```h
#ifndef COMHAND_H
#define COMHAND_H

/**
 @file comhand.h
 @brief Command handler interface for the OS.
* Reads from the polling input and executes commands.
*/

/**
 * @brief Prints a welcome message and penguin ASCII art to the terminal
 */
void com_startup(void);

/**
 * @brief Trim function to remove \\n and \r from the string.
 * @param str string variable to trim
 */
void trim_Input(char *str);

/**
 * @brief Enters a loop and waits for the user to input commands.
 */
void comhand(void);

#endif

```

```c
#include "comhand.h"
#include <string.h>
#include <sys_req.h>
#include "help.h"
#include "exit.h"
#include "version.h"
#include "clock.h"
#include "showPCB.h"
#include "setPriority.h"
#include "ready.h"
#include "init.h"
#include "block.h"
#include "yield.h"
#include "loadR3.h"
#include "alarm.h"
#include "mcb/allocate.h"
#include "mcb/free.h"
#include "mcb/show.h"


// macaroni penguin ASCII image on startup
void com_startup(void) {
    sys_req(WRITE, COM1, "\r\n-------------------------------------------------------
----------------------\r\n", 80);
    sys_req(WRITE, COM1, "\r\n", 3);

    const char *bannerPart1 =
        "      \033[33m\\|/\033[0m         Welcome to
\033[33mMacaroniOS\033[0m!\r\n"
        "   -=(o ).        Version \033[36m";
    sys_req(WRITE, COM1, bannerPart1, strlen(bannerPart1));

    version_latest();

    const char *bannerPart2 =
        "\033[0m"
        "      '.-.\\\r\n"
        "      /|  \\\\      CS450: Operating Systems Structure\r\n"
        "     ' |  ||\r\n"
        "       _\\_):,_    Type '\033[33mhelp\033[0m' for a list of commands or
'\033[31mexit\033[0m' to shutdown.\r\n\r\n";
    sys_req(WRITE, COM1, bannerPart2, strlen(bannerPart2));
}

// trim Function for input (trims \r and \n from input)
void trim_Input(char *str){
    int length = strlen(str);
    while (length > 0 && (str[length - 1] == '\n' || str[length - 1] == '\r')){
        str[length - 1] = '\0';
        length --;
    }
}

// command handler (DOES NOT handle arguments)
void comhand(void)
{
    // startup message
    //com_startup();

    //TimeZone corrections
```

```
    tz_correction();

    // loop through the entire buffer
    for (;;) {
        char buf[100] = {0};

        // read command
        int nread = sys_req(READ, COM1, buf, sizeof(buf));
        buf[nread] = '\0'; // adds null terminator
        trim_Input(buf);    // trims \r\n
        sys_req(WRITE, COM1, "\r\n", 2); // Start new line

        // Yield CPU after reading input to allow multitasking
        yield();

        // to test some commands, we create user functions, but they are deprecated before "production"
        char* deprecated_msg = "\033[31mError: This function is depreciated and no longer usable.\033[0m\r\n";

        // command logic: each function handles its own argument(s) for better encapsulation
        if (strncmp(buf, "exit", 4) == 0) {
            char *args = buf + 4;
            while (*args == ' ') args++;
            if (exit_command(args)){
                sys_req(EXIT);
            }
        }
        else if (strncmp(buf, "shutdown", 8) == 0) {
            char *args = buf + 8;
            while (*args == ' ') args++;
            if (exit_command(args)){
                sys_req(EXIT);
            }
        }
        else if (strncmp(buf, "version", 7) == 0) {
            char *args = buf + 7;
            while (*args == ' ') args++;
            version_command(args);
        }
        else if (strncmp(buf, "help", 4) == 0) {
            char *args = buf + 4;
            while (*args == ' ') args++;
            help_command(args);
        }
        else if (strncmp(buf, "clock", 5) == 0) {
            char *args = buf + 5;
            while (*args == ' ') args++;
            clock_command(args);
        }
        else if (strncmp(buf, "show pcb", 8) == 0) {
            char *args = buf + 8;
            while (*args == ' ') args++;
            show_pcb_command(args);
        }
        else if (strncmp(buf, "priority set", 12) == 0) {
            char *args = buf + 12;
            while (*args == ' ') args++;
            set_priority_command(args);
```

```
115         }
116         else if (strncmp(buf, "suspend", 7) == 0) {
117             char *args = buf + 7;
118             while (*args == ' ') args++;
119             suspend_command(args);
120         }
121         else if (strncmp(buf, "resume", 6) == 0) {
122             char *args = buf + 6;
123             while (*args == ' ') args++;
124             resume_command(args);
125         }
126         else if (strncmp(buf, "create", 6) == 0) {
127             // char *args = buf + 6;
128             // while (*args == ' ') args++;
129             // create_pcb_command(args);
130             sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
131         }
132         else if (strncmp(buf, "delete", 6) == 0) {
133             char *args = buf + 6;
134             while (*args == ' ') args++;
135             delete_pcb_command(args);
136         }
137         else if (strncmp(buf, "block", 5) == 0) {
138             //char *args = buf + 5;
139             //while (*args == ' ') args++;
140             //block_pcb_command(args);
141             sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
142 
143         }
144         else if (strncmp(buf, "unblock", 7) == 0) {
145             //char *args = buf + 7;
146             //while (*args == ' ') args++;
147             //unblock_pcb_command(args);
148             sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
149 
150         }
151         else if (buf[0] == '\0') {
152             sys_req(WRITE, COM1, "\r", 2);
153         }
154         else if (strncmp(buf, "clear", 5) == 0) {
155             // this only moves the entire terminal up
156             sys_req(WRITE, COM1, "\033[2J\033[H", 7);
157             //com_startup();
158         }
159         else if (strncmp(buf, "yield", 5)==0){
160             // char *args = buf + 5;
161             // while (*args == ' ') args++;
162             // yield_command(args);
163             sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
164         }
165         else if (strncmp(buf, "load", 4)==0){
166             char *args = buf+4;
167             while (*args == ' ') args++;
168             load_command(args);
169         }
170         else if (strncmp(buf, "alarm", 5)==0){
171             char *args = buf+5;
172             while (*args == ' ') args++;
173             alarm_command(args);
174         }
```

```
        else if (strncmp(buf, "allocate", 8) == 0) {
            // char *args = buf + 8;
            // while (*args == ' ') args++;
            // allocate_command(args);
            sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
        }
        else if (strncmp(buf, "free", 4) == 0) {
            // char *args = buf + 4;
            // while (*args == ' ') args++;
            // free_command(args);
            sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
        }
        else if (strncmp(buf, "show mcb", 8) == 0) {
            char *args = buf + 8;
            while (*args == ' ') args++;
            show_mcb_command(args);
        }
        else {
            const char *invalidMsg = "\033[31mInvalid command. Please try
again.\033[0m\r\n\r\n";
            sys_req(WRITE, COM1, invalidMsg, strlen(invalidMsg));
            help_message();
        }
    }
}

```

```h
#ifndef ITOA_H
#define ITOA_H

/**
 * @file itoa.h
 * @brief Declaration for interger-to-ASCII conversion
 */

 /**
  * @brief Converts an integer to a C-string.
  * @param num The integer to convert.
  * @param buffer Pointer to an array to store the string.
  */

void itoa(int num, char* buffer);

#endif

```

```
void itoa(int num, char* buffer) {
  // get the absolute value of num
  int numLeft;
  if (num < 0) {
    numLeft = -1 * num;
  }
  else {
    numLeft = num;
  }

  // find the maximum power of 10 needed
  int maxPower = 0;
  while (numLeft > 1) {
    numLeft = numLeft / 10;
    if (numLeft < 1) {
      break;
    }
    else {
      maxPower++;
    }
  }

  // reset numLeft to the value of num
  if (num < 0) {
    numLeft = -1 * num;
  }
  else {
    numLeft = num;
  }

  // initialize variables
  int numToChar;
  int tenToThePower;

  // convert negative int to string
  if (num < 0) {
    buffer[0] = '-';
    for (int i = 1; i < maxPower + 2; i++) {
      // convert each int to char
      tenToThePower = 1;
      for (int j = 0; j < maxPower - i + 1; j++) {
        tenToThePower = tenToThePower * 10;
      }

      // write each char to buffer
      numToChar = (int)(numLeft / tenToThePower);
      buffer[i] = numToChar + '0';
      numLeft = numLeft - numToChar * tenToThePower;
    }
  }
  // convert positive int to string
  else {
    for (int i = 0; i < maxPower + 1; i++) {
      // convert each int to char
      tenToThePower = 1;
      for (int j = 0; j < maxPower - i; j++) {
        tenToThePower = tenToThePower * 10;
      }

      // write each char to buffer
```

```
      numToChar = (int)(numLeft / tenToThePower);
      buffer[i] = numToChar + '0';
      numLeft = numLeft - numToChar * tenToThePower;
    }
  }

  // add null terminator
  if (num < 0) {
    buffer[maxPower + 2] = '\0';
  }
  else {
     buffer[maxPower + 1] = '\0';
  }
}

```

```h
#ifndef ITOBCD_H
#define ITOBCD_H

/**
 @file itoBCD.h
 @brief Function that converts an integer into a string
* that is representative of the binary coded decimal
* format of the input integer.
*/

/**
 Convert an integer to an Binary Coded Decimal
 @param int Integer being converted into a Binary Coded Decimal
 @param s A buffer to hold the created string
*/

void itoBCD(int num, char* buffer);

#endif
```

```
void itoBCD(int num, char* buffer) {

    // Variables used
  int numLeft = num;
  int power = 0;
  char BCDval[4];
  int tenToThe;
  int numTo;

    //Gets highest power of ten
  while (numLeft > 0) {
    numLeft = (int)(numLeft / 10);
    power++;
  }
  numLeft = num;

    /* This for loop is used to convert an integer into binary coded decimal
    *  Determines the digit in each place starting with the highest power of ten
    *  in the number. When a digit is found, the correct binary coded decimal value
    *  is selected and input into a string
    */
  for (int count = 0; count < power; count++) {
    numTo = numLeft;
    tenToThe = 1;
    for (int j = 0; j < power-count-1; j++) {
      tenToThe = tenToThe * 10;
    }
    numLeft = numLeft % tenToThe;
    numTo = (numTo - numLeft)/tenToThe;
    // Switch-Case determines what BCD value based on digit value
    if (numTo == 1) {
      BCDval[0] = '0';
      BCDval[1] = '0';
      BCDval[2] = '0';
      BCDval[3] = '1';
    }
    else if (numTo == 2) {
      BCDval[0] = '0';
      BCDval[1] = '0';
      BCDval[2] = '1';
      BCDval[3] = '0';
    }
    else if (numTo == 3) {
      BCDval[0] = '0';
      BCDval[1] = '0';
      BCDval[2] = '1';
      BCDval[3] = '1';
    }
    else if (numTo == 4) {
      BCDval[0] = '0';
      BCDval[1] = '1';
      BCDval[2] = '0';
      BCDval[3] = '0';
    }
    else if (numTo == 5) {
      BCDval[0] = '0';
      BCDval[1] = '1';
      BCDval[2] = '0';
      BCDval[3] = '1';
    }
```

```
    else if (numTo == 6) {
      BCDval[0] = '0';
      BCDval[1] = '1';
      BCDval[2] = '1';
      BCDval[3] = '0';
    }
    else if (numTo == 7) {
      BCDval[0] = '0';
      BCDval[1] = '1';
      BCDval[2] = '1';
      BCDval[3] = '1';
    }
    else if (numTo == 8) {
      BCDval[0] = '1';
      BCDval[1] = '0';
      BCDval[2] = '0';
      BCDval[3] = '0';
    }
    else if (numTo == 9) {
      BCDval[0] = '1';
      BCDval[1] = '0';
      BCDval[2] = '0';
      BCDval[3] = '1';
    }
    else if (numTo == 0){
      BCDval[0] = '0';
      BCDval[1] = '0';
      BCDval[2] = '0';
      BCDval[3] = '0';
    }

    // Adds BCD value of digit into string, appends onto end
    for (int j = 0; j < 4; j++) {
      buffer[j + 4*count] = BCDval[j];

    }
  }

  // Adds null terminator to end of string
  buffer[4 * power] = '\0';
}

```

```c
/**
 * @file pcb.h
 * @brief Process Control Block queue and stack functions.
 *
 * Defines the data structure types and functions for operating
 * the PCB queues.
 * - Ready Queue: PCBs waiting to be executed by priority.
 * - Blocked Queue: PCBs waiting on something else to happen before its ready.
 * - Suspended-Ready Queue: PCBs not in the stack but ready.
 * - Suspended-Blocked Queue: PCBs not in the stack and blocked.
 *
 */

#ifndef PCB_H
#define PCB_H
#include <stdint.h>
#define PCB_NAME_MAX_LEN 16
#define PCB_STACK_MIN_SIZE 1024

enum process_class{
    CLASS_SYSTEM = 0,
    CLASS_USER = 1
};

enum execution_state{
    STATE_READY = 0,
    STATE_RUNNING = 1,
    STATE_BLOCKED = 2
};

enum dispatch_state{
    DISPATCH_ACTIVE = 0,
    DISPATCH_SUSPENDED = 1
};

struct context{
    uint32_t gs, fs, es, ds;
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    uint32_t eip, cs, eflags;
} __attribute__((packed));

struct pcb{
    char name[PCB_NAME_MAX_LEN];
    enum process_class process_class;
    int priority; // 0 (highest) to 9
    enum execution_state execution_state;
    enum dispatch_state dispatch_state;
    void *args;  // optional pointer for process-specific data (i.e alarm)
    void *stack; // Dynamically allocated, might manually allocate based on memory management.
    struct context* contextPtr;
    struct pcb* next;
    struct pcb* prev;
};

struct pcb_queue{
    struct pcb* head;
    struct pcb* tail;
};

```

```
// Queue initialization
extern struct pcb_queue ready_queue;
extern struct pcb_queue blocked_queue;
extern struct pcb_queue suspended_ready_queue;
extern struct pcb_queue suspended_blocked_queue;

/**
 * @brief Allocates a new PCB and its stack.
 *
 * Initializes all members to zero and sets stack pointer.
 *
 * @return Pointer to the allocated PCB, or NULL if allocation fails.
 */
struct pcb* pcb_allocate(void);

/**
 * @brief Frees a previously allocated PCB and its stack.
 *
 * @param ptr Pointer to the PCB to free.
 * @return 0 on success, -1 if ptr is NULL.
 */
int pcb_free(struct pcb* ptr);

/**
 * @brief Initializes a PCB with given name, class, and priority.
 *
 * Sets default execution and dispatch states.
 *
 * @param name Name of the process.
 * @param process_class Class of the process (SYSTEM/USER).
 * @param priority Initial priority (0-9).
 * @return Pointer to initialized PCB, or NULL if allocation/setup fails.
 */
struct pcb* pcb_setup(const char* name, int process_class, int priority, void
(*function)(void));

/**
 * @brief Finds a PCB by its name in all queues.
 *
 * @param name Name of the process to find.
 * @return Pointer to the PCB if found, NULL otherwise.
 */
struct pcb* pcb_find(const char* name);

/**
 * @brief Insert a PCB into the appropriate queue.
 *
 * Chooses the correct queue based on the PCB's dispatch and execution state,
 * then inserts it. If the target queue is the ready queue, PCBs are inserted
 * by priority (FIFO within same priority). Otherwise, they are appended FIFO
 * at the tail.
 *
 * @param ptr Pointer to the PCB to insert.
 */
void pcb_insert(struct pcb* ptr);

/**
 * @brief Remove a PCB from its current queue.
 *
 * Locates the PCB's queue based on its state, then detaches it by
```

```
 * fixing neighboring links. Clears the PCB's next/prev pointers.
 *
 * @param ptr Pointer to the PCB to remove.
 * @return 0 on success, -1 if PCB is NULL or not found in any queue.
 */
int pcb_remove(struct pcb* ptr);

#endif

```

```c
#include "pcb.h"
#include <string.h>
#include <sys_req.h>
#include <memory.h>
#include <showPCB.h>

/**
 * TODO
 * Will add in error messaging at different points
 */

// Queue Initialization
struct pcb_queue ready_queue = { NULL, NULL };
struct pcb_queue blocked_queue = { NULL, NULL };
struct pcb_queue suspended_ready_queue = { NULL, NULL };
struct pcb_queue suspended_blocked_queue = { NULL, NULL };

struct pcb* pcb_allocate(void){
    struct pcb* ptr =(struct pcb*) sys_alloc_mem(sizeof(struct pcb));
    if (!ptr) return NULL;
    memset(ptr, 0, sizeof(struct pcb)); // initialize to 0
    ptr->stack = sys_alloc_mem(PCB_STACK_MIN_SIZE);

    // if allocation fails
    if(!ptr->stack){
        sys_free_mem(ptr);
        return NULL;
    }

    memset(ptr->stack, 0, PCB_STACK_MIN_SIZE);
    ptr->contextPtr = (struct context*)((uint8_t *)ptr->stack + PCB_STACK_MIN_SIZE -
sizeof(struct context)); // Downward stack movement
    ptr->next = NULL;
    ptr->prev = NULL;

    return ptr;
}

int pcb_free(struct pcb* ptr){
    if (!ptr) return -1;
    if (ptr->stack) sys_free_mem(ptr->stack);
    sys_free_mem(ptr);
    return 0;
}

struct pcb* pcb_setup(const char* name, int process_class, int priority, void
(*function)(void)){
    if (!name || strlen(name) >= PCB_NAME_MAX_LEN) return NULL;
    if (priority < 0 || priority > 9) return NULL; // Also handle in User Functions
    if (pcb_find(name)!=NULL){
        char* errorMsg = "\033[31mError: Process with name '";
        sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
        sys_req(WRITE, COM1, name, strlen(name));
        errorMsg = "' already exists\033[0m\r\n";
        sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
        return NULL;
    }
    struct pcb* ptr = pcb_allocate();
    if (!ptr) return NULL;
    strncpy(ptr->name, name, PCB_NAME_MAX_LEN - 1);
```

```
    ptr->name[PCB_NAME_MAX_LEN - 1] = '\0';
    ptr->process_class = process_class;
    ptr->priority = priority;
    ptr->execution_state = STATE_READY;
    ptr->dispatch_state = DISPATCH_ACTIVE;
    ptr->contextPtr->fs = 0x10;
    ptr->contextPtr->gs = 0x10;
    ptr->contextPtr->ds = 0x10;
    ptr->contextPtr->es = 0x10;
    ptr->contextPtr->cs = 0x08;
    ptr->contextPtr->ebp = 0;
    ptr->contextPtr->esp = (uint32_t)((uint8_t *)ptr->contextPtr + sizeof(struct
context));
    ptr->contextPtr->eflags = 0x202;
    ptr->contextPtr->eip = (uint32_t)function;
    return ptr;
}

struct pcb* pcb_find(const char* name){
    if (!name) return NULL;
    struct pcb_queue* queues[] = {
        &ready_queue,
        &blocked_queue,
        &suspended_ready_queue,
        &suspended_blocked_queue
    };
    for (int i = 0; i < 4; i++){
        struct pcb* find_ptr = queues[i]->head;
        while(find_ptr){
            if (strcmp(find_ptr->name, name) == 0){
                return find_ptr;
            }
            find_ptr = find_ptr->next;
        }
    }
    return NULL; // Not found
}

void pcb_insert(struct pcb* ptr) {
    // Safety check
    if (!ptr) return;

    struct pcb_queue* q_ptr = NULL;

    // Select correct queue based on dispatch and execution state
    if (ptr->dispatch_state == DISPATCH_SUSPENDED) {
        if (ptr->execution_state == STATE_BLOCKED) {
            q_ptr = &suspended_blocked_queue;
        } else {
            q_ptr = &suspended_ready_queue;
        }
    } else {
        if (ptr->execution_state == STATE_BLOCKED) {
            q_ptr = &blocked_queue;
        } else {
            q_ptr = &ready_queue;
        }
    }

    // Insert PCB into the chosen queue
```

```
    if (q_ptr == &ready_queue) {
        // Priority-ordered insert (FIFO within same priority)
        struct pcb* cur = q_ptr->head;
        while (cur && cur->priority <= ptr->priority) {
            cur = cur->next;
        }

        // if cur is NULL, we reached the end of the list
        if (!cur) {
            // Insert at tail if no higher-priority PCB found
            if (!q_ptr->head) {
                q_ptr->head = q_ptr->tail = ptr;
            } else {
                q_ptr->tail->next = ptr;
                ptr->prev = q_ptr->tail;
                q_ptr->tail = ptr;
            }
        }
        // else insert before cur
        else {
            ptr->next = cur;
            ptr->prev = cur->prev;
            if (cur->prev) cur->prev->next = ptr;
            else q_ptr->head = ptr;
            cur->prev = ptr;
        }
    } else {
    // For non-ready queues: insert new PCB at the tail (FIFO order).

    // Case 1: The queue is empty (no head yet).
    if (!q_ptr->head) {
        // Make the new PCB both the head and the tail of the queue,
        // since it's the only element.
        q_ptr->head = q_ptr->tail = ptr;
    } else {
        // Case 2: Queue already has at least one PCB.

        // Link the current tail's 'next' pointer to the new PCB.
        q_ptr->tail->next = ptr;

        // Link the new PCB's 'prev' pointer back to the old tail.
        ptr->prev = q_ptr->tail;

        // Update the queue's tail to point to the new PCB.
        // Now 'ptr' becomes the new last element.
        q_ptr->tail = ptr;
    }
}

}

int pcb_remove(struct pcb* ptr) {
    if (!ptr) return -1;

    struct pcb_queue* q_ptr = NULL;

    // Determine which queue the PCB belongs to
    if (ptr->dispatch_state == DISPATCH_SUSPENDED) {
        if (ptr->execution_state == STATE_BLOCKED)
            q_ptr = &suspended_blocked_queue;
```

```
        else
            q_ptr = &suspended_ready_queue;
    } else {
        if (ptr->execution_state == STATE_BLOCKED)
            q_ptr = &blocked_queue;
        else
            q_ptr = &ready_queue;
    }

    // Nothing to remove if queue is empty
    if (!q_ptr || !q_ptr->head) return -1;

    // Relink neighbors to bypass 'ptr'
    if (ptr->prev) ptr->prev->next = ptr->next;
    else q_ptr->head = ptr->next;

    if (ptr->next) ptr->next->prev = ptr->prev;
    else q_ptr->tail = ptr->prev;

    // Clear links
    ptr->next = ptr->prev = NULL;

    return 0;
}

```

```
bits 32
global serial_isr

extern serial_interrupt     ; void serial_interrupt(void);

serial_isr:
    ; Save general-purpose registers first
    pusha

    ; Save segment registers
    push ds
    push es
    push fs
    push gs

    ; Ensure we have the kernel data segment loaded
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; Call the C-level serial interrupt handler
    call serial_interrupt

    ; Restore segment registers
    pop gs
    pop fs
    pop es
    pop ds

    ; Restore general-purpose registers
    popa

    iret

```

```c
#ifndef MPX_INTERRUPTS_H
#define MPX_INTERRUPTS_H

#include <stdint.h>
#include <stddef.h>
#include <sys_req.h>
#include <stdlib.h>

#include <mpx/device.h>

#define RING_BUFFER_SIZE 100

/**
 * @file mpx/interrupts.h
 * @brief Kernel functions related to software and hardware interrupts
 */

/** Disable interrupts */
#define cli() __asm__ volatile ("cli")

/** Enable interrupts */
#define sti() __asm__ volatile ("sti")

/* Core interrupt setup */
void irq_init(void);
void pic_init(void);
void idt_init(void);
void idt_install(int vector, void (*handler)(void *));

/* ---------- Device / I/O state enums ---------- */

typedef enum {
    DEV_CLOSED = 0,
    DEV_IDLE,
    DEV_READING,
    DEV_WRITING
} dev_state_t;

typedef enum {
    open   = 1,
    closed = 0
} open_status;


typedef enum {
    allocated   = 1,
    unallocated = 0
} allocation_status;

typedef enum {
    no_event        = 0,
    unhandled_event = 1
} event_flag;

typedef enum {
    IO_READ = 0,
    IO_WRITE
} io_op_t;

/* Forward declarations */
```

```
struct pcb;
struct dcb;

/**
 * @struct iocb
 * @brief I/O Control Block for one read or write request.
 */
typedef struct iocb {
    struct pcb *proc;
    struct dcb *dev;
    io_op_t     op;

    char       *buf;
    size_t      size;
    size_t      byte_count;

    struct iocb *next;
} iocb_t;

/**
 * @struct dcb
 * @brief Device Control Block for a serial port.
 */
typedef struct dcb {
    device            device;
    dev_state_t       state;
    open_status       open_status;
    allocation_status allocation_status;
    event_flag        event_flag;

    /* Ring buffer for inputs when no read is active */
    char   ring_buffer[RING_BUFFER_SIZE];
    size_t ring_head;
    size_t ring_tail;
    size_t ring_count;

    /* I/O queue for this device */
    iocb_t *q_head;
    iocb_t *q_tail;
    iocb_t *current;

    uint16_t base_port;
} dcb_t;


/**
 * @brief Get the DCB for a given serial device.
 */
dcb_t *get_dcb_for_device(device dev);

/**
 * @brief Open a serial device for interrupt-driven I/O.
 *
 * Error codes follow the R6 notes:
 *  -100 invalid device
 *  -101 invalid event flag pointer (not used here, but reserved)
 *  -102 invalid baud (no divisor)
 *  -103 already open
 */
int serial_open(device dev, int speed);
```

```
121
122 /**
123  * @brief Close a serial device.
124  *
125  * Error codes (from notes):
126  *  -201 device not open
127  */
128 int serial_close(device dev);
129
130 /**
131  * @brief Begin an interrupt-driven read on dev.
132  *
133  * This does NOT block. It:
134  *  - assumes dcb->current points to the IOCB for this op
135  *  - initializes state/event_flag
136  *  - copies any pre-typed chars from the ring buffer
137  *  - returns 0 on success, or negative error.
138  */
139 int serial_read(device dev, char *buf, size_t len);
140
141 /**
142  * @brief Begin an interrupt-driven write on dev.
143  *
144  * This does NOT block. It:
145  *  - assumes dcb->current points to the IOCB for this op
146  *  - writes first byte to THR
147  *  - enables THR empty interrupts
148  *  - returns 0 on success, or negative error.
149  */
150 int serial_write(device dev, char *buf, size_t len);
151
152 /**
153  * @brief Serial ISR
154  */
155 void serial_interrupt(void);
156
157 #endif
158
```

```c
#include <string.h>
#include <mpx/io.h>
#include <mpx/serial.h>
#include <mpx/interrupts.h>
#include <memory.h>
#include <itoa.h>

extern void serial_isr(void *);

/************ PIC + UART DEFINES ************/

#define PIC_CMD  0x20
#define PIC_MASK 0x21
#define PIC_EOI  0x20

// UART register offsets (must match serial.c)
enum uart_registers {
    RBR = 0,  // Receive Buffer
    THR = 0,  // Transmitter Holding
    DLL = 0,  // Divisor Latch LSB
    IER = 1,  // Interrupt Enable
    DLM = 1,  // Divisor Latch MSB
    IIR = 2,  // Interrupt Identification
    FCR = 2,  // FIFO Control
    LCR = 3,  // Line Control
    MCR = 4,  // Modem Control
    LSR = 5,  // Line Status
    MSR = 6,  // Modem Status
    SCR = 7,  // Scratch
};

/************ DCBs PER DEVICE ************/

/* Standard PC base ports */
#define COM1_BASE 0x3F8
#define COM2_BASE 0x2F8
#define COM3_BASE 0x3E8
#define COM4_BASE 0x2E8

static dcb_t com1_dcb = { .device = COM1, .state = DEV_CLOSED, .open_status = closed, .allocation_status = unallocated, .event_flag = no_event, .base_port = COM1_BASE };
static dcb_t com2_dcb = { .device = COM2, .state = DEV_CLOSED, .open_status = closed, .allocation_status = unallocated, .event_flag = no_event, .base_port = COM2_BASE };
static dcb_t com3_dcb = { .device = COM3, .state = DEV_CLOSED, .open_status = closed, .allocation_status = unallocated, .event_flag = no_event, .base_port = COM3_BASE };
static dcb_t com4_dcb = { .device = COM4, .state = DEV_CLOSED, .open_status = closed, .allocation_status = unallocated, .event_flag = no_event, .base_port = COM4_BASE };

dcb_t *get_dcb_for_device(device dev)
{
    switch (dev) {
    case COM1: return &com1_dcb;
    case COM2: return &com2_dcb;
    case COM3: return &com3_dcb;
    case COM4: return &com4_dcb;
    default:   return NULL;
```

```
    }
}

static int irq_for_device(device dev)
{
    switch (dev) {
    case COM1: return 4;
    case COM2: return 3;
    case COM3: return 4;   // typical PC mapping
    case COM4: return 3;
    default:   return -1;
    }
}

/************ PIC HELPERS ************/

static void pic_mask_irq(int irq)
{
    if (irq < 0) return;
    cli();
    uint8_t mask = inb(PIC_MASK);
    mask |= (1 << irq);     // 1 = disable
    outb(PIC_MASK, mask);
    sti();
}

static void pic_unmask_irq(int irq)
{
    if (irq < 0) return;
    cli();

    /* debug prints removed */
    uint8_t mask = inb(PIC_MASK);
    mask &= ~(1 << irq);    // 0 = enable
    outb(PIC_MASK, mask);
    sti();
}

/************ RING BUFFER HELPERS ************/

static void ring_put(dcb_t *dcb, char c)
{
    if (dcb->ring_count >= RING_BUFFER_SIZE) {
        // Overflow: drop char (simple policy)
        return;
    }
    dcb->ring_buffer[dcb->ring_head] = c;
    dcb->ring_head = (dcb->ring_head + 1) % RING_BUFFER_SIZE;
    dcb->ring_count++;
}

static int ring_get(dcb_t *dcb, char *out)
{
    if (dcb->ring_count == 0) {
        return 0;
    }
    *out = dcb->ring_buffer[dcb->ring_tail];
    dcb->ring_tail = (dcb->ring_tail + 1) % RING_BUFFER_SIZE;
    dcb->ring_count--;
    return 1;
```

```
}


/************ OPEN / CLOSE ************/

int serial_open(device dev, int speed)
{
    dcb_t *dcb = get_dcb_for_device(dev);
    if (!dcb) {
        return -100;  // invalid device
    }

    int      irq  = irq_for_device(dev);
    uint16_t base = dcb->base_port;

    // Already open?
    if (dcb->open_status == open) {
        return -103;
    }

    // Baud divisor must divide 115200
    if (speed <= 0 || (115200 % speed) != 0) {
        return -102;
    }

    // Disable UART interrupts
    outb(base + IER, 0x00);

    // Set baud rate
    uint16_t divisor      = (uint16_t)(115200 / speed);
    uint8_t  divisor_low  = divisor & 0x00FF;
    uint8_t  divisor_high = (divisor >> 8) & 0x00FF;

    outb(base + LCR, 0x80);          // DLAB = 1
    outb(base + DLL, divisor_low);
    outb(base + DLM, divisor_high);
    outb(base + LCR, 0x03);          // 8 bits, no parity, 1 stop, DLAB=0

    // enable interrupts
    outb(base + MCR, 0x08);

    // Clear pending status
    (void)inb(base + LSR);
    (void)inb(base + RBR);
    (void)inb(base + IIR);
    (void)inb(base + MSR);

    // Initialize DCB
    memset(dcb->ring_buffer, 0, sizeof(dcb->ring_buffer));
    dcb->ring_head         = 0;
    dcb->ring_tail         = 0;
    dcb->ring_count        = 0;
    dcb->device            = dev;
    dcb->open_status       = open;
    dcb->allocation_status = allocated;
    dcb->state             = DEV_IDLE;
    dcb->event_flag        = no_event;
    dcb->q_head            = NULL;
    dcb->q_tail            = NULL;
    dcb->current           = NULL;
```

```
173
174     // Install ISR and unmask IRQ
175     if (irq >= 0) {
176         idt_install(0x20 + irq, serial_isr);
177         pic_unmask_irq(irq);
178     }
179
180     // Enable receive data available interrupts (bit 0)
181     outb(base + IER, 0x01);
182
183     return 0;
184 }
185
186 int serial_close(device dev)
187 {
188     dcb_t *dcb = get_dcb_for_device(dev);
189     if (!dcb || dcb->open_status != open) {
190         return -201; // not open
191     }
192
193     uint16_t base = dcb->base_port;
194     int      irq  = irq_for_device(dev);
195
196     // Disable UART interrupts
197     outb(base + IER, 0x00);
198
199     // Mask PIC line
200     pic_mask_irq(irq);
201
202     dcb->open_status       = closed;
203     dcb->allocation_status = unallocated;
204     dcb->state             = DEV_CLOSED;
205     dcb->event_flag        = no_event;
206     dcb->ring_count        = 0;
207     dcb->q_head            = NULL;
208     dcb->q_tail            = NULL;
209     dcb->current           = NULL;
210
211     return 0;
212 }
213
214 /************ INPUT / OUTPUT INTERRUPT HELPERS ************/
215
216 static void serial_input_interrupt(dcb_t *dcb)
217 {
218     uint16_t base = dcb->base_port;
219
220     while (inb(base + LSR) & 0x01) {    // Data ready
221         char c = inb(base + RBR);
222
223         // Echo back the received character
224         outb(base + THR, c);
225
226         if (dcb->state == DEV_READING && dcb->current != NULL) {
227             iocb_t *io = dcb->current;
228
229             if (io->byte_count < io->size - 1) {
230                 io->buf[io->byte_count++] = c;
231             }
232
```

```
            // Stop on CR/LF or full buffer
            if (c == '\r' || c == '\n' || io->byte_count >= io->size - 1) {
                // Remove CR/LF from stored buffer
                if (io->byte_count > 0 &&
                    (io->buf[io->byte_count - 1] == '\r' ||
                     io->buf[io->byte_count - 1] == '\n')) {
                    io->byte_count--;
                }
                io->buf[io->byte_count] = '\0';

                dcb->state      = DEV_IDLE;
                dcb->event_flag = unhandled_event;
            }
        } else {
            // No active read – stash into ring buffer
            ring_put(dcb, c);
        }
    }
}

static void serial_output_interrupt(dcb_t *dcb)
{
    uint16_t base = dcb->base_port;
    iocb_t  *io   = dcb->current;

    if (!io || dcb->state != DEV_WRITING) {
        // No active write; interrupts are off
        uint8_t ier = inb(base + IER);
        ier &= ~0x02;
        outb(base + IER, ier);
        return;
    }

    if (io->byte_count < io->size) {
        outb(base + THR, (uint8_t)io->buf[io->byte_count++]);
    }

    if (io->byte_count >= io->size) {
        uint8_t ier = inb(base + IER);
        ier &= ~0x02;
        outb(base + IER, ier);

        dcb->state      = DEV_IDLE;
        dcb->event_flag = unhandled_event;
    }
}

/************SERIAL ISR************/

void serial_interrupt(void)
{
    cli();
    device devs[] = { COM1, COM2, COM3, COM4 };

    for (int i = 0; i < 4; ++i) {
        dcb_t *dcb = get_dcb_for_device(devs[i]);
        if (!dcb || dcb->open_status != open)
            continue;

        uint16_t base = dcb->base_port;
```

```
        uint8_t  iir  = inb(base + IIR);

        // Bit 0 == 1 => no interrupt pending
        if (iir & 0x01)
            continue;

        // Bits 3:1 = interrupt reason
        switch (iir & 0x0E) {
        case 0x04:  // RX data available
        case 0x0C:  // RX timeout
            serial_input_interrupt(dcb);
            break;

        case 0x02:  // TX
            serial_output_interrupt(dcb);
            break;

        default:
            // Clear
            (void)inb(base + LSR);
            (void)inb(base + RBR);
            (void)inb(base + MSR);
            break;
        }
    }

    // End Of Interrupt to PIC
    outb(PIC_CMD, PIC_EOI);

    sti();
}

/************ Read And Write ************/

int serial_read(device dev, char *buf, size_t len)
{
    dcb_t *dcb = get_dcb_for_device(dev);
    if (!dcb || dcb->open_status != open)
        return -301;    // invalid or not open

    if (!buf || len == 0)
        return -302;    // invalid buffer/len

    if (dcb->state != DEV_IDLE && dcb->state != DEV_READING)
        return -304;    // busy

    iocb_t *io = dcb->current;
    if (!io)
        return -304;    // scheduler error: no IOCB present

    io->buf        = buf;
    io->size       = len;
    io->byte_count = 0;

    dcb->state      = DEV_READING;
    dcb->event_flag = no_event;

    // Use any chars already in the ring buffer
    char c;
    while (io->byte_count < io->size - 1 && ring_get(dcb, &c)) {
```

```
        if (c == '\r' || c == '\n') {
            // End of line; don't store CR/LF
            io->buf[io->byte_count] = '\0';
            dcb->state              = DEV_IDLE;
            dcb->event_flag         = unhandled_event;
            return 0;
        }
        io->buf[io->byte_count++] = c;
    }

    // If buffer filled before newline, complete immediately
    if (io->byte_count >= io->size - 1) {
        io->buf[io->byte_count] = '\0';
        dcb->state              = DEV_IDLE;
        dcb->event_flag         = unhandled_event;
    }

    return 0;
}

int serial_write(device dev, char *buf, size_t len)
{
    dcb_t *dcb = get_dcb_for_device(dev);
    if (!dcb || dcb->open_status != open)
        return -401;                // invalid or not open

    if (!buf || len == 0)
        return -402;                // invalid buffer/len

    // Device must be idle or already writing (io_schedule sets this up)
    if (dcb->state != DEV_IDLE && dcb->state != DEV_WRITING)
        return -404;                // busy

    // I/O scheduler must have set up current IOCB
    iocb_t *io = dcb->current;
    if (!io)
        return -404;                // scheduler error: no IOCB present

    // Initialize IOCB for this write
    io->buf        = buf;
    io->size       = len;
    io->byte_count = 0;

    dcb->state      = DEV_WRITING;
    dcb->event_flag = no_event;

    uint16_t base = dcb->base_port;

    /*
     * Kick out the FIRST byte only (per R6 spec).
     * Do NOT spin in a while-loop forever; just send if THR is empty.
     * If it's not empty yet, the next TX interrupt will pick it up.
     */
    uint8_t lsr = inb(base + LSR);
    if ((lsr & 0x20) != 0 && io->size > 0) {
        outb(base + THR, (uint8_t)io->buf[0]);
        io->byte_count = 1;
    }

    /* Enable transmit holding register empty interrupts (bit 1 in IER) */
```

```
    uint8_t ier = inb(base + IER);
    ier |= 0x02;                    // enable TX interrupt
    outb(base + IER, ier);

    // Asynchronous: ISR will finish the transfer and set event_flag.
    return 0;
}

```

```c
#ifndef MPX_SERIAL_H
#define MPX_SERIAL_H

#include <stddef.h>
#include <mpx/device.h>

/**
 @file mpx/serial.h
 @brief Kernel functions and constants for handling serial I/O
*/

/**
 * @brief Performs trimming and validation on a command before adding to history.
 * @param buffer The command string to clean
 * @return 0 on success, -1 if empty/invalid
 */
int history_errors(char *buffer);

/**
 * @brief Adds a command to the history buffer and performs shifting when the buffer is full.
 * @param buffer The command to add
 * @return 0 always
 */
int history_handler(char *buffer);

/**
 * @brief Initializes devices for user input and output
 * @param device A serial port to initialize (COM1, COM2, COM3, or COM4)
 * @return 0 on success, non-zero on failure
 */
int serial_init(device dev);

/**
 * @brief Writes a buffer to a serial port
 * @param device The serial port to output to
 * @param buffer A pointer to an array of characters to output
 * @param len The number of bytes to write
 * @return The number of bytes written
 */
int serial_out(device dev, const char *buffer, size_t len);

/**
 * @brief Reads a string from a serial port
 * @param device The serial port to read data from
 * @param buffer A buffer to write data into as it is read from the serial port
 * @param count The maximum number of bytes to read
 * @return The number of bytes read on success, a negative number on failure
 */
int serial_poll(device dev, char *buffer, size_t len);

int serial_open(device dev, int speed);
int serial_close(device dev);
int serial_write(device dev, char *buf, size_t len);
int serial_read(device dev, char *buf, size_t len);

#endif

```

```c
#include <mpx/io.h>
#include <mpx/serial.h>
#include <sys_req.h>
#include <string.h>
#include <memory.h>
#include <mpx/interrupts.h>

////////// DEFINE VARIABLES ///////////////

#define HISTORY_SIZE 5
#define MAX_CMD_LEN 128

static char command_history[HISTORY_SIZE][MAX_CMD_LEN];
static int history_count = 0;
static int history_index = 0;

enum uart_registers {
  RBR = 0,  // Receive Buffer
  THR = 0,  // Transmitter Holding
  DLL = 0,  // Divisor Latch LSB
  IER = 1,  // Interrupt Enable
  DLM = 1,  // Divisor Latch MSB
  IIR = 2,  // Interrupt Identification
  FCR = 2,  // FIFO Control
  LCR = 3,  // Line Control
  MCR = 4,  // Modem Control
  LSR = 5,  // Line Status
  MSR = 6,  // Modem Status
  SCR = 7,  // Scratch
};

static int initialized[4] = { 0 };

////////// HISTORY FUNCTIONS ///////////////

int history_errors(char *buffer) {
  // ensure buffer is not empty
  if (buffer == NULL || strlen(buffer) == 0) {
    return -1;
  }

  // trim leading spaces
  size_t index = 0;
  while (buffer[index] == ' ' && index < strlen(buffer)) {
    index++;
  }
  if (index > 0) {
    size_t j = 0;
    while (buffer[index + j] != '\0' && (index + j) < strlen(buffer)) {
      buffer[j] = buffer[index + j];
      j++;
    }
    buffer[j] = '\0';
  }

  // trim trailing spaces
  size_t end = strlen(buffer);
  while (end > 0 && buffer[end - 1] == ' ') {
    end--;
  }
```

```
  buffer[end] = '\0';

  // ignore if buffer is now empty after trimming
  if (strlen(buffer) == 0) {
    return -1;
  }

  return 0;
}

int history_handler(char *buffer) {
  // handle input based on array size
  if (history_count < HISTORY_SIZE) {
    strncpy(command_history[history_count], buffer, MAX_CMD_LEN - 1);
    command_history[history_count][MAX_CMD_LEN - 1] = '\0';
    history_count++;
  }

  // buffer is full
  else {
    for (int i = 1; i < HISTORY_SIZE; i++) {
      strncpy(command_history[i - 1], command_history[i], MAX_CMD_LEN - 1);
      command_history[i - 1][MAX_CMD_LEN - 1] = '\0';
    }
    strncpy(command_history[HISTORY_SIZE - 1], buffer, MAX_CMD_LEN - 1);
    command_history[HISTORY_SIZE - 1][MAX_CMD_LEN - 1] = '\0';
  }

  history_index = history_count;

  return 0;
}



////////// SERIAL FUNCTIONS ///////////////

static int serial_devno(device dev)
{
  switch (dev) {
  case COM1: return 0;
  case COM2: return 1;
  case COM3: return 2;
  case COM4: return 3;
  }
  return -1;
}

int serial_init(device dev)
{
  int dno = serial_devno(dev);
  if (dno == -1) {
    return -1;
  }
  outb(dev + IER, 0x00);  //disable interrupts
  outb(dev + LCR, 0x80);  //set line control register
  outb(dev + DLL, 115200 / 9600); //set bsd least sig bit
  outb(dev + DLM, 0x00);  //brd most significant bit
  outb(dev + LCR, 0x03);  //lock divisor; 8bits, no parity, one stop
  outb(dev + FCR, 0xC7);  //enable fifo, clear, 14byte threshold
```

```
  outb(dev + MCR, 0x0B);  //enable interrupts, rts/dsr set
  (void)inb(dev);   //read bit to reset port
  initialized[dno] = 1;
  return 0;
}

int serial_out(device dev, const char *buffer, size_t len)
{
  int dno = serial_devno(dev);
  if (dno == -1 || initialized[dno] == 0) {
    return -1;
  }
  for (size_t i = 0; i < len; i++) {
    outb(dev, buffer[i]);
  }
  return (int)len;
}


int serial_poll(device dev, char *buffer, size_t len)
{
  // This function must properly handle the following based off ASCII:
  // - alphanumerics (a-z, A-Z, 0-9) and regular special keys (/!#$)
  // - space, backspace, delete, arrow keys (up down left right)
  // - carriage returns (\r) and new lines (\n)

  // define some ASCII characters
  const char ESC_KEY = 27;
  const char BACKSPACE = 8;
  const char DELETE = 127;

  // format the serial terminal to look like a penguin
  sys_req(WRITE, COM1, "( ')> ", 6);

  // read the input while the buffer is not full
  int index = 0;
  char input;
  char specialKey;
  int endOfLine = 0;
  while (index < (int)len - 1) {
    input = inb(dev);

    // alphanumerics, spaces, and special keys
    if ((input >= 'a' && input <= 'z') ||
      (input >= 'A' && input <= 'Z') ||
      (input >= '0' && input <= '9') ||
      (input >= '!' && input <= '/') ||
      (input == ' ') || (input == ':')) {

      buffer[index] = input;
      outb(dev, input); // echo
      index++;
      endOfLine++;
    }

    // backspace and delete
    else if ((input == BACKSPACE || input == DELETE) && (index > 0)) {
      // remove the last character from the buffer
      index--;
      endOfLine--;
```

```
      buffer[index] = '\0';

      // erase the last character from the terminal
      outb(dev, '\b');
      outb(dev, ' ');
      outb(dev, '\b');
    }

    // CR and LF (user is done)
    else if (input == '\n' || input == '\r') {
      // end the line
      buffer[endOfLine] = '\0';
      outb(dev, input);

      // handle command history and reset to "empty line" position
      history_handler(buffer);
      history_index = history_count;

      return endOfLine;
    }

    // handles multi-byte characters that use the escape character
    else if (input == ESC_KEY) {
      input = inb(dev);
      while (1) {
        specialKey = inb(dev);

        // command history error checking
        if (history_index < 0) history_index = 0;
        if (history_index > history_count) history_index = history_count;

        // UP arrow
        if (specialKey == 'A' && history_count > 0) {
          // load the last command
          if (history_index > 0 && history_index < HISTORY_SIZE) history_index--;

          // move the cursor to the end and clear the line
          while (index < endOfLine) {
            sys_req(WRITE, COM1, "\033[C", 3);
            index++;
          }
          for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);

          // write the last command and set new index and endOfLine
          strncpy(buffer, command_history[history_index], MAX_CMD_LEN - 1);
          buffer[MAX_CMD_LEN - 1] = '\0';
          sys_req(WRITE, COM1, buffer, strlen(buffer));
          endOfLine = strlen(buffer);
          index = endOfLine;

          break;
        }

        // DOWN arrow
        else if (specialKey == 'B') {
          // default behavior for command history
          if (history_index < history_count - 1) {
            // load next command
            history_index++;

```

```
            // move the cursor to the end and clear the line
            while (index < endOfLine) {
              sys_req(WRITE, COM1, "\033[C", 3);
              index++;
            }
            for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);

            // write the next command and set new index and endOfLine
            strncpy(buffer, command_history[history_index], MAX_CMD_LEN - 1);
            buffer[MAX_CMD_LEN - 1] = '\0';
            sys_req(WRITE, COM1, buffer, strlen(buffer));
            endOfLine = strlen(buffer);
            index = endOfLine;
          }
          // clear the line when you get to the end of the history
          else if (history_index == history_count - 1) {
            // clear line (move to "new" command position)
            history_index = history_count;
            while (index < endOfLine) {
              sys_req(WRITE, COM1, "\033[C", 3);  // move cursor right
              index++;
            }
            for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);

            // reset the buffer, index, and endOfLine
            buffer[0] = '\0';
            endOfLine = 0;
            index = 0;
          }

          break;
        }

        // RIGHT arrow (until you get to the end of the characters)
        else if (specialKey == 'C' && index < endOfLine) {
          sys_req(WRITE, COM1, "\033[C", 3);
          index++;
          break;
        }

        // LEFT arrow (until you get to the start of the characters)
        else if (specialKey == 'D' && index != 0) {
          sys_req(WRITE, COM1, "\033[D", 3);
          index--;
          break;
        }

        // DELETE key (ESC[3~)
        else if (specialKey == '3') {
          // ensure its the delete key
          while (input != '~'){
            input = inb(dev);
          }

          // delete until the last character
          if (index < (endOfLine-1)) {
            for (int i = index; i < endOfLine - 1; i++) {
              buffer[i] = buffer[i + 1];
              outb(dev, buffer[i]);
            }
```

```
            outb(dev, ' ');
            for (int i = index; i < endOfLine; i++){
              sys_req(WRITE, COM1, "\033[D", 3);
            }
            endOfLine--;
          }
          // delete the last character
          else if (index == (endOfLine-1)){
            buffer[index] = '\0';
            sys_req(WRITE, COM1, " \033[D", 4);
            endOfLine--;
          }
          else {
            // ignore sequence
          }
          break;
        }
      }
    } // end of multi byte chars

  }

  // returns -1 if buffer overflow or when there is an error
  buffer[index] = '\0';
  return -1;
}

```

```
bits 32
global sys_call_isr

;;; System call interrupt handler. To be implemented in Module R3.
extern sys_call     ; The C function that sys_call_isr will call
sys_call_isr:
  pusha       ; Pushes general purpose registers

  push ds       ; Pushes remaining registers
  push es
  push fs
  push gs

  mov eax, esp
  push eax
  call sys_call

  mov esp, eax

  pop gs        ; Pops last 4 registers pushed
  pop fs
  pop es
  pop ds

  popa        ; Pops general purpose registers

  iret

```

```h
#ifndef SYS_CALL_H
#define SYS_CALL_H
#include <pcb.h>
#include <sys_req.h>

/**
 * @file sys_call.h
 * @brief Handles context switching when sys_req function is used
 * @param curContext Pointer to the current context
 */
struct context *sys_call(struct context *curContext);

/**
 * @brief Helper that returns the current process
 */
struct pcb* sys_get_current_process(void);

#endif

```

```c
#include "comhand.h"
#include <string.h>
#include <itoa.h>
#include <sys_req.h>
#include "help.h"
#include "exit.h"
#include "version.h"
#include "clock.h"
#include "showPCB.h"
#include "setPriority.h"
#include "ready.h"
#include "init.h"
#include "block.h"
#include "yield.h"
#include <mpx/interrupts.h>
#include <mpx/serial.h>
#include <memory.h>
#include <mpx/device.h>
#include <pcb.h>

/* Pointer for current process */
static struct pcb *curProc = NULL;

/* Pointer to system stack/kernel context */
static struct context *sysStackPtr = NULL;

/* Helper to return current process */
struct pcb *sys_get_current_process(void)
{
    return curProc;
}

/*
 * Scan all serial DCBs. For any with event_flag == unhandled_event:
 *  - take the IOCB from dcb->current
 *  - move the process from BLOCKED to READY
 *  - set its saved eax = #bytes transferred
 *  - free the IOCB
 *  - start the next IOCB in the queue, if any
 */
static void io_handle_completions(void)
{
    device devs[] = { COM1, COM2, COM3, COM4 };

    for (int i = 0; i < 4; ++i) {
        dcb_t *dcb = get_dcb_for_device(devs[i]);
        if (!dcb || dcb->open_status != open)
            continue;

        if (dcb->event_flag == unhandled_event && dcb->current != NULL) {
            iocb_t      *io = dcb->current;
            struct pcb  *p  = io->proc;

            /* Wake the blocked process */
            if (p != NULL) {
                pcb_remove(p);                  // remove from blocked queue
                p->execution_state = STATE_READY;
                pcb_insert(p);                  // insert into ready queue


```

```
                if (p->contextPtr) {
                    p->contextPtr->eax = (int)io->byte_count;
                }
            }

            /* Clear DCB */
            dcb->current    = NULL;
            dcb->state      = DEV_IDLE;
            dcb->event_flag = no_event;

            /* Grab next IOCB, if any */
            iocb_t *next = NULL;
            if (dcb->q_head) {
                next       = dcb->q_head;
                dcb->q_head = next->next;
                if (!dcb->q_head)
                    dcb->q_tail = NULL;
                next->next = NULL;
            }

            /* Free finished IOCB */
            sys_free_mem(io);

            /* Start next request, if present */
            if (next) {
                dcb->current    = next;
                dcb->state      = (next->op == IO_READ) ? DEV_READING : DEV_WRITING;
                dcb->event_flag = no_event;

                if (next->op == IO_READ) {
                    serial_read(dcb->device, next->buf, next->size);
                } else {
                    serial_write(dcb->device, next->buf, next->size);
                }
            }
        }
    }
}

/*
 * io_scheduler:
 *  - Builds an IOCB for the given request.
 *  - If device idle, starts I/O immediately (serial_read/serial_write).
 *  - If busy, enqueues the IOCB.
 *
 * Return values:
 *  < 0  : error (immediate; caller does not block)
 *  > 0  : I/O completed synchronously, return that byte count
 *  == 0 : I/O started and will complete asynchronously; caller should BLOCK.
 */
static int io_schedule(struct context *ctx)
{
    op_code op  = (op_code)ctx->eax;
    device dev  = (device)ctx->ebx;
    char  *buf  = (char *)ctx->ecx;
    size_t len  = (size_t)ctx->edx;

    dcb_t *dcb = get_dcb_for_device(dev);
    if (!dcb || dcb->open_status != open) {
        return (op == READ) ? -301 : -401;
```

```
    }

    if (!buf || len == 0) {
        return (op == READ) ? -302 : -402;
    }

    /* Allocate IOCB */
    iocb_t *io = (iocb_t *)sys_alloc_mem(sizeof(iocb_t));
    if (!io) {
        return -1;  // out of memory
    }

    memset(io, 0, sizeof(iocb_t));
    io->proc       = curProc;
    io->dev        = dcb;
    io->buf        = buf;
    io->size       = len;
    io->byte_count = 0;
    io->next       = NULL;

    if (op == READ) {
        io->op = IO_READ;
    } else if (op == WRITE) {
        io->op = IO_WRITE;
    } else {
        sys_free_mem(io);
        return -1;
    }

    /* If device idle, start immediately; else enqueue */
    if (dcb->current == NULL && dcb->state == DEV_IDLE) {
        dcb->current    = io;
        dcb->event_flag = no_event;
        dcb->state      = (io->op == IO_READ) ? DEV_READING : DEV_WRITING;

        int rc;
        if (io->op == IO_READ)
            rc = serial_read(dcb->device, buf, len);
        else
            rc = serial_write(dcb->device, buf, len);

        if (rc < 0) {
            dcb->current = NULL;
            dcb->state   = DEV_IDLE;
            sys_free_mem(io);
            return rc;
        }

        /* If completed immediately */
        if (dcb->event_flag == unhandled_event) {
            int done = (int)io->byte_count;
            sys_free_mem(io);
            dcb->current    = NULL;
            dcb->state      = DEV_IDLE;
            dcb->event_flag = no_event;
            return done;
        }

        /* Otherwise, I/O is in progress; caller must block. */
    } else {
```

```
        /* Device busy: append to queue */
        if (dcb->q_tail)
            dcb->q_tail->next = io;
        else
            dcb->q_head = io;
        dcb->q_tail = io;
    }

    return 0;   // started, will complete asynchronously
}

struct context *sys_call(struct context *curContext)
{
    /* Save kernel context */
    if (curProc == NULL) {
        sysStackPtr = curContext;
    }

    /* First: handle any completed I/O */
    io_handle_completions();

    /* Next ready process */
    struct pcb *newProc = ready_queue.head;

    /* IDLE: */
    if (curContext->eax == IDLE) {
        if (newProc == NULL) {
            /* No other process ready; just return to current process */
            if (curProc != NULL) {
                curContext->eax = -1;
                return curContext;  // back to same process
            } else {
                /* No processes at all; return to kernel */
                curContext->eax = -1;
                return curContext;
            }
        } else {
            if (curProc == NULL) {
                pcb_remove(newProc);
                curProc          = newProc;
                curContext->eax  = -1;
                return newProc->contextPtr;
            } else {
                curProc->contextPtr = curContext;
                pcb_insert(curProc);
                pcb_remove(newProc);
                curProc          = newProc;
                curContext->eax  = -1;
                return newProc->contextPtr;
            }
        }
    }

    /* EXIT: */
    else if (curContext->eax == EXIT) {
        if (newProc == NULL) {
            pcb_free(curProc);
            curProc          = NULL;
            sysStackPtr->eax = -1;
            return sysStackPtr;
```

```
        } else {
            pcb_free(curProc);
            pcb_remove(newProc);
            curProc    = newProc;
            curContext = curProc->contextPtr;
            curContext->eax = -1;
            return newProc->contextPtr;
        }
    }

    /* READ or WRITE */
    else if (curContext->eax == READ || curContext->eax == WRITE) {
        int rc = io_schedule(curContext);

        /* Immediate error: return to caller with error code */
        if (rc < 0) {
            curContext->eax = rc;
            return curContext;
        }

        /* Synchronous completion: rc is byte count */
        if (rc > 0) {
            curContext->eax = rc;
            return curContext;
        }

        /*
         * Asynchronous I/O started:
         *  - Mark current process BLOCKED
         *  - Save its context
         *  - Put it in blocked queue
         *  - Dispatch next READY process or return to kernel
         */
        if (curProc != NULL) {

            /* Remove from whatever queue it is currently in */
            pcb_remove(curProc);

            /* Mark blocked */
            curProc->execution_state = STATE_BLOCKED;
            curProc->contextPtr      = curContext;

            /* Insert into blocked queue properly */
            pcb_insert(curProc);
        }


        newProc = ready_queue.head;

        if (newProc == NULL) {
            // No other process ready then go idle
            return curContext;
        }
        else {
            pcb_remove(newProc);
            curProc = newProc;
            return newProc->contextPtr;
        }
    }

```

```
    /* Any other opcode */
    else {
        curContext->eax = -1;
        return curContext;
    }
}

```

```c
#ifndef MPX_VM_H
#define MPX_VM_H
#include <stddef.h>

/**
 @file mpx/vm.h
 @brief Kernel functions for virtual memory and primitive allocation
*/

/**
 * @brief Memory Control Block (MCB) structure.
 */
struct mcb{
    int startAddr;
    int size;
    struct mcb* next;
    struct mcb* prev;
    // Only needed if doing single list
    // enum alloc_stat alloc_stat;
};

struct mcb_list{
    struct mcb* head;
    struct mcb* tail;
};

extern struct mcb_list mem_allocated_list;
extern struct mcb_list mem_free_list;

/**
 Allocates memory from a primitive heap.
 @param size The size of memory to allocate
 @param align If non-zero, align the allocation to a page boundary
 @param phys_addr If non-NULL, a pointer to a pointer that will
                  hold the physical address of the new memory
 @return The newly allocated memory
 */
void *kmalloc(size_t size, int align, void **phys_addr);

/**
 Initializes the kernel page directory and initial kernel heap area.
 Performs identity mapping of the kernel frames such that the virtual
 addresses are equivalent to the physical addresses.
*/
void vm_init(void);

/**
 * @brief Initializes the heap with a given size.
 * @param size The total size of the heap to initialize.
 */
void initialize_heap(size_t size);

/**
 * @brief Allocates a block of memory of the requested size.
 * @param req_size The size of memory to allocate.
 * @return Pointer to the allocated memory, or NULL on failure.
 */
void *allocate_memory(size_t req_size);

/**
```

```
 * @brief Frees a previously allocated block of memory.
 * @param ptr Pointer to the memory block to free.
 * @return 0 on success, -1 on failure.
 */
int free_memory(void *ptr);

#endif

```

```c
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <mpx/vm.h>
#include <mpx/serial.h>
#include <sys_req.h>

struct mcb_list mem_allocated_list = { NULL, NULL };
struct mcb_list mem_free_list = { NULL, NULL };

void initialize_heap(size_t size){
    void *addr = kmalloc(size, 0, NULL);
    if (!addr) return;

    // Place the initial MCB at the start of the returned region.
    struct mcb* initial_mcb = (struct mcb*)addr;

    // Data address starts immediately after the MCB
    initial_mcb->startAddr = (int)((uintptr_t)addr + sizeof(struct mcb));

    // Usable size is the total minus the MCB header (16 bytes)
    initial_mcb->size = (int)(size - sizeof(struct mcb));
    initial_mcb->next = NULL;
    initial_mcb->prev = NULL;

    // Start lists as free
    mem_free_list.head = mem_free_list.tail = initial_mcb;
    mem_allocated_list.head = mem_allocated_list.tail = NULL;
}

static void unlink_mcb(struct mcb_list *list, struct mcb *node){
    if (!list || !node) return;
    if (node->prev) node->prev->next = node->next;
    else list->head = node->next;

    if (node->next) node->next->prev = node->prev;
    else list->tail = node->prev;
    // Remove MCB from a list
    node->next = node->prev = NULL;
}

static void append_mcb(struct mcb_list *list, struct mcb *node){
    if (!list || !node) return;
    node->next = NULL;
    node->prev = list->tail;
    if (list->tail) list->tail->next = node;
    else list->head = node;
    // Add MCB to tail of a list
    list->tail = node;
}

void *allocate_memory(size_t req_size){
    if (req_size == 0) return NULL;

    // First Fit - Look for the first block that fits
    struct mcb *cur = mem_free_list.head;
    while (cur && (size_t)cur->size < req_size) cur = cur->next;
    if (!cur) return NULL;

    // If block is large enough, split it
```

```
    size_t remaining = (size_t)cur->size - req_size;
    if (remaining > sizeof(struct mcb)){

        // Create MCB for remaining free block
        uintptr_t alloc_data_addr = (uintptr_t)cur->startAddr;
        uintptr_t new_mcb_addr = alloc_data_addr + req_size;
        struct mcb *new_mcb = (struct mcb *)new_mcb_addr;

        // Set up the Free MCB
        new_mcb->startAddr = (int)(new_mcb_addr + sizeof(struct mcb));
        new_mcb->size = (int)(remaining - sizeof(struct mcb));

        // Replace cur in free list with the new MCB
        new_mcb->prev = cur->prev;
        new_mcb->next = cur->next;
        if (cur->prev) cur->prev->next = new_mcb;
        else mem_free_list.head = new_mcb;
        if (cur->next) cur->next->prev = new_mcb;
        else mem_free_list.tail = new_mcb;

        // Shrink cur to allocated size
        cur->size = (int)req_size;
        cur->next = cur->prev = NULL;

        // Add cur to allocated list
        append_mcb(&mem_allocated_list, cur);

        // Set pointer to unallocated space
        return (void *) (uintptr_t) cur->startAddr;
    } else {
        // If not enough room then allocate the whole block
        // Remove from free list
        unlink_mcb(&mem_free_list, cur);

        // Add to allocated list
        append_mcb(&mem_allocated_list, cur);

        return (void *) (uintptr_t) cur->startAddr;
    }
}

int free_memory(void *ptr){
    if (!ptr) return -1;

    // Convert user pointer to MCB pointer
    struct mcb *m = (struct mcb *)((uintptr_t)ptr - sizeof(struct mcb));

    // Check if MCB is in allocated list
    struct mcb *cur = mem_allocated_list.head;
    while (cur && cur != m)
        cur = cur->next;

    if (!cur)
        return -1;  // Not found → invalid free()

    // If found remove from allocated list
    unlink_mcb(&mem_allocated_list, m);

    // Insert into free list in order
    struct mcb *f = mem_free_list.head;
```

```
    struct mcb *prev = NULL;

    while (f && (uintptr_t)f < (uintptr_t)m) {
        prev = f;
        f = f->next;
    }

    // Insert between two blocks
    m->prev = prev;
    m->next = f;
    if (prev) prev->next = m;
    else mem_free_list.head = m;

    if (f) f->prev = m;
    else mem_free_list.tail = m;

    // Merge with the previous block if adjacent
    if (m->prev) {
        uintptr_t prev_end = (uintptr_t)m->prev->startAddr + m->prev->size;
        uintptr_t m_addr = (uintptr_t)m->startAddr - sizeof(struct mcb);

        if (prev_end == m_addr) {
            m->prev->size += sizeof(struct mcb) + m->size;

            // After merge remove the front block (m) from the list
            if (m->next) m->next->prev = m->prev;
            else mem_free_list.tail = m->prev;

            m->prev->next = m->next;

            // After merge use prev block as the whole merged mcb
            m = m->prev;
        }
    }

    // Merge with the next block if adjacent
    if (m->next) {
        uintptr_t m_end = (uintptr_t)m->startAddr + m->size;
        uintptr_t next_addr = (uintptr_t)m->next;

        if (m_end == next_addr) {
            struct mcb *next = m->next;

            // Merge next into m
            m->size += sizeof(struct mcb) + next->size;

            // After merge remove the next block from the list
            m->next = next->next;
            if (next->next) next->next->prev = m;
            else mem_free_list.tail = m;
        }
    }

    return 0;
}
```

```h
#ifndef MCB_ALLOCATE_H
#define MCB_ALLOCATE_H

#include <stddef.h>

/**
 * @file mcb/allocate.h
 * @brief Shell command handler for allocating memory via allocate_memory() and printing the resulting address in hex.
 */

/**
 * @brief Prints help/usage information for the `allocate` command.
 */
void allocate_help(void);

/**
 * @brief Executes the allocate command by calling the kernel function.
 * @param args  The argument string passed by the shell: size between 1-50,000 or help.
 * @return 0 on success, -1 on error.
 */
int allocate_command(char *args);

#endif

```

```c
#include <mpx/serial.h>
#include <sys_req.h>
#include "mcb/allocate.h"
#include <mpx/vm.h>
#include <string.h>
#include <stdlib.h>

static void print_hex(int num) {
    // setup variables
    char hex_chars[] = "0123456789ABCDEF";
    char buf[11];   // 0x + 8 digits + '\0'
    int i;

    // create and print buffer
    buf[0] = '0';
    buf[1] = 'x';
    for (i = 0; i < 8; i++) {
        int shift = (7 - i) * 4;
        buf[2 + i] = hex_chars[(num >> shift) & 0xF];
    }
    buf[10] = '\0';

    sys_req(WRITE, COM1, buf, 10);
}

void allocate_help(void) {
    const char *helpMsg =
        "\r\n\033[33mallocate\033[0m [<\033[36msize\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33mallocate\033[0m <\033[36msize\033[0m>    allocates 1-50,000 bytes and prints its address in hex.\r\n"
        "  \033[33mallocate\033[0m \033[36mhelp\033[0m      prints this help message\r\n\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

int allocate_command(char *args) {
    // error checking and cleanup
    if (!args) return -1;
    while (*args == ' ') args++;

    // help command
    if (strncmp(args, "help", 4) == 0) {
        allocate_help();
        return 0;
    }

    // check args
    int size = atoi(args);
    if (size <= 0 || size > 50000) {
        const char *msg = "\033[31mError: invalid size.\033[0m\r\n";
        sys_req(WRITE, COM1, msg, strlen(msg));
        allocate_help();
        return -1;
    }

    // call kernel function
    void *addr = allocate_memory(size);
    if (!addr) {
        const char *err = "\033[31mAllocation failed in kernel function.\033[0m\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
```

```
        allocate_help();
        return -1;
    }

    // successfully allocated
    const char *success_msg = "\033[32mSuccessfully allocated:\033[0m ";
    sys_req(WRITE, COM1, success_msg, strlen(success_msg));
    print_hex((int)addr);
    const char *success_addr = "\r\n";
    sys_req(WRITE, COM1, success_addr, strlen(success_addr));

    return 0;
}

```

```h
#ifndef MCB_FREE_H
#define MCB_FREE_H

/**
 * @file mcb/free.h
 * @brief Shell command handler for freeing memory.
 */

/**
 * @brief Prints help/usage information for the `free` command.
 */
void free_help(void);

/**
 * @brief Executes the free command by calling the kernel function.
 * @param args String containing the hex address to free.
 * @return 0 on success, -1 on error.
 */
int free_command(char *args);

#endif

```

```c
#include <mpx/vm.h>
#include <mpx/serial.h>
#include <sys_req.h>
#include <stddef.h>
#include <stdint.h>
#include "mcb/free.h"
#include "string.h"

static void print_hex(int num) {
    // setup variables
    char hex_chars[] = "0123456789ABCDEF";
    char buf[11];   // 0x + 8 digits + '\0'
    int i;

    // create and print buffer
    buf[0] = '0';
    buf[1] = 'x';
    for (i = 0; i < 8; i++) {
        int shift = (7 - i) * 4;
        buf[2 + i] = hex_chars[(num >> shift) & 0xF];
    }
    buf[10] = '\0';

    sys_req(WRITE, COM1, buf, 10);
}

void free_help(void) {
    const char *helpMsg =
        "\r\n\033[33mfree\033[0m [<\033[36maddress\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33mfree\033[0m <\033[36maddress\033[0m>    frees the memory at the address given.\r\n"
        "  \033[33mfree\033[0m \033[36mhelp\033[0m         prints this help message.\r\n\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

int free_command(char *args)
{
    // error checking
    if (!args) {
        free_help();
        return -1;
    }

    // check args
    while (*args == ' ') args++;
    if (*args == '\0') {
        free_help();
        return -1;
    }

    // help command
    if (strncmp(args, "help", 4) == 0) {
        free_help();
        return 0;
    }

    // parse hex string (optional 0x prefix)
    int addr = 0;
    if (args[0] == '0' && (args[1] == 'x' || args[1] == 'X')) {
```

```
        args += 2;
    }
    while ((*args >= '0' && *args <= '9') || (*args >= 'a' && *args <= 'f') || (*args >= 'A' && *args <= 'F')) {
        int val = 0;
        if (*args >= '0' && *args <= '9') val = *args - '0';
        else if (*args >= 'a' && *args <= 'f') val = *args - 'a' + 10;
        else if (*args >= 'A' && *args <= 'F') val = *args - 'A' + 10;

        addr = (addr << 4) | val;
        args++;
    }

    // free memory
    if (free_memory((void *)addr) == 0) {
        const char *msg = "\033[32mSuccessfully freed:\033[0m ";
        sys_req(WRITE, COM1, msg, strlen(msg));
        print_hex(addr);
        sys_req(WRITE, COM1, "\r\n", 2);
        return 0;
    }
    else {
        const char *err = "\033[31mError: Invalid address.\033[0m\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
        free_help();
        return -1;
    }
}

```

```c
#ifndef MCB_SHOW_H
#define MCB_SHOW_H

/**
 * @file show.h
 * @brief Functions for displaying information about MCBs.
 */

/**
 * @brief Prints help information for the "show mcb" command.
 */
void show_mcb_help(void);

/**
 * @brief Handles the "show mcb" shell command.
 * @param args  User argument string: address or help.
 * @return The address parsed on success, or -1 on failure.
 */
int show_mcb_command(char *args);

/**
 * @brief Prints allocated memory blocks.
 */
void show_mcb_allocated(void);

/**
 * @brief Prints free memory blocks.
 */
void show_mcb_free(void);

/**
 * @brief Prints all memory blocks.
 */
void show_mcb_all(void);

#endif

```

```c
#include "mcb/show.h"
#include <mpx/vm.h>
#include <mpx/serial.h>
#include <sys_req.h>
#include <string.h>

static void print_hex(int num) {
    // setup variables
    char hex_chars[] = "0123456789ABCDEF";
    char buf[11];   // 0x + 8 digits + '\0'
    int i;

    // create and print buffer
    buf[0] = '0';
    buf[1] = 'x';
    for (i = 0; i < 8; i++) {
        int shift = (7 - i) * 4;
        buf[2 + i] = hex_chars[(num >> shift) & 0xF];
    }
    buf[10] = '\0';

    sys_req(WRITE, COM1, buf, 10);
}

static void print_dec(int num) {
    // setup variables for 32-bit int
    char buf[12];
    int i = 0, j, k;
    char temp;

    if (num == 0) {
        buf[0] = '0';
        buf[1] = '\0';
        sys_req(WRITE, COM1, buf, 1);
        return;
    }

    while (num > 0 && i < 11) {
        buf[i++] = '0' + (num % 10);
        num /= 10;
    }

    // reverse the buffer
    for (j = 0, k = i - 1; j < k; j++, k--) {
        temp = buf[j];
        buf[j] = buf[k];
        buf[k] = temp;
    }
    buf[i] = '\0';
    sys_req(WRITE, COM1, buf, i);
}

void show_mcb_help(void) {
    const char *helpMsg =
        "\r\n\033[33mshow mcb\033[0m [\033[36mallocated\033[0m|\033[36mfree\033[0m|
<\033[36maddress\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33mshow mcb\033[0m              shows info about all memory.\r\n"
        "  \033[33mshow mcb\033[0m \033[36mallocated\033[0m    shows info about all
allocated memory.\r\n"
```

```
        "  \033[33mshow mcb\033[0m \033[36mfree\033[0m         shows info about all free memory.\r\n"
        "  \033[33mshow mcb\033[0m <\033[36maddress\033[0m>    shows info about the memory at the address given.\r\n"
        "  \033[33mshow mcb\033[0m \033[36mhelp\033[0m         prints this help message.\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

//Show allocated function

void show_mcb_allocated(void){
    //setup variable
    struct mcb *cur;
    int mcb_size = 0;
    int mcb_addr = 0;

    //iterate through loop
    for (cur = mem_allocated_list.head; cur; cur = cur->next) {
        //assign variables
        mcb_addr=cur->startAddr;
        mcb_size=cur->size;

        //print address
        sys_req(WRITE, COM1, "\r\nAllocated Memory\n", strlen("\r\nAllocated Memory\n"));
        sys_req(WRITE, COM1, "Start Address: ", strlen("Start Address: "));
        print_hex(mcb_addr);

        //print size
        sys_req(WRITE, COM1, "\nSize: ", strlen("\nSize: "));
        print_dec(mcb_size);
        sys_req(WRITE, COM1, "\n\n", strlen("\n\n"));
    }

}

//Show free function

void show_mcb_free(void){
    //setup variable
    struct mcb *cur;
    int mcb_size = 0;
    int mcb_addr = 0;

    //iterate through loop
    for (cur = mem_free_list.head; cur; cur = cur->next) {
        //assign variables
        mcb_addr=cur->startAddr;
        mcb_size=cur->size;

        //print address
        sys_req(WRITE, COM1, "\r\nFree Memory\n", strlen("\r\nFree Memory\n"));
        sys_req(WRITE, COM1, "Start Address: ", strlen("Start Address: "));
        print_hex(mcb_addr);

        //print size
        sys_req(WRITE, COM1, "\nSize: ", strlen("\nSize: "));
        print_dec(mcb_size);
```

```
        sys_req(WRITE, COM1, "\n\n", strlen("\n\n"));
    }
}

//Show all function

void show_mcb_all(void){
    show_mcb_allocated();
    show_mcb_free();
}

int show_mcb_command(char *args) {
    // setup variables and check args
    int addr = 0;
    int i = 0;
    if (args == NULL) return -1;

    // help command
    if (strncmp(args, "help", 4) == 0) {
        show_mcb_help();
        return 0;
    }
    else if (strncmp(args, "allocated", 9) == 0) {
        show_mcb_allocated();
        return 0;
    }
    else if (strncmp(args, "free", 4) == 0) {
        show_mcb_free();
        return 0;
    }
    else if (args == NULL || *args=='\0' || (strcmp(args, "all")==0)){
        show_mcb_all();
        return 0;
    }

    // parse hex string from args
    if (args[0] == '0' && (args[1] == 'x' || args[1] == 'X')) i = 2;

    while (args[i] != '\0') {
        char c = args[i];
        int val = 0;
        if (c >= '0' && c <= '9') val = c - '0';
        else if (c >= 'A' && c <= 'F') val = 10 + (c - 'A');
        else if (c >= 'a' && c <= 'f') val = 10 + (c - 'a');
        else break; // invalid char
        addr = (addr << 4) | val;
        i++;
    }

    // setup more variables
    struct mcb *cur;
    int found = 0;

    // check allocated list
    for (cur = mem_allocated_list.head; cur; cur = cur->next) {
        if (cur->startAddr == addr) {
            const char *header_msg = "\r\nAllocated Block(s):\r\n";
            sys_req(WRITE, COM1, header_msg, strlen(header_msg));

            const char *addr_msg = "  Start address = .";
```

```
            sys_req(WRITE, COM1, addr_msg, strlen(addr_msg));
            print_hex(cur->startAddr);

            const char *size_msg = ", Size = ";
            sys_req(WRITE, COM1, size_msg, strlen(size_msg));
            print_dec(cur->size);

            sys_req(WRITE, COM1, "\r\n", 2);
            found = 1;
        }
    }

    sys_req(WRITE, COM1, "\r\n", 2);

    // check free list if not found
    for (cur = mem_free_list.head; cur; cur = cur->next) {
        if (cur->startAddr == addr) {
            const char *header_msg = "\r\nFree Block(s):\r\n";
            sys_req(WRITE, COM1, header_msg, strlen(header_msg));

            const char *addr_msg = "  Start address = ";
            sys_req(WRITE, COM1, addr_msg, strlen(addr_msg));
            print_hex(cur->startAddr);

            const char *size_msg = ", Size = .";
            sys_req(WRITE, COM1, size_msg, strlen(size_msg));
            print_dec(cur->size);

            sys_req(WRITE, COM1, "\r\n", 2);
            found = 1;
        }
    }

    if (!found) {
        const char *err = "Error: Block not found.\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
        return -1;
    }

    return addr;
}
```

```
#ifndef BLOCK_H
#define BLOCK_H

#include <pcb.h>
#include <sys_req.h>
#include <string.h>



 /**
  * @brief Function for handling command input for blocking PCB, called by comhand
  * @param args Arguments for handling block command
  */
void block_pcb_command(const char* args);


/**
 * @brief Provides help for how to use the block command
 */
void block_help(void);


/**
 * @brief Block the desired PCB
 * @param name Argument stores the name of desired PCB
 */
void block_pcb(const char* name);


/**
 * @brief Function for handling command input for unblocking PCB, called by comhand
 * @param args Arguments for handling unblock command
 */
void unblock_pcb_command(const char* args);


/**
 * @brief Provides help for how to use the unblock command
 */
void unblock_help(void);


/**
 * @brief Unblock the desired PCB
 * @param name Argument stores the name of desired PCB
 */
void unblock_pcb(const char* name);

#endif

```

```c
#include <pcb.h>
#include <sys_req.h>
#include <string.h>
#include <stdlib.h>
#include "block.h"
#include "showPCB.h"

/////////////////////////////////////////////////////
//block function


//help message for block
void block_help(void){
    const char *helpMessage =
        "\r\n\033[33mblock\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33mblock\033[0m <\033[36mname\033[0m>      block the requested process with the given name (1-16 characters)\r\n"
        "  \033[33mblock\033[0m \033[36mhelp\033[0m        prints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}

//blocks the selected process
void block_pcb(const char* name){
    //check to see if process is already blocked
    struct pcb* p = pcb_find(name);
    if(p->execution_state==STATE_BLOCKED){
        sys_req(WRITE, COM1, "\033[31mProcess has already been blocked.\033[0m\n", 44);
    }

    // Checks if the name is appropriate length
    else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16 characters\033[0m\n", 59);
        block_help();
    }

    // Checks if the process exists
    else if (p == NULL){
        sys_req(WRITE, COM1, "\033[31mCould not find process: use 'show' for list of processes\033[0m\n", 67);
        show_pcb_help();
    }

    //block the selected process
    else{
        pcb_remove(p);
        p->execution_state=STATE_BLOCKED;
        pcb_insert(p);
        sys_req(WRITE, COM1, "\033[36mProcess blocked successfully\033[0m\n", 39);
    }
}

//function for use in comhand
void block_pcb_command(const char* args){
    //validation
    for (int i=0; i<(int)strlen(args); i++){
        if (args[i] == ' '){
```

```
            sys_req(WRITE, COM1, "\033[31mError: Please ensure a valid name is given\033[0m\n", 47);
            block_help();
            return;
        }
    }
    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is given\033[0m\n", 47);
        block_help();
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        block_help();
    }

    // Handles blocking process
    else{
            block_pcb(args);
    }
}

////////////////////////////////////////////////////
//unblock function


//help message for unblock
void unblock_help(void){
    const char *helpMessage =
        "\r\n\033[33munblock\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33munblock\033[0m <\033[36mname\033[0m>      unblock the requested process with the given name (1-16 characters)\r\n"
        "  \033[33munblock\033[0m \033[36mhelp\033[0m        prints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}

//unblocks the selected process
void unblock_pcb(const char* name){
    //check to see if process is already unblocked
    struct pcb* p = pcb_find(name);
    if(p->execution_state==STATE_READY){
        sys_req(WRITE, COM1, "\033[31mProcess has already been unblocked.\033[0m\n", 46);
    }

    // Checks if the name is appropriate length
    else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16 characters\033[0m\n", 59);
        unblock_help();
    }

    // Checks if the process exists
    else if (p == NULL){
        sys_req(WRITE, COM1, "\033[31mCould not find process: use 'show' for list of processes\033[0m\n", 67);
        show_pcb_help();
```

```
    }

    //unblock the selected process
    else{
        pcb_remove(p);
        p->execution_state=STATE_READY;
        pcb_insert(p);
        sys_req(WRITE, COM1, "\033[36mProcess unblocked successfully\033[0m\n", 41);
    }
}

//function for use in comhand
void unblock_pcb_command(const char* args){
    //validation
    for (int i=0; i<(int)strlen(args); i++){
        if (args[i] == ' '){
            sys_req(WRITE, COM1, "\033[31mError: Please ensure a valid name is
given\033[0m\n", 47);
            unblock_help();
            return;
        }
    }
    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is
given\033[0m\n", 47);
        unblock_help();
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        unblock_help();
    }

    // Handles unblocking process
    else{
            unblock_pcb(args);
    }
}

```

```c
#ifndef INIT_H
#define INIT_H

#include <pcb.h>
#include <sys_req.h>
#include <string.h>

 /**
 * @brief Function for handling command input for creating PCB, called by comhand
 * @param args Arguments for handling create command
 */
void create_pcb_command(const char* args);


/**
 * @brief Provides help for how to use the create command
 */
void create_help(void);


/**
 * @brief Create the desired PCB
 * @param name Argument stores the desired name of PCB
 * @param process_class Argument stores the desired class of PCB
 * @param priority Argument stores the desired priority of PCB
 */
void create_pcb(const char* name, int process_class, int priority);


 /**
 * @brief Function for handling command input for deleting PCB, called by comhand
 * @param args Arguments for handling delete command
 */
void delete_pcb_command(const char* args);


/**
 * @brief Provides help for how to use the delete command
 */
void delete_help(void);


/**
 * @brief Delete the desired PCB
 * @param name Argument stores the name of desired PCB
 */
void delete_pcb(const char* name);

#endif

```

```c
#include <pcb.h>
#include <sys_req.h>
#include <string.h>
#include <stdlib.h>
#include "init.h"
#include "showPCB.h"

////////////////////////////////////////////////////
//create function


//help message for create
void create_help(void){
  const char *helpMessage =
    "\r\n\033[33mcreate\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m]
[<\033[36mclass\033[0m>] [<\033[36mpriority\033[0m>]\r\n"
    "  \033[33mcreate\033[0m <\033[36mname\033[0m> <\033[36mclass\033[0m>
<\033[36mpriority\033[0m>    create a new process with the given:\r\n"
    "     name (1-16 characters)\r\n"
    "     class (0=system or 1=user)\r\n"
    "     priority (0=highest to 9=lowest)\r\n"
    "  \033[33mcreate\033[0m \033[36mhelp\033[0m    prints this message\r\n"
    "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}

//creates a new pcb and inserts it into the queue
/*
void create_pcb(const char* name, int process_class, int priority){
    struct pcb* p = pcb_find(name);

    // Checks if the process exists
    if (p != NULL){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given process already
exists\033[0m\n", 53);
    }

    // Checks if the name is appropriate length
    else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16
characters\033[0m\n", 59);
        create_help();
    }

    // Checks if the process class is valid
    else if (process_class != CLASS_SYSTEM && process_class != CLASS_USER){
        sys_req(WRITE, COM1, "\033[31mInvalid Process: Given class must be 0
(system) or 1 (user)\033[0m\n", 70);
        create_help();
    }

    // Validates priority
    else if (priority>9 || priority<0){
        sys_req(WRITE, COM1, "\033[31mInvalid Priority: Priority level must be
between 0 (Highest) and 9 (Lowest)\033[0m\n", 86);
        create_help();
    }

    else {
        struct pcb* newProcess = pcb_setup(name, process_class, priority);
```

```
        pcb_insert(newProcess);
        sys_req(WRITE, COM1, "\033[36mProcess created successfully\033[0m\n", 39);
    }

}
*/

//function for use in comhand
void create_pcb_command(const char* args){

    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and priority are given\033[0m\n", 69);
        create_help();
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        create_help();
    }

    // Handles getting process name, class, and priority number
    /*
    else{
        // get a substring for the process name from the args
        int temp = 0;
        for (int i=0; i<(int)strlen(args); i++){
            if (args[i] == ' ' && i<(int)strlen(args) && i != 0){
                temp = i;
                break;
            }
        }
        //parse the substring above into name
        if (temp != 0){
            char name[temp + 1];
            for (int i=0; i<temp; i++){
                name[i] = args[i];
            }
            name[temp] = '\0'; // Added in null terminator
            //parse process class
            char class[2];
            char priority_char[2];
            const char *numArg = args + temp +1;
            strncpy(class, numArg, 1);
            class[1]='\0';
            int process_class = atoi(class);
            if(class[0]<'0'||class[0]>'1' || *(numArg+1)!=' '){
                sys_req(WRITE, COM1, "\033[31mError: Class must be valid number\033[0m\n", 44);
                create_help();
                return;
            }
            //parse process priority
            numArg+=2;
            strncpy(priority_char, numArg, 1);
            priority_char[1]='\0';
            int priority = atoi(priority_char);
            if((priority_char[0]<'0' || priority_char[0]>'9') || *(numArg+1)!='\0'){
                sys_req(WRITE, COM1, "\033[31mError: Priority must be valid number\033[0m\n", 47);
```

```
                create_help();
                return;
            }
            create_pcb(name, process_class, priority);
        }
        else{
            sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and priority are given\033[0m\n", 69);
            create_help();
        }
    }
    */
}


//////////////////////////////////////////////////////
//delete function



//help message for delete
void delete_help(void){
    const char *helpMessage =
        "\r\n\033[33mdelete\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33mdelete\033[0m <\033[36mname\033[0m>    deletes the requested user process with the given name (1-16 characters)\r\n"
        "  \033[33mdelete\033[0m \033[36mhelp\033[0m      prints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}


//Finds the requested process and removes it from the queue
void delete_pcb(const char* name){
    struct pcb* proc_to_delete  = pcb_find(name);

    // Checks if the process exists
    if (proc_to_delete  == NULL){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given process does not exist\033[0m\n", 53);
    }

    // Checks if the name is appropriate length
    else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16 characters\033[0m\n", 59);
        delete_help();
    }

    // Checks if the process class
    else if (proc_to_delete ->process_class == CLASS_SYSTEM){
        sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process is a kernel/system level process\033[0m\n", 74);
    }

    else{
        pcb_remove(proc_to_delete );
        pcb_free(proc_to_delete );
        sys_req(WRITE, COM1, "\033[36mProcess deleted successfully\033[0m\n", 39);
    }
```

```
}

//function for use in comhand
void delete_pcb_command(const char* args){

    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is
given\033[0m\n", 47);
        delete_help();
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        delete_help();
    }

    // Handles getting process name
    else{
        delete_pcb(args);
    }
}

```

```c
#ifndef PCB_READY_H
#define PCB_READY_H

/**
 * @file ready.h
 * @brief Commands to suspend or resume a process.
 */

#include "pcb.h"

/**
 * @brief Prints the help message for the suspend command.
 */
void suspend_help(void);

/**
 * @brief Puts a non-system process in the suspended state, and moves it to the appropriate queue.
 * @param process_name Process's name (checks for validity)
 * @return int 0 for success, -1 for invalid process/name, -2 if given a system process, and 1 if already suspended.
 */
int suspend_pcb(const char* process_name);

/**
 * @brief Handles command-line arguments for the suspend command.
 * @param args Command argument string (process name or "help").
 */
void suspend_command(const char *args);

/**
 * @brief Prints the help message for the resume command.
 */
void resume_help(void);

/**
 * @brief Puts a process in the active (not suspended) state, and moves it to the appropriate queue.
 * @param process_name Process's name (checks for validity)
 * @return int 0 for success, -1 for invalid process/name, and 1 if already active.
 */
int resume_pcb(const char* process_name);

/**
 * @brief Handles command-line arguments for the resume command.
 * @param args Command argument string (process name or "help").
 */
void resume_command(const char *args);

#endif

```

```c
#include <sys_req.h>
#include <string.h>

#include "pcb.h"

// Because the commands "suspend" and "resume" share so much code, they were
// put into the same file for convenience, called "ready".



////////////////////////////////////////////////////////////////////////////
// suspend command

// help message for suspend
void suspend_help(void) {
    const char *helpMsg =
        "\r\n\033[33msuspend\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "  \033[33msuspend\033[0m <\033[36mname\033[0m>    Puts a non-system process in the suspended state, and moves it to the appropriate queue.\r\n"
        "  \033[33msuspend\033[0m \033[36mhelp\033[0m      prints this message\r\n\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

// Puts a non-system process in the suspended state, and moves it to the appropriate queue.
int suspend_pcb(const char* process_name) {
    // check for invalid name or process
    if (!process_name || strlen(process_name) == 0) return -1;
    struct pcb* found_pcb_ptr = pcb_find(process_name);
    if (!found_pcb_ptr) return -1;

    // check if system process
    if (found_pcb_ptr->process_class == CLASS_SYSTEM) return -2;

    // check if already suspended
    if (found_pcb_ptr->dispatch_state == DISPATCH_SUSPENDED) return 1;

    // put the process into the correct queue
    if (pcb_remove(found_pcb_ptr) != 0) return 1;
    found_pcb_ptr->dispatch_state = DISPATCH_SUSPENDED;
    pcb_insert(found_pcb_ptr);

    return 0;
}

// argument handler for suspend
void suspend_command(const char *args) {
    if (args == NULL || *args == '\0' || strcmp(args, "help") == 0) {
        suspend_help();
        return;
    }

    int result = suspend_pcb(args);
    if (result == -1) {
        const char *argMsg = "\033[31mInvalid process/name. Please try again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
        suspend_help();
    }
```

```
    else if (result == -2) {
        const char *argMsg = "\033[31mProcess is a system process and cannot be suspended.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }
    else if (result == 1) {
        const char *argMsg = "\033[31mThe process is already suspended.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }
    else if (result == 0) {
        const char *argMsg = "\033[36mThe process has been suspended.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }
    else {
        const char *argMsg = "\033[31mInvalid argument. Please try again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }

    return;
}



////////////////////////////////////////////////////////////////////////////
// resume command

// help message for resume
void resume_help(void) {
    const char *helpMsg =
        "\r\n\033[33mresume\033[0m [<\033[36mname\033[0m>|\033[36mall\033[0m]|\033[36mhelp\033[0m]\r\n"
        "  \033[33mresume\033[0m <\033[36mname\033[0m>    Puts a process in the active (not suspended) state, and moves it to the appropriate queue.\r\n"
        "  \033[33mresume\033[0m \033[36mall\033[0m      Puts all suspended processes in the ready state, and moves it to the appropriate queue.\r\n"
        "  \033[33mresume\033[0m \033[36mhelp\033[0m      prints this message\r\n\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

// Puts a process in the active (not suspended) state, and moves it to the appropriate queue.
int resume_pcb(const char* process_name) {
    // check for invalid name or process
    if (!process_name || strlen(process_name) == 0) return -1;
    struct pcb* found_pcb_ptr = pcb_find(process_name);
    if (!found_pcb_ptr) return -1;

    // check if already active
    if (found_pcb_ptr->dispatch_state == DISPATCH_ACTIVE) return 1;

    // put the process into the correct queue
    if (pcb_remove(found_pcb_ptr) != 0) return 1;
    found_pcb_ptr->dispatch_state = DISPATCH_ACTIVE;
    pcb_insert(found_pcb_ptr);

    return 0;
}

```

```
// argument handler for resume
void resume_command(const char *args) {
    if (args == NULL || *args == '\0' || strcmp(args, "help") == 0) {
        resume_help();
        return;
    }
    else if (args == NULL || *args == '\0' || strcmp(args, "all") == 0) {
        struct pcb* curPtr = suspended_ready_queue.head;
        struct pcb* tempPtr;
        // Checks if there are any processes in the ready queue
        if (curPtr == NULL){
            sys_req(WRITE, COM1, "No processes in the suspended ready queue\n", 42);
        }

        // If there are processes, move through queue and print each

        else{
            char* message = "\033[32mResuming all Processes:\033[0m\n";
            sys_req(WRITE, COM1, message, strlen(message));
            while(curPtr){
                tempPtr = curPtr->next;
                resume_pcb(curPtr->name);
                curPtr = tempPtr;
            }
        }
        return;
    }


    int result = resume_pcb(args);
    if (result == -1) {
        const char *argMsg = "\033[31mInvalid process/name. Please try
again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
        resume_help();
    }
    else if (result == 1) {
        const char *argMsg = "\033[31mThe process is already active.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }
    else if (result == 0) {
        const char *argMsg = "\033[36mThe process has been resumed.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }
    else {
        const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
    }

    return;
}

```

```h
#ifndef SETPRIORITY_H
#define SETPRIORITY_H

#include <pcb.h>
#include <sys_req.h>
#include <string.h>

/**
 * @file setPtriority.h
 * @brief Provides ability to set priority for processes
 */

/**
 * @brief Function for handling command input for setting priority, called by comhand
 * @param args Arguments for handling set priority command
 */
void set_priority_command(const char* args);

/**
 * @brief Provides help for how to use the priority set command
 */
void set_priority_help(void);

/**
 * @brief Sets priority of a process
 * @param name The name of the process that is being modified
 * @param newPriority The priority that the process is to be set to
 */
void setPriority(char* name, int newPriority);


#endif

```

```c
#include <pcb.h>
#include <sys_req.h>
#include <string.h>
#include <stdlib.h>

/**
 * Function for setting priority. Takes a string (name) and priority (integer)
 */
void setPriority(char* name, int newPriority){
    struct pcb* pcbPTR = pcb_find(name);

    // Checks if the process exists
    if (pcbPTR == NULL){
        sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process does not exist\033[0m\n", 56);
    }

    // Checks if the process class
    else if (pcbPTR->process_class == CLASS_SYSTEM){
        sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process is a kernel/system level process\033[0m\n", 74);
    }

    // Validates priority is possible
    else if (newPriority>9 || newPriority<0){
        sys_req(WRITE, COM1, "\033[31mInvalid Priority: Priority level must be between 0 (Highest) and 9 (Lowest)\033[0m\n", 86);
    }

    // Sets priority
    else if (newPriority<=9 && newPriority>=0){
        /*
         * To Fix Page fault errors, changed the priority set to remove the process from the queue, change its priority and reinsert it.
         */
        (void) pcb_remove(pcbPTR); /* ignore error code; pcb_remove will find the PCB if it's in a queue */
        pcbPTR->priority = newPriority;
        pcb_insert(pcbPTR);
    }
}

/**
 * Help message for setting priority
 */
void set_priority_help(void){
    const char *helpMessage =
        "\r\n\033[33mpriority set\033[0m [<\033[36mname\033[0m>|\033[36mhelp\033[0m] [<\033[36mpriority\033[0m>]\r\n"
        "  \033[33mpriority set\033[0m <\033[36mname\033[0m> <\033[36mpriority\033[0m>    sets the priority of a process to an integer priority value\r\n"
        "  \033[33mpriority set\033[0m \033[36mhelp\033[0m                 prints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}

/**
```

```
 * Creates function for use in command handler, for setting priority and getting help
 */
void set_priority_command(const char* args){

    // Checks for arguments
    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name and priority are given\033[0m\n", 61);
        set_priority_help();
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        set_priority_help();
    }

    // Handles getting process name and priority number
    else{
        int isNumber = 1;
        int temp = 0;
        for (int i=0; i<(int)strlen(args); i++){
            if (args[i] == ' ' && i<(int)strlen(args) && i != 0){
                temp = i;
                break;
            }
        }
        if (temp != 0){
            char name[temp + 1];
            for (int i=0; i<temp; i++){
                name[i] = args[i];
            }
            name[temp] = '\0'; // Added in null terminator
            const char *numArg = args + temp;
            for (int i = 0; i<(int)strlen(numArg)-1; i++){
                if ((numArg[i] <= '0' || numArg[i] >= '9') && (numArg[i]!='\0' && numArg[i]!=' ')){
                    isNumber = 0;
                    break;
                }
            }
            if (isNumber == 1){
                int priority = atoi(numArg);
                setPriority(name, priority);
            }
            else {
                sys_req(WRITE, COM1, "\033[31mError: Please ensure the priority argument is a number\033[0m\n", 64);
                set_priority_help();
            }
        }
        else{
            sys_req(WRITE, COM1, "\033[31mError: Please ensure a name and priority are given\033[0m\n", 61);
            set_priority_help();
        }
    }
}


```

```h
#ifndef SHOWPCB_H
#define SHOWPCB_H

#include <pcb.h>
#include <sys_req.h>
#include <string.h>
#include <comhand.h>
#include <itoa.h>

/**
 * @file showPCB.h
 * @brief Provides ability to display a specific PCB, all PCBs, or PCBs in specific queues
 */

/**
 * @brief Handles which functions should be used, for use in comhand
 * @param args Arguments used for performing correct function
 */
void show_pcb_command(const char* args);

/**
 * @brief Displays the details for a PCB if the PCB exists
 * @param name The name of the PCB to be displayed
 */
void showPCB(const char* name);

/**
 * @brief Displays all PCBs in the ready queue
 */
void showReady(void);

/**
 * @brief Displays all PCBs in the blocked queue
 */
void showBlocked(void);

/**
 * @brief Displays all PCBs in all queues
 */
void showAllPCB(void);

/**
 * @brief Help function for the show command
 */
void show_pcb_help(void);

/**
 * @brief Displays all PCBs in the suspended queues
 */
void showSuspended(void);

#endif

```

```
#include <pcb.h>
#include <sys_req.h>
#include <string.h>
#include <comhand.h>
#include <itoa.h>

/**
 * Help message for showing pcbs
 */
void show_pcb_help(void){
    const char *helpMessage =
        "\r\n\033[33mshow pcb\033[0m
[<\033[36mname\033[0m>|\033[36mready\033[0m|\033[36mblocked\033[0m|\033[36msuspended
\033[0m|\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
        "  \033[33mshow pcb\033[0m <\033[36mname\033[0m>     prints details for the
named process\r\n"
        "  \033[33mshow pcb\033[0m \033[36mready\033[0m      prints details of
processes in ready queue\r\n"
        "  \033[33mshow pcb\033[0m \033[36mblocked\033[0m    prints details of
processes in blocked queue\r\n"
        "  \033[33mshow pcb\033[0m \033[36msuspended\033[0m  prints details of
processes in the suspended queues\r\n"
        "  \033[33mshow pcb\033[0m \033[36mall\033[0m        prints details of all
processes\r\n"
        "  \033[33mshow pcb\033[0m \033[36mhelp\033[0m       prints this
message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
}

/**
 * Takes process name (string) and prints details for the process
 */
void showPCB(const char* name){
    char buffer[20];
    struct pcb* pcbPTR = pcb_find(name);

    // Checks if the process exists
    if (pcbPTR == NULL){
        sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process does not
exist\033[0m\n", 56);
        show_pcb_help();
    }
    else {

        // Print name
        sys_req(WRITE, COM1, "Name: ", strlen("Name: "));
        sys_req(WRITE, COM1, pcbPTR->name, strlen(pcbPTR->name));
        sys_req(WRITE, COM1, "\n", 1);

        // Print Class
        sys_req(WRITE, COM1, "Class: ", strlen("Class: "));
        if (pcbPTR->process_class == CLASS_SYSTEM){
            sys_req(WRITE, COM1, "System/Kernel\n", strlen("System/Kernel\n"));
        }
        else{
            sys_req(WRITE, COM1, "User\n", strlen("User\n"));
        }

        // Print state
```

```
52          sys_req(WRITE, COM1, "State: ", strlen("State: "));
53          if (pcbPTR->execution_state == STATE_READY){
54              sys_req(WRITE, COM1, "Ready\n", strlen("Ready\n"));
55          }
56          else if (pcbPTR->execution_state == STATE_RUNNING){
57              sys_req(WRITE, COM1, "Running\n", strlen("Running\n"));
58          }
59          else {
60              sys_req(WRITE, COM1, "Blocked\n", strlen("Blocked\n"));
61          }
62
63          // Print suspension state
64          sys_req(WRITE, COM1, "Suspension Status: ", strlen("Suspension Status: "));
65          if (pcbPTR->dispatch_state == DISPATCH_ACTIVE){
66              sys_req(WRITE, COM1, "Not suspended\n", strlen("Not suspended\n"));
67          }
68          else {
69              sys_req(WRITE, COM1, "Suspended\n", strlen("Suspended\n"));
70          }
71
72          // Print priority
73          sys_req(WRITE, COM1, "Priority: ", strlen("Priority: "));
74          itoa(pcbPTR->priority, buffer);
75          sys_req(WRITE, COM1, buffer, strlen(buffer));
76          sys_req(WRITE, COM1, "\n\n", 2);
77      }
78  }
79
80  /**
81   * Prints all processes in the ready queue
82   */
83  void showReady(void){
84      struct pcb* nextPtr = ready_queue.head;
85
86      // Checks if there are any processes in the ready queue
87      if (nextPtr == NULL){
88          sys_req(WRITE, COM1, "No processes in the ready queue\n", 32);
89      }
90
91      // If there are processes, move through queue and print each
92      else{
93          sys_req(WRITE, COM1, "\033[32mReady Processes:\033[0m\n", 27);
94          while(nextPtr){
95              showPCB(nextPtr->name);
96              nextPtr = nextPtr->next;
97          }
98      }
99  }
100
101 /**
102  * Prints all processes in the blocked queue
103  */
104 void showBlocked(void){
105     struct pcb* nextPtr = blocked_queue.head;
106
107     // Checks if there are any processes in the blocked queue
108     if (nextPtr == NULL){
109         sys_req(WRITE, COM1, "No processes in the blocked queue\n", 34);
110     }
111
```

```
112     // If there are processes, move through queue and print each
113     else{
114         sys_req(WRITE, COM1, "\033[31mBlocked Processes:\033[0m\n\r", 29);
115         while(nextPtr){
116             showPCB(nextPtr->name);
117             nextPtr = nextPtr->next;
118         }
119     }
120 }
121 
122 void showSuspended(void){
123     struct pcb* nextPtr = suspended_ready_queue.head;
124 
125     // Checks if there are suspended-ready processes
126     if (nextPtr == NULL){
127         sys_req(WRITE, COM1, "No processes in the suspended-ready queue\n", 42);
128     }
129     else{
130         sys_req(WRITE, COM1, "\033[33mSuspended Ready Processes:\033[0m\n\r", 37);
131         while(nextPtr){
132             showPCB(nextPtr->name);
133             nextPtr = nextPtr->next;
134         }
135     }
136 
137     nextPtr = suspended_blocked_queue.head;
138 
139     if (nextPtr == NULL){
140         sys_req(WRITE, COM1, "No processes in the suspended-blocked queue\n", 44);
141     }
142     else{
143         sys_req(WRITE, COM1, "\033[33mSuspended Blocked Processes:\033[0m\n\r", 39);
144         while(nextPtr){
145             showPCB(nextPtr->name);
146             nextPtr = nextPtr->next;
147         }
148     }
149 }
150 
151 /**
152  * Prints all processes
153  */
154 void showAllPCB(void){
155     showReady();
156     showBlocked();
157     showSuspended();
158 }
159 
160 
161 /**
162  * Function for use in command handler. Handles arguments for showing pcbs
163  */
164 void show_pcb_command(const char* args){
165 
166     // Checks if there are no arguments, or if argument is all
167     if (args == NULL || *args=='\0' || (strcmp(args, "all")==0)){
168         showAllPCB();
169     }
170 
171     // Checks if the argument is ready
```

```
    else if (strncmp(args, "ready", 5) == 0){
        showReady();
    }

    // Checks if the argument is blocked
    else if (strncmp(args, "blocked", 7) == 0){
        showBlocked();
    }

    // Checks if the argument is help
    else if (strncmp(args, "help", 4)==0){
        show_pcb_help();
    }

    else if (strncmp(args, "suspended", 9)==0){
        showSuspended();
    }

    // Handles any other input
    else{
        showPCB(args);
    }
}

```

```h
#ifndef ALARM_H
#define ALARM_H

/**
 * @brief Struct that stores data relating to the alarm
 * @struct stores Seconds, Minutes, Hours, and Message
 */
typedef struct {
    int hour;
    int minute;
    int second;
    char message[100];
} AlarmData;

/**
 * @brief Create an alarm with at the specified HH:MM:SS with specified message
 */
void alarm(void);

/**
 * @brief Creates a new pcb with alarm data
 * @param data Passes relevant data such as time (Hours, minutes, seconds) and alarm message
 */
void alarm_create(AlarmData* data);

/**
 * @brief Main handler for the alarm command.
 * @param args The argument string passed after 'alarm'
 */
void alarm_command(const char* args);

/**
 * @brief Prints help information related to the alarm command.
 */
void alarm_help(void);

#endif

```

```c
#include "clock.h"
#include <string.h>
#include <sys_req.h>
#include <stdint.h>
#include <mpx/interrupts.h>
#include <mpx/io.h>
#include "stdlib.h"
#include "itoa.h"
#include "alarm.h"
#include "clock.h"
#include <pcb.h>
#include <processes.h>
#include <loadR3.h>
#include "sys_req.h"
#include "sys_call.h"
#include "memory.h"


void alarm(void) {
    struct pcb* self = sys_get_current_process();
    //error checking
    if (!self) {
        // Shouldn't happen when properly dispatched, but guard anyway
        sys_req(EXIT);
        return;
    }
    AlarmData* data = (AlarmData*) self->args;

    while (1) {
        rtc_time_t current_time;
        get_time(&current_time);

        if ((current_time.hour > data->hour) ||
            (current_time.hour == data->hour && current_time.minute > data->minute)
||
            (current_time.hour == data->hour && current_time.minute == data->minute
&& current_time.second >= data->second)) {
            sys_req(WRITE, COM1, data->message, strlen(data->message));
            sys_req(WRITE, COM1, "\r\n", 2);
            // free data before exiting so it doesn't leak or become dangling
            sys_free_mem(data);
            self->args = NULL; // avoid dangling pointer
            break;
        }
        sys_req(IDLE);
    }
    sys_req(EXIT);
}

void alarm_create(AlarmData* data){
    // Allocate memory for alarm data
    AlarmData* newData = (AlarmData*) sys_alloc_mem(sizeof(AlarmData));
    if (!newData) {
        sys_req(WRITE, COM1, "\033[31mError: Memory allocation failed.\033[0m\r\n",
53);
        return;
    }

    // Copy fields
    newData->hour = data->hour;
```

```
    newData->minute = data->minute;
    newData->second = data->second;
    strncpy(newData->message, data->message, sizeof(newData->message) - 1);
    newData->message[sizeof(newData->message) - 1] = '\0';

    // Generate unique name
    static int alarm_counter = 1;
    char name[16];
    my_strcpy(name, "alarm");
    char numbuf[8];
    itoa(alarm_counter++, numbuf);
    my_strcat(name, numbuf);

    // Create and queue PCB
    struct pcb* newProc = pcb_setup(name, CLASS_USER, 1, alarm);
    //error checking
    if (!newProc) {
        sys_free_mem(data);
        const char *err = "\033[31mError: Failed to setup alarm
process.\033[0m\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
        return;
    }
    newProc->args = newData;
    pcb_insert(newProc);

    char* mes = "\033[32mAlarm set successfully.\033[0m\r\n";

    sys_req(WRITE, COM1, mes, strlen(mes));
}

void alarm_command(const char* args){
    if (args == NULL || *args =='\0'){
        sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and
priority are given\033[0m\n", 69);
        alarm_help();
        return;
    }

    // Checks if argument is help
    else if (strcmp(args, "help")==0){
        alarm_help();
        return;
    }
    char time_str[9]; //to hold time substring of args
    const char* time_ptr = strncpy(time_str, args, 8);
    const char* message = args + 9;  // skip "HH:MM:SS "


    //error test message
    if (strlen(message) > 100 || message == NULL || *message =='\0') {
        const char *err = "\033[31mMust include valid message\033[0m\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
        alarm_help();
        return;
    }
    // Expected format: HH:MM:SS
    if (strlen(time_ptr) != 8 || time_ptr[2] != ':' || time_ptr[5] != ':') {
        const char *err = "\033[31mInvalid format. Use set time
<HH:MM:SS>\033[0m\r\n";
```

```
        sys_req(WRITE, COM1, err, strlen(err));
        alarm_help();
        return;
    }

    char buf[3];
    buf[2] = '\0';

    buf[0] = time_ptr[0]; buf[1] = time_ptr[1];
    int hour = atoi(buf);

    buf[0] = time_ptr[3]; buf[1] = time_ptr[4];
    int minute = atoi(buf);

    buf[0] = time_ptr[6]; buf[1] = time_ptr[7];
    int second = atoi(buf);

    if (hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 || second > 59) {
        const char *err = "\033[31mInvalid time values.\033[0m\r\n";
        sys_req(WRITE, COM1, err, strlen(err));
        alarm_help();
        return;
    }
    AlarmData* data = (AlarmData*)sys_alloc_mem(sizeof(AlarmData));
    data->hour=hour;
    data->minute=minute;
    data->second=second;
    strncpy(data->message, message, sizeof(data->message) - 1);
    data->message[sizeof(data->message) - 1] = '\0';//ensure null termination
    alarm_create(data);
}

void alarm_help(void){
    const char *helpMsg =
        "\r\n\033[33malarm\033[0m [\033[36mhelp\033[0m|<\033[36mtime\033[0m>] [<\033[36mmessage\033[0m>]\r\n"
        "  \033[33malarm\033[0m <\033[36mtime\033[0m> <\033[36mmessage\033[0m> time in form HH:MM:SS (hours 0-23, minutes/seconds 0-59) and message (under 100 chars)\r\n"
        "  \033[33malarm\033[0m \033[36mhelp\033[0m                prints this message.\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

```

```c
#ifndef CLOCK_H
#define CLOCK_H

#include <string.h>
#include <sys_req.h>
#include <stdint.h>
#include <mpx/interrupts.h>
#include <mpx/io.h>

/**
 * @file clock.h
 * @brief Handles accesses to the Real Time Clock (RTC)
 */

/**
 * @brief Struct that stores data relating to the time
 * @struct stores Seconds, Minutes, and Hours
 */
typedef struct {
    uint8_t second;
    uint8_t minute;
    uint8_t hour;
} rtc_time_t;

/**
 * @brief Struct that stores data relating to the date
 * @struct stores Day, Month, and Year
 */
typedef struct {
    uint8_t day;
    uint8_t month;
    uint8_t year;   // Last two digits of the year
} rtc_date_t;


/**
 * @brief Accesses the RTC and stores the current time into specified struct
 * @param time Stores the current Seconds, Minutes, and Hours
 */
void get_time(rtc_time_t *time);

/**
 * @brief Stores the specified time into associated RTC registers
 * @param time Passes user defined Seconds, Minutes, and Hours
 */
void set_time(const rtc_time_t *time);

/**
 * @brief Accesses the RTC and stores the current date into specified struct
 * @param date Stores the current Day, Month, and Year
 */
void get_date(rtc_date_t *date);

/**
 * @brief Stores the specified date into associated RTC registers
 * @param date Passes user defined Day, Month, and Year
 */
void set_date(const rtc_date_t *date);

/**
```

```
 * @brief Prints the specified time in the form HH:MM:SS
 * @param time Passes the Seconds, Minutes, and Hours
 */
void print_time(rtc_time_t *time);

/**
 * @brief Prints the specified date in the form MM/DD/YY
 * @param time Passes the Day, Month, and Year
 */
void print_date(const rtc_date_t *date);

/**
 * @brief Prints help information related to the clock command.
 */
void clock_help(void);

/**
 * @brief Main handler for the clock command.
 * @param args The argument string passed after 'clock'
 */
void clock_command(const char *args);

//---- Helper Functions ----//

/**
 * @brief Concatenates src to the end of dest
 * @param dest The argument string to be appended to
 * @param src The argument string to append to
 */
void my_strcat(char *dest, const char *src);

/**
 * @brief Copies src to dest
 * @param dest Stores a copy of src
 * @param src Copied to dest
 */
void my_strcpy(char *dest, const char *src);

/**
 * @brief Writes data to an RTC register
 * @param reg Address of specified RTC register
 * @param value The byte to write to the RTC register
 */
void rtc_write(uint8_t reg, uint8_t value);

/**
 * @brief Reads data from an RTC register
 * @param reg Address of specified RTC register
 */
uint8_t rtc_read(uint8_t reg);

/**
 * @brief Converts an integer from binary to Binary Coded Decimal (BCD)
 * @param value The value to convert
 * @returns The specified value as a BCD
 */
uint8_t bin_to_bcd(uint8_t value);

/**
 * @brief Converts an integer from Binary Coded Decimal (BCD) to binary
```

```
 * @param value The value to convert
 * @returns The specified value in binary
 */
uint8_t bcd_to_bin(uint8_t value);

/**
 * @brief Converts the standard RTC timezone (UTC) to EST
 */
void tz_correction(void);

#endif

```

```c
#include "clock.h"
#include <string.h>
#include <sys_req.h>
#include <stdint.h>
#include <mpx/interrupts.h>
#include <mpx/io.h>
#include "stdlib.h"
#include "itoa.h"

#define CMOS_ADDRESS 0x70
#define CMOS_DATA    0x71

// RTC register indexes
#define REG_SECONDS   0x00
#define REG_MINUTES   0x02
#define REG_HOURS     0x04
#define REG_DAY       0x07
#define REG_MONTH     0x08
#define REG_YEAR      0x09
#define REG_STATUS_B  0x0B

// --- Helpers ---
uint8_t bcd_to_bin(uint8_t value) {
    return ((value >> 4) * 10) + (value & 0x0F);
}

uint8_t bin_to_bcd(uint8_t value) {
    return ((value / 10) << 4) | (value % 10);
}

uint8_t rtc_read(uint8_t reg) {
    outb(CMOS_ADDRESS, reg);
    return inb(CMOS_DATA);
}

void rtc_write(uint8_t reg, uint8_t value) {
    outb(CMOS_ADDRESS, reg);
    outb(CMOS_DATA, value);
}

void my_strcpy(char *dest, const char *src) {
    while (*src) {
        *dest++ = *src++;
    }
    *dest = '\0';
}

void my_strcat(char *dest, const char *src) {
    while (*dest) dest++;
    while (*src) {
        *dest++ = *src++;
    }
    *dest = '\0';
}

void clock_command(const char *args){
    if (args == NULL || *args == '\0') {
        rtc_time_t t;
        rtc_date_t d;
        get_time(&t);
```

```
        get_date(&d);
        print_date(&d);
        print_time(&t);
    }
    else if (strcmp(args, "get time") == 0) {
        rtc_time_t t;
        get_time(&t);
        print_time(&t);
    }
    else if (strcmp(args, "get date") == 0) {
        rtc_date_t d;
        get_date(&d);
        print_date(&d);
    }
    else if (strncmp(args, "set time ", 9) == 0) {
        const char *val = args + 9;  // skip "set time "

        // Expected format: HH:MM:SS
        if (strlen(val) != 8 || val[2] != ':' || val[5] != ':') {
            const char *err = "\033[31mInvalid format. Use set time
<HH:MM:SS>\033[0m\r\n";
            sys_req(WRITE, COM1, err, strlen(err));
            clock_help();
            return;
        }

        char buf[3];
        buf[2] = '\0';

        buf[0] = val[0]; buf[1] = val[1];
        int hour = atoi(buf);

        buf[0] = val[3]; buf[1] = val[4];
        int minute = atoi(buf);

        buf[0] = val[6]; buf[1] = val[7];
        int second = atoi(buf);

        if (hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 ||
second > 59) {
            const char *err = "\033[31mInvalid time values.\033[0m\r\n";
            sys_req(WRITE, COM1, err, strlen(err));
            clock_help();
            return;
        }

        rtc_time_t t = { second, minute, hour };
        set_time(&t);
        //print_time(&t);
    }
    else if (strncmp(args, "set date ", 9) == 0) {
        const char *val = args + 9;  // skip "set date "

        // Expected format: DD/MM/YY
        if (strlen(val) != 8 || val[2] != '/' || val[5] != '/') {
            const char *err = "\033[31mInvalid format. Use set date
<MM/DD/YY>\033[0m\r\n";
            sys_req(WRITE, COM1, err, strlen(err));
            clock_help();
            return;
```

```
        }

        char buf[3];
        buf[2] = '\0';

        buf[0] = val[0]; buf[1] = val[1];
        int month = atoi(buf);

        buf[0] = val[3]; buf[1] = val[4];
        int day = atoi(buf);

        buf[0] = val[6]; buf[1] = val[7];
        int year = atoi(buf);

        if (day < 1 || day > 31 || month < 1 || month > 12) {
            const char *err = "\033[31mInvalid date values.\033[0m\r\n";
            sys_req(WRITE, COM1, err, strlen(err));
            clock_help();
            return;
        }

        rtc_date_t d = { day, month, year };
        set_date(&d);
        //print_date(&d);
    }
    else if (strcmp(args, "help") == 0) {
        clock_help();
    }
    else {
        const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
        clock_help();
    }
}

void print_time(rtc_time_t *time){
    char buffer[100];
    char num[4];

    //time->hour = (time->hour + 24 - 4) % 24;
    my_strcpy(buffer, "\r\nTime: ");

    if (time->hour < 10) my_strcat(buffer, "0");
    itoa(time->hour, num);
    my_strcat(buffer, num);
    my_strcat(buffer, ":");

    if (time->minute < 10) my_strcat(buffer, "0");
    itoa(time->minute, num);
    my_strcat(buffer, num);
    my_strcat(buffer, ":");

    if (time->second < 10) my_strcat(buffer, "0");
    itoa(time->second, num);
    my_strcat(buffer, num);
    my_strcat(buffer, "\r\n\n");

    size_t len = strlen(buffer);

```

```
    sys_req(WRITE, COM1, buffer, len);
}

void print_date(const rtc_date_t *date){
    char buffer[100];
    char num[4];

    my_strcpy(buffer, "\r\nDate: ");

    if (date->month < 10) my_strcat(buffer, "0");
    itoa(date->month, num);
    my_strcat(buffer, num);
    my_strcat(buffer, "/");

    if (date->day < 10) my_strcat(buffer, "0");
    itoa(date->day, num);
    my_strcat(buffer, num);
    my_strcat(buffer, "/");

    if (date->year < 10) my_strcat(buffer, "0");
    itoa(date->year, num);
    my_strcat(buffer, num);
    my_strcat(buffer, "\r\n\n");

    size_t len = strlen(buffer);

    sys_req(WRITE, COM1, buffer, len);
}

void get_time(rtc_time_t *time) {
    uint8_t regB = rtc_read(REG_STATUS_B);

    time->second = rtc_read(REG_SECONDS);
    time->minute = rtc_read(REG_MINUTES);
    time->hour   = rtc_read(REG_HOURS);

    // Convert from BCD if needed
    if (!(regB & 0x04)) {
        time->second = bcd_to_bin(time->second);
        time->minute = bcd_to_bin(time->minute);
        time->hour   = bcd_to_bin(time->hour);
    }
}

void set_time(const rtc_time_t *time) {
    uint8_t regB = rtc_read(REG_STATUS_B);

    cli();
    if (!(regB & 0x04)) { // BCD mode
        rtc_write(REG_SECONDS, bin_to_bcd(time->second));
        rtc_write(REG_MINUTES, bin_to_bcd(time->minute));
        rtc_write(REG_HOURS,   bin_to_bcd(time->hour));
    } else { // Binary mode
        rtc_write(REG_SECONDS, time->second);
        rtc_write(REG_MINUTES, time->minute);
        rtc_write(REG_HOURS,   time->hour);
    }
    sti();
}

```

```
void get_date(rtc_date_t *date) {
    uint8_t regB = rtc_read(REG_STATUS_B);

    date->day   = rtc_read(REG_DAY);
    date->month = rtc_read(REG_MONTH);
    date->year  = rtc_read(REG_YEAR);

    // Convert from BCD if needed
    if (!(regB & 0x04)) {
        date->day   = bcd_to_bin(date->day);
        date->month = bcd_to_bin(date->month);
        date->year  = bcd_to_bin(date->year);
    }
}

void set_date(const rtc_date_t *date) {
    uint8_t regB = rtc_read(REG_STATUS_B);

    cli();
    if (!(regB & 0x04)) { // BCD mode
        rtc_write(REG_DAY,   bin_to_bcd(date->day));
        rtc_write(REG_MONTH, bin_to_bcd(date->month));
        rtc_write(REG_YEAR,  bin_to_bcd(date->year));
    } else { // Binary mode
        rtc_write(REG_DAY,   date->day);
        rtc_write(REG_MONTH, date->month);
        rtc_write(REG_YEAR,  date->year);
    }
    sti();
}

void clock_help(void) {
    const char *helpMsg =
        "\r\n\033[33mclock\033[0m
[\033[36mget\033[0m|\033[36mset\033[0m|\033[36mhelp\033[0m]  [\033[36mtime\033[0m
<\033[36mHH:MM:SS\033[0m>|\033[36mdate\033[0m <\033[36mMM/DD/YY\033[0m>]\r\n"
        "  \033[33mclock\033[0m                       prints the current date and
time.\r\n"
        "  \033[33mclock\033[0m \033[36mget\033[0m \033[36mtime\033[0m
prints the current time as: hour:minute:second.\r\n"
        "  \033[33mclock\033[0m \033[36mget\033[0m \033[36mdate\033[0m
prints the current date as: month/day/year.\r\n"
        "  \033[33mclock\033[0m \033[36mset\033[0m \033[36mtime\033[0m
<\033[36mHH:MM:SS\033[0m>    sets the current time to: hour:minute:second.\r\n"
        "  \033[33mclock\033[0m \033[36mset\033[0m \033[36mdate\033[0m
<\033[36mMM/DD/YY\033[0m>    sets the current date to: month/day/year.\r\n"
        "  \033[33mclock\033[0m \033[36mhelp\033[0m                   prints this
message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

void tz_correction(void){
    rtc_time_t t;
    rtc_date_t d;
    get_time(&t);
    get_date(&d);

    // Determine if subtracting 4 hours goes to previous day
    int new_hour = t.hour - 4;
```

```
    if (new_hour < 0) {
        new_hour += 24;

        // decrement day
        d.day--;

        // handle month/year wraparound
        if (d.day < 1) {
            d.month--;
            if (d.month < 1) {
                d.month = 12;
                d.year--;
            }

            // set day to last day of new month
            switch (d.month) {
                case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                    d.day = 31; break;
                case 4: case 6: case 9: case 11:
                    d.day = 30; break;
                case 2:
                    d.day = (d.year % 4 == 0) ? 29 : 28;
                    break;
            }
        }
    }

    t.hour = new_hour;

    // commit corrected time and date
    set_time(&t);
    set_date(&d);
}

```

```c
#ifndef EXIT_H
#define EXIT_H
/**
 * @file exit.h
 * @author Caleb Edwards
 * @brief Header file for the exit command used in the command handler.
 * Exits the terminal when called and confirmed by the user.
 */

void exit_help(void);

 /**
  * @brief Begins the shutdown process when the user
  * types 'exit' in the terminal. Confirmation by typing
  * 'Y' or 'n' is then required to completely exit.
  * @param arg_counter Counts the number of arguments input.
  * @param arg_vector Stores the arguments.
  * @return int return 1 to confirm exit and 0 to return to terminal.
  */
int exit_command(const char *args);

#endif

```

```c
#include <sys_req.h>
#include <string.h>
#include "comhand.h"
#include "exit.h"
#include <sys_req.h>

void exit_help(void) {
    const char *helpMsg =
        "\r\n\033[33mexit\033[0m [\033[36mhelp\033[0m|\033[36mforce\033[0m]\r\n"
        "  \033[33mexit\033[0m          prompts for confirmation before shutting down\r\n"
        "  \033[33mexit\033[0m \033[36mhelp\033[0m     displays this message\r\n"
        "  \033[33mexit\033[0m \033[36mforce\033[0m    shuts down without confirmation\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

int exit_command(const char *args) {
    if (args == NULL || *args == '\0') {
        // exit confirmation
        const char *confirmationMsg = "\r\nExit? (Y/n): \r\n";
        sys_req(WRITE, COM1, confirmationMsg, strlen(confirmationMsg));

        // exit confirmation logic
        while (1) {
            char confirmation[2] = {0};
            int nreadExit = sys_req(READ, COM1, confirmation, sizeof(confirmation));
            confirmation[nreadExit] = '\0';

            if (confirmation[0] == 'y' || confirmation[0] == 'Y') {
                const char *exitMsg = "\r\nExiting...\r\n";
                sys_req(WRITE, COM1, exitMsg, strlen(exitMsg));
                return 1; // tell comhand to exit

            } else if (confirmation[0] == 'n' || confirmation[0] == 'N') {
                const char *returnMsg = "\r\nReturning...\r\n";
                sys_req(WRITE, COM1, returnMsg, strlen(returnMsg));
                return 0;
            } else {
                const char *invalidMsg = "\r\nInvalid input. Please type 'Y' or 'n'.\r\n";
                sys_req(WRITE, COM1, invalidMsg, strlen(invalidMsg));
            }
        }
    }
    else {
        if (strcmp(args, "help") == 0) {
            exit_help();
      return 0;
        }
  else if (strcmp(args, "force") == 0 || strcmp(args, "f") == 0) {
      return 1;

        }
        else {
            const char *argMsg = "\033[31mInvalid argument. Please try again.\033[0m\r\n";
            sys_req(WRITE, COM1, argMsg, strlen(argMsg));
            exit_help();
```

```
57              return 0;
58          }
59      }
60
61      return 0;
62  }
63
```

```c
#ifndef HELP_H
#define HELP_H
/**
 * @file help.h
 * @author Caleb Edwards
 * @brief Header for the help command used in command handler.
 * Used to list the commands available to the user.
 */

void help_message(void);

/**
 * @brief Prints all commands available into the terminal
 * when the user types 'help' in the input.
 */
void help_command(const char *args);

/**
 * @brief Prints all commands individual help functions.
 */
void help_verbose(void);
#endif

```

```
#include <sys_req.h>
#include <string.h>
#include "help.h"
#include "exit.h"
#include "version.h"
#include "clock.h"
#include "setPriority.h"
#include "showPCB.h"
#include "ready.h"
#include "init.h"
#include "block.h"
#include "yield.h"
#include "loadR3.h"
#include "mcb/allocate.h"
#include "mcb/free.h"
#include "mcb/show.h"
#include "alarm.h"



void help_message(void) {
    // command help
    const char *helpMsg =
        "List of Commands [arguments]:\r\n\r\n"
        "\033[33mhelp\033[0m            [\033[36mverbose\033[0m]\r\n"
        "\033[33mversion\033[0m
[\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
        "\033[33mexit\033[0m
[\033[36mhelp\033[0m|\033[36mforce\033[0m]\r\n"
        "\033[33mclock\033[0m
[\033[36mget\033[0m|\033[36mset\033[0m|\033[36mhelp\033[0m]  [<\033[36mdate\033[0m>|
<\033[36mtime\033[0m>]\r\n"
        "\033[33mcreate\033[0m          [<\033[36mname\033[0m>|\033[36mhelp\033[0m]
[<\033[36mclass\033[0m>] [<\033[36mpriority\033[0m>]\r\n"
        "\033[33mdelete\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33malarm\033[0m           [\033[36mhelp\033[0m|<\033[36mtime\033[0m>]
[<\033[36mmessage\033[0m>]\r\n"
        "\033[33mshow pcb\033[0m
[<\033[36mname\033[0m>|\033[36mready\033[0m|\033[36mblocked\033[0m|\033[36mall\033[0
m|\033[36mhelp\033[0m]\r\n"
        "\033[33mpriority set\033[0m    [<\033[36mname\033[0m>|\033[36mhelp\033[0m]
[<\033[36mpriority\033[0m>]\r\n"
        "\033[33msuspend\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33mresume\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33mblock\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33munblock\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33myield\033[0m           [\033[36mhelp\033[0m]\r\n"
        "\033[33mload\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m|\033[36msuspended\033[0m]\r\n"
        "\033[33mallocate\033[0m
[<\033[36mbytes\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33mfree\033[0m
[<\033[36maddress\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33mshow mcb\033[0m
[\033[36mallocated\033[0m|\033[36mfree\033[0m|
```

```
<\033[36maddress\033[0m>|\033[36mhelp\033[0m]\r\n"
        "\033[33mclear\033[0m\r\n\r\n"
  "For more help, run '\033[33mhelp\033[0m \033[36mverbose\033[0m' or see the user guide at https://github.com/WVU-CS450/MacaroniPenguins.\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

void help_help(void) {
    const char *helpMsg =
        "List of Commands [arguments]:\r\n\r\n"
        "\033[33mhelp\033[0m [\033[36mverbose\033[0m]\r\n"
        "  \033[33mhelp\033[0m            prints a basic help message\r\n"
        "  \033[33mhelp\033[0m \033[36mverbose\033[0m    prints this message.\r\n\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

void help_verbose(void) {
    help_help();
    exit_help();
    version_help();
    clock_help();
    create_help();
    delete_help();
    alarm_help();
    show_pcb_help();
    set_priority_help();
    suspend_help();
    resume_help();
    block_help();
    unblock_help();
    yield_help();
    load_help();
    allocate_help();
    free_help();
    show_mcb_help();

    const char *clearMsg =
        "\r\n";
    sys_req(WRITE, COM1, clearMsg, strlen(clearMsg));

    const char *docMsg =
        "\033[33mclear\033[0m clears the terminal\r\n";
    sys_req(WRITE, COM1, docMsg, strlen(docMsg));
}

void help_command(const char *args) {

    if (args == NULL || *args == '\0') {
        help_message();
    }
    else if (strcmp(args, "help") == 0) {
        help_message();
    }
    else if (strcmp(args, "verbose") == 0 || strcmp(args, "v") == 0 || strcmp(args, "all") == 0 || strcmp(args, "a") == 0) {
        help_verbose();
    }
    else {
```

```
98          const char *argMsg = "\033[31mInvalid argument. Please try
    again.\033[0m\r\n\r\n";
99          sys_req(WRITE, COM1, argMsg, strlen(argMsg));
100         help_help();
101     }
102 }
103
104
```

```h
#ifndef LOADR3_H
#define LOADR3_H

/**
 * @file loadR3.h
 * @brief Handles loading the R3 processes into the system
 */

 /**
  * @brief Displays a help message for loading the processes
  */
void load_help(void);

/**
 * @brief Loads all 5 of the R3 processes
 */
void loadR3(void);


/**
 * @brief Loads all 5 of the processes, but in a suspended state
 */
void loadR3_suspended(void);

/**
 * @brief Loads a specific R3 process
 * @param name The name of a specific process to load
 */
void loadProcess(const char* name);

/**
 * @brief Handles the selection of which load function to call
 * @param args Argument for selecting which load function, for example "help"
 */
void load_command(const char* args);

#endif

```

```
#include <pcb.h>
#include <processes.h>
#include <loadR3.h>
#include "sys_req.h"
#include "string.h"
#include "ready.h"

// Displays help message
void load_help(void){
    char *helpMsg = "\r\n\033[33mload\033[0m
[<\033[36mname\033[0m>|\033[36mhelp\033[0m|\033[36msuspended\033[0m]\r\n"
        "  \033[33mload\033[0m            loads all 5 premade processes\r\n"
        "  \033[33mload \033[0m<\033[36mname\033[0m>     loads a specific process,
name is like 'proc1'\r\n"
        "  \033[33mload \033[36msuspended  \033[0mloads all 5 processes as
suspended\r\n"
        "  \033[33mload \033[36mhelp \033[0m      prints this message\r\n"
        "\r\n";
        sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

// Loads all R3 processes
void loadR3(void){
    // Struct used for setup, reused for each process
    struct pcb* newProc = pcb_setup("proc1", CLASS_USER, 1, proc1);
    pcb_insert(newProc);
    newProc = pcb_setup("proc2", CLASS_USER, 1, proc2);
    pcb_insert(newProc);
    newProc = pcb_setup("proc3", CLASS_USER, 1, proc3);
    pcb_insert(newProc);
    newProc = pcb_setup("proc4", CLASS_USER, 1, proc4);
    pcb_insert(newProc);
    newProc = pcb_setup("proc5", CLASS_USER, 1, proc5);
    pcb_insert(newProc);
    char *msg = "Processes have been loaded\r\n";
    sys_req(WRITE, COM1, msg, strlen(msg));
}

// Loads and supsends all processes
void loadR3_suspended(void){
    loadR3();
    suspend_pcb("proc1");
    suspend_pcb("proc2");
    suspend_pcb("proc3");
    suspend_pcb("proc4");
    suspend_pcb("proc5");
}

// Loads a specific process
void loadProcess(const char* name){
    // Struct for a named process
    struct pcb* newProc;
    char *msg = "Process has been loaded\r\n";
    if (strcmp(name, "proc1")==0){
        newProc = pcb_setup("proc1", CLASS_USER, 1, proc1);
        pcb_insert(newProc);
        sys_req(WRITE, COM1, msg, strlen(msg));
    }
    else if (strcmp(name, "proc2")==0){
        newProc = pcb_setup("proc2", CLASS_USER, 1, proc2);
```

```
        pcb_insert(newProc);
        sys_req(WRITE, COM1, msg, strlen(msg));
    }
    else if (strcmp(name, "proc3")==0){
        newProc = pcb_setup("proc3", CLASS_USER, 1, proc3);
        pcb_insert(newProc);
        sys_req(WRITE, COM1, msg, strlen(msg));
    }
    else if (strcmp(name, "proc4")==0){
        newProc = pcb_setup("proc4", CLASS_USER, 1, proc4);
        pcb_insert(newProc);
        sys_req(WRITE, COM1, msg, strlen(msg));
    }
    else if (strcmp(name, "proc5")==0){
        newProc = pcb_setup("proc5", CLASS_USER, 1, proc5);
        pcb_insert(newProc);
        sys_req(WRITE, COM1, msg, strlen(msg));
    }
    else{
        char *errorMsg = "\033[31mError: Please enter a valid process name, like
'proc1'\033[0m\r\n";
        sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
        load_help();
    }
}

void load_command(const char* args){
    if (args==NULL || *args == '\0'){
        loadR3();
    }
    else if (strncmp(args, "suspended", 9)==0){
        loadR3_suspended();
    }
    else if (strncmp(args, "help", 4)==0){
        load_help();
    }
    else{
        loadProcess(args);
    }
}

```

```h
#ifndef VERSION_H
#define VERSION_H

#ifndef GIT_DATE
#define GIT_DATE "unknown"
#endif

#ifndef GIT_HASH
#define GIT_HASH "unknown"
#endif

#ifndef GIT_DIRTY
#define GIT_DIRTY "unknown"
#endif

/**
 * @file version.h
 * @brief Displays the current version of MacaroniOS.
 */


/**
 * @brief Prints help information related to the version command.
 */
void version_help(void);

/**
 * @brief Displays the latest version.
 */
void version_latest(void);

/**
 * @brief Displays the past and present versions.
 */
void version_history(void);

/**
 * @brief Main handler for the version command.
 * @param args The argument string passed after 'version'
 */
void version_command(const char *args);

#endif

```

```c
#include <sys_req.h>
#include <string.h>
#include "version.h"

void version_help(void) {
    const char *helpMsg =
        "\r\n\033[33mversion\033[0m [\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
        "  \033[33mversion\033[0m         prints information about the current version as: R<x> <date> <hash> (<status>)\r\n"
        "  \033[33mversion\033[0m \033[36mall\033[0m     prints information about all version history as: R<x> <date> <hash>\r\n"
        "  \033[33mversion\033[0m \033[36mhelp\033[0m    prints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

void version_latest(void) {
    char buffer[128];
    char *write_ptr = buffer;
    const char *read_ptr;

    // main version
    *write_ptr++ = 'R';
    *write_ptr++ = '6';
    *write_ptr++ = ' ';

    // git metadata
    read_ptr = GIT_DATE;
    while (*read_ptr) {
        *write_ptr++ = *read_ptr++;
    }
    *write_ptr++ = ' ';

    read_ptr = GIT_HASH;
    while (*read_ptr) {
        *write_ptr++ = *read_ptr++;
    }
    *write_ptr++ = ' ';

    *write_ptr++ = '(';
    read_ptr = GIT_DIRTY;
    while (*read_ptr) {
        *write_ptr++ = *read_ptr++;
    }
    *write_ptr++ = ')';

    *write_ptr++ = '\r';
    *write_ptr++ = '\n';
    *write_ptr = '\0';

    // print out
    sys_req(WRITE, COM1, buffer, write_ptr - buffer);
}

void version_history(void) {
    const char *historyMsg =
        "R5 2025-11-20 431691d\r\n"
        "R4 2025-10-31 266935a\r\n"
        "R3 2025-10-20 9b0243f\r\n"
        "R2 2025-10-03 15c9229\r\n"
```

```
      "R1 2025-09-12 956ee13\r\n"
        "R0 2025-08-24 6738747\r\n";
    sys_req(WRITE, COM1, historyMsg, strlen(historyMsg));
}

void version_command(const char *args) {
    if (args == NULL || *args == '\0') {
        version_latest();
    }
    else if (strcmp(args, "all") == 0) {
        version_latest();
        version_history();
    }
    else if (strcmp(args, "help") == 0) {
        version_help();
    }
    else {
        const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
        sys_req(WRITE, COM1, argMsg, strlen(argMsg));
        version_help();
    }
}

```

```h
#ifndef YIELD_H
#define YIELD_H

/**
 * @file yield.h
 * @brief Adds yield functionality so that a process can be set to IDLE
 */

 /**
  * @brief Yields the current process
  */
void yield(void);

/**
 * @brief Prints a help message for the yield function
 */
void yield_help(void);

/**
 * @brief Handles arguments given in the command handler for yield
 * @param args Arguments for the yield command
 */
void yield_command(const char* args);

#endif

```

```
#include "sys_req.h"
#include <string.h>

void yield(void){
    sys_req(IDLE);
}

void yield_help(void){
    const char *helpMsg = "\r\n\033[33myield\033[0m [\033[36mhelp\033[0m]\r\n"
        "  \033[33myield\033[0m         yields the currently running process\r\n"
        "  \033[33myield \033[36mhelp    \033[0mprints this message\r\n"
        "\r\n";
    sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
}

void yield_command(const char* args){
    if (args == NULL || *args=='\0'){
        yield();
    }
    else if (strcmp(args, "help")==0){
        yield_help();
    }
    else{

    }
}

```