SE350 RTX Project Report

Group G013

Term W13
April 4, 2013

Group members:

- Mohammad Shahroze Khan (ms22khan, #20381206)
- Nicholas Kim (n24kim, #20380036)
- Alireza Pourhaj (apourhaj, #20385214)
- Manodasan Wignarajah (mwignara, #20371224)

Table of Contents

Introduction

This is our project report for the SE350 RTX design project, in which we were tasked to create a simple operating system for the Keil MCB1700 ARM boards.

The project was separated into three successive parts, each building on the last.

1. Part 1 laid the foundations for the operating system, with the memory, process, and context switching modules implemented. Six user-level processes were created to test the foundations, and output a message. Interrupts were not a consideration for this part of the project; output was done using provided polling routines.
2. Part 2 added more advanced features expected in an operating system, such as message sending, an internal timer, and interrupt handling. In addition to the user-level processes, keyboard input was required to be handled, with a set of commands for handling a system wall clock implemented.
3. Part 3 was largely a stress test of Part 2, with provided test processes (named A, B, and C) forcing us to redesign and refactor much of the code we wrote for part 2.

The operating system was developed on Keil's µVision 4 IDE, and should theoretically work on the provided simulator (testing, however, was done on the actual boards). A serial port is used to feed the board's I/O to a PuTTY terminal, where keyboard input is accepted and all displayed messages are outputted.

Global Information

This part of the report describes the data structures underlying our operating system and the constants/initialization tables used.

Memory

Overview

The first and most important aspect of our operating system is memory - in particular, managing it and getting access to it when needed. As such, we begin with code for requesting and releasing blocks of memory, using a memory data structure to help encapsulate the details.

The memory available for our purposes is a block of contiguous space, with upper bound 0x10008000 (268468224 in decimal) and lower bound &Image$$RW_IRAM1$$ZI$$Limit, a predefined symbol. We begin by dividing this memory into equally-sized blocks, where the size is a customizable #define constant MEMORY_BLOCK_SIZE (default value 512). The number of available memory blocks is then computed and stored in a global variable named NUM_MEMORY_BLOCKS, which is incremented and decremented as memory is released or allocated, respectively. The variable can also be set manually if an artificial restriction on memory is desired.

Structure
To properly organize the available memory to meet all our needs, we created a header for each memory block, where the header was a structure called MemNode with the following elements:

- an integer representing the memory block ID (unique identifier)
- a flag representing whether the block is in use or not
- an integer representing the actual address of the raw memory block (that the user will use) - *note that this is the only part visible to the end user*
- a pointer to the next MemNode (for the purposes of chaining MemNodes as a singly linked list)

Because the size of each memory block (as determined by the constant MEMORY_BLOCK_SIZE) represents the size allocated to the user, each memory block in our operating system is in fact slightly larger than MEMORY_BLOCK_SIZE - we add the size of a MemNode to each block to accommodate the header. The exact code is:

```
MEMORY_SIZE = (unsigned int) ((unsigned) 0x10008000 - (unsigned int)
&Image$$RW_IRAM1$$ZI$$Limit);
NUM_MEMORY_BLOCKS = MEMORY_SIZE / (MEMORY_BLOCK_SIZE + sizeof(MemNode));
```

Refer to Figure 2 in Appendix A for a diagram summary.

Initialization and Running
Like all other operating system mechanics, memory is initialized on startup, using a memory_init function that sets up each memory block and associated MemNode structure.

Once initialized, the memory module remains passive until it receives requests for memory and/or the corresponding deallocation requests. If there is ever too much memory requested (i.e. NUM_MEMORY_BLOCKS reaches 0 and another block is requested), the request_mem_block function will have a blocking event (tagged as memory blocked, to distinguish it from other blocking events) issued, and the block will only be released once the release_mem_block function is called.

Process Control Block

Overview
Every process running on our operating system is represented internally with a process control block (PCB) structure. The structure contains the information required for the operating system to handle the associated process, e.g. process ID, priority, state.

Processes running on our operating system can broadly be classified into two categories: user and system (kernel). System processes include services fundamental to the operating system, such as I/O and timer routines, while user processes are arbitrary processes that perform tasks using the operating system essentials. The latter are typically denied permission to call kernel-level functions, relying instead on an indirection mechanism, which creates user-level aliases for certain kernel-level functions.

Note that all processes are known prior to operating system startup, and no process is ever created or destroyed during the operating system's runtime.

Structure

Each PCB is a structure containing the following info:

- an integer representing the process ID (unique identifier)
- an integer representing the priority
    - Valid values for priority range from 0-4, with a lower number representing higher priority. 4 is reserved for the *null process* (a process which serves no purpose except to run when no other process is scheduled).
    - 0 and 1 are typically used for system processes, to ensure that they run when needed. 1-3 are typical user process values. Of course, these are only guidelines, and the operating system technically continues to work if they are not followed.
- an enumeration representing the type of the process
    - Possible values include USER, INTERRUPT, SYSTEM, DEBUG.
    - The latter three are different types of system processes, while the first represents all user processes.
- an enumeration representing the state of the process
    - Possible values include NEW, RDY, RUN, BLKD, INTERRUPT, EXIT.
- an enumeration representing the what the process is currently blocked on
    - Possible values include NONE (i.e. normal), MEM_BLKD (blocked due to lack of memory), MSG_BLKD (blocked waiting for message), SEM_BLKD (blocked due to semaphore guard)
- a pointer representing the master stack pointer of the process
- a pointer to the next PCB (for the purposes of chaining PCBs as a singly linked list)
- a pointer to the head of the mailbox of the process (where messages are received; chained as a singly linked list)

Refer to Figure 1 in Appendix A for a diagram summary.

Process

Overview

Accompanying the PCB structure is a process structure, which stores information that, while not required for the kernel to run, is nevertheless crucial for enabling a process to run. As such, it is designed to hold a pointer to an accompanying PCB internally, with only a little bit of extra information; where not needed, PCB structures are passed around instead of the full process structure.

Structure

Each process structure contains the following info:

- a pointer to the accompanying PCB (described in the appropriate section)
- an integer representing the starting address of the process

    - This is not touched by the kernel main API, and is instead handled by setting up the exception stack frame during the initialization of processes.
- a pointer representing the stack allocated to the process, for storage of the process' own local variables, etc.

Message

Overview

Also fundamental to our operating system are messages, which, as the name implies, are packets of data sent from one process to another through a central mechanism, and received similarly. These messages are used for a variety of purposes at the system level, such as the wall clock process sending messages to the CRT process to display the current time. Of course, user processes are also free to make use of the messaging system through the same indirection mechanism employed by other kernel-level services.

There are a number of important facts regarding the use of messages:

- Send and receive operations are atomic, thanks to the use of a semaphore. Interrupts are also disabled during the time these operations take place.
- There are two types of receive functions, a blocking and non-blocking variant respectively. The non-blocking variant returns NULL in the event that it finds no message in the queue for that particular process.
- There are two types of send functions, an instant and a delayed variant respectively. The delayed variant relies on sending an instant message to the timer, and then sending the real message to the appropriate process after the correct amount of time has elapsed.
- Each message requires a single memory block for its storage. If memory cannot be allocated, an internal check inside the message functions will fail, causing no message to be sent and an error code returned.

Structure

Each message is a structure containing the following info:

- an integer representing the process ID of the sender
- an integer representing the process ID of the receiver
- an integer representing the type of message
    - This variable is technically somewhat arbitrary, as processes decide for themselves what "types" of messages they wish to listen for. However, the current convention used by our operating system is:
    - 0 = empty message, 1 = integer, 2 = kernel level messages, 3 = count report, 4 = command registration, 5 = wake up 10 (used by a test process)
- a void pointer representing the raw body (data) of the message
    - The receiver of the message must know in advance what the type and structure of the data being sent is.
- a pointer to the next message (for purposes of chaining messages as a singly linked list)
- an integer representing the time when the message is expected to reach the destination process (for purposes of delayed sending)

Note that when creating a message, only the destination process ID, the message type, and the message contents are actually set by the sending process. The other fields are filled in automatically by the message functions upon sending.

Refer to Figure 2 in Appendix A for a diagram summary.

Queues

Overview

While not a feature of operating systems per se, in the same sense as memory allocation, processes, and so forth, our heavy use of queue (and priority queue) structures prompted us to design a generic queue structure and functions, which reduced the amount of duplicate code for handling PCB and message queues, and permitted easier debugging (queue code in one place) Special-casing some PCB functionality, however, was left in for the purposes of convenience.

Structure

A queue is represented by a single pointer pointing to the head of what is assumed to be a singly linked list, and a priority queue is represented by an array of such pointers (where the index of each pointer represents the priority).

The functions for managing these queues include:

- insert_pq (insert element into PCB priority queue)
- remove_pq (remove element from PCB priority queue)
- get_process (retrieves next available process from PCB priority queue; higher priority processes obviously take precedence)
- lookup_pid_pq (returns the PCB of the process with a certain ID, or NULL on error)
- enqueue_q (generic insertion into queue)
- dequeue_q (generic removal from queue)

All generic functionality was implemented with void* pointers, with an additional parameter specifying what the void* pointer really points to, to enable conversion in the generic functions.

Primitives

crt_display.c

non-blocking output

- Uses messages to receive output requests.
- Can output chars and numbers (using a function to convert an integers to a char array)
- Handles the hot-key key press
    - We have one hot-key to print out the following
        - process id

            - priority
            - status
                - if blocked, what they are blocked on

        - The hotkey supports all the user processes and a few system processes such as the change priority system process, keyboard system process, and wall clock system process.
        - Concatenates everything using a concatenation function in order to reduce the number of messages sent (to CRT_SYSTEM_PROCESS for outputting purposes).
- reverse_c_string(start, end) -> used to reverse the char representation of integers. (123 becomes 321 after getting converted to an integer, so we need to reverse it)

Psuedocode

```
// initialize the hotkey process
void hotkey_init(void)
  request memory block for the hotkey process
  initialize hotkey_pcb
  initialize stack pointer
  insert the hotkey pcb into the process queue
  request memory block for the hotkey reserved message block

// initialize the crt process
void crt_init(void)
  request memory block for the crt process
  initialize crt pcb
  initialize stack pointer
  insert the crt pcb into the process queue
  request memory block for the hotkey reserved message block

// function responsible for initiating the printing
void crt_print(input)
  request a memory block
  set the “input” as the message’s data
  set message type to 1
  send the message to CRT_PID

// when integers are converted to char*, they are reversed, therefore we need to reverse them
void reverse_c_string(start, end)
  reverse the string in place

// convert an integer to its char* representation
void int2str(int input, char* beginning)
  if zero, return ‘0’
  while input > 0
    convert the last digit to its char representation
    divide by 10
    increment the buffer holding the string representation of the the number
  reverse the buffer
  add null terminating character

// takes in a pcb, and concatanates the processes’ id, state and priority to the output
```

```
void hot_key_helper(states, output, pcb)
  iterate = pcb
  buffer = int2str(iterate->pid)
  concat (output, buffer)
  buffer = int2str(iterate->priority)
  concat (output, buffer);
  concat (output, iterate->state)
  buffer = "\n\r\0"
  concat (output, buffer)
```

process.c

```
// inserts the PCB into the ready queue
int insert_process_pq(pcb)
  insert pcb from readyQueue

// removes the PCB from the ready queue
int remove_process_pq(pcb)
  remove pcb from readyQueue

// performs a lookup on the PID (for all known pids in the system)
PCB lookup_pid(pid)
  for i = 0...n
    if (pcb_list[i].pid == pid)
      return pcb_list[i]
  switch (pid)
    case TIMER_PID:
      return timer_pcb;
    case CRT_PID:
      return crt_pcb;
    case WALL_CLOCK_PID:
      return wall_clock_pcb;
    case HOTKEY_PID:
      return hotkey_pcb;
    case KEYBOARD_PID:
      return keyboard_pcb;
    case SET_PROCESS_PCB_PID:
      return set_process_pcb_pcb;

// given a PID returns the priority of the PCB if found
int k_get_process_priority(process_ID)
  PCB process = lookup_pid(process_ID)
  if (process exists)
    return process->priority
  else return -1

// sets the priority of the PCB given the PID, otherwise returns -1 if not found or incorrect
priority
int k_set_process_priority(process_ID, priority)
  PCB process = lookup_pid(process_ID)
  if (process does not exist) return -1
  if (0 <= priority < NUM_PRIORITIES)
    remove process from current_queue
    process->priority = priority
    insert process to previous_queue
  else return -1

// initializes the process module
void process_init(void)
```

```
  insert usr_procs into ready_queue
  current_process = NULL_proc

// gets the interrupt state
// (100 = crt, 010 = keyboard, 001 = timer, 111 = all enabled)
int k_get_interrupt_state(void)
  return current_interrupt_state

// sets the interrupt state given 3-bits (BBB)
// newState = 3 bit format (000, 001, 010, 011, 100, 101, 110, 111)
void k_set_interrupt_state(newState)
  if (newState & 4 == 1) enable_timer
  if (newState & 2 == 1) enable keyboard
  if (newState & 1 == 1) enable crt
  current_interrupt_state = newState

// sets the current_process to the NULL process
void process_reset(void)
  current_process = NULL_proc
```

pq.c

```
// general purpose insert_pq that takes in a priority queue and PCB and performs a priority //
based enqueue
int insert_pq(PCB pq[], PCB p)
  if (p == NULL || !  (0 <= p->priority < MUM_PRIORITIES))
    return -1
  pr_head = pq[p->priority]
  p->next = NULL
  if (pr_head does not exist) pq[p->priority] = p
  while (pr_head->next exists)
    pr_head = pr_head->next
  pr_head->next = p
  return 0

// general purpose remove_pq that takes in a priority queue and a PCB and performs a dequeue //
based on priority
int remove_pq(PCB pq[], PCB p)
  if (p == NULL || !  (0 <= p->priority < MUM_PRIORITIES))
  return -1

  find p in pq
  p.previous = p->next
  return 0;

// returns the head of the priority queue (lowest priority process)
PCB get_process(PCB pq[])
  return head of pq

// returns the PCB given the PID in the given priority queue
PCB *lookup_pid_pq(PCB* pq[], int pid)
  for i = 0...NUM_PRIORITIES
    for j = 0...n
      if pq[j].pid == pid
        return pq[j]
  return NULL

// general purpose enqueue function that performs an enqueue operation on MSG/PCB queues
```

```
int enqueue_q(void* pq_generic, void* p_generic, q_type type)
  p_generic->next = NULL;
  if (*pq_generic == NULL)
    *pq_generic = p_generic
  pq_generic.lastElementInQueue = p_generic;
  return 0;

// general purpose dequeue function that performs a dequeue on MSG/PCB queues
void *dequeue_q(void* pq_generic, q_type type)
  return dequeue pq_generic.headOfQueue
```

message.c

```
semaphore send;
semaphore receive;

// initializes the message module
void message_init(void)
  semInit(send)
  semInit(receive)

// non-blocking request used to retrieve head of the mailbox given a PID if it exists
MSG get_message_pid(int process_ID)
  atomic(on)

  PCB process = lookup_pid(process_ID)
  if (process exists)
    atomic(off)
    return process->msgHEAD
  else
    atomic(off)
    return NULL

// non-blocking request used to retrieve head of the mailbox given a PCB if it exists
MSG k_get_message(PCB pcb)
  atomic(on)
  if (pcb does not exist) return NULL
  atomic(off)
  return pcb->msgHEAD

// non-blocking Send: sends the message atomically to the destination_PCB’s mailbox + releases
processor if destination has //greater priority
int send_message_global(int dest_process_ID, void *MessageEnvelope, int router_process_pid, long
delay)

  atomic(on)
  msg->destination_pid = dest_process_ID
  msg->next = NULL
  msg->expirt_time = get_current_time() + delay

  dest_proc = lookup_pid(router_process_pid)

  if (dest_proc->state == BLKD && dest_proc->status == MSG_BLKD && delay == 0)
    remove_pq(msg_pq, dest_proc)
    dest_proc->state = RDY;
    dest_proc->status = NONE;
    insert_process_pq(dest_proc) // inserts back to ready queue
  atomic(off)
  if (dest_proc->priority < current_process->priority && !isI-Process(current_process && msg-
>msg_type != 4)
    k_release_processor();

// routes to send_message_global
int k_send_message(int process_ID, void *MessageEnvelope)
  return send_message_global(process_ID, MessageEnvelope, process_ID, 0)

// routes to send_message_global
int k_send_message(int process_ID, void *MessageEnvelope)
  return send_message_global(process_ID, MessageEnvelope, process_ID, 0)

// blocking Receive: runs atomically until the message is found and returns the msg. Blocks the
process and releases if //message not current found.
void *k_receive_message(int *sender_ID)
  atomic(off)

  while (current_process->head == (void *)0)
    dest_proc = lookup_pid_pq(msg_pq, current_process->pid)
    if (dest_proc == NULL)
      current_process->state = BLKD; // Set state to BLKD
      current_process->status = MSG_BLKD; // Set status to MSG_BLKD
      insert_pq((PCB**)msg_pq, current_process);
    atomic(off)
    k_release_processor()
    atomic(on)

  msg = dequeue_q(current_process->head, MSG_T)
  atomic(off)
  return msg
```

semaphore.c

```
// initializes the semaphore
void semInit(semaphore *s)
  s->count = 1
  s->queue = NULL

// the caller gets the 'lock' if it is available, otherwise it is blocked and placed in the
semaphore queue
void semWait(semaphore *s)
  atomic(on)
  s->count--;
  if (s->count < 0)
    current_process->state = BLKD
    current_process->status = SEM_BLKD
    remove_process_pq(current_process)
    enqueue_q(&(s->queue), current_process, PCB_T)
  atomic(off)
  k_release_processor()

// the caller lets go of the 'lock', and it unblocks the head of the queue if it exists (ie.
there was a process waiting for this 'lock'
void semSignal(semaphore *s)
  atomic(on)
  s->count++;
  if ( s->count <= 0)
```

```
    PCB p = dequeue_q(s->queue, PCB_T)
    p->state = RDY
    p->status = NONE
    insert_process_pq(p)
  atomic(off)
```

main.c

```
// initializes all the modules (memory, process, system processes, interrupts, etc.)
void __init()
  call to init of all modules
  reset current process

// main function
int main
  SystemInit()
  atomic(on)
  __init()
  atomic(off)
  set control to MSP mode
  release_processor()
```

memory.c

```
// initializes memory module by setting up memory blocks
void memory_init()
  NUM_MEMORY_BLOCKS = TOTAL_MEMORY_AVAILABLE/(MEMORY_BLOCK_SIZE + HEADER_SIZE)
  for i = 0...NUM_MEMORY_BLOCKS
    memory_block = ith memory block
    set memory_block number to i
    set memory_block to not used
    set address to block pointer address + HEADER_SIZE
    set next to this address + MEMORY_BLOCK_SIZE

// system api call to request memory block, if none blocks process
void k_request_memory_block()
  if (no memory blocks left)
    set process to blocked
    insert in blocked queue
    release_processor

  while (memory_block != null)
    if (!used)
      set memory_block to used
      return memory_block address

    memory_block = next memory_block

// system api call to release memory block, unblocks a process if any blocked on memory
void k_release_memory_block(mem_block)
  while (memory_block != null)
    if (memory_block == mem_block)
      set memory_block to not used
    if (any processes blocked of memory)
      unblock one process
      insert unblocked process on ready queue
      return
   memory_block = next memory_block
```

process.c

```
// gets pid next process to be scheduled
int scheduler()
  for i = 0...NUM_PRIORITIES
    get process from priority queue[i] // first process in linked list
    while (process != null && process can not be scheduled)
      process = next process in linked list

    if(process != null)
      return pid of process

  return null process  // none can be scheduled

// system api call to tell OS that current process is done, and to schedule next
int k_release_processor()
  pid = scheduler() // next process
  k_context_switch(pcb of pid)

// switches context to run given process
int k_context_switch(process)
  atomic(on)
  old_process = current_process
  current_process = process
  if (current_process == null)
    atomic(off)
    return error

  if (current_process->state == new)
    save old_process msp to __get_MSP
    if (old_process state != blkd)
      set old_process state to interrupted if interrupted or otherwise ready
      insert old_process to ready queue

    remove current_process from ready queue
    set current_process state to run
    set_MSP(current_process msp)
    atomic(off)
    __rte()
  else if (current_process state == ready || interrupted)
    save old_process msp to __get_MSP
    if (old_process state != blkd)
      set old_process state to interrupted if interrupted or otherwise ready
      insert old_process to ready queue
    remove current_process from ready queue
    set current_process state to run
    set_MSP(current_process msp)
  else
    atomic(off)
    return error
  atomic(off)
  return 0
```

timer.c

```
// gets current time
long get_current_time()
  return time
```

wall_clock.c

```
// initializes wall clock process and performs command registration
void wall_clock_init()
  set up pcb for wall clock process as system process
  set up stack
  add to ready queue
  send command registration message to keyboard for command W

// converts time string to seconds
int hms_to_ts(char *hms)
  hour = hms[0 - 1]
  min = hms[3 - 4]
  secs = hms[6 - 7]
  return hour * 3600 + min * 60 + secs

// converts seconds to time string
void ts_to_hms(int ts, char *buffer)
  hour = get hour from ts
  min = get minutes from ts
  sec = get seconds from ts
  set buffer[0 - 1] = hour
  set buffer[2] = ':'
  set buffer[3 - 4] = min
  set buffer[5] = ':'
  set buffer[6 - 7] = sec
  set buffer[8 - 9] = '\r\0'
```

Interrupts

This section summarizes the various interrupts in our operating system and their mechanisms.

Overview

Interrupts can broadly be classified into software and hardware interrupts. Software interrupts are essentially the mechanism via which processes invoke kernel services, while hardware interrupts are those triggered by the hardware themselves, i.e. timer and UART interrupts.

Software

Memory module

request_memory_block

- Allows user processes to request memory block to store data
- Blocks process if there is no memory left
- Returns a memory address which user process can store data at

release_memory_block

- Allows user processes to release allocated memory block, giving it back to available memory
- Parameter is the address of the block which was given to user process by

request_memory_block
- Unblocks the highest-priority process waiting on memory, if any

Process module

get_process_priority
- Allows retrieval of the priority of the process represented by the PID
- Parameter is the PID of the process whose priority is required
- Returns the priority if process exists, or-1 if it doesn't exist

set_process_priority
- Allows setting the priority of the process represented by the PID
- Parameter is the PID of the process, and the priority which the process is being set to
- Returns the set priority if successful or -1 if not

release_processor
- Informs kernel that the current process is done what it needs to do, and lets go of processor for another process
- Returns 0 if successful, -1 otherwise

k_context_switch
- Used by interrupts and system processes to force a context switch onto a certain process
- Parameter is the PCB pointer of the process to switch to
- Returns 0 if successful, -1 otherwise

CRT display module

crt_print
- Allows printing to console
- Parameter is the string to print

crt_output_int
- Allows printing integers to console if required by user process (provided for convenience)
- Parameter is int to print to console

Message module

get_message_pid
- Non-blocking receive used as a lookup for the mailbox (queue) of message for system processes

- Retrieves a message pending for the process if available
- Parameter is process_ID, the PID for the process whose mailbox will be searched for a message
- Return null if mailbox empty, else a pointer to the message

get_message_pcb

- Non-blocking receive used as a lookup for the mailbox (queue) of message for system processes
- Retrieves a message pending for the process if available
- Parameter is PCB, the PCB for the process whose mailbox will be searched for a message
- Return null if mailbox empty, else a pointer to the message

send_message

- Non blocking call to send a message from one process to the mailbox (queue) of another process
- First parameter is process_ID, the PID of the destination
- Second parameter is MessageEnvelope, a memory block containing the message structure

receive_message

- Used by process to retrieve message from its mailbox (queue)
- Blocks process if there is no message in the mailbox until there is one
- Parameter is sender_ID, the PID of the sender, used for output
- Returns the message

delayed_send

- Non blocking call to send a message from one process to the mailbox (queue) of another after a certain delay
- First parameter is process_ID, the PID of the destination
- Second parameter is MessageEnvelope, a memory block containing the message structure
- Third parameter is delay, which indicates after how many milliseconds to send the message

Hardware

Overview - Timer

Timer interrupts are set up to be triggered every millisecond by the hardware. This is done by setting the appropriate values in the timer register data structure so that it triggers every millisecond and enabling the setup timer. When one is triggered, the registers are saved by the interrupt handler, and the timer I-process is ran.

This I-process essentially increments the time (stored in milliseconds), and checks for any new delayed send messages. If there are any, they are added to their own queue and, in addition, the I-process looks for any sent messages with an expired delay. If any exist, they are sent to the appropriate process.

Pseudocode - Timer

```
// initializes timer interrupts and timer i-process
uint32_t timer_init(timer)
  if (timer == 0)
    set up timer interrupt
    set up pcb for timer i-process
    set up stack of process
    insert process in queue

// called when timer interrupt is triggered, calls timer interrupt handler
__asm void TIMER0_IRQHandler()
  push registers
  call c_TIMER0_IRQHandler
  pop registers

// handles timer interrupt
void c_TIMER0_IRQHandler()
  atomic(on)
  save current process
  context switch to timer i-process
  context switch to saved process
  if (preempt flag set)
    release_processor() // preempt
  atomic(off)

// timer i-process increments time, and performs necessary operations for delayed send
// messages(receiving new messages and sending expired messages)
void timeout_i_process()
  increment time
  while (msg to be received)
    enqueue(msg)

  while (exists expired message)
    remove message
    send message to pid in message
    if (process being sent to is higher priority than saved process)
      preempt by setting flag and break
```

Overview - UART

UART interrupts are setup to be triggered when a key is pressed.

The i_uart_init function simply sets up everything required to receive input from a particular port number (in this case 0) and to display output.

The interrupt handler is triggered when there is keyboard input or when the CRT is done processing the character output. If the former, the key being pressed is read (clearing the read flag) and, if the hotkey was pressed, simply context switches to the hotkey routine, or the character is passed along to the keyboard decoder routine otherwise. If the latter, the flag is set to indicate that the CRT is ready to output another character.

The uart_i_process function is used for displaying output. It accepts a buffer of characters and serially prints them out one character at a time whenever the CRT hardware is ready to process the next one, which is indicated by the flag set in the interrupt handler.

Pseudocode - UART

```
// initializes uart interrupts
uint32_t i_uart_init(n_uart)
  set up the pins
  set up the uart transmission configuration
  set up other required uart settings for interrupt
  enable uart interrupt
  Allocate un-allocatable memory block to use for hotkey command

// called when uart0 interrupt is triggered, calls uart interrupt handler
__asm void UART0_IRQHandler()
  push registers
  call c_UART0_IRQHandler
  pop registers

// process uart serial output to console display
void uart_i_process( uint32_t n_uart, uint8_t *p_buffer, uint32_t len )
  check if the call is for the supported uart
  disable keyboard interrupts
  while (there are characters to print)
    wait for flag to be set to signal crt is ready for next character
    set THR register with the current character to print
    set flag to 0 to identify crt is processing again
    go to next character
  enable keyboard interrupts
```

System and User Processes

This section describes the various processes on our system in detail, summarizing each process with accompanying pseudocode.

Null process

- Purpose: Give the processor something to run when there is nothing else to run
- Assumptions: None
- Dependencies: None
- Pseudocode:

```
void null_process()
  loop forever
    release_processor()
```

Test processes 1 - 6

Processes 1-6 were used to stress test the OS, and to check to make sure various components of the OS were working as intended.

proc1

- Purpose: Send two messages to process 2 (to make sure multiple messages could be sent to a process)
- Assumptions: None
- Dependencies: Have memory blocks to use for message sending; otherwise it'll block
- Pseudocode:

```
void proc1(void)
  ret_val <- 10
  status <- 10
  send_status <- 10
  x1 <- 10, x2 <- 20
  msg <- null
  msg2 <-  null
  print testing info ("START \n\r total # of tests)

  if (msg1_status = 0)
    msg <- request_memory_block()
    initialize msg
    send_status = send_message(2, msg)
    msg1_status = 1
  if (msg2_status = 0)
    msg2 <- request_memory_block()
    initialize msg
    send_status = send_message(2, msg2)
    msg2_status = 1

  if send_status = 0 or (msg1_status = 1 and msg2_status = 1)
    success
  else
    test failed

  ret_val <- release_processor()
```

proc2

- Purpose: Receive the two messages sent by proc1 (To test receiving multiple/queue holding multiple messages)
- Assumptions: proc1 successfully sends the messages to proc2
- Dependencies: Depends on proc1's messages.
- Pseudocode:

```
void proc2(void)
  ret_val <- 10
  msg <- null
  msg2 <- null
  pass <- 0;
  sender_pid <- -1
  release_status <- 0
```

```
 TEST2 <- 0
 msg = recieve_message(sender_pid)
 msg = recieve_message(sender_pid)
 if (msg and msg2 contain the expected data sent by proc1)
   pass++
   release_status <- release_memory_block(msg)
   release_status = release_status | release_memory_block(msg2)
   msg1_status = 0
   msg2_status = 0

 if (release_status = 0)
   pass++

 if (pass = 2)
   successful
 else
   test failed
```

proc3

- Purpose: Test for multiple send/multiple receive
- Assumptions: None
- Dependencies: Depends on delayed message to work properly
- Pseudocode:

```
void proc3(void)
  while (1)
    msg <- initialize message
    delayed_send to proc3
    received_msg = received_msg(the delayed msg that was just sent)
    if (recieved_msg->content is what we expected)
      pass <- 1

    if (pass = 1)
      successful
    else
      test failed
    release_memory_block(msg)
    ret_val = release_processor()
```

proc4

- Purpose: Test process for getting and setting process priorities
- Assumptions: None
- Dependencies: Set priority process to work properly
- Pseudocode:

```
void proc4(void)
  while(1)
    pass <- 0
    i <- get_process_priority(4)
    if (i = 2)
      pass++
    i <- set_process_priority(4, 3)
    i <- get_process_priority(4)
    if(i = 3)
```

```
      pass++
    i <- set_process_priority(4, 2)
    i <- get_process_priority(4)
    if(i = 2)
      pass++
    if (pass = 3)
      successful
    else
      test failed

    ret_val <- release_processor()
```

proc5

- Purpose: Test registers/general variable allocation
- Assumptions: None
- Dependencies: None
- Pseudocode:

```
void proc5(void)
  while(1)
    a=1, b=2, ...., y = 25, z = 26
    a +=1, b+=1, ..., y+=1, z+=1
    if(a = 2, b = 3, ..., y = 26, y = 27)
      pass <- 1
    if(pass = 1)
      successful
    else
      test failed
    ret_val = release_processor()
```

proc6

- Purpose: End of tests, print results
- Assumptions: None
- Dependencies: None
- Pseudocode:

```
void proc6(void)
  while(1)
    test_num_passed <- 0
   foreach proc
      if passed their test
        test_num_passed++

    print out the number of tests passed/failed
```

Processes A, B and C

- These three processes were used for verifying the behaviour of the system under heavy stress scenarios, one being the near depletion of system resources. Processes A, B and C stress test the depletion of memory blocks.
    - Process A:
        - Summary: Registers with Command Decoder as handler of command %Z. If %Z is the inputted command, in an infinite loop increment a counter, and send a message to Process B (content of the message is the counter value).
        - Purpose: Test the command registration functionality, consume memory blocks and send messages to process B.
        - Assumptions: None made
        - Dependencies: Having a successful command registration (with %Z) and having memory blocks available
        - Pseudocode:

```
procA(void)
  p <- request_memory_block
  register with Command Decoder as handler of %Z commands
  loop forever
    p <- receive a message
    if the message(p) contains the %Z command then
      release_memory_block(p)
      exit the loop
    else
      release_memory_block(p)
    endif
endloop
num = 0
loop forever
  p <- request memory block to be used as a message envelope
  set message_type field of p to “count_report”
  set msg_data[0] field of p to num
  send the message(p) to process B
  num = num + 1
  release_processor()
endloop
```

    - Process B:
        - Summary: Forward messages received from Process A to Process B
        - Purpose: Test message forwarding.
        - Assumptions: None made
        - Dependencies: Successfully receiving messages from the message queue
        - Pseudocode :

```
procB(void)
  loop forever
```

```
    receive a message
    send the message to process C
  endloop
```

- Process C:

  - Summary: Prints out “Process C” every 10 seconds (uses delayed message (written by Shah), to implement the 10 second hibernates).
  - Purpose: Test sending out delayed messages (10 seconds) and freeing the memory blocks used by process A.
  - Assumptions: None made
  - Dependencies: Successfully receiving messages from the message queue
  - Pseudocode:

```
procC(void)
  perform any needed initialization and create a local message queue
  loop forever
    if (local message queue is empty) then
      p <- receive a message
    else
      p <- dequeue the first message from the local message queue
    endif
    if msg_type of p == “count_report” then
      if msg_data[0] of p is evenly divisible by 20 then
        send "Process C" to CRT display using msg envelope p
        hibernate for 10 sec
      endif
    endif
    deallocate message envelope p
    release_processor()
  endloop
```

CRT System Process

- Purpose: Performs blocking receive, and output the message’s data when it’s received
- Assumptions: The data passed in for printing is terminated by the null terminating character
- Dependencies: None
- Pseudocode:

```
crt_system_process(void)
  msg = perform a blocking listen
  output = msg->data
  length = output.length
  uart_i_process(0, output, length)  //call the provided function to output to display
```

Hotkey System Process

- Purpose: Handle the hotkey press
- Assumptions: There is a memory block set aside for this process, since it’s supposed to work, even if we are out of memory blocks.
- Dependencies: 1 memory block which should be met due to assumption
- Pseudocode:

```
void hot_key_handler(void)
  while(1)
    table_header = "\n\rProc_id  Priority Status\n\r"
    setup state char arrays
    p_state[1] = New
    p_state[2] = Ready
               ...
    foreach userproc
      hot_key_handler(p_states, output, (void)* 0)

      hot_key_helper(p_states, output, crt_pcb->pid, crt_pcb);
      hot_key_helper(p_states, output, wall_clock_pcb->pid, wall_clock_pcb);
      hot_key_helper(p_states, output, set_process_pcb_pcb->pid, set_process_pcb_pcb);
    output += ‘\0’
    send_messae(CRT_PID, output)
```

Wall Clock System Process

- Summary: Displays wall clock upon command is received, and allows to alter wall clock displayed time
- Purpose: To display wall clock to user
- Dependencies: There is at least one memory block free to receive command
- Pseudocode:

```
void wall_clock(void)
  while(1)
    receive command message

    if(command == reset)
      set time to 0
      set clock to running
    else if(command == set time)
      time = ts_to_hms(time passed in message)
      set clock to running
      release memory block of message

    while(clock_running)
      if(check for command message has message)
        if(command == reset)
          set time to 0
           set clock to running
        else if(command == set time)
           time = ts_to_hms(time passed in message)
          set clock to running
        else if(command == terminate)
          set clock to not running
          release memory block of message
          break
       release memory block of message
      if(1 second has passed since last update)
        set time = ts_to_hms(previous time in seconds + 1, buffer)
        crt_print(buffer)
```

Set Priority System Process

- Summary: Used to set the priority of different processes when the keyboard command it registered is used
- Dependencies: There is at least one memory block left for it to receive the command from the user
- Pseudocode:

```
void set_process_pcb_proc(void)
  Receive command
  Parse parameters from command into pid and priority
  set_process_priority(pid, priority)
  Send error message to crt if return result indicates error
```

Initialization

Overview

Broadly speaking, initialization is done in our top-level main function, where we call the initialization functions for each of our operating system components, disabling interrupts while doing so in order to ensure initialization proceeds smoothly. Our top-level code ends by simply calling our release_processor function, which begins the main event loop of our operating system.

The individual initialization functions generally do not take parameters, but can be customized by modifying certain values in the respective functions themselves.

Memory

Memory initialization is relatively straightforward, with the entire free memory space being divided into blocks of size MEMORY_BLOCK_SIZE (defined in memory.h) plus the space for each block's associated MemNode header. The number of memory blocks available is computed and stored as NUM_MEMORY_BLOCKS, but can be manually modified later if an artificial restriction is desired.

Process

Note: the naming here is a slight misnomer because "process_init", as our function is called, specifically initializes user processes and then inserts them into the process queue (system processes are initialized individually and separately).

As the internal workings of individual user processes are technically unknown to the kernel, each user process is initialized in a simple generic fashion, using a loop to march through an array of function pointers and initialize each process in that manner. The procedure is as follows:

1. Every user process is given its own PCB and priority structure (see the appropriate section for details). Using these, the processes' priorities (read off a customizable array), types (user), states (new), blocking statuses (none), mailbox queues (empty), starting address (using the function pointers), and stacks (allocated a memory block to store its local variables) are all set.
2. Every process is then inserted into the process priority queue as appropriate.
3. The first process in the priority queue (normally null process, but technically customizable) is then removed from the queue and the current process is set to that process.

Message

As messages are a passive component of the operating system (used on demand), message initialization merely consists of initializing the send/receive semaphores associated with it.

Keyboard, CRT, Hotkey, Wall Clock, Set Process Priority

These are the system processes, whose initializations are done in much the same manner as a user process initialization, except that the process type is set to INTERRUPT, SYSTEM, or DEBUG instead of USER.

UART

UART initialization code was largely given to us as part of the sample code, which we used with permission. However, one addendum of our own is worth noting; namely, we pre-allocate a memory block for messages sent to the hotkey process, so that even in the event that another process depletes available memory, the hotkey functionality will not be affected.

Implementation / Test Plan

Work Division and Timeline

Part 1 of the operating system project was largely a full-group effort, as virtually all the code was written with everyone present. Memory code was handled by Manodasan and Nicholas, process code by Mohammad, context switching code by Manodasan, and the user test process code by Alireza and Nicholas.

With part 2, we took a slightly different approach and split up the work ahead of time as follows:

- messages: Mohammad
- timer: Manodasan
- CRT display and hotkey: Alireza
- keyboard and wall clock: Nicholas

After writing our individual parts, we then met in the lab to integrate, debug, and test each other's code.

We then returned to immediate full-group effort with part 3, with bug fixes from part 2 predominating and everyone contributing towards fixing them. Concerning the small amount of new requirements for part 3, Alireza wrote the new test processes (ABC), Nicholas wrote the set process priority command functionality (i.e. %Z command), and Manodasan rewrote our scheduling functionality to accommodate our introduction (in code) of the conceptual difference between user and system processes (described in more detail in our design changes section).

Development Tools and Code Storage

Development was done on Keil's µVision 4 IDE, which automatically handled our building of our code and gave us the debugging tools we used for testing (e.g. simulator, memory watch).

Testing was primarily done using the user process functions (and processes A, B, C), rewriting as necessary to test new aspects of our operating system. While the simulator was used for part 1 and some of part 2, we eventually made the decision to test exclusively on hardware afterwards, as certain discrepancies between the simulator and the hardware began to make themselves apparent.

We used the revision control system tool Git for storage of our project, using the online service GitHub as our remote server. As Git is available on the ECE computers, this turned out to be an exceptionally convenient choice.

Major Design Changes

This section of the report describes the major design changes we made between various parts of the lab, as well as explaining the rationale behind each.

Lab 1 → Lab 2

In general, virtually no major changes were made in the transition between parts 1 and 2 of the lab. The only exception was the introduction of the concept of atomicity - while atomicity existed in lab 1, it was a redundant concept because we had no interrupts, meaning no system API call could be interrupted and put the operating system in an undefined state.

To accomplish atomicity, we used a helper class which permitted us to disable and enable specific interrupts by using bit flags, so that we can guarantee no interrupt can trigger which would leave the operating system in an undefined state.

Lab 2 → Lab 3

The transition to part 3, on the other hand, did require certain changes to our design. Specific instances include:

- Certain system processes such as the CRT display and wall clock were switched to directly by using a direct context switch call, instead of going through the scheduler. This was fixed in part 3, resulting in more elegant code and saving us the trouble of worrying about where we currently were in the code (what type of process are we in at the moment, is there some interrupt service routine, etc.).
- Our CRT process used to share the stack with the currently running process, which increased the chances of stack overflow. The CRT process was given its own stack in part 3, diminishing the chances of stack overflow and permitting the operating system to work with a smaller stack size (we had previously enlarged it to accommodate this issue).
- We introduced a new type of process, system, in part 3. System processes have the distinct feature of waiting on a command to activate them, and being blocked if there is no command to process, e.g. CRT waits for messages with data to print, and gets blocked if no messages are available.
- Storing keyboard input was previously handled in the interrupt service routine, with a buffer filling up and the buffer contents being sent to the keyboard decoder process when Enter was pressed. This has been moved to the keyboard decoder process itself, emphasizing the fact that the ISR is only meant to process the key press and nothing else.
- Command registration was hardcoded in part 2 due to the fact that there were a limited number of commands, all of which were known in advance. Due to the need for dynamic command registration in part 3, we implemented command registration through messages to the keyboard command decoder, sent during initialization of the processes.

Timing Analysis

These timings represent the average execution time of these function calls in about the first 10 seconds.

request_memory_block	1.88μs
send_message	1.48μs
receive_message	1.34μs

request_memory_block takes the longest time out of the three timed functions, due to the presence of a while loop searching for a free memory block. send_message has the second longest time and recieve_message has the lowest time because

- send_message has more lines of code to execute than recieve_message
- send_message contains an enqueue, which requires us to traverse to the end of a list one step at a time with a loop, while in receive_message, we have a dequeue operation, which only takes the first node out of the list, saving us the traversal of a list

Lessons Learned

Part 1

Technical

- We learned to work within severe memory constraints (32 KB vs. modern computers with many GB)
- We were introduced to scheduling processes and maintaining blocked event queues in addition to a process ready queue, demonstrating how a simple OS would manage such issues
- We learned the value of using the user processes for testing, as running them revealed bugs that we would not have found otherwise (and enabled us to fix them)
- We learned the value of commenting our code, as it enabled us to quickly understand other members' code without always having to consult them

Non-technical

- We learned that having all four members surround a single computer is generally not a good idea, as two people usually get deeply into the code and the other two become bystanders
- We learned that having everyone around causes ideas to be thrown out quickly, and in general accelerates decision making

Part 2

Technical

- We learned the basics of how processes communicate (in particular, with messages)
- We learned that modularizing our code base was of massive benefit, in particular writing generic functions for queue processing, which enabled isolation of bugs and reduced code complexity (less duplication)

- We learned to handle interrupts safely, which required us to have a solid understanding of the possible outcomes (e.g. designing lean vs. heavy ISRs, out-of-memory considerations, null checks, non-blocking functions required for ISRs)
- We learned that to design our code to be non-blocking for the most part, we needed to handle edge cases that could lead to deadlocks

Non-technical

- We learned that using GitHub and the branching schema permitted easier code merging and resolution of conflicts
- We learned the value of code reviews (particularly for resolving merge conflicts)
- We learned that dividing up work ahead of time tends to result in work being done faster, but not necessarily more efficiently - incorrect assumptions about each other's code often led to long debugging sessions
- We learned the importance of making sure everyone has a solid understanding of what exactly they are writing (in particular, requirements and specs)

Part 3

Technical

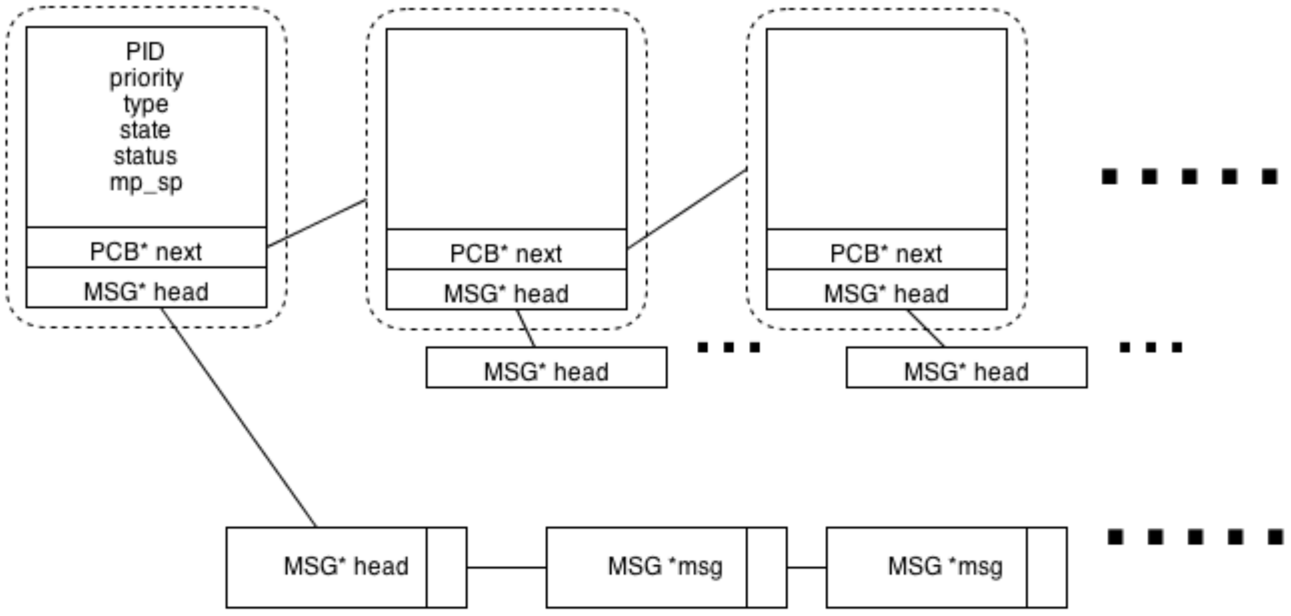
- Using the processes A, B, and C showed us certain lecture concepts such as atomicity (multiprogramming on a uniprocessor machine), race conditions, and deadlock
- By refactoring our code from part 2, we were able to fix certain bugs in our part 2 code that showed up during the running of processes A, B, and C
- Understanding the order of execution required a solid understanding of our operating system during the demo, in particular when explaining the consequences of certain priority choices; we broke down our OS in the debugger to help us gain a better understanding

Non-technical

- Overall, we caught a glimpse of the difficulty and the work that goes into creating an operating system, as well as understanding some of the mechanics behind group collaboration and code review

Appendix A: Diagrams

Figure 1: PCB



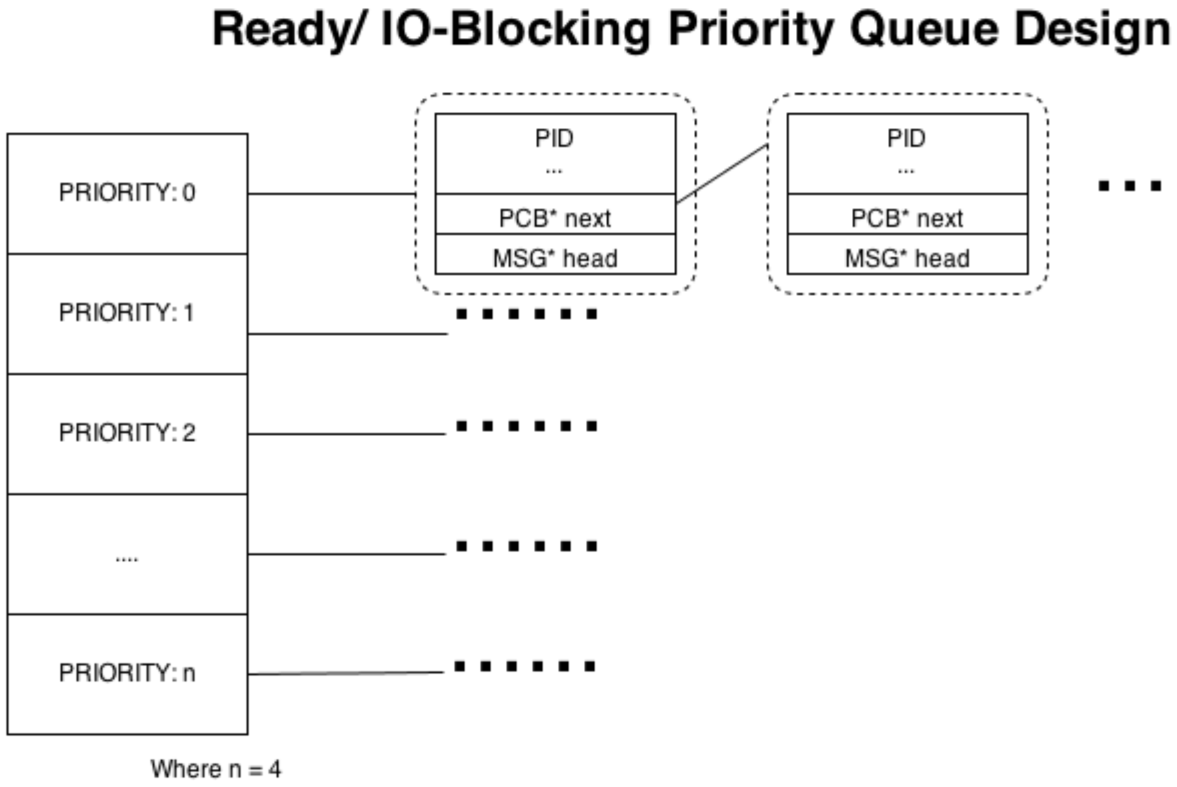
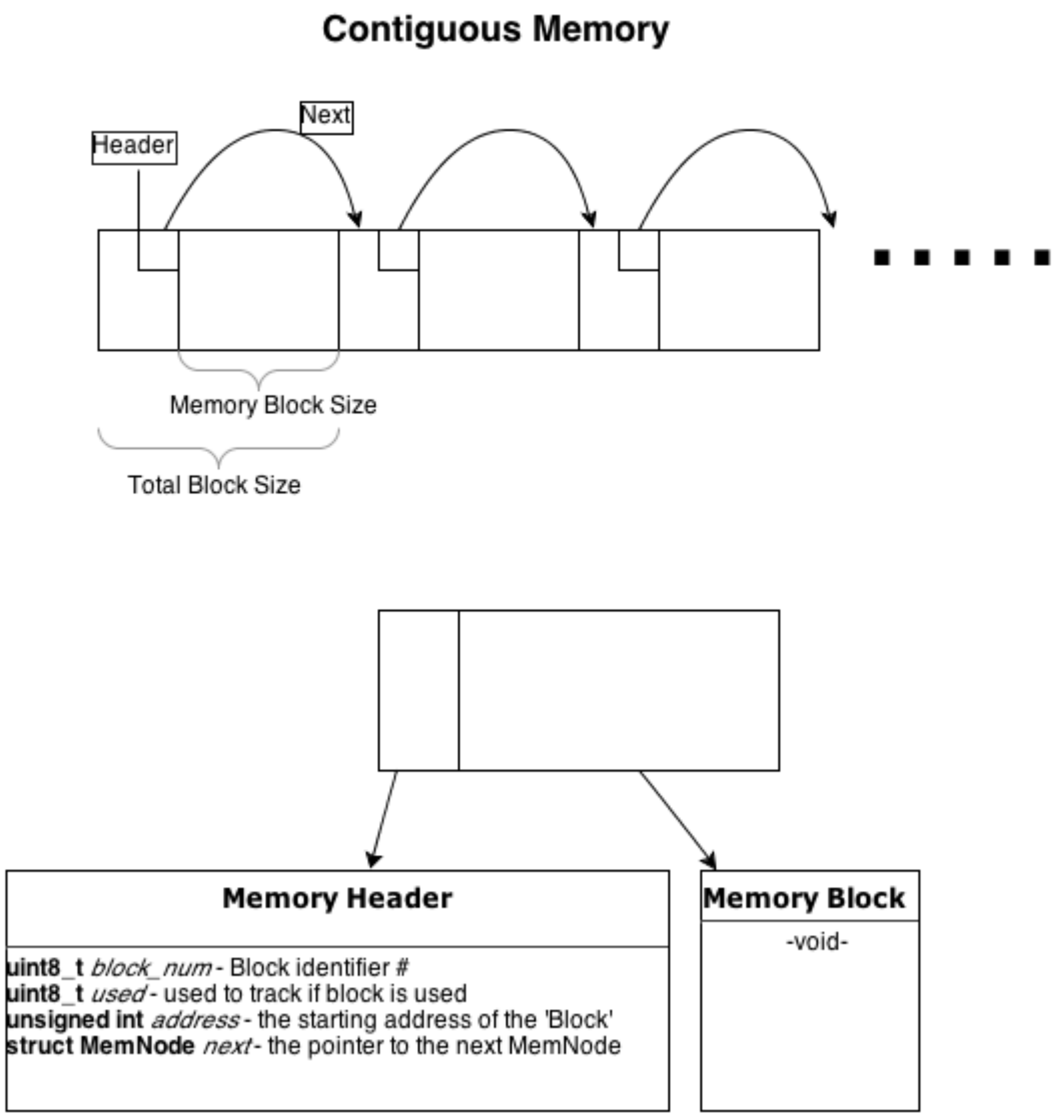


Figure 2: Memory


Contiguous Memory
Header
Next
Memory Block Size
Total Block Size
Memory Header
uint8_t block_num - Block identifier #
uint8_t used - used to track if block is used
unsigned int address - the starting address of the 'Block'
struct MemNode next - the pointer to the next MemNode
Memory Block
-void-