CS360 Lecture notes

LICENSE

CS360 Lecture notes
Umask and Other System Calls

- James S. Plank
- Directory: **/home/jplank/cs360/notes/Umask-And-Others**
- Start Date: In the 1990's.
- Latest revision: Wed Feb 23 12:11:47 EST 2022
- Git Hash: 96d4718adaafb536f67d2e708e8ff0af3ca2c1aa

This is a catch-all lecture, to go over some system calls that I don't go over in other parts of the class. With the exception of **umask**, they are straightforward, so if I end up being short on lecture time, I will skip them and just refer you here. Many of them will be useful for you when you write **jtar**.

Umask

Umask is a system call that handles the "file mode creation mask." Here's a man page (from my Mac in 2018):

```
NAME
    umask -- set file creation mode mask

SYNOPSIS
    #include <sys/stat.h>

    mode_t
    umask(mode_t cmask);

DESCRIPTION
    The umask() routine sets the process's file mode creation mask to cmask and returns
    the previous value of the mask.  The 9 low-order access permission bits of cmask are
    used by system calls, including open(2), mkdir(2), mkfifo(2), and mknod(2) to turn
    off corresponding bits requested in file mode.  (See chmod(2)).  This clearing allows
    each user to restrict the default access to his files.

    The default mask value is S_IWGRP | S_IWOTH (022, write access for the owner only).
    Child processes inherit the mask of the calling process.

RETURN VALUES
    The previous value of the file mode mask is returned by the call.

ERRORS
    The umask() function is always successful.
```

The "file creation mask" (which I will call the "umask" out of habit) is a nine-bit number. If a bit in the umask is set, then whenever you make a system call that creates a file, that bit in the protection mode will be turned off.

Formally, when you specify a **mode** when you open a file, the real protection mode will be:

`(mode & ~ umask)`

Until you get used to "AND-NOT", it can be confusing. If you have a mask m, then:

- `x = (a & m)` will set x to all of the bits in a in the positions of the one bits of m. Put another way, it will "turn off" all of the zero bits of m in a.
- `x = (a & ~ m)` will set x to all of the bits in a in the positions of the zero bits of m. Put another way, it will "turn off" all of the one bits of m in a.

So, the umask "turns off" protection bits. The point of the umask is to allow programs to create files with the following protection modes:

- Regular text and data files may be opened with the mode 0666.
- Directories and executable files may be opened with the mode 0777.

The user can tailor the protection modes to his or her liking with the umask.

In the examples that follow, I'm not going to make the system call, but simply use the **umask** command, which does the same thing, but in the current shell. If I type `umask` into my shell, then it will tell me the current umask, in octal:

```
UNIX> umask
22
UNIX> echo "Hi" > f1.txt
UNIX> umask 0
UNIX> echo "Hi" > f2.txt
UNIX> umask 77
UNIX> echo "Hi" > f3.txt
UNIX> umask 777
UNIX> echo "Hi" > f4.txt
UNIX> ls -l f?.txt
-rw-r--r--. 1 plank loci 3 Feb 13 09:53 f1.txt
-rw-rw-rw-. 1 plank loci 3 Feb 13 09:53 f2.txt
-rw-------. 1 plank loci 3 Feb 13 09:53 f3.txt
----------. 1 plank loci 3 Feb 13 09:54 f4.txt
UNIX>
```

The shell, when it opens a file for output redirection, uses a mode of 0666. As you can see from above:

- When my umask is 022, then the "group" and "world" write bits are turned off, so the file has a protection mode of 0644.
- When my umask is 0, then no bits are turned off, and the file has a protection mode of 0666.
- When my umask is 077, then all of the group and world bits are turned off, so the file has a protection mode of 0600.
- When my umask is 0777, then all nine bits, so the file has a protection mode of 0000. You'll note, that I was still allowed to write "Hi" into it, because the **open()** call gave me a legal file descriptor for writing. It's just that no other process can now open the file.

The same thing is true of directories:

```
UNIX> rm -rf f?.txt
UNIX> umask 22
UNIX> mkdir d1
UNIX> umask 0
UNIX> mkdir d2
UNIX> umask 077
UNIX> mkdir d3
UNIX> umask 0777
UNIX> mkdir d4
```

```
UNIX> ls -l | grep 'd.$'
drwxr-xr-x. 2 plank loci     6 Feb 13 09:59 d1
drwxrwxrwx. 2 plank loci     6 Feb 13 09:59 d2
drwx------. 2 plank loci     6 Feb 13 09:59 d3
d---------. 2 plank loci     6 Feb 13 10:00 d4
UNIX> rm -rf d?
UNIX> umask 22
UNIX>
```

You'll note, in the **umask** command, I don't need to include the initial 0 – it interprets its argument in octal. In the system call, you should specify octal.

Random File/Inode System calls.

These are sketchy because they are straightforward.

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.

The man page for **chmod()** – "man -s 2 chmod" shows you a bunch of **#define's** from **¡sys/stat.h¿** that are useful for accessing individual bits from the mode.

Quiz yourself on your understanding of how **open()** and **chmod()** interact. Compile and run [**src/o1.c**](**https://web.eecs.utk.ed**
And-Others/src/o1.c):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    int fd;

    printf("Opening the file:\n");
    fd = open("f1.txt", O_WRONLY | O_CREAT | O_TRUNC);
    sleep(1);

    printf("Doing chmod\n");
    chmod("f1.txt", 0000);
    sleep(1);

    printf("Doing write\n");
    write(fd, "Hi\n", 3);

    return 0;
}
```

```
UNIX> bin/o1
Opening the file:
Doing chmod
Doing write
UNIX> ls -l f1.txt
What will this show as the protection mode and the size of the file?
UNIX> cat f1.t xt
What will this do?
UNIX>
```

The answers are as follows:

```
UNIX> ls -l f1.txt
----------. 1 plank loci 3 Feb 13 10:07 f1.txt
UNIX>
```

The file descriptor is a valid file descriptor for writing. The **chmod()** command did not do anything to the open file, so the process can successfully write with it. That is why the file's size is three, and not zero. The protection mode, of course, was changed by the **chmod** command.

```
UNIX> cat f1.txt
cat: f1.txt: Permission denied
UNIX>
```

Since the protection mode was "----------", the **cat** program received an error when it tried to open the file (most likely with **fopen()**, which calls **open()**).

Suppose we run bin/o1 again:

```
UNIX> bin/o1
Opening the file:
Doing chmod
Doing write
UNIX> ls -l f1.txt
----------. 1 plank loci 3 Feb 13 10:07 f1.txt
UNIX>
```

You'll note that the modification time of **f1.txt** has not changed. This is because the **open()** call failed and returned -1. The file was not truncated or modified in any way. That's why the modification time is unchanged. The **chmod()** command succeeded, but the **write()** system call was given a file descriptor of -1, so it failed too.

Let's kill that file:

```
UNIX> rm -f f1.txt
UNIX>
```

---

link, unlink, remove, rename: These are straightforward: **link(char *f1, char *f2)** works just like:

```
UNIX> ln f1 f2
```

f2 has to be a file – it 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. Take a look at **[src/o2.c](https://web.eecs.utk.edu/ jplank/plank/classes/cs360/360/notes/Umask-And-Others/src/o2.c)**:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    int fd;
    char s[11];
    int i;

    // printf("Opening f1.txt and putting \"Fun Fun\" into s.\n");
    strcpy(s, "Fun Fun\n");
    fd = open("f1.txt", O_RDONLY);
    sleep(1);

    // printf("Removing f1.txt\n");
    remove("f1.txt");
    sleep(1);

    // printf("Listing f1.txt, and reading 10 bytes from the open file descriptor.\n");
    system("ls -l f1.txt");
    i = read(fd, s, 10);
    s[i] = '\0';
    // printf("Read returned %d: %d %s\n", i, fd, s);
    return 0;
}
```

This program opens **f1.txt** for reading, sleeps for a second, and then removes **f1.txt**. It sleeps again, performs a long listing, and then tries to read 10 bytes from the open file. The question is—what happens when we remove **f1.txt**? Will the read call succeed, or fail because the file is gone?

```
UNIX> rm -f f1.txt
UNIX> echo "Jim Plank" > f1.txt
UNIX> bin/o2
Opening f1.txt and putting "Fun Fun" into s.
Removing f1.txt
Listing f1.txt, and reading 10 bytes from the open file descriptor.
ls: cannot access f1.txt: No such file or directory
Read returned 10: 3 Jim Plank

UNIX>
```

The `ls` command shows that **f1.txt** is indeed gone after the `remove()` call. However, the operating system does not delete the file until the last file descriptor to it is closed. For that reason, the `read()` call succeeds.

Try `bin/o2` again—since **f1.txt** was removed, it does not exist now:

```
UNIX> bin/o2
Opening f1.txt and putting "Fun Fun" into s.
Removing f1.txt
Listing f1.txt, and reading 10 bytes from the open file descriptor.
ls: cannot access f1.txt: No such file or directory
Read returned -1: -1 Fun Fun

UNIX>
```

What happened? First, the `open()` call failed and returned -1. Thus, the `read()` call also failed and returned -1. Since the `read()` call failed, the bytes of `s` were never overwritten. Therefore, when we printed them out, we got "Fun Fun." Make sure you understand this code and its output. It is deterministic—we are not getting segmentation violations or random behavior with these calls—we are simply getting well-defined errors in our system calls.

---

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

```
UNIX> mv f1 f2
```

---

symlink and **readlink**: These routines mess with symbolic links. You will need them in your **jtar** assignment. Go ahead and read the man pages for these.

---

mkdir and **rmdir**: Again straightforward, and like:

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

Read the man pages.

---

chdir, getcwd: These are like the shell commands **cd** and **pwd**. You will not need these for **jtar**, but can use them if you'd like.

---

utime: This system call lets you change the time fields of a file's inode. It 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**). As always, read the man page. When working with time values, you need to be aware of a few data structures:

- **time_t**: This is a **long**, which contains the number of seconds since time began (January 1, 1970). The **time()** system call returns the current time on your machine as a **time_t**. Call it with **time(0)**.
- **struct timeval**: This has the following definition:

```
struct timeval {
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
};
```

These give you a finer resolution. You can get the current time with this resolution by calling **gettimeofday()**.

- **struct tm**. This one has the following definition:

```
struct tm {
    int tm_sec;         /* seconds */
    int tm_min;         /* minutes */
    int tm_hour;        /* hours */
    int tm_mday;        /* day of the month */
    int tm_mon;         /* month */
    int tm_year;        /* year */
    int tm_wday;        /* day of the week */
    int tm_yday;        /* day in the year */
    int tm_isdst;       /* daylight saving time */
};
```

Do "man ctime" to see a list of procedures that convert between **time_t** data types, **struct tm** data types, and strings. Useful ones are **ctime()**, **localtime()**, **mktime()**, **asctime()**, and **strftime()**. I'll go over a few of these below.

---

time(), localtime(), asctime() and mktime()

Here are synposes:

```
#include <time.h>

time_t time(time_t *tloc);                  /* Returns or fills in the current time as a
    time_t */
struct tm *localtime(const time_t *clock); /* Turns a time_t into a (struct tm *) */
char *asctime(const struct tm *timeptr);   /* Returns a printable string from a (struct
    tm *) */
time_t mktime(const struct tm *timeptr);   /* Turns a (struct tm *) into a time_t */
```

With **time()**, if you give it an argument of NULL or 0, it ignores the argument.

You should always be careful when something returns a pointer for you. Unless the man page specifies that it is created with `malloc()`, you should assume it is not created with `malloc()`. Let me illustrate below with `localtime()`. The program is `src/t1.c`.

```
/* This program shows time(), localtime() and asctime(). */

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    time_t now;
    struct tm *tm1, *tm2;

    /* Set now to the current time,
        use localtime() to convert it to a (struct tm *)
        and print them both out. */

    now = time(0);
```

```
    tm1 = localtime(&now);
    printf("Seconds: %ld.  Asctime(tm1): %s\n", now, asctime(tm1));

    /* Add an hour to "now" and do the same, using
        tm2 for the (struct tm *) */

    now += 3600;
    tm2 = localtime(&now);
    printf("Seconds: %ld.  Asctime(tm2): %s\n", now, asctime(tm2));

    /* Print out tm1.  Is that what you expect? */

    printf("Asctime(tm1): %s\n", asctime(tm1));

    /* Print the pointers.*/

    printf("0x%lx 0x%lx\n", (long unsigned int) tm1, (long unsigned int) tm2);
    return 0;
}
```

This is straightforward until the last line:

```
UNIX> bin/t1
Seconds: 1645637115.  Asctime(tm1): Wed Feb 23 12:25:15 2022   # You'll note asctime() includes a newline.

Seconds: 1645640715.  Asctime(tm2): Wed Feb 23 13:25:15 2022   # We've added an hour to the time_t,
                                                                 and it's reflected in tm2.

Asctime(tm1): Wed Feb 23 13:25:15 2022                         # You'll note, tm1 has "changed" too.

0x7fb062c017a0 0x7fb062c017a0                                  # Why?  Because localtime() always
                                                                 returns the same pointer.
UNIX>
```

Manupulating time with the (struct tm *)

If you're manipulating time, it's typically better to convert to a **(struct tm *)** and manipulate that. The program **[src/t2.c](https://web.eecs.utk.edu/ jplank/plank/classes/cs360/360/notes/Umask-And-Others/src/t2.c)** demonstrates by successively adding a year to the current time for 9 years:

```
/* This program shows how you can use the (struct tm *) to manipulate time. */

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    time_t years[10];         /* Now, and every year up to 9 years from now */
    struct tm *tm1;
    int i;
```

```
    /* Set years[0] to now, convert to tm1 and print. */

    years[0] = time(0);
    tm1 = localtime(&years[0]);
    printf("Base time: %ld. %14s  %s", years[0], "", asctime(tm1));

    /* Now use the (struct tm *) to increment the years, and print.
        When you run this, you'll see that leap years are handled correctly. */

    for (i = 1; i < 10; i++) {
    tm1->tm_year += 1;
    years[i] = mktime(tm1);
    printf("Year +%d:   %ld. Diff: %ld. %s", i, years[i], years[i]-years[i-1],
        asctime(tm1));
    }

    return 0;
}
```

When we run it, you'll see it handles the leap years correctly—their **time_t**'s differ from the non-leap years:

```
UNIX> bin/t2
Base time: 1645637432.                 Wed Feb 23 12:30:32 2022
Year +1:   1677173432. Diff: 31536000. Thu Feb 23 12:30:32 2023
Year +2:   1708709432. Diff: 31536000. Fri Feb 23 12:30:32 2024
Year +3:   1740331832. Diff: 31622400. Sun Feb 23 12:30:32 2025     # Leap year handled correctly
Year +4:   1771867832. Diff: 31536000. Mon Feb 23 12:30:32 2026
Year +5:   1803403832. Diff: 31536000. Tue Feb 23 12:30:32 2027
Year +6:   1834939832. Diff: 31536000. Wed Feb 23 12:30:32 2028
Year +7:   1866562232. Diff: 31622400. Fri Feb 23 12:30:32 2029     # Leap year handled correctly
Year +8:   1898098232. Diff: 31536000. Sat Feb 23 12:30:32 2030
Year +9:   1929634232. Diff: 31536000. Sun Feb 23 12:30:32 2031
UNIX>
```
