Exceptional Control Flow, Cont: Processes and Signals

B&O Readings: Chap 8
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Note: these slides were originally created in parts by Markus Püschel at Carnegie Mellon University

Modeling `fork` with Process Graphs

- **A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - Each vertex is the execution of a statement
  - a -> b means `a` happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges
- **Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - Total ordering of vertices where all edges point from left to right

fork Example: Nested forks in parent

```
void fork2()
{
    printf("L0\n");
    if (Fork() != 0) {
        printf("L1\n");
        if (Fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}                          forks.c
```
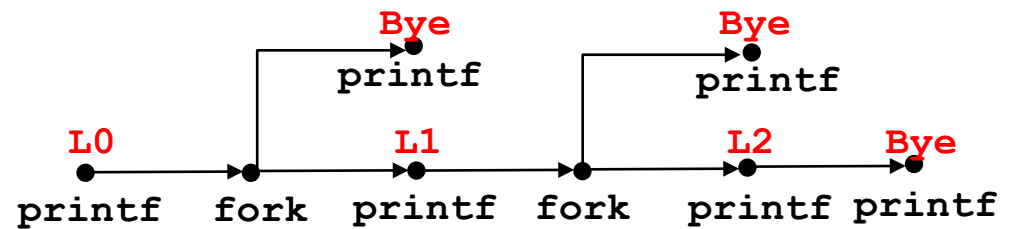

Feasible output?
L0
L1
Bye
Bye
L2
Bye

Yes

Feasible output?
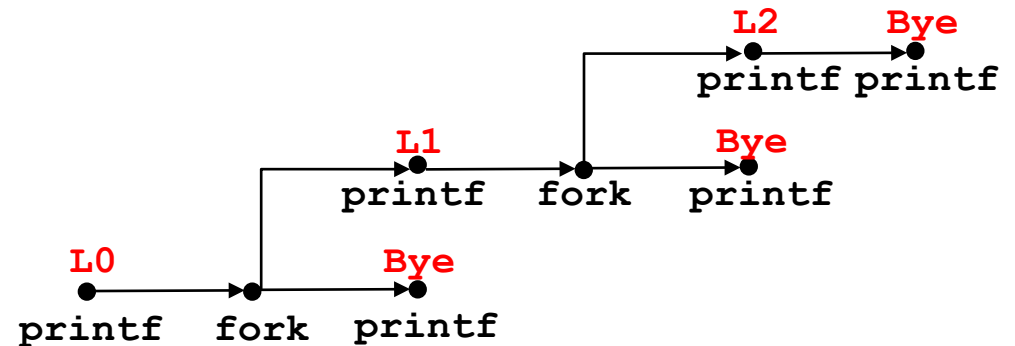L0
Bye
L1
Bye
Bye
L2

No

fork Example: Nested forks in children

```c
void fork3()
{

    printf("L0\n");
    if (Fork() == 0) {
        printf("L1\n");
        if (Fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c

Feasible output?
L0
Bye
L1
Bye
Bye
L2

No

Feasible output?
L0
Bye
L1
L2
Bye
Bye

Yes

fork Example: Two consecutive forks

```c
void fork4()
{
    int x = 0;
    printf("L0\n");
    if(Fork() == 0) x++;
    printf("L1,%d\n", x);
    Fork();
    printf("Bye\n");
}                           forks.c
```

Feasible output?	Feasible output?
L0	L0
L1,1	L1,0
Bye	Bye
Bye	L1,1
L1,0	Bye
Bye	Bye
Bye	Bye

Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a "zombie"
    - Living corpse, half alive and half dead

- **Reaping**
  - Performed by parent on terminated child to get child's exit status (using `wait` or `waitpid`)
  - Kernel then deletes zombie child process

- **What if parent doesn't reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
  - So, should reap children explicitly in long-running processes
    - e.g., shells and servers

Zombie Example

```c
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 */
    }
}
```
forks.c

```
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 -9 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" (i.e., a zombie)

- Killing parent allows child to be reaped by **init**

49

Non-terminating Child Example

```c
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);
    }
}
```
forks.c

```
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 child explicitly, or else will keep running indefinitely

50

Synchronizing with Children

- **Parent reaps a child by calling the `wait/waitpid` function**

- `pid_t wait(int *child_status)`
- `pid_t waitpid(pid_t pid, int *child_status,`
                `int option);`
  - 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 integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
    - See textbook for details

`wait`: Synchronizing with Children

```c
void fork9() {
    int child_status;

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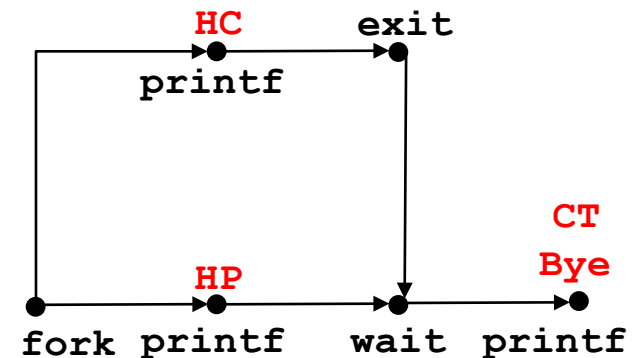
Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

Another wait Example

- If multiple children completed, will take in arbitrary order

- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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);
    }
}
                                            forks.c
```

`waitpid`: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook or man page)

```c
void fork11() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; 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 terminate abnormally\n", wpid);
    }
}
```
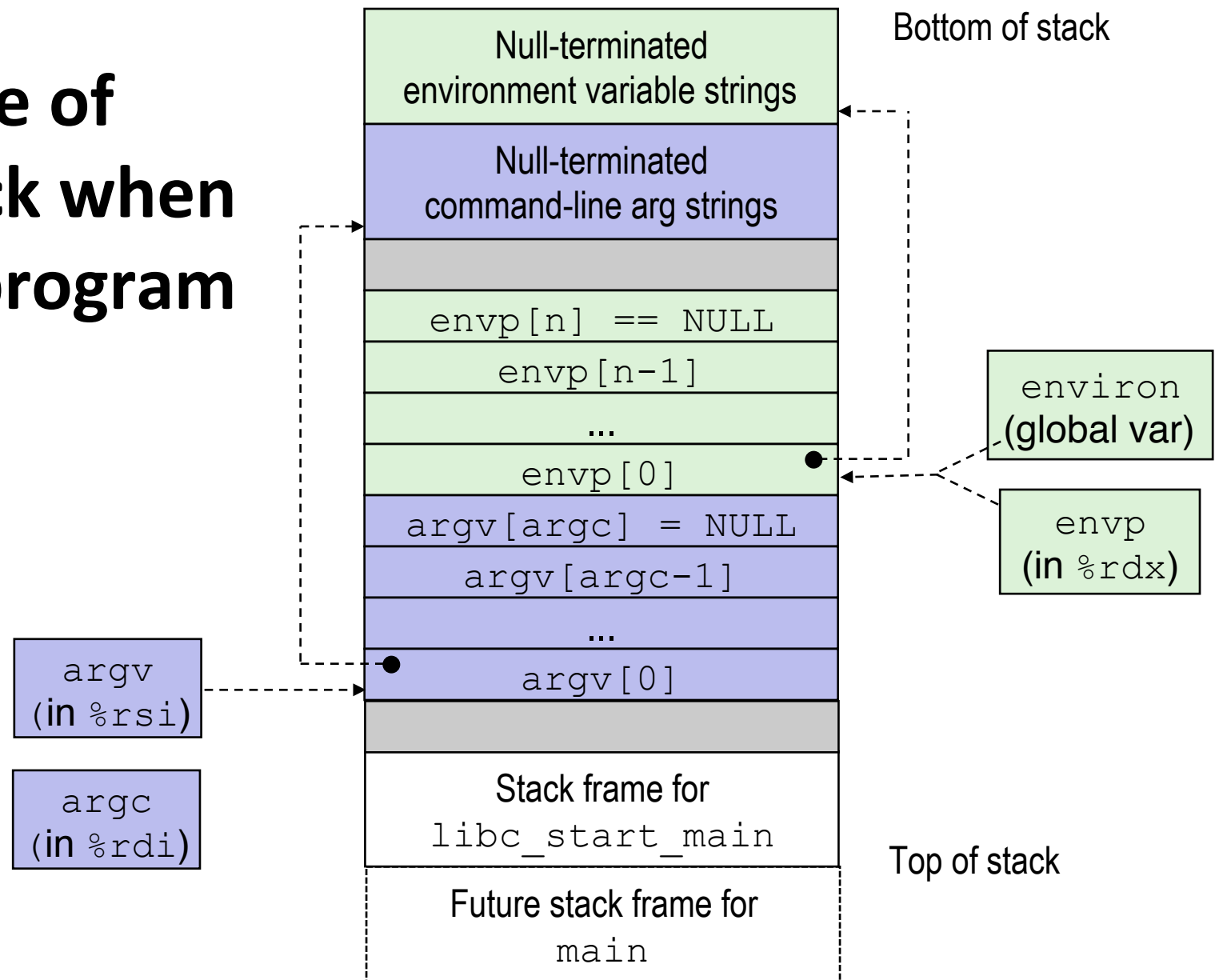
forks.c

`execve`: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
  - Executable file `filename`
    - Can be object file or script file (e.g., `#!/bin/bash`)
  - with argument list `argv`
    - By convention `argv[0]==filename`
  - and environment variable list `envp`
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

Structure of the stack when a new program starts

Null-terminated environment variable strings	Bottom of stack
Null-terminated command-line arg strings	
`envp[n] == NULL`	
`envp[n-1]`	
...	`environ` (global var)
`envp[0]`	
`argv[argc] = NULL`	`envp` (in `%rdx`)
`argv[argc-1]`	
...	
`argv[0]`	
Stack frame for `libc_start_main`	
Future stack frame for `main`	Top of stack

`argv` (in `%rsi`)

`argc` (in `%rdi`)

execve Example

- **Executes** `"/bin/ls -lt /usr/include"` **in child process using current environment:**

```
                    ┌─────────────────────────┐
                    │ myargv[argc] = NULL     │
(argc == 3)         ├─────────────────────────┤
                    │ myargv[2]               │ ───────→  "/usr/include"
                    ├─────────────────────────┤
                    │ myargv[1]               │ ───────→  "-lt"
                    ├─────────────────────────┤
myargv  ──────────→ │ myargv[0]               │ ───────→  "/bin/ls"
                    └─────────────────────────┘

                    ┌─────────────────────────┐
                    │ envp[n] = NULL          │
                    ├─────────────────────────┤
                    │ envp[n-1]               │ ───────→  "PWD=/usr/droh"
                    ├─────────────────────────┤
                    │ …                       │
                    ├─────────────────────────┤
environ  ─────────→ │ envp[0]                 │ ───────→  "USER=droh"
                    └─────────────────────────┘
```
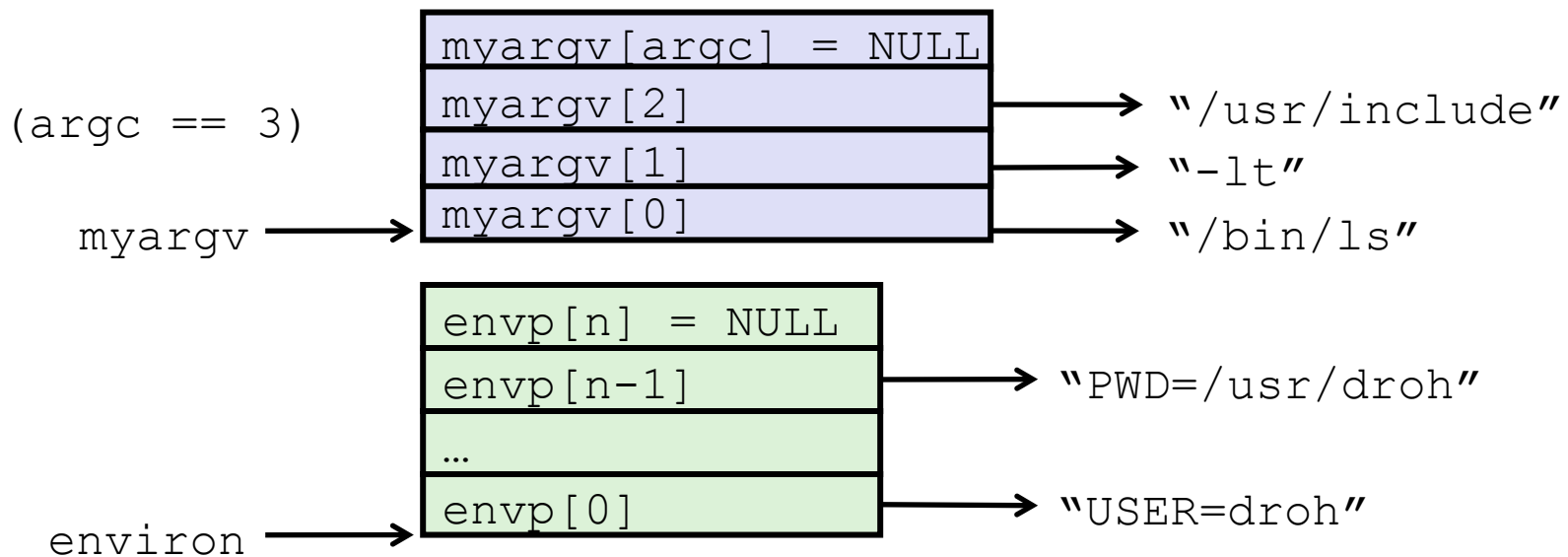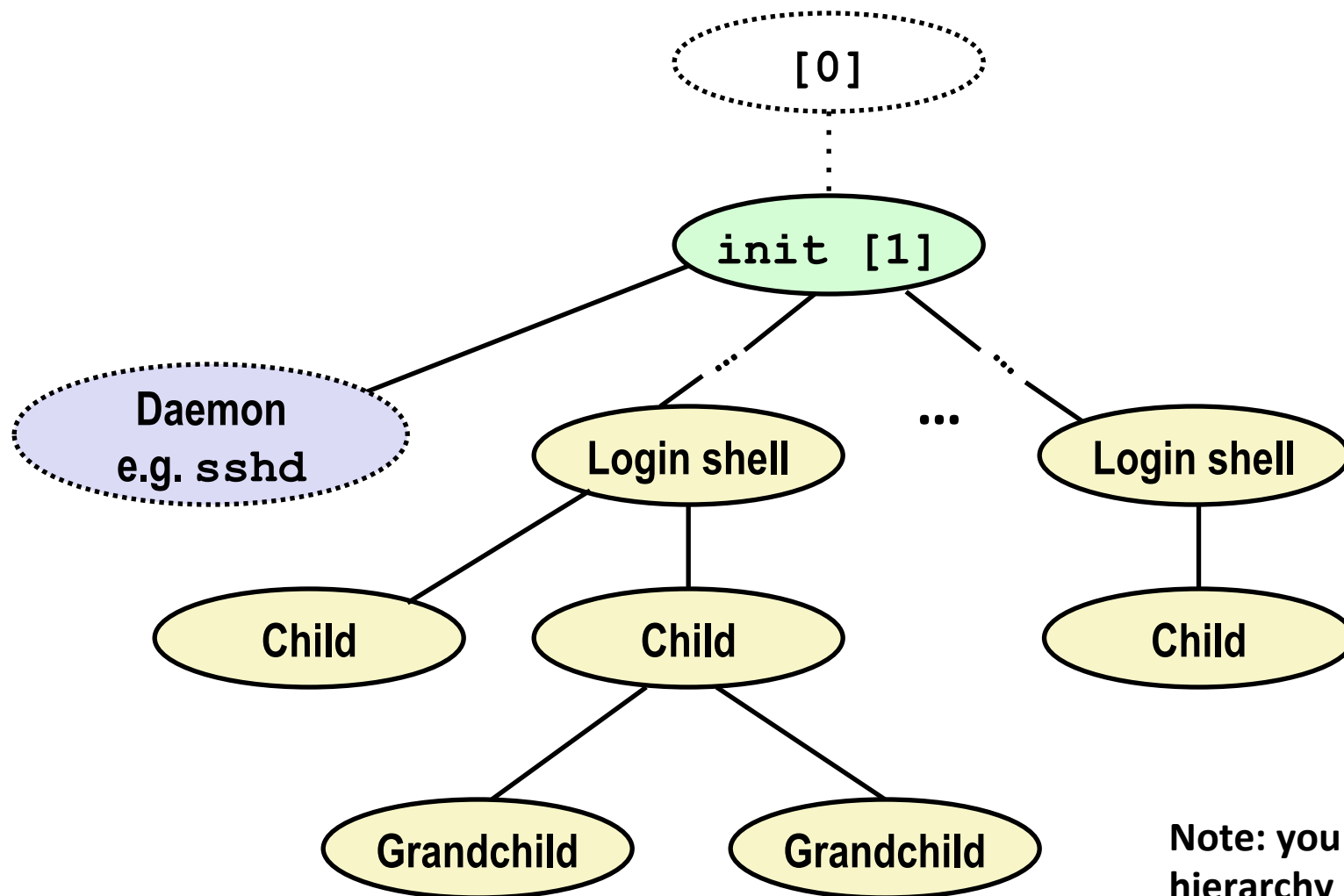
```c
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Today

- **Process**
- **Shells**
- **Signals**
- **Nonlocal jumps**

Linux Process Hierarchy

Note: you can view the hierarchy using the Linux `pstree` command

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/tcsh`   BSD Unix C shell
  - `bash`   "Bourne-Again" Shell   (default Linux shell)

```c
int main()
{
    char cmdline[MAXLINE]; /* command line */

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
                                      shellex.c
```

Execution is a sequence of read/ evaluate steps

Simple Shell `eval` Function

```c
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    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);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

shellex.c

61

Problem with Simple Shell Example

- **Our example shell correctly waits for and reaps foreground jobs**

- **But what about background jobs?**
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory

ECF to the Rescue!

- **Solution: Exceptional control flow**
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix / Linux, the alert mechanism is called a *signal*

Today

- **Process**

- **Shells**

- **Signals**

- **Nonlocal jumps**

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - Akin to exceptions and interrupts
  - Sent from the kernel to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only information in a signal is its ID and the fact that it arrived

| ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Concepts: Sending a Signal

- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process

Signal Concepts: Receiving a Signal (Kernel)

- **Suppose kernel is returning from an exception handler and is ready to pass control to process *B***
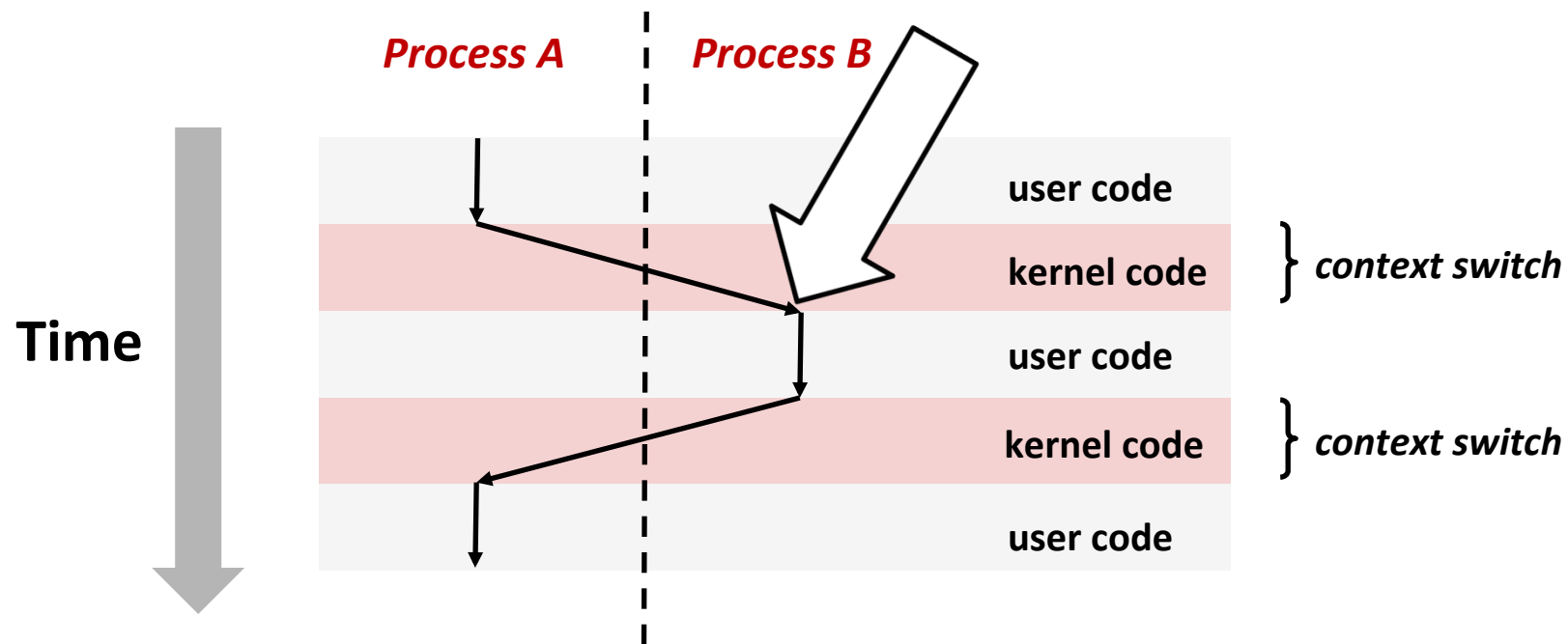
Signal Concepts: Pending and Blocked Signals

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

- **A process can *block* the receipt of certain signals**
    - Blocked signals can be delivered, but will not be received until the signal is unblocked

- **A pending signal is received at most once**

Signal Concepts: Pending/Blocked Bits

- **Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **`pending`: represents the set of pending signals**
    - Kernel sets bit k in **`pending`** when a signal of type k is delivered
    - Kernel clears bit k in **`pending`** when a signal of type k is received

  - **`blocked`: represents the set of blocked signals**
    - Can be set and cleared by using the **`sigprocmask`** function
    - Also referred to as the *signal mask*.

Signal Concepts: Receiving a Signal (Kernel)

- **Suppose kernel is returning from an exception handler and is ready to pass control to process *B***
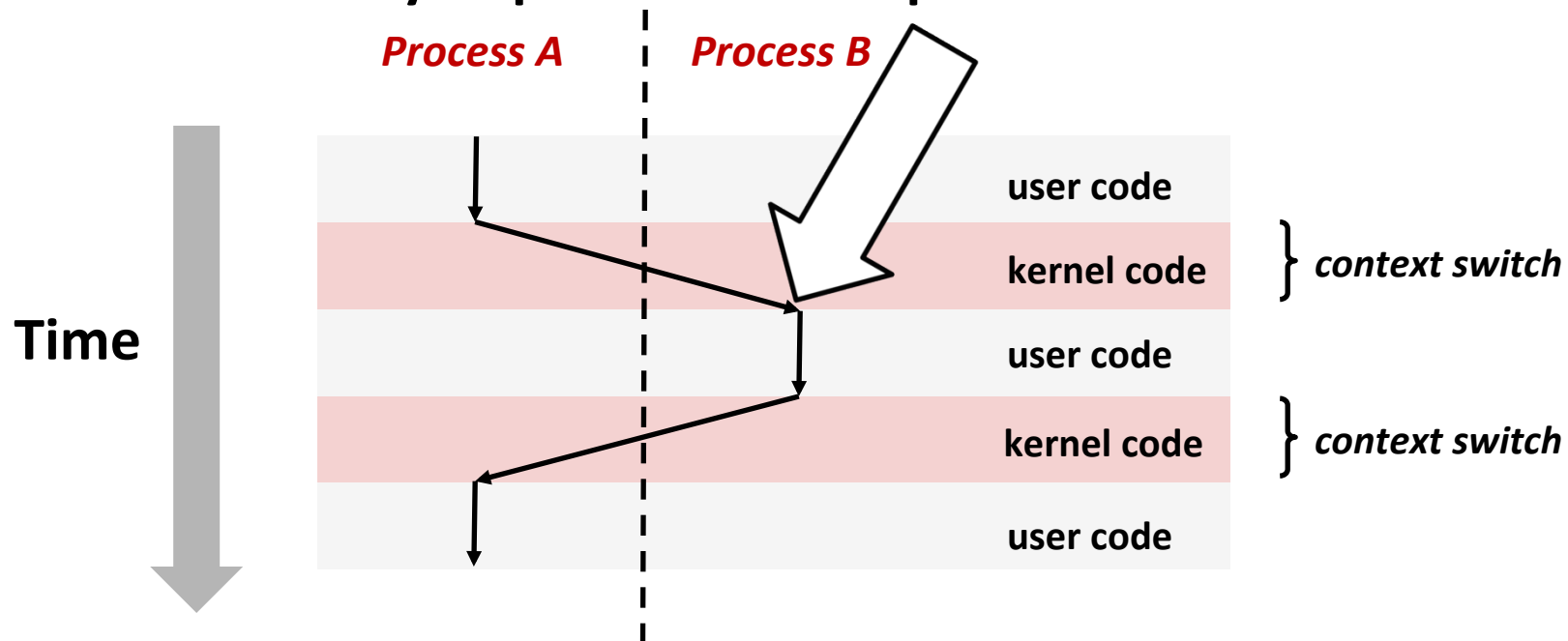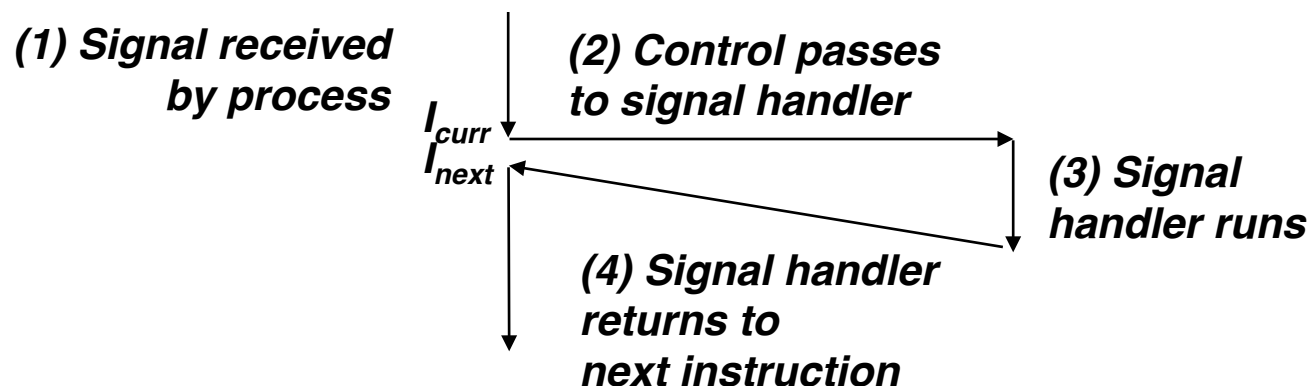

- **Kernel computes `pnb = pending & ~blocked`**
  - If (`pnb == 0`), pass control to next instruction in the logical flow for *B*
  - Else, choose least nonzero bit *k* in `pnb` and force process *p* to **receive** signal *k*

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`
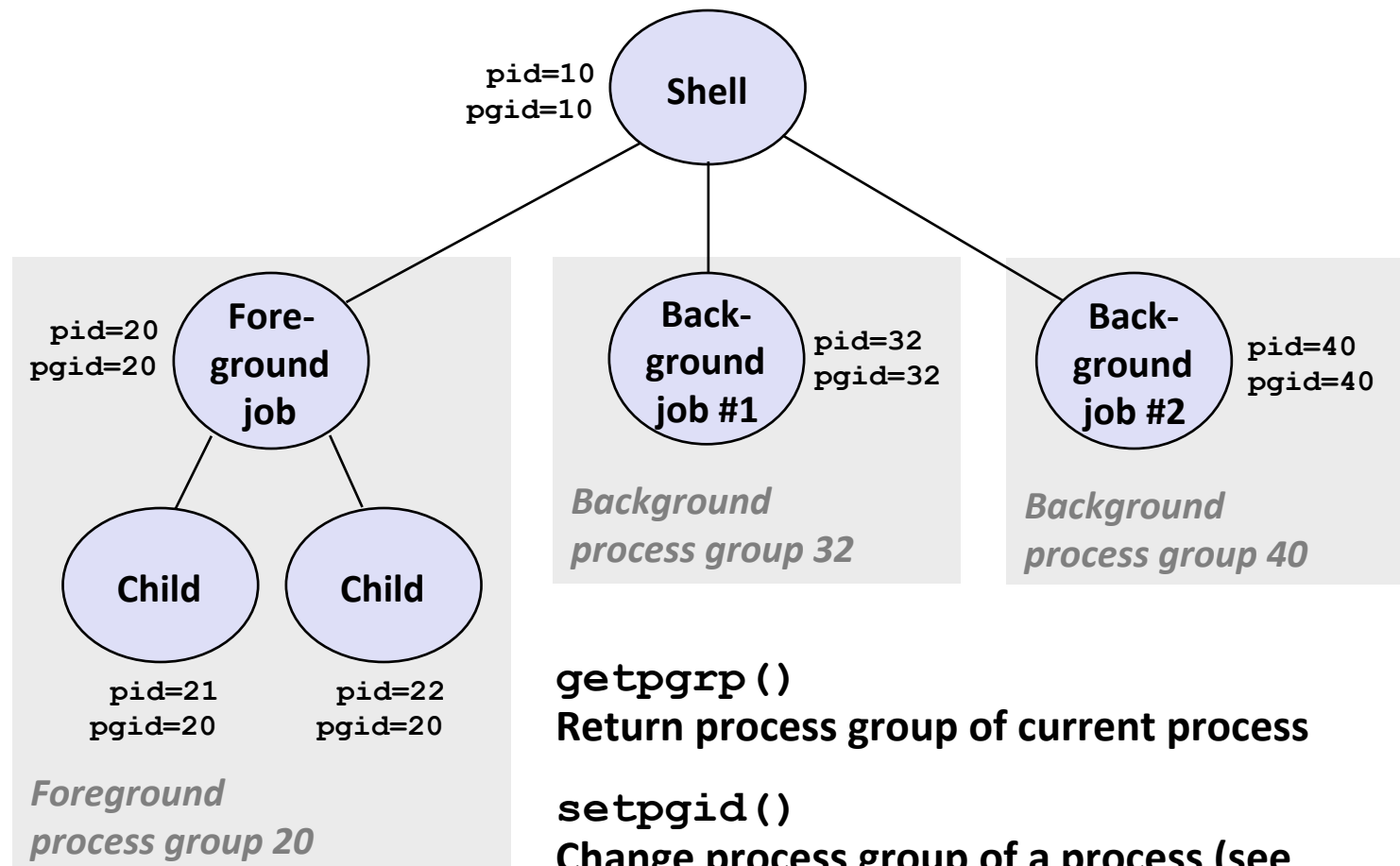 - Pass control to next instruction in logical flow for *p*

Signal Concepts: Receiving a Signal (Process)

- **A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal**
  - The receipt of the signal triggers some *action* by the destination process

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

(1) Signal received by process

I_{curr}
I_{next}

(2) Control passes to signal handler

(3) Signal handler runs

(4) Signal handler returns to next instruction

Sending Signals: Process Groups

- **Every process belongs to exactly one process group**


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

```
pid=20
pgid=20
```
Foreground job

Background job #1
```
pid=32
pgid=32
```

Background job #2
```
pid=40
pgid=40
```

Child

Child

```
pid=21
pgid=20
```

```
pid=22
pgid=20
```

Foreground process group 20

Background process group 32

Background process group 40

`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process (see text for details)

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
linux>
```
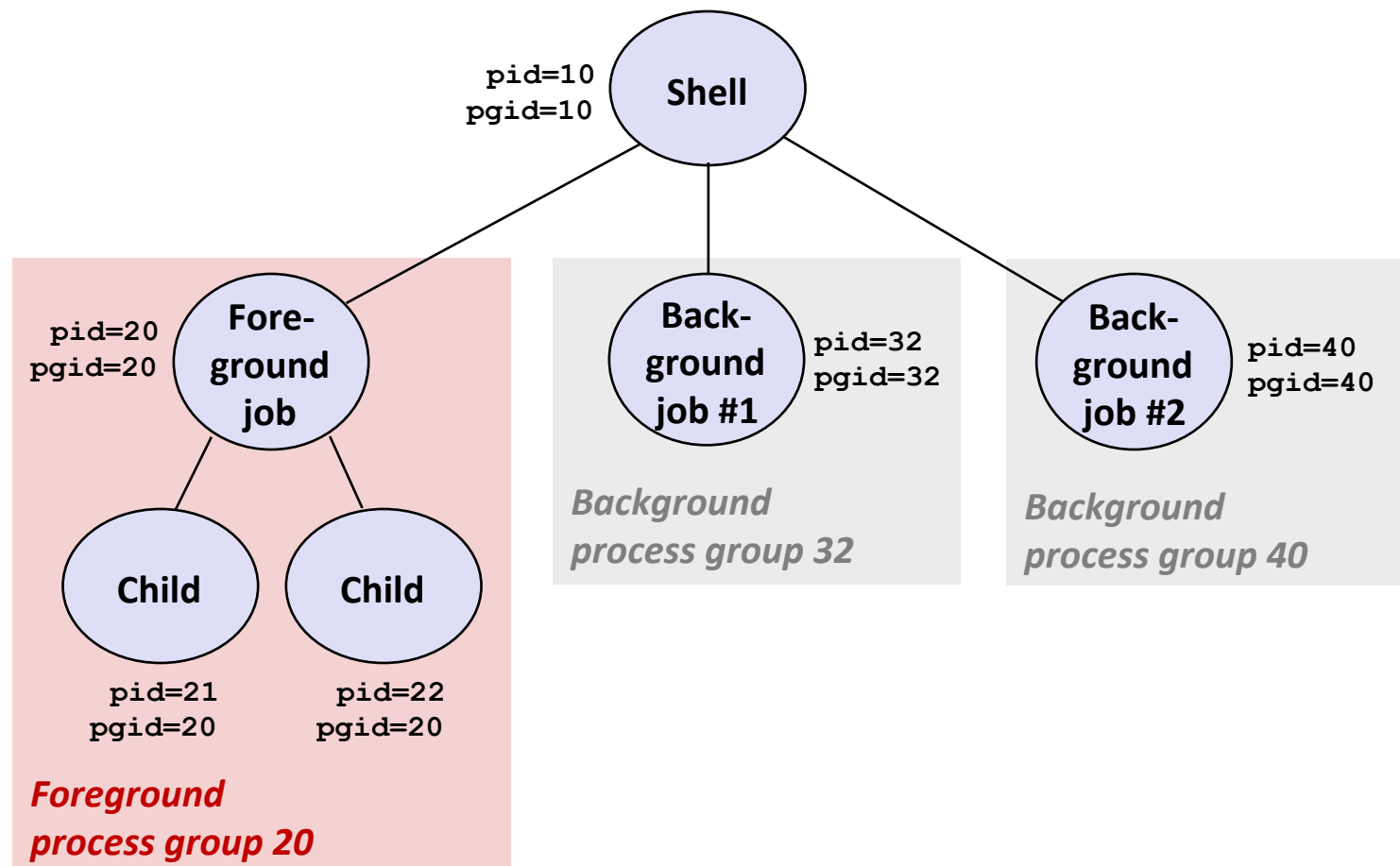
Sending Signals from the Keyboard

- **Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process

Example of `ctrl-c` and `ctrl-z`

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

STAT (process state) Legend:

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

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

See "man ps" for more details

Sending Signals with `kill` Function

```c
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

forks.c

Default Signal Handler

- **Each signal type has a predefined *default action*, which is one of:**
  - The process terminates
  - 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 user-level *signal handler*
    - Called when process receives signal of type `signum`
    - Referred to as *"installing"* the handler
    - Executing handler is called *"catching"* or *"handling"* the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handling Example

```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```
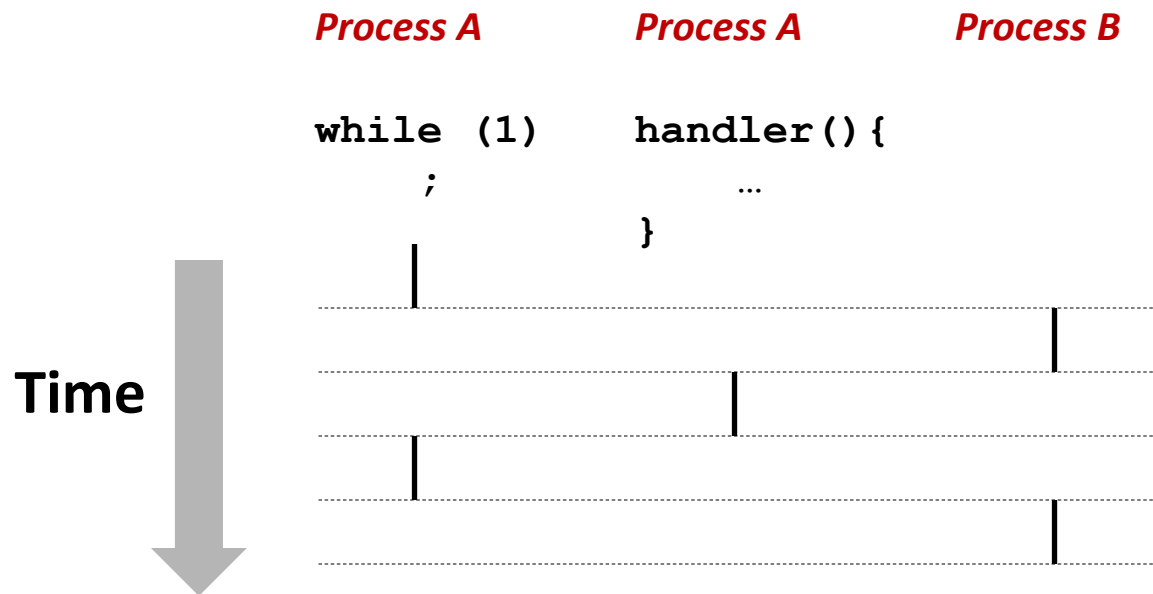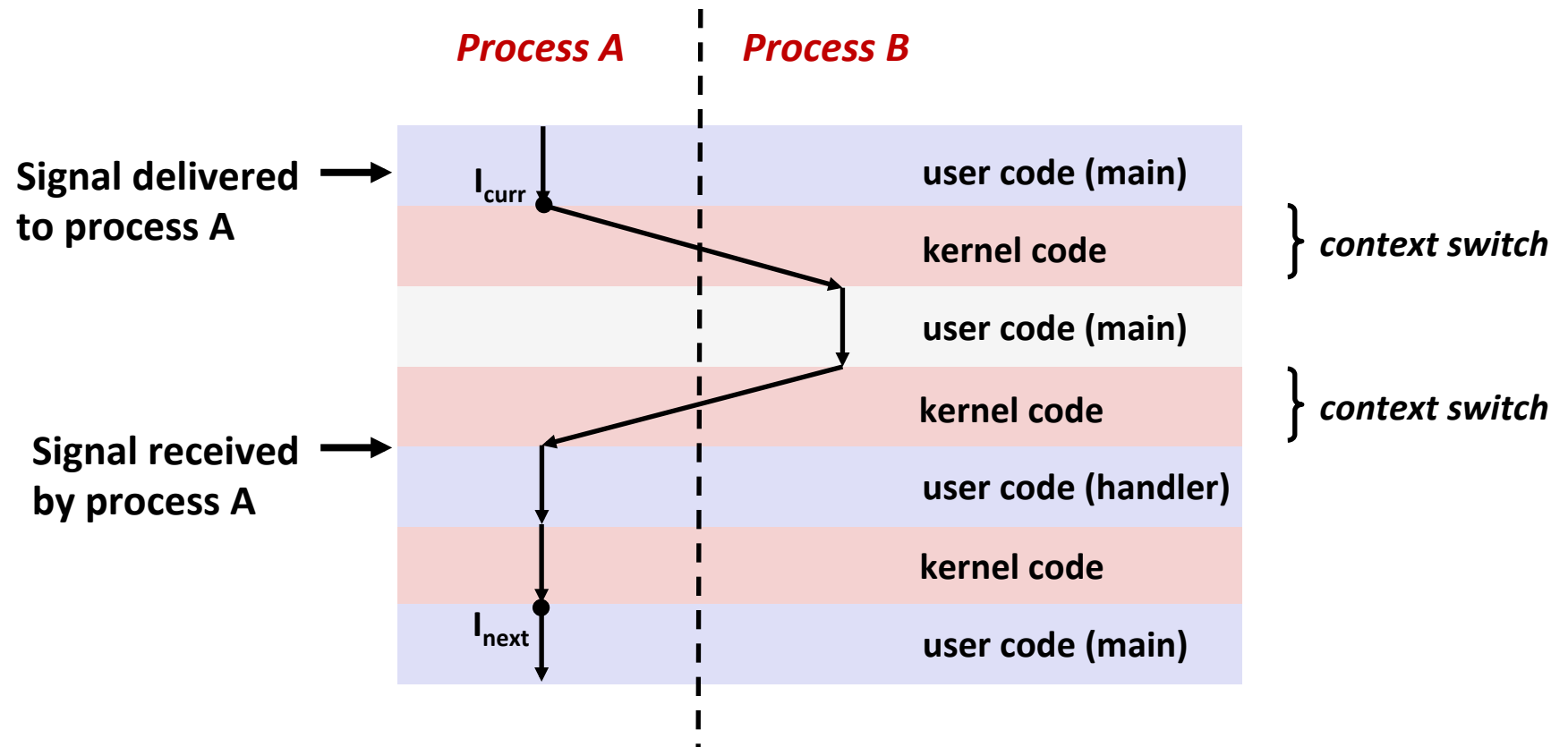
sigint.c

Signals Handlers as Concurrent Flows

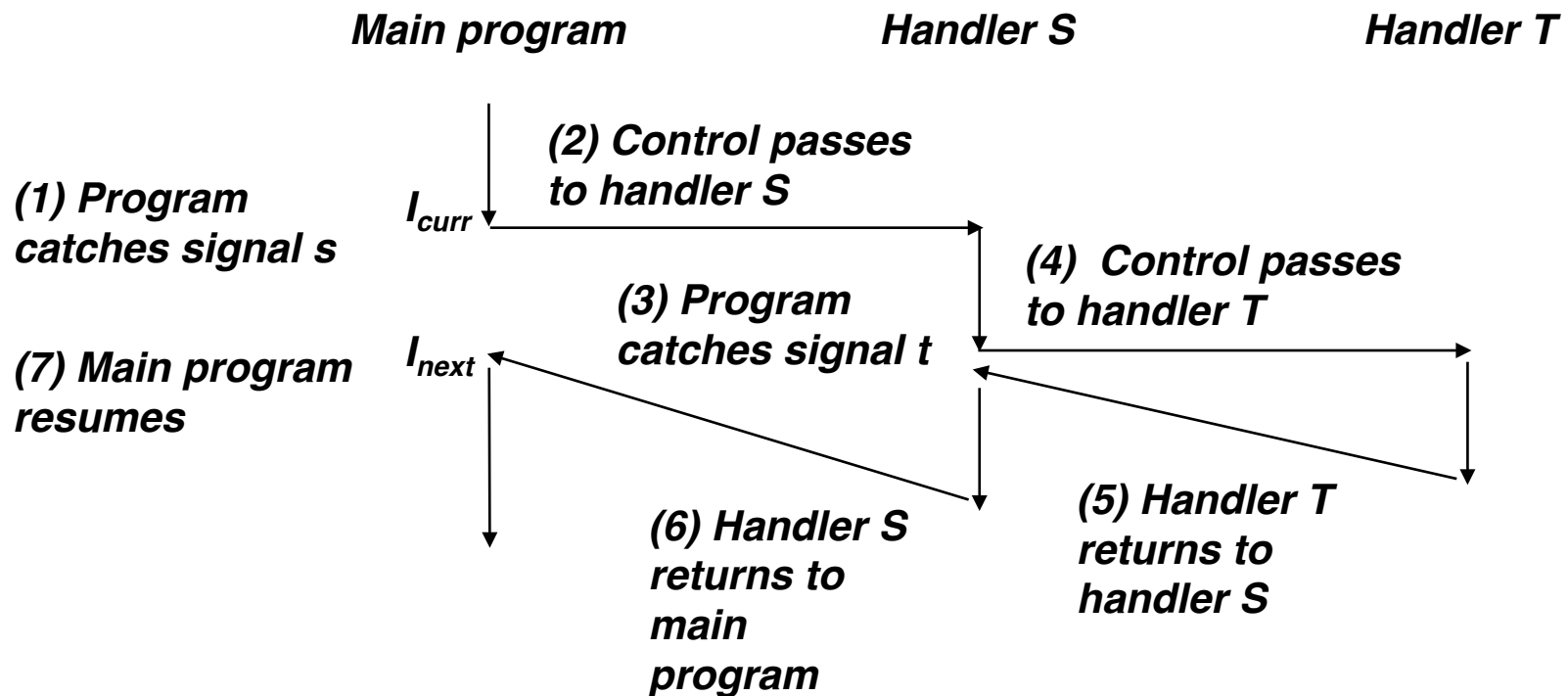■ **A signal handler is a separate logical flow (not process) that runs concurrently with the main program**

Process A   *Process A*   *Process B*

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

Time

Another View of Signal Handlers as Concurrent Flows

Nested Signal Handlers

■ **Handlers can be interrupted by other handlers**

Main program   **Handler S**   **Handler T**

(1) Program catches signal s

I_{curr}

(2) Control passes to handler S

(4) Control passes to handler T

(3) Program catches signal t

(7) Main program resumes

I_{next}

(5) Handler T returns to handler S

(6) Handler S returns to main program

Safe Signal Handling

- **Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - Shared data structures can become corrupted.

- **We'll explore concurrency issues later in the term.**

- **For now here are some guidelines to help you avoid trouble.**

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf, sprintf, malloc,` and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags as `volatile sig_atomic_t`**
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals