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 function`memcpy`.

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

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.