Recitation 2: Processes and File I/O

Department of Computer Science and Engineering
University of Minnesota

June 16, 2025

Overview

- Process APIs: `fork()`, `exec()`, `wait()`
- File I/O APIs
- Directory APIs: `opendir()`, `readdir()`, `closedir()`
- Please check the man pages for more information on any APIs discussion: `man api`, `man 2 api`, and `man 3 api`, where `api` could be `fork`, `exec`, `read`, `write`
- Exercises
  - 1: Processes
  - 2: File I/O
  - 3: Directory

Spawn a Process(sample code: p1.c)

```
#include <sys/types.h>
#include <unistd.h>
// Creates a new process which is an exact copy of the calling process
pid_t fork();
// Return values:
// 0: currently in child process
// process id: currently in parent process
// -1: failed creating process
```

Executing an external program(sample code p2.c)

- Fork duplicates the exact code of calling process, instead we could replace it with a new image using `exec()` family.
- There are two variants. One where we already know the parameters ahead of time (`execl*`) and another where parameters are defined on the go or are variable (`execv*`). We will only cover four exec functions. Please refer to the man page of exec for more information.
- The first argument is always the program name in `execl*` and `execv*`
- `execl*`: Explicitly specify comma separated arguments list. Always end the argument list with a NULL.
  - ```
    int execl(const char *pathname, const char *arg, ..., (char *) NULL);
    ```
  - ```
    int execlp(const char *file, const char *arg, ..., (char *) NULL);
    ```

Executing an external program (cont.)

- `execv*`: Variable set of arguments could be provided. Always end the variable argument list with a NULL.
  - `int execv(const char *pathname, char *const argv[]);`
  - `int execvp(const char *file, char *const argv[]);`
- The `p` variant of `execl*` and `execv*` first searches for the program in the path mentioned in the PATH variable if the name doesn't have a '/'.
- All `exec()` returns -1 on failure.

Waiting for a process(sample code: p3.c and p4.c)

```
// Once the parent process spawns a child, it could wait for the child to
// complete execution. There are two APIs to do it.
#include <sys/types.h>
#include <sys/wait.h>
// wait for any one child
pid_t wait(int *wstatus);
// wait for a specific child
pid_t waitpid(pid_t pid, int *wstatus, int options);
// wstatus and options are usually set to NULL and 0 respectively
// pid = -1: wait for any child process == wait(NULL)
```

File I/O System calls(sample code: p4.c)

- System level file operations for manipulating files.
- Child process inherits file descriptors (fds) from parent and its shared, meaning, the movement of fds will be same across processes.
  - `int open(const char *pathname, int flags);`
  - `int open(const char *pathname, int flags, mode_t mode);`
- `pathname`: filename
- `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, `O_APPEND`
- `mode`: Used with `O_CREAT` to set the permissions of the file: `S_IRWXU`, `S_IRUSR`, `S_IWUSR`, `S_IXUSR`. `R`, read, `W`: write, `X`: execute, `U/USR`: user level.

File I/O System calls (cont.)

- `ssize_t read(int fd, void *buf, size_t count);`, reads at most `count` bytes data from the file associated with `fd` into the buffer `buf`.
- `ssize_t write(int fd, const void *buf, size_t count);`, writes at most `count` `count` bytes data from the buffer `buf` to the file associated with `fd`.
- `int close(int fd);`, closes file descriptor.
- `read()` and `write()` return the number of bytes read or written, or -1 on error.

Seeking System File I/O

```
// File descriptor could be moved to a desired location within the file
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
// whence can be SEEK_SET, SEEK_CUR, SEEK_END
// SEEK_SET: The file offset is set to offset bytes.
// SEEK_CUR: The file offset is set to its current
// location plus offset bytes.
// SEEK_END: The file offset is set to the size of
// the file plus offset bytes.
```

File I/O - Stardard I/O(sample code: p5.c)

- Utilizes file pointers (`FILE *`) instead of file descriptors (internally file descriptors are
- `FILE *fopen(const char *pathname, const char *mode);`
- `mode`: r, w,a (read, write, append). r+ (r & w), w+ (r & w, new file opened, if it doesn't exist; existing file truncated), a+ (r & w, new file opened , if it doesn't exists)
- `int fclose(FILE *stream);`, Closes the file stream and returns 0 on success. used)

File I/O - Stardard I/O(cont.)

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`, read `nmemb` items of datasize `size` to `ptr` from the file.
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)` write `nmemb` items of datasize `size` from `ptr` to the file
- `fread` and `fwrite` returns the number of items read and written respectively on success, 0 on failure.

Seeking Standard File I/O

```
// File pointer could be moved to a desired location within the file
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
// whence: file offset setter
// SEEK_SET: The file offset is set to offset bytes.
// SEEK_CUR: The file offset is set to its current location plus offset by
// SEEK_END: The file offset is set to the size of the file plus offset by
// Also, check ftell and rewind in man page of fseek
```

Directory Operations(sample code: p6.c)

```
// Directory structures could be access using a set of APIs
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
// Opens a directory stream and returns a pointer to it
// Returns a NULL on error
```

Directory Operations(cont.)

```
struct dirent *readdir(DIR *dirp);
// Read directory entries one at a time. The returned
// dirent structure has field to uniquely identify each entry
struct dirent {
ino_t d_ino;
/* Inode number */
off_t d_off;
/* Not an offset; see below */
unsigned short d_reclen; /* Length of this record */
unsigned char d_type;
/* Type of file; not supported by all filesystem
types */
char d_name[256]; /* Null-terminated filename */
};
// dtype : DT_DIR: directory, DT_REG: file
```

Directory Operations(cont.)

- `int closedir(DIR *dirp);`, Closes the directory stream and returns 0 on success. -1 on failure.

Exercesise

- In samples/p3.c, we saw the creation of a chain of processes. For this exercise, you will complete **fanout.c** to create a fan-out, i.e., there will be 'n'child processes spawned by a single main processes. Please refer to **fanout.c** for an example and the expected output.
- Complete the code provided in **chain.c** to create a chain of 'n'child processes that write their pid to a file **pids.txt**. samples/p3.c and samples/p4.c or samples/p5.c has required codes to complete the exercise. You may use system file I/O or standard file I/O
- Complete the **traverse.c** code to traverse the given 'nestdir'folder recursively and report the names files and directories. Refer to samples/p6.c for a single level directory traversal.
- Finally, once you have completed all the above exercises, complete **callall.c**, where twp child processes will call fanout.o and traverse.o using exec variants mentioned in the code.

Deliverables

- Submit the **tar.gz** exercises folder to Canvas by Jun 20th, 11:59 pm.
  - fanout.c
  - chain.c, pids,txt
  - traverse.c, nestdir
  - callall.c
  - Makefile