait** for the child's death. You get no credit unless the third argument of **open** is written with the macros **S_IRUSR**, **S_IWUSR**, etc. You get no credit if you turn on any of the **x** bits of any file. You get no credit if you use the macro **S_IRDWR**. You get no credit if you call the **creat** system call.

✎ Extra credit. Have the child change both its standard input and its standard output.

▲

▼ Homework 2.16: write a shell that accepts <, >, and >>

Change **mysh** to accept lines of standard input like this:

```
prog
prog < infile
prog > outfile
prog < infile > outfile
prog arg1 arg2 arg3 < infile arg4 arg5 arg6 >> outfile &
```

where **prog** is the name of any program and **infile** and **outfile** are filenames. For simplicity, assume there are one or more blanks and/or tabs on either side of each **<** and **>**.

Between its birth and its **execlp**, each child should call a function declared as

```
int redirect(int new_argc, char **new_argv);
```

This function should look for the redirection symbols **<**, **>**, etc., and change the child's standard input and output to the requested source and destination. The parent should have nothing to do with the redirection symbols.

If the child opens a **<** file, it should give **O_RDONLY** and **0** to **open** as the second and third arguments. If the child opens a **>** file, it should give **O_CREAT | O_TRUNC | O_WRONLY** and **S_IRUSR | S_IWUSR** (i.e., **rw-------**) to **open** as the second and third arguments. If the child opens a **>>** file, it should give **O_CREAT | O_WRONLY** and **S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH** to **open** as the second and third arguments. The child must also call

```
#include <sys/types.h>
#include <unistd.h>

    if (lseek(fd, 0L, SEEK_END) < 0) {    /*  Curry pp. 65−68 */
        perror
        exit
    }
```

immediately after **open**'ing a **>>** file, assuming that **fd** holds the return value of the **open**.

Don't do it like this:

```
    if (strcmp(new_argv[i], "<"
        open(, O_RDONLY, 0
        dup2(, 0)
        close
    } else if (strcmp(new_argv[i], ">"
        open(, O_CREAT | O_TRUNC | O_WRONLY, S_ISUSR | S_IWUSR
        dup2(, 1)
        close
    } else if (strcmp(new_argv[i], "2>"
        open(, O_CREAT | O_TRUNC | O_WRONLY, S_ISUSR | S_IWUSR
        dup2(, 2)
        close
    } else /* etc. */
```

Instead, make an array of structures whose initialization is something like

```
/* The second field is the file descriptor number. */
typedef struct {
    ...
} redirect_t;

redirect_t a[] = {
    {"<",   0,    O_RDONLY,                           0},

    {">",   1,    O_CREAT | O_TRUNC | O_WRONLY,       S_IRUSR | S_IWUSR},
    {">>",  1,    O_CREAT | O_WRONLY,                 S_IRUSR | S_IWUSR},

    {"2>",  2,    O_CREAT | O_TRUNC | O_WRONLY,       S_IRUSR | S_IWUSR},
    {"2>>", 2,    O_CREAT | O_WRONLY,                 S_IRUSR | S_IWUSR},

    {NULL,  -1,   0,                                  0}    /* end of data */
};
```

Remove the **>**, **<**, **>>**, and the following filename from the **new_argv** array before giving birth, just as you removed the ampersand and asterisk. Move all the subsequent words two positions forward, including the **(char *)0** after the last word.

If you're really brave, try to use **chsh** or the **-s** option of **passwd** to change the seventh field of your line in the **/etc/passwd** file to **/home1/a/abc1234/bin/mysh**. See **finger**(1), **chsh**(1).
▲

http://i5.nyu.edu/~mm64/x52.9544/src/winner

```
#!/bin/perl

$_ = <STDIN>;
chomp($_);   #Remove the trailing newline from $_.

if (length($_) != 9 || $_ =~ /[^XO ]/) {
    die "$0: Input line must be nine X’s, O’s, or blanks.";
}

#Insert a dash after the 3rd character and after the 6th character.
#For example, OOXXXOOXX becomes OOX-XXO-OXX.

$_ =~ s/(...)(...)(...)/\1-\2-\3/;

if (
    $_ =~ /([XO])\1\1/   ||      #any row
    $_ =~ /([XO])...\1...\1/ ||     #any column
    $_ =~ /([XO])..-.\1.-..\1/ ||   #the main diagonal
    $_ =~ /([XO])-.\1.-\1/) {       #the other diagonal

    print "$1 is a winner.\n";
    exit 0;
}

print "No one has won yet.\n";
exit 1;
```

□