6.824 - Fall 2002

Getting started with 6.824 labs

A gentle crash course

Introduction

The 6.824 labs require you to program C/C++ on FreeBSD machines. This document should help you on your way with logging into the machines, reading manual pages, compiling C/C++ programs, and understanding `Makefile`-s that we will give you.

Logging in with ssh

Students enrolled in this class log into class machines using ssh, with a username and password. Other Unix versions are available on http://www.openssh.com/. Windows versions come with Cygwin. If you just want ssh for Windows without the fancy stuff, you can also just install PuTTY.

Compiling and linking simple programs

A compiler takes one or more source files and produces object files that contain machine code. Object files contain machine code and usually have a `.o` extension. C source files usually have a `.c` extension and C++ files usually have a `.C` or `.cc` extension. Object files may have undefined symbols (names of functions or global variables); linking object files into an executable resolves these undefined symbols. Note that one of the object files must have a `main()` function to create an executable.

Object files can be linked together to form an executable, provided that one (and only one) of the object files defines a `main()` function.

An executable is similar to an object file in that it contains machine code, but an executable contains some additional bits that allow the operating system to start the program and run it.

Compiling a source file into the equivalent object file is done using `c++ -c`. For example:

```
% c++ -c hello.C -o hello.o
```

The `-c` flag indicates pure compilation without linking. Even if `hello.C` defines `main()`, the resulting `hello.o` is not executable. To compile a source file into an executable, omit the `-c` flag:

```
% c++ hello.C -o hello
```

This creates an executable `hello`. Note that `c++` will bark at you if you try to create an executable without a definition of `main()`.

Note that all these examples assume you are compiling C++ files. Use `gcc` in stead of `c++` if you are compiling plain old C files.

Multiple object files can be linked together into one executable. Suppose we have the following files:

common.h

```
#ifndef __COMMON_H
#define __COMMON_H
void hello();
void bye();
#endif // __COMMON_H
```

hello.C

```
#include <stdio.h>
#include "common.h"
void hello() {
  printf("hello!\n");
}
```

bye.C

```
#include <stdio.h>
#include "common.h"
void bye() {
  printf("bye!\n");
}
```

main.C

```
#include "common.h"
int main() {
  hello();
  bye();
}
```

To turn these files into a working executable, do:

```
% c++ main.C hello.C bye.C -o hellobye
```

This compiles `main.C`, `bye.C`, and `hello.C` and links them together into an executable called `hellobye`. The header file `common.h` declares---but does not define---`hello()` and `bye()` so that the compiler can compile `main.C`, without knowing the details of their implementation.

Linking is only possible when no two object files try to define C or C++ functions or global variables with the same name. For example, try declaring a variable `int foo` in both `hello.C` and `bye.C`. Compiling each file separately is no problem, but linking fails with a `multiple definition of `foo'` error. The same is true for functions. Depending on what you want, you can avoid this problem in one of a number of ways:

- If you want `hello.C` and `bye.C` to **share** variable `foo`, you can declare one of them as `extern int foo`.
- If you want `hello.C` and `bye.C` to each have their own independent variable named `foo`, then declare one or both of them as `static int foo`. Keyword `static` limits the scope of the declaration to just the current file.
- The best way to solve this problem is to avoid global variables altogether; use function arguments, local variables, or C++ class members instead.

Makefiles

While doing the 6.824 labs you probably don't need to create your own Makefiles from scratch, because we will give them to you. However, you may need to modify them. This section gives a quick overview of what a Makefile is and how it is used.

A Makefile is used in conjuction with `gmake` and describes how different source and header files in a project relate and how to compile and link them. A Makefile usually bears the meaningful name `Makefile` and its contents are probably roughly similar to this example:

Makefile

```
hellobye : libhellobye.a main.o
        c++ main.o -L. -lhellobye -o hellobye

libhellobye.a : hello.o bye.o
        ar cru libhellobye.a hello.o bye.o
# this command puts hello.o and bye.o in a library libhellobye.a

hello.o : hello.C common.h
        c++ -c hello.C -o hello.o

bye.o : bye.C common.h
        c++ -c bye.C -o bye.o

main.o : main.C common.h
        c++ -c main.C -o main.o

clean :
        rm -f *.a *.o hellobye
```

This Makefile has 6 sections. Each section describes a **dependency** on the first line and some **action** on the second. A dependency consists of a target file (before the colon) and prerequisite files (after the colon). Lines starting with # are comments.

When you run `gmake`, it will execute a target's action if any of the prerequisite files have been modified since the target was last modified. `gmake` executes the action in much the same way as if you had typed it to the shell. `gmake` only tries to make sure that the very first target in the `Makefile` is up to date. However, if the first target depends on other targets, `gmake` may end up executing multiple actions.

The first section in the Makefile above states that the file `hellobye` depends on `libhellobye.a` and `main.o`. If either `libhellobye.a` or `main.o` is newer than `hellobye`, then `hellobye` is rebuilt. Building the whole project is now as simple as typing

```
% gmake
```

If you write your Makefile correctly, gmake will only compile and link those parts of a project that have changed since the last `gmake`. If your project is big and the full compile/link process takes a long time, then gmake is your friend. It saves time.

Notice that the last section in this Makefile is not a real dependency, but allows the user to conveniently call

```
% gmake clean
```

to remove all object files, the library, and the executable.

Makefiles can get pretty complicated, but this gentle introduction may very well suffice for your 6.824 needs. If you're interested, read the full gmake manual (`info make`). It's fabulous.

Libasync in Linux

The lab machines run FreeBSD. However, you may be able to compile and run libasync programs on Linux.

We will not answer Linux-related questions! We discourage you from writing libasync programs on anything else but our lab machines! The grade you get is based only on our own test results on the FreeBSD lab machines. Make sure you verify that the results you get on Linux are the same on FreeBSD.

First and foremost: as opposed to FreeBSD, you need to link your binaries with `-lresolv`. Your friendly teaching assistant has made this really easy for you. If you are using Makefiles that we give you, you can just type:

```
% gmake LFLAGS="-lresolv"
```

The rest of this section explains how to compile and install SFS on your Linux system. You need SFS to compile most of the lab assignments.

The rest of this section assumes you are running RedHat. I myself run RedHat 7.3, but I've had success with 7.2, also. This doesn't mean you can't get it to work on other Linuces.

First of all, you should use GCC 3.1. Forget about RedHat's broken GCC 2.96. Even if, by a stroke of fortune, SFS compiles with GCC 2.96, the resulting executables are broken. I use GCC 3.1 without removing the standard GCC 2.96 installation by installing GCC 3.1 in a separate directory. (I first configured GCC 3.1 with `--prefix=/usr/local/gcc-3.1/`, then ran `gmake install`.)

Before trying to compile anything with GCC 3.1, make sure to prepend `/usr/local/gcc-3.1/bin/` before any other path in environment variable `PATH`. Similarly, prepend `/usr/local/gcc-3.1/lib/` to `LD\_LIBRARY\_PATH`. Verify that you are running the right compiler by doing `gcc -v`.

Libasync is part of SFS. Always check out the latest CVS snapshot using the anonymous CVS repository. You don't have to fully install SFS; a successful build suffices. Grab a coffee---building SFS may take a few hours, depending on your processing power. Online documentation tells you about the SFS compilation process. Libasync is in the `async/` subdirectory in the SFS source tree. You may have to struggle through some compiler errors. I think I fixed most or all of them. Here's a summary of my fixes:

```
Index: agent/sfsauthmgr.C
72c72
< auth_sess_mgr::timeout (bool cb = false)
---
> auth_sess_mgr::timeout (bool cb)
181c181
< authmgr::timeout (bool cb = false)
---
> authmgr::timeout (bool cb)

Index: arpc/authopaque.C
29c29
<   bool_t xdr_opaque_auth(XDR *, struct opaque_auth *);
---
>   bool_t xdr_opaque_auth(XDR *, struct opaque_auth *) __THROW;
```

```
Index: async/daemonize.C
294,295c294,295
< void abort (void) __THROW __attribute__ ((noreturn));
< void exit (int) __THROW __attribute__ ((noreturn));
---
> // void abort (void) __THROW __attribute__ ((noreturn));
> // void exit (int) __THROW __attribute__ ((noreturn));

Index: authserv/authclient.C
31c31
< extern "C" char *crypt (const char *, const char *);
---
> extern "C" char *crypt (const char *, const char *) __THROW;

Index: authserv/authmisc.C
30,32c30,32
< extern "C" char *getusershell(void);
< extern "C" void setusershell(void);
< extern "C" void endusershell(void);
---
> extern "C" char *getusershell(void) __THROW;
> extern "C" void setusershell(void) __THROW;
> extern "C" void endusershell(void) __THROW;

Index: crypt/bigint.h
35c35
< extern "C" {
---
> // extern "C" {
37c37
< }
---
> // }

Index: rex/connect.c
27a28
> #include <stdlib.h>

Index: sfsrodb/sfsrodb.C
48c48
< extern int errno;
---
> // extern int errno;
```

Most of these fixes solve one of these problems:

- GCC 3.1 does not allow a default parameter to be specified in both the header file and the source file. Remove one of the defaults. For example:

  `auth_sess_mgr::timeout (bool cb = false)`

  in `sfsauthmgr.C`, around line 72, should be

  `auth_sess_mgr::timeout (bool cb)`

  since the `= false` part is specified somewhere else, also.

- If you get errors about function definitions being different, try appending `__THROW` to the function definition in the SFS source tree. Apparently, FreeBSD and Linux header files differ somewhat. For example:

```
bool_t xdr_opaque_auth(XDR *, struct opaque_auth *);
```

in `authopaque.C`, around line 29 should be

```
bool_t xdr_opaque_auth(XDR *, struct opaque_auth *) __THROW;
```