CSE 531 – Distributed and Multiprocessor Operating Systems
Fall 2019
Project #2
Due Date: Oct 7th

Submission instructions at end.

Explanation: More explanation about this project is covered in lectures.

Groups:

Please do the project in groups of 2.
This rule *enforced* for all students.

Overview:

For this project you are to write routines that will allow a program to run, with multiple threads, using NON-preemptive scheduling on most Unix/Linux operating systems.

Step 1: Queues

The queue must be a circular doubly linked consisting of q-elements. A queue consists of a head-pointer and a set of q-elements. The first q-element in the queue is the first element in the queue. The head pointer points to the first element. See the figures for a better explanation. A q-element is a structure, consisting of a prev and next pointer, and a payload consisting of 1 integer.

The functions that you implement are:

1. **item = NewItem**(); // returns a pointer to a new q-element, uses memory allocation
2. **head = newQueue**() // creates a empty queue, that is the header pointer is set to null.
3. **AddQueue**(head, item) // adds a queue item, pointed to by “item”, to the queue pointed to by head.
4. item = **DelQueue**(head) // deletes an item from head and returns a pointer to the deleted item. If the queue is already empty, flag error.

Note: All the routines work on pointers. They do ***not*** copy q-elements. Also they to not allocate/deallocate space (except NewItem()). You may choose to implement a **FreeItem**(item) function – optional.

All the above routines and data structures are to be placed in 1 file, called “q.h”. Do not include other files into this file.

Test. Then re-test.
The Q routines are the cause on 95% of the problems with this and subsequent projects.

Step 2: PCB and context:

All the built-in types and functions used in this step come from a library. To get them, use the line:

#include <ucontext.h>

The queue items used for handling the threads is the TCB_t. A TCB_t is a structure that contains a next pointer, a previous pointer. A thread_id (int) and a data item called context, of the type “ucontext_t”.

Initializing a TCB can be done as follows (real code):

```
void init_TCB (TCB_t *tcb, void *function, void *stackP, int stack_size)
// arguments to init_TCB are
//   1. pointer to the function, to be executed
//   2. pointer to the thread stack
//   3. size of the stack
{
   memset(tcb, '\0', sizeof(TCB_t));       // wash, rinse
   getcontext(&tcb->context);              // have to get parent context, else snow forms on hell
   tcb->context.uc_stack.ss_sp = stackP;
   tcb->context.uc_stack.ss_size = (size_t) stack_size;
   makecontext(&tcb->context, function, 0);// context is now cooked
}
```

Put the code in a file called TCB.h and include this file in **q.h** so that q.h can use the type TCB_t. Each q-element now holds one struct of type TCB_t.

Step 3:

Declare two global variables: ReadyQ and Curr_Thread. ReadyQ points to a queue of TCBs and Curr_Thread points to the thread under execution (one TCB)

Write the routine to “start thread” and “yield” and the “run” routine which cranks up the works. You will need to include q.h, Put these routines (and the globals) in the file “**threads.h”**

```
void start_thread(void (*function)(void))
{ // begin pseudo code
     allocate a stack (via malloc) of a certain size (choose 8192)
     allocate a TCB (via malloc)
     call init_TCB with appropriate arguments
     Add a thread_id (use a counter)
     call addQ to add this TCB into the “ReadyQ” which is a global header pointer
  //end pseudo code
}
```

```
void run()
{   // real code
    Curr_Thread = DelQueue(ReadyQ)
    ucontext_t parent;     // get a place to store the main context, for faking
    getcontext(&parent);   // magic sauce
    swapcontext(&parent, &(Curr_Thread->conext));  // start the first thread
}
```

```
void yield() // similar to run
{  TCB_t Prev_Thread;
   AddQueue(Ready_Q, Curr_Thread);
   Prev_Thread = Curr_Thread;
   Curr_Thread = DelQueue(ReadyQ);
   swap the context, from Prev_Thread to the thread pointed to Curr_Thread
}
```

You may want to add a function to print the thread_id, given a pointer of type TCB_t.

Step 4:

Roll the code and light the fire.

Remember to include threads.h which includes q.h, which includes TCB.h, which includes ucontext.h
Write a few functions with infinite loops (put an yield in each loop).
Start them up, from main, using start_thread
Call run() and watch. // note try to write thread functions that are meaningful, use global and local variables
Call friends and family

SUBMIT:

Your project must consist of 4 files

1. TCB.h (uses ucontext.h)
2. q.h (includes TCB.h)
3. threads.h (includes q.h)
4. thread_test.c (includes threads.h) – must contain your name(s) in comments @ beginning

(make sure the compile command, “gcc thread_test.c” does the correct compilation).

All 4 files are to be ZIPPED into one zip or gzip file.

Names of 2 team members must be a comment in the file (submit only once please).

Upload the ZIP file to Canvas