

# Exceptional Control Flow

15-213/15-513: Introduction to Computer Systems  
17<sup>th</sup> Lecture, March 21, 2024

# Today

- **Exceptional Control Flow**
- Exceptions
- Signals
- If we have time: Nonlocal Jumps

# Problem with Simple Shell Example

- **Shell designed to run indefinitely**
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- **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
  - Could run the entire computer out of memory
    - More likely, run out of PIDs

# Printers Used to Catch on Fire



# Highly Exceptional Control Flow

```

234 static int lp_check_status(int minor)
235 {
236     int error = 0;
237     unsigned int last = lp_table[minor].last_error;
238     unsigned char status = r_str(minor);
239     if ((status & LP_PERRORP) && !(LP_F(minor) & LP_CAREFUL))
240         /* No error. */
241         last = 0;
242     else if ((status & LP_POUTPA)) {
243         if (last != LP_POUTPA) {
244             last = LP_POUTPA;
245             printk(KERN_INFO "lp%d out of paper\n", minor);
246         }
247         error = -ENOSPC;
248     } else if (!(status & LP_PSELECD)) {
249         if (last != LP_PSELECD) {
250             last = LP_PSELECD;
251             printk(KERN_INFO "lp%d off-line\n", minor);
252         }
253         error = -EIO;
254     } else if (!(status & LP_PERRORP)) {
255         if (last != LP_PERRORP) {
256             last = LP_PERRORP;
257             printk(KERN_INFO "lp%d on fire\n", minor);
258         }
259         error = -EIO;
260     } else {
261         last = 0; /* Come here if LP_CAREFUL is set and no
262                 errors are reported. */
263     }
264
265     lp_table[minor].last_error = last;
266
267     if (last != 0)
268         lp_error(minor);
269
270     return error;
271 }

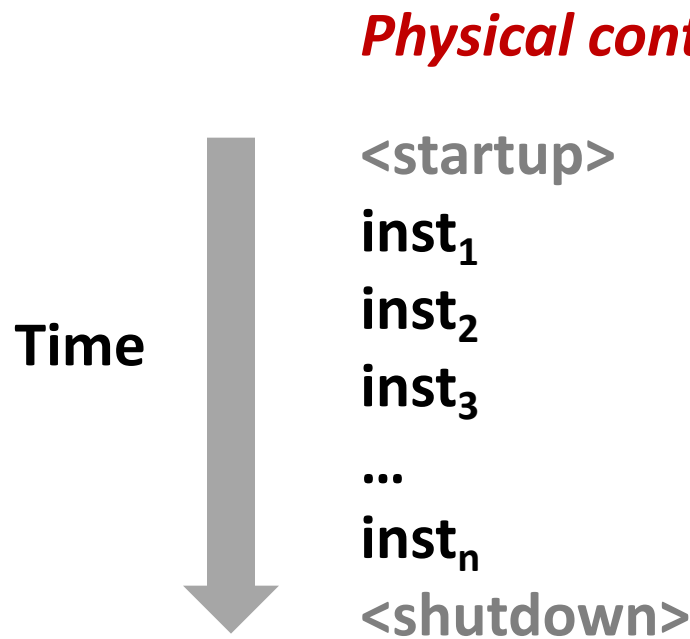
```

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/char/lp.c?h=v5.0-rc3>

# Control Flow

## ■ Processors do only one thing:

- From startup to shutdown, each CPU core simply reads and executes (interprets) a sequence of instructions, one at a time \*
- This sequence is the CPU's *control flow* (or *flow of control*)



- \* Externally, from an architectural viewpoint (internally, the CPU may use parallel out-of-order execution)

# Altering the Control Flow

## ■ Up to now: two mechanisms for changing control flow:

- Jumps and branches
- Call and return

React to changes in *program state*

## ■ Insufficient for a useful system:

Difficult to react to changes in *system state*

- Data arrives from a disk or a network adapter
- Instruction divides by zero
- User hits Ctrl-C at the keyboard
- System timer expires

## ■ System needs mechanisms for “exceptional control flow”

# Exceptional Control Flow

- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

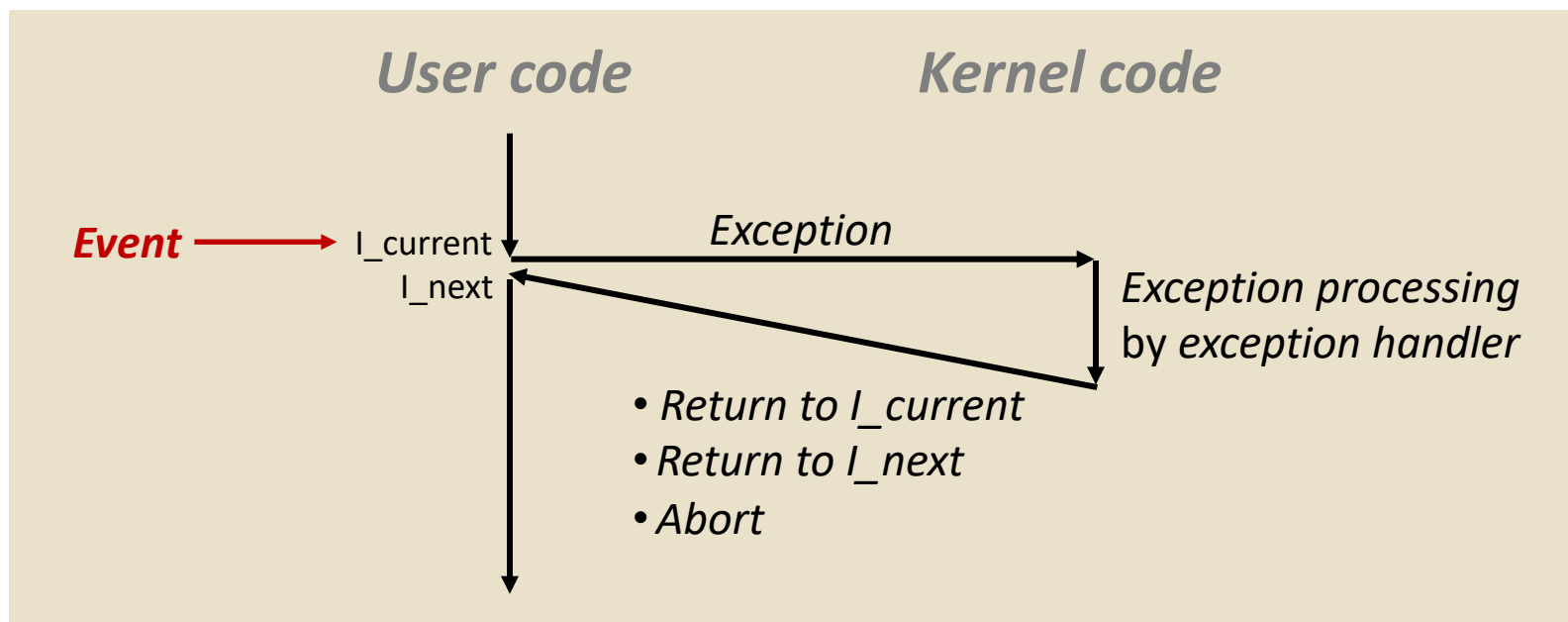


# Today

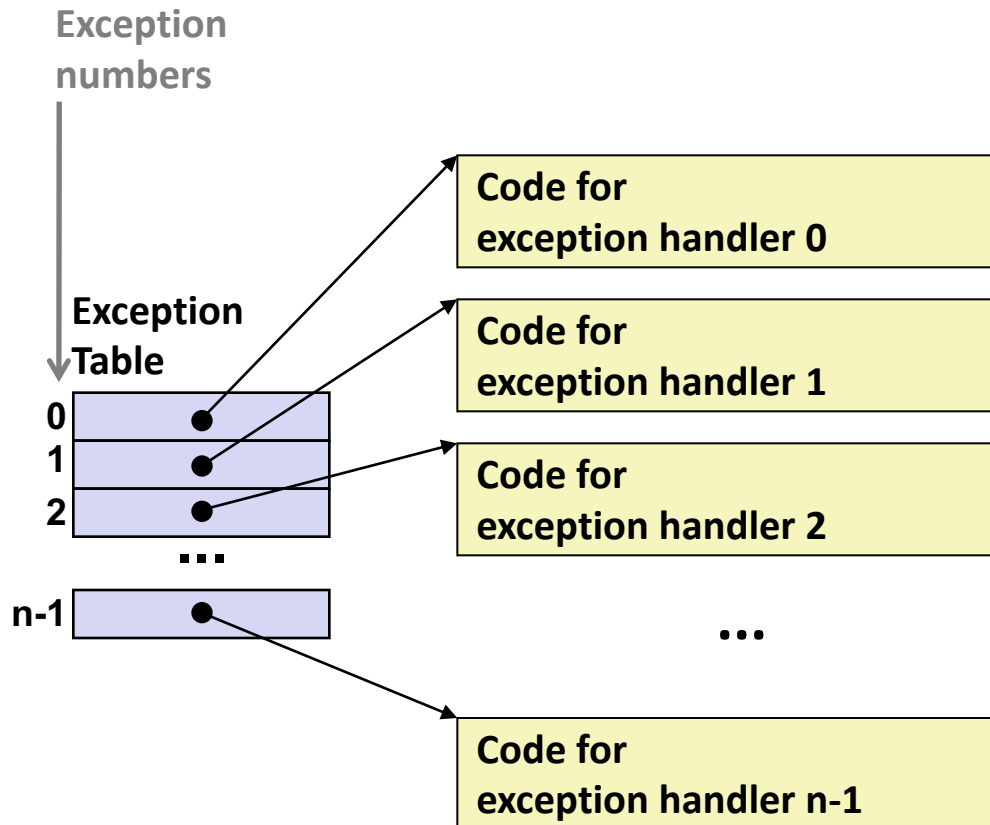
- Exceptional Control Flow
- **Exceptions**
- Signals
- If we have time: Nonlocal Jumps

# Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

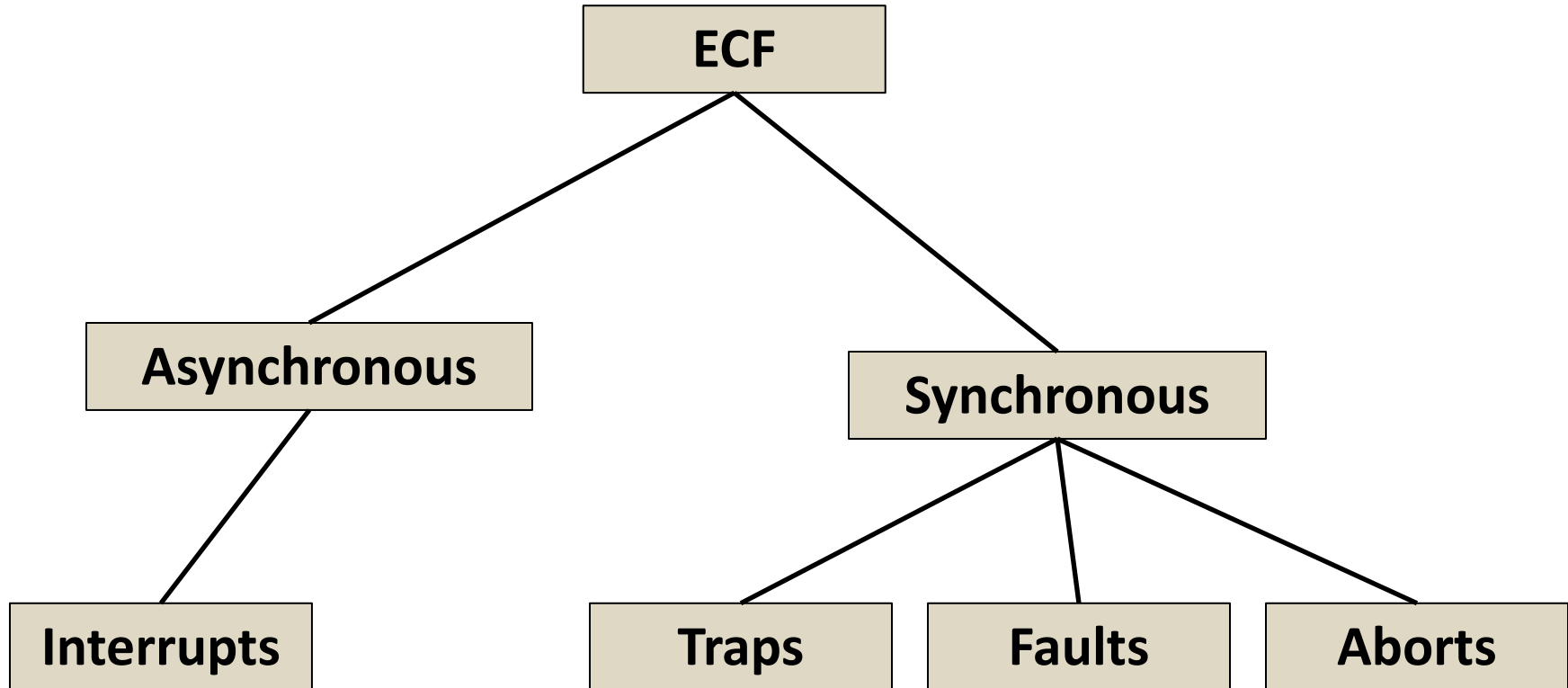


# Exception Tables



- Each type of event has a unique exception number  $k$
- $k$  = index into exception table (a.k.a. interrupt vector)
- Handler  $k$  is called each time exception  $k$  occurs

# Taxonomy of Hardware ECF



# Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to “next” instruction
- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

# Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional, set program up to “trip the trap” and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

# System Calls

- Each x86-64 system call has a unique ID number
- Examples:

<i>Number</i>	<i>Name</i>	<i>Description</i>
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

# System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

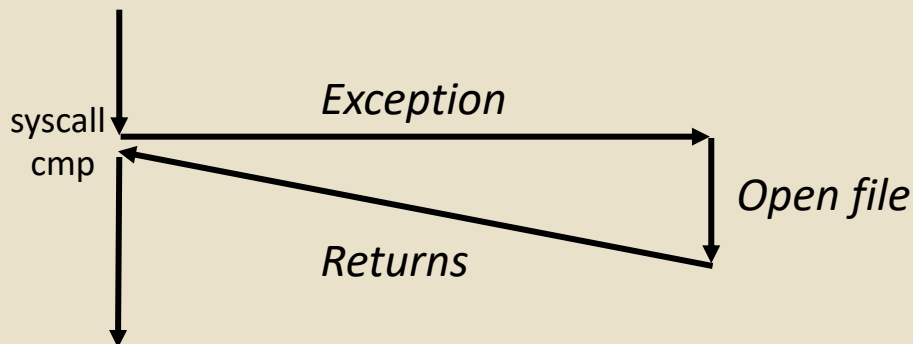
```

0000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00      mov  $0x2,%eax    # open is syscall
#2
e5d7e:  0f 05                syscall
# Return value in %rax
e5d80:  48 3d 01 f0 ff ff    cmp  $0xffffffffffffffff001,%rax
...
e5dfa:  c3                   retq

```

*User code*

*Kernel code*



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`



# System Call

- User calls: `open (f`
- Calls `__open` function

```

000000000000e5d70
...
e5d79:  b8 02 00
e5d7e:  0f 05
%rax
e5d80:  48 3d 01
...
e5dfa:  c3
  
```

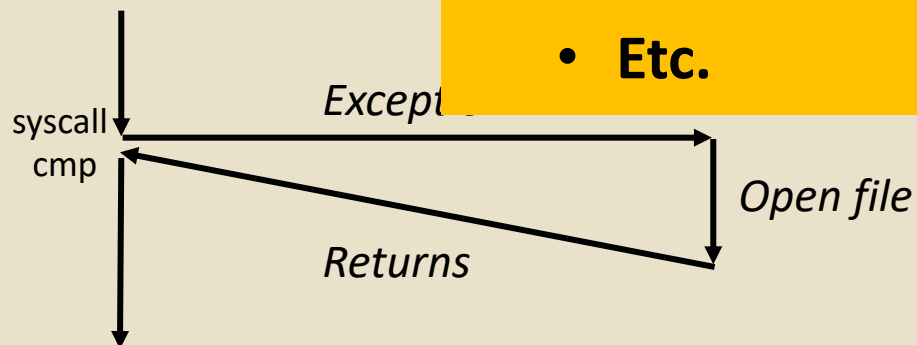
Almost like a function call

- Transfer of control
- On return, executes next instruction
- Passes arguments using calling convention
- Gets result in `%rax`

One Important exception!

- Executed by Kernel
- Different set of privileges
- And other differences:
  - E.g., “address” of “function” is in `%rax`
  - Uses `errno`
  - Etc.

*User code*



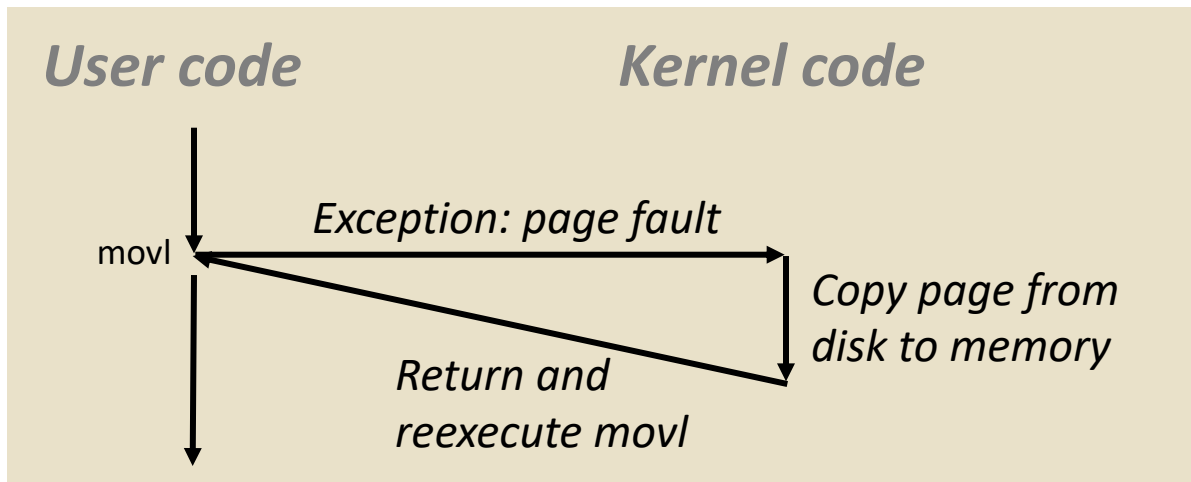
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

# Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

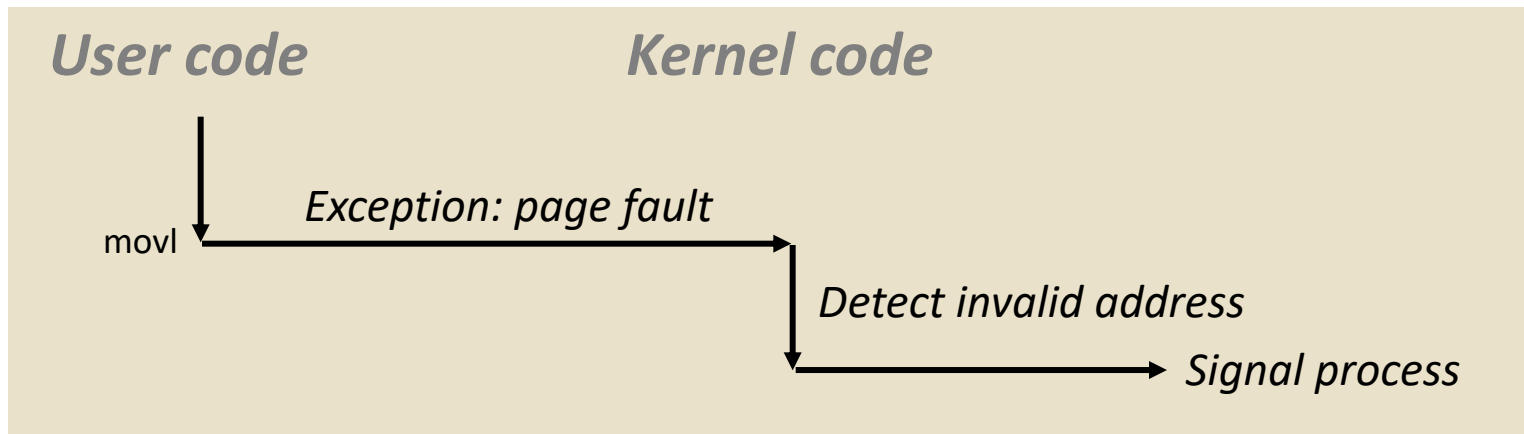
```
80483b7:      c7 05 10 9d 04 08 0d  movl   $0xd,0x8049d10
```



# Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d  movl    $0xd,0x804e360
```



- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”

# Quiz

<https://canvas.cmu.edu/courses/37116/quizzes/109925>

# Today

- Exceptional Control Flow
- Exceptions
- **Signals**
- If we have time: Nonlocal Jumps

# ECF Exists at All Levels of a System

## ■ Exceptions

- Hardware and operating system kernel software

## ■ Process Context Switch

- Hardware timer and kernel software

## ■ Signals

- Kernel software and application software

## ■ Nonlocal jumps

- Application code

# Problem with Simple Shell Example

- **Shell designed to run indefinitely**
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- **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, the alert mechanism is called a *signal*



# Signals

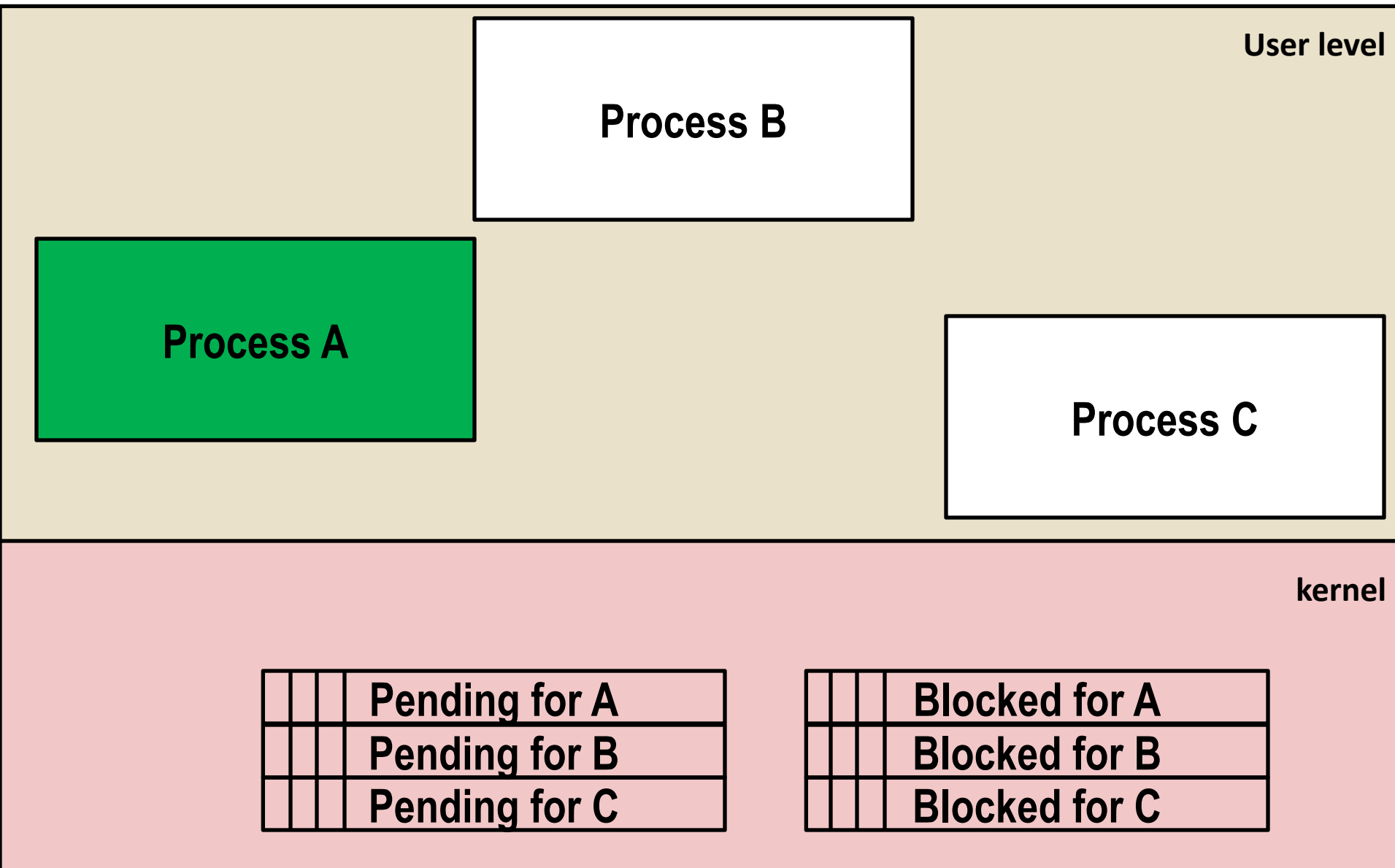
- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - Akin to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only information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

<i>ID</i>	<i>Name</i>	<i>Default Action</i>	<i>Corresponding Event</i>
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

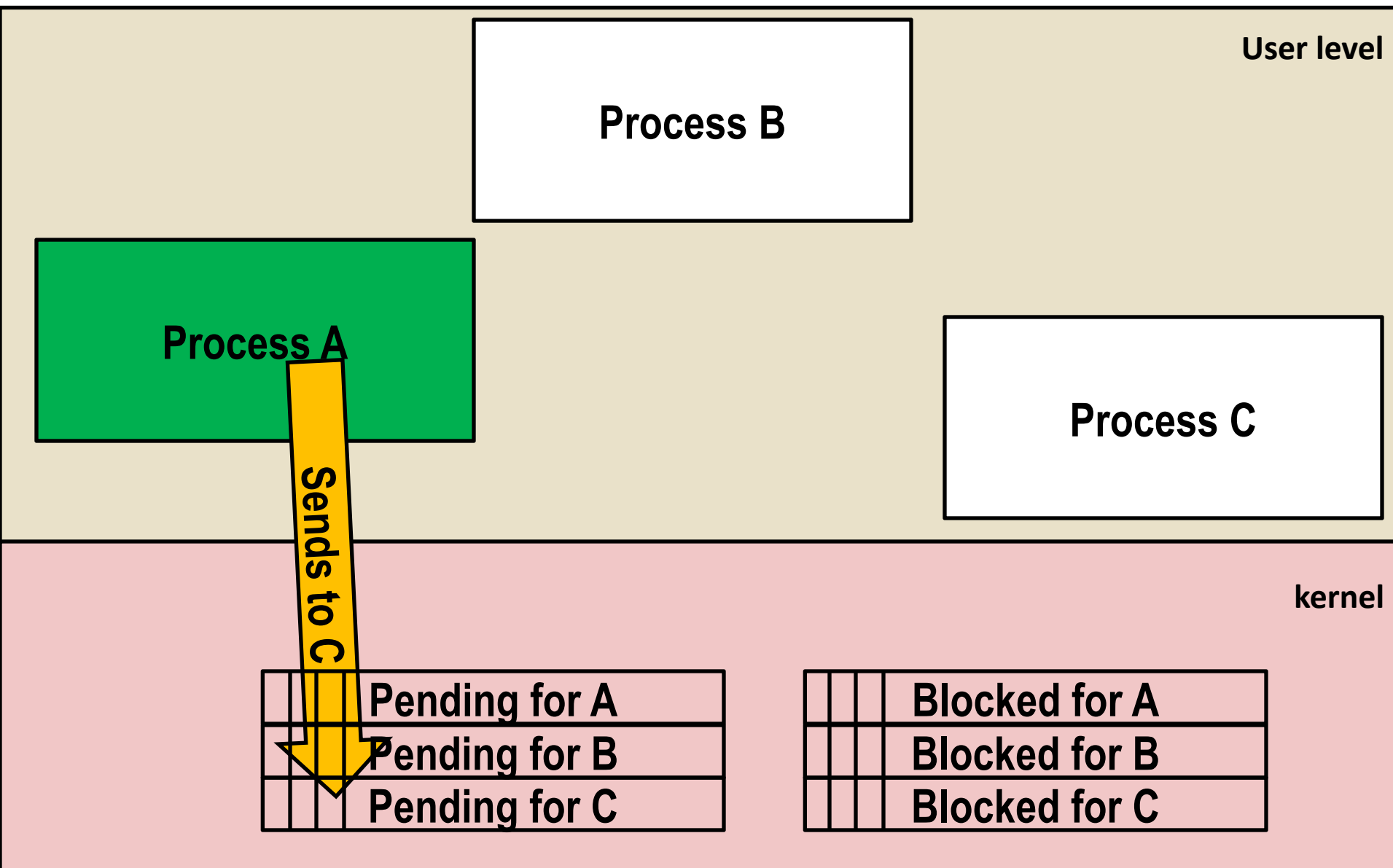
# Signal Concepts: Sending a Signal

- Kernel *sends* 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

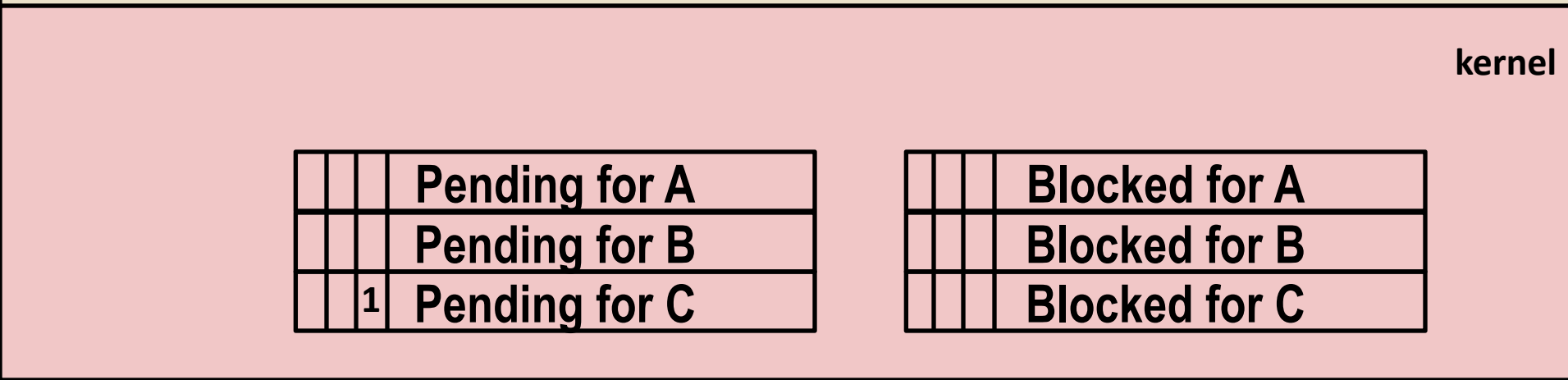
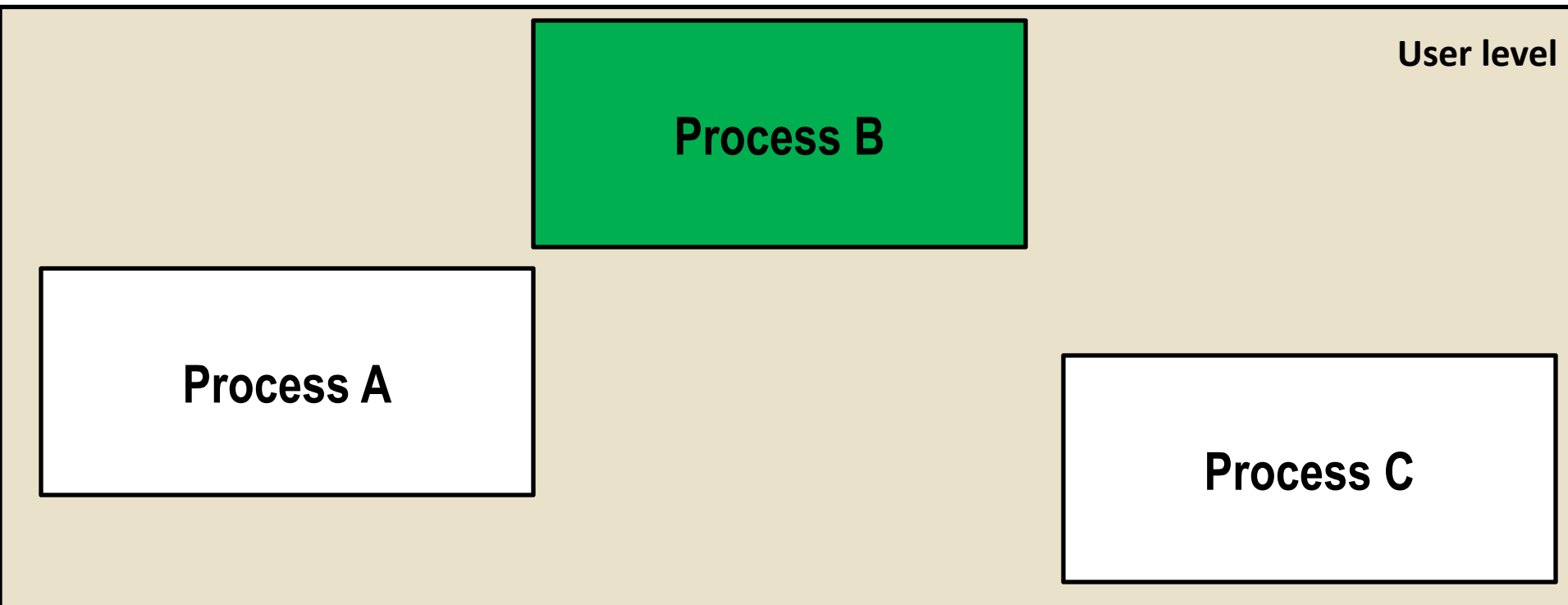
# Signal Concepts: Sending a Signal



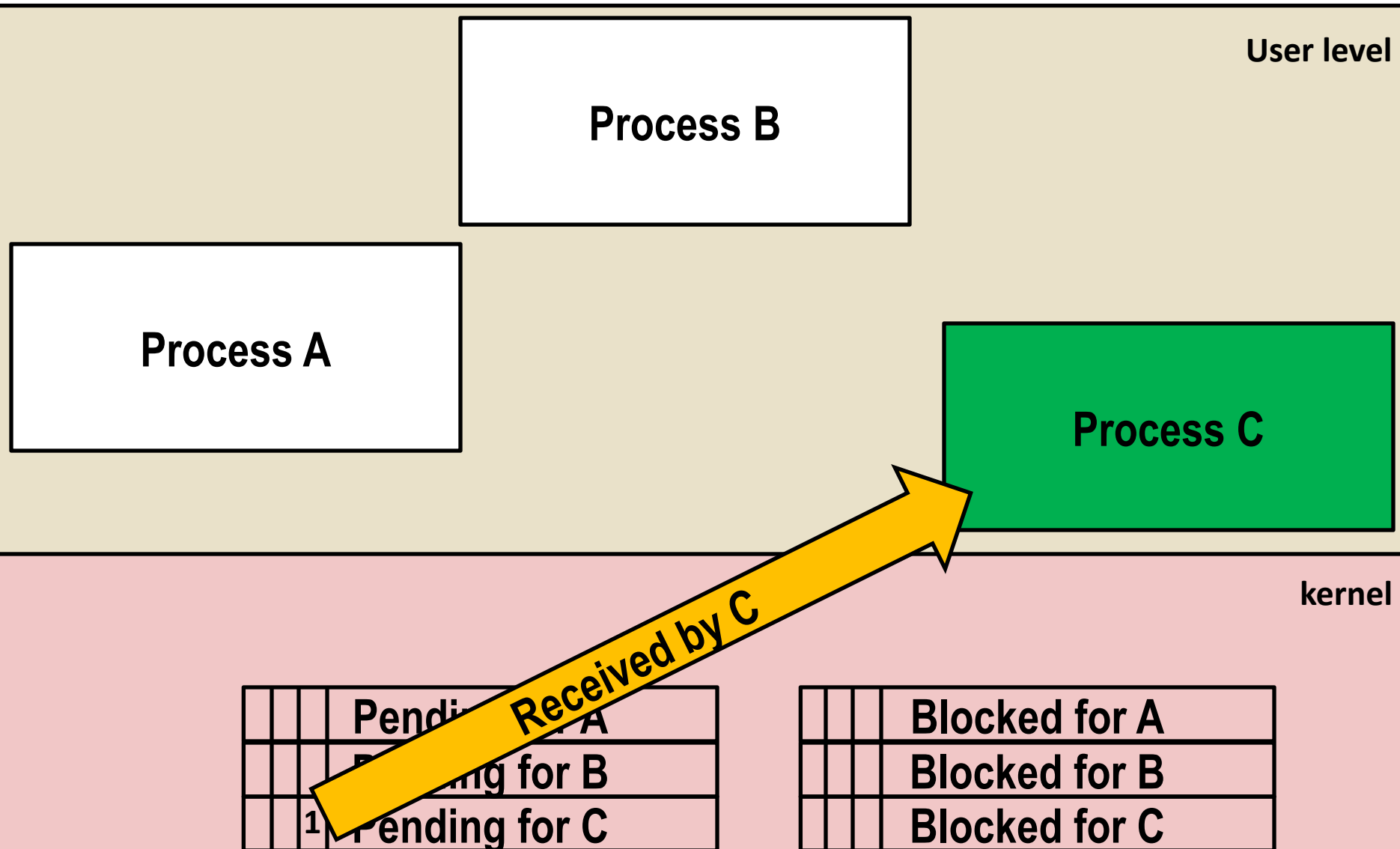
# Signal Concepts: Sending a Signal



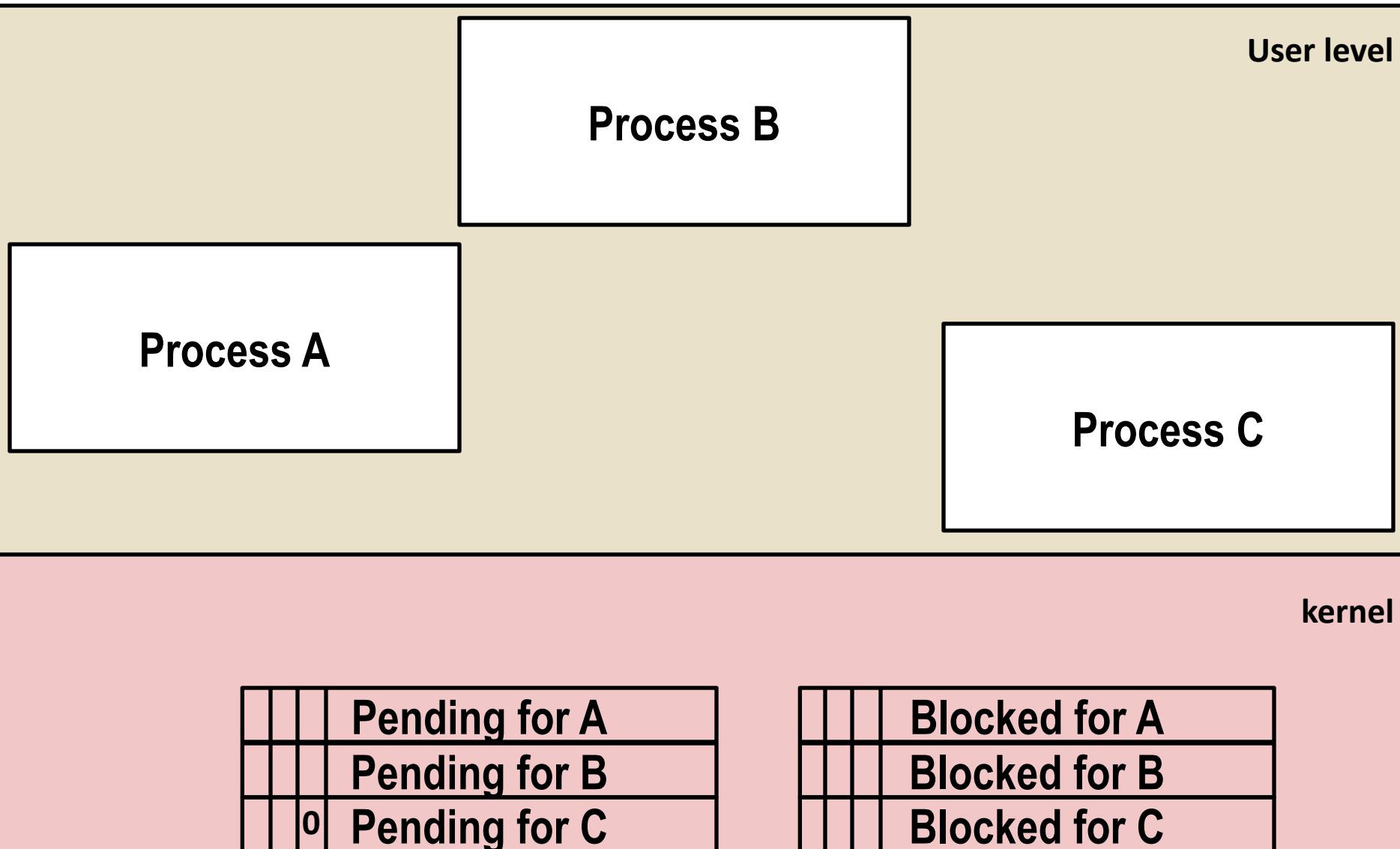
# Signal Concepts: Sending a Signal



# Signal Concepts: Sending a Signal

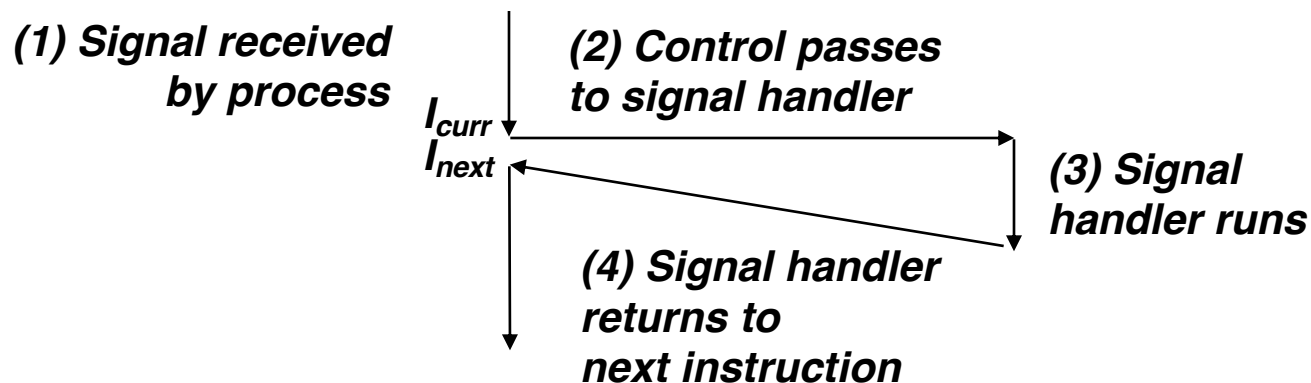


# Signal Concepts: Sending a Signal



# Signal Concepts: Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the signal
- 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:





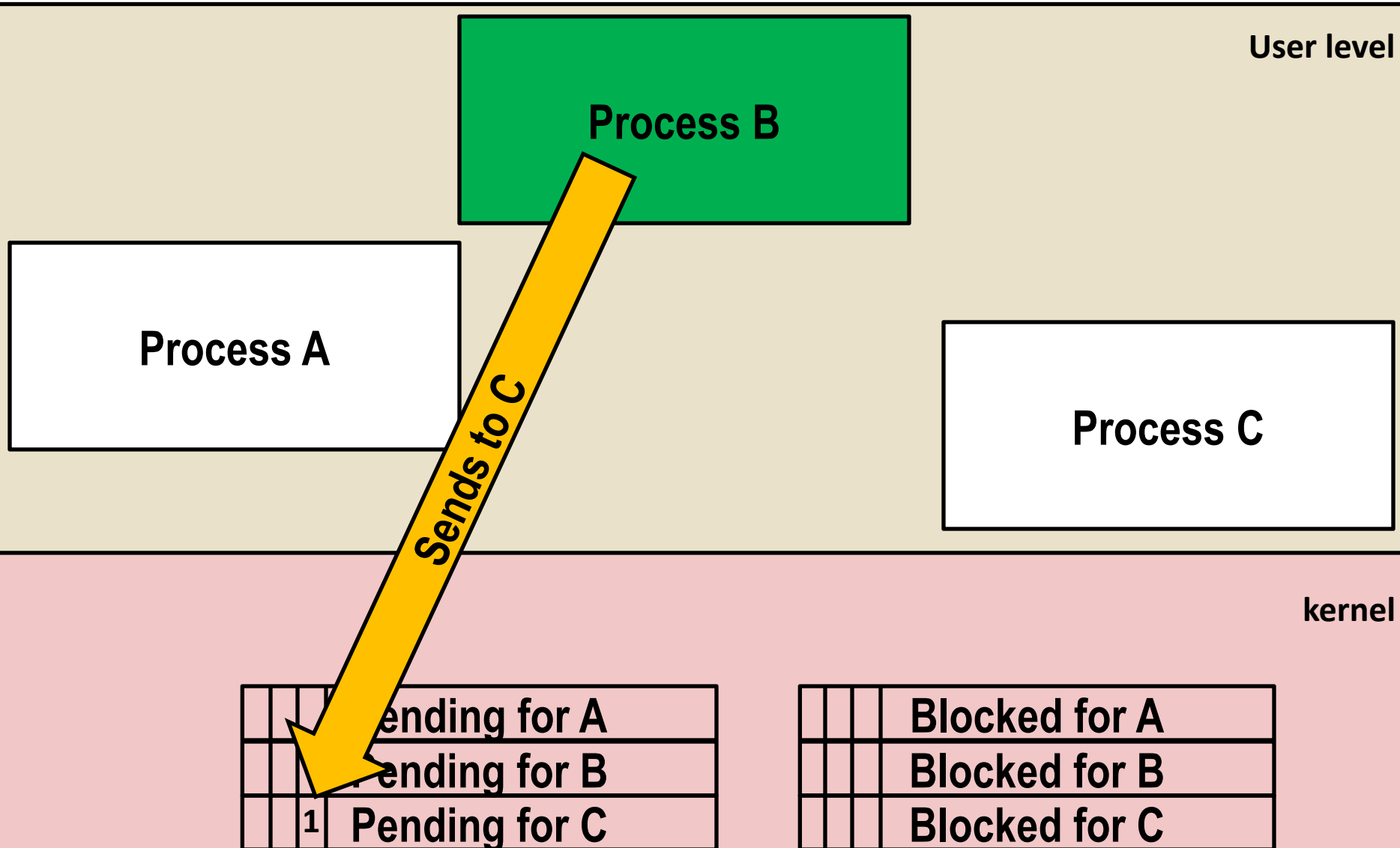
# Signal Concepts: Pending and Blocked Signals

- A signal is *pending* if sent but not yet received
  - There can be at most one pending signal of each 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 sent, but will not be received until the signal is unblocked
  - Some signals cannot be blocked (SIGKILL, SIGSTOP) or can only be blocked when sent by other processes (SIGSEGV, SIGILL, etc)
- A pending signal is received at most once

# Signal Concepts: Pending/Blocked Bits

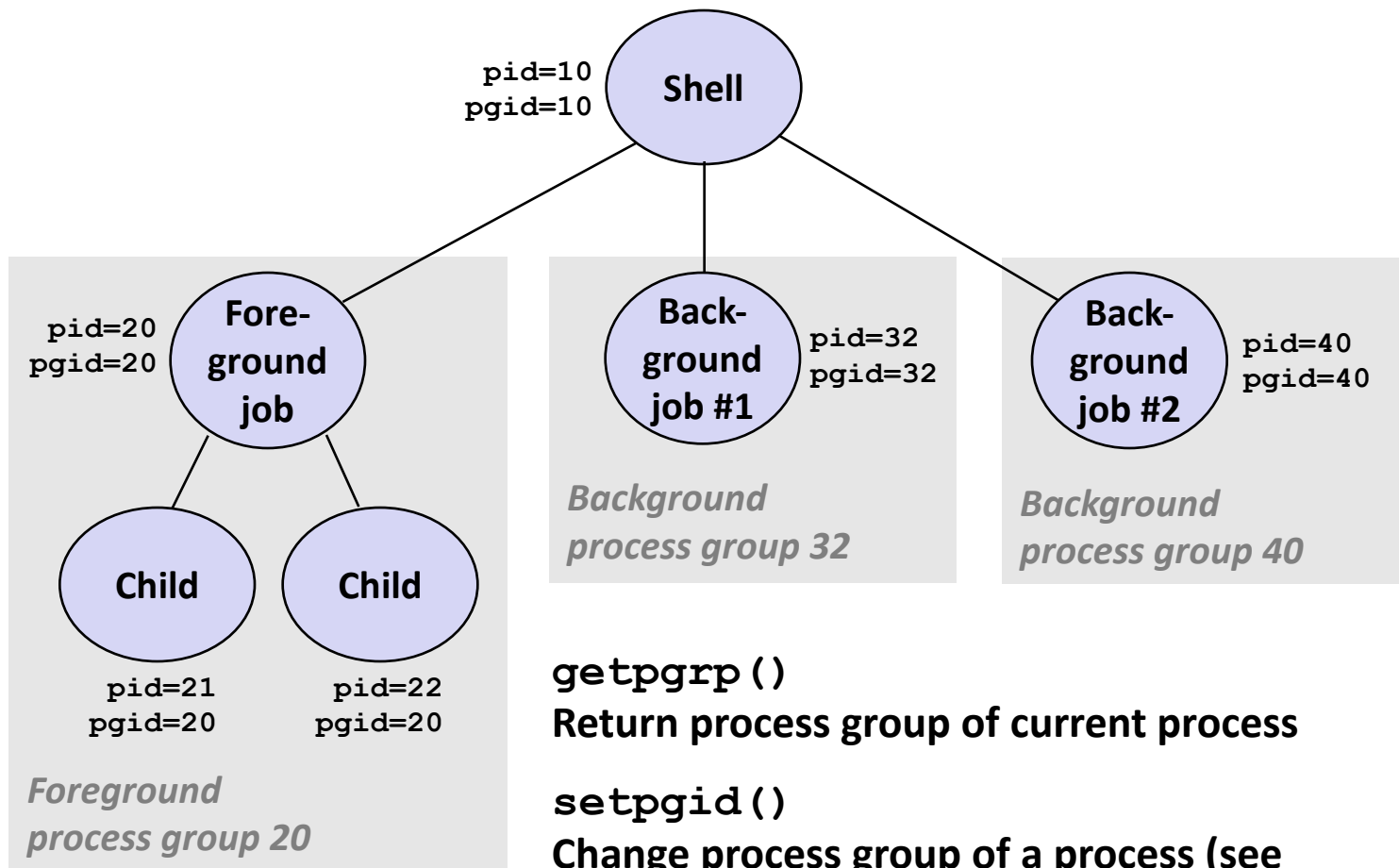
- Kernel maintains **pending** and **blocked** bit vectors in the context of each process
  - **pending**: represents the set of pending signals
    - Kernel sets bit *k* in **pending** when a signal of type *k* is sent
    - Kernel clears bit *k* in **pending** when a signal of type *k* is received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



# 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

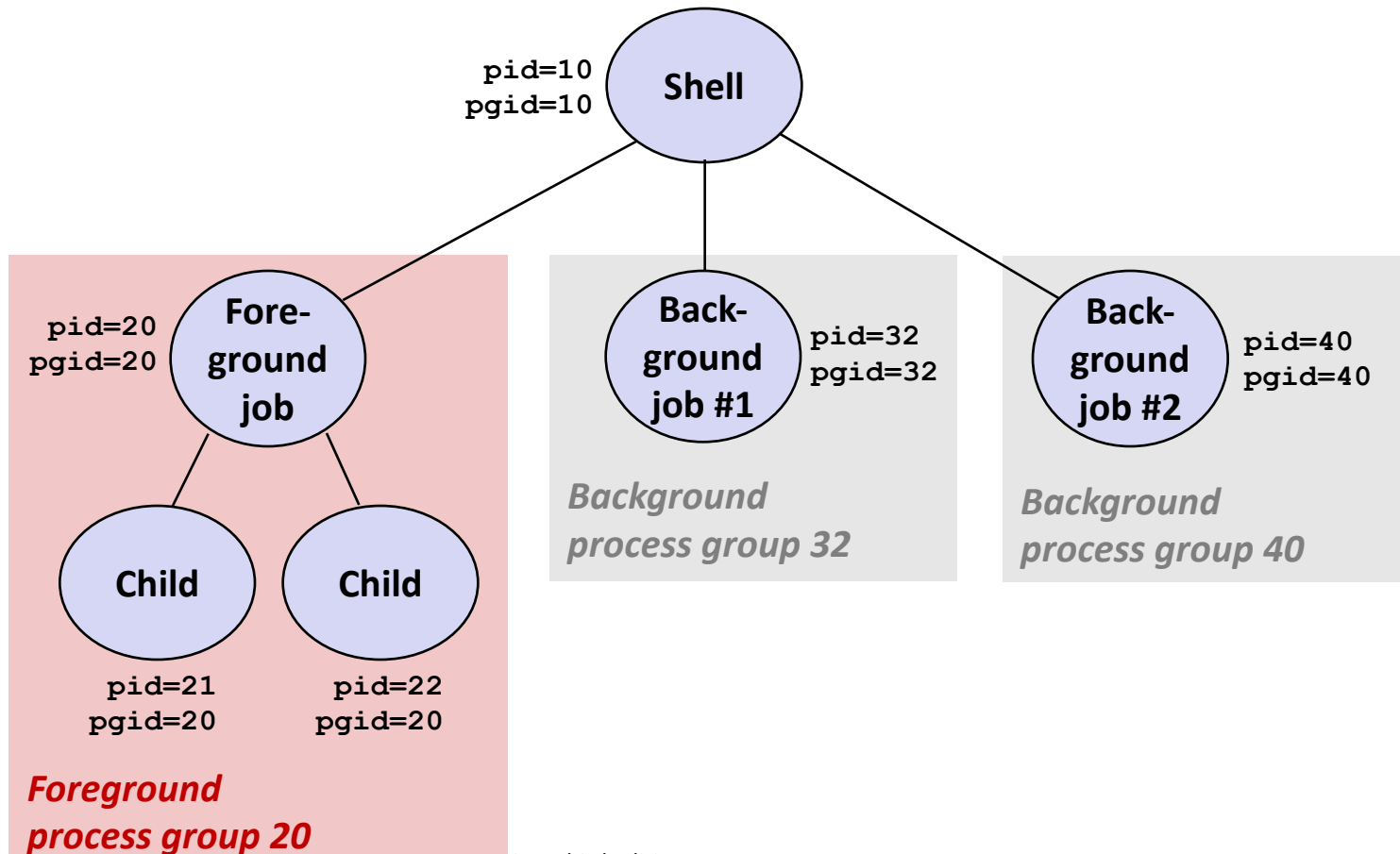
- `/bin/kill -9 -24817`  
Send SIGKILL to every process in process group 24817

```
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

```
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);
    }

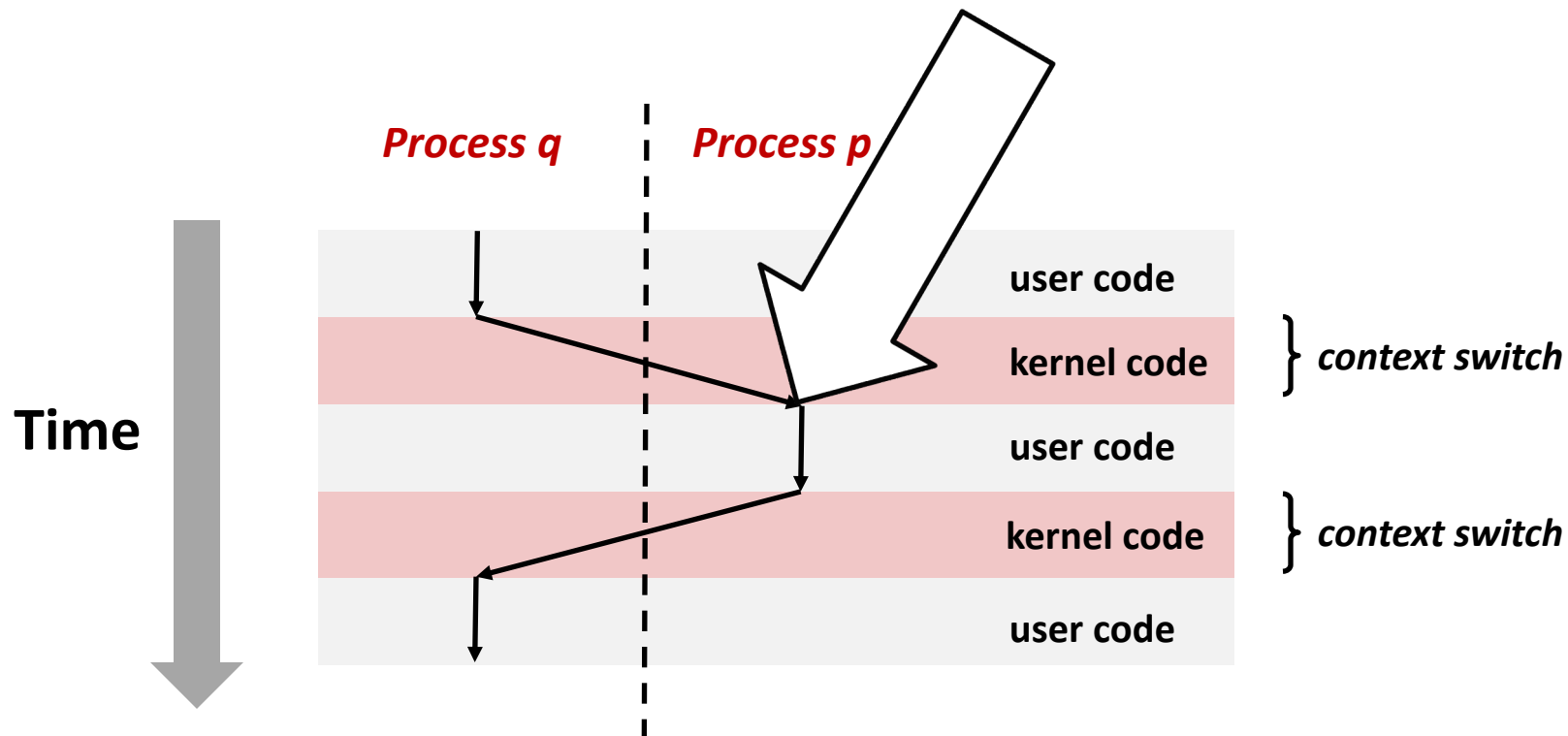
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}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

*forks.c*



# Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process  $p$



# Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process  $p$
- Kernel computes  $\mathbf{pnb} = \mathbf{pending} \ \& \ \sim\mathbf{blocked}$ 
  - The set of pending nonblocked signals for process  $p$
- If ( $\mathbf{pnb} == 0$ )
  - Pass control to next instruction in the logical flow for  $p$
- Else
  - Choose least nonzero bit  $k$  in  $\mathbf{pnb}$  and force process  $p$  to *receive* signal  $k$
  - The receipt of the signal triggers some *action* by  $p$
  - Repeat for all nonzero  $k$  in  $\mathbf{pnb}$
  - 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)`
- 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

```
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");

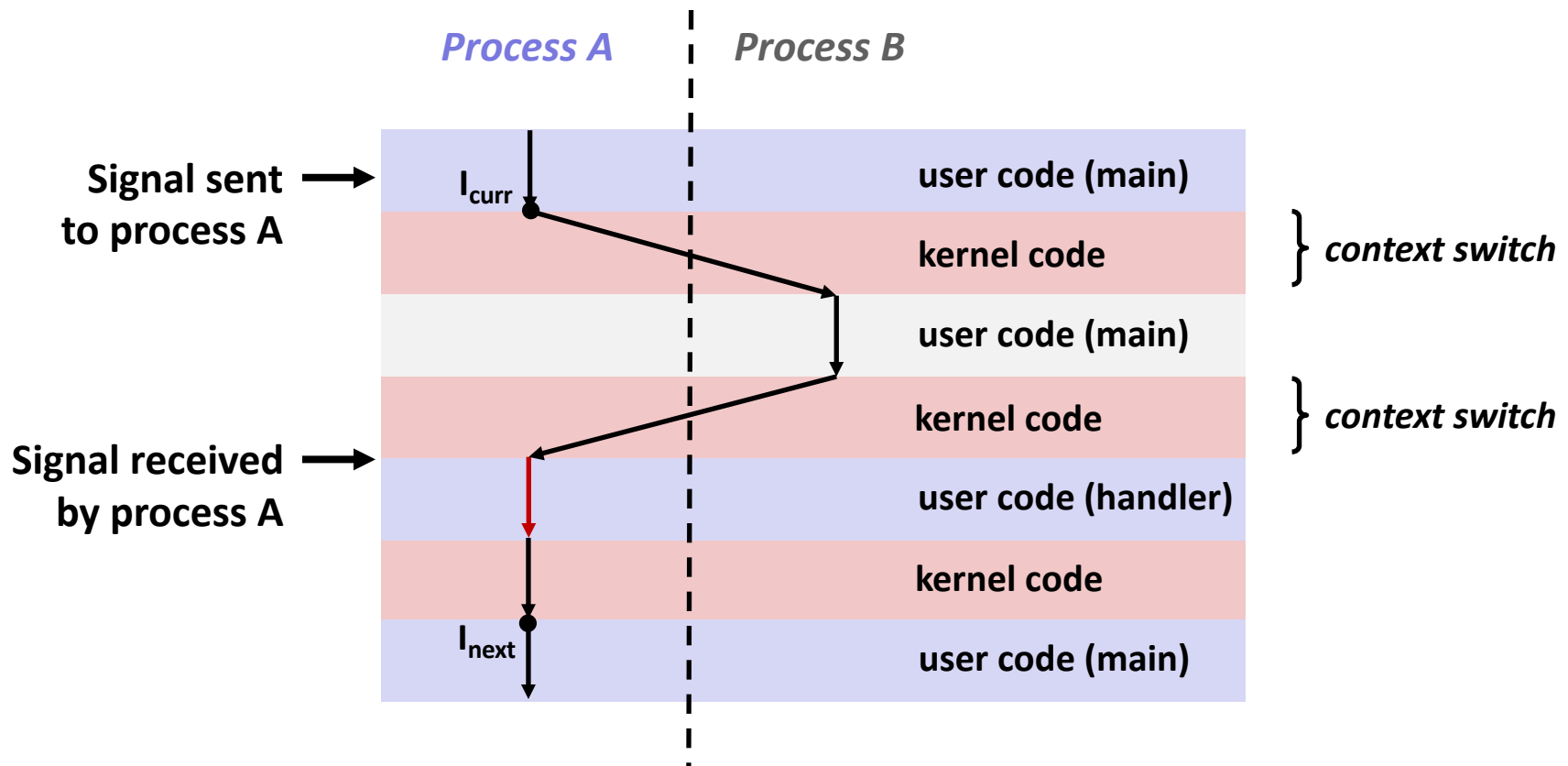
    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```

sigint.c

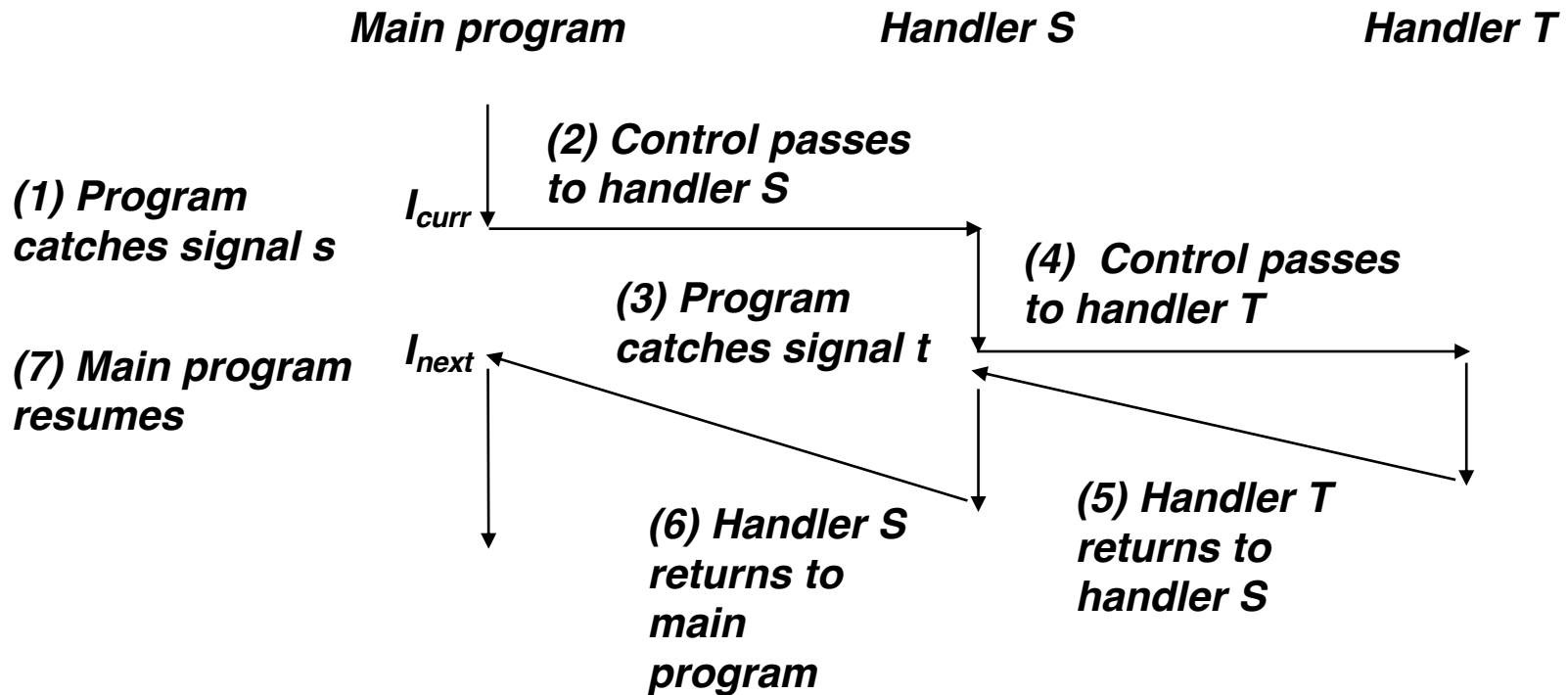


# Another View of Signal Handlers as Concurrent Flows



# Nested Signal Handlers

- Handlers can be interrupted by other handlers





# Blocking and Unblocking Signals

## ■ Implicit blocking mechanism

- Kernel blocks any pending signals of type currently being handled
- e.g., a SIGINT handler can't be interrupted by another SIGINT

## ■ Explicit blocking and unblocking mechanism

- `sigprocmask` function

## ■ Supporting functions

- `sigemptyset` – Create empty set
- `sigfillset` – Add every signal number to set
- `sigaddset` – Add signal number to set
- `sigdelset` – Delete signal number from set

# Temporarily Blocking Signals

```
sigset_t mask, prev_mask;

sigemptyset(&mask);
sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
sigprocmask(SIG_BLOCK, &mask, &prev_mask);

•   /* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

# 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

# Async-Signal-Safety

- Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals
- Posix guarantees 117 functions to be async-signal-safe
  - Source: “man 7 signal-safety”
  - Popular functions on the list:
    - `_exit`, `write`, `wait`, `waitpid`, `sleep`, `kill`
  - Popular functions that are **not** on the list:
    - `printf`, `sprintf`, `malloc`, `exit`
    - Unfortunate fact: `write` is the only async-signal-safe output function

# Safe Formatted Output: Option #1

- Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers

- `ssize_t sio_puts(char s[]) /* Put string */`
- `ssize_t sio_putl(long v) /* Put long */`
- `void sio_error(char s[]) /* Put msg & exit */`

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

sigintsafe.c

# Safe Formatted Output: Option #2

- Use the new & improved reentrant `sio_printf!`
  - Handles restricted class of `printf` format strings
    - Recognizes: `%c %s %d %u %x %%`
    - Size designators `'l'` and `'z'`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    sio_printf("So you think you can stop the bomb"
              " (process %d) with ctrl-%c, do you?\n",
              (int) getpid(), 'c');
    sleep(2);
    sio_puts("Well...");
    sleep(1);
    sio_puts("OK. :-)\n");
    _exit(0);
}
```

sigintsafe.c

# Correct Signal Handling

```
volatile int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    sio_puts("Handler reaped child ");
    sio_putl((long)pid);
    sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

This code is incorrect!

```
void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(0); /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

N == 5

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
...(hangs)
```

forks.c

- Pending signals are not queued
  - For each signal type, one bit indicates whether or not signal is pending...
  - ...thus at most one pending signal of any particular type.
- You can't use signals to count events, such as children terminating.



# Correct Signal Handling

## ■ Must wait for all terminated child processes

- Put `wait` in a loop to reap all terminated children

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        sio_puts("Handler reaped child ");
        sio_putl((long)pid);
        sio_puts(" \n");
    }
    if (errno != ECHILD)
        sio_error("wait error");
    errno = olderrno;
}
```

```
whaleshark> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
whaleshark>
```

# Synchronizing Flows to Avoid Races

## ■ SIGCHLD handler for a simple shell

- Blocks all signals while running critical code

```
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (pid != 0 && errno != ECHILD)
        sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

# Synchronizing Flows to Avoid Races

- Simple shell with a subtle synchronization error because it assumes parent runs before child

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    int n = N; /* N = 5 */
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (n-- > 0) {
        if ((pid = fork()) == 0) { /* Child */
            execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid); /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

procmask1.c

# Corrected Shell Program Without Race

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;
    int n = N; /* N = 5 */
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (n--) {
        sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = fork()) == 0) { /* Child process */
            sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid); /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}
```

# Explicitly Waiting for Signals

- Handlers for program explicitly waiting for SIGCHLD to arrive

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

# Explicitly Waiting for Signals

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    int n = N; /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);

    while (n--) {
        sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid = 0;
        sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

        /* Wait for SIGCHLD to be received (wasteful!) */
        while (!pid)
            ;
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    printf("\n");
    exit(0);
}
```

Similar to a shell waiting for a foreground job to terminate.

waitforsignal.c

# Explicitly Waiting for Signals

```
while (!pid)
    ;
```

## ■ Program is correct, but very wasteful

- Program in busy-wait loop

```
while (!pid) /* Race! */
    pause();
```

## ■ Possible race condition

- Between checking pid and starting pause, might receive signal

```
while (!pid) /* Too slow! */
    sleep(1);
```

## ■ Safe, but slow

- Will take up to one second to respond

# Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`
- Equivalent to atomic (uninterruptable) version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev);  
pause();  
sigprocmask(SIG_SETMASK, &prev, NULL);
```



# Waiting for Signals with `sigsuspend`

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    int n = N; /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    while (n--) {
        sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (fork() == 0) /* Child */
            exit(0);

        /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            sigsuspend(&prev);
        /* Optionally unblock SIGCHLD */
        sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    printf("\n");
    exit(0);
}
```

sigsuspend.c

# Today

- Exceptional Control Flow
- Exceptions
- Signals
- **If we have time: Nonlocal Jumps**

# 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 (stack pointer, base pointer, PC value) from jump buffer `j`
- Set `%eax` (the return value) to `i`
- Jump to the location indicated by the PC stored in jump buf `j`

# setjmp/longjmp Example

- Goal: return directly to original caller from a deeply-nested function

```
/* Deeply nested function foo */  
void foo(void)  
{  
    if (error1)  
        longjmp(buf, 1);  
    bar();  
}  
  
void bar(void)  
{  
    if (error2)  
        longjmp(buf, 2);  
}
```

## setjmp/longjmp Example (cont)

```
jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
        case 0:
            foo();
            break;
        case 1:
            printf("Detected an error1 condition in foo\n");
            break;
        case 2:
            printf("Detected an error2 condition in foo\n");
            break;
        default:
            printf("Unknown error condition in foo\n");
    }
    exit(0);
}
```

# Limitations of Nonlocal Jumps

## ■ Works within stack discipline

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

```

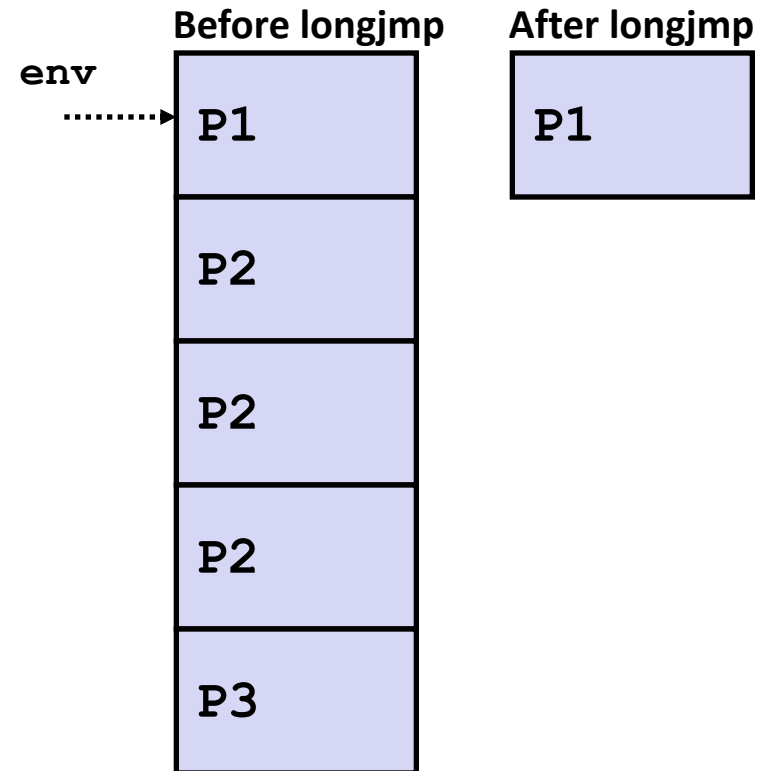
jmp_buf env;

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

```

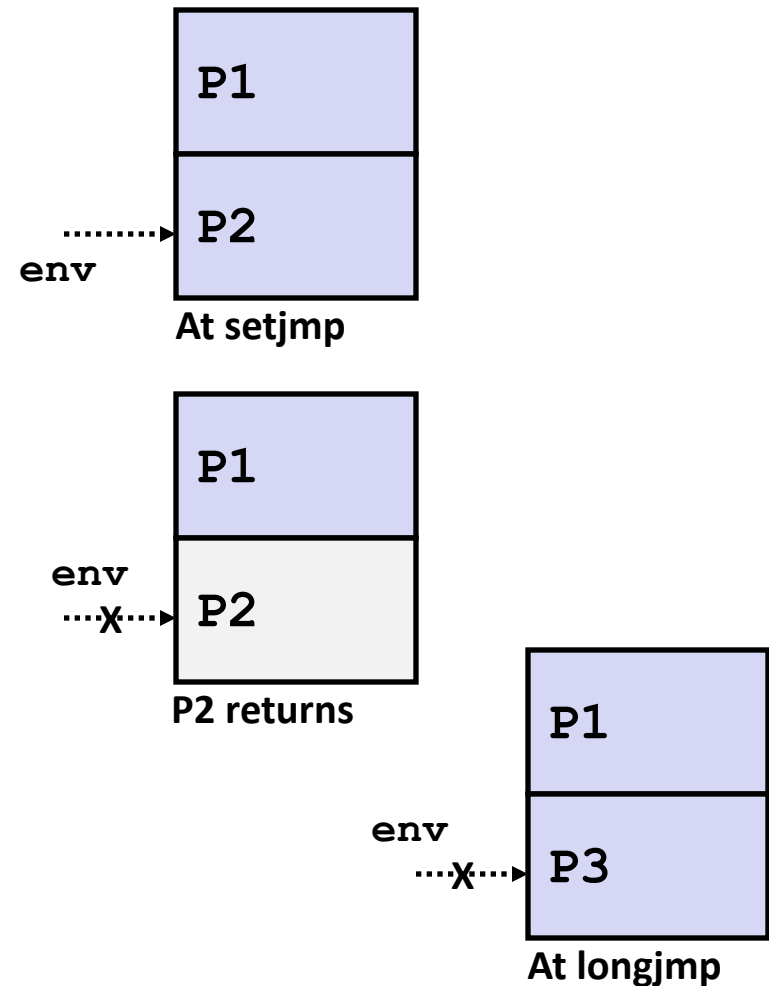
jmp_buf env;

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

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

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

```





# Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

int main()
{
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts("starting\n");
    }
    else
        Sio_puts("restarting\n");

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting
processing... ← Ctrl-c
processing...
restarting
processing... ← Ctrl-c
processing...
processing...
```

restart.c