The Signal Concept

**Generating Signals**
- **Command Line:** `kill -9 8371`
  - Unconditionally kill process 8371
  - Equivalent to `kill -KILL 8371` (send SIGKILL to process 8371)
- **Keyboard:** `ctrl-c`
  - Equivalent to sending SIGINT signal to shell (and its proc group)
- **Program:**
  ```c
  if (kill(8371, SIGTERM) == -1) . . .
  ```

Signals are **asynchronous** event notifications
- Like **software interrupt** but sent by OS to process
- Every signal has a name and a positive integer
  - See `signal(7)`
  - `<signal.h>`: Mapping between names and numbers
    - See `/usr/include/bits/signum.h` (e.g., SIGKILL \(\rightarrow 9\))

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
- **SIGTTIN:** Background process attempts to read from controlling terminal

**Implementation Dependent**
- **SIGCHLD:** Child process terminates
- **SIGQUIT:** User presses quit (`ctrl-q`) key \(\rightarrow\) produces core dump
- See `signal(7)` for list of signals and their values

Project A Keyboard-Generated Signals

**Example:**
- `ls -l | grep Oct > xxx &`
- `find /usr -mtime -60 | grep '*.c'`

**Effect of `ctrl-c` on user processes**
- Terminate all foreground (FG) processes
  - all processes in the FG pipeline
- Return control to command line prompt
- Should NOT terminate background processes or the interactive shell `xssh`

**Unix handling of `ctrl-c`**
- Send SIGINT to all processes in FG process group

**Strategy**
- Put each pipeline in a process group
Terminology

- **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**

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 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 signal
  - `signal` is sometimes the unreliable signals version

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

Code Example 1 (1)

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

int main(int argc, char *argv[]) {
    struct sigaction action, oaction;
    sigemptyset(&action.sa_mask); // additional 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\n");
    }
    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`: Any signal except SIGKILL and SIGSTOP
- `act`: SIG_IGN or SIG_DFL or ptr to handler function
- `oact`: Where to save old action if non-null
- Return 0 if successful; -1 otherwise

**sigaction structure**

```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;      // OBSOLETE
    void (*sa_restorer)(void); // OBSOLETE
};
```

**Code Example 1 (3)**

- `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`
  - Additional signals to be blocked before calling handler
  - `Mask is reset to previous value upon return from handler`

- `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 signal is delivered to process
- `sigemptyset`: Exclude all signals in signal set
- `sigismember`: Return 1 if signal is in 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`
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 value of a static variable, uses malloc/free, or uses 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 !!!**
    - fprintf is NOT async-safe (but usually OK for simple testing)
- **waitpid( ) returns -1 if signal delivered**

Handling Interrupted System Call

- Example: When **waitpid( )** returns -1
  - Check if **errno** is equal to EINTR
  - If so, call **waitpid( )** again
- Alternative
  - Automatically restart interrupted system call

```c
struct sigaction action, oaction;
sigemptyset (&action.sa_mask); // additional signals
action.sa_flags = SA_RESTART; // restart syscall
action.sa_handler = handle_alarm; // the handler code
if (sigaction(SIGALRM, &action, &oaction) < 0) Error...
```

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;
}
```
Effect of Fork and Exec on Child

- **Fork**
  - All pending signals are cleared (discarded)
- **Exec**
  - All pending signals are cleared (discarded)
  - Default signal sets are restored
    - i.e., Signals will be handled in default manner

**xssh Handling of ctrl-c (1)**

- Example: Abort long-running FG command
  - Ctrl-c causes SIGINT to be delivered to FG process group
    - Default: Abnormal termination
- Interactive shell should go back to prompt
  - Use `sigtimeout` and `siglongjmp`
    - May require unraveling the program stack
- Child should terminate
- xssh can’t just ignore SIGINT
  - If ctrl-c happens between Fork and Execvp, child will continue to execvp command

```
   Ignored                  Child drops SIGINT
   Ignored                  Default (Terminate)
   Fork                     Execvp
                         SIGINT handled ok here
```

**xssh Handling of ctrl-c (2)**

- Correct behavior
  - Interactive shell goes back to display prompt
  - Child terminates

```
   Default               Sigaction
   Catch   Sigprocmask
   Block   Fork
   Block   Sigaction
   Sigprocmask
   Catch

   Default               Child
   Sigprocmask
   Block               Execvp
   SIGINT handled
   Default (Terminate)
```

**sigprocmask Function**

```
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
Siglongjmp and Sigsetjmp Example

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 … }
}

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);

Avoiding Common Mistakes

- Use sigaction, NOT signal, system call
  - But OK if OS redefines signal as sigaction

- Avoid serious race conditions
  - A signal handler is an asynchronous thread of control
  - Use signal mask to block unwanted interruptions of a handler

- Make signal handlers do little work
  - If event requires a lot of processing
    - Raise a flag in the handler
    - Have the normal code path check the flag and do the work

- Avoid calling functions that are not async-signal safe inside signal handlers if it can be interrupted by a signal

Setting The Process Group

- Ideas
  - Interactive shell and FG pipeline in FG process group
  - Choose one BG pipeline process to be group leader (pgid = pid)
  - Put all processes in the BG pipeline in the group of the leader

- setpgid( pid, pgid )
  - Set process group of process pid to pgid
    - setpgid( 0, pgid )
      - pid = PID of current process
    - setpgid( pid, 0 )
      - pgid = pid
    - setpgid( 0, 0 )
      - pid = PID of current process
      - pgid = pid

- setpgid( pid )
  - Same as pgid( pid, 0 )

- setpgid( )
  - Same as setpgid( 0, 0 )