**Memory Management**

Using generic linked list to keep track of free memory blocks

Memory management API provides an interface for processes to interact to when requesting or returning memory.

**Global Variables Used**:

**gHeap** - list to hold free memory blocks,

**MEM\_BLOCK\_SIZE**: 128 bytes,

**NUM\_MEM\_BLOCKS**: 30

**mem\_blocks:** holds all memory blocks

|  |
| --- |
| METHOD 1: void MEMORY\_INIT(void) |
| allocate memory for RTX\_IMAGE  allocate memory for PCB pointers  allocate memory for each process stack  update stack pointer  allocate memory for heap, size equals NUM\_MEM\_BLOCKS\*128 bytes |

|  |
| --- |
| METHOD 2: void\* K\_REQUEST\_MEMORY\_BLOCK(void) |
| while no memory available  add the calling process to the BLOCKED\_ON\_RESOURCE queue  k\_release\_processor();  once is memory available  pop the memory block from the heap  and return it to the calling process |

|  |
| --- |
| METHOD 3: K\_RELEASE\_MEMORY\_BLOCK(void\*) |
| Check if the block being freed is valid  If the block is valid  Add the block back the heap (gHeap List)  Follow preemption policy to assign the memory to processes |

|  |
| --- |
| METHOD 4: K\_RELEASE\_MEMORY\_BLOCK\_VALID(void\* p\_mem\_block) |
| ***// A helper function to check if the memory block being released is actually valid***  return RTX\_ERR if p\_mem\_block is invalid  return RTX\_ERR if address pointed by p\_mem\_block is outside the bounds  return RTX\_ERR if the p\_mem\_block size is not 128 bytes  return RTX\_ERR if trying to free a block that is already free  return RTX\_OK |

**Processor Management**

Firstly, this file contains the following global variables:

PCB \*\*gp\_pcbs -- An array of pcb pointers. This array contains all the PCBs for our OS.

PCB \*gp\_current\_process -- A pointer that always points to the current running process.

Afterwards, we declare two queues, the ready queue and the blocked queue. They are both global variables. Then we have our process initialization tables, which contain initialization information for all the processes of our OS.

The function infinite\_loop(void) simply calls release\_processor() forever, and the null process is assigned this procedure. So in essence, all the null process does is call the release\_processor().

Now the function process\_init() initializes all the processes in a system. It calls the set\_test\_proc() function, which fills out the initialization information for all the user processes into a table called g\_test\_procs[], and then process initialization information is copied from there into g\_proc\_table[]. The NULL process is also initialized explicitly beforehand.

Now, we loop through the g\_proc\_table[], and initialize the contents of our gp\_pcbs i.e initializing all the PCBs in our OS. We copy over the pid, priority, and sp, and we initialize all the states to NEW. Afterwards, we push every pcb onto the ready queue.

Now the scheduler(void) function simply picks the pcb of the next process to run. It makes sure that there is a process available on the ready queue to run, and that it is not blocked. Otherwise, it will return the NULL process.

The process\_switch() function simply takes in the previous running process, and sets the gp\_current\_process to running. It takes care of various state information (such as setting gp\_current\_process state to ready, etc.).

The k\_release\_processor(void) simply calls the scheduler to determine the next ready process, and then pushes the old process back to the end of the ready queue.

The method k\_enqueue\_blocked\_on\_resource\_process(PCB \*pcb) sets the state of the pcb passed in to BLOCKED\_ON\_RESOURCE and enqueues it on to the back of the blocked queue.

The method k\_dequeue\_blocked\_on\_resource\_process(void) dequeues the next available process in blocked\_on\_resource queue and returns it.

The method get and set priorites do exactly what is expected: get and set priorities of the required PCBs.

The method check\_preemption() checks if the PCB in the front of the ready queue should preempt the current running process.

User test processes

There are 6 user test processes. The first 3 (the minimum required to test all features) behave like unit tests, testing each case of each function against the specification.

The unit tests were designed with debuggability in mind:

* The tests are easy to read. Each test says what it's evaluating, what process it starts from, and what process it ends at.
* The tests have very little boilerplate. Boilerplate is more code that can have bugs that a simpler test harness can prevent.
* The test output is easy to interpret. Specifically, any failures output the name and line of the failed test. Unfortunately, to output "total n tests" at the start, it's necessary to hard-code the number of tests. This is because the number of tests is unknown at the start of the user tests.
* The tests are comprehensive. This makes it easier to resolve ambiguity in the specification and verify that behaviour doesn't change over time. In particular, the tests cover some edge cases for process priority, where breaking changes could result in deadlocks.
* The tests are deterministic, assuming the operating system is deterministic. This makes test results reproducible. Unfortunately, this means the tests only test correctness, not performance. On many architectures, performance-related test results can be harder to reproduce, since hardware manufacturers may make architectural design decisions that improve average performance while introducing nondeterministic behaviour. The test results could depend on the branch predictor and/or cache state, rather than just the program code.

The tests work by tracking the current state and marking the line of code where one state ends and another begins. Each state performs one or more checks, including a check to see if the next state is the expected one. These state checks are coded in-line and rely on the compiler's string interning to check for equality. The state checks are directly coded as string literals so other processes can check the state without using a global declaration at the top of the file, visible from both processes' functions.

Most tests run in the lowest-numbered process(es) possible. For example, the only 2 tests in process 3 are the round-robin scheduling test and the resource contention tests. FIFO-semantics for round robin scheduling are only noticeable when there are more than 2 processes, and the resource contention tests need at least 2 processes blocked on resources and 1 running process.

Of the other 6 processes, the last 3 behave like normal programs, and theoretically should not interfere with the first 3. These simply exercise the processor and expect the correct behaviour, performing some math and some recursion (quicksort) logic. They have some complex behaviour, so running them in parallel will test all sorts of interleavings of processes. For example, the quicksort implementation does the recursion in one process (proc5) and the partitioning in another (proc6). There's another process (proc4) that changes its own priority repeatedly. It guarantees it eventually sets itself to the lowest priority and releases the processor, but it detected a livelock condition.

The combination of the two kinds of tests allows the first 3 processes to accurately report when and how a specific API is failing and the last 3 processes to detect more complex failures.

Linear list

To help implement the queueing for processes and for memory management, there's a generic linear list. The linear list can store items of any type that supports the sizeof operator. By pulling out the queue logic, the queue logic can be coded once and debugged once, separately from the rest of the kernel.

The linear list is like a double-ended queue with a LL\_FOREACH operation. It supports the basic operations:

* LL\_DECLARE: Declare and statically initialize a linear list or array of linear lists.
* LL\_CAPACITY: Return the maximum capacity of the linear list.
* LL\_SIZE: Return the current size of the linear list, the number of pushes minus the number of pops.
* LL\_{PUSH,POP}\_{FRONT,BACK}: Push/pop an element to/from either end of the list.
* LL\_FOREACH: Execute a statement for each element from the front to the back. In addition, the list can be copied using the standard functionmemcpy.

The list is statically allocated because the memory management would also need to use it. In addition, a zero-initialized list is empty, making it easy to allocate arrays of empty lists. This makes it unnecessary to initialize lists to the empty state.

To allow the same functions to work on queues with different capacity and type, the interface uses macros. At each call site, the macros perform type and capacity checks. Unfortunately, in C, there are no template functions, so macros are the only way to perform these checks. This makes using the lists less error-prone.

Priority Queue

To store process IDs for ready and blocked processes by priority, as well as determine which should be the next process that should be run/unblocked and run, an array of the generic Linear Lists above indexed by priority is used.

The priority queue can support up to 5 priorities as that is the number of priorities to be supported by processes, and a set capacity for each linked list per priority.

2 global priority queues for pids are used: a ready queue and a blocked on resource queue

The priority queue supports these operations:

* push\_process: pushes a process ID into a list at the index of the priority
* pop\_process: pops a pid at a given priority
* pop\_first\_process: traverses the queues in decreasing order of priority and pops the first pid it finds
* peek\_process\_front: returns the first pid at the beginning of the queue for the given priority
* peek\_front: traverses the queues in decreasing order of priority and returns the first pid it finds
* peek\_process\_back: returns the first pid at the end of the queue for the given priority
* change\_priority: traverses the queue to find a given pid, then removes the pid from the queue indexed by the pid's current priority and enqueues it to the queue for the new priority the pid is to be assigned to
* move\_process: finds the pid in one priority queue (from\_queue), removes it from that queue, then adds the pid into the other queue (to\_queue) keeping its priority the same
* clear\_queue: removes all pids from a priority queue
* copy\_queue: pushes all of the pids of one queue (from\_queue) into the other (to\_queue), then clears the first queue
* is\_pid\_queue\_empty: checks if a given queue is empty by calling LL\_SIZE
* remove\_from\_queue: removes a pid from a given queue, and returns the removed pid
* queue\_contains\_node: checks if a given queue contains a given pid

Sorted Message List/queue

To support the sending and queuing of messages and delays, a second type of queue was needed. The sorted message queue is essentially a singly linked list with each node being a MSG\_BUF (as provided in common.h) with the next element or link provided by the mp\_next field.

The message buffer list is guaranteed to be sorted by the time which the message should be sent in ascending order. The message list supports the following operations:

* is\_queue\_empty: checks if the message queue is empty (by checking if it’s null)
* peek\_message: returns the message at the start of the queue
* dequeue\_message: dequeues the first message that has a delay/send time before or at the current g\_timer\_count, and returns it. If no suitable message is found, this returns null
* enqueue\_message: takes a message, then Iterates through the list until the next element’s delay/send time is greater than the provided message’s delay/time or is null, then inserts the provided message just before that element (or at the end of the list if there are no elements with a greater delay/send time)

Check Preemption

When a memory block is to be released or a process is to be set to a new priority, preemption must be checked before the operation ends.

Preemption checking starts first by checking if the heap has free memory, and if it does, the pids of all blocked blocked on resource processes are moved out of the blocked on resource queue and into the ready queue and the PCBs for each of those processes have their state changed from BLOCKED\_ON\_RESOURCE to RDY.

The priority of the process with the highest priority in the ready queue is then checked, and if its priority is higher than that of the current process, then the processor is released.

KCD Process

The KCD process is responsible for registering new commands and taking keyboard input. It first receives a message from a sender and then it determines the type, then the appropriate action for handling the message is done. An error is outputted if the message is either or null or the message type is invalid. Note that in the below pseudocode, the memory block is only released if the message is null and its type is invalid.

|  |
| --- |
| METHOD 1: void proc\_kcd(void) |
| while (true)  receive message from sender\_id  determine message type  if message is null  output error message and continue  else if message type is command registration  kcd\_process\_command\_registration(message) and continue  else if sender\_id is PID\_UART\_IPROC  kcd\_handle\_keyboard\_input() and continue  else  output error message  release memory block for message |

To register a command, the message is simply forwarded to the kcd’s message queue to be sent. The delay/send time of the message is set to 0 so it can be sent immediately.

|  |
| --- |
| METHOD 2: void kcd\_process\_command\_registration(message) |
| Set message delay to 0  Enqueue\_message |

To handle keyboard input, the input is parsed character by character, and added to a command until a CR (‘\r’) is encountered, at which point, kcd\_process\_keyboard\_input is called. Kcd\_process\_keyboard\_input takes the message and compares it to a list of registered messages. When a registered command is matched with the input, a message buffer is created by requesting a memory block with the contents of the message copied into the block, and the block containing the message is sent towards the appropriate process.

|  |
| --- |
| METHOD 3: void kcd\_handle\_keyboard\_input(void) |
| Initialize a global temporary message buffer  for all characters c in input  if c is newline  break  else if c is ‘\r’  end command with null terminator  kcd\_process\_keyboard\_input(message)  set command length to 0  else if c is null char  return  else  if command length is less than message max length  add c to message buffer’s text and 1 to command length |

|  |
| --- |
| METHOD 4: void kcd\_process\_keyboard\_input (message) |
| For all registered commands c  If message text matches c  Request new memory block  Copy message into block  Send message |

**Inter-Process Communication**

The RTX support message based Inter-Process Communication. Messages are sent between processes using message envelopes. Each process has a mailbox which is a queue of message envelopes.

The message consists of two parts header which contains information like *sender\_pid*, receiver\_pid, and other kernel data (like message delay time) that is used by the operating system. The envelope contains the data to be sent and it’s type.

To send the message, a process must first call request\_memory\_block for an envelope. The message envelope is then populates and

|  |
| --- |
| **ALGORITHM 1:** k\_send\_message(receiver\_pid, message) |
| if message is not valid  return RTX\_ERR;  k\_send\_message\_helper(sender\_pid, receiver\_pid, message);  //k\_check\_premption checks if any preemption condition is satisfied  k\_check\_preemption(); |

|  |
| --- |
| **ALGORITHM 2:** k\_send\_message\_helper(sender, receiver, message) |
| Get the PCB of the receiver message  Push the message the to the receiver message’s message queue  If(receiver.state == BLOCKED\_ON\_RECEIVE)  receiver.state = READY  Put the receiver process in the ready queue |

|  |
| --- |
| **ALGORITHM 2:** k\_receive\_message(sender\_pid) |
| //This is a blocking receive  While message queue of current process is empty:  Set the state of the current process to BLOCKED\_ON\_RECEIVE  Release\_processor();  Dequeue message from current process’s message queue  And update the sender\_pid  Return message |

|  |
| --- |
| **ALGORITHM 3:** k\_receive\_message\_non\_blocking(sender\_pid) |
| //This is a non-blocking receive  If there is a message in the message queue of the calling process  return the message  Else  return NULL |

Primitive Timing

To time the primitives, we programmed a second timer to count the number of cycles each primitive takes in a single call. This was chosen because we can't call each primitive thousands of times consecutively without blocking the current process, so we needed an accurate way of timing one iteration. The board has 3 timers, Timer 0, Timer 1, and the SysTick timer, and the LEARN discussion board says that using the SysTick timer is acceptable, so we implemented timing both the SysTick timer and Timer 1, depending on whether the macro USE\_SYSTICK is defined.

Referring to the ARM documentation, we timed the execution of 4 statements in C:

* The empty statement
* p = request\_memory\_block();
* send\_message(caller\_pid, p);
* q = receive\_message(&sender);

This also allows us to also time the time for the interrupt handler to run. The timings are given in clock cycles, which can be converted to seconds by dividing by the clock frequency.

According to the ARM documentation, the resulting cycle count needs to by decreased by 2, unless it's already 1 (due to pipelining), in which case it's decreased to 0 instead of -1.

All of this calculation can be performed in user space.

Since the board's \_\_CORE\_CLK (clock frequency) is 100,000,000 Hz (100 MHz), we can compute the time for each primitive operation. These are the timings for each primitive, plus the empty statement, using the formula in the ARM Cortex-M3 documentation in nanoseconds (ns, where 1,000,000,000 ns = 1 second):

* The empty statement takes 0 cycles (0 ns)
* p = request\_memory\_block(); takes 116 cycles (1160 ns)
* send\_message(caller\_pid, p); takes 788 cycles (7880 ns)
* q = receive\_message(&sender); takes 185 cycles (1850 ns)

We repeated 3 iterations per run and also ran the whole program multiple times to ensure consistency.

Here is the timing algorithm, which is repeated for each statement.

|  |
| --- |
| **ALGORITHM 1:** TEST\_TIME(statement) |
| start = get\_cycle\_count();  statement;  end = get\_cycle\_count();  print "statement took" compute\_difference(start, end) "cycles at" \_\_CORE\_CLK " Hz" |