CSE 330 - Operating Systems
Spring 2019
Project #1
Due Date: Feb, 7 2019, 11:59 PM

Strictly Individual Projects

Part 1: **(This part of the project is not graded.)**

Overview:

For this project you are to write routines that perform standard queuing functions. The functions work on multiple queues, and structure each queue as a doubly linked, circular list.

Data Structures:

A queue consists of a head-pointer and a set of q-elements.

A q-element is a structure, consisting of a prev and next pointer, and a payload consisting of 1 integer. The header is a pointer to the first element of the queue. *The head pointer is "null" if the q is empty.*

Functions:

The functions that you implement are:

1. **item = NewItem**(); // returns a pointer to a new q-element
2. **InitQueue**(&head) // creates a empty queue, pointed to by the variable head.
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
5. **RotateQ**(&head) // Moves the header pointer to the next element in the queue. This is equivalent to AddQ(&head, DeleteQ(&head)), but is simpler to use and more efficient to implement.

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.

Implementation and Testing

The routines should be implemented in C under the Linux operating system. If you are not familiar with Linux and/or do not have it installed. Please use a Ubuntu Virtual Machine. For more details/questions post requests on the discussion board.

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. The test routines can be in "proj-1.c" and this will include q.h and other standard header files.

Write test routines that thoroughly test the queue implementation. Use multiple queues. Pay special attention to deleting the last element of a q. Also make sure "RotateQ" is behaves properly (strange behavior of this routine means the insert/delete routines have bugs.)

Further warning: Bugs in the Q routines have been the #1 cause for strange errors in the project, always. Careful that you get it right, else things will go bump later.

Part 2: **(This part of the project is graded.)**

Overview:
Using the queuing routines (not all will be used) this project will implement the ability to run multiple functions as threads, using non-preemptive scheduling.

Step 1: TCB and context:
The queue items defined in your q.h file has to changed, to be of type TCB_t. The TCB_t and an initialization routine are provided in a header file tcb.h. Note that TCB_t has previous and next pointers along with an ucontext_t field to store a thread context.

There is a routine in the tcb.h file called init_TCB, which is used to initialize a TCB for a new thread. The arguments to init_TCB are:

```
1. pointer to the function, to be executed
2. pointer to the thread stack
3. size of the stack
```

Step 1 consists of understanding tcb.h and changing the q-element type in your q.h file to TCB_t

Step 2:
Write a routine called start_thread and put it in a file called "**threads.h".** Also routines defined in Step 3 will go into this file. You will need to include q.h into threads.h. The code for start_thread looks like:

```
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
     call addQ to add this TCB into the "RunQ" which is a global header
pointer
  //end pseudo code
}
```

Step 3:
Write the routines called "yield" and "run" which cranks up the works (and put them in threads.h). These routines are defined as:

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

void yield() // similar to run
{
   rotate the run Q;
   swap the context, from previous thread to the thread pointed to by RunQ
}
```

Step 4:
Write a driver program in a file called thread_test.c. Into thread_test.c.c include threads.h which includes q.h, which includes TCB.h, which includes ucontext.h.

Declare a global RunQ.

Write a few functions with infinite loops (put a yield in each loop). Note: try to write thread functions that are meaningful, use global and local variables

In main, initialize RunQ, start threads using the functions you defined using start_thread.

Call run() and watch.

---

tcb.h

```
#include <ucontext.h>

typedef struct TCB_t {
     struct TCB_t     *next;
     struct TCB_t     *prev;
     ucontext_t      context;
} TCB_t;



void init_TCB (TCB_t *tcb, void *function, void *stackP, int stack_size)
{
    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
}
```

SUBMIT:

Your project must consist of 4 files

1. TCB.h (uses ucontext.h) // this can be a copy of the file provided
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. The name of this file should reflect the name of the student (abbreviated, do not make it too long).

Submit in Blackboard.

Note: Grading is on Ubuntu. It is in your interest to make sure the program compiles and runs in the target test platform.