CMPUT 379
Signal Handling

Signals Demo

What happens when you:

- Press CTRL + C
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a `SIGINT` signal
- Press CTRL + Z
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a `SIGSTP` signal

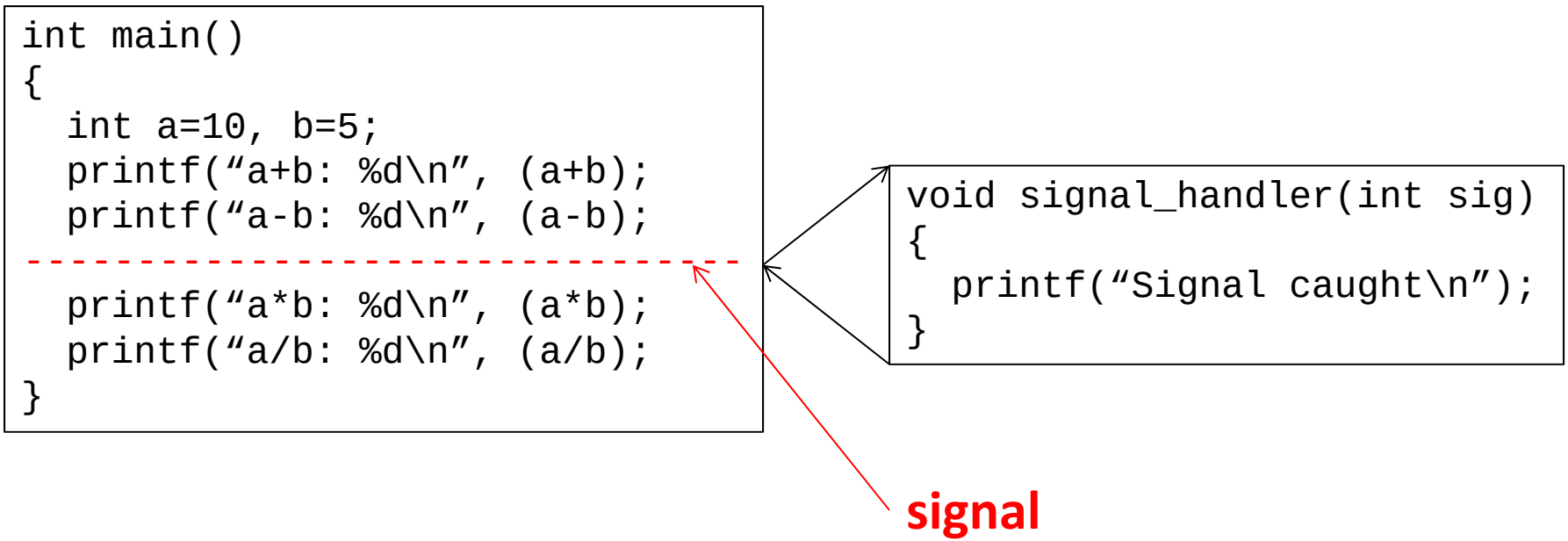
What are signals?

- Event generated by UNIX and Linux systems
  - in response to some condition
  - Process may in turn take some action
- A way for OS to communicate to a process

What are signals?

- Notification of an event
  - Event gains attention of the OS
  - OS stops the application process immediately, sending it a signal
  - Signal handler executes to completion
  - Application process resumes where it left off

What are signals?



Terminology

- Raise
  - Term used to indicate generation of a signal
- Catch
  - Term used to indicate receipt of a signal
- Signals can be acted upon
- Signals can also be ignored

Conditions for signals

- Error conditions
  - Memory segment violations
  - Floating-point processor errors
  - Illegal instructions
- Explicitly sent from one process to another
  - Inter-process communication

Example condition

- Process makes illegal memory reference
  - Event gains attention of the OS
  - OS stops application process immediately, sending it a `SIGSEGV` signal
  - Signal handler for `SIGSEGV` signal executes to completion
    - Default signal handler for `SIGSEGV` signal prints "Segmentation Fault" and exits process

Types (I)

Name	Description
SIGABORT	Process abort
SIGALRM	Alarm clock
SIGFPE	Floating point exception
SIGHUP	Hangup
SIGILL	Illegal instruction
SIGINT	Terminal interrupt
SIGKILL	Kill process
SIGPIPE	Write on pipe with no reader
SIGQUIT	Terminal quit
SIGSEGV	Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Types (II)

Name	Description
SIGCHLD	Child process has stopped or exited
SIGCONT	Continue executing, if stopped
SIGSTOP	Stop executing
SIGTSTP	Terminal stop signal
SIGTTIN	Background process trying to read
SIGTTOU	Background process trying to write

Sending signals via keystrokes

- CTRL + C -> `SIGINT`
  - Default handler exits process
- CTRL + Z -> `SIGTSTP`
  - Default handler suspends process
- CTRL + \ -> `SIGQUIT`
  - Default handler exits process

Sending signals via commands

- `kill` command
  - `kill –signal pid`
    - Send a signal of type **signal** to the process with id **pid**
- Examples
  - `kill -2 1234`
  - `kill –INT 1234`

Sending signals via commands

- `kill()`

```
#include <sys/types.h>
#include <signal.h>

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

- Sending process must have permission
  - Both processes must have the same user ID
  - Superuser can send signal to any process
- Return value
  - Success: `0`
  - Error: `-1` (`errno` is set appropriately)
    - `EINVAL` if invalid
    - `EPERM` if no permission
    - `ESRCH` if specified process does not exist

Sending signals via commands

- `raise()`
  - Commands OS to send a signal to current process

```
#include <sys/types.h>
#include <signal.h>

int raise(int sig);
```

- Return value
  - Success: `0`
  - Error: `-1` (`errno` is set appropriately)

Sending signals via commands

- `alarm()`
  - Schedule a `SIGALRM` at some time in future

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

- Processing delays and scheduling uncertainties
- Value of `0` cancels any outsanding alarm request
- Each process can have only one outstanding alarm
- Calling `alarm` before signal is received will cause alarm to be rescheduled

Signal handling

- Each signal type has a default handler
  - Most default handlers exit the process
- A program can install its own handler for signals of any type
  - Exceptions
    - `SIGKILL`
      - Default handler exits the process
      - Catchable termination signal is SIGTERM
    - `SIGSTOP`
      - Default handler suspends the process
      - Can resume process with signal SIGCONT
      - Catchable suspension signal is SIGTSTP

How to handle signals?

- `signal()`

```
#include <signal.h>

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

- `signal()` returns a function which is the previous value of the function set up to handle the signal
- Second argument is the function indicating the signal handler to install for `sig`
  - OR one of these two special values
    - `SIG_IGN` – Ignore the signal
    - `SIG_DFL` – Restore default behavior

Example

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

void ouch(int sig)
{
  printf("OUCH – I got signal %d\n", sig);
  (void) signal(SIGINT, SIG_DFL);
}

int main()
{
  (void) signal(SIGINT, ouch);
  while(1)
  {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Signal handling

- Can install same signal handler for multiple signals

```
(void) signal(SIGINT, sig_handler);
(void) signal(SIGHUP, sig_handler);
(void) signal(SIGILL, sig_handler);
(void) signal(SIGFPE, sig_handler);
(void) signal(SIGABRT, sig_handler);
(void) signal(SIGTRAP, sig_handler);
(void) signal(SIGQUIT, sig_handler);
...
```

POSIX Signal Handling

- **C90** standard
  - `signal()` and `raise()` functions
    - Works across all systems (UNIX, Linux, Windows) but …
    - Works **differently** across some systems!!!
      - Blocked signals during handler execution
      - Reinstall handler after every signal invocation
    - Does not provide mechanism to block signals

POSIX Signal Handling

- POSIX standard
  - `sigaction()` and `sigprocmask()` functions
    - Works the same across all POSIX-compliant UNIX systems (Linux, Solaris etc) but…
    - Do not work on non-UNIX systems (e.g. Windows)
    - Provides mechanism to block signals in general

Robust interface

- `sigaction()`

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

- `struct sigaction` has at least
  - `void (*) (int) sa_handler` `// function, SIG_DFL, SIG_IGN`
  - `sigset_t sa_mask` `// signals to block in sa_handler`
  - `int sa_flags` `// signal action modifiers`

- Signals of type X are automatically blocked when executing handler for signals of type X

- Return value
  - Success: `0`
  - Error: `-1` (`errno` is set appropriately)

Example

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

void ouch(int sig)
{
  printf(“OUCH! - I got signal %d\n”, sig);
}

int main()
{
  struct sigaction act;

  act.sa_handler = ouch;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;

  sigaction(SIGINT, &act, 0);

  while(1) {
    printf(“Hello World!\n”);
    sleep(1);
  }
}
```

setjmp() and longjmp()

- Routines to perform complex flow of control in C/Unix

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- In the setjmp() call, env is initialized with information about current state of the stack
- longjmp() call causes the stack to be reset to its env value.
- Execution restarts after the setjmp() call, but this time setjmp() returns val

Example

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

static jmp_buf env;

void sig_hander(int signo) {
  longjmp(env, 1);
}

int main() {
  (void) signal(SIGSEGV, sig_handler);

  if(!setjmp(env))
    printf("Jump marker set\n");
  else {
    printf("Segmentation fault occurred");
    exit(-1);
  }

  char *str = "Hello";
  str[2] = 'Z';
}
```

Status of variables

- POSIX standard says
  - Global and static variable values will not be changed by the `longjmp()` call
- Nothing is specified about local variables
  - Most implementations do not roll back their values

References

- "Beginning Linux Programming" – 4th edition
- COS 217 Spring 2008 – Princeton University
  - www.cs.princeton.edu/courses/archive/spr08/cos217/lectures/23Signals.ppt
- CS 355 Spring 2010 – Bryn Mawr College
  - www.cs.brynmawr.edu/cs355/labs/lab02.pdf
- CSCI 1730 Spring 2012 – University of Georgia
  - www.cs.uga.edu/~eileen/1730/Notes/signals-UNIX.ppt
- "Use reentrant functions for safer signal handling" – Dipak Jha (IBM)
  - http://www.ibm.com/developerworks/linux/library/l-reent/index.html