
CS11 C++ DGC

Spring 2006-2007

Lecture 7

This Week's Lab

- Build a basic instant messaging program using Spread
 - Console-based user interface, using ncurses
 - User can type chat messages or commands
 - Also need to handle incoming messages from Spread
 - Suggests using multithreaded programming
 - Problem: ncurses is not thread-safe!
 - Use a different approach: asynchronous IO operations from a single thread
-

The ncurses API

- A programming library to support text-based user interfaces on a console
 - Original version was named “curses”
 - ncurses = “New Curses”
 - Used by a wide variety of command-line programs
 - e.g. vim, ncftp, nano, lynx, tin
 - Provides simple “windowing” abstraction
 - Create non-overlapping windows in the console
 - Position the cursor, read/write text on console
 - To build programs with ncurses:
 - Link with `-lncurses`
 - API/libraries are available on virtually all Linux systems
-

ncurses Chat Console

- Simple chat-console class is provided:



```
~/projects/nchat
Wow this chat program is cool!
Users:
-----
Input: Yes I know
```

- Very simple to use:
 - Constructor initializes UI, destructor cleans it up
 - Functions to write a chat message, get input, etc.

Asynchronous IO

- Two sources of input for the chat program:
 - Console input from the user
 - Incoming messages from `SP_receive()`
 - `SP_receive()` is a blocking call
 - `ncurses` is not thread-safe
 - Only one thread should interact with `ncurses` at a time
 - Safest/easiest for `ncurses` interactions to originate from only one thread
 - Solution: use asynchronous IO in program
 - IO operations that return even when no input is immediately available
 - `Spread` and `ncurses` both provide ways of doing this
-

Asynchronous IO with ncurses

- **ncurses:**
 - Can configure “read character” functions to return after a timeout
 - `bool ChatConsole::getUserInput(string &result);`
 - If a complete message is available: message is stored in **result**, and **true** is returned
 - If a complete message isn't yet available, **result** is left unchanged, and function returns **false**
 - Times out after 0.2 seconds
 - Returns in a reasonable amount of time, even when no input is available
 - Periodically call the user-input function
 - When a whole message is finally available, perform appropriate processing
-

Asynchronous IO with Spread

- **`SP_receive()`** is a blocking call
 - Won't return until a new message is available (or if an error occurs, of course)
 - Spread provides an **`SP_poll()`** function
 - **`int SP_poll(mailbox mbox)`**
 - Returns number of bytes available for reading
 - If result > 0, **`SP_receive()`** won't block
 - Write a loop to process Spread messages:
 - As long as **`SP_poll()`** returns positive value, call **`SP_receive()`** and handle incoming message
-

Message Handling Loop

- Construct a message handling loop:
 - Do:
 - Check for a complete message from console input
 - If a message is available, perform appropriate processing
 - May be a “quit” command, for example
 - While there is Spread data to receive:
 - Receive a Spread message and handle it
 - Until user decides to exit chat program.
 - Don't want to peg the CPU!
 - Best if some operation in the loop involves a timeout
 - Console input involves a timeout, so we are in good shape
-

Chat Messages

- This week's chat program just sends and receives chat messages
 - Similar to `sprecho`, but a single program can send and receive messages
 - When a user enters a message:
 - Send it to the Spread chat group
 - (If you use `SELF_DISCARD`, also write message to chat window.)
 - When a message arrives from Spread:
 - Pull out the message text and add it to the chat window
-

Messages and C++ Strings

- C++ `string` class is very useful for simple text messages
 - Dynamically allocated, resizable string
 - Provides many features and benefits over `char*` strings
 - Generally painless to use in very complex ways
 - Prefer `string` to `char*`, wherever possible!
 - `#include <string>`
 - Can send a single string for message body in chat messages
 - How to convert between an array of characters and a C++ `string` object?
-

Messages and C++ Strings (2)

- `string` provides `char*` conversion functions
 - `c_str()` and `data()` return a pointer to a `char*` version of the string
 - Memory for `char*` version is managed by `string` object
 - Don't cache the pointer. Don't try to free the memory yourself.
 - `c_str()` returns a zero-terminated `char*`
 - `data()` returns a non-terminated `char*`
 - `copy(...)` copies the string's contents into an existing `char*` buffer
 - You provide the buffer (you deallocate it too, if necessary)
-

Using Strings to Send Messages

- Can use `c_str()` or `data()` for sending Spread messages
 - With `data()`, no trailing `\0` on string

```
string msg = ...;
```

```
rc = SP_multicast(mbox, svcType, ...  
    msg.length(), msg.data());
```
 - With `c_str()`, don't forget trailing `\0` on string

```
rc = SP_multicast(mbox, svcType, ...  
    msg.length() + 1, msg.c_str());
```
- If C string functions are used to process incoming Spread messages, use `c_str()` and not `data()`
 - e.g. `strcmp()`, `strcpy()`, `strdup()`, `strchr()`

Using Strings to Receive Messages

- Message data is received into a `char` buffer

```
char msgBuf[1024];  
...  
rc = SP_receive(mbox, &svcType, ...,  
               sizeof(msgBuf), msgBuf);
```

- How to convert `char` buffer data back into `string` object?

- Can use `string` constructor, or `assign()` function

```
string::string(const char *str, size_type length);  
string::assign(const char *str, size_type num);
```

- Example:

```
...  
// rc contains number of bytes received  
string msgText(msgBuf, rc);
```

String-Streams

- C++ also provides string-backed streams
 - `#include <sstream>`
 - `stringstream` – general input/output string-stream
 - `istringstream` – for input only
 - `ostringstream` – for output only
- Very useful for formatting or parsing messages
- Example:

```
ostringstream oss;  
oss << "Received " << numBytes << " bytes.";  
  
// Retrieve a copy of stream's contents  
string msg = oss.str();  
console.addChatMessage(msg);
```

Lab 7

- Console UI code is provided
 - `ChatConsole` class manages ncurses
 - Constructor sets up ncurses
 - Destructor cleans it up
 - Error-handling code isn't great, but whatever. 😊
 - `nchat.cc` contains example code
 - How to use the `ChatConsole` class
 - An example main-loop to poll the console for input
 - First step: write a Makefile and get it building
-

Lab 7 (2)

- Once you have chat console code working:
 - Add `Spread` code to connect/disconnect
 - Add “send chat message” and “receive chat message” operations to main loop
 - Use `SP_poll()` to keep `SP_receive()` from blocking user input
 - Feel free to reuse code from `sprecho`
-