**RTX Part 2a Document**

**System.h**

**Macros:**

* **PROCESS\_STACK\_SIZE 1024** – This is the amount of memory in bytes allocated to each process’ stack
* **KERNEL\_STACK\_SIZE 1024 –** This is used for the allocation of the kernel stack in memory
* **NUM\_PROCESSES 6 –** This is the number of processes. There are 6 for this part of the project.

**Structs:**

* **s\_pcb**
  + **UINT32 m\_process\_ID –** This is the process ID. With 6 processes, the process IDs go from 1 to 6.
  + **UINT8 m\_priority –** This is the priority level of each process. There are four priority levels, with 0 being the highest priority and 3 being the lowest priority.
  + **UINT8 m\_state –** This refers to the state of each process. 0 means the process is blocked, 1 means the process is ready, and 2 means the process is running.
  + **UINT32 m\_stack –** This is the current stack pointer of the process
  + **VOID (\*m\_entry)() –** This is the starting point of the test process function in memory. It is a pointer that points to that location in memory.

**System.c**

**Global Variables:**

* **struct s\_pcb g\_null\_proc –** This is the null process
* **struct s\_pcb g\_test\_proc\_table[NUM\_PROCESSES] –** This is an array of the processes being used for testing. It contains the IDs of the test processes.
* **struct s\_pcb g\_priority\_ready[4][NUM\_PROCESSES] –** This is a 2D array that acts as a queue for each priority level. Each priority level from 0 to 3 has room for 6 processes. This is redundancy which allows for the possibility of each process having the same priority.
* **SINT8 g\_priority\_ready\_tracker[4][2] –** This tracks the head and tail of each priority queue. The 0th index of the second array field is the head and the 1st index is the tail.
* **struct s\_pcb \*g\_current\_process –** A pointer to the address of the current process running
* **UINT32 \*g\_kernelStack –** A pointer to the address of the beginning of the kernel stack
* **UINT32 g\_asmBridge –** A temporary variable, meant to allow us to move values into registers using asm commands
* **UINT32 g\_first\_run –** A binary flag that signifies whether or not the scheduler has been run yet

**Functions:**

**VOID sys\_init()**

* **Purpose –** To simply initialize the process handling system. It runs only once at the beginning of execution
* **Pseudocode –** At first, the stack pointer A7 is set to point to the beginning of the kernel stack frame. This is done by setting A7 to &\_\_end. Then, asm\_trap\_entry is moved to memory at location 10000080. The priority queues are then initialized, by using two for loops, one nested inside the other. These nested for loops set each index of g\_priority\_ready\_tracker[][] to -1 initially. This is done because g\_priority\_ready\_tracker[][] is meant to keep track of the head and tail of the priority queues, and since there is nothing in those queues there are no heads or tails.   
    
  Next, the null process is set-up with initial values. Its process ID is set to 0, its priority is set to 4, its stack location is set to &\_\_end + KERNEL STACK SIZE and its entry is set to the memory location labeled “null\_process”. Then, a pointer (UINT32 \* addr) is used to set-up the exception stack frame of the null process. addr’s value is set to the null process’s stack location, and then it is pointed to the location of the null process entry. addr is then decremented, and its value is set to 0x40000000. The first word (4000) is meant to set the admin field of the ESF, and the second word (0000) is used to set the SR. For redundancy, a for loop is then used to decrement addr, and fill each memory location pointed to by addr to 0x00000000. This for loop is executed 16 times (0 to 15). This is to make sure that the stack is clean before adding any addresses or data registers to it.  
    
  Next, the processes are initialized. This is done by using a for loop that iterates once for each process (for a total of 6 times). During each iteration, the following occurs. The process ID is set to the loop index + 1. The priority of the process is set to 3. The location of its stack is set to &\_\_end + KERNEL STACK SIZE + (loop index + 1)\*(PROCESS\_STACK\_SIZE). The process’s state is set to 1. After these values are set, there is some debug output to help communicate with the user. The debug output states which process is running, where in memory its stack begins, and its entry location. Then the ESF for the process is constructed in the same method as described above for the null process. The last thing done in this for loop is adding the process to the ready queue, considering its priority. This is done by pushing it on to the ready queue. Outside of the for loop, the entry location for each test process is set to its respective memory location label (test\_proc\_1, test\_proc\_2, etc.). Next, the kernel stack location is saved by moving the address pointed to by A7 to a temporary global variable called g\_kernelStack. Scheduler() is then called which loads the next process’ stack pointer into another temp global variable g\_asmBridge. The value in g\_asmBridge is then moved to A7 so that the stack pointer now points to the new process’ stack. Lastly, all of the address and data registers are backed up onto the stack, and then rte is called.

**VOID scheduler(VOID)**

* **Purpose**
* **Pseudocode –** The first thing this function does is check to see if it is the first time the scheduler function has been called. This is done by checking the g\_first\_run variable, which is 0 if the scheduler has been previously called. If g\_first\_run is equal to 0, then the current process’s information is saved as follows: the stack location of the current process is set to the stack pointer plus an offset of 0xC and the state of the current process is set to 1. After the if statement that checked g\_first\_run, set g\_first\_run to 0 to record that the scheduler has been called.  
    
  To determine what the next process to switch to is, a for loop is used that attempts to pop the next ready process of highest priority from the priority ready queue. The pop will only succeed if that is the next process to be run. So, the for loop gets broken when the pop is successful. If no pops were successful at all, it means that no ready process was found. In this case, the null process needs to be started. This is accomplished by setting g\_current\_process equal to &g\_null\_proc. Finally, the process state of the selected process is set to “running” (which is a value of 2) and its stack is restored.

**SINT8 release\_processor()**

* **Purpose**
* **Pseudocode –** At the beginning of the function, the stack pointer is offset to ensure that the ESF doesn’t interfere with the return address of the process when it is released. Then the function issues a Trap #0 command to finish releasing the process. The function then returns 0.

**SINT8 send\_message(UINT8 process\_ID, VOID \* MessageEnvelope)**

* **Purpose –** To be implemented in part b
* **Pseudocode –** This function returns 0.

**VOID \* receive\_message(UINT8 \* sender\_ID)**

* **Purpose -** To be implemented in part b
* **Pseudocode –** This function returns (VOID\*)0.

**SINT8 set\_process\_priority(UINT8 process\_ID, UINT8 priority)**

* **Purpose –** This function is for changing the priority level of a process. The priority of the process should be changed, and then the process needs to be placed in the correct priority stack location. Returns 0 if successful, -1 if not.
* **Pseudocode –** This function iterates through the test process array, using a for loop to iterate once for each process. It checks the process ID passed in as an argument and compares it with the process ID of the process in the table at the given loop index. If the process IDs match up, the priority of that process is set to whatever priority is passed into the function. The function then returns 0. If the process ID does NOT match up with any of the processes in the table, the function returns -1.

**SINT8 get\_process\_priority(UINT8 process\_ID)**

* **Purpose –** This function is for returning the priority level of a process. It finds the process in the array and simply returns its priority. If unsuccessful, it returns -1.
* **Pseudocode –** This function also iterates through the test process array, using a for loop to iterate once for each process. For each iteration it checks to see if the process ID in the process table at the loop index matches the process ID passed in as an argument. If they match, the function simply returns the priority of the process in the table. If the process ID was never found in the loop, the function returns -1 instead.

**SINT8 pop(UINT8 priority, struct s\_pcb \*\* catcher)**

* **Purpose –** This function is called when switching processes. It is called from the scheduler function
* **Pseudocode –** Check if the list is empty. Return -1 if empty, since you can’t pop from an empty list. Otherwise, remove the head of the process list and then update head and tail pointers. The popped element is stored in a catcher variable which is passed as a parameter to the function. The function returns 0 if successful.

**SINT8 push(UINT8 priority, struct s\_pcb \* new\_back)**

* **Purpose –** The purpose of the function is to push the next ready process of the highest priority to the end of the process ready queue.
* **Pseudocode –** Check to see if the process queue is empty. If it is, simply set head and tail of the process queue to the new object being pushed. Else if check if the process queue is full (size equals 6). If it is, return -1 because nothing can be pushed. Else, the process queue is neither completely full nor empty. In this case, set the back pointer of the last object in the list to point to the new object, and set the tail pointer to point to the new object being pushed.