Chapter 8:
Exceptional Control Flow II
or
How a (shell) program interacts with the OS and I/O hardware

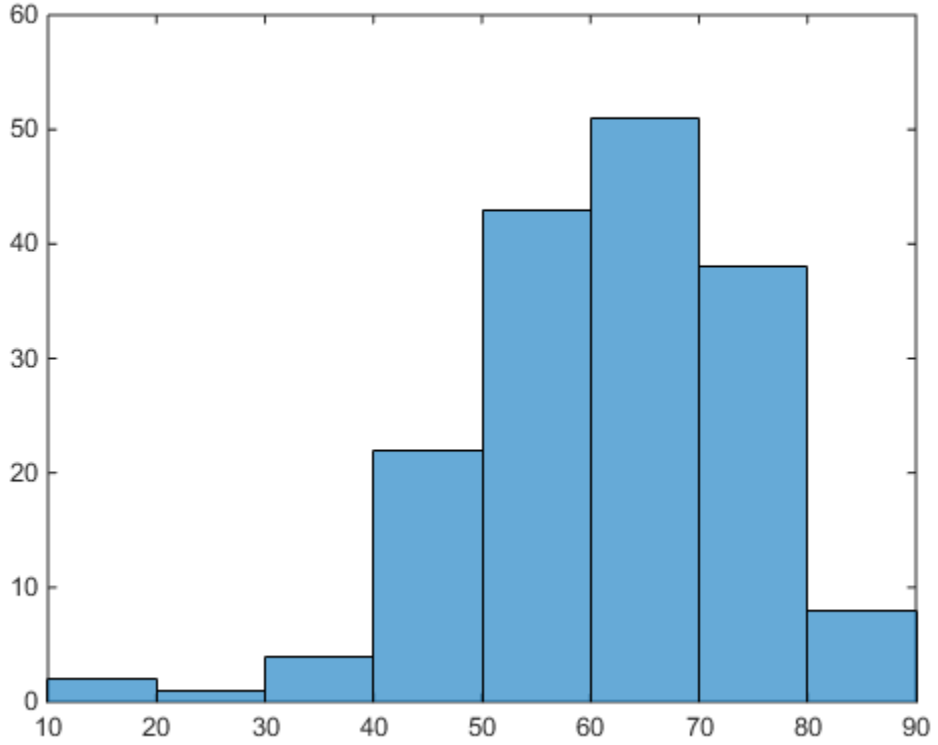
Useful for Shell Lab

Topics

- **Shells**
- **Signals**

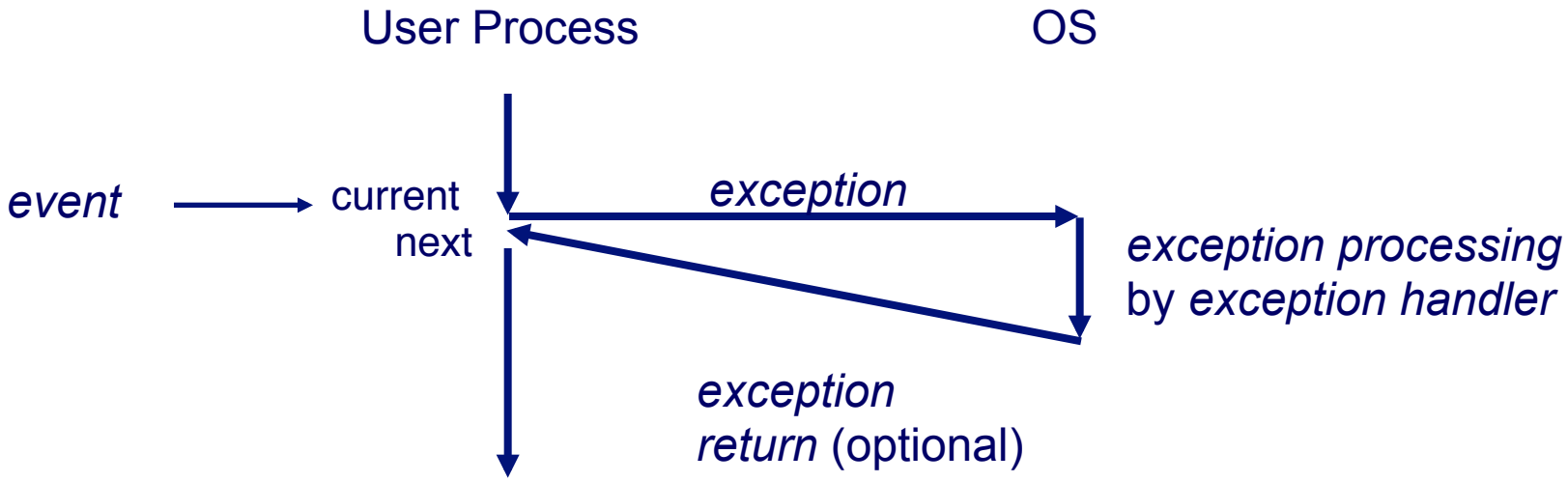
Announcements

- Performance Lab grading this week
- Shell Lab due after Fall Break, Monday Dec 8 by 8 am
- Midterm #2 returned in TA office hours this Thursday & Friday, or in recitation after Fall Break
  - Ave & Median = 61 of 90
  - Std dev = 12.5
  - Solutions posted soon
- Reading:
  - Read Chapter 8 except 8.6 (no nonlocal jumps)
  - Read Chapter 9, except 9.6 and 9.7 (no case study and no memory mapping)



Recap...

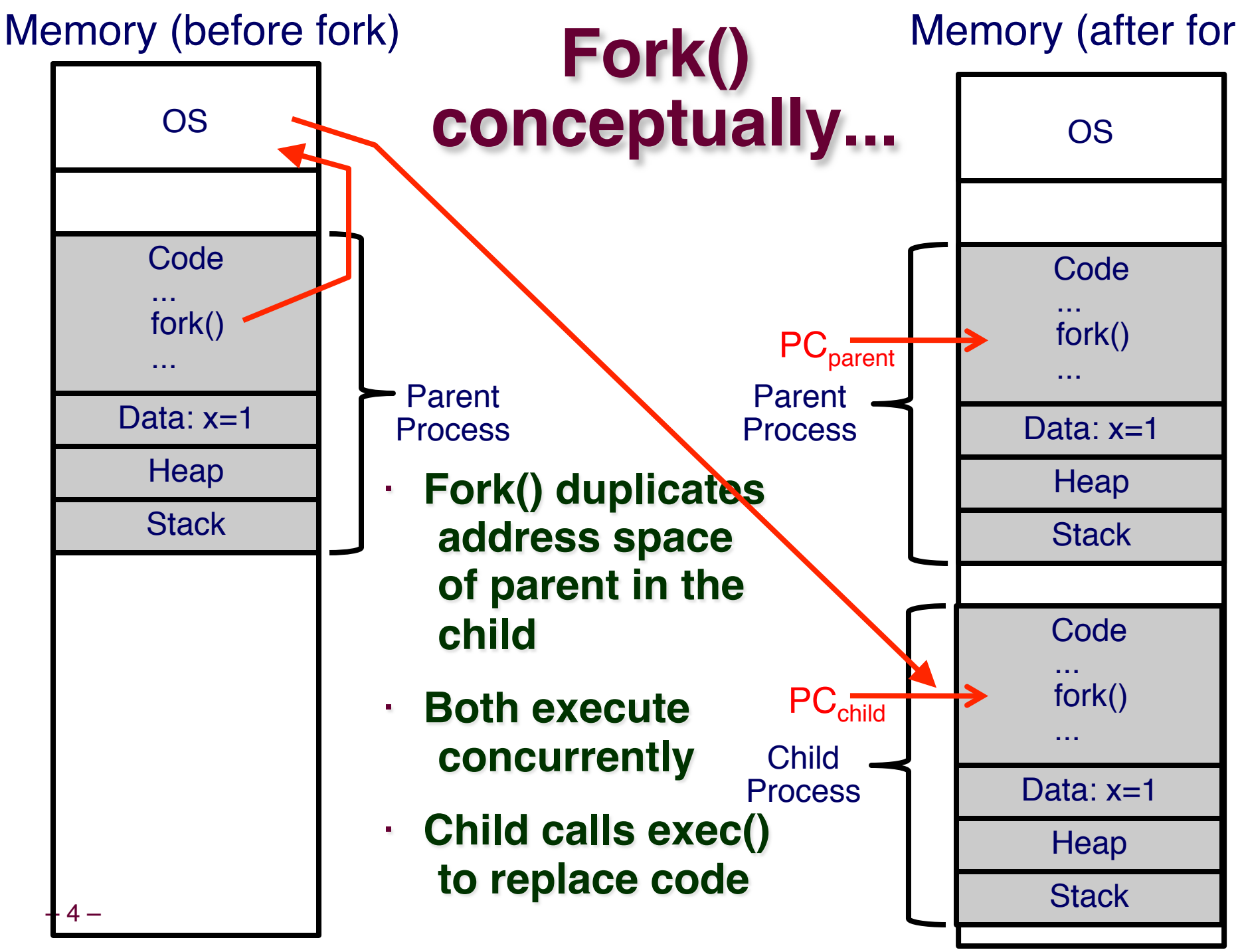
- **Exceptional control flow**



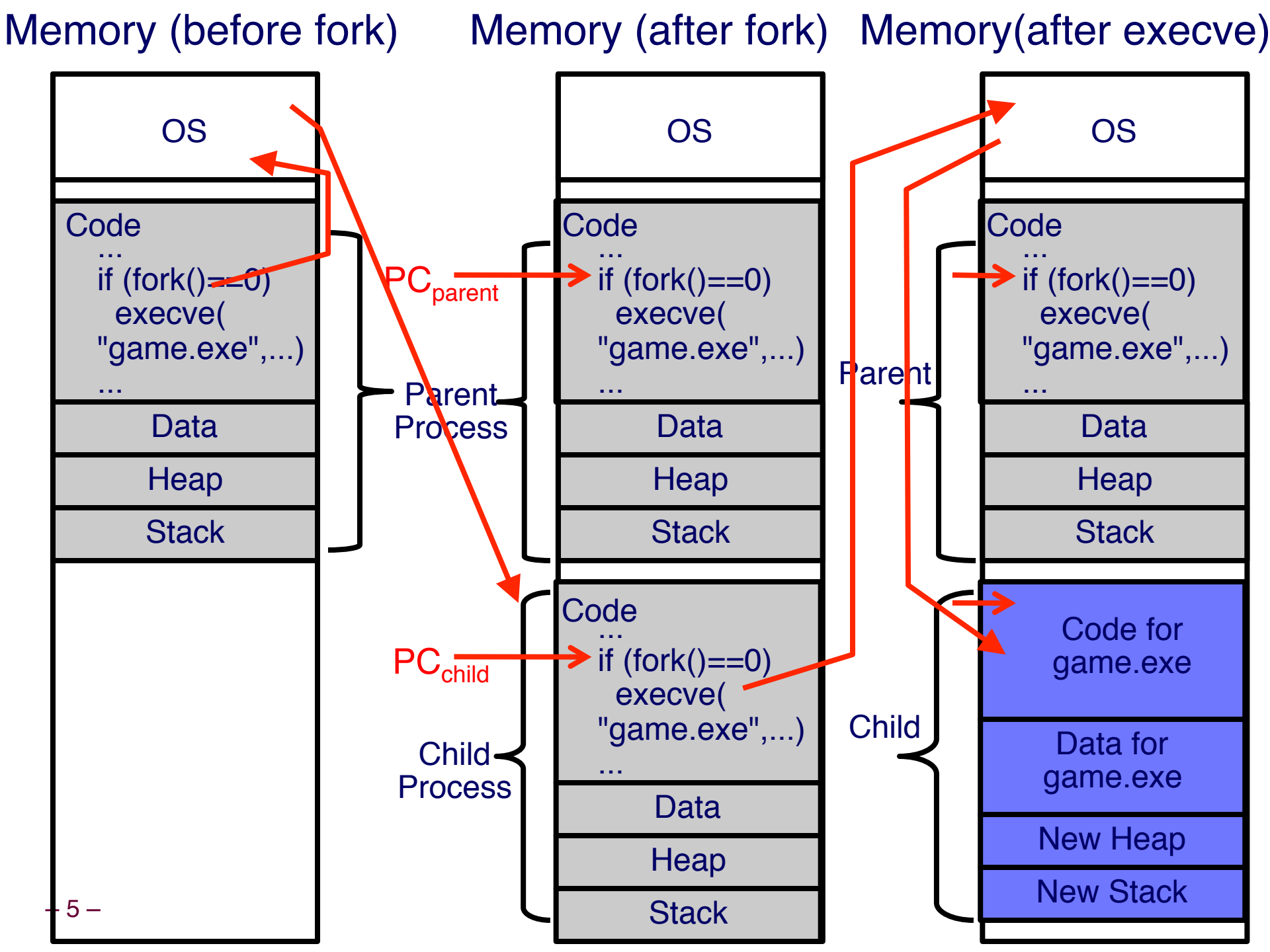
  - **Types of exceptions include hardware interrupts, traps/software interrupts (system calls to OS), faults (divide by zero, page fault, memory fault), and aborts (hardware failures)**

- **Define a process**
  - **Actively executing program loaded into memory**
  - **Given illusion it has its own CPU (context switching) & memory**
  - **create a child process in Unix/Linux with `fork`() command**
  - **parent calls `wait`() or `waitpid`() to reap child processes**

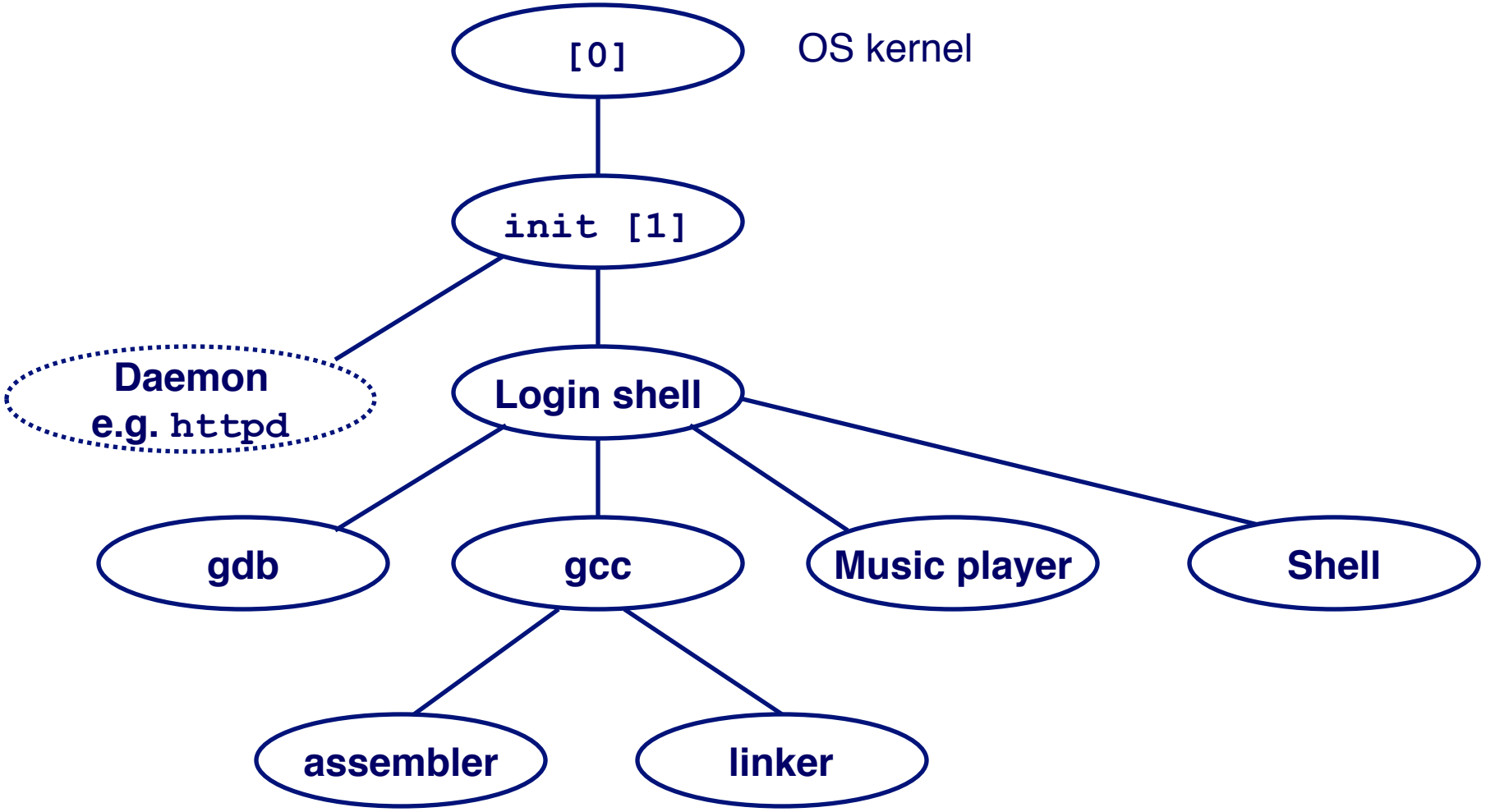
Fork() conceptually...



- **Fork() duplicates address space of parent in the child**
- **Both execute concurrently**
- **Child calls exec() to replace code**


Memory (before fork)
Memory (after fork)
Memory(after execve)
OS
Code
...
if (fork()==0)
execve(
"game.exe",...)
...
Data
Heap
Stack
PC parent
PC child
Parent Process
Child Process
Parent
Child
Code for game.exe
Data for game.exe
New Heap
New Stack

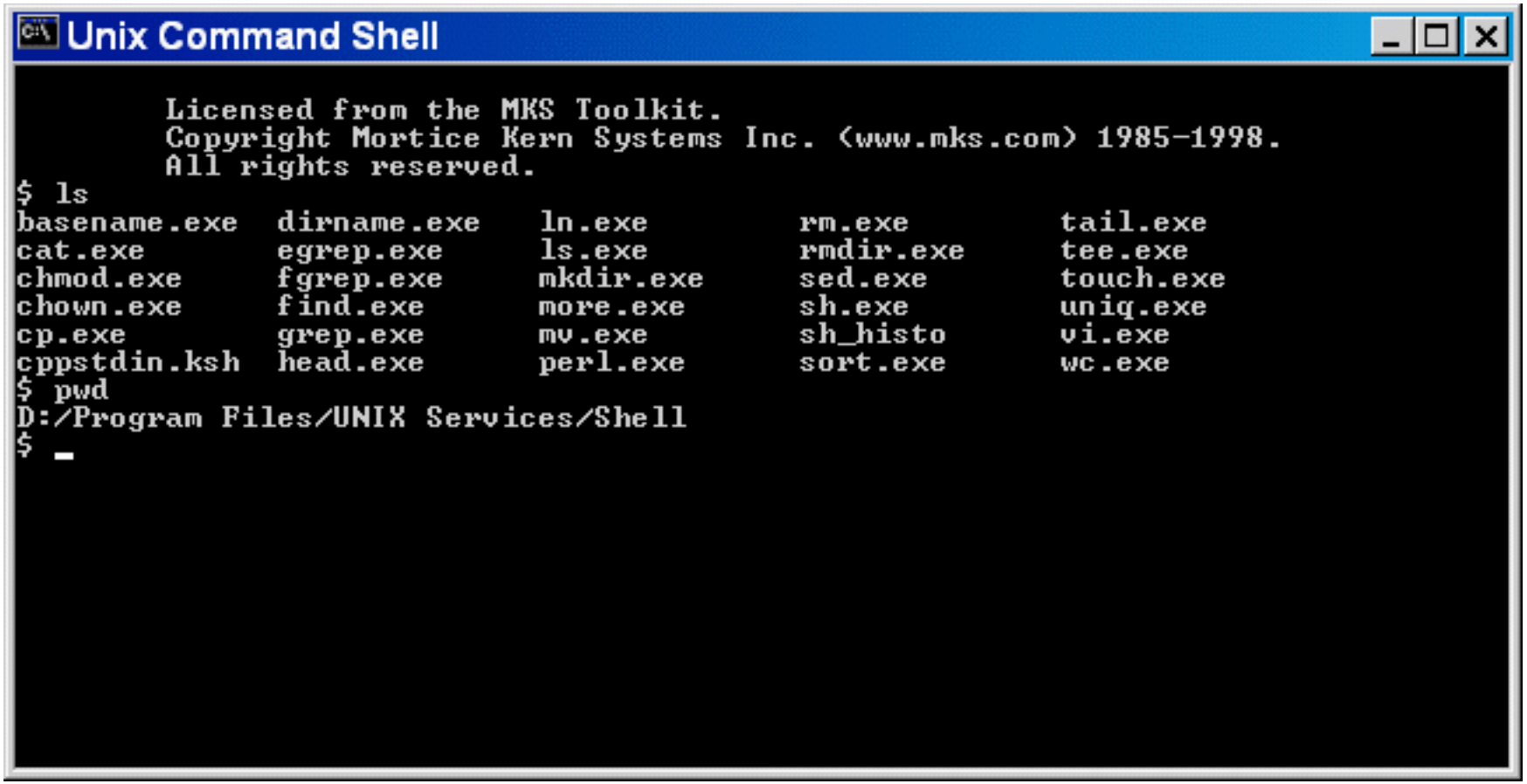
Unix Process Hierarchy Example



Shell Programs

A *shell* is an application program that runs programs on behalf of the user.

- Also called a terminal



Shell Programs

Example Types:

- `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);
    }
}
```

A typical shell is simply a loop that gets a string from the command line and executes it

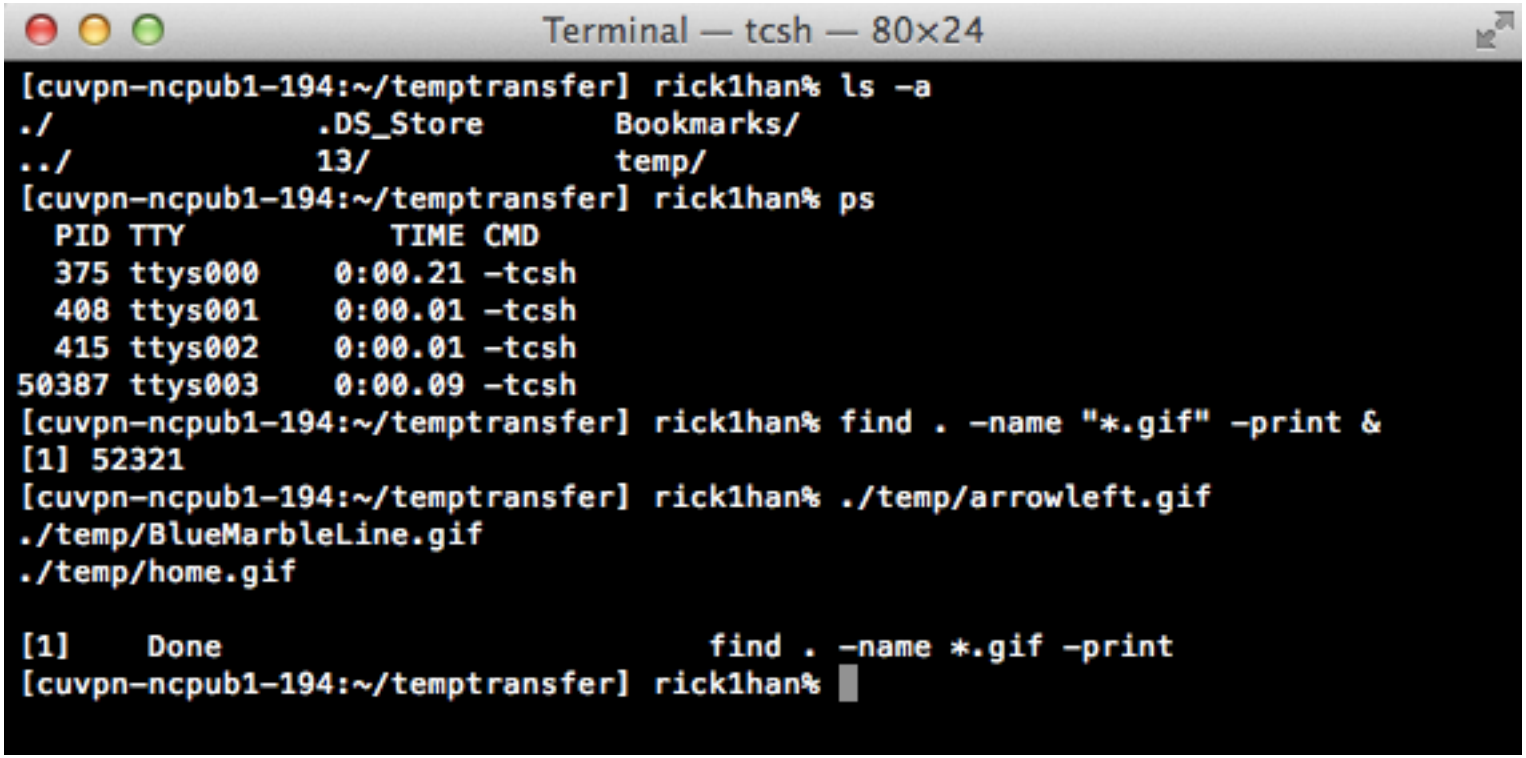
Execution is a sequence of read/evaluate steps

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);
    }
}
```

Background vs Foreground Processes

Note how "ls –a" and "ps" both run in the foreground



Note how "find … & " runs in the *background* due to the "&" at the end of the command line

Keeping Track of Background Processes

```
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.**

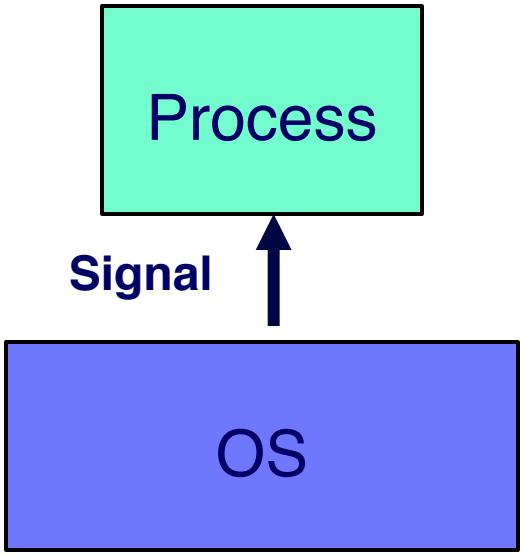
Solutions to reaping background jobs:

- **Use `waitpid`(pid, status, WNOHANG)**
  - **return immediately if no child has exited, allowing the parent to keep running and not block. Have to keep calling waitpid() in a polling-style loop.**
- **Use 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.
- Signals are identified by small integer ID's
- The only information in a signal is its ID and the fact that it arrived.

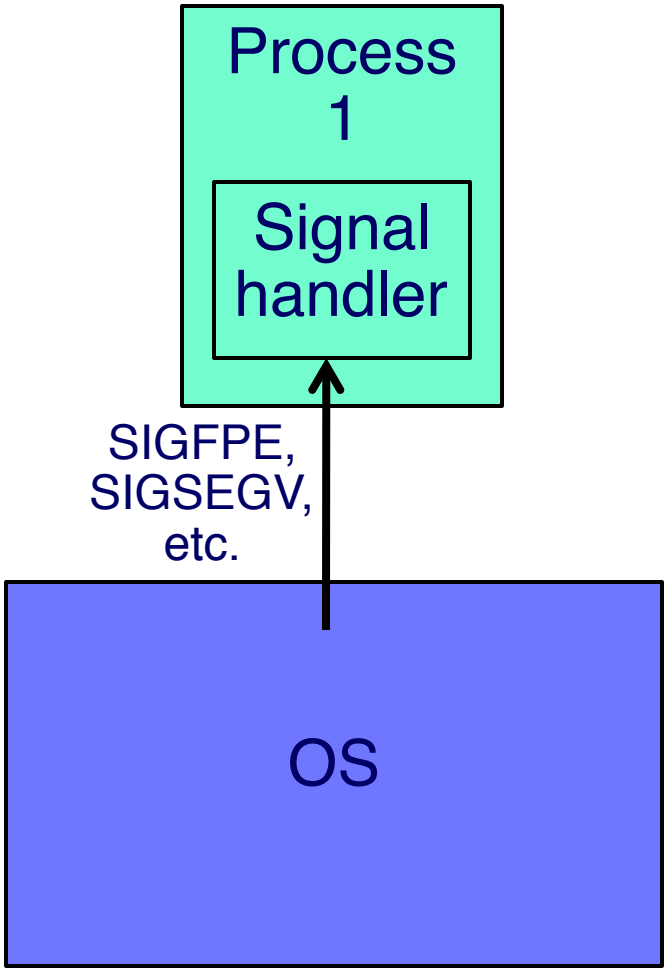


ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Interrupt from keyboard (`ctl-c`)
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate & Dump	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated

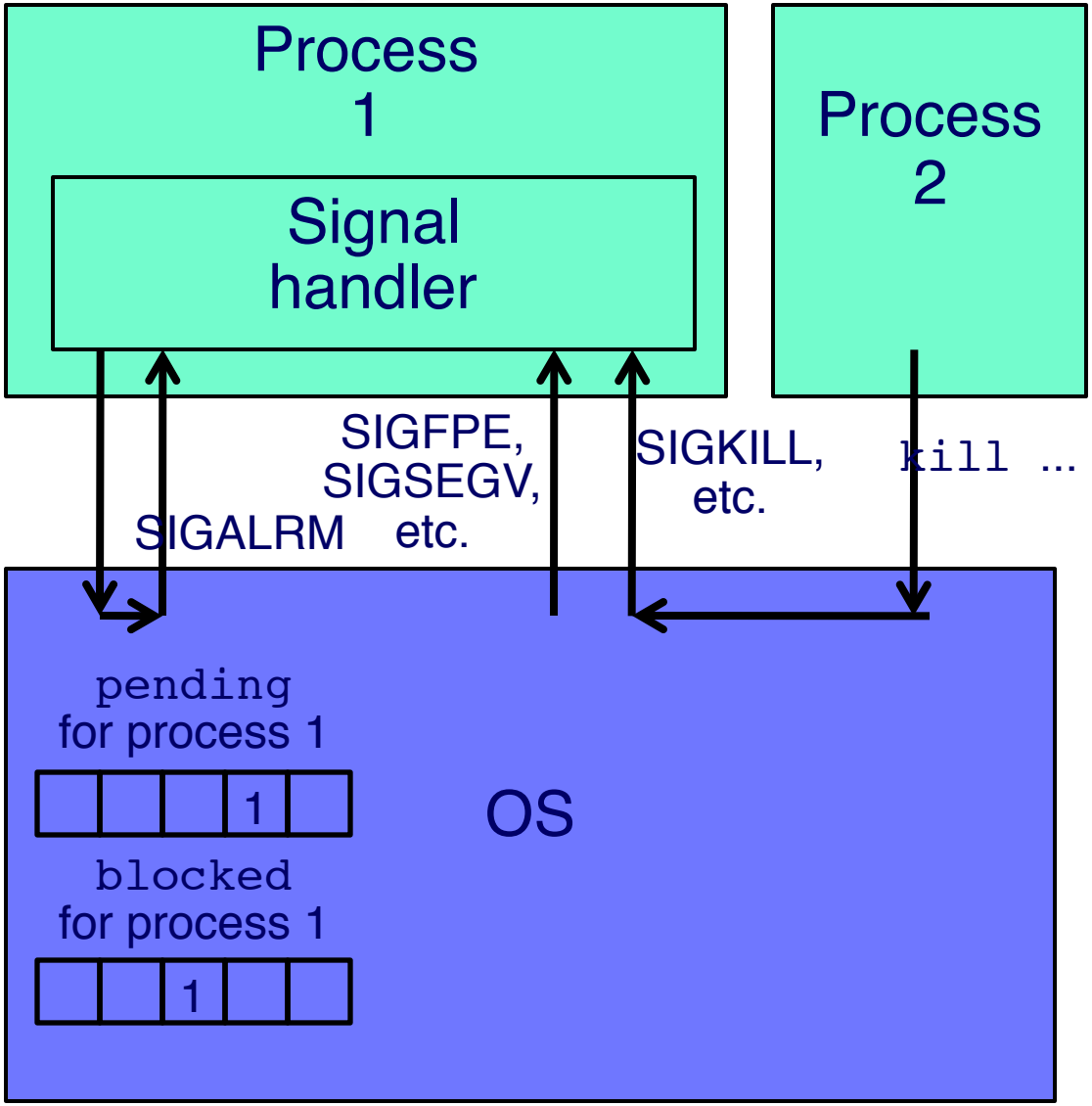
Signal Concepts

Sending 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, e.g. SIGKILL or SIGUSR1 or 2



Signal Concepts – A Complete View



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.

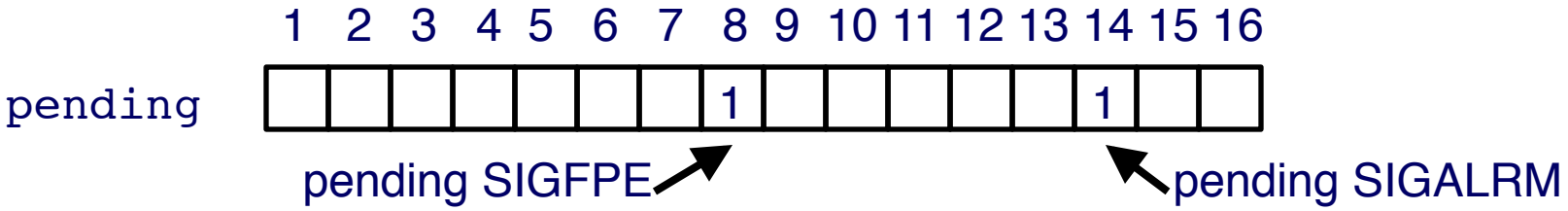
- Blocked signals can be delivered, but will not be received until the signal is unblocked.

A pending signal is received at most once.

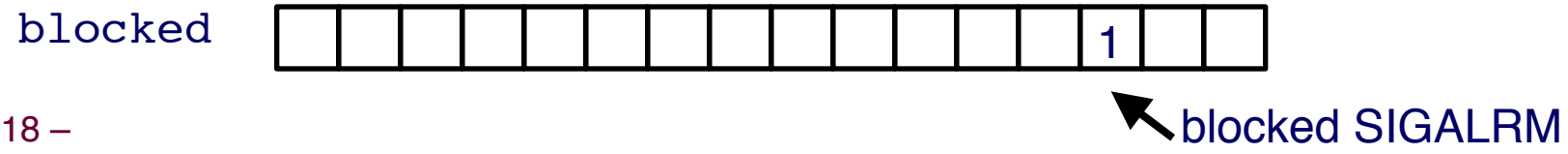
Signal Concepts

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**



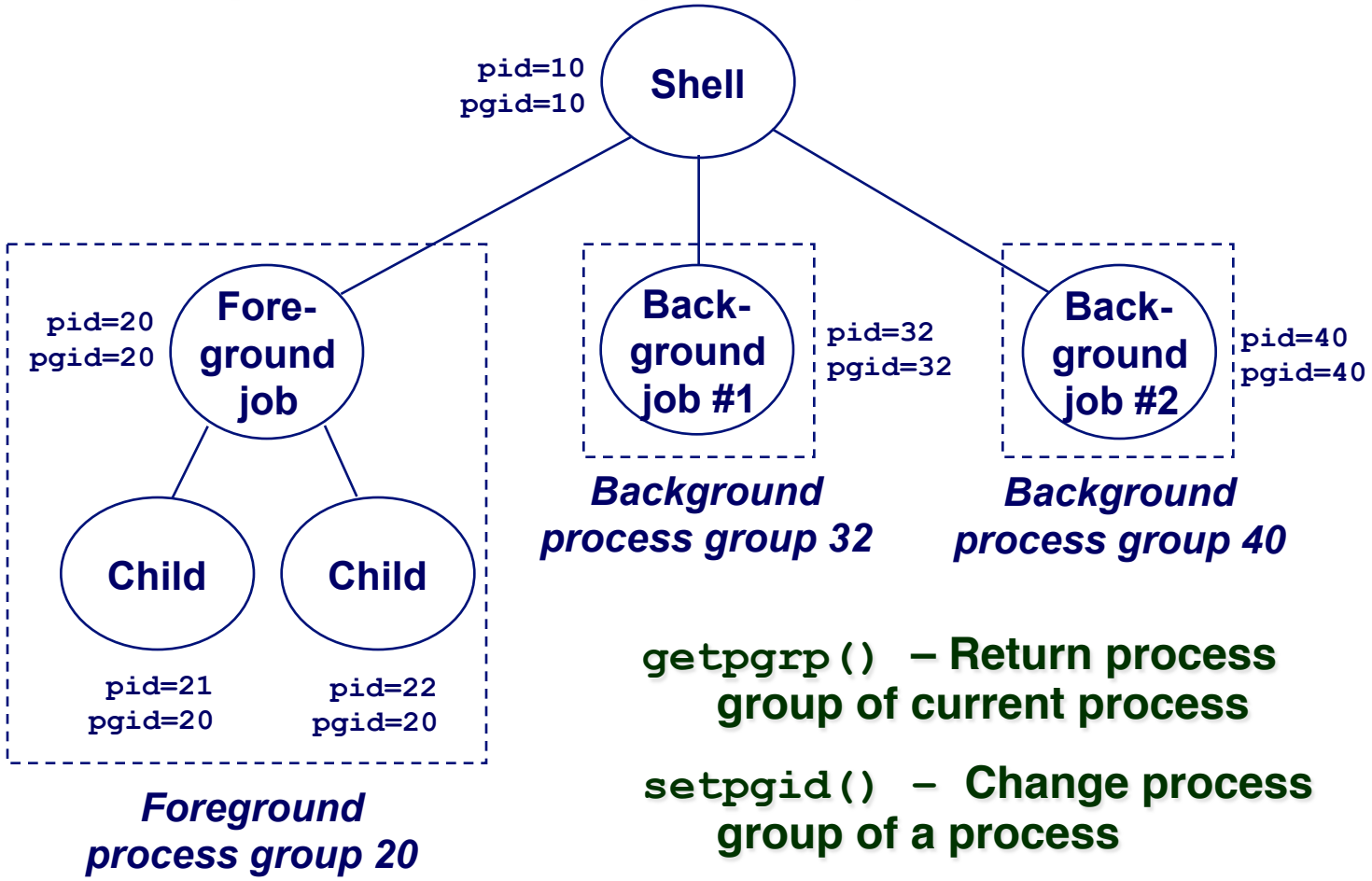
- **`blocked` – represents the set of blocked signals**
  - **Can be set and cleared by the application using the `sigprocmask` function.**



Signaling Process Groups

Every process belongs to exactly one process group

Child belongs to parent's group by default



`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

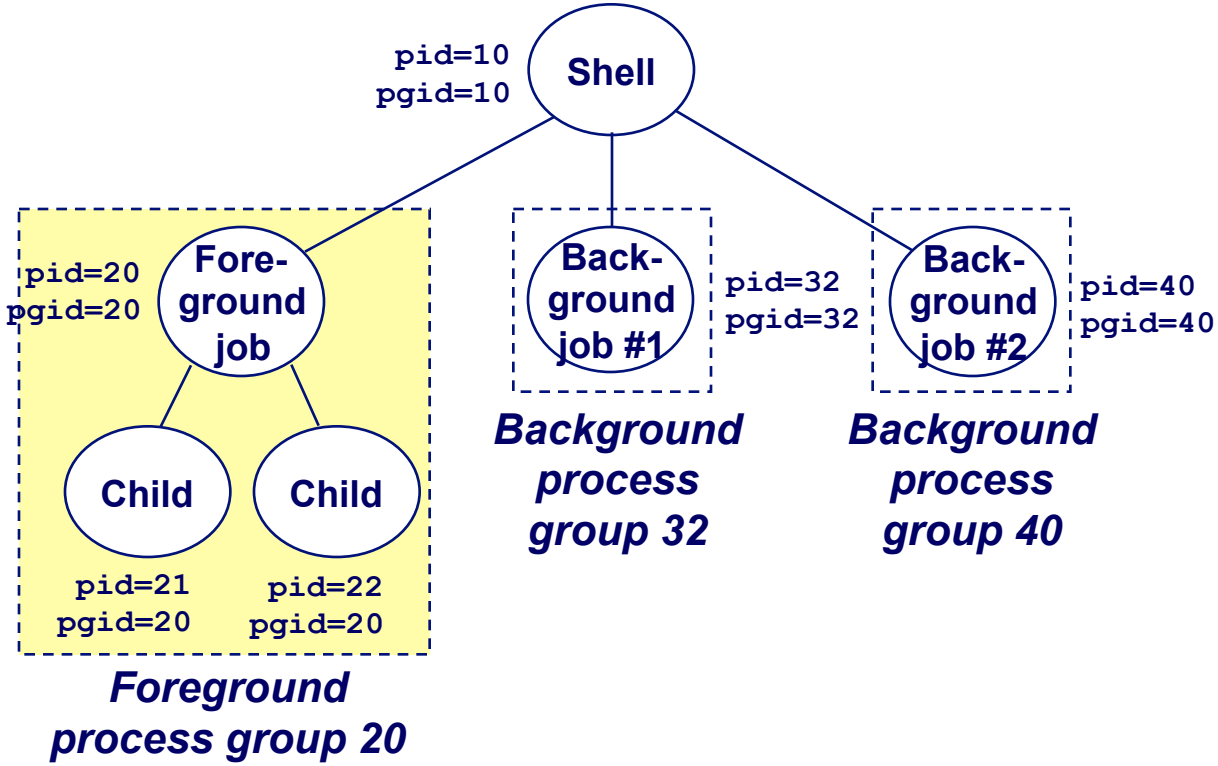
- `kill -9 24818`
  - Send SIGKILL to process 24818
- `kill -9 -24817`
  - Send SIGKILL to every process in process group 24817.

```
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);
    }
}
```

Receiving Signals

Suppose the OS 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 that was interrupted by receipt of the signal.**

Installing Signal Handlers (2)

```
void handler1(int sig_num) {
    /* handle SIGINT */
    ...
}

void handler2(int sig_num) {
    /* handle SIGALRM */
    ...
}

int main() {
    ...
    signal(SIGINT, handler1);  // register handler1 to
                               catch SIGINT signals
    signal(SIGALRM, handler2); // register handler2 to
                               catch SIGALRM signals
    ....

    while (1) {
        /* main loop's processing can be interrupted by
           handlers above */
        DoSomething();
    }
}
```

Signal Handling Example

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}

void fork13()  // similar to fork12()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    for (i = 0; i < 5; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinit

    /* Parent: send SIGINT (signal 2)
       to each child */
    for(i=0; i<5; i++) {
        printf("Sending SIGINT to...");
        kill(pid[5-i-1],SIGINT);
    }

    /* wait on 5 children and printf
      "Child ... terminated with exit status
      ..." when each child exits */
}
```

```
linux> ./forks 13
Sending SIGINT to process 24973
Sending SIGINT to 24974
Sending SIGINT to 24975
Sending SIGINT to 24976
Sending SIGINT to 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>
```

Portable Signal Handling

The `sigaction` function portably modifies the default action associated with the receipt of signal `signum`:

- `int sigaction(int signum, struct sigaction *act, struct sigaction *oldact)`

Usage:

- **to set the handler, declare a struct sigaction action, and set action.sa_handler = handler;**
- **same behavior as `signal` for SIG_IGN and SIG_DFL: ignore or revert to the default action on receipt of signals of type `signum`.**
- **More portable than `signal` in handling interrupted slow system calls**
  - **users specify how to handle interrupted slow system calls, either restarting them or aborting them by setting for example action.sa_flags = SA_RESTART to restart system calls**

Signal Handler Funkiness

```
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);  /* send SIGCHLD to parent */
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
}
```

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
- Suppose N=3, then 3 child processes are forked, but if all 3 call exit(0) in close succession, then only 2 of 3 SIGCHLD signals are caught - 1 is handled, 1 is pending & the last is lost!

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 = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig,
pid);
    }
}

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

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) {
  }
}
```

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>
```

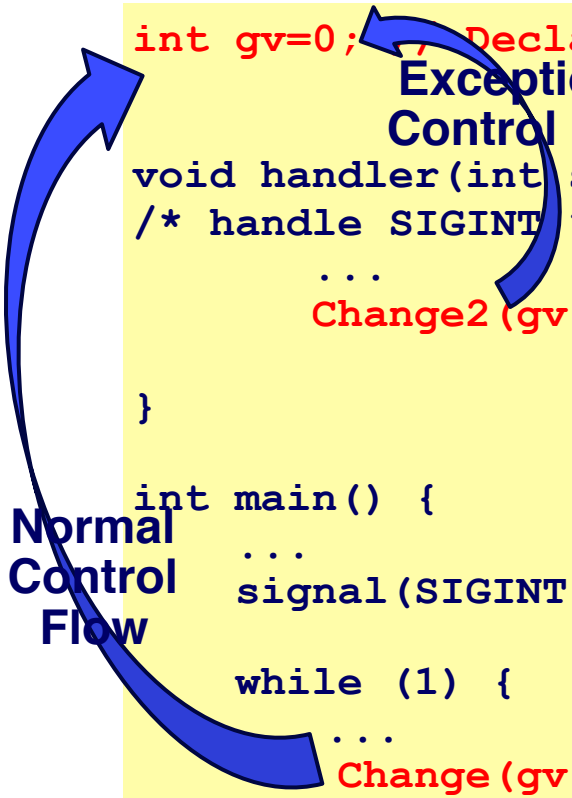
A General Signal Handler

```
void handler(int sig_num) {
    switch(sig_num) {
        case SIGINT:
            /* handle SIGINT */
            ...
            break;
        case SIGALRM:
            /* handle SIGALRM */
            ...
            break;
        case SIGCHLD:
            /* handle SIGCHLD */
            ...
            break;
    }
}

int main() {
    ...
    signal(SIGINT, handler);  // register handler for SIGINT
    signal(SIGALRM, handler); // register handler for SIGALRM
    signal(SIGCHLD, handler); // register handler for SIGCHLD
    while (1) {
       DoSomething();
    }
}
```

Designing Asynchronous Programs

- Normally, we have control flow that modifies program state
- Asynchronous interruptions like signals can also modify program state outside of the normal control flow – program becomes event-driven
- Must be aware that global variables/state can change outside the normal control flow and that an executing program can be interrupted anywhere in its control flow
  - a *race condition* may occur in which normal control flow *races* with exceptional control flow to modify state, leading to unpredictable results

Designing Asynchronous Programs




```
int gv=0;  // Declare global variable

void handler(int sig_num) {
/* handle SIGINT */
        ...
        Change2(gv); // Change value of gv in a different way,
                              e.g. gv = gv * 2
}

int main() {
    ...
    signal(SIGINT, handler);  // register handler for SIGINT

    while (1) {
        ...
        Change(gv);  // Change value of global variable,
        ...                   e.g. gv++
    }
}
```

- As a programmer, you must be aware that normal control flow can be interrupted at *any instruction*, and global variable state may be changed *at any time* by signal handlers

Race Condition Example

```
int ccount = 2;  /* global variable */

void child_handler(int sig) {
    int temp2;
    /* ccount-- */
    temp2 = ccount; temp2--; ccount = temp2;
    ...
}

void main() {
    pid_t pid[2];
    int i, temp;
    signal(SIGCHLD, child_handler);

    for (i = 0; i < 2; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child */
            exit(0);  /* send SIGCHLD to parent */
        }
    for (i = 0; i < 2; i++) {
        /* ccount++ */
        temp = ccount; temp++; ccount = temp;
    }
}
```

- Both the parent's main() and the parent's signal handler modify the global variable ccount
- expect ccount = 2 after both children have completed
- Let parent fork 2 children. Parent executes 1st & sets temp = ccount (= 2)
- 1st child exit() interrupts parent's control flow, calling child_handler(), which decrements ccount to 1
- then parent resumes & set ccount to 3!
- then 2nd child exits(), which decrements ccount back to 2.
- in last step of for loop, parent increments ccount back to 3!

Designing Asynchronous Programs

- To avoid race conditions of the previous slide, use synchronization primitives – see Operating Systems course
- Another race condition can occur inside a general signal handler
  - For a particular signal type, only one flow of execution at a time inside a signal handler, but...
  - If there are two different types of signals, then while execution is inside of the general signal handler handling signal1, it may be interrupted to handle a different signal2, which brings
  - If the case statements are both modifying the same variable, e.g. ccount++ and ccount--, then can have a race condition
  - *good practice is to mask out or block all other signals before handling a signal*

Race Condition Example 2

```
void child_handler(int sig) {
    pid_t pid;
    while ((pid = waitpid(...)) > 0) /* reap
                                        zombie child */
        deletejob(pid);
    ...
}

void main() {
    ...
    signal(SIGCHLD, child_handler);

    while (1) {
        if ((pid = fork()) == 0) {
            /* Child */
            execve("/bin/date", argv, NULL);
        }

        /* Parent */
        addjob(pid);
    }
}
```

- expect deletejob() to occur after addjob()
- But, could have the following sequence:
- parent executes fork(), but child runs first
- child completes "/bin/date" causing SIGCHLD to be sent to parent
- parent handles SIGCHLD first, calling deletejob() first
- then parent resumes normal control flow, calling addjob() last!
- system now thinks that there is a job, though the child no longer exists!

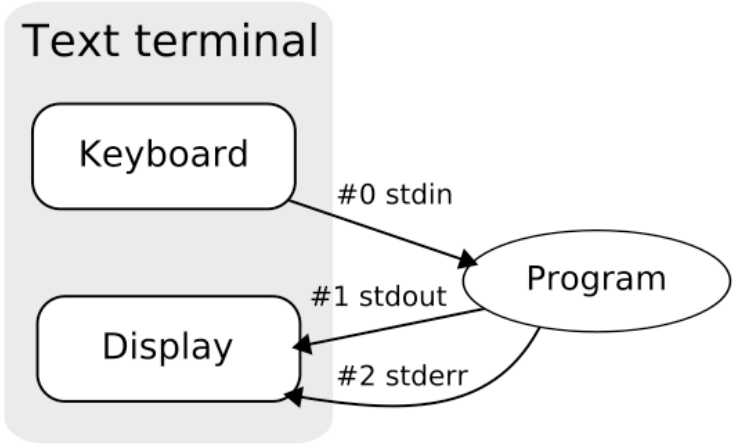
Race Condition Example 2 (cont.)

```
void child_handler(int sig) {
    pid_t pid;
    while ((pid = waitpid(...)) > 0) /* reap
                                        zombie child */
        deletejob(pid);
    ...
}

void main() {
    ...
    signal(SIGCHLD, child_handler);

    while (1) {
         ...
         sigaddset(&mask, SIGCHLD);
         sigprocmask(SIG_BLOCK, &mask, NULL);
        if ((pid = fork()) == 0) {
            /* Child */
           sigprocmask(SIG_UNBLOCK, &mask, NULL);
            execve("/bin/date", argv, NULL);
        }

         /* Parent */
         addjob(pid);
         sigprocmask(SIG_UNBLOCK, &mask, NULL);
    }
}
```

- Solution: want parent to execute first and call addjob() first, so before forking make sure to block SIGCHLD signals
- After calling addjob(), unblock SIGCHLD signals
- Therefore, deletejob() always occurs after addjob()
- There is no longer any race condition
- Note, we also should unblock SIGCHLD signals in the children, which inherit the set of blocked signals from the parent, but don't really need to block any of those signals.

Processes and File descriptors

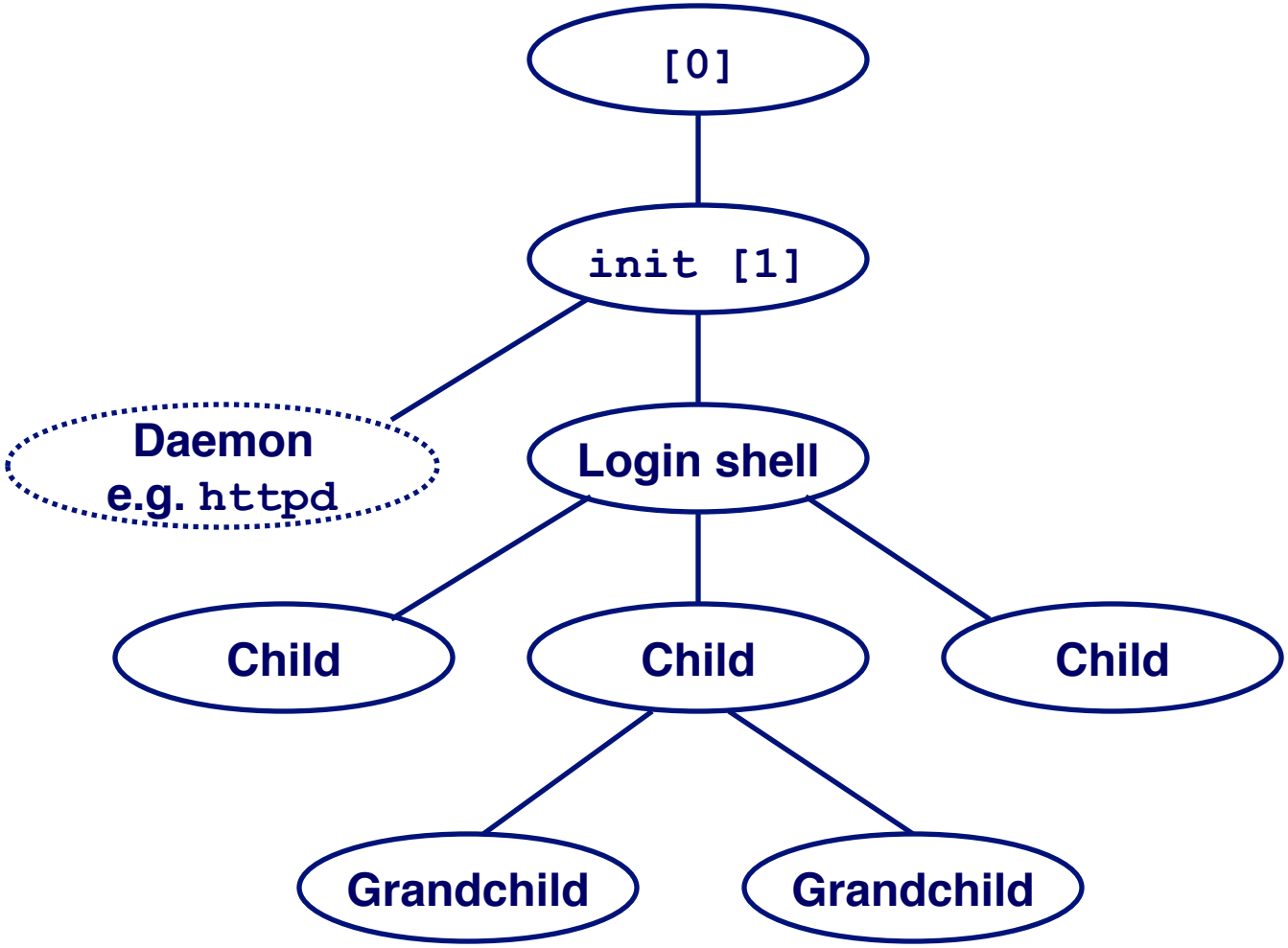
- A process typically has a set of default file descriptors assigned to it:
  - stdin = standard input (e.g. keyboard) = file descriptor 0
  - stdout = standard output (e.g. display) = file descriptor 1
  - stderr = standard error (e.g. display) = file descriptor 2
  - Usage: e.g. fgets(stdin, ...), fprintf(stdout, ...)



- Can redirect stderr to stdout using `dup2(oldfd, newfd)`, which makes newfd be the copy of oldfd, closing newfd first if necessary:
  - dup2(1,2) duplicates stdout into stderr – helpful for shell lab

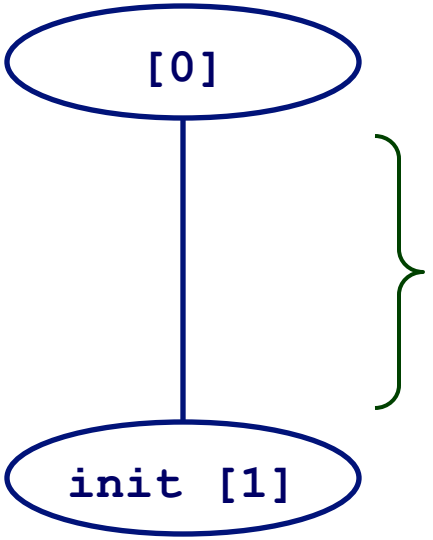
Supplementary Slides

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.

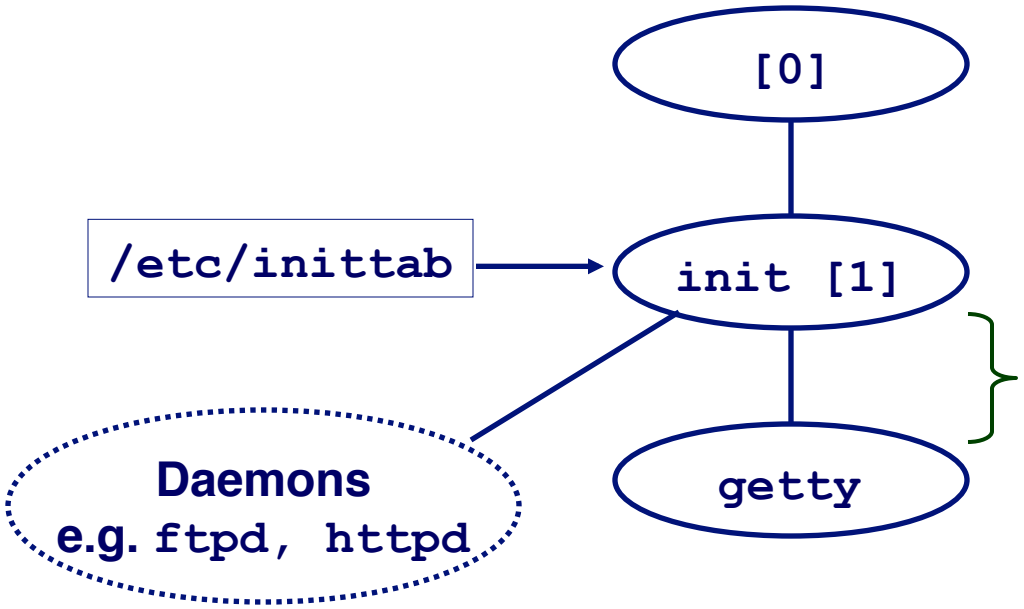


Process 0: handcrafted kernel process

Process 0 forks child process 1

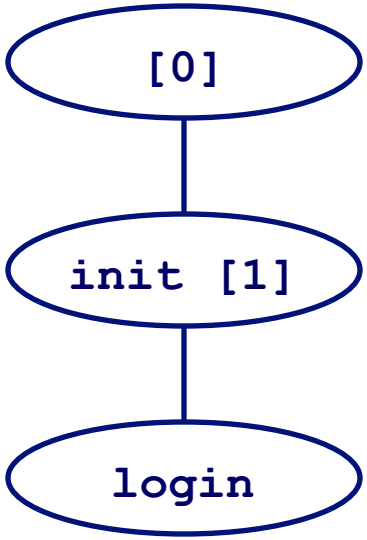
Child process 1 execs `/sbin/init`

Unix Startup: Step 2



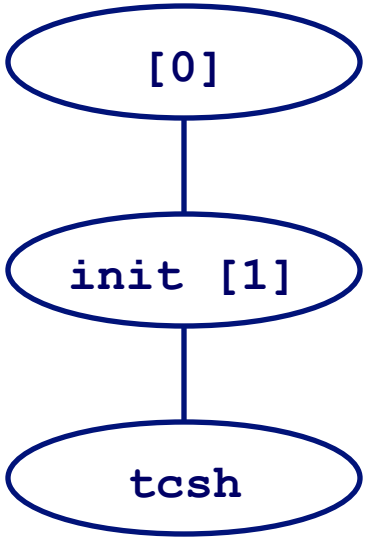
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

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`

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)`

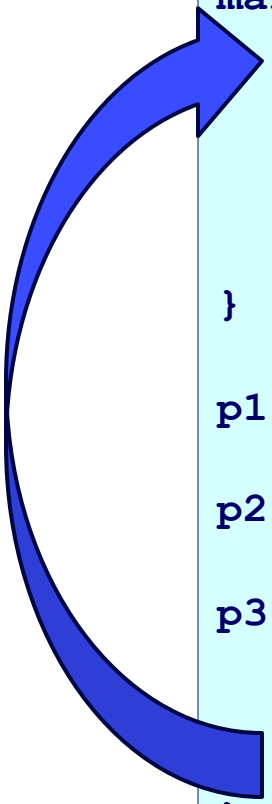
- Meaning:
  - return from the `setjmp` remembered by jump buffer `j` again...
  - ...this time returning `i` instead of 0
- Called after `setjmp`
- Called once, but never returns

`longjmp` Implementation:

- Restore register context from jump buffer `j`
- Set `%eax` (the return value) to `i`
- Jump to the location indicated by the PC stored in jump buf `j`.

`sigsetjmp()` and `siglongjmp()` are similar versions that can be used with signals

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 */
}

p1() {... p2(); ...}

p2() {... p3(); ...}

p3() {
   <error checking code>
   if (error)
      longjmp(buf, 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");
  }
}
```

```
bass> a.out
starting
processing...
processing...
restarting
processing...
processing...
processing...
restarting
processing...
restarting
processing...
processing...
```

Ctrl-c

Ctrl-c

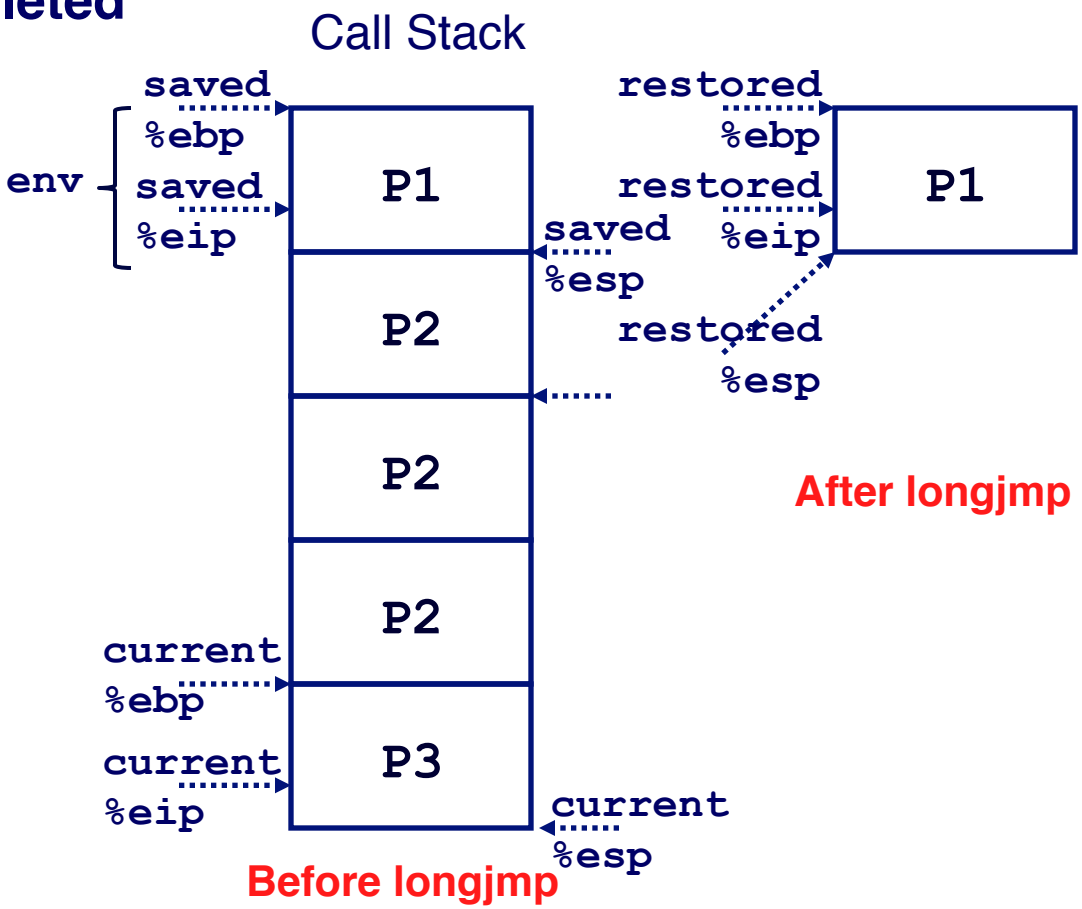
Ctrl-c

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);
}
```

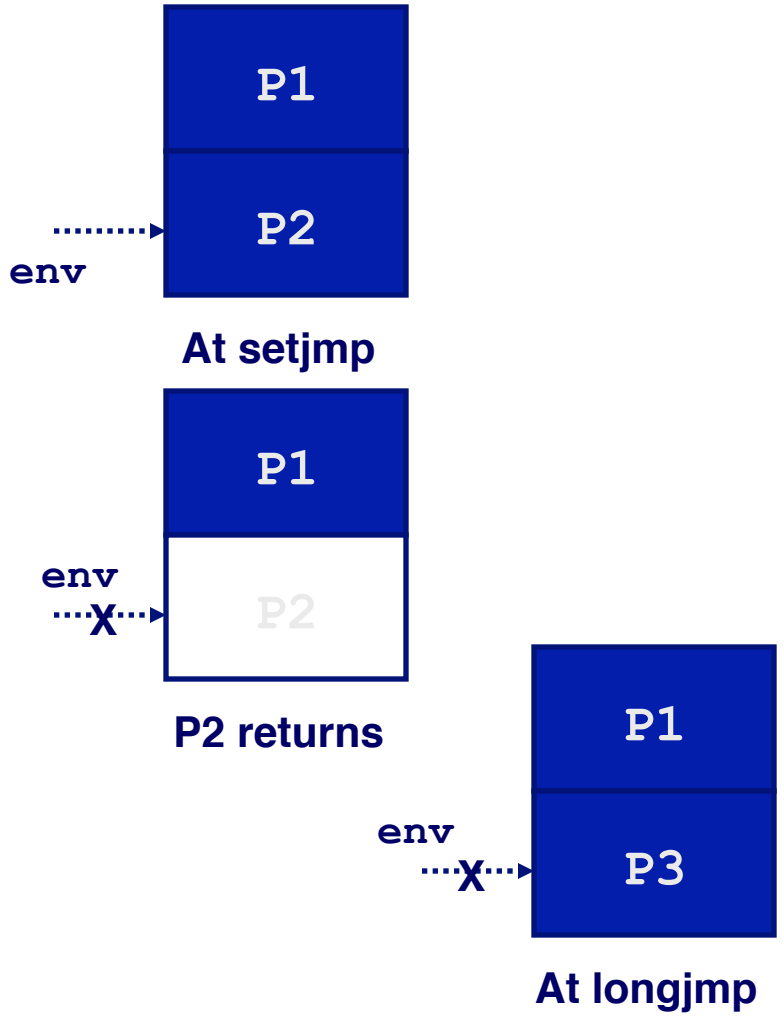


Limitations of Long Jumps (cont.)

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