1 Initialization

At the start of our OS, we disabled interrupts before we initialize our system, otherwise our interrupt handlers cannot run properly. Once everything was initialized properly, we re-enabled interrupts and called our scheduler to dispatch the first process.

1.1 System APIs

The first stage of our initialization was to setup our system APIs. These allowed us to use `printf`, to use interrupt-driven IO on UART0, to use program-driven IO on UART1 (for debugging), and to receive timer interrupts every second.

- `SystemInit()`
- `init_printf(putc, NULL)`
- `uart_irq_init(0)`
- `uart1_init()`
- `timer_init(0)`

As an added bonus, we also enabled the use of our board's LCD screen. From the example code found in uVision's installation directory, we were able to write text and draw basic geometric shapes onto the LCD. Since the LCD API uses polling for IO like UART1, we decided that it was infeasible to use it for system functionalities such as the Wall Clock. Instead, we decided to make the LCD screen display our OS's logo, doge (See Appendix TODO for more background history).

1.2 Memory

After the system APIs have been initialized, we then initialized and allocated our system memory. We did this before any of our other code runs because the OS itself only contained pointers to our data structures. Since our data structures were allocated at this stage, any references to these structures would either hard fault or experience non-deterministic behaviour. The entire memory map of our system after initialization is shown on Figure 1.2.2.

1.2.1 Kernel Memory Space

After the process image, we dynamically allocated space for all of our kernel's data structures. We chose to allocate our data structures dynamically instead of at compile time because we were first "zeroing" out our entire memory space from `0x10008000` down to the end of our image with `0xDEADBEEF`. This made it easier to debug faults because if any registers (especially the link register) is `0xDEADBEEF` instead of 0 or some other random value, then we would instantly know that it was caused by uninitialized memory. In our kernel space, we initialized the following data structures:

- **5 Ready, 5 memory blocked, and 5 message blocked queues:**

  We had three states (ready, memory blocked, and message blocked) that each needed five queues, one for each priority. In total, we had 15 queues, each implemented as a doubly-linked list:

  ```
  linkedlist_t
      node_t* first;
      node_t* last;
      int length;
  ```

- **16 PCB Nodes:** (one for each PCB)

  Unlike a traditional linked list implementation, our linked list did not dynamically allocate and free memory. Instead, our implementation simply changed the pointers because we did not want our kernel to be blocked on memory or get pre-empted while it is running the scheduler. We wrapped our PCB struct with this node, instead of putting the next and previous pointers inside the PCB itself, was because we wanted the linked list implementation to be generic enough for other kernel features (such as the message queues).

  ```
  node_t
      node_t* next;
      node_t* prev;
      void* value;
  ```

- **16 PCBs:** (one for each process)

  Our PCBs themselves contain all of the information that the kernel needs to know about a process.

  ```
  pcb_t
      uint32_t pid;
      PROCESS_STATE state;
      uint32_t priority;
      uint32_t* stack_ptr;
      linkedlist_t msg_queue;
  ```

  The `PROCESS_STATE` type is a enum that we use to keep track of the current state. Currently we have 6 states: new, ready, running, memory blocked, message blocked, and exit.

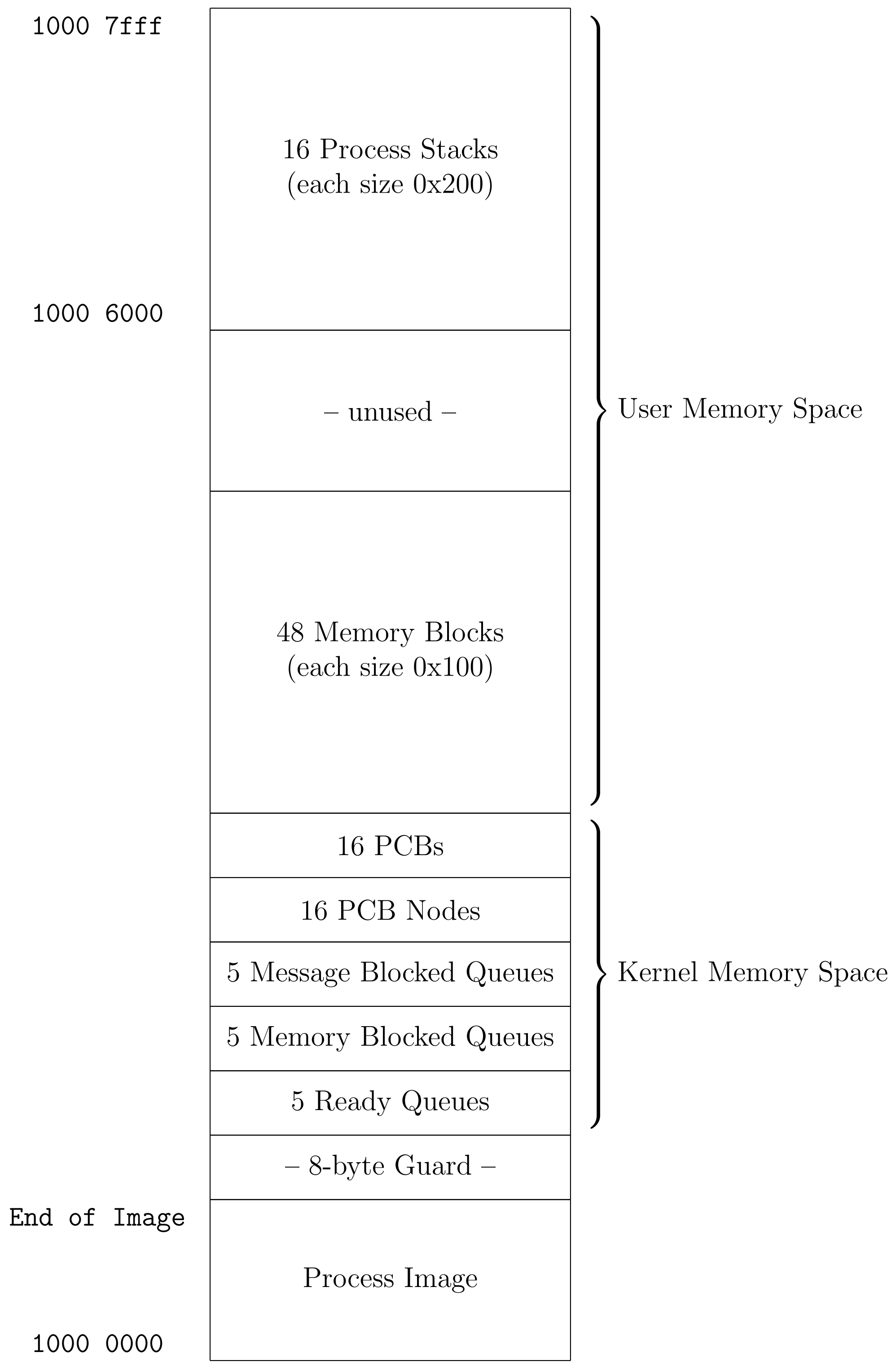
1.2.2 User Memory Space

After we finished initializing our kernel's memory space, we then used the space above the PCBs for our memory blocks. However, before we initialized the memory blocks, we first byte aligned our heap pointer after the PCBs with `0x100`. This made it easier to debug with the memory window at the cost of at most 0xFF memory.

The memory blocks themselves have the same "header" as a regular node because we can then cast the memory blocks to nodes and reuse our linked list implementation. See Section TODO for more details about the memory blocks.

```
mem_blk_t
    mem_blk_t* next;
    mem_blk_t* prev;
    void* data;
    uint32_t padding;
```

Figure 1: Memory Map after Initialization



1.3 Processes

After the memory had been allocated, we then move on to initializing our PCBs with their corresponding process information.

1.3.1 Process tables

First we call these methods to fill in our process tables:

```
set_test_procs() // sets test processes 1-6
set_user_procs() // sets stress tests A-C, wall clock, and change priority
k_set_procs() // sets the null process, KCD, CRT, and i-processes
```

These methods filled in our process tables (which had been allocated at compile time instead of at runtime in the previous step):

```
proc_image_t k_proc_table[NUM_K_PROCESSES] // size 5 (processes 0, 12-15)
proc_image_t g_test_procs[NUM_TEST_PROCESSES] // size 6 (processes 1-6)
proc_image_t u_proc_table[NUM_USER_PROCESSES] // size 5 (processes 7-11)
```

And each entry of our process table contained:

```
proc_image_t
    uint32_t pid
    PROCESS_PRIORITY priority
    func_ptr_t proc_start
```

The `PROCESS_PRIORITY` type is a enum that we use to designate the priority of a process from `HIGHEST` being 0 to `LOWEST` being 4. The `func_ptr_t` type is function pointer that points to where our process actually resides in memory.

1.3.2 Putting processes onto the ready queue

After our process tables were filled, we then iterated over all the processes in the process tables and setup our PCBs:

1. Copy the `pid` and `priority` into their corresponding PCBs.
2. Set the PCB state to `NEW`.
3. Initialize the PCB's message queue to be an empty linked list. Since the linked list was not declared as a pointer, its members (`first`, `last`, `length`) were already within the PCB struct.

4. Assign a stack of size `0x200` from the top of the RAM (at `0x1000 8000`) or just below the previously allocated stack to the current process. The `stack_ptr` in PCB itself, however, points 8 words below the start of the stack to allow the scheduler to work properly.

   As specified in the board's documentation, to go from handler mode (scheduler) to thread mode (a user process), we need to use the `BL 0xFFFF FFF9` instruction. This special instruction pops values off the stack frame, as shown below, and copies them into their corresponding registers.

Figure 2: Stack after calling `BL 0xFFFF FFF9`

– NULL –	← SP is here after resuming to thread mode
xPSR = `0x1000 0000`	
PC = `proc_start`	
LR = `0x0`	
R12 = `0x0`	
R3 = `0x0`	
R2 = `0x0`	
R1 = `0x0`	
R0 = `0x0`	← SP initially points here in handler mode

Normally, these values are be automatically set when we make a system call from thread mode. However because our OS is initialized in handler mode, we need to manually set these default values (as shown above) the first time so that the scheduler don't hard fault.

Also, although the i-processes did not actually need a stack because they used the kernel stack whenever they got invoked, it made initialization a lot easier because our loop did not need to check process types. In addition, it was also at no cost to us because we still had leftover free space in our RAM in the end.