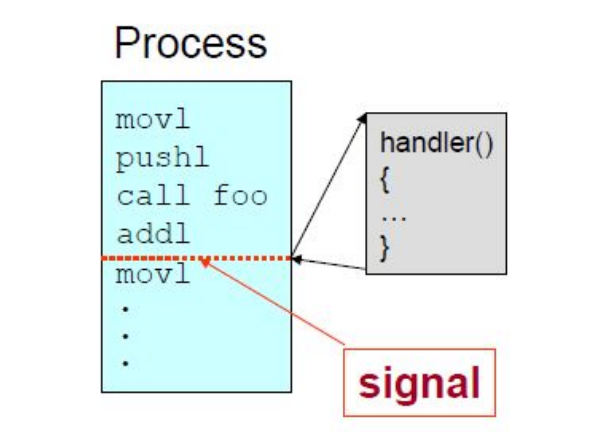
Recitation 7

—

Signals

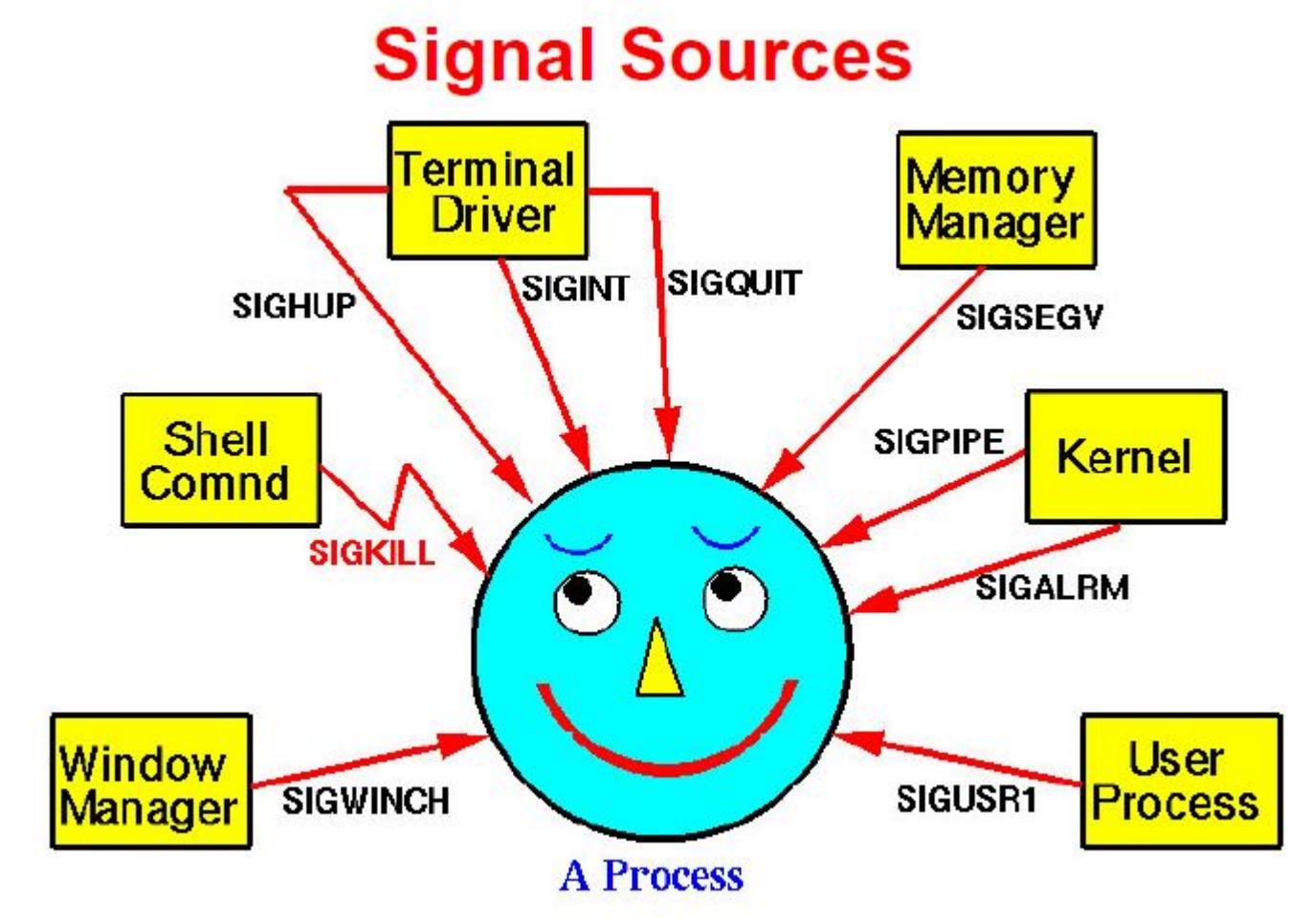
Signals: Introduction

1. A signal is an asynchronous process event which is delivered to a process.
2. Asynchronous implies, it can occur anytime.
3. An event generates a signal. OS stops the process immediately. Signal handler executes and completes. The process resumes where it left off.



Examples of Signals

1. **User types Ctrl+C:** Event generates the “interrupt” signal (SIGINT). OS stops the process immediately. Default handler terminates the process.
2. **Process makes illegal memory reference:** Event generates “segmentation fault” signal (SIGSEGV). OS stops the process immediately. Default handler terminates the process, and dumps core.
3. **Hardware:** Division by 0.
4. **Kernel:** Notifying an I/O device for which a process has been waiting is available
5. **Other processes:** A child notifies its parent that it has terminated.


Signal Sources
Terminal Driver
Memory Manager
SIGHUP
SIGINT
SIGQUIT
SIGSEGV
Shell Comnd
SIGPIPE
Kernel
SIGKILL
SIGALRM
Window Manager
SIGWINCH
SIGUSR1
User Process
A Process

Sending Signals From Keyboard

Steps:

1. Pressing keys generates interrupts to the OS.
2. OS interprets key sequence and sends a signal.

OS sends a signal to the running process:

1. Ctrl+c: INT signal. By default, process terminates immediately.
2. Ctrl+z: TSTP signal. By default, process suspends execution.
3. Ctrl+\: By default, process terminates immediately and creates a core image.

Sending Signals From The Shell

- **`kill -<signal> <PID>`**
  - Example: `kill -INT 1234`
    - Send the INT signal to process with PID 1234
    - Same as pressing Ctrl-C if process 1234 is running
  - If no signal name or number is specified, the default is to send an SIGTERM signal to the process,
- **`fg` (foreground)**
  - On UNIX shells, this command sends a `CONT` signal
  - Resume execution of the process (that was suspended with Ctrl-Z or a command "`bg`")
  - See man pages for `fg` and `bg`

Sending Signal From a Program

- The kill command is implemented by a system call

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Example: send a signal to itself

```
if (kill(getpid(), SIGABRT))
    exit(0);
```

Some predefined signals in UNIX

```
#define SIGHUP      1     /* Hangup (POSIX).  */
#define SIGINT      2     /* Interrupt (ANSI).  */
#define SIGQUIT     3     /* Quit (POSIX).  */
#define SIGILL      4     /* Illegal instruction (ANSI).  */
#define SIGTRAP     5     /* Trace trap (POSIX).  */
#define SIGABRT     6     /* Abort (ANSI).  */
#define SIGFPE      8     /* Floating-point exception (ANSI).  */
#define SIGKILL     9     /* Kill, unblockable (POSIX).  */
#define SIGUSR1     10    /* User-defined signal 1 (POSIX).  */
#define SIGSEGV     11    /* Segmentation violation (ANSI).  */
#define SIGUSR2     12    /* User-defined signal 2 (POSIX).  */
#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 SIGTSTP     20    /* Keyboard stop (POSIX).  */
#define SIGTTIN     21    /* Background read from tty (POSIX).  */
#define SIGTTOU     22    /* Background write to tty (POSIX).  */
#define SIGPROF     27    /* Profiling alarm clock (4.2 BSD).  */
```

More About Signals

1. Signal names are defined in signal.h
2. You can **ignore** some signals.
3. You can catch and handle some signals.
4. Signals have default handlers. Usually terminate the process and generate the core image.
5. Programs can override default for most signals. They can define their own handlers and ignore certain signals or temporarily block them.
6. STOP and KILL are not “catchable” in user programs. KILL terminates the process immediately. The catchable termination signal is TERM.
7. STOP suspends the process immediately. You can resume the process with CONT signal. Catchable suspension signal is TSTP.

The Signal Function

```
void (*signal(int, void (*)(int)))(int);
```

- `signal()` is a function that accepts *two* arguments and returns a pointer to a function that takes one argument, the signal handler, and returns nothing. If the call fails, it returns `SIG_ERR`.
- The arguments are
  - The first is an integer (*i.e.*, `int`), a *signal name*.
  - The second is a function that accepts an `int` argument and returns nothing, the *signal handler*.
  - If you want to ignore a signal, use `SIG_IGN` as the second argument.
  - If you want to use the default way to handle a signal, use `SIG_DFL` as the second argument.

Example

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

void sighandler(int);

int main () {
   signal(SIGINT, sighandler);

   while(1) {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum) {
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
~
~
```

Output

```
-sh-4.2$ ./a.out
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
^CCaught signal 2, coming out...
```

Some Examples

- **The following ignores signal** `SIGINT`

```
signal(SIGINT, SIG_IGN);
```

- **The following uses the default way to handle** `SIGALRM`

```
signal(SIGALRM, SIG_DFL);
```

- **The following installs function** `INThandler()` **as the signal handler for signal** `SIGINT`

```
signal(SIGINT, INThandler);
```

Installing a signal handler

- Predefined signal handlers
  - `SIG_DFL`: Default handler
  - `SIG_IGN`: Ignore the signal
- To install a handler, use

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

  - Handler will be invoked, when signal `sig` occurs
  - Return the old handler on success; `SIG_ERR` on error
  - On most UNIX systems, after the handler executes, the OS resets the handler to `SIG_DFL`

Example: Clean up Temporary Files

- Program generates a lot of intermediate results
  - Store the data in a temporary file (e.g., “temp.xxx”)
  - Remove the file when the program ends (i.e., unlink)

```
#include <stdio.h>

char *tmpfile = "temp.xxx";
int main() {
    FILE *fp;

    fp = fopen(tmpfile, "rw");

    …
    fclose(fp);
    unlink(tmpfile);
    return(0);
}
```

Problem: What about ctrl+c?

- What if user hits control-C to interrupt the process
  - Generates a SIGINT signal to the process
  - Default handling of SIGINT is to terminate the process
- Problem: the temporary file is not removed
  - Process dies before `unlink(tmpfile)` is performed
  - Can lead to lots of temporary files lying around
- Challenge in solving the problem
  - Control-C could happen at any time
  - Which line of code will be interrupted???
- Solution: signal handler
  - Define a “clean-up” function to remove the file
  - Install the function as a signal handler

Solution: Clean-up signal handler

```
#include <stdio.h>
#include <signal.h>
char *tmpfile = "temp.xxx";

void cleanup() {
   unlink(tmpfile);
   exit(1);
}
int main() {
   if (signal(SIGINT, cleanup) == SIG_ERR)
      fprintf(stderr, "Cannot set up signal\n");

   ...
   return(0);
}
```

Ignoring a signal

- Completely disregards the signal
  - Signal is delivered and "ignore" handler takes no action
  - E.g., `signal(SIGINT, SIG_IGN)` to ignore the ctrl-C
- Example: background processes (e.g., "a.out &")
  - Many processes are invoked from the same terminal
    - And, just one is receiving input from the keyboard
  - Yet, a signal is sent to *all* of these processes
    - Causes *all* processes to receive the control-C
  - Solution: shell arranges to ignore interrupts
    - All *background* processes use the SIG_IGN handler

Example: Clean Up Signal Handler

```
#include <stdio.h>
#include <signal.h>
char *tmpfile = "temp.xxx";

void cleanup() {
   unlink(tmpfile);
   exit(1);
}
int main() {
   if (signal(SIGINT, cleanup) == SIG_ERR)
      fprintf(stderr, "Cannot set up signal\n");

   ...
   return(0);
}
```

Problem: What if this is a background process that was *ignoring* SIGINT???

Solution: Check for Ignore handler

- `signal()` system call returns previous handler
  - E.g., `signal(SIGINT, SIG_IGN)`
    - Returns `SIG_IGN` if signal was being ignored
    - Sets the handler (back) to `SIG_IGN`
- Solution: check the value of previous handler
  - If previous handler was "ignore"
    - Continue to ignore the interrupt signal
  - Else
    - Change the handler to "cleanup"

Solution: Modified Signal Call

```
#include <stdio.h>
#include <signal.h>
char *tmpfile = "temp.xxx";

void cleanup() {
   unlink(tmpfile);
   exit(1);
}
int main() {
   if (signal(SIGINT, SIG_IGN) != SIG_IGN)
      signal(SIGINT, cleanup);

   ...
   return(0);
}
```

Solution: If SIGINT was ignored simply keep on ignoring it!

Installing a Signal Handler

1. Prepare a function that accepts an integer, a signal name, to be a signal handler.
2. Call `signal()` with a signal name as the first argument and the signal handler as the second.
3. When the signal you want to handle occurs, *your* signal handler is called with the argument the signal name that just occurred.
4. Two important notes:
   a. You might want to ignore that signal in your handler
   b. Before returning from your signal handler, don't forget to re-install it.

Another example

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

void  INThandler(int);

void main(void)
{
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, INThandler);
    while (1)
        pause();
}
```

```
void  INThandler(int sig)
{
   char c;
   signal(sig, SIG_IGN);
   printf("Ouch, did you hit Ctrl-C?\n",
          "Do you really want to quit [y/n]?");
   c = getchar();
   if (c == 'y' || c = 'Y')
      exit(0);
   else
      signal(SIGINT, INThandler);
}
```

ignore the signal first

reinstall the signal handler

Handling Multiple Signal Types (1/2)

- **You can install multiple signal handlers:**

```
signal(SIGINT, INThandler);
signal(SIGQUIT, QUIThandler);

void  INThandler(int sig)
{
    // SIGINT handler code
}

void  QUIThandler(int sig)
{
    // SIGQUIT handler code
}
```

Handling Multiple Signal Types (2/2)

- **Or, you can use one signal handler and install it multiple times**

```
signal(SIGINT, SIGhandler);
signal(SIGQUIT, SIGhandler);

void SIGhandler(int sig)
{
   switch (sig) {
      case SIGINT:    // code for SIGINT
      case SIGQUIT:   // code for SIGQUIT
      default:        // other signal types
   }
}
```

Blocking a signal

1. To temporarily defer handling the signal. Process can prevent signals from occurring, while ensuring the signal is not forgotten.
2. The process can then handle the signal later.
3. Two ways to block signals:
   a. Affect all signal handlers: sigprocmask()
   b. Affect a specific action: sigaction()

Blocking Signals

- Each process has a signal mask in the kernel
  - OS uses the mask to decide which signals to deliver
  - User program can modify mask with `sigprocmask()`
- `int sigprocmask()` with three parameters
  - How to modify the signal mask (int how)
    - `SIG_BLOCK:` Add `set` to the current mask
    - `SIG_UNBLOCK:` Remove `set` from the current mask
    - `SIG_SETMASK:` Install `set` as the signal mask
  - Set of signals to modify `(const sigset_t *set)`
  - Old signals that were blocked `(sigset_t *oset)`
- **Functions for constructing sets**
  - `sigemptyset(), sigaddset(), …`

Example: Block Interrupt Signal

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

sigset_t newsigset;

int main() {
   sigemptyset(&newsigset);
   sigaddset(&newsigset, SIGINT);
   if (sigprocmask(SIG_BLOCK, &newsigset, NULL) < 0)
      fprintf(stderr, "Could not block signal\n");
   …
}
```

Send a signal to a process

- Use Unix system call `kill()` to send a signal to another process:

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

- `kill()` sends the `sig` signal to process with ID `pid`.
- So, you must find some way to know the process ID of the process a signal is sent to.

Example: Process a(1)

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void SIGINT_handler(int);
void SIGQUIT_handler(int);

int     ShmID;
pid_t   *ShmPTR;
```

used to save shared memory ID

my PID will be stored here

Example: Process a(2)

```
void  main(void)
{
   int    i;
   pid_t  pid = getpid();
   key_y  MyKey;

   signal(SIGINT, SIGINT_handler);
   signal(SIGQUIT, SIGQUIT_handler);
   MyKey = ftok("./", 'a');
   ShmID = shmget(MyKey, sizeof(pid_t), IPC_CREAT|0666);
   ShmPTR = (pid_t *) shmat(shmID, NULL, 0);
   *ShmPTR = pid;
   for (i = 0; ; i++) {
      printf("From process %d: %d\n", pid, i);
      sleep(1);
   }
}
```

Example

```
void SIGINT_handler(int sig)
{
   signal(sig, SIG_IGN);
   printf("From SIGINT: got a Ctrl-C signal %d\n", sig);
   signal(sig, SIGINT_handler);
}

void SIGQUIT_handler(int sig)
{
   signal(sig, SIG_IGN);
   printf("From SIGQUIT: got a Ctrl-\\ signal %d\n", sig);
   printf("From SIGQUIT: quitting\n");
   shmdt(ShmPTR);
   shmctl(ShmID, IPC_RMID, NULL);
   exit(0);
}
```

use `Ctrl-C` *to interrupt*

use `Ctrl-\` *to kill this program*

Example: Process b(1)

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

Void main(void)
{
   pid_t  pid, *ShmPTR;
   key_t  MyKey;
   int    ShmID;
   char   c;

   MyKey = ftok("./", 'a');
   ShmID = shmget(MyKey, sizeof(pid_t), 0666);
   ShmPTR = (pid_t *) shmat(ShmID, NULL, 0);
   pid    = *ShmPTR;
   shmdt(ShmPTR);   /* see next page */
```

detach the shared memory after taking the `pid`

Example: Process b(2)

```
    while (1) {
        printf("(i for interrupt or k for kill)? ");
        c = getchar();
        if (c == 'i' || c == 'I') {
            kill(pid, SIGINT);
            printf("A SIGKILL signal has been sent\n");
        }
        else if (c == 'k' || c == 'K') {
            printf("About to sent a SIGQUIT signal\n");
            kill(pid, SIGQUIT);
            exit(0);
        }
        else
            printf("Wrong keypress (%c).  Try again!\n", c);
    }
}
```