# Operating Systems

## WiSe 2015/2016

## — Lab Assignment 1 —

Linux Processes, Threads and IPC

— Submission Email —

os-lab@deeds.informatik.tu-darmstadt.de

— Course Website —

www.deeds.informatik.tu-darmstadt.de/teaching/courses/ws20152016/operating-systems/news/

| | |
|---|---|
| Publication: | Mon, Oct 26, 2015 |
| Submission: | Tue, Nov 10, 2015 before 23:59 |
| Testing: | Fri, Nov 13, 2015 (room A213) |

# Preface

The purpose of this lab is to get familiar with the Linux development environment and the basics of process & thread management as well as different inter-process communication (IPC) mechanisms. Beside a brief introduction to processes and threads, we will cover a subset of the IPC mechanisms that are presented in the lecture, namely pipes, message queues and shared memory.

This lab contains multiple sections covering the different topics. Each section contains both programming tasks and questions that have to be answered. Although the programming tasks build upon each other, you have to clearly separate them is your solution by providing distinct source files for each programming task. You have to use `make` for compiling your code, i.e., you need to *supply appropriate Makefiles* for that purpose. After compiling your code, we want to have one separate executable that implements your solution for each programming task. For answering the stated questions, create a *plain text file* and write down your answers. ***Do not use other file formats such as Office formats or PDF!***

You should do the lab on your own machine. Make sure that you use a recent Linux distribution that includes recent (but not necessarily the latest) versions of the Linux kernel and the required build tools. Most common distributions allow for easy installation and upgrade/downgrade of the required tools. The documentation and examples provided in this lab were tested and verified in our testing environment (Debian 8.2) using a *Linux 3.16* kernel, *GCC 4.9*, and *GNU Make 4.0*. We recommend that you use a similar setup to ensure that your solutions work properly in our testing environment. ***If you use different kernel or tool versions, it is your responsibility to check for compatibility with our testing environment!***

Make sure you send your lab solution as tar archive via email to the address stated on the cover page before the deadline. You have to work in groups of 4 to 5 students. Do not submit individual solutions but send one email per group! ***In your submission email, include the names, matriculation numbers and email addresses of ALL group members.*** The testing date (see cover page) is mandatory for all group members. Therefore, ***indicate collisions with other TU courses on the testing date*** so that we can assign you a suitable time slot. Assigned time slots are fixed and cannot be changed. Students that do not appear for the assigned testing time, will get no bonus points for that lab. During the test, we discuss details of your solutions with you to verify that you are the original authors and have a good understanding of your code. The amount of bonus points you achieve depends on your performance in the testing session.

Please regularly check the course website as we publish up-to-date information and further updates there.

Questions regarding the lab can be sent to: os-lab@deeds.informatik.tu-darmstadt.de

Good Luck!

# General Advice

For the programming tasks, you have to use the *C programming language*, the *GCC compiler*, and the *Make* build tool. We expect that you not only provide a *working and robust solution* to each of the tasks, but also adhere to *a consistent, structured, clean, and documented coding style*. Keep your code simple and as close to the tasks as possible. Do not implement more than what is asked.

For all programming tasks, you have to *implement proper error handling* mechanisms in your solution. Function calls can fail and their failure must be handled properly. The failure of a function call is usually indicated by its return value being set to a magic value (e.g. `-1` or `NULL`), and often `errno`, a global variable holding additional error information, is set. You can query this information using the `perror` or `strerror` functions – please make use of these or similar mechanisms.

Make sure that your code compiles without any warnings by using the `-Wall` compiler switch during compilation (e.g., `gcc -Wall -o executable source.c`). If your code produces warnings during the testing session, we expect that you are able to explain each and every warning and justify why you did not fix it.

Linux systems provide an extensive online manual, the so-called man pages. Please make extensive use of this feature when you work on the tasks of this lab as we only provide a very brief description of some important functions and tools. You can access the man pages with the following command line:

```
man [section] <command-name>
```

Alternatively, there are also online versions of the man pages available, e.g., at

https://www.kernel.org/doc/man-pages

But be aware that online versions may not be complete compared to a local installation.

Section 3 refers to C library functions and section 2 to system calls. These two should be the most interesting sections for you. The usage of section numbers is optional, but may be necessary when identical command names exist in different sections. Also, do not hesitate to search for additional information on the web or at the ULB Darmstadt.

For building your solution for each task, you have to use the `make` tool and write appropriate Makefiles instead of directly invoking the compiler. You may choose to write a separate Makefile for each programming task, or you may write one Makefile that can be used for building all the solutions. If you do not know how to use the make tool, have a look at its man pages. You may also have a look at online tutorials such as the following ones.

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

http://mrbook.org/blog/tutorials/make/

# 1) Process Management

Process management includes the creation of new processes, program execution, and process termination. It is not directly related to IPC, but in order to create processes that communicate with each other, you need to understand some process management basics.

A process can create a new child process by calling the `fork` function, see `man 2 fork`.

```
#include <unistd.h>

// returns 0 in child, process ID of child in parent, -1 on error
pid_t fork(void);
```

The `fork` function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. Both the child and the parent continue their execution with the instruction that follows the `fork` call. The child is a copy of the parent, i.e., the child gets a copy of the parent's data (heap and stack) and they share the same text segment.

A parent process can wait for the termination of its child processes by calling the `wait` or `waitpid` function. In this lab, we only consider `wait`, as it has simpler semantics and is sufficient for our application, see `man 2 wait`.

```
#include <sys/types.h>
#include <sys/wait.h>

// returns process ID if OK, -1 on error
pid_t wait(int *status);
```

The `wait` function blocks the caller until any child process terminates. If a child has already terminated, `wait` returns immediately with that child's status. The target integer of the `status` pointer stores the termination status of the child process, but by passing a `NULL` pointer as argument, this information can be ignored.

### Task 1.1

> Implement a program that spawns a child process and waits for its child to finish execution. Make use of the `fork` and `wait` functions. After spawning the child, each process has to print a text message that states its own process id (PID) and whether it is the parent or child process. Moreover, the parent has to print messages indicating the creation and termination of the child including the child's process id. Print all messages to `stderr` rather than `stdout` (`e.g. use fprintf(stderr, ...)`)!
>
> Use the `usleep` function to artificially slow down your program. Have a look at `man 3 usleep` to learn how to use the function. Add a 150 ms (ms = millisecond) delay right after the `fork` call to slow down both the parent and the child process. Add another 500 ms delay in the child process after printing the child message to prolong its execution time.
>
> Extend your program with a global variable `int my_value` that is initialized to `42`. In the child process, change `my_value` to `18951` before the child prints any messages. Both the parent and child have to print `my_value` along with the message that states their PID and the parent has to print it in the child termination message as well.

> Do not forget to implement proper error checking as this is very important for real world programs. You have to at least print error messages and terminate the program with a proper exit code!

### Question 1.1

> Execute your program from task 1.1 at least 100 times. Is the output of your program always exactly the same? Which value does the parent process print for `my_value`? What does the child print? Explain your observations.

# 2) Multithreading with POSIX Threads

Using multiple threads is often considered more efficient than using multiple processes. In Linux, using the POSIX threads library is a common choice for implementing multithreaded applications. Have a look at `man 7 pthreads` for an overview of the library.

For this lab, the most important library functions are as follows. See `man 3 <function name>` for detailed explanations for each function.

```
#include <pthread.h>

// thread creation; returns 0 on success, error number on error
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

// terminate calling thread with retval as exit value
void pthread_exit(void *retval);

// wait for the termination of the specified thread
// returns 0 on success, error number on error
int pthread_join(pthread_t thread, void **retval);
```

New threads within a process are created with the `pthread_create` function, which takes a pointer to the function that should be executed in the new thread (`start_routine`). For this lab, you can stick to the following code example for thread creation, termination and joining. However, make sure to understand how the code works and what the individual functions do.

```
void* my_thread_function(void *arg) {
    ...
    pthread_exit(0);
}
...
pthread_t my_thread;
err = pthread_create(&my_thread, NULL, my_thread_function, NULL);
...
err = pthread_join(my_thread, (void**)&ptr_to_thread_ret);
...
```

In order to build programs that use the pthreads library, you have to link your program accordingly. An example invocation of GCC for a program that uses pthreads may look like the following.

```
gcc -Wall program.c –lpthread
```

### Task 2.1

> Modify your program from task 1.1 to use multi-threading instead of multi-processing.

### Question 2.1

> Execute your program from task 2.1 at least 100 times. Is the output of your program always exactly the same? Which values do the two threads print for `my_value`? Do you spot a difference compared to your program from task 1.1? Explain your observations.

## 3) IPC with Pipes

Pipes are the oldest form of UNIX System IPC. Pipes are communication channels between two processes and usually operate in a half-duplex mode, i.e., data flows in only one direction at a time. Pipes can only be shared between processes that have a common ancestor. A common usage pattern for pipes is that a pipe is created by a process via the `pipe` function, then the process calls `fork`, and the pipe is used between the parent and the child. See `man 2 pipe` for details on the function.

```
#include <unistd.h>

// returns 0 if OK, -1 on error
int pipe(int filedes[2]);
```

The function returns two file descriptors through the `filedes` parameter: `filedes[0]` is open for reading, `filedes[1]` is open for writing. How the file descriptors are used after the fork depends on the desired data flow direction. For a pipe from the parent to the child, the parent closes the read end of the pipe (`filedes[0]`), and the child closes the write end (`filedes[1]`). The pipe's file descriptors are used like a file, i.e., the same functions that are used for handling file I/O (`read`, `write`, `close`) also apply for pipes. See `man 2 {read, write, close}` for details.

### Task 3.1

> Modify your program from task 1.1 so that it uses pipe-based IPC to pass data from the parent to the child. The parent has to send the following string to the child process via the pipe: **"**`Hi, I am your parent. My PID=%d and my_value=%d`**"**. Before sending the string to the child, the parent first has to replace the first `%d` with its own process id and the second `%d` with its value for `my_value`. Use the commonly used string and print functions (e.g. see `man 3 sprintf`). Do not forget that strings in C are `NULL` terminated.
>
> After receiving the parent's string, the child has to print the message as last message after its 500 ms sleep and before it finishes execution.

### Question 3.1

> Execute your program from task 3.1 at least 100 times. Is the output always the same? What is printed for `my_value`? Explain your observations.

### Question 3.2

> Can you image reasons for and against using pipes for communication within multi-threaded software?

# 4) IPC with POSIX Message Queues

POSIX message queues allow processes to exchange data in the form of messages. A message queue is a linked list of messages stored within the kernel and identified by a message queue descriptor. Message queues are created and opened using `mq_open`, which returns the message queue descriptor for the opened message queue. Each message queue needs a unique name of the form `/MessageQueueName` – a `NULL`-terminated string with a leading slash and no subsequent slashes. After opening a message queue, the sending process can add messages to the queue using `mq_send`. The message can then be read by another process via `mq_receive`. When a process has finished using the queue, it closes it using `mq_close`, and when the queue is no longer required, it has to be deleted with `mq_unlink`.

For this lab, the most important functions are listed below. See `man 3 <function name>` for detailed explanations for each function; see `man 7 mq_overview` for a general overview.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

// create/open queue; returns queue descriptor, -1 on error
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);

// send message to queue; returns 0 if OK, -1 on error
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);

// receive message from queue
// returns length of received message, -1 on error
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);

// get queue attributes; returns 0 if OK, -1 on error
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

// close queue; returns: 0 if OK, -1 on error
int mq_close(mqd_t mqdes);

// remove queue; returns 0 if OK, -1 on error
int mq_unlink(const char *name);
```

The function `mq_open` returns a message queue descriptor (MQD), which is used for accessing the queue. The input parameters are: a unique queue `name` (identifier), the operation flags `oflag` (specifies read or write operation, creation options, etc.), the `mode` (permissions of a newly created queue), and an optional message queue attribute `attr`.

Message queues are usually used according to one of the following approaches. The first is similar pipes approach from above where `mq_open` is called before the fork and the MQD is shared among the parent and child process. The second option is to share the name of the message queue (e.g. `/MyMessageQueue`) and call `mq_open` in the parent and the child. Here

the parent would create the queue and the child would only open it. Preferable option is the second one as it allows processes of different ancestors to access the same queue.

The `mq_send` function adds the message pointed to by the `msg_ptr` to the queue specified by `mqdes`. The `msg_len` parameter specifies the length of the message in bytes. The `msg_prio` parameter specifies the message priority (placement within queue).

The `mq_receive` function is used to receive the oldest message from the queue specified by `mqdes`. If the size of the buffer in bytes specified by the `msg_len` is less than the maximum possible message size of the queue (`mq_msgsize` attribute of the queue), the function fails and returns an error. Otherwise, the received message is removed from the queue and copied to the buffer pointed to by `msg_ptr`. You can use the `mq_getattr` function to obtain the maximum message size (see man page) and the `malloc` function to allocate the amount of buffer space needed. Do not forget to de-allocate the buffer using `free` when it is not needed any more.

When a process has finished using the queue (i.e., it has successfully sent or received its message(s)), the process has to close the queue using `mq_close`. The last process to finish using a message queue has to take care of deleting it using `mq_unlink`. Else the message queue will remain within the system until reboot and subsequent calls to `mq_open` might fail or lead to unexpected behavior.

In order to build programs that use POSIX message queues, you have to link your program with the real-time extension library. An example invocation of GCC for a program that uses message queues may look like the following.

```
gcc -Wall program.c –lrt
```

## Task 4.1

> Modify your program from task 3.1 to use a POSIX message queue instead of pipes. Use `/DEEDS_lab1_mq` as the unique queue name. The parent has to create and delete the queue, whereas the child only opens and closes the queue (second option described above). Please ensure to unlink the message queue whenever your parent process finishes! Take special care to also include the unlinking in your error handling code to ensure a clean termination even in error cases!

## Question 4.1

> Execute your program at least 100 times. Do you observe significant differences compared to your program from task 3.1? In which scenarios would you use message queues, in which pipes?

## Question 4.2

> Do you see use cases or reasons for using message queues within a multi-threaded software?

# 5) IPC with POSIX Shared Memory

Shared memory allows processes to share a region of memory. This is the fastest form of IPC since the data does not need to be copied between processes. On the other hand, using shared memory requires explicit synchronization among processes to avoid inconsistencies caused by concurrent read/write access. However, in this lab, we simply ignore such synchronization issues and observe what happens.

The usage of POSIX shared memory is to some extent similar to that of POSIX message queues. Shared Memory Objects (SMOs) are created and opened using `shm_open`, which returns a file descriptor that is used to refer to the open SMO in subsequent calls. Each SMO is identified by a unique name of the form `/SharedMemName` – a null-terminated string with a leading slash and no subsequent slashes. After opening a SMO, the size of the SMO needs to be set via `ftruncate`. The functions `mmap` and `munmap` are used to map or unmap the shared memory object into the virtual address space of the calling process. Data is written to, and read from, the SMO by operating on the memory region addressed by the pointer returned by `mmap`. When a process has finished using the SMO, it closes it using `close`, and when the SMO is no longer required, it has to be deleted using `shm_unlink`.

For this lab, the most important functions are listed below. See `man 2/3 <function name>` for detailed explanations for each function; see `man 7 shm_overview` for a general overview.

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

// create/open SMO; returns SMO handle, -1 on error
int shm_open(const char *name, int oflag, mode_t mode);

// set SMO length; returns 0 if OK, -1 on error
int ftruncate(int fd, off_t length);

// map SMO into memory
// returns pointer to the mapped area, MAP_FAILED on error
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
           off_t offset);

// unmap SMO memory region; returns 0 if OK, -1 on error
int munmap(void *addr, size_t length);

// remove SMO; returns 0 if OK, -1 on error
int shm_unlink(const char *name);
```

The function `shm_open` needs the following parameters: a unique shared memory `name`, the operation flag `oflag` (specifies read or write operation, creation options, etc.), and the `mode` (permissions of a newly created SMO). After creation, a SMO has a length of zero, which needs to be adjusted using `ftruncate`.

Processes map the SMO into their virtual address space by calling `mmap`. The starting address for the new mapping is specified in `addr` and the `length` parameter specifies the length of the mapping. The `prot` parameter describes the desired memory protection for the mapping (and must not conflict with the open mode). The `flags` parameter determines whether updates to

the mapping are visible to other processes mapping the same region. Processes access the SMO through the pointer returned by `mmap`. When a process has finished using the SMO, it unmaps the SMO using `munmap`.

The last process to finish using a SMO has to take care of deleting it using `shm_unlink`. Else the SMO will remain within the system until reboot and subsequent calls to `shm_open` might fail or lead to unexpected behavior.

In order to build programs that use POSIX shared memory, you have to link your program with the real-time extension library. An example invocation of GCC for a program that uses message queues may look like the following.

```
gcc -Wall program.c –lrt
```

## Task 5.1

> Modify your program from task 4.1 to use POSIX shared memory instead of message queues. Use `/DEEDS_lab1_shm` as the unique SMO name. The parent has to create and delete the SMO, whereas the child only opens and closes the SMO. Please ensure to unlink the SMO whenever your parent process finishes! Take special care to also include the unlinking in your error handling code to ensure a clean termination even in error cases!
>
> Extend your program so that the child sends a similar string back to the parent right after receiving the parent's string. The parent has to print the string received from the child before waiting for the child's termination. Do not use additional synchronization mechanisms for this task!

## Question 5.1

> Execute your program from task 5.1 at least 100 times. Do you observe differences in the executions? Explain your observations. In which scenarios would you use shared memory objects, in which message queues?

## Question 5.2

> Do you see use cases or reasons for using SMOs within a multi-threaded software?