CS 3214: Computer Systems
Lecture 7: Signals

Instructor: Huaicheng Li

Sept 13 2022

VIRGINIA TECH™

Signals

- A small message to notify a process of an event
- Similar to exceptions and interrupts
- Who generates signals?
  - Self, other-processes, the kernel
- Signal types (interger ID's, e.g., <32)

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

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

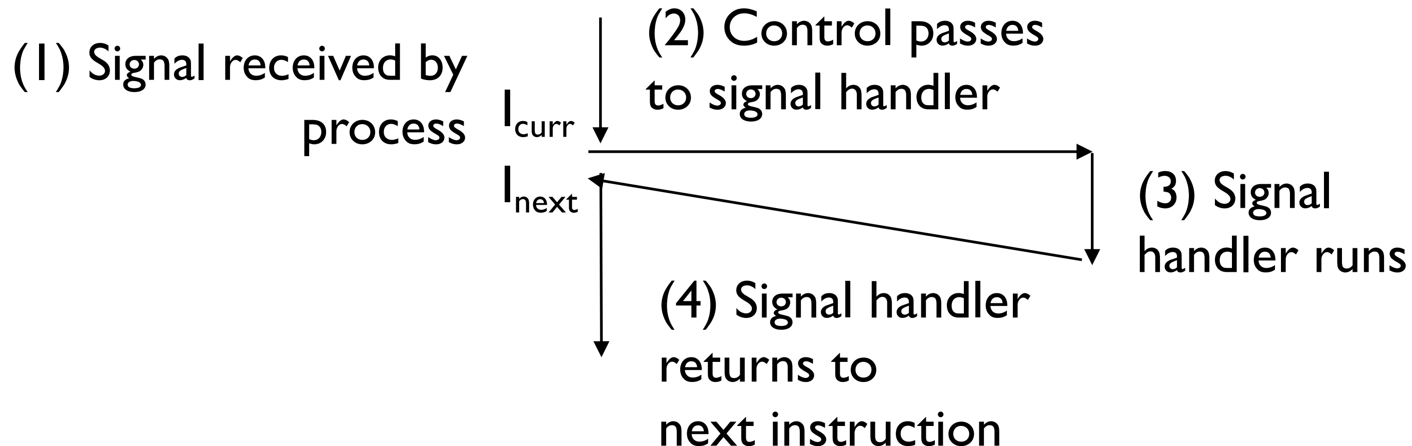
(1) These are sent by the kernel, e.g., terminal device driver
(2) SIGKILL and SIGSTOP cannot be caught or ignored
(3) Default action: terminate the process

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()*

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



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*

Safe Signal Handling

```c
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.

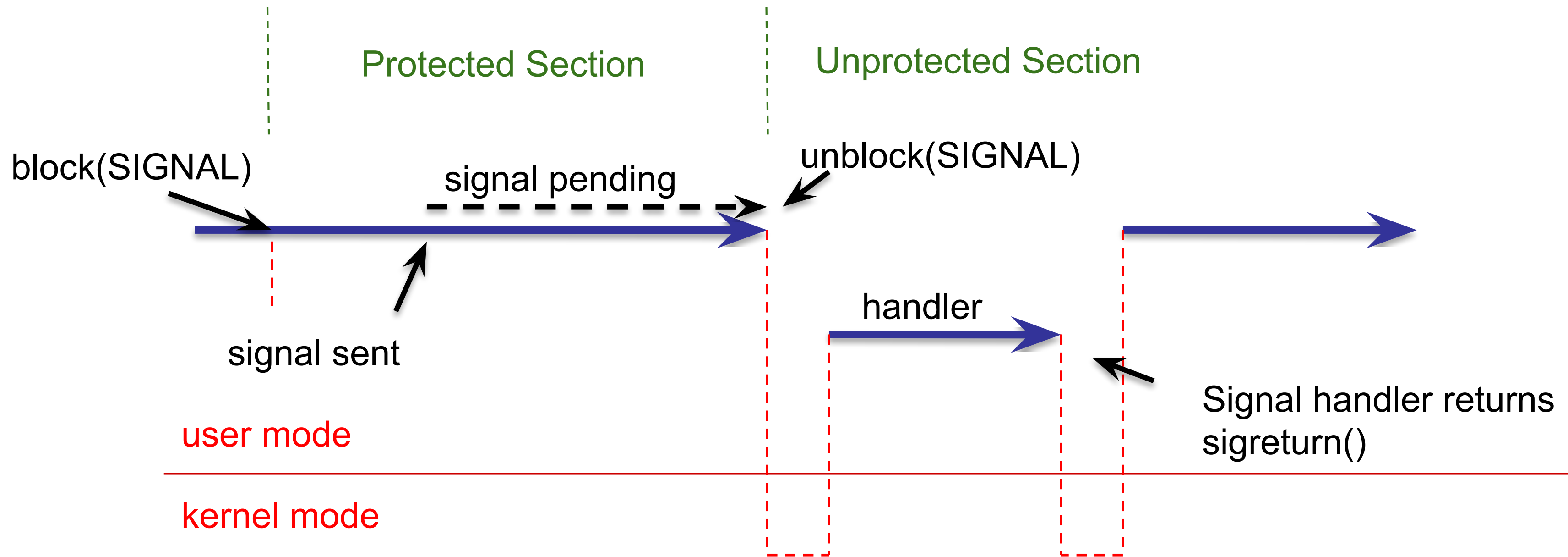
Safe Signal Handling

- 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*

Async-Signal-Safety

- Man 7 signal
- Safe: _exit(), write(), wait(), waitpid(), sleep(), kill()
- Unsafe: printf(), sprint(), malloc(), exit()

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.

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

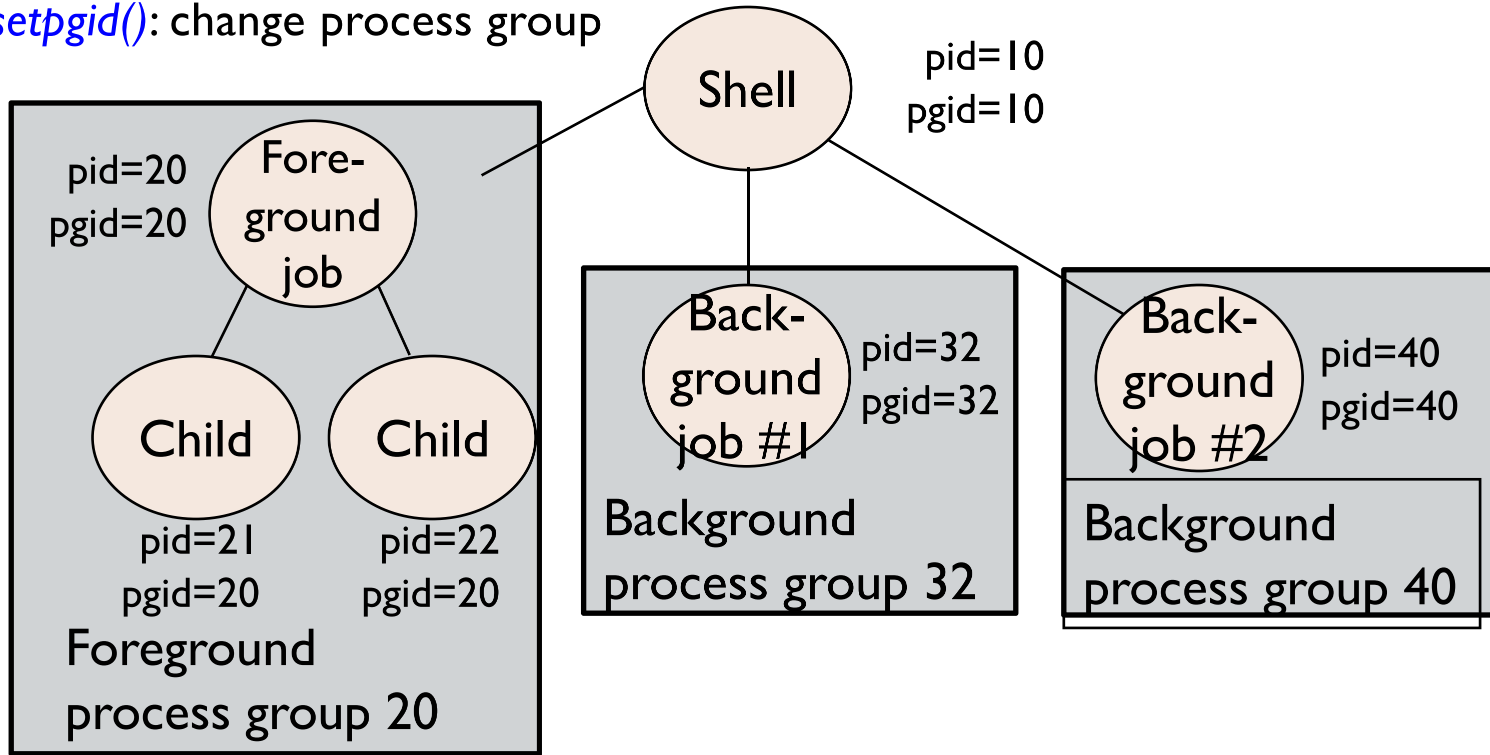
```
sigset_t mask, prev_mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);

/* 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);
```

Process Group

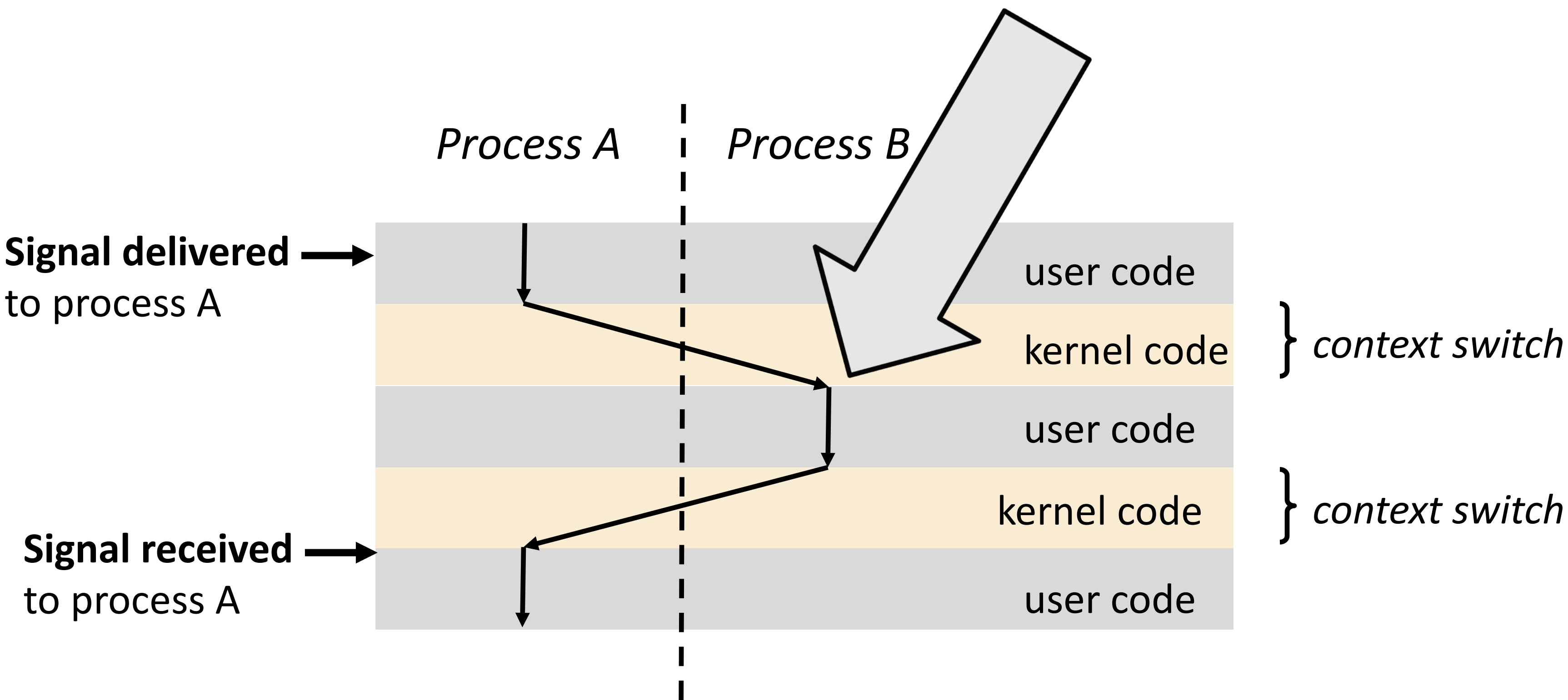
- One process belongs to one process group
  - *getpgrp()*, get process group of current process
  - *setpgid()*: change process group



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

Receiving Signals



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

Installing Signal Handlers

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

```
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;
}
```