ECE-350 L3 Report

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

DATA STRUCTURES AND ALGORITHMS

Ring Buffer

The ring buffer holds the data within the mailbox in an efficient and easy to use way. It supports the following operations: push, pop and peek. Push is used to write len bytes into the buffer from the supplied memory address. Pop reads the first message out of the buffer and updates the read/write regions of the buffer. Peek returns the size of the fist message in the ring buffer. One design decision that was made that is not standard to the typical ring buffer implementation is the field bytes_written. This is used as there is no clean way to check if the right buffer is empty otherwise. This also makes it easy to find things like free space as it is held in the data structure directly.

```
typedef struct ring_buffer
{
   // The buffer to store the data in
   U8 *buf;

   // The length of the buffer
   U32 length;

   // The amount of bytes that have been written
   U32 bytes_written;

   // head -> where the stored messages start
   U32 read_idx;

   // tail -> where the messages end
   U32 write_idx;

} ring_buffer;

void init_ring_buffer_queue(ring_buffer *p_rb, U8 *buf, U32 length);
U32 get_ring_buffer_free_bytes(ring_buffer *p_rb);
int peek_ring_buffer(ring_buffer* p_rb);
int push_ring_buffer(ring_buffer* p_rb, const void *buf, size_t len);
int pop_ring_buffer(ring_buffer *p_rb, void *buf);
```

Mailbox

The mailbox data structure holds all of the essential data for the mailboxes. The first field holds whether the mailbox is initialized or not. The next field holds the FIFO queue of senders that are blocked in sending. Next for consistency we use a single-item queue to hold a blocked reader. Finally we store a ring buffer in the mailbox to hold the messages.

Note: the buffer for the mailbox ring buffer will be allocated dynamically when the mailbox is created.

```c
typedef struct mailbox
{
   int initialized;

   // Senders that are blocked on the buffer
   TCB_queue sender_waiting_queue;

   // Reader that is blocked | should only ever be one item
   TCB_queue reader_waiting_queue;

   // Buffer for the messages
   ring_buffer rb;
} mailbox;
```

TCB Queue

This is reused from the last lab to hold the waiting senders/ receivers.

```c
typedef struct TCB_queue
{
   TCB *HEAD;
   TCB *TAIL;
} TCB_queue;

void enqueue_back(struct TCB_queue *q, struct tcb *task);
void enqueue_front(struct TCB_queue *q, struct tcb *task);
void remove_tcb_given_tid(struct TCB_queue *q, task_t tid);
TCB *queue_pop(struct TCB_queue *q);
void init_queue(TCB_queue *tcb_q);
```

TEST SUITE DESCRIPTIONS

Note: we use helper functions initialize_msg_(MSG_TYPE) and validate_message in our test suites.

Test Suite 1 - ae_tasks1_G10.c

Tests the scheduler functions correctly and is invoked again with the new BLK states:

- Two new states will be main focus
    - BLK_SEND
        - Task enters this state when it calls send_msg() but the receiving task's mailbox is full/does not have enough space to receive the incoming message.
        - Task exits this state and becomes READY when the mailbox of the receiving task has space to receive the message.
        - Test that when a task is in BLK_SEND state, the scheduler returns the new task that should be run
        - Test that when a task exits the BLK_SEND state, the scheduler can return this task.
    - BLK_RECV
        - Task enters this state when it calls recv_msg() and its own mailbox is empty (there are no messages in the tasks mailbox)
        - Task exits this state and becomes READY when it's own mailbox is not empty (a message has been delivered to the mailbox)
        - Test that when a task is in BLK_RECV state, the scheduler returns the new task that should be run
        - Test that when a task exits the BLK_RECV state, the scheduler can return this task.
- Some regression done in this test:
    - tsk_set_prio()
    - Preemption
    - tsk_create()

Test 1:

- Tests that tasks enter BLK_SEND and BLK_RECV correctly
- Tests that the scheduler correctly reschedules after a task enters BLK_SEND and BLK_RECV states.
- Tests that tasks can exit BLK_SEND and BLK_RECV states correctly.
- Tests that after exiting BLK_SEND and BLK_RECV states, the tasks are placed correctly back to their respective priority ready queue. (If unblocked from BLK_SEND, should be added to the BACK of the respective priority queue. If unblocked from BLK_RECV, should be added to the BACK of the respective priority queue.)
- Test preemption when a task in BLK_SEND and BLK_RECV states is unblocked and is a greater priority than the currently running task. (Preempted tasks are always added to the FRONT of their respective ready queue.)

Steps:

- priv_task1 created with MEDIUM priority.
    - Update order of execution array.
- Priv_task1 creates a mailbox for itself.
    - **TEST 1**
- Priv_task1 creates task2 with LOW priority then task 1 with HIGH priority
    - **TEST 2 & TEST 3**
- Task1 preempts priv_task1 and creates a mailbox for itself
    - Update order of execution array.
    - **TEST 4**
- Task1 sends a message to itself that is it's entire mailbox size, thus filling its mailbox.
    - **TEST 5**
- Task1 calls mbx_get() to check that its mailbox is now full.
    - **TEST 6**
- Task1 changes its own priority to LOW, scheduler now selects priv_task1 to be run
- Priv_task1 creates a message and sends it to task1 which has a full mailbox, priv_task1 is now in the BLK_SEND state and scheduler is called again to tell task1 to run again.
    - **TEST 7**
    - Update order of execution array.
- Task1 yields so that task2 can run
    - Update order of execution array.
- Task2 creates a mailbox for itself
    - **TEST 8**
- Task2 calls recv_msg() which should fail as its mailbox is empty. Task2 is not in BLK_RECV state and scheduler calls task1 to run
    - **TEST 9**
    - Update order of execution array.
- Task1 calls recv_msg() to unblock priv_task1 from BLK_SEND state
    - **TEST 10**
- Since priv_task1 is unlocked it preempts task1 and runs
    - Update order of execution array.
- Priv_task1 unblocks task2 by sending it a message
    - **TEST 11**
- Priv_task1 exits, scheduler runs task1
- Task1 exits, scheduler runs task2
- Task2 checks the order or execution array and verifies that the scheduler made the correct decisions depending on priority and state
    - **TEST 12**
- Task2 exits
- Test exits

Test Suite 2 - ae_tasks2_G10.c

fifo message queue structure, backlog structure, tsk_exit dealloc, send msg to self

NOTE: test cases 1-6 execute in order

Test Suite 2 creates 3 tasks (task 1-3) which all send messages of specified lengths to the first task (priv_task1), completely filling its mailbox then adding one message to the backlog (BLK_SEND). We then let priv_task1 receive each message, which should appear in the order they were sent.

This suite contains the following 6 test cases:

- **TEST CASE 1**
  - verifies that priv_task1 sends message to itself successfully
  - checks that mbx_get of priv_task1 decreases by size of message
- **TEST CASE 2**
  - verifies that task1 sends message to priv_task1 successfully
  - checks that mbx_get of priv_task1 decreases by size of message
    - fills half of remaining space in mailbox
- **TEST CASE 3**
  - verifies that task2 sends message to priv_task1 successfully
  - checks that mbx_get of priv_task1 decreases by size of message
    - fills remaining space in mailbox
- **TEST CASE 4**
  - verifies that non-blocking send from task3 to priv_task1 fails since no space
  - checks that mbx_get of priv_task1 remains the same (0)
- **TEST CASE 5**
  - verifies that priv_task1 successfully receives all the messages in order
  - checks that the message data follows expect order from FIFO, with task3 last
- **TEST CASE 6**
  - verifies tsk_exit deallocates mailbox + make DORMANT, mbx_get(3) fails
    - task1 exits, becomes dormant and allows task2 to reuse tid=3, etc.

<u>**STEPS:**</u>

- <u>priv_task1</u> (prio = lowest) (tid=1)
  - create_mbx (primary receiver for tasks)
  - *send_msg to task1 (itself)?
    - **TEST CASE 1**
  - create task1 (prio = med) - WILL pre-empt
  - create task2 (prio = high) - WILL pre-empt
  - create task3 (prio = low) - WILL pre-empt
  - recv_msg until message buffer + sending task-waiting list empty (while loop)
    - **TEST CASE 5**
  - here, tasks 1-3 exited, deallocated mailbox, check mbx_get(3) since reused
    - **TEST CASE 6**
  - test_exit
- <u>task1</u> (prio=med) (tid=3)
  - send_msg to fill half of task1 mailbox (blocking)
  - mbx_get(priv_task1) to check decreasing size
    - **TEST CASE 2**
  - tsk_exit
- <u>task2</u> (prio=high) (tid=3)
  - send_msg_nb to fill other half of task1 mailbox (non-blocking) - NOW FULL
  - mbx_get(task1) to check decreasing size (0)

        - **TEST CASE 3**
    - tsk_exit
- task3 (prio=low) (tid=3)
    - send_msg_nb to fill half of task1 mailbox (non-blocking)
        - should output error, since full
    - mbx_get(task1) to check same size
        - **TEST CASE 4**
    - send_msg to fill half of task1 mailbox (blocking)
    - tsk_exit

Test Suite 3 - ae_tasks3_G10.c

test a series of blocking and unblocking receives / sends repeatedly to check for leaks and errors with resetting/initialization

test for:

- Blocking send blocking if the mailbox is full
- Blocking receive blocking if the mailbox is empty
- Blocking receive unblocking senders correctly
- Blocking send unlocking readers correctly
- Non-blocking calls fail if there is no space / no message in the mailbox
- Tasks cleanup correctly

STEPS:

1. Task 1 creates Task 2 with medium priority
2. Task 1 creates a mailbox
3. Task 1 calls receive message - blocked
4. Task 2 creates Task 3 with a low priority
5. Task 2 sends a message to Task 1 - preempts
6. Task 1 calls receive message - blocked
7. Task 2 creates a mailbox
8. Task 2 calls receive message - blocked
9. Task 3 creates a mailbox
10. Task 3 sends a message to Task 2 - preempts
11. Task 2 sends a message to Task 1 - preempts
12. Task 1 sends a message that fits into the mailbox of Task 3
13. Task 1 sets its priority to low - preempts
14. Task 2 senses a message to Task 3 that is too big to fit - blocked
15. Task 1 Sends a message to Task 3 - blocked
16. Task 3 receives a message and unblocks Task 1 and Task 2 - preempts
17. Task 2 sets its priority to lowest - preempts
18. Task 3 receives the other two messages and exits - dormant
19. Task 1 sends a message that fits into the mailbox if Task 2
20. Task 1 sends a message that does not fit into the mailbox of Task 2 - blocked
21. Task 2 receives a message and unblocks Task 1
22. Task 1 sends a nb message to Task 2 that succeeds
23. Task 1 sends a nb message to Task 2 that fails (too large)

24. Task 1 exits
25. Task 2 verifies all of its messages and exits

Test Suite 4 - ae_tasks4_G10.c

test KCD, recv_msg w/ most recently registered, re-register command id, edge cases

NOTE: test cases 1-11 execute in order

Test Suite 3 creates 2 tasks (task 1-2) which each register command ids through TID_KCD then waits to receive a message (BLK_RECV). Next, we have priv_task1 send (simulated) KEY_IN messages that should be forwarded to task 1-2 unblocking them, testing specifically that the message is forwarded to the task that most recently registered the command id. We test a task's ability to re-register a command id then send it back to itself (thru KCD). We also verify that mbx_ls and tsk_ls return the correct number of tasks at various points. Finally, we check edge cases of sending (simulated) KEY_IN messages that (first) have no associated registered task (should pass + console print “Command not found”), and (second) represent an invalid command without a ‘%’ (should pass + console print “Invalid command”).

This suite contains the following 11 test cases:

- **TEST CASE 1**
    - verifies that task1 successfully registers command ids W and X
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv
- **TEST CASE 2**
    - verifies that task2 successfully registers command id W
    - checks that mbx_get(TID_KCD) goes down
        - KCD task will not preempt since same prio, msg stuck in mailbox
- **TEST CASE 3**
    - verifies that tasks 1-2 are created successfully
- **TEST CASE 4**
    - verifies sent command id %W goes to task2, since most recently registered
    - verifies that the message data contains the exact above string sent from KCD
- **TEST CASE 5**
    - checks that we can successfully re-register command id W to task2
    - checks that mbx_get(TID_KCD) goes down
        - KCD task will not preempt since same prio, msg stuck in mailbox
- **TEST CASE 6**
    - checks that task2 successfully sends+receives command id %W to/from itself
    - verifies that the message data contains the exact above string sent from KCD
- **TEST CASE 7**
    - checks that mbx_ls=3 (tid=3, 14, 15)
    - checks that tsk_ls=6 (tid=0, 1, 2, 3, 14, 15)
- **TEST CASE 8**
    - verifies sent command id %X goes to task1, since only registered
    - verifies that the message data contains the exact above string sent from KCD

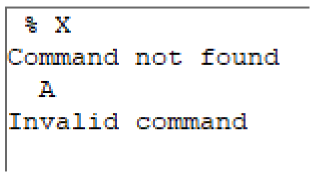
- **TEST CASE 9**
    - checks that mbx_ls=2 (tid=14, 15)
    - checks that tsk_ls=5 (tid=0, 1, 2, 14, 15)
- **TEST CASE 10**
    - verifies that command id %X successfully sent to KCD
        - command id not registered anymore (task1 exited), no matching task
        - console should print “Command not found”
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv
- **TEST CASE 11**
    - verifies that invalid command id A successfully sent to KCD
        - invalid command id should not lead anywhere
        - console should print “Invalid command”
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv

STEPS:

- priv_task1 (prio = low) (tid=1)
    - create task1 (prio = med) - WILL pre-empt (run to recv_msg)
    - create task2 (prio = high) - WILL pre-empt (run to recv_msg)
        - **TEST CASE 3**
    - send command id %W first time (should go task2, most recent registered)
    - mbx_ls - should return tid=3, 14, 15
    - tsk_ls - should return tid=0, 1, 2, 3, 14, 15
        - **TEST CASE 7**
    - send command id %X first time (should go task1, only registered)
    - mbx_ls - should return tid=14, 15
    - tsk_ls - should return tid=0, 1, 2, 14, 15
        - **TEST CASE 9**
    - send command id %X second time (console print “Command not found”)
        - **TEST CASE 10**
    - send invalid command id A (console print “Invalid command”)
        - **TEST CASE 11**
    - test_exit
- task1 (prio=med) (tid=3)
    - create_mbx
    - register command id %W
    - register command id %X
        - **TEST CASE 1**
    - recv_msg from priv_task1 - should receive %X - KCD_CMD
        - **TEST CASE 8**
    - task_exit - will now go to task2 again, deallocate+delete mailbox
- task2 (prio=high) (tid=4)
    - create_mbx
    - register command id %W (will take precedence since most recent)
        - **TEST CASE 2**
    - recv_msg from priv_task1 - should receive %W
        - **TEST CASE 4**

- re-register command id %W
    - **TEST CASE 5**
- send command id %W second time (should redirect back to itself)
- recv_msg from TID_KCD
    - **TEST CASE 6**
- task_exit - will now go to priv_task1 again, deallocate+delete mailbox

Expected console output:



```
 % X
Command not found
  A
Invalid command
```

Test Suite 5 - ae_tasks5_G10.c

Main focus of this test case should be:

- UART interrupt handler
    1.
        - Keyboard input forwarded to mailbox of KCD using KEY_IN message
        - Then KCD forwards the message to the console display task's mailbox to be echoed/displayed back to the console.
    2.
        - Keyboard input keys are queued (inputted one at a time) so that receiving carriage return key (enter key) KCD dequeues all previous keys and combines them to construct a single string with those keys
    3.
        - If the string is %LT, KCD will print the task ID and state of all non-Dormant tasks on the console.
    4.
        - If the string is %LM, KCD will print the task ID and state of all non-Dormant tasks WITH A MAILBOX on the console.
        - Should also print the mailbox ID associated with the task and the amount of tasks that are in the sending task-waiting list.
    5.
        - Test that when TID_KCD and TID_CON (KCD and Console display task) can preempt and also verify their mailbox behaviours when preempting and not preempting
            - When preempting, the KCD or Console Display Task should immediately recv thus, the mailbox size remained unchanged (because it received the message so it is cleared out of the mailbox)
                - task1() has MEDIUM prio, so it will be preempted
            - When not preemption, the KCD or Console Display Task should not recv immediately so the mailbox is filled with a message so mailbox size decreases.
                - task1() has HIGH prio, so it will not be preempted
- Console Display Task

1. Only responds to one message type that being the DISPLAY message type. So test that it ignores the other types.
2. Message body can contain control characters such as newline, ANSI escape sequences, etc…
3. Displays the message string to the console terminal.
4. Can disable the UART0 transmit interrupt when all characters of the string have been displayed.

<u>NOTE: test cases 1-4 execute in order</u>

This suite contains the following 4 test cases:

- **TEST CASE 1**
    - verifies that invalid command id A successfully sent to KCD
        - invalid command id should not lead anywhere
        - console should print "Invalid command"
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv
- **TEST CASE 2**
    - verify that %LT and %LM are successfully sent to KCD
        - manually check console output for listed mailboxes, tasks
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv
- **TEST CASE 3**
    - verify that generic DISPLAY message can be directly sent to CON
        - manually check console output for printed message data
    - checks that mbx_get(TID_KCD) stays the same size, unchanged
        - KCD task will preempt since higher prio, immediately recv
- **TEST CASE 4**
    - verify that non-DISPLAY message (DEFAULT) sent to CON
        - manually check console output, should not print anything at all
    - checks that mbx_get(TID_KCD) goes down
        - KCD task will not preempt since same prio, msg stuck in mailbox

<u>STEPS:</u>

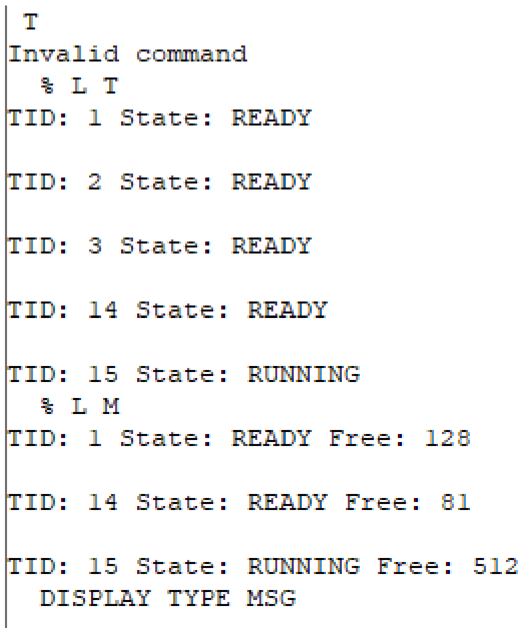
- <u>priv_task1</u> (prio = low) (tid=1)
    - create_mbx for itself
    - create task1 (prio = high) - WILL pre-empt
    - create task2 (prio = high) - WILL pre-empt
    - test_exit
- <u>task1</u> (prio=med) (tid=3) TID_KCD and TID_CON WILL preempt => mailbox same
    - send invalid command T to KCD (no %)
        - **TEST CASE 1**
    - send %LT and %LM to TID_KCD
        - **TEST CASE 2**
    - send DISPLAY type msg directly to TID_CON
        - **TEST CASE 3**
    - tsk_exit
- <u>task2</u> (prio=high) (tid=4) - TID_KCD and TID_CON WON'T preempt => mailbox lower

- send non-DISPLAY type msg to TID_CON - should pass, TID_CON ignores
    - **TEST CASE 4**
- tsk_exit

NOTE: TID_KCD and TID_CON have HIGH priority, so for:

- task1 (MEDIUM prio) - TID_KCD and TID_CON will preempt and immediately recv, so mailbox size unchanged
- task2 (HIGH prio) - TID_KCD and TID_CON will not preempt (task2 will finish first, so mailbox size changes)

<u>Expected console output:</u>



```
 T
Invalid command
  % L T
TID: 1 State: READY

TID: 2 State: READY

TID: 3 State: READY

TID: 14 State: READY

TID: 15 State: RUNNING
  % L M
TID: 1 State: READY Free: 128

TID: 14 State: READY Free: 81

TID: 15 State: RUNNING Free: 512
  DISPLAY TYPE MSG
```