ECE-350 L2 Report

Tony, Shen, t48shen
Moyez Farook, Mansoor, mfmansoo
Anthony Natale, De Luca, andeluca
Anson, Wan, a23wan

DATA STRUCTURES AND ALGORITHMS

For this lab, we reused structures from Lab1 for memory allocation, deallocation.

TCB

The task control block is a data structure that contains various information about the task it is associated with. In this lab each task has a privilege, priority, task id, and state which are all variables within the TCB structure. There are also references to the previous and next TCB for use in the queue data structure described later.

We also added a few elements to the TCB structure to facilitate task/memory functions:

```
typedef struct tcb
{
    struct tcb *prev; /**< prev tcb, not used in the starter code
    struct tcb *next; /**< next tcb, not used in the starter code
    U32 *msp;         /**< kernel sp of the task, TCB_MSP_OFFSET = 8
    U8 priv;          /**< = 0 unprivileged, =1 privileged,
    U8 tid;           /**< task id
    U8 prio;          /**< scheduling priority
    U8 state;         /**< task state

    // ADDED for tsk_get:
    void (*ptask)();  /**< task entry address
    U32 *usp;         /**< user stack pointer
    U32 u_stack_size; /**< user stack size in bytes

    // ADDED for kernel
    U32 *u_sp_base; // Base address for the USP

    U32 *u_sp_end; // End address for the USP
} TCB;
```

Queue

The queue data structure is a first-in first-out structure that is used to organise the various tasks that are in the READY state. For each priority, there is a queue that holds all the tasks that are ready to be run. For this Lab, there are 5 priorities including the null task priority and thus there are 5 ready queues created to organise the tasks of those 5 priorities. In the null task case, it is always alone in the null task priority queue which is the lowest priority. In our

implementation we have both an array called ready_TCB_queues as well as dormant_TCB_queue. This allows us to quickly reuse any task blocks which have been set to dormant.

The queue data structure was implemented by using a singly linked list. This linked list is a list of TCB structures for each of the tasks with references to the next TCB in the list. There is also a HEAD and TAIL reference to the beginning and end of the queue which is used in conjunction with the queue's functions when placing a task back into the queue after preemption or yielding.

Scheduler Algorithm

The scheduler returns the TCB of the task to be run next. The algorithm that does this checks the ready queues in highest priority order and returns the TCB that is at the HEAD of the highest priority queue that is not empty. If all the priority queues are checked to be empty, the null task TCB will always be returned as the null task is always in it's own null task priority queue. The task function of the TCB that is returned is then run as the next task. After most tsk_functions, we will call either k_tsk_run_new to select a new task to run with the help of the scheduler. Some functions will call k_tsk_run_new_helper instead, which allows us to choose between reinserting the TCB at the front (preemption) or back (yield) based on whether or not the switch was voluntary.

TEST SUITE DESCRIPTIONS

Test Suite 1 - ae_tasks1_G10.c

This test ensures that we can allocate the max number of tasks, have them all exit and then do it over again. We decided to test for this as it will find errors in the creation and clean up of tasks. This includes things like forgetting to clean up memory, forgetting to reset information, and forgetting to move tasks to a dormant queue. The test case will be described in pseudocode below to give a better idea of how we achieve this.

```
create priv_task1 with a priority of HIGH

// NOTE: this task should not ever run
create task1 with a priority of LOWEST


priv_task1():
    for i in range(NUM_TEST_RUNS):
        run_test()
```

```
    test_exit()

run_test():
    set itself to HIGH priority

    update exec sequence

    for i in range(TASKS_TO_CREATE):
        // MEDIUM priority so it does not preempt
        create dummy task with a MEDIUM priority
        check to ensure that create returns okay

    // This works because we push new items to the back of the queue
    // and they all have the same priority
    set the expected order to be the order that the tids are created

    // The dummy tasks run
    set itself to LOW priority

    // All tasks have finished
    check to ensure the execution sequence is correct

dummy_task():
    update exec sequence
    exit the task
```

Test Suite 2 - ae_tasks2_G10.c

This test ensures that a task can successfully change its own priority by calling tsk_set_prio and then resume running once it becomes the highest priority task again as well as checking that we cannot create more than the max number of allowed tasks. It does this by creating a task of high priority which will then create a copy of itself and then change its own priority so that the copy it created will begin to run and this will repeat until the maximum number of tasks have been created in which the last task attempting to create a new task should fail and then finish its own work and exit. After this all the tasks in order of priority will finish their work and exit, concluding the test.

Test Suite 3 - ae_tasks3_G10.c

This test checks to ensure that tasks can use the memory allocation from IRAM1 correctly. To do this we have two tasks that pass execution back and forth that allocate and deallocate memory, ensuring that everything is working as expected. To illustrate how the code works pseudocode is provided below.

```
create priv_task1 with a priority of HIGH
create task1 with a priority of MEDIUM


priv_task1():
    // Case 1
    void * ptr_1 = mem_alloc(1024)
    ensure the pointer is not null

    // Case 2
    ensure the number of free blocks is 2

    // Case 3
    mem_dealloc(ptr_1)
    ensure deallocation returns correctly

    // Case 4
    // this should be the entire memory
    ensure the number of free blocks is 1

    // Case 5
    void * ptr_2 = mem_alloc(1024)
    ensure the pointer is not null

    set the priority of this task to MEDIUM

    // Case 6
    // Because set_prio is considered involuntary
    ensure that this task has not been preempted

    // task1 should run after this
    tsk_yeild()

    // Case 11
    ensure that task1 has run at this point
```

```
    // Case 12
    // We should receive a non-null pointer
    void *ptr_3 = mem_alloc(1024);
    ensure the pointer is not null


    // Case 13
    ensure the number of free blocks is 0

    // Case 14
    mem_dealloc(ptr_2);
    ensure that we can deallocate with no problem

    // Case 15
    // It should have one memory block left
    ensure the number of free blocks is 1

    // Case 16
    // It should deallocate with no issue
    mem_dealloc(ptr_3);
    ensure the deallocation is successful

    tsk_exit();

task1():
    // This is done to ensure the preemption is done correctly
    // when setting priority
    set a global flag indicating task1 has run

    // Case 7
    ensure the number of free blocks is 2

    // Case 8
    // we should not be able to allocate the entire ram
    void * ptr_1 = mem_alloc(4096)
    ensure the pointer is null

    // Case 9
    // We should be able to allocate half of the ram
    void *ptr_2 = mem_alloc(2048);
    ensure the pointer is not null
```

```
// Case 10
// There should be 2 free blocks
ensure the number of free blocks is 2

// Move back to the other task
tsk_yield();

// Case 17
// It should deallocate with no issue
mem_dealloc(ptr_2);
ensure we deallocate successfully

// Case 18
ensure the number of free blocks is 1


test_exit();
```

Test Suite 4 - ae_tasks4_G10.c

This test checks the functionality of our tsk_get function and ensures that unprivileged tasks can only change the priority of unprivileged tasks. We do this by creating a privileged task and an unprivileged task and start by running the privileged task. Then the privileged task lowers its own priority to lower than that of the unprivileged task, allowing it to run and finally the unprivileged task will attempt to change the priority of the privileged task and fail, create another unprivileged task, change that tasks priority and succeed, and then finish running in accordance to priority. Just before the test ends we call tsk_get and run a check to make sure that the information retrieved by tsk_get matches that of the currently running task and then the test concludes.