COMP 530H
Fall 2014

Programming Assignment 3
Module for named-event services
Date assigned: September 26, 2014
Date due: October 10, 2014

Implement a named-event 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 a character string name. 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 name. The scope of an event is system wide so that any process may access an event by its identifier. Assume that processes either use a "well-known" set of event names or use some out-of-band communication mechanism (e.g., files) to share a set of names. The means used by processes to learn and share system-wide event names is outside the scope of this assignment. Specifically, your module should implement the following set of operations: Note that you are free to define the exact syntax used in the character strings exchanged between your module and the calling user program.

- ***event_create*** (parameter is the character string name to be assigned to the event; returns an integer identifier (> 0) or an error indication). This call is used to instantiate a new event that includes a wait queue for blocked processes.
- ***event_id*** (parameter is a character string event name; returns an integer identifier (> 0) or an error indication). This call is used to map an event name to an integer event identifier for a previously created event. The identifier is used in other calls (below).
- ***event_wait*** (parameters are an integer identifier for an event and an integer that indicates exclusive (value 1) or non-exclusive (value 0) treatment when the event is signaled; returns an integer identifier of the event when the event is signaled, 0 when the event is destroyed, or an error indication immediately). 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 to the user process 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.
- ***event_signal*** (parameter is an integer event identifier; returns 0 or an error indication). This call is used to unblock one or more processes (depending on the process exclusive/non-exclusive flag).
- ***event_destroy*** (parameter is an integer identifier for an event; returns 0 or an error indication). 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 (name and identifier are no longer valid).

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). The calling program should receive sufficient information so that it can determine if the call has been successful or has failed.

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. In general, the semantics for the blocking and unblocking operations for processes using these named 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>. <linux/wait.h> and <linux/preempt.h> (for enabling and disabling kernel preemption).

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 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 defined for your named events. Therefore, the example on pp. 59 can be simplified by eliminating the while {} loop and the check for external (Linux OS) signals.

Give careful consideration to the 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. Write your test programs in a way that will be useful for demonstrating your tests to me (see below).

***Submitting your program*:**

Follow the instructions from assignment 1. Also, for this assignment I will want to schedule a 30 minute meeting with each of you so you can demonstrate for me how your module is tested. I will email with instructions for scheduling the demos after the assignment is due. The demos will take place in my office and will be most effective if you bring a laptop so we can use my office projection system.