CSCI 210 Systems Programming

Week 11
Signals and Signal Processing

slides adapted from
Cornel CS4414 and University at Buffalo CSE220

Overview

- Exceptions
- Signals
- Signal Handling

Recap: Kernel Mode versus User Mode

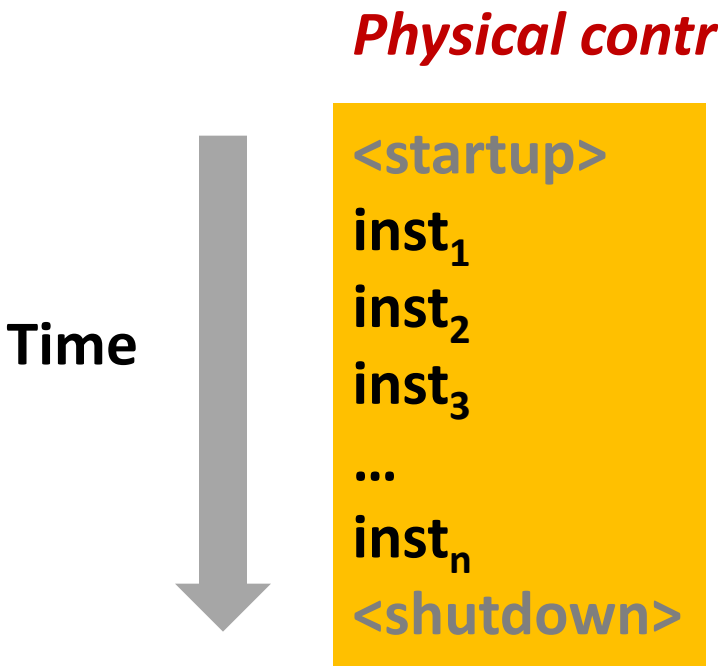
- User mode and kernel mode are two process modes in an OS that differ in the level of access and privileges granted to the code running in each mode
  - A user program runs in User mode, but several switches from the User mode to the Kernel mode occur while the process is handled by the OS.
  - Kernel mode is an elevated, higher privilege mode with direct control over the hardware and system resources.
  - A process may switch to the kernel mode due to these exceptions:
    - Interrupts
    - Traps
    - System calls

Recap – Interrupts and Traps (Exceptions)

- Interrupts are signals from external devices to the CPU, requesting for CPU service. While executing in user mode, the CPU's interrupts are enabled so that it will respond to any interrupts.
  - When an interrupt occurs, the CPU will enter the kernel mode to handle the interrupt, which also causes the process to enter the kernel mode.
- Traps are error conditions, such as invalid address, illegal instruction, divide by 0, etc., which are recognized by the CPU as exceptions, causing it to enter the kernel mode to deal with the error.
  - In Unix/Linux, the kernel trap handler converts the trap reason to a signal number and delivers the signal to the process.
  - For most signals, the default action of a process is to terminate.

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU 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*)



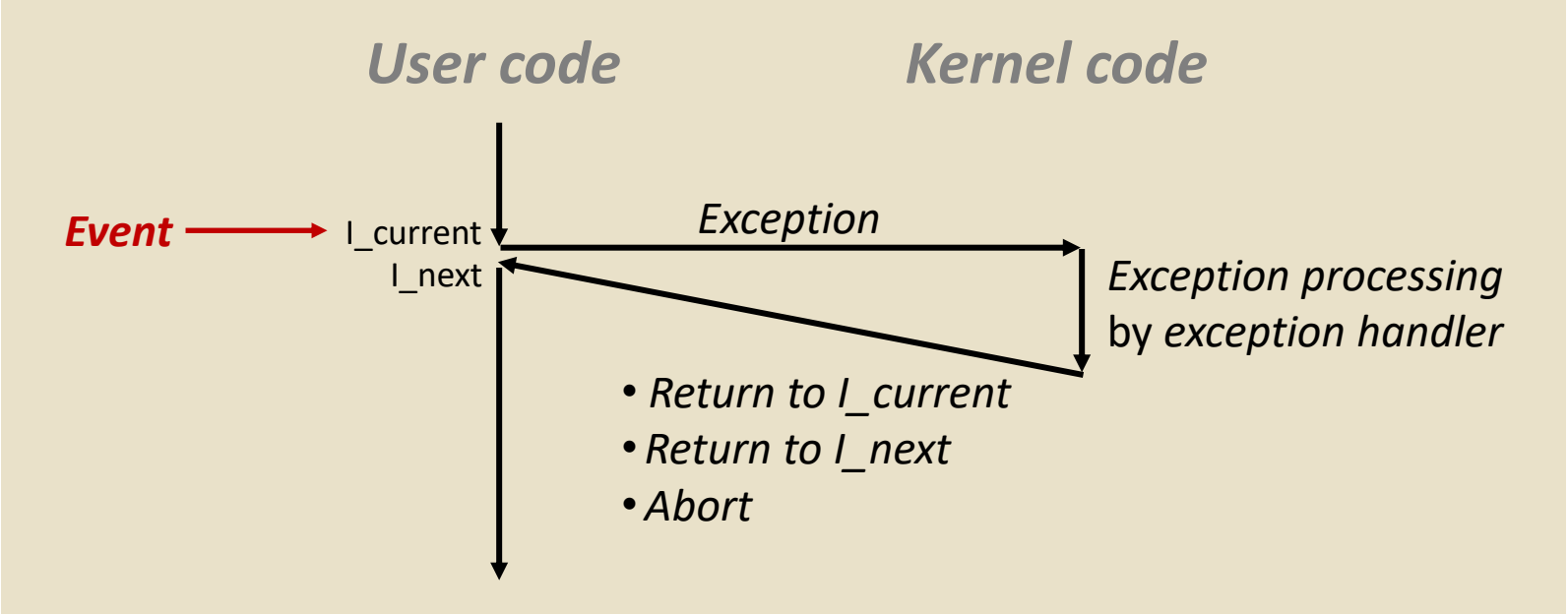
Altering the Control Flow

- Programming constructs: if, break, continue, etc. (they are essentially branches and jumps in User Mode)

- Insufficient: We also need 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… Timer expires…
  - Switch to Kernel Mode

Altering the Control Flow

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



Exceptions: several “flavors” but many commonalities

- All exceptions “seize control,” generally by forcing the immediate execution of a handler procedure, no matter what your process was doing.

- When a hardware device wants to signal that something needs attention, or has gone wrong, we say that the device triggers an interrupt. Linux generalizes this and views all forms of exceptions as being like interrupts.

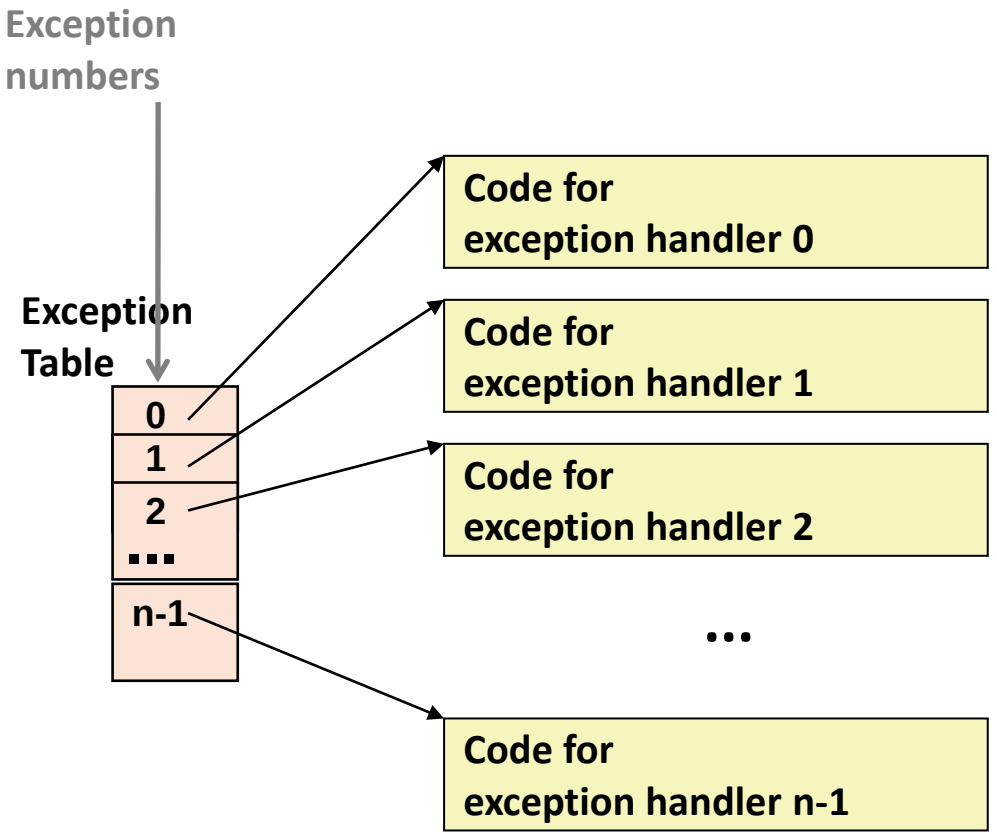
- Once this occurs, we can “handle” the exception in ways that might hide it, or we may need to stop some task entirely (like with ^C).

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - 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
  - Process context switch
    - Implemented by OS software and hardware timer
  - Signals
    - Implemented by OS software
  - Nonlocal jumps: setjmp() and longjmp()
    - Implemented by C runtime library

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

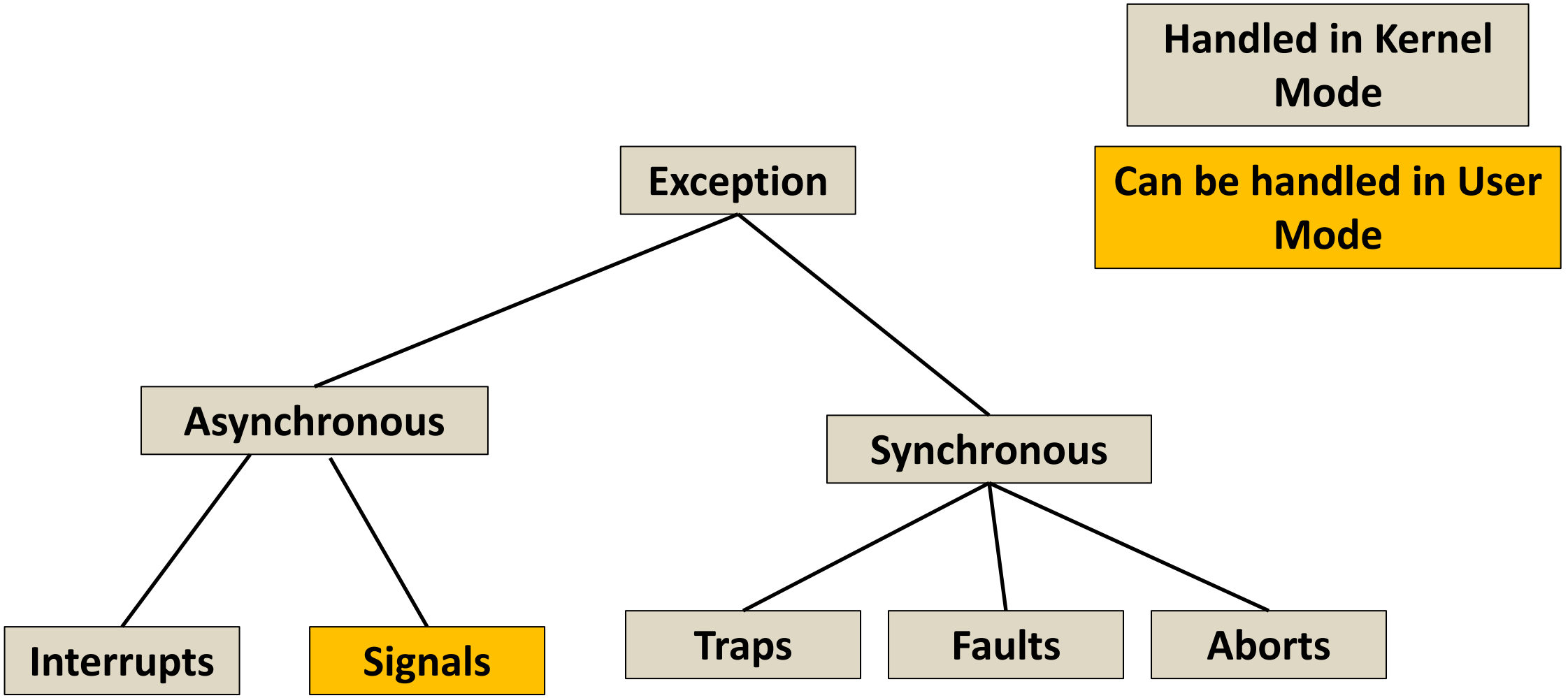


Exception tables

- The kernel has one for interrupts.
- Each process has one for signals.

- The entries are simply the addresses of the handler methods. A special exception handler turns the exception into a kind of procedure call, at which the handler runs like normal code.

(Partial) Taxonomy



Asynchronous Exceptions (Interrupts)

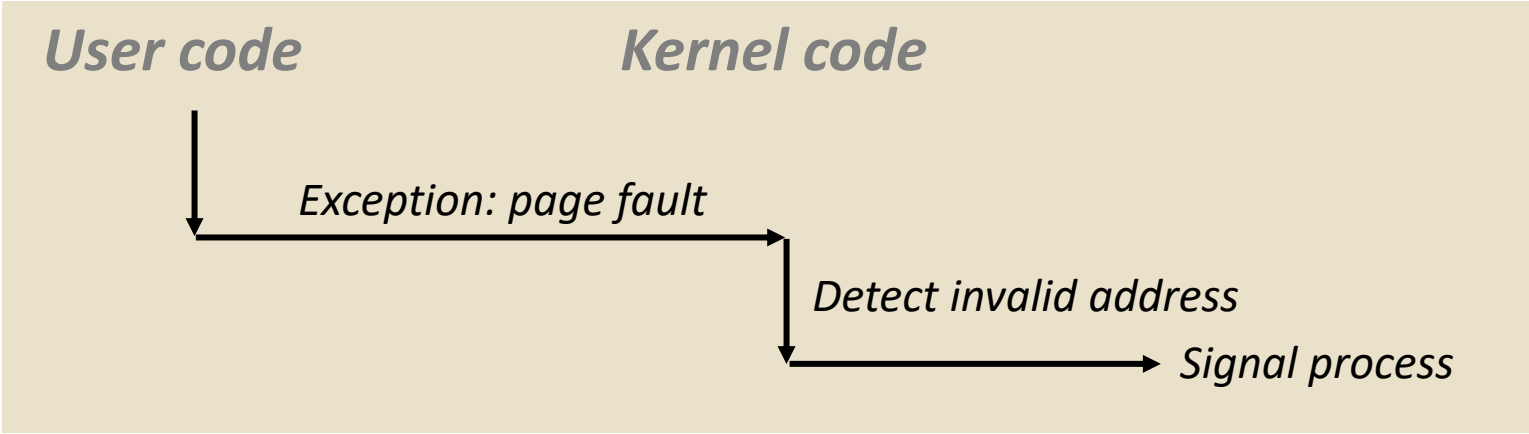
- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to the instruction that was about to execute
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt.
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Typing a character or hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network, or 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. Control resumes at “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… Aborts current program
    - Examples: illegal instruction, parity error, machine check

Fault Example: Invalid Memory Reference

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



Sends SIGSEGV signal to user process

User process exits with “segmentation fault”

Linux signals

- Linux uses a variety of signals to “tell” an active process about exceptions relevant to it. The approach mimics what the hardware does for 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
- The signal must be caught or ignored. Some signals are ignored by default. Others must be caught and will terminate the process if not.
- To catch a signal, a process (or some library it uses) must register a “signal handler” procedure. Linux will pause normal execution and call the handler. When the handler returns, the interrupted logic resumes.

List of Linux signals

SIGABRT Abort signal from abort(3)
SIGALRM Timer signal from alarm(2)
SIGBUS Bus error (bad memory access)
SIGCHLD Child stopped or terminated
SIGCONT Continue if stopped
SIGEMT Emulator trap
SIGFPE Floating-point exception
SIGHUP User logged out or controlling process terminated
SIGILL Illegal Instruction
SIGINFO A synonym for SIGPWR
SIGINT Interrupt from keyboard
SIGIO I/O now possible (4.2BSD)
SIGIOT IOT trap. A synonym for SIGABRT
SIGKILL Kill signal (cannot be caught or ignored)
SIGLOST File lock lost (unused)
SIGPIPE Broken pipe: write to pipe with no readers
SIGPOLL Pollable event (Sys V); synonym for SIGIO

- SIGPROF Profiling timer expired
  SIGPWR Power failure (System V)
  SIGQUIT Quit from keyboard
  SIGSEGV Invalid memory reference
  SIGSTOP Stop process
  SIGTSTP Stop typed at terminal
  SIGSYS Bad system call (SVr4)
  SIGTERM Termination signal
  SIGTRAP Trace/breakpoint trap
  SIGTTIN Terminal input for background process
  SIGTTOU Terminal output for background process
  SIGURG Urgent condition on socket (4.2BSD)
  SIGUSR1 User-defined signal 1
  SIGUSR2 User-defined signal 2
  SIGVTALRM Virtual alarm clock (4.2BSD)
  SIGXCPU CPU time limit exceeded (4.2BSD)
  SIGXFSZ File size limit exceeded (4.2BSD)
  SIGWINCH Window resize signal (4.3BSD, Sun)

List of Linux Signals (signal.h)

```
#define SIGHUP          1
#define SIGINT          2
#define SIGQUIT         3
#define SIGILL          4
#define SIGTRAP         5
#define SIGABRT         6
#define SIGIOT          6
#define SIGBUS          7
#define SIGFPE          8
#define SIGKILL         9
#define SIGUSR1        10
#define SIGSEGV        11
#define SIGUSR2        12
#define SIGPIPE        13
#define SIGALRM        14
#define SIGTERM        15
```

```
#define SIGSTKFLT      16
#define SIGCHLD        17
#define SIGCONT        18
#define SIGSTOP        19
#define SIGTSTP        20
#define SIGTTIN        21
#define SIGTTOU        22
#define SIGURG         23
#define SIGXCPU        24
#define SIGXFSZ        25
#define SIGVTALRM      26
#define SIGPROF        27
#define SIGWINCH       28
#define SIGPOLL        29
#define SIGPWR         30
#define SIGSYS         31
```

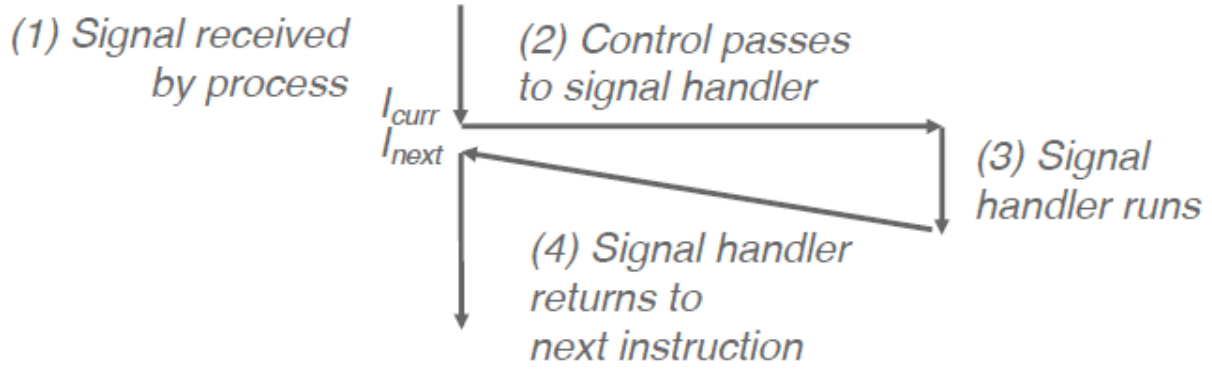
Sending a Signal

- Kernel sends (delivers) 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

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
- 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
    - Like a hardware exception handler being called in response to an asynchronous interrupt



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 delivered, but will not be received until the signal is unblocked
- A pending signal is received at most once

Managing pending and blocked signals

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

Sending signals with the kill command

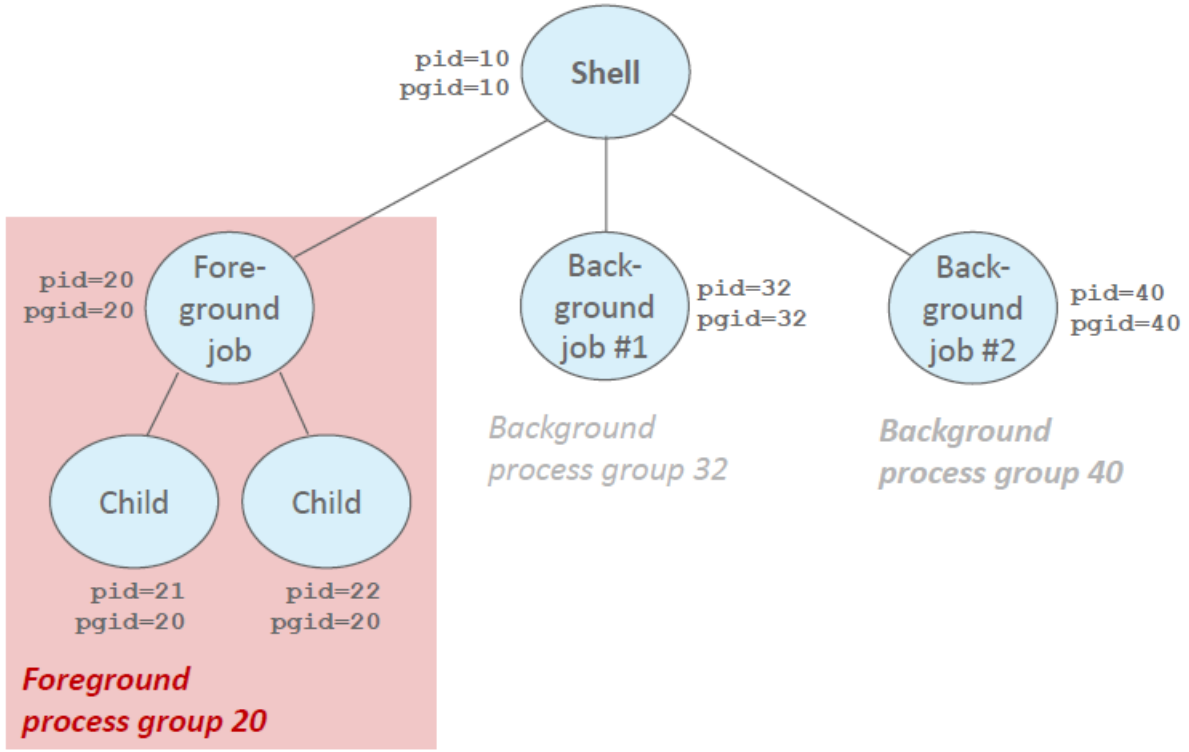
- Can send any signal to a process or to a process group
  - See **`man kill`**
- An interesting example:

  - **`main(){ while(1); }`**

  - **`kill -s 11 pid`**

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



Receiving Signals

- Suppose kernel is returning from an exception handler and is ready 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
    - Choose 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

Signal Handlers

- The **`signal`** function can be used to modify the default action associated with the receipt of a signal:
  - **`sighandler_t *signal(int signum, sighandler_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`**
    - Also 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

`sighandler_t`

- `typedef void (*sighandler_t)(int);`
- A function pointer to a function that gets an int and returns void

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) {
                fprintf(stderr,"signal error");
                exit (1);
        }
        /* Wait for the receipt of a signal */
        pause();
        return 0;
}
```

Notes on the signal() function

The signal() system call is available in all Unix-like systems but it has some undesirable features.

(1). Before executing the installed signal catcher, the signal handler is usually reset to DEFault. In order to catch the next occurrence of the same signal, the catcher must be installed again. This may lead to a race condition between the next signal and reinstalling the signal handler. In contrast, sigaction() automatically blocks the next signal while executing the current catcher, so there is no race condition.

(2). Signal() can not block other signals. If needed, the user must use sigprocmask() to explicitly block/unblock other signals. In contrast, sigaction() can specify other signals to be blocked.

(3). Signal() can only transmit a signal number to the catcher function. Sigaction() can transmit addition information about the signal.

(4). Signal() may not work for threads in a multi-threaded program. Sigaction() works for threads.

(5). Signal() may vary across different versions of Unix. Sigaction() is the POISX standard, which is more portable.

From our textbook Systems Programming in Unix/Linux by K.C. Wang
See the example in Section 6.3.5

Using sigaction() instead

- Gives us greater flexibility, e.g., retrieve the current disposition of a signal without changing it
- More portable than signal()

```
int sigaction(int sig, const struct sigaction *new, struct sigaction *old);
```

- Returns 0 on success, or -1 on error

```
struct sigaction {
       void (*sa_handler)(int);     /* Address of handler */
       sigset_t      sa_mask;       /* Signal blocked during handler
                                       invocation */
       int           sa_flags;      /* Flags controlling handler
                                     /* invocation */
       void (*sa_restorer)(void);  /* Not for application use */
};
```

Blocking signals (temporarily)

```
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);
```

gdb and exceptions

- Allows you to understand where an exception occurred.

- **gdb uses exception handlers**

Exceptions at the language level

- Many programming languages have features to help you manage exceptions.
- For Linux signals, this is done purely through library procedures that register that register the desired handler method.
- But for program exceptions, a program might halt, or there may be a way to manage the exception and resume execution.
- One big difference: Linux can restart a program at the exact instruction and in the exact state it was in prior to an interrupt or signal. But a programming language generally can’t resume the same instruction after an event like a zero divide, so we need a way to transfer control to “alternative logic”