Operating Systems

WS 2014/2015

Lab 1 – Linux Inter-process Communication (IPC)

Submission deadline: November 18, 2014

Testing: November 21, 2014 in E302

Form groups of 2 to 3 students!

Send solution to: os-lab@deeds.informatik.tu-darmstadt.de

Preface

The purpose of this lab is to get familiar with different inter-process communication (IPC) mechanisms in Linux. We will cover a subset of the mechanisms that are presented during the lecture, namely pipes, message queues and shared memory. Furthermore, a short introduction to process management functionality is given.

If you do the lab on your own machine, make sure that you use a recent Linux distribution that includes recent versions of the Linux kernel and the required build tools. Most common distributions will allow you to make automatic updates to the latest version. The examples provided in the lab were all tested and verified on the latest stable version of the 3.16 kernel. Do not use 2.6.x or earlier kernel versions, as some of the mechanisms and interfaces presented in the labs may not be present or work differently with old Linux versions.

Make sure you send the solution of the lab via email to the address mentioned on the cover page before the deadline (**November 18, 2014**). You have to **work in groups of 2 to 3 people**, do **refrain from submitting solutions individually!** In your submission, **include the names, matriculation numbers and e-mail addresses of ALL group members.** Currently, we target to test your solutions to the lab problems on November 21 in seminar room E302. In your submission mail, **indicate possible collisions on that day with an exact timeslot**, so we can schedule you accordingly – elsewise you have to accept our assignment. During the test we will discuss details of your solutions with you to verify that you are the original author and understand your own code well. In case of updates, information will be made available during the lecture or exercise, as well as on the course web page.

Questions regarding the lab can be sent to:

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

Good Luck!

General Advice

This lab provides hands-on experience on the implementation and use of several inter-process communication (IPC) mechanisms in Linux. As such, you will be programming in the C programming language and we expect that you not only provide a working and robust solution to each of the problems, but also adhere to a consistent, structured, clean and documented coding style.
We also expect that you 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.
Furthermore, please make sure that your program compiles without any warnings by using the **`-Wall`** compiler switch during compilation (e.g., `gcc -Wall -o executable source.c`).

Linux provides its users with an extensive online manual, the so-called man pages. Please make extensive use of this feature when you try to solve the problems of this lab, as we will only provide you with a very brief description of each function. They are accessed through the shell by issuing the command:

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

Alternatively, there are also online version available, e.g., at
https://www.kernel.org/doc/man-pages

Section 3 refers to C library functions and section 2 to system calls. These two should be the sections that are most interesting to 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.

1 Process Management

Process management handles 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.

An existing process can create a new one by calling the `fork` function.

```
#include <unistd.h>

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

The new process created by fork is called the child process. 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 executing with the instruction that follows the call to fork, and the child is a copy of the parent. The child gets a copy of the parent's data space, heap, and stack. It is noteworthy that parent and child do not share these portions of memory, but operate on individual copies. The parent and child share the 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.

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

pid_t wait(int *stat_loc);
                                    // Returns: process ID if OK; -1 on error
```

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

Exercise 1: Implement a program that spawns a child process by using the fork and wait functions. Each process entity shall print a text message that indicates whether it is the parent's or the child's thread of execution, and the parent process shall wait for the child process to terminate.

2 IPC using Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have the limitations that they are often only half-duplex (i.e., data flows in only one direction) and that they can only be shared between processes that have a common ancestor. Normally, a pipe is created by a process via the pipe function, that process calls fork, and the pipe is used between the parent and the child.

```
#include <unistd.h>

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

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. A pipe in a single process is

next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child or vice versa.
What happens after the fork depends on which direction of data flow we want. 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]`). After closing the corresponding file descriptors with the `close` function, the file descriptors can be written to, or read from, by using the `write` and `read` functions.

Exercise 2: Extend your solution from Exercise 1, so that it implements pipe-based IPC between the parent and the child. The two processes shall exchange a message between each other that is either a constant, hardcoded string, or a user-supplied input. Upon successful reception, the receiving process shall output the received message to the console (e.g. via `printf`). The direction of data flow is up to you to decide.

3 IPC using 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 a message queue descriptor that is used to refer to the open message queue in later calls. Each message queue is identified by a 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 by calling `mq_send`, which can 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 can be deleted using `mq_unlink`. For a general overview of POSIX message queues, please refer to the manpages on `mq_overview`.

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

mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
                        // Returns: Message queue descriptor; -1 on error
```

After successful invocation, the function `mq_open` returns a message queue descriptor (MQD), which is used for further addressing. The input parameters are a unique message queue `name` or identifier, the operation flag `oflag` (specifies read or write operation, creation options, etc.), `mode` argument (permissions of a newly created queue) and an optional message queue attribute `attr`.
Due to the implementation of the `mq_open` function, there are two distinct approaches to use it. The first is similar to the approach taken in the previous section

"IPC using Pipes", as 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, of which one has to create the message queue first, before it can be opened by the second process (synchronization!). The advantage of this addressing scheme is that processes of different ancestors can access the queue.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
                                        // Returns: 0 if OK; -1 on error
```

The mq_send function adds the message pointed to by the argument msg_ptr to the message queue specified by mqdes. The msg_len argument specifies the length of the message in bytes pointed to by msg_ptr and will be placed in the queue at the position indicated by msg_prio.

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
                         // Returns: Length of selected message; -1 on error
```

The mq_receive function is used to receive the oldest of the highest priority message(s) from the message queue specified by mqdes. If the size of the buffer in bytes, specified by the msg_len argument, is less than the mq_msgsize attribute of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the msg_ptr argument.
In order to provide a buffer of mq_msgsize size, the message queue attributes shall be obtained by mq_getattr.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
                                        // Returns: 0 if OK; -1 on error
```

The member mq_msgsize of mqstat contains the size of the message queue's buffer. The malloc function can be used to allocate a suitably sized buffer to be provided to mq_receive. When the allocated memory is no longer required, it shall be de-allocated using the free function.
When a process has finished using the queue (i.e., it has successfully sent or received its message(s)), the process closes the queue using mq_close.

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
                                        // Returns: 0 if OK; -1 on error
```

One of the processes should also take care of deleting the message queue using mq_unlink, when it is no longer required. Else the message queue will remain within the system until reboot and subsequent calls to mq_open might fail or lead to unexpected behavior.

```
#include <mqueue.h>

int mq_unlink(const char *name);
                                        // Returns: 0 if OK; -1 on error
```

Exercise 3: Extend your solution from Exercise 1, so that it implements IPC based on POSIX message queues between the parent and the child. The two processes shall exchange a message between each other that is either a constant, hardcoded string, or a user-supplied input. Upon successful reception, the receiving process shall output the received message to the console (e.g. via printf). The direction of data flow is up to you to decide. The function mq_open offers two distinct options to use it, either before or after the fork (as described above), and you may decide which option you prefer to implement. Please ensure to unlink the message queue after successful transmission and closure of the MQD.
Please note: The mq_* set of functions require you to link to the real-time library, which is done by appending **-lrt** to the gcc command.

4 IPC using POSIX Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between processes. On the other hand, using shared memory requires synchronization among multiple processes to avoid inconsistencies caused by concurrent read/write access. Often semaphores are used to synchronize shared memory access, but to keep things simple, we will use an implicit synchronization instead. By waiting in the parent for the child to return it can be guaranteed that the write to the shared memory is complete and that it can be read from.
The semantics 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 handle that is used to refer to the open SMO in subsequent calls. Each SMO is identified by a 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 accessed 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 can be deleted using `shm_unlink`. For a general overview of POSIX shared memory, please refer to the manpages on `shm_overview` in section 7.

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

int shm_open(const char *name, int oflag, mode_t mode);
                                   // Returns: File descriptor; -1 on error
```

After successful invocation, the function `shm_open` returns a file descriptor for a shared memory object (SMO), which is used for further addressing. Necessary input parameters are a unique shared memory `name` or identifier, the operation flag `oflag` (specifies read or write operation, creation options, etc.) and the `mode` argument (permissions of a newly created SMO).
Due to the implementation of the `shm_open` function, there are two distinct approaches to use it, similar to `mq_open` of the message queue exercise. Please keep in mind that synchronization might be required if you call `shm_open` after the fork, as one process will be creating the SMO and it shall not be accessed by the other process before creation has successfully finalized.
As a newly created SMO has a length of zero, we adjust its size using `ftruncate`.

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

int ftruncate(int fd, off_t length);
                                          // Returns: 0 if OK; -1 on error
```

After forking a child process, both processes map the SMO into their virtual address space by calling `mmap`.

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

void *mmap(void *addr, size_t length, int prot, int flags, int
fd, off_t offset);
              // Returns: Pointer to the mapped area; MAP_FAILED on error

int munmap(void *addr, size_t length);
                                          // Returns: 0 if OK; -1 on error
```

The starting address for the new mapping is specified in `addr` and the `length` argument specifies the length of the mapping. The `prot` argument describes the desired memory protection of the mapping (and must not conflict with the open

mode of the file!). The `flags` argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.
The SMO can be accessed through the pointer returned by `mmap`. When a process has finished using the SMO (i.e., it has successfully sent or received its message(s)) it unmaps the SMO using `munmap`.

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

int shm_unlink(const char *name);
                                         // Returns: 0 if OK; -1 on error
```

One of the processes should also take care of deleting the SMO using `shm_unlink`, when it is no longer required. Else the SMO will remain within the system until reboot and subsequent calls to `shm_open` might fail or lead to unexpected behavior.

Exercise 4: Extend your solution from Exercise 1 or modify your solution from Exercise 3, so that it implements IPC based on POSIX shared memory between the parent and the child. The two processes shall exchange a message that is either a constant hardcoded string, or a user-supplied input. Upon successful reception, the receiving process shall output the received message to the console (e.g. via `printf`). The direction of data flow is up to you to decide. The function `shm_open` offers two distinct options to use it, either before or after the fork (as described above), and you may decide which option you prefer to implement. The same holds true for the placement of `mmap`. Please ensure to unlink the SMO after successful transmission and unmapping. Also keep in mind to properly synchronize the parent and the child using the `wait` function.
Please note: The `shm_*` set of functions require you to link to the real-time library, which is done by appending **`-lrt`** to the gcc command.