CS 3214: Computer Systems
Lecture 6: Signals

Instructor: Huaicheng Li

Sept 12 2024

VIRGINIA TECH™

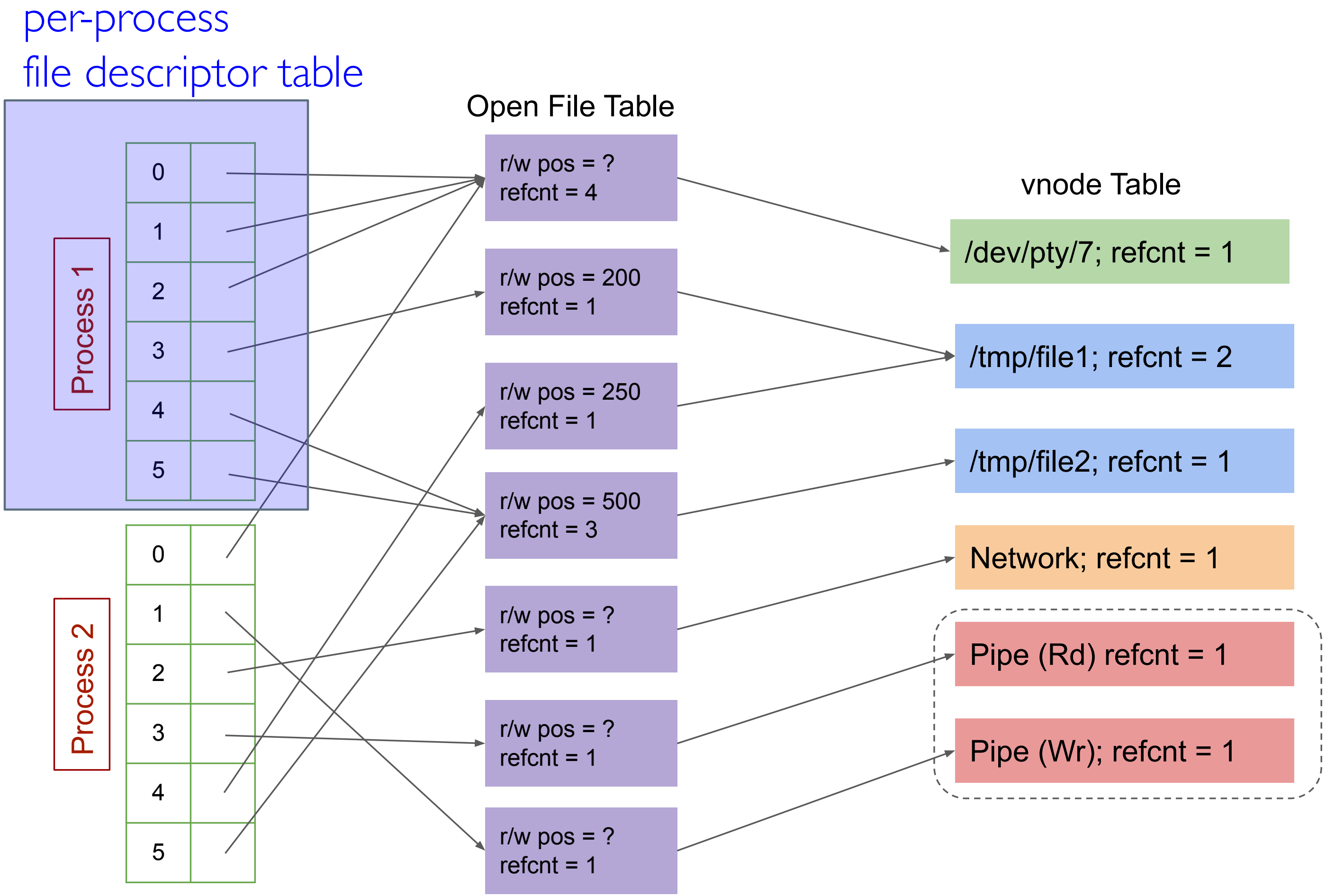
Administrivia

- Project 1 is released!
  - Lab session next week, details to be announced soon

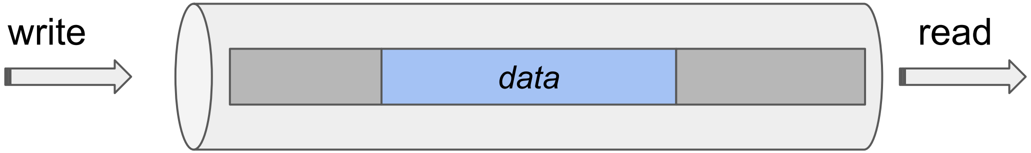
Recap & Today's Topics

- ☐ Recap on file descriptors and pipes
- ☐ Signals
  - Why do we need it?
  - How does it work?
  - How do we use it?
  - Cute demos!

File Descriptors



Pipes



- Writers:
  - can store data in the pipe as long as there is space
  - blocks if pipe is full until reader drains pipe
- Readers:
  - drains pipe by reading from it
  - if empty, blocks until writer writes data
- Pipes provide a classic “bounded buffer” abstraction that
  - **is safe:** no race conditions, no shared memory, handled by kernel
  - **provides flow control that automatically controls relative progress:** e.g., if writer is BLOCKED, but reader is READY, it’ll be scheduled. And vice versa.
  - **Created unnamed;** file descriptor table entry provide for automatic cleanup

Motivation for Signals (Why Should I Care?)

- For inter-process communication
- Process control (e.g., pause job execution)
- Handling unexpected conditions (e.g., segfault)
- Async event handling (e.g., timeout, background job)
- Debugging (inspect program execution on the fly)
- …

Unix Signals

- Unix Signals present a uniform mechanism that allows the kernel to inform processes of events of interest from a small predefined set (<32)
  - Traditionally represented by their integer number
  - Sometimes associated with some optional additional information
- Two types of signals
  - **Synchronous:** caused by something the process did (aka "internally generated event")
  - **Asynchronous:** not related to what the process currently does (aka "externally generated event")
- Uniform API includes provisions for programs to determine actions to be taken for signals, which include
  - terminating the process, optionally with core dump
  - ignoring the signal
  - invoking a user-defined handler
  - stopping the process (in the job control sense)
  - continuing the process
- Sensible *default actions* support user control and fail-stop behavior when faults occur

Synchronous Signals

SIGILL (1)	Illegal Instruction
SIGABRT (1)	Program called abort()
SIGFPE (1)	Floating Point Exception (e.g. integer division by zero)
SIGSEGV (1)	Segmentation Fault - catch all for memory and privilege violations
SIGPIPE (1)	Broken Pipe - attempt to write to a closed pipe
SIGTTIN (2)	Terminal input - attempt to read from terminal while in background
SIGTTOU (2)	Terminal output - attempt to write to terminal while in background

(1) Default action: terminate the process
(2) Default action: stop the process

Asynchronous Signals

SIGINT (1, 3)	Interrupt: user typed Ctrl-C
SIGQUIT (1, 3)	Interrupt: user typed Ctrl-\
SIGTERM (3)	User typed kill pid (default)
SIGKILL (2, 3)	User typed kill -9 pid (urgent)
SIGALRM (1, 3)	An alarm timer went off (alarm(2))
SIGCHLD (1)	A child process terminated or was stopped
SIGTSTP (1)	Terminal stop: user typed Ctrl-Z
SIGSTOP (2)	User typed kill -STOP pid

(1) These are sent by the kernel, e.g., terminal device driver

(2) SIGKILL and SIGSTOP cannot be caught or ignored

(3) Default action: terminate the process

How Signals Work

- First, a signal is *sent (via the kernel)* to a target process
  - Some signals are sent internally by the kernel (e.g. SIGALRM, SIGINT, SIGCHLD)
  - User processes can use the kill(2) system call to send signals to each other (subject to permission)
  - The kill(1) command or your shell's built-in kill command do just that.
  - raise(3) sends a signal to the current process
- This action makes the signal become *"pending"*
- Then (possibly some time later) the target process *receives the signal* and performs the action (ignore, terminate, or call handler)
- **Aside:** the details of how processes learn about pending signals and how they react to them are complicated, but handled by the kernel
- Here we focus on what user programmers need to observe when using signals

Sending a Signal

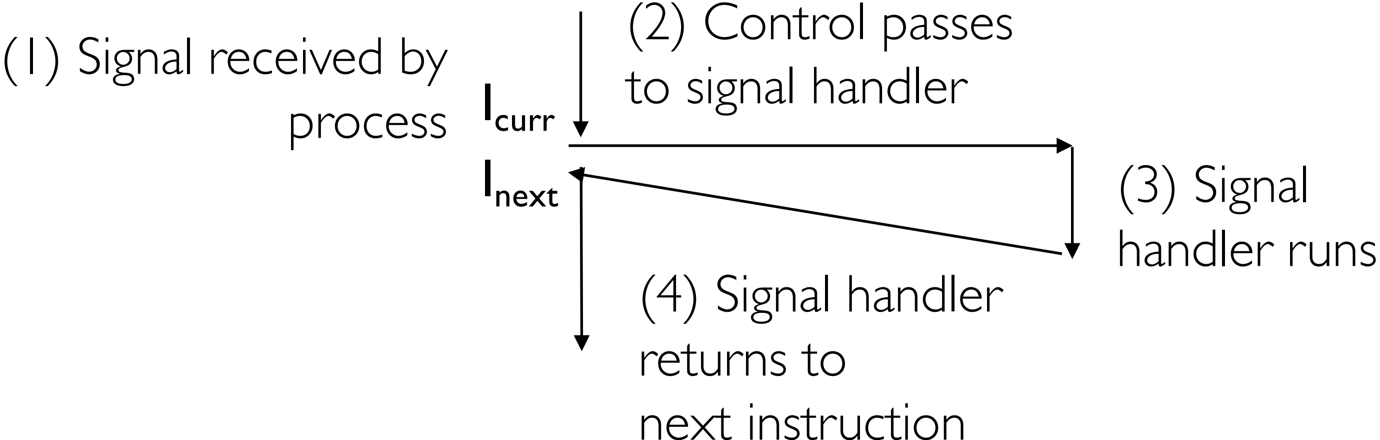
- Kernel sends a signal to a destination process by updating some state in the context of the destination process
  - divide-by-zero (SIGFPE)
  - Termination of a child process (SIGCHLD)
- Another process has invoked *kill()* system call to explicitly request the kernel to send a signal to the destination process
- *raise(), signaling within the same process*

Pending and Blocked Signals

- *Pending: sent but not yet received*
  - At most one pending signal of any particular type
  - Signals are not queued (On/Off)
- A process can block the receipt of certain signals
  - Blocked signal can be delivered, but will not be received until the signal is unblocked
- A pending signal is received at most once
- Kernel maintains pending and blocked ***bit vectors*** in the context of each process
  - Pending: kernel sets/clears certain bits when a signal is delivered/received
  - Blocked: *sigprocmask(), aka, signal mask*

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
- Possible reactions
  - Ignore the signal (do nothing)
  - Terminate the process (e.g., with core dump)
  - Catch the signal by executing a user-level function called signal handler



(1) Signal received by process

I_{curr}

I_{next}

(2) Control passes to signal handler

(3) Signal handler runs

(4) Signal handler returns to next instruction

Signals Don’t Queue

- Each signal represents a bit in the target process’s pending mask saying whether the signal has been sent (but not yet received)
- Thus, sending a signal that’s already pending has no effect
- This applies to internally triggered signals as well: notably, multiple children that terminate while SIGCHLD is pending will result in a single delivery of SIGCHLD
- More like railway signals (on/off) than individual messages

Control Flow (Asynchronous Notification)

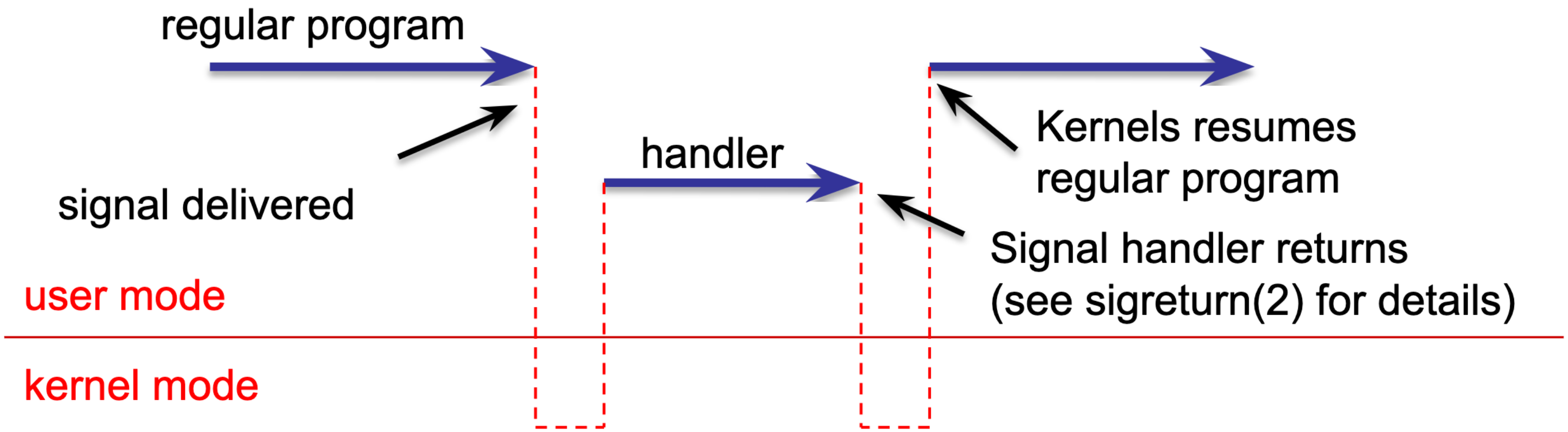


Figure 1: If a user-defined signal handler is set, it may interrupt the current program at any point. After the execution and return of the handler, the original program continues.

Safe Signal Handling

```
void
list_insert (struct list_elem *before,
             struct list_elem *elem)
{
  elem->prev = before->prev;
  elem->next = before;
  before->prev->next = elem;
  before->prev = elem;
}
```

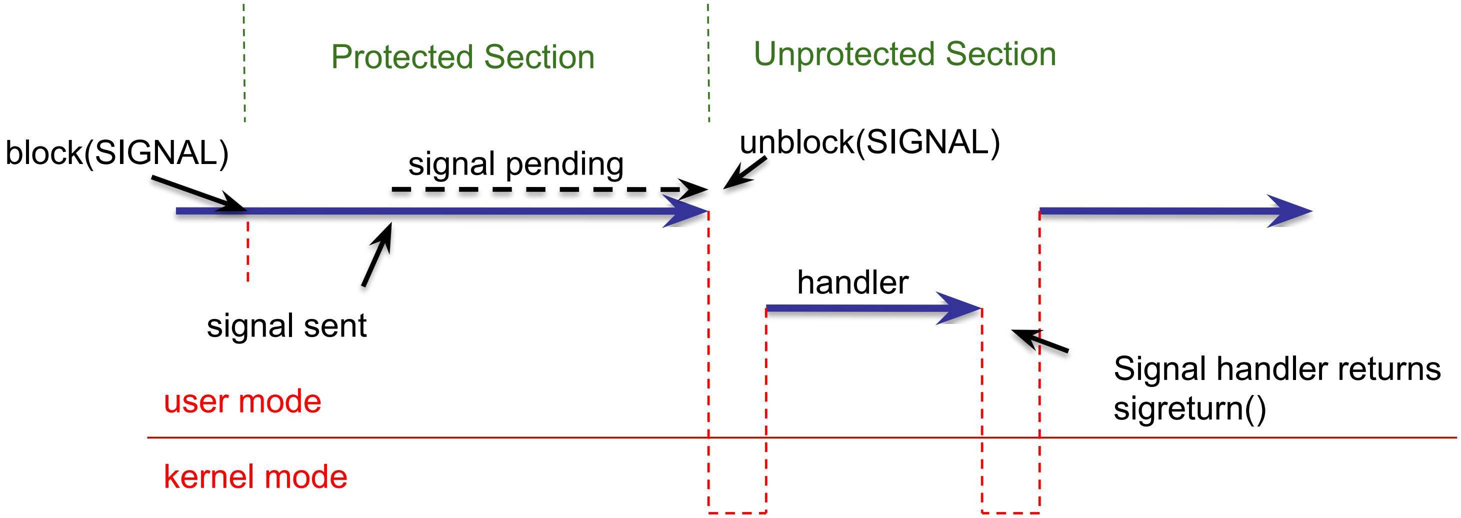
```
list_insert:
    movq    (%rdi), %rax
    movq    %rdi, 8(%rsi)
    movq    %rax, (%rsi)
    movq    %rsi, 8(%rax)
    movq    %rsi, (%rdi)
    ret
```

If a signal arrives in the middle of `list_insert()`, the manipulated list's list element are in a partially linked state. If the signal handler now takes a path where the same list is being accessed (iterated over, etc.), inconsistent behavior will result. This situation must be avoided.

Async-Signal-Safety

- *Is it safe to manipulate data from a signal handler while that same data is being manipulated by the program that was executing (and interrupted) when the signal was delivered?*
- *In general, is it safe to call a function from a signal handler while that same function was executing when the signal was delivered?*
- Answer: it depends.
- POSIX defines a list of functions for which it is safe, so-called async-signal-safe functions, see signal-safety(7) for a list
- *printf()* is not async-signal-safe (acquires the console lock)
- Two strategies to write async-signal-safe programs:
  - don't call async-signal-unsafe function in a signal handler
  - block signals while calling unsafe functions in the main control flow (or when manipulating shared data)

Blocking/Unblocking Signals



- *If signals are masked/blocked* most of the time in the main program, signal handlers can call most functions, but signal delivery may be delayed.
- *If a signal is not masked most of the time,* signal handlers must be very carefully implemented. In practice, coarse-grained solutions are perfectly acceptable unless there is a requirement that bounds the maximum allowed latency in which to react to a signal.
- *Side note:* OS face the same trade-off when implementing (hardware) interrupt handlers.

Safe Signal Handling

- Concurrent with main program
- Guidelines to avoid trouble
    - Keep handlers simple
    - Only use async-signal-safe functions (no *printf*)
    - Save and restore *errno* on entry and exit to avoid overwrite
    - Temporarily blocking all signals to protect access to shared data structures
    - Declare global variables as *volatile* to prevent compiler from storing them in a register
    - Declare global flags as *volatile sig_atomic_t*

Signal APIs

- Uniform APIs for programs to determine actions to be taken for signals
  - Terminating the process, core dump
  - Ignoring the signal
  - Invoking a user-defined handler
  - Stop the process
  - Continuing the process

Blocking/Unblocking Signals

- Explicit blocking/unblocking: *sigprocmask()*
- *Others*
    - *sigemptyset()* – create empty set
    - *sigfillset()* – Add every signal number to set
    - *sigaddset()* – Add signal number to set
    - *sigdelset()* – Delete signal number from set

```
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);
```

Installing Signal Handlers

- *Handler_t *signal(int signum, handler_t *handler)*

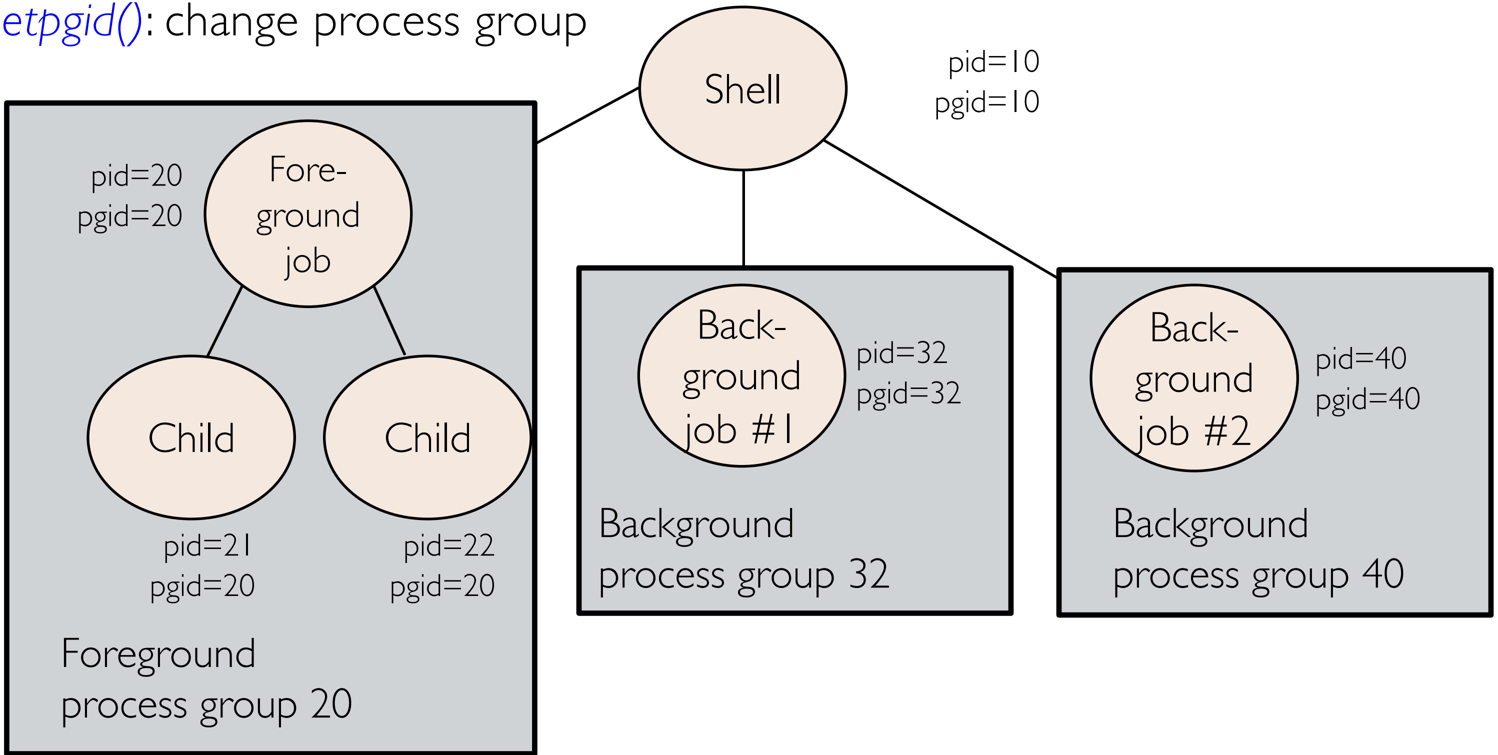
```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");
    /* Wait for the receipt of a signal */
    pause();
    return 0;
}
```

For Reference: Kill: Sending signals

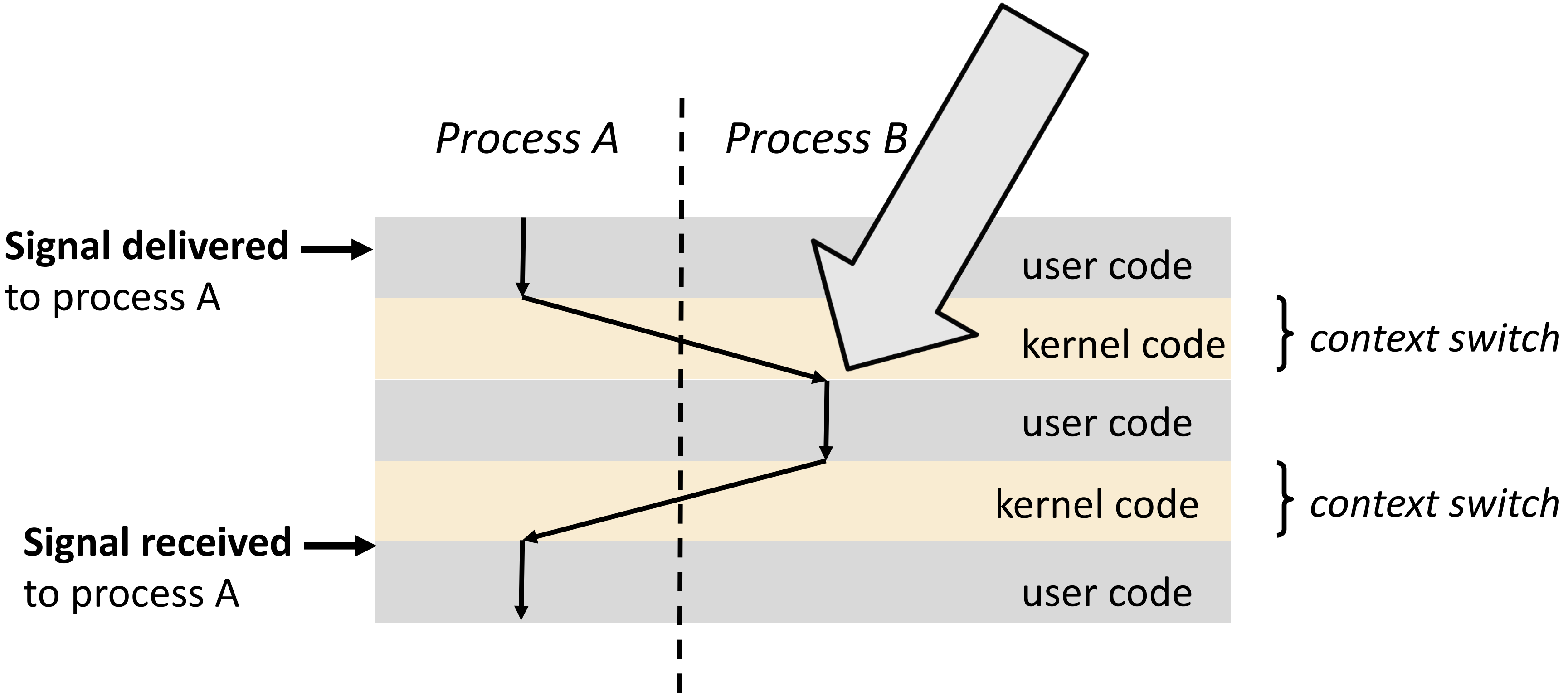
- Kill -9 1000: Send SIGKILL to process 1000
- Kill -9 -1000: Send SIGKILL to every process in process group 1000
- Ctrl-C: SIGINT
- Ctrl-Z: SIGTSTP

Process Group

- One process belongs to one process group
  - *getpgrp()*, get process group of current process
  - *setpgid()*: change process group



Receiving Signals



Process A

Process B

Signal delivered to process A

Signal received to process A

user code

kernel code

context switch

user code

kernel code

context switch

user code