Lecture 3

Review of C Programming Tools

Unix File I/O System Calls

Review of C Programming Tools

Compilation

Linkage

The Four Stages of Compilation

- preprocessing
- compilation
- assembly
- linking

gcc driver program (toplev.c)

- cpp: C PreProcessor
- cc1: RTL (Register Transfer Language) processor
- as: assembler
- ld: loader (linker)

The GNU CC Compilation Process

- GCC is portable:
  - multiplatform (intel, MIPS, RISC, Sparc, Motorola, etc.)
  - multiOS (BSD,AIX, Linux, HPUX, mach, IRIX, minix, msdos, Solaris, Windoze, etc.)
  - Multilingual (C, Objective C, C++, Fortran, etc.)
- Single first parsing pass that generates a parsing tree

The GNU CC Compilation Process

- Register Transfer Language generation
  - close to 30 additional passes operate on RTL Expressions (RTXs), constructed from partial syntax trees
  - gcc –c –dr *filename.c*
  - RTL is Lisp-like
    - cond(if_then_else *cond then else*)
    - (eq: *m x y*)
    - (set *lval x*)
    - (call *function numargs*)
    - (parallel [*x0 x1 x2 xn*])
- Final output is assembly language, obtained by mapping RTX to a machine dependency dictionary
  - ~/mark/pub/51081/compiler/i386.md

Assembler Tasks

- converts assembly source code into machine instructions, producing an “object” file (called “.o”)

Loader (Linker) tasks

- The Loader (linker) creates an executable process image within a file, and makes sure that any functions or subprocesses needed are available or known. Library functions that are used by the code are linked in, either statically or dynamically.

Preprocessor Options

- -E preprocess only: send preprocessed output to standard out--no compile
  - output file: file.c -> file.i file.cpp -> file.ii
- -M produce dependencies for make to stdout (voluble)
- -C keep comments in output (used with -E above):
  - -E -C
- -H printer Header dependency tree
- -dM Tell preprocessor to output only a list of macro defs in effect at end of preprocessing. (used with -E above)
  - gcc -E -dM funcs.c |grep MAX

Compiler Options

- -c compile only
- -S send assembler output source to *.s
  - output file: file.c -> file.s
- -w Suppress All Warnings
  - gcc warnings.c
  - gcc -w warnings.c
- -W Produce warnings about side-effects (falling out of a function)
  - gcc -W warnings.c

Compiler Options (cont)

- -I Specify additional include file paths
- -Wall Produce many warnings about questionable practices; implicit declarations, newlines in comments, questionable lack of parentheses, uninitialized variable usage, unused variables, etc.
  - gcc -Wall warnings.c
- -pedantic Warn on violations from ANSI compatibility (only reports violations required by ANSI spec).
  - gcc -pedantic warnings.c

Compiler Options (cont)

- -O optimize (1,2,3,0)
  - -O,-O1 base optimizations, no auto inlines, no loops
  - -O2 performs additional optimizations except inline-functions optimization and loop optimization
  - -O3 also turns on inline-functions and loop optimization
  - -O1 default
- -g include debug info (can tell it what debugger):
  - -gcoff COFF format for sdb (System V < Release 4)
  - -gstabs for dbx on BSD
  - -gxcoff for dbx on IBM RS/6000 systems
  - -gdwarf for sdb on System V Release 4

Compiler Options (cont)

- -save-temps save temp files (foo.i, foo.s, foo.o)
- -print-search-dirs print the install, program, and libraries paths
- -gprof create profiling output for gprof
- -v verbose output (useful at times)
- -nostartfiles skip linking of standard start files, like /usr/lib/crt[0,1].o, /usr/lib/crti.o, etc.
- -static link only to static (.a=archive) libraries
- -shared if possible, prefer shared libraries over static

Assembler Options (use gcc -Wa,-*options* to pass options to assembler)

- -ahl generate high level assembly language source
  - gcc -Wa,-ahl warnings.c
- -as generate a listing of the symbol table
  - gcc -Wa,-as warnings.c

Linker Options (use gcc -Wl,-*options* to pass options to the loader)

- gcc passes any unknown options to the linker
- -l lib (default naming convention lib*lib*.a)
- -L lib path (in addition to default /usr/lib and /lib)
- -s strip final executable code of symbol and relocation tables
  - gcc -w –g warnings.c ; ls -l a.out ; gcc -w -Wl,-s warnings.c ; ls -l a.out
- -M create load Map to stdout

Static Libraries and ar (cd /pub/51081/static.library)

- Create a static library: the ar command:
  - ar [rcdusx] libname objectfiles ...
- Options
  - rcs: add new files to the library and create an index (ranlib) (c == create the library if it doesn't exist)
  - rus: update the object files in the library
  - ds: delete one or more object files from a library
  - x: extract (copy) an object file from a library (remains in library)
  - v: verbose output

Steps in Creating a Static Library (cd ~mark/pub/51081/static.library)

- First, compile (-c) the library source code:
  - gcc -Wall -g -c libhello.c
- Next, create the static library (libhello.a)
  - ar rcs libhello.a libhello.o
- Next, compile the file that will *use* the library
  - gcc -Wall -g -c hello.c
- Finally, link the user of the library to the static library
  - gcc hello.o -lc -L. -lhello -o hello
- Execute: ./hello

Shared Libraries
(cd /pub/51081/shared.library)

- Benefits of using shared libraries over static libraries:
  - saves disk space—library code is in library, not each executable
  - fixing a bug in the library doesn't require recompile of dependent executables.
  - saves RAM—only one copy of the library sits in memory, and all dependent executables running share that same code.

Shared Library Naming Structure

- soname: libc.so.5
  - minor *v*ersion and *r*elease number:
    - libc.so.5.*v*.*r* eg: libc.so.5.3.1
  - a soft link libc.so.5 exists and points to the *real* library libc.so.5.3.1
    - that way, a program can be linked to look for libc.so.5, and upgrading from release to libc.so.5.3.2 just involves resetting the symbolic link libc.so.5 from libc.so.5.3.1 to libc.so.5.3.2.
    - ldconfig does this automatically for system libraries (man ldconfig, /etc/ld.so.conf)

Building a shared library: Stage 1: Compile the library source

- Compile library sources with -fPIC (Position Independent Code):
  - gcc -fPIC -Wall -g -c libhello.c
  - This creates a new shared object file called libhello.o, the object file representation of the new library you just compiled
- Create the *release* shared library by linking the library code against the C library for best results on all systems:
  - gcc -g -shared –Wl,-soname,libhello.so.1 -o libhello.so.1.0.1 libhello.o –lc
  - This creates a new release shared library called libhello.so.1.0.1

Building a shared library: Stage 2: Create Links

- Create a soft link from the *minor* version to the release library:
  - ln -sf libhello.so.1.0.1 libhello.so.1.0
- Create a soft link from the *major* version to the *minor* version of the library:
  - ln -sf libhello.so.1.0 libhello.so.1
- Create a soft link for the *linker* to use when linking applications against the new release library:
  - ln -sf libhello.so.1.0.1 libhello.so

Building a shared library: Stage 3: Link Client Code and Run

- Compile (-c) the client code that will *use* the release library:
  - gcc -Wall -g -c hello.c
- Create the dependent executable by using -L to tell the linker where to look for the library (i.e., in the current directory) and to link against the shared library (-lhello == libhello.so):
  - gcc -Wall -g -o hello hello.c -L. -lhello
- Run the app:
  - LD_LIBRARY_PATH=. ./hello

How do Shared Libraries Work?

- When a program runs that depends on a shared library (discover with ldd progname), the dynamic linker will attempt to find the shared library referenced by the soname
- Once all libraries are found, the dependent code is dynamically linked to your program, which is then executed
- Reference: The Linux Program-Library HOWTO

Unix File I/O

Unix System Calls

- System calls are low level functions the operating system makes available to applications via a defined API (Application Programming Interface)
- System calls represent the *interface* the kernel presents to user applications

A File is a File is a File
--Gertrude Stein

- Remember, “Everything in Unix is a File”
- This means that all low-level I/O is done by reading and writing file handles, regardless of what particular peripheral device is being accessed—a tape, a socket, even your terminal, they are all *files*.
- Low level I/O is performed by making *system calls*

User and Kernel Space

- System memory is divided into two parts:
  - user space
    - a process executing in user space is executing in user mode
    - each user process is protected (isolated) from another (except for shared memory segments and mmapings in IPC)
  - kernel space
    - a process executing in kernel space is executing in kernel mode
- Kernel space is the area wherein the kernel executes
- User space is the area where a user program normally executes, *except when it performs a system call.*

Anatomy of a System Call

- A System Call is an explicit request to the kernel made via a software interrupt
- The standard C Library (libc) provides *wrapper routines*, which basically provide a user space API for all system calls, thus facilitating the context switch from user to kernel mode
- The wrapper routine (in Linux) makes an interrupt call 0x80 (vector 128 in the Interrupt Descriptor Table)
- The wrapper routine makes a call to a system call handler (sometimes called the “call gate”), which executes in kernel mode
- The system call handler in turns calls the system call interrupt service routine (ISR), which also executes in kernel mode.

Regardless…

- Regardless of the type of file you are reading or writing, the general strategy remains the same:
  - creat() a file
  - open() a file
  - read() a file
  - write() a file
  - close() a file
- These functions constitute Unix *Unbuffered I/O*
- ALL files are referenced by an integer *file descriptor* (0 == STDIN, 1 == STDOUT, 2 == STDERR)

read() and write()

- Low level system calls return a count of the number of *bytes* processed (read or written)
- This count may be less than the amount requested
- A value of 0 indicates EOF
- A value of –1 indicates ERROR
- The BUFSIZ #define (8192, 512)

A Poor Man's cat (~mark/pub/51081/io/simple.cat.c)

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char * argv [])
{
   char buf[BUFSIZ];
   int numread;
   while((numread = read(0, buf, sizeof(buf))) > 0)
       write(1, buf, numread);
   exit(0);
}
```

- Question: Why didn't we have to open file handles 0 and 1?

read()

```
#include <unistd.h>
ssize_t read(int fd, void * buf, size_t
count);
```

- If read() is successful, it returns the number of bytes read
- If it returns 0, it indicates EOF
- If unsuccessful, it returns –1 and sets errno

write()

```
#include <unistd.h>
ssize_t write(int fd, void * buf, size_t
count);
```

- If write() is successful, it returns the number of bytes written to the file descriptor, this will usually equal *count*
- If it returns 0, it indicates 0 bytes were written
- If unsuccessful, it returns –1 and sets errno

open()

```
#include <fcntl.h>
int open(const char * path, int flags[, mode_t
   mode]);
```

- *flags* may be OR'd together:
  - O_RDONLY open for reading only
  - O_WRONLY open for writing only
  - O_RDRW open for both reading and writing
  - O_APPEND open for appending to the end of file
  - O_TRUNC truncate to 0 length if file exists
  - O_CREAT create the file if it doesn't exist
- *path* is the pathname of the file to open/create
- file descriptor is returned on success, -1 on error

creat()

- Dennis Ritchie was once asked what was the single biggest thing he regretted about the C language. He said “leaving off the ‘e’ on creat()”.
- The creat() system call creates a file with certain permissions:

```
int creat(const char * filename, mode_t mode);
```

- The mode lets you specifiy the permissions assigned to the file after creation
- The file is opened for *writing* only

open() (create file)

- When we use the O_CREAT flag with open(), we need to define the mode (rights mask from **sys/stat.h**):
  - S_IRUSR read permission granted to OWNER
  - S_IWUSR write permission granted to OWNER
  - S_IXUSR execute permission granted to OWNER
  - S_IRGRP read permission granted to GROUP
    - etc.
  - S_IROTH read permission granted to OTHERS
    - etc.
- Example:

```
int fd = open("/path/to/file", O_CREAT, S_IRUSR |
S_IWUSR | S_IXUSR | S_IRGRP | S_IROTH);
```

close()

```
#include <unistd.h>
int close( int fd );
```

- close() closes a file descriptor (fd) that has been opened.
- Example: ~mark/pub/51081/io/mycat.c

lseek()
(~mark/pub/50181/lseek/myseek.c)

```
#include <sys/types.h>
#include <unistd.h>
long lseek(int fd, long offset, int startingpoint)
```

- lseek moves the current file pointer of the file associated with file descriptor *fd* to a new position for the next read/write call
- *offset* is given in number of bytes, either positive or negative from *startingpoint*
- *startingpoint* may be one of:
  - SEEK_SET move from beginning of the file
  - SEEK_CUR move from current position
  - SEEK_END move from the end of the file

Error Handling (~mark/pub/51081/io/myfailedcat.c)

- System calls set a *global* integer called errno on error:
  - extern int errno; /* defined in /usr/include/errno.h */
- The constants that errno may be set to are defined in </usr/include/asm/errno.h>. For example:
  - EPERM operation not permitted
  - ENOENT no such file or directory (not there)
  - EIO I/O error
  - EEXIST file already exists
  - ENODEV no such device exists
  - EINVAL invalid argument passed

```
#include <stdio.h>
void perror(const char * s);
```

stat():

int stat(const char * *pathname*; struct stat **buf*);

- The stat() system call returns a structure (into a buffer you pass in) representing all the stat values for a given filename. This information includes:
  - the file's mode (permissions)
  - inode number
  - number of hard links
  - user id of owner of file
  - group id of owner of file
  - file size
  - last access, modification, change times
  - less /usr/include/sys/stat.h => /usr/include/bits/stat.h
  - less /usr/include/sys/types.h (S_IFMT, S_IFCHR, etc.)
- Example: ~/UofC/51081/pub/51081/stat/mystat.c