Program and OS Interaction: Multitasking and Signals

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
Topics

- Multitasking, shells
- Signals
Process Abstraction Review

- **Process: executing program**
  - State includes:
    - memory image
    - register values including the program counter (current instruction)
    - file handles + etc.

- **OS runs processes concurrently**
  - 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
Programmer’s Model of Multitasking

- **Basic functions**
  - `fork` spawns a duplicate 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, never returns (except on error)

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

[0]

init [1]

Daemon
e.g. httpd

Login shell

Child

Child

Child

Grandchild

Grandchild
Shell Programs

- **Shell**: user-level program that runs programs for the user
  - `sh/bash/ksh` - Original Unix shell / Bourne-Again shell / **Korn shell**
    - bash/ksh are more feature rich versions of sh
  - `csh/tcsh` - BSD Unix C shell
    - tcsh is just a more feature rich version of csh

```c
int main() {
    char cmdline[MAXLINE];
    while (1) {
        /* prompt for cmd */
        printf("> ");

        /* read cmd */
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate cmd */
        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);
    }
}
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) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else
            printf("%d %s", pid, cmdline);
    }
}

Correctly reaps foreground jobs

What about background jobs?
What about “Background Job”?

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

  ```bash
  linux> sleep 7200 # shell stuck for 2 hours
  ```

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

  ```bash
  linux> sleep 7200 &
  [1] 907
  Linux>
  ```

  PID of background process

  Background Job#
```c
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.
", argv[0]);
                exit(0);
            }
        }
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0) {
                unix_error("waitfg: waitpid error");
            }
        } else {
            printf("%d %s", pid, cmdline);
        }
    }
}
```

Correctly reaps foreground jobs

What about background jobs?
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
    - Won’t be reaped unit shell terminates
    - Zombies hold onto resources (memory usage, etc.)
    - Run out of the maxproc quota imposed by OS

Solution: Signals
- Have OS interrupt the shell process to alert it when a background job completes

```bash
linux> ulimit -u 6111
```
Topics

- Multitasking, shells
- Signals
Signals

- A **signal** is a small message that notifies a process of an event
  - Signals are for processes | exceptions/interrupts are for OS
  - sent from the kernel (sometimes at the request of another process) to a process (an “upcall”)
  - only information in a signal is an integer ID (1-30)

- Each signal type has a **default action**, e.g.

<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>Kills the program (cannot override/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>15</td>
<td>SIGTERM</td>
<td>Terminate</td>
<td>Asks program to terminate (program exits cleanly)</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Sending a Signal

- Kernel sends a signal to a *destination process* by updating some state in that process

- When to send a signal:
  - Detection of a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - A process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process (`kill` syscall can send any kind of signal, not just `SIGKILL`)
Receiving a Signal

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

- Three 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
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
    - Further signals of the same type 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

- 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
Process Groups

- Every process belongs to exactly one process group

- 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 sends a SIGINT while ctrl-z sends a SIGTSTP to every job in the foreground process group.
  - SIGINT (ctrl-c) – default action is to terminate each process
  - SIGTSTP (ctrl-z) – default action is to stop (suspend) each process

```
<table>
<thead>
<tr>
<th>Process Group</th>
<th>PID</th>
<th>Pgroupid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Foreground job</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>Background job #1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Background job #2</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>
```

Diagram:
- Shell
  - Foreground job
    - Child
    - Child
  - Background job #1
  - Background job #2

- Process group 20 (foreground)
- Process group 32
- Process group 40
Example of `ctrl-c` and `ctrl-z`

```bash
linux> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
```

```bash
linux> 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
```

```bash
linux> fg
./forks 17
<types ctrl-c>
```

```bash
linux> ps w
  PID   TTY     STAT   TIME   COMMAND
27699 pts/8   Ss    0:00   -tcsh
28110 pts/8   R+     0:00   ps w
```

> – prompt
ps – process status command
fg – foreground command

**STAT (Status) First letter:**
S: sleeping
T: stopped
R: running

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

See “man ps” for more details
Sending Signals with `kill` Function

```c
void fork12()
{
    /* assume N is a constant set to 5 */
    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 \( p \)

- Kernel computes \( \text{pnb} = \text{pending} \land \neg \text{blocked} \)
  - 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 \textit{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

- User program can alter default signal action using signal syscall:
  - `handler_t *signal(int signum, handler_t *handler)`

- Different values for `handler` for signals of type `signum`:
  - SIG_IGN: ignore
  - SIG_DFL: revert to the default action
  - The address of a `function` to call
    - Called when process receives signal of type `signum`
    - Executing handler is called “catching” or “handling” the signal
    - When the handler executes its return statement, control passes back to the instruction of the process that was interrupted
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 status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
Signals Handlers Subtlety

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

- “concurrently” in the “not sequential” sense

```c
while (1) {
    handler();
    ...
}
```
Signal Handlers as Concurrent Flows

Signal delivered

\[ I_{\text{curr}} \]

\[ \rightarrow \]

user code (main)

kernel code

user code (main)

context switch

Signal received

\[ I_{\text{next}} \]

\[ \rightarrow \]

user code (handler)

kernel code

user code (main)

context switch

Process A

Process B
Signal Handler Race

```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 */
}
```

Does this program always exit?

- Pending signals are not queued
  - For each signal type, there is a single bit indicating whether or not signal is pending
    - Even if multiple signals were sent

```
linux> ./forks 14
Received SIGCHLD signal 17 for process 21344
Received SIGCHLD signal 17 for process 21345
```
Living With Nonqueuing Signals

- Must check for all terminated jobs
  - Typically loop with `wait`

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

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

```bash
linux> forks 15
Received signal 17 from process 27476
Received signal 17 from process 27477
Received signal 17 from process 27478
Received signal 17 from process 27479
Received signal 17 from process 27480
linux>
```
More Signal Handler Funkiness

- Signal arrival during long system calls (say a `read`)
- Signal handler interrupts `read` call
  - Linux: upon return from signal handler, the `read` call is restarted
  - Some other flavors of Unix can cause the `read` call to fail with an `EINTERR error number (errno)
    in this case, the program can restart the system call

- Subtle differences like these complicate the writing of portable code that uses signals
Sample handler: Ctrl-c/SIGINT

```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...");
    sleep(1);
    printf("OK\n");
    exit(0);
}

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

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

```c
#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 */
    }
}
```

```
linux> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```
Async-Signal-Safety

- Function is `async-signal-safe` if either reentrant (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
  - `write` is on the list, `printf` is not
- One solution: async-signal-safe wrapper for `printf`:

```c
void safe_printf(const char *format, ...) {
  char buf[MAXS];
  va_list args;

  va_start(args, format); /* reentrant */
  vsnprintf(buf, sizeof(buf), format, args); /* reentrant */
  va_end(args); /* reentrant */
  write(1, buf, strlen(buf)); /* async-signal-safe */
}
```

safe_printf.c
Today

- Multitasking, shells
- Signals
- Nonlocal jumps
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 buffer `j`
```c
#include <setjmp.h>
jmp_buf buf;

main() {
    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);
}
```
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 <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

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c

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

Ctrl-c
Summary

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

- **Some caveats**
  - Very high overhead
    - >10,000 clock cycles
    - Only use for exceptional conditions
  - Don’t have queues
    - Just one bit for each pending signal type

- **Nonlocal jumps provide exceptional control flow within process**
  - Within constraints of stack discipline