Fall 2004 Handout 4

Which ports on i5 already have a server bound to them?

```
1$ netstat -an -f inet -P tcp | head

TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.*                  *.*                0      0 24576      0 IDLE
      *.111                *.*                0      0 24576      0 LISTEN
      *.*                  *.*                0      0 24576      0 IDLE
      *.21                 *.*                0      0 24576      0 LISTEN
      *.515                *.*                0      0 24576      0 LISTEN
      *.24346              *.*                0      0 24576      0 LISTEN
```

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

```
#!/bin/ksh
#Output the ports that already have a server bound to them.

netstat -an -f inet -P tcp |
awk 'NR > 4 {print $1}' |
awk -F. '{print $NF}' |
sort -n |
uniq

exit 0
```

Here's the output of the above shellscript piped through **`pr -7 -l6 -s -t`** (minus lowercase L):

```
*       515     3306    8009    13782   28063   32786
21      939     4045    8080    13783   28064   51469
22      940     5987    11959   22370   32771   51471
25      941     8005    13722   24346   32774   51473
80      942     8008    13724   28062   32775   51474
111     2410
```

Bind a server to a port: Curry pp. 400−401, 404−405

The **`echo`**, **`daytime`**, and **`finger`** programs on **`acf.nyu.edu`** are called *servers*: they wait for other programs called *clients* to ask for something, and then respond to the request. The program you write for the previous homework was a client. Now you will write a server.

Each server listens to requests that arrive at a given port number. For example, the **`echo`** server on **`www.uu.nl`** is *bound* to port number 7.

When run on **`i5.nyu.edu`**, the following server binds itself to port number 10566 on **`i5.nyu.edu`**. Use **`INADDR_ANY`** instead of **`128.122.253.142`** to bind a server to a port on the host on which it is running; see Curry p. 400 and **`tcp`**(7). I picked the big port number 10566 to lessen the chance that some other server was using the same port number on **`i5.nyu.edu`**. Actually, 0566 are the

last four digits of my social security number:

```
1     long ss;                /* Print a long with percent lowercase LD. */
2     printf ("The last four digits are %04ld.\n", ss % 10000);

      The last four digits are 0566.
```

Only the superuser is allowed bind a server to a port number less than **IPPORT_RESERVED** (Curry pp. 412−413).

If several clients on several hosts try to **connect** to your server simultaneously, they will have to wait in line. The second argument of **listen** (Curry pp. 369−370) gives the maximum length of this queue. If more than this number of clients attempt to **connect**, the **connect** function in the unlucky clients will return −1. **listen** will deposit the number **ECONNREFUSED** or **ETIMEDOUT** (**#define**'d in the file **/usr/include/errno.h**) into the variable **errno** of the unlucky clients, causing **perror** to print **Connection refused** or **Connection timed out**.

The **accept** system call (Curry p. 370) will make the server go to sleep until a client tries to **connect** to it. Use the return value of **accept** as the file descriptor for all subsequent communication with the client. It can be used as the first argument of either **read** (line 56) or **write** (line 64) For convenience, the server should call **fdopen "r+"** so that it could **fprintf** and **fscanf** instead of **write** and **read**.

At line 69, after finishing its dialog with the client, the server could use the original file descriptor (**s**) as the first argument of another call to **accept**. This would let the server sleep until another client **connect**'s to it. To keep the server simple, we choose not to do this. This server will talk to only the first client that **connect**'s to it, and will then **exit**.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/myserver.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <ctype.h>   /* for tolower */
 4
 5 #include <sys/types.h>
 6 #include <sys/socket.h>
 7 #include <netinet/in.h>
 8
 9 int main(int argc, char **argv)
10 {
11     const u_short port = 10566;            /* sin_port field of struct sockaddr_in */
12     int s = socket(AF_INET, SOCK_STREAM, 0); /* file descriptor to accept a client */
13     struct sockaddr_in myaddress;
14     struct sockaddr_in clientaddress;
15     size_t length = sizeof clientaddress;
16     int client;   /* file descriptor to talk to client */
17     char dotted[INET6_ADDRSTRLEN];          /* for inet_ntop */
18     char buffer[2];
19
20     if (s < 0) {
21         perror(argv[0]);
22         return 1;
23     }
24
25     bzero((char *)&myaddress, sizeof myaddress);
26     myaddress.sin_family = AF_INET;
27     myaddress.sin_port = htons(port);
28     myaddress.sin_addr.s_addr = INADDR_ANY;
29
```



```
30      if (bind(s, (struct sockaddr *)&myaddress, sizeof myaddress) != 0) {
31          perror(argv[0]);
32          return 2;
33      }
34
35      if (listen(s, SOMAXCONN) != 0) {
36          perror(argv[0]);
37          return 3;
38      }
39
40      client = accept(s, (struct sockaddr *)&clientaddress, &length);
41      if (client < 0) {
42          perror(argv[0]);
43          return 4;
44      }
45
46      if (inet_ntop(AF_INET, &clientaddress.sin_addr, dotted, sizeof dotted)
47          == NULL) {
48          perror(argv[0]);
49          return 5;
50      }
51
52      fprintf(stderr, "I have accepted a client whose IP address is %s.\n", dotted);
53
54      /* The service provided by this server is merely to input one character
55      and then output it in lowercase, followed by a newline. */
56      if (read(client, buffer, 1) != 1) {
57          perror(argv[0]);
58          return 6;
59      }
60
61      buffer[0] = tolower(buffer[0]);
62      buffer[1] = '\n';
63
64      if (write(client, buffer, sizeof buffer) != sizeof buffer) {
65          perror(argv[0]);
66          return 7;
67      }
68
69      if (shutdown(client, SHUT_RDWR) != 0) {
70          perror(argv[0]);
71          return 8;
72      }
73
74      return EXIT_SUCCESS;
75  }
```

Run the above program like this on its initial voyage:

```
1$ gcc -o ~/bin/myserver myserver.c -lsocket -lnsl
2$ ls -l ~/bin/myserver
3$ myserver                                    Nothing is supposed to happen.
```

If **myserver** blows up, say


```
4$ echo $?
```

immediately afterwards to see its exit status. If **myserver** doesn't blow up during the first 30 seconds, kill it with **control-c** and run it in the background with **nohup** (KP p. 33, 35) so that it will not die when you log out:

```
5$ nohup myserver &
6$ ps -f | more

7$ netstat -an -f inet -P tcp | awk '2 <= NR && NR <= 4 || $1 ~ /.10566$/'
TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.10566              *.*                0      0 24576      0 LISTEN
```

Then on any Internet host (including **i5.nyu.edu** itself) you can run **mytelnet**, or simply

```
$ telnet i5.nyu.edu 10566
A                                   You type this line.
a                                   It types this line and telnet types the next.
Connection closed by foreign host.
```

myserver return's from **main** when it is done with its first and only client. If you want to kill the server before it has talked to a client, use **ps** and **kill -9**.

If you try to **rm** a program while it is running, you may get the **text file busy** error message. For example, you will get this message if you try to recompile a C program while it is running. Simply **kill -9** the process and then try to **rm** again. For a minute or two after the server terminates, the socket will remain in the **TIME_WAIT** state:

```
8$ netstat -an -f inet -P tcp | grep 10566
i5.nyu.edu.10566     i5.nyu.edu.43734     32768      0 32768      0 TIME_WAIT
```

If you try to run the server again while the socket is still in this state, the **bind** in line 31 will give you

```
Address already in use
```

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

```
#!/bin/perl
#The same server, in Perl.
use Socket;

use FileHandle;
STDOUT->autoflush();

socket(S, AF_INET, SOCK_STREAM, 0) or die "$0: $!";
bind(S, sockaddr_in(10566, INADDR_ANY)) or die "$0: $!";
state();

listen(S, SOMAXCONN) or die "$0: $!";
state();

$clientaddress = accept(CLIENT, S) or die "$0: $!";
CLIENT->autoflush();

($port, $ip) = sockaddr_in($clientaddress);
print "I have accepted a client whose IP address is ",
    inet_ntoa($ip), ".\n";
state();

$_ = <CLIENT>;
$_ =~ tr/A-Z/a-z/;   #Can remove the $_ =~
print CLIENT $_;     #Can remove the $_

shutdown(CLIENT, SHUT_RDWR) or die "$0: $!";
state();
exit 0;

sub state {
    print 'State of socket is ',
        'netstat -an -f inet -P tcp |\
        awk 'NR > 4 && \$1 ~ /\.10566\$/ {print \$NF}' |\
        tail -1';
}
```

```
9$ nohup myserver > myserver.log 2>&1 &

10$ telnet i5.nyu.edu 10566
Hello
hello
Connection closed by foreign host.

11$ cat myserver.log
State of socket is BOUND            line 10
State of socket is LISTEN           line 13
I have accepted a client whose IP address is 128.122.253.142.
State of socket is ESTABLISHED      line 21
State of socket is TIME_WAIT        line 28
```

Serve more than one client simultaneously: Bach pp. 386−387; Curry pp. 350−352

> No man can serve two masters.
>
> *—Matthew 6:24*

To serve two or more clients, your server must **`read`** from and **`write`** to one of them while also **`accept`**'ing another one. The easiest way to do both, simultaneously, is to have your server give birth immediately after each **`accept`**'ance of a new client. The child will devote itself entirely to its dialog with the new client, and will **`exit`** when it is done talking to the client. The parent, meanwhile, will have no further relations to do with the new client. It will do nothing but **`accept`** additional clients.

In the following program, all i/o is performed with the file descriptor **`client`**. The other file descriptor, **`s`**, is only for **`accept`**'ing new clients. If you want to use the standard i/o library, it therefore makes sense to **`fdopen`** only the **`client`** file descriptor, not **`s`**. And since only the child, not the parent, will perform the i/o, you should **`fdopen client`** in the child. The parent should do nothing with the **`client`** file descriptor except **`close`** it immediately after the **`fork`**.

—Source code on the Web at **`http://i5.nyu.edu/~mm64/x52.9544/src/myserver2.c`**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <ctype.h>
 4
 5 #include <sys/types.h>
 6 #include <sys/socket.h>
 7 #include <netinet/in.h>
 8 #include <sys/wait.h>
 9
10 int main(int argc, char **argv)
11 {
12     const u_short port = 10566;  /* sin_port field of struct sockaddr_in */
13     int s = socket(AF_INET, SOCK_STREAM, 0); /* file descriptor to accept a client */
14     struct sockaddr_in myaddress;
15     struct sockaddr_in clientaddress;
16     size_t length = sizeof clientaddress;
17     int client;  /* file descriptor to talk to client */
18     char dotted[INET6_ADDRSTRLEN];          /* for inet_ntop */
19     pid_t pid;
20     char c;
21     char buffer[2];
22     int status;
23
24     if (s < 0) {
25         perror(argv[0]);
26         return 1;
27     }
28
29     bzero((char *)&myaddress, sizeof myaddress);
30     myaddress.sin_family = AF_INET;
31     myaddress.sin_port = htons(port);
32     myaddress.sin_addr.s_addr = INADDR_ANY;
33
34     if (bind(s, (struct sockaddr *)&myaddress, sizeof myaddress) != 0) {
35         perror(argv[0]);
36         return 2;
37     }
38
```



```
39      if (listen(s, SOMAXCONN) != 0) {
40          perror(argv[0]);
41          return 3;
42      }
43
44      for (;;) {
45          client = accept(s, (struct sockaddr *)&clientaddress, &length);
46          if (client < 0) {
47              perror(argv[0]);
48              return 4;
49          }
50
51          if (inet_ntop(AF_INET, &clientaddress.sin_addr,
52              dotted, sizeof dotted) == NULL) {
53              perror(argv[0]);
54              return 5;
55          }
56
57          fprintf(stderr, "I have accepted a client whose IP address is %s.\n",
58              dotted);
59
60          pid = fork();
61          if (pid < 0) {
62              perror(argv[0]);
63              return 6;
64          }
65
66          if (pid == 0) {
67              /* Arrive here if I am the child.  The service provided
68              by this server is merely to input one character and then
69              output it in lowercase, followed by a newline. */
70
71              if (read(client, &c, 1) != 1) {
72                  perror(argv[0]);
73                  return 1;
74              }
75
76              buffer[0] = tolower(c);
77              buffer[1] = '\n';
78
79              if (write(client, buffer, sizeof buffer) != sizeof buffer) {
80                  perror(argv[0]);
81                  return 2;
82              }
83
84              if (shutdown(client, SHUT_RDWR) != 0) {
85                  perror(argv[0]);
86                  return 3;
87              }
88
89              return EXIT_SUCCESS;
90          }
91
92          /* Arrive here if I am the parent. */
```

```
 93          if (close(client) != 0) {
 94              perror(argv[0]);
 95              return 7;
 96          }
 97
 98          /* Harvest any children that are ripe (i.e., zombies). */
 99          while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
100              printf("child PID %d: ", pid);
101              if (WIFEXITED(status)) {
102                  printf("exit status was %d.\n", WEXITSTATUS(status));
103              } else if (WIFSIGNALED(status)) {
104                  printf("terminated by signal number %d.\n", WTERMSIG(status));
105              } else if (WIFSTOPPED(status)) {
106                  printf("stopped by signal number %d.\n", WSTOPSIG(status));
107              } else {
108                  fprintf(stderr, "%s: child %d came to unknown end.\n",
109                      argv[0], pid);
110                  return EXIT_FAILURE;
111              }
112          }
113      }
114
115      return EXIT_SUCCESS;
116  }
```

Run this server with **nohup** as shown above. Use **ps** and **kill -9** to kill it when you no longer want it to **accept** clients.

Instead of writing a server that gives birth to server-children, the superuser can have the **inetd** give birth to server-children. See **inetd**(8) and **inetd.conf**(4).

In Perl, run in *taint mode* with **-T** (O'Reilly Perl book, p. 356) to prevent outsiders from making mischief.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/myserver2**

```
 1 #!/bin/perl -T
 2 #The same server, in Perl.
 3 use Socket;
 4 use FileHandle;
 5 use POSIX;
 6
 7 socket(S, AF_INET, SOCK_STREAM, 0)      or die "$0: $!";
 8 bind(S, sockaddr_in(10566, INADDR_ANY)) or die "$0: $!";
 9 listen(S, SOMAXCONN)                    or die "$0: $!";
10
11 for (;;) {
12     $clientaddress = accept(CLIENT, S) or die "$0: $!";
13
14     ($port, $ip) = sockaddr_in($clientaddress);
15     print "I have accepted a client whose IP address is ",
16         inet_ntoa($ip), ".\n";
17
18     CLIENT->autoflush();
19
20     $pid = fork();
21     defined $pid or die "$0: $!";
```



```
22
23      if ($pid == 0) {
24          #Arrive here if I am the child.
25          $_ = <CLIENT>;
26          tr/A-Z/a-z/;
27          print CLIENT;
28          shutdown(CLIENT, SHUT_RDWR) or die "$0: $!";
29          exit 0;
30      }
31
32      #Arrive here if I am the parent.
33      close(CLIENT) or die "$0: $!";
34
35      while (waitpid(-1, WNOHANG)) {
36          if (WIFEXITED($?)) {
37              print "My child's exit status was ",
38                  WEXITSTATUS($?), "\n";
39          } elsif (WIFSIGNALED($?)) {
40              print "My child was terminated by signal number ",
41                  WTERMSIG($?), ".\n";
42          } elsif (WIFSTOPPED($?)) {
43              print "My child was stopped by signal number ",
44                  WSTOPSIG($?), ".\n";
45          } else {
46              print STDERR "$0: couldn't find out how child $pid ended up.\n";
47          }
48      }
49  }
50
51  exit 0;
```

▼ Homework 4.1: write a server that can serve two or more clients simultaneously

Do not attempt this assignment until we discuss it in class. Write a server named **`myserver`** that will simultaneously serve two or more clients running on any hosts as shown above. If **`myserver`** runs on the host **`i5.nyu.edu`**, it should bind itself to port number 10000 plus the last four digits of your social security number. **`nohup`** the server on the date when this assignment is due, and **`kill -9`** it one week later.

Hand in the **`myserver.c`**, the hostname of the host on which it is running (e.g., **`i5.nyu.edu`**), the host's IP address (e.g., **`128.122.253.142`**), and the port number (e.g., **`10566`**). Also put a note in your World Wide Web home page giving the hostname and IP address of the machine where server runs, the port number of server, Describe what it does and how to use it, i.e., what input, if any, the client must send through the socket.

`myserver` can provide each client any service you wish, but you must use **`fscanf`**, **`fgetc`**, or **`getc`** instead of **`read`**, and you must use **`fprintf`**, **`fputc`**, or **`putc`** instead of **`write`**. For example:

(1) **`myserver`** can merely output to each client a brief message such as **`You are client number n.`** In this case, the server performs output but no input.

(2) Write an interactive server (performing both input and output) providing the service shown in **`$S45/prog.c`** (on the web at **`http://i5.nyu.edu/~mm64/x52.9545/public_html/prog.c`**)

(3) If **`myserver`** is running on **`i5.nyu.edu`**, it can output to each client a picture of the moon. Have the child **`popen`** the program **`/home1/m/mm64/bin/moon`**, and write a **`while-getc`** loop to input bytes from **`moon`** and output them one-by-one through the socket to the client. This will make the

output of my **moon** program available to the whole Internet.

(4) Write an interactive server that will translate its input into Pig Latin.

Test your server with either **telnet** or **mytelnet**.

▲

Signals: Bach pp. 200−211; Curry pp. 239−282; KP pp. 150−152, 225−230; K&R pp. 200, 255

A *signal* is a small integer that can be fired at a process; To see all the signal numbers,

```
1$ man -s 3head signal
2$ kill -l | more                    minus lowercase L; ksh93(1) pp. 27−28
```

The file **/usr/include/sys/iso/signal_iso.h** defines a macro for each signal number:

```
3$ grep '^#define[    ][    ]*SIG[A-Za-z0-9_]' /usr/include/sys/iso/signal_iso.h |
    head -27
#define SIGHUP    1     /* hangup */
#define SIGINT    2     /* interrupt (rubout) */
#define SIGQUIT   3     /* quit (ASCII FS) */
#define SIGILL    4     /* illegal instruction (not reset when caught) */
#define SIGTRAP   5     /* trace trap (not reset when caught) */
#define SIGIOT    6     /* IOT instruction */
#define SIGABRT   6     /* used by abort, replace SIGIOT in the future */
#define SIGEMT    7     /* EMT instruction */
#define SIGFPE    8     /* floating point exception */
#define SIGKILL   9     /* kill (cannot be caught or ignored) */
#define SIGBUS    10    /* bus error */
#define SIGSEGV   11    /* segmentation violation */
#define SIGSYS    12    /* bad argument to system call */
#define SIGPIPE   13    /* write on a pipe with no one to read it */
#define SIGALRM   14    /* alarm clock */
#define SIGTERM   15    /* software termination signal from kill */
#define SIGUSR1   16    /* user defined signal 1 */
#define SIGUSR2   17    /* user defined signal 2 */
#define SIGCLD    18    /* child status change */
#define SIGCHLD   18    /* child status change alias (POSIX) */
#define SIGPWR    19    /* power-fail restart */
#define SIGWINCH  20    /* window size change */
#define SIGURG    21    /* urgent socket condition */
#define SIGPOLL   22    /* pollable event occured */
#define SIGSTOP   23    /* stop (cannot be caught or ignored) */
#define SIGTSTP   24    /* user stop requested from tty */
#define SIGCONT   25    /* stopped process has been continued */

4$ grep '^#define[    ][    ]*SIG_' /usr/include/sys/iso/signal_iso.h |
    head -20
#define SIG_DFL   ((void (*)(int)) 0)  /* restore the default action */
#define SIG_IGN   ((void (*)(int)) 1)  /* ignore the signal */
#define SIG_ERR   ((void (*)(int))-1)  /* bad return value of signal function */
```

Exceptions

An *exception* is a signal that the kernel automatically sends to any program that attempts an illegal act. (A C++ ''exception'' is something else entirely; see Bjarne Stroustrup, *The C++ Programming Language, Third Edition*.)

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/exception.c**

```
 1 #include <stdlib.h>
 2
 3 int main()
 4 {
 5     char *p = NULL;
 6
 7     for (;;) {      /* deliberately suicidal loop: fill all of memory with A's. */
 8         *p++ = 'A';
 9     }
10
11     return EXIT_SUCCESS;   /* Never reaches here. */
12 }
```

```
1$ gcc -o ~/bin/exception -g exception.c          minus lowercase g for gdb
2$ ls -l ~/bin/exception.o
3$ exception
Memory fault(coredump)

4$ echo $?                                  Must do this immediately after program's demise.
267

5$ bc
obase = 2                                   request binary output
267                                         You type this.
100001011                                   It types this.

obase = 16                                  You type this.
267                                         You type this.
10B                                         It types this: the lowest 7 bits of 10B is SIGSEGV.
control-d

6$ grep SIGSEGV /usr/include/sys/iso/signal_iso.h
#define  SIGSEGV 11  /* segmentation violation */

7$ ls -l
-rw-------   1 mm64     users      72160 Mar   8 10:34 core
8$ rm core                                  if you don't plan to examine the core file with dbx.
```

Run the debugger

If you compiled with the **-g** option of **gcc**, you can examine the **core** file dumped by your executable **prog** with

```
1$ gdb exception core
Program terminated with signal 11, Segmentation Fault.
#0  0x105c4 in main () at exception.c:8
8                       *p++ = 'A';
(gdb) print p
$1 = 0x0
(gdb) help
(gdb) run
(gdb) quit
2$
```


What happens if you try to change the value of a **static const** variable in C? What happens if you try to change the value of a non-**static const** variable in C? What happens if you let the user **scanf** the values of two **double** variables, **a** and **b**, and then try to **printf a/b** when **b** is zero? What happens when the two variables are **int**'s?

Interrupts

An *interrupt* is a signal that one process sends to another. For example, when you type **control-c**, **control-**, or **control-z** at the terminal, the shell sends the signal **#define**'d in **/usr/include/signal.h** to the process you are running.

The commands

```
1$ kill -9 12345                    12345 is the PID number of a hapless program.
2$ kill -KILL 12345
```

send signal number 9 (**SIGKILL**) to the process whose **PID** number is 12345. The name **kill** is a misnomer. You're not necessarily killing the process; you're merely sending it a signal. Most signals, however, are deadly or narcotic. See **kill**(1) for the **kill** command, **kill**(2) for the **kill** system call:

```
3$ whatis kill
kill    kill (1) - terminate or signal processes
kill    kill (2) - send a signal to a process or a group of processes
```

ksh88 has **$ERRNO**; **ksh93** doesn't.

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

```
#!/bin/ksh88
#Send the SIGKILL signal to the process whose PID number is 12345.
#See Kernighan & Pike p. 140 for $?; pp. 93, 141-2 for 1>&2.

kill -KILL 12345
if [[ $? -ne 0 ]]
then
    echo $0: couldn\'t kill PID 12345, errno == $ERRNO 1>&2
    exit 1
fi

exit 0
```

```
4$ sigkill
kill: 12345: no such process
sigkill: couldn't kill PID 12345, errno == 3

5$ grep ESRCH /usr/include/sys/errno.h
#define  ESRCH    3    /* No such process          */
```

The following shellscript needs the **{**curly braces**}** (KP pp. 168−169) because the precedence of **||** (KP pp. 143−144) is higher than that of **;**. If you had used parentheses instead of curly braces, the **echo** and **exit** would have been executed by another copy of the Korn shell. In that case, the **exit** would have had no effect: it would have merely terminated the second copy.

http://i5.nyu.edu/~mm64/x52.9544/src/sigkill2

```
#!/bin/ksh88
#Another way to do the same thing.

kill -KILL 12345 || {
    echo $0: couldn\'t kill 12345, errno $ERRNO 1>&2
    exit 1
}

exit 0
```

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/sigkill.c**

```
 1 /* Send the SIGKILL signal to the process whose PID number is 12345. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <signal.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     if (kill(12345, SIGKILL) != 0) {
 9         perror(argv[0]);
10         return EXIT_FAILURE;
11     }
12
13     return EXIT_SUCCESS;
14 }
```

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

```
#!/bin/perl
#Send the SIGKILL signal to the process whose PID number is 12345.

kill('SIGKILL', 12345) or die "$0: $!";
exit 0;
```

▼ Homework 4.2: send a signal to yourself

Two C functions that return the PID number of processes you might want to signal are **getpid**, which returns the PID number of the process itself (like **$$** in a shellscript or perlscript), and **getppid**, which returns the **PID** number of the parent of the process. In Perl, the variable **$$** is the PID of the process. See **perlvar**(1), p. 8.

The Bell Labs chess program *Belle* was programmed to dump core whenever it sensed that it was losing. Write a program that convincingly dumps core by sending itself the **SIGSEGV** signal with **kill**. Use **getpid** or **$$** to discover your own **PID** number. But first, **sleep** a few seconds and output some garbage and a random error message.

Or write a program that stops itself with a **SIGTSTP**, in effect **control-z**'ing itself. Then say

```
1$ jobs
```

or

```
2$ ps
```

to verify that the process is alive but **Stopped**. Then restart the process with

```
3$ fg
```

to let it run to completion.

▲

Ignore a signal: Curry pp. 248−250

By default, most signals kill the process that receives them. Among the few exceptions are the signals that merely stop process the process: **SIGSTOP** and **SIGTSTP**.

The following programs make themselves ignore the **SIGINT** signal for 10 seconds. During this time, you will not be able to kill them with a **control-c**. You will have to stop them with a **control-z** (which is really a **SIGTSTP**) and then say **kill -9** (which is really a **SIGKILL**).

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

```
#!/bin/ksh
#For trap, See KR p. 150-152, or ksh93(1) p. 31.

trap '' 2   #You can change 2 to INT.
i=1
while [[ $i -le 10 ]]
do
    echo I am ignoring the SIGINT signal.
    sleep 1
    let ++i
done

trap 2
while true
do
    echo I am once again vulnerable to the SIGINT signal.
    sleep 1
done
```

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/ignore.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <signal.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     int i;
 9
10     if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
11         perror(argv[0]);
12         return 1;
13     }
14
15     for (i = 1; i <= 10; ++i) {
16         printf("I am ignoring the SIGINT signal.\n");
17         sleep(1);
18     }
19
20     if (signal(SIGINT, SIG_DFL) == SIG_ERR) {
21         perror(argv[0]);
22         return 2;
23     }
24
```



```
25      for (;;) {
26          printf("I am once again vulnerable to the SIGINT signal.\n");
27          sleep(1);
28      }
29  }
```

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

```
#!/bin/perl
#%SIG is an associative array; see Handout 3, p. 12;
#Kernighan & Pike pp. 123-134.

$SIG{INT} = 'IGNORE';
for ($i = 1; $i <= 10; ++$i) {
    print "I am ignoring the SIGINT signal.\n";
    sleep 1;
}

$SIG{INT} = 'DEFAULT';
for (;;) {
    print "I am once again vulnerable to the SIGINT signal.\n";
    sleep 1;
}
```

▼ Homework 4.3: refuse to stop

Write a program that will ignore the **SIGTSTP** signal, so that it can't be stopped with **control-z**.

▲

Catch a signal and then resume what you were doing: Curry pp. 248−250

If you type **control-c** when runing **vi**, it merely beeps at you. If you type **control-c** when runing **gdb**, it ignores you. Both programs then resume what they were doing before.

If the first argument of **trap** contains more than one command, they must be separated by semicolons. See KP p. 151.

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

```
#!/bin/ksh

trap 'echo Pitiful human, your control-c cannot harm me.' 2

while true
do
    echo I am prepared to handle the SIGINT signal.
    sleep 1
done
```

In C, the same signal handler can handle two different signals. The first argument of the handler tells it which signal it received.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/resume.c**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <unistd.h>
5
```



```
 6 void handle_sigint(int sig); /* function declaration */
 7
 8 int main(int argc, char **argv)
 9 {
10     if (signal(SIGINT, handle_sigint) == SIG_ERR) {
11         perror(argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     for (;;) {
16         printf("I am prepared to handle the SIGINT signal.\n");
17         sleep(1);
18     }
19
20     return EXIT_SUCCESS;
21 }
22
23 /* This function will be called whenever a SIGINT arrives. */
24
25 void handle_sigint(int sig)  /* function definition */
26 {
27     printf("Pitiful human, your control-c (signal number %d) cannot harm me.\n",
28         sig);
29 }
```

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

```
#!/bin/perl

$SIG{INT} = 'handle_sigint';

for (;;) {
    print "I am prepared to handle the SIGINT signal.\n";
    sleep(1);
}

#This subroutine will be called whenever a SIGINT arrives.

sub handle_sigint {
    local($sig) = @_;
    print "Pitiful human, your control-c (SIG$sig) cannot harm me.\n";
}
```

Catch a signal, set a variable, and resume what you were doing

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/set.c**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <curses.h>  /* for napms */
5
6 void handle_sigint(int sig);
7 void fascinator(void);
8 int fascinated = 1;  /* fascinator returns when this becomes 0 */
9
```

```
10 int main(int argc, char **argv)
11 {
12     if (signal(SIGINT, handle_sigint) == SIG_ERR) {
13         perror(argv[0]);
14         return EXIT_FAILURE;
15     }
16
17     printf("Press control-c when you're thoroughly hypnotized:  ");
18     fascinator();
19
20     printf("\n");
21     printf("Okay, now we can go on to something else.\n");
22     return EXIT_SUCCESS;
23 }
24
25 /* Return from this function when fascinated becomes 0. */
26
27 void fascinator(void)
28 {
29     int i;
30     static char a[] = {'|', '/', '-', '\\'};  /* clockwise propeller */
31     const size_t n = sizeof a / sizeof a[0];
32
33     for (i = 0; fascinated; ++i) {
34         printf("\b%c", a[i % n]);    /* backspace: \*(Kr p. 38 */
35         fflush(stdout);
36         napms(250);  /* sleep for milliseconds */
37     }
38 }
39
40 void handle_sigint(int sig)  /* called upon receipt of SIGINT */
41 {
42     fascinated = 0;
43 }
```

For **napms**, See Curry pp. 264−267.

```
1$ gcc -o ~/bin/set set.c -lcurses
2$ ls -l ~/bin/set
3$ ~/set                                  because set is a word in the shell language
```


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

```
#!/bin/perl

$| = 1;    #flush output buffer immediatey after each print
@a = ('|', '/', '-', '\\');
$n = @a;   #number of elements

$fascinated = 1;
$SIG{INT} = 'handle_sigint';
print "Press control-c when you're thoroughly hypnotized:  ";

for ($i = 0; $fascinated; ++$i) {
    print "\b$a[$i % $n]";
    sleep 1;
}

print "\n";
print "Okay, now we can go on to something else.\n";
exit 0;

sub handle_sigint {
    $fascinated = 0;
}
```

Catch a signal and longjmp back to the top of the program

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/backtotop.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <setjmp.h>
 4 #include <signal.h>
 5
 6 void handle_sigint(int sig);
 7 void fascinator(void);
 8 jmp_buf back_to_main;
 9
10 int main(int argc, char **argv)
11 {
12     int i = setjmp(back_to_main);
13     if (i == 1) {
14         printf("\n");
15         printf("Okay, now we can go on to something else.\n");
16         /* other stuff */
17         return EXIT_SUCCESS;
18     }
19
20     if (signal(SIGINT, handle_sigint) == SIG_ERR) {
21         perror(argv[0]);
22         return EXIT_FAILURE;
23     }
24
25     printf("Press control-c when you're thoroughly hypnotized:  ");
26     fascinator();
27 }
28
```



```
29 /* Only a signal could let you escape from this function. */
30
31 void fascinator(void)
32 {
33     int i;
34     static char a[] = {'|', '/', '-', '\\'};  /* clockwise propeller */
35     const size_t n = sizeof a / sizeof a[0];
36
37     for (i = 0;; ++i) {
38         printf("\b%c", a[i % n]);    /* backspace: \*(Kr p. 38 */
39         fflush(stdout);
40         napms(250);  /* sleep for milliseconds */
41     }
42 }
43
44 void handle_sigint(int sig)  /* called upon receipt of SIGINT */
45 {
46     longjmp(back_to_main, 1);
47 }
```

For **napms**, see Curry pp. 215−216.

```
1$ gcc -o ~/bin/backtotop backtotop.c -lcurses
2$ ls -l ~/bin/backtotop
3$ backtotop
```

Catch the death-of-child signal: Bach pp. 213−217

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9544/src/sigchld.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <signal.h>
 4 #include <setjmp.h>
 5 #include <sys/wait.h>
 6
 7 void handle_sigchld(int sig);
 8 jmp_buf alldone;
 9
10 int main(int argc, char **argv)
11 {
12     pid_t pid;
13     int status;
14
15     if (setjmp(alldone) == 1) {
16
17         /* Arrive here after the child has died. */
18         pid = wait(&status);
19         printf("My child's PID number was %d.\n", pid);
20         if (WIFEXITED(status)) {
21             printf("My child's exit status was %d.\n", WEXITSTATUS(status));
22         }
23         return EXIT_SUCCESS;
24     }
25
26     if (signal(SIGCHLD, handle_sigchld) == SIG_ERR) {
```



```
27          perror(argv[0]);
28          return 1;
29      }
30
31      pid = fork();
32      if (pid < 0) {
33          perror(argv[0]);
34          return 2;
35      }
36
37      if (pid == 0) {
38          /* Arrive here if I am the child. */
39          execl("/bin/grep", "grep", "-q", "Yorick",
40              "/home/m/mm64/public_html/x52.9545/src/Shakespeare.complete",
41              (char *)0);
42          perror(argv[0]);
43          return 3;       /* different from grep's exit status */
44      }
45
46      /* Arrive here if I am the parent. */
47      for (;;) {
48          printf("Parent does other work here.\n");
49          /* The other work does not necessarily have to be inside a loop. */
50          sleep(1);
51      }
52  }
53
54  void handle_sigchld(int sig)
55  {
56      longjmp(alldone, 1);
57  }
```

```
1$ prog
Parent does other work here.
Parent does other work here.
Parent does other work here.
My child's PID number was 5788.
My child's exit status was 0.
```

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

```
#!/bin/perl -w
use POSIX;   #for the W functions

$SIG{CHLD} = 'death_of_child';

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

if ($pid == 0) {
    #Arrive here if I am the child.
    sleep 3;
    exit 0;
}

#Arrive here if I am the parent.
#Copy the standard input to the standard output until the death of child.
while (($c = getc()) ne '') {
    print $c;
}

sub death_of_child {
    print "My child is dead.\n";
    exit 0;
}
```

▼ Homework 4.4: catch the death-of-child signal

The parent in Handout 15, pp. 11−13 does not break out of the **`while`** loop in lines 68−71 until the user types **`control-d`**. Make it break out of the loop as soon as the user types **`control-d`** or as soon as its child dies, whichever comes first.

Before the loop begins (in fact, even before giving birth), call **`signal`** to set up a handler named **`handle_sigchld`**. The function **`handle_sigchld`** should simply change the value of a variable from true to false. The parent will then **`shutdown`**, **`wait`**, and terminate itself. This is the ''future improvement'' in Handout 15, p. 13.

▲

Set the alarm: Bach pp. 260−261 Curry pp. 243, 258−261 KP pp. 229−230

Your process can immediately send itself whatever signal it wishes by calling

```
1     if (kill(getpid(), SIGWHATEVER) != 0) {
2         perror(argv[0]);
3         return EXIT_FAILURE;
4     }
```

Your process can also ask the kernel to send it a **`SIGALRM`** signal in 30 seconds. Be sure to call **`signal(SIGALRM`** before you call **`alarm`**:

```
5 #include <unistd.h>
6 void handle_sigalrm(int sig);    /* function declaration atop .c file */
7
8     /* Be ready to handle the SIGALRM when it arrives. */
9     if (signal(SIGALRM, handle_sigalrm) == SIG_ERR) {
10        perror(argv[0]);
11        return EXIT_FAILURE;
12    }
13
```



```
14      /* Ask the kernel to send me a SIGALRM in 30 seconds. */
15      alarm(30);
```

□

