Programming Assignment #3: MyThreads Package (a User Level Threads Package)

Due: see WebCT for the deadline (TA in-charge: Xing Shi Cai <xingshi.cai@mail.mcgill.ca>)

What is required as part of this assignment?

In this assignment, you will develop a user-level threads package. This provides a simplified replacement for Pthreads that you were using in the previous assignment. You develop Mythreads library which provides a preemptive multi-threading on Linux. Because it is built as a user library without any kernel support, it does not handle system calls properly (more on this later).

You will implement the thread package completely in C (i.e., no assembly code is necessary)[1]. It is important for you to recall that a thread is an independent execution of a program code. The multiple independent runs of the program share the resources that belong to the parent process.

To enable an independent run we need to provide a stack to each executing instance. Because multiple instances can share the same processor resource, we need to enable the multiple executing instances to multiplex on the processor. To enable multiplexing, we need to save and restore the executing context. This way the processor can be executing one instance and then switch over to another instance by saving the current executing context and restoring the next one. Fortunately, Linux provides system libraries for manipulating user thread contexts. In Linux, the user thread context, hold saved registers, thread execution stack, and blocked signals. See the man page for **`ucontext`** for more details. As part the ucontext, following functions are provided:

```
int getcontext(ucontext_t *);
int setcontext(const ucontext_t *);
void makecontext(ucontext_t *, void (*)(void), int, ...);
int swapcontext(ucontext_t *, const ucontext_t *);
```

Your thread package should provide the following or ***equivalent*** functions.

`int init_my_threads();`

This function initializes all the global data structures for the thread system. Mythreads package will maintain many global data structures such as the runqueue, a table for thread control blocks. It is your responsibility to define the actual data structures. One of the constraints you have the need to accommodate **`ucontext_t`** inside the data structures.

---

[1] You **cannot use** Pthreads or equivalent Linux/UNIX thread packages in implementing the MyThreads package.

`int create_my_thread(char *threadname, void (*threadfunc)(), int stacksize);`

This function creates a new thread. It returns an integer that points to the thread control block that is allocated to the newly created thread in the thread control block table. If the library is unable to create the new thread, it returns -1 and prints out an error message. This function is responsible for allocating the stack and setting up the user context appropriately. The newly created thread starts running the **`threadfunc`** function when it starts. The **`threadname`** is stored in the thread control block and is printed for information purposes. A newly created thread is in the RUNNABLE state when it is inserted into the system. Depending on your system design, the newly created thread might be included in the runqueue.

`void exit_my_thread();`

This function is called at the end of the function that was invoked by the thread. This function will remove the thread from the runqueue (i.e., the thread does not need to run any more). However, the entry in the thread control block table could be still left in there. However, the state of the thread control block entry should be set to an EXIT state.

`void runthreads();`

In MyThreads, threads are created by the create_my_thread() function. The create_my_thread() function needs to be executed by the main thread (the default thread of process – the one running before MyThreads created any threads). Even after all threads are created, the main thread will still keep running. To actually run the threads, you need to run the runthreads(). The runthreads() switches control from the main thread to one of the threads in the runqueue.

In addition to switching over to the threads in the runqueue, the runthreads() function activates the thread switcher. The thread switcher is an interval timer that triggers context switches every **`quantum`** nanoseconds.

`void set_quantum_size(int quantum);`

Sets the quantum size of the round robin scheduler. The round robin scheduler is pretty simple it just picks the next thread from the runqueue and appends the current to the end of the runqueue. Then it switches over to the new thread.

`int create_semaphore(int value);`

This function creates a semaphore and sets its initial value to the given parameter. The init_my_threads() function would have initialized the semaphores table and set the total number of active semaphore count to zero. You insert an entry into this table. Each entry of this table will be a structure that defines the complete state of the semaphore. It should also have a queue to hold the threads that will be waiting on the semaphore.

`void semaphore_wait(int semaphore);`

When a thread calls this function, the value of the semaphore is decremented. If the value goes below 0, the thread is put into a WAIT state. That means calling thread is taken out of the runqueue if the value of the semaphore goes below 0. In page 172 of the textbook, an implementation of the mutex_lock function for user level threads is shown. The semaphore wait is very similar. However, there are some differences as well. In MyThreads, you have preemptive multithreading which is implemented through signal based interrupts. You need to block the signals (more on the exact signal to block later) while manipulating the semaphore internal parameters. Remember to enable the signal based interrupts as soon as possible. Otherwise, the multithreading process will stop working!

Notice that the semaphore is denoted by an integer. It is actually an index into the active semaphore table maintained by the MyThreads library. The semaphore_wait() function needs to access the record corresponding to the given semaphore and then manipulate its contents.

```
void semaphore_signal(int semaphore);
```

When a thread calls this function, the value of the semaphore is incremented. If value is not greater than 0, then we should have at least on thread waiting on it. The thread at the top of the wait queue associated with the semaphore is dequeued and enqueued in the runqueue. The state of the thread is changed to RUNNABLE. Again check the implementation given in page 172 of the textbook.

```
void destroy_semaphore(int semaphore);
```

This function removes a semaphore from the system. A call to this function while threads are waiting on the semaphore should fail. That is the removal process should fail with an appropriate error message. If there are no threads waiting, this function will proceed with the removal after checking whether the current value is the same as the initial value of the semaphore. If the values are different, then a warning message is printed before the semaphore is destroyed.

```
void my_threads_state();
```

This function prints the state of all threads that are maintained by the library at any given time. For each thread, it prints the following information in a tabular form: thread name, thread state (print as a string RUNNING, BLOCKED, EXIT, etc), and amount of time run on CPU.

Proposed approach

Designing and implementing appropriate data structures is an important part of the assignment. You need to start with the thread control block table. Each entry in this table will look something like the following (intentionally incomplete):

```
typedef struct _mythread_control_block {
      ucontext_t context;
      char thread_name[THREAD_NAME_LEN];
      int thread_id;
      ….
} mythread_control_block;
```

The table itself needs to be directly addressable and can have an upper limit. So this naturally lends to an array type structure. The thread ID is an index into this table.

In addition to the thread control block table, you will have many queues: runqueue and wait queues associated with the semaphores. To implement these queues, you can use the libslack library (www.libslack.org). It is already installed in the CS lab machines. You can installed it in your machines from the project's website. Below is a simple program that illustrates enqueuing and dequeuing integers in libslack provided lists (equivalent to queues). I suggest that you use integers in the runqueue and wait queues to point to the thread structures. This simplifies the programming effort (particularly the memory management).

```
#include <slack/std.h>
#include <slack/list.h>

void main()
{
	List *l = list_create(NULL);
	int i = 100;
	int j = 200;
	int k = 300;
	// Append integers (enqueue operations)
	l = list_append_int(l, i);
	l = list_append_int(l, j);
	l = list_append_int(l, k);

	// Dequeue integers
	int q = list_shift_int(l);
	printf("Value of q = %d\n", q);
}
```

Suppose the above program is stored in **`ltest.c`** use the following command line to compile it.

```
gcc –o ltest ltest.c -DHAVE_PTHREAD_RWLOCK=1 -lslack
```

The libslack library provides support for many more data types. You can check their documentation for the full set of data types provided by the library and the associated API for creating and manipulating them.

Getting the user context management done is the important part of this assignment. The user context APIs provided by Linux greatly simplifies this part of the assignment. The **`makecontext`** man page in Linux provides an example program that shows how the user context management functions can be used to get the context, manipulate the context by setting the starting function, and swapping the context. Below is a portion of the program in the **`makecontext`** man page.

```
	if (getcontext(&uctx_func1) == -1)
		handle_error("getcontext");
	uctx_func1.uc_stack.ss_sp = func1_stack;
	uctx_func1.uc_stack.ss_size = sizeof(func1_stack);
	uctx_func1.uc_link = &uctx_main;
	makecontext(&uctx_func1, func1, 0);
```

The program fragment above shows how **`getcontext`** and **`makecontext`** can be used to create a thread. In the program fragment a thread is created to run func1 and the stack is set from memory that is allocated by the user. It should be observed that the setup also specifies the context that should be switched to once the function exits. You can use this information to determine how your system should behave when the thread completes. The program fragment below shows how swap the context from one thread to another (see also page 171 of the textbook).

```
	printf("main: swapcontext(&uctx_main, &uctx_func2)\n");
	if (swapcontext(&uctx_main, &uctx_func2) == -1)
		handle_error("swapcontext");
```

For more information and full listing of the sample program, see the man page for makecontext. A web copy of the man page can be found at http://linux.die.net/man/3/makecontext.

To drive the preemption process, you need a timer. You can use the setitimer() function for this purpose. It sends an SIGALRM. You can invoke a given function at each SIGALRM using a code fragment like the following:

```
struct itimerval tval;

sigset(SIGALRM, handler);

tval.it_interval.tv_sec = 0;
tval.it_interval.tv_usec = 100;
tval.it_value.tv_sec = 0;
tval.it_value.tv_usec = 100;
setitimer(ITIMER_REAL, &tval, 0);
```

This code fragment calls the handler every 100 nanoseconds. This is not the recommended strategy by Linux. You should use sigaction() instead of sigset() to make your code portable. For the assignment, sigset() is acceptable and it works!

The last piece of the puzzle is signal blocking. You need this for implementing the semaphores. Check pages 82-83 of the textbook for guidelines on how to handle signal blocking and unblocking. Remember you need to unblock the soonest to get multi-threading going with minimal interruption.

What to Hand in

Submit the following files separately:

1. A README file with:
    1. Your name and McGill ID number
    2. If you have not fully implemented all, then list the parts that work so that you can be sure to receive credit for the parts you do have working. Indicate any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit.
2. All source files needed to compile, run and test your code. If multiple source files are present, provide a Makefile for compiling them. Do not submit object or executable files.
3. Output from your testing of your program.

Suggestions

1. Make sure your source code could compile in Linux. Test them on the Trottier lab machines.

Part of the grading scheme you might want to know (to keep the TA happy):

1. 0 points if it does not compile.
2. A README text file must come with the assignment, especially if it does not implement all the features

or has some known errors. (In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.). Otherwise, the highest mark you can get is half of the total.
3. Name your parameters and functions properly by following one of the naming rules, such as Camel, Pascal, and even Hungarian notation (not recommended), to make your program clear enough to readers. (It could be you in 6 months later.). Otherwise, the highest mark you can get is half of the total.
4. Proper comments should be written in your program, unless your program is highly well-organized. Otherwise, the highest mark you can get is half of the total.
5. Use indentation correctly in your program, otherwise you will lose 20 percents out of 100.