

# Signals

Instructor: Linyi Li

*Slides adapted from Dr. B. Fraser*

# Topics

- 1) We can create processes, but **how can they communicate?**
  - a) How can regular code with loops and functions **handle an asynchronous communication?**
  - b) How can a **child send a message to the parent?**

# Introduction to Signals

# Signals

- Signals are..
    - Programs and the kernel can send signals to itself or other programs.
  - Wonka Golden Ticket Example
    - Parent process spawns children to search for a golden ticket.
    - Parent: ..
    - Child: ..
- when discovered a ticket.



# Pseudocode for Signals

- Parent

```
handler() {  
    print "Found ticket!"  
}  
  
main() {  
    pid = fork()  
  
    if (pid != 0) {  
        register signal handler  
        wait forever  
    }  
}
```

sigaction(...)

- Child

```
...  
main() {  
    ...  
    if(pid != 0) {  
        ...  
    } else {  
        if (found_ticket()) {  
            signal parent  
        }  
    }  
}
```

kill(...)

# Function Pointers

- Variables

- Normal variables hold values.
- Pointers hold the address of a variable.
- Function pointers..

They allow us to pass around (and call) functions

my\_function

A curved arrow originates from the 'my\_function' label and points to the 'handler()' function definition.

```
handler() {  
    print "Found ticket!"  
}
```

# Why Function Pointers?

- Imagine an embedded system receiving bluetooth data.
  - How does the bluetooth module / library ..
- Idea 1:
  - Application just keep asking it!
  - Slow, power hungry!
- Idea 2:
  - Have bluetooth module directly execute our application's code!
  - How? Have the module to call our function.
  - How? Give it..

# Coding with Function Pointers

function\_pointers.c

```
1 #include <stdio.h>
2
3 void happy(int score) {
4     printf("%d is great!\n", score);
5 }
6
7 void sad(int score) {
8     printf("%d sucks!\n", score);
9 }
10
11 int main() {
12     // Declare function pointer variable
13     void (*my_function)(int);
14
15     // Change value, just like a variable; no ()
16     my_function = happy;
17
18     for (int i = 0; i < 10; i++) {
19         // Call it
20         my_function(i);
21     }
22
23     return 0;
24 }
```

Looks complex, but  
it's just the prototype with:

- a) .. Variable name in brackets
- b) .. \* before the name

Can also use:  
`my_function = &happy;`

Call the function pointer like it's  
.. just a normal function.



# ABCD: Function Pointers

- Which of the following gets the address of a function?

(a) &foo()  
(b) \*foo()  
(c) &foo  
(d) foo

- Which of the following correctly creates a function pointer named `func` that points to `int foo(char a, int b)`?

(a) `int (*foo)(char a, int b) = func;`  
(b) `int (*func)(char a, int b) = foo;`  
(c) `int *(foo)(char a, int b) = func;`  
(d) `int *(func)(char a, int b) = foo;`

# Coding with Signals

# man 7 signal

- Run: `man 7 signal`

– **Some examples** (scroll down to **Standard signals**)

- Integer symbols
- **SIGINT**: CTRL-C
- **SIGKILL**: kill call
- **SIGSEGV**: Invalid memory reference

|   |                         |
|---|-------------------------|
| <code>printf("SIGINT=%d\n", SIGINT);</code>   | <code>SIGINT=2</code>   |
| <code>printf("SIGKILL=%d\n", SIGKILL);</code> | <code>SIGKILL=9</code>  |
| <code>printf("SIGTERM=%d\n", SIGTERM);</code> | <code>SIGTERM=15</code> |
| <code>printf("SIGSEGV=%d\n", SIGSEGV);</code> | <code>SIGSEGV=11</code> |
| <code>printf("SIGUSR1=%d\n", SIGUSR1);</code> | <code>SIGUSR1=10</code> |
| <code>printf("SIGUSR2=%d\n", SIGUSR2);</code> | <code>SIGUSR2=12</code> |

– **How to send a signal** (scroll up to **Sending a signal**)

- `raise()`: to itself
- `kill()`: to a process

– **Signal handler**

- `man sigaction`
- The important part is filling out **struct sigaction**.

**When using signals, you need to use signal safe functions.**

# Signals and Function Pointers

- To receive a signal we must:
  - write a function to handle a certain signal.
  - register handler with Linux using `sigaction()`:  
pass it a function pointer to our handler.

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

Signal to  
handle,  
such as  
**SIGSEGV**

Struct configuring our handler.

**struct sigaction**

```
.sa_handler = ..  
.sa_flags = ..  
.sa_mask = ..
```

Gives us  
back the  
old signal  
handler.

# Sigaction

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

- **signum**

- The signal number to register the handler

- **act**

- Specify action to perform

- **Recall: Define a struct C:** `struct sigaction act;`

- Contain three major fields:

1. `act.sa_handler = handler_func;`

- A pointer `void (*sa_handler)(int)` to signal handler, receiving the signal number
- `SIG_DFL` (default) or `SIG_IGN` (ignore)

2. `act.sa_flags = 0;`

- A mask to modify the behavior of signal. By default we use `0`

3. `sigemptyset(&act.sa_mask);`

- Specify signals should be blocked during signal handler execution in addition to the triggering signal

- **oldact**

- Nullable

- Return back original handler

# Signal Safety

- When using signals, you need to use signal safe functions in handler

Run: `man 7 signal-safety`

**async-signal-safe function**: can be **safely called** from within a signal handler

- The function should guarantee not to interfere any operation being interrupted

**Example**: all **stdio** library functions are not **async-signal-safe**!

Reason:

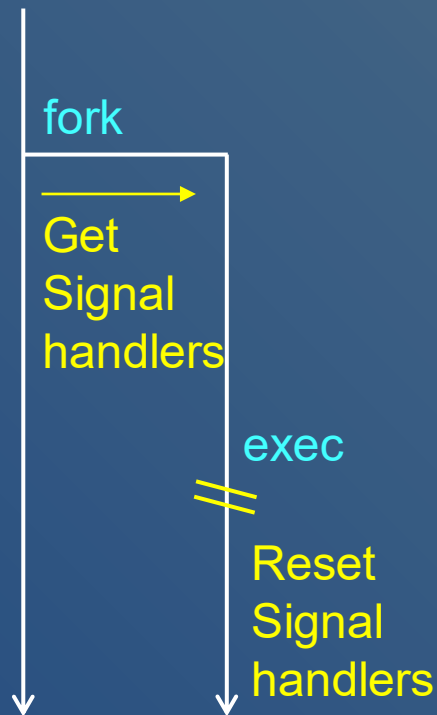
- When performing buffered I/O, need allocated data buffer & pointers
- When main program partially updates the buffer, signal handler that uses it result in wrong buffer status

**Workaround**: use `read()`, `write()` instead

- The file descriptor for stdin/out is **STDIN\_FILENO**, **STDOUT\_FILENO**

# Signal with Fork

- `fork()`: **inherit** signal handler in the new process
- `fork() + exec()`: **not inherit** signal handler (normally)



# Activity: sigaction()

- (10 min) Write a program that:
  - use `sigaction()` to install a `SIGINT` signal handler:  
Print "CTRL-C pressed"
  - infinite loop calling `sleep()`
- Test using CTRL-C to test
  - Use `bt` to send `SIGINT`, and `kill`
- Hints
  - Use `write(STDOUT_FILENO, ....)` instead of `printf()` (not signal safe)
  - `sigaction()`'s struct:
    - Create a struct, then one at a time initialize the fields
    - Set the `.sa_handler` to your function.
    - Set the `.sa_flags` to 0 (don't need any here)
    - Initialize `.sa_mask` to empty; `man sigemptyset()`



# Code

- Note function pointers
- Note struct initialization
  - Pass by ptr

```
sig_handle_sigint.c +
1 #define _POSIX_C_SOURCE 200809
2 #include <signal.h>
3 #include <stdbool.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <unistd.h>
8
9 static char *message = "CTRL-C Pressed\n";
10 void handle_sigint(int signum) {
11     write(STDOUT_FILENO, message, strlen(message));
12     // printf("%s", message); // Don't use; not signal safe.
13 }
14
15 int main() {
16
17     struct sigaction act;
18     act.sa_handler = handle_sigint;
19     act.sa_flags = 0;
20     sigemptyset(&act.sa_mask);
21
22     // Register signal handler
23     int ret = sigaction(SIGINT, &act, NULL);
24     if (ret == -1) {
25         perror("Sigaction() failed");
26         exit(EXIT_FAILURE);
27     }
28
29     while (true) {
30         sleep(1);
31     }
32 }
```

# Activity: kill()

- (5 min) Write a program that creates two processes:
  - parent process should:
    - use `sigaction()` install `SIGINT` signal handler.  
Print "CTRL-C pressed"
    - infinite loop calling `sleep()`
  - child process should:
    - infinite loop that periodically sends `SIGINT` to the parent & sleeps
- Hint
  - `kill()`

# Code

```
42     } else {  
43         // Child to send signals  
44         while (true) {  
45             sleep(2);  
46             printf("HEY Parent!\n");  
47             if (kill(getppid(), SIGINT) == -1) {  
48                 perror("Unable to send signal to parent.");  
49                 exit(EXIT_FAILURE);  
50             }  
51         }  
52     }
```

# ABCD: Signals

- What is **wrong** with this **signal handler** for SIGINT?

```
void do_signal(int signum) {  
    printf("Signal %d\n", signum);  
}
```

- (a) It has the wrong name.
- (b) It has the wrong arguments.
- (c) It has the wrong return type.
- (d) It calls the wrong function.

- What is the **data type** of the *second* argument to **sigaction()**?

- (a) Function pointer to signal handler.
- (b) Pointer to a struct which contains a function pointer.
- (c) The signal number to respond to.
- (d) Pointer to the mask of signals to block while in the signal handler

# Summary

- Signals are notifications with specific meanings.
  - Allow asynchronous communication.
- Configure to receive using `sigaction()`
  - Configuration done with a `struct`
  - Set signal handler with a function pointer
- Send any signal with `kill()`