Operating Systems

WS 2014/2015

Lab 3 – LKM: Intermodule Communication, Synchronization, Work Queues

Submission deadline: January 13, 2015

Testing: January 16, 2015

Form groups of 2 to 3 students!

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

Goal

The focus of this lab is on understanding the interaction and communication mechanism of Linux kernel modules, how to defer work using workqueues, how to use semaphores and mutexes for synchronization, and how to provide parameters to kernel modules.

Producer - Consumer System

For this lab you will be implementing 3 types of modules to realize a producer - consumer system, as depicted in Figure 1. Please note that the arrows don't indicate data flow, but interface invocation. Both, the producer and the consumer modules, use functions provided by the FIFO queue.

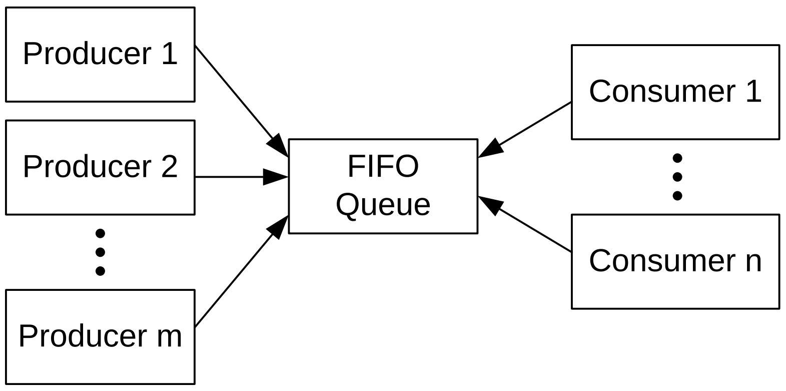

Figure 1: Producer – Consumer System

Module 1: FIFO Queue

The FIFO queue module implements a FIFO buffer of fixed size (e.g. 32) and provides an interface to the producer and consumer module(s) that allows to put items into the buffer and to remove items from the buffer. As the FIFO queue has to support the concurrent access of several producers and consumers, it needs to provide synchronization mechanisms, e.g. via mutexes and semaphores (cf. lecture 8).

Module 2: Producer

The producer module produces items, which the FIFO can store (e.g., a character or an integer), at a certain rate (items/second) and puts them into the FIFO queue. The queue's empty and fill counts are controlled by separate semaphores that must be provided by the FIFO queue. The *item* that the producer produces, as well as the *production rate* (in items/second), can be configured by the user at module load-time. When the call to insmod is made, a parameter called `item` defines the produced item. Different instances of the producer module can be configured to produce different items. Furthermore, a parameter `rate` is passed, which defines how many items are produced per second.

Module 3: Consumer

The consumer module consumes items from the FIFO queue, effectively removing them from the queue. Just as for the producer, the queue's empty and fill counts are controlled by semaphores provided by the FIFO queue. The consumer's *consumption rate* (in items/second) can be configured by the user at module load-time. When the call to insmod is made, a parameter `rate` is passed, which defines how many items are consumed per second.

Intermodule Communication

In the above example, the producer and consumer modules need to communicate with the FIFO queue module in some way. In Linux, this is done via intermodule communication. For Linux 2.4, the mechanism of choice was `inter_module_register` and adjacent functions (cf. Linux Device Drivers, 2nd Edition, Chapter 11). However, this mechanism was deprecated in Linux 2.6 onwards. Instead, a simple scheme based on symbol name import and export is used.

```
// Example: export the symbol of a kernel module function
#include <linux/module.h>

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`, the programmer can export a symbol (e.g., a function or variable name) 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 code looks as follows.

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

extern int square(int x);
```

The module loader will check all `extern` declarations and try to resolve them. In case a symbol declared as `extern` cannot be resolved, loading the module will fail.

Semaphores and Mutexes

The FIFO queue provides semaphores for the queue's empty and fill count, as well as a mutex for accessing the queue itself to either add or remove items. 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 because workqueue functions are executed in the context of a special kernel process and can sleep. Further information on workqueues is provided below.

Module Parameters

Parameters can be assigned to kernel modules at load-time by `insmod` or `modprobe`, for instance, to provide initialization values or to configure module behavior. In order to load the module `mymodule` with the value `123` assigned to parameter `myparam`, the following command would be used:

```
insmod mymodule myparam=123
```

The parameter declaration inside the module is as follows.

```
// Example: Declare a parameter inside a module
#include <linux/module.h>

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/).

Rate Control

The rate at which producers and consumers produce and consume items may be configured at module load-time. To implement the rate control mechanism, we recommend using workqueues, as they allow to schedule delayed work (i.e., producing or consuming an item at a later point in time). Further, workqueues run in a special kernel process, thus allowing blocking mutex/semaphore invocations to sleep.

Workqueues are documented in Chapter 7 of LDD3, section Workqueues, pages 205ff. As the Linux kernel is constantly changing (and so are its interfaces), the workqueue API description of LDD is slightly outdated. More recent information on the workqueue API can be found in the respective kernel sources and documentation:

- workqueue.txt in the *Documentation* folder of Linux kernel tree
  https://github.com/torvalds/linux/blob/master/Documentation/workqueue.txt
- workqueue.h in the kernel tree: include/linux/workqueue.h
  https://github.com/torvalds/linux/blob/master/include/linux/workqueue.h
- workqueue.c in the kernel tree: kernel/workqueue.c
  https://github.com/torvalds/linux/blob/master/kernel/workqueue.c

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
*
* Kill off a pending delayed_work.
*
* 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);
```

Problem Statement

Implement a producer - consumer system with three separate loadable kernel modules as described above. The FIFO queue module contains the actual buffer and exports an interface to the producer and consumer modules for adding and removing items. The FIFO module also provides the required means for synchronization via mutexes and semaphores.
Producer and consumer modules can access the FIFO concurrently. While the producer(s) constantly fill the queue at a certain rate, the consumer(s) empty the queue at a certain rate. The item that a producer instance generates, as well as the production and consumption rate, can be configured via module parameters.

Note: To run several instances of the producer / consumer, it might be necessary to rename the source code and compile several instances of the same module, to be loaded with insmod. **In your submitted solution please only include one instance of the producer and consumer!**