Week 06 2

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

October 19, 2023

Announcements

- midterm room
  - still waiting
  - I'll announce something Monday
- midterm is next class!
- A2 is over
  - A3 is up
    - pt.1 you can already do!
    - pt.2 you'll need to learn

System calls

- why are they different
  - than functions?
  - than other programs?
- execution stops
  - at least for *your* program
  - and you wait for results
    - from the OS!

How can we deal with errors?

- there are two main ways
- for system calls to report an error
- some system calls return an `int`
  - and the value `-1` indicates error
- some system calls return pointers
  - so `NULL` indicates an error

These are limited

- there is not much info
- regarding *what* error occurred
- so there is a global `int`
  - the variable `errno`
    - that the OS will also change
    - in addition to return value

errno

- a variety of error codes
- simply represent many
  - possible system errors
- rather than interpret these
  - the numbers returned
- the system provides
  - `perror()`

`perror(char *msg)`

- a function for error printing
  - a consistent error report
  - for system calls
- the `msg` is *pre-pended*
  - so the error message will
  - start with this
  - and then be a standardized
    - message for the `errno`

Good practice

- try to check all of your
  - system calls
- AI code completion
  - makes this profoundly easy

WORKSHEET

`errors.pdf`

Function Pointers

- remember that code
  - is in memory!
- if we had pointers to values
  - why not pointers to code?

Recall arrays

- suppose we said
  - `int arr[5]`
- then, `arr` refers to:
  - the starting memory addr
    - of the whole array

functions are the same!

```
void my_func(int *arg1, int arg2){
    *arg1 = arg2;
}

int main(){
    // stuff
    void (*func_variable)(int *, int) = my_func;
    // stuff
}
```

The notation is clunky

- since C is so reliant
  - on knowing which types
  - are relevant to code
    - (because it needs to allocate space!)
- but now, you could say

```
int x, y;
func_variable(&x, y);
```

Why would you ever use this?

- Consider how useful
  - this is *within* a `struct`

```
tyepdef struct almost_object{
    int value;
    char *name;
    void (*__init__)(almost_object self,
                     int value, char* name);
} AlmostObject;
```