COMP 530H Lab
Fall 2017

Programming Assignment 3
Module for barrier synchronization
Date assigned: September 15, 2017
Date due: September 29, 2017

Implement a service module to be called by user programs for barrier-style synchronization. In this form of user-level synchronization, one or more processes may concurrently wait for a globally-scoped event identified by an integer value. The waiting process(es) are blocked on kernel wait queues until another process indicates that unblocking should occur by "signaling" the event that has that identifier. The scope of an event is system wide so that any process may access an event by its identifier. Assume that synchronizing processes either use a "well-known" set of event identifiers or use some out-of-band communication mechanism (e.g., files) to share a set of identifiers. The means used by processes to learn and share system-wide event identifiers is outside the scope of this assignment.

Specifically, your module should implement the following set of operations: The input to your module from a calling program is a character string with the following syntax:

<operation name><space><integer-1>[<space><integer-2>]

Where the element in **[ ]** indicates that it is not used for all operations. The character string names of the operations are shown below in bold. The value represented by **<integer-1>** must be in the range **(0 – 99)** and **<integer-2>** represents only the values, **0 and 1.**

- **event_create** The **<integer-1>** parameter is the character representation of the integer value to be assigned as the event identifier for a new event. An attempt to create an event using an already existing identifier is an error. This call is used to instantiate a new event that includes a wait queue for processes blocked on it. On successful completion, the module should return a character string containing only the **<integer-1>** value from the caller input string. If the operation fails for any reason, it should return a string containing only the value **-1**.
- **event_wait** The **<integer-1>** parameter is the character representation of the integer event identifier the process is waiting to be signaled. The **<integer-2>** character representation of an integer indicates **exclusive (value 1)** or **non-exclusive (value 0)** treatment when the event is signaled. This call is used to block a process by placing it on the event's wait queue until the event is 'signaled' by another process. Note that if successful, this call does not return in the user process (it becomes blocked while executing inside your module) unless and until the event is signaled or destroyed. This means that the **write()** call in the user process will not complete until it is unblocked. It can then do the **read()** call to determine the result. On successful completion, the module should return a character string containing only the **<integer-1>** value from the caller input string. If the operation fails for any reason, it should return a string containing only the value **-1**.
- **event_signal** The **<integer-1>** parameter is the character representation of the integer value of the event identifier to be signaled. This call is used to unblock one or more processes (depending on the process **exclusive/non-exclusive** flag). On successful completion, the module should return a character string containing only the **<integer-1>** value from the caller input string. If the operation fails for any reason, it should return a string containing only the value **-1**.
- **event_destroy** The **<integer-1>** parameter is the character representation of the integer value of the event identifier to be destroyed. This call unblocks all blocked processes independent of their **exclusive/non-exclusive** flag and makes the

event and its wait queue unavailable to all processes (identifier is no longer valid until another **event_create** is called with the same value). On successful completion, the module should return a character string containing only the **<integer-1>** value from the caller input string. If the operation fails for any reason, it should return a string containing only the value **-1**.

As indicated above, a process that waits can select either **exclusive** or **non-exclusive** treatment when it is unblocked by a signal. The selection is implemented with the kernel functions **add_wait_queue()** and **prepare_to_wait()** or **add_wait_queue_exclusive()** and **prepare_to_wait_exclusive()**. When a wait queue is unblocked with the kernel function **wake_up()**, all **non-exclusive** blocked processes (if any) are made **RUNNABLE** along with exactly one **exclusive** blocked process (if any). These semantics for process unblocking are generally more useful when they are not both used on the same wait queue, but this separation is not required.

You will need to use the Linux representation of wait queues and the various functions for placing processes on wait queues to block them, removing processes from wait queues to unblock them and invoking the scheduler (**schedule()**). In general, the semantics for the blocking and unblocking operations for processes using these events should be the same as for process blocking and unblocking on the kernel wait queues. For this assignment, consider all blocked processes to be in the TASK_NORMAL state. The important include files for this assignment are **<llinux/sched.h>**. and **<linux/wait.h>** .

A tutorial and examples of Linux wait queue handling can be found in the book by R. Love, Linux Kernel Development, 3rd Ed., pp. 58-59 and 61 (Waking Up). Note that the text discusses the concepts and example in terms of an abstract condition variable that may be TRUE or FALSE. You will not need to use the concept of a condition variable because there is an explicit wait/signal synchronization model that will be used by calling user tasks. Therefore, the example on pp. 59 can be simplified by eliminating the **while {}** loop and the check for external (Linux OS) signals. When handling both exclusive and non_exclusive processes as required in this assignment, it is necessary to use the proper functions to add processes to a wait queue and prepare to wait. The reason is that **wake_up()** expects the wait queue to be managed such that the non-exclusive processes are inserted at the front of the queue (LIFO) and exclusive processes at the tail. This insertion order is handled by the functions with (for tail insertion) or without (for head insertion) the suffix **_exclusive** in the name. If the queue is not ordered this way, the first exclusive process encountered by **wake_up()** terminates the search of the queue for non-exclusive processes.

In this assignment you will need to consider how you will deal with potential concurrent use of any global state or with preemption causing conflicting calls for a given event. Your module should deal with any errors that can occur (but not be burdened with tests for errors that cannot occur). Give careful consideration to the caller programs you will write to test your module for both the usual cases (for example, **event_signal()** with one or more proceses blocked on the event) and for the corner cases (for example, **event_signal()** with no proceses blocked on the event). Since processes that use the **event_wait()** operation will block, you will need other processes using the **event_signal()** operation for unblocking.

Submitting your program:

Follow the instructions from assignment 1. You will need to submit your C source code, your include file that is to be used for common definitions shared with a caller program, and a Makefile. Note that you can choose your own name for the module and the debugfs directory and file.