Signal Handling

Signals

- Software notification to a process of an event
- *Generated* by an event
- *Delivered* when a process receiving the signal takes an action based on that signal
- Signal *lifetime*
  - Interval between signal generation and delivery
  - There may be considerable time between generation and delivery
- Signal *pending* if it has been generated but not delivered
- Process state must be *running* at the time of signal delivery
- *Catching* a signal
  - Process executes a *signal handler* upon delivery
  - Process can install a signal handler by calling `sigaction(2)` with the name of a user-written function
    - `sigaction(2)` may also be called with `SIG_DFL` or `SIG_IGN` instead of a handler
      - These two actions are not considered to be catching the signal
      - `SIG_DFL` implies default action for the signal
      - `SIG_IGN` implies that the signal should be ignored by the process
      - If signal is ignored, it is thrown away at delivery and has no effect on the process
- *Process signal mask*
  - Decides the action in addition to the current signal handler for that signal
  - Contains a list of currently *blocked signals*
    - Blocked signals are not thrown away as ignored signals
    - If a pending signal is blocked, it is delivered when the process unblocks that signal
  - Process blocks a signal by changing its signal mask by using `sigprocmask(2)`

Generating signals

- Each signal has a symbolic name starting with `SIG`
- All signal names defined in `signal.h`

<table>
<thead>
<tr>
<th>No.</th>
<th>Signal</th>
<th>Description</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>Hang-up (death) on controlling terminal</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Interactive attention signal (usually <code>^C</code>)</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>Interactive termination: core dump (usually `^</td>
<td>`)</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>Invalid hardware instruction</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>Trace trap</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>Process abort</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>7</td>
<td>SIGIOT</td>
<td>IOT trap</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>10</td>
<td>SIGUSR1</td>
<td>User-defined signal 1</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>12</td>
<td>SIGUSR2</td>
<td>User-defined signal 2</td>
<td>Abnormal termination</td>
</tr>
</tbody>
</table>
• Two signals (SIGUSR1 and SIGUSR2) are available for users and do not have a preassigned use

• Some signals such as SIGFPE and SIGSEGV are generated upon certain errors; others are generated by specific calls such as alarm

• You can generate signals from the shell with the kill command
  
  – Historically, many signals’ default action is to terminate the process
  – You can use the signal number or signal name to send a signal
    
    ```
    kill -9 pid
    kill -KILL pid
    ```

• You can get all possible signals by using the command kill -l

• The kill(2) system call
  
  – Used to send signals to a process owned by the user from within the process
  
  – Prototype is given by
    
    ```
    #include <signal.h>
    int kill ( pid_t pid, int sig );
    ```

  – Returns 0 on success; −1 on failure and sets errno
  – A user may send a signal to a process that he owns
    
    * kill determines permissions by comparing UID of caller and target
  – Sending SIGUSR1 to process 3423
    
    ```
    if ( kill ( 3423, SIGUSR1 ) == -1 )
        perror ( "Failed to send the SIGUSR1 signal" );
    ```

  – A child may kill its parent by
    
    ```
    if ( kill ( getppid(), SIGTERM ) == -1 )
        perror ( "Failed to kill parent" );
    ```

  – A process can send signal to itself by using the library function raise(3)
    
    ```
    #include <signal.h>
    int raise ( int sig );
    ```

• Key press

  – A key press causes a hardware interrupt to be handled by the keyboard device driver
  
    * Device driver may perform buffering and editing of keyboard input
  – Two special characters, INTR and QUIT, cause the device driver to send a signal to the foreground process group
  
    * INTR or ´C is used to send SIGINT to foreground process group
    * QUIT can be set by user send sends the SIGQUIT signal
    * You can see the characters by using stty -a

• alarm(2)

  – Sends a SIGALARM signal to the calling process after specified number of seconds
    
    ```
    #include <unistd.h>
    unsigned int alarm ( unsigned int seconds );
    ```

    * Returns the number of seconds remaining on the alarm before the call reset the value, or 0 if no alarm was set
    * It never reports an error
  – Requests to alarm are not stacked
A call to alarm before the previous timer elapses resets the alarm to new value

- Default action for alarm is to terminate the process
- alarm.c

Manipulating signal masks and signal sets

- Process can temporarily prevent a signal from being delivered by blocking it
  - Blocked signals have no effect till they are delivered
- Difference between blocking and ignoring a signal
- sigset_t
  - Used to specify operations on signals (blocking/unblocking) as a set
  - Manipulated by the following functions, using set as a pointer to a set of signals (sigset_t *) and signum being the signal number to be manipulated

```c
#include <signal.h>

int sigaddset ( sigset_t * set, int signum );
int sigdelset ( sigset_t * set, int signum );
int sigemptyset ( sigset_t * set );
int sigfillset ( sigset_t * set );
int sigismember ( const sigset * set, int signo );

* sigemptyset initializes set to an empty set of signals
* sigfillset initializes set to a filled set of all signals
* sigaddset adds the signal signum to the set
* sigdelset removes the signal signum from the set
* sigismember tests whether signum is a member of set
  - sigismember returns 1 if the signal is a member of the set; 0 otherwise
  - Other functions return 0 on success, -1 on failure and set errno

- We can initialize a set with two signals as follows:

```c
sigset_t sig_mask;
if (( sigemptyset ( &sig_mask ) == -1 ) ||
    ( sigaddset ( &sig_mask, SIGINT ) == -1 ) ||
    ( sigaddset ( &sig_mask, SIGQUIT ) == -1 ) )
  perror ( "Failed to set up signal mask" );
```

- sigprocmask function
  - Used to examine and change blocked signals

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

- how specifies the manner in which the signal mask is to be modified; it can take one of the three values
  1. SIG_BLOCK – Add set to the signals that are currently blocked
  2. SIG_UNBLOCK – Signals in set are removed from the current set of blocked signals; you can unblock a signal that is not currently blocked
  3. SIG_SETMASK – Set of blocked signals is set to set
  - set is the signal set to be modified; if set is NULL, no modification is made
  - If oldset is no NULL, the previous value of signal mask is stored in it;
Two signals – `SIGKILL` and `SIGSTOP` cannot be ignored; request to ignore those is silently ignored

- Program `signals.c`
- Function `makepair.c`
  - Takes two path names as parameters and creates two named pipes with these names