SE350 Operating Systems Documentation

Gaisano, G., Janecka, L., Mak, R., Schneider, C.

April 5, 2015

1 Introduction

Clarisse

- What is this document about
- What is the purpose of the document?
- What does the document contain?

2 Global Variable Documentation

Everyone

- What does each variable store?
- Why is there a variable to store this?
- What do your global data structures look like?
- What functions use it?

Global Variables

VARIABLE_NAME Stores x for y purpose, used by:

- z

Global Data Structures

DATA_STRUCTURE_NAME Stores x for y purpose, used by:

Structure Name	**Purpose**	**Properties**
Queue	generic queue	• **Element *first**: pointer to first element in the Queue. Is NULL if there are no elements. • **Element *last**: pointer to last element in the Queue. Is NULL if there are no elements.
Element	Generic queue element	• **Element *next**: pointer to next element in queue. NULL if the last element • **void *data**: pointer to element data • **void *block**: pointer to memory block element resides in
PCB	**Process Control Block**. Used to store process identification data and status, such as running state and progress in process	**U32 *mp_sp**: stack pointer of process • **U32 m_pid**: process id • **PROC_STATE_E m_state**: current running state of the process • **int m_priority**: process priority (low value = high priority) • **Queue *mailbox**: pointer to the process' mailbox. Contains all messages passed to process. After initialization, remains unchanged, as only the first and last pointers change.

Block	Represents a chunk of free memory. Is returned to the user on request_memory_block	• **int pid**: the id of the process that currently owns the Block. Is NULL if Block is free. • **Block *next**: pointer to the next Block in the free memory block list. Is NULL if block is in use/allocated to a user process.
msgbuf	Stores the message's data	• **int mtype**: the message type (types defined in CONSTANTS section) • **char mtext[size]**: char array containing the message data. Size is dependent on BLOCK_SIZE
Envelope	The header for a message. Contains necessary information for delivery. Is the structure that is passed around in the message-passing system.	• **int sender_id**: pid of sending process • **int destination_id**: pid of destination process • **int time**: message timestamp • **int delay**: time delay to send message (seconds) • **msgbuf *message**: pointer to message data

- z

3 Kernel API

Clarisse

- All kernel fuctions
- What does each function do?
- What does proper use of this function look like?
- What cleanup is necessary afterwards, if any?

- Does my documentation cover its behaviour in all scenarios?
- Is this described more efficiently through pseudo?

4 Interrupts and their Handlers/Processes

Ginelle

Global Variables: timed_q

Major Design Changes

- Should made a special i-process queue instead of putting in them in the ready queue with other user processes
- Not have i-processes depend on message passing. Instead, have the i-process block itself on finishing with input and be unblocked by the interrupt handler.

Questions

- What interrupts are enabled by your OS? Interrupts:

Interrupt Handlers	**Description**	**Functionality**
Timer	Increments a counter after every clock tick. For each second passed, decrements all messages in the delayed timed queue.	• pushes delayed messages to appropriate destination process mailbox when message delay has passed.
Keyboard	captures keyboard input, composes and sends a message to the UART i-process	• prints debugging hot keys to UART1 output. • sends key press to KCD for command processing.

i-process	Description	Active
UART_iprocess	blah	blah
KCD	blah	blah

- **How does the OS handle those interrupts?** For the Timer Handler, once a message's delay had expired, the interrupt pushed the message to the mailbox of the destination process. All the logic for the timer-related interrupts are in the Timer

Handler. For the Keyboard Handler, it sends a message to the UART i-process, who then sends that message to the KCD i-process for command decoding. While waiting for keyboard input, both i-processes are blocked on received, waiting for messages to arrive. All these processes are given the highest priority to ensure that they interrupt any current processes.

- What do your interrupt processes do?
- Does my documentation cover its behaviour in all scenarios?
- Is this described more efficiently through pseudo?

5 System and User Processes

RayMak

- What system processes are in the OS?
- What is the purpose of each system process?
- Waht does each system process do?
- what services do each of the system processes depend on?
- What system processes does each user processes use?

6 Initialization

Lara

- What steps does your OS take to boot?
- What parameters does your OS have?
- How are these parameters tuned?

7 Testing

7.1 Test Handler

Testing was done by creating a test handler that handled the set up, tear down, and printing of each test. The test handler was contained in *usr_proc.c* allowing test files to

be separated and still use the handler. The test handler starts by printing the starting information, then it receives a message from each test and frees the memory used for that message. Once each of the five tests have finished the test handler is unblocked and it prints the ending information. A global variable is used to keep track of how many tests have failed to be printed.

7.2 Test Procedure

Each test is initialized to LOW priority and the dummy test functions (A, B, and C) are initialized to LOWEST priority. When a test starts it sets its own priority and the property of any dummy tests that it uses to MEDIUM, this is done to ensure that only one test is running at a time in an isolated environment, ensuring that no tests interfere with each other. During its execution the test only messes with the priorities of the dummy test functions to keep this isolated state. Each test has a local variable **failed** which is used to count the number of internal assertions that have failed. Each test function also has a global counter (**function_name_count**) associated with it to allow assertions to be made between processes. When a test finishes it sets its priority to LOWEST and calls an endTest function. This function sends a message to the test handler, sets each of the dummy test functions and the test handler back to priority LOWEST, sets the global counters to 0, and checks the failed variable to print the correct output and update the global failed test counter. Essentially the endTest attempts to return everything to the same starting state and print the correct output at the end of each test.

7.3 Part 1 Tests

The part one tests focused on testing memory management and process priority switching (in *usr_proc1.c*).

7.3.1 Test 1

This test runs basic unit tests on getting and setting process priority. It starts with getting the priority of process A and checking that it is correct and that the appropriate ready queue contains A. It then tests preemption by setting process A to HIGH. Within A its global counter is incremented and the processor is released. This counter is checked to make sure that a did execute and return and the current priority of A is checked. This can also be con Finally the test attempts to set A to a invalid priority to ensure that an error is returned.

7.3.2 Test 2

This test runs basic unit tests on memory allocation and releasing. It starts by requesting a memory block and checking that the MSP has been updated appropriately. It then releases that memory block and checks that the MSP returns to the value it had initially. Next it checks the error codes by requesting a memory block, freeing it, then checking that the return value is RTX_OK. The test then attempts to free the same memory block again and checks that the return value was RTX_ERR.

7.3.3 Test 3

This test checks that memory ownership is enforced. It uses a global variable **test3_mem** which is requested by test3. The test then sets B to MEDIUM and releases the processor. B then attempts to release test3_mem, records the value in a global variable, and returns to test3. Test3 checks that B got a error when it attempted to release test3_mem and releases it.

7.3.4 Test 4

This test checks what happens when we reach the end of memory. The test starts by requesting a bock of memory **test4_mem**. This will be used later. The test then releases to C, Within test C we request exactly the number of free memory blocks +1. This should cause C to be blocked and the processor to switch back to test4. Here test4 checks that C has been put on the blocked on resource queue correctly. Then test4 releases the test4_mem block which should trigger preemption back to C. Here C frees all of the memory that it allocated and returns to test4. Test4 checks that the amount of free memory blocks at the start of the test equals the amount of free memory at the end of the test to ensure that we have no memory leaks.

7.3.5 Test 5

This test focuses on ensuring that the way we manipulate our queues does not introduce memory leaks. We create generic queue elements containing two pointers (to the next element and to the data contained in the element). For efficiency we store multiple elements in a single memory block. Test5 starts by allocating enough elements to fill a memory block and checking that a new block has been allocated for them. It then released each of those elements and checks that the memory block allocated for them is returned correctly.

7.4 Part 2 Tests

The part two tests focused on testing message passing (in *usr_proc2.c*).

Note: the order of these tests seems a little odd, this is done to prevent tests from interfering with each other.

7.4.1 Test 1

This test tests sending a receiving multiple messages. The test starts by sending a configurable number of messages to B (this value is stored in a variable NUM_TEST_MESSAGES_MAILBOX) and checking that B's mailbox is of appropriate size. The test then switches to B who iterates through its mail box freeing each message, then returns to test1. This also tests that the special memory ownership related to message sending works since B is the one who releases the memory associated with the message. This test can be used to evaluate the maximum number of messages the system can process at the same time.

7.4.2 Test 2

This test also tests sending and receiving multiple messages, but this test sends and receives one at a time instead of sending and receiving blocks of messages. The test records the number of free memory blocks then switches to B. Within a for loop of configurable size (stored in global variable NUM_TEST_MESSAGES) B creates a message, sends it, increments a counter (**messages_sent**), and switches to test1. Within a similarly sized loop, test1 receives a message, deallocates it, increments a counter (**messages_received**), and switches to B. Once the message sending is done we check that the ending number of free memory blocks is equal to the starting number of free memory blocks and that the number of messages sent is equal to the number of messages receieved. This ensures that there is no memory leak or lost messages. This test also runs the same procedure sending messages to itself. In theory this test can run for any number of messages.

7.4.3 Test 3

This test tests blocking and blocking from the blocked on receive queue. This test starts by immidiatly switching to C. C attempts to receive a message. It should be blocked and switch back to test3. Test3 checks that C is now on the blocked on receive queue then sends C a message and releases the processor. C should unblock, receive the message, increment a counter, and switch back to test3. Test3 checks the counter to make sure that C finished executing and checks that C has been removed from the blocked on receive queue.

7.4.4 Test 4

This test tests preemption on the blocked on receive queue. The test starts by setting A to HIGH priority which should preempt to A. A then attempts to receive a message and gets blocked with should preempt back to test4. Test 4 sends a message to A which should preempt back to A. A increments its counter and releases the processor. Then test4 just checks that A incremented its counter signifying that it finished running.

7.4.5 Test 5

This test contains the unit test for send, receive, and delayed send. The test starts by switching to A. A then sends two messages to test5, one with a delay and one without in that order (both messages contain different text to tell them apart), and releases the processor. Test5 now receives both messages. It checks that the sender was correctly sent and that the message text of the first message equal what we expect it to. It similarly checks that the text of the second message matches what should have been sent. This also ensures that the messages arrived in the correct order. Since the delayed message was sent first if it did not delay it would arrive first. We also watch the test to see that the test has the appropriate delay before printing.

7.4.6 Manual Tests

The wallclock was tested by manually entering each of the following commands after the automatic tests finished.

Manual tests:

- %W (wait a few seconds to watch it increment)
- %WT
- %W (wait a few seconds (check that the starting value is correct)
- %WR (check that clock resets)
- %WS 12:59:55 (watch to make sure it returns to 00:00:00)
- %WS 33:00:00 (make sure that it does not set to invalid times)
- r (check that the ready queue is printed correctly)
- b (check that the blocked on resource queue is printed correctly)
- m (check that the blocked on receive queue is printed correctly)

- leave the wall clock running for a few minutes

7.5 Part 3 Tests

Part three is different from the other tests since it is implementing given pseudo code (in *usr_proc3.c*) with all other test process empty and uses a specialized test handler (in *usr_proc_p3.c*). The automatic version of the test is simply the pseudo code listed in the assignment requirements.

Manual tests:

- the above listed wall clock tests
- %C 5 3 (move test4 to a new priority)
- r (check that test4 has changed priority)
- %C 5 12 (check invalid priority)
- %C 45 1 (check invalid process)
- %C 10 3 (check forbidden processes)
- %C 5 4 (check forbidden priority)
- %C 8 0(check moving blocked processes)

7.6 Running the Tests

Tests were run automatically by the test handler (with the exceptions of the few listed manual tests). Each part's tests were contained in their own file separate from the test handler. To run a different set of tests we linked in that test file and ran it. Some exceptions were made for part 3 which required a different test handler due to its unique nature. Testing was done almost exclusively on the debugger, testing with the board only at the very end to make sure that the demo would run smoothly. This was done both to protect the board and allow us to the many tools built into the debugger.

8 Major Design Changes

Everyone

- What design decisions ended up being a mistake?
- What were the major stumbling blocks?

- What would you do differently if you started over?
- What design issues did your OS have?

9 Timing Analysis

Everyone

10 Conclusion

Clarisse