ME241 – Introduction to Computer Structures & Real-Time Systems

Lab Project 5: Mutual Exclusion

Prepared by:

Nathan Tang (20890522)

Eddy Zhou (20873917)

Submitted on:

1 December 2022

Part 1: Restructuring Thread Implementation

To simplify gizmOS, the previous implementation from Lab 3 was used. With a linked list implementation, the thread node has some old thread node members from Lab 3 and new thread node members for the mutex implementation in the following lab. Its members and its characteristics can be seen in Table 1 below. As well, the linked list members remain the same from Lab 3. Its members and characteristics can be found in Table 2 below. The members and their uses will be thoroughly explained in associated functions in later sections. All definitions can be found in `osDefs.h`.

Table 1. Members and Characteristics of Thread Node

`struct threadNode`	
`uint32_t* threadStackP`	Stores the thread stack pointer of the thread.
`void (*threadFunc)(void* args)`	Stores the pointer of the function to be run when the thread runs.
`int status`	Stores the status of the thread (such as if it is created, destroyed, etc).
`uint32_t runTimer`	Stores the amount fo time the thread has to run before it pre-empts.
`uint32_t mutexTimer`	Stores the time that the current thread is willing to wait on the mutex
`uint32_t mutexID`	Stores the mutex ID of the mutex that the thread is waiting on (if another thread is in possession of the mutex). Initialized/set to -1 whenever the thread is not waiting on any mutex.
`struct threadNode* next`	Stores the pointer to the next thread node in the linked list.

Table 2. Members and Characteristics of Thread Linked List

`struct threadLinkedList`	
`threadNode* head`	Stores the pointer to the head of the linked list.
`threadNode* tail`	Stores a pointer to the last thread (node) in the linked list
`int size`	Stores the size of the linked list.

Part 2: Mutex Implementation

To store the mutexes in gizmOS, a struct defined as `mutex` was created. To implement a data structure that stores the threads waiting on any given unavailable mutex, a queue was implemented. The members and characteristics of the mutex type, mutex queue node and the mutex queue can be seen below in Tables 3, 4, and 5 respectively.

Table 3. Members and Characteristics of Mutex

`struct mutex`	
`bool available`	Indicates whether the current mutex is available or not.
`mutexQueue* threadQ`	Stores a pointer to the mutexQueue associated with the following mutex

Note that the mutex holds a pointer to the `mutexQueue`, a queue which indicates the threads waiting on the mutex (and the order the threads are waiting on the mutex) if it is not available when the thread attempts to acquire the mutex.

Table 4. Members and Characteristics of Mutex Queue Node

`struct mutexQueueNode`	
`struct mutexQueueNode* next`	Stores the pointer to the next mutex queue node in the linked list.
`threadNode* tNode`	Stores a pointer to the thread that is waiting on the mutex.

Table 5. Members and Characteristics of mutexQueue

`struct mutexQueue`	
`struct mutexQueueNode* head`	Stores the pointer to the head of the queue.

In `_kernelCore.c`, an array of type `mutex` is declared with a maximum number of mutexes being `MAXNUMMUTEX` (5). The array is statically allocated and can later be accessed by functions in multiple files to manipulate mutexes.

Part 3: osMutexCreate()

A mutex can be created by calling the function `osMutexCreate()` defined in `_kernelCore.c`. When called, the function initializes the mutex at index `mutexIDCount` of the array. The initialization includes setting the mutex to initially available, and dynamically allocating a `mutexQueue` to store any threads that would potentially wait on this mutex in the future. The value of the `mutexIDCount` is incremented for any future mutex initialization and the value of the currently initialized mutex is returned as a success indicator. `osMutexCreate()` is called in `kernelInit()` when the kernel is initialized.

Part 4: osAcquireMutex() and osReleaseMutex()

A mutex can be acquired by calling the function `osAcquireMutex()` in `_kernelCore.c`. The function accepts two parameters; `uint32_t mutexID` which specifies the mutex that the thread wishes to acquire and `uint32_t timeout` which sepcifies how long the thread is willing to wait on the mutex.

The function first checks to see if the mutex is available (not in the possession of any other thread). If it is not, the function simply makes the availability of the mutex false and returns `ACQUIRE_SUCCESS` (indicating that the thread has acquired the mutex). Otherwise, the function dynamically allocates a `mutexQueueNode` which will hold the current thread which wishes to wait on the mutex. The `mutexQueueNode` is then added to the back of the queue and the thread's `timeout` is set to the argument passed in. The status of the thread is then set to `WAITING`, a state where the thread cannot run due to lack of resources and the mutex that the thread is waiting on is recorded by the thread using the `mutexID` member.

Then, `osYield()` is called to switch threads to a thread that can be scheduled. The `scheduler()` function has been reverted back to a round-robin implementation for simplification. It simply loops through the entire linked list of thread nodes to identify a thread that is not in `WAITING` state. When the original thread calling `osAcquireMutex()` is rescheduled by the scheduler, this thread will continue off from when it called `osYield()`within the `osAcquireMutex()` function. This is when `osAcquireMutex()` then checks if the thread has successfully acquired the mutex or has timed out. Should the thread have timed out, then `osAcquireMutex()` will return 1 to indicate that a timeout error has occurred. To check whether the thread has timed out or not, it checks whether the thread's timeout counter has hit <= 0. This means that the thread has failed to meet the timeout time and was set to `ACTIVE` by the `SysTick_Handler()`.

`osReleaseMutex()` simply checks to see if the mutex is available. If it is, then an error of 1 is returned to indicate that a mutex that is already available cannot be released. Otherwise, the mutex is set to true and a 0 success value is returned.

Part 5: SysTick_Handler()

Just as in previous labs, the `runTimer` of the currently running node is decremented on every call to enforce pre-emption if a task is taking too long. If the `runTimer` ever is 0, the it is reset and the node is set to `ACTIVE`. Pre-emption occurs and the scheduler determines the next node that is able ot run. A context switch occurs and the scheduled node becomes the new running node.

The `SysTick_Handler()` function also decrements the `mutexTimer` of a thread node if it is in the `WAITING` state and its `mutexTimer` is not set to `OS_WAITFOREVER` (defined in `osDefs.h` and its intended use is to make threads wait on a mutex forever). If the `mutexTimer` is 0, indicating that the thread can no longer wait on the mutex it was waiting on, the `mutexID` is obtained from the thread to identify which mutex it was waiting on. The queue of the identified mutex is searched through for the thread that no longer to wishes to wait on the mutex and the corresponding node is removed.

The `SysTick_Handler()` also checks if any of the mutexes are available (for each mutex stored in the array). If a mutex is found to be available and there is a thread waiting on the mutex in the queue, the thread is made to be `ACTIVE` and no longer waits on the mutex. This ensures that the next node to take the mutex is the one just made `ACTIVE`.

Part 6: Test Cases

A simple test case was used at first to determine the correct use of mutexes. This case was not included in the lab project requirements but was used to indicate correctness. In the testing `threadA` (which prints a large string of “a”) and `threadB` (which prints a large string of “b”), it was found that using mutexes the full string for each thread is printed before switching as shown in in Figure 1. Not using mutexes causes pre-emption, switching between both threads to print a sequence of both a and b as show below in Figure 2.

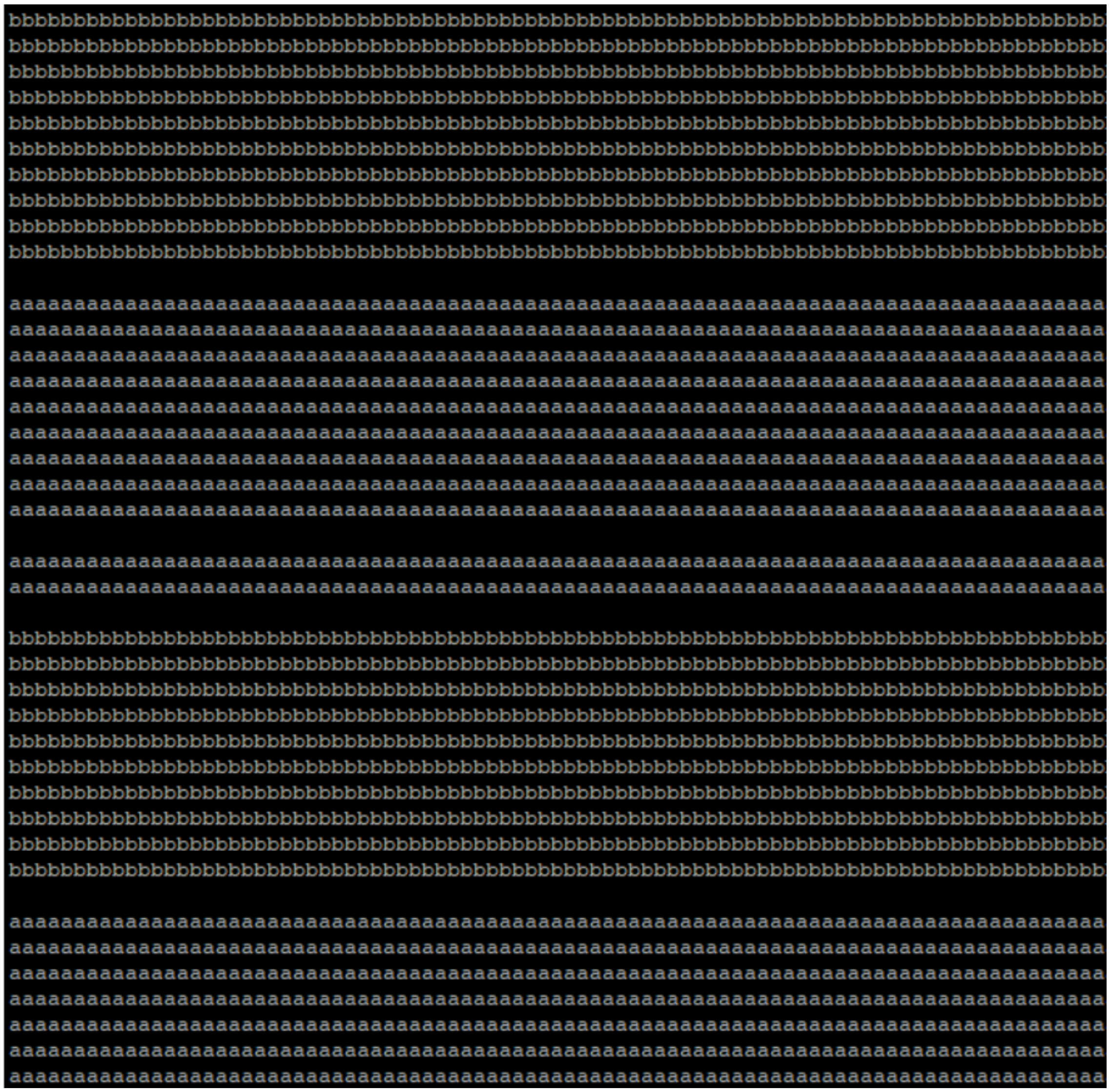

Figure 1. threadA and threadB executed with Mutexes

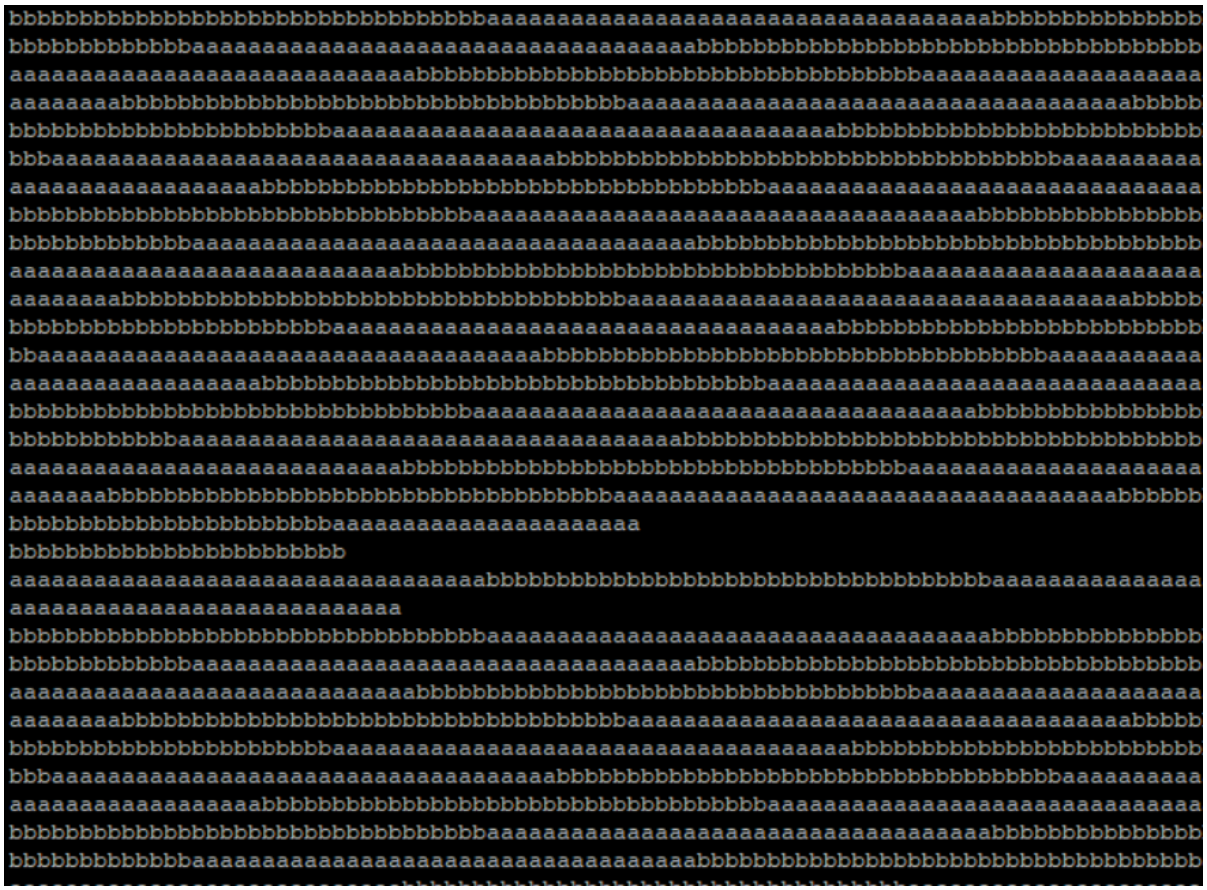

Figure 2. threadA and threadB executed with Mutexes

For the first test case, 3 threads were used to print a short string. Before printing the string, the thread is instructed to take mutex with ID 0. When printing the string is done, the same mutex is released and the thread cooperatively yields for the next. The threads are also set to wait `OS_WAITFOREVER` to wait on the mutex forever if it is unavailable. The output is shown below in Figure 3. The threads were not pre-empted in midst of printing the string which indicates proper operation of mutexes.

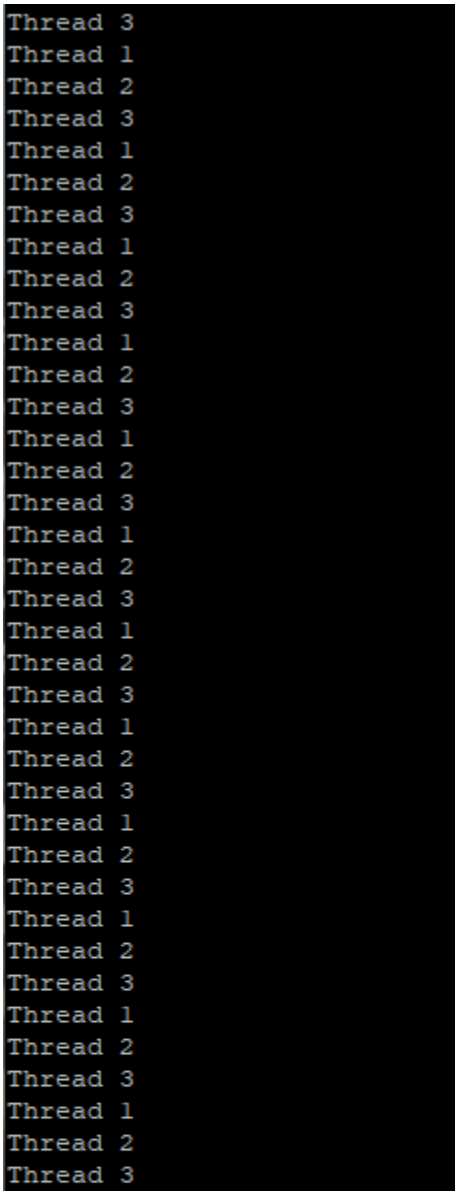

Figure 3. Thread1, thread2, and thread3 executed with Mutexes

For the last test case, two mutexes were used. The first with ID 0 is used to control access to a variable x while the second with ID 1 is used to control the LED usage on the board. The first thread simply increments x. The second sets the LEDs to 0x71 and the third sets the value of the LEDs to x%47. As seen in the additional print values, none of the operations on x are pre-empted and the whole operation finishes before the next is executed, indicating that the mutexes work. Figure 4 shows the console while Figures 5 and 6 show the LED status at different stages. Due to the nature of how fast the LEDs flash, a video was taken and a photo of 0x71 LEDs and x%47 LEDs were taken in midst of transitioning LEDs for evidence. Please note that the output indicates "x%49" but in fact, x%47 was used in the calculations and this is a minor error.

```
LEDs > x%49=21
LEDs > 0x71
x++: 69
LEDs > x%49=22
LEDs > 0x71
x++: 70
LEDs > x%49=23
LEDs > 0x71
x++: 71
LEDs > x%49=24
LEDs > 0x71
x++: 72
LEDs > x%49=25
LEDs > 0x71
x++: 73
LEDs > x%49=26
LEDs > 0x71
x++: 74
LEDs > x%49=27
LEDs > 0x71
x++: 75
LEDs > x%49=28
LEDs > 0x71
x++: 76
LEDs > x%49=29
LEDs > 0x71
x++: 77
LEDs > x%49=30
LEDs > 0x71
x++: 78
LEDs > x%49=31
LEDs > 0x71
x++: 79
LEDs > x%49=32
LEDs > 0x71
x++: 80
LEDs > x%49=33
LEDs > 0x71
x++: 81
LEDs > x%49=34
LEDs > 0x71
```

Figure 4. Console Output for threads used to increment x, access x, and set LEDs with Mutex

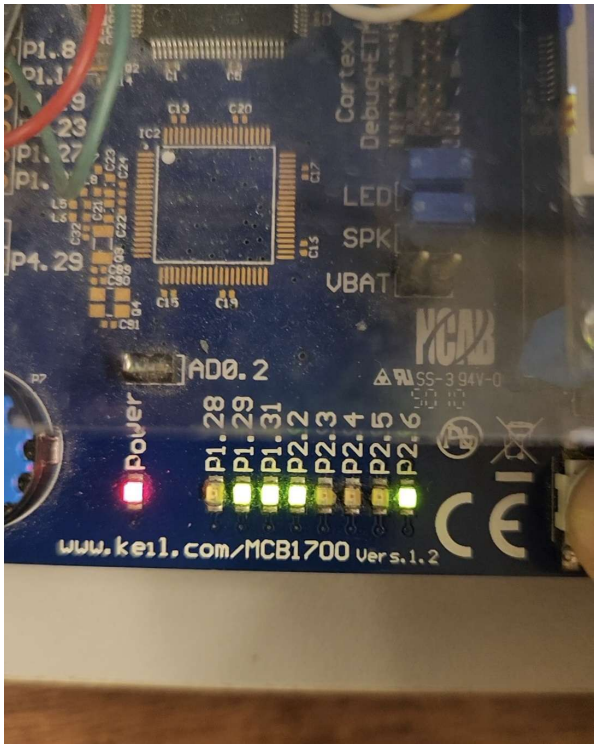

Figure 4. LED output for setting LEDs to 0x71

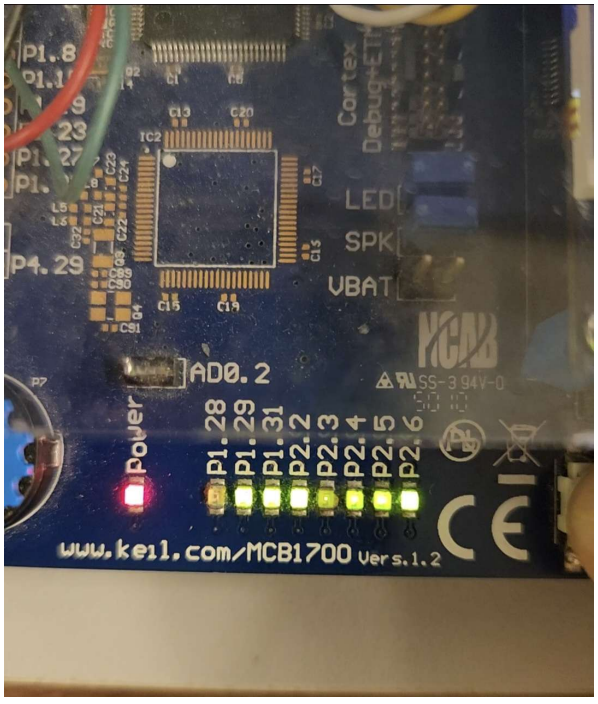

Figure 4. LED output transitioning for setting LEDs to x%47