§The Unix Family Tree
BIOS

bootloader

kernel

“handcrafted” process
"Daemons" e.g., sshd, httpd

- **kernel**
  - **init**
    - **fork & exec**
      - **getty**

- **fork & exec**
kernel

init

shell (e.g. sh)

exec
kernel

init

shell (e.g. sh)

(a fork-ing party!)

user process

user process

user process

user process
(or, for the GUI-inclined)

```
kernel

init

display manager (e.g., xdm)

X Server (e.g., XFree86)

window manager (e.g., twm)
```
window manager (e.g. twm)

terminal emulator (e.g. xterm)

shell (e.g. sh)

user process

user process

user process

user process
§The Shell (aka the CLI)
the original operating system user interface
essential function: let the user issue requests to the operating system
e.g., fork/exec a program,
manage processes (list/stop/term),
browse/manipulate the file system
(a read-eval-print-loop REPL for the OS)
pid_t pid;
char buf[80], *argv[10];

while (1) {
    /* print prompt */
    printf("$ ");

    /* read command and build argv */
    fgets(buf, 80, stdin);
    for (i=0, argv[0] = strtok(buf, "\n");
        argv[i];
        argv[++i] = strtok(NULL, "\n");
    /* fork and run command in child */
    if ((pid = fork()) == 0)
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0);
        }
    /* wait for completion in parent */
    waitpid(pid, NULL, 0);
}
Demo:

examples/processes/simple_shell1.c
... but we are far from done :-}
all shells provide *task management* features

i.e., to run, track and manage *multiple* processes at a time
distinguish between *foreground* (fg) and *background* (bg) processes

- fg process “blocks” additional commands from being run

- can have multiple bg processes at once
some shell conventions:

- start bg process: `prog_name &`

- `fg/bg`: move a process into fg/bg
Demo:

`/bin/zsh`

(using "myspin.c")

```
./myspin 5
./myspin 5 &
fg %1
./myspin 5
ctrl-Z; bg/fg
```
fgets(buf, 80, stdin);

/* check if bg job requested */
if (buf[strlen(buf)-2] == '&') {
   bg = 1;
   buf[strlen(buf)-2] = 0;
} else
   bg = 0;

for (i=0, argv[0] = strtok(buf, " 
"); argv[i];
    argv[++i] = strtok(NULL, " 
");

/* fork and run command in child */
if ((pid = fork()) == 0)
   if (execvp(argv[0], argv) < 0) {
      printf("Command not found
");
      exit(0);
   }

/* wait for completion only if bg */
if (!bg) {
   waitpid(pid, NULL, 0);
}
Demo:

examples/processes/simple_shell2.c
background zombies!!!
if (!bg) {
    /* wait for fg job completion */
    waitpid(pid, NULL, 0);
}

/* ... and machine-gun down bg zombies */
while (waitpid(-1, NULL, WNOHANG) > 0);
(this is a really, really lousy hack.)

- inefficient & ugly

- no guarantee when reaping will occur
what we really want is a way to be *notified* when a child turns into a zombie

… so that we can run our reaping code
“notification” $\rightarrow$ exceptional control flow
§ Signals
signals are messages delivered by the kernel to user processes

- in response to OS events (e.g., segfault)
- or at the request of other processes
how “delivered”?
- by executing a *handler function* in the receiving process
aspects of signal processing:

1. *sending* a signal to a process
2. *registering* a handler for a given signal
3. *delivering* a signal (kernel mechanism)
4. *designing* a signal handler
1. *sending* a signal to a process

```c
int kill(pid_t pid, int sig);
```
<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>terminate process</td>
<td>terminal line hangup</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>terminate process</td>
<td>interrupt program</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>create core image</td>
<td>quit program</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>create core image</td>
<td>abort program (formerly SIGIOT)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>terminate process</td>
<td>kill program</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>create core image</td>
<td>bus error</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>create core image</td>
<td>segmentation violation</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>create core image</td>
<td>non-existent system call invoked</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>terminate process</td>
<td>write on a pipe with no reader</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>terminate process</td>
<td>real-time timer expired</td>
</tr>
<tr>
<td>17</td>
<td>SIGSTOP</td>
<td>stop process</td>
<td>stop (cannot be caught or ignored)</td>
</tr>
<tr>
<td>18</td>
<td>SIGTSTP</td>
<td>stop process</td>
<td>stop signal generated from keyboard</td>
</tr>
<tr>
<td>19</td>
<td>SIGCONT</td>
<td>discard signal</td>
<td>continue after stop</td>
</tr>
<tr>
<td>20</td>
<td>SIGCHLD</td>
<td>discard signal</td>
<td>child status has changed</td>
</tr>
<tr>
<td>30</td>
<td>SIGUSR1</td>
<td>terminate process</td>
<td>User defined signal 1</td>
</tr>
<tr>
<td>31</td>
<td>SIGUSR2</td>
<td>terminate process</td>
<td>User defined signal 2</td>
</tr>
</tbody>
</table>
```c
int main () {
    int stat;
    pid_t pid;
    if ((pid = fork()) == 0)
        while(1);
    else {
        kill(pid, SIGINT);
        wait(&stat);
        if (WIFSIGNALED(stat))
            psignal(WTERMSIG(stat),
                    "Child term due to");
    }
}
```

Child term due to: Interrupt
sometimes it’s convenient to be able to send a signal to *multiple* processes at once
mechanism: *process groups*
/* set pid's group to given pgid */
int setpgid(pid_t pid, pid_t pgid);

/* set caller's gid equal to its pid */
pid_t setpgrp();
a process automatically inherits its parent’s pgid when forked

- the founder of a group (i.e., whose pid = pgid) is the group leader

- become a group leader via setpgid
int \textbf{kill}(\texttt{pid_t pid, int sig});

if \textbf{kill} is given a \textit{negative} value for \texttt{pid},
\texttt{sig} is sent to \textit{all processes} with \texttt{gid = abs(pid)}
Computer
Science

setpgrep

shell
pid=10, pgid=10

user process
pid=11, pgid=11

user process
pid=12, pgid=10
Computer Science

shell
pid=10, pgid=10

user process
pid=11, pgid=11

setpgrp

user process
pid=12, pgid=12

user process
pid=13, pgid=12

user process
pid=11, pgid=11
user process pid=11, pgid=11

shell pid=10, pgid=10

user process pid=12, pgid=12

user process pid=13, pgid=12

kill(-12, SIGINT)
if ((pid = fork()) == 0) {
    setpgrp(); /* child establishes new group */
    printf("Child pgid = %d\n", getpgrp());
    for (i=0; i<3; i++)
        /* grandchildren inherit child's group */
        if (fork() == 0)
            while(1);
    while(1);
}
else {
    sleep(1);
    if (fork() == 0) {
        sprintf(buf, "%d", pid);
        execvp("ps", "ps", ":-0pgid", ":-g", buf, NULL);
    }
    sleep(1);
    kill(-pid, SIGINT);
}
if (fork() == 0) {
    setpgrp(); /* child establishes new group */
    printf("Child pgid = %d\n", getpgrp());
    for (i=0; i<3; i++)
        if (fork() == 0)
            while (1);
    while (1);
} else {
    sleep(1);
    if (fork() == 0) {
        sprintf(buf, "%d", pid);
        execlp("ps", "ps", "-Opgid", "-g", buf, NULL);
    }
    sleep(1);
    kill(-pid, SIGINT);
}

$ ./a.out
Child pgid = 26470
    PID  PGID   TT  STAT      TIME COMMAND
 26470 26470 s005  R      0:00.40 ./a.out
 26471 26470 s005  R      0:00.40 ./a.out
 26472 26470 s005  R      0:00.42 ./a.out
 26473 26470 s005  R      0:00.39 ./a.out

$ ps -g 26470
    PID  STAT   TT  STAT      TIME COMMAND
2. registering a handler for a given signal

```c
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```
sig_t signal(int sig, sig_t func);

func can be one of:

- a pointer to a signal handler function
- SIG_IGN, to ignore the signal
- SIG_DFL, to use the default action
sig_t signal(int sig, sig_t func);

some signals cannot be caught/ignored!
- i.e., default action is mandatory
- e.g., SIGKILL (9) = DIE!
```c
int main () {
    signal(SIGINT, SIG_IGN);
    kill(getpid(), SIGINT);
    while(1) {
        sleep(1);
        printf("And I still live!!!\n");
    }
    return 0;
}
```

And I still live!!!
And I still live!!!
^CAnd I still live!!!
And I still live!!!
^CAnd I still live!!!
^C^C^CAnd I still live!!!
Q: how does \(^C \rightarrow \text{SIGINT}\) ?

A: the terminal emulator (tty device) maps keystrokes to signals, which are sent to the session leader’s process group (typically, login shell)
$ stty -a
speed 9600 baud; 50 rows; 110 columns;
...
cchars: discard = ^O; dsusp = ^Y; eof = ^D; intr = ^C;
    lnext = ^V; quit = ^\; reprint = ^R; start = ^Q;
status = ^T; stop = ^S; susp = ^Z; weerase = ^W;
controlling tty

SIGINT

\^C

must forward signal to FG group

user process pid=11, pgid=11

user process pid=12, pgid=12

user process pid=13, pgid=12

shell pid=10, pgid=10
pid_t cpid;

int main () {
    if ((cpid = fork()) == 0) {
        signal(SIGINT, child_handler);
        setpgid(); /* child becomes group leader */
        while (1);
    }

    signal(SIGINT, parent_handler);
    while (1); /* parent doesn’t term by SIGINT! */
}

void parent_handler(int sig) {
    printf("Relaying SIGINT to child\n");
    kill(-cpid, SIGINT); /* send sig to child group */
}

void child_handler(int sig) {
    printf("Child dying...\n");
    exit(0);
}

$ ./a.out
^CRelaying SIGINT to child
Child dying...
† child processes inherit their parent’s signal handlers!
‡ but lose them when exec-ing a program
pid_t cpid;

int main () {
    signal(SIGINT, parent_handler);
    
    if ((cpid = fork()) == 0) {
        setpgrp(); /* child becomes group leader */
        while(1);
    }
    
    while (1); /* parent doesn’t term by SIGINT! */
}

void parent_handler(int sig) {
    printf("Relaying SIGINT to child\n");
    kill(-cpid, SIGINT); /* send sig to child group */
}

$ ./a.out
^CRelaying SIGINT to child
Relaying SIGINT to child
Relaying SIGINT to child
Relaying SIGINT to child
Relaying SIGINT to child
...

(kill to 0 = send sig to my own pgrp)
```c
void sigint_handler (int sig) {
    printf("Signal %d received\n", sig);
    sleep(1);
}

int main () {
    signal(SIGINT, sigint_handler);
    while (1) {
        pause(); /* pauses until signal */
        printf("Back in main\n");
    }
}
```
Demo:

examples/processes/sighandler1.c
3. *delivering* a signal (kernel mechanism)
per-process kernel structures: 2 *bit vectors*

- “pending” – 1 bit per pending signal
- “blocked” – 1 bit per blocked signal
adjusting blocked signals (*signal mask*):

```c
int sigprocmask(int how, /* SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
    const sigset_t *set, /* specified signals */
    sigset_t *oset);      /* gets previous mask */
```

(SIGKILL & SIGTSTP can’t be blocked!)
note: a newly forked child will inherit its parent’s blocked vector, but its pending vector will start out empty!
pending

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

blocked

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

0
```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
kill(the_pid, SIGINT);

sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);

kill(the_pid, SIGINT);
before resuming this process, kernel computes

\[ \text{pending} \land \neg \text{blocked} \]

\begin{verbatim}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

does not happen
(pending & ~blocked) ⇒ ∅
i.e., no signals to deliver — resume regular control flow
pending

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

blocked

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

```c
kill(the_pid, SIGTERM);
kill(the_pid, SIGUSR1);
```
<table>
<thead>
<tr>
<th></th>
<th>pending</th>
<th>&amp; ~blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>pending</th>
<th>&amp; ~blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>0</td>
<td>10000000000000010000000000000000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

deliver signals in order
(i.e., ignore, terminate, or run handler)
/* (user space code) */

void handler(int sig) {
    ...
}

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
/* (user space code) */
void handler(int sig) {
    ...  
}

mark signal as “delivered”
(and block this signal until the handler returns)
Q: what happens if a signal is received as its handler is running?
/* (user space code) */
void handler(int sig) {
    ...
    ...
    ...
    ...
    ...}

A: mark it as pending, but don’t run the handler again! (signal currently blocked)
Q: what happens if a signal is sent many times before its handler is run?
Q: what can we do?
A: nothing. (we can’t queue signals!)
Q: what happens if a signal is received as a handler for a lower priority one is already running?
A: we *preempt* the lower priority handler (and resume it — if possible — later)
4. *designing* a signal handler
Q: what can go wrong?
```c
struct foo { int x, y, z; } f;

int main () {
    int i = 1;

    f = (struct foo){ 0, 0, 0 };  
    signal(SIGALRM, tick);

    alarm(1); /* send SIGALRM in 1s */

    while(1) {
        f = (struct foo){ i, i, i };  
        i = (i + 1) % 100;
    }
}

void tick(int s) {
    printf("%d %d %d\n", f.x, f.y, f.z);
    alarm(1); /* send SIGALRM in 1s */
}
```
```c
int main () {
    int i;
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    for (i=0; i<10; i++) {
        if (fork() == 0) {
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
        }
    } while(1) pause();
}

void handler(int s) {
    static int x = 10, y = 20;
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```
```c
int x = 10, y = 20;

int main () {
    int i;
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
    for (i=0; i<10; i++) {
        if (fork() == 0) {
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
        }
    }
    while(1) pause();
}

void handler1(int s) { swapglobs(); }

void handler2(int s) { swapglobs(); }

void swapglobs() {
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```
lesson 1: signals can be delivered at any time
- may interrupt any nonatomic operation
- problematic if using global variables!
design goal 1: minimize use of global variables in sighandlers

- if needed, ideally use data that can be read/written atomically (most primitives)
**lesson 2**: a sighandler may execute in overlapping fashion (with itself)

- when used to handle multiple signals
design goal 2: prefer separate handlers for different signals

- otherwise, must design handlers to be *reentrant* — i.e., able to be called again (re-entered) when already executing
lesson 3: execution of sighandlers for separate signals may overlap

- any functions they call may have overlapping execution
design goal 3: keep sighandlers simple; minimize calls to other functions

- any functions called by sighandlers should be reentrant!
pid_t fg_pid;

int main () {
    ...
    signal(SIGCHLD, sigchld_handler);
    while (1) {
        ... /* read & parse command */
        if ((pid = fork()) == 0) {
            ... /* exec program in child */
        }
        if (!bg) {
            /* wait for fg job completion */
            fg_pid = pid;
            while (fg_pid != -1)
                sleep(1);
        }
    }
}

void sigchld_handler(int sig) {
    pid_t pid;
    /* centralized wait-ing (fg & bg) */
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        if (pid == fg_pid)
            fg_pid = -1;
    }
}
insidious problem caused by concurrency
need to be sure that certain events take place without interruption (no signals!)
direct approach: block signals
```c
void sigchld_handler(int sig) {
  ...
  fg_pid = -1;
}

int main () {
  ...
  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);

  signal(SIGCHLD, sigchld_handler);
  while (1) {
    ... /* read & parse command */
    sigprocmask(SIG_BLOCK, &mask, NULL);
    if ((pid = fork()) == 0) {
      ... /* exec program in child */
    }
    if (!bg) {
      /* wait for fg job completion */
      fg_pid = pid;
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      while (fg_pid != -1)
        sleep(1);
    }
  }
}
```

SIGCHLD is blocked!

(should also unblock signals in child)
† can also block signals when forced to call non-reentrant functions from sighandlers
Bonus: non-local jumps
what if we don’t want to continue where we left off after handling a signal?
```c
int main () {
    int choice;
    char buf[80];
    void (*menufns[])(void)[80] = { data_entry, data_analysis ];
    while(1) {
        print_menu();
        fgets(buf, 80, stdin);
        choice = atoi(buf);
        (*menufns[choice-1])();
    }
}

void print_menu() {
    printf("1: Data entry\n");
    printf("2: Data analysis\n");
    printf("Enter choice: ");
}

void data_entry() {
    int i;
    char buf[80];
    for (i=0; i<1000000; i++) {
        printf("> ");
        fgets(buf, 80, stdin);
        process_entry(buf);
    }
}
```

$ ./dproc
1: Data entry
2: Data analysis
Enter choice: 1
> 1,820,1840,3880
> 2,20,2084,2848
> 3,328,3840,28402
> 4,580,3780,8890
> 5,7,80,2788,28
> 6,9304,880,28801
> 7,928,2830,188
... 
> ^C
$

(dropped to shell)
```c
int reset = 0;

int main () {
    ...
    signal(SIGINT, sigint_handler);
    while(1) {
        ...
    }
}

void sigint_handler(int sig) {
    reset = 1;
}

void data_entry() {
    int i;
    char buf[80];
    for (i=0; i<1000000 && !reset; i++) {
        printf("> ");
        fgets(buf, 80, stdin);
        if (reset)
            break;
        process_entry(buf);
    }
    reset = 0;
}
```
painful: “reset” requires many changes to program logic, and potentially introduces reentrancy problems
```c
int main () {
...

while(1) {
    print_menu();
    fgets(buf, 80, stdin);
    choice = atoi(buf);
    (*menufns[choice-1])();
}
}

void sigint_handler(int sig) {
    ...
    ...
    ...
}

void data_entry() {
    int i;
    char buf[80];
    for (i=0; i<1000000; i++) {
        printf("> ");
        fgets(buf, 80, stdin);
        process_entry(buf);
    }
}
```

would prefer a direct jump to the reset position
/* save calling environment in env; return 0 */
int setjmp(jmp_buf env);

/* restore environment from env; "return" val */
void longjmp(jmp_buf env, int val);
restriction: `longjmp` destination must be in a `calling` frame (i.e., further up the stack)
jmp_buf env;

int main () {
    if (setjmp(env)) {
        printf("Restarting...\n");
    }
    signal(SIGINT, sigint_handler);
    while(1) {
        ...
    }
}

void sigint_handler(int sig) {
    longjmp(env, 1);
}

void data_entry() {
    int i;
    char buf[80];
    for (i=0; i<1000000; i++) {
        printf("> ");
        fgets(buf, 80, stdin);
        process_entry(buf);
    }
}

$ ./dproc
1: Data entry
2: Data analysis
Enter choice: 1
> 1,820,1840,3880
> 2,20,2084,2848
> 3,328,3840,28402
> 4,580,3780,8890
> 5,7,80,2788,28
> 6,9304,880,28801
> 7,928,2830,188
...
> ^C Restarting...
1: Data entry
2: Data analysis
Enter choice:
set/longjmp can also be used to implement exception handling
typedef enum {
    e_type_1 = 1,
    e_type_2,
    e_type_3
} ex_t;

jmp_buf last_env;

int main () {
    ex_t e;
    if ((e = setjmp(last_env)) == 0) {
        /* "try" calling foo */
        foo();
        printf("main completing normally\n");
    } else {
        printf("main caught exception %d\n", e);
    }
}
void foo() {
    ex_t e;
    jmp_buf saved_env;
    memcpy(saved_env, last_env, sizeof(jmp_buf));

    /* following is analogous to try-catch */
    if ((e = setjmp(last_env)) == 0) {
        bar();
        printf("foo completing normally\n");
    } else {
        switch(e) {
        case e_type_1:
            printf("foo caught exception %d\n", e);
            break;
        default:
            printf("foo re-throwing exception %d\n", e);
            memcpy(last_env, saved_env, sizeof(jmp_buf));
            longjmp(last_env, e);
        }
        memcpy(last_env, saved_env, sizeof(jmp_buf));
    }
}

void bar() {
    return;
}
```c
void foo() {
    ex_t e;
    jmp_buf saved_env;
    memcpy(saved_env, last_env, sizeof(jmp_buf));

    /* following is analogous to try-catch */
    if ((e = setjmp(last_env)) == 0) {
        bar();
        printf("foo completing normally\n");
    } else {
        switch(e) {
        case e_type_1:
            printf("foo caught exception %d\n", e);
            break;
        default:
            printf("foo re-throwing exception %d\n", e);
            memcpy(last_env, saved_env, sizeof(jmp_buf));
            longjmp(last_env, e);
        }
    }
    memcpy(last_env, saved_env, sizeof(jmp_buf));
}

void bar() {
    /* "throw" exception */
    longjmp(last_env, e_type_1);
}
```

void foo() {
    ex_t e;
    jmp_buf saved_env;
    memcpy(saved_env, last_env, sizeof(jmp_buf));

    /* following is analogous to try-catch */
    if ((e = setjmp(last_env)) == 0) {
        bar();
        printf("foo completing normally\n");
    } else {
        switch(e) {
        case e_type_1:
            printf("foo caught exception %d\n", e);
            break;
        default:
            printf("foo re-throwing exception %d\n", e);
            memcpy(last_env, saved_env, sizeof(jmp_buf));
            longjmp(last_env, e);
        }
    }
    memcpy(last_env, saved_env, sizeof(jmp_buf));
}

void bar() {
    /* "throw" exception */
    longjmp(last_env, e_type_2);
}