Processes & Signals

CS 475

This lecture is based on Chapter 8 of Computer Systems:
A Programmer's Perspective (Bryant & O'Halloran)

Processes

Def: A *process* is an instance of a running program.
- **One of the most profound ideas in computer science.**
- **Not the same as "program" or "processor"**

**Process provides each program with two key abstractions:**
- **Logical control flow**
  - **Each program seems to have exclusive use of the CPU.**
- **Private address space**
  - **Each program seems to have exclusive use of main memory.**

How are these Illusions maintained?
- **Process executions interleaved (multitasking)**
- **Address spaces managed by virtual memory system**

Logical Control Flows
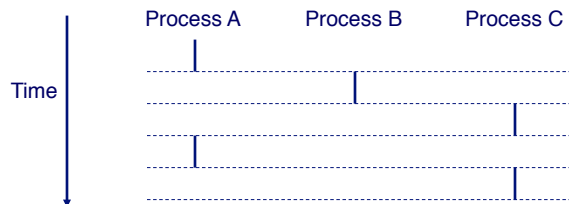
Each process has its own logical control flow

Process A   Process B   Process C
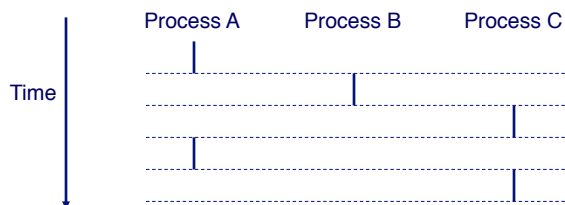
Time

– 3 –   CS 475

Concurrent Processes

Two processes *run concurrently (are concurrent)* if their flows overlap in time.

Otherwise, they are *sequential.*

Examples:

- **Concurrent: A & B, A & C**
- **Sequential: B & C**

Process A   Process B   Process C

Time

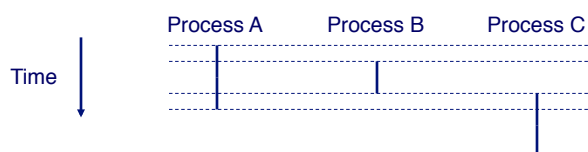– 4 –   CS 475

1/24/10

User View of Concurrent Processes

Control flows for concurrent processes are physically disjoint in time.

However, we can think of concurrent processes are running in parallel with each other.
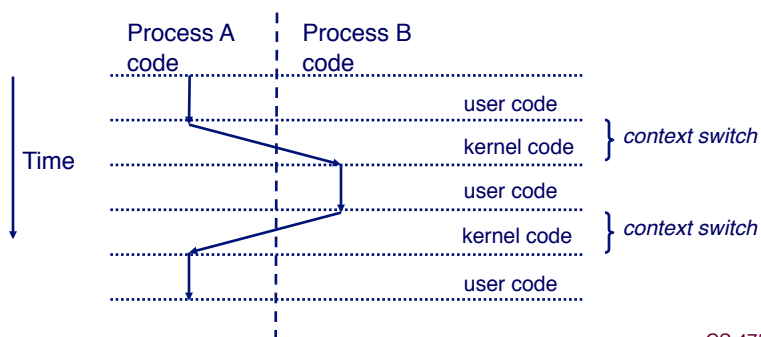

CS 475

Context Switching

Processes are managed by a shared chunk of OS code called the *kernel*

- **Important: the kernel is not a separate process, but rather runs as part of some user process**

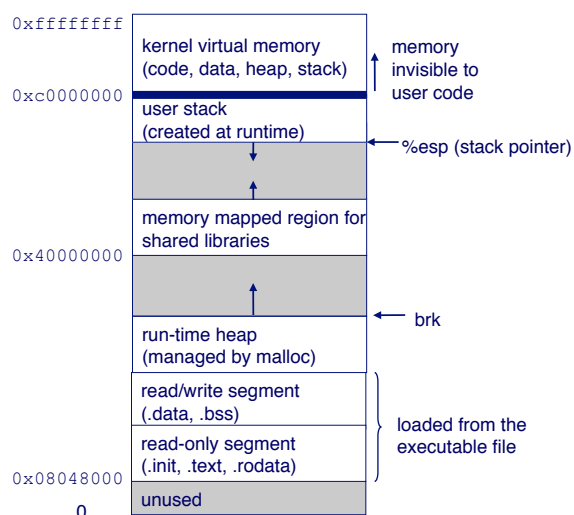**Control flow passes from one process to another via a *context switch*.**

– 6 –   CS 475

3

Private Address Spaces

Each process has its own private address space.

```
0xffffffff
              kernel virtual memory          memory
              (code, data, heap, stack)      invisible to
0xc0000000                                   user code
              user stack
              (created at runtime)           ←─ %esp (stack pointer)


              memory mapped region for
              shared libraries
0x40000000

                                             ←─ brk
              run-time heap
              (managed by malloc)
              read/write segment
              (.data, .bss)                  } loaded from the
              read-only segment                executable file
              (.init, .text, .rodata)
0x08048000
              unused
        0
```

– 7 –   CS 475

`fork`: Creating new processes

int fork(void)

- **creates a new process (child process) that is identical to the calling process (parent process)**
- **returns 0 to the child process**
- **returns child's `pid` to the parent process**

```
if (fork() == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

– 8 –   CS 475

Fork Example #1

Key Points

- **Parent and child both run same code**
  - ● **Distinguish parent from child by return value from `fork`**
- **Start with same state, but each has private copy**
  - ● **Including shared output file descriptor**
  - ● **Relative ordering of their print statements undefined**

```c
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
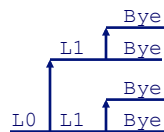
– 9 –   CS 475

Fork Example #2

Key Points

- **Both parent and child can continue forking**

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
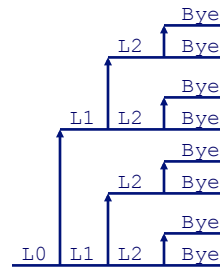

– 10 –   CS 475

5

Fork Example #3

Key Points

- **Both parent and child can continue forking**

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
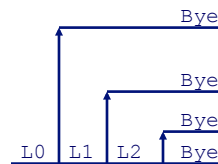
```
                                  Bye
                          L2 │ Bye
                          ↑
                                  Bye
                  L1 │ L2 │ Bye
                  ↑
                                  Bye
                          L2 │ Bye
                          ↑
                                  Bye
          L0 │ L1 │ L2 │ Bye
```

– 11 –   CS 475

Fork Example #4

Key Points

- **Both parent and child can continue forking**

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
    printf("L1\n");
    if (fork() != 0) {
        printf("L2\n");
        fork();
    }
    }
    printf("Bye\n");
}
```

```
                          Bye
          ↑
                          Bye
                  ↑
                          Bye
          L0 │ L1 │ L2 │ Bye
```

– 12 –   CS 475

6

Fork Example #5

Key Points

- **Both parent and child can continue forking**

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
   printf("L1\n");
   if (fork() == 0) {
        printf("L2\n");
        fork();
   }
    }
    printf("Bye\n");
}
```
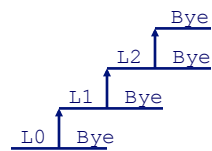
```
                                      Bye
                               L2    Bye
                         L1    Bye
                   L0    Bye
```

– 13 –   CS 475

exit: Destroying Process

void exit(int status)

- **exits a process**
  - **Normally return with status 0**
- **atexit() registers functions to be executed upon exit**

```
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

– 14 –   CS 475

7

Zombies

Idea

- **When process terminates, still consumes system resources**
  - ● **Various tables maintained by OS**
- **Called a "zombie"**
  - ● **Living corpse, half alive and half dead**

Reaping

- **Performed by parent on terminated child**
- **Parent is given exit status information**
- **Kernel discards process**

What if Parent Doesn't Reap?

- **If any parent terminates without reaping a child, then child will be reaped by `init` process**
- **Only need explicit reaping for long-running processes**
  - ● **E.g., shells and servers**

– 15 –   CS 475

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
    /* Child */
    printf("Terminating Child, PID = %d\n",
            getpid());
    exit(0);
    } else {
    printf("Running Parent, PID = %d\n",
            getpid());
    while (1)
        ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **`ps` shows child process as "defunct"**
- **Killing parent allows child to be reaped**

– 16 –   CS 475

8

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
    /* Child */
    printf("Running Child, PID = %d\n",
           getpid());
    while (1)
        ; /* Infinite loop */
    } else {
    printf("Terminating Parent, PID = %d\n",
           getpid());
    exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

– 17 –   CS 475

`wait:` Synchronizing with children

int wait(int *child_status)
- suspends current process until one of its children terminates
- return value is the **pid** of the child process that terminated
- if **child_status != NULL**, then the object it points to will be set to a status indicating why the child process terminated

– 18 –   CS 475

9

wait: Synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```

HC Bye

HP   CT Bye

– 19 –   CS 475

Wait Example

- **If multiple children completed, will take in arbitrary order**
- **Can use macros WIFEXITED and WEXITSTATUS to get information about exit status**

```
void fork10()
{
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++)
   if ((pid[i] = fork()) == 0)
      exit(100+i); /* Child */
   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 terminate abnormally\n", wpid);
   }
}
```

Waitpid

- **waitpid(pid, &status, options)**
  - **Can wait for specific process**
  - **Various options**

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
    pid_t wpid = waitpid(pid[i], &child_status, 0);
    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);
    }
```

– 21 –   CS 475

Wait/Waitpid Example Outputs

Using `wait` (`fork10`)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using `waitpid` (`fork11`)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

– 22 –   CS 475

exec: Running new programs

int execl(char *path, char *arg0, char *arg1, ..., 0)

- **loads and runs executable at `path` with args `arg0`, `arg1`, ...**
  - **`path` is the complete path of an executable**
  - **`arg0` becomes the name of the process**
    - » **typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`**
  - **"real" arguments to the executable start with `arg1`, etc.**
  - **list of args is terminated by a `(char *)0` argument**
- **returns `-1` if error, otherwise doesn't return!**

```
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

– 23 –   CS 475

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**
  - **Even though most systems can only execute one process at a time**
  - **Except possibly with lower performance than if running alone**

– 24 –   CS 475

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**
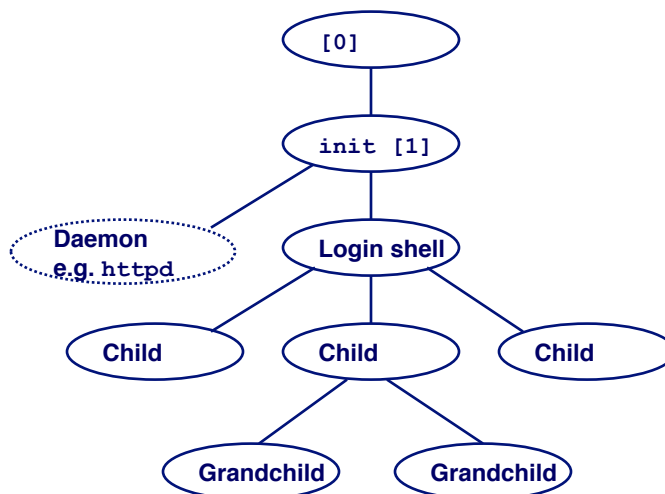 - Called once, (normally) never returns

Programming Challenge

- **Understanding the nonstandard semantics of the functions**
- **Avoiding improper use of system resources**
  - E.g. "Fork bombs" can disable a system.

– 25 –   CS 475

Unix Process Hierarchy

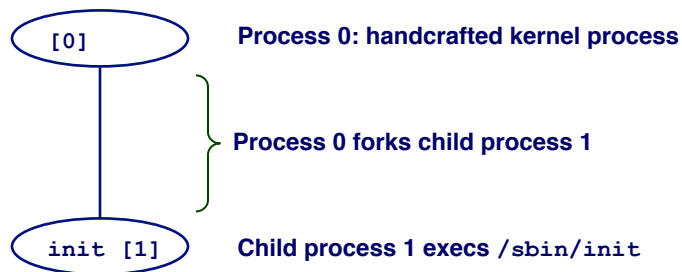– 26 –   CS 475
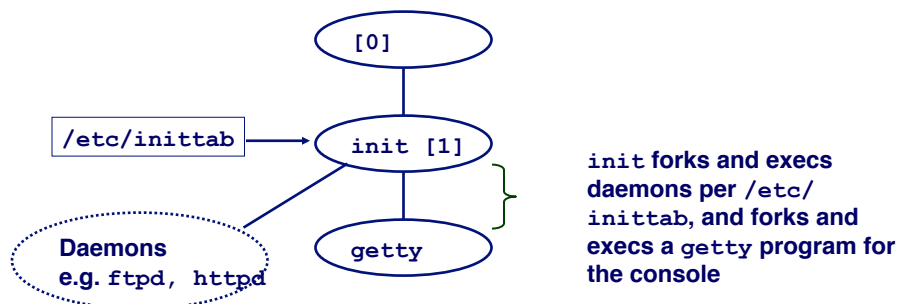
Unix Startup: Step 1

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.

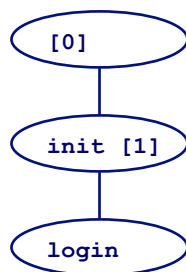[0] — Process 0: handcrafted kernel process

Process 0 forks child process 1

init [1] — Child process 1 execs `/sbin/init`

– 27 –   CS 475

Unix Startup: Step 2

[0]

`/etc/inittab` → init [1]

Daemons
e.g. `ftpd, httpd`

getty

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

– 28 –   CS 475

14

1/24/10

Unix Startup: Step 3

[0]

init [1]

login

The `getty` **process execs a** `login` **program**

– 29 –

CS 475

Unix Startup: Step 4

[0]

init [1]
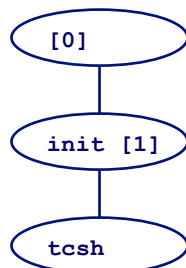
tcsh

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

– 30 –

CS 475

15

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

– 31 – CS 475

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);
    }
}
```

– 32 – CS 475

16

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*.

CS 475

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

CS 475

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 (cont)

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 (cont)

A signal is *pending* if it has been 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

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

Every process belongs to exactly one process group

```
pid=10
pgid=10    Shell
```

```
pid=20     Fore-          Back-          Back-         pid=40
pgid=20    ground         ground         ground        pgid=40
           job            job #1         job #2
                          pid=32
                          pgid=32
```

```
Child    Child
pid=21   pid=22
pgid=20  pgid=20
```

Background process group 32

Background process group 40

Foreground process group 20

`getpgrp()` – **Return process group of current process**

`setpgid()` – **Change process group of a process**
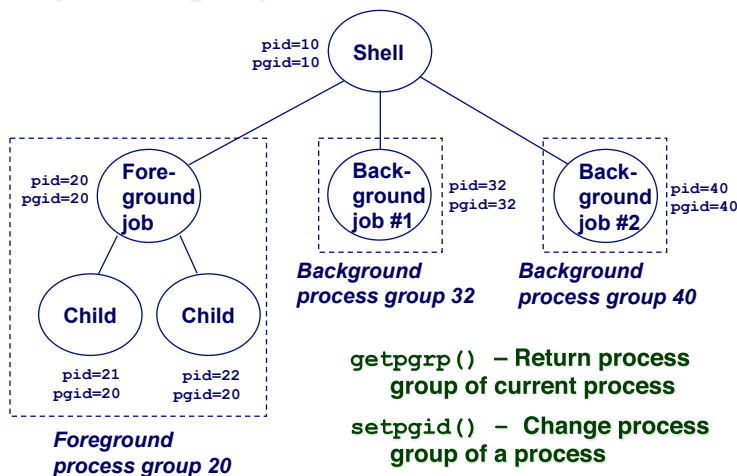
– 39 –   CS 475

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>
```
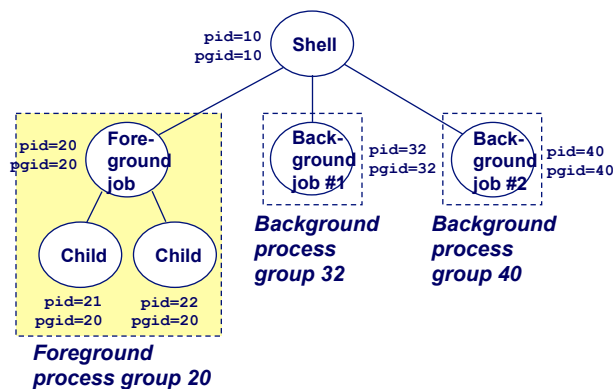
– 40 –   CS 475

20

Sending Signals from the Keyboard

Typing ctrl-c (ctrl-z) sends a SIGTERM (SIGTSTP) to every job in the foreground process group.

- **SIGTERM – default action is to terminate each process**
- **SIGTSTP – default action is to stop (suspend) each process**

```
                              pid=10    ┌──────┐
                              pgid=10   │ Shell│
                                        └──────┘

          ┌ ─ ─ ─ ─ ─ ─ ─ ─ ┐    ┌ ─ ─ ─ ─ ┐   ┌ ─ ─ ─ ─ ┐
                    ┌──────┐         ┌──────┐       ┌──────┐
 pid=20            │ Fore-│       │ Back-│ pid=32 │ Back-│ pid=40
 pgid=20          │ground│       │ground│ pgid=32│ground│ pgid=40
          │        │ job  │  │    │job #1│   │    │job #2│   │
                    └──────┘         └──────┘       └──────┘
          │                    │  └ ─ ─ ─ ─ ┘   └ ─ ─ ─ ─ ┘
               ┌──────┐ ┌──────┐    Background    Background
          │    │ Child│ │ Child│ │  process       process
               └──────┘ └──────┘    group 32       group 40
          │   pid=21   pid=22    │
              pgid=20   pgid=20
          └ ─ ─ ─ ─ ─ ─ ─ ─ ┘
              Foreground
              process group 20
```

– 41 –   CS 475

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
```

– 42 –   CS 475

21

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);
    }
}
```

– 43 –   CS 475

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*.

– 44 –   CS 475

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

```
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>
```

– 47 –   CS 475

Signal Handler Funkiness

Pending signals are not queued

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    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) {
        /* Child: Exit */
        exit(0);
    }
    while (ccount > 0)
    pause();/* Suspend until signal occurs */
}
```

- **For each signal type, just have single bit indicating whether or not signal is pending**
- **Even if multiple processes have sent this signal**

– 48 –   CS 475

24

Living With Nonqueuing Signals

Must check for all terminated jobs

- **Typically loop with `waitpid`**

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

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

– 49 –   CS 475

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) {
  }
}
```

– 50 –   CS 475

25

A Program That Reacts to Internally Generated Events

```
#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>
```

– 51 –   CS 475

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

– 52 –   CS 475

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`.**

– 53 –   CS 475

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)
}
```

– 54 –   CS 475

27

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");
```

```c
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...
```

– 55 –   CS 475

Limitations of Nonlocal Jumps

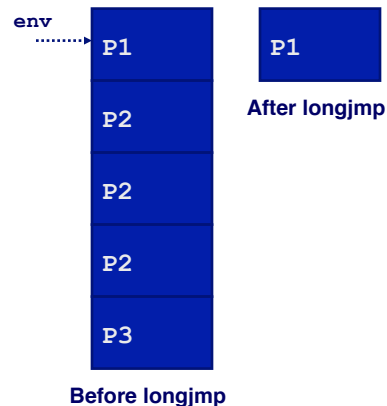### Works within stack discipline

- 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);
}
```

env ········▸

P1
P2
P2
P2
P3

Before longjmp

P1

After longjmp

– 56 –   CS 475

28

Limitations of Long Jumps (cont.)
Works within stack discipline

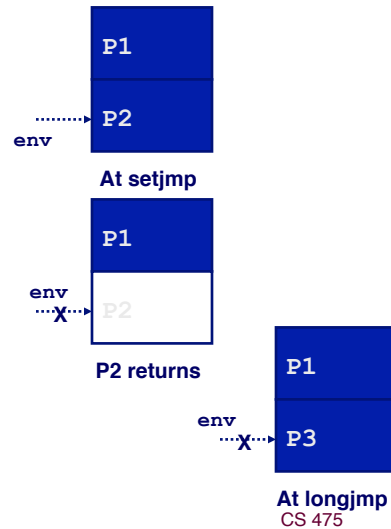- Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

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

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

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

P1

env ········► P2

At setjmp

P1

env ····X···· P2

P2 returns

P1

env ····X···► P3

At longjmp

– 57 –
CS 475

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

– 58 –
CS 475

29

Summarizing (cont.)

Spawning Processes

- **Call to `fork`**
  - One call, two returns

Terminating Processes

- **Call `exit`**
  - One call, no return

Reaping Processes

- **Call `wait` or `waitpid`**

Replacing Program Executed by Process

- **Call `execl` (or variant)**
  - One call, (normally) no return

CS 475