Lab 2 Report

Description of Functions

k_tsk_create

1. We start k_tsk_create by error checking against cases such as
    - If the given task pointer is null
    - If the task entry pointer is null
    - If we have already reached the max number of tasks
    - If the stack size to create is too small
    - If the stack size to create is too big
    - If the stack size is not 8 byte aligned
    - If the given priority is invalid

2. After the initial error checks, we search for a free TCB

3. We then set up the the appropriate task information such as priority, stack size, entry or if its privileged

4. Call k_tsk_create_new

5. Increment the number of active tasks

k_tsk_exit

1. Set the current task's state to be 'dormant'
2. Set the current task to what the scheduler feels is appropriate
3. Deallocate the previously running tasks stack
4. Decrement the number of currently running tasks
5. Pop that task id off the heap we use as a priority queue
6. Run an error check to see if we should actually keep running the old task
7. Set the state of the current task to 'running' if needed
8. Switch to the old stack if needed

k_tsk_set_prio

1. Run checks for the following cases
    1. Invalid priority
    2. if the given task is already dormant
    3. if the current tasks priority is the same we can return early
    4. If the task id is valid
2. If the current task is privilidged, let it set whatever priority it wants
3. If the given task is privilidged, error
4. otherwise set the priority as it was requested
5. Update our heap(which acts as a priority queue) with the newly set priority
6. Call k_tsk_yield so that a new task can run if needed

k_tsk_get

1. Run checks for the following cases
    1. If the given buffer is null
    2. if the given taskID maps to a dormant task
    3. If the given taskID is valid
2. populate the buffer with the correct information from the global list of TCBs
    1. this includes setting kernal and user stack pointers

Scheduler

Our underlying data structure that handles the scheduling of tasks is a priority queue. Under the hood, this is implemented using a binary heap. This lets us pop off the highest priority runnable task in O(1) while also being able to insert items in O(logn).

The scheduler function extracts the TID off the top of the heap (priority queue). It also checks if the current task should keep running. In the case that a new task needs to be run, it removes the next runnable task from the heap and then inserts the currently running task.

There are a few helper methods that we use to make working with the heap easier to manage:

find_value

- This method iterates through the heap to find a given task id
- this operation runs in o(n) but given that the heap is at most, MAX_TASKS(16), the linear time search is acceptable

compare

- a simplified comparison to compare the priority of two TCBs that get stored in the heap
- this is used when "heapifying"

Swap

- Swaps two tcb's pointers

left/right child

- returns the left or right child of a parent using pointer arithmetic to save on using space on pointers

Parents

- returns the parents of a node using pointer arithmetic to save on using space on pointers

Increase key

- iteratively moves a TCB up the heap until it is in the correct position

Decrease key

- Iteratively pushes a TCB deeper into the heap until it is in the correct position

Insert

- inserts a TCB into the heap in the correct position

ExtractMax

- fetches the highest priority runnable task

Maximum

- Extracts the TID of the highest priority runnable task

Reset priority

- sets the priority of a given TCB and then moves it to the correct position in the keap

RemoveID

- removes a given TCB from the heap

Mandatory Testing Scenarios

Creating and terminating user-mode tasks

Sanity test

1. Create task A with a variety of stack sizes
2. let the task run and use the stack
3. Call tsk exit

Create and destroy multiple tasks

1. Create tasks until the max number of tasks are created

2. get tasks for all TIDs to validate that get works as expected
3. assert that calling tsk_create fails
4. tsk_exit
5. tsk_create should re-use the TID

Scheduling tasks with the same priorities

1. Create tsks all with the same priority
2. Pop off the tasks using the scheduluer
3. Assert that the popped off tasks are in the same order they were created

Scheduling tasks with different priorities

Sanity tests

1. assert using PRIO_NULL fails
2. assert using an invalid TID fails
3. assert set prio on dormant task fails
4. assert set prio on kernal tasks fails
5. assert a simple test with valid set pio passes and can then have tsk_get be called

Ability to deallocate memory that is owned

1. Create a task that owns data and calls malloc
2. From that task, try to dealloc the date
3. asset success

No ability to deallocate memory that is not owned

1. Create a task that owns data and calls malloc
2. create a task that try to dealloc the first task's data

3. assert that dealloc fails

Getting correct information on tasks

1. Create a task
2. call get on the task and verify in memory that the correct task was fetched
3. validate that stack size, task id, priority, state, privilege, stack pointers are correct
4. repeat the above for multiple tasks created at once

Extra Testing Scenarios

tsk create failure tests

1. Create a task with a stack size too small
2. assert failure
3. create task with memory that is not 8 byte aligned
4. assert failure
5. create tasks with invalid priority input
6. assert failure
7. create tasks with null priority
8. assert failure
9. create tasks with invalid tid
10. assert failure
11. create tasks with invalid task function
12. assert failure

tsk get failure tests

1. create task with ID 0
2. assert failure
3. use a null buffer
4. assert failure
5. use a task id that is out of bounds
6. assert failure
7. use a task that is dormant
8. assert failure

Set priority tests

1. set a task prioity as low
2. get the task
3. assert the priority was set correctly
4. set task priority to high
5. get the task
6. assert the priority was set correctly

Set prio failure tests

1. create task with ID 0
2. assert failure
3. use a null buffer
4. assert failure
5. use a task id that is out of bounds

6. assert failure
7. use a task that is dormant
8. assert failure
9. use an invalid priority
10. assert failure

tsk failures at max task bounds

1. create tasks until at max_tasks bound
2. assert next creation fails

TCB reusal test

1. create task
2. yield task
3. creates task using same tid
4. assert that task id reused
5. tsk exit