SIGNALS

Tanzir Ahmed

CSCE 313 Spring 2021

Signal: User-Mode Exceptional Flow

- A Signal is a **one-word** message with
  - *Each has a numerical code*
- So far, exceptional control flow features have been usable only by the operating system
  - *Recall handling interrupts and exceptions*
  - *All exception handlers run in **protected (Kernel) mode***
- Would like similar capabilities for **user mode code**
  - *Inter-Process communication to facilitate 'exceptional control flow' is subject of today's discussion*
  - *We will discuss them through "Signals" mechanism*
    - .....but also in a broader context beyond just IPC
- For instance, when we press CTRL-C key we ask the Kernel to send the interrupt signal to the currently running process

Major Purposes of Signals

1. Obviously, Inter Process Communication
2. A way for humans to interact with programs using the terminal. For instance,
   - *Ctrl+C sends a interrupt signal SIGINT*
   - *Ctrl+D sends EOF signal*
   - *Ctrl+Z suspends the process by sending SIGSTP*
3. Most importantly, a mechanism for the Kernel for making processes behave according to specifications.
   - *SIGSEGV is sent on memory exceptions*
   - *SIGILL sent on seeing illegal instructions*
   - *SIGPIPE when on attempts to use a broken pipe*

List of Signals

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

Terminate and also dump core (for further investigation)

Only signal ignored by default, unless overridden

Process scheduler??

Source of table: http://man7.org/linux/man-pages/man7/signal.7.html

Multiple values mean that they are different based on architecture

What can a Process do about a Signal?

Tell the Kernel what to do with a signal:

1. **Accept Default action.** All signals have a default action

   signal (SIGINT, SIG_DFL)

2. **Ignore the signal.** Works for most signals

   signal (SIGINT, SIG_IGN)

   cannot ignore SIGKILL and SIGSTOP; also unwise to ignore hardware exception signals

3. **Catch the signal (call a function).** Tell the Kernel to invoke a given function (signal handler) whenever signal occurs.

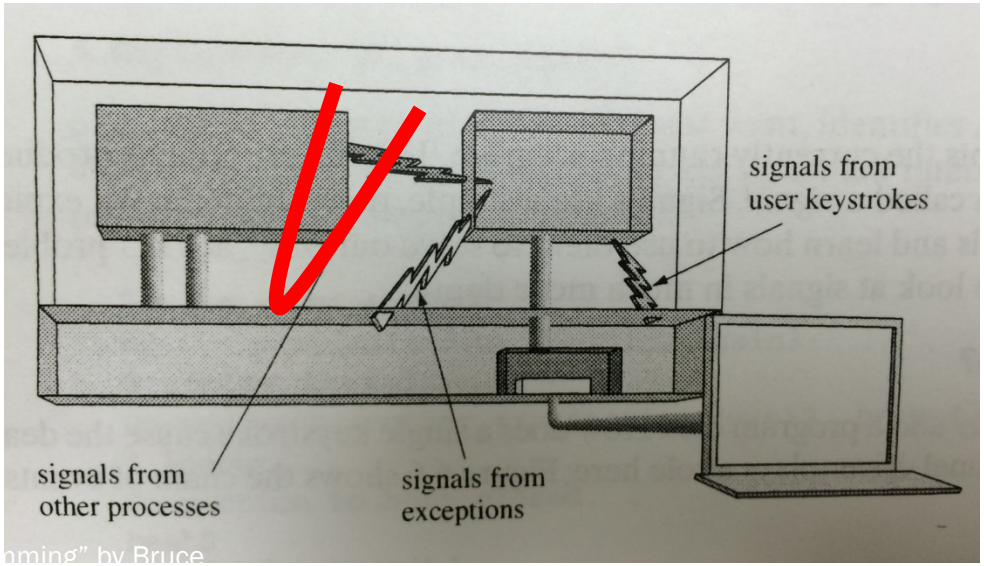
   signal (SIGINT, foo)

Signals – Some Points

- The "Action" in the previous page in many cases indicate the "Default Action"
  - *You can **"override"** the action for some signals (e.g., **SIGINT, SIGUSRx, SIGTERM, SIGSTP**), while for others (**SIGKILL, SIGILL, SIGFPE, SIGSEGV, SIGSTOP**) you **cannot override***
  - *"Overriding" is also called "signal handling", or "catching"*
- SIGCHLD is the only signal so far that is **Ignored** by default
  - *Most others will kill the process except **SIGCONT** and **SIGSTP***
- Does that mean you can kill any process by sending any signal??
  - NO. You can only signal processes that you created
- **SIGSTP** (suspends a process) and **SIGCONT** (conitues/unsuspends a process) are very useful for "**job control**"
  - *How to control a process group tied by a shell session?*
  - *All processes that you run make a group of processes that need special controlling*
  - *You may suspend a process, run it in the background, move to foreground*
  - *What happens to standard input/output of a background process?*

Where do Signals come from?

- Today we'll look at Signals in a broader context
  - *IPC is one of the contexts (one process sending to another)*
- Others are facilitated by Users and Kernel
  - *[Users] Signals generated by external Input devices*
  - *[Kernel] Exceptions*



Where do Signals come from?

(USER) Terminal-generated signals: triggered when user presses certain key on terminal. (e.g. ^C)

`kill(2)` *function: Sends any signal to another process.*

`kill(1)` *command: The command-line interface to* `kill(2)`

`raise(3)`*: Sends a signal to itself*

A user can send signals to only his owned processes

(Kernel) Exception-generated signals: CPU execution detects condition and notifies Kernel. (e.g. **`SIGFPE`** divide by 0, **`SIGSEGV`** invalid memory reference, **`SIGILL`** when an instruction with illegal opcode is found)

(PROCESSES) Software-condition generated signals: Triggered by software event (e.g. **`SIGURG`** by out-of-band data on network connection, **`SIGPIPE`** by broken pipe, **`SIGALRM`** by timer)

Generating Signals: `kill(2)` and `raise(3)`

```
#include <signal.h>

int kill(pid_t pid, int sig);
   /* send signal 'sig' to process 'pid' */
```

```
/* example: send signal SIGUSR1 to process 1234 */
if (kill(1234, SIGUSR1) == -1)
  perror("Failed to send SIGUSR1 signal");

/* example: kill parent process */
if (kill(getppid(), SIGTERM) == -1)
  perror("Failed to kill parent");
```

```
#include <signal.h>

int raise(int sig);
   /* Sends signal 'sig' to itself.
      Part of ANSI C library! */
```

Raise sends a signal to the executing process
Kill sends a signal to the specified process

Signals and the Kernel

- Many of the signals are **generated by** the Kernel in response to events and exceptions received
  - *SIGFPE – FP exception*
  - *SIGILL – Illegal instruction*
  - *SIGSEGV – Segment Violation*
- All others are **routed through** the Kernel, if not originating from the Kernel itself
  - *E.g. SIGHUP (terminal hang), SIGINT (CTRL-C keyboard), SIGTSTP (CTRL-Z), SIGKILL (KILL)*

Simple Signal Handling: Example

linux2.cse.tamu.edu - PuTTY

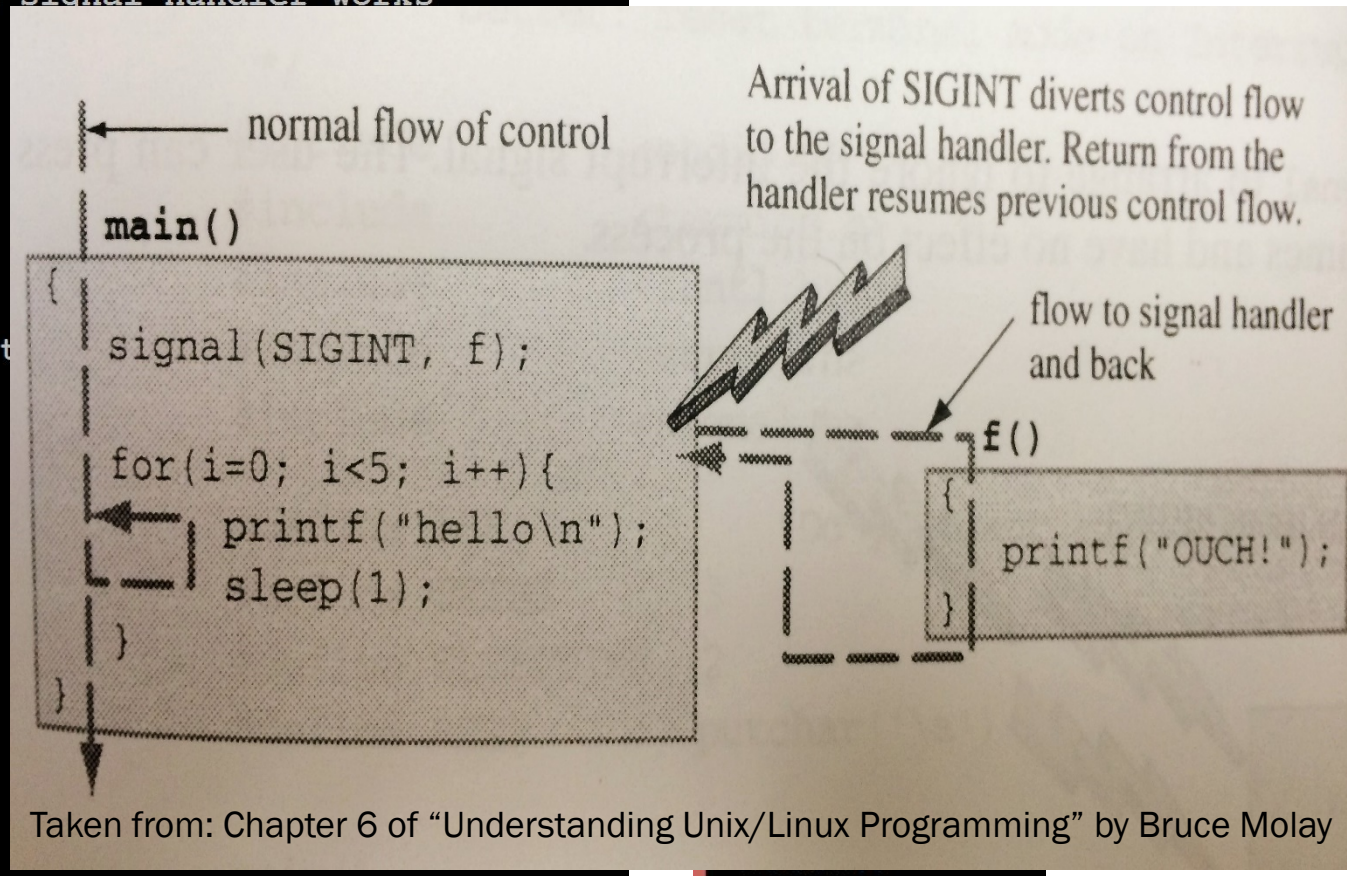
```
:: more sigdemo.c
/* sigdemo.c - shows how a signal handler works
 * credit: Bruce Molay
 */

#include <stdio.h>
#include <signal.h>

main()
{
  void f(int); /* declare t
  int i;

  signal(SIGINT, f);
  for (i=0; i<5; i++) {
    printf("hello\n");
    sleep(1);
  }
}

void f(int signum)
{
  printf("OUCH!\n");
}
```



Taken from: Chapter 6 of “Understanding Unix/Linux Programming” by Bruce Molay

Example – One Handler for Multiple Signals

```
void sig_usr(int signo) { /*argument is signal number*/
  if      (signo == SIGUSR1) printf("received SIGUSR1\n");
  else if (signo == SIGUSR2) printf("received SIGUSR2\n");
  else    error_dump("received signal %d\n", signo);
  return;
}

int main (void) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    perror("cannot catch signal SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    perror("cannot catch signal SIGUSR2");
  for(;;) pause();
}
```

Reaping Child Processes

- A child process being terminated generates SIGCHLD for the parent process
- The parent can handle this signal by reaping children
- This is asynchronous, i.e., the parent process can do other things instead of continually waiting

```
void my_handler (int sig){
    wait (0);
}
int main (){
    // install handler
    signal (SIGCHLD, my_handler);

    // create 5 child procs,
    // i-th proc sleeps i sec and then dies
    for (int i=1; i<=5; i++){
        int pid = fork ();
        if (pid == 0){
            sleep (i);
            return 0;
        }
    }
    // parent in an infinite loop
    // busy doing something else
    while (true){
        cout << "Relaxing" << endl;
        sleep (1);
    }
}
```

Signal Blocking

- A process has the option of manually **blocking** the delivery of a signal.
  - *Signal remains blocked until process either (a) unblocks the signal, or (b) changes the action to ignore the signal.*
- Blocking is automatic for a signal type when the same signal is being handled
  - *First, SIGINT received causes the SIGINT handler (if any) to be called*
  - *Also causes SIGINT to be blocked for the process - If another SIGINT occurs during Handler execution, it is recorded in the pending bit vector, but not delivered*
  - *When handler returns, signals of that type can start to be delivered again*
- The following function is used to block a signal:
  - ***`sigprocmask`*** *(manipulate the set of blocked signals for the process)*

Properly Reaping Child Processes Under Simultaneous Termination

- A child process being terminated generates SIGCHLD for the parent process
- The parent can handle this signal by reaping children
- This is asynchronous, i.e., the parent process can do other things instead of continually waiting

```
void my_handler (int sig){
     while (waitpid (-1, 0, 0) != -1)
          ;
}
int main (){
    // install handler
    signal (SIGCHLD, my_handler);

    // create 5 child procs,
    // i-th proc sleeps i sec and then dies
    for (int i=1; i<=5; i++){
        int pid = fork ();
        if (pid == 0){
            sleep (5);//all die
simult.     return 0;
        }
    }
    // parent in an infinite loop
    // busy doing something else
    while (true){
        cout << "Relaxing" << endl;
        sleep (1);
    }
}
```