Section 3
Project 0 reflection, More C lessons, Project 1 pitfalls

Project 0

Reflections?

Coding Style

- Having neat code makes a world of difference
  - If I can’t read your code and understand what its doing – you will lose points! (Especially if there are bugs)
- Properly indent nested blocks
- Always comment functions declared in .h files with their “contract”
  - What does it do?
  - What does it return?
  - What assumptions does it make about its arguments?
  - How does it indicate an error condition?

Coding Style

- Write comments for tricky implementation sections:
  - Bad Comment:

```
somePtr = NULL;   // Set somePtr to NULL
```

What a useless comment!

  - Good Comment:

```
somePtr = NULL;   // Always reset the pointer to
    // NULL after the shared memory
    // it points to has been freed
```

Coding Style

- Always use header guards. Why?

```
#ifndef FULL_PATH_TO_FILE_H_
#define FULL_PATH_TO_FILE_H_

// all your header file code here

#endif  /* FULL_PATH_TO_FILE_H_ */
```

Coding Style

- Be consistent with your naming.
- For functions
  - `set_hash_function()` style is ok and most common in C
  - `SetHashFunction()` style also ok, just pick one and stick to it!
  - End typenames in _t, eg

    ```
    typedef foo_struct * foo_t;
    ```
  - Don't abbreviate ambiguous variable names

    ```
    int n_comp_conns;  // BAD - what is this?
    int num_completed_connections;  // OK
    ```

What’s Wrong Here?

```
void add(key_t k, value_t v) {
   …
   ht_node_t node = (ht_node_t)malloc(sizeof(ht_node));
   node->k = k;
   node->v = v;
   …
}

ht_node_t lookup(key_t k) {
  …
  return node;
}
```

Memory Management

- Always be explicit about who owns memory.
- Consider:

```
void do_stuff (char * buff, int len) {
     …
     free(buff);
}

int main() {
     char * mybuff = (char*)malloc(SZ * sizeof(char));
     do_stuff(mybuff, SZ);  // This frees mybuff
     …
     free(mybuff);  // Double free - Undefined behavior!
}
```

Memory Management

- Consider one of two solutions:

```
// do stuff assumes ownership of buffer buff and
// deallocates memory allocated for buff.
void do_stuff(char * buff, int len);
```

- Or

```
// Caller owns the memory pointed to by buff.
void do_stuff(char * buff,  int len) {

    // do not free(buff) here!

}
```

- Either way – memory ownership is explicit.

Your hash table?

```
int main() {
  …
  key_t k = (key_t)malloc(...)
  value_t v = (value_t)malloc(...)
  ht.add(k,v);
  v = NULL;
  ht.remove(k);
  // What's happened to k or v?
}
```

Two best solutions

- Two solutions:
  - Either...
    - Client releases ownership of allocated memory
    - Hash table must free elements
    - If client tries to free added elements, she causes a crash
  - ...or
    - Client maintains ownership of allocated memory
    - Hash table not resposnible from freeing elements
    - If client does not maintain pointers, she causes a memory leak
- Which is better?

Memory Management

- When to free memory?
- What if two different places are holding on to a reference?
  - Reference counting. Drawbacks?
  - This is why platforms like Java and .NET have Garbage Collection.

Project 1 Notes

- Project 1
  - Due Monday, April 25th At 11:59pm!
- You should have started already! If not, do it now!
- New starting procedure
  - And start-up troubles can stall you for days
    - (trust me)

Project 1 Notes

- For changed Linux source files:
  - Give full path names in your modified files write-up
    - USE “./arch/i386/kernel/process.c”
    - NOT “process.c” – there are many of these
  - Maintain directories when submitting changed files:
    - When I extract your changed files, they should go to the right directory, so it is unambiguous which file you changed
    - This is easy to do with tar

Watch out for...

- What architecture the code you're reading is for:
  - You'll want x86
  - And 32-bit!
- You're working on the latest stable...
  - ...but a lot of online resources are for older versions! ...even the previous slide...
- Your environment
  - VMWare is supported by us. Virtualbox is possible but you'll have to solve some problems yourself.

Linux directory structure

- mm → memory management
- ipc → interprocess communication
- fs → files system
- include → user exposed headers
- kernel → core OS
- arch → architecture specific code
  - Much of the lower level implementation is here