
Lecture 7: Signals and Events

CSC 469H1F

Fall 2006

Angela Demke Brown



Signals

- Software equivalent of hardware interrupts
- Allows process to respond to asynchronous external events (or synchronous internal events)
 - Process may specify its own signal handlers or may use OS default action
 - Defaults include
 - Ignoring the signal
 - Terminating all threads in the process (with or without a core dump)
 - Stopping all threads in the process
 - Resuming all threads in the process
- Provide a simple form of inter-process communication (IPC)

Signal Terminology

- **Posting** - action taken when event occurs that process needs to be notified of (aka signal generation)
- **Delivery** - action taken when process recognizes arrival of event (aka signal handling)
- **Catching** - if user-level signal handler is invoked, process is said to catch the signal
- **Pending** - signals that have been posted, but not yet delivered

User-Level View

- Write a signal handler function
 - E.g. handle SIGINT (interrupt signal) ourselves

```
void sigint_handler(int sig) {  
    fprintf(stderr, "Interrupted!\n");  
    close(tmp_file_fd);  
    unlink(tmp_file_name);  
}
```

- Install it:

```
struct sigaction new_action, old_action;  
new_action.sa_handler = sigint_handler;  
sigaction(SIGINT, &new_action, &old_action);
```

Other user-level actions

- Block signal delivery by masking signals
 - Similar in spirit to disabling interrupts
 - `sigsetmask(how, newset, oldset)`
- Specify that signal handlers run on separate stack
 - `sigaltstack(signal_stack, old_signal_stack)`
- Retrieve list of pending signals
 - `sigpending(signal_set)`
- Block process until signal is posted
 - `sigsuspend(signal_mask)`
- Send signal to process
 - `kill(pid, signal_number)`

Complications

- Handler may execute at any time
 - Need to be careful of manipulating global state in signal handler
- Signal delivery may interrupt execution of signal handler!
 - code should be re-entrant
 - Should block signals if this is not acceptable
- In some implementations (System V Unix, older Linux kernel, libc4,5), handler is reset to default action when it is dispatched
 - Can lead to ugly races... default is often terminate process
- Only one signal handler per signal per process
 - Can't use in library code
- In many implementations, no signal queuing

Kernel View

- Define fixed set of signals, identified numerically
 - E.g. `#define SIGKILL 9 /* kill program */`
 - Signal sets are bitvectors; each bit position gives the status of corresponding signal
- Process structure has field to mark pending signals
 - FreeBSD: `sigset_t p_siglist;`
- Thread structure has similar field to mark pending signals for each thread
 - `sigset_t td_siglist;`

Signal Posting (FreeBSD)

- Mark bit for specified signal in process' p_siglist, and set process to run
 - Process is woken up if in interruptible sleep
 - Many blocking system calls can be interrupted by signals!
- If process is multi-threaded, search for appropriate thread to post signal to
 - Synchronous signals (caused by thread's execution) are posted only to that thread
 - Other signals search thread list for first thread not masking signal and add to that thread's td_siglist
 - If all threads are masking signals, mark process p_siglist
- Some actions can be taken immediately
 - E.g., stopping or continuing the process

Signal Delivery

- Thread checks pending signals (at least once) each time it enters kernel
- If user-level handler exists, arranges for that handler to be invoked
 - Saves signal state on stack
 - Sets up registers to begin executing user-mode signal handler trampoline
 - Trampoline calls signal handler function
 - When handler returns, trampoline makes `sigreturn()` system call
 - OS cleans up stack

Using Signals

- Used to implement timers
 - E.g. send SIGALRM after N seconds
- Used in some programming language interpreters to implement language-defined exceptions
 - E.g. JamVM, SableVM (open source Java VMs) implement NULL pointer checks by catching the SIGSEGV that the access causes, and then handling it according to the Java specification
- Simple "X has occurred" communication between processes
 - E.g. parent forks child and wants to know when child has completed initialization before continuing, child sends signal to parent, or parent wants to tell all children to stop after a certain amount of time has elapsed
- Portability can be a concern as different systems have different signal behavior
 - E.g. Linux implements signal queues so multiple signals of the same time can be recorded, but FreeBSD just has the bit marking so repeated signals can be lost