Operating Systems – spring 2022
Assignment 7

Instructor: Hans P. Reiser

Submission Deadline: Monday, March 7th, 2022 – 23:59

A new assignment will be published every week, right after (or a bit before) the last one was due. It must be completed before its submission deadline. (Hard deadlines, no extension.)

T-Questions are theory homework assignments. **The answers to the assignments must be uploaded to Canvas (in the quiz).**

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Submission instructions can also be found in first assignment.

The topic of this assignment is interprocess communication.

T-Question 7.1: Interprocess Communication

a. What are the two fundamental models of interprocess communication? Give a short explanation for each. **2 T-pt**

b. Which of the two IPC mechanisms (shared memory, messages passing) is usually easier to use for an application developer? Justify your answer! **2 T-pt**

c. Read the Linux manual page of the `futex` system call and answer the following questions: **3 T-pt**

- Is it a synchronization system calls that is mainly intended for implementing (1) spinlocks, (2) queue-based blocking locks, or (3) hybrid two-phase locks that want to to put a thread to sleep if longer waiting time is expected?
- Is it a synchronization mechanism that is suitable (1) only for threads of a single process, (2) only for synchronization across separated processes, or (3) for both (threads of a single process, multiple processes)?
- Is it a system call that enables non-blocking synchronization? Justify (briefly, one sentence is sufficient.)

d. (Anonymous) pipes are an IPC mechanisms in Linux (see `pipe` system call). How do you classify this IPC mechanisms regarding (1) shared memory or message passing? (2a) if shared memory: using synchronization with spin locks, with blocking locks (waiting queues), or non-blocking synchronization? (2b) if message passing: direct messages or indirect messages (mailbox)? Without buffering (zero capacity) or with buffering (bounded capacity)? **3 T-pt**

P-Question 7.1: Pipes

Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `pipe.c`.

In assignment 2 you wrote a simple program starter that used the `fork`, `exec` and `waitpid` system calls to start a program and wait for its exit. Your solution had to return the exit status of the new process or the special value 127 to indicate error conditions in your own program. However, the solution could not properly detect if the `fork`ed child failed to `exec` or if the started program normally exited with status code 127.

In this question you will extend the program starter to receive proper error information.

a. Complete the function `run_program` so that the `fork`ed child transmits the error number `errno` to the parent (the program starter), if the call to `exec` fails. Your function should fulfill the following requirements: **4 P-pt**

- Returns -1 on any error; the exit status of the child process otherwise.
- Allows the caller to read the correct error number from `errno`, as explained in the following items:
- Uses the `pipe2` system call to create a new pipe and sets the necessary flags to avoid leakage of the pipe's file descriptors into the `exec`ed program. (Note: unlike the basic `pipe` system call, `pipe2` has an additonal argument with flags that can be used – among other things – to indicate that the file descripter should automatically be closed upon an `exec` system call.)
- Uses the `write`/`read` system calls to write to and read from the pipe, transmitting – on error – the error number `errno` of `exec` from the child to the parent. The parent shall then set its `errno` to the received value.
- Closes unnecessary pipe endings in the parent and child respectively as soon as possible and fully closes the pipe before returning to the caller. **Do not leak file descriptors!**

Hints: Assume that the `write` and `read` system calls do not fail and ignore errors from the `close` system call. The template only marks the most important locations that need to be extended, you need to add further code to fulfill all requirements.

```
int run_program(char *file_path, char *argv[]);
```

Additional note: To keep things simple, the template version assumes that the program name is included in `argv` by the caller, so unlike the sample solution of assignment 2, the template does not add the program name at the beginning of `argv`.

P-Question 7.2: Message Queues

Download the template **p2** for this assignment from Canvas. You may only modify and upload the file `message_queue.c`.

In this question you will write a simple client-/server application using two processes and a mailbox (known as message queue in Linux) for communication. The server accepts commands from the client and performs corresponding actions. The message format is defined in the template by the `Message` structure, the available commands are defined in the `Command` enumeration.

a. Implement the server in the `runServer` function. The server should adhere to the following criteria: **6 P-pt**

- Returns -1 on any error, 0 otherwise.
- Creates and initializes a new message queue, which takes `Message` structures and provides space for 10 messages, using the `mq_open` call and appropriate flags for read-only access and `QUEUE_NAME` as name.
- Fails if the message queue already exists.
- Receives messages from the client and processes them until an exit condition is met.
- The template already contains an implementation of the following commands:

  CmdExit Exits the server

  CmdAdd Adds the two parameters supplied in the message and prints the result with the `FORMAT_STRING_ADD` format string.

  CmdSubtract Subtracts the second message parameter from the first one and prints the result with the `FORMAT_STRING_SUBTRACT` format string.
- Exists on error or on reception of an unknown command.
- Closes the message queue on exit and unlinks it

```
int runServer();
```

Implement the client functions. Assume the message queue to be already created by the server and ready to be opened for write access by the client.

```
mqd_t startClient();
int sendExitTask(mqd_t client);
int sendAddTask(mqd_t client, int operand1, int operand2);
int sendSubtractTask(mqd_t client, int operand1, int operand2);
int stopClient(mqd_t client);
```

Total:
10 T-pt
10 P-pt