Exceptional Control Flow II

Today

- Process Hierarchy
- Shells
- Signals
- Nonlocal jumps

Next time

- I/O

ECF exists at all levels of a system

- Exceptions
  - Hardware and operating system kernel software

Previous Lecture

- Concurrent processes
  - Hardware timer and kernel software
- Signals
  - Kernel software
- Non-local jumps
  - Application code

This Lecture

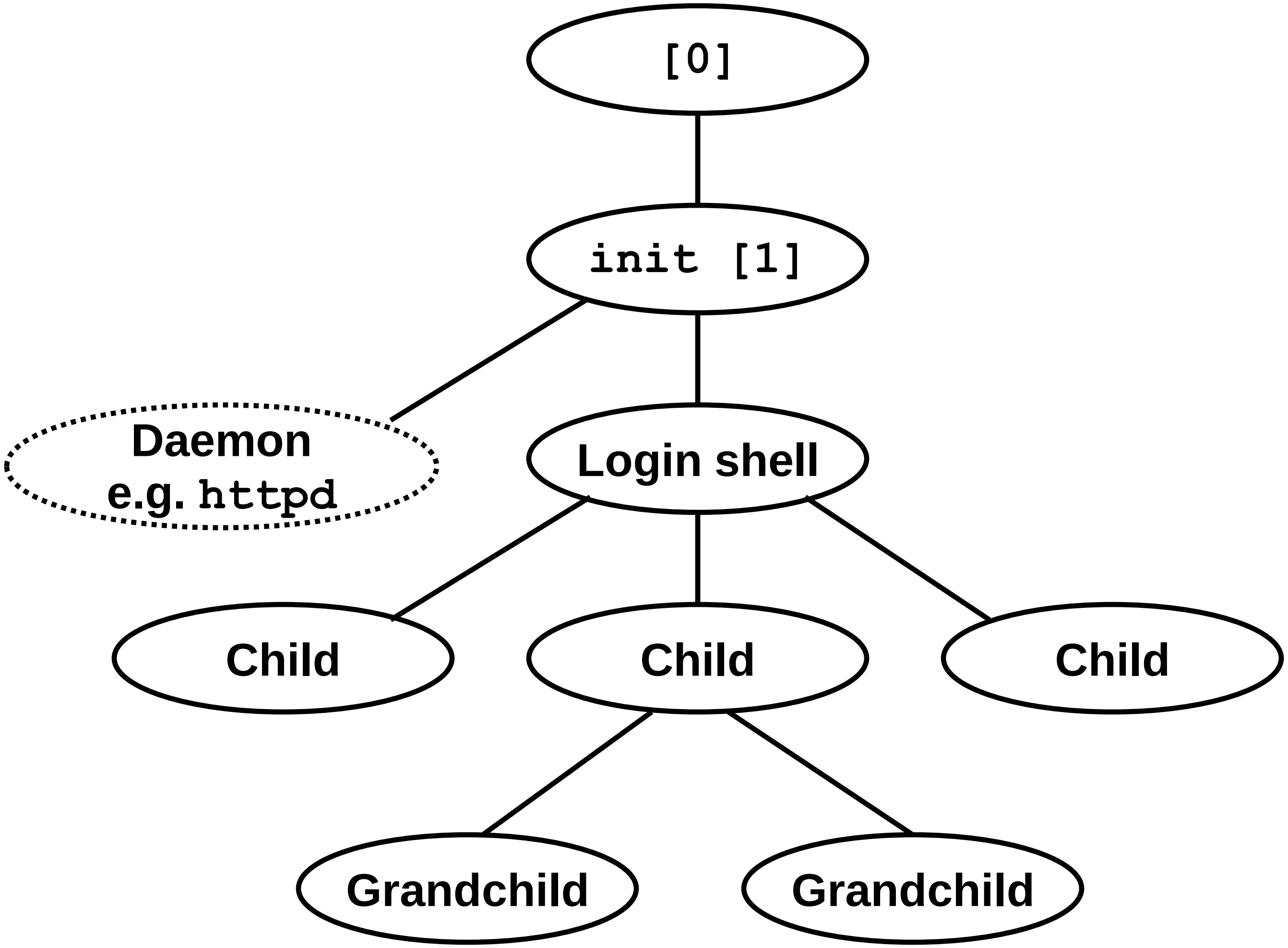
The world of multitasking

- System runs many processes concurrently
  - Process: executing program
    - State consists of memory image + register values + program counter
  - Continually 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
    - Except possibly with lower performance
    - Even though most systems can only execute one at a time

Programmer's model of multitasking

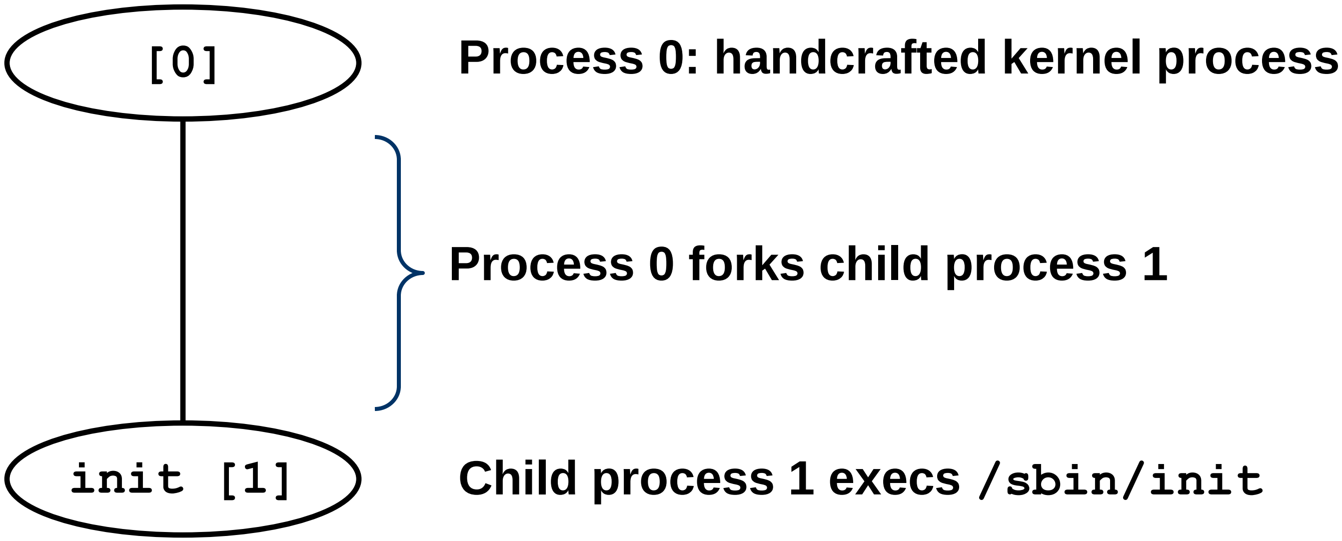
- Basic functions
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts it into "zombie" status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run a new program in an 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

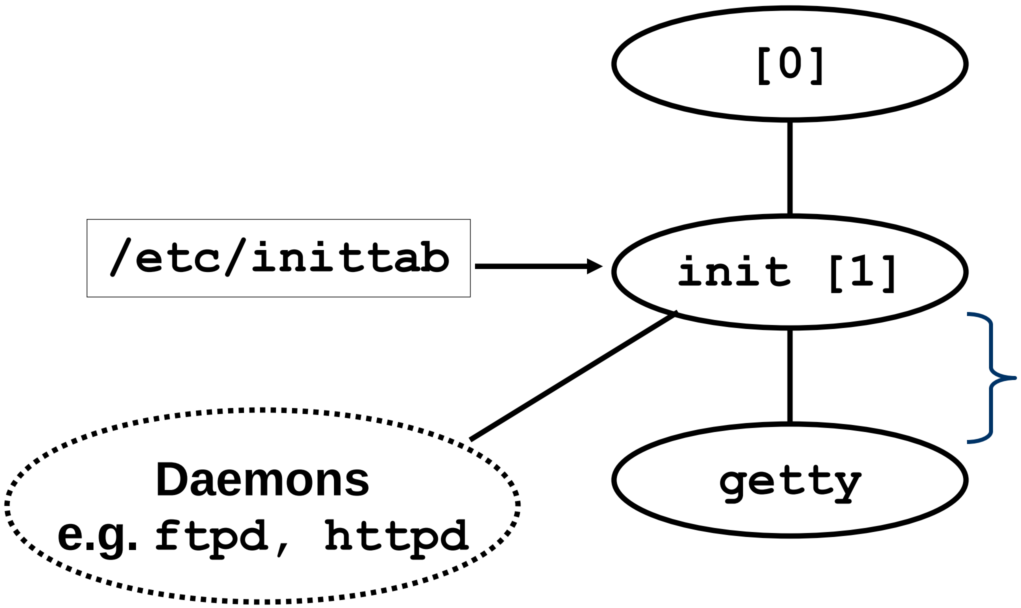


Unix startup: Step 1

1. **Pushing reset button loads the `PC` with the address of a small bootstrap program.**
2. **Bootstrap program loads the boot block (disk block 0).**
3. **Boot block program loads kernel binary (e.g., `/boot/vmlinux`)**
4. **Boot block program passes control to kernel.**
5. **Kernel handcrafts the data structures for process 0.**

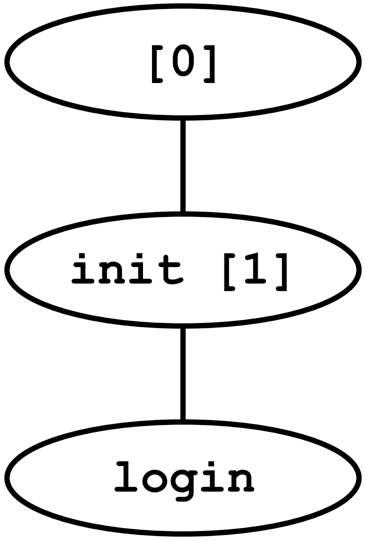


Unix startup: Step 2



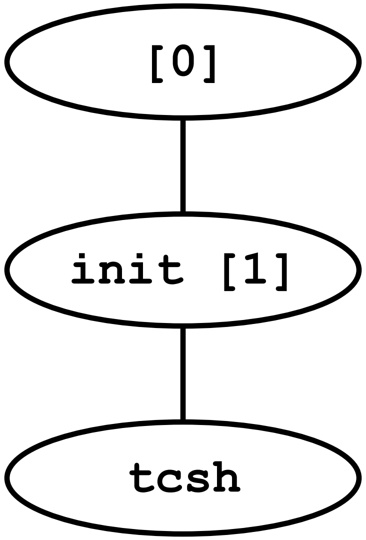
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

Unix startup: Step 3



The `getty` process execs a `login` program

Unix startup: Step 4



`login` reads login and passwd.
if OK, it execs a *shell.*
if not OK, it execs another `getty`

Shell programs

- A *shell* is an application program that runs programs on behalf of the user.
  - `sh` – Original Unix Bourne Shell
  - `csh` – BSD Unix C Shell
  - `tcsh` – Enhanced C Shell
  - `bash` –Bourne-Again Shell

```
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);
    }
}
```

Problem with simple shell 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.
  - Creates a memory leak that will eventually crash the kernel when it runs out of memory.
- Solution: Reaping background jobs requires a mechanism 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
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived

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

Signal concepts – sending

- 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

- 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.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process.
    - *Catch* the signal by executing a user-level function called a signal handler.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Signal concepts – pending

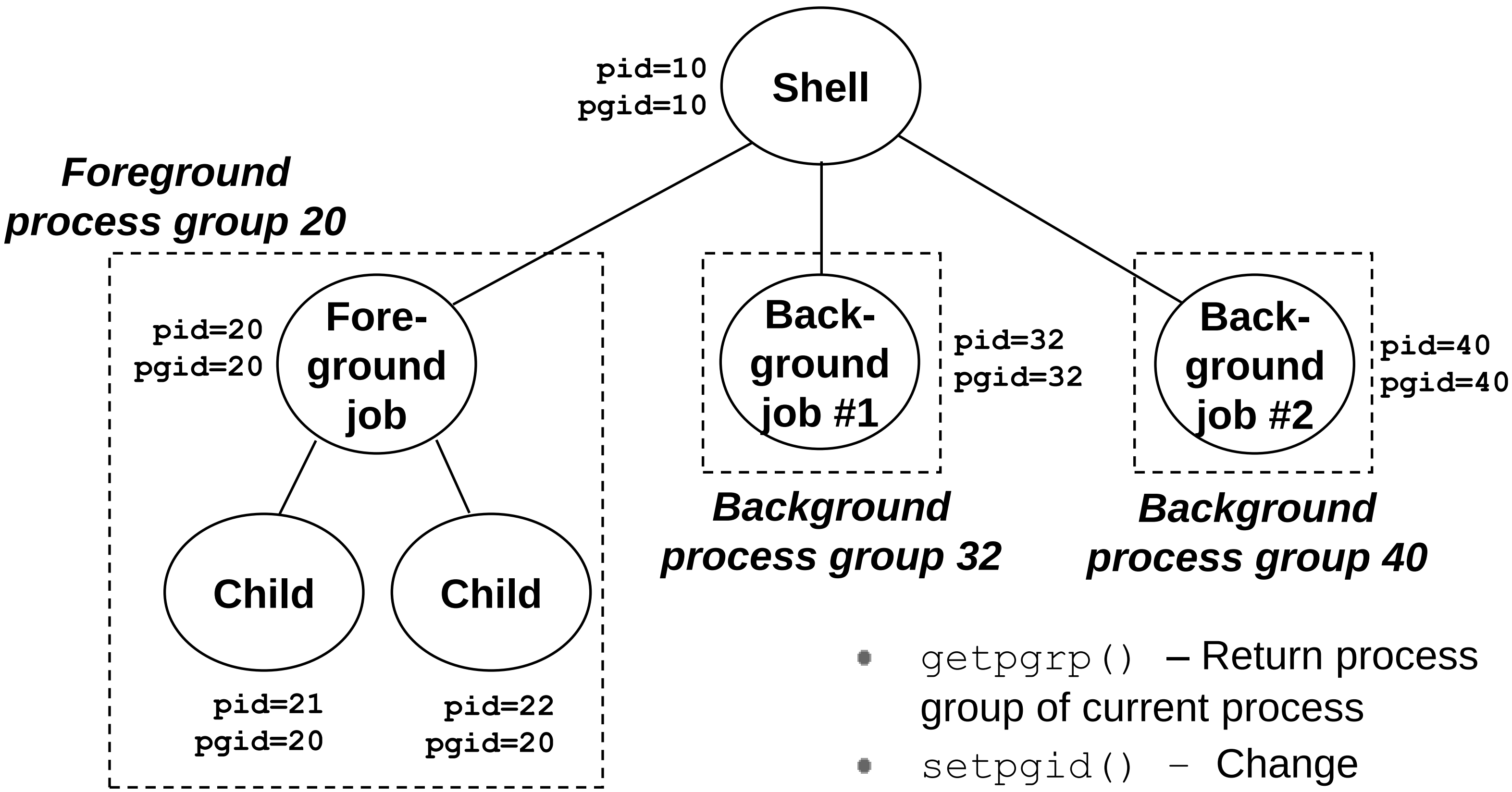
- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process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

- 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` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - `blocked` – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

Process groups

- All mechanisms for sending signals to processes rely on the notion of process group
- 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 `kill` program

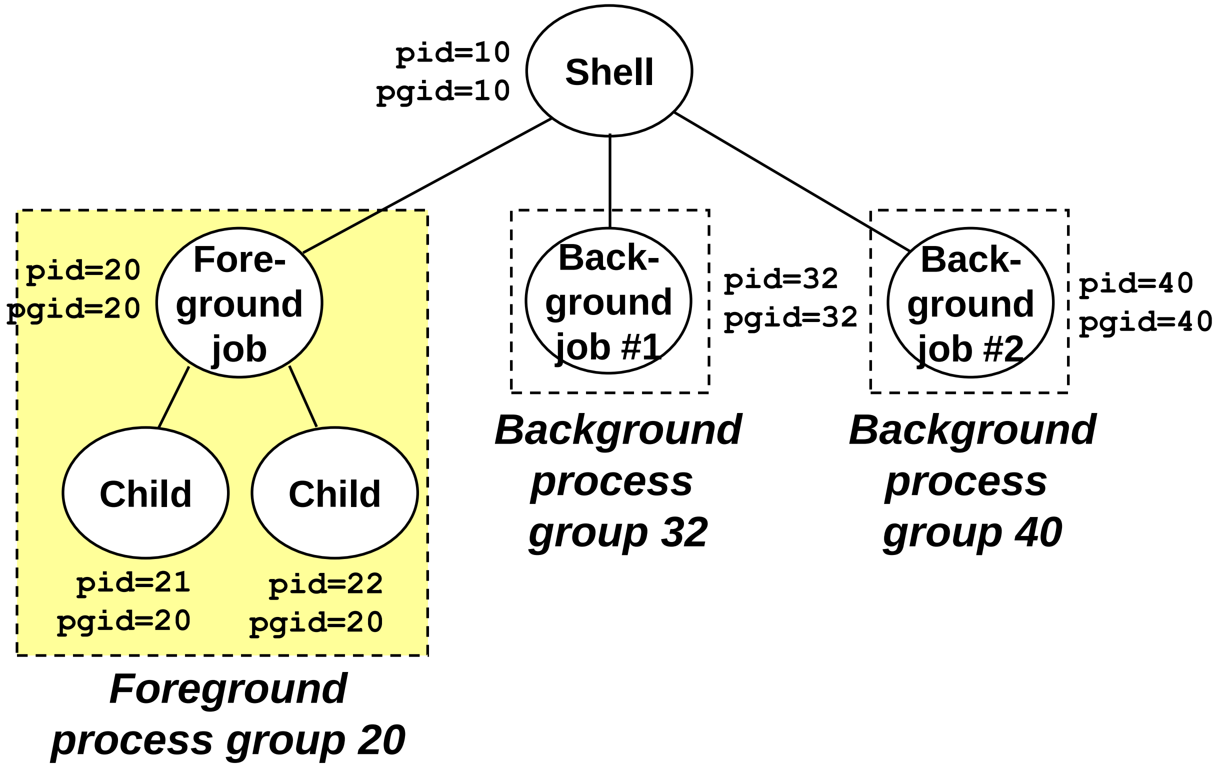
- `kill` program sends arbitrary signal to a process or process group
- Examples
  - `kill -9 24818`
    - Send SIGKILL to process 24818
  - `kill -9 -24817`
    - Send SIGKILL to every process in process group 24817.

```
linux> ./forks 16
linux> 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> 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 keyboard

- 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`

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
 <typed ctrl-z>
Suspended
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24867 pts/2    T      0:01 ./forks 17
24868 pts/2    T      0:01 ./forks 17
24869 pts/2    R      0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24870 pts/2    R      0:00 ps a
```

Sending signals with `kill` function

```
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);
    }
}
```

```
linux % ./fork12
Killing process 578
Killing process 579
Killing process 580
Killing process 581
Killing process 582
Child 578 terminated abnormally
Child 580 terminated abnormally
Child 582 terminated abnormally
Child 581 terminated abnormally
Child 579 terminated abnormally
```

Receiving signals

- Suppose kernel is returning from 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

- `Signal` 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 “*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

```
void int_handler(int sig)
{
    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);

    . . .
}
```

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```

Signal handler funkiness

```
int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler1) == SIG_ERR)
        unix_error("signal error");

    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

Parent installs SIGCHLD handler

… and create three children

Each child says “hi”, sleeps for 1sec and leaves

Parent waits for an input and process it; modeled as an infinite loop

Signal handler funkiness

```
void handler1(int sig)
{
    pid_t pid;

    if ((pid = waitpid(-1, NULL, 0)) < 0)
        unix_error("waitpid error");
    printf("Handler reaped child %d\n", (int)pid);
    Sleep(2);
    return;
}
```

```
linux> ./signal1
Hello from child 2916
Hello from child 2917
Hello from child 2918
Handler reaped child 2916
Handler reaped child 2917
<cr>
Parent processing input
^Z
[1]+  Stopped                    ./signal1
linux> ps
  PID TTY          TIME CMD
 2235 pts/2    00:00:00 bash
 2915 pts/2    00:01:05 signal1
 2918 pts/2    00:00:00 signal1 <defunct>
 2921 pts/2    00:00:00 ps
```

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

```
void handler2(int sig)
{
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}
```

Must check all terminated jobs – typically loop with wait

```
linux> ./signal2
Hello from child 2983
Hello from child 2984
Hello from child 2985
Handler reaped child 2983
Handler reaped child 2984
Handler reaped child 2985
<cr>
Parent processing input
```

OK, no zombies left; still, there is a portability problem there …

Slow system calls interrupted

```
int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler1) == SIG_ERR)
        unix_error("signal error");

    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

In some systems (Solaris) slow system calls like read are not restarted automatically after interrupted

Slow system calls interrupted

```
int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler1) == SIG_ERR)
        unix_error("signal error");

    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

In some systems (Solaris) slow system calls like read are not restarted automatically after interrupted

Signal handler funkiness

```
int main() {
    int i, n;
    char buf[MAXBUF];
    pid_t pid;

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");

    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        pid = Fork();
        if (pid == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Manually restart the read call if it is interrupted */
    while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        if (errno != EINTR)
            unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

EINTR return code indicates read returned prematurely after interrupted

External event handling

- A program that reacts to externally generated events (ctrl-c)

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

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

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

Internal event handling

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  printf("BEEP\n");
  fflush(stdout);

  if (++beeps < 5)
    alarm(1);
  else {
    printf("BOOM!\n");
    exit(0);
  }
}
```

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

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

```
linux> a.out
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 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 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

- An typical application – return from a deeply nested function call when detecting an error

```
#include <setjmp.h>
jmp_buf buf;

main()
{
   int rc;
   rc = setjmp(buf);
   if (rc == 0) /* First time through */
     p1(); /* p1 calls p2, which calls p3 */
   else if (rc == 1) {
     printf("back in main, from p3, due to an error\n");
   else
     ...

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

Putting it all together

- Another use – not returning from a handler to the interrupted instruction but to another specific location
- 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");
  }
}
```

```
bass> a.out
starting
processing...
processing...
restarting          Ctrl-c
processing...
processing...
processing...
restarting          Ctrl-c
processing...
restarting          Ctrl-c
processing...
processing...
```

Limitations of nonlocal jumps

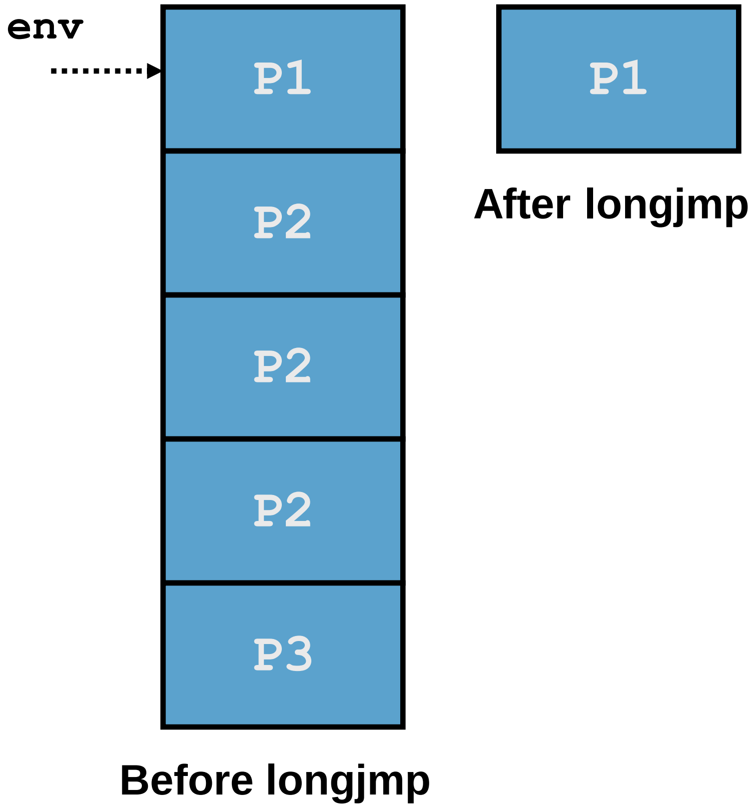
- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```
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.)

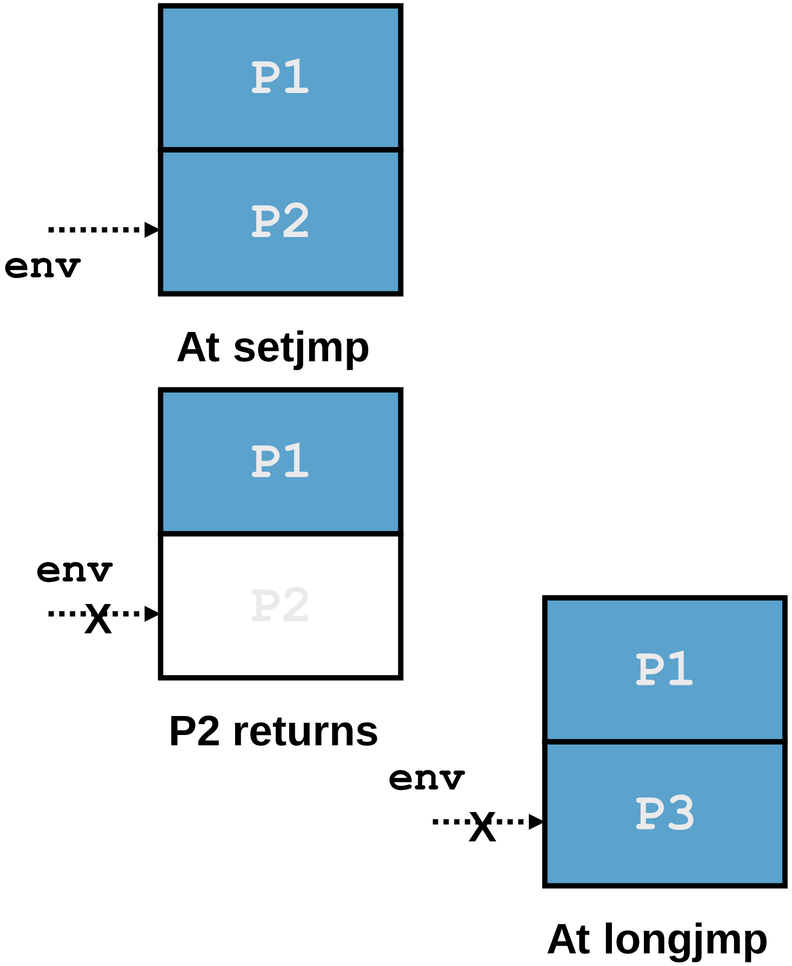
- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

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

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

P3()
{
  longjmp(env, 1);
}
```



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