

# 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 learned that user level programs interact with the OS kernel through an application program interface (API), which consists of *system calls* for accessing OS functionalities. 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 those system calls discussed in the lecture. Please refer to the lecture notes to see working examples.

#### 1. Process management: *fork*, *\_exit* and *wait*

Interface	Include files
<code>pid_t fork(void)</code>	<code>&lt;unistd.h&gt;</code>
<code>void _exit(int status)</code>	<code>&lt;unistd.h&gt;</code>
<code>pid_t wait(int *status)</code>	<code>&lt;sys/types.h&gt;</code> , <code>&lt;sys/wait.h&gt;</code>
<code>pid_t waitpid(pid_t pid, int *status, int options)</code>	<code>&lt;sys/types.h&gt;</code> , <code>&lt;sys/wait.h&gt;</code>

*fork()* creates a new process. 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. It will get a new PID (process id), which will be returned to the parent process. (The child can call *getpid()* to find its own PID.)

*\_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 C library function *exit()* without the underscore, which performs its own cleanup before calling *\_exit()*. Most of the time you should use *exit()* instead of *\_exit()*. The *status* argument is a value to pass back to the parent process.

A process which executes *fork()* should arrange to call *wait* or *waitpid* some time later. This waits for the termination of a process (for *wait* any child, and for *waitpid*, the child with the specified PID), and retrieves the *status* value returned by the child. The OS can reuse the child PID only after *wait* or *waitpid* has been called.

#### 2. Shared memory manipulation: *shmget*, *shmat*, *shmctl*

Processes can communicate by sharing a common piece of memory with the “System V” shared memory construct. (The name is due to the first Unix system implementing this functionality.)

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

Within the OS, each piece of shared memory allocated this way has an identifier, which is a

number generated by the OS “randomly”. Optionally, a unique key may be associated with it as well, which is an integer chosen by individual programs.

To use shared memory, some process must create it, and all the communicating processes must get its identifier. Both can be done by *shmget*, which returns an identifier to the shared memory. A *key* is used to specify which shared memory is needed, and the identifier is returned. With *shmflg* set to *IPC\_CREAT*, a piece of shared memory will be created if it did not exist. In this case, the flag *IPC\_EXCL* can be added to *shmflg* if you want to make sure the shared memory with that key does not exist before, and the *key* can be specified as *IPC\_PRIVATE* if new shared memory with no key is desired. If no *key* is used, some other mechanism (e.g., the filesystem) must be used to communicate the identifier of the shared memory.

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()*.

Just like any OS objects allocated to users, shared memory allows users to specify whether another user is allowed to use the shared memory. This is done by “permission bits”, which are added to the *shmflg* when the shared memory is *created*. For now, always use the permission 0700 as in the lecture notes, which allows no other users to attach to the shared memory.

Finally, *shmctl()* can be 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. The actual removal will be delayed until all its mappings are detached.

After playing with shared memory for a while, the system might have some shared memory regions left uncollected. You can use the *ipcs* command to see the shared memory regions you have created, and use *ipcrm* to remove them manually. See their manpages for details.

### 3. 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
<b>int</b> <i>open</i> (const char * <i>pathname</i> , int <i>flags</i> )	<fcntl.h>, <sys/types.h>, <sys/stat.h>
ssize_t <i>read</i> (int <i>fd</i> , void * <i>buf</i> , size_t <i>count</i> )	<unistd.h>
ssize_t <i>write</i> (int <i>fd</i> , void * <i>buf</i> , size_t <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. File descriptors can then be used as the *fd* argument of the other system calls to identify the file. 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 and 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 `read()` system call attempts to read up to `count` bytes from the file specified by `fd` into the buffer starting at `buf`. If the call succeeds, the number of bytes read is returned. The file position is advanced by this number, so the next read will not read at 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 a program finishes using a file, it should use `close()` to free up resources allocated by the OS. This also happens automatically if the program terminates. If no error is detected, it returns 0.

The following program shows an example, which performs a file copy.

```
#include <unistd.h>
#include <sys/fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#define BUF_SIZE 4096

int main(int argc, char **argv) {
    char buffer[BUF_SIZE];
    int sfile, dfile, 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: %s\n", argv[1], strerror(errno));
        return 1;
    }
    dfile = open(argv[2], O_WRONLY|O_CREAT);
    if (dfile == -1) {
        fprintf(stderr, "Cannot open file %s: %s\n", argv[2], strerror(errno));
        return 1;
    }
    while ((byteRead = read(sfile, buffer, BUF_SIZE)) > 0)
        write(dfile, buffer, byteRead);
    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.

#### 4. 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*.

Other information are described by the *flag* argument. All mappings must have either *MAP\_PRIVATE* or *MAP\_SHARED* in the *flag*, specifying a private or shared mapping. A private mapping is, well, private: if you write on the mapped memory, only your own process see the change. In contrast, for a shared mapping, any write to the region is a file write, and is seen by all other processes.

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 above.

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.