Data storage in our linked structures

From the beginning of our project, we knew that we would need to store different types of data in linked structures for different use cases. For example, we knew that nodes in our memory heap structure would have different data storage requirements compared to nodes in our process control structures (e.g. ready queue, blocked queue).

Since we wanted to avoid creating linked structures that were coupled with the type of data they were storing, we needed to devise a way to make them as generic as possible.

We chose to make a generic node structure, `k_node_t`, with only one member: a pointer to the next `k_node_t`:

```
typedef struct k_node_t {
    struct k_node_t *p_next;
} k_node_t;
```

We then designed our linked structures to accommodate nodes of this type:

```
typedef struct k_list_t {
    k_node_t *p_first;
} k_list_t;

typedef struct k_queue_t {
    k_node_t *p_first;
    k_node_t *p_last;
} k_queue_t;
```

This way, if a certain use case necessitates having additional storage in each node, we can simply declare a new type with the additional required members. For example, here is a node that can store a pointer to a process control block:

```
typedef struct k_pcb_node_t {
    struct k_pcb_node_t *p_next;
    k_pcb_t *p_pcb;
} k_pcb_node_t;
```

As long as the first member remains a pointer to the next node, the structure members will line up in memory and our linked structures will accommodate these nodes (assuming we cast `k_pcb_node_t` pointers to `k_node_t` pointers on insertion, and vice versa on removal).

Modeling blocks in our memory heap

During the planning phase, we chose to implement our memory heap as a linked list. However, we were not sure how nodes in this linked list should keep track of the blocks that they would represent.

In addition to having a pointer to the next node, our first thought was that each node structure should have a pointer to the beginning address of the memory block that it represents:

```
typedef struct k_mem_blk_node_t {
    struct k_mem_blk_node_t *p_next;
    void *p_mem_blk_start_addr;
} k_mem_blk_node_t;
```

After further thought, we realized that since the size of each memory block is known at compile time, we can simply space the nodes apart at (sizeof(k_node_t) + BLOCK_SIZE) intervals, obviating the need for a member such as void *p_mem_blk_start_addr. Thus, we were able to use the unmodified node structure (k_node_t, outlined in the previous section) for our memory heap.

Pointer arithmetic

For our memory heap, we chose to structure blocks with the header in front of the block itself. This means that when we remove a node from our heap in k_request_memory_block(), we need to increment node's address by 4 bytes (the size of the header, sizeof(k_node_t)) before returning the address to the user. Similarly, in k_release_memory_block(void *p_mem_blk), we need to decrement the address of p_mem_blk by 4 bytes in order to get the correct pointer to the node for re-insertion into the heap.

When we originally implemented this pointer arithmetic in k_request_memory_block(), we did the following:

```
k_node_t *p_mem_blk = get_node(gp_heap);
p_mem_blk += sizeof(k_node_t);
```

If the value of p_mem_blk is, for example, 0x10000000, we would expect the resulting value to be 0x10000004. However, the above code produces a result of 0x10000010. This is because the compiler knows the size of k_node_t, and is multiplying the addend by sizeof(k_node_t), resulting in an addend of 16 instead of 4. Therefore, we revised the addend to 1 instead of 4:

```
k_node_t *p_mem_blk = get_node(gp_heap);
p_mem_blk += 1;
```

Since the block pointer we receive in k_request_memory_block(void *p_mem_blk) is not typed as a k_node_t, the compiler does not multiply the subtrahend by sizeof(k_node_t) for us. Instead of doing this multiplication ourselves, we chose to cast p_mem_blk as a pointer to a k_node_t and simply use a subtrahend of 1:

```
k_node_t *p_blk = p_mem_blk;
p_blk -= 1;
```

This will correctly subtract the size of our header from the address of the returned block.

Context switching

In order to context-switch to processes that have not yet executed, it is necessary to pop a pre-made exception stack frame off of the stack (by calling the inline assembly function `__rte()`) so that the new process has an initial context.

Since we only want to call `__rte()` if a process has not executed before, we needed to add a sixth process state, `NEW`. This gives us more fine-grained process control behavior for `NEW` vs. `READY` processes.

Process control blocks

At the beginning of our `memory_init()` function, we allocate an array of process control block (PCB) nodes (`k_pcb_node_t`), one for each process (`NUM_TEST_PROCS`). Since this is an array, it is indexed in array notation (`0` to `NUM_TEST_PROCS-1`).

When we need to access a PCB node for an arbitrary process, `p_process`, it makes sense to access this array using the process identifier (`p_process->m_pid`). However, process identifiers do not begin at 0; they begin at 1. So, in order to access the PCB node for `p_process`, we need to access our array of PCB nodes at index `(p_process->m_pid-1)`. Although this is a simple concept, it took an extraordinary amount of time to debug.

Dequeuing from the ready queue

In `request_memory_block()`, it is expected that the calling process may become blocked in the case that there is no available memory in the heap.

Originally, our implementation of `k_request_memory_block()` would attempt to dequeue the calling process from the ready queue and enqueue it in the blocked queue. This conceptually made sense, as it should be in the `BLOCKED_ON_RESOURCE` state, not the `READY` state, next time we call `k_release_processor()`.

However, we know that if a process is able to call `request_memory_block()`, then it must currently be in the `EXECUTING` state. Thus, it must have already been dequeued from the ready queue in order to have been scheduled in the first place. So, all we need to do when starved for memory is add the calling process to the blocked queue:

```
while (is_list_empty(gp_heap)) {    // no available memory in the heap
    if (k_enqueue_blocked_process(gp_current_process) == RTX_OK) {
        k_release_processor();
    }
}
```

Dynamic allocation in the kernel

In our initial implementation of the kernel, we dynamically requested and returned heap memory for PCB nodes as we enqueued and dequeued them from the ready and blocked queues. For naïve test processes, this was not a problem.

Once we started to request memory blocks in our test processes, we noticed a fatal flaw in this approach. If a process attempted to request a memory block when there was no available memory in the heap, we would attempt to add it to the blocked queue, as shown in the previous section.

However, since we were dynamically requesting heap memory for PCB nodes to be enqueued in the blocked queue (by calling `k_request_memory_block()`), we would never succeed in enqueueing the process in the blocked queue. Instead, we would infinitely recurse into `k_request_memory_block()`.

To solve this problem, we decided to statically allocate all of our PCB nodes in `memory_init()`, obviating the need to ever dynamically request memory in the kernel:

```
/* allocate memory for pcb node pointers */
gp_pcb_nodes = (k_pcb_node_t **)p_end;
p_end += NUM_TEST_PROCS * sizeof(k_pcb_node_t *);

/* allocate memory for each pcb node */
for (i = 0; i < NUM_TEST_PROCS; i++) {
    gp_pcb_nodes[i] = (k_pcb_node_t *)p_end;
    p_end += sizeof(k_pcb_node_t);
}
```