Lab 3 Report

Group Members:

Geoffery Qin (c22qin)

Rebecca Zhou (yx5zhou)

Kehan Xing (k5xing)

Yiru Wu (y645wu)

Data Structure

Ring Buffer (MB)

- This structure is used to store the message in the mailbox

variable name	type	purpose
memory_start	U8*	Stores the start of the ring buffer which will be returned from the **k_mpool_alloc** function
memory_end	U8*	Stores the end of the ring buffer which is **memory_start** plus the size of the buffer
capacity	size_t	Size of the buffer
current_size	size_t	Current capacity of the buffer
head	U8*	Stores the current location of read (receiving message)
tail	U8*	Stores the current location of write (sending message)

Mailbox (MAILBOX)

- This is the structure of the mailbox which will contain a ring buffer and waiting list.
- The waiting list will only be used for blocking receive and send

variable name	type	purpose
active	int	Set to 1 if the mailbox is active
receiver_id	task_t	Stores the task id of the owner of the mailbox
memory_buffer	MB	Instantiation of Ring buffer
waiting list	QUEUE	Instantiation of Waiting List

ready_queue (QUEUE)

- This structure is used for waiting list and ready queues

variable name	type	purpose
HEAD	TCB*	Points to the head of QUEUE
TAIL	TCB*	Points to the tail of QUEUE
current_size	unsigned int	Stores the current size of the QUEUE

cmdnode (CMDNODE)

variable name	type	purpose
next	CMDNODE*	Points to the next CMDNODE in the CMDLIST
command	char	Stores the command character
command_handler_tid	task_t	Stores the command handler tid

cmdlist (CMDLIST)

variable name	type	purpose
head	CMDNODE*	Points to the head of the list
tail	CMDNODE*	Points to the tail of the list

inputnode (INPUTNODE)

variable name	type	purpose
next	INPUTNODE*	Points to the next INPUTNODE in the INPUTQUEUE
character	char	Stores the character

inputqueue (INPUTQUEUE)

variable name	type	purpose
head	INPUTNODE*	Points to the head of the queue
tail	INPUTNODE*	Points to the tail of the queue

Helper Functions

void mem_cpy(U8 dest, U8* src, size_t cpy_length)*

This function works as the memcpy() function in the C library. It copies *length* characters from memory area *src* to memory area *dest*.

int can_fit(MB memory_buf, size_t message_size)*

This function checks whether there's enough space in the buffer for a specific message size. The function returns –1 if the buffer is full, 0 if the remaining space is smaller than the message size, and 1 if there's enough space for the message.

int push_buffer(MB memory_buf, U8* buf)*

This function copies the received message to the message buffer of the receiver's mailbox. It checks if there's enough space for the message using *can_fit* function, and copies the message using *mem_cpy* function. Since we are using ring buffer for the message buffers, the function splits the message into two if we need more memory from the start of the buffer. This function returns 1 if the message is successfully copied and returns 0 otherwise.

int pop_buffer(MB memory_buf, U8* buf, size_t length)*

This function copies the message in the message buffer to a new buffer *buf* and remove it from the message buffer. It first copies the message header to get the message length, and then copies the entire message if *length* is larger than the message length. This function returns 1 if the message is successfully copied, 0 if there's no message in the message buffer and –1 if *length* is smaller than the message length.

Algorithm Functions

int mbx_create(size_t size)

This function creates a mailbox for the current running task with the input size. The new created mailbox will be saved in *g_mailboxes* array. The message buffer and the waiting list of the mailbox are also initialized. The function returns the mailbox id of current running task if a mailbox is successfully created, and return –1 otherwise.

*int send_msg(task_t receiver_tid, const void *buf)*

This is the function that does blocking send. It first checks for all the error conditions that are mentioned in the lab manual. The function can_fit checks whether or not this message can fit in the receiver's buffer and we also check if the current waiting list is empty. If both conditions are met, we can directly put the message in the buffer by calling the push_buffer function. If the buffer is not big enough to take the message or there is any message waiting to send, we then push the TCB of the sending task to the waiting list which is a FIFO queue and set itself to "BLK_SEND". If the receiving task is in the state of BLK_RECV, we will unblock it and put it back to the ready queue.

*int send_msg_nb(task_t receiver_tid, const void *buf)*

This is the function that does non-blocking send. It first checks for all the error conditions that are mentioned in the lab manual. In addition to the errors in block sending, if the waiting list is not empty or the message cannot fit in the buffer, we also return RTX_ERR and set errno bit to ENOSPC. The logic of putting the message in the buffer is the same as send_msg(). If the receiving task is in the state of BLK_RECV, we will unblock it and put it back to the ready queue.

*int recv_msg(void *buf, size_t len)*

This is the function that does blocking receive. It first checks for all the error conditions that are included in the lab manual. By calling pop_buffer we can check if there is any message in the ring buffer. If there

isn't any message, it will set itself to the BLK_RECV state and call schedular to make new running task arrangement. If a message is successfully popped from the buffer, we then check the waiting list from the front of the queue to the back for the blocked messages. In a FIFO convention, if the message can fit in the buffer, we move it into the buffer and remove the task from waiting list and push it to the ready queue. At the end we call schedular to see if we have unblocked any higher priority tasks along the way.

*int recv_msg_nb(void *buf, size_t len)*

This is the function that does blocking receive. It first checks for all the error conditions that are included in the lab manual. In addition to the errors in the block sending, if there is no message in the buffer, we immediately returns and set error bit to ENOMSG. Everything else is exactly the same as recv_msg.

*int tsk_ls(task_t *buf, size_t count)*

This function saves the task ids of the first *count* non-dormant tasks of *g_tcbs* array in *buf.* If the number of non-dormant tasks is smaller than *count,* the function will count the number of non-dormant tasks using the variable *non_dormant*, and will save *non_dormant* task ids in *buf*. This function will return minimum of *count* and *non_dormant*.

*int mbx_ls(task_t *buf, size_t count)*

This function saves the task ids of the first *count* tasks with a mailbox of *g_tcbs* array in *buf.* If the number of tasks with a mailbox is smaller than *count,* the function will count the number of tasks with a mailbox using the variable *with_mailbox*, and will save *with_mailbox* task ids in *buf*. This function will return minimum of *count* and *with_mailbox.*

int mbx_get(task_t tid);

This function returns the free space in the maibox of *g_tcbs[tid].* It returns –1 if the task is dormant or the task doesn't have a mailbox.

Test Cases

`ae_tasks1_G12.c`

This test suite creates 1 task during initialization, then the initial task will create other three tasks. Overall, it contains 4 tasks in total, 2 high priority and 2 medium priority. It mainly tests the blocking send / receive with priority setting. The test suite covers the following scenarios.

- Send message to a high priority blocking receive task, the CPU should be preempted by the unblocked high priority task.
- Receive message to leave enough space for high priority blocking send task, the CPU should be preempted by the unblocked high priority task.
- If a task been preempted, it should be put to the head of ready queue.
- If a task calling yield or been blocked, it should be put to the tail of ready queue.
- Same priority would not preempt each other.
- Using ring buffer repeatedly.
- No memory leak.

If everything passes, we should get a score of 27/27 at the end.

ae_tasks2_G12.c

This test suite creates 1 task during initialization, then the initial task will create other two high priority tasks. It mainly tests the size limit and basic send/receive in wither blocking / non-blocking situation. The test suite covers the following scenarios.

- Send message to no-mailbox receiver.
- Send message to non-exist receiver
- Receiver create multiply mailboxes.
- Message size less than minimum limit size.
- Message exceed mailbox size.
- Message buffer is null.
- No enough space for message.
- Check exist mailbox before and after task exit
- Send message to dormant task
- Check alive task before and after task exit.
- Check mailbox space before and after message sending.
- Blocking receiving.
- Blocking sending.

If everything passes, we should get a score of 33/33 at the end.

ae_tasks3_G12.c

This test suite creates 1 task during initialization, then the initial task will create other ten high priority tasks. It mainly tests the mbx_ls and tsk_ls. The test suite covers the following scenarios.
Based on this scenario, we tested the following cases:

- Memory leak.
- When task exit, mailbox is been deallocated.
- When task exit, mbx_ls will not return dormant task tid
- When task exit, tsk_ls will not return dormant task tid.
- When new task created after other task exiting, tsk_ls will return current tids

If everything passes, we should get a score of 22/22 at the end.