Exceptional Control Flow: Signals and Nonlocal Jumps

15-213: Introduction to Computer Systems
14th Lecture, Oct. 9, 1014

Instructors:
Greg Ganger, Greg Kesden, and Dave O’Hallaron
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

Previous Lecture

This Lecture

Textbook and supplemental slides
Today

- Shells
- Signals
- Nonlocal jumps
Unix Process Hierarchy

- [0]
  - init [1]
    - Daemon e.g. httpd
    - Login shell
      - Child
      - Child
      - Child
        - Grandchild
        - Grandchild
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** BSD Unix C shell (**tcsh**: enhanced **csh** at CMU and elsewhere)
- **bash** “Bourne-Again” Shell

```c
int main() {
    char cmdline[MAXLINE];

    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]; /* argv for execve() */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* process id */

    bg = parseline(cmdline, argv);
    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);
            }
        }
        if (!bg) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
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
Signals

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>Interrupt (e.g., ctl-c from keyboard)</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 &amp; Dump</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

(2) Control passes to signal handler

I_{curr} \quad I_{next}

(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*. 
Sending Signals: Process Groups

- Every process belongs to exactly one process group

```plaintext
- Foreground job
  - Child
  - Child

- Background job #1
  - Background process group 32
    - Child
    - Child

- Background job #2
  - Background process group 40

getpgrp()
Return process group of current process

setpgid()
Change process group of a process
```
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
```
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

```
+----------------------------------+
| Foreground job                   |
| ppid=20                          |
| ppgid=20                         |
+----------------------------------+
| Child                            |
| ppid=21                          |
| ppgid=20                         |
+----------------------------------+
| Child                            |
| ppid=22                          |
| ppgid=20                         |
+----------------------------------+
| Background job #1                |
| ppid=32                          |
| ppgid=32                         |
+----------------------------------+
| Background job #2                |
| ppid=40                          |
| ppgid=40                         |
+----------------------------------+
```

**Background process group 32**

**Background process group 40**

**Foreground process group 20**
Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
/types ctrl-z>
Suspended
bluefish> ps w
   PID TTY STAT TIME COMMAND
  27699 pts/8 Ss  0:00 -tcsh
  28107 pts/8 T  0:01 ./forks 17
  28108 pts/8 T  0:01 ./forks 17
  28109 pts/8 R+  0:00 ps w
bluefish> fg
./forks 17
/types ctrl-c>
bluefish> ps w
   PID TTY STAT TIME COMMAND
  27699 pts/8 Ss  0:00 -tcsh
  28110 pts/8 R+  0:00 ps w
```

STAT (process state) Legend:

**First letter:**
- S: sleeping
- T: stopped
- R: running

**Second letter:**
- s: session leader
- +: foreground proc group

See “man ps” for more details
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    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 $p$

Important: All context switches are initiated by calling some exception handler.
Receiving Signals

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

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

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

- Else
  - Choose least nonzero bit $k$ in $\text{pnb}$ and force process $p$ to receive signal $k$
  - The receipt of the signal triggers some action by $p$
  - Repeat for all nonzero $k$ in $\text{pnb}$
  - 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 terminates and dumps core
  - 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 `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
#include "csapp.h"

void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\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)
    handler()
    ...
```

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

Process B
```
Another View of Signal Handlers as Concurrent Flows

Signal delivered

Process A

- $I_{curr}$
- Signal received

Process B

- User code (main)
- Kernel code
- User code (main)
- Kernel code
- User code (handler)

\{ context switch \}

- User code (main)

\{ context switch \}

$\rightarrow$
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);
```
Guidelines for Writing Safe Handlers

- 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:APP2e 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:
    - exit, **printf**, sprintf, malloc
Safely Generating Formatted Output

- Option 1: temporarily block all signals during each call to printf EVERYWHERE in the program.

```c
/* safe_printf – async-signal-safe wrapper for printf */
void safe_printf(const char *format, ...) {
    char buf[MAXBUF];
    va_list args;
    sigset_t mask, prev_mask;

    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    va_start(args, format);
    vsnprintf(buf, sizeof(buf), format, args);
    va_end(args);
    write(1, buf, strlen(buf));

    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
}
```
Safely Generating Formatted Output

- Option 2: 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 */`

- The new `csapp.c` file is available on the course schedule page and your shell lab handout materials.

```c
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    sio_puts("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    sio_puts("Well...".mvp);
    sleep(1);
    sio_puts("OK. :-\n");
    _exit(0);
}
```
Signal Handler Funkiness

Pending signals are not queued
- For each signal type, just have single bit indicating whether or not signal is pending
- Even if multiple processes have sent this signal
- This program may get stuck in final loop

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    safe_printf("Received signal %d from process %d\n", sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1); /* deschedule child */
            exit(0); /* Child: Exit */
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}```
Signal Handler Funkiness

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal
  - This program may get stuck in final loop

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    safe_printf(
        "Received signal %d from process %d\n", sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
    {
        pid_t pid[i] = fork();
        if ((pid[i] != 0) && (i < N))
        {
            sleep(1); /* deschedule child */
            exit(0); /* Child: Exit */
        }
    }
    while (ccount > 0)
    {
        pause(); /* Suspend until signal occurs */
    }
```
Living With Nonqueueing Signals

- Must wait for all terminated jobs
  - Have handler loop with `waitpid` to get all jobs

```c
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    int n = 0;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        safe_printf("Received signal %d from process %d.  n = %d\n", sig, pid, n++);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```
Living With Nonqueuing Signals

- Must wait for all terminated jobs
  - Have handler loop with `waitpid` to get all jobs

```c
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    int n = 0;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        safe_printf("Received signal %d from process %d.  n = %d\n", sig, pid, n++);
    }
}
```

```c
void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

greatwhite> forks 15
Received signal 17 from process 27476.  n = 0
Received signal 17 from process 27477.  n = 0
Received signal 17 from process 27478.  n = 0
Received signal 17 from process 27479.  n = 1
Received signal 17 from process 27480.  n = 0
```
Portable Signal Handling

- Different versions of Unix can have different signal handling semantics
  - Restore action to default after catching signal
  - Some interrupted system calls can return with 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 ssigns 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;
}
```
Corrected Shell Program w/o Race

```c
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

```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);
}
```
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 waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```
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);
```
Waiting for Signals with `sigsuspend`

```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);

        /* 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
Summary

- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler

- **Nonlocal jumps provide exceptional control flow within process**
  - Within constraints of stack discipline
Additional slides
What Is a “Background Job”? 

- Users generally run one command at a time
  - Type command, read output, type another command

- Some programs run “for a long time”
  - Example: “delete this file in two hours”

```
unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

- A “background” job is a process we don't want to wait for

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix> # ready for next command
```
The World of Multitasking

- System runs many processes concurrently

- Process: executing program
  - State includes memory image + register values + program counter

- Regularly switches from one process to another
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority

- Appears to user(s) as if all processes executing simultaneously
  - Even though systems can only execute one process (or a small number of processes) at a time
  - Except possibly with lower performance than if running alone
Programmer’s Model of Multitasking

- **Basic functions**
  - `fork` spawns new process
    - Called once, returns twice
  - `exit` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait` and `waitpid` wait for and reap terminated children
  - `execve` runs new program in existing process
    - Called once, (normally) never returns

- **Programming challenge**
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g. “Fork bombs” can disable a system
**Signal Handling Example**

```c
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            while(1); /* child infinite loop */
        }
    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);
    }
}
```

linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
```
A Program That Reacts to Internally Generated Events

#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    safe_printf("BEEP\n");

    if (++beeps < 5)
        alarm(1);
    else {
        safe_printf("BOOM!\n");
        _exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */

    while (1) {
        /* handler returns here */
    }
}

internal.c

linux> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
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`
# include <setjmp.h>

jmp_buf buf;

tmp() {
    if (setjmp(buf) != 0) {
        printf("back in main due to an error\n");
    else
        printf("first time through\n");
        p1(); /* p1 calls p2, which calls p3 */
    }
    ...

p3() {
    <error checking code>
    if (error)
        longjmp(buf, 1)
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);  }
```

Before longjmp

| env | P1 | P2 | P2 | P3 |

After longjmp

| env | P1 | P2 | P2 | P3 |
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);
}
```

Diagram:
- `P1` calls `P2`
- `P2` returns
- `longjmp` to `P1`
- `setjmp` in `P2`
Putting It All Together: A Program That Restarts Itself When \texttt{ctrl-c’d}

```c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

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

main() {
    signal(SIGINT, handler);
    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");

    while(1) {
        sleep(1);
        printf("processing...\n");
    }
}
```

greatwhite> ./restart
starting
processing...
processing...
restarting
processing...
processing...
restarting
processing.
processing...
processing...

```
Ctrl-c

Ctrl-c

```

restart.c
A Program That Reacts to Externally Generated Events (Ctrl-c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    safe_printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    safe_printf("Well...\n");
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
    }
}
```

external.c

```bash
linux> ./external
<ctrl-c>
You think hitting ctrl-c will stop the bomb?
Well...OK
linux>
```