Operating Systems – spring 2023
Tutorial-Assignment 8

Instructor: Hans P. Reiser

Submission Deadline: Monday, March 6th, 2023 – 23:59

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

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

The topic of this assignment is interprocess communication.

Question 8.1: Interprocess Communication (IPC)

a. How can concurrent activities (threads, processes) interact in a (local) system?

b. Asynchronous `send` operations require a buffer for sent but not yet received messages. Discuss possible locations for this message buffer and evaluate them.

Question 8.2: Race Conditions

a. Explain the term *race condition* with this scenario: Two people try to access a bank account simultaneously. One person tries to deposit 100 Euros, while another wants to withdraw 50 Euros. These actions trigger two update operations in a central bank system. Both operations run in "parallel" on the same computer, each represented by a single thread executing the following code:

```
current = get_balance();
current += delta;
set_balance(current);
```

where `delta` is either 100 or -50 in our example.

b. Determine the lower and upper bounds of the final value of the shared variable `tally` as printed in the following program:

```
const int N = 50;
int tally;


void total ()
{
    for( int i = 0; i < N; ++i )
        tally += 1;
}
```

```
int main ()
{
    tally = 0;

    #pragma omp parallel for
    for( int i = 0; i < 2; ++i )
        total();

    printf( "%d\n", tally );
    return 0;
}
```

Note: The #pragma tells the compiler to create code to run parallel threads for each loop step. Assume that threads can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

c. Suppose that an arbitrary number $t > 2$ of parallel threads are performing the above procedure `total`. What (if any) influence does the value of t have on the range of the final values of `tally`?

d. Finally consider a modified `total` routine:

```
void total ()
{
    for( int i = 0; i < N; ++i )
    {
        tally += 1;
        sched_yield();
    }
}
```

What will be printed in the one-to-one model, when a voluntary yield is added?

Question 8.3: Critical Sections

a. Explain the term *critical section*.

b. Explain the two core requirements that a valid synchronization primitive for critical sections as to fulfill. Furthermore explain additional desirable properties.

c. Recap the banking example from the previous question. How could the race condition be avoided?

Question 8.4: Pipes

a. Use cases for (anonymous) pipes: The `pipe` system call creates a pipe (in form of a pair of file descriptors) within the same process. For what kind of applications can you imagine that such an IPC channel can be useful? Do you know any application that you have recently used and that uses this IPC mechanism?

b. In assignment 6, you may have used the command line command `(valgrind --tool=lackey --trace-mem=yes ls -la 2>&1 1>/dev/null) | python transform-lackey.py` Can you explain this command line, and how it relates to the IPC mechanism of pipes?

c. Write a C program for Linux that creates two child processes, `ls` and `less`, and uses an ordinary pipe to redirect the standard output of `ls` to the standard input of `less`.

T-Question 8.1: Interprocess Communication

a. The two fundamental models of interprocess communication are shared memory and message passing. Which of the two models is usually easier to use for an application developer? Justify your answer! **2 T-pt**

b. Which of the two models of interprocess communication usually does not require using system calls for exchanging data between two processes? Briefly explain why! **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 system calls to perform atomic instructions, for example for atomically changing a lock state from not acquired to acquired?
- Is it a system call that is useful for implementing hybrid two-phase locks, as it can be used to put a thread to sleep if longer waiting time is expected?
- Is it a system call that enables non-blocking synchronization?

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 8.1: 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.

MacOS hint: The `pipe2` system call is not available on MacOS. Instead, you could use the `pipe` system call combined with `fcntl` system calls for each of the file descriptors to set the close on exec flag (this works on MacOS and Linux). We accept both version, but make sure the code you submit compiles and runs on Linux.

```
int run_program(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 simply uses argv[0] as the program name.

P-Question 8.2: 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