**RTX Part 2b Document**

**System.h**

**Macros:**

* **KERNEL\_STACK\_SIZE 2048 –** This is used for the allocation of the kernel stack in memory
* **NUM\_PRIORITIES 5 –** Number of priority levels that can be assigned to the processes
* **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.
  + **SINT8 msg\_waiting –** This is the ID of the process that is trying to send a message to this process. It is -1 if there is no message waiting.
  + **UINT32 msg\_box[NUM\_PROCESSES] –** This is an array that holds all of the messages that are inbound to the process
* **s\_pcb\_queue\_item**
  + **s\_pcb\_queue\_item \* next –** Points to the next item in the linked list
  + **s\_pcb \* data** – Pointer to the PCB data location in memory
* **s\_pcb\_queue**
  + **s\_pcb\_queue\_item \* front** – Pointer to the head of the linked list
  + **s\_pcb\_queue\_item \* back** – Pointer to the tail of the linked list
  + **UINT8 num\_slots** – Max number of items that the queue can hold

**System.c**

**Global Variables:**

* **struct s\_pcb\_queue g\_priority\_queues[NUM\_PRIORITIES] –** This is an array of the priority queues. There is one queue for each priority level.
* **struct s\_pcb\_queue\_item g\_queue\_slots[NUM\_PROCESSES] –** This is an array of ready queue slots.
* **struct s\_pcb\_queue g\_mem\_blocking\_queue[NUM\_PRIORITIES] –** Array of the priorities of all processes that are being blocked
* **struct s\_pcb\_queue\_item g\_mem\_blocking\_queue\_slots[NUM\_PROCESSES] –** This is an array of all of the processes that are blocked waiting on memory
* **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
* **extern UINT32 g\_free\_mem** – Keeps track of the beginning of the free memory region

**Functions (only ones relevant to part 2b) included):**

**VOID \* request\_memory\_block()**

* **Purpose ­–** Issues the trap command to request a memory block. Returns a pointer to the result from the memory block trap handler.
* **Pseudocode** – Initialize a void pointer called returnVal and set it to 0. Back up all address and data registers onto the stack. Issue a TRAP #3 command. Point returnVal to the result from the memory block trap handler. Restore all of the address and data registers by popping them off the stack. Return the variable returnVal.

**VOID request\_memory\_block\_trap\_handler()**

* **Purpose** – This is the function that gets called when the request memory block function issues a TRAP #3 command. It looks for a free block in memory and allocates it to the process requesting it.
* **Pseudocode**  - Initialize a void pointer called freeBlock and set it to 0. Using a for loop, iterate through gp\_mem\_pool\_lookup[] for each memory block in the memory pool. Look for the first block that is unallocated (gp\_mem\_pool\_lookup[i] equals 0). If one is found, mark it as allocated, and point freeBlock at the memory location of the free block.   
    
  If no unallocated memory block is found, this means the current process must be set to blocked**.** The current process is added to the appropriate blocking queue for its priority level. Release processor is called. Once a memory block becomes available, the function will resume at this point and point freeBlock to the now available memory location. The value pointed to by freeBlock is then passed into D2 so that request memory block can accept the value.

**SINT8 release\_memory\_block(VOID \* memory\_block)**

* **Purpose** – This function issues the trap command to release a specific block of memory that is being released by a process. It accepts a pointer to the memory block that is to be released. It returns a 0 if successful, and it returns -1 if unsuccessful.
* **Pseudocode** – Create an integer variable called returnVal and set it to 0. Back up all address and data registers onto the stack. Put memory\_block (passed as an argument) into D2 to pass to the trap handler. Issue a TRAP #4 command. Take the return value from D2 which was issued by the trap handler, and store it in returnVal. Restore all of the address and data registers by popping them off the stack. Return returnVal.

**VOID release\_memory\_block\_trap\_handler()**

* **Purpose** – This is the function that gets called when a TRAP #4 command is issued. It looks for the specified block in memory and releases it.
* **Pseudocode –** Create a void pointer called memory\_block. Store the pointer to the block that is to be released in this variable, by moving it from D2. This value was sent by the release memory block function. Iterate through the memory pool list and make sure there is a match with the memory block to be released. When it is found, set its flag to signify that it’s now unallocated.   
    
  Now, iterate through the list of processes that are blocked and waiting for memory. Unblock the highest priority process that is waiting for a memory block. If all of the blocked\_mem queues are empty, then no processes are waiting for a memory block and the function can simply return. The function sends the value 0 to D2 if it was successful. It sends the value -1 to D2 if it was not able to locate the memory block to be released, or it was already available in the first place.

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

* **Purpose –** Issues the trap command to move to kernel mode and call send message trap handler. Accepts process\_ID and MessageEnvelope as arguments. The former represents which process the message is being sent to, and the latter represents the contents of the message. Returns 0 if successful, -1 if not.
* **Pseudocode –** Backup all address and data registers onto the stack. Place process\_ID and messageEnvelope into D2 and D3 respectively to pass to the send message trap handler. Issue a TRAP #5 command. Take the return value from the trap handler, and store it in a variable called returnVal. Restore the address and data registers by popping them off the stack. Return returnVal.

**VOID send\_message\_trap\_handler()**

* **Purpose** – Gets called when TRAP #5 command is issued. It allows a process to send a message to another process.
* **Pseudocode** – Move process\_ID and messageEnvelope from D2 and D3 into local variables. Check if process\_ID is valid. If not, send message fails and it returns -1. This is done by creating a flag called process\_ID\_valid. It is initially set to -1. A loop iterates through the list of process\_IDs and compares each one to the process\_ID sent to the function as an argument. If it matches with one of the IDs, process\_ID\_valid is set to 0 which means that the process is valid.  
    
  Insert sender and receiver process IDs into MessageEnvelope and add the message to the receiving processes’ message box. Check if the receiving process is blocking while waiting for a message from the sending process. If it is, set its state to ready and add it to the appropriate ready queue. Since the message send was successful, return 0.

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

* **Purpose -** Issues the trap command to move to kernel mode and call receive message trap handler. Accepts sender\_ID as an argument representing the process ID of the process that is sending the message. Returns 0 if successful, -1 if not.
* **Pseudocode –** Backup all address and data registers onto the stack. Place sender\_ID into D2 to pass to the receive message trap handler. Issue a TRAP #6 command to call the receive message trap handler. Take the return value from D1 and store it in the variable called returnVal. Restore the contents of the address and data registers by popping them off the stack. Return returnVal.

**VOID receive\_message\_trap\_handler()**

* **Purpose –** Gets called when TRAP #6 command is issued. It allows a process to receive a message from a specified sender.
* **Pseudocode** – Move sender\_ID from D2 into a local variable. Iterate through the current process’ message box and check if there is a msg waiting from the process with the sender’s ID. If there is, return a pointer to that message. If there is not, the current process blocks and releases processor. Returns a 0 when successful.