University of Waterloo

ECE 350 Lab2 Report

Group 12

Group Members:

Geoffery Qin (c22qin)

Rebecca Zhou (yx5zhou)

Kehan Xing (k5xing)

Yiru Wu (y645wu)

Lab 2 Task Management

Data Structure

Important Variables

variable name	type	purpose
ready_queue	Array of QUEUE	This array is used to order tasks by priority. QUEUE is used to store same priority tasks; array is used to separate different priority. This array will always contains all the ready priorities that are currently in the system
g_tcbs	array	This array is used to store tcb of tasks with pid as index
QUEUE	Array of TCBs	This structure is used to store all tasks in the same priority that are ready

Functions

Helper Functions

<priority_queue.c>

TCB pop_head (QUEUE* queue)*

This function will pop and return the first tcb in specific priority queue. After, we set both the prev and next pointer of the TCB to null, then set its state to running.

TCB pop_tail (QUEUE* queue)*

This function will pop and return the last TCB in specific priority queue. After, we set both the prev and next pointer of the TCB to null, then set its state to running.

int pop_task (TCB t_task, QUEUE* queue)*

This function will remove the specific task from the ready_queue. This task is specified through the parameter TCB* t_task.

int push_head (TCB t_head, QUEUE* queue)*

This function will the push the TCB passed in through the parameter to the beginning of a specific priority queue. The next pointer of new head will point to the old address that the HEAD pointer of the queue was pointing to and the prev pointer will point to HEAD pointer of the queue.

int push_tail (TCB t_tail, QUEUE* queue)*

This function will the push the TCB passed in through the parameter to the end of a specific priority queue. The prev pointer of new tail will point to the old address that the TAIL pointer of the queue was pointing to and the next pointer will point to TAIL pointer of the queue.

<k_task.c>

int priority2order(U8 priority)
U8 order2priority(int order)

These two functions convert between the priority macros and the order macros of the ready queue.

Algorithm Functions

<k_task.c>

*int rtx_init (RTX_SYS_INFO *sys_info, TASK_INIT *tasks, int num_tasks)*

This function initializes the priority queue and pushes the initial tasks into the queue.

*TCB *scheduler(void)*

When this function is called, it will always pop and return the first TCB in the highest priority queue that contains a task. There is no task existing in the system, NULL_TASK will be returned.

*int tsk_create(task_t *task, void (*task_entry)(void),U8 prio, U32 stack_size);*

This function is called to create a new user-space unprivileged task which creates a TASK_INIT structure and update corresponding TCB in the system. The function first check if stack_size parameter is larger or equal to PROC_STACK_SIZE, and will allocate PROC_STACK_SIZE if stack_size is less than PROC_STACK_SIZE. Then, the function will check g_tcbs[] array to find a DORMANT task entry to assign the new created task's tid. The function might returns -1 and set errno to EAGAIN if the system has reached maximum number of tasks. It also check if the parameter prio is a valid priority value defined in the system, and it will return -1 and set errno to EINVAL if the parameter prio is invalid. The function will then create a TASK_INIT structure for the new task and set prio, priv, task_entry, tid, and u_stack_size. Subsequently, the function will call tsk_create_new to dynamic allocate space for user-stack and update corresponding TCB's values and registers. The function might return -1 and set errno to ENOMEM if there is not enough memory to support the operation. Once the k_tsk_create_new returns RTX_OK, the function will push the new task's TCB to the tail of the ready_queue corresponding to its priority level and stores the tid in the address pointed by *task parameter. Lastly, the function checks if the new created task has a higher priority than the current running task, and will push the current running task to the head of its ready_queue and perform a task switch if the new created task has a higher priority than current running task.

int k_tsk_yield (void)

This function is called when a task voluntarily relinquishes the CPU. The task will be pushed to the back of the ready queue of its priority, and then we use the scheduler to find the task with the highest priority in the ready queue. The program will switch to run the found task after this.

void k_tsk_exit (void)

This function is used to exit a task. The state of the task will be changed to DORMANT so it will not be found in the ready queue anymore. The user stack of this task will be deallocated from our memory pool. Then the scheduler will find the task with the highest priority, and the program will switch to run the found task.

int k_tsk_set_prio (task_t task_id, U8 prio)

This function is used to set the priority of tasks in order to organize the order for yielding. Input the task_id and assigned priority, the function will compare the assigned priority and current task' priority to decide whether suspends current task and switches to high priority task. Then it changes the order of assigned task in ready_queue to update its priority.

*int k_tsk_get (task_t tid, RTX_TASK_INFO *buffer)*

This function is used to return the information about specified task. The function will find the tcb of task by tid and return the information stored in tcb.

Test Cases

`ae_tasks1_G12.c`

This test suite creates 3 tasks during initialization – 2 privileged tasks and 1 unprivileged task. Overall, it creates 5 tasks in total. It mainly tests the functionality of `tsk_get_prio.` The test suite covers the following scenarios.

- When an invalid ID or unknown level is entered
- When an unprivileged task tries to set the priority of a privileged task which is not permitted
- When a privileged task sets an unprivileged task's priority
- When an unprivileged task sets another unprivileged task's priority
- When a privileged task sets itself another priority
- When an unprivileged task sets itself another priority
- When a privilege task sets another privileged task's priority

If everything passes, we should get a score of 13/13 at the end.

ae_tasks2_G12.c

This test suite creates 2 tasks during initialization - 1 privileged task and 1 unprivileged task. Inside of the unprivileged function, it will try to create another unprivileged task and get its information using `tsk_get()`. Afterwards, it will try to call `tsk_set_prio()` on the newly created task to change its priority, then try to get its information again using `tsk_get()` . We wang to make sure that we are getting the up-to-date value of our tasks. The test suite covers the following scenarios.

- Calling `tsk_get()` on a newly created task
- Change the priority of the task and acquire the same information again to check whether we are geting the updated information
- Calling `tsk_get()` on a function itself
- Change the priority of the task itself and acquire the same information again to check whether we are getting the updated information

ae_tasks3_G12.c

This test suite wants to test the edge case when the whole memory pool is allocated or the whole tcb array list is occupied. It creates two tasks during initialization (priv_task and dumy task1), and they are both size of 0x2000. Then we create another two tasks inside of priv_task with size 0x2000 and size 0x1000. Since the null task takes a size of 0x200, we cannot allocate another task with 0x1000. Based on this scenario, we tested the following cases:

- The tsk_create function fails if we want to create another task with size 0x1000
- The tsk_create function succeeds if we want to create another task with size 0x1000 after exiting previous tasks
- The tsk_create function succeeds if we want to create the 15th task. This test checks if the previous failed task is allocated in the tcb array by mistake.
- The tsk_create function fails if we want to create the 16th task since we can only create MAX_TASKS-1 tasks
- The tsk_create function succeeds if we want to create the 16th task after exiting a previous task
- All the created tasks exit by order

If everything passes, we should get a score of 6/6 at the end.

ae_tasks4_G12.c

This test suite creates 2 privileged tasks during initialization. Overall, it creates 3 tasks in total. It tests tsk_get_prio, tsk_create, mem_alloc, mem_dealloc, tsk_yield, tsk_exit, and data safe. The test suite covers the following scenarios.

- When set priority during initialization, high priority task will run first

- After set priority of another task to the same priority as current task, yield could switch between them.
- Data is safe after switching tasks
- When exit, highest priority task will run according to the scheduler

If everything passes, we should get a score of 12/12 at the end.

ae_tasks5_G12.c

This test suite tests memory allocation that are not power of 2. There are three tasks created in total and they will each have a user stack size that is not a power of 2. Inside of each task, we examine whether or not the user stack size is the closest power of 2 of the requested memory allocations. If is not, our test would fail. There are 4 tests in total and they all check for the same variable inside the tcb – user-stack-size.