Lecture 3

Introduction to Unix Systems Programming:

Unix File I/O System Calls

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

make

What is make?

- make is used to:
  - save time by not recompiling files that haven't changed
  - make sure all files that have changed *do* get recompiled

The Concept

- make is a program that will update targets on the basis of changes in dependencies.
- Although it is mostly used to build software by compiling and linking, it can be used to manage any construction project that involves creating something based on something else (e.g., using nroff over a series of book chapters).
- A makefile is nothing more than dependencies and rules. A rule describes HOW to create the target from the dependencies.

Calling Convention and Options

- -n don't make, but print out what would be done
- -k keep going, don't stop on errors, which is the default
- -f run makefile specified by filename
- Default makefile naming convention
  - makefile
  - Makefile

Dependencies and Rules

- Dependencies and Syntax
  - target: dep1 dep2 depn
  - make will build the first target it finds
  - this target is commonly called "all"
    - all: bigapp
- Rules
  - It is a rule that *every* rule must begin with a single TAB character!
    - [TAB] gcc -c 1.c
- make has several built-in rules
  - make -p will show them to you
- *Examples (~mark/pub/51081/makefile.demo): simple, make1*

Macros and Multiple Targets

- a MACRO is a substitutable syntax to give flexibility and genericity to rules
- Forms:
  - MACRONAME=value
  - access with either:
    - $(MACRONAME) or
    - ${MACRONAME} or (sometimes) $MACRONAME
  - undefine a MACRO with:
    - MACRONAME=
- A macro can be redefined at the command line:
  - make CC=aCC #for HP Ansi compiler
- *Examples:* (make2, make3)

Suffix Rules

- a Suffix Rule is a directive that applies rules and macros to generic suffixes
- tell make about a new suffix: SUFFIXES: .cpp
- tell make how to compile it: .cpp.o:
- then the rule: $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
- Built in suffix macros:
  - **$@** The full name of the current target
  - **$?** A list of modified dependencies (a list of files newer than the target on which the target depends)
  - **$<** The single file that is newer than the target on which the target is dependent
  - **$*** The name of the target file, WITHOUT its suffix (i.e., without the .c or .cpp, etc.)
- *examples (make5)*

Debugging with gdb and ddd

What is a bug?

- a bug exists when executable code returns or causes results that are unintended or undesirable.
  - You can only have a bug in code that's compiled or a shell script that's executed by the shell (ie. the compiler or shell do not give errors about compilation).
- Don't confuse design errors with code bugs (don't confuse design with implementation)

Finding bugs

- Problem statement: Code runs fast and furious--we must find out "where" in the code the problem *originates*.
- Solution statement:
  - attempt to make bug repeatable--this is empirical analysis, pure and simple.
  - printf's can help, displaying variables, but they're limited.
    - gcc -o cinfo -DDEBUG cinfo.c
    - cinfo
  - __DATE__, __TIME__, __LINE__
- *Examples:* (in ~mark/pub/51081/debug) cinfo.c

Interactive Debuggers

- But interactive debuggers are MUCH better, because they offer:
  - run time code stepping
  - variable analysis and modification
  - breakpoints (multiple forms)
- Compile for debugging: -g
  - *Try* to void optimizing when debugging
- remaining problems:
  - loop tracing (problem doesn't arise until loop has executed 1M times)
  - Optimization problems
  - Intermittency
- *Examples:* debug3 (gdb); debug4 (ddd)