CMPUT 379
Lab 1 - Signal Handling

Winter 2018

Slides adapted from:
Mark Stevens (mjsteven@ualberta.ca)
Sepehr Kazemian(skazemia@ualberta.ca)

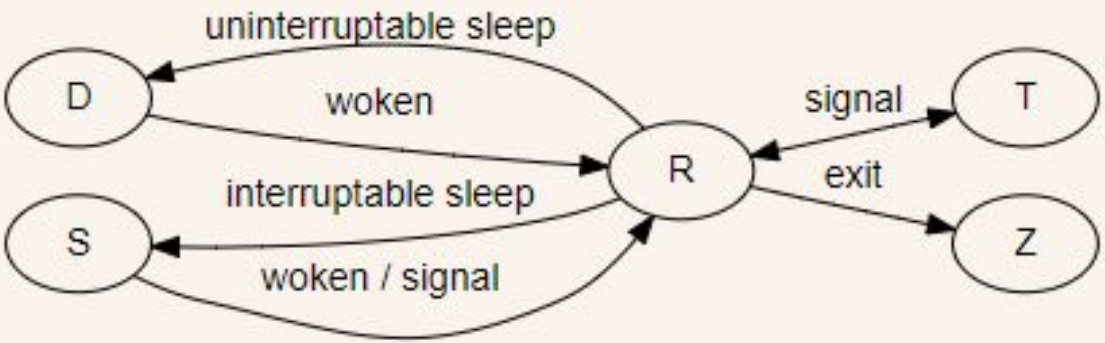
Thanks to:
Muhammad Waqar
Christopher Solinas

Introduction:
How Linux works?

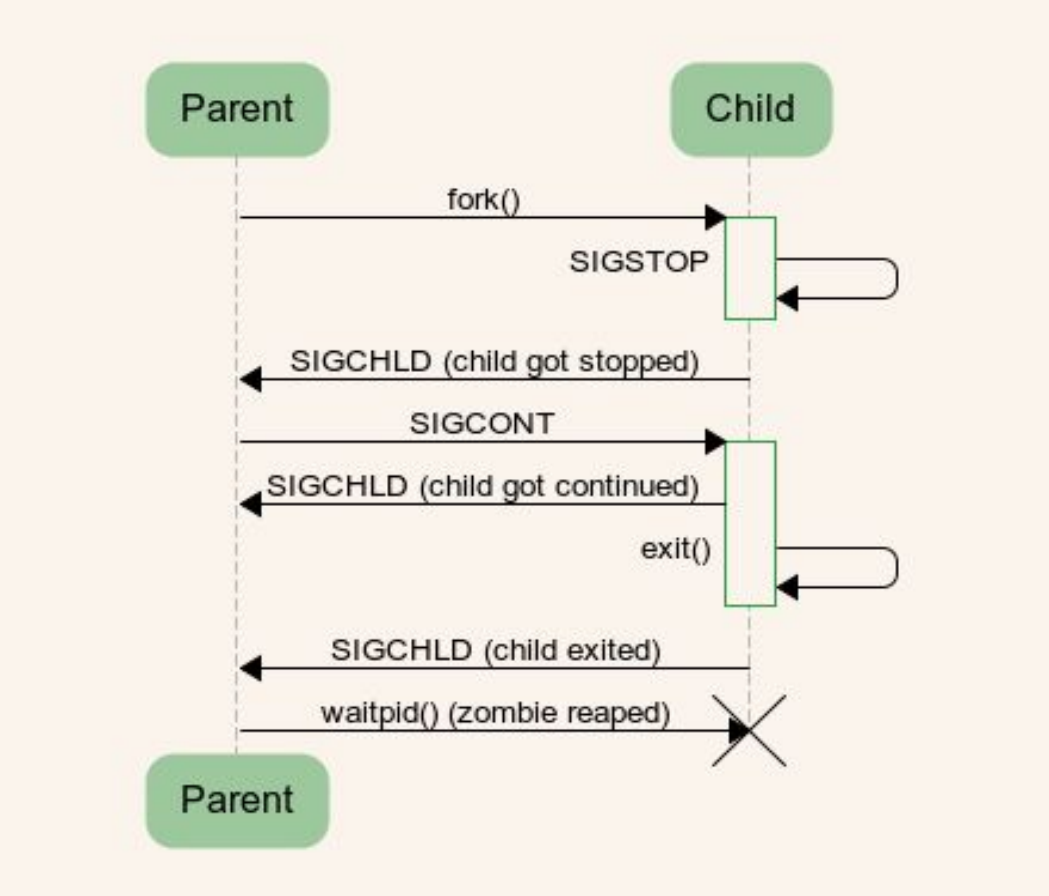
Let's use "ps" to look:

```
PROCESS STATE CODES
   R  running or runnable (on run queue)
   D  uninterruptible sleep (usually IO)
   S  interruptible sleep (waiting for an event to complete)
   Z  defunct/zombie, terminated but not reaped by its parent
   T  stopped, either by a job control signal or because
      it is being traced
   [...]
```

Introduction:
How Linux works?



Example:


Parent
Child
fork()
SIGSTOP
SIGCHLD (child got stopped)
SIGCONT
SIGCHLD (child got continued)
exit()
SIGCHLD (child exited)
waitpid() (zombie reaped)
Parent

Today

1. Signals
   a. What are they?
   b. Why do they exist?
   c. What do we do with signals?
   d. How do we stop signals? sigprocmask()
   e. Why block signals?
2. Signal Handling
   a. signal()?
   b. sigaction()?
   c. Why sigaction() over signal()?

Signals

What happens when you:

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

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

What are signals?

- Asynchronous events generated by UNIX and Linux systems, relating to a running process
  - In response to some condition
  - Process may in turn take some action
- A way for OS to communicate to a process

Basic Signal Flow

1. OS becomes aware of an event (signal) for a process.
2. OS can do one of three things:
   a. Ignore the signal, letting the process continue its normal execution.
   b. Let the process catch the signal. This can be done through custom-made signal handlers that execute from within the process.
   c. Let the signal's default action occur. This default action is usually to terminate the process.
3. Some signals cannot be ignored or handled. Others may result i undefined behavior.

Signal Handler

A **signal handler** is a function which is called by the target environment when the corresponding **signal** occurs.

Signal Handler Flow

1. Signal handlers can be created and specified for a given signal.
2. When the signal is received by the OS, it calls this handler function
3. Signal handler function then executes to completion.
4. Application process resumes where it left off before the signal was raised

Example:

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

void sig_handler(int signo)
{
  if (signo == SIGINT)
    printf("received SIGINT\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
  printf("\ncan't catch SIGINT\n");
  // A long long wait so that we can easily issue a signal to this process
  while(1)
    sleep(1);
  return 0;
}
```

Example:

```
$ ./sigfunc
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
```

Terminology

- Raise
  - Term used to indicate generation of a signal
- Catch
  - Term used to indicate receipt of a signal

Note

A **signal handler** can be specified for all but two **signals** (SIGKILL and SIGSTOP cannot be caught, blocked or ignored)

Common Use Cases

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

Example

- 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

Keystroke Signals

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

Linux 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`

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

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)

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 outstanding alarm request
- Each process can have only one outstanding alarm

- Calling `alarm` before signal is received will cause alarm to be rescheduled

Blocking Signals - sigprocmask()

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
    sigset_t *oldset);
```

- Used to manipulate the list of symbols that are blocked for a process
- Return
  - 0 on success
  - -1 on failure, sets `errno` to error code
- `how`
  - add, remove, set the list
- `*set`
  - the set of signals to block
- `*oldset`
  - the old set of signals to block

How to handle signals? sigaction()

```
sigset_t sa_mask;
```

- `sa_mask`
  - The mask of signals to block in signal handler
- `sigemptyset()`
  - clears the signal mask
- `sigdellset()`
  - fills the signal mask
- `sigaddset()`
  - add a signal to the mask
- `sigemptyset()`
  - remove a signal from the mask

Why block signals?

- Allows you to complete critical code without interruption
  - The signal is triggered after you unblock the signals
- Avoid race conditions in exception handling

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()

- `signal()`

```
#include <signal.h>

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

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

Example - signal()

```
#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 - signal()

- 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,  ;
(void)  signal(SIGQUIT,  sig_handler)
...                      ;
                         sig_handler)
                         ;
```

How to handle signals? sigaction()

- `sigaction()`

```
#include <signal.h>

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

struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int,siginfo_t *, void *);
   sigset_t sa_mask;
   int   sa_flag;
   void (*sa_restorer)(void);
}
```

How to handle signals? sigaction()

```
#include <signal.h>

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

- Return
  - 0 on success
  - -1 on failure, sets `errno` to error code
- signum
  - the signal to handle
- *act
  - the signal handling struct to handle the signal moving forward
- *oldact
  - the old signal handling struct
  - can be `NULL` if you don't want to save it

How to handle signals? sigaction()

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int,siginfo_t *, void *);
    sigset_t sa_mask;
    int   sa_flag;
    void (*sa_restorer)(void);
}
```

- `*sa_handler & *sa_sigaction`
  - The signal handler
  - Use only one of them
- `sa_mask`
  - Data structure controlling which signals can interrupt the signal handler
- `sa_flag`
  - bitmask flag for additional options
- `*sa_restorer`
  - Obsolete may not even exist on modern OS

signal() vs sigaction()

- `signal()` is specified in the C90 language standard
- Works across all platforms
  - (Unix, Linux, Windows)
- Works differently across different systems
  - You might have to reinstall handler after every signal invocation
- Does not provide mechanism to block signals

signal() vs sigaction()

- `sigaction()` is the POSIX compatible
  - Works on Unix and some other OSs
- Provides a method to block signals for the handler
  - Much safer because of this

Use `sigaction()` over `signal()` in MOST case

References

- "Advanced Programming in the UNIX Environment" – Third Edition
- "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

References

- https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/
- https://en.wikipedia.org/wiki/C_signal_handling
- https://idea.popcount.org/2012-12-11-linux-process-states/