What is a Signal?

• A signal is a software interrupt delivered to a process by the OS because:
  – it did something (segfault, FPE)
  – the user did something (pressed ^C)
  – another process wants to tell it something (SIGUSR?)

• Sending a signal is one way a process can communicate with other processes

• Some signals is asynchronous, they may be raised at any time (user pressing ^C)

• Some signals are directly related to hardware (illegal instruction, arithmetic exception, such as attempt to divide by 0) - synchronous signals
Common Signals

- SIGHUP (1): hangup - sent to a process when its controlling terminal has disconnected
- SIGINT (2): interrupt - Ctrl-C pressed by user
- SIGQUIT (3): quit - Ctrl-\ pressed by user
- SIGILL (4): Illegal instruction (default core)
- SIGABRT (6): Abort process
- SIGKILL (9): kill (cannot be caught or ignored)
- SIGSEGV (11): Segmentation fault
- SIGALRM (14): Alarm clock timeout
- SIGUSR[1,2]: User-defined signals
- `kill –l` will list all signals

Process Groups

- Every process belongs to exactly one process group.

- `getpgid()` – Return process group of current process
- `setpgid()` – Change process group of a process
Sending Signals

- **Sending signals from the keyboard**
  - Typing `ctrl-c` (`ctrl-z`) sends a **SIGINT** (**SIGTSTP**) to every job in the foreground process group.
    - **SIGINT**: default action is to terminate each process.
    - **SIGTSTP**: default action is to stop (suspend) each process.

Signals from Keyboard

The most common way of sending signals to processes is using the keyboard:

- **Ctrl-C**: Causes the system to send an INT signal (**SIGINT**) to the running process.
- **Ctrl-Z**: causes the system to send a TSTP signal (**SIGTSTP**) to the running process.
- **Ctrl-\**: causes the system to send a ABRT signal (**SIGABRT**) to the running process.
Signals from Command-Line

• The `kill` command has the following format:

```
kill [options] pid
```

– `-l` lists all the signals you can send
– `-n signal` is a signal number
– the default is to send a `TERM` signal to the process.

• The `fg` command will resume execution of the process (that was suspended with Ctrl-Z), by sending it a `CONT` signal.

```
$ kill 10231 // SIGTERM : default signal
$ kill -9 10231 // SIGKILL
```

Signal Disposition

• Ignore the signal (most signals can simply be ignored, except SIGKILL and SIGSTOP)
• Handle the signal disposition via a `signal handler` routine. This allows us to gracefully shutdown a program when the user presses Ctrl-C (SIGINT).
• Block the signal. In this case, the OS queues signals for possible later delivery
• Let the default apply (usually process termination)
**Actions on Signal**

```
do_signal()
```

- Ignoring the signal
- Executing the default action
  - Executing the signal handler

**Default Actions**

- **Abort** – terminate the process after generating a dump
- **Exit** – terminate the process without generating a dump
- **Ignore** – the signal is ignored
- **Stop** – suspends the process
- **Continue** – resumes the process, if suspended
# Default Signal Actions (BSD)

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>terminate process</td>
<td>terminal line hangup</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>terminate process</td>
<td>interrupt program</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>create core image</td>
<td>quit program</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>create core image</td>
<td>illegal instruction</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>create core image</td>
<td>trace trap</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>create core image</td>
<td>abort program</td>
</tr>
<tr>
<td>7</td>
<td>SIGEMT</td>
<td>create core image</td>
<td>emulate instruction</td>
</tr>
<tr>
<td>8</td>
<td>SIGFPE</td>
<td>create core image</td>
<td>floating-point exception</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>terminate process</td>
<td>kill program</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>create core image</td>
<td>bus error</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>create core image</td>
<td>segmentation violation</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>create core image</td>
<td>non-existent system call</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>terminate process</td>
<td>write on a pipe with</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>terminate process</td>
<td>real-time timer expired</td>
</tr>
<tr>
<td>15</td>
<td>SIGTERM</td>
<td>terminate process</td>
<td>software termination</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SIGURG</td>
<td>discard signal</td>
<td>urgent condition present on socket</td>
</tr>
<tr>
<td>17</td>
<td>SIGSTOP</td>
<td>stop process</td>
<td>stop (cannot be caught or ignored)</td>
</tr>
<tr>
<td>18</td>
<td>SIGTSTP</td>
<td>stop process</td>
<td>stop signal from keyb</td>
</tr>
<tr>
<td>19</td>
<td>SIGCONT</td>
<td>stop process</td>
<td>continue after stop</td>
</tr>
<tr>
<td>20</td>
<td>SIGCHLD</td>
<td>discard signal</td>
<td>child status has changed</td>
</tr>
<tr>
<td>21</td>
<td>SIGTTIN</td>
<td>stop process</td>
<td>background read attempt</td>
</tr>
<tr>
<td>22</td>
<td>SIGTTOU</td>
<td>stop process</td>
<td>background write attempt</td>
</tr>
<tr>
<td>23</td>
<td>SIGIO</td>
<td>discard signal</td>
<td>I/O is possible on a desc</td>
</tr>
<tr>
<td>24</td>
<td>SIGXCPU</td>
<td>terminate process</td>
<td>cpu time limit exceeded</td>
</tr>
<tr>
<td>25</td>
<td>SIGXFSZ</td>
<td>terminate process</td>
<td>file size limit exceeded</td>
</tr>
<tr>
<td>26</td>
<td>SIGVTALRM</td>
<td>terminate process</td>
<td>virtual time alarm</td>
</tr>
<tr>
<td>27</td>
<td>SIGPROF</td>
<td>terminate process</td>
<td>profiling timer alarm</td>
</tr>
<tr>
<td>28</td>
<td>SIGWINCH</td>
<td>discard signal</td>
<td>Window size change</td>
</tr>
<tr>
<td>29</td>
<td>SIGINFO</td>
<td>discard signal</td>
<td>status request from keyboard</td>
</tr>
<tr>
<td>30</td>
<td>SIGUSR1</td>
<td>terminate process</td>
<td>User defined signal 1</td>
</tr>
<tr>
<td>31</td>
<td>SIGUSR2</td>
<td>terminate process</td>
<td>User defined signal 2</td>
</tr>
<tr>
<td>32</td>
<td>SIGTHR</td>
<td>terminate process</td>
<td>thread interrupt</td>
</tr>
</tbody>
</table>
Catching the Signal

User Mode

Kernel Mode

Actions:

• SIG_DSL: Reset to default Action
• SIG_IGN: Ignore Signal
• func(): user defined function

typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
Non-Catchable Signals

- Most signals may be caught by the process, but there are a few signals that the process cannot catch, and cause the process to terminate.
  - For example: KILL and STOP.
- If you install no signal handlers of your own the runtime environment sets up a set of default signal handlers.
  - For example:
    - The default signal handler for the TERM signal calls the exit().
    - The default handler for the ABRT is to dump the process's memory image into a file, and then exit.

Catching a Signal

```c
main(int ac, char *av[])
{
    void    inthandler(int);
    void    quithandler(int);
    char    input[100];

    signal( SIGINT, inthandler ); //set trap
    signal( SIGQUIT, quithandler ); //set trap

    do {
        printf("\nType a message\n");
        if ( gets(input) == NULL )
            perror("Saw EOF ");
        else
            printf("You typed: %s\n", input);
    }
    while( strcmp( input , "quit" ) != 0 );
}
```
Catching a Signal (cont.)

```c
void inthandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(2);
    printf("  Leaving inthandler \n");
}

void quithandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(3);
    printf("  Leaving quithandler \n");
    exit(0)
}
```

User Defined Signals

```c
main(int ac, char *av[])
{
    void    signalhandler(int);

    signal( SIGUSR1, signalhandler );
    signal( SIGUSR2, signalhandler );

    while(1) pause();
}

void signalhandler(int s)
{
    printf(" Received signal %d\n", s );
}
```
STOP & CONT Signals

```c
main(int ac, char *av[]) {
    signal( SIGSTOP, signalhandler );
    signal( SIGCONT, signalhandler );

    int i=0;
    while(1){
        printf("i=%d\n", i++);
        sleep(1);
    }
}

void signalhandler(int s) {
    printf("Received signal %d\n", s);
}
```

Alarming Signals

- SIGALRM can be used as a kind of “alarm clock” for a process
- By setting a disposition for SIGALRM, a process can set an alarm to go off in x seconds with the call:
  - unsigned int alarm(unsigned int numseconds)
- Alarms can be interrupted by other signals
- Examples: mysleep.c, impatient.c
Alarm Signal

```c
main()
{
    void wakeup();

    printf("about to sleep for 4 seconds\n");
    signal(SIGALRM, wakeup); /* catch it */
    alarm(4); /* set clock */
    pause(); /* sleep */
    printf("Morning so soon?\n"); /* back to work */
}

void wakeup()
{
    printf("Wakeup: Alarm received from kernel!\n");
}
```

Interval Timers

```c
#include <sys/time.h>
• int getitimer(int which, struct itimerval *value);
• int setitimer(int which, const struct itimerval *value,
    struct itimerval *ovalue);
```

Three Timers:
• ITIMER_REAL: decrements in real time
• ITIMER_VIRTUAL: decrements only when the process is executing
• ITIMER_PROF: decrements both when the process executes and when the system is executing on behalf of the process.
Interval Timer Struct

```c
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};

struct timeval {
    long tv_sec;                /* seconds */
    long tv_usec;               /* microseconds */
};
```

Interval Time Example

```c
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>

void main()
{
    char x[200];

    signal(SIGALRM, hello);
    set_ticker(5,1);

    while(1)
    {
        printf("enter a word: ");
        fgets(x, 200, stdin);
        printf(">>> %s", x);
    }
}
```
void set_ticker(int start, int interval)
{
    struct itimerval new_timeset;

    new_timeset.it_interval.tv_sec  = interval;
    new_timeset.it_interval.tv_usec = 0;
    new_timeset.it_value.tv_sec     = start  ;
    new_timeset.it_value.tv_usec    = 0 ;

    return setitimer(ITIMER_REAL, &new_timeset, NULL);
}

void hello(int s)
{
    static int      counter = 5;

    printf("hello\n");
    counter--;
    printf("* TICK: counter is now %d\n", counter);
    if ( counter == 0 ){
        printf("* TICK: Time is up!\n");
        exit(0);
        counter = 5;
    }
}
Summary

• Signals
  - Signal Types & Actions
  - Catching Signals
  - STOP & CONT Signals
  - ALARM Signals
  - Interval Timers
  - Generating & Catching Signals

• Read Ch 10 from Stevens Book

Acknowledgments

• Advanced Programming in the Unix Environment by R. Stevens
• The C Programming Language by B. Kernighan and D. Ritchie
• Understanding Unix/Linux Programming by B. Molay
• Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), and M. Shcklette (UChicago).