CMPSC 311 - Introduction to Systems Programming

Module: Signals

Professor Patrick McDaniel
Fall 2013
Exam 2 scores

- Average: 70.48
- Median: 77.75
- High: 108
- Low: 31
UNIX Signals

• A *signal* is a special message sent through the OS to tell a process (or thread) of some desire or event.

• The process execution stops and special “*signal handler*” code runs.

• The process can resume operation after the signal handling is complete.
Signal types (abbreviated)

Each signal is identified by a number:

```c
/* Signals. */
#define SIGHUP          1       /* Hangup (POSIX). */
#define SIGINT          2       /* Interrupt (ANSI). */
#define SIGQUIT         3       /* Quit (POSIX). */
#define SIGABRT         6       /* Abort (ANSI). */
#define SIGFPE          8       /* Floating-point exception (ANSI). */
#define SIGKILL         9       /* Kill, unblockable (POSIX). */
#define SIGSEGV         11      /* Segmentation violation (ANSI). */
#define SIGTERM         15      /* Termination (ANSI). */
#define SIGSTKFLT       16      /* Stack fault. */
#define SIGCHLD         17      /* Child status has changed (POSIX). */
#define SIGCONT         18      /* Continue (POSIX). */
#define SIGSYS          31      /* Bad system call. */
```

All the signals are `#define(d)` in

```
/usr/include/bits/signum.h
```
Signals as process control

- The operating system uses signals to control how the process runs, or stops running.
  - Signals are sent on errors

```c
#define SIGILL 4       /* Illegal instruction (ANSI). */
#define SIGTRAP 5      /* Trace trap (POSIX). */
#define SIGIOT 6       /* IOT trap (4.2 BSD). */
#define SIGBUS 7       /* BUS error (4.2 BSD). */
#define SIGFPE 8       /* Floating-point exception (ANSI). */
#define SIGSEGV 11     /* Segmentation violation (ANSI). */
#define SIGUSR1 10     /* User-defined signal 1 (POSIX). */
#define SIGUSR2 12     /* User-defined signal 2 (POSIX). */
#define SIGKILL 9      /* Kill, unblockable (POSIX). */
#define SIGCONT 18     /* Continue (POSIX). */
#define SIGSTOP 19     /* Stop, unblockable (POSIX). */
```

- Signals can be used by other applications too

- Control the process execution
Process IDs

- Every process running on the OS is given a unique number called its process ID
  - This is what is used in the OS and for process control to reference that specific running program instance.
- To find a process ID for a program, use the `ps` utility to find the number.

```
$ ps -U mcdaniel

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>30908</td>
<td>?</td>
<td>00:00:00</td>
<td>gnome-keyring-daemon</td>
</tr>
<tr>
<td>30919</td>
<td>?</td>
<td>00:00:00</td>
<td>gnome-session</td>
</tr>
<tr>
<td>30964</td>
<td>?</td>
<td>00:00:00</td>
<td>ssh-agent</td>
</tr>
<tr>
<td>30967</td>
<td>?</td>
<td>00:00:00</td>
<td>dbus-launch</td>
</tr>
<tr>
<td>30968</td>
<td>?</td>
<td>00:00:01</td>
<td>dbus-daemon</td>
</tr>
<tr>
<td>30978</td>
<td>?</td>
<td>00:00:00</td>
<td>at-spi-bus-laun</td>
</tr>
<tr>
<td>30982</td>
<td>?</td>
<td>00:00:00</td>
<td>dbus-daemon</td>
</tr>
<tr>
<td>30985</td>
<td>?</td>
<td>00:00:00</td>
<td>at-spi2-registr</td>
</tr>
<tr>
<td>30999</td>
<td>?</td>
<td>00:00:02</td>
<td>gnome-settings-</td>
</tr>
<tr>
<td>31009</td>
<td>?</td>
<td>00:00:00</td>
<td>pulseaudio</td>
</tr>
<tr>
<td>31011</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfsd</td>
</tr>
<tr>
<td>31017</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfsd-fuse</td>
</tr>
<tr>
<td>31031</td>
<td>?</td>
<td>00:02:43</td>
<td>compiz</td>
</tr>
<tr>
<td>31041</td>
<td>?</td>
<td>00:00:00</td>
<td>dconf-service</td>
</tr>
<tr>
<td>31044</td>
<td>?</td>
<td>00:00:00</td>
<td>gnome-fallback-</td>
</tr>
<tr>
<td>31045</td>
<td>?</td>
<td>00:00:06</td>
<td>nautilus</td>
</tr>
<tr>
<td>31047</td>
<td>?</td>
<td>00:00:01</td>
<td>nm-applet</td>
</tr>
<tr>
<td>31048</td>
<td>?</td>
<td>00:00:41</td>
<td>vmtoolsd</td>
</tr>
<tr>
<td>31049</td>
<td>?</td>
<td>00:00:00</td>
<td>polkit-gnome-au</td>
</tr>
<tr>
<td>31064</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfs-udisks2-vo</td>
</tr>
<tr>
<td>31079</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfs-photov2-vo</td>
</tr>
<tr>
<td>31083</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfs-afc-volume</td>
</tr>
<tr>
<td>31090</td>
<td>?</td>
<td>00:00:00</td>
<td>gvfs-mtp-volume</td>
</tr>
</tbody>
</table>
...
**kill**

- **Kill** is a program that sends signals to processes.

  \[
  \text{kill [-<sig>] <pid>}
  \]

- Where `<sig>` is the signal number and `<pid>` is the process ID of the running program you want to send the signal.

  ‣ If no SIGNUM is given, then SIGTERM is used by default.

```bash
$ ps -U mcdaniel
57613 pts/4   00:00:00 signals
$ kill -1 57613
$ kill -2 57613
$ kill -9 57613
```

```bash
$ ./signals
Sleeping ...zzzzz ..... Signal handler got a SIGHUP!
Signals received : 1
Woken up!!
Sleeping ...zzzzz ..... Signal handler got a SIGNINT!
Signals received : 2
Woken up!!
Sleeping ...zzzzz ..... Killed
```
SIGTERM VS. SIGKILL

- **SIGTERM** interrupts the program and asks it to shut down, which by default it does.
  - Sometimes this does not work (for instance when the process is in a locked state)
  - It is often desirable to add a signal handler to handle the SIGTERM, so that it can gracefully shut down the process, cleanup memory, close files, etc.

- **SIGKILL** kills the process
  - Can lead to inconsistent state, because there is no opportunity to gracefully shutdown the process.

**Definition**: the term *graceful shutdown* refers to the proper and complete sync with secondary storage, disposal of resources, and normal termination.
killall

• **Killall** is a program that sends signals to all instances of a particular program.

  `killall [-<sig>] <name>`

• Where `<sig>` is the signal number and `<pid>` is the name of running program you want to send the signal.
  ‣ If no SIGNUM is given, then SIGTERM is used by default.

```bash
$ killall -1 signals
$ killall -2 signals
$ killall -SIGKILL signals

$ ./signals
Sleeping ...zzzzz ....
Signal handler got a SIGHUP!
Signals received : 1
Woken up!!
Sleeping ...zzzzz ....
Signal handler got a SIGNINT!
Signals received : 2
Woken up!!
Sleeping ...zzzzz ....
Killed
```
raise() 

• raise allows a process to send signals to itself.

    int raise(int sig);

• There are a range of reasons why a process might want to do this.
  ‣ Suspend itself (SIGSTOP)
  ‣ Kill itself (SIGKILL)
  ‣ Reset its configuration (SIGHUP)
  ‣ User defined signals (SIGUSR1..)

```c
void suicide_signal( void ) {
    raise( SIGKILL );
    return; // This will never be reached
}
```
You can create your own signal handlers simply by creating a function

```c
void <fname>( int <var name> )
```

and passing a `function pointer` to the function

```c
sighandler_t signal(int signum, sighandler_t handler);
```

Thereafter, whenever a signal of the type signo is raised, your program is called instead of the default handler.

```c
void signal_handler( int no ) {
    printf( "Sig handler got a [%d]\n", no );
    return;
}
```

```c
signal( SIGHUP, signal_handler );
signal( SIGINT, signal_handler );
```
Function pointers

- A function pointer is a pointer to a function that can be assigned, passed as parameters and called

\[ \text{<return>} (*\text{<var>})(\text{<params>}); \]

- Where
  - \text{<return>} is the return type of the function
  - \text{<var>} is the variable names
  - \text{<params>} are the parameters, separated by commas

```c
int myfunc( int i ) {
    printf( "Got into function with %d\n", i );
    return( 0 );
}

int main( void ) {
    int (*func)(int);
    func = myfunc;
    func( 7 );
    return( 0 );
}
```

$ ./signals
Got into function with 7
$
An alternate approach

• The **`sigaction()`** system call changes the action taken by a process on receipt of a specific signal.

```c
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

• Where:
  ```plaintext
  ‣ `signum` - is the signal number to be handled
  ‣ `act` - is a structure containing information about the new handler, NULL means ignore the signal
  ‣ `oldact` - is a pointer to the previously assigned handler, as assigned in call to function
  ```

```c
struct sigaction new_action, old_action;
new_action.sa_handler = signal_handler;
new_action.sa_flags = SA_NODEFER | SA_ONSTACK;
sigaction( SIGINT, &new_action, &old_action );
```
Why another API?

• Many argue that the `sigaction` function is better:
  ‣ The signal() function does not block other signals from arriving while the current handler is executing; sigaction() can block other signals until the current handler returns.
  ‣ The signal() function resets the signal action back to SIG_DFL (default) for almost all signals.
  ‣ Better tuning of signals/controls of process through flags
    • SA_NODEFER - don’t suspend signals while in handler
    • SA_ONSTACK - provide alternate stack for signal handler
    • SA_RESETHAND - Restore the signal action to the default upon entry to the signal handler.

Note: *In general, sigaction is preferred over signal.*
Putting it all together ...

```c
void signal_handler( int no ) {
    if ( no == SIGHUP ) {
        printf( "Signal handler got a SIGHUP!\n" );
    } else if ( no == SIGINT ) {
        printf( "Signal handler got a SIGINT!\n" );
    }
    return;
}

void cleanup_handler( int no ) {
    return; // Cleanup here
}

int main( void ) {
    struct sigaction new_action, old_action; // Setup the signal actions
    new_action.sa_handler = signal_handler;
    new_action.sa_flags = SA_NODEFER | SA_ONSTACK;
    sigaction( SIGINT, &new_action, &old_action );

    signal( SIGHUP, signal_handler ); // Setup the signal handlers
    signal( SIGTERM, cleanup_handler );

    while ( 1 ) {
        printf( "Sleeping ...zzzzz ....\n" );
        select( 0, NULL, NULL, NULL, NULL );
        printf( "Woken up!!\n" );
    }

    // Return successfully
    return( 0 );
}
```