Process Management III

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
Interesting question:

Why are `fork` & `exec` separate syscalls?

/* i.e., why not: */

fork_and_exec("/bin/ls", ...)

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
A1: we might really want to just create duplicates of the current process (e.g.?)
A2: we might want to replace the current program without creating a new process
A3 (more subtle): we might want to “tweak” a process \textit{before} running a program in it
§The Unix Family Tree
BIOS

bootloader

kernel

“handcrafted” process
/etc/inittab

fork & exec

init

kernel
"Daemons"
e.g., sshd, httpd

fork & exec

kernel

init

fork & exec

getty
user process

shell (e.g. sh)

init

kernel

(a fork-ing party!)
(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, " 
");
        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 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
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\n");
        exit(0);
    }

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

examples/processes/simple_shell2.c
background zombies!!!
/* background zombie reaping? */

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 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” → 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 **kill**(*pid_t pid, int sig*);

if **kill** is given a *negative* value for **pid**, **sig** is sent to *all processes* with **gid** = **abs**(**pid**)
user process
pid=11, pgid=10

shell
pid=10, pgid=10
A diagram showing two user processes and a shell process. The user process with pid=11, pgid=11 is connected to the shell process with pid=10, pgid=10 via the `setpgid` function. The shell process is connected to another user process with pid=12, pgid=10.
user process
pid=11, pgid=11

shell
pid=10, pgid=10

user process
pid=12, pgid=12

user process
pid=13, pgid=12

setpgrp
Computer Science

User process: pid=11, pgid=11
User process: pid=12, pgid=12
User process: pid=13, pgid=12
Shell: pid=10, pgid=10

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);
        execlp("ps", "ps", "-Opgid", ",", buf, NULL);
    }
    sleep(1);
    kill(-pid, SIGINT);
}
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

1. sending a signal to a process

```c
int kill(pid_t pid, int sig);
```
2. registering a handler for a given signal

typedef void (*sig_t) (int);

sig_t signal(int sig, sig_t func);
sig_t signal(int sig, sig_t func);

- `func` is typically a pointer to a signal handler function — “callback” API
- some signals cannot be caught!
  (e.g., SIGKILL)
sig_t signal(int sig, sig_t func);

- func can also take special values:
  - SIG_IGN: ignore signal
  - SIG_DFL: use default action
int main () {
    signal(SIGINT, SIG_IGN);

    kill(getpid(), SIGINT);

    while(1) {
        sleep(1);
        printf("And I still live!!!\n");
    }
    return 0;
}
Q: how does ^C → 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; werase = ^W;
SIGINT

must forward signal to FG group

controlling tty

user process
pid=11, pgid=11

user process
pid=12, pgid=12

user process
pid=13, pgid=12

shell
pid=10, pgid=10

^C
pid_t cpid;

int main () {
    if ((cpid = fork()) == 0) {
        signal(SIGINT, child_handler);
        setpgid(0); /* 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
```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 \textit{(signal mask)}:

\begin{verbatim}
int sigprocmask(int how, /* SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
               const sigset_t *set, /* specified signals */
               sigset_t *oset);     /* gets previous mask */
\end{verbatim}

(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

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 0 0 0 0 0 0 0

blocked

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 0 0 0 0 0 0 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);

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

kill(the_pid, SIGINT);
```
Computer Science

<table>
<thead>
<tr>
<th>pending</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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 0 0 1 0 0 0</td>
</tr>
<tr>
<td>blocked</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>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 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Before resuming this process, kernel computes pending & ~blocked

<table>
<thead>
<tr>
<th>pending</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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 0 0 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1</td>
</tr>
<tr>
<td>&amp; ~blocked</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1</td>
</tr>
</tbody>
</table>

|         | 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 0 0 0 0 0 0 |

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
(pending & \sim\text{blocked}) \Rightarrow 0

i.e., no signals to deliver — resume regular control flow
kill(the_pid, SIGTERM);

kill(the_pid, SIGUSR1);

```
pending
0 1 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 1 0 0
```

```
blocked
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 0 0 1 0 0
```
pending

& ~blocked

31

0 1 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 1 0 0
0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1
0

31

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 0 0 0
0
deliver signals in order
(i.e., ignore, terminate, or run handler)
/* (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?
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);
}
```
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 signal handlers 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!
Back to background job reaping ...
int main () {
    ...
    while (1) {
        ...
        fgets(buf, 100, stdin);
        ...
        if ((pid = fork()) == 0) {
            if (execvp(argv[0], argv) < 0) {
                printf("Command not found\n");
                exit(0);
            }
        }
        if (!bg) {
            waitpid(pid, NULL, 0);
        }
    }
  }
```c
int main () {
    ...
    signal(SIGCHLD, sigchld_handler);
    
    while (1) {
        ...
        if ((pid = fork()) == 0) {
            ...
        }
        if (!bg) {
            waitpid(pid, NULL, 0);
        }
    }
    ...
}

void sigchld_handler(int sig) {
    pid_t pid;
    printf("sigchld handler called\n");
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        /* Q: why a loop? */
        printf("Reaping in sigchld handler\n");
    }
}
```

reaps before handler is called!
pid_t fg_pid = -1;

int main () {
    ...
    signal(SIGCHLD, sigchld_handler);

    while (1) {
        ...
        \[1\] if ((pid = fork()) == 0) {
            ...
        }

        if (!bg) {
            \[2\] fg_pid = pid;
            while (fg_pid != -1)
                sleep(1);
        }
    }
    ...
}

void sigchld_handler(int sig) {
    pid_t pid;
    printf("sigchld handler called\n");
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        printf("Reaping in sigchld handler\n");
        \[4\] if (fg_pid == pid)
            fg_pid = -1;
    }
}
```c
pid_t fg_pid = -1;

int main () {
    ...
    signal(SIGCHLD, sigchld_handler);

    while (1) {
        ...
        ❶ if ((pid = fork()) == 0) {
            ...
        }

        if (!bg) {
            4 fg_pid = pid;
            while (fg_pid != -1)
                sleep(1);
        }
    }
}

void sigchld_handler(int sig) {
    pid_t pid;
    printf("sigchld handler called\n");
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        printf("Reaping in sigchld handler\n");
        3 if (fg_pid == pid)
            fg_pid = -1;
    }
}
```

$ echo hello
hello
sigchld handler called
Reaping in sigchld handler

(hangs)
insidious *race condition* caused by *concurrency* (can’t predict when child will terminate / when signal will arrive)

need to ensure that certain sequences of events *cannot be interrupted*
direct approach: block signals
```c
int main () {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    ... 
    while (1) {
        ... 
        sigprocmask(SIG_BLOCK, &mask, NULL);
        ❶ if ((pid = fork()) == 0) {
            ... 
        }
        if (!bg) {
            ❷ fg_pid = pid;
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            while (fg_pid != -1)
                sleep(1);
        }
    }
    ... 
}

void sigchld_handler(int sig) {
    ...
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        if (fg_pid == pid)
            fg_pid = -1;
    }
}
```

- Ensures ❶, ❷ cannot be interrupted by ❸.
† can also block signals when forced to call non-reentrant functions from sighandlers
Another form of ECF: 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) = { data_entry,
                              data_analysis };
    while(1) {
        print_menu();
        fgets(buf, 80, stdin);
        choice = atoi(buf);
        (*menufns[choice-1])(void);
    }
}

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;
}
```

```
$ ./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
1: Data entry
2: Data analysis
Enter choice:
```
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 context in env; return 0 */
int setjmp(jmp_buf env);

/* restore saved context; "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...
");
  }
  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);
  }
}
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;
}
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);
}
some third-party libraries support similar behavior without explicitly saving context (possible due to direct access to memory)
e.g., libunwind
```c
void baz() {
    unw_cursor_t cursor;
    unw_context_t uc;
    unw_word_t ip, off;
    char name[80];

    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    do {
        unw_get_proc_name(&cursor, name, 80, &off);
        unw_get_reg(&cursor, UNW_REG_IP, &ip);
        printf("%5s :: ip = %lx\n", name, (long)ip);
        if (strcmp(name, "main") == 0)
            unw_resume(&cursor);
    } while (unw_step(&cursor));
}

void bar() {
    baz(); printf("Returned from baz\n");
}

void foo() {
    bar(); printf("Returned from bar\n");
}

int main() {
    foo(); printf("Returned from foo\n");
    return 0;
}
```

$ ./a.out
baz :: ip = 10e8d2c1f
bar :: ip = 10e8d2d59
foo :: ip = 10e8d2d79
main :: ip = 10e8d2d9d
Returned from foo