Housekeeping

• Paper reading assigned for next class

Last time...

• We’ve discussed how the OS schedules the CPU
  – And how to block a process on a resource (disk, network)

• Today:
  – How do processes block on each other?
  – And more generally communicate?

What is a signal?

• Like an interrupt, but for applications
  – <64 numbers with specific meanings
  – A process can raise a signal to another process or thread
  – A process or thread registers a handler function

• For both IPC and delivery of hardware exceptions
  – Application-level handlers: divzero, segfaults, etc.

• No “message” beyond the signal was raised
  – And maybe a little metadata
    • PID of sender, faulting address, etc.
    • But platform-specific (non-portable)

Signals and Inter-Process Communication

Don Porter

Outline

• Signals
  – Overview and APIs
  – Handlers
  – Kernel-level delivery
  – Interrupted system calls

• Interprocess Communication (IPC)
  – Pipes and FIFOs
  – System V IPC
  – Windows Analogs

Logical Diagram

Today’s Lecture Process Coordination
Example

```
int main() {
    signal(SIGUSR1, &usr_handler);
    ...  // Register usr_handler() to handle SIGUSR1
}
```

Send signal to PID 300

```
kill(300, SIGUSR1);
```

Basic Model

- Application registers handlers with signal or sigaction
- Send signals with kill and friends
  - Or raised by hardware exception handlers in kernel
- Signal delivery jumps to signal handler
  - Irregular control flow, similar to an interrupt

API names are admittedly confusing

Signal Types

- See man 7 signal for the full list: (varies by sys/arch)
  - SIGTSTP – 1 – Stop typed at terminal (Ctrl+Z)
  - SIGKILL – 9 – Kill a process, for realizzes
  - SIGSEGV – 11 – Segmentation fault
  - SIGPIPE – 13 – Broken pipe (write with no readers)
  - SIGALRM – 14 – Timer
  - SIGUSR1 – 10 – User-defined signal 1
  - SIGCHLD – 17 – Child stopped or terminated
  - SIGSTOP – 19 – Stop a process
  - SIGCONT – 18 – Continue if stopped

Language Exceptions

- Signals are the underlying mechanism for Exceptions and catch blocks
- JVM or other runtime system sets signal handlers
  - Signal handler causes execution to jump to the catch block

Signal Handler Control Flow

From Understanding the Linux Kernel
Alternate Stacks

- Signal handlers execute on a different stack than program execution.
  - Why?
    - Safety: App can ensure stack is actually mapped
    - Avoid assumptions about application not using space below rsp
  - Set with `sigaltstack()` system call
- Like an interrupt handler, kernel pushes register state on interrupt stack
  - Return to kernel with `sigreturn()` system call
  - App can change its own on-stack register state!

Nested Signals

- What happens when you get a signal in the signal handler?
- And why should you care?

The Problem with Nesting

```c
int main() {
  /* ... */
  signal(SIGINT, &handler);
  signal(SIGTERM, &handler);
  /* ... */

int handler() {
  free(buf1);
  free(buf2);
}
```

Nested Signals

- The original `signal()` specification was a total mess!
  - Now deprecated—do not use!
- New `sigaction()` API lets you specify this in detail
  - What signals are blocked (and delivered on `sigreturn`)
  - Similar to disabling hardware interrupts
- As you might guess, blocking system calls inside of a signal handler are only safe with careful use of `sigaction()`

Application vs. Kernel

- App: signals appear to be delivered roughly immediately
- Kernel (lazy):
  - Send a signal == mark a pending signal in the task
  - And make runnable if blocked with TASK_INTERRUPTIBLE flag
  - Check pending signals on return from interrupt or syscall
  - Deliver if pending

Example

```c
int main() {
  read();
}

int usr_handler() { ...

Mark pending signal, Block unblock

Send signal to PID 300
```
Interrupted System Calls

• If a system call blocks in the INTERRUPTIBLE state, a signal wakes it up
• Yet signals are delivered on return from a system call
• How is this resolved?
• The system call fails with a special error code
  – EINTR and friends
  – Many system calls transparently retry after sigreturn
  – Some do not – check for EINTR in your applications!

Default handlers

• Signals have default handlers:
  – Ignore, kill, suspend, continue, dump core
  – These execute inside the kernel
• Installing a handler with signal/sigaction overrides the default
• A few (SIGKILL) cannot be overridden

RT Signals

• Default signals are only in 2 states: signaled or not
  – If I send 2 SIGUSR1’s to a process, only one may be delivered
  – If system is slow and I furiously hit Ctrl+C over and over, only one SIGINT delivered
• Real time (RT) signals keep a count
  – Deliver one signal for each one sent

Signal Summary

• Abstraction like hardware interrupts
  – Some care must be taken to block other interrupts
  – Easy to write buggy handlers and miss EINTR
• Understand control flow from application and kernel perspective
• Understand basic APIs

Other IPC

• Pipes, Sockets, and FIFOs
• System V IPC
• Windows comparison

Pipes

• Stream of bytes between two processes
• Read and write like a file handle
  – But not anywhere in the hierarchical file system
  – And not persistent
  – And no cursor or seek()-ing
  – Actually, 2 handles: a read handle and a write handle
• Primarily used for parent/child communication
  – Parent creates a pipe, child inherits it
Example

```c
int pipe_fd[2];
int rv = pipe(pipe_fd);
int pid = fork();
if (pid == 0) {
    close(pipe_fd[1]); // Close unused write end
    dup2(pipe_fd[0], 0); // Make the read end stdin
    exec("grep", "quack");
} else {
    close(pipe_fd[0]); // Close unused read end
}
```

FIFOs (aka Named Pipes)

• Existing pipes can't be opened—only inherited
  — Or passed over a Unix Domain Socket (beyond today's lec)
• FIFOs, or Named Pipes, add an interface for opening existing pipes

FIFOs, or Named Pipes, add an interface for opening existing pipes

Sockets

• Similar to pipes, except for network connections
• Setup and connection management is a bit trickier
  — A topic for another day (or class)

Select

• What if I want to block until one of several handles has data ready to read?
• Read will block on one handle, but perhaps miss data on a second...
• Select will block a process until a handle has data available
  — Useful for applications that use pipes, sockets, etc.

Synthesis Example: The Shell

• Almost all 'commands' are really binaries
  — /bin/ls
• Key abstraction: Redirection over pipes
  — ‘>’, ‘<’, and ‘|’ implemented by the shell itself

Shell Example

• Ex: `ls | grep foo`
• Implementation sketch:
  — Shell parses the entire string
  — Sets up chain of pipes
  — Forks and exec’s ‘ls’ and ‘grep’ separately
  — Wait on output from ‘grep’, print to console
Job control in a shell

- Shell keeps its own “scheduler” for background processes
- How to:
  - Put a process in the background?
    - SIGTSTP handler catches Ctrl-Z
    - Send SIGSTOP to current foreground child
  - Resume execution (fg)?
    - Send SIGCONT to paused child, use waitpid() to block until finished
  - Execute in background (bg)?
    - Send SIGCONT to paused child, but block on terminal input

Other hints

- Splice(), tee(), and similar calls are useful for connecting pipes together
  - Avoids copying data into and out-of application

System V IPC

- Semaphores – Lock
- Message Queues – Like a mail box, “small” messages
- Shared Memory – particularly useful
  - A region of non-COW anonymous memory
  - Map at a given address using shmat()
- Can persist longer than an application
  - Must be explicitly deleted
  - Can leak at system level
  - But cleared after a reboot

System V Keys and IDs

- Programmers pick arbitrary 32-bit keys
  - Use these keys to name shared abstractions
- Find a key using shmget(), msgget(), etc.
  - Kernel internally maps key to a 32-bit ID

Windows Comparison

- Hardware exceptions are treated separately from IPC
  - Upcalls to ntdll.dll (libc equivalent), to call handlers
- All IPC types can be represented as handles
  - Process termination/suspend/resume signaled with process handles
  - Signals can be an Event handle
  - Semaphores and Mutexes have handles
  - Shared memory equally complicated (but still handles)
- Single select()-like API to wait on a handle to be signaled

Summary

- Understand signals
- Understand high-level properties of pipes and other Unix IPC abstractions
  - High-level comparison with Windows