Eldor Bekpulatov (eb654)
Carol Zhang (cz233)

Lab 4: Locks

Intro

A lock is an abstract synchronization tool used to limit access to a resource or data when multiple active processes are competing for it. In this lab we implemented locks build on top of what we implemented in the last lab with concurrency and time-sharing using the FRDM-K64F microcontroller. Lock we implemented limits the number of processes that can have access to a resource or data at any moment in time to one. By implementing lock structure and 3 simple abstract functions like init_lock(), lock() and unlock(), and we were able to successfully complete this lab.

Design and Implementation

To begin, we focused on out structures. We needed a good way to encapsulate a way to assign a series of processes to a specific lock, and they would have to be in some orderly fashion. A good way to implement such structure was using queue in lock structure. This was accomplished having linked list of process structures we implemented in our last lab. These process nodes in linked list would have to be mutually exclusive from our general process queue, and so to indicate if the process is the queue of the lock or the process queue, we added an additional attribute to out process_state structure called blocked. This attribute would be set to one to indicate if it has been added to lock's queue, zero if it had been added to process_queue. Lock structure should also have an attribute to indicate if there are any processes currently holding the lock or waiting for the lock. We added an attribute state to lock which is set to zero when no process is holding or competing for the lock, one otherwise. Figure 1 below shows the modified and new structures with necessary global variables pointing to them.

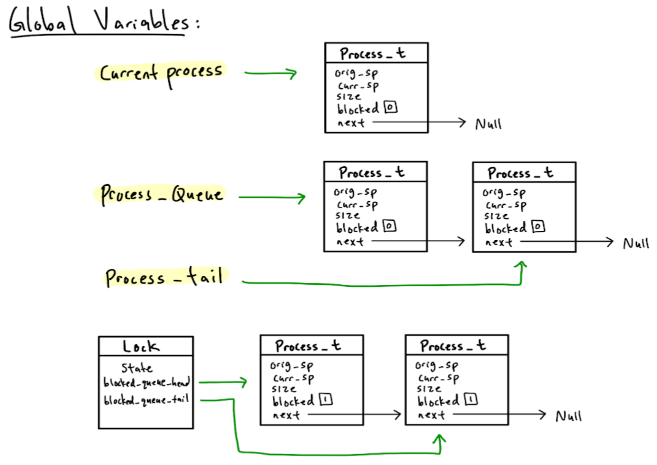
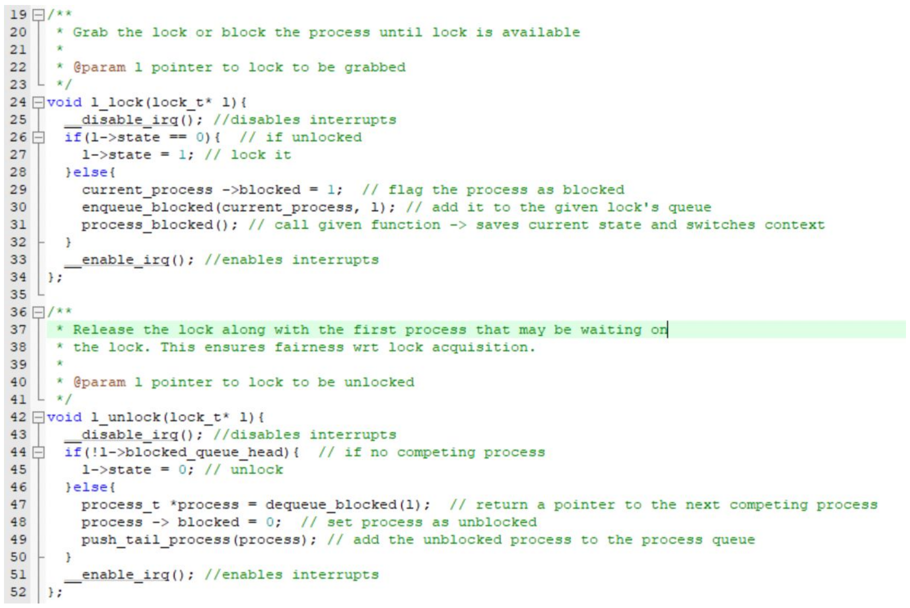


Figure 1. Illustrates the general queue of processes as well as the competing processes for a single lock. We call the head of the competing processes the

Once finalized our changes to our structures, we then had to implement our 3 functions. init_lock() was extremely straightforward. Lock would be initialized as unlocked (0) and there would be no competing processes for the lock, so out lock queue would be empty. lock() and unlock() were also simple in logic and symmetric in many ways. Both had to be done in an atomic cycle, so we disabled interrupts at the beginning of the functions and re-enabled them before exiting the functions. For the lock(), first we checked if the lock is unlocked if unlocked we simply lock it, else we update the current process's blocked flag and add the current process to the lock's queue and call the process_blocked function, which basically lets another process from process_queue take over the cpu. For unlock(), we check if there are any competing processes in the lock's queue, if its empty we release the lock, else we dequeue the first element from the lock's queue, update it's blocked flag, and put at the back of the process queue. This allows for fairness with respect to other non-competing processes. Enqueuing and dequeuing were practically given by the provided lab files, we simply renamed the variables. Figure 2 below shows the code snippets for the lock() and unlock() functions.

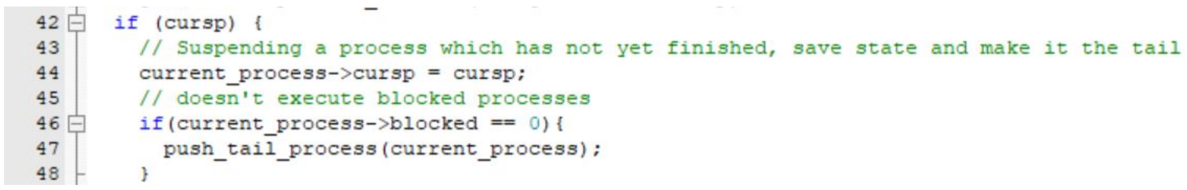


```
/**
 * Grab the lock or block the process until lock is available
 *
 * @param l pointer to lock to be grabbed
 */
void l_lock(lock_t* l){
  __disable_irq(); //disables interrupts
  if(l->state == 0){  // if unlocked
    l->state = 1; // lock it
  }else{
    current_process ->blocked = 1;  // flag the process as blocked
    enqueue_blocked(current_process, l); // add it to the given lock's queue
    process_blocked(); // call given function -> saves current state and switches context
  }
  __enable_irq(); //enables interrupts
};

/**
 * Release the lock along with the first process that may be waiting on
 * the lock. This ensures fairness wrt lock acquisition.
 *
 * @param l pointer to lock to be unlocked
 */
void l_unlock(lock_t* l){
  __disable_irq(); //disables interrupts
  if(!l->blocked_queue_head){  // if no competing process
    l->state = 0; // unlock
  }else{
    process_t *process = dequeue_blocked(l);  // return a pointer to the next competing process
    process -> blocked = 0;  // set process as unblocked
    push_tail_process(process); // add the unblocked process to the process queue
  }
  __enable_irq(); //enables interrupts
};
```

Figure 2. Code for locking and unlocking functions

Lastly, we had to modify our two functions from lab3 to allow for locking capability while time sharing a single processor. There were two changes mainly, first was the create_process(). Since, we had updated out process state structure, we needed to initialize the new attribute (blocked) with a zero when creating the process node. Second, change was made to the select_process() function. Before we were updating the currsp of current process and re-enqueueing the process back into the process_queue of it had not yet finished. Now, a process could have been updated to blocked process, meaning it was trying or using a resource and it was moved to the lock's queue, so we don't need to add the blocked processes to the process queue. Figure 3 shows the code snippet of the changes made to the process_select() function.



```
if (cursp) {
  // Suspending a process which has not yet finished, save state and make it the tail
  current_process->cursp = cursp;
  // doesn't execute blocked processes
  if(current_process->blocked == 0){
    push_tail_process(current_process);
  }
```

Figure 3. Conditional statement to check if the current_process can be enqueued to process_queue based on its locking parameter

Testing

We created two unique test cases to ensure that our implementation is correct. The first test case runs two processes that are protected with locks and a third unprotected process. We locked on-off toggles of red and blue LED's and didn't lock the green's. Since both blue and red are locked, we expect to them toggle independently without ever interfering with each other, whereas green led can collide with red or blue. Each color should toggle on and off five times each. Outcome was exactly what we expected. This showed that our lock implementation can run both locking and non-locking processes concurrently.

The second test case again runs three processes but now however with two locks. Process A and Process B will not share a lock, however Process C must have both locks to complete its critical section. We designed this test case in such a way that red LED is locked by a different lock than blue one. However, green LED must have both of their locks to toggle. Our expected result was that red and blue LEDs will always collide and green will never collide with either color. The result was just that. This test showed that our implementation can handle more than a single lock.

Documenting/writing

Documenting the lab was done through Google Docs, where each member practiced live editing and communicated through tools such as text comments. Any updates were recorded, reviewed and annotated by each member. This method of documentation proved to be highly effective.

Work distribution

Implementing, debugging, testing and writing the lab report was done by Eldor. Carol participated in during the implementation of the lab and provided helpful digital diagram shown in Figure 1.