CS360 Lecture notes -- Logfiles and more on I/O

- Jim Plank
- Directory: **/blugreen/homes/plank/cs360/notes/Logfile**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Logfile/lecture.html**

This lecture covers more various aspects of file i/o. It finishes chapter 3 in the book.

Writing log files

Try the following command on your machine:

```
UNIX> last | head -60
```

You'll see that it prints out the 60 most recent logins to your machine. This is a good example of the use of a "log file." What happens is that the system program responsible for letting users log into a system appends the login time, and the logout time to the file "**/var/adm/wtmp**". The "**last**" program reads "**/var/adm/wtmp**" and prints it to the screen an a readable form.

Now, suppose we wanted to write a program to append to a log. For example, suppose I wanted to log whenever someone used my **fill** program (do you remember the **fill** program from CS302? If not, **fill** is just a simple program that is used to justify text). The first thing I'd do is write a procedure something like the one below, and then put a call to it in my **fill** program.

(This is in **logfill1.c**, which is a program that simply makes a call to **write_to_fill_log**).

```
#define LOG_FILE "/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file"

write_to_fill_log()
{
  char *username;
  long t;
  FILE *f;

  username = getenv("USER");
  t = time(0);

  f = fopen(LOG_FILE, "a");
  /* error check */

  fprintf(f, "%s %ld\n", username, t);
  fclose(f);
}
```

As before, feel free to copy all the .c files and the makefile to a directory of your own, and compile and run them. Here, go ahead and run **logfill1**. Then check the file **/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file** and see if indeed your log entry is there. Read the manual page for **time** (man -s 2 time) to see what the time number is. You can do this multiple times. I don't care.

Now, this seems to work fine, but the question arises -- what happens if two people in different processes call

write_to_fill_log() simultaneously? Specifically, what happens if one process gets switched off the CPU by the operating system just after the call to **fopen()**, and another process runs and calls **write_to_fill_log()**? You can get the answer in chapter 3 of the book -- what happens is that both processes see the same value for the end of file, and thus will both write to the same location in the file. This may well corrupt the log file. You might say "so, how many times are two people going to be calling **write_to_fill_log()** simultaneously? Should I really care about this?" The answer is maybe. If you think the chances that two people will call this routine simultaneously are sufficiently negligible, then don't worry about it. However, if these chances are non-trivial (as for the "**last**" routine above), then you have to do something about it.

The way you can deal with this problem is via the flag **O_SYNC** in the **open()** system call. Read the man page for a full description. The working code for **write_to_fill_log()** is in **logfill2.c**, and the log file is **/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file2**. Again, try it out and see if your entry is in the log file.

```
#include < fcntl.h >
#include < stdio.h >
#include < stdlib.h >

#define LOG_FILE "/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file2"

write_to_fill_log()
{
  char *username;
  long t;
  int fd;
  char s[100];

  username = getenv("USER");
  t = time(0);

  fd = open(LOG_FILE, O_APPEND | O_SYNC | O_CREAT | O_WRONLY, 0666);

  if (fd < 0) {
    fprintf(stderr, "Can't write log file %s\n", LOG_FILE);
    return;
  }

  sprintf(s, "%s %ld\n", username, t);
  write(fd, s, strlen(s));
  close(fd);
}
```

Atomic Actions

An atomic action is a sequence of actions that gets executed as a whole, uninterrupted by the CPU. They are important in systems programming as they can make guarantees that would be impossible to make otherwise.

Two good examples of atomic actions are :

- 1. **Write()** calls when a file has been opened (**O_APPEND | O_SYNC**).

  When you do a write to such a file, two things happen: An **lseek()** is done to the end of file, and then a write is performed. If these two actions are not performed atomically, then the end of file could change

(by another process writing to it) in between the **lseek()** and the **write()**. Thus, it is necessary for the operating system to ensure that these actions are atomic.

Now, note that if you are not anticipating that a file will be shared, you don't have to worry about these things -- just **open()** or **fopen()** it normally. It's only special files (such as log files) that have such requirements.

- 2. Opening a file only if it doesn't exist.

  Suppose you want to open the file **f1** for writing only if it doesn't already exist. Then you might try the following segment of code:

```
int fd;

fd = open("f1", O_WRONLY);
if (fd < 0) {
  fd = open("f1", O_WRONLY | OCREAT, 0644);
} else {
  close(fd);
  fprintf(stderr, "Error: f1 already exists\n");
  exit(1);
}
```

  Does this ensure that **f1** will only be opened if it doesn't already exist? The answer is yes only if the segment of code could be executed atomically. Otherwise, the CPU can interrupt the program between the first **fd = open()** statement and the **if** statement, and some other process can change the status of the file **f1**. Since you can't guarantee that your C code will be executed uninterrupted, Unix provides a way to open a file if and only if it doesn't already exist. That is:

```
fd = open("f1", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

  It checks for existence, and returns a fd value >= 0 only if if **f1** doesn't already exist.

In other parts of this class (and in other parts of computer science, especially parallel processing and databases), you will hear the term ``executing x atomically.'' That means, as described here, that x must be executed without interruption. Note that you can't make up your own atomic actions. In other words, you can't specify to the operating system that some block of code be executed atomically. Instead, there are some atomic actions (like the ones described above) that the operating system has implemented for you. In other courses (e.g. databases, operating systems, and maybe later in this class if we get to threads), you'll be able to define your own atomic actions.

Section 3.11 of the book covers atomic actions as well. Please give it a reading.

Umask

Read chapter 4.8, and the **umask** man page (say "**man -s 2 umask**"):

```
umask() sets the process's file creation mask  to  mask  and
returns  the  previous  value  of the mask.  The low-order 9
bits of mask are used whenever a file is  created,  clearing
corresponding  bits  in  the  file access permissions.  (see
```

```
     stat(2V)).  This clearing restricts the default access to  a
     file.

     The mask is inherited by child processes.
```

When you call **umask** from a program, or from the shell, it changes the "File creation mask". This mask consists of 9 bits. Whenever a file is created, for example by **open()**, **creat()**, or **mkdir()**, and a mode **m** is specified, then the file is created with the mode:

```
(m & ~umask)
```

Umask the system call returns the old **umask** value.

For example, look at the following program (**um1.c**):

```
main()
{
  int i;
  int old_mask;

  old_mask = umask(0);
  i = open("f1", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  close(i);
  printf("created f1: 0666\n");
  i = open("f2", O_WRONLY | O_CREAT | O_TRUNC, 0200);
  close(i);
  printf("created f2: 0200\n");

  umask(022);
  i = open("f3", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  close(i);
  printf("created f3: %o\n", 0666 & ~022 & 0777);
  i = open("f4", O_WRONLY | O_CREAT | O_TRUNC, 0777);
  close(i);
  printf("created f4: %o\n", 0777 & ~022 & 0777);
  i = open("f5", O_WRONLY | O_CREAT | O_TRUNC, 0200);
  close(i);
  printf("created f5: %o\n", 0200 & ~022 & 0777);
}
```

```
UNIX> um1
created f1: 0666
created f2: 0200
created f3: 644
created f4: 755
created f5: 200
UNIX> ls -l f?
-rw-rw-rw-  1 plank           0 Sep 28 15:05 f1
--w-------  1 plank           0 Sep 28 15:05 f2
-rw-r--r--  1 plank           0 Sep 28 15:05 f3
-rwxr-xr-x  1 plank           0 Sep 28 15:05 f4
--w-------  1 plank           0 Sep 28 15:05 f5
UNIX>
```

The **umask** value is set per process, not per user. So, if your shell's **umask** is 022, and you have a program set it to 0, then that does not affect the shell:

```
UNIX> cat um2.c
```

```
main()
{
  umask(0);
}
UNIX> umask
22
UNIX> um2
UNIX> umask
22
UNIX>
```

Random File/Inode System calls.

- chmod(char *path, mode_t mode) -- Works just like chmod when executed from the shell. E.g. chmod("f1", 0600) will set the protection of file f1 to be rw- for you, and --- for everyone else.

  Read section 4.9 for a more thorough description, including use of the mode bits like S_IRUSR, etc.

- chown() -- ignore.

- link, unlink, remove, rename: Section 4.15 of the book

  pretty straightforward: link(char *f1, char *f2) works just like

  ```
  UNIX> ln f1 f2
  ```

  f2 has to be a filename though, and cannot be a directory.

  unlink(char *f1) works like

  ```
  UNIX> rm f1
  ```

  remove(char *f1) works like unlink, but it also works for (empty) directories. unlink fails on directories.

  rename(char *f1, char *f2) works just like

  ```
  UNIX> mv f1 f2
  ```

- symlink and readlink: Read chapter 4.17. These routines mess with symbolic links.

- utime: Read chapter 4.19. These routines let you change the time fields of a file's inode. This system call looks like it should be illegal (for example, one could write a program to make it look like one has finished his homework on time...), but it is very handy, especially for writing tar (and jtar).

- mkdir and rmdir: Again straightforward, and like

  ```
  UNIX> mkdir ...
  UNIX> rmdir ...
  ```

  Read chapter 4.20

- chdir, getcwd: Like

```
UNIX> cd ..
UNIX> pwd
```

Read chapter 4.22. You will not need these for jtar, but can use them if you'd like.