RTX Project Report

Craig, Shale
20371384
sakcraig@uwaterloo.ca

Klen, Alex
20372654
ayklen@uwaterloo.ca

Menakuru, Sanjay
20374915
smenakur@uwaterloo.ca

Wei, Jonathan
20376489
j25wei@uwaterloo.ca

April 7, 2013

Abstract

Using the Keil Development Environment (Keil IDE) and LPC1768 boards using the ARM instruction set, an operating system was implemented with basic functionality throughout this project. The operating system offers a stable and succinct API, and also tries to ensure high-performance. It offers a message-passing interface between processes, and allows processes to release the processor to share CPU time. Additionally, we implemented memory management of the on-board memory, allowing processes to dynamically acquire and release memory during our operating-system's runtime. I/O between keyboard input and display output on a UART connection was also implemented, which allows users to interact with processes on the system in near real-time. Overall, this operating system offers a minimal but fast API, and is easily extendible with features necessary to implement a larger and more feature-complete operating system with minimal overhead.

Contents

List of Algorithms

List of Figures

Part I

Introduction

As part of our Operating Systems course, we implemented a micro-kernel for the LPC1768. This board uses the Cortex M-3 processor, which implements the ARM processor architecture. In this report, we discuss our implementation of said micro-kernel in detail.

First, we present detailed descriptions of the algorithms and data structures used inside our kernel. We perform a theoretical analysis of these algorithms and data structures. We provide some basic measurements as to the efficacy of said algorithms and data structures.

Next, we discuss our testing procedure. In particular, we do a detailed examination of our test processes that we ran atop our micro-kernel. We provide pseudo-code and describe the functionality of these processes.

Last, we talk about the lessons we have learned in completing this project.

Part II

Kernel Implementation

Chapter 1

Scheduler

1.1 Description

The scheduler is responsible for dividing processor time amongst all of the system's processes. It handles context switching between processes, deciding which process to schedule next, and preempting or blocking processes when appropriate. Processes can be in one of the following states: new, ready, running, blocked on memory, blocked on message, or unused. 'New' is a state that processes are given after being initialized so that the scheduler can properly context switch to them for the first time. The running state signifies that the process is the only currently running process in the system. Processes which are ready are placed in a priority queue, and the scheduler will select the ready process with the highest priority to run next. Processes which are blocked on memory are put into another priority queue called the blocked-on-memory queue, and when memory is freed, the blocked process with the highest priority is unblocked. Processes that are blocked on a message remain blocked until they are sent a message.

Priorities are implemented using an 8-bit unsigned integer. A lower number represents a higher priority. The top bit is reserved for identifying whether the priority is of kernel or user level. A kernel-level priority has a leading 0 and a user priority has a leading 1, forcing all kernel-level priorities to be higher than user-level ones. The last 7 bits represent all of the priority levels possible in the system (priorities up to $2^7 - 1$ for each priority level). A process can be preempted when it frees memory or sends a message. If the process that it unblocks has a higher priority, then the currently

running process is preempted.

The ready queues are priority queues implemented using heaps. The decision was made to use a priority queue for all ready/blocked processes instead of a queue for each priority level because it allowed the system to support an arbitrary number of priorities without having the release_processor runtime depend on the number of priorities.

Pseudocode for process initialization is listed in Algorithm 1.

Algorithm 1 Process Initialization Pseudocode

```
1:  function K_INITPROCESSES
2:      Initialize process-ready queue
3:      Initialize blocked-on-memory queue
4:      for each process p, index i do
5:          p.pid ← i
6:          p.stack ← ACQUIREMEMORYBLOCK() three times + memory block size   ▷ Use three memory blocks for each process stack and place stack pointer at end of third one
7:          push initial required values onto stack
8:          set p.startLocation to appropriate process function location   ▷ Reserve a debug envelope for each process to be able to display a debug message even if there is no free memory in the system
9:          p.debugEnvelope ← acquireMemoryBlock()
10:         p.state ← new
11:         add p to the process-ready queue
12:     end for
13:
14:     Initialize CRT proc
15: end function
```

See Algorithm 2 for a pseudocode implementation for the kernel-level function k_releaseProcessor. This function is called any time the current process should be preempted and a new process should be scheduled. The function takes a parameter that specifies what the reason is for the context switch. The release reason can be one of the following: a voluntary yield, a process priority change, memory was freed, the system is out of memory, a message was sent, or a message is being received. The scheduling behavior can be different for some of these reasons.

Algorithm 2 Release Processor Pseudocode

```
1: function K_RELEASEPROCESSOR(releaseReason)      ▷ First we process the changed states of a few interrupt-driven systems here to avoid ISRs modifying kernel state while they are interrupting kernel methods
2:     K_PROCESSUARTOUTPUT()
3:     K_PROCESSUARTINPUT()
4:     K_PROCESSDELAYEDMESSAGES()
                    ▷ These variables are set depending on the release reason
5:     define targetState, ▷ The state to give the currently running process
6:          srcQueue,          ▷ The queue to schedule the next process from
7:          dstQueue           ▷ The queue to insert the current process into
8:     if releaseReason = out of memory then
9:         srcQueue ← process-ready queue
10:        dstQueue ← blocked-on-memory queue
11:        targetState ← blocked on memory
12:    else if releaseReason = message was received then
13:        if the current process's mail queue is empty then
14:            srcQueue ← process-ready queue
15:            dstQueue ← NULL                    ▷ Don't place in any queue
16:            targetState ← blocked on message
17:        else
18:            srcQueue ← process-ready queue
19:            dstQueue ← process-ready queue
20:            targetState ← ready
21:        end if
22:    else
23:        srcQueue ← process-ready queue
24:        dstQueue ← process-ready queue
25:        targetState ← ready
26:        if currentProcess is the null process then          ▷ Keep the null process out of the ready queue - handle it separately
27:            dstQueue ← NULL
28:        end if
29:    end if
30:    if srcQueue ≠ NULL and srcQueue.size > 0 then
31:        nextProc ← SRCQUEUE.POP()
32:    else
33:        nextProc ← the null process
34:    end if
```

```
35:     if currentProcess ≠ NULL then                    ▷ Save old process info
36:         currentProcess.stack ← getStackPointer()
37:         currentProcess.state ← targetState
38:         if dstQueue ≠ NULL then
39:             dstQueue.add(currentProcess)
40:         end if
41:     end if
42:     oldState ← nextProc.state
43:     nextProc.state ← running
44:     currentProcess ← nextProc
45:     setStackPointer(nextProc.stack)
46:     if oldState = new then
47:         __rte()                  ▷ If first time, pop the exception stack frame
48:     end if
49: end function
```

1.2 Running Time Analysis

The runtime for k_initProcesses is $O(n \log n)$, where n is the number of processes in the system. This is because each processes is inserted into the process-ready queue, which is implemented as a heap and therefore taking $O(\log n)$ to insert an element. This function could be trivially modified to achieve a runtime of $O(n)$ time by putting all of the heap elements in an array first, and then heapifying it afterwards. Since there is a small, fixed number of processes in the system, this small optimization wasn't completed since this function only runs once at system startup.

The runtime for k_releaseProcessor, with the assumption that the UART and delayed message processing at the beginning will typically take $O(1)$ time, takes $O(\log n)$ time, where n is the number of processes in the system. It is bounded by removing and inserting from the source and destination priority queues (heaps). This provides good performance, and we can have an arbitrary number of priorities in the system because a priority queue was used instead of a queue per priority level.

Chapter 2

Memory Allocator

2.1 Description

The memory allocator was designed with two orthogonal layers. The first is called the 'block layer', and is responsible for allocating and deallocating blocks at a low level. The second is called the 'metadata layer'; this layer builds upon the work done by the block layer, and adds the ability to store metadata about blocks. We will describe both of these layers independently.

2.1.1 Block Layer

The block layer has a conceptually simple role. It is responsible for managing a pool of contiguous memory. Given a start memory address, an end memory address, and a block size, the block layer must provide two pieces of functionality. It must support allocating a block from the pool, and it must support freeing a block back into the pool. One design constraint is that it must prefer reusing a previously allocated block that has been freed, rather than handing out a block that has never been allocated before. The reason for this constraint will become clear when discussing the metadata layer.

See Algorithm 3 for a pseudocode implementation of this layer. Note that this layer returns an error code when it runs out of memory. Further, this layer does some rudimentary sanity checking but doesn't handle some obvious error conditions. For example, this layer allows blocks to be inserted into the free list twice (assuming a trivial linked list implementation). This is not an error; it is the responsibility of the metadata layer to never allow this to happen. Finally, it is important to point out that this layer automatically

zeroes memory. This is slightly detrimental to performance, but allows user processes to use binary comparisons between structs without caring about garbage in the struct's padding. We deemed this convenience worth the slight performance overhead.

2.1.2 Metadata Layer

The metadata layer builds atop the block layer, and adds the ability for the allocator to store metadata about blocks. Since it is built upon the block layer, it must fit its metadata storage in blocks. To avoid a costly setup phase, the metadata layer uses a lazy, arena-based storage scheme. The general version of the metadata layer supports an arbitrary amount of metadata per block ranging from some non-zero amount of metadata to half the block size worth of metadata.

For the sake of description, let us call the number of metadata fields that fit inside a block k. That is to say, one block can hold the metadata for k blocks. The metadata layer discriminates between two types of blocks, data blocks and arena header blocks. The metadata layer implements a function that takes a block and returns its associated arena header block. For our purposes, this function acts as the identity function when passed an arena header block. These definitions lead to the notion that an arena header block is responsible for storing the metadata of itself, as well as the following $k-1$ blocks. The metadata layer is responsible for serving the user-facing API.

See Algorithm 4 for a pseudocode implementation of this layer. Note that this layer can also access the variables defined by the block layer (they can be found in Algorithm 3). Also note that we have elided the code of several trivial helper functions. For example, we did not specify the body of the 'get_header' function. Since the implementation of this function was some simple modular arithmetic, we did not feel it was worth wasting the reader's time with such minutiae.

While the metadata layer is parameterized on k, for our actual implementation, we chose k to be equal to our block size; this implies the metadata for each process fits inside 1 byte. Indeed, we stored the owner pid of each block as the only metadata, and the bitwidth of the pid type in our system was 8.

Algorithm 3 Block layer pseudocode

```
1: blockSize ← 2^7                                  ▷ Blocksize in bytes (configurable)
2: startAddr ← 0                                    ▷ Start address of pool (inclusive)
3: endAddr ← 2^16                                   ▷ End address of pool (exclusive)
4: nextAddr ← startAddr                             ▷ Next available address in pool
5: freeList ← {}                                    ▷ List of freed blocks
6:
7: function ALLOC_BLOCK()
8:     if |freeList| > 0 then                       ▷ Check free-list
9:         blk ← freeList[0]
10:        freeList = freeList − {blk}
11:        ZERO(blk)
12:        return blk
13:    end if
14:
15:    if nextAddr + blockSize >= endAddr then      ▷ Out-of-memory
16:        return −1
17:    end if
18:
19:    blk ← nextAddr
20:    nextAddr ← nextAddr + blockSize
21:    ZERO(blk)
22:    return blk
23: end function
24:
25: function FREE_BLOCK(blk)
26:    if blk < startAddr ∨ nextAddr <= blk ∨ endAddr <= blk then
27:        return                                   ▷ Outside valid range
28:    end if
29:
30:    if blk − startAddr ≢ 0 mod blockSize then
31:        return                                   ▷ Unaligned
32:    end if
33:
34:    freeList = freeList ∪ {blk}
35: end function
```

Algorithm 4 Metadata layer pseudocode

```
1: k ← 2^7                                          ▷ Blocks per arena (configurable)
2: arenaSize ← k * blockSize                        ▷ Arena size (in bytes)
3: function REQUEST_MEMORY_BLOCK(pid)
4:     header ← −1
5:     blk ← −1
6:
7:     blk ← ALLOC_BLOCK()
8:     if blk = −1 then
9:         return −1
10:    end if
11:
12:    header ← GET_HEADER(blk)
13:    if blk = header then
14:        blk ← ALLOC_BLOCK()
15:    end if
16:
17:    if blk = −1 then
18:        return −1
19:    end if
20:
21:    SET_OWNER(header, blk, pid)
22:    return blk
23: end function
24:
25: function FREE_MEMORY_BLOCK(blk, pid)
26:    header ← GET_HEADER(blk)
27:    if header = −1 then
28:        return −1
29:    end if
30:
31:    if ¬IS_OWNER(header, blk, pid) then
32:        return −1
33:    end if
34:
35:    SET_OWNER(header, blk, 0)
36:    FREE_BLOCK(blk)
37: end function
```

2.2 Theoretical Analysis

Both layers were carefully designed to be $O(1)$ for all operations. This should be trivial by inspection, as all the functions do a tiny amount of arithmetic, and considering we used a linked-list as the implementation of the free list.

A slightly more interesting analysis is the space overhead of our metadata scheme. The metadata layer uses $\frac{k-1}{k}$ blocks for storing user data; this yields a storage overhead of $\frac{1}{k}$. For our implementation, we picked $k = 2^7$, yielding an overhead of 0.78%, or a utilization rate of 99.22%.

2.3 Measurements

We found that on average the memory allocator would take $787.2ns$ to return a fully zeroed memory block. This was determined by taking the average of 5 trials, where each trial did a few hundred calls to the allocation routine. The raw data from the trials can be found in Appendix A.

Chapter 3

Message Passing

3.1 Description

This micro-kernel supports three user API functions for message passing: send_message, receive_message, and send_delayed_message. Each function requires the user process to use a special data structure called Envelope, which is available in a provided header file. The following is a table of the fields available on the Envelope data structure.

Type	Name
uint32_t	messageType
char[MESSAGEDATA_SIZE_BYTES]	messageData
ProcId	srcPid
ProcId	dstPid
uint32_t	sendTime
Envelope*	next

To obtain an envelope, a user process must acquire a memory block and then cast it to the Envelope type. When sending a message, the user should set the messageType and messageData fields to values they wish to send. The srcPid and dstPid fields may be filled out, but the kernel will overwrite them to their correct values. The rest of the fields are reserved for kernel use and they will be overwritten after the user sends the message. After sending an envelope, the user process no longer owns that block of memory and they should clear all references to it.

The kernel maintains a priority queue of delayed messages (where priority is the time it should be delivered) for the system and a message queue for each process. A function called k_processDelayedMessages (see Algorithm 8)

is called in release_processor, and it delivers any delayed messages that have waited for their specified delay. The message queue for each process is where messages are stored when they're delivered to a process that is not blocked on message.

Sending a message may cause the sending process to be preempted. This will occur if the receiving process is blocked on receiving a message and it has a higher priority. Calling receive_message will cause the process to block on message, unless it has messages in its message queue. Sending a delayed message, however, will never cause the sender to block. See Algorithm 5 for a pseudocode implementation of send_message, Algorithm 6 for receive_message, and 7 for send_delayed_message.

Algorithm 5 Send Message

```
1:  function K_SENDMESSAGE(envelope, srcPid, dstPid)
2:      Initialize reserved kernel fields of envelope to a zeroed state
3:      envelope.srcPid = srcPid
4:      envelope.dstPid = dstPid
5:      if srcPid or dstPid is invalid then
6:          return error code
7:      end if
8:      dstPCB ← PCB of process with pid dstPid
9:      srcPCB ← PCB of process with pid srcPid
10:     if srcPid owns memory block envelope then
11:         Set owner of envelope to dstPid
12:     else
13:         return error code
14:     end if
15:     Add envelope to dstPCB's message queue
16:     if dstPCB.state = blocked on message then          ▷ Unblock receiver
17:         dstPCB.state ← ready
18:         Add dstPCB to process-ready queue
19:         if dstPCB.priority < currentProcess.priority then
20:             K_RELEASEPROCESSOR() ▷ Preempt if unblocked process has
    higher priority
21:         end if
22:     end if
23: end function
```

Algorithm 6 Receive Message

```
1: function K_RECEIVEMESSAGE
2:     message ← CURRENTPROC.MESSAGEQUEUE.TOP()
3:     while message = NULL do                    ▷ Loop until message exists
4:         K_RELEASEPROCESSOR(message being received)    ▷ Blocks this process
5:         message ← CURRENTPROC.MESSAGEQUEUE.TOP()
6:     end while
7:     CURRENTPROC.MESSAGEQUEUE.POP()
8:     message.next ← NULL    ▷ Clear next pointer so user doesn't have next message
9:     Set owner of message to dstPid
10:    return message
11: end function
```

Algorithm 7 Send Delayed Message

```
1: function K_SENDDELAYEDMESSAGE(envelope, srcPid, dstPid, delay)
2:     Initialize reserved kernel fields of envelope to a zeroed state
3:     if srcPid or dstPid is invalid then
4:         return error code
5:     end if
6:     if srcPid owns memory block envelope then
7:         Set owner of envelope to kernel
8:     else
9:         return error code
10:    end if
11:    envelope.sendTime ← K_GETTIME() + delay    ▷ Set the time the envelope will be delivered
12:    envelope.srcPid ← srcPid
13:    envelope.dstPid ← dstPid
14:    Add envelope to kernel delayed message queue
15: end function
```

Algorithm 8 Process Delayed Messages

```
1: function K_PROCESSDELAYEDMESSAGES
2:     messageQueue ← kernel delayed message queue
3:     currentTime ← K_GETTIME()
4:     if messageQueue is empty then
5:         return
6:     end if
7:     loop
8:         message ← MESSAGEQUEUE.TOP()
9:         if message = NULL or message.sendTime > currentTime then
10:            return     ▷ Loop until there are no more messages to deliver
11:        end if
12:        MESSAGEQUEUE.POP()
13:        Set owner of message to message.srcPid    ▷ Set to srcPid so that
   we can use k_sendMessage
14:        K_SENDMESSAGE(message, message.srcPid, message.dstPid)
15:    end loop
16: end function
```

3.2 Running Time Analysis

The runtime of k_sendMessage (assuming process is not preempted) is $O(1)$ because all operations take constant time. Adding a message to the receiving process's message queue takes constant time because it is simply a linked- list. The runtime of k_receiveMessage (assuming there is a message to receive) is also $O(1)$ because it only looks at the message at the head of the queue.

The runtime of k_sendDelayedMessage is bounded by the time it takes to add a message to the kernel delayed message queue. This is a priority queue implemented as a heap, so it takes $O(\log n)$ time for insertion. This is a good runtime for sending a delayed message, considering that delayed messages are not necessarily delivered in the order they are sent. A heap allows efficient retrieval of the earliest message without having to sort all of the messages.

The runtime of k_processDelayedMessages depends upon how many delayed messages need to be delivered at the same time. Each message that needs to be delivered is removed from the kernel delayed message queue, which is implemented with a heap. This makes the runtime $(m \log n)$, where

m is the number of messages that need to be sent immediately, and n is the number of delayed messages currently waiting in the system. This runtime is reasonable because it is very unlikely that a large number of messages need to be delivered at exactly the same time. It is possible to improve this runtime by using a skip list. This would allow binary searching for the last message in the queue that needs to be sent, and then linearly removing and delivering all messages from the head of the list to the found message. This optimization would result in worst-case $O(\log n + m)$ time. This optimization wasn't done because the small likelihood of m being large didn't justify a much more complex implementation.

3.3 Measurements

The measured average runtime of k_sendMessage was $1.09\mu s$, and the average runtime of k_receiveMessage was $0.722\ \mu s$. This is reasonable because both of these operations take $O(1)$. The measurements did not change much with the different trials, except in the case of trial 4, where the Memory Muncher process did not run. In this case, k_receiveMessage decreased its average runtime to $0.667\ \mu s$. This is because when profiling average runtimes, the amount of time the processes is blocked on receiving a message is included in the average runtime of k_receiveMessage. This agrees with the observation that when the Memory Muncher process was not allowed to run, the average runtime of k_releaseProcessor improved because there was enough memory to send messages more often than when the Memory Muncher was running. In the other cases the runtimes did not change significantly, which corresponds with a $O(1)$ runtime of these functions. The raw data can be viewed in Appendix A.

Chapter 4

I/O

I/O interactions were designed to provide a simple user-facing interface while guaranteeing a high level of functionality and speed in low-memory situations. Functionality was split between separate input and output processes, each of which were responsible for providing their respective functionality. By separating functionality, we were able to take advantage of the benefits of microkernel architecture instead of monolithic architecture. In the context of our OS, both input and output tasks consisted of interacting solely with UART (Universal Asynchronous Receiver / Transmitter) connections to the boards used in this lab. We will describe both input and output independently in their respective sections.

4.1 Input

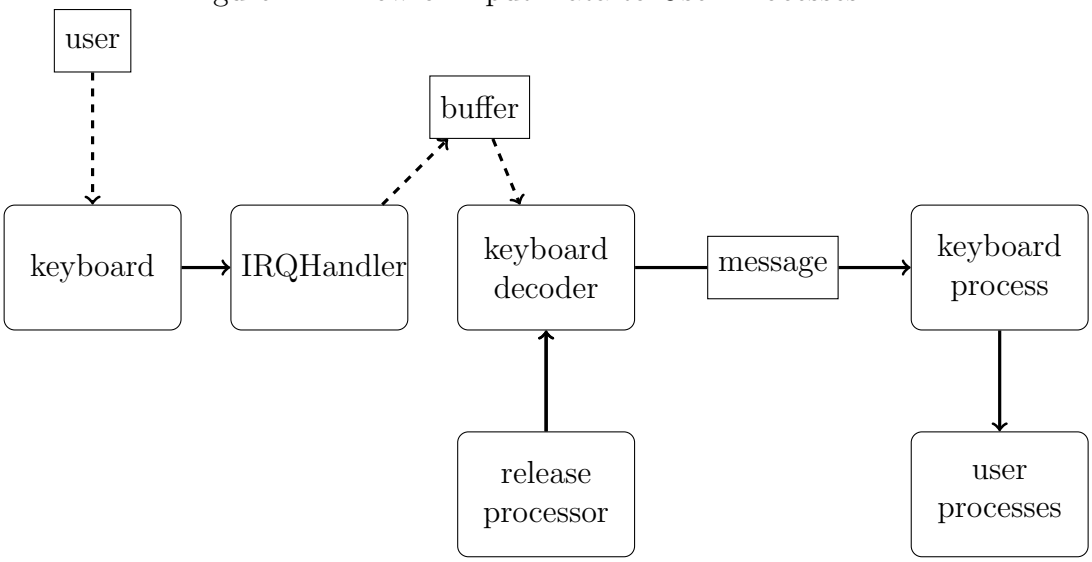
In this assignment, the primary input device is keyboard input. Keyboard input triggers the UART0_IRQHandler assembly interrupt. This passes the input to the c_UART0_IRQHandler method that puts the events in a buffer. Whenever release_processor is called, messages in the buffer are sent to the keyboard decoder method to decode the new events in the buffer. In the case that input is a hot-key, it immediately executes the relevant execution path. In the case that input is a complete command, a process that has registered for that command will receive a message containing the command. If no processes have been registered, the message is freed.

Figure 4.1 illustrates this process in a visually apparent manner, and Algorithm 9 expresses this flow algorithmically.

Algorithm 9 Uart Input Pseudocode

```
1: function __asm UART0_IRQHandler
2:     ...
3:     BL c_UART0_IRQHandler
4:     ...
5: end function
6: function c_UART0_IRQHandler
7:     ...
8:     mark message as seen
9:     uart_receive_char_isr(newChar);
10:    ...
11: end function
12: function uart_receive_char_isr(newChar)
13:    if (writeIndex + 1) % bufferSize == readIndex then
14:        bufferOverflow = true
15:        return
16:    end if
17:    buffer[writeIndex] = newChar;
18:    writeIndex = (writeIndex + 1)
19: end function
20: function releaseProcessor
21:    ...
22:    processUartInput();
23:    ...
24: end function
25: function processUartInput
26:    ...
27:    while buffer has more do
28:        char newChar = buffer[readIndex];
29:        readIndex = (readIndex + 1)
30:        if newChar is a newline then
31:            send a message to keyboard that a newline has occurred
32:            continue;
33:        else if newChar is SHOW_DEBUG_PROCESSES then
34:            print debug information, send it to the CRT proc
35:            continue;
36:        end if
37:    end while
38:    ...
39: end function
```

Figure 4.1: Flow of Input Data to User Processes



4.2 Output

4.2.1 Basic Output

At a basic level, user processes send messages to the CRT process, which buffers it into a stream. The method uart_send_char_isr consumes the buffer and sends it through UART to the PuTTY on the PC. Additionally, the same UART0_IRQHandler used in Section 4.1 triggers uart_send_char_isr when the uart has notified us it is able to send more bytes. Figure 4.2.1 is a pictorial representation of this procedure. In the next subsection 4.2.2, we will explain how we exceeded the specifications which caused our implementation to be more complicated than this.

4.2.2 Output Enhancements

We decided to implement a larger feature set than required in the application specification in order to make our PuTTY interface feel and act more like a command line found in a traditional computer. Our features include a text input bar that stays at the bottom while users type, coloured text, backspace support, and a few other enhancements.

To achieve this, we've moved the buffered keyboard input into an envelope

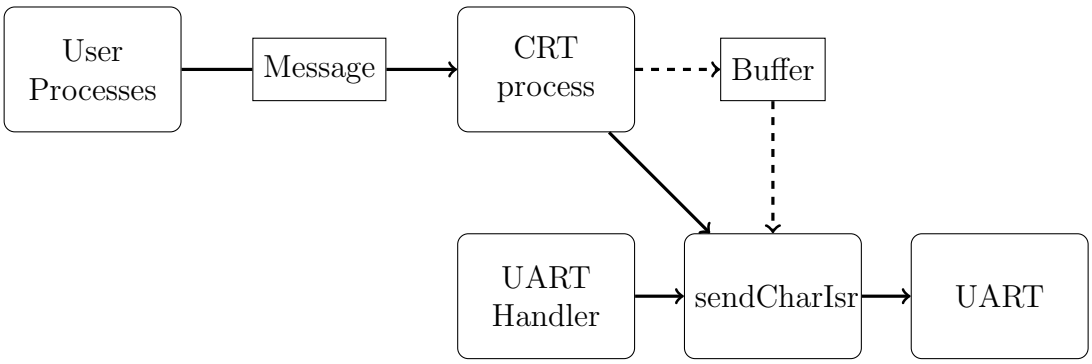


Figure 4.2: Flow of Output Data to Uart

belonging to a CRTData object. Whenever the buffer is cleared by a user pressing the enter key, we move the cursor upwards, and write the content of the buffer in the cleared space. Similarly, if a process requests to print, we move the cursor upwards, and write the content of the message into the cleared space. By shifting the cursor's position, we are able to maintain the currently buffered user text at the bottom of the screen, allowing the user to see their text as they enter it. Implementing deletion only required parsing a special character in a similar way to a hot-key. Inside the pseudocode algorithm (10) listed for this subsection, the CRT process loads messages from the buffer that CRT_advance loads with instructions to move characters, print content from processes, and show current input buffer status (i.e. what the user has typed).

Algorithm 10 Uart Output Pseudocode

```
1: function CRT_PROC
2:     loop
3:         nextMessage = receive_message(NULL)
4:         pushEnvelope(nextMessage)
5:         while ( docrt_hasData && ((writeIndex + 1)
6:             outputBuffer[writeIndex] = crt_getData()
7:             writeIndex = (writeIndex + 1)
8:         end while
9:         uart_send_char_isr(outputBuffer)
10:    end loop
11: end function
12: function CRT_ADVANCE
13:    Print the text bar. Remove characters that are not matching.
14:    Process output
15:    while hasQueuedEnvelopes do
16:        Print next queued envelope until a null byte occurrs.
17:    end while
18:    move the cursor to the user position
19: end function
```

Chapter 5

Misc

5.1 Bridge Layer

5.1.1 Description

We added a bridge layer in order to allow our user processes to access the kernel-level functions safely. These bridging functions allow us to pass global data structures into the kernel functions, as well as do additional validation or work that only needs to be done when a user process calls the function as opposed to when it is called from within kernel space.

Examples of this where additional functionality was added are listed below.

acquireMemoryBlock We want this to be a blocking call when the system is out of memory, so we loop in the bridge function. We alternate between calling the k_acquireMemoryBlock function (which will return an error code if there is no memory left) and calling k_releaseProcessor until the user process successfully obtains a block.

releaseMemoryBlock If there are processes that are blocked on memory, then the highest priority process should receive the released memory block. The highest priority blocked process is placed back into the ready queue, and we preempt the current process if the blocked process is of higher priority.

sendMessage The sending process is preempted if the message sent unblocks a process of higher priority. This is done here so we guarantee

that we only call k_releaseProcessor if it was a user process of lower priority that made the sendMessage call.

receiveMessage The output parameter is set here, if available.

5.2 Get/Set Priority

5.2.1 Description

These functions return and set priorities of processes, respectively. Validation is done to ensure that only proper process IDs and priorities are passed in. k_setProcessPriority will preempt the currently running process if it is of lower priority.

k_get_process_priority simply verifies if the passed process ID is valid and returns the priority of that process, or -1 otherwise. As k_set_process_priority is more complicated, its pseudocode has been included under Algorithm 11.

There are two helper functions used: setMaskPriority and pqChangedPriority. setMaskPriority sets the new priority to retain the old priority's leading bits, while setting its last 7 bits to the desired priority. pqChangedPriority modifies the given process' priority in the queue and rebalances the backing heap to keep it in sorted order.

Overall, this function takes at most $O(\log(n))$ time where n is the number of processes in the queue that the process whose priority is being changed resides in. All other operations in the function are constant time.

Algorithm 11 Kernel set process priority pseudocode

```
1: function K_SETPROCESSPRIORITY(pid, priority)
2:     modifiedProcess ← K_GETPCB(pid)
3:                                   ▷ Returns NULL if pid is invalid
4:     if modifiedProcess = NULL || modifiedProcess = nullProcess then
5:         return EINVAL
6:     end if
7:
8:     SETMASKPRIORITY(modifiedProcess, oldPriority, newPriority) ▷ Set the priority to the masked new priority
9:
10:    PQCHANGEDPRIORITY(readyQueue, memoryQueue, modifiedProcess) ▷ Change priority state in the queues, safe to call if the process isn't in the queue
11:
12:    if we improved our own priority or top process has same or worse priority then
13:        return SUCCESS                    ▷ Don't preempt ourselves
14:    end if
15:
16:    K_RELEASEPROCESSOR(CHANGED_PRIORITY)
17:
18:    return SUCCESS
19: end function
```

Part III

User-level Processes

Chapter 6

Set Priority Command Process

6.1 Description

This process is responsible for allowing the user to change process priorities through terminal commands. It is registered to the %C command, and will attempt to set a new priority level to the given process ID. Validation is done on the input parameters to ensure that they are actually valid inputs before calling set_process_priority(pid, priority). We check to ensure that we have a space between %C, the pid, and the priority. We also ensure that the pid is valid (within our allowed range of integers) and that the priority is valid (between 0 and 255 inclusive). The set_process_priority process itself satisfies the requirements that the change be immediate and that the target process could be located in the ready or blocked resource queue.

See Algorithm 12 for a pseudocode implementation of this user process. Note that this process, among others, makes use of a custom implementation of a read_uint32 helper function that we created. The pseudocode for that function is included in Algorithm 13.

As a user process, we require memory in order to send a message to it (as the user), so we will be unable to send commands to it (as well as other user processes) in a situation where we are out of memory. Therefore, we are unable to use this process to stop a misbehaving process that is holding and not releasing memory if we are already out of memory.

Algorithm 12 Set priority command process pseudocode

```
1: function setPriorityProcess
2:     envelope ← request_memory_block
3:     init_registration_envelope(envelope, c)
4:     send_message(KEYBOARD_PID, envelope)
5:
6:     loop
7:         envelope ← receive_message(NULL)
8:
9:         if envelope → srcPid ≠ KEYBOARD_PID then
10:            release_memory_block(envelope)
11:            continue
12:        end if
13:
14:        if envelope message isn't prefixed with %C then
15:            release_memory_block(envelope)
16:            continue
17:        end if
18:
19:        status ← processSetMessage(envelope → message)    ▷ This will also call set_process_priority if the contents pass validation
20:        if status = EINVAL then    ▷ Print error message upon failure to validate
21:            printSetErrorMessage(envelope)
22:        else
23:            release_memory_block(envelope)
24:        end if
25:    end loop
26: end function
```

Algorithm 13 Read uint32 from char array pseudocode

```
1:  function READ_UINT32(buf, out)                    ▷ buf is a char pointer
2:      number ← 0                                    ▷ The integer result
3:      read ← 0                                      ▷ Number of characters read
4:
5:      if buf = NULL then
6:          return 0                                  ▷ Return if there's nothing to read
7:      end if
8:
9:      while read < LEN(buf) do
10:         if ¬IS_NUMERIC(buf[0]) then
11:             break
12:         end if
13:
14:         number ← number × 10
15:         number ← number + TO_INT(buf[0])
16:         read ← read + 1
17:         buf ← buf + 1
18:     end while
19:
20:     if out ≠ NULL then
21:         out[0] ← number
22:     end if
23:
24:     return read
25: end function
```

Chapter 7

24-Hour Wall Clock Display Process

7.1 Description

This process is responsible for acting as a digital clock, printing the time at one second intervals. The user can start the clock at 00:00:00 with the command %WR, can stop output with %WT, and can set an arbitrary time using %WS HH:MM:SS. The process will reject any messages passed to it that either don't begin with the registered %W command, or are malformed.

For the %WS HH:MM:SS command, we check that there is a space in between %WS and HH:MM:SS. As well, we make sure that there are colons in between HH, MM, and SS. We also ensure that HH is between 00 and 23 inclusive, and MM and SS are both between 00 and 59 inclusive.

See Algorithm 14 for a pseudocode implementation of this user process. Note that this process, among others, makes use of a few helper functions. It uses read_uint32, write_uint32, as well as write_string. See Algorithm 15 for the pseudocode of write_uint32, and Algorithm 16 for the pseudocode of write_string. In addition, a user API call get_time is also used. This simply obtains the current system counter, which is incremented every millisecond by the timer ISR.

Algorithm 14 Wall clock process pseudocode

```
1: receivedEnvelope ← NULL                          ▷ Envelope for message passing
2: selfEnvelope ← REQUEST_MEMORY_BLOCK  ▷ Cached envelope for self delayed messages
3: cmdType ← TERMINATE                                          ▷ Action to take
4: currentTime ← 0                                          ▷ Cached current time
5: offset ← 0                                          ▷ Offset for calculating time
6: isRunning ← 0                                  ▷ Boolean value for running state
7:
8: function parseClockMessage
9:     status ← 0
10:    envelope ← receivedEnvelope
11:
12:    if envelope → srcPid = CLOCK_PID then   ▷ Received wake up, don't release memory
13:        if isRunning then                          ▷ If not terminated, print time
14:            cmdType ← PRINT_TIME
15:        end if
16:        return
17:    else if envelope → srcPid ≠ KEYBOARD_PID then      ▷ Ignore unwanted message
18:        RELEASE_MEMORY_BLOCK(envelope)
19:        return
20:    end if
21:
22:    if envelope → message[2] ='R' then
23:        cmdType ← RESET_TIME
24:    else if envelope → message[2] ='T' then
25:        cmdType ← TERMINATE
26:    else if envelope → message[2] ='S' then
27:        cmdType ← SET_TIME
28:    else
29:        return
30:    end if
31:
32:    if cmdType = RESET_TIME then
33:        offset ← −1 × currentTime
```

```
34:         if isRunning = 0 then                          ▷ Start clock if necessary
35:             cmdType ← PRINT_TIME
36:         end if
37:     else if cmdType = SET_TIME then
38:         status, offset ← parseTime(envelope)           ▷ Sets offset if validation is passed
39:         if status = SUCCESS&&isRunning = 0 then        ▷ Start clock if necessary
40:             cmdType ← PRINT_TIME
41:         end if
42:     else if cmdType = TERMINATE then
43:         isRunning ← 0
44:     end if
45:
46:     release_memory_block(envelope)
47: end function
48:
49: function clockProcess
50:     envelope ← request_memory_block
51:     init_registration_envelope(envelope, w)
52:     send_message(KEYBOARD_PID, envelope)
53:
54:     loop
55:         receivedEnvelope ← receive_message
56:         currentTime ← get_time
57:
58:         parseClockMessage
59:
60:         if cmdType = PRINT_TIME then                   ▷ Print the time in HH:MM:SS using the calculated offset, and wake self in 1 second
61:             printTime(currentTime, offset)
62:             isRunning ← 1
63:             delayed_send(CLOCK_PID, selfEnvelope, 1000)
64:         end if
65:     end loop
66: end function
```

Algorithm 15 Write uint32 to char array pseudocode

```
1: function WRITE_UINT32(buf, number, minDigits)        ▷ buf is a char pointer
2:     tempNumber ← number
3:     numDigits ← 0                                    ▷ Number of digits available
4:
5:     while tempNumber > 0 do
6:         numDigits ← numDigits + 1
7:         tempNumber ← tempNumber/10
8:     end while
9:
10:    if minDigits < 1 then
11:        minDigits ← 1
12:    end if
13:
14:    if numDigits < minDigits then
15:        numDigits ← minDigits
16:    end if
17:
18:    if numDigits > LEN(buf) then
19:        numDigits ← LEN(buf)
20:    end if
21:
22:    tempNumber ← numDigits
23:
24:    while numDigits > 0 do
25:        if buf ≠ NULL then
26:            buf[numDigits − 1] ← TO_CHAR(number%10)
27:        end if
28:
29:        number ← number/10
30:        numDigits ← numDigits − 1
31:    end while
32:
33:    return tempNumber                                ▷ Returns number of digits read
34: end function
```

Algorithm 16 Write string to char array pseudocode

```
1: function WRITE_STRING(buf, message)                ▷ buf is a char pointer
2:     written ← 0                                    ▷ Number of characters written
3:     bufLen ← LEN(buf)
4:
5:     if msg = NULL then
6:         return 0                                   ▷ Return if there's nothing to write
7:     end if
8:
9:     while msg[0] ≠ NULL_CHAR&&bufLen > 0 do
10:        if buf ≠ NULL then
11:            buf[0] ← msg[0]
12:            buf ← buf + 1
13:        end if
14:
15:        msg ← msg + 1
16:        bufLen ← bufLen − 1
17:        written ← written + 1
18:    end while
19:
20:    return written
21: end function
```

Chapter 8

Stress Processes A, B, and C

8.1 Description

These processes were used to stress test the system. Process A was registered under command %Z. Upon starting, it would continuously obtain a memory block when possible and send it off as a message to process B, containing the value of a counter that it incremented every time it went around the loop. Process B would redirect any messages to process C. Process C had its own extra message queue (on top of the one that we already had for each process). If there was something in its extra queue, we would dequeue from that first, before processing any new messages from its regular queue. Then, if we received a message from process A, we would check if the counter that was included is divisible by 20. If so, we send a message to the CRT process to print a message to the screen, then sleep for 10 seconds. During this time, messages from process A (through process B) would continue to accumulate, eventually running out of memory when there is none left to use to send messages. After waking up, C would process all the messages in the queue, releasing their memory, and the cycle would begin again.

This set of operations would quickly deprive our system of memory. It was designed to see how the system would respond in a no-memory situation. When we ran this during testing, we would run into deadlocks in the situation when the priority of process C was less than or equal to process A, as process A only requested memory blocks, while process C was the one that released memory. The scenario would be when all the memory is locked up inside process C's local message queue, and it blocks during the call to sleep (which

requires an envelope in order to send itself a message). To this end, we modified the sleep function such that this scenario wouldn't occur. We would still be out of memory, but we wouldn't deadlock - we would simply undergo the 10 second cycle of flushing all memory, then being out of memory again until the next time process C wakes up.

To do this, we preallocated envelopes for processes that needed to use the sleep function (process C as well as some of our own user-level test processes). That way, sleep doesn't require a call to request_memory_block, so it's a non-blocking call (apart from the sleep itself).

The pseudocode for processes A, B, and C are not included as they were implemented off of the given pseudocode in the project description. However, an additional envelope was preallocated for process C so that its sleep call wouldn't block. See Algorithm 17 for a pseudocode implementation of the sleep function, which was modified from the given pseudocode in order to include our additional functionality. We also make a user call to pid(), which is a function returning the process ID of the currently running process.

Algorithm 17 Sleep function pseudocode

```
1: function SLEEP(ms, listEnv, sleepEnv) ▷ listEnv is a double pointer to Envelope to be used as an additional message queue, while sleepEnv is a pointer to a preallocated Envelope to be used for sending the delayed message
2:     currentTime ← GET_TIME
3:     targetTime ← currentTime + ms
4:     sleepEnv → messageType ← MT_SLEEP
5:     DELAYED_SEND(PID, sleepEnv, targetTime − currentTime)
6:
7:     if listEnv ≠ NULL then ▷ Advance to back of passed queue if available
8:         while listEnv[0] ≠ NULL do
9:             listEnv ← ADDROF(listEnv[0] → next)
10:        end while
11:    end if
12:
13:    loop
14:        env ← RECEIVE_MESSAGE
15:
16:        if env → messageType = MT_SLEEP then ▷ If it's the sleep message, then we can wake up
17:            return
18:        else if listEnv ≠ NULL then ▷ Push onto queue if available
19:            env → next ← NULL
20:            listEnv[0] ← env
21:            listEnv ← ADDROF(env → next)
22:        else ▷ Send it to ourselves on a delay to process later otherwise
23:            DELAYED_SEND(PID, env, ms)
24:        end if
25:    end loop
26: end function
```

Chapter 9

User-Level Test Processes

9.1 Description

We had various user-level processes that we created for the first deliverable, and continued to use throughout creating our OS. As we continued adding more features, the user processes were modified as well. Most notably, they were modified in order to incorporate a sleep function, which we had already implemented for the second deliverable after adding message passing. As we set the processes to sleep for different delays, it is possible to see the interleaving of message printing by the CRT process when their delays synchronise. The processes are listed below.

9.1.1 'funProcess'

This process would send 5 messages to the CRT process to print the numbers 1, 2, 3, 4, and 5. It would then sleep for 3000 milliseconds. This would be repeated indefinitely. This process, along with the other processes, had to be modified for the second deliverable when we switched from straight UART polling to passing messages to the CRT process for printing.

9.1.2 'schizophrenicProcess'

This process is similar to 'funProcess', printing the numbers 9, 8, 7, 6, and 5. It would then sleep for 4000 milliseconds. This would be repeated indefinitely.

9.1.3 'fibProcess'

This process would print the first 40 or so Fibonacci numbers. It would print these 5 at a time, then sleep for 1000 milliseconds. After reaching the Fibonacci number just under 1000000000, it would restart at 1. This would be repeated indefinitely.

9.1.4 'memoryMuncherProcess'

This process would request all available memory blocks, printing the addresses of the obtained memory blocks. When it has obtained all available blocks, it would try to acquire another block, which would cause the process to block, waiting for an additional memory block to become available. Prior to this, it would set the priority of 'releaseProcess' to the highest possible, which would free a preallocated block of memory, allowing this process to continue. After becoming unblocked, it would free all held memory blocks and then block on receiving a message indefinitely.

9.1.5 'releaseProcess'

This process was used in conjunction to the above 'memoryMuncherProcess'. At startup, this process had the highest priority. It would preallocate a memory block (used to unblock 'memoryMuncherProcess'), then set its own priority to the lowest possible. It would then call release_processor. Our other user processes were running at higher priority than this one was set to, so it would require a set_process_priority in order to get it to run again, provided by 'memoryMuncherProcess'. It would continue executing after having its priority set to the highest possible again by 'memoryMuncherProcess', and free its single block. After doing so, it would block on receiving a message indefinitely. The combination of these two processes allowed us to test low to no memory situations to ensure our memory allocator was working as expected, as well as to ensure that our APIs for setting and getting priorities were working correctly.

Part IV

Lessons Learned

During the implementation phases of our OS project we went through several deliverables, which allowed us to learn various lessons. At times, we decided to learn from our mistakes and not to make the same mistakes repetitively. In other cases, we did not learn from our mistakes, and lost a lot of time in the process.

Our development process consisted in writing code on our laptops, and committing it to a centralized version control system. In our case, we used Git, an open-source distributed version control system. When group members decided code they had written was of adequate quality, they pushed their code to a new branch on a Git server. The new code would get a thorough review from team members, and when they were happy with the new code, the branch was merged into the main development source tree. This process saved us a lot of time, as many bugs were caught as they were being introduced, instead of in debugging. If we had not used version control, and didn't use the peer code-review system, it is entirely possible we wouldn't have been able to complete the lab successfully. This was a very good organizational decision that we started early on, and kept with for the duration of the project.

Since we only attempted compilation with Keil near the deliverable deadline, we had to make a large number of purely syntactical fixes to fit the requirements of the Keil IDE. For example, we occasionally did not pre-declare variables at the top of blocks, or forgot to put a newline at the end of a file. This caused us much undue stress, as we raced towards the first deliverable making sure our code compiled correctly inside Keil. Luckily, our peer review system ensured our first deliverable was mainly bug-free, which saved us time actually testing on the board.

One of the issues that seemed to continually plague us (until we made it easier to detect), was incorrectly accessing memory that had not been initialized. This caused many unusual bugs that would only occur in some circumstances where memory we accessed would contain unusual data. By changing our memory allocator to always zero-initialize memory returned, we were able to quickly identify invalid memory accesses. This in turn helped us fix these kinds of bugs very quickly.

In some initial assignments, we ran into problems with looming deadlines. As we learned from these experiences, we started working on assignments earlier and earlier, in order to reduce our stress at submission time. One of the side effects was allowing us to write our enhanced CRT output display.

One of the overall goals for our operating system was to provide efficient

implementations to all user-side operating system calls. By maintaining the intention to reduce the runtime of operations (i.e. $O(n) \rightarrow O(\log(n))$, etc.), we feel we are not only able to say our implementations are faster than naive implementations, but guarantee performance to be better.

Overall, we are very happy with our implementation. It succinctly implements the required portions of the project (memory, messages, I/O, etc.) while ensuring high performance on a stable API.

Appendix A

Raw Measurement Data

We profiled the performance of the memory allocator using the Keil IDE's built- in profiling tools. There were 5 trials completed with different total runtimes and different sets of processes active. The Trial Information table shows the runtime of the system for each trial and notes about how the trial deviated from the normal setup of the system (which defaults to having all processes on, without the clock or the stress processes started). The next table shows the total and average runtimes of the measured functions for each trial.

A.1 Trial Information

Trial	Total Runtime	Notes
1	4.219	Normal (no stress processes)
2	7.754	Wall clock
3	8.487	Normal (no stress processes)
4	6.5	No Memory Muncher or Release Process
5	30.988	Stress processes

A.2 Function Runtime Profiling

Function	Trial	Time (μs)	# of Calls	Average time / call (μs)
k_sendMessage	1	601.58	552	1.090
k_receiveMessage	1	408.22	565	0.723
k_acquireMemoryBlock	1	244.12	294	0.830
k_sendMessage	2	647.44	594	1.090
k_receiveMessage	2	437.78	606	0.722
k_acquireMemoryBlock	2	258.68	320	0.808
k_sendMessage	3	630.99	579	1.090
k_receiveMessage	3	426.83	591	0.722
k_acquireMemoryBlock	3	259.24	321	0.808
k_sendMessage	4	108.80	100	1.088
k_receiveMessage	4	74.44	110	0.677
k_acquireMemoryBlock	4	92.47	123	0.752
k_sendMessage	5	750.63	687	1.093
k_receiveMessage	5	497.09	693	0.717
k_acquireMemoryBlock	5	329.90	447	0.738

Appendix B

Work Breakdown

While we do have extensive logs of who committed different parts of the code, it is difficult to say who worked on what. This is because we often pair-programmed, and it would be unfair to say that the 'credit', such as it were, goes to the person who committed the code. Also, we did detailed code-reviews, so the other team members often put in an equal amount of time reviewing the code as did the person writing the code. We believe it is fair to say that we all put an equal amount of time into this project.

As to how we split up the work, we used a bug tracker to list all the requirements for a particular deliverable. Then, on a first come, first served basis, one of us would claim a bug, and then work on fixing that part of the code. This quick and informal mechanism meant that we were not often blocked waiting for meetings with other team members to begin working on the project.