CS6456 (F2016) Operating Systems

Project 3

Title: Creating a User-Level Thread Package (Part II)
Due: October 25, 2016 (11:00 a.m.)
Points: 20

(1) Project Description

In this project, you will gain experience with pre-emptive and priority scheduling, signals, and implementation of synchronization primitives. The project is an extension of project 2 for implementing user-level thread package.

(2) Library Interface

Your project will be implemented in Linux using C/C++. Recall that the old API calls to support were as follows; now the priority field in *uthread_create()* will need to be supported.

- **int uthread_init()**
  Initialize the thread library. The function shall return 0 when it succeeds and -1 otherwise. For this project, the function shall fail when the library is already initialized.

- **int uthread_create(void (*func)(int), int val, int pri)**
  Create a new thread with priority *pri* that starts execution with function *func()* whose argument is *val*; i.e. the prototype of *func* is *void func(int val)*. The *uthread_create()* function shall return the identifier of the thread thus created. The thread ID should start from 0, where the main thread is always assigned 0 as its thread ID and other threads created in the main thread are assigned thread IDs starting from 1. When creating a thread, a stack of 1 MB is allocated. The created thread should not run immediately; it should be put at the end of the queue of threads waiting to run. **Now the priority field will need to be supported.** The main thread should be given the lowest priority (by convention, UNIX uses lower nonnegative integers to represent higher priority). You may assume that the priority passed to the function for threads ranges from 0 to 99, where 0 represents the highest priority and 99 the lowest.

- **int uthread_yield()**
  For the currently running thread to relinquish the CPU. See section 3 below for details on the new, pre-emptive, priority-aware scheduling policy. On Linux, this call always succeeds. So your program may just return 0 after the call is successfully executed.

- **void uthread_exit(void *retval)**
  Terminate the currently running thread and returns a value via *retval* that is available to other threads. The call also makes the scheduler select another thread to run, if there is one, according to the pre-emption-aware, priority-based scheduling algorithm explained in section 3 below. If the exiting thread is the only thread in the process, the entire process should exit. All threads, including the main thread, should exit via *uthread_exit()*; behavior is undefined if any thread fails to do so. Note the function always succeeds and returns nothing.

The following shows the new calls to be implemented in your project.

- **int uthread_mutex_init(uthread_mutex_t *mutex)**
  Instantiate the *mutex* lock. The function shall return 0 on successful return and -1 otherwise. For this project, the function shall fail when the *mutex* requested for initialization has already been initialized.

- **int uthread_mutex_lock(uthread_mutex_t *mutex)**
  The mutex object referenced by *mutex* will be locked. If the *mutex* is already locked, the calling thread is **suspended** until the *mutex* is available — **no busy-waiting allowed**. The function shall return 0 on successful return and -1 otherwise. For this project, the function shall fail if the value specified by *mutex* does not refer to an initialized mutex object.

- **int uthread_mutex_unlock(uthread_mutex_t *mutex)**
  Release the mutex object referenced by *mutex* if no thread is waiting on it; otherwise, leave the mutex object locked but allow the highest-priority waiting thread to proceed. The function shall return 0 on successful return and -1 otherwise. For this project, the function shall fail if the value specified by *mutex* does not refer to an initialized mutex object or is called twice without making a lock call in between.

Whenever a thread calls *lock*, it can no longer be interrupted by any other thread. Once it calls *unlock*, the thread resumes its normal status and will be scheduled whenever an alarm signal is received. Having the *lock* and *unlock* functions available is useful to protect small pieces of code (critical sections) during which you do not want the current thread to be interrupted. For example, *lock* and *unlock* should be used whenever your thread library manipulates global data structures (such as a list of running threads). A thread is supposed to call *unlock* only when it has previously performed a *lock* operation. Moreover, the result of calling *lock* twice without invoking an intermediate *unlock* is undefined.

- **int uthread_join(uthread_tid_t tid, void *retval)**
  The *uthread_join()* function shall suspend execution of the calling thread until the target thread terminates. If the target thread has already terminated at the time when *uthread_join()* is called, then *uthread_join()* returns immediately. On return from a successful *uthread_join()* call with a non-NULL *retval* argument, the value passed to *uthread_exit()* by the terminating thread shall be made available in the location referenced by *retval*. The results of multiple simultaneous calls to *uthread_join()* specifying the same target thread are undefined. As you can see, with *uthread_join()*, there is now the need to correctly handle the exit code of threads that terminate. To make your life easier, we assume that a thread always calls *uthread_exit()* to exit its execution. Note that *uthread_join()* returns 0 on successful return and -1 otherwise. For this project, the function can fail if another thread is already waiting to join with this thread or no thread with the thread ID (returned from *uthread_create()*) could be found.

- **int usem_init(usem_t *sem, int pshared, unsigned value)**
  The *usem_init()* function shall initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore shall be *value*. The value has to be less than *SEM_VALUE_MAX*, which is 65,536 for this project. The *pshared* argument has to be always zero, which means that the semaphore is shared among threads of the process. The function returns 0 when succeeds and -1 otherwise. The call shall fail if the semaphore has already been initialized or *value* exceeds *SEM_VALUE_MAX*.

- **int usem_destroy(usem_t *sem)**
  usem_destroy() destroys the unnamed semaphore at the address pointed to by *sem*. Only a semaphore that has been initialized by *usem_init()* should be destroyed using *usem_destroy()*. Destroying a semaphore that other threads are currently blocked on (in *usem_wait()*) produces an error. Using a semaphore that has been destroyed also produces an error. The function returns 0 when succeeds and -1 otherwise. The call shall fail if the semaphore does not exist, has blocked threads, or has already been destroyed.

- **int usem_wait(usem_t *sem)**
  usem_wait() decrements the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement (i.e. the semaphore value rises above zero). Note that in this implementation, the value of the semaphore **never** falls below zero (unlike how it is handled in some semaphore wait definitions). The function returns 0 when succeeds and -1 otherwise. The call shall fail if the semaphore has not been initialized (i.e. does not exist yet) or destroyed already.

- **int usem_post(usem_t *sem)**
  usem_post() increments the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another thread blocked in a *usem_wait()* call will be woken up (in a FIFO manner) and proceeds to lock the semaphore. Note that when a thread is woken up and takes the lock as part of *usem_post()*, the value of the semaphore will remain zero. The function returns 0 when succeeds and -1 otherwise. The call shall fail if the semaphore has not been initialized (i.e. does not exist yet) or has been destroyed.

(3) Thread Scheduler

Every thread is given a time slice of 1 millisecond. When a time slice expires or a yield is called, a scheduling decision is made to find the next executable thread. Your library should support 10 different priority levels, with 0 being the highest and 9 the lowest. Threads are classified into one of the 10 queues using their priorities passed to the *uthread_create()* function. For example, threads with priorities ranging from 0 to 9 are queued in class 1, threads with priorities ranging from 10 to 19 are queued in class 1, and so on. Use round-robin scheduling to find the next runnable thread among threads of same priority level. Lower priority levels should not be serviced unless there are no runnable threads at higher levels. For simplicity, **priorities are fixed** (i.e. feedback scheduling is **not** allowed**)**.

(4) Implementation

In this project, you will build upon the thread library that you developed for the previous project (i.e. project 2). Thus, as a first step, make sure that everything works well. Then, extend your library as outlined above.

Adding the *lock* and the *unlock* functions is straightforward. To lock, you just need a way to make sure that the currently running thread can no longer be interrupted by an alarm signal. For this, you can make use of the *sigprocmask()* function. To unlock, simply re-enable (unblock) the alarm signal, again using *sigprocmask()*. Once you have *lock* and *unlock*, use them to protect all accesses to global data structures. You will probably need to call these functions when implementing the semaphore routines.

To implement the *pthread_join()* function, you will probably need to introduce a BLOCKED state for your threads. Whenever a thread is blocked, it cannot be selected by the scheduler. A thread becomes blocked when it attempts to join a thread that is still running.

In addition to blocking threads that wait for (attempt to join) active threads, you might also need to modify *pthread_exit()*. In particular, whenever a thread exits, you cannot immediately clean up everything that is related to it. Instead, you need to retain its return value, because other

threads might later want to get this return value by calling *pthread_join()*. That is, once a thread exits, it is not immediately gone, but becomes a "zombie" thread (very similar to the situation with processes). Once a thread's exit value is collected via a call to *pthread_join()*, you can free all resources related to this thread.

When implementing semaphores, you might want to create a semaphore structure that stores the current value, a pointer to a queue for threads that are waiting, and a flag that indicates whether the semaphore is initialized.

Once you have your semaphore data structure, just go ahead and implement the semaphore functions as described above. Make sure that you test a scenario where a semaphore is initialized with a value of 1, and multiple threads use this semaphore to manage access to a critical code section that requires mutual exclusion (i.e., they invoke *sem_wait()* before entering the critical section). When using your semaphore, only a single thread should be in the critical section at any time. The other threads need to wait until the first thread exits and calls *sem_post()*. At this point, the next thread can proceed.

(5) Useful UNIX System Calls

- **For thread scheduling:** Use *getitimer()* and *setitimer()* with *ITIMER_VIRTUAL* to catch *SIGVTALRM*. The signal handler would implement the necessary thread scheduling functionality. Use the *signal()* system call to implement your signal handler to handle signal *SIGVTALRM* when the event occurs.

- **Mutex lock:** The *lock()* and *unlock()* operations have to be *atomic*. Atomic execution can be enforced by disabling interrupts using *sigprocmask()*/*sigaction()* or related functions.

You may not use primitives from the OS that implement the above functionalities, or existing thread packages (e.g. semaphores), other than signal/timer functions. You also may not implement lock/unlock by simply blocking signals for the duration between lock and unlock (although you probably want to block them at the beginning of each of those functions and unblock them before returning). That would defeat the purpose of this project. You must implement the scheduler and mutual-exclusion primitives yourself.

In your written evaluation (at least two pages long in PDF), be sure to discuss your implementation, specifically: (1) how each of the new functionalities above is implemented; (2) the design decisions you made; and (3) an explanation of why you made those decisions.

Miscellaneous

(1) The project does NOT allow team work.
(2) You must use a makefile to compile your program.
(3) Your program must be compiled and run successfully on the Debian VM.
(4) Your implementation must be in C/C++.
(5) Turn your project in via **collab** in a **tar** file consisting of (i) all headers you have created; (ii) source code file for project 3 (you may use any file name for your library program, although **threadsLib2.c** is a good one to choose); (iii) a make file; and (iv) a write-up showing your design and implementation of the project (at least two pages long in **PDF** format). The **tar** file should be named as follows: **p3**, followed by your computing ID, and followed by the file extension (i.e. **tar**). For example, the **tar** file turned in by **John Smith** should be named **p3js7nk.tar**, where **js7nk** is the computing ID of **John Smith**.
(6) Your program should be successfully compiled. Programs failing to compile will NOT receive any points.