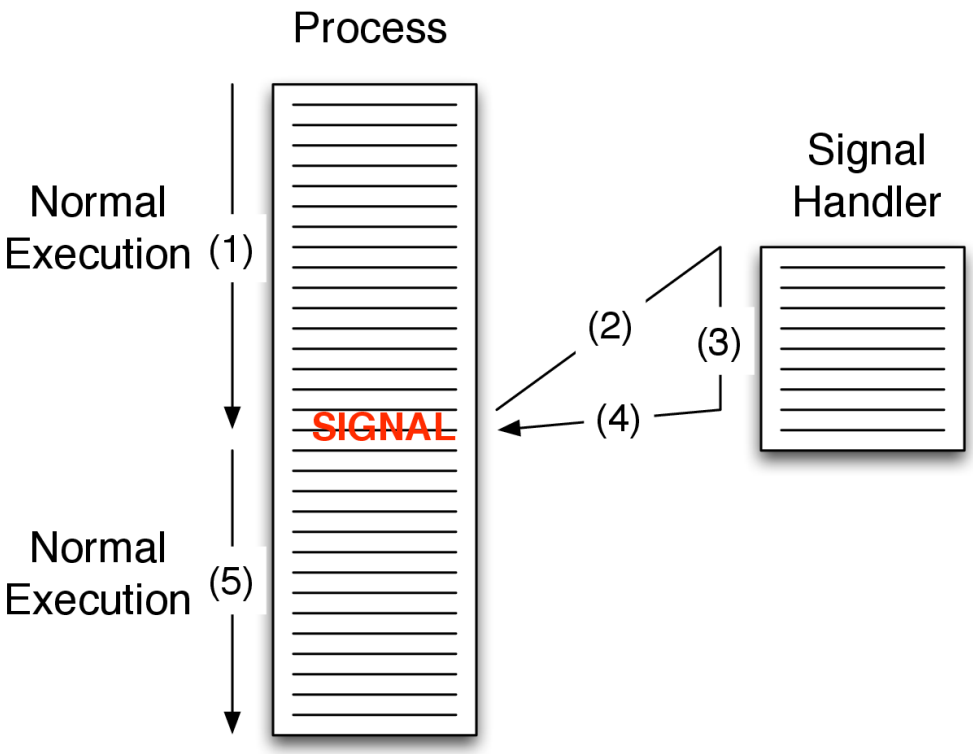
Signals

Devin J. Pohly <djpohly@cse.psu.edu>

Unix signals

- *Signals* are special messages sent through the OS to tell a process (or thread) of some request or event
- Process execution stops and special *signal handler* code runs
  - The process may resume operation after the signal handler finishes
- You can only send signals to your own processes
  - The OS and root can send signals to any process



Signal names

- Each signal is identified by a number:

```
/* 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 SIGPIPE     13  /* Broken pipe (POSIX).  */
#define SIGALRM     14  /* Alarm clock (POSIX).  */
#define SIGTERM     15  /* Termination (ANSI).  */
#define SIGCHLD     17  /* Child status has changed (POSIX).  */
#define SIGCONT     18  /* Continue (POSIX).  */
#define SIGSTOP     19  /* Stop, unblockable (POSIX).  */
#define SIGSYS      31  /* Bad system call.  */
```

- All the signals are `#defined` in header files:
  - To use them, `#include <signal.h>`
  - Actual definitions are in `/usr/include/bits/signum.h`
  - Note: use names when possible; the numbers are not portable!

Signals as process control

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

```
#define   SIGILL        4    /* Illegal instruction (ANSI).  */
#define   SIGTRAP       5    /* Trace trap (POSIX).  */
#define   SIGABRT       6    /* Abort (ANSI).  */
#define   SIGBUS        7    /* BUS error (4.2 BSD).  */
#define   SIGFPE        8    /* Floating-point exception (ANSI).  */
#define   SIGSEGV       11   /* Segmentation violation (ANSI).  */
```

  - Signals can be used by applications as well:

```
#define   SIGUSR1       10   /* User-defined signal 1 (POSIX).  */
#define   SIGUSR2       12   /* User-defined signal 2 (POSIX).  */
```

  - Signals can control process execution:

```
#define   SIGINT        2    /* Interrupt (ANSI).  */
#define   SIGKILL       9    /* Kill, unblockable (POSIX).  */
#define   SIGTERM       15   /* Termination (ANSI).  */
#define   SIGCONT       18   /* Continue (POSIX).  */
#define   SIGSTOP       19   /* Stop, unblockable (POSIX).  */
```

Process IDs

- Every process running on the OS is given a unique number called its *process ID* (or *pid*)
  - This is what is used in the OS and for process control to reference one specific running program instance
- To see process IDs for current processes, use the `ps` utility

```
$ ps -U djpohly
  PID TTY          TIME CMD
  699 ?        00:00:00 systemd
  714 ?        00:00:00 dbus-daemon
  729 ?        01:36:18 mpd
 1021 ?        00:00:00 dbus-daemon
 1022 ?        16:07:09 jackd
 2335 pts/7    00:00:00 bash
 2684 pts/7    00:00:00 ps
 2685 pts/7    00:00:00 xsel
 3710 ?        00:00:00 bash
 3711 ?        00:00:00 startx
 3730 ?        00:00:00 xinit
 3731 ?        01:44:09 X
 3734 ?        00:00:00 .xinitrc
 3739 ?        00:06:27 xcompmgr
 3741 ?        00:03:32 urxvtd
 3745 ?        00:00:12 dwm
 8254 ?        01:09:07 firefox
 8660 pts/3    00:00:00 bash
 8718 pts/4    00:00:00 bash
 9378 ?        00:00:00 dbus-launch
 9379 ?        00:00:00 dbus-daemon
 9381 ?        00:00:00 at-spi-bus-laun
 9387 ?        00:00:09 at-spi2-registr
10470 pts/0    00:00:05 mutt
12920 pts/1    00:00:00 dtach
12922 ?        00:00:00 dtach
12923 pts/2    00:00:20 mutt
16119 ?        00:00:00 mpc
23895 ?        00:00:00 mpdfilter
26870 ?        00:00:03 zathura
27372 ?        00:00:00 loimpress
27388 ?        00:00:00 oosplash
27400 ?        00:00:47 soffice.bin
...
```

The `kill` command

- To send a signal to a process, use the `kill` command (named after its most common use...):

  `kill -SIG pid(s)`

  - SIG: the signal name (e.g., ABRT) or number (e.g., 6)
    - `SIGTERM` is the default if no `-SIG` argument is given
  - pid(s): process IDs of one or more processes to which to send the given signal

```
$ ./signals
Main screen turn on...
We get SIGHUP!
Main screen turn on...
^CWe get SIGINT!
Main screen turn on...
We get SIGTERM!
Exiting cleanly
```

```
$ ps -U djpohly | grep signals
11921 pts/7    00:00:00 signals
$ kill -1 11921
$ # typed Ctrl-C on program
$ kill -TERM 11921
```

SIGTERM vs. SIGKILL

- `SIGTERM` interrupts the program and asks it nicely to shut down
  - Sometimes this will not work (e.g., if the process is in a locked state)
  - It is often desirable to add a signal handler for `SIGTERM` so that your program can clean up memory, close files, and terminate of its own accord
    - This is referred to as a *graceful shutdown*
- **`SIGKILL`** kills the process unceremoniously
  - Can lead to inconsistent state, buffers not written to file, certain system resources not being released, etc.
  - Effective, but should be a last resort
    - If someone tells you to "**`kill -9`**" a process, that's `SIGKILL`

The `killall` command

- The `killall` program sends signals to *all* instances of a particular program:

  `killall -SIG name(s)`

  - SIG: same as for `kill`
  - name(s): one or more program names to which to send the signals

```
$ ./signals
Main screen turn on...
We get SIGHUP!
Main screen turn on...
We get SIGINT!
Main screen turn on...
Killed
```

```
$ killall -HUP signals
$ killall -2 signals
$ killall -KILL signals
$ # ^ that was overkill :(
```

Signal dispositions

- If a process receives a signal for which it does not provide a handler, the signal's default action or *disposition* is invoked:
  - Most signals: terminate the process
  - Some signals: terminate the process and generate a core dump for debugging if enabled (e.g., `SIGSEGV`, `SIGABRT`)
  - A couple of signals: ignored
  - Certain signals: pause the process (`SIGSTOP`) or resume execution (`SIGCONT`)

Meanwhile, back in C...

- The `kill` function sends a signal to a process:

```
int kill(pid_t pid, int sig)
```

  - pid: process ID
  - sig: signal number (use the named constants, e.g., `SIGTERM`)
  - Return value: 0 for success

Raising a signal

- *Raising* a signal simply means a process sending the signal to itself:

```
int raise(int sig);
```

- There are a range of reasons why a process might want to do this:
  - Suspend itself (`SIGSTOP`)
  - Terminate itself or its threads (`SIGTERM`, `SIGKILL`)
  - User-defined signals (`SIGUSR1`, …)
- Return value: 0 for success

Process-defined handlers

- You can create your own signal handler simply by creating a function with the following signature:

```
void funcname(int argname)
```

- To set this function as the handler for a particular signal, you get a *function pointer* to the handler and pass it to the `signal` function:

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

  - "`sighandler_t`" is a function pointer typedef
  - Return value: previous signal handler, or `SIG_ERR`

```
void signal_handler(int num) {
    printf("Got signal #%d\n", num);
}
```

```
signal(SIGHUP, signal_handler);
signal(SIGINT, signal_handler);
```

Function pointers

- A *function pointer* is a variable that points to a function; it can be assigned, passed as a parameter, and called like a regular function.
- To declare a function pointer:

```
rettype (*var)(argtypes);
```

  - rettype: return type of the function
  - var: name of the pointer variable being declared
  - argtypes: types of the arguments, separated by commas
- Probably the most confusing syntax in all of C/C++!
  - Frequently a typedef (like `sighandler_t`) is used to help

Using function pointers

- Just like the name of an array is a pointer to the array, the name of a function is a pointer to the function
  - C will let you use the & operator with the function name for clarity, but it's not required

```c
void foo(int num) {
    printf("foo: num=%d\n", num);
}

void bar(int num) {
    printf("bar: num=%d\n", num);
}

int main(int argc, char **argv) {
    // Declare, assign, and call fp
    void (*fp)(int);
    fp = foo;
    fp(42);
    fp = &bar;
    fp(74);
    return 0;
}
```

```
$ ./fptest
foo: num=42
bar: num=74
```

A new approach

- The sigaction function changes the action taken by a process on receipt of a specific signal:

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

  - signum: what signal to handle
  - act: pointer to a structure containing information about the new handler, NULL means don't change the existing one
  - oldact: pointer to a structure where the old handler info will be stored, NULL means don't we don't want it
  - `man sigaction` to see what the `sigaction` struct contains

Why another API?

- `sigaction` is newer, cleaner, and generally preferred:
  - `signal` does not block other signals from arriving during the current handler (which can cause a race condition); `sigaction` can block other signals until the current handler returns.
  - On some systems, `signal` resets the signal to its default disposition after the handler executes. On others, it doesn't.
  - `sigaction` provides flags for more control over signal behavior:
    - `SA_NODEFER`: don't suspend signals while handler is executing
    - `SA_ONSTACK`: use an alternate stack for the signal handler
    - `SA_RESETHAND`: restore the signal to default disposition when the handler is executed
    - And others, see `man sigaction`

All at once now

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int running = 1;

void signal_handler(int num)
{
     switch (num) {
          case SIGHUP:
               printf("We get SIGHUP!\n");
               break;
          case SIGINT:
               printf("We get SIGINT!\n");
               break;
          case SIGTERM:
               // End the program cleanly
               printf("We get SIGTERM!\n");
               running = 0;
               break;
     }
}
```

```c
int main(int argc, char **argv)
{
     struct sigaction new_act, old_act;
     new_act.sa_handler = signal_handler;
     new_act.sa_flags = 0;
     sigaction(SIGINT, &new_act, &old_act);

     signal(SIGHUP, signal_handler);
     signal(SIGTERM, signal_handler);

     while (running) {
          printf("Main screen turn on...\n");
          // Wait for a signal
          pause();
     }

     // Do code cleanup here
     printf("Exiting cleanly\n");

     return 0;
}
```