The Signal Concept

- Signals are software interrupts
  - Provides a way of notifying a process of an event
- Every signal has a name and a positive integer
  - See signal(7)
  - `<signal.h>` defines mapping between names and numbers

Generating Signals

- **Command Line:**
  - `kill -KILL 8371`
  - Unconditionally kill process 9371
  - Equivalent to `kill -9 8371`

- **Keyboard:**
  - `ctrl-c`
  - Interactive interrupt key
  - Equivalent to sending SIGINT signal to shell (and its proc group)

- **Program:**
  - `if (kill(8371, SIGTERM) == -1) . . .`

Reliable Signals

- A signal is **generated** by certain events
  - Hardware exception (divide by 0)
  - A software condition becomes true (alarm timer expires)
  - A terminal generates a signal (kill command)
  - The kill(2) system call
  - OS Kernel sets a flag in the process table for the signal

- Signal is **delivered** to a process when signal action is taken

- A signal is **pending** if it is generated, but not delivered

- A process can:
  - **Block** the delivery of a signal or
  - **Ignore** the signal (throw it away)

- If a signal is generated more than once while blocked, the user can have:
  - One delivery (typical case) or
  - Many deliveries (i.e., queue the signals)

- The process **signal mask** indicates the blocked signals

Some Signals and Default Actions

Abnormal Termination

- **SIGINT:** User presses interrupt (usually `ctrl-c`) key
- **SIGPIPE:** Process writes to a pipe after reader has quit
- **SIGALRM:** Alarm clock expires
- **SIGTERM:** Terminate (kill) process
- **SIGKILL:** Terminate (kill) process (Can’t be caught/ignored)

Stop Process

- **SIGTSTP:** User presses suspend (`ctrl-z`) key
- **SIGSTOP:** Stop process (Can’t be caught/ignored)
  - SIGCONT continues process
- **SIGTIN:** Background process attempting to read

Implementation Dependent

- **SIGCHILD:** Child process terminates
- **SIGQUIT:** User presses quit (`ctrl-|`) key ➔ produces core dump
Handling A Signal

- **Semantics**: When signal X occurs, do Y.
  - Y is called the disposition or action

- **Action Choices**
  - **Ignore** the signal
    - Works for most signals
    - SIGKILL and SIGSTOP can NOT be ignored
  - **Catch** the signal
    - Call a signal handler (user-written function)
  - Let the default action occur
    - Every signal has a default action
    - In most cases, terminate the process

Concepts

- **Signal Mask**
  - Indicates the set of signals which should be blocked
    - Blocking means hold for later delivery (different from ignore)
    - Ignore means throw signal away
  - Type is sigset_t
    - Manipulated by five functions
      - sigaddset, sigdelset, sigemptyset, sigfillset, sigismember
    - sigprocmask used to read/write a process' signal mask

- **Signal handler**
  - The function that is called when a signal is caught

- **Signal action**
  - Action associated with a signal
  - sigaction is used to examine/specify action for a signal

Code Example 1 (1)

```c
static void handle_alarm(int signo) {
    ... code to handle alarm ...
    return; // or exit
}

int main(int argc, char *argv[]) {
    struct sigaction action, oaction;
    sigemptyset(&action.sa_mask); // block NO signals
    action.sa_flags = 0; // no special options
    action.sa_handler = handle_alarm; // the handler code
    if (sigaction(SIGALRM, &action, &oaction) < 0) Error...
    for (int i=0; i<5; i++) {
        alarm(2); // send SIGALRM in 2 sec
        pause(); // suspend until get signal
        printf("I woke up!
");
    }
    return 0;
}
```

Code Example 1 (2)

```c
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact)
    
    // Changes the action taken by a process for a signal
    // 'sig' can be any signal except SIGKILL and SIGSTOP
    // Install new action if 'act' is non-null
    // Save old action if 'act' is non-null
    // Return 0 if successful; -1 otherwise

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; // additional signals to block
    int sa_flags;
    void (*sa_restorer)(void); // OBSOLETE
}
```
Code Example 1 (3)

```c
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; // additional signals to block
    int sa_flags;
    void (*sa_restorer)(void); // OBSOLETE

    sa_handler and sa_sigaction
    » Mutually exclusive function pointers
    » SIG_DFL: Default action; SIG_IGN: Ignore signal
    » Or a pointer to user-defined signal handler
    sa_mask
    » Mask of additional signals that should be blocked
    sa_flags
    » Modifies behavior of signal handling
}
```

Signal Related Functions

- **kill**: Send a signal to a process
- **raise**: Send a signal to itself
- **alarm**: Set a timer that will expire in the future
  » Generate SIGALRM when timer expires
- **pause**: Suspend calling process until a signal is delivered to process
- **sigemptyset**: Exclude all signals in signal set
- **sigismember**: Return 1 if a signal is in a signal set
- **sigprocmask**: Examine/modify which signals are blocked from deliver to a process
- **sigpending**: Return blocked/pending signal set
- **sigaction**: Examine/modify action for a signal

Signal Sets

- A **signal set** is a data type (sigset_t) used to represent multiple signals
  » Each signal is represented by a single bit
  » Solaris uses 128 bits
  » Linux uses 1024 bits
- Initialize signal set by first turning all bits either off or on (use sigemptyset or sigfillset)
- Then add and delete specific signals in the set (use sigaddset or sigdelset)

sigprocmask Function

```c
int sigprocmask (int how, const sigset_t *set,
                 sigset_t *oset);
```

- Returns 0 if OK, -1 otherwise
- If oset is nonnull, current signal mask is returned through oset
- If set is nonnull, how indicates how signal mask is modified
  » SIG_BLOCK (=0): New mask is union of old and new mask
  » SIG_UNBLOCK (=1): set indicates signals to unblock
  » SIG_SETMASK (=2): New mask will be in set
- A pending, unblocked signal will be delivered after return from sigprocmask
**sigprocmask Example**

```c
void printMask(void) {
    sigset_t sigset;
    int old_errno;

    old_errno = errno; // if called by sig hndlr
    if (sigprocmask(SIG_BLOCK, NULL, &sigset) < 0) Err...
    if (sigismember(&sigset, SIGALRM))
        printf("SIGALRM will be handled\n");
    if (sigismember(&sigset, SIGVTALRM))
        printf("SIGVTALRM will be handled\n");
    if (sigismember(&sigset, SIGUSR1))
        printf("SIGUSR1 will be handled\n");
    errno = old_errno; // restore
}
```

**Sigsuspend and Sigwait**

- **Mechanisms for suspending a process until a signal occurs**
  - Methods for waiting for an event without busy waiting
  - Allows other processes to use CPU

- **sigsuspend**
  - *Atomically* sets signal mask and suspends process until a signal is caught
    - Old signal mask is restored after return
  - int sigsuspend(const sigset_t *new_sigmask);

- **sigwait**
  - Blocks until any of specified signals is pending and then removes that signal from the set of pending signals and unblocks
  - int sigwait(const sigset_t *sigmask, int *signum);
  - Does not change process signal mask (*unlike sigsuspend*)

**Sigsuspend Example (1)**

- Wait for a single signal `signum`
- Assume signal handler
  - Is setup for `signum` signal
  - Sets `sigrcvd` to 1

```c
static volatile sig_atomic_t sigrcvd = 0;
sigset_t maskall, maskmost, maskold;
int signum = SIGUSR1;
sigfillset(&maskall); // all bits on
sigfillset(&maskmost); // all bits on except `signum`
sigdelset(&maskmost, signum); // .
sigprocmask(SIG_SETMASK, &maskall, &maskold); // new
if (sigrcvd == 0) sigsuspend(&maskmost);
sigprocmask(SIG_SETMASK, &maskold, NULL); // restore mask
```

**Sigsuspend Example (2)**

- *volatile*
  - Value can change asynchronously
  - So compiler should not throw away sigsuspend call even though it looks like `sigrcvd` doesn't change
- *sig_atomic_t*
  - Code that accesses `sigrcvd` is in a critical section
  - The data type should be small enough to be accessed atomically
**Sigwait Example**

- Use sigwait to count the number of times SIGUSR1 signal is delivered to process
  - Note: No signal handler is necessary since the signal is always blocked
  ```c
  int sigcount = 0;
  int signo;
  int signum = SIGUSR1;
  sigset_t sigset;
  if ( (sigemptyset(&sigset) == -1) ||
       (sigaddset(&sigset, signum) == -1) ||
       (sigprocmask(SIG_BLOCK, &sigset, NULL) == -1) ) ... Error ...
  for ( ; ; ) {
    if (sigwait(&sigset, &signo) == -1) ... Error ...
    ++sigcount;
  }
  ```

**Difficulties With Handling Signals**

- Whether POSIX functions that are interrupted by signals should be restarted
  - What happens when a process catches a signal while it is executing a library function?
  - Slow versus fast I/O
- Signal handlers calling non-reentrant functions
  - Any function that changes the value of a static variable, use malloc/free, or use a global data structure
- Handling errors that use the system global variable errno
  - **Async-Safe Function**
    - Problem is that a function may have to wait for a signal handler to complete before it completes
    - Predictable results when called from within a signal handler

---

**Signal Interrupts System Call**

- Signal handling depends on how fast the call is
  - Terminal I/O is slow (may block for a long time)
  - Disk I/O is fast (may block for a short time)
  - Some calls (e.g., getpid) do not block at all
- Slow POSIX calls are interrupted by signals
  - They return after the signal handler returns
  - Return code is -1 with errno = EINTR
  - No clear way to determine which functions can get interrupted except by looking for errno = EINTR in man page of the function !!!
  - **Be careful when calling functions from inside signal handlers !!!**
    - printf is NOT async-safe

**Handling errno**

- A function sets errno because of an error but a signal handler is called before the error message is printed
  - The signal handler could change errno because of an error
  - It should restore the proper value of errno before returning

```c
void myhandler(int signum) {
  int old_errno;
  old_errno = errno;
  ... do something ...
  errno = old_errno;
}
```
Extending xssh

- **xssh should go back to prompt if user enters ctrl-c**
  - Example: Abort long-running foreground command
  - Ctrl-c causes SIGINT to be delivered to process
    - Default: Abnormal termination
    - Use `sigsetjmp` and `siglongjmp`
      - May require unraveling the program stack if buried deep into many function calls
- **xssh should protect the updating of the backgrounded process status table**
  - Use `sigprocmask`

Siglongjmp and Sigsetjmp

- **sigsetjmp** is analogous to a statement label
  - Like placing a marker at the current location
  - `int sigsetjmp(sigjmp_buf env, int savemask);`
    - `env` is initialized with information needed to jump back to the current location
    - Save the current signal mask in the `env` buffer if `savemask` is nonzero
    - Return value is 0 when directly called
    - Return value is `val` argument of `siglongjmp(env, val)` otherwise
- **siglongjmp** is analogous to a goto statement
  - `void siglongjmp(sigjmp_buf env, int val);`

Siglongjmp and Sigsetjmp Example

```c
static sigjmp_buf jmpbuf;
static volatile sig_atomic_t jumpok = 0;
static void handler(int signo) {
    if (jumpok == 0) return;
    siglongjmp(jmpbuf, 1);
}

int main(void) {
    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = handler;
    if ( (sigemptyset(&act.sa_mask) == -1) ||
        (sigaction(SIGINT, &act, NULL) == -1) ) ... Error/Exit ...
    sigsetjmp(jmpbuf, 1); // return here from handler
    jumpok = 1;
    for ( ; ; ) { ... Main Loop ... }
}
```