Systems Programming

Signals

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.



Review: Exception and Process

- **Exception**
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **Process**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on any single core
  - Each process appears to have
    - 1) total control of processor and
    - 2) private memory space

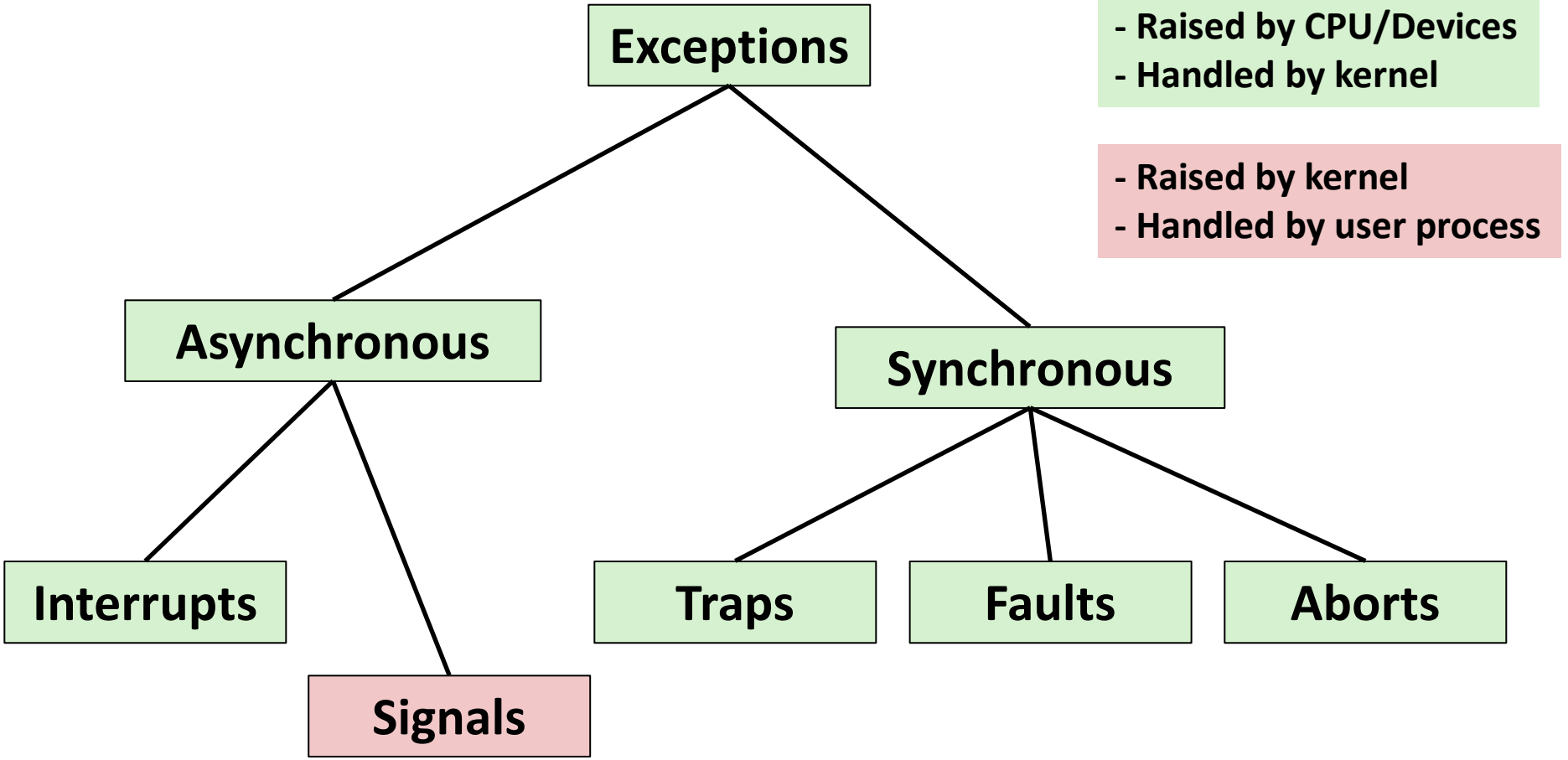
Review (cont.)

- **Spawning processes**
  - Call `fork`
  - One call, two returns
- **Process completion**
  - Call `exit`
  - One call, no return
- **Reaping and waiting for processes**
  - Call `wait` or `waitpid`
- **Loading and running programs**
  - Call `execve` (or variant)
  - One call, (normally) no return

`execve`: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
  - Executable file **`filename`**
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=blee`)
    - `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

(partial) Taxonomy

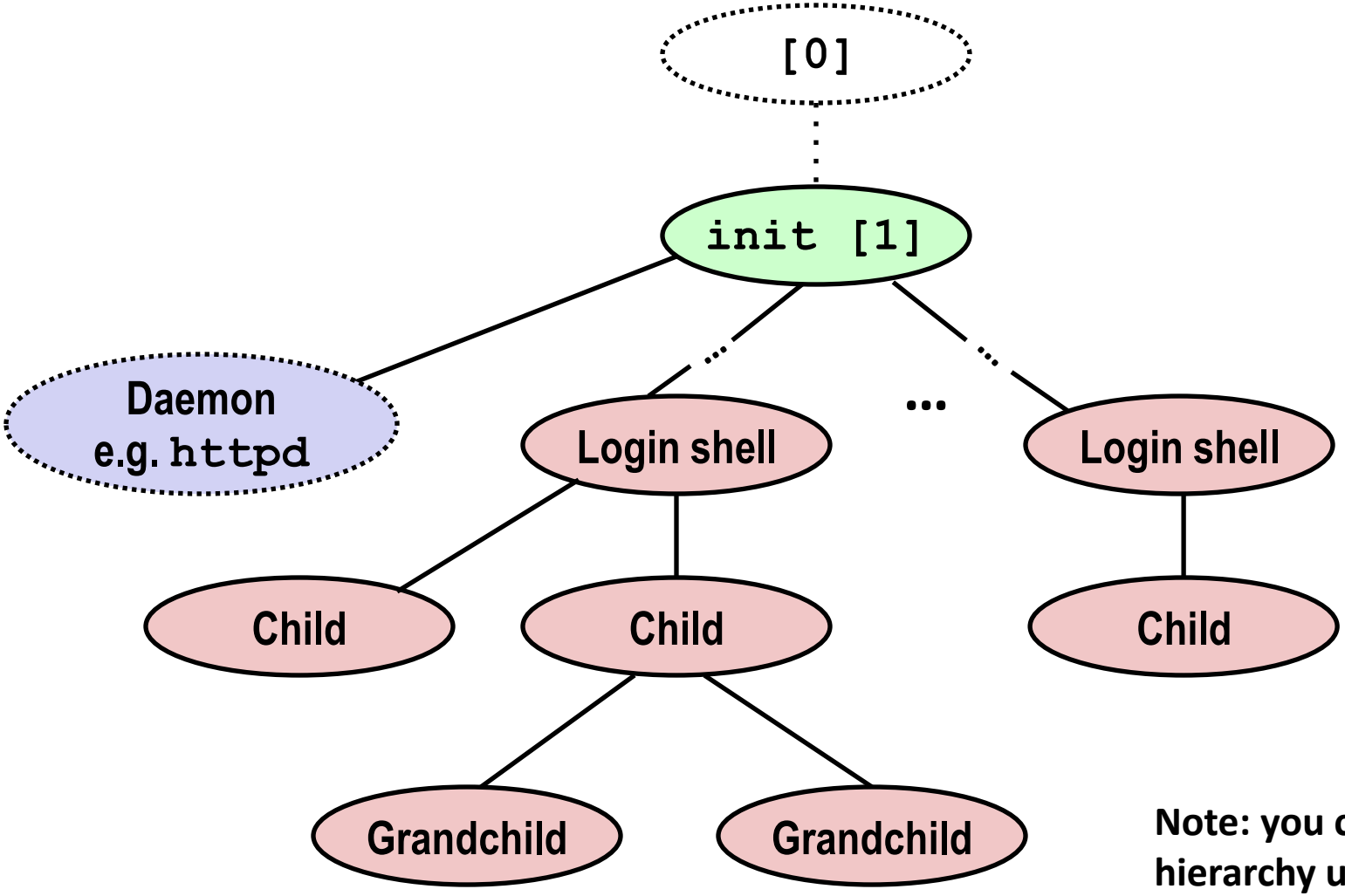


Today

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



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)

- **Simple shell**
  - Described in the textbook, starting at p. 753
  - Implementation of a very elementary shell
  - Purpose
    - Understand what happens when you type commands
    - Understand use and operation of process control operations



Simple Shell Example

```
$ ./shellex
> /bin/ls -l csapp.c
-rw-r--r-- 1 bryant users 23053 Jun 15  2015 csapp.c
> /bin/ps
  PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32017 pts/2    00:00:00 shellex
32019 pts/2    00:00:00 ps
> /bin/sleep 10 &
32031 /bin/sleep 10 &
> /bin/ps
 PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32024 pts/2    00:00:00 emacs
32030 pts/2    00:00:00 shellex
32031 pts/2    00:00:00 sleep
32033 pts/2    00:00:00 ps
> quit
```

Must give full pathnames for programs

Run program in background

Sleep is running in background

Simple Shell Implementation

- **Basic loop**
  - Read line from command line
  - Execute the requested operation
    - Built-in command (only one implemented is **`quit`**)
    - Load and execute program from file

```
int main(int argc, char** argv)
{
    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

```
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);
```

`parseline` will

1) parse ‘buf’ into ‘argv’
2) return whether input line ended in ‘&’

shellex.c

Simple Shell `eval` Function

```
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 */
```

Ignore empty lines.

shellex.c

Simple Shell `eval` Function

```
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 it is a ‘built in’ command, then handle it here in this program.

Otherwise fork/exec the program specified in argv[0]

shellex.c

Simple Shell `eval` Function

```
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 */
```

Create child

shellex.c

Simple Shell `eval` Function

```
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);
            }
        }
```

Start **`argv[0]`**.

Remember **`execve`** only returns on error.

shellex.c

Simple Shell `eval` Function

```
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");
        }
```

If running child in foreground, wait until it is done.

shellex.c

Simple Shell `eval` Function

```
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;
}
```

If running child in background, print pid and continue doing other stuff.

Simple Shell `eval` Function

```
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;
}
```

Oops. *There is a problem with this code.*

shellex.c

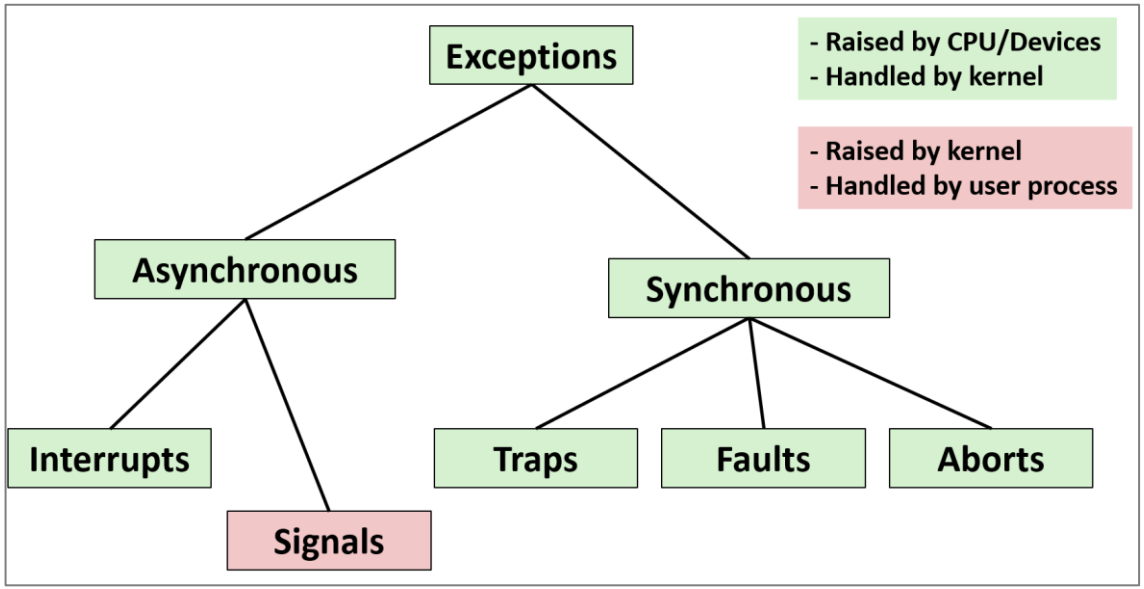
Problem with Simple Shell Example

- **Shell designed to run indefinitely**
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- **In the previous 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



Signal comes to the rescue!

- **Solution: Exceptional control flow**
  - The kernel will “signal” a designated process to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***



Today

- Shells
- **Signals**

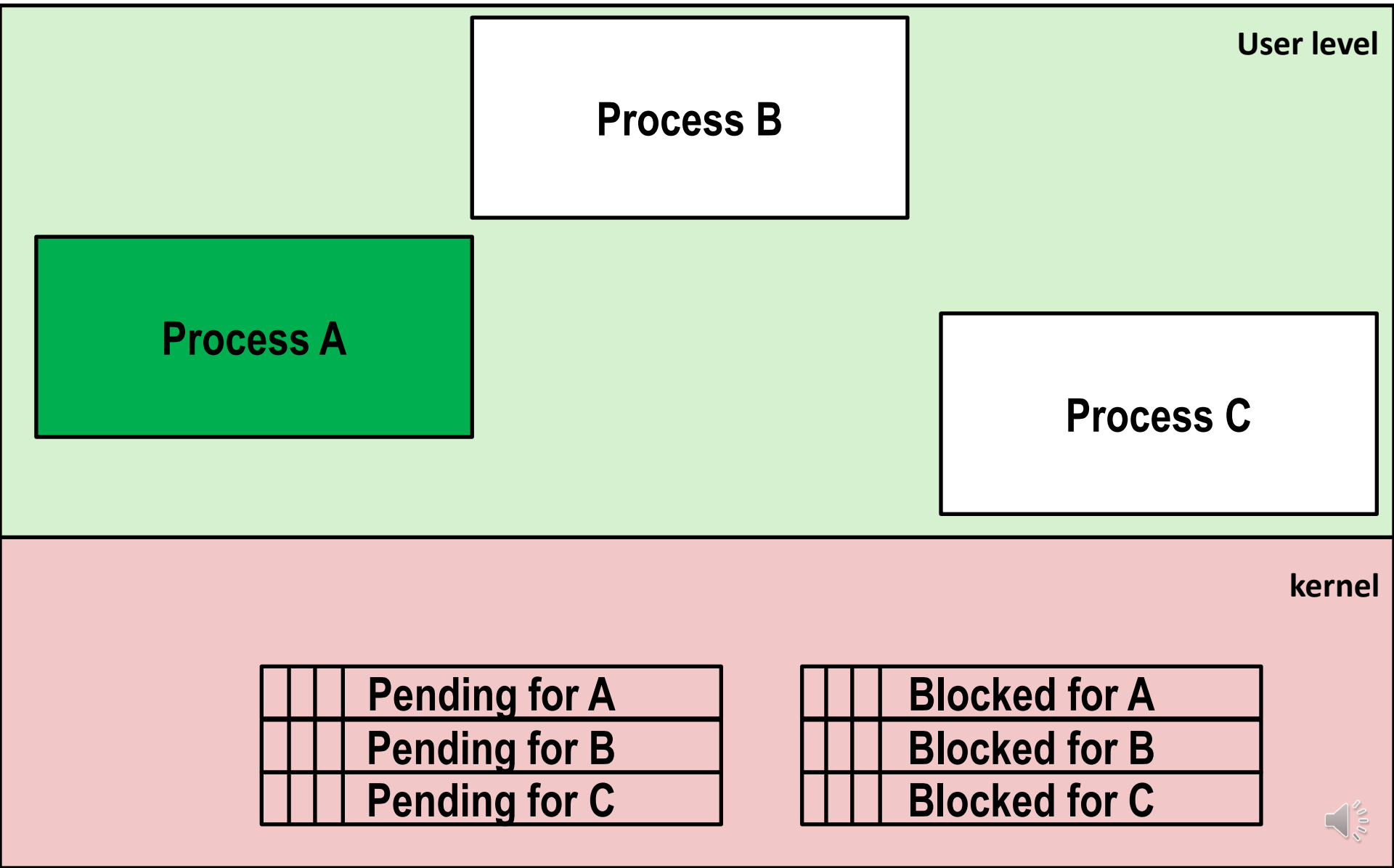


Signals

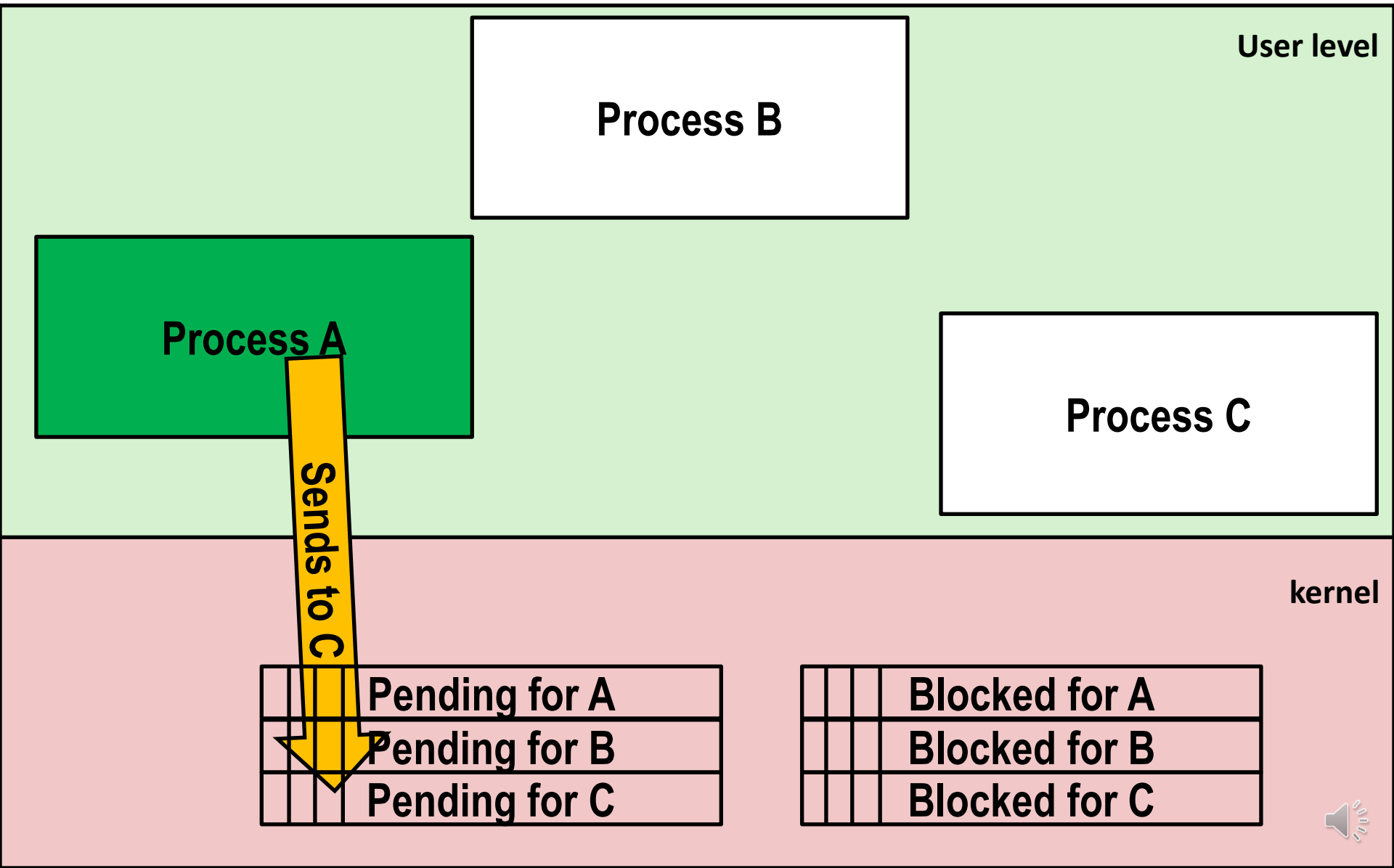
- **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - Akin to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only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

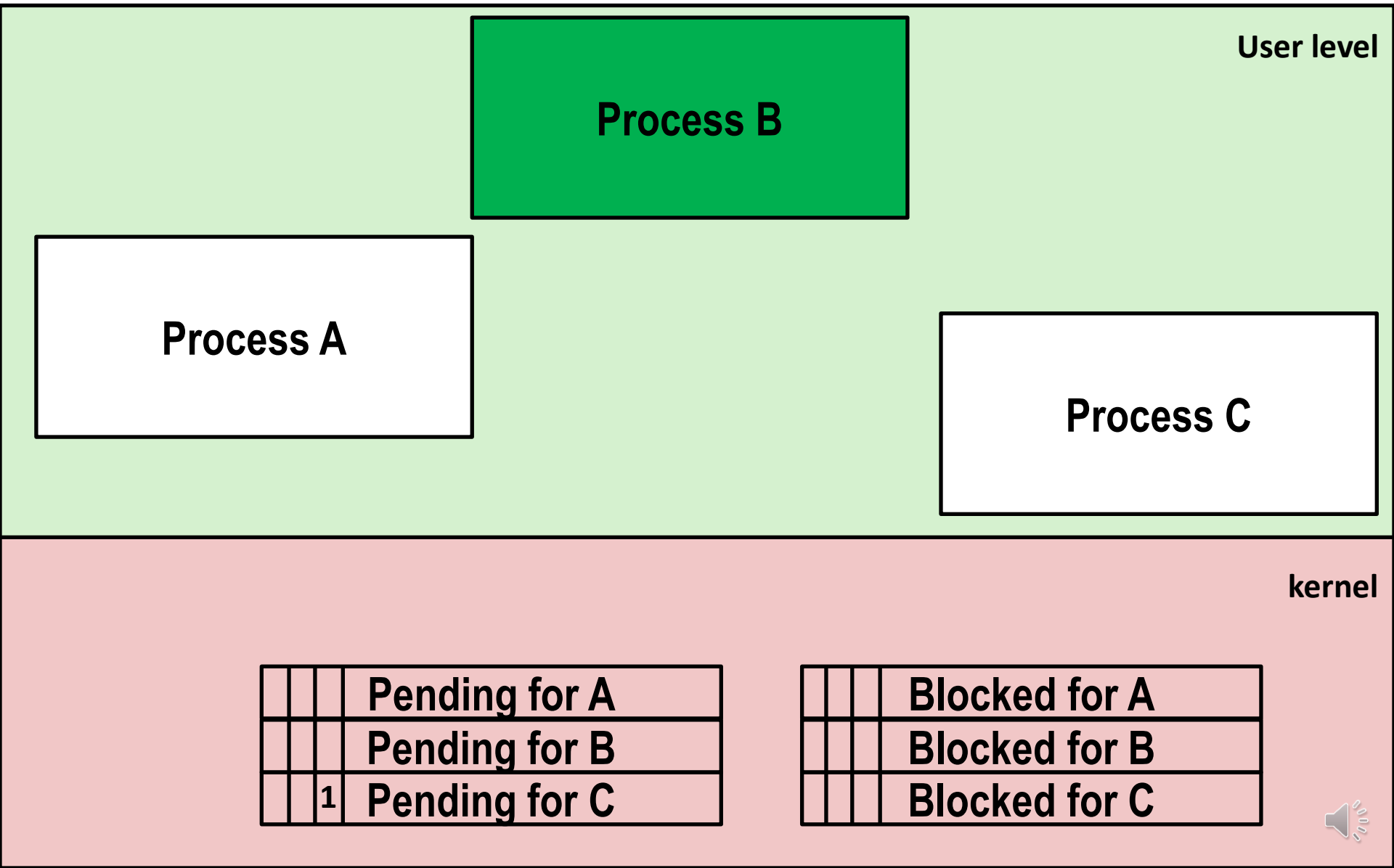
Signal Concepts: Sending a Signal



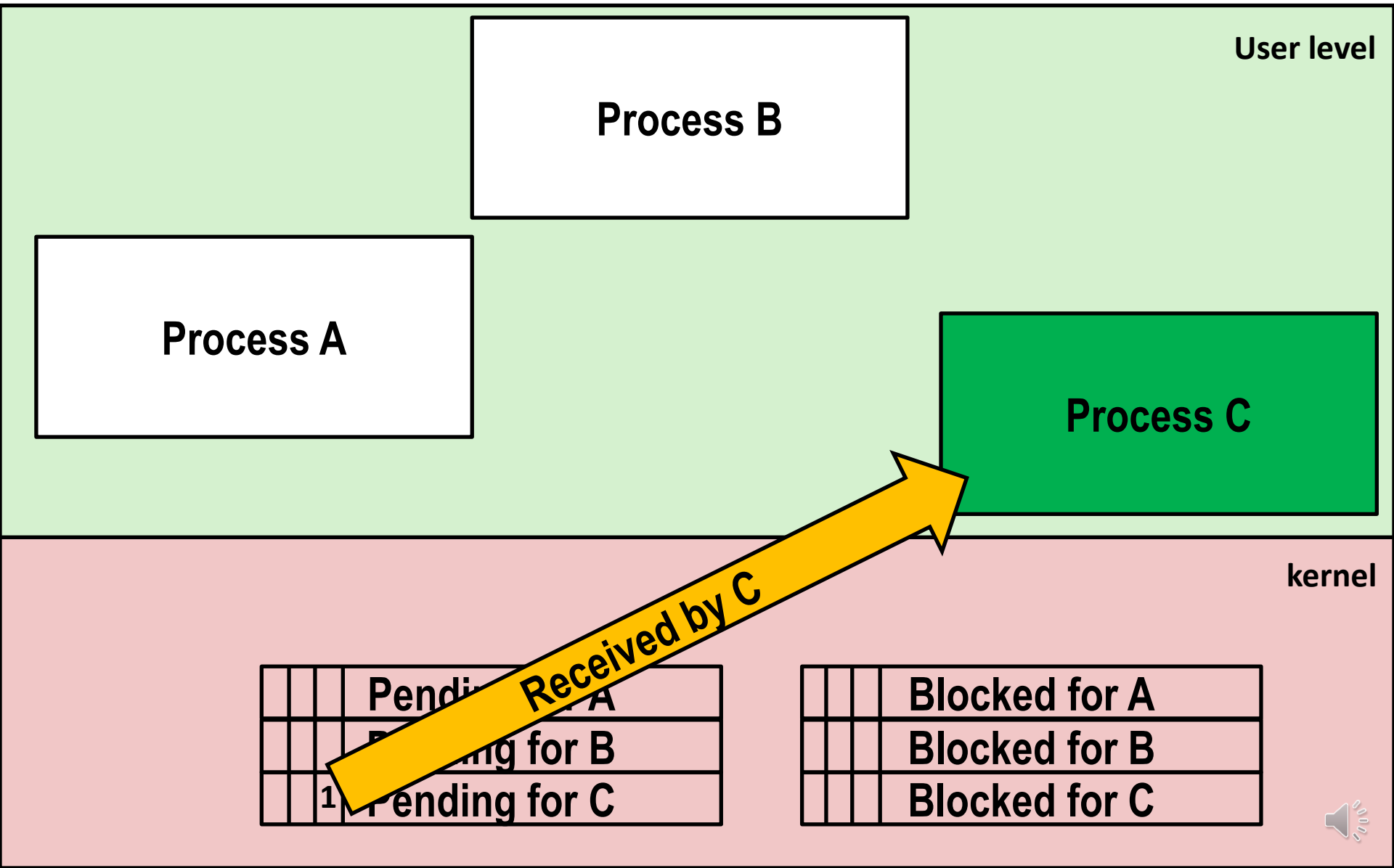
Signal Concepts: Sending a Signal



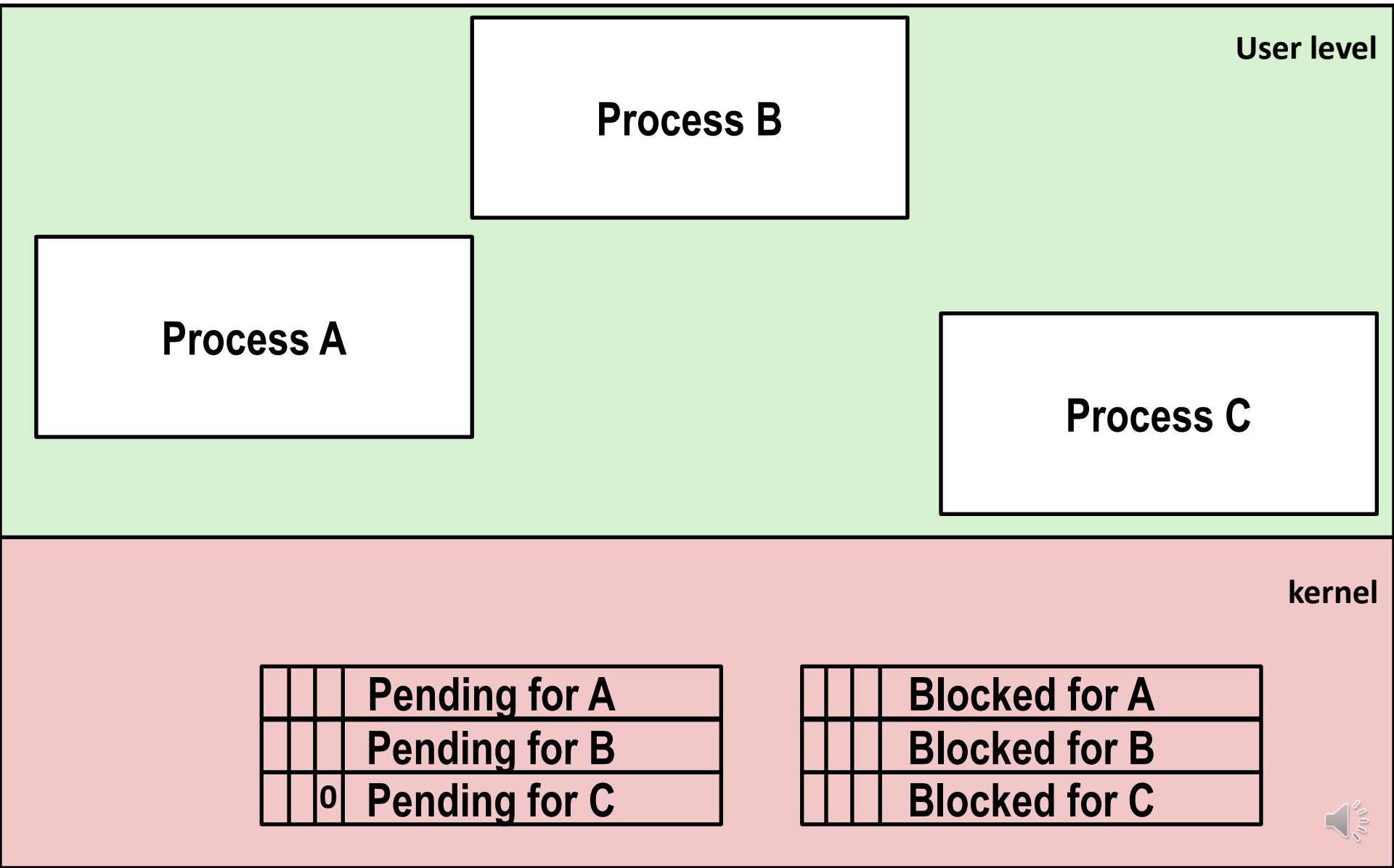
Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal



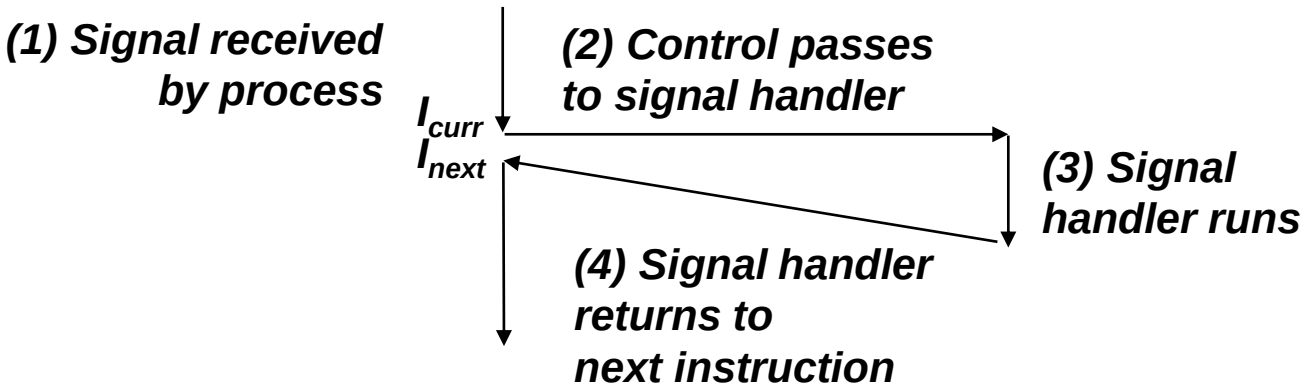
Signal Concepts: Sending a Signal



Signal Concepts: Receiving a Signal

- **A destination process *receives* a signal when it is sent by the kernel**

- **Some possible ways to react by the destination process:**
  - ***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:



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 pended, and it will be received after the signal is unblocked

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

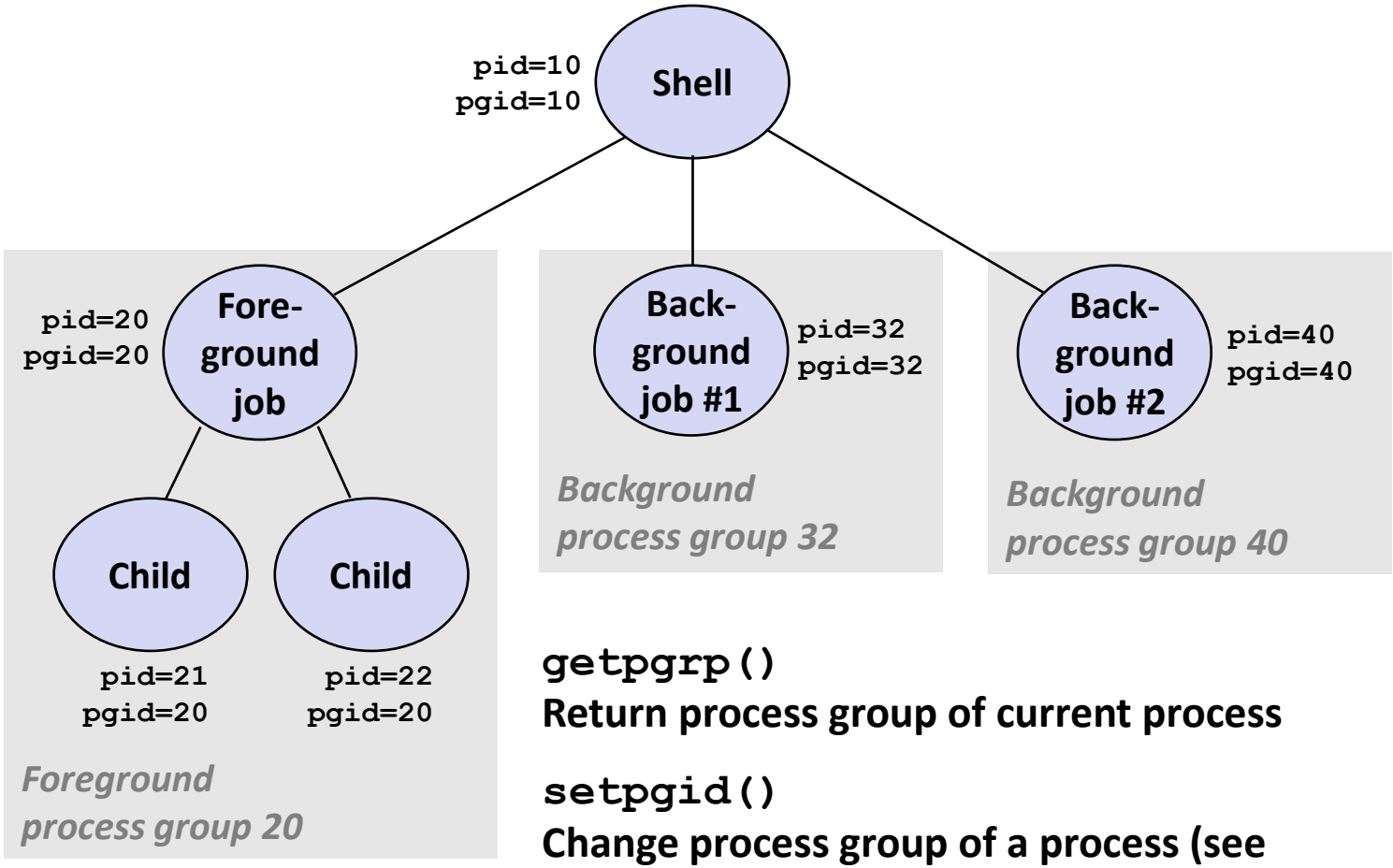


Signal Concepts: Pending/Blocked Bits

- **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`** syscall
    - Also referred to as the *signal mask*.

Sending Signals: Process Groups

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



`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

```
$ ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

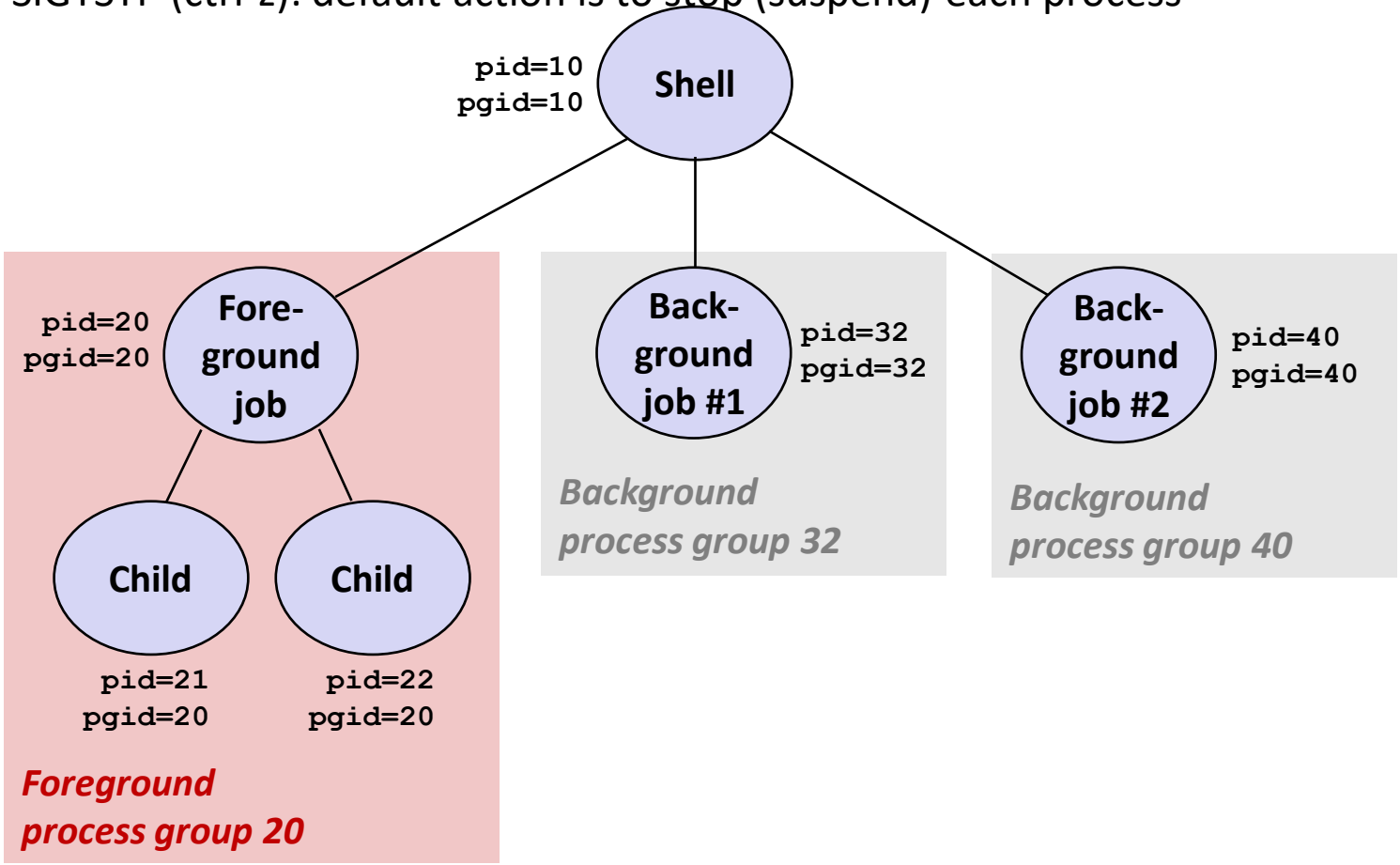
$ ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps

$ /bin/kill -9 -24817

$ ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
```

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 (ctrl-c): default action is to terminate each process
  - SIGTSTP (ctrl-z): default action is to stop (suspend) each process



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

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

$ 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

$ fg
./forks 17
<types ctrl-c>

$ 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` syscall

```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

```
kill(2)

NAME
       kill - send signal to a process

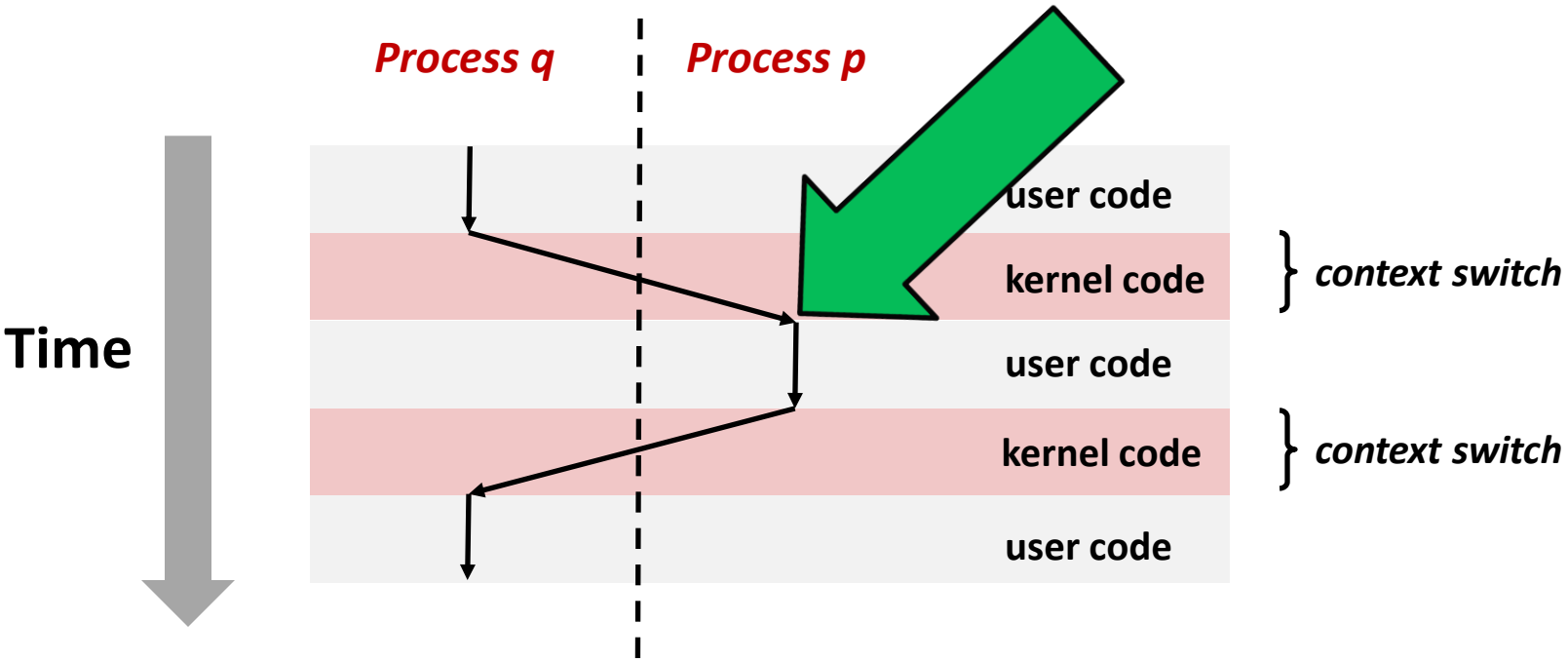
LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <signal.h>

       int kill(pid_t pid, int sig);
```

Receiving Signals

- The signal is delivered when kernel is about to pass control to process *p*



Receiving Signals

- **The signal is delivered when kernel is about 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**
  - Find 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 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)`

- **Possible 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 ***"dispatching", "catching"*** or ***"handling"*** the signal

Signal Handling Example

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

    /* Sleep until a signal is delivered. */
    pause();

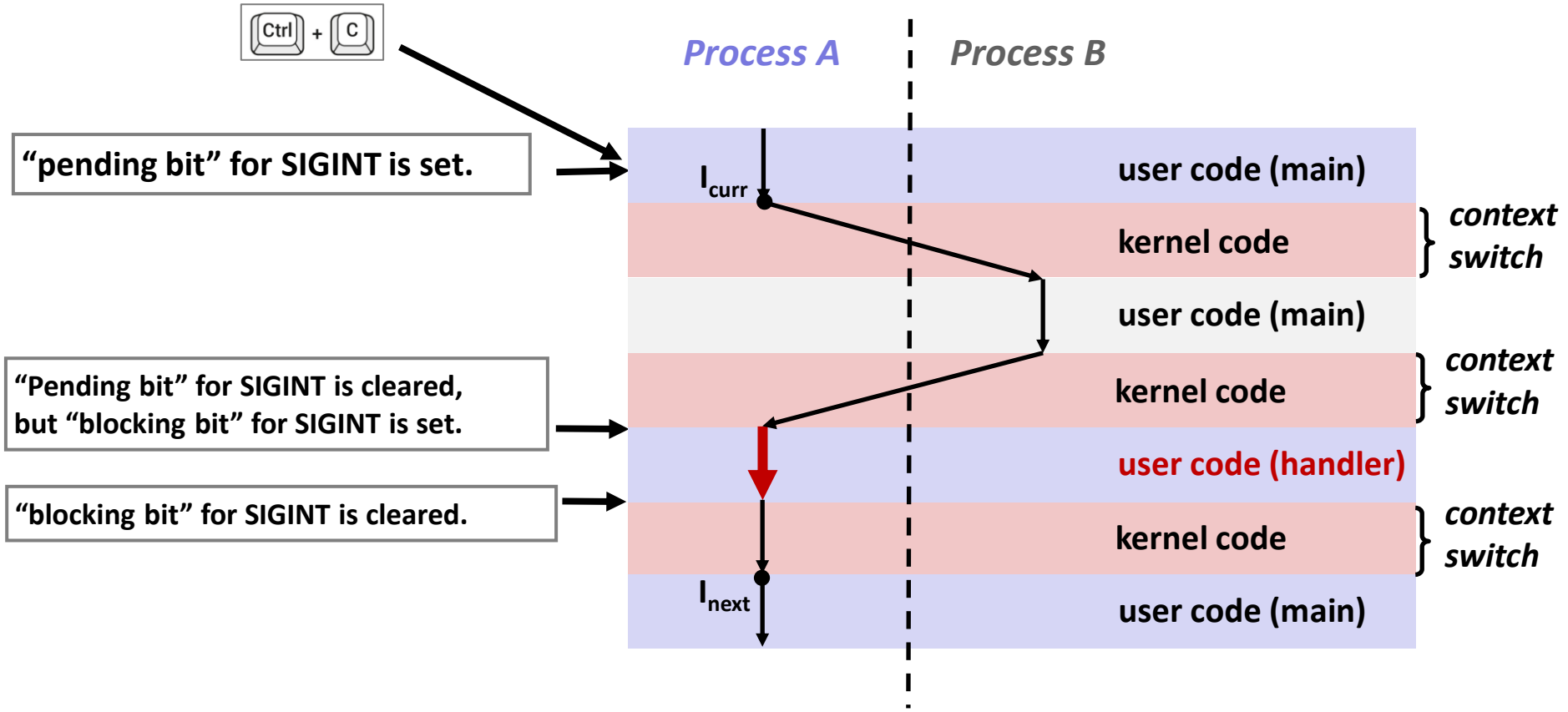
    return 0;
}
```

Q. What would happen if you keep typing “ctrl-C”?

sigint.c



Signal Handlers as Concurrent Flows



Example: Nested and Same Signals

```
void check_sig_status(void) {
    sigset_t sigset;
    bool is_pending;
    bool is_blocked;

    if (sigpending(&sigset) != 0)
        perror("sigpending() error");

    if (sigismember(&sigset, SIGINT))
        is_pending = true;
    else
        is_pending = false;

    if (sigprocmask(SIG_BLOCK, NULL, &sigset) != 0)
        perror("sigprocmask() error");

    if (sigismember(&sigset, SIGINT))
        is_blocked = true;
    else
        is_blocked = false;

    printf("STATUS: pending [%d] block [%d]\n",
           is_pending, is_blocked);
}
```

```
void sigint_handler(int sig)
{
    static int count = 0;
    printf("[*] [%d] sigint_handler() START\n",
           ++count);
    check_sig_status();
    sleep(3);
    check_sig_status();
    printf("[*] [%d] sigint_handler() END\n",
           count);
}

int main(void)
{
    check_sig_status();

    signal(SIGINT, sigint_handler);

    check_sig_status();
    while (1)
        pause();
    return 0;
}
```

Example: Nested and Same Signals

- **SIGINT**

```
$ ./check_sig
STATUS: pending [0] block [0]
STATUS: pending [0] block [0]
^C
[*] [1] sigint_handler() START
STATUS: pending [0] block [1]
STATUS: pending [0] block [1]
[*] [1] sigint_handler() END
```

- **Nested SIGINT**

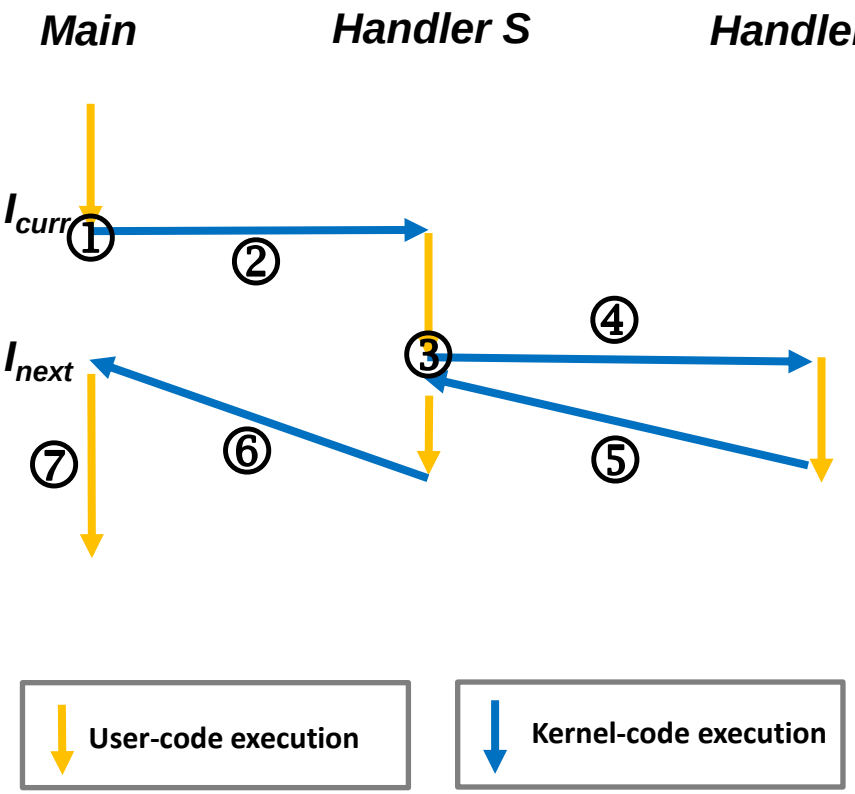
```
$ ./check_sig
STATUS: pending [0] block [0]
STATUS: pending [0] block [0]
^C
[*] [1] sigint_handler() START
STATUS: pending [0] block [1]
^C
STATUS: pending [1] block [1]
[*] [1] sigint_handler() END
[*] [2] sigint_handler() START
STATUS: pending [0] block [1]
STATUS: pending [0] block [1]
[*] [2] sigint_handler() END
```

- **Nested SIGINT (two nested SIGINT)**

```
$ ./check_sig
STATUS: pending [0] block [0]
STATUS: pending [0] block [0]
^C
[*] [1] sigint_handler() START
STATUS: pending [0] block [1]
^C
^C
STATUS: pending [1] block [1]
[*] [1] sigint_handler() END
[*] [2] sigint_handler() START
STATUS: pending [0] block [1]
STATUS: pending [0] block [1]
[*] [2] sigint_handler() END
```

Nested and Different Signals

- Handlers can be interrupted by other handlers



① While scheduling, kernel notices "signal S" is pending.

② Kernel dispatches "signal S".

③ While scheduling, kernel notices "signal T" is pending.

④ Kernel dispatches "Signal T".

⑤ Handler T returns to the kernel (i.e., sigreturn).
➔ Kernel resumes the process execution.

⑥ Handler S returns to the kernel.
➔ Kernel resumes the process execution.

⑦ The process continues its execution

Blocking and Unblocking Signals

- **Implicit blocking mechanism**
  - Kernel blocks any pending signals of the type currently being handled
  - e.g., a SIGINT handler can't be interrupted by another SIGINT
  - Q. Do you see the difference between
    - Nested and same signals
    - Nested and different signals

- **Explicit blocking and unblocking mechanism**
  - `sigprocmask` function

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**
- **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`, and `malloc` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other nested 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
- **Check the textbook for more details!**



Wrong Example

```
enum { MAX_MSG_SIZE = 24 };
char *err_msg;

void handler(int signum) {
  strcpy(err_msg, "SIGINT encountered.");
}

int main(void) {
  signal(SIGINT, handler);

  err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  return 0;
}
```

Correct Example

```
enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
  e_flag = 1;
}

int main(void) {
  char *err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }

  signal(SIGINT, handler);
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  if (e_flag) {
    strcpy(err_msg, "SIGINT received.");
  }
  return 0;
}
```

Async-Signal-Safety

- **Function is *async-signal-safe* if either reentrant or non-interruptible by signals**
  - Reentrant: All variables stored on stack frame, so it's not using global/heap variables to maintain its state. [CS:APP3e 12.7.2]

- **Posix guarantees 117 functions to be async-signal-safe**
  - Source: "`man 7 signal-safety`"
  - Popular async-signal-safe functions:
    - `_exit, write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** async-signal-safe:
    - `printf, sprintf, malloc`
    - Unfortunate fact: `write` is the only async-signal-safe output function

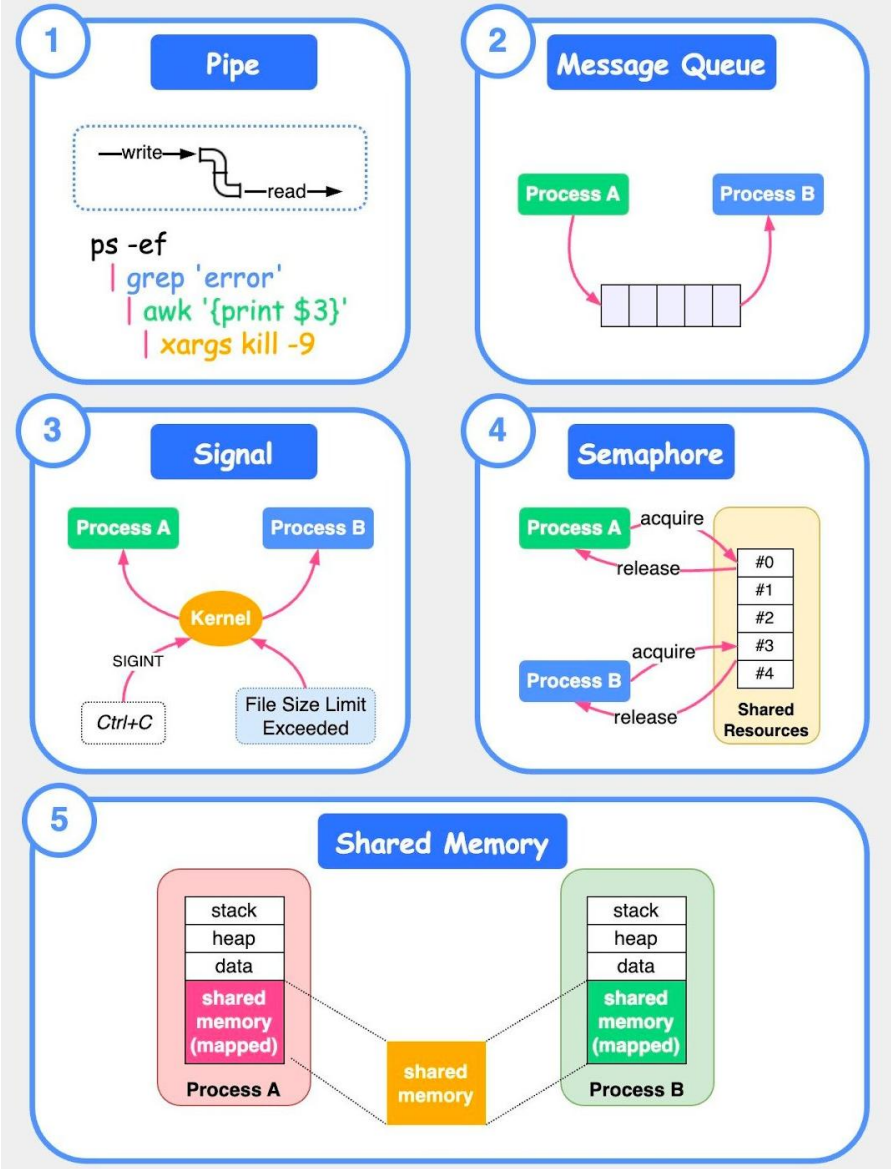


CVE-2024-6387



This signal handler (SIGALRM) is designed to close the connection, but it mistakenly calls functions like `syslog()`, which are not safe to execute in this asynchronous context. These functions can invoke other non-async-signal-safe functions like `malloc()` and `free()`, leading to inconsistent states and potential heap corruption. Consequently, an attacker can exploit this vulnerability to execute arbitrary code on the server, resulting in remote code execution with root privileges. This type of vulnerability is critical because it allows unauthenticated remote attackers to gain full control over the affected system.

Inter-Process Communication (IPC)



blog.bytebytego.com

Summary

- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Be very careful when writing signal handlers



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

volatile sig_atomic_t e_flag = 0;

void handle_signal(int signal) {
    e_flag = 1;  // Set flag to signal main program.
}

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

    while (!e_flag) {
        printf("Running...\n");
        sleep(1);
    }

    printf("Signal received, exiting.\n");
    return 0;
}
```