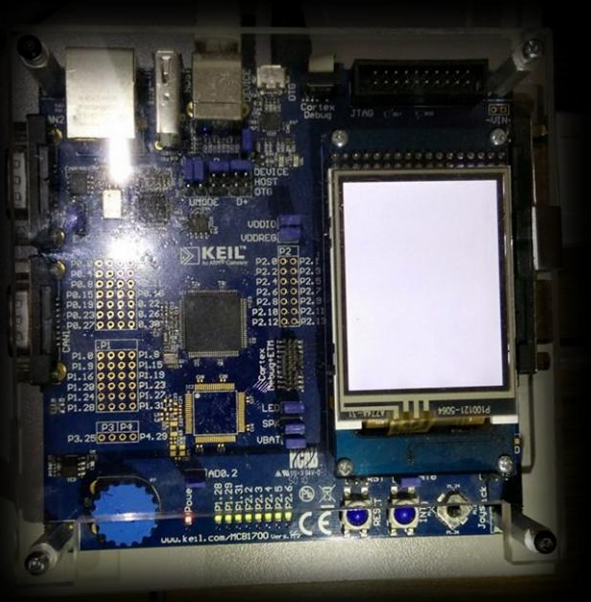

SE350 RTX Project Documentation

Angad Kashyap (20466050)
askashya@uwaterloo.ca

Mike Wang (20465276)
yc8wang@uwaterloo.ca

Tong Tong (20460179)
t7tong@uwaterloo.ca

Vishal Bollu (20474731)
vbollu@uwaterloo.ca

University of Waterloo

3A Software Engineering

4/6/2015

Table of Contents

List of Figures

List of Tables

1 Introduction

This document outlines the key concepts and structure used to build a small real-time executive (i.e. operating system) on the Keil MCB1700 board with an NXP LPC1768 microcontroller. It describes a very basic implementation of a kernel that supports context switching and uses memory management techniques in order to efficiently handle high process load and stress. This operating system implementation makes use of priority levels, pre-emption, message-based interprocedural communication, simple timing services, system console I/O and debugging features.

The purpose of this documentation is to outline the exact components and specifications of our implementation of the operating system. This document provides high level explanations and descriptions of the way in which the different components interact. It also summarizes the decisions made and why certain implementations were chosen with respect to others.

The rest of this document describes each of the components of the operating system, including global variables, memory and process initialization, the core kernel application programming interface, interrupts and their respective handlers, system and user processes and quality assurance.

2 Software Design Description

2.1 GLOBAL VARIABLES

The following subsections describe the global variables used throughout the operating system.

2.1.1 Memory

`gp_stack`

This is a pointer to the top of the stack at the RAM high address and is used in memory initialization. The stack grows downwards towards lower addresses.

`beginHeap, p_end`

These are pointers to the start and end addresses of the heap in memory respectively.

`beginMemMap`

This is an array with a size equal to the number of memory blocks. This array represents whether a memory block has been taken are not: 0 represents a free block and 1 represents a block in use.

2.1.2 Processes

`gp_pcbs`

This is an array of pointers to process control blocks (PCBs) and has an array size equal to the number of processes. The PCB data structure keeps track of a process' main stack pointer, process identifier (PID), state, priority and messaging queue.

`gp_pcb_nodes`

This is an array of PCB node pointers and has an array size equal to the number of processes. The PCB node data structure consists of a pointer to the next PCB node and a pointer to a process' PCB.

`g_proc_table`

This is an array of `PROC_INIT` pointers and has an array size equal to the number of processes. The `PROC_INIT` data structure keeps track of the process' priority, PID, stack size, starting address pointer and stack pointer.

gp_current_process

This is a pointer to the PCB of the currently executing process.

ready_priority_queue

This is an array of priority queues that holds PCB nodes of the same priority in each queue. A PCB node is enqueued whenever its associated process is ready for execution and dequeued when its associated process is the next to be run. This is used mainly for context switching.

blocked_on_memory_queue

This is an array of priority queues that holds PCB nodes of the same priority in each queue. A PCB node is enqueued whenever its associated process is blocked because it is waiting for memory to be allocated to it. A PCB node is dequeued when its associated process receives the memory it was waiting for. This is used mainly for context switching.

blocked_on_receive_list

This is a PCB node linked list that tracks all of the processes that are waiting to receive a message. When a process receives the message it was waiting for, its PCB node is removed from the linked list.

g_kc_reg

This is an array of `KC_LIST` data structures which store registered keyboard commands and the associated PID.

g_timer_count

This is a counter for the timer i-process that keeps track of the current system time. This is used for both the wall clock process as well as the delayed messages. The counter increments every millisecond when a timer interrupt occurs.

show_wclock

This is an integer boolean variable used to determine whether the clock is currently being shown in the console.

2.1.3 Interprocess Communication

t_queue

This is a message queue that keeps track of all the sent messages that have not expired as of the current system time count. The contents of the queue are kept in sorted order with the closest-to-expiring messages at the head. The messages are dequeued and placed in the appropriate process' message queue when they have reached their expiry date.

g_input_buffer

This is a char array that is used as a buffer to hold the keyboard input.

g_input_buffer_index

This is an index counter for the next available place in the char array.

g_curr_p

This is a pointer to the current message that is being printed to console in the UART i-process.

g_char_out_index

This is an index counter that specifies the next character in the current message to be printed to console.

2.2 INITIALIZATION

2.2.1 Operating System Parameters

The following is a list of the operating system parameters which can be modified to tune the operating system:

- `USR_SZ_STACK` – the stack size allocated to each process in bytes
- `MEMORY_BLOCK_SIZE` – the size of each memory block in the heap in bytes
- `NUM_OF_MEMBLOCKS` – the maximum number of memory blocks
- `NUM_TEST_PROCS` – the maximum number of user test processes
- `KC_MAX_CHAR` – the maximum length of each keyboard command identifier
- `KC_MAX_COMMANDS` – the maximum number of registered keyboard command identifiers
- `MAX_ENVELOPE` – the maximum number of messages in use at a time

2.2.2 Memory Initialization

The memory portion of the operating system initialization takes care of allocating memory for PCBs and setting up memory management structures. It starts with placing 4 bytes of padding at the bottom of the usable RAM (adding 4 bytes to the `p_end` pointer starting at the lower address). Figure 3 shows how the rest of memory is allocated.

```
1  void memory_init(void)
2  {
3     lowAddrOfRAM ← &Image$$RW_IRAM1$$ZI$$Limit;
4
5     // calculate beginning and end address of PCB pointers
6     globalPCBs ← lowAddrOfRAM;
7     lowAddrOfRAM ← lowAddrOfRAM + NUM_PROCS * sizeof(PCB *);
8
9     allocate memory and assign pointers for each PCB;
10
11    // calculate beginning and end address of PCB node pointers
12    globalPCBNodes ← lowAddrOfRAM;
13    lowAddrOfRAM ← lowAddrOfRAM + NUM_PROCS * sizeof(PCB_NODE *);
14
15    allocate memory and assign pointers for each PCB node;
16    prepare for alloc_stack() to allocate memory for stacks;
17
18    // Calculate beginning of heap pointers
19    beginningOfMemoryMap ← lowAddrOfRAM;
20    beginningOfHeap ← beginningOfMemoryMap + NUM_OF_MEMBLOCKS;
21
22    initialize each memory block flag with 0; // indicating free blocks
23 }
```

Figure 1: Memory initialization pseudocode

Figure 2 shows a visual representation of the memory allocation. Figure 3 explains the heap structure.

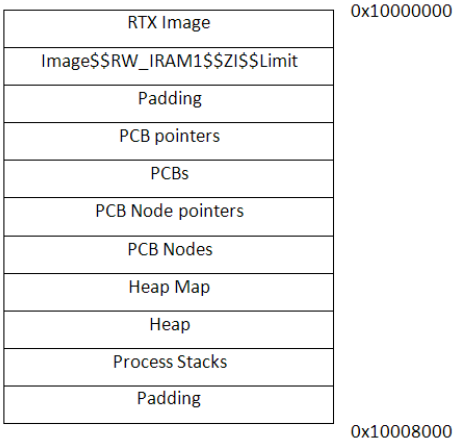

Figure 2: Memory allocation diagram

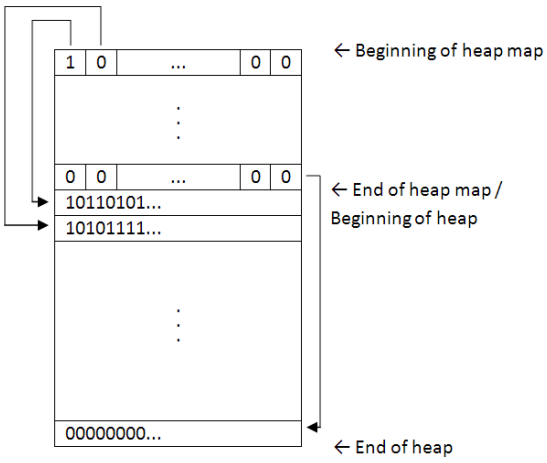

Figure 3: Heap structure

2.2.3 Process Initialization

Process initialization begins with placing all of the information related to each process into a global process table. This information includes the process identification number (PID), the priority, the stack size and the starting PC value for the process. The processes and their associated PIDs are shown in Table 1. New PCBs are created for each of the processes in the global process table. Here, stack space is allocated for the PCBs using `alloc_stack`. At the same time, PCB nodes are created for each process to be used in the ready priority queue and the blocked queues. The blocked on receive queue and blocked on memory queue are both initialized to be empty. Then all of the PCB nodes of processes that are not interrupt handlers are enqueued in the ready priority queue according to their priorities. The global keyboard commands array is also initialized to be empty.

Process	PID	Process	PID
null process	0	stress test B	8
user process 1	1	stress test C	9
user process 2	2	set priority command process	10
user process 3	3	wall clock process	11
user process 4	4	KCD process	12
user process 5	5	CRT process	13
user process 6	6	timer i-process	14
stress test A	7	UART i-process	15

Table 1: Processes

2.3 MEMORY MANAGEMENT

2.3.1 Requesting Memory

When a process requests a memory block, a check for free memory blocks is performed. As long as all of the memory blocks are in use, the current process is pushed into the blocked on memory queue and `k_release_processor` is called so that the next process with a ready state can be executed while this process waits for memory. If there are memory blocks available, a memory block will be given to the current process and its status in the memory map will be set to taken. Figure 4 shows the pseudocode for requesting memory. The method returns the address of the memory block to be allocated to the waiting process.

```
1   void *k_request_memory_block (void) {
2       rVoid ← beginningOfHeap;
3       atomic (on);
4       while (memory is empty)
5       {
6           k_block_current_processs();
7           atomic (off);
8           k_release_processor();
9           atomic (on);
10      }
11
12      mark the memory block as taken in the memory map;
13      atomic (off);
14      return (void*) (rVoid + i * MEMORY_BLOCK_SIZE);
15  }
```

Figure 4: Requesting memory blocks pseudocode

2.3.2 Releasing Memory

When a process no longer needs a memory block, it will invoke the method described in Figure 5. This method checks the validity of the memory block pointer. If it is a valid pointer that points to the beginning of a memory block, then the memory block's associated memory map element is updated with an available status. If the blocked on memory queue is not empty, the first process with the highest priority is removed from it and is added to the appropriate ready queue. `k_release_processor` is called to see if the current process needs to be switched out.

```
1   int k_release_memory_block(void* memoryBlock) {
2       atomic (on);
3
4       if (memoryBlock  is NULL or outside allocated memory space) {
5           return RTX_ERR;
6       }
7
8       index ← (memoryBlock - beginningOfHeap) / MEMORY_BLOCK_SIZE;
9       beginningOfMemoryMap[index] = 0;
10      atomic (off);
11
12      k_ready_first_blocked(); // unblock the first process blocked on memory
13      k_release_processor();
14      return RTX_OK;
15  }
```

Figure 5: Releasing memory blocks pseudocode

2.4 PROCESSOR MANAGEMENT

2.4.1 Scheduling

The scheduler is invoked when there is a need to context switch into the next process. This method first checks if the `uart_preemption_flag` is set to true. If it is, then the scheduler returns the UART interrupt handler. If it is not set to true, then the scheduler iterates through the ready priority queue and dequeues the PCB node with the highest priority using a first-in first-out policy. The scheduler then returns a pointer to this PCB node. Note that the null process is in the ready priority queue at the lowest level. Figure 6 provides the pseudocode for the scheduler.

```
1  PCB* scheduler(void) {
2      // check if there are any system processes ready to run
3      if(readyPriorityQueue[SYS_PROC] is not empty){
4          return (dequeue readyPriorityQueue[SYS_PROC]).pcb;
5      }
6
7      // check the user procs (last/default is the null process)
8      // return the PCB of the highest priority
9      for i = 0 to 4 do {
10         if(readyPriorityQueue[i] is not empty){
11             return (dequeue ready_priority_queue[i]).pcb;
12             }
13         }
14     return NULL;
15 }
```

Figure 6: Scheduler pseudocode

2.4.2 Process Priority

Each process has its own priority which is set at initialization as well as through the `k_set_process_priority` method. The priority of a process is what the operating system uses to determine which process is to be run next and which process should receive the next available memory block.

The `k_set_process_priority` method takes in a PID and a new priority value, and changes the priority of the process with the associated PID. If the modified process does not have a blocked state, it will be dequeued from its current queue in the ready priority queue and re-inserted back into the appropriate queue. Also, if this modified process has a higher priority than the currently running process, then `k_release_processor` will be called to pre-empt the current process with the modified process. Figure 7 presents this logic workflow.

```
1   int k_set_process_priority(int pid, int priority) {
2       if (pid is invalid or priority is invalid) {
3           return RTX_ERR;
4       }
5
6       node ← globalPCBNodes[pid];
7
8       if(node.pcb.priority is not priority) {
9           if (node.pcb.state is not BLOCKED_ON_MEMORY and
10              node.pcb is not globalCurrentProcess) {
11              remove node from ready readyPriorityQueue;
12              node.pcb.priority ← priority;
13              enqueue node onto readyPriorityQueue;
14          }
15
16          node.pcb.priority ← priority;
17          k_release_processor();
18      }
19      return RTX_OK;
20  }
```

Figure 7: Set process priority pseudocode

The `k_get_process_priority` method takes in a PID and returns the priority value for the node with that PID. An RTX_ERR value is returned if the process with the given PID is not found in the `gp_pcb_nodes` array.

2.4.3 Context Switching

Context switching occurs as a process runs to completion or if the operating system deems that there is a need for pre-emption. The `k_release_processor` method keeps track of the current running process, giving it an alias of the old process as it searches for a process to pre-empt with. If the old process is not null and has a run state, then its state will be changed to ready and the process' PCB will be enqueued in the appropriate ready queue. This method changes the current process pointer to the new process as returned by the scheduler. If the old process is null, which is the case the operating system first starts up, the old process is set to the result of the scheduler. Then `process_switch` is called on the old process to set the chosen process as the current process and execute it. Figure 8 shows the entire pseudocode of method.

```
1  int k_release_processor (void) {
2      oldPCB ← globalCurrentProcess;
3      if (globalCurrentProcess is not NULL)
4      {
5          if(globalCurrentProcess.state is RUN){
6              globalCurrentProcess.state ← RDY;
7              enqueue globalCurrentProcess' PCB node onto readyPriorityQueue;
8          }
9      }
10
11     globalCurrentProcess ← scheduler(); // get the next process to run
12
13     if ( globalCurrentProcess is NULL  ) {
14         globalCurrentProcess ← oldPCB; // revert back to the old process
15         return RTX_ERR;
16     }
17
18     if (oldPCB is NULL ) {
19         oldPCB ← globalCurrentProcess;
20     }
21
22     process_switch(oldPCB); // context switch
23     globalCurrentProcess.state ← RUN;
24     return RTX_OK;
25 }
```

Figure 8: Release processor pseudocode

2.5 INTERPROCESS COMMUNICATION

2.5.1 Sending Messages

Processes send messages to each other through the envelope data structure which contains the following information about a message: sender PID, receiver PID, message type, delay in milliseconds, message (char array) and a pointer to the next message which is used in the timeout queue for delayed send messages.

The `k_send_message` method takes in a PID and a pointer to an envelope. The envelope is enqueued to the message queue of the target process. If the target process is in the blocked on receive queue, it is removed and added to the ready priority queue. The pseudocode can be found in Figure 9. A call to `k_release_processor` to switch the current process with the target process would only occur if the target process priority is greater than the current process' priority and if all of the expired messages have been delivered to their respective receiving processes.

```
1  int k_send_message (int targetPID, void* msgEnvelope) {
2      atomic (on);
3      enqueue msgEnvelope onto targetProcess.msgQueue;
4
5      if (targetProcess.state is BLOCKED_ON_RECEIVE) {
6          remove targetProcess from blockedOnReceiveList;
7          targetProcess.state ← READY;
8          enqueue targetProcess onto readyPriorityQueue;
9
10         if (targetProcess.priority > currentProcess.priority){
11             k_release_processor ();
12         }
13     }
14     atomic (off);
15     return RTX_OK;
16 }
```

Figure 9: Sending a message pseudocode

2.5.2 Receiving Messages

Receiving messages requires, as a parameter, a pointer to an integer representing the sender PID. If the receiving process has an empty message queue, its state is changed to blocked on receive and it is added to the blocked on receive list. Then `k_release_processor` would be called so that a new process can be executed. If the message queue is not empty, then dequeue the first message. Figure 10 presents the full pseudocode for the `k_receive_message` method.

```
1  void* k_receive_message (int* senderID) {
2      atomic (on);
3      get currentProcess;
4
5      while (currentProcess.msgQueue is empty) {
6          currentProcess.state ← BLOCKED_ON_RECEIVE;
7          add targetProcess to blockedOnReceiveList;
8          k_release_processor ();
9      }
10
11     msgEnvelope ← dequeue currentProcess.msgQueue;
12     senderID ← msgEnvelope.senderPID;
13     atomic (off);
14     return (void*) msgEnvelope;
15 }
```

Figure 10: Receiving a message pseudocode

2.5.3 Delayed Messages

Sending delayed messages is similar to sending regular messages, as seen in Figure 11, except with the addition of a delay value which tells the timer i-process to send the messages only when they have expired (i.e. the current system time has met or exceeded the delay value).

```
1  int k_delayed_send (int targetPID, void* msgEnvelope, int delay) {
2      atomic (on);
3      msgEnvelope.delay ← globalTimerCount + delay;
4      k_send_message (TIMER_PID, msgEnvelope);
5      atomic (off);
6      return RTX_OK;
7  }
```

Figure 11: Sending a delayed message pseudocode

2.6 INTERRUPT PROCESSES

The operating system has two processes, timer i-process and UART i-process, that are never executed using the ready priority queue, blocked on memory queue and blocked on receive list scheduling structure. Instead, they are scheduled by their interrupt handlers which are called when an interrupt signal is received. Before the i-process associated with a particular interrupt is executed, the currently running process' context would be saved on the stack. During the execution of an i-process, the previously running process may be pre-empted if the i-processes causes a higher priority process to be unblocked. The saved context is popped off the stack when the i-process has run to completion and the previously running process will continue its execution if it has not been pre-empted.

2.6.1 Timer I-Process

Since the operating system needs the ability to send delayed messages, timing services are required. One of the MCB1700 board timers, TIMER0, is used in the implementation of the timer interrupts which fires every millisecond. Each time the timer interrupt fires, the `timer_i_proc` method – the timer i-process – is invoked, dispatching the expired messages when appropriate. This i-process uses the `g_timer_count` value to determine whether a message has expired and sends the message to the specified process by enqueuing it in the process' message queue. Figure 12 shows a simplified version of the timer i-process pseudocode.

```
1   void timer_i_proc (void) {
2       atomic (on);
3       LPC_TIM0.IR ← (1 << 0);
4       envelope ← k_non_blocking_receive_message (TIMER_PID);
5
6       while (envelope not NULL) {
7           insert envelope into timeoutQueue in sorted order by delay;
8           envelope ← k_non_blocking_receive_message (TIMER_PID);
9       }
10
11      while (timeoutQueue.head is not NULL and
12              timeoutQueue.head.delay <= globalTimerCount) {
13          cur ← dequeue timeoutQueue;
14          k_send_message (cur.destination_pid, cur);
15      }
16
17      globalTimerCount++;
18      atomic (off);
19      k_release_processor ();
20  }
```

Figure 12: Timer i-process pseudocode

2.6.2 UART I-Process

The UART i-process is another interrupt process and has two main responsibilities:

1. Reading input in from the keyboard – This requires parsing through a sequence of characters terminated by the endline character. Once the input has been taken in, the UART i-process sends a message to the KCD process with the entire input as the message content.

2. Printing output to console – The CRT process sends messages to the UART i-process to have them be printed to console.

Figure 13 shows the pseudocode for both reading in input and printing output.

The UART i-process also provides three debug hotkeys that allows the developer to print the state of the processes at any given time using three special characters:

- ! prints the PID of the currently running process and the contents of the ready priority queue according to the priorities of each process by printing their PIDs

- @ prints the contents of the blocked on memory queue according to the priorities of each process by printing their PIDs

- # prints the contents of the blocked on receive list by printing their PIDs

```
1  void uart_i_proc (void) {
2      atomic (on);
3
4      if (UART0.IIR is in read status) {
5          envelope ← k_request_memory_block ();
6          envelope.message_type ← MSG_CRT_DISPLAY;
7          envelope.message ← characterRead;
8          k_send_message (CRT_PID, envelope);
9
10         if (characterRead is not '\r') {
11             put characterRead in globalInputBuffer;
12         } else {
13             put end of line character in globalInputBuffer;
14
15             envelope ← k_request_memory_block ();
16             envelope.message_type ← MSG_CONSOLE_INPUT;
17             envelope.message ← globalInputBuffer;
18             k_send_message(KCD_PID, envelope);
19         }
20     } else if (UART0.IIR is in write status) {
21         if (globalCurrentEnvelope  is NULL) {
22             globalCurrentEnvelope ← k_non_block_receive_message(UART_IPROC_PID);
23         }
24
25         if (globalCurrentEnvelope is not NULL) {
26             UART0.THR ← globalCurrentEnvelope.message [globalOutCharacterCounter];
27             globalOutCharacterCounter ++;
28
29             if (globalCurrentEnvelope.message [globalOutCharacterCounter - 1] is '\0') {
30                 unmark UART0 for output;
31                 k_non_block_release_memory_block(globalCurrentEnvelope);
32                 globalOutCharacterCounter ← 0;
33             }
34         }
35     }
36     atomic (off);
37 }
```

Figure 13: UART i-process pseudocode

2.7 SYSTEM PROCESSES

2.7.1 Null Process

The null process is a simple infinite loop that just keeps calling k_release_processor. It has the lowest priority and should only run if there are no other processes that can or needs to be executing at a given time.

2.7.2 KCD Process

The Keyboard Command Decoder (KCD) process, described in Figure 14, receives messages from the UART i-process and decides what to do with the message content. If the message is a command registration request, the KCD process will add the new identifier to the `g_kc_reg` array and associate it with the specified PID. Otherwise, the KCD process will redirect the message to whichever process that was registered with the command identifier in the message.

```
1  void kcd_proc (void)
2  {
3      while (true) {
4          msgEnvelope ← receive_message (sender);
5          if (msgEnvelope.message_type is MSG_COMMAND_REGISTRATION) {
6              if (there is an available spot in the globalKeyboardCommands array) {
7                  register the command with the associated PID;
8              }
9          } else if (msgEnvelope.message_type is MSG_CONSOLE_INPUT) {
10             command ← command portion from msgEnvelope.message
11
12             for j ← 0 to KC_MAX_COMMANDS – 1 do {
13                 if (command equals globalKeyboardCommands[j].command) {
14                     KCDMsgEnvelope ← request_memory_block();
15                     KCDMsgEnvelope.message_type ← MSG_KCD_DISPATCH;
16                     KCDMsgEnvelope.message ← msgEnvelope.message;
17                     send_message(globalKeyboardCommands[j].pid, KCDMsgEnvelope);
18                     break;
19                 }
20             }
21             release_memory_block(msgEnvelope);
22         }
23     }
24 }
```

Figure 14: KCD process pseudocode

2.7.3 CRT Display Process

The CRT display process receives messages from the KCD process and redirects the message body to UART i-process for printing out to console. It also triggers UART interrupts by marking the pointer to the memory mapped device UART for output. This ensures that the UART i-process will run and print the sent message. Figure 15 shows the pseudocode for the CRT display process.

```
1  void crt_proc (void)
2  {
3     while (true) {
4        envelope ← receive_message (NULL);
5        pUart ← LPC_UART0;
6
7        if (envelope.message_type is MSG_CRT_DISPLAY) {
8           send_message (UART_IPROC_PID, envelope);
9           mark pUart for output;
10       } else {
11          release_memory_block (envelope);
12       }
13    }
14 }
```

Figure 15: CRT display process pseudocode

2.8 USER PROCESSES

2.8.1 24 Hour Wall Clock Display Process

The wall clock process prints a digital clock to console in the format of HH:MM:SS with values depending on the command that was given. The process begins with registering the following command identifiers:

- "%WR" for resetting the wall clock to 00:00:00
- "%WS hh:mm:ss" for setting the wall clock to the values as specified by hh:mm:ss
- "%WT" for terminating the wall clock and stop it from displaying in the console

This process sends delayed messages to itself every second to tell itself to update the time value and prints the new time to console by sending a message with this new time to the CRT process.

2.8.2 Set Priority Command Process

The set priority command process begins with registering the "%C" command identifier with the KCD process. The arguments following the command identifier are the PID of the process to be modified and priority value that the specified process' priority should be set to. The set priority command process checks for valid input and prints out an error message if any one of the arguments is invalid. If the input is indeed valid, `set_process_priority` will be called with the two arguments as the required parameters.

2.8.3 User Tests

Throughout the development of the operating system, correct behaviour for all functionalities was ensured through testing. With every modification, the new code was tested to confirm that there were no regression bugs. The new code was tested manually and automatically. The automated tests were implemented in the user processes where each user process tested a separate basic functionality.

All the test processes originally start with a priority of `LOW`. Process 1 requests two memory blocks and does a delayed send to Process 2 and 3 at 4 seconds and 6 seconds respectively. It then pre-empts Process 3 by changing its priority to `HIGH`. Next, Process 3 runs and attempts to receive a message but gets blocked on receive because its message box is empty. Process 2 will then run and it will get blocked on receive as well. Process 4 will run, request a memory block and pre-empt Process 5 by changing its priority to `HIGH`. Process 5 gets blocked on receive, causing Process 6 to run. Process 6 attempts to request all of the memory blocks (which is 32 blocks for this version of the operating system) but it gets block on memory because 3 memory blocks are already in use. Process 1 will run to completion and invoke a context switch. This triggers Process 4 to run while Process 2 and 3 are blocked on receive and it sends a message to Process 5, unblocking it. Process 5 then runs to completion, freeing the memory block used in its message. Process 6 gets unblocked as a result but is blocked again after consuming one more memory block. The operating system cycles between Process 1, 4 and 5 until the timer completes a delayed send to Process 2 after 4 seconds has elapsed since the start of the tests. Process 2 gets unblocked and it frees the memory block used by its message, unblocking Process 6. Process 6 then gets blocked again after it consumes the released memory. The operating system now cycles between Process 1, 4, 5 and 2 until the timer completes a delayed send to Process 3 after 6 seconds has elapsed since the start of the tests. Then Process 3 gets unblocked and releases a memory which causes Process 6 to be unblocked. Process 6 consumes the released memory, at which it has acquired all of the memory blocks. Finally, it frees all of the memory blocks, completing the execution of this integration test suite.

Additionally, the debugger was used to perform manual testing. This testing acted as a basic sanity check to ensure that the KCD, CRT and other input dependent processes ran as expected. Unit tests were not used because testing processes such as the CRT and the wall clock required checking the display which was not possible. However the use of the automated integration tests along with manual testing were sufficient in reducing the number of bugs in the source code.

3 Timing Analysis

In order to time the functions as accurately as possible, a new timer needed to be implemented that does not use interrupts to increment a value. The functions that need to be timed disable interrupts so while these functions are running, timers using interrupts will not increment as desired. Therefore a new timer was needed:

```
1  pTimer = (LPC_TIM_TypeDef*) LPC_TIM1;
2  pTimer->PR = 0;
3  pTimer->TCR = BIT(0);
4  function_timer = &pTimer->TC;
```

Figure 16: Timing analysis timer

This new timer will increment the value in the memory address `pTimer->TC` once every 800 nanoseconds, calculated by:

$$2 \times (1 + 0) \times \frac{1}{25} \times 10^{-6} = 800 \text{ ns}$$

As an experiment, two different processes were run to collect time.

3.1 Experiment 1

Figure 17 presents the logic workflow for the first experiment.

```
1  for i = 1 to 30{
2      request_memory_block
3      send_message(to self)
4      receive_message
5      release_memory_block
6  }
```

Figure 17: Experiment 1 pseudocode

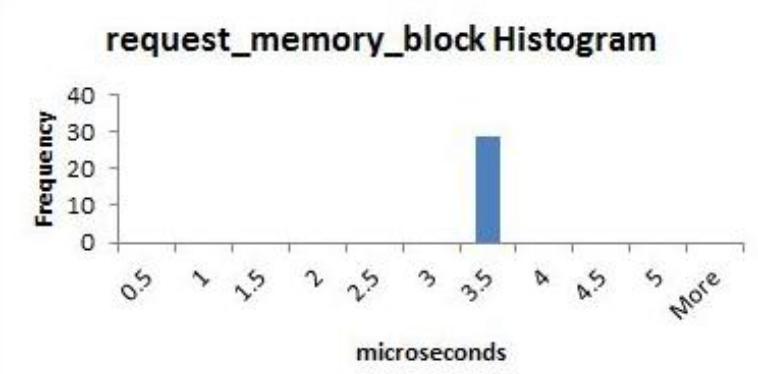

Figure 18: Results for requesting a memory block in Experiment 1

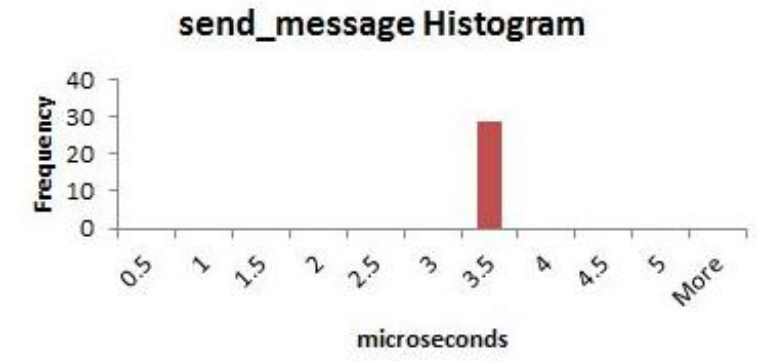

Figure 19: Results for sending a message in Experiment 1

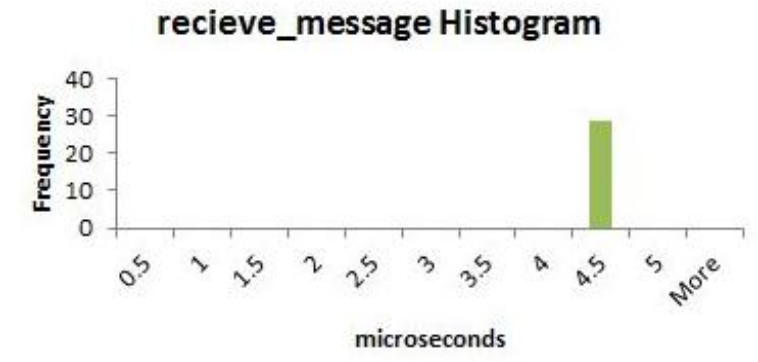

Figure 20: Results for receiving a message in Experiment 1

Summary of Results (no interference)

Function	Average Time (microseconds)
`request_memory_block`	3.44
`send_message`	3.44
`recieve_message`	4.4

Analysis

`request_memory_block`, `send_message` and `receive_message` always take the same amount of time: 3.44 microseconds, 3.44 microseconds and 4.4 microseconds respectively. The results are consistent because the function that executed this experiment was at the highest priority level and all of the memory blocks are free. In addition, there were no interrupts from the user or timer because the experiment finished before 1 millisecond had elapsed.

3.2 Experiment 2

The second experiment's pseudocode is shown in Figure 21.

```
1  for i = 1 to 29 {
2  request_memory_block
3  send_message(to self)
4  receive_message
5  }
```

Figure 21: Experiment 2 pseudocode

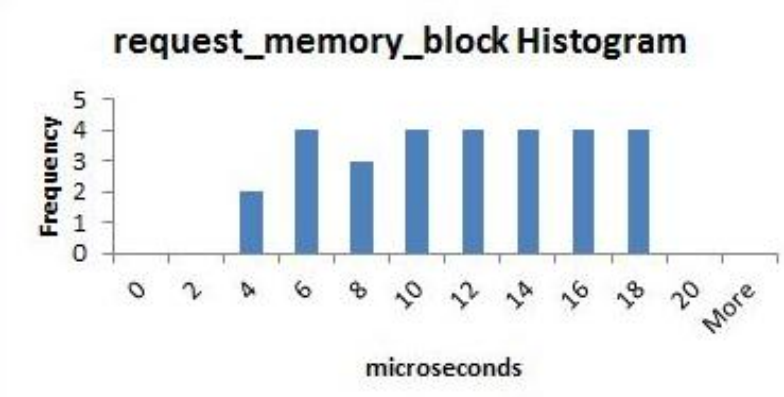

Figure 22: Results for requesting a memory block in Experiment 2

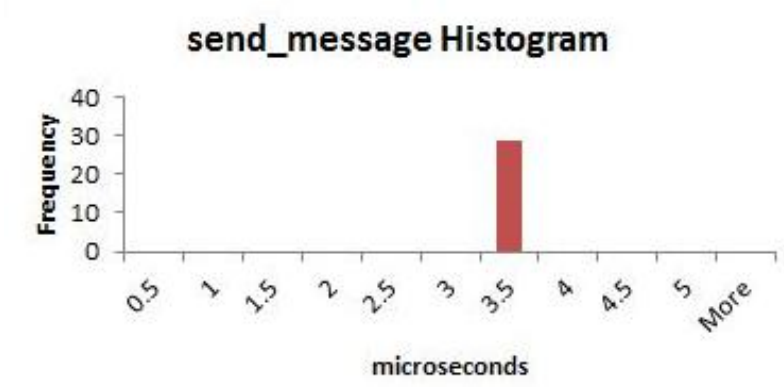

Figure 23: Results for sending a message in Experiment 2

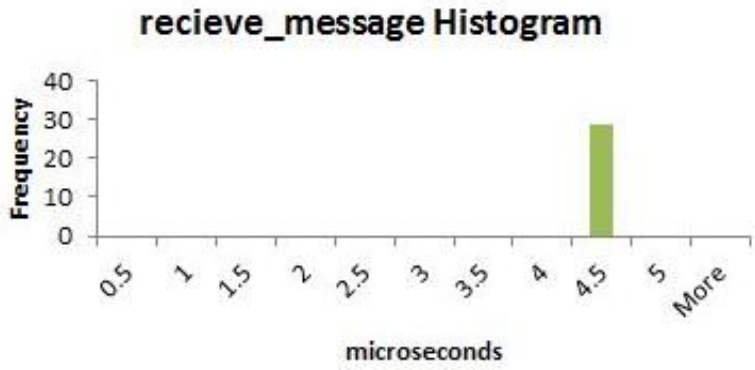

Figure 24: Results for receiving a message in Experiment 2

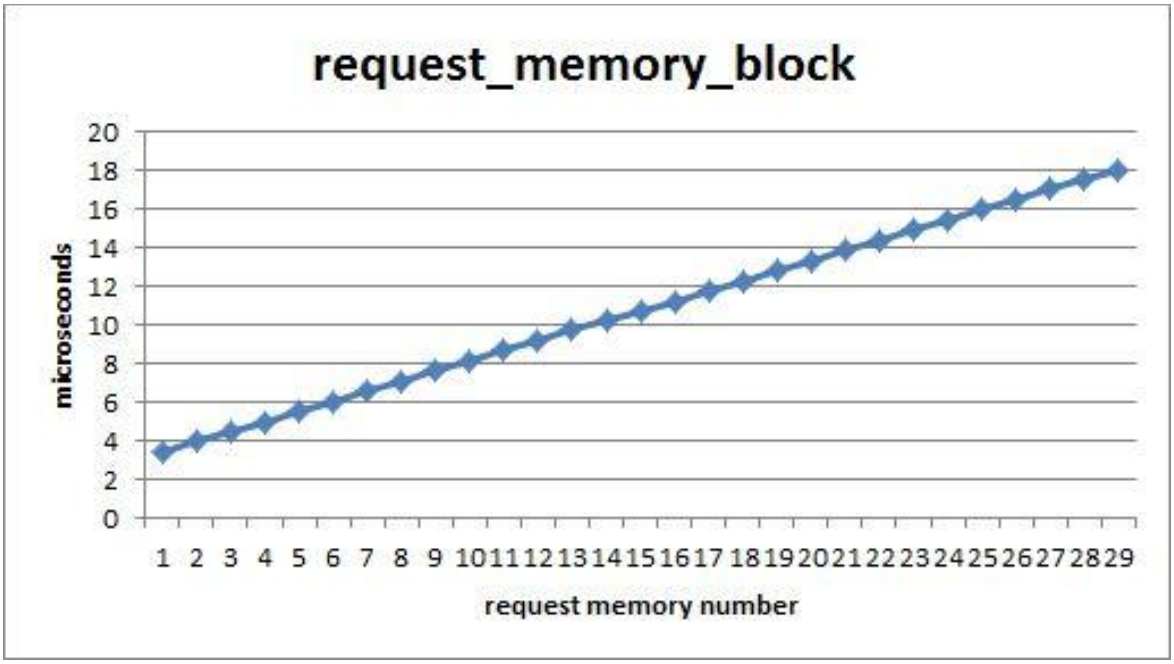

Figure 25: Results for requesting a memory block with variations in the number of blocks requested in Experiment 2

Analysis

In the second experiment, all of the memory blocks are free prior to running the second experiment function. Similar to Experiment 1, the experiment function is running at the highest priority with no interrupts from the user or the timer. The difference between the second experiment and the first experiment lies in the fact that memory is not released. Since memory is not released in the second experiment and due to the implementation of memory, the time taken by each `request_memory_block` takes grows linearly as shown in Figure 25. Requesting memory is implemented by having an array of integers that keep track of memory block usage. If the first block is consumed, the first integer in the array has a value of 1. Otherwise, the value is 0. When requesting the i^{th} memory block, i - 1 memory blocks would have already been consumed. In `request_memory_block`, there is a for loop that returns the first available memory block. Therefore, on the memory request for block i, the for loop iterates i times to find an available memory block. In the second experiment, 1 to 29 memory blocks are requested. For every request, the for loop in the `request_memory_block` call does one more iteration, thereby increasing the time it takes to get a memory block linearly.

3.3 General Analysis

Another point to note is the relation between the times in the first experiment. The functions `request_memory_block` and `send_message` take the same amount of time while `request_message` takes longer than the previous two. `request_memory_block` and `send_message` take the same amount of time because they execute same number of assembly instructions while `request_message` executes more. `request_message` has more instructions to execute because it calls two other functions: `msg_empty` and `dequeue_env_queue`. Branching to these functions, executing and returning from them is the reason for the longer execution time in `request_message`.

4 Conclusions and Thoughts

4.1 Design changes

Part 1

Firstly, we learned that pair programming was essential so everyone can work together and has knowledge of all the other components. Thus when it comes to debugging, everyone can help out and actively fix bugs. Also, we did not reference the specific kernel space and user space functions properly at first and kept getting hard faults when we initially tried to call the kernel methods (i.e. `k_release_memory`) directly from inside the user processes since it was trying to access restricted areas of memory. Another key takeaway was the debugging skills central to succeeding in the other parts of the lab. We learned how to effectively use the debugger to step through memory and use the variable watcher. We gained a solid understanding of the C language and why variables were being overwritten.

Part 2

We had to remember to modify the global stack pointer for the PCBs to accommodate the individual message receives intended for a process. Another tricky part was implementing the `k_release_processor` method and being extremely careful to remember when to enable interrupts and when to disable interrupts. We needed to be careful of not overwriting variables before context switching. Another important takeaway was being careful to deallocate the messages at the right place (i.e. the receiving function should handle the deallocation of the memory blocks associated with sending a message). There were several instances where we were getting errors due to the fact that we were not deallocating memory.

Part 3

After vigorously testing the first two parts, we thought our implementation was nearly perfect. But the main lesson learned here was that it never hurts to keep on testing. Through the stress tests, we found and ironed out some of the kinks in our earlier implementations. This entailed refining cases made in Part 2 to handle extra cases when memory was full.

Part 4

The main lesson learned in this last part was to never leave documentation to the end. Also, with respect to the timer, it was interesting to see how the board actually uses the timers and how timing is implemented on the board.

4.2 Lessons Learned

Mistakes Along the Way

From the first two parts, we realized that we did not allow pre-emption to occur when setting the priority of a process for the case when a lower priority process was set to the highest priority.

Major Stumbling Blocks

The biggest challenge of this project was finding a good memory management strategy and coming up with the structure of the ready queues. Once we decided to have a priority queue implementation for the ready queues and finalized how memory management was going to be implemented (i.e. the maximum number of PCBs and all the associated data for each process in the PCB) the other two parts became easier. A lot of changes had to be made in the k_memory.c file to accommodate the envelope data structure for interprocedural communication in Part 2 and in hindsight, it would have been simpler if we accounted for that in Part 1.

Starting Over

If given the chance to start over, we would have liked to implement the ready priority queue dynamically with more priorities so that each queue for each priority would be dynamic in size. Currently, we limit the size of each queue since we were given exact specifications. Also, since we were given the specifications for the processes, it was possible to hard code the PCB pointers to make the initialization simpler. If we were to account for threading and other advanced features, this operating system would require a lot of modifications.

4.3 Responsibilities

	Tonia	Mike	Angad	Vishal
P1	ready queue, debugging, user processes	memory management, debugging	context switching, Debugging	process priority, user processes
P2	debug hotkeys, debugging, timer i-process	interprocedural communication	wall clock process	Timer i-process, KCD process
P3	process priority, debugging	stress test A, debugging	stress tests B and C, debuggin	process priority, debugging
P4	documentation	documentation	documentation	documentation