Exceptional Control Flow: Signals and Nonlocal Jumps

CS-281: Introduction to Computer Systems

Instructor:
Dr. Thomas C. Bressoud

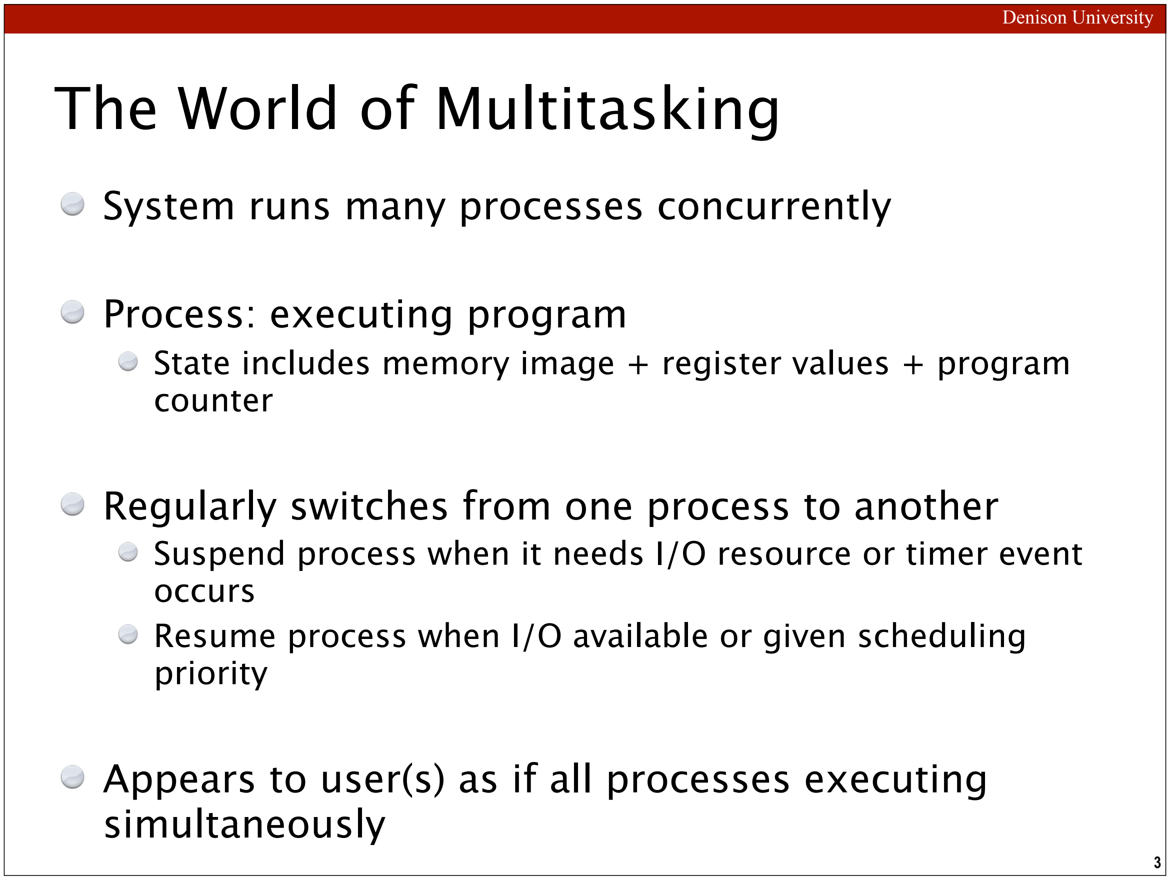
ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software
- Process Context Switch
  - Hardware timer and kernel software

Previously

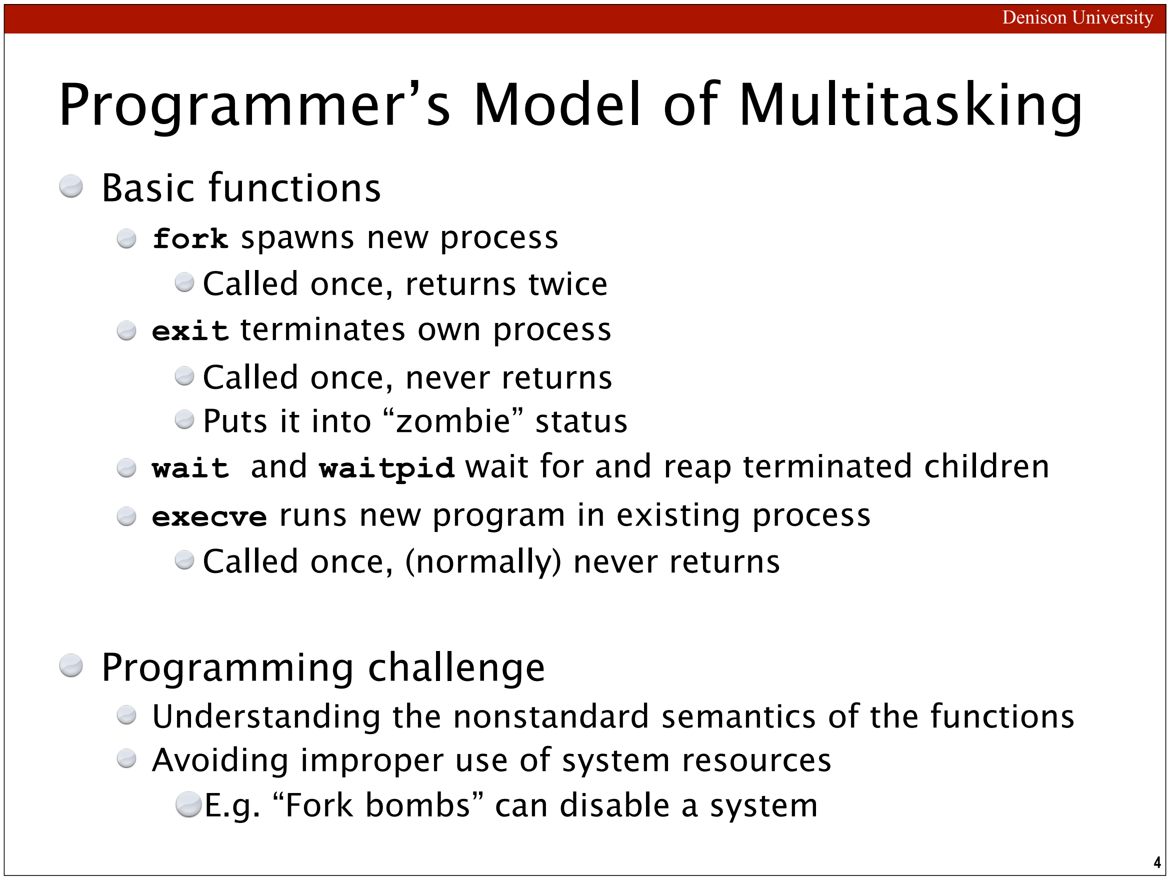
- Signals
  - Kernel software
- Nonlocal jumps
  - Application code

Now



The World of Multitasking

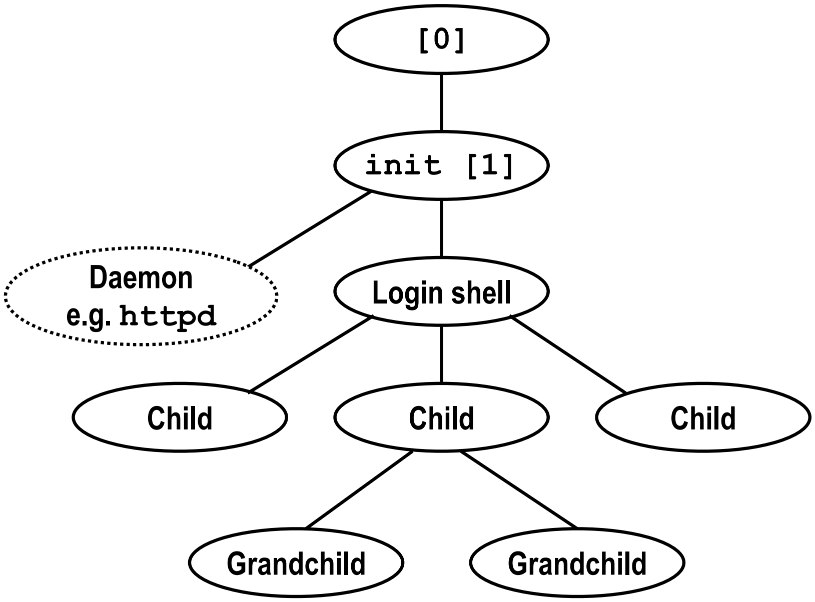
- System runs many processes concurrently
- Process: executing program
  - State includes memory image + register values + program counter
- Regularly switches from one process to another
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority
- Appears to user(s) as if all processes executing simultaneously



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

Unix Process Hierarchy



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)

```
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

Simple Shell `eval` Function

```
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);
    }
}
```

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
```

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
  - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

```
unix> limit maxproc          # csh syntax
maxproc       202752
unix> ulimit -u              # bash syntax
202752
```

ECF to the Rescue!

- Problem
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion
  - Regular control flow is “wait until running job completes, then reap it”
- 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**

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

ID Name	Default Action	Corresponding Event
2 SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9 SIGKILL	Terminate	Kill program (cannot override or
11 SIGSEGV	Terminate &	Segmentation violation
14 SIGALRM	Terminate	Timer signal
17 SIGCHLD	Ignore	Child stopped or terminated

Sending a Signal

- A signal is generated and the kernel sends 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

Receiving a Signal

- The kernel delivers and a destination process receives a signal when it is forced by the kernel to react in some way 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 in response to an asynchronous interrupt

Pending and Blocked Signals

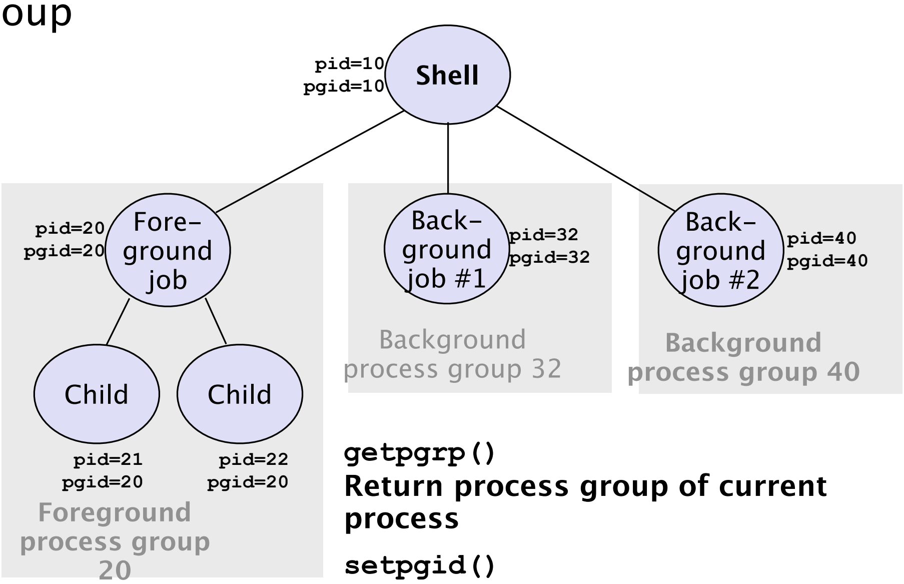
- A signal is pending if sent (generated) but not yet received (delivered)
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- A process can block the receipt of certain signals
  - Blocked signals can be generated, 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

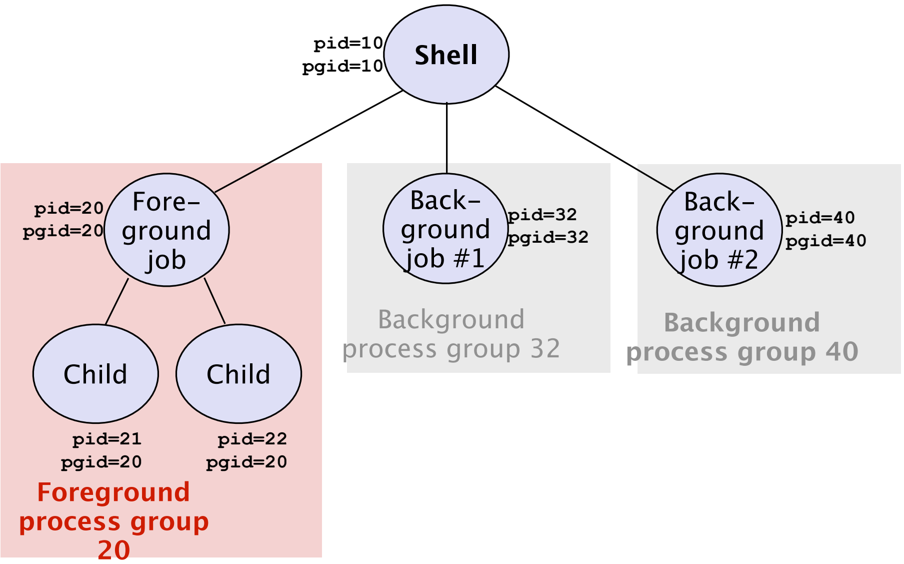
- /bin/kill program sends arbitrary signal to a process or process group

- 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
linux>
```

Sending Signals from the

- Typing ctrl-c (ctrl-z) sends 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 ctrl-c and ctrl-z

```
219k> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
219k> 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
219k> fg
./forks 17
<types ctrl-c>
219k> 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

Sending Signals with kill

```
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
- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process p
- If (`pnb == 0`)
  - Pass control to next instruction in the logical flow for p
- Else
  - Choose least nonzero bit k in **`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 **`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

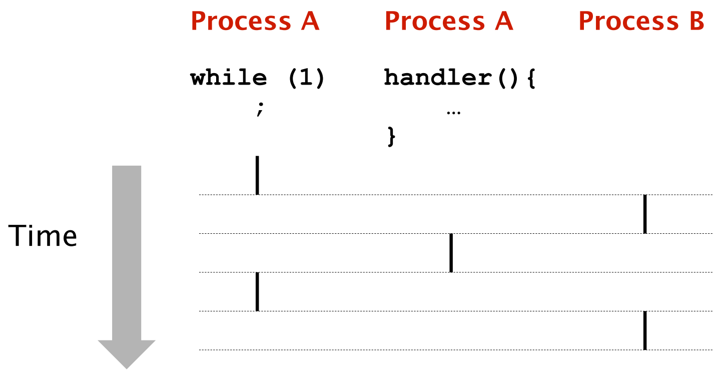
Signal Handling Example

```
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 infin
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n'
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_sta
        if (WIFEXITED(child_status))
            printf("Child %d termina
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
219k> ./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
Process 25419 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
219k>
```

Signals Handlers as Concurrent Flows

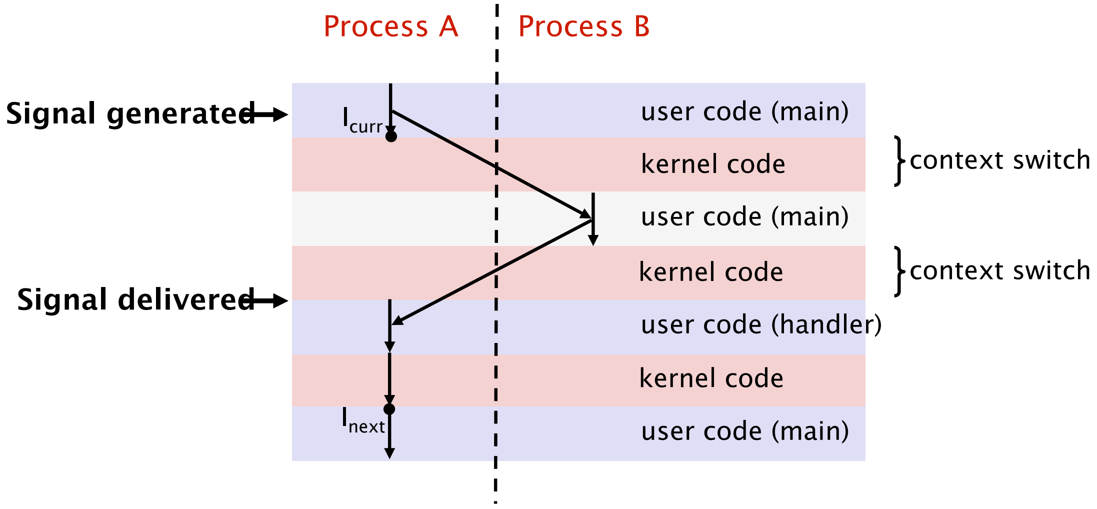
- A signal handler is a separate logical flow (not process) that runs concurrently with the main program
  - "concurrently" in the "not sequential" sense



Process A Process A Process B

```
while (1)     handler(){
     ;            …
              }
```

Another View of Signal Handlers as Concurrent Flows



Process A Process B

Signal generated

Signal delivered

user code (main)

kernel code

user code (main)

kernel code

user code (handler)

kernel code

user code (main)

context switch

context switch

Signal Handler Funkiness

```
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()
          sleep(1); /* des
          exit(0);  /* Chi
      }
    while (ccount > 0)
      pause(); /* Suspend until signal occurs */
}
```

```
219k> ./forks 14
Received SIGCHLD signal 17 for process 21344
Received SIGCHLD signal 17 for process 21345
```

- 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

Living With Nonqueuing Signals

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

```
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(SIGC
    . . .
}
```

```
219k> ./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
219k>
```

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 automatically
  - Some other flavors of Unix can cause the **`read`** call to fail with an **`EINTER`** error number (**`errno`**)
    in this case, the application program can restart the slow system call
- Subtle differences like these complicate the writing of portable code that uses signals
  - Consult your textbook for details

A Program That Reacts to Externally Generated Events (Ctrl-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 ctl-c handler */
  while(1) {
  }
}
```

```
219k> ./external
<ctrl-c>
You think hitting ctrl-c will stop
the bomb?
Well...OK
219k>
```

external.c

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 */
  }
}
```

```
219k> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
219k>
```

internal.c

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`:

```
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

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`**

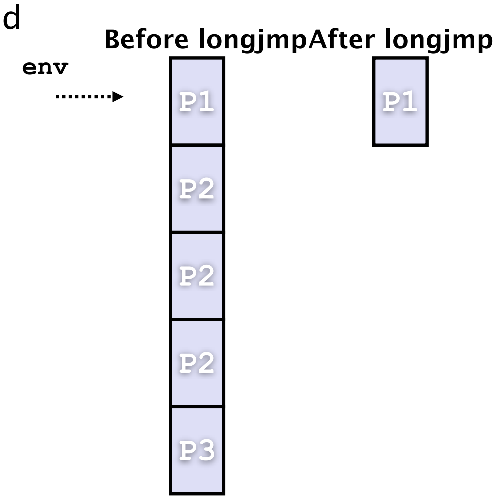
setjmp/longjmp Example

```
#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

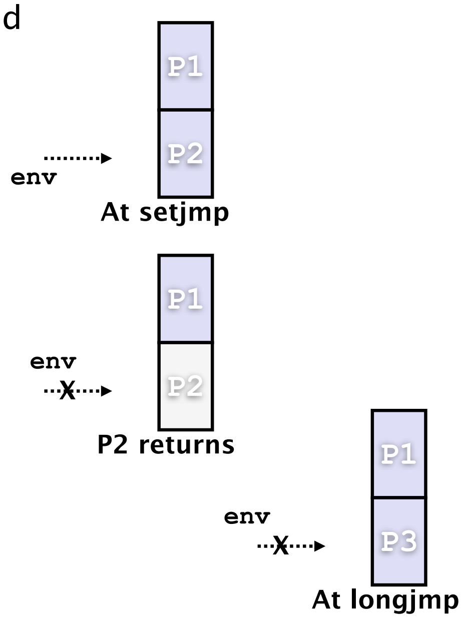
```
jmp_buf env;

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

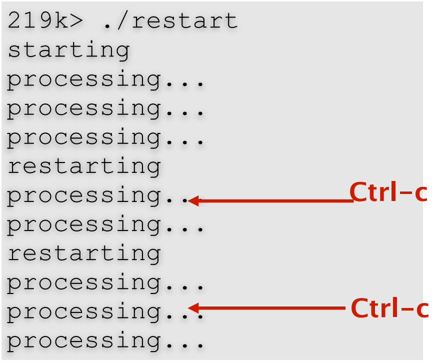
```
jmp_buf env;

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

```
#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");
                }
}
```

restart.c



```
219k> ./restart
starting
processing...
processing...
processing...
restarting
processing...
processing...
restarting
processing...
processing...
processing...
```

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