Operating Systems

WiSe 2015/2016

— Lab Assignment 3 —

Linux Loadable Kernel Modules:
Producer-Consumer, Synchronization & Work Queues

— 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, Dec 14, 2015
Submission:	Tue, Feb 2, 2016 before 23:59
Testing:	Fri, Feb 5, 2016 (room A213)

Preface

The purpose of this lab is to gain deeper experience with the Linux kernel and learn how to use kernel level synchronization mechanisms and work queues. You will build upon your experience from previous labs to design a producer-consumer system that operates across the kernel-user space boundary.

This lab contains multiple sections covering the different topics. Each numbered section contains both programming tasks and questions that have to be answered. Although the programming tasks may build upon each other, you have to clearly separate them in 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 or LKM 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!* ***Make sure that every group member has detailed knowledge of the whole lab solution! We expect that every group member can answer questions related to all tasks.***

You should do the lab on your own machine. Since you will work on the kernel level, we strongly recommend doing all the programming tasks *inside a virtual machine* in order not to damage your 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 to send your lab solution as compressed 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.***

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`). Proper error handling and cleanup in error cases is especially important for kernel code since improper error handling can have severe effects on your system. Please note that you cannot use the same approaches for in-kernel error handling as you would do for user-space programs, e.g., printing to `stderr` and calling `exit()` does not work inside the kernel.

Make sure that *your* code compiles without any warnings by using the `-Wall` compiler switch during compilation. 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, section 2 to system calls, and section 9 to general kernel routines. These three 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. For building Linux kernel modules, your Makefiles need to follow a specific structure, which is explained below. 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/

Producer–Consumer System

This lab focuses on building a producer-consumer system that supports n producers and m consumers as illustrated in Figure 1. The numbers n and m are not predetermined and can change during execution. The producers write data items to a shared FIFO queue while the consumers read data items from the queue. Each producer produces data items at an individual rate and each consumer consumes data items at an individual rate.

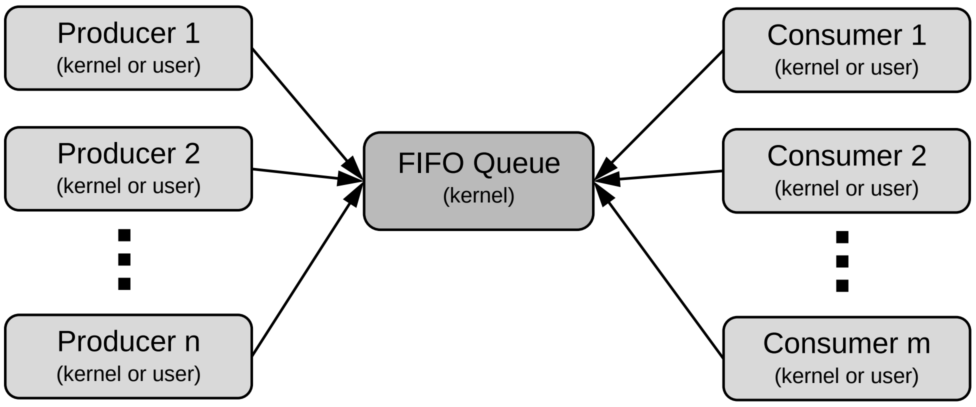

Figure 1: Producer-Consumer System

Access to the shared FIFO queue needs to be synchronized to ensure that no data items are overwritten and that no data items are consumed more than once. Moreover, access to the FIFO queue needs to be blocking, i.e., producers that try to add to the queue while it is full wait until space becomes available and consumers that try to consume while the queue is empty wait until new data items arrive.

The FIFO queue has to be implemented as LKM within the kernel and provide an appropriate interface to both user space programs and other LKMs for accessing the FIFO contents. Producers and consumers reside in both domains, user and kernel space. Kernel space producers/consumers have to be implemented as LKMs whereas user space producers/consumers have to be implemented as ordinary C programs.

Intermodule Communication

In the described producer-consumer system, producers and consumers that reside in kernel space need to be implemented as LKMs. Hence, a mechanism for exchanging information between modules is needed to allow them to access the FIFO queue, which is also an LKM. In Linux, this can be easily achieved by means of ex- and importing symbols. A symbol in this context refers to a function or a global variable as illustrated in the following example.

```
// Example: export the symbol of a module function

int square(int x) {
    return x*x;
}
EXPORT_SYMBOL(square);
// alternatively:
// EXPORT_SYMBOL_GPL(square);
```

By using the macros `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL`, a developer can export a symbol (in this case the function `square`) for use by another module. The difference between both macros is that the `_GPL` version makes the symbols only available to modules with a GPL compatible license. In the importing module, the corresponding code looks as follows.

```
// Example: import a kernel module function

extern int square(int x);
```

The module loader checks all `extern` declarations and tries to resolve them. In case a symbol declared as `extern` cannot be resolved, the attempt to load the module fails.

Module Parameters

LKMs can be passed parameter values at load-time (`insmod` or `modprobe`), similar to command line arguments for user space programs. LKMs have to explicitly declare global variables as module parameters. The provided load-time values are assigned to the module variables before the module initialization routine is invoked. This ability is useful to provide initialization values or to configure module behavior. For instance, the module `mymodule` can be loaded with the value 123 assigned to parameter `myparam` as follows:

```
insmod mymodule myparam=123
```

The corresponding parameter declaration inside `mymodule` would look as follows:

```
// Example: Declare a module parameter

static int myparam = 0;
module_param(myparam, int, 0);
```

The `module_param` macro takes 3 arguments: the name of the variable, its type, and permissions for the corresponding file in sysfs. For further information, please refer to the section “Module Parameters” of Chapter 2 of Linux Device Drivers (http://lwn.net/Kernel/LDD3/) or the kernel documentation.

Semaphores and Mutexes

The FIFO queue should use semaphores and mutexes to ensure correct race free and blocking operation. For instance, semaphores could be used for keeping track of the queue’s empty and fill counts while mutexes could be used to protect its read and write pointers. Please refer to Chapter 5 of Linux Device Drivers (http://lwn.net/Kernel/LDD3) for an in-depth description of both concepts.

Care must be taken when semaphores and mutexes are used inside the kernel as sleeping in interrupt context is not possible, i.e., blocking down/wait operations are not permitted. We therefore use workqueues (cf. Chapter 7 of LDD3) to handle item production and consumption within kernel space because workqueue functions are executed in the context of a special kernel process and can sleep. Further information on workqueues is provided below.

Rate Control & Linux Work Queues

The rate at which producers and consumers produce and consume items has to be configurable at module load-time (kernel space) and program start-time (user space). To implement the rate control mechanism in the kernel, we recommend using workqueues as they allow to schedule delayed work (producing/consuming at a later point in time). Workqueues run in a special kernel process, thus conveniently allowing blocking mutex/semaphore invocations.

Workqueues are documented in Chapter 7 of LDD3, section Workqueues. As the Linux kernel is constantly changing (and so are its interfaces), the workqueue API description of LDD is slightly outdated. More recent information (for Linux 3.16) on the workqueue API can be found in the respective kernel sources and documentation:

- workqueue.txt in the Documentation folder of Linux kernel tree
  http://lxr.free-electrons.com/source/Documentation/workqueue.txt?v=3.16
- workqueue.h in the kernel tree: include/linux/workqueue.h
  http://lxr.free-electrons.com/source/include/linux/workqueue.h?v=3.16
- workqueue.c in the kernel tree: kernel/workqueue.c
  http://lxr.free-electrons.com/source/kernel/workqueue.c?v=3.16

For your convenience, we provide a compact documentation of the most important API calls below. Workqueues are represented by a `workqueue_struct` that can be created using the `alloc_workqueue` function as follows:

```
/**
* alloc_workqueue - allocate a workqueue
* @name: printf format for the name of the workqueue
* @flags: WQ_* flags
* @max_active: max in-flight work items, 0 for default
* @args: args for @fmt
*
* Allocate a workqueue with the specified parameters. For detailed
* information on WQ_* flags, please refer to Documentation/workqueue.txt.
*
* RETURNS:
* Pointer to the allocated workqueue on success, %NULL on failure.
*/
struct workqueue_struct* alloc_workqueue(name, flags, max_active, args...);

Hints:
- a simple string works just fine for @name
- use WQ_UNBOUND for @flags
- use 1 for @max_active

Example:
struct workqueue_struct *wq = alloc_workqueue("lab-wq", WQ_UNBOUND, 1);
```

After use, e.g., when the kernel module is unloaded, workqueues can be safely destroyed using the `destroy_workqueue` function as follows:

```
/**
* destroy_workqueue - safely terminate a workqueue
* @wq: target workqueue
*
* Safely destroy a workqueue. All work currently pending will be done
* first, i.e., blocks until queue is empty.
*/
void destroy_workqueue(struct workqueue_struct *wq);

Example:
destroy_workqueue(wq);
```

Delayed work items that can be put in a workqueue are represented by a `delayed_work` struct. Delayed work items can be declared using the macro `DECLARE_DELAYED_WORK` as follows:

```
/**
* Macro DECLARE_DELAYED_WORK - declare a delayed work item
* @name name of declared delayed_work structure
* @fn function to be called in workqueue
*
* This macro declares a struct delayed_work with name @name that executes
* function @fn.
*/
DECLARE_DELAYED_WORK(name, fn);

Example:
DECLARE_DELAYED_WORK(lab_work, lab_work_handler);
```

Work items are added to the workqueue using the function `queue_delayed_work`, and queued work may be cancelled using the function `cancel_delayed_work`.

```
/**
* queue_delayed_work - queue work on a workqueue after delay
* @wq: workqueue to use
* @dwork: delayable work to queue
* @delay: number of jiffies to wait before queueing
*
* Work @dwork is added to workqueue @wq after @delay jiffies.
*/
static bool queue_delayed_work(struct workqueue_struct *wq,
                               struct delayed_work *dwork,
                               unsigned long delay);

Example:
// queue work lab_work after 2 seconds in queue wq
queue_delayed_work(wq, &lab_work, 2*HZ);

/**
* cancel_delayed_work - cancel a delayed work
* @dwork: delayed_work to cancel
*
* Return: %true if @dwork was pending and canceled; %false if it wasn't
* pending.
*
* Note:
* The work callback function may still be running on return, unless
* it returns %true and the work doesn't re-arm itself. Explicitly flush
or
* use cancel_delayed_work_sync() to wait on it.
*/
bool cancel_delayed_work(struct delayed_work *dwork);

Example:
cancel_delayed_work(&lab_work);
```

1) FIFO Queue

The FIFO queue has to be implemented as LKM in a similar, but more powerful, manner to the FIFO implementation in lab 2. The FIFO module has to export a convenient interface (functions) to other kernel modules that allows to add and remove data items while ensuring correct synchronization and blocking behavior. In addition to this kernel level interface, the FIFO module has to provide a user level interface by providing character device nodes in `/dev` for file based access from user space (cf. lab 2).

The FIFO stores data items that contain three pieces of information as specified in the code snippet below. `qid` is a sequence number assigned by the FIFO upon item insertion; sequence numbers must be unique and consecutive (item 1, 2, … `MAX_UINT`). `time` is the system time in seconds (cf. `do_gettimeofday`) when the producer created the data item. `msg` is the actual payload that points to a text message created by the producer. Note that all data items and data item messages should be allocated using `kmalloc` and deallocated using `kfree`. If the producer/consumer resides in kernel space, allocation/deallocation is the responsibility of the producer/consumer. In the user space case, however, it is the responsibility of the provided user level FIFO interface to handle allocation and deallocation.

```
// FIFO data item
struct data_item {
    unsigned int qid;           // queue sequence number for item
    unsigned long long time;    // timestamp (seconds) of item creation
    char * msg;                 // NULL terminated C string
}
```

Task 1.1

Design and implement the FIFO LKM as described. The FIFO size, i.e., the number of data items that can be stored, has to be configurable at module load-time via module parameters; set 32 items as default size. Provide a convenient in-kernel interface for adding and removing data items. The FIFO and its in-kernel interface has to directly use the `data_item` struct as specified above. Make sure to get the synchronization right by using semaphores and mutexes. Also provide a read/write interface to user space programs by means of device nodes similar to lab 2. The user space interface has to work via text I/O, i.e., user space programs read and write text descriptions of data items rather than directly using the `data_item` struct. Hence, the FIFO user level interface has to convert between the user level text representation and the in-kernel struct representation. The user level text representation is a simple CSV (comma separated values) format: "`qid,time,msg`". The following provides an example of how the user space interface should work.

```
$ echo -n "0,1451001600,Merry Christmas!" > /dev/deeds_fifo
$ echo -n "0,1451001601,And happy new year!" > /dev/deeds_fifo
$ cat /dev/deeds_fifo
1,1451001600,Merry Christmas!
$ cat /dev/deeds_fifo
2,1451001601,And happy new year!
$ cat /dev/deeds_fifo
  => blocks/waits until another item becomes available
```

Each user space read operation has to return one data item and each write operation has to write one data item. Make sure to return appropriate error codes for error cases, e.g., if the read buffer is too small or the data item text representation is malformed. Do not forget that FIFO access has to be blocking not only from kernel but also from user space. We recommend to invoke the in-kernel interface functions for implementing the internals of the user level interface.

Task 1.2

Extend the FIFO module to provide FIFO statistics via `/proc/deeds_fifo_stats`. The virtual file has to be read-only and contain current statistical data on the FIFO in a nice human readable text format. The following values have to be in it: (1) current FIFO fill level (number of stored items, number of empty slots, percentage of filled slots), (2) current data item sequence number, (3) overall number of performed insertions and removals, (4) current number of user level producers and consumers. Feel free to add additional values here.

Task 1.3

Write a test program or script for testing the user level interface of your FIFO implementation. Check all the corner cases and make sure that the synchronization works as expected.

2) Producers & Consumers

In order to complete the producer-consumer system, kernel und user level producers and consumers with rate control have to be implemented as LKMs and C programs. The individual rates have to be configurable via LKM and command line parameters. The kernel level producers/consumers directly use the in-kernel FIFO interface and insert/remove `data_item` struct instances. The user level producers/consumers, on the other hand, use the FIFO's dev interface. The number of producers and consumers in either domain are not predetermined and can change during runtime.

Task 2.1

Design and implement both a kernel level producer LKM and a consumer LKM. Use workqueues for implementing the rate control. Use LKM parameters for configuring the rates and, for the producer, also for configuring the message (`msg` in `data_item` struct). Provide another LKM parameter for assigning each LKM instance a name string. Note that both LKMs have to be generic enough to run several instances of them. In order to run several instances of the same LKM, it is necessary to rename the source file and compile it for each instance. This is necessary because the kernel does not support loading more than one LKM with the same name.

While the producers add items to the FIFO at the configured rates, the consumers remove the items at the configured rates from the FIFO and print the item contents using `printk`. The print format has to be as follows: `"[%s][%u][%llu] %s\n"`, where the first `%s` is replaced by the instance name (the module parameter), `%u` is replaced by the item sequence number, `%llu` is replaced by the item creation time, and the last `%s` is replaced by the item message. Make sure to get the item and message allocation and deallocation right as kernel level producers/consumers are directly responsible for this.

Take special care that your LKMs are cleanly unloadable (`rmmod`) without leaking memory and without indefinitely blocking unload attempts. Possibly, you also have to tweak the in-kernel FIFO interface.

Task 2.2

Design and implement a user level C program that can act as both a producer and a consumer. Whether the program executes in producer or consumer mode has to be configurable via a command line parameter. Also provide command line parameters for assigning an instance name string and for configuring the rate and, for producers, the message. Consider using `getopt` for handling command line parameters, see `man 3 getopt` for details. Keep your C program generic enough so that multiple instances in the different modes can be used concurrently.

In producer mode, the program has to add items to the FIFO at the configured rates, in consumer mode, it has to remove the items at the configured rates from the FIFO and print the item contents to `STDOUT`, e.g., using `printf`. Use the same print format as in the previous task. Make sure to get the user level FIFO protocol right. Remember that item and message allocation and deallocation on the kernel side is the responsibility of the FIFO dev interface, but do not leak memory on the user side either.

Take special care that your program is cleanly interruptible, i.e., it has to cleanly terminate upon receiving a `SIGINT` (e.g. hitting ctrl-c in the shell) without indefinitely blocking. Possibly, you also have to tweak the FIFO dev interface.

Task 2.3

Write a (shell) script for testing a producer-consumer scenario using your FIFO and producer/consumer implementations. Configure your FIFO to provide 43 item slots. Use 2 instances of kernel level producers that produce at a rate of 2 items/sec and 3 instances of user level producers where 2 produce 3 items/sec and one produces 5 items/sec. Use 1 kernel level consumer that consumes 6 items/sec and 2 user level consumers where one consumes 2 items/sec and the other consumes 4 items/sec. Make sure to give all your producers and consumers unique instance name strings so that their output is distinguishable. Also configure all your producers to produce funny messages.

Your script has to save all output from the producers and consumers in a text file. Consider using `dmesg` for retrieving the LKM output and IO redirection (`">"`) for capturing the C program output. Moreover, your script has to periodically read the FIFO statistics (ProcFS) and save them in another file.

Run the described scenario for about 2 minutes and observe the behavior. Make sure that the behavior you observe is correct; repair your implementation if you observe incorrect behavior. Include the gathered text files with the output in your submission.

How long does it take in this scenario until the FIFO is full? Change the production and consumption rates so that you can observe cases in which the FIFO gets empty. Also include log files for this second scenario in your submission.