

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 2

A more careful look to some Linux System calls

In the lecture, you 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 briefly in the lectures, and we will use these system calls during our upcoming tutorial. The primary 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
<i>pid_t fork</i> (void)	<unistd.h>
void _exit (int status)	<unistd.h>
<i>pid_t wait</i> (int *status)	<sys/types.h>, <sys/wait.h>
<i>pid_t waitpid</i> (<i>pid_t pid</i> , int *status , int options)	<sys/types.h>, <sys/wait.h>

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. **Each process has an unique PID**, and no process should get the PID 0. The return value for the child is thus not a PID. It is just an indication that the process is the child created by the *fork* function. (The child can call *getpid()* to find its own PID and *getppid()* to get the parent’s PID.) Once forked, the parent and child process will run **concurrently**.

_exit() terminates a process. If called, it is the last system call made by a process, since the process will be collected by the OS immediately afterwards (except that the PID cannot be reused, see *waitpid()* described below). 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, which can obtain it using the *wait()* system call.

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 its parent calls *wait()* or *waitpid()*.

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

The `open()` system call is used to “open a file”, which prepare the process for using the file named by `pathname`. A **file descriptor (FD)** is returned by `open()`, which is a small integer to be used as the `fd` argument of the other system calls to identify the file (We will soon see that there are other calls apart from `open()` that produce FDs). 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 this case, the `O_EXCL` flag can be used to specify that the system call should fail if the file already exists before.

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`, and the `write()` system call attempts to **write up to `count` bytes from the file**. If the call succeeds, the number of bytes read or written is returned. The file position is advanced by this number, so the next read or write will be perform at the next file position. It is possible for the returned number to be smaller than the requested number of bytes to read or write, e.g., when end of file is encountered during a read. Note also that `read()` and `write()` does **not** expect arguments to be C-style strings (i.e., null-terminated).

After a program finishes using a file, it should use `close()` to free up resources allocated by the OS. This 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) {
        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;
}
```

You can debug the program in Unix to find that *sfile* = 3 and *dfile* = 4. The FDs 0, 1 and 2 are normally the standard input, output and error devices respectively.

3. Memory mapping and shared memory: *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. It is more efficient than reading and writing through *read* and *write*, since no system call is needed once the mapping is established.

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. For a private mapping, if a process write on the mapped memory, only that process sees the change. For a shared mapping, memory writes are a file writes, and are 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_ANONYMOUS*. In this case, the *fd* and *offset* arguments are not used. This allows memory to be allocated outside the normal heap. This also provides a way for **multiple processes to share memory**—a common parent allocate memory using *mmap* with *MAP_SHARED* and *MAP_ANONYMOUS*, and then *fork()*; both the parent and the child will then share the same memory. If two processes do not have a common parent, they can share memory by mapping to an actual file, or by using a slightly more complicated interface (see *shm_open()* and *shm_unlink()* in Solaris documentation).

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.