Exceptional Control Flow: Signals and Nonlocal Jumps

CS351: Systems Programming
Day 17: Oct. 20, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Heads up

DOE’s Office of Science Is Now Accepting Applications for Summer 2023 Undergraduate Internships

*Students Will Conduct Research and Technical Projects at National Laboratories*

Applications are currently being accepted for the Summer 2023 term of two undergraduate internship programs offered by the Department of Energy (DOE) Office of Science: the Science Undergraduate Laboratory Internships (SULI) program and the Community College Internships (CCI) program. The [application deadline is January 10, 2023, at 5:00 p.m. EST.](#)
Next time: back to in-person in SB104

- Tuesday: 4th lab will be assigned – what we’re covering will be useful for that lab.
- Monday: deadline for 3rd lab assignment
ECF Exists at All Levels of a System

- **Exceptions**
  - Hardware and operating system kernel software

- **Process Context Switch**
  - Hardware timer and kernel software

- **Signals**
  - Kernel software and application software

- **Nonlocal jumps**
  - Application code
Today

- Shells
- Signals
- Nonlocal jumps
Linux Process Hierarchy

Note: you can view the hierarchy using the Linux `pstree` command
Shell Programs

- A **shell** is an application program that runs programs on behalf of the user.
  - sh: Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - csh/tcsh: BSD Unix C shell
  - bash: “Bourne-Again” Shell (default Linux shell)

```c
int main()
{
    char cmdline[MAXLINE]; /* command line */

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

*Execution is a sequence of read/evaluate steps*
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE]; /* Holds modified command line */
    int bg; /* Should the job run in bg or fg? */
    pid_t pid; /* Process id */

    strcpy(buf, cmdline);
    bg = parse_line(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {
            /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
    } /* Parent waits for foreground job to terminate */
    if (!bg) {
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitfg: waitpid error");
    } else
        printf("%d %s", pid, cmdline);
    return;
}
Problem with Simple Shell Example

- Our example shell correctly waits for and reaps foreground jobs

- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
ECF to the Rescue!

Solution: Exceptional control flow
- The kernel will interrupt regular processing to alert us when a background process completes
- In Unix, the alert mechanism is called a *signal*
Today

- Shells
- Signals
- Nonlocal jumps
A **signal** is a small message that notifies a process that an event of some type has occurred in the system:

- Akin to exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Signal type is identified by small integer ID’s (1-30)
- Only information in a signal is its ID and the fact that it arrived

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>User typed ctrl-c</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Signal Concepts: Sending a Signal

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD).
  - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.
Signal Concepts: Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.

Some possible ways to react:

- **Ignore** the signal (do nothing)
- **Terminate** the process (with optional core dump)
- **Catch** the signal by executing a user-level function called *signal handler*
  - Akin to a hardware exception handler being called in response to an asynchronous interrupt:

1. Signal received by process $I_{next}$
2. Control passes to signal handler
3. Signal handler runs
4. Signal handler returns to next instruction
Signal Concepts: Pending and Blocked Signals

- **A signal is pending if sent but not yet received**
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **A process can block the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- **A pending signal is received at most once**
Signal Concepts: Pending/Blocked Bits

- Kernel maintains **pending** and **blocked** bit vectors in the context of each process
  - **pending**: represents the set of pending signals
    - Kernel sets bit k in **pending** when a signal of type k is delivered
    - Kernel clears bit k in **pending** when a signal of type k is received
  - **blocked**: represents the set of blocked signals
    - Can be set and cleared by using the **sigprocmask** function
    - Also referred to as the **signal mask**.
Every process belongs to exactly one process group

- **getpgrp()**: Return process group of current process
- **setpgid()**: Change process group of a process (see text for details)
 Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process or process group

- **Examples**
  - `/bin/kill -9 24818`
    Send SIGKILL to process 24818
  - `/bin/kill -9 -24817`
    Send SIGKILL to every process in process group 24817

---

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
   PID  TTY          TIME    CMD
24788 pts/2    00:00:00  tcsh
24818 pts/2    00:00:02  forks
24819 pts/2    00:00:02  forks
24820 pts/2    00:00:00  ps

linux> /bin/kill -9 -24817
linux> ps
   PID  TTY          TIME    CMD
24788 pts/2    00:00:00  tcsh
24823 pts/2    00:00:00  ps
```

Illinois Tech CS351 Fall 2022
Sending Signals from the Keyboard

- Typing `ctrl-c` (`ctrl-z`) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process
Example of \texttt{ctrl-c} and \texttt{ctrl-z}

\texttt{linux}\textgreater; ./forks 17
Child: \texttt{pid=28108} pgrp=28107
Parent: \texttt{pid=28107} pgrp=28107
\texttt{<types ctrl-z>}
Suspended
\texttt{linux}\textgreater; ps w
\begin{tabular}{lrrl}
PID & TTY & STAT & TIME COMMAND \\
27699 & pts/8 & Ss & 0:00 \texttt{-tcsh} \\
28107 & pts/8 & T & 0:01 ./forks 17 \\
28108 & pts/8 & T & 0:01 ./forks 17 \\
28109 & pts/8 & R+ & 0:00 \texttt{ps w} \\
\end{tabular}
\texttt{linux}\textgreater; fg
./forks 17
\texttt{<types ctrl-c>}
\texttt{linux}\textgreater; ps w
\begin{tabular}{lrrl}
PID & TTY & STAT & TIME COMMAND \\
27699 & pts/8 & Ss & 0:00 \texttt{-tcsh} \\
28110 & pts/8 & R+ & 0:00 \texttt{ps w} \\
\end{tabular}

\textbf{STAT} (process state) Legend:

\textbf{First letter:}
S: sleeping
T: stopped
R: running

\textbf{Second letter:}
s: session leader
+: foreground proc group

See “\texttt{man ps}” for more details
Sending Signals with `kill` Function

```c
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}  
```
Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process $p$
Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process $p$

- Kernel computes $p_{nb} = \text{pending} \& \sim\text{blocked}$
  - The set of pending nonblocked signals for process $p$

- If ($p_{nb} == 0$)
  - Pass control to next instruction in the logical flow for $p$

- Else
  - Choose least nonzero bit $k$ in $p_{nb}$ and force process $p$ to receive signal $k$
  - The receipt of the signal triggers some action by $p$
  - Repeat for all nonzero $k$ in $p_{nb}$
  - Pass control to next instruction in logical flow for $p$
Default Actions

Each signal type has a predefined default action, which is one of:

- The process terminates
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal
Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`

- Different values for `handler`:
  - `SIG_IGN`: ignore signals of type `signum`
  - `SIG_DFL`: revert to the default action on receipt of signals of type `signum`
  - Otherwise, `handler` is the address of a user-level `signal handler`
    - Called when process receives signal of type `signum`
    - Referred to as “installing” the handler
    - Executing handler is called “catching” or “handling” the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal
Signal Handling Example

```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("Received ctrl-c\n");
    sleep(2);
    printf("Processing...\n");
    fflush(stdout);
    sleep(1);
    printf("Complete\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```
Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program.

```
while (1)
  ;
  ... 
  
Process A
```

```
handler()
```

```
Process B
```

Time
Another View of Signal Handlers as Concurrent Flows

Signal delivered to process A

Signal received by process A

Process A

Process B

I<sub>curr</sub>

context switch

I<sub>next</sub>

user code (main)

kernel code

user code (main)

kernel code

user code (handler)

kernel code

user code (main)
Nested Signal Handlers

- Handlers can be interrupted by other handlers

(1) Program catches signal s

(2) Control passes to handler S

(3) Program catches signal t

(4) Control passes to handler T

(5) Handler T returns to handler S

(6) Handler S returns to main program

(7) Main program resumes
Blocking and Unblocking Signals

- **Implicit blocking mechanism**
  - Kernel blocks any pending signals of type currently being handled.
  - E.g., A SIGINT handler can’t be interrupted by another SIGINT

- **Explicit blocking and unblocking mechanism**
  - `sigprocmask` function

- **Supporting functions**
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set
Temporarily Blocking Signals

```c
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

/* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```
Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and share the same global data structures.
  - Shared data structures can become corrupted.

- We’ll explore concurrency issues later in the term.

- For now here are some guidelines to help you avoid trouble.
Guidelines for Writing Safe Handlers

■ G0: Keep your handlers as simple as possible
  ▪ e.g., Set a global flag and return

■ G1: Call only async-signal-safe functions in your handlers
  ▪ `printf`, `sprintf`, `malloc`, and `exit` are not safe!

■ G2: Save and restore `errno` on entry and exit
  ▪ So that other handlers don’t overwrite your value of `errno`

■ G3: Protect accesses to shared data structures by temporarily blocking all signals.
  ▪ To prevent possible corruption

■ G4: Declare global variables as `volatile`
  ▪ To prevent compiler from storing them in a register

■ G5: Declare global flags as `volatile sig_atomic_t`
  ▪ `flag`: variable that is only read or written (e.g. `flag = 1`, not `flag++`)
  ▪ Flag declared this way does not need to be protected like other globals
Async-Signal-Safety

- Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.

- Posix guarantees 117 functions to be async-signal-safe
  - Source: “man 7 signal”
  - Popular functions on the list:
    - _exit, write, wait, waitpid, sleep, kill
  - Popular functions that are not on the list:
    - printf, sprintf, malloc, exit
    - Unfortunate fact: write is the only async-signal-safe output function
Safely Generating Formatted Output

- Use the reentrant SIO (Safe I/O library) from csapp.c in your handlers.
  - ssize_t sio_puts(char s[]) /* Put string */
  - ssize_t sio_putl(long v)   /* Put long */
  - void sio_error(char s[])   /* Put msg & exit */

```c
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    Sio_puts("Received ctrl-c\n");
    sleep(2);
    Sio_puts("Processing...");
    sleep(1);
    Sio_puts("Complete\n");
    _exit(0);
}
```
The code snippet demonstrates how to handle signals correctly, especially in a scenario where you need to monitor child processes. It uses a `count` variable to keep track of the number of children, and a `child_handler` function to handle signals from child processes. The `fork14` function forks N processes and waits for them to complete, then counts the number of remaining children and iterates until all children have been reaped.

```c
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0); /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */

    . . .
}
```

The output of the program shows that the `child_handler` function successfully reaps the children and decrements the `ccount` variable accordingly.

```
linux> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```
Correct Signal Handling

- Must wait for all terminated child processes
  - Put `wait` in a loop to reap all terminated children

```c
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
linux> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
linux>
```
Portable Signal Handling

- Ugh! Different versions of Unix can have different signal handling semantics
  - Some older systems restore action to default after catching signal
  - Some interrupted system calls can return with errno == EINTR
  - Some systems don’t block signals of the type being handled

- Solution: `sigaction`

```c
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```
Synchronizing Flows to Avoid Races

- Simple shell with a subtle synchronization error because it assumes parent runs before child.

```c
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;

    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```
Synchronizing Flows to Avoid Races

- SIGCHLD handler for a simple shell

```c
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}
Explicitly Waiting for Signals

- Handlers for program explicitly waiting for SIGCHLD to arrive.

```c
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c
Explicitly Waiting for Signals

```c
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid = 0;
        Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
        /* Wait for SIGCHLD to be received (wasteful!) */
        while (!pid)
            ;
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

Similar to a shell waiting for a foreground job to terminate.
 Explicitly Waiting for Signals

- Program is correct, but very wasteful
- Other options:

  ```c
  while (!pid) /* Race! */
  pause();
  ```
  ```c
  while (!pid) /* Too slow! */
  sleep(1);
  ```

- Solution: sigsuspend
Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`

- Equivalent to atomic (uninterruptable) version of:

```c
    sigprocmask(SIG_BLOCK, &mask, &prev);
    pause();
    sigprocmask(SIG_SETMASK, &prev, NULL);
```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BBLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0); /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

        /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
Today

- Shells
- Signals
- Nonlocal jumps
  - Consult your textbook and additional slides
Summary

- Signals provide process-level exception handling
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Be very careful when writing signal handlers

- Nonlocal jumps provide exceptional control flow within process
  - Within constraints of stack discipline
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- The form is anonymous. (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I’ll remind about this at each lecture.
Additional slides
Nonlocal Jumps: `setjmp/longjmp`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling

- `int setjmp(jmp_buf j)`
  - Must be called before `longjmp`
  - Identifies a return site for a subsequent `longjmp`
  - Called once, returns one or more times

- Implementation:
  - Remember where you are by storing the current `register context`, `stack pointer`, and `PC value` in `jmp_buf`
  - Return 0
**setjmp/longjmp (cont)**

- **void longjmp(jmp_buf j, int i)**
  - Meaning:
    - return from the *setjmp* remembered by jump buffer *j* again ...
    - ... this time returning *i* instead of 0
  - Called after *setjmp*
  - Called **once**, but **never** returns

- **longjmp Implementation:**
  - Restore register context (stack pointer, base pointer, PC value) from jump buffer *j*
  - Set `%eax` (the return value) to *i*
  - Jump to the location indicated by the PC stored in jump buf *j*
setjmp/longjmp Example

- Goal: return directly to original caller from a deeply-nested function

```c
/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```
jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}
Limitations of Nonlocal Jumps

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    } else {
        P2();
    }
}

P2()
{
    . . . P2(); . . . P3();
}

P3()
{
    longjmp(env, 1);
}
```
Limitations of Long Jumps (cont.)

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    P2(); P3();
}

P2()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    }
}

P3()
{
    longjmp(env, 1);
}
```
Putting It All Together: A Program That Restarts Itself When ctrl-c’d

```c
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

int main()
{
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts("starting\n");
    } else
        Sio_puts("restarting\n");

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

```
linux> ./restart
starting
processing...
processing...
processing...
restarting
processing...
restarting
processing...
restarting
processing...
processing...
```