Assignment 3. User-level Threads for Priority-Based Preemptive Scheduling (100 points)

In this assignment, you are required to develop a thread library that supports:

1. Preemptive scheduling of user-level threads with assigned priorities.
2. Timer related functions for periodic execution of a user-level thread.
3. Mutex lock and unlock for resource sharing between user-level threads.
4. Basic priority inheritance protocol and stack-based priority ceiling protocol

A thread is an independent unit of execution that has its own stack and set of registers (e.g., PC). For system or kernel-level threads, e.g., pthread, the creation and exit of a thread is supported by O/S with a set of system calls and the execution of a thread is managed by the O/S. On the other hand, a user-level thread, abberivated as uthread, is scheduled and managed in user level without interaction with the O/S. Hence, multiple uthreads will be seen as a single thread for the O/S. To schedule and manage uthreads, a user-level thread library will need to manipulate the context of a system thread (i.e., stack and registers) with the use of timer and signal.

In this assignment, you are required to develop a user-level thread library, *libuthread.a*, as a form of Linux static library. The user-level thread library, written in C for 64-bit Linux, is an implementation of a collection of APIs that mimic the behavior of the pthread library with additional supports of the two real-time scheduling protocols. A header file, *uthread.h*, that defines the API is provided. The header file is not to be modified for your implementation. Since the library implementation is platform dependent, it is required that your library and test applications should be compiled and run on the Red-Hat Linux in the BYENG 217 lab. Our test applications will include "*uthread.h*" and we will compile the test applications with your user thread library as follows,

```
gcc -o test_app test_app.c -L. -luthread -lrt
```

In this assignment, we assume that there exists only a single Linux thread at any given time instance. We further assume that, for the sake of simplicity, the test applications are well-written (i.e., no bug/error), and the main thread has the highest priority and is structured as follows,

```
int main() {
  1. initialize mutex
  2. create uthreads
  3. may enable stack-based priority ceiling protocol
  4. join the uthreads
    // note: main() does not have any computation to perform and all computation is
       done in uthreads.
}
```

The first step of this assignment is to support uthread creation, exit, and join.

- *int uthread_create(uthread_t *tid, void *(*start)(void *), void *args, int priority)*: to create a uthread with the priority of "*priority*". 0 is the highest priority and 100 is the lowest priority. Upon completion, the ID of the created uthread shall be stored in *tid*. "*start*" and "*args*" are the uthread start function and argument passed, respectively (as in pthread_create).

- *int uthread_join(uthread_t tid, void **value_ptr)*: to join a uthread as in pthread_join().
- *void uthread_exit(void *value_ptr)*: to exit the calling uthread. It is implicitly called when a thread "return *value_ptr;*" The joining thread shall receive *value_ptr* when it returns from *uthread_join().*

The calling thread of *uthread_create*() can create a new uthread by setting up stack and registers of the child uthread, and setting PC to the address of the child thread's start function. This can be done with Linux *setjmp(jmp_buf env)* call to obtain the context of the calling thread and by manipulating the context for the child uthread. The uthreads can be scheduled with the use of Linux *setjmp(jmp_buf env)* and *longjmp(jmp_buf env, int val)* calls. At any scheduling instance of a uthread (e.g., timer expiration, blocking on mutex lock), *setjmp*() is called to save the current context of the uthread and the scheduler is invoked. The scheduler calls *longjmp*() to switch context to the next selected uthread.

To support periodic execution of a user thread, two timer functions are needed.

- *int uthread_gettime(struct timespec *tp)*: to read the current time in the clock *CLOCK_MONOTONIC* and to store it in *tp*. It will be a simple wrapper function for *clock_gettime(CLOCK_MONOTONIC, &tp)*.
- *int uthread_abstime_nanosleep(const struct timespec *request)*: to put the calling uthread to sleeping mode and wake it up at the time specified in *request* in the clock *CLOCK_MONOTONIC*. The behavior of the calling uthread is similar to calling *clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &request, 0)*.

Note that calling *uthread_abstime_nanosleep()* shouldn't actually result in sleeping (suspendion) as there is only one Linux thread (main()). The function will (1) trigger a timer that expires at the time specified in *request*, and (2) call the scheduler to select the next highest priority ready uthread. When the timer is expired, a signal handler is invoked. The signal handler will (1) index which uthread's timer is expired, (2) change the uthread's state to ready, and (3) call the scheduler.

There can be time instance when no uthread is in ready state. To handle the case, we can (1) run an idle uthread or, (2) suspend the scheduler and wake it up at the next timer expiration instant. If you use an idle uthread, do not make it waste CPU cycles. The body of the idle thread will look something like, *while(1) { sleep(1); }*

For resource sharing and priority inheritance protocol support, the library should include:

- *int uthread_mutex_init(uthread_mutex_t *mutex, const int attr)*: to initialize the mutex referenced by "*mutex*" with the attribute specified by "*attr*". When multiple uthreads are blocking for "*mutex*", the next owner of "*mutex*" is the highest priority uthread among the waiting uthreads. When *"attr"=UTHREAD_MUTEX_ATTR_PI*, the basic priority inheritance protocol is applied for "*mutex*".
- *int uthread_mutex_lock(uthread_mutex_t *mutex)*: to lock the mutex object referenced by "*mutex*".
- *int uthread_mutex_unlock(uthread_mutex_t *mutex)*: to unlock the mutex object referenced by "*mutex*".

For enabling the stack-based priority ceiling protocol, the library should include:

- *int uthread_srp_enable(resource_entry_t *resource_table, int n)*: to enable the stack-based priority ceiling protocol. This function should be called after creating all the uthreads and before main() starts joining threads. "*resource_table*" defines the resource table needed by the protocol and "*n*" is the number of resources. Please refer to "*uthread.h*" for more details.

The last requirement is to make the library routines reentrant. That is, you need to guarantee atomicity when accessing critical sections inside the library implementation (e.g., accessing run queue or mutex structure). In this assignment, we assume that there exists only a single Linux thread execution at any time instance, and the only execution instance that can preempt thread's execution is the timer signal. Therefore, entering and exiting a critical section can be simply masking and unmasking the timer signal, respectively.

Implementation hints (not required to follow)

Scheduling: To implement a user-level thread library, first you will need a data structure for each uthread. The data structure is often called a thread control block (TCB). TCB contains variousinformation for scheduling and resource management. In our case, a TCB should contain at least, uthread's context, priority, and state:

```
struct tcb {
  int state;
  jmp_buf context;
  int priority;
  //some other information continues
};
```

A uthread can be blocked due to unavailability of resource, delayed, or runnable (ready). "state" represents the uthread's current state. The scheduler's job is to pick the highest priority runnable (ready) uthread (since, in this assignment, we only consider priority-based preemptive scheduling). "context", which is saved by calling *setjmp(tcb->context)*, contains all registers and stack pointer for the current context of the uthread.

Once the scheduler picks the next uthread to run, we need to be able to switch context from the current uthread to the selected next uthread. This task can be done by the use of setjmp() and longjmp() calls. The current running uthread saves the current context into its TCB by calling setjmp(). By calling longjmp() with the context saved in the next uthread's TCB, the next uthread resumes its execution. For instance, assume that a uthread is blocked on uthread_mutex_lock(),

```
int uthread_mutex_lock(uthread_mutex_t *mutex) {
  ……
  if (mutex is not available) {
    tcb->state = BLOCKED; //tcb is TCB for the current thread
    if (setjmp(tcb->context)==0) {    //save context
      call the scheduler to pick the next thread to run and context switch
    } else {
       When this thread resumes, it will back to here
    }
  }
  …….
}
```

And in the scheduler function, the new thread will be scheduled.

```
void scheduler_func() {
  //Select the next thread
  //next_tcb = TCB of the selected thread
  longjmp(next_tcb->context, some_non_zero); //resume execution for the next thread
}
```

For more details and usages of setjmp() and longjmp(), please refer to the man pages and examples on the web.

Create and exit thread: The hardest part would be starting a new uthread (in other words, creating the very first context). Creating the very first context for a new uthread is to allocate a stack and adjust PC to point to the start address of the thread's start function. First, the creating thread calls setjmp(*jmp_buf*). We can then manipulate *jmp_buf* for the very first context, and use *jmp_buf* as the new thread''s initial context. *Jmp_buf* for 64-bit x86 architecture is defined in glibc as follows,

```
#define JB_RBX    0
#define JB_RBP    1
#define JB_R12    2
#define JB_R13    3
#define JB_R14    4
#define JB_R15    5
#define JB_RSP    6
#define JB_PC     7
#define JB_SIZE (8*8)
```

We only need to modify the 6th element of *jmp_buf* for stack pointer and the 7th element of *jmp_buf* for PC. However, just assigning 6th and 7th elements of jmp_buf does not make it work as we expect. Glibc encrypts contents of jmp_buf for security reasons. We need to assign our stack pointer and PC after "mangling" the values as follows,

```
long pointer_mangle(void* p)
{
  long ret;
  asm volatile(
          "mov %1, %%rax; \n"
          "xor %%fs:0x30, %%rax;"
          "rol $0x11, %%rax;"
          "mov %%rax, %0;"
          : "=r" (ret)
          : "r" (p)
          : "%rax"
  );
  return ret;
}
```

Creating a new stack will be simply allocating a chunk of memory (e.g., through malloc()). Let's assume that each uthread has 8192 bytes of stack. Setting PC can be done by setting the 7th element of *jmp_buf* to the address of the thread's start function. However, there are some complications in passing an argument to the new uthread. In 64-bit x86 calling convention, the first function argument is passed in a register before the actual call is made, and setting PC for the start function implies that we missed a chance of passing the argument. One workaround is to (1) put the start function and the argument in TCB of a thread, (2) push indexing information of the TCB onto below the top of stack, and (3) assign a wrapper function as the start function to the 7th element of *jmp_buf*. When the wrapper function starts to run for a new uthread, it will get the start function and argument by reading the indexing information of the TCB which are pushed onto the stack in step (2) and invoke the real start function with the argument. The start wrapper function for a new thread will look like,

```
void *start_wrapper()
{
   void *ptr;
   //get the argument we pushed onto the stack
   asm volatile(
            "mov 0x10(%%rbp), %0; \n"
            : "=r" (ptr)
   );
   //ptr may contain a pointer to TCB or thread id
   // that can index the corresponding thread
   //here let's assume ptr is a pointer to TCB
   //the following is calling the start function with args
   return ((tcb_t *)ptr)->start(((tcb_t*)ptr)->args);
}
```

In 64-bit x86 calling convention, the caller pushes the return address onto the stack and starts executing the function. In our case, we can push the address of an exit wrapper function onto the stack, and the wrapper function will read the return value from the thread start function and call uthead_exit().

```
void exit_wrapper()
{
   long exit_code = 0;
   asm volatile ("mov %%rax, %0":
            "=r"(exit_code)::"%rax");
   uthread_exit((void *)exit_code);
}
```

Creating a thread can be summarized as shown in Figure 1.

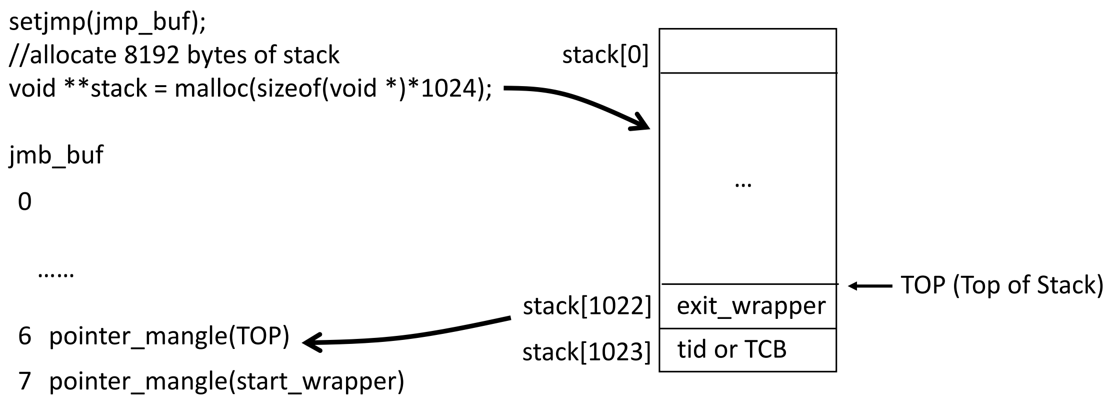


Figure 1.

Timer and signal: For implementation of timers, you will need to install a signal handler with sigaction() call. You may need to keep a timer for each thread since multiple threads may be waiting for the next periods. For the usage and examples of a timer, please see the man page of "*timer_create*".

Compile library: Executing the following commands in your Makefile shall generate "libuthread.a"

```
gcc -c "source files" -lrt
ar rc libuthread.a ".o files"
ranlib libuthread.a
```

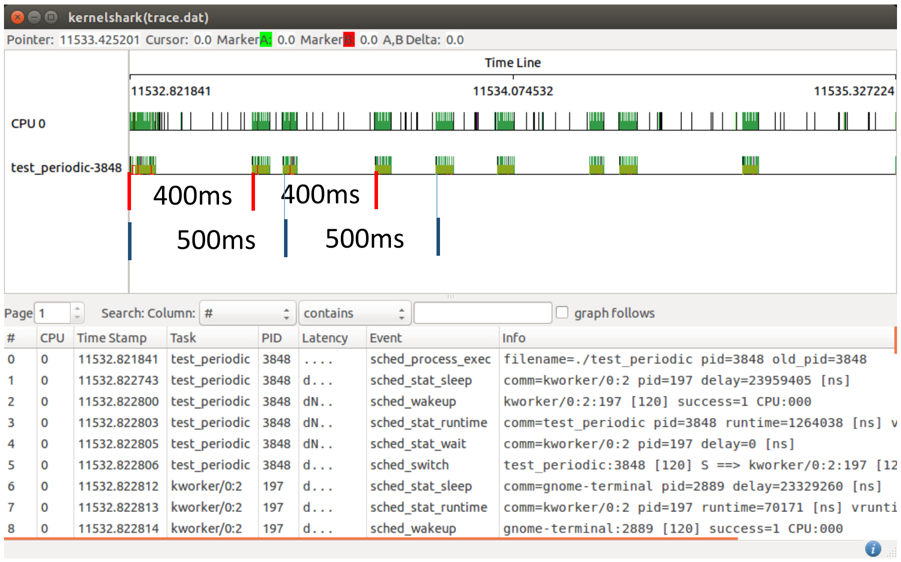
Expected results from the test applications

Five test applications are provided. We may use other test applications in addition to the five programs.

(1) "test_create_join.c": Since priority(thread1) > priority(thread2) > priority(thread0), threads will be executed in the order. The arguments and the returned values should be printed as expected.

```
thread 1 - arg = 1
thread 2 - arg = 2
thread 0 - arg = 0
thread 0 returned 0
thread 1 returned 11
thread 2 returned 22
```

(2) "test_periodic.c": Thread0's period is 500ms and thread1's period 400ms. So the periods should be observed with kernelshark.



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	11532.821841	test_periodic	3848	....	sched_process_exec	filename=./test_periodic pid=3848 old_pid=3848
1	0	11532.822743	test_periodic	3848	d...	sched_stat_sleep	comm=kworker/0:2 pid=197 delay=23959405 [ns]
2	0	11532.822800	test_periodic	3848	dN..	sched_wakeup	kworker/0:2:197 [120] success=1 CPU:000
3	0	11532.822803	test_periodic	3848	dN..	sched_stat_runtime	comm=test_periodic pid=3848 runtime=1264038 [ns] v
4	0	11532.822805	test_periodic	3848	dN..	sched_stat_wait	comm=kworker/0:2 pid=197 delay=0 [ns]
5	0	11532.822806	test_periodic	3848	d...	sched_switch	test_periodic:3848 [120] S ==> kworker/0:2:197 [12
6	0	11532.822812	kworker/0:2	197	d...	sched_stat_sleep	comm=gnome-terminal pid=2889 delay=23329260 [ns]
7	0	11532.822813	kworker/0:2	197	d...	sched_stat_runtime	comm=kworker/0:2 pid=197 runtime=70171 [ns] vrunti
8	0	11532.822814	kworker/0:2	197	d...	sched_wakeup	gnome-terminal:2889 [120] success=1 CPU:000

(3) "test_lock.c": There is a priority inversion in the program. When lock_attr = UTHREAD_MUTEX_ATTR_NONE, "g = 22" should be printed. When lock_attr = UTHREAD_MUTEX_ATTR_PI "g = 130" should be printed.

(4) "test_srp.c": When uthread_srp_enable() is not called, i.e., stack-based priority ceiling is not enabled, the middle priority thread performs the operation on the variable "g" first. Hence, the result "g = 20" should be printed. When uthread_srp_enable() is called, the lower priority thread performs the operation on "g" first. Thus, the result "g = 110" should be printed.

(5) "test_deadlock.c": Without uthread_srp_enable() call, the program is deadlocked. With uthread_srp_enable() call, the deadlock is prevented and the following message should be printed.

```
thread id: 0, period: 500ms
thread id: 1, period: 490ms
L took umutex[0]
L took umutex[1]
H took umutex[1]
H took umutex[0]
thread 0 returned 0
thread 1 returned 0
```

Reference

[1] User Mode Thread Library, Operating Systems. University of California, Santa Barbara. http://www.cs.ucsb.edu/~chris/teaching/cs170/projects/proj2.html

Due Date

This is assignment is due at 11:59pm on April 9.

What to Turn in for Grading

- Before submission, please make sure your project works on the Red-Hat Linux in the BYENG 217 lab. Personal demo with your own laptop/PC will not be acceptable. Also, make sure your library do not print out any debugging message.
- Create a directory *cse522-firstname-lastname*. The directory without any subdirectories should contain (1) Makefile, (2) readme.txt, and (3) source files and headers for the user thread library. Please do not include any test applications. Typing “make” shall generate “libuthread.a”.
- Comment your source files properly and make sure there will be no warnings when compiling your library.
- Compress the directory into a zip file named *cse522-assign3-firstname-lastname.zip*. Please note that, for convenience, we only accept zip files. Points will be deducted if the name conversion and the directory structure are not followed.
- Submit the zip archive to Blackboard by the due date and time.