CS 2510 - Computer Operating Systems

Fall Term 2013

Project 1
Due Date – 10/15/2014

1. Introduction

A process in a conventional UNIX environment has a private address space in which a single thread of control, which is initiated by invoking the function main(), lives. If the application requires the execution of concurrent activities, then as many child processes as needed must be forked and organized. The UNIX implementation of a process requires high overhead in terms of context switching overhead, process slots, file descriptors and page tables. This may be a waste of resources for a class of applications based on a client/server model of interaction, where a process has to be created by a server for each client request.

The purpose of this project is to implement a light weight user-level thread (LWT) scheduling system without the need to modify the Linux kernel. Your system must allow the creation and concurrent execution of threads of control within a Linux task, in a true preemptive manner. The LWT system must be capable of creating threads and performing mini context switches to share the CPU time among the task threads, based on the specified scheduling policy.

2. LWT Design Overview

The basic components of the LWT system include:

- **A thread scheduler** – The thread scheduler schedules and performs a mini context switch between the threads. The scheduler can be implemented as a SIGALRM handler, using the **ualarm()** system call to generate a SIGALRM signal in a periodic manner. The basic policy to achieve thread scheduling is a preemptive round robin scheme[1].
- **A thread context** – Each thread is provided with its own context which is a private stack area, stack pointer, and other thread attributes such as its current state and environment on the last context switch. When a context switch is performed, the context of the interrupted thread is saved. This operation involves:
    - Saving the current stack pointer value of the thread[1], and
    - Performing a **setjmp()** to save other registers.

[1] Extra credits will be awarded for implementing a static or dynamic priority-based handling mechanism to handle real-time requirements.

Upon saving the context of the interrupted thread, the next thread is fetched. This operation involves:
    - Setting the system stack pointer to point to the thread stack pointer, and
    - Performing a **longjmp()**.
- **Semaphores** – A semaphore, **S**, is an integer variable that, apart from initialization, is accessed only through two standard atomic operations, namely **P()** and **V()**. To avoid busy waiting, a thread that executes the P() operation and finds that the semaphore value is not positive, blocks. The block operation places the thread into a waiting queue associated with the semaphore. Then control is transferred to the scheduler, which selects the next thread to execute.

3. LWT Design and Runtime Environment

A thread in the LWT system may be in one of the following possible states:

- **lwt_READY –** In this state, the thread is eligible for a CPU time slice by the scheduler. Notice that several threads may be in this state at the same time.
- **thrd_WAIT** – A thread moves into this state when it issues a **thrd_wait()** system call. In this state, the thread loses its eligibility to contend for a CPU time slice, and awaits the termination of the thread it is waiting for.
- **sem_WAIT** – A thread moves into this state when it is waiting for a semaphore to be released.
- **lwt_SLEEP**: A thread moves into this state when it issues **lwt_sleep()** system call.
- **lwt_EXIT**: A thread moves into this state when it issues a **lwt_exit()** system call. The scheduler, whenever it takes over, releases the private working area of the exiting thread and performs the required modifications on the thread queue.

Several issues must be addressed for the proper implementation of the light weight thread system. These issues are discussed in the following sections.

3.1 Initiating the Thread Subsystem

In order to allow an application to use the light weight thread system, an **lwt_init()** procedure must be invoked before creating any threads. The purpose of this procedure is to allocate and initialize properly the required data structure and setup the initial thread. More specifically, the procedure virtually creates the program first thread by creating the thread queue and placing the appropriate attributes in the thread entry. We will assume that 16 KB of stack area is sufficient for most applications. In essence, we are assuming that the depth of recursion and the size of the data structure requested by any function do not exceed 16 KB. In addition, the procedure should install the SIGALRM handler. The procedure must be invoked with a **lwt_quantum** argument, which represents the time slice in **μ** seconds for each light weight thread.

3.2 Thread Creation

Upon initializing the LWT system, the application creates a new thread by invoking the system primitive, **lwt_create()**. The primitive must create a private working area for

the thread, and proceed to immediately execute the thread. The invoked call returns to the caller a unique thread descriptor. The arguments of the primitive must include the name of the routine to be executed by the thread, the arguments and their numbers, and the name of the thread itself. We assume that these parameters are correctly defined by the programmer.

3.3 Thread Termination

A thread must explicitly call a system primitive, **lwt_exit()**, when it desires to leave the system. The primitive must be called with a status value argument which is either returned to another thread which performed a **thrd_wait()** on this thread or returned to the system (shell or other processes waiting for this process) if the thread is the last thread running in the program. If no other thread is waiting for the exiting thread or if it is not the last thread in the system then the exit status is ignored. When a thread performs a **lwt_exit()** its state is changed to **lwt_EXIT** and all threads waiting for the exiting thread are awaken and their states are changed from **thrd_WAIT** to **lwt_READY**.

3.4 Waiting for other Threads

A parent thread forking another thread may wish to wait until the newly generated thread terminates. This achieved by invoking the primitive **thrd_wait(),** which moves the calling thread into **thrd_WAIT** state. The waiting thread will be awakened when the child thread it is waiting for exists. The waiting thread can obtain the exit status of the exiting thread.

3.5 Thread sleep

A thread may wish to sleep for a given number of seconds by invoking the **lwt_sleep()** system call, which moves the calling thread into the **lwt_SLEEP** state . The thread will then be prevented from execution for *approximately* the given number of seconds.

4. **Synchronization**

As threads live in the same address space and can use the same global variables, a set of synchronization primitives must exist to allow the application programmer to protect the shared resources. The synchronization primitives implemented in the light weight thread system are based on semaphores.

A semaphore in the light weight thread system has:

- A value, which is set to zero when the semaphore is free, and
- A thread queue, to hold all threads currently waiting for the semaphore to be released.
- Two primitives P() and V() operations. The P() operation operation starts by disabling any context switch to another thread. If the semaphore is available the semaphore is given to the calling thread, the semaphore value is changed to one and context switch is allowed. If the semaphore is currently locked by another

thread, then the calling thread is queued on the semaphore, its state is changed to **sem_WAIT** and control is passed (the context switch is enabled) to the next ready thread (there should be at least one runnable thread, at least the one currently holding the semaphore, or the system is in deadlock.) When the thread owning the semaphore releases it using a V() operation the first thread queued on the semaphore is dequeued and its state is changed to **lwt_READY**. Hence, the semaphore locking works in a strict first request first acquire mode.

The programmer must declare semaphores as global variables as many threads will be synchronized by the semaphore. A set of semaphores is used for synchronization of the different activities that need exclusive access of shared resources. The programmer is responsible for correct usage of the semaphores.

5. Project requirement

For a full credit, you must:

1. Implement the LWT scheduling system, including all the primitives for thread control and management.
2. Demonstrate the LWT scheduling using the **Producer-Consumer** problem, a common paradigm for cooperating processes. Your implementation must allow for multiple producers and multiple consumers to execute concurrently.
3. A well-documented source code for your implementation, in a tar file. The file should be made available in a folder (YOURNAME_CS2510) in your public directory with access control set to allow the instructor and the TA to access the file.
4. A report describing the basic components of the system architecture, the main design decisions of the architecture.

6. References

[1] **GCC-Inline-Assembly-HOWTO**, http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.