Signals and Inter-Process Communication (IPC)

Nima Honarmand
Outline

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

• Interprocess Communication (IPC)
  • Pipes and FIFOs
  • System V IPC
What is a Signal?

• Like an interrupt, but for applications
  • < 64 numbers with specific meanings
  • Sending: A process can raise a signal to another process or thread
  • Sending: Kernel can send signals to processes or threads
  • Receiving: 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 + a little metadata
  • PID of sender, faulting address, etc.
Example

Pid 300

```c
int main() {
    ... 
    signal(SIGUSR1, &usr_handler);
    ... 
}
```

Register `usr_handler()` to handle `SIGUSR1`
Example

Send signal to PID 300

```c
int main() {
    ...
}

int usr_handler() {
    ...
}

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
Some Signal Types

See man7 signal for the full list: (varies by sys/arch)

SIGTSTP: Stop typed at terminal (Ctrl+Z)

SIGKILL: Kill a process

SIGSEGV: Segmentation fault

SIGPIPE: Broken pipe (write with no readers)

SIGALRM: Timer

SIGUSR1: User-defined signal 1

SIGCHLD: Child stopped or terminated

SIGSTOP: Stop a process

SIGCONT: 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

Figure 11-2. Catching a signal

Source: Understanding the Linux Kernel
Alternate Stacks

• Signal handlers can execute on a different stack than program execution.
  • 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!
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, SIGSTOP) cannot be overridden
Signal Delivery

• Kernel is 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();
    kill(300, SIGUSR1);
}
int usr_handler() {
    ...}
```

What happens to `read`?

Mark pending signal, unblock

Send signal to PID 300

Pid 300

INTERRUPTIBLE
Interrupted System Calls

• If a system call blocks in the TASK_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!
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);
}
```

Double free!

Calls `munmap()`

Another signal delivered on return from `munmap()`
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()`
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
Other IPC

• Pipes, FIFOs, and Sockets

• System V IPC
Pipes

- Stream of bytes between two processes
  - Stored in a buffer in the kernel
- Read and write like a file descriptor
  - 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 (a.k.a. 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
Sockets

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

• 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 using standard file descriptors 0, 1, and 2
  • 0: standard input
  • 1: standard output
  • 2: standard error
  • ‘>’, ‘<’, and ‘|’ implemented by the shell itself
Shell Example

• Example: `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:
  • How to suspend the foreground process?
    • SIGTSTP default 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 Pipe-related API

- `splice()`, `tee()`, and similar calls are useful for connecting pipes together
- Avoid copying data into and out-of application
System V IPC (1)

• Semaphores – Lock
  • Kernel-managed semaphore identified with a system-wide ID
  • `semget()`, `semctl()`, `semop()`

• Message Queues – Like a mail box, “small” messages
  • A linked list in the kernel, identified with a system-wide ID
  • `msgget()`, `msgctl()`, `msgsnd()`, `msgrcv()`
System V IPC (2)

• Shared Memory – particularly useful
  • A region of non-COW anonymous memory, identified with a system-wide ID
  • `shmget()`, `shmat()`, `shmdt()`
  • Get using `shmget()` and map at a given address using `shmat()`

• Can persist longer than an application
  • Must be explicitly deleted using `shmdt()`
  • Can leak at system level
  • But cleared after a reboot