CS 241
Section Week #5

9/22/11

Topics This Section

- File I/O
- Advanced C

POP QUIZ!

POP QUIZ! (not really)

- What is type definition for this function:
  - int *foo(const int* input, char *msg)
- T/F When a thread is finished it should call *exit.*
- What is the one exception to grading where you can leak memory? (Case and amount)
- T/F Robin was wearing green in lecture yesterday.

POP QUIZ! (not really)

- What is type definition for this function:
  - int *foo(const int* input, char *msg)

  int *(*)(const int*, char*)

POP QUIZ! (not really)

- What is type definition for this function:
  - int *foo(const int* input, char *msg)
- T/F When a thread is finished it should call *exit.*

False. It should call pthread_exit or simply return.

POP QUIZ! (not really)

- What is type definition for this function:
  - int *foo(const int* input, char *msg)
- T/F When a thread is finished it should call *exit.*
- What is the one exception to grading where you can leak memory? (Case and amount)

If you call pthread_exit(); 5 blocks.

POP QUIZ! (not really)

- What is type definition for this function:
  - int *foo(const int* input, char *msg)
- T/F When a thread is finished it should call *exit.*
- What is the one exception to grading where you can leak memory? (Case and amount)
- T/F Robin was wearing green in lecture yesterday.

If you missed this you haven't been keeping up on lectures

File Input/Output

File I/O in C

MP2 requires you to read and write text files in C.

Two primary means of doing I/O in C:

Through lightly-wrapped system calls

`open()`, `close()`, `read()`, `write()`, etc

Through C-language standards

`fopen()`, `fclose()`, `fread()`, `fwrite()`, etc

File I/O in C

Opening a file (Method #1):

```
fopen(const char *filename, const char *mode);
```

filename: path to file to open

mode: what do you wish to do with the file?

- **`r`**: read only
- **`r+`**: read and write (file must already exist)

- **`w`**: write (or overwrite) a file
- **`w+`**: write (or overwrite) a file and allow for reading

- **`a`**: append to the end of the file (works for new files, too)
- **`a+`**: appends to end of file and allows for reading anywhere in the file; however, writing will always occur as an append

File I/O in C

Opening a file (Method #2):

```
open(const char *filename, int flags, int mode);
```

filename: path to file to open

flags: what do you wish to do with the file?

One of the following is required:

O_RDONLY, **O_WRONLY**, **O_RDWR**

And any number of these flags (yo "add" these flags, simply binary-OR them together):

O_APPEND: Similar to "a+" in fopen()

O_CREAT: Allows creation of a file if it doesn't exist

O_SYNC: Allows for synchronous I/O (thread-safeness)

mode: what permissions should the new file have?

(**S_IRUSR** | **S_IWUSR**) creates a user read-write file.

Opening Files in C

Return value of opening a file:

Having called **`open()`** or **`fopen()`**, they will both create an entry in the OS's file descriptor table.

Specifics of a file descriptor table will be covered in-depth in the second-half of CS 241.

Both **`open()`** and **`fopen()`** returns information about its file descriptor:

`open()`: Returns an int.

`fopen()`: Returns a **`(FILE *)`**.

Reading Files in C

Two ways to read files in C:

```
fread(void *ptr, size_t size, size_t count, FILE *s);
```

`*ptr`: Where should the data be read into?

`size`: What is the size of each piece of data?

`count`: How many pieces?

`*s`: What (FILE *) do we read from?

```
read(int fd, void *buf, size_t count);
```

`fd:` What file do we read from?

`*buf:` Where should the data be read into?

`count:` How many bytes should be read?

Reading Files in C

Reading more advancely…

`fscanf(FILE *stream, const char *format, …);`

Allows for reading at a semantic-level (eg: ints, doubles, etc) rather than a byte-level.The format string (**`*format`**) is of the same format as **`printf()`**.

`fgets(char *s, int size, FILE *stream);`

reads in at most **`size -1`** characters from stream and stores them into the buffer pointed to by s. Reading stops after an **`EOF`** or a newline. If a newline is read, it is stored into the buffer. A **`'\0'`** is stored after the last character in the buffer.

Writing Files in C

Writing is a lot like reading…

```
fwrite(void *ptr, size_t size, size_t count, FILE *s);
```

Writing of bytes with (**`FILE *`**).

```
write(int fd, void *buf, size_t count);
```

Writing of bytes with a file descriptor (**`int`**)

```
fprintf(FILE *stream, const char *format, …);
```

Formatted writing to files (works like **`printf()`**)

Closing Files in C

Always close your files!

```
fclose(FILE *stream);
close(int fd);
```

`write()`, and especially **`fwrite()/fprintf()`**, may be buffered before being written out to disk.

If a file is never closed after writing:

- the new data may never be written on the actual file;
- the files may be corrupted.

Advanced C

Playing with Structs

How do we reduce the size of the struct?

```
typedef struct _name_t{
    int size;
    int bool;
} name_t;
```

Playing with Structs

How do we initialize the struct in one line?

```
typedef struct _name_t{
   int size:31;  //31 bits
   int bool:1;  //1 bit
} name_t;
```

Playing with Structs

How do we initialize only bool?

```
typedef struct _name_t{
   int size:31;  //31 bits
   int bool:1;  //1 bit
} name_t;
name_t var = {0, 1}; //size = 0, bool = 1
```

Playing with Structs

How do we initialize only bool?

```
typedef struct _name_t{
   int size:31;  //31 bits
   int bool:1;  //1 bit
} name_t;
name_t var = {.bool=1};
```