CSc 352
System Calls and Function Pointers

Benjamin Dicken

Announcements

- Extensions
- PAs 8 and 9
- Next week we should be back to in-person

System Calls

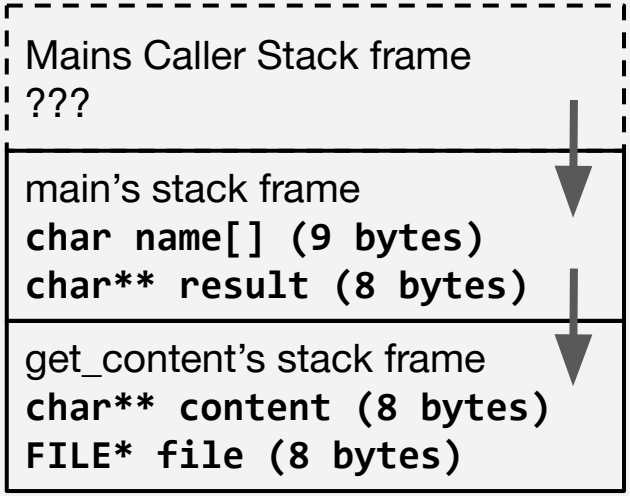
- There are some tasks that a “regular” user process does not have the privileges to do
  - Communicate with the hard drive / ssd
  - Communicating with the network hardware
  - Create a new process
- User programs can use **system calls** to request these things

The Stack

```c
#include <stdio.h>

char** get_content(char* file_name) {
  char** content;
  FILE* file = fopen(file_name, "r");
  // Load file contents into content variable
  return content;
}

int main() {
  char name[] = "file.txt";
  char** result = get_content(name);
  // other stuff
  return 0;
}
```

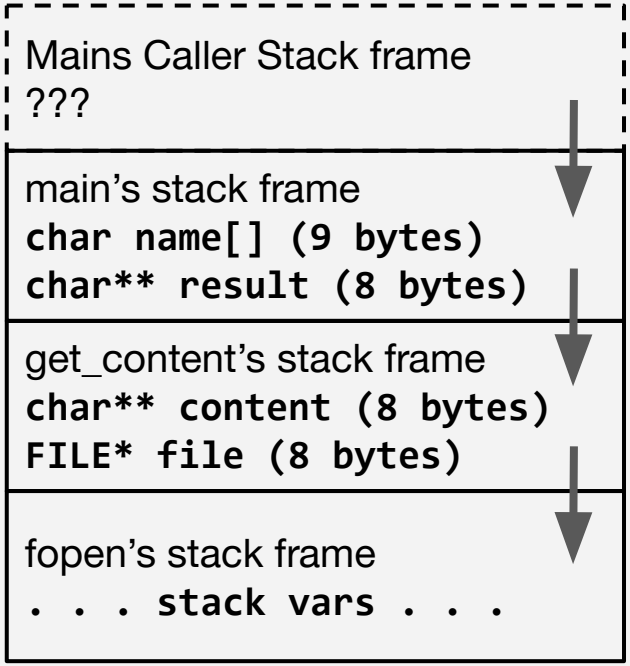


The Stack

```c
#include <stdio.h>

char** get_content(char* file_name) {
  char** content;
  FILE* file = fopen(file_name, "r");
  // Load file contents into content variable
  return content;
}

int main() {
  char name[] = "file.txt";
  char** result = get_content(name);
  // other stuff
  return 0;
}
```

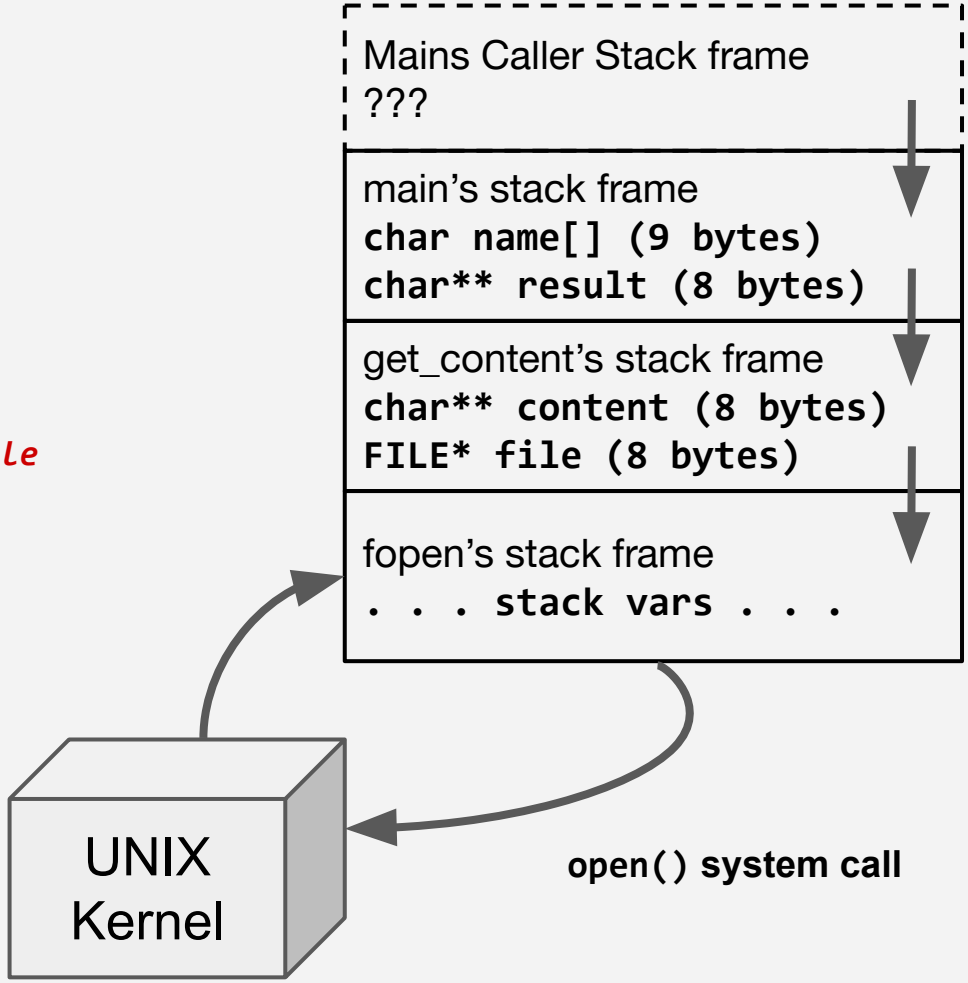


The Stack

```c
#include <stdio.h>

char** get_content(char* file_name) {
  char** content;
  FILE* file = fopen(file_name, "r");
  // Load file contents into content variable
  return content;
}

int main() {
  char name[] = "file.txt";
  char** result = get_content(name);
  // other stuff
  return 0;
}
```



System Calls

Each process (and even each thread) gets a:

- User stack - for executing the user code
- Kernel stack - for executing things at the kernel level, such as a system call

System Calls

https://www.cs.miami.edu/home/geoff/Courses/CSC521-04F/Content/UNIXProgramming/UNIXSystemCalls.shtml

Many calls available

Generally, call these via library calls as intermediary / abstraction rather than directly

Library Functions Abstracting Sys Calls

- `fopen` is a standard library function, may invoke the `open` sys call
- `fscanf` is a standard library function, may invoke the `read` sys call
- `fclose` is a standard library function, may invoke the `close` sys call
- `printf` is a standard library function, may invoke the `write` sys call
- . . . .

Strace

the **`strace`** command can be used to see what system calls a process invokes

`$ strace ./a.out 2> out`

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

int main() {
  char buffer[256];
  FILE* test_file = fopen("test.txt", "r");
  while (fgets(buffer, 255, test_file) != NULL) {
    printf("LINE: %s\n", buffer);
  }
  fclose(test_file);
  return 0;
}
```

Activity

Strace

Go to your home directory

Use **`strace`** and determine what system call is used the most if you run ls

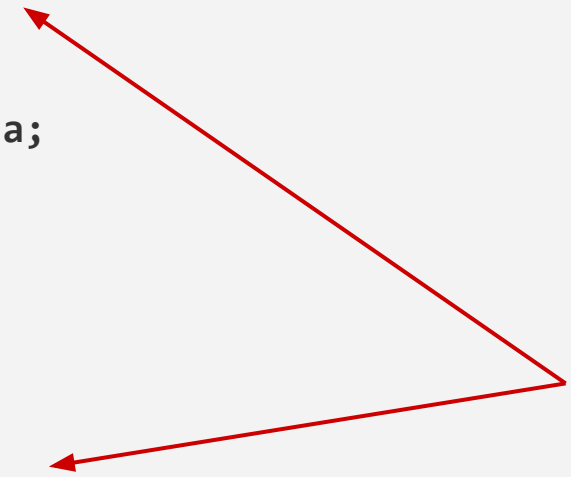
Function Pointers

- Can call functions by their pointer (address), rather than by name
- Can store pointers to functions in variables!

Two Functions

```
int summation(int a) {
  int result = 0;
  while(a > 0) {
    result = result + a;
    a--;
  }
  return result;
}

int factorial(int a) {
  int result = 1;
  while(a > 0) {
    result = result * a;
    a--;
  }
  return result;
}
```



Same return value, same number of parameters and parameter types

Function Pointers

```c
int main(int argc, char** argv) {
  int result = 0;
  int (*action)(int);
  int value = 5;

  action = factorial;
  if (argc >= 2 && argv[1][0] == 's') {
    action = summation;
  }

  result = (*action)(value);
  printf("%d\n", result);

  return 0;
}
```

This is a function pointer variable

Assigning a function pointer variable

Call function via pointer

Function Pointers

- Why are function pointers useful?
- Think of particular examples where it could come in handy

Object-Oriented

- In OO languages such as Python and Java, we can
  - Create classes, that define both **variables** and **functions**
  - Instantiates instances of classes (objects)
  - Call functions via an object with dot syntax

```
Car x = new Car();
x.honkHorn();
```

  - Can avoid directly accessing the variables, use getters and setters, etc

Object-Oriented

```
public class Car {
  private String color;
  private String horn_sound;
  private int horse_power;
  private double latitude;
  private double longitude;

  public Car() { . . . }

  public void honkHorn() { . . . }
  public void showColor() { . . . }
  public void move (double lat, double  lng) { . . . }
}
```

```
public static void main(String[] args) {
  Car x = new Car();
  x.honkHorn();
}
```

Object-Oriented

Can we do something similar in C?

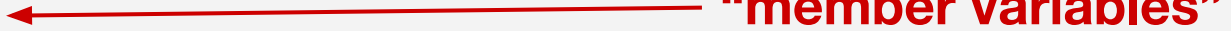
Object-Oriented

Can we do something similar in C?

Yes (but it’s a bit uglier :)

Define a Struct with Functions

```
typedef struct Car {
  char* color;
  char* horn_sound;
  int horse_power;
  double latitude;
  double longitude;
  void (*honk_horn)(struct Car* this);
  void (*show_color)(struct Car* this);
  void (*move)(struct Car* this, double lat, double  lng);
} Car;
```



```c
void honk_horn(struct Car* this) {
  if (this->horn_sound == NULL) {
    printf("honk!\n");
  } else {
    printf("%s\n", this->horn_sound);
  }
}

void show_color(struct Car* this) {
  if (this->color == NULL) {
    printf("red\n");
  } else {
    printf("%s\n", this->color);
  }
}

void move(struct Car* this, double lat, double lng) {
  this->latitude  = lat;
  this->longitude = lng;
}
```

Define the functions

```
#define NEW_CAR(hp, lat, lng) \
 (Car) {          \
  NULL, NULL,     \
  hp, lat, lng,   \
  honk_horn,      \
  show_color,     \
  move            \
 };
```

Define a “constructor” with CPP

Use it!

```
int main(int argc, char** argv) {

  Car plain = NEW_CAR(200, 0, 0);

  plain.show_color(&plain);
  plain.honk_horn(&plain);
  plain.move(&plain, 1.0, 2.0);

  return 0;
}
```