Fall 2005 Handout 7

Another Perl program with a loop: a TCP port scanner

A TCP port number is 16 bits, so there are $2^{16} = 65,536$ possible TCP port numbers. (Ditto for UDP ports.) Every one of them is a potential entry point for an attacker. The underscore in the **65_536** in line 18 stands for a comma.

Line 24 creates the ''filehandle'' **S** just as line 23 creates the variable **$address**. Both variables must be fed to the **connect** function in line 27. The **AF_INET** in line 24 means ''IP version 4''; the **SOCK_STREAM** means ''TCP''.

The functions in lines 24 and 34 (and 27) are Unix system calls. They are part of the operating system, and therefore deposit their error messages into the special variable **$!**. On the other hand, the functions in lines 13 and 23 (and 29) are not part of the operating system, so we have to write our own error messages for them.

The **connect** function in line 27 might take a long time to return to us. If we're still stuck in the **connect** after 15 seconds, line 25 will wake us up with an alarm. Line 16 will then send us to the subroutine in line 40. The subroutine does nothing: it sends us straight back to the **connect** in line 27. But the visit to the subroutine will make the **connect** terminate and return the undefined value.

See also

http://scan.sygate.com/

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/port_scanner.pl

```
 1 #!/bin/perl
 2 #Output the numbers of the TCP ports we were able to open on the given host.
 3 #The argument is the hostname or IP address of the given host,
 4 #e.g., i5.nyu.edu or 128.122.253.152
 5
 6 use Socket;
 7
 8 if (@ARGV != 1) {
 9     die "$0: requires one argument";
10 }
11
12 $hostname = $ARGV[0];
13 $ip = inet_aton($hostname)
14     or die "$0: argument $hostname must be a hostname or dotted IP address";
15
16 $SIG{ALRM} = 'handle_sigalrm';
17
18 for ($port = 0; $port < 65_536; ++$port) {
19     if ($port % 1000 == 0) {
20         print "port $port\n";   #to assure us that we're making progress
21     }
22
23     $address = sockaddr_in($port, $ip) or die "$0: sockaddr_in failed";
```

```
24      socket(S, AF_INET, SOCK_STREAM, 0) or die "$0: $!";
25      alarm(15);
26
27      if (connect(S, $address)) {
28          print "$hostname $port";
29          $servername = getservbyport($port, 'tcp');
30          if (defined $servername) {
31              print " $servername";
32          }
33          print "\n";
34          shutdown(S, SHUT_RDWR) or die "$0: $!";
35      }
36  }
37
38  exit 0;
39
40  sub handle_sigalrm {
41          if ($_[0] ne 'ALRM') {
42                  die "$0: received signal $_[0] instead of ALRM";
43          }
44  }
```

```
1$ port_scanner.pl 128.122.253.152 | head | cat -n
     1  port 0
     2  128.122.253.152 22 ssh
     3  128.122.253.152 25 smtp
     4  128.122.253.152 80
     5  128.122.253.152 111 sunrpc
     6  128.122.253.152 898
     7  port 1000
```

The **/etc/nsswitch.conf** file (p. 271) tells the **getservbyport** function in the above line 29 to get its information from the file **/etc/services**.

```
2$ awk '$1 == "services:"' /etc/nsswitch.conf
services:   files

3$ awk '$2 == "22/tcp"' /etc/services
ssh     22/tcp              # Secure Shell

4$ man -s 4 services
```

The wildcard has one blank and one tab in the **[**square brackets**]**.

```
5$ lynx -source http://www.iana.org/assignments/port-numbers | grep '[    ]22/tcp'
ssh             22/tcp    SSH Remote Login Protocol
```

▼ Homework 7.1: probe for security holes

Combine the above Perl program with the **localhosts.pl** in Handout 1, p. 26 to try to connect to every port on every host on your local network. The first command line argument will be the name or IP address of the network, in our case **NYU-FDDI253-128-NET** or **128.122.253.128** (Handout 1, p. 21). The second command line argument will be the number of network bits in the netmask. For our network, it would be **26**.

Use classic nested **for** loops. We saw the outer one in Handout 1, p. 27, line 33 of **localhosts.pl**.


```
1 for ($i = $network + 1; $i < $broadcast; ++$i) {
2     for ($port = 0; $port < 65_536; ++$port) {
```

Run the Perl program in the background because it takes too long; **nohup** will keep the script going even if you log out.

```
1$ nohup localhosts.pl > myscript.out 2>&1 &
2$
```

Hand in the script and a few lines of output. Do any interesting port numbers respond?
▲

fork without exec

See Bach pp. 148, 192−200; David Curry's O'Reilly book *Using C on the UNIX System* 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

```
 1 #include <stdio.h>    /* for printf and perror */
 2 #include <stdlib.h>   /* for EXIT_SUCCESS */
 3 #include <unistd.h>   /* for fork */
 4
 5 int main(int argc, char **argv)
 6 {
 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.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/fork.pl

```
1 #!/bin/perl
2
3 print "hello\n";
4 defined fork() or die "$0: $!";
5 print "goodbye\n";
6
```

```
 7 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 7.2: 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; David Curry's O'Reilly book *Using C on the UNIX System* 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 7.3: how many times will it print "hello"?

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

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

```
24 }
```

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

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/fork3.pl

```
 1 #!/bin/perl
 2
 3 defined fork() or die "$0: $!";
 4 defined fork() or die "$0: $!";
 5 defined fork() or die "$0: $!";
 6
 7 print "hello\n";
 8 exit 0;
```

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

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

6$ chmod 755 fork3.pl          Make it executable: change mode to rwxr-xr-x
7$ ls -l fork3.pl
8$ fork3.pl | cat -n
```

How not to use fork

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

```
 1 /* For pedagogical purposes only.  Do not try this! */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     for (;;) {
 9         if (fork() < 0) {
10             perror(argv[0]);
11             return EXIT_FAILURE;
12         }
13     }
14 }
```

```
 1 #!/bin/perl
 2 #For pedagogical purposes only.  Do not try this!
 3
 4 for (;;) {
 5     defined fork() or die "$0: $!";
 6 }
```

Parent and child

See David Curry's O'Reilly book *Using C on the UNIX System* 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.

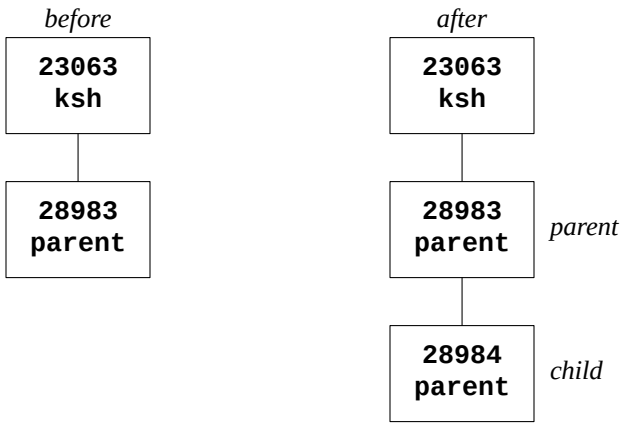
```
1 /* Excerpt from /usr/include/sys/types.h.
2 typedef int pid_t;
```

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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/types.h>   /* for pid_t */
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     pid_t pid;
 9
10     printf("My PID is %d and my parent's PID is %d.\n", getpid(), getppid());
11     fflush(NULL);
12
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) {
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/parent.pl

```
 1 #!/bin/perl
 2 use FileHandle;   #for autoflush
 3 STDOUT->autoflush(1);
 4
 5 $ppid = ‘ps -o ppid= -p $$‘;
 6 chomp $ppid;
 7 print "My PID is $$ and my parent’s PID is $ppid.\n";
 8
 9 $pid = fork();
10 defined $pid or die "$0: $!";
11
12 $ppid = ‘ps -o ppid= -p $$‘;
13 chomp $ppid;
14 print "My PID is $$ and my parent’s PID is $ppid.  fork returned $pid.\n";
15
16 exit 0;
```

Make the parent and child do different things

See David Curry’s O’Reilly book *Using C on the UNIX System* pp. 296−298.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9544/src/different1.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      } else {
17          printf("I am the parent.\n");
18      }
19
20      return EXIT_SUCCESS;
21  }
```

```
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.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/different1.pl

```
 1 #!/bin/perl
 2 use FileHandle;   #for autoflush
 3 STDOUT->autoflush(1);
 4
 5 $pid = fork();
 6 defined $pid or die "$0: $!";
 7
 8 if ($pid == 0) {
 9     print "I am the child.\n";
10 } else {
11     print "I am the parent.\n";
12 }
13
14 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.
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/different2.pl

```
 1 #!/bin/perl
 2 use FileHandle;   #for autoflush
 3 STDOUT->autoflush(1);
 4
 5 $pid = fork();
 6 defined $pid or die "$0: $!";
 7
 8 if ($pid == 0) {
 9     print "I am the child.\n";
10     exit 0;
11 }
12
13 print "I am the parent.\n";
14 exit 0;
```

exec without fork

See Bach pp. 217−227; David Curry's O'Reilly book *Using C on the UNIX System* 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.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/exec.pl

```
 1 #!/bin/perl
 2
 3 use FileHandle;
 4 STDOUT->autoflush(1);
 5 print "I am about to transform myself into the cal program.\n";
 6
 7 exec {'/bin/cal'} 'cal', '9', '1752';   #first arg has braces, not comma
 8 warn "$0: $!";
 9 exit 2;       #a number different from any that could be retunned by cal
```

▼ **Homework 7.4: 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; David Curry's O'Reilly book *Using C on the UNIX System* 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 7.5: 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 7.6: 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; David Curry's O'Reilly book *Using C on the UNIX System* 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?

▲

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; David Curry's O'Reilly book *Using C on the UNIX System* pp. 301−309; KP pp. 222−225.

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

```
1 #include <stdio.h>
```



```
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     pid_t pid = fork();
 9     int status;
10
11     if (pid < 0) {
12         perror(argv[0]);
13         return 1;
14     }
15
16     if (pid == 0) {
17         /* Arrive here if I am the child. */
18         execl("/usr/xpg4/bin/grep",
19             "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
20         perror(argv[0]);
21         return 3;       /* different from grep's exit status */
22     }
23
24     /* Arrive here if I am the parent. */
25     pid = wait(&status);
26     if (pid < 0) {
27         perror(argv[0]);
28         return 2;
29     }
30     printf("My child's PID number was %d.\n", pid);
31
32     if (WIFEXITED(status)) {
33         printf("My child's exit status was %d.\n", WEXITSTATUS(status));
34     }
35
36     else if (WIFSIGNALED(status)) {
37         printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
38     }
39
40     else if (WIFSTOPPED(status)) {
41         printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
42     }
43
44     else {
45         fprintf(stderr, "%s: couldn't find out how child ended up.\n", argv[0]);
46         return 3;
47     }
48
49     return EXIT_SUCCESS;
50 }
```

```
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 7.7: 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.

▲

fork-exec-wait in Perl

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

```
 1 #!/bin/perl
 2 use POSIX;
 3
 4 $pid = fork();
 5 defined $pid or die "$0: $!";
 6
 7 if ($pid == 0) {
 8     #Arrive here if I am the child.
 9     exec {'/usr/xpg4/bin/grep'} 'grep', '-q', '^abc1234:', '/etc/passwd';
10     warn "$0: $!";
11     exit 3;
12 }
13
14 #Arrive here if I am the parent.
15 $pid = wait();
16 if ($pid < 0) {
17     die "$0: $!";
18 }
19 print "My child's PID number was $pid.\n";
20
21 if (WIFEXITED($?)) {
22     print "My child's exit status was ", WEXITSTATUS($?), ".\n";
23 }
24
25 elsif (WIFSIGNALED($?)) {
26     print "My child was terminated by signal number ", WTERMSIG($?), ".\n";
27 }
28
29 elsif (WIFSTOPPED($?)) {
30     print "My child was stopped by signal number ", WSTOPSIG($?), ".\n";
31 }
32
33 else {
34     die "$0: couldn't find out how child ended up.";
35 }
36
37 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

```
38 $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 David Curry's O'Reilly book *Using C on the UNIX System* 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`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     pid_t pid1 = fork();
 9     pid_t pid2;
10     int status;
11
12     if (pid1 < 0) {
13         perror(argv[0]);
14         return 1;
15     }
16
17     if (pid1 == 0) {
18         /* Arrive here if I am the first child. */
19         execl("/usr/xpg4/bin/grep",
20             "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
21         perror(argv[0]);
22         return 3;   /* different from grep's exit status */
23     }
24
25     /* Arrive here if I am the parent. */
26     pid2 = fork();
27     if (pid2 < 0) {
28         perror(argv[0]);
29         return 2;
30     }
31
32     if (pid2 == 0) {
33         /* Arrive here if I am the second child.  mailx -e returns
34         exit status 0 if there is mail waiting for you, 1 otherwise. */
35         execl("/bin/mailx", "mailx", "-e", (char *)0);
36         perror(argv[0]);
37         return 2;   /* different from mail's exit status */
38     }
```

```
39
40      /* Arrive here if I am the parent.  I don't necessarily have to wait
41      for my children in the order in which they were born. */
42
43      waitpid(pid2, &status, 0);
44      if (WIFEXITED(status)) {
45          if (WEXITSTATUS(status) == 0) {
46              printf("There is mail waiting for you.\n");
47          } else if (WEXITSTATUS(status) == 1) {
48              printf("There is no mail waiting for you.\n");
49          } else {
50              printf("My second child couldn't turn into the mailx program.\n");
51          }
52      } else if (WIFSIGNALED(status)) {
53          printf("My second child (mailx) was terminated by signal number %d.\n",
54              WTERMSIG(status));
55      } else if (WIFSTOPPED(status)) {
56          printf("My second child (mailx) was stopped by signal number %d.\n",
57              WSTOPSIG(status));
58      }
59
60      waitpid(pid1, &status, 0);
61      if (WIFEXITED(status)) {
62          if (WEXITSTATUS(status) == 0) {
63              printf("abc1234 has an account.\n");
64          } else if (WEXITSTATUS(status) == 1) {
65              printf("abc1234 has no account.\n");
66          } else if (WEXITSTATUS(status) == 2) {
67              printf("My first child (grep) couldn't search /etc/passwd.\n");
68          } else {
69              printf("My first child couldn't turn into the grep program.\n");
70          }
71      } else if (WIFSIGNALED(status)) {
72          printf("My first child (grep) was terminated by signal number %d.\n",
73              WTERMSIG(status));
74      } else if (WIFSTOPPED(status)) {
75          printf("My first child (grep) was stopped by signal number %d.\n",
76              WSTOPSIG(status));
77      }
78
79      return EXIT_SUCCESS;
80  }
```

```
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 David Curry's O'Reilly book *Using C on the UNIX System* 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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     pid_t pid = fork();
 9     int status;
10
11     if (pid < 0) {
12         perror(argv[0]);
13         return EXIT_FAILURE;
14     }
15
16     if (pid == 0) {
17         /* Arrive here if I am the child. */
18         execl("/usr/xpg4/bin/grep",
19             "grep", "-q", "^abc1234:", "/etc/passwd", (char *)0);
20         perror(argv[0]);
21         return 3;   /* different from grep's exit status */
22     }
23
24     /* Arrive here if I am the parent. */
25
26     if (waitpid(pid, &status, WNOHANG) == 0) {
27         printf("The child isn't ready for harvesting yet, which is\n");
28         printf("okay because I have plenty of other work to do.\n");
29     } else if (WIFEXITED(status)) {
30         if (WEXITSTATUS(status) == 0) {
31             printf("abc1234 has an account.\n");
32         } else if (WEXITSTATUS(status) == 1) {
33             printf("abc1234 has no account.\n");
34         } else {
35             printf("My child (grep) couldn't search /etc/passwd.\n");
36         }
37     } else if (WIFSIGNALED(status)) {
38         printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
39     } else if (WIFSTOPPED(status)) {
40         printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
41     }
42
43     pid = waitpid(-1, &status, WNOHANG);
44     if (pid == 0) {
45         printf("I have no children which are ready to be harvested right now.\n");
46     } else {
47         printf("My child's PID number was %d and his exit status was %d.\n",
48             pid, WEXITSTATUS(status));
49     }
50
51     pid = wait(&status);
52     if (pid < 0) {
```

```
53          printf("I have no other children at all.\n");
54      } else {
55          printf("My child's PID number was %d and his exit status was %d.\n",
56              pid, WEXITSTATUS(status));
57      }
58
59      return EXIT_SUCCESS;
60  }
```

```
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.
```

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

```
 1 #!/bin/perl
 2 use POSIX;
 3
 4 $pid = fork();
 5 defined $pid or die "$0: $!";
 6
 7 if ($pid == 0) {
 8     #Arrive here if I am the child.
 9     exec {'/usr/xpg4/bin/grep'} 'grep', '-q', '^abc1234:', '/etc/passwd';
10     warn "$0: $!";
11     exit 3;
12 }
13
14 #Arrive here if I am the parent.
15 if (waitpid($pid, WNOHANG)) {
16     if (WIFEXITED($?)) {
17         print "The child's exit status is ", WEXITSTATUS($?), "\n";
18     }
19     exit 0;
20 }
21
22 print "The child isn't ready for harvesting yet.\n";
23 wait();
24 if (WIFEXITED($?)) {
25     print "The child's exit status is ", WEXITSTATUS($?), "\n";
26 }
27
28 exit 0;
```

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

See a zombie.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/zombie.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>     /* for perror, EXIT_SUCCESS, EXIT_FAILURE, system */
 3 #include <unistd.h>     /* for fork, getpid, getppid */
 4 #include <sys/wait.h>   /* for WIFEXITED and WEXITSTATUS */
 5
 6 int main(int argc, char **argv)
 7 {
 8     char buffer[1000];
 9     int status;
10     int s;
11     pid_t p;
12
13     const pid_t pid = fork();
14     if (pid < 0) {
15         perror(argv[0]);
16         return 1;
17     }
18
19     if (pid == 0) {
20         /* Arrive here if I am the child. */
21         return EXIT_SUCCESS;   /* Turn into a zombie immediately. */
22     }
23
24     /* Arrive here if I am the parent. */
25     sleep(1);   /* Give the child time to turn into a zombie. */
26
27     printf("My PID is %d and my child's PID is %d.\n", getpid(), pid);
28     /* minus lowercase L for "long" */
29     sprintf(buffer, "ps -l -p %d -p %d", getpid(), pid);
30     system(buffer);
31
32     p = wait(&status);   /* Harvest the zombie. */
33     if (p != pid) {
34         fprintf(stderr, "%s: my child is %d, not %d.\n", argv[0], pid, p);
35         return 2;
36     }
37
38     if (!WIFEXITED(status)) {
39         fprintf(stderr, "%s: parent didn't receive exit status from child.\n",
40             argv[0]);
41         return 3;
42     }
43
44     s = WEXITSTATUS(status);
45     if (s != EXIT_SUCCESS) {
46         fprintf(stderr,
47             "%s: parent received exit status %d instead of %d from child.\n",
48             argv[0], s, EXIT_SUCCESS);
49         return 4;
50     }
51
```



```
52     return EXIT_SUCCESS;
53 }
```

```
   F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
   8 S 50766 25429 25428  0  47 20        ?    122        ? pts/6    0:00 24741.ex
   8 Z 50766 25430 25429  0   0                                     0:00 <defunct>
   My PID is 25429 and my child's PID is 25430.
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/zombie.pl

```
 1 #!/bin/perl
 2 use POSIX;
 3
 4 $pid = fork();
 5 defined $pid or die "$0: $!";
 6
 7 if ($pid == 0) {
 8     #Arrive here if I am the child.
 9     exit 0;   #Turn into a zombie immediately.
10 }
11
12 #Arrive here if I am the parent.
13 sleep 1;   #Give the child time to turn into a zombie.
14
15 print "My PID is $$ and my child's PID is $pid.\n",
16     `ps -l -p $$ -p $pid`;   #minus lowercase L for "long"
17
18 $p = wait();   #Harvest the zombie.
19 if ($p != $pid) {
20     warn "$0: my child is $pid, not $p.";
21     exit 1;
22 }
23
24 if (!WIFEXITED($?)) {
25     warn "$0: parent didn't receive exit status from child.";
26     exit 2;
27 }
28
29 $status = WEXITSTATUS($?);
30 if ($status != 0) {
31     warn "$0: parent received exit status $status instead of 0 from child.";
32     exit 3;
33 }
34
35 exit 0;
```

The **S** column is the status of each process. The `back quotes` in line 16 make the parent give birth to a second child (**ps**), and make the parent sleep until this second child is dead. That's why the parent's status is **S** (sleeping):

```
1$ zombie.pl
My PID is 25466 and my child's PID is 25467.
F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
8 S 50766 25466 25465  0  57 20        ?    496        ? pts/6    0:00 zombie.p
8 Z 50766 25467 25466  0   0                                     0:00 <defunct>
```

Server and client

In a conversation between two programs, the one that sends the first packet is the *client;* the other one is the *server.*

The server starts running first, and waits for a client to talk to it. In fact, the server probably runs 24 hours per day, and continues to run after the conversation is over. Moreover, a server probably talks to many clients simultaneously.

Either the server or the client can terminate the conversation. In the following examples, the **daytime** server at port 13 and the **finger** server at port 79 terminate the conversation. On the other hand, the client who talks to the **echo** server at port 7 decides when it's time to terminate.

/etc/services

See pp. 47−48; **services**(4); and

http://www.iana.org/assignments/port-numbers

sed outputs a copy of the **/etc/services** file with the comments stripped away. Inside the argument of **awk**, a regular expression must be surrounded by slashes. If the regular expression contains a slash, it must therefore be preceded by a backslash.

The following caret ^ means ''start of the second field'', not ''start of the entire line''.

```
1$ sed 's/#.*//' /etc/services | awk '$2 ~ /^(7|13|21|23|53|79|520)\//' | more
echo    7/tcp
echo    7/udp
daytime 13/tcp
daytime 13/udp
ftp     21/tcp
telnet  23/tcp
domain  53/udp
domain  53/tcp
finger  79/tcp
route   520/udp router routed
```

What telnet really does

telnet is a client that lets you say whatever you want to any server that speaks TCP. When your **telnet** terminates the conversation, it says **Connection closed**. When the server terminates the conversation, your **telnet** says (in iambic tetrameter) **Connection closed by foreign host**.

Try **labinfu.unipv.it** (Università degli Studi di Pavia; 193.204.35.58).

```
1$ telnet labinfu.unipv.it 7
Trying 193.204.35.58...
Connected to labinfu.unipv.it.
Escape character is '^]'.
"And this also," said Marlow suddenly,
"And this also," said Marlow suddenly,
"has been one of the dark places of the earth."
"has been one of the dark places of the earth."
control-]
telnet> help
telnet> quit
Connection closed.
2$
```

The following server (**daytime**) requires no input from the client. When the server terminates your conversation, **telnet** says **Connection closed by foreign host**:

```
3$ telnet www.urz.uni-heidelberg.de 13
Trying 129.206.218.89...
Connected to www.urz.uni-heidelberg.de.
Escape character is '^]'.
Thu Dec 22 14:51:15 2005
Connection closed by foreign host.
4$

5$ date
Thu Dec 22 08:51:15 EST 2005
```

The following server requires a newline, optionally preceded by a loginname:

```
6$ telnet aixmita1.urz.uni-heidelberg.de 79
Trying 129.206.218.160...
Connected to aixmita1.urz.uni-heidelberg.de.
Escape character is '^]'.
RETURN
  User     Real Name          What    Idle  TTY  Host      Console Location
a86      Horst Koeppel      bash      1:01    0 aixterm6   (euler.pci.uni-he)
bgarbrec Bjoern Garbrecht   ssh2      0:02    3 aixterm1   (aixserv0.urz.uni)
by6      Nora Haerdle       bash     1 day    2 aixterm7   (129.206.119.60)
etc.
Connection closed by foreign host.
7$

8$ telnet aixmita1.urz.uni-heidelberg.de 79
Trying 129.206.218.160...
Connected to aixmita1.urz.uni-heidelberg.de.
Escape character is '^]'.
a86RETURN
  User     Real Name          What    Idle  TTY  Host      Console Location
a86      Horst Koeppel      bash      1:01    0 aixterm6   (euler.pci.uni-he)
Connection closed by foreign host.
9$
```

Sockets

Each computer that is connected to the Internet is called a *host.* A *socket* is like a pipe, except that (1) the same socket can be used for both input and output, and (2) the two processes can run on different hosts. See David Curry's O'Reilly book *Using C on the UNIX System* p. 391. The process that runs on the host where you're logged in is called the *local* process; the one on the other host is the *remote* process.

The *address family* of a socket tells how far apart the two processes are allowed to be. A socket whose address family is **AF_UNIX** (David Curry's O'Reilly book *Using C on the UNIX System* p. 367) can be used only for communication between two processes that are running on the same Unix machine. An **AF_UNIX** socket is given a name and is stored in a directory, so you can see it with **ls -l**. It's similar to a named pipe:

```
1$ cd /tmp
2$ ls -l | grep '^s' | more
srwxrwxrwx   1 mysql    other          0 Sep 26 18:18 mysql.sock
```

A socket whose address family is **AF_INET** (David Curry's O'Reilly book *Using C on the UNIX System* p. 399) can be used for communication between two processes that are running on different hosts. An **AF_INET** socket is given a *port number* instead of a name and directory.

Communicate via a socket

See Bach pp. 383−388; David Curry's O'Reilly book *Using C on the UNIX System* pp. 398−399, 401−404, 405−407. Every host on the Internet has an *IP address;* David Curry's O'Reilly book *Using C on the UNIX System* calls it an *host address* on pp. 393−394. For example, the IP address of the host **www.unipv.it** (Università degli Studi di Pavia) is

```
1$ /usr/sbin/nslookup www.unipv.it
Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

Name:    www.unipv.it
Address:  193.204.35.36
```

written with three dots separating the four octets.

There is a server named **echo** at port 7 on the host **www.unipv.it** that is waiting for you to send it data. It will send a copy of the data back to you.

```
 1 /* Excerpts from /usr/include/sys/socket.h. */
 2
 3 /* Address family: the first argument of socket, and the sin_family field
 4 of struct sockaddr_in. */
 5 #define AF_UNIX 1             /* two processes on the same Unix machine */
 6 #define AF_INET 2             /* two processes on different hosts, IPv4 */
 7 #define AF_INET6 26           /* two process on different hosts, IPv6 */
 8
 9 /* Socket type: the second argument of the socket function. */
10 #define SOCK_STREAM 1         /* stream socket (TCP) */
11 #define SOCK_DGRAM 2          /* datagram socket (UDP) */
12
13 /* The second argument of the shutdown function. */
14 #define SHUT_RD   0           /* Disables further receive operations */
15 #define SHUT_WR   1           /* Disables further send operations */
16 #define SHUT_RDWR 2           /* Disables further send and receive operations
17
18 /* Maximum queue length specifiable by listen. */
```



```
19 #define SOMAXCONN 1024

20 /* Excerpts from /usr/include/sys/types.h. */
21 typedef unsigned short u_short;
22 typedef unsigned int u_int;

23 /* Excerpts from /usr/include/netinet/in.h. */
24
25 /* Only the superuser can bind to a port whose number is smaller than this.
26 David Curry's O'Reilly book Using C on the UNIX System pp. 412-413. */
27 #define IPPORT_RESERVED 1024
28
29 /* Ports > IPPORT_USERRESERVED are reserved for servers, not necessarily
30 run by the superuser. */
31 #define IPPORT_USERRESERVED 5000
32
33 /* Tell server to accept a connection on any interface, David Curry's O'Reilly book Usin
34 #define INADDR_ANY 0x00000000
35
36 typedef uint32_t in_addr_t;
37
38 struct in_addr {
39     union {
40         struct { uint8_t s_b1, s_b2, s_b3, s_b4; } _S_un_b;
41         struct { uint16_t s_w1, s_w2; } _S_un_w;
42         in_addr_t _S_addr;
43     } _S_un;
44 };
45
46 #define s_addr  _S_un._S_addr
47
48 struct sockaddr_in {          /* Internet socket address */
49     u_short sin_family;       /* address family: AF_UNIX or AF_INET */
50     u_short sin_port;         /* port number from /etc/services */
51     struct in_addr sin_addr;  /* 32-bit IP address from nslookup */
52     char sin_zero[8];         /* padding */
53 };
```

I discovered which header file contained the definition of **struct sockaddr_in** by running just the C preprocessor. Then I used **vi** to search downwards for the first occurrence of **sockaddr_in**. Then I searched upwards from there for the first occurrence of **/usr/include**.

```
2$ gcc -E socket.c > socket.E
3$ vi socket.E
/sockaddr_in                    Slash searches downwards.
?/usr/include                   Question mark searches upwards.
```

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

```
1 /* Write one character to the program at port 7 (echo) of labinfu.unipv.it
2 Then read one character back. */
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
```



```
 7 #include <sys/socket.h>
 8
 9 int main(int argc, char **argv)
10 {
11     const int s = socket(AF_INET, SOCK_STREAM, 0); /* file descriptor */
12     struct sockaddr_in echo;
13     char buffer[INET6_ADDRSTRLEN];          /* for inet_ntop */
14     int i;                                  /* return value of inet_pton */
15     char c = 'A';
16
17     if (s < 0) {
18         perror(argv[0]);
19         return 1;
20     }
21
22     bzero((char *)&echo, sizeof echo);   /* fill the echo structure with zeroes */
23     echo.sin_family = AF_INET;           /* Internet address family: IPv4 */
24     echo.sin_port = htons(7);            /* echo is at port number 7 */
25
26     /* IP address of labinfu.unipv.it: */
27     i = inet_pton(AF_INET, "193.204.35.58", &echo.sin_addr);
28     if (i == 0) {
29         fprintf(stderr, "%s: bad dotted string to IP address\n", argv[0]);
30         return 2;
31     }
32
33     if (i != 1) {
34         perror(argv[0]);   /* bad address family */
35         return 3;
36     }
37
38     if (inet_ntop(AF_INET, &echo.sin_addr, buffer, sizeof buffer) == NULL) {
39         perror(argv[0]);
40         return 4;
41     }
42     printf("Trying %s...\n", buffer);
43
44     if (connect(s, (struct sockaddr *)&echo, sizeof echo) != 0) {
45         perror(argv[0]);
46         return 5;
47     }
48
49     printf("Connected to labinfu.unipv.it.\n");
50
51     if (write(s, &c, 1) != 1) {
52         perror(argv[0]);
53         return 6;
54     }
55
56     c = '\0';   /* Prove that the same character comes back from the server. */
57
58     if (read(s, &c, 1) != 1) {
59         perror(argv[0]);
60         return 7;
```



```
61      }
62
63      printf("%c\n", c);
64
65      if (shutdown(s, SHUT_RDWR) != 0) {    /* Curry pp. 128-409 */
66          perror(argv[0]);
67          return 8;
68      }
69
70      printf("Connection closed.\n");
71      return EXIT_SUCCESS;
72  }
```

```
4$ gcc -o ~/bin/socket socket.c -lsocket -lnsl "Networking Services Library"
5$ ls -l ~/bin/socket
6$ socket
Trying 193.204.35.58...
Connected to labinfu.unipv.it.
A
Connection closed.

7$ echo $?
0
```

Network byte order

See David Curry's O'Reilly book *Using C on the UNIX System* pp. 397−398; **htons**(3), **htonl**(3). Use use *type punning* to print the contents of **s** byte-by-byte:

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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/types.h>
 4 #include <netinet/in.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     unsigned short s = 0x1234;     /* decimal 4660, binary 00010010 00110100 */
 9     unsigned char *p = (char *)&s; /* Let the value of p be the address of s. */
10
11     printf("The value of s is hexadecimal %04X.\n", s);
12     printf("The address of s is %p.\n", p);
13     printf("The number of bytes in s is %d.\n\n", sizeof s);
14
15     printf("The contents of address %p is hexadecimal %02X.\n",   p,     p[0]);
16     printf("The contents of address %p is hexadecimal %02X.\n\n", p + 1, p[1]);
17
18     printf("htons returns %04X.\n", htons(s));    /* "host to network short" */
19     return EXIT_SUCCESS;
20 }
```

You could split line 9 to

```
 9     char *p;
10     p = (char *)&s;      /* Let the value of p be the address of s. */
```

—but why would you want to?

```
1$ gcc -o ~/bin/byteorder byteorder.c
2$ ls -l ~/bin/byteorder
3$ byteorder
The value of s is hexadecimal 1234.
The address of s is ffbff926.
The number of bytes in s is 2.

The contents of address ffbff926 is hexadecimal 12.
The contents of address ffbff927 is hexadecimal 34.

htons returns 1234.
```

	ffbff925	ffbff926	ffbff927	ffbff928	
		12 00010010	34 00110100		

▼ Homework 7.8: find the byte order on your machine at work

What is the **sizeof** and byte order on your machine of these three variables:

```
 1     short s = 0x1234;
 2     int i;
 3     long l = 0x12345678;
 4
 5     if (sizeof(int) == 2) {
 6         i = 0x1234;
 7     } else if (sizeof(int) == 4) {
 8         i = 0x12345678;
 9     } else {
10         fprintf(stderr, "sizeof(int) == %d\n", sizeof(int));
11     }
```

▲

socket in Perl: pp. 217, 348−355, 498−500 in O'Reilly Perl book

We saw **inet_aton**, in X52.9547 Handout 1, p. 16; **sockaddr_in** and **connect** in X52.9547 Handout 7, p. 1.

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

```
 1 #!/bin/perl
 2 #Output one line to the echo server at port 7 of labinfu.unipv.it
 3 #(193.204.35.58).  Then input the line back from the server.
 4
 5 use Socket;       #for AF_INET, etc.
 6 use FileHandle;   #for autoflush
 7
 8 socket(S, AF_INET, SOCK_STREAM, 0) or die "$0: $!";
 9 S->autoflush();      #pp. 130, 444 in O'Reilly Perl book
10
11 $ip = inet_aton('193.204.35.58')   or die "$0: inet_aton failed";
12 $address = sockaddr_in(7, $ip)     or die "$0: sockaddr_in failed";
13
```

```
14 print "Trying 193.204.35.58...\n";
15 connect(S, $address)              or die "$0: $!";
16 print "Connected to labinfu.unipv.it;\n";
17
18 print S "Hello, there.\n";   #Write one line to the socket.
19 $line = <S>;                 #Read one line from the socket.
20 print $line;
21
22 shutdown(S, SHUT_RDWR)            or die "$0: $!";
23 print "Connection closed.\n";
24 exit 0;
```

```
1$ chmod 755 socket.pl
2$ mv socket.pl ~/bin
3$ ls -l ~/bin/socket.pl
4$ socket
Trying 193.204.35.58...
Connected to labinfu.unipv.it;
Hello, there.
Connection closed.
```

Make the perlscript executable: **rwxr-xr-x**
if the perlscript is not already in **~/bin**

socket in Java:

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

```
 1 import java.io.*;
 2 import java.net.*;
 3
 4 class Echo {
 5     static public void main(String[] argv) {
 6         try {
 7             System.out.println("Trying 193.204.35.58...");
 8             Socket echo = new Socket("193.204.35.58", 7);
 9             System.out.println("Connected to labinfu.unipv.it.");
10
11             OutputStream out = echo.getOutputStream();
12             InputStream in = echo.getInputStream();
13
14             byte[] buffer = new byte[100];
15             buffer[0] = 'A';
16             out.write(buffer, 0, 1);
17
18             buffer[0] = '\0';
19             int len = in.read(buffer);
20             if (len > 0) {
21                 System.out.write(buffer, 0, len);
22             }
23             System.out.print('\n');
24
25             echo.close();
26         }
27
28         catch (UnknownHostException e) {
29             e.printStackTrace(System.err);
```

```
30          }
31
32          catch (IOException e) {
33              e.printStackTrace(System.err);
34          }
35
36          System.out.println("Connection closed.");
37          System.exit(0);
38      }
39  }
```

```
1$ javac Echo.java                  Run the java compiler.
2$ ls -l Echo.class
3$ java Echo                        Run the java interpreter (a.k.a the Java Virtual Machine).
Trying 193.204.35.58...
Connected to labinfu.unipv.it.
A
Connection closed.
```

□

