

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 2

A more careful look to some Linux System calls

In the lecture, you have learnt that user level programs interacts with the OS kernel through an application program interface (API). We usually call functions in this interface to be a *system call*. Some of them are illustrated in the lectures, and we will use these system calls during our upcoming tutorial. It is essential to have a deeper understanding of those system calls before attending the tutorial. The canonical source of information about system calls is the manual pages, accessible using the `man` command. However, the manual pages may be too technical for you at this moment, and this note provides a more gentle description to the system calls. Please refer to the lecture notes to see working examples.

1. File related system calls: *open*, *read*, *write* and *close*

In Unix, files and I/O are performed in a similar fashion, using the following system calls.

Interface	Include files
int <i>open</i> (const char * <i>pathname</i> , int <i>flags</i>)	<fcntl.h>, <sys/types.h>, <sys/stat.h>
<i>ssize_t</i> <i>read</i> (int <i>fd</i> , void * <i>buf</i> , <i>size_t</i> <i>count</i>)	<unistd.h>
<i>ssize_t</i> <i>write</i> (int <i>fd</i> , void * <i>buf</i> , <i>size_t</i> <i>count</i>)	<unistd.h>
int <i>close</i> (int <i>fd</i>)	<unistd.h>

The *open()* system call is used to “open a file”, which converts a *pathname* into a file descriptor, which can then be used as the *fd* argument of the other system calls. In the *open()* system call, *flag* is one of *O_RDONLY*, *O_WRONLY* or *O_RDWR*, which requests opening the file for read only, write only and both read/write respectively. In addition, *flag* may be bitwise-or’d (using the `|` operator) with one or more flags to modify its behaviour. For example, the *O_CREAT* flag specifies that the OS should attempt to create the file if it is not in the file system. (Without *O_CREAT*, it is an error to open a non-existent file.)

In general, if a system call fails, `-1` is returned, and the **int**-type global variable *errno* is set to identify the error. To use this variable, include the header file <errno.h>. For example, if you attempt to open a file which is actually a directory, *errno* is set to *EISDIR* (21). If you want a string rather than just an integer, you can use the *strerror(int)* function defined in <string.h>, which takes the error number like *EISDIR* as an argument and returns to you a C-style string like “*Is a directory*”. The string is owned by the C library and should never be deallocated.

The *open()* system call attempts to read up to *count* bytes from the file specified by *fd* into the buffer starting at *buf*. If the call succeed, the number of bytes read is returned. The file position is advanced by this number, so the next read will not read the same file position. It is possible for the returned number to be smaller than the requested number of bytes to read, e.g., when end of file is encountered, or reading further bytes from an I/O device would require waiting.

The *write()* system call writes up to *count* bytes of the buffer starting at *buf* to the file specified by *fd*. All precautions of *read()* also holds for *write()*.

After finishing using a file, a program should use *close()* to free up system resources allocated by the OS. If no error is detected, it returns 0. The following program shows an example, which perform a file copy.

```
#include <unistd.h>
```

```
#include <sys/fcntl.h>
#include <stdio.h>

#define BUF_SIZE 1024
char buffer[BUF_SIZE];
int sfile, dfile;

int main(int argc, char **argv) {
    int byteRead;

    if (argc < 3) { /* check whether enough argument */
        fprintf(stderr, "Not enough arguments!\n");
        fprintf(stderr, "Usage: %s source_file destination_file\n", argv[0]);
        return 0;
    }
    sfile = open(argv[1], O_RDONLY);
    if (sfile == -1) {
        fprintf(stderr, "Cannot open file %s\n", argv[1]);
        return 1;
    }
    dfile = open(argv[2], O_WRONLY|O_CREAT);
    if (dfile == -1) {
        fprintf(stderr, "Cannot open file %s\n", argv[2]);
        close(sfile);
        return 1;
    }
    while ((byteRead = read(sfile, buffer, BUF_SIZE)) > 0)
        write(dfile, buffer, byteRead);
    close(dfile);
    close(sfile);
    return 0;
}
```

If you run the program in Unix and debug it, you will notice that `sfile` and `dfile` are allocated numbers 3 and 4 respectively. This is because 0, 1 and 2 are used for the standard input, standard output and standard error respectively.

2. Memory mapping: `mmap`, `munmap`

Files can be mapped to memory, so that reading files are done by reading the memory. This is done efficiently: the file is actually read only when needed.

Interface

```
void *mmap(void *start, size_t length, int prot, int flags, int
fd, off_t offset)
int munmap(void *start, size_t length)
```

Include files

```
<unistd.h>, <sys/mman.h>
<unistd.h>, <sys/mman.h>
```

The `mmap()` system call maps `length` bytes from the file specified by `fd`, starting at file position `offset`. If `start` is not `NULL`, the OS will map it into the memory starting at `start` if possible; otherwise the address is chosen by the OS. The `munmap()` system call remove such a mapping.

The desired memory protection is described by *prot*. *PROT_EXEC* specifies that the memory can contain code to be executed, *PROT_READ* allows the memory to be read, *PROT_WRITE* allows the memory to be written to. As usual, these flags can be combined using bitwise-or. If none is intended, one can use *PROT_NONE*.

All mappings are either private or shared, set by the *flags* argument by using *MAP_PRIVATE* or *MAP_SHARED*. This is meaningful when two or more processes have writable mapping of the same file. For a private mapping, such writes are private to the individual processes, and the writes are discarded without changing the files. For a shared mapping, such writes affect all processes, and the file is modified accordingly.

One can also use *mmap* to create memory regions that is not associated with any file, by adding the flag *MAP_ANON*. In this case, the *fd* and *offset* arguments are not used. This allows memory to be allocated outside the normal heap, and provides an alternative to the shared memory mechanism described later in this note.

We say that system calls generally use -1 to signify errors. The *mmap* system call is no exception. However, since it normally returns a pointer, the returned value is (**void ***)-1.

3. Process management: *fork* and *_exit*

Interface	Include files
<i>pid_t</i> <i>fork</i> (void)	< <i>unistd.h</i> >
void <i>_exit</i> (int <i>status</i>)	< <i>unistd.h</i> >

fork() creates a new process, which have a different PID (process ID) and PPID (parent process ID). The process calling *fork*() is said to be the parent process, and the process created by *fork*() is said to be the child. On success, 0 is returned for the child process, and the child PID is returned to the parent process. On failure, -1 is returned.

_exit() terminates a process. This is usually the last system call made by a process, since the process will be collected by the OS immediately afterwards. There is a function *exit*() without the underscore in the C library, which performs its own cleanup before calling *_exit*(). The parent process will receive a notification about the termination of the child process, which will receive *status* specified by the child.

4. Shared memory manipulation: *shmget*, *shmat*, *shmctl*

Processes can communicate by sharing a common piece of memory. There are a few ways to set this up in Linux. For example, *mmap* can be used. However, by far the most flexible method is to the one defined by System V, a version of Unix. This interface should be considered portable, since nowadays both BSD and System V Unix systems provide these calls.

Interface	Include files
int <i>shmget</i> (<i>key_t</i> <i>key</i> , int <i>size</i> , int <i>shmflg</i>)	< <i>sys/ipc.h</i> >, < <i>sys/shm.h</i> >
void <i>*shmat</i> (int <i>shmid</i> , const void <i>*shmaddr</i> , int <i>shmflg</i>)	< <i>sys/types.h</i> >, < <i>sys/shm.h</i> >
int <i>shmdt</i> (const void <i>*shmaddr</i>)	< <i>sys/types.h</i> >, < <i>sys/shm.h</i> >
int <i>shmctl</i> (int <i>shmid</i> , int <i>cmd</i> , struct <i>shmid_ds</i> <i>*buf</i>)	< <i>sys/ipc.h</i> >, < <i>sys/shm.h</i> >

Within the OS, each shared memory allocated this way has an identifier, which is a system-wide

number that increment every time a new shared memory is allocated. Optionally, a unique key may be associated with it as well, which is an arbitrary integer chosen by individual programs. In practice, most program do not use a key, since it is impossible to ensure different programs use different keys. The `ipcs` command lists all the shared memory (and other IPC constructs) currently in the system.

To create a piece of shared memory, one calls `shmget` with `shmflg` set to `IPC_CREAT`. A *key* can be passed, which associates a fixed integer with the allocated memory. Most programs use the constant `IPC_PRIVATE` as the key, meaning that no key is needed. On the other hand, if one decides to use a key, it is desirable to use the flag `IPC_EXCL` as well, to flag an error if another program is using the same key. The returned value is the identifier, which can be used as the *shmid* argument of the other system calls.

It requires some methods for another program to retrieve the identifier before the programs can communicate using the shared memory. It may be done by the filesystem, for example, by writing *shmid* to a file for another to read. If a program uses a fixed key, `shmget` (without `IPC_CREAT`) can be used as well.

After the identifier of a shared memory is obtained, a program can map the shared memory into its own address space using `shmat()`. It returns the starting address of the memory used to access the shared memory. From then on, one can use the shared memory just like those dynamically allocated by `malloc()`. To unmap the shared memory, one uses `shmdt()`.

Both `shmget()` and `shmat()` takes an argument `shmflg`, which allows permission of the memory to be specified. For example, `SHM_R` allows the memory to be read, and `SHM_W` allows the memory to be written. To obtain or change the permission, or other information, of a piece of shared memory, one can use `shmctl()`. To obtain the information, one set `cmd` to `IPC_STAT`. To change the information, one set `cmd` to `IPC_SET`. The information is set via `buf`, which points to a data structure defined as follows:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation perms */
    int shm_segsz; /* size of segment (bytes) */
    time_t shm_atime; /* last attach time */
    time_t shm_dtime; /* last detach time */
    time_t shm_ctime; /* last change time */
    unsigned short shm_cpid; /* pid of creator */
    unsigned short shm_lpid; /* pid of last operator */
    short shm_nattch; /* no. of current attaches */
};
```

Finally, `shmctl()` is also used to remove a shared memory, by setting `cmd` to `IPC_RMID`. This should be done once the shared memory is known to be unneeded, to free up the memory of the OS.

5. Concluding notes

We have just seen a deeper descriptions of some key system calls of Linux. But remember that the manual pages are the primary source of information. Once you gain enough knowledge from this course, you must make sure that you can read information from there.