signals

Eric Missimer

Signals – Sending Messages to a Program

- Can be thought of as a software interrupt
- Signals are one form of inter-process communication (IPC) one process sends a signal to another
- Can also be used by the kernel to notify the process

Signal Generation

- Hardware Exceptions
- SIGSEGV – segmentation fault
- SIGFPE – fatal arithmetic error

Signal Generation

- Hardware Exceptions
- SIGSEGV – segmentation fault
- SIGFPE – fatal arithmetic error
- Software Reasons
- SIGALRM – generated when a process timer expires

So what happens when a signal occurs?

- Ignore the signal – at least most of them
- Catch the signal – inform the kernel what function to call when the signal occurs
- Apply the default action – most default actions terminate the process

Typical Signals

- SIGINT – generated by terminal driver ctrl+c
- SIGKILL – guarantees a process will be killed, cannot be caught
- SIGTERM – sent by the `kill` command, signals that a process should terminate, can be caught
- SIGTSTP – stop a process to later be resumed
- SIGSTOP – stops a process to later be resumed, cannot be caught
- SIGCONT – resume a stopped process

Under the hood

- Either a hardware event or another process generates the signal
- Permission check if it is another process
- If it is SIGKILL or SIGSTOP the kernel handles the signal
- Otherwise control flow is passed to the process's signal handler while in the context of the process
- After process finishes the signal handler (and assuming it did not exit) control flow will return to where the process was interrupted

Changing signal handler

- `typedef void (*sighandler_t)(int)`
- `sighandler_t signal(int signum, sighandler_t handler)`
  - signal() returns the previous value of the signal handler, or `SIG_ERR` on error.
  - Deprecated

Changing signal handler

- `typedef void (*sighandler_t)(int)`
- `sighandler_t signal(int signum, sighandler_t handler)`
  - signal() returns the previous value of the signal handler, or `SIG_ERR` on error.
  - Deprecated
- `int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact)`
  - More powerful than signal, `struct sigaction` has flags to define certain behaviors
  - Doesn't reset itself to the default handler after a signal occurs which `signal` may do
  - Well defined behavior, `signal`'s behavior can vary

Sending a Signal

- `raise(int sig)` – Send a signal to the calling process
- `kill(pid_t pid, int sig)` – Send a signal to another process
  - `pid > 0` – signal sig is sent to the process with the ID specified by pid
  - `pid == 0` – sig is sent to every process in the process group of the calling process
  - `pid == -1` – sig is sent to every process for which the calling process has permission to send signals, except for process 1
  - `pid < -1` – sig is sent to every process in the process group whose ID is -pid
- Need permission to send signal to another process, typically real or effective user ID must equal the real/effective user ID of the target process

alarm and pause

- `alarm(unsigned seconds)`
  - Causes the system to generate a `SIGALARM` after the specified number of seconds
  - Need to do something with the signal

alarm and pause

- `alarm(unsigned seconds)`
  - Causes the system to generate a `SIGALARM` after the specified number of seconds
  - Need to do something with the signal
- `ualarm(useconds_t usecs, useconds_t interval)` – smaller resolution, if interval is nonzero, the signal will repeat itself every interval seconds after the first.

alarm and pause

- `alarm(unsigned seconds)`
  - Causes the system to generate a `SIGALARM` after the specified number of seconds
  - Need to do something with the signal
- `ualarm(useconds_t usecs, useconds_t interval)` – smaller resolution, if interval is nonzero, the signal will repeat itself every interval seconds after the first.
- Only 1 alarm clock per process
- Don't mix the two

alarm and pause

- `alarm(unsigned seconds)`
  - Causes the system to generate a `SIGALARM` after the specified number of seconds
  - Need to do something with the signal
- `ualarm(useconds_t usecs, useconds_t interval)` – smaller resolution, if interval is nonzero, the signal will repeat itself every interval seconds after the first.
- Only 1 alarm clock per process
- Don't mix the two
- `pause` – suspends a process until a signal is sent

Signal Sets

- Want to manipulate multiple signals at a single time
- `int sigemptyset(sigset_t *set)` – empties set
- `int sigfillset(sigset_t *set)` – includes everything
- `int sigaddset(sigset_t *set, int signo)` – add signal
- `int sigdelset(sigset_t *set, int signo)` – remove signal
- `int sigismember(sigset_t *set, int signo)` – tests for membership

sigprocmask

- `sigprocmask(int how, const sigset_t *set, sigset_t *oset)`
- Used to get and set the signal mask for a process
- `oset` will point to the old signal mask if not NULL
- `how`
  - `SIG_BLOCK` – add signals in `set` to signal mask
  - `SIG_UNBLOCK` – remove signals in `set` to signal mask
  - `SIG_SETMASK` – `set` is now the signal process mask

sigpending

- `int sigpending(sigset_t *set)`
- returns the set of signals that are currently blocked from delivery

sigsetjmp and siglongjmp

- We can longjmp from a signal handler rather than returning normally
- `void siglongjmp(sigjmp_buf env, int val)`

sigsetjmp and siglongjmp

- We can longjmp from a signal handler rather than returning normally
- `void siglongjmp(sigjmp_buf env, int val)`
- What about the setjmp?

sigsetjmp and siglongjmp

- We can longjmp from a signal handler rather than returning normally
- `void siglongjmp(sigjmp_buf env, int val)`
- What about the setjmp?
- `int sigsetjmp(sigjmp_buf env, int savemask)`
- What is this savemask?

sigsetjmp and siglongjmp

- We can longjmp from a signal handler rather than returning normally
- `void siglongjmp(sigjmp_buf env, int val)`
- What about the setjmp?
- `int sigsetjmp(sigjmp_buf env, int savemask)`
- What is this savemask?
- savemask argument is not 0, `sigsetjmp()` will also save the current signal mask