# CS 241: Systems Programming
# Lecture 22. Signals

Fall 2023

Prof. Stephen Checkoway

How does a process request that the kernel perform an action on the process's behalf?

A. It calls a function in the kernel

B. It calls a function in libc

C. It makes a system call

D. It makes a hypervisor call

E. It changes switches the processor into kernel mode and then performs the action

# Signals

Signals are a mechanism for the kernel to inform a process that some event has occurred

- A single bit of information: event X occurred (possibly multiple times!)

System calls are for process -> kernel communication

Signals are for (extremely limited) kernel -> process communication

# Common signals: signal(7)

**SIGINT** — Interrupt from keyboard (ctrl-C on the terminal)
**SIGQUIT** — Quit from keyboard (ctrl-\ on the terminal)
**SIGILL** — Illegal instruction
**SIGABRT** — Signal from abort() (or assert() which calls abort())
**SIGFPE** — Floating point exception; integer divide by 0 on some systems
**SIGKILL** — Kill signal, cannot be handled or ignored
**SIGSEGV** — Segmentation fault
**SIGPIPE** — Write to pipe with no readers
**SIGTERM** — Termination signal
**SIGCHLD** — Child stopped or terminated
**SIGSTOP** — Suspend the process (ctrl-Z on the terminal)
**SIGCONT** — Resume the process (fg or bg on terminal)
**SIGWINCH** — Terminal window resized

# Similar sounding signals

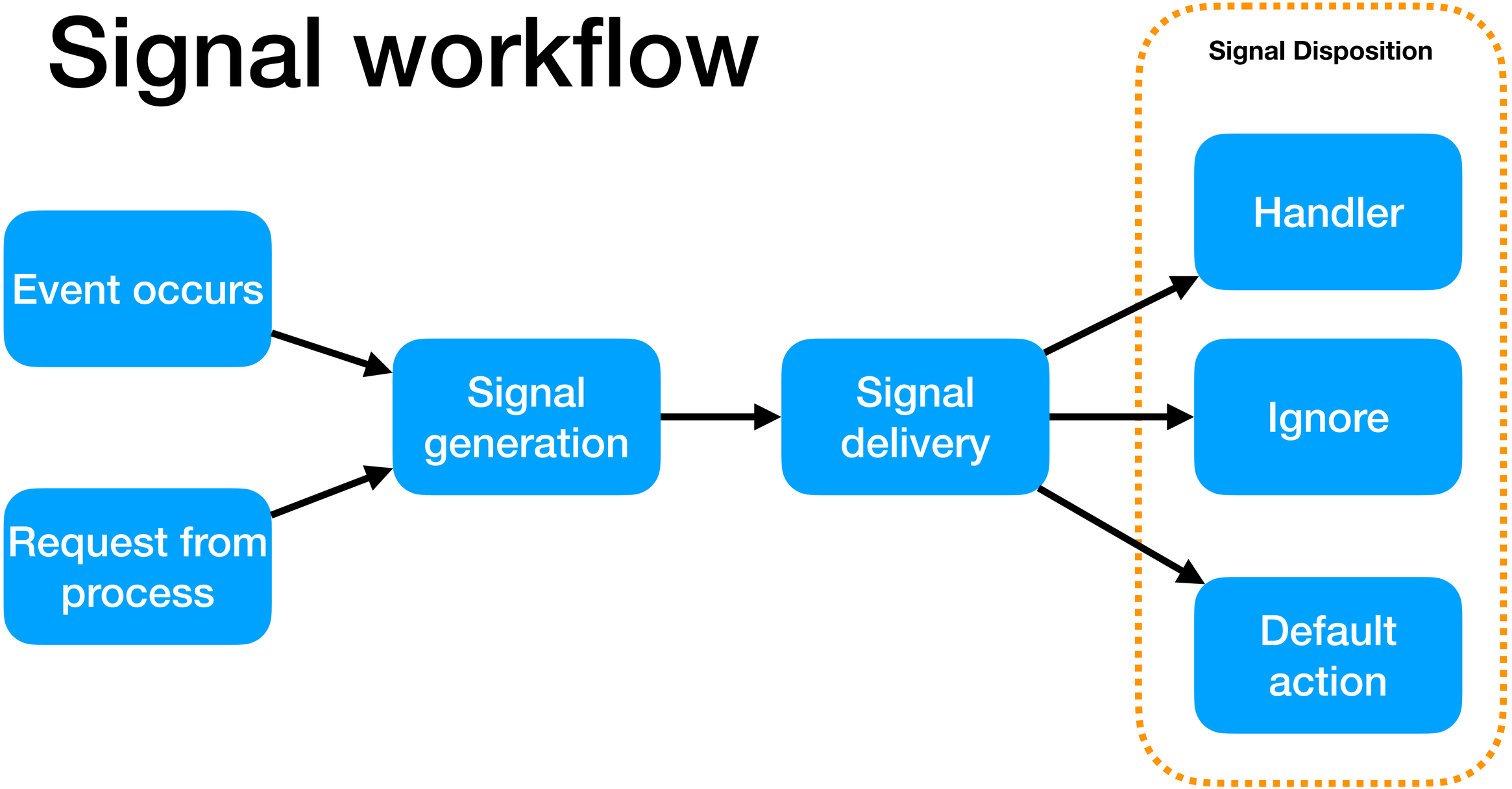
**SIGINT** — Interrupt from keyboard (ctrl-C on the terminal)
**SIGQUIT** — Quit from keyboard (ctrl-\ on the terminal)
**SIGKILL** — Kill signal, cannot be handled or ignored
**SIGTERM** — Termination signal
**SIGSTOP** — Suspend the process (ctrl-Z on the terminal)

**SIGINT** and **SIGQUIT** should only come from the user typing at the terminal

If one process wants to stop another, it should (typically) request the process terminate via **SIGTERM** and, if after a few seconds it hasn't, use **SIGKILL**

**SIGSTOP** is about job control, not about terminating processes

# Signal workflow



# Event or request

Some event occurs

- Ctrl-C, or a child process exits or …

A process requests a signal be sent to itself or another process

- The kill system call specifies a signal to send

# Signal generation

The kernel maps the event/request to a signal number

The kernel sets a bit indicating the particular signal is **pending** (meaning it will be delivered) for the target process

# Signal delivery

Before returning to a user process after a system call or context switch, the kernel checks the set of pending signals and the set of signals the process is **blocking**

If a pending signal is not blocked, signal delivery occurs

Action taken depends on the **signal disposition**

- If a signal handler has been registered, it is called by the kernel in the context of the process
- If the particular signal is ignored, nothing happens
- Otherwise the default action occurs

Default action

- Some signals are ignored (like SIGCHLD)
- Some cause the process to be terminated (like SIGINT)

# Signal handlers

Signal handlers are just functions that are called asynchronously in response to a signal

Signal handlers run in the context of the process and have access to all of the process's memory

**Signal handlers are extremely limited in what they can safely do**

# Blocked signals

Processes can request that delivery of particular signals be blocked

When a blocked signal is generated, it remains pending until the signal is unblocked

- When a signal is unblocked and is pending, it is delivered immediately

Typically, a process will block signals for a short period of time and then unblock

If a process never wants to receive a signal, it can set the signal's disposition to ignored

# Signal delivery delay

Signal delivery is deferred until the kernel next returns to the process

- At the completion of a system call
- The next time the process is scheduled to run

Some system calls can be interrupted, others cannot

- System calls like read(2) and write(2) can read/write less than requested when interrupted by a signal; return value reflects this
- Other calls may return `-1` and set **`errno`** to **`EINTR`** to indicate it was interrupted

Only one of each (standard) signal may be pending at a time

Consider the following sequence of events

- The process installs a signal handler for **`SIGINT`**
- The process masks (blocks) **`SIGINT`**
- The user presses ctrl-c twice
- The process unmasks (unblocks) **`SIGINT`**

Which of the following is correct?

A. The handler never runs

B. The handler runs the first time ctrl-c is pressed

C. The handler runs both times ctrl-c is pressed

D. The handler runs once after the signal is unmasked

E. The handler runs twice after the signal is unmasked

# Sending a signal

From the shell: `kill(1)` or `killall(1)`

- `$ kill -9 1234 # Send SIGKILL (signal 9) to PID 1234`
- `$ kill -l # List all of the signals`

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

- Sends signal sig to process pid
- Different behavior depending on pid < 0, pid = 0, pid > 0, sig = 0, sig > 0

```
int raise(int sig);
```

- Sends signal sig to the own process

# Setting a handler

Use sigaction(2)

- Takes a `const` pointer to a struct that holds a new handler and flags
- Takes a pointer to a struct that will be filled in with the old handler and flags
- flags specify the behavior of interrupted system calls, what information is given to the signal handler, and whether the same signal can be received while its handler is running
- Read the man page!

# Blocking signals

Signal masks (which indicate which signals are blocked) can be manipulated with `sigprocmask(2)`

# Signal handler limitations

Signal handlers run asynchronously compared to the rest of the code

There is a real danger of a signal handler modifying data that the main program is currently accessing

- This is undefined behavior

Signal handlers have extreme limitations:

- No allocating memory
- A very restricted set of system calls are allowed
- No touching data other code can use **nonatomically**

# “Safe” signal handlers

The only really safe signal handler (this is a slight overstatement) is one that atomically sets a bool

Pseudo code:

```
received_signal = false

fn handler():
    atomically set received_signal = true

fn main():
    register handler as signal handler for SIGINT
    loop:
        let input = read_from_stdin()
        interrupted = atomically read received_signal and set it to false
        if interrupted:
            handle the Ctrl-C
```

# Nonatomic operations

Consider the code like x = x + 1;

The processor has to perform three concrete actions

- Load the current value of x from memory
- Add 1 to that value
- Store the new value of x back into memory

This process is **not atomic**

- E.g., the process could be interrupted by a signal after loading x from memory but before storing the new value back, if the signal handler modified x, then its modification would be overwritten once the main code is running again

Signal handlers can be interrupted by signals which means that the signal handler can be run in response to a signal while it is already running!

Does the following pseudocode for a signal handler that counts how many times the handler is called work correctly? Why or why not?

```
count = 0
fn handler():
    count += 1
```

A. Yes

B. No

C. It depends

# Atomic operations

Modern processors all have support for performing atomic operations on basic types like bools and integers

Programming languages have varying levels of support for atomics (C only added support in 2011!)

Rust has fantastic support for atomic operations

# AtomicBool

```rust
use std::sync::atomic::{AtomicBool, Ordering};

// Create a new AtomicBool initially set to false.
let val = AtomicBool::new(false);

// Atomically sets val to true
val.store(true, Ordering::Relaxed);

// Atomically loads val; assigns the result to x.
let x = val.load(Ordering::Relaxed);

// Atomically sets val to false and returns the old value of val
let old = val.swap(false, Ordering::Relaxed);
```

# Not mutable!

```
use std::sync::atomic::{AtomicBool, Ordering};

// Create a new AtomicBool initially set to false.
let val = AtomicBool::new(false);

// Atomically sets val to true
val.store(true, Ordering::Relaxed);

// Atomically loads val; assigns the result to x.
let x = val.load(Ordering::Relaxed);

// Atomically sets val to false and returns the old value of val
let old = val.swap(false, Ordering::Relaxed);
```

val isn’t mutable

This modifies it!

As does this!

# Mutating nonmutable data

Since every operation on an AtomicBool is atomic, it’s safe to let multiple pieces of code operate on it simultaneously

Put another way, modification through shared references is safe

Consequence: We can use global, nonmutable atomic variables and modify the values, e.g., from a signal handler

# Ordering

Each of the .store(), .load(), and .swap() methods take an Ordering enum

The Ordering variant used dictates how the hardware relates the atomic operation with other memory reads (loads) and writes (stores)

This isn’t relevant for our use case so we always use Ordering::Relaxed

# Signal handling in Rust

1. Create a global AtomicBool variable for each signal of interest

2. Create a signal handler function that sets the appropriate AtomicBool for the received signal **and does nothing else**

3. Use libc::sigaction() to register the signal handler

# The AtomicBool and handler

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static INTERRUPTED: AtomicBool = AtomicBool::new(false);

extern "C" fn handler(_sig: libc::c_int) {
    INTERRUPTED.store(true, Ordering::Relaxed);
}
```

extern “C” tells rustc to compile the function in the manner expected by C code since that’s what the kernel expects too

# Setting the signal handler

```
unsafe {
    let action = libc::sigaction {
        sa_sigaction: handler as libc::sighandler_t,
        ..std::mem::zeroed()
    };

    if libc::sigaction(libc::SIGINT, &action, std::ptr::null_mut()) < 0 {
        return Err(io::Error::last_os_error());
    }
}
```

Set the sa_sigaction field to handler (cast to a sighandler_t)
Set the rest of the fields to zero

Call sigaction(), passing the action structure and check the return value

# Changing the signal disposition

In general, we call sigaction() to change the signal disposition to one of

- Call a handler
- Ignore the signal
- Perform the default action

Sets the disposition to ignored
Use SIG_DFL to set the disposition to the default action

```
unsafe {
    let action = libc::sigaction {
        sa_sigaction: libc::SIG_IGN,
        ..std::mem::zeroed()
    };

    if libc::sigaction(libc::SIGINT, &action, std::ptr::null_mut()) < 0 {
        return Err(io::Error::last_os_error());
    }
}
```