ECE 354: Project 2 Part B

Group: ECE.354.S11-G031
Members: Ben Ridder (brridder), Casey Banner (cccbanne), David Janssen (dajjanss)

This document is concerned with the high level implementation of the specified primitives. As such, most of the low level back end is omitted for simplicity. Any kernel structures required to understand the primitive are described when the primitive is discussed.

RTX Initialization of Memory Management

The final phase of the RTX initialization is the memory management phase. Memory blocks are allocated beginning at the start of free memory. The function call `init_memory(void* memory_start)`, as seen in Listing 1. A free list is used to implement linking and management of the memory blocks. A free list is a singly linked list that stores the address of the next free memory in the first four bytes of the current memory block. The only pointer required is to the head of this list. Push and pop operations can be done in $O(1)$ time. Over-allocation and double deallocation prevention is handled by a bit field containing a bit for each memory block. When the memory is allocation the associated bit is set to 1 and for when it is released it is set back to 0.

Listing 1: Pseudo code for RTX initialization of memory management

```
void init_memory(void* memory_start){

    head of memory = start of free memory + (number of memory blocks - 1)*(size of
        memory block)

    for each block in free list:
        store address of next free memory block in current block

    initialize the memory allocation bit field to 0
}
```

Request and Release Memory Block

The `request_memory_block()` and `release_memory_block()` primitives are used by the operating system to allocate and deallocate memory blocks for processes.

Requesting a memory block is shown in pseudo code in Listing 2. First a mode switch to kernel mode is made. After the mode switch, a check is made to see if any memory blocks are available. If there are no available memory blocks, the process is moved to a blocked state and the processor is released. If memory is available, the memory block is popped off of the free list. The block index is retrieved and the appropriate memory allocation bit is set high. Finally, the memory block address is returned.

Listing 2: Pseudo code for requesting memory blocks

```
void* request_memory_block(){
    mode switch to kernel mode

    if no available memory blocks:
        running process state = blocked on memory
        release_processor()
    else:
        pop block of memory off the free list
        get block index
        set corresponding bit in memory allocation field to 1
```

```
    return address of block
}
```

Releasing a memory block is shown in the pseudo code in Listing 3. First a system call is made using the address of the memory block as a parameter. After the mode switch is performed, a check is made to determine the index in the allocation field of the block. This index is used to see if this block is currently allocated. If it is not then the system returns `RTX_ERROR`. If the block has been allocated then the block is pushed back on to the free list and that bit in the memory allocation field is set back to 0. Next, a check is made to see if there are any processes currently blocked on memory. If any are found, the first blocked process has its state set to ready and `RTX_SUCCESS` is returned. If there are no processes blocked on memory then `RTX_SUCCESS` is returned right away.

Listing 3: Pseudo code for releasing memory blocks

```
int release_memory_block(void* memory_block){
    mode switch to kernel mode

    if memory block is not allocated:
        return RTX_ERROR
    else:
        push block back on to free list
        set correct bit in memory allocation field to 0

            if there are processes blocked on memory:
                set state of first blocked process to ready

        return RTX_SUCCESS
}
```

Send and Receive Messages

The primitives `send_message(...)` and `receive_message(...)` are used by processes to send and receive messages between them. Each process has its own message queue which is setup upon process initialization. When a process sends a message it is enqueued to the message queue of the receiving process. When a process attempts to receive a message it polls its own message queue.

Listing 4 describes the functionality of `send_message(...)`. First, a system call is made with the process id of the receiver and a pointer to the message envelope. The system call then switches into kernel mode. Once the mode switch is complete the message is then pushed on the end of the recipient's message queue. There is no size limit on the message queue so this will never cause an error. A check is then made to see if the recipient process is currently blocked on receiving messages. If this is the case the state of this process is set to ready. Then `RTX_SUCCESS` is returned.

Listing 4: Pseudo code for sending messages

```
int send_message(int process_id, void* message_envelope){
    mode switch to kernel mode

    push message on to end of message queue

    if receiving process is blocked on messages:
        set state of receiving process to ready

    return RTX_SUCCESS
}
```

Pseudo code for `receive_message(...)` is shown in Listing 5. A system call is made to switch the system into kernel mode. In this system call a pointer for storing the process id of the sender is passed in as a parameter. Upon switching to kernel mode the running process checks to see if its message queue is empty. If the message queue is empty the process is now blocked on messages. The processes' state is changed to reflect this and release processor is called. If the queue is not empty the message is popped off the queue, the process id of the sender is written to the `sender_id` pointer, and the message is returned.

Listing 5: Pseudo code for receiving messages

```
void* receive_message(int* sender_id){
    mode switch to kernel mode

    if message queue for running process is empty:
        running process state = blocked on messages
        release_processor()
    else:
        pop message off queue
        *sender_id = process id of the message sender
        return message
}
```