Today

- Directory system calls
- Dynamic Memory Management
Directory related system calls

- `mkdir()` – creates a directory
- `readdir()` – reads a directory
- `opendir()` – opens a directory
- `getcwd()` – returns the current working directory
mkdir

- `int mkdir ( char *pathname, mode_t mode)`
- `'pathname'` specifies name of new directory (can be absolute or relative)
- `'mode'` specifies permissions of new directory
- Returns 0 on success, -1 on failure
readdir

- struct dirent *readdir (DIR *dirp)
- Returns a pointer to the next directory entry in the directory stream dirp
- struct dirent {
  ino_t d_ino; /* inode number */
  char d_name[256]; /* filename */
  /*other fields (see man page)*/
}
- Returns NULL on reaching end of directory stream or if an error occurred
opendir

- `DIR *opendir (char *pathname)`
- `'pathname' specifies name of the directory (can be absolute or relative)`
- Returns a pointer to the directory stream positioned at the first entry in the directory
- Returns NULL on error
getcwd

• char *getcwd (char *buf, size_t size)
• Returns null-terminated string containing absolute path of current working directory in buf which is of length size
• Returns NULL on error
Programs we will discuss

Run each of these programs:
- mkdir.c
- getcwd.c
- is_dir.c

Problems to work on offline:
- Walk through directory and identify files and sub-directories
- Make a copy of an input directory
Dynamic Memory Management

• Local variables within a function are allocated on the stack, and deleted when the function exits.

• If you want a variable to persist after the function exits, need `malloc` to dynamically allocate memory.
Dynamic Memory Management

• Static allocation:
  
  ```c
  int array[10];
  ```

• Dynamic allocation:
  
  ```c
  int *array = (int *) malloc(10*sizeof(int));
  ```

• Since both are pointers, both can be accessed with `array[0] = 42;`
Common Memory Problems

• Memory Leaks
• Buffer Overflow
• Segmentation Fault / Illegal Access
Memory Leaks

“free() the malloc()s!”

• If you call malloc, the block it returns is NEVER automatically freed up.
• You MUST call free on the memory address returned by malloc to free up the block.
• You MUST keep a pointer to it so that you can call free later. If you don’t, the block is “lost” somewhere in memory. Too many of these and you’ll run out of memory!
Memory Leaks

- `free` must be called EXACTLY once. Repeat calls will result in unexpected behavior.

```c
char *first = (char*)malloc(10*sizeof(char));
char *second = first;
//Notice that both pointers point to the same memory block.
free(second); //This is good.
free(first); //This is redundant - crash!
```
Buffer Overflows

“Taking a long walk off a short array”

• Common causes:
  – Array index miscalculation or “off-by-one” error
  – Trying to store a string into a memory block too small to fit it into

• Can lead to segmentation faults or security holes.
Segmentation Fault

• Occurs when a process tries to access a memory address illegally.
• Process gets signal **SIGSEGV**
• Usually causes a crash.
• Common causes:
  – Trying to access a **NULL** pointer
  – Buffer overflows into an invalid memory space
  – Trying to access an uninitialized pointer
Exercise

• Find bugs in `buggy1.c` and `buggy2.c`

• Make sure you don't write such terrible code!
Questions?