Operating Systems – spring 2024
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.

Lab 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.

Lab 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?

Lab 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?

Lab 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. If you run this command in a Linux shell `cat Makefile | wc`
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. Assume you have two processes:

- a computation process that works on a CPU-heavy, long-running computation (this process knows about the progress of computation in percentage of completing the work).
- a GUI process that shows the progress of the computation to the user (updating the information once per second).

You – as a developer – have the choice to use either anonymous pipes or shared memory for the necessary inter-process communication. Is there a strong reason for preferring one over the other choice for this sample application? Justify your answer! **2 T-pt**

b. Consider a scenario where various applications, such as a terminal program, a GUI program, and/or a web service, are used to show the progress of the computation. How would this scenario influence the choice of an IPC mechanism? You want to have a simple approach that facilitates the dynamic start or stop of any application showing the computation progress. Which approach would you choose? Make a clear single choice out of the two options in part (a) and justify your choice! **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. Which of the following statements on IPC (shared-memory or message passing) are correct? **3 T-pt**

correct	incorrect	
☐	☐	Send operations are always non-blocking if buffering is used.
☐	☐	The smallest possible size of a shared memory segment depends on hardware characteristics.
☐	☐	An anonymous pipe in Linux is bidirectional.
☐	☐	Anonymous pipes in Linux do not use buffering (“zero-capacity”).
☐	☐	Anonymous pipes in Linux can be classified as IPC mechanism with indirect messages (“mailbox”).
☐	☐	`shm_open`, `shmget`, and `shm_close` are system calls used for IPC using message passing

.

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. The template `pipe.c` is a very similar program.

In this assignment you will extend this program to capture the output (on STDOUT) of the program you execute with `exec`.

a. Complete the function `get_output` to capture the output of the program started by `exec`. Your function should fulfill the following requirements: **4 P-pt**

- Returns NULL on any error;
- Return a copy of the first line of the output of the program, in a memory region allocated on the heap with `malloc`. A line is a string up to the first newline ('\n') character.
- You may truncate the first line if it is longer than 1024 bytes.
- You should assume that the caller of `get_output` later calls `free` for this memory.
- Close unnecessary pipe endings in the parent and child respectively as soon as possible and fully closes the pipe before returning to the caller.
- Note: You will need to use the `dup2` system call to replace the standard output file descriptor (STDOUT_FILENO) with one end of your pipe before executing `execvp`.
- One (secret) test will be running a program that produces a lot of output (like running "cat trace-stydemo.txt").

```
int get_output(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.

  CmdMul Multiplies the two parameters supplied in the message and prints the result with the `FORMAT_STRING_MUL` 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 sendMulTask(mqd_t client, int operand1, int operand2);
int stopClient(mqd_t client);
```

Total:
10 T-pt
10 P-pt