Signals

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

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

Introduction to Signals

Signals

- *Signals are **notifications with specific meanings***
  - Programs and the kernel can send signals to themselves or other programs
- Wonka Golden Ticket Example
  - Parent process spawns children to search for a golden ticket
  - Parent registers a signal handle
  - Child sends a signal to parent when it discovers a ticket

Pseudocode for Signals

- *Parent*

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

main() {
    pid = fork()

    if (pid != 0) {
        register signal handler
        wait forever
    }
}
```

sigaction(…)

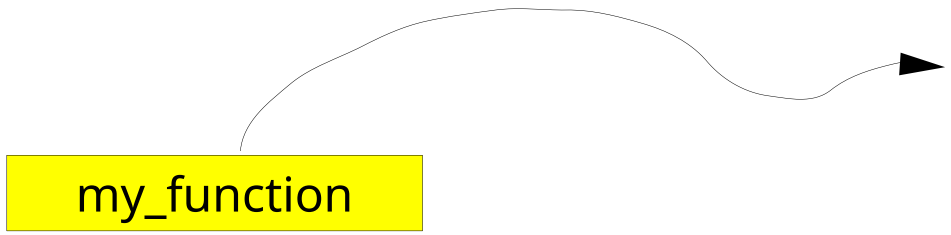
- *Child*

```
...
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 hold the address of a function
  - They allow us to pass around (and call) functions

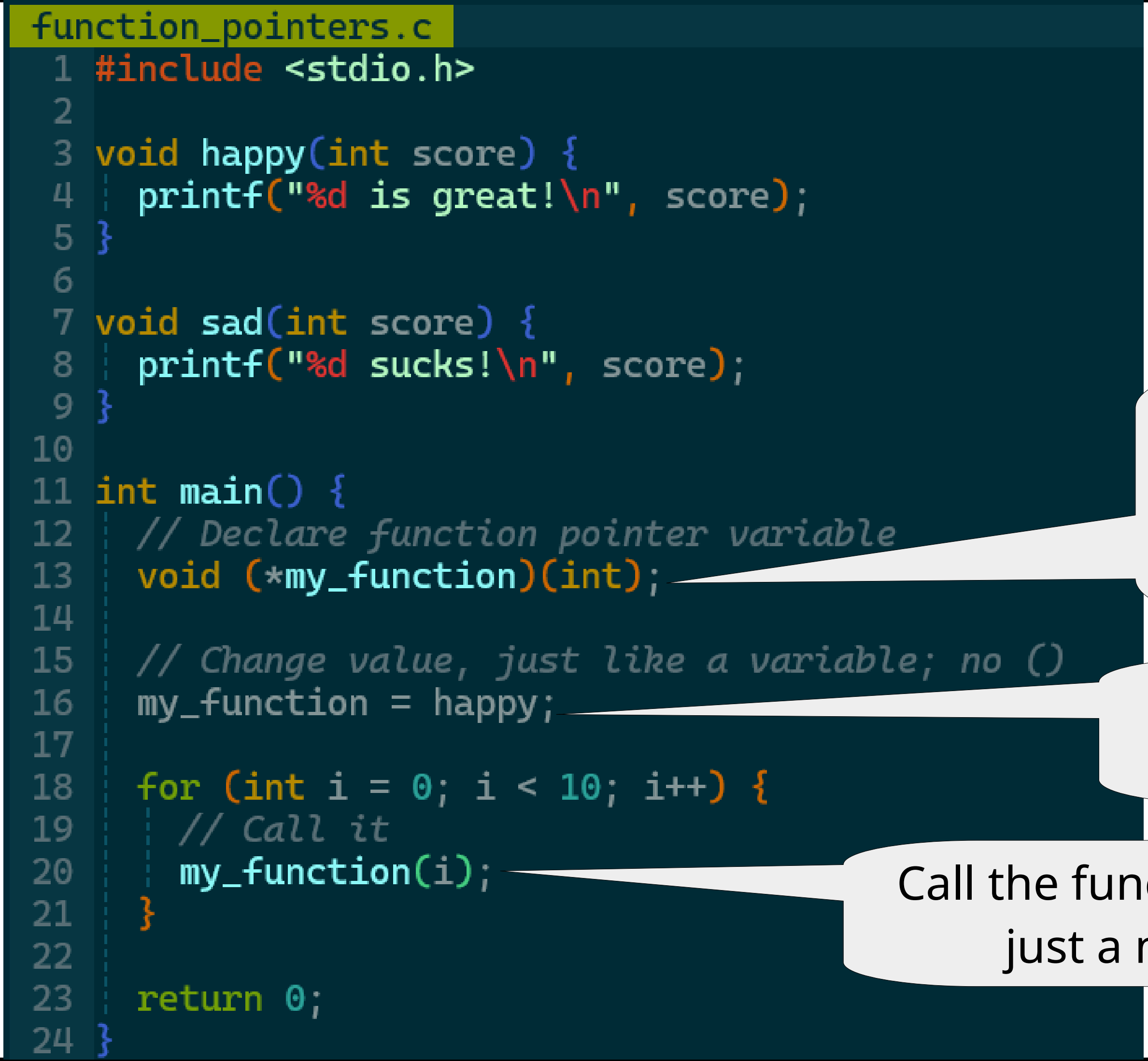


```
handler() {
    print "Found ticket!"
}
```

Why Function Pointers?

- *Imagine an embedded system receiving bluetooth data*
  - How does the bluetooth module / library tell the rest of the system there is data available?
- *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 a function pointer

Coding with Function Pointers



function_pointers.c

```c
#include <stdio.h>

void happy(int score) {
  printf("%d is great!\n", score);
}

void sad(int score) {
  printf("%d sucks!\n", score);
}

int main() {
  // Declare function pointer variable
  void (*my_function)(int);

  // Change value, just like a variable; no ()
  my_function = happy;

  for (int i = 0; i < 10; i++) {
    // Call it
    my_function(i);
  }

  return 0;
}
```

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

Audience Participation - 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

Coding with Signals

- *To receive a signal we must*
  - write a function to handle a certain signal
  - register a handler with Linux using sigaction() by passing 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` = Our handler
`.sa_flags` = Custom flags (0)
`.sa_mask` = Set with `sigemptyset()`

Gives us back the old signal handler.

man 7 signal

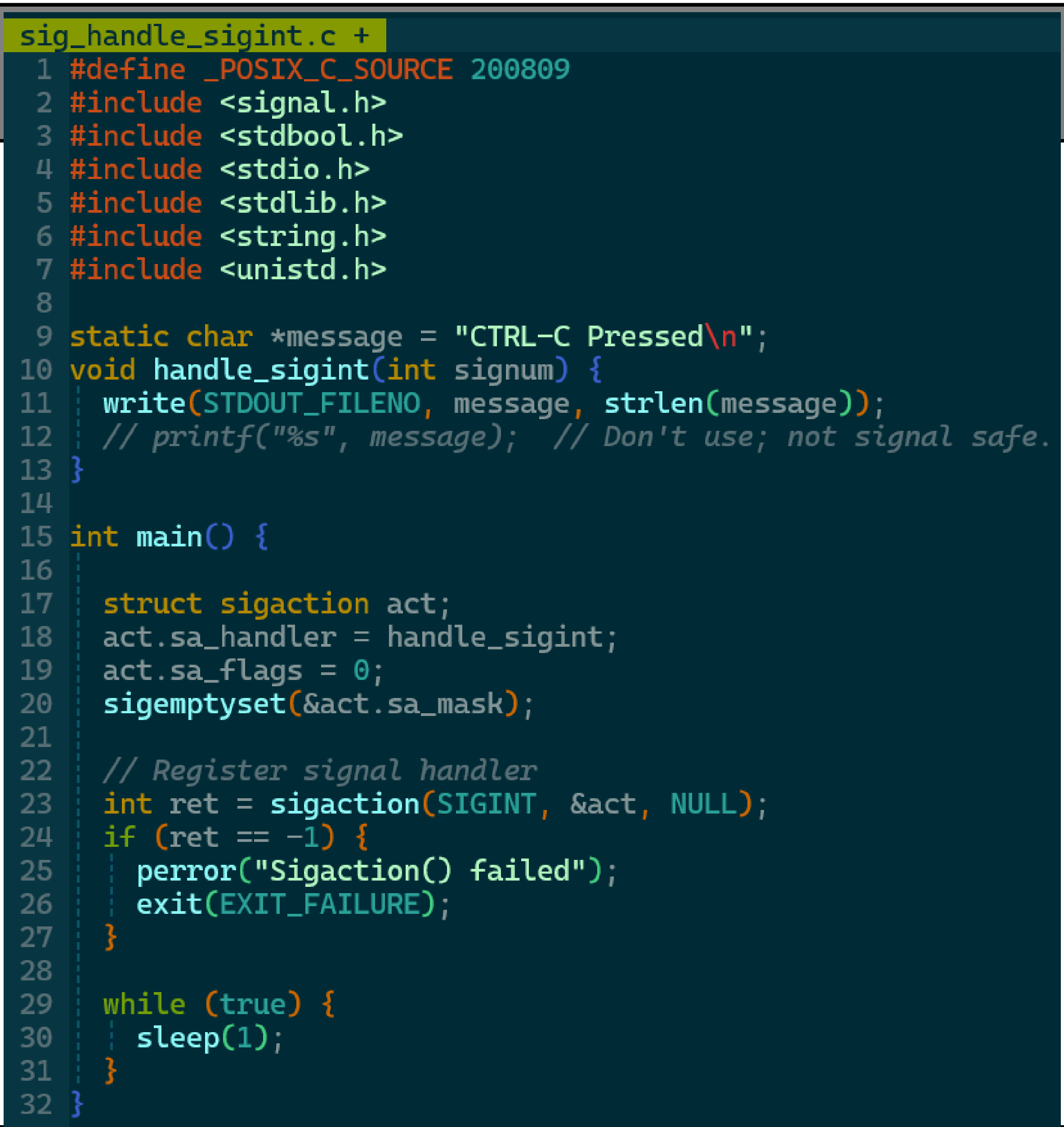
- *Run `man 7 signal`*
  - *Some examples (scroll down to "Standard signals")*
    - **`SIGINT`:** interrupt, CTRL-C
    - **`SIGKILL`:** kill call
    - **`SIGSEGV`:** invalid memory reference
  - *How to send a signal (scroll up to `Sending a signal`)*
    - **`raise()`:** to self
    - **`kill()`:** to another process
  - *Signal handler*
    - **`man sigaction`**
    - The important part is filling out `struct sigaction`
    - Look at feature test macros for sigaction
  - *When handling signals, you need to use **signal safe functions***
    - `man signal-safety` for a list of async-signal-safe functions

Activity - `sigaction()`

- *Write a program that handles `SIGINT` (10m)*
  - Use `sigaction()` to install a `SIGINT` signal handler
  - Handler should print "CTRL-C pressed"
  - Wait (call `sleep()`)
- *Test using CTRL-C*
  - Use btop (or ps, kill) to send SIGINT and kill
- *Hints*
  - Use write(STDOUT_FILENO, ….) instead of printf() (not signal safe)
  - *`sigaction()`'s struct*
    - Declare/allocate a struct, then initialize the fields one by one
    - 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`

Solution Code

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



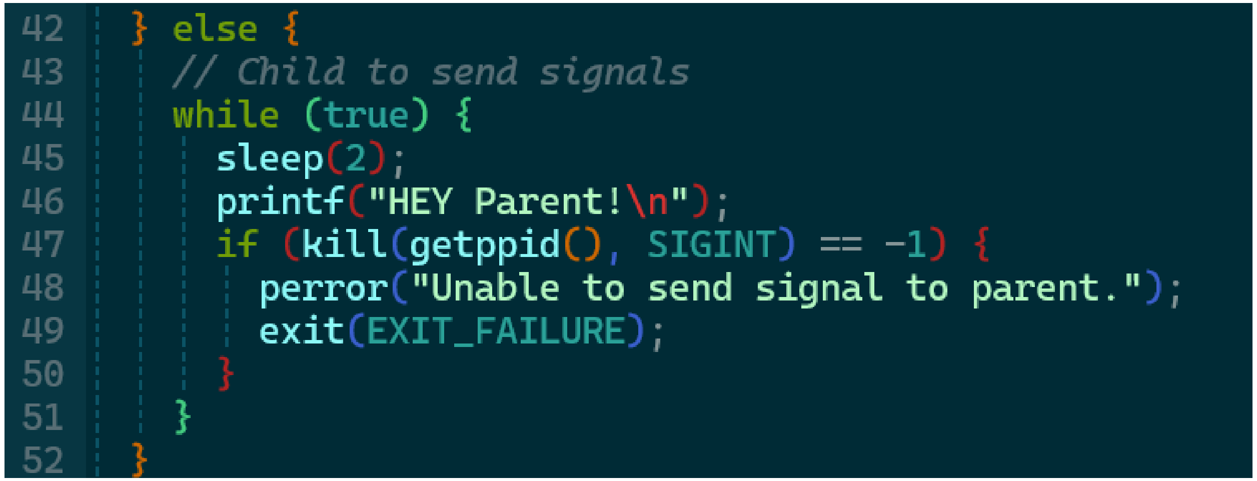
sig_handle_sigint.c +

```c
#define _POSIX_C_SOURCE 200809
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static char *message = "CTRL-C Pressed\n";
void handle_sigint(int signum) {
  write(STDOUT_FILENO, message, strlen(message));
  // printf("%s", message);  // Don't use; not signal safe.
}

int main() {

  struct sigaction act;
  act.sa_handler = handle_sigint;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);

  // Register signal handler
  int ret = sigaction(SIGINT, &act, NULL);
  if (ret == -1) {
    perror("Sigaction() failed");
    exit(EXIT_FAILURE);
  }

  while (true) {
    sleep(1);
  }
}
```

Activity - `kill()`

- *Write a program that creates two processes (5m)*
  - *Parent process*
    - Use `sigaction()` install `SIGINT` signal handler
    - Handler should print "CTRL-C pressed"
    - Wait (call `sleep()`)
  - *Child process*
    - Periodically send `SIGINT` to the parent in a loop
    - Wait between signals (call `sleep()`)
- *Hint*
  - Use `kill()`
  - Remember `fork()`?

Solution Code



```c
} else {
  // Child to send signals
  while (true) {
    sleep(2);
    printf("HEY Parent!\n");
    if (kill(getppid(), SIGINT) == -1) {
      perror("Unable to send signal to parent.");
      exit(EXIT_FAILURE);
    }
  }
}
```

Audience Participation - 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()`*