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 RTX initialization of memory management is called by `init()`. This function then uses the beginning of free memory to call `init_memory(void* memory_start)`. The pseudo code for this initialization can be see in Listing 1. The data structure that is used to implement our memory management is known as a free list. A free list is 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 pointer required is to the head of the free list. This allows the list to be traversed in $O(1)$ time. In order to prevent against allocating the same block multiple times and also double deallocation a bit field with a bit for each memory block was created. This bit field stores a value of 0 if the memory is free and a value 1 if it is currently allocated, this check can also be made in $O(1)$.

Listing 1: Pseudo code for RTX initialization of memory management

```
void init(){
    init_memory(void* memory_start)
}

void init_memory(void* memory_start){
    declare local variables

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

    iterate over all blocks 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 as 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 releases the processor. If there are available memory blocks the block is popped off of the free list and a check is made to see what the index of the next available block is. That bit in the memory allocation field is then set to 1 and then the address of that block 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 in order to switch kernel mode. After the mode switch a check is made to determine the index in the allocation field of the block. This index is then used to see if this block is currently allocated. If it is not currently allocated 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. A check is then made to see if there are any processes currently blocked on memory. If there are then 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 set to NULL when the process is initialized. Whereas, 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.

The pseudo code for `send_message(...)` can be seen in Listing 4. First a system call is made using the process id of the process intended to receive the message and the message itself as arguments. The system call then switches into kernel mode. Once the mode switch is complete the message is then pushed on the end of the 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 process being sent a message is currently blocked on messages. If it is 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 paramater. 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, its 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 passed in 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
}
```