

```
1 #ifndef COMHAND_H
2 #define COMHAND_H
3
4 /**
5  * @file comhand.h
6  * @brief Command handler interface for the OS.
7  * Reads from the polling input and executes commands.
8 */
9
10 /**
11  * @brief Prints a welcome message and penguin ASCII art to the terminal
12  */
13 void com_startup(void);
14
15 /**
16  * @brief Trim function to remove \\n and \\r from the string.
17  * @param str string variable to trim
18  */
19 void trim_Input(char *str);
20
21 /**
22  * @brief Enters a loop and waits for the user to input commands.
23  */
24 void comhand(void);
25
26#endif
27
```

```

1 #include "comhand.h"
2 #include <string.h>
3 #include <sys_req.h>
4 #include "help.h"
5 #include "exit.h"
6 #include "version.h"
7 #include "clock.h"
8 #include "showPCB.h"
9 #include "setPriority.h"
10 #include "ready.h"
11 #include "init.h"
12 #include "block.h"
13 #include "yield.h"
14 #include "loadR3.h"
15 #include "alarm.h"
16 #include "mcb/allocate.h"
17 #include "mcb/free.h"
18 #include "mcb/show.h"
19
20
21 // macaroni penguin ASCII image on startup
22 void com_startup(void) {
23     sys_req(WRITE, COM1, "\r\n-----\r\n", 80);
24     sys_req(WRITE, COM1, "\r\n", 3);
25
26     const char *bannerPart1 =
27         "\033[33m\|\033[0m      Welcome to
\033[33mMacaroniOS\033[0m!\r\n"
28         " --(o ).      Version \033[36m";
29     sys_req(WRITE, COM1, bannerPart1, strlen(bannerPart1));
30
31     version_latest();
32
33     const char *bannerPart2 =
34         "\033[0m"
35         ".-\.\r\n"
36         "/| \\\\"      CS450: Operating Systems Structure\r\n"
37         "' | ||\r\n"
38         "_\\_) :,_    Type '\033[33mhelp\033[0m' for a list of commands or
'\033[31mexit\033[0m' to shutdown.\r\n\r\n";
39     sys_req(WRITE, COM1, bannerPart2, strlen(bannerPart2));
40 }
41
42 // trim Function for input (trims \r and \n from input)
43 void trim_Input(char *str){
44     int length = strlen(str);
45     while (length > 0 && (str[length - 1] == '\n' || str[length - 1] == '\r')){
46         str[length - 1] = '\0';
47         length--;
48     }
49 }
50
51 // command handler (DOES NOT handle arguments)
52 void comhand(void)
53 {
54     // startup message
55     //com_startup();
56
57     //TimeZone corrections

```

```

58     tz_correction();
59
60     // loop through the entire buffer
61     for (;;) {
62         char buf[100] = {0};
63
64         // read command
65         int nread = sys_req(READ, COM1, buf, sizeof(buf));
66         buf[nread] = '\0'; // adds null terminator
67         trim_Input(buf); // trims \r\n
68         sys_req(WRITE, COM1, "\r\n", 2); // Start new line
69
70         // Yield CPU after reading input to allow multitasking
71         yield();
72
73         // to test some commands, we create user functions, but they are deprecated
before "production"
74         char* deprecated_msg = "\033[31mError: This function is deprecated and no
longer usable.\033[0m\r\n";
75
76         // command logic: each function handles its own argument(s) for better
encapsulation
77         if (strncmp(buf, "exit", 4) == 0) {
78             char *args = buf + 4;
79             while (*args == ' ') args++;
80             if (exit_command(args)){
81                 sys_req(EXIT);
82             }
83         }
84         else if (strncmp(buf, "shutdown", 8) == 0) {
85             char *args = buf + 8;
86             while (*args == ' ') args++;
87             if (exit_command(args)){
88                 sys_req(EXIT);
89             }
90         }
91         else if (strncmp(buf, "version", 7) == 0) {
92             char *args = buf + 7;
93             while (*args == ' ') args++;
94             version_command(args);
95         }
96         else if (strncmp(buf, "help", 4) == 0) {
97             char *args = buf + 4;
98             while (*args == ' ') args++;
99             help_command(args);
100        }
101        else if (strncmp(buf, "clock", 5) == 0) {
102            char *args = buf + 5;
103            while (*args == ' ') args++;
104            clock_command(args);
105        }
106        else if (strncmp(buf, "show pcb", 8) == 0) {
107            char *args = buf + 8;
108            while (*args == ' ') args++;
109            show_pcb_command(args);
110        }
111        else if (strncmp(buf, "priority set", 12) == 0) {
112            char *args = buf + 12;
113            while (*args == ' ') args++;
114            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 }

```

```
175     else if (strncmp(buf, "allocate", 8) == 0) {
176         // char *args = buf + 8;
177         // while (*args == ' ') args++;
178         // allocate_command(args);
179         sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
180     }
181     else if (strncmp(buf, "free", 4) == 0) {
182         // char *args = buf + 4;
183         // while (*args == ' ') args++;
184         // free_command(args);
185         sys_req(WRITE, COM1, deprecated_msg, strlen(deprecated_msg));
186     }
187     else if (strncmp(buf, "show mcb", 8) == 0) {
188         char *args = buf + 8;
189         while (*args == ' ') args++;
190         show_mcb_command(args);
191     }
192     else {
193         const char *invalidMsg = "\033[31mInvalid command. Please try
again.\033[0m\r\n\r\n";
194         sys_req(WRITE, COM1, invalidMsg, strlen(invalidMsg));
195         help_message();
196     }
197 }
198 }
199 }
```

```
1 #ifndef ITOA_H
2 #define ITOA_H
3
4 /**
5  * @file itoa.h
6  * @brief Declaration for interger-to-ASCII conversion
7  */
8
9 /**
10 * @brief Converts an integer to a C-string.
11 * @param num The integer to convert.
12 * @param buffer Pointer to an array to store the string.
13 */
14
15 void itoa(int num, char* buffer);
16
17 #endif
18
```

```

1 void itoa(int num, char* buffer) {
2     // get the absolute value of num
3     int numLeft;
4     if (num < 0) {
5         numLeft = -1 * num;
6     }
7     else {
8         numLeft = num;
9     }
10
11    // find the maximum power of 10 needed
12    int maxPower = 0;
13    while (numLeft > 1) {
14        numLeft = numLeft / 10;
15        if (numLeft < 1) {
16            break;
17        }
18        else {
19            maxPower++;
20        }
21    }
22
23    // reset numLeft to the value of num
24    if (num < 0) {
25        numLeft = -1 * num;
26    }
27    else {
28        numLeft = num;
29    }
30
31    // initialize variables
32    int numToChar;
33    int tenToThePower;
34
35    // convert negative int to string
36    if (num < 0) {
37        buffer[0] = '-';
38        for (int i = 1; i < maxPower + 2; i++) {
39            // convert each int to char
40            tenToThePower = 1;
41            for (int j = 0; j < maxPower - i + 1; j++) {
42                tenToThePower = tenToThePower * 10;
43            }
44
45            // write each char to buffer
46            numToChar = (int)(numLeft / tenToThePower);
47            buffer[i] = numToChar + '0';
48            numLeft = numLeft - numToChar * tenToThePower;
49        }
50    }
51    // convert positive int to string
52    else {
53        for (int i = 0; i < maxPower + 1; i++) {
54            // convert each int to char
55            tenToThePower = 1;
56            for (int j = 0; j < maxPower - i; j++) {
57                tenToThePower = tenToThePower * 10;
58            }
59
60            // write each char to buffer

```

```
61     numToChar = (int)(numLeft / tenToThePower);
62     buffer[i] = numToChar + '0';
63     numLeft = numLeft - numToChar * tenToThePower;
64 }
65 }
66
67 // add null terminator
68 if (num < 0) {
69     buffer[maxPower + 2] = '\0';
70 }
71 else {
72     buffer[maxPower + 1] = '\0';
73 }
74 }
75 }
```

```
1 #ifndef ITOBCD_H
2 #define ITOBCD_H
3
4 /**
5  * @file itoBCD.h
6  * @brief Function that converts an integer into a string
7  * that is representative of the binary coded decimal
8  * format of the input integer.
9 */
10
11 /**
12  * Convert an integer to an Binary Coded Decimal
13  * @param int Integer being converted into a Binary Coded Decimal
14  * @param s A buffer to hold the created string
15 */
16
17 void itoBCD(int num, char* buffer);
18
19#endif
```

```
1 void itoBCD(int num, char* buffer) {
2
3     // Variables used
4     int numLeft = num;
5     int power = 0;
6     char BCDval[4];
7     int tenToThe;
8     int numTo;
9
10    //Gets highest power of ten
11    while (numLeft > 0) {
12        numLeft = (int)(numLeft / 10);
13        power++;
14    }
15    numLeft = num;
16
17    /* This for loop is used to convert an integer into binary coded decimal
18     * Determines the digit in each place starting with the highest power of ten
19     * in the number. When a digit is found, the correct binary coded decimal value
20     * is selected and input into a string
21     */
22    for (int count = 0; count < power; count++) {
23        numTo = numLeft;
24        tenToThe = 1;
25        for (int j = 0; j < power-count-1; j++) {
26            tenToThe = tenToThe * 10;
27        }
28        numLeft = numLeft % tenToThe;
29        numTo = (numTo - numLeft)/tenToThe;
30        // Switch-Case determines what BCD value based on digit value
31        if (numTo == 1) {
32            BCDval[0] = '0';
33            BCDval[1] = '0';
34            BCDval[2] = '0';
35            BCDval[3] = '1';
36        }
37        else if (numTo == 2) {
38            BCDval[0] = '0';
39            BCDval[1] = '0';
40            BCDval[2] = '1';
41            BCDval[3] = '0';
42        }
43        else if (numTo == 3) {
44            BCDval[0] = '0';
45            BCDval[1] = '0';
46            BCDval[2] = '1';
47            BCDval[3] = '1';
48        }
49        else if (numTo == 4) {
50            BCDval[0] = '0';
51            BCDval[1] = '1';
52            BCDval[2] = '0';
53            BCDval[3] = '0';
54        }
55        else if (numTo == 5) {
56            BCDval[0] = '0';
57            BCDval[1] = '1';
58            BCDval[2] = '0';
59            BCDval[3] = '1';
60    }
```

```
61     else if (numTo == 6) {
62         BCDval[0] = '0';
63         BCDval[1] = '1';
64         BCDval[2] = '1';
65         BCDval[3] = '0';
66     }
67     else if (numTo == 7) {
68         BCDval[0] = '0';
69         BCDval[1] = '1';
70         BCDval[2] = '1';
71         BCDval[3] = '1';
72     }
73     else if (numTo == 8) {
74         BCDval[0] = '1';
75         BCDval[1] = '0';
76         BCDval[2] = '0';
77         BCDval[3] = '0';
78     }
79     else if (numTo == 9) {
80         BCDval[0] = '1';
81         BCDval[1] = '0';
82         BCDval[2] = '0';
83         BCDval[3] = '1';
84     }
85     else if (numTo == 0){
86         BCDval[0] = '0';
87         BCDval[1] = '0';
88         BCDval[2] = '0';
89         BCDval[3] = '0';
90     }
91
92     // Adds BCD value of digit into string, appends onto end
93     for (int j = 0; j < 4; j++) {
94         buffer[j + 4*count] = BCDval[j];
95     }
96 }
97 }
98
99 // Adds null terminator to end of string
100 buffer[4 * power] = '\0';
101 }
102 }
```

```

1 /**
2  * @file pcb.h
3  * @brief Process Control Block queue and stack functions.
4  *
5  * Defines the data structure types and functions for operating
6  * the PCB queues.
7  * - Ready Queue: PCBs waiting to be executed by priority.
8  * - Blocked Queue: PCBs waiting on something else to happen before its ready.
9  * - Suspended-Ready Queue: PCBs not in the stack but ready.
10 * - Suspended-Blocked Queue: PCBs not in the stack and blocked.
11 *
12 */
13
14 #ifndef PCB_H
15 #define PCB_H
16 #include <stdint.h>
17 #define PCB_NAME_MAX_LEN 16
18 #define PCB_STACK_MIN_SIZE 1024
19
20 enum process_class{
21     CLASS_SYSTEM = 0,
22     CLASS_USER = 1
23 };
24
25 enum execution_state{
26     STATE_READY = 0,
27     STATE_RUNNING = 1,
28     STATE_BLOCKED = 2
29 };
30
31 enum dispatch_state{
32     DISPATCH_ACTIVE = 0,
33     DISPATCH_SUSPENDED = 1
34 };
35
36 struct context{
37     uint32_t gs, fs, es, ds;
38     uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
39     uint32_t eip, cs, eflags;
40 } __attribute__((packed));
41
42 struct pcb{
43     char name[PCB_NAME_MAX_LEN];
44     enum process_class process_class;
45     int priority; // 0 (highest) to 9
46     enum execution_state execution_state;
47     enum dispatch_state dispatch_state;
48     void *args; // optional pointer for process-specific data (i.e alarm)
49     void *stack; // Dynamically allocated, might manually allocate based on memory
management.
50     struct context* contextPtr;
51     struct pcb* next;
52     struct pcb* prev;
53 };
54
55 struct pcb_queue{
56     struct pcb* head;
57     struct pcb* tail;
58 };
59

```

```

60 // Queue initialization
61 extern struct pcb_queue ready_queue;
62 extern struct pcb_queue blocked_queue;
63 extern struct pcb_queue suspended_ready_queue;
64 extern struct pcb_queue suspended_blocked_queue;
65
66 /**
67 * @brief Allocates a new PCB and its stack.
68 *
69 * Initializes all members to zero and sets stack pointer.
70 *
71 * @return Pointer to the allocated PCB, or NULL if allocation fails.
72 */
73 struct pcb* pcb_allocate(void);
74
75 /**
76 * @brief Frees a previously allocated PCB and its stack.
77 *
78 * @param ptr Pointer to the PCB to free.
79 * @return 0 on success, -1 if ptr is NULL.
80 */
81 int pcb_free(struct pcb* ptr);
82
83 /**
84 * @brief Initializes a PCB with given name, class, and priority.
85 *
86 * Sets default execution and dispatch states.
87 *
88 * @param name Name of the process.
89 * @param process_class Class of the process (SYSTEM/USER).
90 * @param priority Initial priority (0-9).
91 * @return Pointer to initialized PCB, or NULL if allocation/setup fails.
92 */
93 struct pcb* pcb_setup(const char* name, int process_class, int priority, void
(*function)(void));
94
95 /**
96 * @brief Finds a PCB by its name in all queues.
97 *
98 * @param name Name of the process to find.
99 * @return Pointer to the PCB if found, NULL otherwise.
100 */
101 struct pcb* pcb_find(const char* name);
102
103 /**
104 * @brief Insert a PCB into the appropriate queue.
105 *
106 * Chooses the correct queue based on the PCB's dispatch and execution state,
107 * then inserts it. If the target queue is the ready queue, PCBs are inserted
108 * by priority (FIFO within same priority). Otherwise, they are appended FIFO
109 * at the tail.
110 *
111 * @param ptr Pointer to the PCB to insert.
112 */
113 void pcb_insert(struct pcb* ptr);
114
115 /**
116 * @brief Remove a PCB from its current queue.
117 *
118 * Locates the PCB's queue based on its state, then detaches it by

```

```
119 * fixing neighboring links. Clears the PCB's next/prev pointers.  
120 *  
121 * @param ptr Pointer to the PCB to remove.  
122 * @return 0 on success, -1 if PCB is NULL or not found in any queue.  
123 */  
124 int pcb_remove(struct pcb* ptr);  
125  
126 #endif  
127
```

```

1 #include "pcb.h"
2 #include <string.h>
3 #include <sys_req.h>
4 #include <memory.h>
5 #include <showPCB.h>
6
7 /**
8  * TODO
9  * Will add in error messaging at different points
10 */
11
12 // Queue Initialization
13 struct pcb_queue ready_queue = { NULL, NULL };
14 struct pcb_queue blocked_queue = { NULL, NULL };
15 struct pcb_queue suspended_ready_queue = { NULL, NULL };
16 struct pcb_queue suspended_blocked_queue = { NULL, NULL };
17
18 struct pcb* pcb_allocate(void){
19     struct pcb* ptr =(struct pcb*) sys_alloc_mem(sizeof(struct pcb));
20     if (!ptr) return NULL;
21     memset(ptr, 0, sizeof(struct pcb)); // initialize to 0
22     ptr->stack = sys_alloc_mem(PCB_STACK_MIN_SIZE);
23
24     // if allocation fails
25     if(!ptr->stack){
26         sys_free_mem(ptr);
27         return NULL;
28     }
29
30     memset(ptr->stack, 0, PCB_STACK_MIN_SIZE);
31     ptr->contextPtr = (struct context*)((uint8_t *)ptr->stack + PCB_STACK_MIN_SIZE -
32     sizeof(struct context)); // Downward stack movement
33     ptr->next = NULL;
34     ptr->prev = NULL;
35
36     return ptr;
37 }
38
39 int pcb_free(struct pcb* ptr){
40     if (!ptr) return -1;
41     if (ptr->stack) sys_free_mem(ptr->stack);
42     sys_free_mem(ptr);
43     return 0;
44 }
45
46 struct pcb* pcb_setup(const char* name, int process_class, int priority, void
47 (*function)(void)){
48     if (!name || strlen(name) >= PCB_NAME_MAX_LEN) return NULL;
49     if (priority < 0 || priority > 9) return NULL; // Also handle in User Functions
50     if (pcb_find(name)!=NULL){
51         char* errorMsg = "\033[31mError: Process with name ''";
52         sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
53         sys_req(WRITE, COM1, name, strlen(name));
54         errorMsg = "' already exists\033[0m\r\n";
55         sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
56     }
57     struct pcb* ptr = pcb_allocate();
58     if (!ptr) return NULL;
59     strncpy(ptr->name, name, PCB_NAME_MAX_LEN - 1);

```

```

59     ptr->name[PCB_NAME_MAX_LEN - 1] = '\0';
60     ptr->process_class = process_class;
61     ptr->priority = priority;
62     ptr->execution_state = STATE_READY;
63     ptr->dispatch_state = DISPATCH_ACTIVE;
64     ptr->contextPtr->fs = 0x10;
65     ptr->contextPtr->gs = 0x10;
66     ptr->contextPtr->ds = 0x10;
67     ptr->contextPtr->es = 0x10;
68     ptr->contextPtr->cs = 0x08;
69     ptr->contextPtr->ebp = 0;
70     ptr->contextPtr->esp = (uint32_t)((uint8_t *)ptr->contextPtr + sizeof(struct
context));
71     ptr->contextPtr->eflags = 0x202;
72     ptr->contextPtr->eip = (uint32_t)function;
73     return ptr;
74 }
75
76 struct pcb* pcb_find(const char* name){
77     if (!name) return NULL;
78     struct pcb_queue* queues[] = {
79         &ready_queue,
80         &blocked_queue,
81         &suspended_ready_queue,
82         &suspended_blocked_queue
83     };
84     for (int i = 0; i < 4; i++){
85         struct pcb* find_ptr = queues[i]->head;
86         while(find_ptr){
87             if (strcmp(find_ptr->name, name) == 0){
88                 return find_ptr;
89             }
90             find_ptr = find_ptr->next;
91         }
92     }
93     return NULL; // Not found
94 }
95
96 void pcb_insert(struct pcb* ptr) {
97     // Safety check
98     if (!ptr) return;
99
100    struct pcb_queue* q_ptr = NULL;
101
102    // Select correct queue based on dispatch and execution state
103    if (ptr->dispatch_state == DISPATCH_SUSPENDED) {
104        if (ptr->execution_state == STATE_BLOCKED) {
105            q_ptr = &suspended_blocked_queue;
106        } else {
107            q_ptr = &suspended_ready_queue;
108        }
109    } else {
110        if (ptr->execution_state == STATE_BLOCKED) {
111            q_ptr = &blocked_queue;
112        } else {
113            q_ptr = &ready_queue;
114        }
115    }
116
117    // Insert PCB into the chosen queue

```

```

118 if (q_ptr == &ready_queue) {
119     // Priority-ordered insert (FIFO within same priority)
120     struct pcb* cur = q_ptr->head;
121     while (cur && cur->priority <= ptr->priority) {
122         cur = cur->next;
123     }
124
125     // If cur is NULL, we reached the end of the list
126     if (!cur) {
127         // Insert at tail if no higher-priority PCB found
128         if (!q_ptr->head) {
129             q_ptr->head = q_ptr->tail = ptr;
130         } else {
131             q_ptr->tail->next = ptr;
132             ptr->prev = q_ptr->tail;
133             q_ptr->tail = ptr;
134         }
135     }
136     // Else insert before cur
137     else {
138         ptr->next = cur;
139         ptr->prev = cur->prev;
140         if (cur->prev) cur->prev->next = ptr;
141         else q_ptr->head = ptr;
142         cur->prev = ptr;
143     }
144 } else {
145     // For non-ready queues: insert new PCB at the tail (FIFO order).
146
147     // Case 1: The queue is empty (no head yet).
148     if (!q_ptr->head) {
149         // Make the new PCB both the head and the tail of the queue,
150         // since it's the only element.
151         q_ptr->head = q_ptr->tail = ptr;
152     } else {
153         // Case 2: Queue already has at least one PCB.
154
155         // Link the current tail's 'next' pointer to the new PCB.
156         q_ptr->tail->next = ptr;
157
158         // Link the new PCB's 'prev' pointer back to the old tail.
159         ptr->prev = q_ptr->tail;
160
161         // Update the queue's tail to point to the new PCB.
162         // Now 'ptr' becomes the new last element.
163         q_ptr->tail = ptr;
164     }
165 }
166
167 }
168
169 int pcb_remove(struct pcb* ptr) {
170     if (!ptr) return -1;
171
172     struct pcb_queue* q_ptr = NULL;
173
174     // Determine which queue the PCB belongs to
175     if (ptr->dispatch_state == DISPATCH_SUSPENDED) {
176         if (ptr->execution_state == STATE_BLOCKED)
177             q_ptr = &suspended_blocked_queue;

```

```
178     else
179         q_ptr = &suspended_ready_queue;
180     } else {
181         if (ptr->execution_state == STATE_BLOCKED)
182             q_ptr = &blocked_queue;
183         else
184             q_ptr = &ready_queue;
185     }
186
187     // Nothing to remove if queue is empty
188     if (!q_ptr || !q_ptr->head) return -1;
189
190     // Relink neighbors to bypass 'ptr'
191     if (ptr->prev) ptr->prev->next = ptr->next;
192     else q_ptr->head = ptr->next;
193
194     if (ptr->next) ptr->next->prev = ptr->prev;
195     else q_ptr->tail = ptr->prev;
196
197     // Clear links
198     ptr->next = ptr->prev = NULL;
199
200     return 0;
201 }
202
```

```
1 bits 32
2 global serial_isr
3
4 extern serial_interrupt      ; void serial_interrupt(void);
5
6 serial_isr:
7     ; Save general-purpose registers first
8     pusha
9
10    ; Save segment registers
11    push ds
12    push es
13    push fs
14    push gs
15
16    ; Ensure we have the kernel data segment loaded
17    mov ax, 0x10
18    mov ds, ax
19    mov es, ax
20    mov fs, ax
21    mov gs, ax
22
23    ; Call the C-level serial interrupt handler
24    call serial_interrupt
25
26    ; Restore segment registers
27    pop gs
28    pop fs
29    pop es
30    pop ds
31
32    ; Restore general-purpose registers
33    popa
34
35    iret
36
```

```
1 #ifndef MPX_INTERRUPTS_H
2 #define MPX_INTERRUPTS_H
3
4 #include <stdint.h>
5 #include <stddef.h>
6 #include <sys_req.h>
7 #include <stdlib.h>
8
9 #include <mpx/device.h>
10
11 #define RING_BUFFER_SIZE 100
12
13 /**
14  * @file mpx/interrupts.h
15  * @brief Kernel functions related to software and hardware interrupts
16  */
17
18 /** Disable interrupts */
19 #define cli() __asm__ volatile ("cli")
20
21 /** Enable interrupts */
22 #define sti() __asm__ volatile ("sti")
23
24 /* Core interrupt setup */
25 void irq_init(void);
26 void pic_init(void);
27 void idt_init(void);
28 void idt_install(int vector, void (*handler)(void *));
29
30 /* ----- Device / I/O state enums ----- */
31
32 typedef enum {
33     DEV_CLOSED = 0,
34     DEV_IDLE,
35     DEV_READING,
36     DEV_WRITING
37 } dev_state_t;
38
39 typedef enum {
40     open = 1,
41     closed = 0
42 } open_status;
43
44
45 typedef enum {
46     allocated = 1,
47     unallocated = 0
48 } allocation_status;
49
50 typedef enum {
51     no_event = 0,
52     unhandled_event = 1
53 } event_flag;
54
55 typedef enum {
56     IO_READ = 0,
57     IO_WRITE
58 } io_op_t;
59
60 /* Forward declarations */
```

```

61 struct pcb;
62 struct dcb;
63
64 /**
65  * @struct iocb
66  * @brief I/O Control Block for one read or write request.
67 */
68 typedef struct iocb {
69     struct pcb *proc;
70     struct dcb *dev;
71     io_op_t      op;
72
73     char        *buf;
74     size_t       size;
75     size_t       byte_count;
76
77     struct iocb *next;
78 } iocb_t;
79
80 /**
81  * @struct dcb
82  * @brief Device Control Block for a serial port.
83 */
84 typedef struct dcb {
85     device          device;
86     dev_state_t    state;
87     open_status    open_status;
88     allocation_status allocation_status;
89     event_flag     event_flag;
90
91     /* Ring buffer for inputs when no read is active */
92     char   ring_buffer[RING_BUFFER_SIZE];
93     size_t ring_head;
94     size_t ring_tail;
95     size_t ring_count;
96
97     /* I/O queue for this device */
98     iocb_t *q_head;
99     iocb_t *q_tail;
100    iocb_t *current;
101
102    uint16_t base_port;
103 } dcb_t;
104
105
106 /**
107  * @brief Get the DCB for a given serial device.
108 */
109 dcb_t *get_dcb_for_device(device dev);
110
111 /**
112  * @brief Open a serial device for interrupt-driven I/O.
113  *
114  * Error codes follow the R6 notes:
115  * -100 invalid device
116  * -101 invalid event flag pointer (not used here, but reserved)
117  * -102 invalid baud (no divisor)
118  * -103 already open
119 */
120 int serial_open(device dev, int speed);

```

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

```

1 #include <string.h>
2 #include <mpx/io.h>
3 #include <mpx/serial.h>
4 #include <mpx/interrupts.h>
5 #include <memory.h>
6 #include <itoa.h>
7
8 extern void serial_isr(void *);
9
10 /***** PIC + UART DEFINES *****/
11
12 #define PIC_CMD 0x20
13 #define PIC_MASK 0x21
14 #define PIC_EOI 0x20
15
16 // UART register offsets (must match serial.c)
17 enum uart_registers {
18     RBR = 0, // Receive Buffer
19     THR = 0, // Transmitter Holding
20     DLL = 0, // Divisor Latch LSB
21     IER = 1, // Interrupt Enable
22     DLM = 1, // Divisor Latch MSB
23     IIR = 2, // Interrupt Identification
24     FCR = 2, // FIFO Control
25     LCR = 3, // Line Control
26     MCR = 4, // Modem Control
27     LSR = 5, // Line Status
28     MSR = 6, // Modem Status
29     SCR = 7, // Scratch
30 };
31
32 /***** DCBs PER DEVICE *****/
33
34 /* Standard PC base ports */
35 #define COM1_BASE 0x3F8
36 #define COM2_BASE 0x2F8
37 #define COM3_BASE 0x3E8
38 #define COM4_BASE 0x2E8
39
40 static dcb_t com1_dcb = { .device = COM1, .state = DEV_CLOSED, .open_status =
41     closed, .allocation_status = unallocated, .event_flag = no_event, .base_port =
42     COM1_BASE };
43 static dcb_t com2_dcb = { .device = COM2, .state = DEV_CLOSED, .open_status =
44     closed, .allocation_status = unallocated, .event_flag = no_event, .base_port =
45     COM2_BASE };
46 static dcb_t com3_dcb = { .device = COM3, .state = DEV_CLOSED, .open_status =
47     closed, .allocation_status = unallocated, .event_flag = no_event, .base_port =
48     COM3_BASE };
49 static dcb_t com4_dcb = { .device = COM4, .state = DEV_CLOSED, .open_status =
50     closed, .allocation_status = unallocated, .event_flag = no_event, .base_port =
51     COM4_BASE };
52
53 dcb_t *get_dcb_for_device(device dev)
54 {
55     switch (dev) {
56         case COM1: return &com1_dcb;
57         case COM2: return &com2_dcb;
58         case COM3: return &com3_dcb;
59         case COM4: return &com4_dcb;
60         default:    return NULL;

```

```

53     }
54 }
55
56 static int irq_for_device(device dev)
57 {
58     switch (dev) {
59     case COM1: return 4;
60     case COM2: return 3;
61     case COM3: return 4;    // typical PC mapping
62     case COM4: return 3;
63     default:   return -1;
64 }
65 }
66
67 /***** PIC HELPERS *****/
68
69 static void pic_mask_irq(int irq)
70 {
71     if (irq < 0) return;
72     cli();
73     uint8_t mask = inb(PIC_MASK);
74     mask |= (1 << irq);      // 1 = disable
75     outb(PIC_MASK, mask);
76     sti();
77 }
78
79 static void pic_unmask_irq(int irq)
80 {
81     if (irq < 0) return;
82     cli();
83
84     /* debug prints removed */
85     uint8_t mask = inb(PIC_MASK);
86     mask &= ~(1 << irq);    // 0 = enable
87     outb(PIC_MASK, mask);
88     sti();
89 }
90
91 /***** RING BUFFER HELPERS *****/
92
93 static void ring_put(dcb_t *dcb, char c)
94 {
95     if (dcb->ring_count >= RING_BUFFER_SIZE) {
96         // Overflow: drop char (simple policy)
97         return;
98     }
99     dcb->ring_buffer[dcb->ring_head] = c;
100    dcb->ring_head = (dcb->ring_head + 1) % RING_BUFFER_SIZE;
101    dcb->ring_count++;
102}
103
104 static int ring_get(dcb_t *dcb, char *out)
105 {
106     if (dcb->ring_count == 0) {
107         return 0;
108     }
109     *out = dcb->ring_buffer[dcb->ring_tail];
110     dcb->ring_tail = (dcb->ring_tail + 1) % RING_BUFFER_SIZE;
111     dcb->ring_count--;
112     return 1;

```

```

113 }
114
115
116 //***** OPEN / CLOSE *****/
117
118 int serial_open(device dev, int speed)
119 {
120     dcb_t *dcb = get_dcb_for_device(dev);
121     if (!dcb) {
122         return -100; // invalid device
123     }
124
125     int     irq  = irq_for_device(dev);
126     uint16_t base = dcb->base_port;
127
128     // Already open?
129     if (dcb->open_status == open) {
130         return -103;
131     }
132
133     // Baud divisor must divide 115200
134     if (speed <= 0 || (115200 % speed) != 0) {
135         return -102;
136     }
137
138     // Disable UART interrupts
139     outb(base + IER, 0x00);
140
141     // Set baud rate
142     uint16_t divisor      = (uint16_t)(115200 / speed);
143     uint8_t  divisor_low   = divisor & 0x00FF;
144     uint8_t  divisor_high = (divisor >> 8) & 0x00FF;
145
146     outb(base + LCR, 0x80);           // DLAB = 1
147     outb(base + DLL, divisor_low);
148     outb(base + DLM, divisor_high);
149     outb(base + LCR, 0x03);           // 8 bits, no parity, 1 stop, DLAB=0
150
151     // enable interrupts
152     outb(base + MCR, 0x08);
153
154     // Clear pending status
155     (void)inb(base + LSR);
156     (void)inb(base + RBR);
157     (void)inb(base + IIR);
158     (void)inb(base + MSR);
159
160     // Initialize DCB
161     memset(dcb->ring_buffer, 0, sizeof(dcb->ring_buffer));
162     dcb->ring_head      = 0;
163     dcb->ring_tail      = 0;
164     dcb->ring_count     = 0;
165     dcb->device          = dev;
166     dcb->open_status     = open;
167     dcb->allocation_status = allocated;
168     dcb->state            = DEV_IDLE;
169     dcb->event_flag       = no_event;
170     dcb->q_head           = NULL;
171     dcb->q_tail           = NULL;
172     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         }

```

```

233     // Stop on CR/LF or full buffer
234     if (c == '\r' || c == '\n' || io->byte_count >= io->size - 1) {
235         // Remove CR/LF from stored buffer
236         if (io->byte_count > 0 &&
237             (io->buf[io->byte_count - 1] == '\r' ||
238              io->buf[io->byte_count - 1] == '\n')) {
239             io->byte_count--;
240         }
241         io->buf[io->byte_count] = '\0';
242
243         dcb->state      = DEV_IDLE;
244         dcb->event_flag = unhandled_event;
245     }
246 } else {
247     // No active read - stash into ring buffer
248     ring_put(dcb, c);
249 }
250 }
251 }
252
253 static void serial_output_interrupt(dcb_t *dcb)
254 {
255     uint16_t base = dcb->base_port;
256     iocb_t *io    = dcb->current;
257
258     if (!io || dcb->state != DEV_WRITING) {
259         // No active write; interrupts are off
260         uint8_t ier = inb(base + IER);
261         ier &= ~0x02;
262         outb(base + IER, ier);
263         return;
264     }
265
266     if (io->byte_count < io->size) {
267         outb(base + THR, (uint8_t)io->buf[io->byte_count++]);
268     }
269
270     if (io->byte_count >= io->size) {
271         uint8_t ier = inb(base + IER);
272         ier &= ~0x02;
273         outb(base + IER, ier);
274
275         dcb->state      = DEV_IDLE;
276         dcb->event_flag = unhandled_event;
277     }
278 }
279
280 /*****SERIAL ISR*****/
281
282 void serial_interrupt(void)
283 {
284     cli();
285     device devs[] = { COM1, COM2, COM3, COM4 };
286
287     for (int i = 0; i < 4; ++i) {
288         dcb_t *dcb = get_dcb_for_device(devs[i]);
289         if (!dcb || dcb->open_status != open)
290             continue;
291
292         uint16_t base = dcb->base_port;

```

```

293     uint8_t iir = inb(base + IIR);
294
295     // Bit 0 == 1 => no interrupt pending
296     if (iir & 0x01)
297         continue;
298
299     // Bits 3:1 = interrupt reason
300     switch (iir & 0x0E) {
301     case 0x04: // RX data available
302     case 0x0C: // RX timeout
303         serial_input_interrupt(dcb);
304         break;
305
306     case 0x02: // TX
307         serial_output_interrupt(dcb);
308         break;
309
310     default:
311         // Clear
312         (void)inb(base + LSR);
313         (void)inb(base + RBR);
314         (void)inb(base + MSR);
315         break;
316     }
317 }
318
319 // End Of Interrupt to PIC
320 outb(PIC_CMD, PIC_EOI);
321
322 sti();
323 }
324
325 /***** Read And Write *****/
326
327 int serial_read(device dev, char *buf, size_t len)
328 {
329     dcb_t *dcb = get_dcb_for_device(dev);
330     if (!dcb || dcb->open_status != open)
331         return -301; // invalid or not open
332
333     if (!buf || len == 0)
334         return -302; // invalid buffer/len
335
336     if (dcb->state != DEV_IDLE && dcb->state != DEV_READING)
337         return -304; // busy
338
339     iocb_t *io = dcb->current;
340     if (!io)
341         return -304; // scheduler error: no IOCB present
342
343     io->buf      = buf;
344     io->size     = len;
345     io->byte_count = 0;
346
347     dcb->state    = DEV_READING;
348     dcb->event_flag = no_event;
349
350     // Use any chars already in the ring buffer
351     char c;
352     while (io->byte_count < io->size - 1 && ring_get(dcb, &c)) {

```

```

353     if (c == '\r' || c == '\n') {
354         // End of line; don't store CR/LF
355         io->buf[io->byte_count] = '\0';
356         dcb->state          = DEV_IDLE;
357         dcb->event_flag      = unhandled_event;
358         return 0;
359     }
360     io->buf[io->byte_count++] = c;
361 }
362
363 // If buffer filled before newline, complete immediately
364 if (io->byte_count >= io->size - 1) {
365     io->buf[io->byte_count] = '\0';
366     dcb->state          = DEV_IDLE;
367     dcb->event_flag      = unhandled_event;
368 }
369
370 return 0;
371 }
372
373 int serial_write(device dev, char *buf, size_t len)
374 {
375     dcb_t *dcb = get_dcb_for_device(dev);
376     if (!dcb || dcb->open_status != open)
377         return -401;           // invalid or not open
378
379     if (!buf || len == 0)
380         return -402;           // invalid buffer/len
381
382     // Device must be idle or already writing (io_schedule sets this up)
383     if (dcb->state != DEV_IDLE && dcb->state != DEV_WRITING)
384         return -404;           // busy
385
386     // I/O scheduler must have set up current IOCB
387     iocb_t *io = dcb->current;
388     if (!io)
389         return -404;           // scheduler error: no IOCB present
390
391     // Initialize IOCB for this write
392     io->buf        = buf;
393     io->size       = len;
394     io->byte_count = 0;
395
396     dcb->state      = DEV_WRITING;
397     dcb->event_flag = no_event;
398
399     uint16_t base = dcb->base_port;
400
401 /*
402  * Kick out the FIRST byte only (per R6 spec).
403  * Do NOT spin in a while-loop forever; just send if THR is empty.
404  * If it's not empty yet, the next TX interrupt will pick it up.
405  */
406     uint8_t lsr = inb(base + LSR);
407     if ((lsr & 0x20) != 0 && io->size > 0) {
408         outb(base + THR, (uint8_t)io->buf[0]);
409         io->byte_count = 1;
410     }
411
412 /* Enable transmit holding register empty interrupts (bit 1 in IER) */

```

```
413     uint8_t ier = inb(base + IER);
414     ier |= 0x02;                      // enable TX interrupt
415     outb(base + IER, ier);
416
417     // Asynchronous: ISR will finish the transfer and set event_flag.
418     return 0;
419 }
420
```

```
1 #ifndef MPX_SERIAL_H
2 #define MPX_SERIAL_H
3
4 #include <stddef.h>
5 #include <mpx/device.h>
6
7 /**
8  * @file mpx/serial.h
9  * @brief Kernel functions and constants for handling serial I/O
10 */
11
12 /**
13  * @brief Performs trimming and validation on a command before adding to history.
14  * @param buffer The command string to clean
15  * @return 0 on success, -1 if empty/invalid
16  */
17 int history_errors(char *buffer);
18
19 /**
20  * @brief Adds a command to the history buffer and performs shifting when the buffer
21  * is full.
22  * @param buffer The command to add
23  * @return 0 always
24  */
25 int history_handler(char *buffer);
26
27 /**
28  * @brief Initializes devices for user input and output
29  * @param device A serial port to initialize (COM1, COM2, COM3, or COM4)
30  * @return 0 on success, non-zero on failure
31  */
32 int serial_init(device dev);
33
34 /**
35  * @brief Writes a buffer to a serial port
36  * @param device The serial port to output to
37  * @param buffer A pointer to an array of characters to output
38  * @param len The number of bytes to write
39  * @return The number of bytes written
40  */
41 int serial_out(device dev, const char *buffer, size_t len);
42
43 /**
44  * @brief Reads a string from a serial port
45  * @param device The serial port to read data from
46  * @param buffer A buffer to write data into as it is read from the serial port
47  * @param count The maximum number of bytes to read
48  * @return The number of bytes read on success, a negative number on failure
49  */
50 int serial_poll(device dev, char *buffer, size_t len);
51
52 int serial_open(device dev, int speed);
53 int serial_close(device dev);
54 int serial_write(device dev, char *buf, size_t len);
55 int serial_read(device dev, char *buf, size_t len);
56
57 #endif
```

```

1 #include <mpx/io.h>
2 #include <mpx/serial.h>
3 #include <sys_req.h>
4 #include <string.h>
5 #include <memory.h>
6 #include <mpx/interrupts.h>
7
8 ////////////// DEFINE VARIABLES /////////////
9
10#define HISTORY_SIZE 5
11#define MAX_CMD_LEN 128
12
13static char command_history[HISTORY_SIZE][MAX_CMD_LEN];
14static int history_count = 0;
15static int history_index = 0;
16
17enum uart_registers {
18    RBR = 0, // Receive Buffer
19    THR = 0, // Transmitter Holding
20    DLL = 0, // Divisor Latch LSB
21    IER = 1, // Interrupt Enable
22    DLM = 1, // Divisor Latch MSB
23    IIR = 2, // Interrupt Identification
24    FCR = 2, // FIFO Control
25    LCR = 3, // Line Control
26    MCR = 4, // Modem Control
27    LSR = 5, // Line Status
28    MSR = 6, // Modem Status
29    SCR = 7, // Scratch
30};
31
32static int initialized[4] = { 0 };
33
34//////////// HISTORY FUNCTIONS /////////////
35
36int history_errors(char *buffer) {
37    // ensure buffer is not empty
38    if (buffer == NULL || strlen(buffer) == 0) {
39        return -1;
40    }
41
42    // trim leading spaces
43    size_t index = 0;
44    while (buffer[index] == ' ' && index < strlen(buffer)) {
45        index++;
46    }
47    if (index > 0) {
48        size_t j = 0;
49        while (buffer[index + j] != '\0' && (index + j) < strlen(buffer)) {
50            buffer[j] = buffer[index + j];
51            j++;
52        }
53        buffer[j] = '\0';
54    }
55
56    // trim trailing spaces
57    size_t end = strlen(buffer);
58    while (end > 0 && buffer[end - 1] == ' ') {
59        end--;
60    }

```

```

61 buffer[end] = '\0';
62
63 // ignore if buffer is now empty after trimming
64 if (strlen(buffer) == 0) {
65     return -1;
66 }
67
68 return 0;
69 }
70
71 int history_handler(char *buffer) {
72     // handle input based on array size
73     if (history_count < HISTORY_SIZE) {
74         strncpy(command_history[history_count], buffer, MAX_CMD_LEN - 1);
75         command_history[history_count][MAX_CMD_LEN - 1] = '\0';
76         history_count++;
77     }
78
79     // buffer is full
80     else {
81         for (int i = 1; i < HISTORY_SIZE; i++) {
82             strncpy(command_history[i - 1], command_history[i], MAX_CMD_LEN - 1);
83             command_history[i - 1][MAX_CMD_LEN - 1] = '\0';
84         }
85         strncpy(command_history[HISTORY_SIZE - 1], buffer, MAX_CMD_LEN - 1);
86         command_history[HISTORY_SIZE - 1][MAX_CMD_LEN - 1] = '\0';
87     }
88
89     history_index = history_count;
90
91     return 0;
92 }
93
94
95
96 ////////////// SERIAL FUNCTIONS /////////////
97
98 static int serial_devno(device dev)
99 {
100     switch (dev) {
101     case COM1: return 0;
102     case COM2: return 1;
103     case COM3: return 2;
104     case COM4: return 3;
105     }
106     return -1;
107 }
108
109 int serial_init(device dev)
110 {
111     int dno = serial_devno(dev);
112     if (dno == -1) {
113         return -1;
114     }
115     outb(dev + IER, 0x00); // disable interrupts
116     outb(dev + LCR, 0x80); // set line control register
117     outb(dev + DLL, 115200 / 9600); // set bsd least sig bit
118     outb(dev + DLM, 0x00); // brd most significant bit
119     outb(dev + LCR, 0x03); // lock divisor; 8bits, no parity, one stop
120     outb(dev + FCR, 0xC7); // enable fifo, clear, 14byte threshold

```

```

121    outb(dev + MCR, 0x0B); //enable interrupts, rts/dsr set
122    (void)inb(dev); //read bit to reset port
123    initialized[dno] = 1;
124    return 0;
125 }
126
127 int serial_out(device dev, const char *buffer, size_t len)
128 {
129     int dno = serial_devno(dev);
130     if (dno == -1 || initialized[dno] == 0) {
131         return -1;
132     }
133     for (size_t i = 0; i < len; i++) {
134         outb(dev, buffer[i]);
135     }
136     return (int)len;
137 }
138
139
140 int serial_poll(device dev, char *buffer, size_t len)
141 {
142     // This function must properly handle the following based off ASCII:
143     // - alphanumerics (a-z, A-Z, 0-9) and regular special keys (/!#$)
144     // - space, backspace, delete, arrow keys (up down left right)
145     // - carriage returns (\r) and new lines (\n)
146
147     // define some ASCII characters
148     const char ESC_KEY = 27;
149     const char BACKSPACE = 8;
150     const char DELETE = 127;
151
152     // format the serial terminal to look like a penguin
153     sys_req(WRITE, COM1, "(> ", 6);
154
155     // read the input while the buffer is not full
156     int index = 0;
157     char input;
158     char specialKey;
159     int endOfLine = 0;
160     while (index < (int)len - 1) {
161         input = inb(dev);
162
163         // alphanumerics, spaces, and special keys
164         if ((input >= 'a' && input <= 'z') ||
165             (input >= 'A' && input <= 'Z') ||
166             (input >= '0' && input <= '9') ||
167             (input >= '!\' && input <= '/') ||
168             (input == ' ') || (input == ':')) {
169
170             buffer[index] = input;
171             outb(dev, input); // echo
172             index++;
173             endOfLine++;
174         }
175
176         // backspace and delete
177         else if ((input == BACKSPACE || input == DELETE) && (index > 0)) {
178             // remove the last character from the buffer
179             index--;
180             endOfLine--;

```

```

181     buffer[index] = '\0';
182
183     // erase the last character from the terminal
184     outb(dev, '\b');
185     outb(dev, ' ');
186     outb(dev, '\b');
187 }
188
189 // CR and LF (user is done)
190 else if (input == '\n' || input == '\r') {
191     // end the line
192     buffer[endOfLine] = '\0';
193     outb(dev, input);
194
195     // handle command history and reset to "empty line" position
196     history_handler(buffer);
197     history_index = history_count;
198
199     return endOfLine;
200 }
201
202 // handles multi-byte characters that use the escape character
203 else if (input == ESC_KEY) {
204     input = inb(dev);
205     while (1) {
206         specialKey = inb(dev);
207
208         // command history error checking
209         if (history_index < 0) history_index = 0;
210         if (history_index > history_count) history_index = history_count;
211
212         // UP arrow
213         if (specialKey == 'A' && history_count > 0) {
214             // load the last command
215             if (history_index > 0 && history_index < HISTORY_SIZE) history_index--;
216
217             // move the cursor to the end and clear the line
218             while (index < endOfLine) {
219                 sys_req(WRITE, COM1, "\033[C", 3);
220                 index++;
221             }
222             for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);
223
224             // write the last command and set new index and endOfLine
225             strncpy(buffer, command_history[history_index], MAX_CMD_LEN - 1);
226             buffer[MAX_CMD_LEN - 1] = '\0';
227             sys_req(WRITE, COM1, buffer, strlen(buffer));
228             endOfLine = strlen(buffer);
229             index = endOfLine;
230
231             break;
232         }
233
234         // DOWN arrow
235         else if (specialKey == 'B') {
236             // default behavior for command history
237             if (history_index < history_count - 1) {
238                 // load next command
239                 history_index++;
240

```

```

241 // move the cursor to the end and clear the line
242 while (index < endOfLine) {
243     sys_req(WRITE, COM1, "\033[C", 3);
244     index++;
245 }
246 for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);
247
248 // write the next command and set new index and endOfLine
249 strncpy(buffer, command_history[history_index], MAX_CMD_LEN - 1);
250 buffer[MAX_CMD_LEN - 1] = '\0';
251 sys_req(WRITE, COM1, buffer, strlen(buffer));
252 endOfLine = strlen(buffer);
253 index = endOfLine;
254 }
255 // clear the line when you get to the end of the history
256 else if (history_index == history_count - 1) {
257     // clear line (move to "new" command position)
258     history_index = history_count;
259     while (index < endOfLine) {
260         sys_req(WRITE, COM1, "\033[C", 3); // move cursor right
261         index++;
262     }
263     for (int i = 0; i < endOfLine; i++) sys_req(WRITE, COM1, "\b \b", 3);
264
265 // reset the buffer, index, and endOfLine
266 buffer[0] = '\0';
267 endOfLine = 0;
268 index = 0;
269 }
270
271 break;
272 }
273
274 // RIGHT arrow (until you get to the end of the characters)
275 else if (specialKey == 'C' && index < endOfLine) {
276     sys_req(WRITE, COM1, "\033[C", 3);
277     index++;
278     break;
279 }
280
281 // LEFT arrow (until you get to the start of the characters)
282 else if (specialKey == 'D' && index != 0) {
283     sys_req(WRITE, COM1, "\033[D", 3);
284     index--;
285     break;
286 }
287
288 // DELETE key (ESC[3~)
289 else if (specialKey == '3') {
290     // ensure its the delete key
291     while (input != '~'){
292         input = inb(dev);
293     }
294
295 // delete until the last character
296 if (index < (endOfLine-1)) {
297     for (int i = index; i < endOfLine - 1; i++) {
298         buffer[i] = buffer[i + 1];
299         outb(dev, buffer[i]);
300     }

```

```
301     outb(dev, ' ');
302     for (int i = index; i < endOfLine; i++){
303         sys_req(WRITE, COM1, "\033[D", 3);
304     }
305     endOfLine--;
306 }
307 // delete the last character
308 else if (index == (endOfLine-1)){
309     buffer[index] = '\0';
310     sys_req(WRITE, COM1, "\033[D", 4);
311     endOfLine--;
312 }
313 else {
314     // ignore sequence
315 }
316 break;
317 }
318 }
319 } // end of multi byte chars
320
321 }
322
323 // returns -1 if buffer overflow or when there is an error
324 buffer[index] = '\0';
325 return -1;
326 }
327 }
```

```
1 bits 32
2 global sys_call_isr
3
4 ;;; System call interrupt handler. To be implemented in Module R3.
5 extern sys_call      ; The C function that sys_call_isr will call
6 sys_call_isr:
7     pusha          ; Pushes general purpose registers
8
9     push ds        ; Pushes remaining registers
10    push es
11    push fs
12    push gs
13
14    mov eax, esp
15    push eax
16    call sys_call
17
18    mov esp, eax
19
20    pop gs         ; Pops last 4 registers pushed
21    pop fs
22    pop es
23    pop ds
24
25    popa          ; Pops general purpose registers
26
27    iret
28
```

```
1 #ifndef SYS_CALL_H
2 #define SYS_CALL_H
3 #include <pcb.h>
4 #include <sys_req.h>
5
6 /**
7  * @file sys_call.h
8  * @brief Handles context switching when sys_req function is used
9  * @param curContext Pointer to the current context
10 */
11 struct context *sys_call(struct context *curContext);
12
13 /**
14  * @brief Helper that returns the current process
15  */
16 struct pcb* sys_get_current_process(void);
17
18 #endif
19
```

```

1 #include "comhand.h"
2 #include <string.h>
3 #include <itoa.h>
4 #include <sys_req.h>
5 #include "help.h"
6 #include "exit.h"
7 #include "version.h"
8 #include "clock.h"
9 #include "showPCB.h"
10 #include "setPriority.h"
11 #include "ready.h"
12 #include "init.h"
13 #include "block.h"
14 #include "yield.h"
15 #include <mpx/interrupts.h>
16 #include <mpx/serial.h>
17 #include <memory.h>
18 #include <mpx/device.h>
19 #include <pcb.h>
20
21 /* Pointer for current process */
22 static struct pcb *curProc = NULL;
23
24 /* Pointer to system stack/kernel context */
25 static struct context *sysStackPtr = NULL;
26
27 /* Helper to return current process */
28 struct pcb *sys_get_current_process(void)
29 {
30     return curProc;
31 }
32
33 /*
34 * Scan all serial DCBs. For any with event_flag == unhandled_event:
35 * - take the IOCB from dcb->current
36 * - move the process from BLOCKED to READY
37 * - set its saved eax = #bytes transferred
38 * - free the IOCB
39 * - start the next IOCB in the queue, if any
40 */
41 static void io_handle_completions(void)
42 {
43     device devs[] = { COM1, COM2, COM3, COM4 };
44
45     for (int i = 0; i < 4; ++i) {
46         dcb_t *dcb = get_dcb_for_device(devs[i]);
47         if (!dcb || dcb->open_status != open)
48             continue;
49
50         if (dcb->event_flag == unhandled_event && dcb->current != NULL) {
51             iocb_t *io = dcb->current;
52             struct pcb *p = io->proc;
53
54             /* Wake the blocked process */
55             if (p != NULL) {
56                 pcb_remove(p); // remove from blocked queue
57                 p->execution_state = STATE_READY;
58                 pcb_insert(p); // insert into ready queue
59             }
60         }
61     }
62 }
```

```

61         if (p->contextPtr) {
62             p->contextPtr->eax = (int)io->byte_count;
63         }
64     }
65
66     /* Clear DCB */
67     dcb->current      = NULL;
68     dcb->state        = DEV_IDLE;
69     dcb->event_flag = no_event;
70
71     /* Grab next IOCB, if any */
72     iocb_t *next = NULL;
73     if (dcb->q_head) {
74         next      = dcb->q_head;
75         dcb->q_head = next->next;
76         if (!dcb->q_head)
77             dcb->q_tail = NULL;
78         next->next = NULL;
79     }
80
81     /* Free finished IOCB */
82     sys_free_mem(io);
83
84     /* Start next request, if present */
85     if (next) {
86         dcb->current      = next;
87         dcb->state        = (next->op == IO_READ) ? DEV_READING : DEV_WRITING;
88         dcb->event_flag = no_event;
89
90         if (next->op == IO_READ) {
91             serial_read(dcb->device, next->buf, next->size);
92         } else {
93             serial_write(dcb->device, next->buf, next->size);
94         }
95     }
96 }
97 }
98 */
99
100 /*
101 * io_scheduler:
102 * - Builds an IOCB for the given request.
103 * - If device idle, starts I/O immediately (serial_read/serial_write).
104 * - If busy, enqueues the IOCB.
105 *
106 * Return values:
107 * < 0 : error (immediate; caller does not block)
108 * > 0 : I/O completed synchronously, return that byte count
109 * == 0 : I/O started and will complete asynchronously; caller should BLOCK.
110 */
111 static int io_schedule(struct context *ctx)
112 {
113     op_code op    = (op_code)ctx->eax;
114     device dev   = (device)ctx->ebx;
115     char *buf    = (char *)ctx->ecx;
116     size_t len    = (size_t)ctx->edx;
117
118     dcb_t *dcb = get_dcb_for_device(dev);
119     if (!dcb || dcb->open_status != open) {
120         return (op == READ) ? -301 : -401;

```

```

121     }
122
123     if (!buf || len == 0) {
124         return (op == READ) ? -302 : -402;
125     }
126
127     /* Allocate IOCB */
128     iocb_t *io = (iocb_t *)sys_alloc_mem(sizeof(iocb_t));
129     if (!io) {
130         return -1; // out of memory
131     }
132
133     memset(io, 0, sizeof(iocb_t));
134     io->proc      = curProc;
135     io->dev       = dcb;
136     io->buf       = buf;
137     io->size      = len;
138     io->byte_count = 0;
139     io->next      = NULL;
140
141     if (op == READ) {
142         io->op = IO_READ;
143     } else if (op == WRITE) {
144         io->op = IO_WRITE;
145     } else {
146         sys_free_mem(io);
147         return -1;
148     }
149
150     /* If device idle, start immediately; else enqueue */
151     if (dcb->current == NULL && dcb->state == DEV_IDLE) {
152         dcb->current      = io;
153         dcb->event_flag   = no_event;
154         dcb->state        = (io->op == IO_READ) ? DEV_READING : DEV_WRITING;
155
156         int rc;
157         if (io->op == IO_READ)
158             rc = serial_read(dcb->device, buf, len);
159         else
160             rc = serial_write(dcb->device, buf, len);
161
162         if (rc < 0) {
163             dcb->current = NULL;
164             dcb->state   = DEV_IDLE;
165             sys_free_mem(io);
166             return rc;
167         }
168
169         /* If completed immediately */
170         if (dcb->event_flag == unhandled_event) {
171             int done = (int)io->byte_count;
172             sys_free_mem(io);
173             dcb->current      = NULL;
174             dcb->state        = DEV_IDLE;
175             dcb->event_flag   = no_event;
176             return done;
177         }
178
179         /* Otherwise, I/O is in progress; caller must block. */
180     } else {

```

```

181     /* Device busy: append to queue */
182     if (dcb->q_tail)
183         dcb->q_tail->next = io;
184     else
185         dcb->q_head = io;
186     dcb->q_tail = io;
187 }
188
189     return 0; // started, will complete asynchronously
190 }
191
192 struct context *sys_call(struct context *curContext)
193 {
194     /* Save kernel context */
195     if (curProc == NULL) {
196         sysStackPtr = curContext;
197     }
198
199     /* First: handle any completed I/O */
200     io_handle_completions();
201
202     /* Next ready process */
203     struct pcb *newProc = ready_queue.head;
204
205     /* IDLE: */
206     if (curContext->eax == IDLE) {
207         if (newProc == NULL) {
208             /* No other process ready; just return to current process */
209             if (curProc != NULL) {
210                 curContext->eax = -1;
211                 return curContext; // back to same process
212             } else {
213                 /* No processes at all; return to kernel */
214                 curContext->eax = -1;
215                 return curContext;
216             }
217         } else {
218             if (curProc == NULL) {
219                 pcb_remove(newProc);
220                 curProc = newProc;
221                 curContext->eax = -1;
222                 return newProc->contextPtr;
223             } else {
224                 curProc->contextPtr = curContext;
225                 pcb_insert(curProc);
226                 pcb_remove(newProc);
227                 curProc = newProc;
228                 curContext->eax = -1;
229                 return newProc->contextPtr;
230             }
231         }
232     }
233
234     /* EXIT: */
235     else if (curContext->eax == EXIT) {
236         if (newProc == NULL) {
237             pcb_free(curProc);
238             curProc = NULL;
239             sysStackPtr->eax = -1;
240             return sysStackPtr;

```

```

241     } else {
242         pcb_free(curProc);
243         pcb_remove(newProc);
244         curProc = newProc;
245         curContext = curProc->contextPtr;
246         curContext->eax = -1;
247         return newProc->contextPtr;
248     }
249 }
250
251 /* READ or WRITE */
252 else if (curContext->eax == READ || curContext->eax == WRITE) {
253     int rc = io_schedule(curContext);
254
255     /* Immediate error: return to caller with error code */
256     if (rc < 0) {
257         curContext->eax = rc;
258         return curContext;
259     }
260
261     /* Synchronous completion: rc is byte count */
262     if (rc > 0) {
263         curContext->eax = rc;
264         return curContext;
265     }
266
267     /*
268      * Asynchronous I/O started:
269      * - Mark current process BLOCKED
270      * - Save its context
271      * - Put it in blocked queue
272      * - Dispatch next READY process or return to kernel
273      */
274     if (curProc != NULL) {
275
276         /* Remove from whatever queue it is currently in */
277         pcb_remove(curProc);
278
279         /* Mark blocked */
280         curProc->execution_state = STATE_BLOCKED;
281         curProc->contextPtr = curContext;
282
283         /* Insert into blocked queue properly */
284         pcb_insert(curProc);
285     }
286
287     newProc = ready_queue.head;
288
289     if (newProc == NULL) {
290         // No other process ready then go idle
291         return curContext;
292     }
293     else {
294         pcb_remove(newProc);
295         curProc = newProc;
296         return newProc->contextPtr;
297     }
298 }
299
300

```

```
301     /* Any other opcode */
302     else {
303         curContext->eax = -1;
304         return curContext;
305     }
306 }
307 }
```

```

1 #ifndef MPX_VM_H
2 #define MPX_VM_H
3 #include <stddef.h>
4
5 /**
6  * @file mpx/vm.h
7  * @brief Kernel functions for virtual memory and primitive allocation
8 */
9
10 /**
11  * @brief Memory Control Block (MCB) structure.
12 */
13 struct mcb{
14     int startAddr;
15     int size;
16     struct mcb* next;
17     struct mcb* prev;
18     // Only needed if doing single list
19     // enum alloc_stat alloc_stat;
20 };
21
22 struct mcb_list{
23     struct mcb* head;
24     struct mcb* tail;
25 };
26
27 extern struct mcb_list mem_allocated_list;
28 extern struct mcb_list mem_free_list;
29
30 /**
31  Allocates memory from a primitive heap.
32  @param size The size of memory to allocate
33  @param align If non-zero, align the allocation to a page boundary
34  @param phys_addr If non-NULL, a pointer to a pointer that will
35                  hold the physical address of the new memory
36  @return The newly allocated memory
37 */
38 void *kmalloc(size_t size, int align, void **phys_addr);
39
40 /**
41  Initializes the kernel page directory and initial kernel heap area.
42  Performs identity mapping of the kernel frames such that the virtual
43  addresses are equivalent to the physical addresses.
44 */
45 void vm_init(void);
46
47 /**
48  * @brief Initializes the heap with a given size.
49  * @param size The total size of the heap to initialize.
50  */
51 void initialize_heap(size_t size);
52
53 /**
54  * @brief Allocates a block of memory of the requested size.
55  * @param req_size The size of memory to allocate.
56  * @return Pointer to the allocated memory, or NULL on failure.
57  */
58 void *allocate_memory(size_t req_size);
59
60 /**

```

```
61 * @brief Frees a previously allocated block of memory.  
62 * @param ptr Pointer to the memory block to free.  
63 * @return 0 on success, -1 on failure.  
64 */  
65 int free_memory(void *ptr);  
66  
67 #endif  
68
```

```

1 #include <stddef.h>
2 #include <stdint.h>
3 #include <string.h>
4 #include <mpx/vm.h>
5 #include <mpx/serial.h>
6 #include <sys_req.h>
7
8 struct mcb_list mem_allocated_list = { NULL, NULL };
9 struct mcb_list mem_free_list = { NULL, NULL };
10
11 void initialize_heap(size_t size){
12     void *addr = kmalloc(size, 0, NULL);
13     if (!addr) return;
14
15     // Place the initial MCB at the start of the returned region.
16     struct mcb* initial_mcb = (struct mcb*)addr;
17
18     // Data address starts immediately after the MCB
19     initial_mcb->startAddr = (int)((uintptr_t)addr + sizeof(struct mcb));
20
21     // Usable size is the total minus the MCB header (16 bytes)
22     initial_mcb->size = (int)(size - sizeof(struct mcb));
23     initial_mcb->next = NULL;
24     initial_mcb->prev = NULL;
25
26     // Start lists as free
27     mem_free_list.head = mem_free_list.tail = initial_mcb;
28     mem_allocated_list.head = mem_allocated_list.tail = NULL;
29 }
30
31 static void unlink_mcb(struct mcb_list *list, struct mcb *node){
32     if (!list || !node) return;
33     if (node->prev) node->prev->next = node->next;
34     else list->head = node->next;
35
36     if (node->next) node->next->prev = node->prev;
37     else list->tail = node->prev;
38     // Remove MCB from a list
39     node->next = node->prev = NULL;
40 }
41
42 static void append_mcb(struct mcb_list *list, struct mcb *node){
43     if (!list || !node) return;
44     node->next = NULL;
45     node->prev = list->tail;
46     if (list->tail) list->tail->next = node;
47     else list->head = node;
48     // Add MCB to tail of a list
49     list->tail = node;
50 }
51
52 void *allocate_memory(size_t req_size){
53     if (req_size == 0) return NULL;
54
55     // First Fit - Look for the first block that fits
56     struct mcb *cur = mem_free_list.head;
57     while (cur && (size_t)cur->size < req_size) cur = cur->next;
58     if (!cur) return NULL;
59
60     // If block is large enough, split it

```

```

61 size_t remaining = (size_t)cur->size - req_size;
62 if (remaining > sizeof(struct mcb)){
63
64     // Create MCB for remaining free block
65     uintptr_t alloc_data_addr = (uintptr_t)cur->startAddr;
66     uintptr_t new_mcb_addr = alloc_data_addr + req_size;
67     struct mcb *new_mcb = (struct mcb *)new_mcb_addr;
68
69     // Set up the Free MCB
70     new_mcb->startAddr = (int)(new_mcb_addr + sizeof(struct mcb));
71     new_mcb->size = (int)(remaining - sizeof(struct mcb));
72
73     // Replace cur in free list with the new MCB
74     new_mcb->prev = cur->prev;
75     new_mcb->next = cur->next;
76     if (cur->prev) cur->prev->next = new_mcb;
77     else mem_free_list.head = new_mcb;
78     if (cur->next) cur->next->prev = new_mcb;
79     else mem_free_list.tail = new_mcb;
80
81     // Shrink cur to allocated size
82     cur->size = (int)req_size;
83     cur->next = cur->prev = NULL;
84
85     // Add cur to allocated list
86     append_mcb(&mem_allocated_list, cur);
87
88     // Set pointer to unallocated space
89     return (void *) (uintptr_t) cur->startAddr;
90 } else {
91     // If not enough room then allocate the whole block
92     // Remove from free list
93     unlink_mcb(&mem_free_list, cur);
94
95     // Add to allocated list
96     append_mcb(&mem_allocated_list, cur);
97
98     return (void *) (uintptr_t) cur->startAddr;
99 }
100 }
101
102 int free_memory(void *ptr){
103     if (!ptr) return -1;
104
105     // Convert user pointer to MCB pointer
106     struct mcb *m = (struct mcb *)((uintptr_t)ptr - sizeof(struct mcb));
107
108     // Check if MCB is in allocated list
109     struct mcb *cur = mem_allocated_list.head;
110     while (cur && cur != m)
111         cur = cur->next;
112
113     if (!cur)
114         return -1; // Not found → invalid free()
115
116     // If found remove from allocated list
117     unlink_mcb(&mem_allocated_list, m);
118
119     // Insert into free list in order
120     struct mcb *f = mem_free_list.head;

```

```

121     struct mcb *prev = NULL;
122
123     while (f && (uintptr_t)f < (uintptr_t)m) {
124         prev = f;
125         f = f->next;
126     }
127
128     // Insert between two blocks
129     m->prev = prev;
130     m->next = f;
131     if (prev) prev->next = m;
132     else mem_free_list.head = m;
133
134     if (f) f->prev = m;
135     else mem_free_list.tail = m;
136
137     // Merge with the previous block if adjacent
138     if (m->prev) {
139         uintptr_t prev_end = (uintptr_t)m->prev->startAddr + m->prev->size;
140         uintptr_t m_addr = (uintptr_t)m->startAddr - sizeof(struct mcb);
141
142         if (prev_end == m_addr) {
143             m->prev->size += sizeof(struct mcb) + m->size;
144
145             // After merge remove the front block (m) from the list
146             if (m->next) m->next->prev = m->prev;
147             else mem_free_list.tail = m->prev;
148
149             m->prev->next = m->next;
150
151             // After merge use prev block as the whole merged mcb
152             m = m->prev;
153         }
154     }
155
156     // Merge with the next block if adjacent
157     if (m->next) {
158         uintptr_t m_end = (uintptr_t)m->startAddr + m->size;
159         uintptr_t next_addr = (uintptr_t)m->next;
160
161         if (m_end == next_addr) {
162             struct mcb *next = m->next;
163
164             // Merge next into m
165             m->size += sizeof(struct mcb) + next->size;
166
167             // After merge remove the next block from the list
168             m->next = next->next;
169             if (next->next) next->next->prev = m;
170             else mem_free_list.tail = m;
171         }
172     }
173
174     return 0;
175 }
176
177
178

```

```
1 #ifndef MCB_ALLOCATE_H
2 #define MCB_ALLOCATE_H
3
4 #include <stddef.h>
5
6 /**
7  * @file mcb/allocate.h
8  * @brief Shell command handler for allocating memory via allocate_memory() and
9  * printing the resulting address in hex.
10 */
11 /**
12  * @brief Prints help/usage information for the `allocate` command.
13  */
14 void allocate_help(void);
15
16 /**
17  * @brief Executes the allocate command by calling the kernel function.
18  * @param args The argument string passed by the shell: size between 1-50,000 or
19  * help.
20  * @return 0 on success, -1 on error.
21 */
22 int allocate_command(char *args);
23#endif
24
```

```

1 #include <mpx/serial.h>
2 #include <sys_req.h>
3 #include "mcb/allocate.h"
4 #include <mpx/vm.h>
5 #include <string.h>
6 #include <stdlib.h>
7
8 static void print_hex(int num) {
9     // setup variables
10    char hex_chars[] = "0123456789ABCDEF";
11    char buf[11]; // 0x + 8 digits + '\0'
12    int i;
13
14    // create and print buffer
15    buf[0] = '0';
16    buf[1] = 'x';
17    for (i = 0; i < 8; i++) {
18        int shift = (7 - i) * 4;
19        buf[2 + i] = hex_chars[(num >> shift) & 0xF];
20    }
21    buf[10] = '\0';
22
23    sys_req(WRITE, COM1, buf, 10);
24 }
25
26 void allocate_help(void) {
27     const char *helpMsg =
28         "\r\n\033[33m allocate\033[0m [<\033[36msize\033[0m>]\033[36m help\033[0m]\r\n"
29         " \033[33m allocate\033[0m <\033[36msize\033[0m>      allocates 1-50,000 bytes
and prints its address in hex.\r\n"
30         " \033[33m allocate\033[0m \033[36m help\033[0m      prints this help
message\r\n\r\n";
31     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
32 }
33
34 int allocate_command(char *args) {
35     // error checking and cleanup
36     if (!args) return -1;
37     while (*args == ' ') args++;
38
39     // help command
40     if (strncmp(args, "help", 4) == 0) {
41         allocate_help();
42         return 0;
43     }
44
45     // check args
46     int size = atoi(args);
47     if (size <= 0 || size > 50000) {
48         const char *msg = "\033[31m Error: invalid size.\033[0m\r\n";
49         sys_req(WRITE, COM1, msg, strlen(msg));
50         allocate_help();
51         return -1;
52     }
53
54     // call kernel function
55     void *addr = allocate_memory(size);
56     if (!addr) {
57         const char *err = "\033[31m Allocation failed in kernel function.\033[0m\r\n";
58         sys_req(WRITE, COM1, err, strlen(err));

```

```
59         allocate_help();
60     return -1;
61 }
62
63 // successfully allocated
64 const char *success_msg = "\033[32mSuccessfully allocated:\033[0m ";
65 sys_req(WRITE, COM1, success_msg, strlen(success_msg));
66 print_hex((int)addr);
67 const char *success_addr = "\r\n";
68 sys_req(WRITE, COM1, success_addr, strlen(success_addr));
69
70 return 0;
71 }
72
```

```
1 #ifndef MCB_FREE_H
2 #define MCB_FREE_H
3
4 /**
5  * @file mcb/free.h
6  * @brief Shell command handler for freeing memory.
7  */
8
9 /**
10 * @brief Prints help/usage information for the `free` command.
11 */
12 void free_help(void);
13
14 /**
15 * @brief Executes the free command by calling the kernel function.
16 * @param args String containing the hex address to free.
17 * @return 0 on success, -1 on error.
18 */
19 int free_command(char *args);
20
21#endif
22
```

```

1 #include <mpx/vm.h>
2 #include <mpx/serial.h>
3 #include <sys_req.h>
4 #include <stddef.h>
5 #include <stdint.h>
6 #include "mcb/free.h"
7 #include "string.h"
8
9 static void print_hex(int num) {
10     // setup variables
11     char hex_chars[] = "0123456789ABCDEF";
12     char buf[11];    // 0x + 8 digits + '\0'
13     int i;
14
15     // create and print buffer
16     buf[0] = '0';
17     buf[1] = 'x';
18     for (i = 0; i < 8; i++) {
19         int shift = (7 - i) * 4;
20         buf[2 + i] = hex_chars[(num >> shift) & 0xF];
21     }
22     buf[10] = '\0';
23
24     sys_req(WRITE, COM1, buf, 10);
25 }
26
27 void free_help(void) {
28     const char *helpMsg =
29         "\r\n\033[33mfree\033[0m [<\033[36maddress\033[0m>]\033[36mhelp\033[0m]\r\n"
30         " \033[33mfree\033[0m <\033[36maddress\033[0m>      frees the memory at the
 address given.\r\n"
31         " \033[33mfree\033[0m \033[36mhelp\033[0m      prints this help
 message.\r\n\r\n";
32     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
33 }
34
35 int free_command(char *args)
36 {
37     // error checking
38     if (!args) {
39         free_help();
40         return -1;
41     }
42
43     // check args
44     while (*args == ' ') args++;
45     if (*args == '\0') {
46         free_help();
47         return -1;
48     }
49
50     // help command
51     if (strncmp(args, "help", 4) == 0) {
52         free_help();
53         return 0;
54     }
55
56     // parse hex string (optional 0x prefix)
57     int addr = 0;
58     if (args[0] == '0' && (args[1] == 'x' || args[1] == 'X')) {

```

```
59         args += 2;
60     }
61     while ((*args >= '0' && *args <= '9') || (*args >= 'a' && *args <= 'f') || (*args
62 >= 'A' && *args <= 'F')) {
63         int val = 0;
64         if (*args >= '0' && *args <= '9') val = *args - '0';
65         else if (*args >= 'a' && *args <= 'f') val = *args - 'a' + 10;
66         else if (*args >= 'A' && *args <= 'F') val = *args - 'A' + 10;
67
68         addr = (addr << 4) | val;
69         args++;
70     }
71
72     // free memory
73     if (free_memory((void *)addr) == 0) {
74         const char *msg = "\033[32mSuccessfully freed:\033[0m ";
75         sys_req(WRITE, COM1, msg, strlen(msg));
76         print_hex(addr);
77         sys_req(WRITE, COM1, "\r\n", 2);
78         return 0;
79     }
80     else {
81         const char *err = "\033[31mError: Invalid address.\033[0m\r\n";
82         sys_req(WRITE, COM1, err, strlen(err));
83         free_help();
84     }
85 }
86 }
```

```
1 #ifndef MCB_SHOW_H
2 #define MCB_SHOW_H
3
4 /**
5  * @file show.h
6  * @brief Functions for displaying information about MCBs.
7 */
8
9 /**
10 * @brief Prints help information for the "show mcb" command.
11 */
12 void show_mcb_help(void);
13
14 /**
15 * @brief Handles the "show mcb" shell command.
16 * @param args User argument string: address or help.
17 * @return The address parsed on success, or -1 on failure.
18 */
19 int show_mcb_command(char *args);
20
21 /**
22 * @brief Prints allocated memory blocks.
23 */
24 void show_mcb_allocated(void);
25
26 /**
27 * @brief Prints free memory blocks.
28 */
29 void show_mcb_free(void);
30
31 /**
32 * @brief Prints all memory blocks.
33 */
34 void show_mcb_all(void);
35
36#endif
37
```

```

1 #include "mcb/show.h"
2 #include <mpx/vm.h>
3 #include <mpx/serial.h>
4 #include <sys_req.h>
5 #include <string.h>
6
7 static void print_hex(int num) {
8     // setup variables
9     char hex_chars[] = "0123456789ABCDEF";
10    char buf[11];    // 0x + 8 digits + '\0'
11    int i;
12
13    // create and print buffer
14    buf[0] = '0';
15    buf[1] = 'x';
16    for (i = 0; i < 8; i++) {
17        int shift = (7 - i) * 4;
18        buf[2 + i] = hex_chars[(num >> shift) & 0xF];
19    }
20    buf[10] = '\0';
21
22    sys_req(WRITE, COM1, buf, 10);
23}
24
25 static void print_dec(int num) {
26     // setup variables for 32-bit int
27     char buf[12];
28     int i = 0, j, k;
29     char temp;
30
31     if (num == 0) {
32         buf[0] = '0';
33         buf[1] = '\0';
34         sys_req(WRITE, COM1, buf, 1);
35         return;
36     }
37
38     while (num > 0 && i < 11) {
39         buf[i++] = '0' + (num % 10);
40         num /= 10;
41     }
42
43     // reverse the buffer
44     for (j = 0, k = i - 1; j < k; j++, k--) {
45         temp = buf[j];
46         buf[j] = buf[k];
47         buf[k] = temp;
48     }
49     buf[i] = '\0';
50     sys_req(WRITE, COM1, buf, i);
51}
52
53 void show_mcb_help(void) {
54     const char *helpMsg =
55         "\r\n\033[33mshow mcb\033[0m [\033[36mallocated\033[0m|\033[36mfree\033[0m|
56         <\033[36maddress\033[0m>]\033[36mhelp\033[0m]\r\n"
57         " \033[33mshow mcb\033[0m           shows info about all memory.\r\n"
58         " \033[33mshow mcb\033[0m \033[36mallocated\033[0m    shows info about all
      allocated memory.\r\n"

```

```

58     " \033[33mshow mcb\033[0m \033[36mfree\033[0m      shows info about all
59     free memory.\r\n"
60     " \033[33mshow mcb\033[0m <\033[36maddress\033[0m>      shows info about the
61     memory at the address given.\r\n"
62     " \033[33mshow mcb\033[0m \033[36mhelp\033[0m      prints this help
63     "\r\n";
64     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
65 }
66
67 //Show allocated function
68
69 void show_mcb_allocated(void){
70     //setup variable
71     struct mcb *cur;
72     int mcb_size = 0;
73     int mcb_addr = 0;
74
75     //iterate through loop
76     for (cur = mem_allocated_list.head; cur; cur = cur->next) {
77         //assign variables
78         mcb_addr=cur->startAddr;
79         mcb_size=cur->size;
80
81         //print address
82         sys_req(WRITE, COM1, "\r\nAllocated Memory\r\n", strlen("\r\nAllocated
Memory\r\n"));
83         sys_req(WRITE, COM1, "Start Address: ", strlen("Start Address: "));
84         print_hex(mcb_addr);
85
86         //print size
87         sys_req(WRITE, COM1, "\nSize: ", strlen("\nSize: "));
88         print_dec(mcb_size);
89         sys_req(WRITE, COM1, "\n\n", strlen("\n\n"));
90     }
91
92 //Show free function
93
94 void show_mcb_free(void){
95     //setup variable
96     struct mcb *cur;
97     int mcb_size = 0;
98     int mcb_addr = 0;
99
100    //iterate through loop
101    for (cur = mem_free_list.head; cur; cur = cur->next) {
102        //assign variables
103        mcb_addr=cur->startAddr;
104        mcb_size=cur->size;
105
106        //print address
107        sys_req(WRITE, COM1, "\r\nFree Memory\r\n", strlen("\r\nFree Memory\r\n"));
108        sys_req(WRITE, COM1, "Start Address: ", strlen("Start Address: "));
109        print_hex(mcb_addr);
110
111        //print size
112        sys_req(WRITE, COM1, "\nSize: ", strlen("\nSize: "));
113        print_dec(mcb_size);

```

```

114         sys_req(WRITE, COM1, "\n\n", strlen("\n\n"));
115     }
116 }
117
118 //Show all function
119
120 void show_mcb_all(void){
121     show_mcb_allocated();
122     show_mcb_free();
123 }
124
125 int show_mcb_command(char *args) {
126     // setup variables and check args
127     int addr = 0;
128     int i = 0;
129     if (args == NULL) return -1;
130
131     // help command
132     if (strncmp(args, "help", 4) == 0) {
133         show_mcb_help();
134         return 0;
135     }
136     else if (strncmp(args, "allocated", 9) == 0) {
137         show_mcb_allocated();
138         return 0;
139     }
140     else if (strncmp(args, "free", 4) == 0) {
141         show_mcb_free();
142         return 0;
143     }
144     else if (args == NULL || *args=='\0' || (strcmp(args, "all")==0)){
145         show_mcb_all();
146         return 0;
147     }
148
149     // parse hex string from args
150     if (args[0] == '0' && (args[1] == 'x' || args[1] == 'X')) i = 2;
151
152     while (args[i] != '\0') {
153         char c = args[i];
154         int val = 0;
155         if (c >= '0' && c <= '9') val = c - '0';
156         else if (c >= 'A' && c <= 'F') val = 10 + (c - 'A');
157         else if (c >= 'a' && c <= 'f') val = 10 + (c - 'a');
158         else break; // invalid char
159         addr = (addr << 4) | val;
160         i++;
161     }
162
163     // setup more variables
164     struct mcb *cur;
165     int found = 0;
166
167     // check allocated list
168     for (cur = mem_allocated_list.head; cur; cur = cur->next) {
169         if (cur->startAddr == addr) {
170             const char *header_msg = "\r\nAllocated Block(s):\r\n";
171             sys_req(WRITE, COM1, header_msg, strlen(header_msg));
172
173             const char *addr_msg = " Start address = .";

```

```

174     sys_req(WRITE, COM1, addr_msg, strlen(addr_msg));
175     print_hex(cur->startAddr);
176
177     const char *size_msg = "", Size = "";
178     sys_req(WRITE, COM1, size_msg, strlen(size_msg));
179     print_dec(cur->size);
180
181     sys_req(WRITE, COM1, "\r\n", 2);
182     found = 1;
183 }
184 }
185
186 sys_req(WRITE, COM1, "\r\n", 2);
187
188 // check free list if not found
189 for (cur = mem_free_list.head; cur; cur = cur->next) {
190     if (cur->startAddr == addr) {
191         const char *header_msg = "\r\nFree Block(s):\r\n";
192         sys_req(WRITE, COM1, header_msg, strlen(header_msg));
193
194         const char *addr_msg = " Start address = ";
195         sys_req(WRITE, COM1, addr_msg, strlen(addr_msg));
196         print_hex(cur->startAddr);
197
198         const char *size_msg = "", Size = ".";
199         sys_req(WRITE, COM1, size_msg, strlen(size_msg));
200         print_dec(cur->size);
201
202         sys_req(WRITE, COM1, "\r\n", 2);
203         found = 1;
204     }
205 }
206
207 if (!found) {
208     const char *err = "Error: Block not found.\r\n";
209     sys_req(WRITE, COM1, err, strlen(err));
210     return -1;
211 }
212
213 return addr;
214 }
215

```

```
1 #ifndef BLOCK_H
2 #define BLOCK_H
3
4 #include <pcb.h>
5 #include <sys_req.h>
6 #include <string.h>
7
8
9
10 /**
11 * @brief Function for handling command input for blocking PCB, called by command
12 * @param args Arguments for handling block command
13 */
14 void block_pcb_command(const char* args);
15
16
17 /**
18 * @brief Provides help for how to use the block command
19 */
20 void block_help(void);
21
22
23 /**
24 * @brief Block the desired PCB
25 * @param name Argument stores the name of desired PCB
26 */
27 void block_pcb(const char* name);
28
29
30 /**
31 * @brief Function for handling command input for unblocking PCB, called by command
32 * @param args Arguments for handling unblock command
33 */
34 void unblock_pcb_command(const char* args);
35
36
37 /**
38 * @brief Provides help for how to use the unblock command
39 */
40 void unblock_help(void);
41
42
43 /**
44 * @brief Unblock the desired PCB
45 * @param name Argument stores the name of desired PCB
46 */
47 void unblock_pcb(const char* name);
48
49#endif
50
```

```

1 #include <pcb.h>
2 #include <sys_req.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include "block.h"
6 #include "showPCB.h"
7
8 /////////////////////////////////
9 //block function
10
11
12 //help message for block
13 void block_help(void){
14     const char *helpMessage =
15         "\r\n\033[33mblock\033[0m [<\033[36mname\033[0m>]\033[36mhelp\033[0m]\r\n"
16         " \033[33mblock\033[0m <\033[36mname\033[0m> block the requested
process with the given name (1-16 characters)\r\n"
17         " \033[33mblock\033[0m \033[36mhelp\033[0m prints this message\r\n"
18         "\r\n";
19     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
20 }
21
22 //blocks the selected process
23 void block_pcb(const char* name){
24     //check to see if process is already blocked
25     struct pcb* p = pcb_find(name);
26     if(p->execution_state==STATE_BLOCKED){
27         sys_req(WRITE, COM1, "\033[31mProcess has already been blocked.\033[0m\n",
44);
28     }
29
30     // Checks if the name is appropriate length
31     else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
32         sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16
characters\033[0m\n", 59);
33         block_help();
34     }
35
36     // Checks if the process exists
37     else if (p == NULL){
38         sys_req(WRITE, COM1, "\033[31mCould not find process: use 'show' for list of
processes\033[0m\n", 67);
39         show_pcb_help();
40     }
41
42     //block the selected process
43     else{
44         pcb_remove(p);
45         p->execution_state=STATE_BLOCKED;
46         pcb_insert(p);
47         sys_req(WRITE, COM1, "\033[36mProcess blocked successfully\033[0m\n", 39);
48     }
49 }
50
51 //function for use in comhand
52 void block_pcb_command(const char* args){
53     //validation
54     for (int i=0; i<(int)strlen(args); i++){
55         if (args[i] == ' '){

```

```

56         sys_req(WRITE, COM1, "\033[31mError: Please ensure a valid name is
57 given\033[0m\n", 47);
58         block_help();
59     return;
60 }
61 if (args == NULL || *args =='\0'){
62     sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is
63 given\033[0m\n", 47);
64     block_help();
65 }
66 // Checks if argument is help
67 else if (strcmp(args, "help")==0){
68     block_help();
69 }
70 // Handles blocking process
71 else{
72     block_pcb(args);
73 }
74 }
75 }
76
77 /////////////////////////////////
78 //unblock function
79
80
81 //help message for unblock
82 void unblock_help(void){
83     const char *helpMessage =
84         "\r\n\033[33munblock\033[0m [<\033[36mname\033[0m>]\033[36mhelp\033[0m]\r\n"
85         " \033[33munblock\033[0m <\033[36mname\033[0m>      unblock the requested
86 process with the given name (1-16 characters)\r\n"
87         " \033[33munblock\033[0m \033[36mhelp\033[0m      prints this
88 message\r\n"
89         "\r\n";
90     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
91 }
92
93 //unblocks the selected process
94 void unblock_pcb(const char* name){
95     //check to see if process is already unblocked
96     struct pcb* p = pcb_find(name);
97     if(p->execution_state==STATE_READY){
98         sys_req(WRITE, COM1, "\033[31mProcess has already been unblocked.\033[0m\n",
99         46);
100    }
101
102    // Checks if the name is appropriate length
103    else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
104        sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16
105 characters\033[0m\n", 59);
106        unblock_help();
107    }
108
109    // Checks if the process exists
110    else if (p == NULL){
111        sys_req(WRITE, COM1, "\033[31mCould not find process: use 'show' for list of
112 processes\033[0m\n", 67);
113        show_pcb_help();

```

```
109     }
110
111     //unblock the selected process
112     else{
113         pcb_remove(p);
114         p->execution_state=STATE_READY;
115         pcb_insert(p);
116         sys_req(WRITE, COM1, "\033[36mProcess unblocked successfully\033[0m\n", 41);
117     }
118 }
119
120 //function for use in command
121 void unblock_pcb_command(const char* args){
122     //validation
123     for (int i=0; i<(int)strlen(args); i++){
124         if (args[i] == ' '){
125             sys_req(WRITE, COM1, "\033[31mError: Please ensure a valid name is
given\033[0m\n", 47);
126             unblock_help();
127             return;
128         }
129     }
130     if (args == NULL || *args =='\0'){
131         sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is
given\033[0m\n", 47);
132         unblock_help();
133     }
134
135     // Checks if argument is help
136     else if (strcmp(args, "help")==0){
137         unblock_help();
138     }
139
140     // Handles unblocking process
141     else{
142         unblock_pcb(args);
143     }
144 }
145 }
```

```
1 #ifndef INIT_H
2 #define INIT_H
3
4 #include <pcb.h>
5 #include <sys_req.h>
6 #include <string.h>
7
8 /**
9 * @brief Function for handling command input for creating PCB, called by command
10 * @param args Arguments for handling create command
11 */
12 void create_pcb_command(const char* args);
13
14
15 /**
16 * @brief Provides help for how to use the create command
17 */
18 void create_help(void);
19
20
21 /**
22 * @brief Create the desired PCB
23 * @param name Argument stores the desired name of PCB
24 * @param process_class Argument stores the desired class of PCB
25 * @param priority Argument stores the desired priority of PCB
26 */
27 void create_pcb(const char* name, int process_class, int priority);
28
29
30 /**
31 * @brief Function for handling command input for deleting PCB, called by command
32 * @param args Arguments for handling delete command
33 */
34 void delete_pcb_command(const char* args);
35
36
37 /**
38 * @brief Provides help for how to use the delete command
39 */
40 void delete_help(void);
41
42
43 /**
44 * @brief Delete the desired PCB
45 * @param name Argument stores the name of desired PCB
46 */
47 void delete_pcb(const char* name);
48
49#endif
50
```

```

1 #include <pcb.h>
2 #include <sys_req.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include "init.h"
6 #include "showPCB.h"
7
8 ///////////////////////////////////////////////////////////////////
9 //create function
10
11
12 //help message for create
13 void create_help(void){
14     const char *helpMessage =
15         "\r\n\033[33mcreate\033[0m [<\033[36mname\033[0m>] \033[36mhelp\033[0m]
16 [<\033[36mclass\033[0m>] [<\033[36mpriority\033[0m>]\r\n"
17     " \033[33mcreate\033[0m <\033[36mname\033[0m> <\033[36mclass\033[0m>
18 <\033[36mpriority\033[0m>      create a new process with the given:\r\n"
19     "      name (1-16 characters)\r\n"
20     "      class (0=system or 1=user)\r\n"
21     "      priority (0=highest to 9=lowest)\r\n"
22     " \033[33mcreate\033[0m \033[36mhelp\033[0m      prints this message\r\n"
23     "\r\n";
24     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
25 }
26
27 //creates a new pcb and inserts it into the queue
28 /*
29 void create_pcb(const char* name, int process_class, int priority){
30     struct pcb* p = pcb_find(name);
31
32     // Checks if the process exists
33     if (p != NULL){
34         sys_req(WRITE, COM1, "\033[31mInvalid Name: Given process already
exists\033[0m\n", 53);
35     }
36
37     // Checks if the name is appropriate length
38     else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
39         sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16
characters\033[0m\n", 59);
40         create_help();
41     }
42
43     // Checks if the process class is valid
44     else if (process_class != CLASS_SYSTEM && process_class != CLASS_USER){
45         sys_req(WRITE, COM1, "\033[31mInvalid Process: Given class must be 0
(system) or 1 (user)\033[0m\n", 70);
46         create_help();
47     }
48
49     // Validates priority
50     else if (priority>9 || priority<0){
51         sys_req(WRITE, COM1, "\033[31mInvalid Priority: Priority level must be
between 0 (Highest) and 9 (Lowest)\033[0m\n", 86);
52         create_help();
53     }
54
55     struct pcb* newProcess = pcb_setup(name, process_class, priority);

```

```

55         pcb_insert(newProcess);
56         sys_req(WRITE, COM1, "\033[36mProcess created successfully\033[0m\n", 39);
57     }
58
59 }
60 */
61
62 //function for use in comhand
63 void create_pcb_command(const char* args){
64
65     if (args == NULL || *args == '\0'){
66         sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and
priority are given\033[0m\n", 69);
67         create_help();
68     }
69
70     // Checks if argument is help
71     else if (strcmp(args, "help")==0){
72         create_help();
73     }
74
75     // Handles getting process name, class, and priority number
76     /*
77     else{
78         // get a substring for the process name from the args
79         int temp = 0;
80         for (int i=0; i<(int)strlen(args); i++){
81             if (args[i] == ' ' && i<(int)strlen(args) && i != 0){
82                 temp = i;
83                 break;
84             }
85         }
86         //parse the substring above into name
87         if (temp != 0){
88             char name[temp + 1];
89             for (int i=0; i<temp; i++){
90                 name[i] = args[i];
91             }
92             name[temp] = '\0'; // Added in null terminator
93             //parse process class
94             char class[2];
95             char priority_char[2];
96             const char *numArg = args + temp +1;
97             strncpy(class, numArg, 1);
98             class[1]='\0';
99             int process_class = atoi(class);
100            if(class[0]<'0'||class[0]>'1' || *(numArg+1)!= ' '){
101                sys_req(WRITE, COM1, "\033[31mError: Class must be valid
number\033[0m\n", 44);
102                create_help();
103                return;
104            }
105            //parse process priority
106            numArg+=2;
107            strncpy(priority_char, numArg, 1);
108            priority_char[1]='\0';
109            int priority = atoi(priority_char);
110            if((priority_char[0]<'0' || priority_char[0]>'9') || *(numArg+1)!= '\0'){
111                sys_req(WRITE, COM1, "\033[31mError: Priority must be valid
number\033[0m\n", 47);

```

```

112         create_help();
113         return;
114     }
115     create_pcb(name, process_class, priority);
116 }
117 else{
118     sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and
priority are given\033[0m\n", 69);
119     create_help();
120 }
121 }
122 */
123 }
124
125
126 ///////////////////////////////////////////////////////////////////
127 //delete function
128
129
130
131 //help message for delete
132 void delete_help(void){
133     const char *helpMessage =
134         "\r\n\033[33mdelete\033[0m [<\033[36mname\033[0m>]\033[36mhelp\033[0m]\r\n"
135         " \033[33mdelete\033[0m <\033[36mname\033[0m>    deletes the requested user
process with the given name (1-16 characters)\r\n"
136         " \033[33mdelete\033[0m \033[36mhelp\033[0m      prints this message\r\n"
137         "\r\n";
138     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
139 }
140
141
142 //Finds the requested process and removes it from the queue
143 void delete_pcb(const char* name){
144     struct pcb* proc_to_delete = pcb_find(name);
145
146     // Checks if the process exists
147     if (proc_to_delete == NULL){
148         sys_req(WRITE, COM1, "\033[31mInvalid Name: Given process does not
exist\033[0m\n", 53);
149     }
150
151     // Checks if the name is appropriate length
152     else if (name == NULL || strlen(name) < 1 || strlen(name) > PCB_NAME_MAX_LEN){
153         sys_req(WRITE, COM1, "\033[31mInvalid Name: Given name must be 1-16
characters\033[0m\n", 59);
154         delete_help();
155     }
156
157     // Checks if the process class
158     else if (proc_to_delete ->process_class == CLASS_SYSTEM){
159         sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process is a
kernel/system level process\033[0m\n", 74);
160     }
161
162     else{
163         pcb_remove(proc_to_delete );
164         pcb_free(proc_to_delete );
165         sys_req(WRITE, COM1, "\033[36mProcess deleted successfully\033[0m\n", 39);
166     }

```

```
167 }
168 //function for use in comhand
169 void delete_pcb_command(const char* args){
170
171     if (args == NULL || *args == '\0'){
172         sys_req(WRITE, COM1, "\033[31mError: Please ensure a name is
173 given\033[0m\n", 47);
174         delete_help();
175     }
176
177     // Checks if argument is help
178     else if (strcmp(args, "help")==0){
179         delete_help();
180     }
181
182     // Handles getting process name
183     else{
184         delete_pcb(args);
185     }
186 }
187
```

```
1 #ifndef PCB_READY_H
2 #define PCB_READY_H
3
4 /**
5  * @file ready.h
6  * @brief Commands to suspend or resume a process.
7  */
8
9 #include "pcb.h"
10
11 /**
12  * @brief Prints the help message for the suspend command.
13  */
14 void suspend_help(void);
15
16 /**
17  * @brief Puts a non-system process in the suspended state, and moves it to the
18  * appropriate queue.
19  * @param process_name Process's name (checks for validity)
20  * @return int 0 for success, -1 for invalid process/name, -2 if given a system
21  * process, and 1 if already suspended.
22  */
23 int suspend_pcb(const char* process_name);
24
25 /**
26  * @brief Handles command-line arguments for the suspend command.
27  * @param args Command argument string (process name or "help").
28  */
29 void suspend_command(const char *args);
30
31 /**
32  * @brief Prints the help message for the resume command.
33  */
34 void resume_help(void);
35
36 /**
37  * @brief Puts a process in the active (not suspended) state, and moves it to the
38  * appropriate queue.
39  * @param process_name Process's name (checks for validity)
40  * @return int 0 for success, -1 for invalid process/name, and 1 if already active.
41  */
42 int resume_pcb(const char* process_name);
43
44 /**
45  * @brief Handles command-line arguments for the resume command.
46  * @param args Command argument string (process name or "help").
47 */
48 void resume_command(const char *args);
49
50#endif
```

```

1 #include <sys_req.h>
2 #include <string.h>
3
4 #include "pcb.h"
5
6 // Because the commands "suspend" and "resume" share so much code, they were
7 // put into the same file for convenience, called "ready".
8
9
10
11 ///////////////////////////////////////////////////////////////////
12 // suspend command
13
14 // help message for suspend
15 void suspend_help(void) {
16     const char *helpMsg =
17         "\r\n\033[33msuspend\033[0m [<\033[36mname\033[0m>]\033[36mhelp\033[0m]\r\n"
18         " \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"
19         " \033[33msuspend\033[0m \033[36mhelp\033[0m      prints this
message\r\n\r\n";
20     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
21 }
22
23 // Puts a non-system process in the suspended state, and moves it to the appropriate
queue.
24 int suspend_pcb(const char* process_name) {
25     // check for invalid name or process
26     if (!process_name || strlen(process_name) == 0) return -1;
27     struct pcb* found_pcb_ptr = pcb_find(process_name);
28     if (!found_pcb_ptr) return -1;
29
30     // check if system process
31     if (found_pcb_ptr->process_class == CLASS_SYSTEM) return -2;
32
33     // check if already suspended
34     if (found_pcb_ptr->dispatch_state == DISPATCH_SUSPENDED) return 1;
35
36     // put the process into the correct queue
37     if (pcb_remove(found_pcb_ptr) != 0) return 1;
38     found_pcb_ptr->dispatch_state = DISPATCH_SUSPENDED;
39     pcb_insert(found_pcb_ptr);
40
41     return 0;
42 }
43
44 // argument handler for suspend
45 void suspend_command(const char *args) {
46     if (args == NULL || *args == '\0' || strcmp(args, "help") == 0) {
47         suspend_help();
48         return;
49     }
50
51     int result = suspend_pcb(args);
52     if (result == -1) {
53         const char *argMsg = "\033[31mInvalid process/name. Please try
again.\033[0m\r\n";
54         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
55         suspend_help();
56     }

```

```

57     else if (result == -2) {
58         const char *argMsg = "\033[31mProcess is a system process and cannot be
59         suspended.\033[0m\r\n";
60     }
61     else if (result == 1) {
62         const char *argMsg = "\033[31mThe process is already suspended.\033[0m\r\n";
63         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
64     }
65     else if (result == 0) {
66         const char *argMsg = "\033[36mThe process has been suspended.\033[0m\r\n";
67         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
68     }
69     else {
70         const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
71         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
72     }
73
74     return;
75 }
76
77
78
79 ///////////////////////////////////////////////////////////////////
80 // resume command
81
82 // help message for resume
83 void resume_help(void) {
84     const char *helpMsg =
85         "\r\n\033[33mresume\033[0m
[<\033[36mname\033[0m>|\033[36mall\033[0m|]\033[36mhelp\033[0m]\r\n"
86         " \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"
87         " \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"
88         " \033[33mresume\033[0m \033[36mhelp\033[0m      prints this
message\r\n\r\n";
89     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
90 }
91
92 // Puts a process in the active (not suspended) state, and moves it to the
appropriate queue.
93 int resume_pcb(const char* process_name) {
94     // check for invalid name or process
95     if (!process_name || strlen(process_name) == 0) return -1;
96     struct pcb* found_pcb_ptr = pcb_find(process_name);
97     if (!found_pcb_ptr) return -1;
98
99     // check if already active
100    if (found_pcb_ptr->dispatch_state == DISPATCH_ACTIVE) return 1;
101
102    // put the process into the correct queue
103    if (pcb_remove(found_pcb_ptr) != 0) return 1;
104    found_pcb_ptr->dispatch_state = DISPATCH_ACTIVE;
105    pcb_insert(found_pcb_ptr);
106
107    return 0;
108 }
109

```

```

110 // argument handler for resume
111 void resume_command(const char *args) {
112     if (args == NULL || *args == '\0' || strcmp(args, "help") == 0) {
113         resume_help();
114         return;
115     }
116     else if (args == NULL || *args == '\0' || strcmp(args, "all") == 0) {
117         struct pcb* curPtr = suspended_ready_queue.head;
118         struct pcb* tempPtr;
119         // Checks if there are any processes in the ready queue
120         if (curPtr == NULL){
121             sys_req(WRITE, COM1, "No processes in the suspended ready queue\n", 42);
122         }
123
124         // If there are processes, move through queue and print each
125
126         else{
127             char* message = "\033[32mResuming all Processes:\033[0m\n";
128             sys_req(WRITE, COM1, message, strlen(message));
129             while(curPtr){
130                 tempPtr = curPtr->next;
131                 resume_pcb(curPtr->name);
132                 curPtr = tempPtr;
133             }
134         }
135     }
136 }
137
138
139     int result = resume_pcb(args);
140     if (result == -1) {
141         const char *argMsg = "\033[31mInvalid process/name. Please try again.\033[0m\r\n";
142         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
143         resume_help();
144     }
145     else if (result == 1) {
146         const char *argMsg = "\033[31mThe process is already active.\033[0m\r\n";
147         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
148     }
149     else if (result == 0) {
150         const char *argMsg = "\033[36mThe process has been resumed.\033[0m\r\n";
151         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
152     }
153     else {
154         const char *argMsg = "\033[31mInvalid argument. Please try again.\033[0m\r\n";
155         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
156     }
157
158     return;
159 }
160

```

```
1 #ifndef SETPRIORITY_H
2 #define SETPRIORITY_H
3
4 #include <pcb.h>
5 #include <sys_req.h>
6 #include <string.h>
7
8 /**
9 * @file setPriority.h
10 * @brief Provides ability to set priority for processes
11 */
12
13 /**
14 * @brief Function for handling command input for setting priority, called by comhand
15 * @param args Arguments for handling set priority command
16 */
17 void set_priority_command(const char* args);
18
19 /**
20 * @brief Provides help for how to use the priority set command
21 */
22 void set_priority_help(void);
23
24 /**
25 * @brief Sets priority of a process
26 * @param name The name of the process that is being modified
27 * @param newPriority The priority that the process is to be set to
28 */
29 void setPriority(char* name, int newPriority);
30
31
32#endif
33
```

```

1 #include <pcb.h>
2 #include <sys_req.h>
3 #include <string.h>
4 #include <stdlib.h>
5
6 /**
7  * Function for setting priority. Takes a string (name) and priority (integer)
8 */
9 void setPriority(char* name, int newPriority){
10     struct pcb* pcbPTR = pcb_find(name);
11
12     // Checks if the process exists
13     if (pcbPTR == NULL){
14         sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process does not
exist\033[0m\n", 56);
15     }
16
17     // Checks if the process class
18     else if (pcbPTR->process_class == CLASS_SYSTEM){
19         sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process is a
kernel/system level process\033[0m\n", 74);
20     }
21
22     // Validates priority is possible
23     else if (newPriority>9 || newPriority<0){
24         sys_req(WRITE, COM1, "\033[31mInvalid Priority: Priority level must be
between 0 (Highest) and 9 (Lowest)\033[0m\n", 86);
25     }
26
27     // Sets priority
28     else if (newPriority<=9 && newPriority>=0){
29         /*
30             * To Fix Page fault errors, changed the priority set to remove the process
from the queue, change its priority and reinsert it.
31             */
32         (void) pcb_remove(pcbPTR); /* ignore error code; pcb_remove will find the
PCB if it's in a queue */
33         pcbPTR->priority = newPriority;
34         pcb_insert(pcbPTR);
35     }
36 }
37
38 /**
39  * Help message for setting priority
40 */
41 void set_priority_help(void){
42     const char *helpMessage =
43         "\r\n\033[33mpriority set\033[0m [<\033[36mname\033[0m>]\033[36mhelp\033[0m
[<\033[36mpriority\033[0m>]\r\n"
44         "\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"
45         "\033[33mpriority set\033[0m \033[36mhelp\033[0m                         prints
this message\r\n"
46         "\r\n";
47     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
48 }
49
50 /**

```

```

51 * Creates function for use in command handler, for setting priority and getting
52 help
53 */
54 void set_priority_command(const char* args){
55
56     // Checks for arguments
57     if (args == NULL || *args == '\0'){
58         sys_req(WRITE, COM1, "\033[31mError: Please ensure a name and priority are
59 given\033[0m\n", 61);
60         set_priority_help();
61     }
62
63     // Checks if argument is help
64     else if (strcmp(args, "help")==0){
65         set_priority_help();
66     }
67
68     // Handles getting process name and priority number
69     else{
70         int isNumber = 1;
71         int temp = 0;
72         for (int i=0; i<(int)strlen(args); i++){
73             if (args[i] == ' ' && i<(int)strlen(args) && i != 0){
74                 temp = i;
75                 break;
76             }
77         }
78         if (temp != 0){
79             char name[temp + 1];
80             for (int i=0; i<temp; i++){
81                 name[i] = args[i];
82             }
83             name[temp] = '\0'; // Added in null terminator
84             const char *numArg = args + temp;
85             for (int i = 0; i<(int)strlen(numArg)-1; i++){
86                 if ((numArg[i] <= '0' || numArg[i] >= '9') && (numArg[i]!='\0' &&
87 numArg[i]!=' ')){
88                     isNumber = 0;
89                     break;
90                 }
91             }
92             if (isNumber == 1){
93                 int priority = atoi(numArg);
94                 setPriority(name, priority);
95             }
96             else {
97                 sys_req(WRITE, COM1, "\033[31mError: Please ensure the priority
98 argument is a number\033[0m\n", 64);
99                 set_priority_help();
100            }
101        }
102    }
103 }
104
105

```



```
1 #ifndef SHOWPCB_H
2 #define SHOWPCB_H
3
4 #include <pcb.h>
5 #include <sys_req.h>
6 #include <string.h>
7 #include <comhand.h>
8 #include <itoa.h>
9
10 /**
11 * @file showPCB.h
12 * @brief Provides ability to display a specific PCB, all PCBs, or PCBs in specific
13 * queues
14 */
15 /**
16 * @brief Handles which functions should be used, for use in comhand
17 * @param args Arguments used for performing correct function
18 */
19 void show_pcb_command(const char* args);
20
21 /**
22 * @brief Displays the details for a PCB if the PCB exists
23 * @param name The name of the PCB to be displayed
24 */
25 void showPCB(const char* name);
26
27 /**
28 * @brief Displays all PCBs in the ready queue
29 */
30 void showReady(void);
31
32 /**
33 * @brief Displays all PCBs in the blocked queue
34 */
35 void showBlocked(void);
36
37 /**
38 * @brief Displays all PCBs in all queues
39 */
40 void showAllPCB(void);
41
42 /**
43 * @brief Help function for the show command
44 */
45 void show_pcb_help(void);
46
47 /**
48 * @brief Displays all PCBs in the suspended queues
49 */
50 void showSuspended(void);
51
52#endif
53
```

```

1 #include <pcb.h>
2 #include <sys_req.h>
3 #include <string.h>
4 #include <comhand.h>
5 #include <itoa.h>
6
7 /**
8  * Help message for showing pcbs
9 */
10 void show_pcb_help(void){
11     const char *helpMessage =
12         "\r\n\033[33mshow pcb\033[0m
13 [<>\033[36mname\033[0m]>|\033[36mready\033[0m|>\033[36mblocked\033[0m|\033[36msuspended
14 \033[0m|\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
15         " \033[33mshow pcb\033[0m <\033[36mname\033[0m> prints details for the
16 named process\r\n"
17         " \033[33mshow pcb\033[0m \033[36mready\033[0m prints details of
18 processes in ready queue\r\n"
19         " \033[33mshow pcb\033[0m \033[36mblocked\033[0m prints details of
20 processes in blocked queue\r\n"
21         " \033[33mshow pcb\033[0m \033[36msuspended\033[0m prints details of
22 processes in the suspended queues\r\n"
23         " \033[33mshow pcb\033[0m \033[36mall\033[0m prints details of all
24 processes\r\n"
25         " \033[33mshow pcb\033[0m \033[36mhelp\033[0m prints this
26 message\r\n"
27         "\r\n";
28     sys_req(WRITE, COM1, helpMessage, strlen(helpMessage));
29 }
30
31 /**
32  * Takes process name (string) and prints details for the process
33 */
34 void showPCB(const char* name){
35     char buffer[20];
36     struct pcb* pcbPTR = pcb_find(name);
37
38     // Checks if the process exists
39     if (pcbPTR == NULL){
40         sys_req(WRITE, COM1, "\033[31mInvalid Process: Given process does not
41 exist\033[0m\n", 56);
42         show_pcb_help();
43     }
44     else {
45
46         // Print name
47         sys_req(WRITE, COM1, "Name: ", strlen("Name: "));
48         sys_req(WRITE, COM1, pcbPTR->name, strlen(pcbPTR->name));
49         sys_req(WRITE, COM1, "\n", 1);
50
51         // Print Class
52         sys_req(WRITE, COM1, "Class: ", strlen("Class: "));
53         if (pcbPTR->process_class == CLASS_SYSTEM){
54             sys_req(WRITE, COM1, "System/Kernel\n", strlen("System/Kernel\n"));
55         }
56         else{
57             sys_req(WRITE, COM1, "User\n", strlen("User\n"));
58         }
59
60         // 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

```

```
172     else if (strcmp(args, "ready", 5) == 0){
173         showReady();
174     }
175
176     // Checks if the argument is blocked
177     else if (strcmp(args, "blocked", 7) == 0){
178         showBlocked();
179     }
180
181     // Checks if the argument is help
182     else if (strcmp(args, "help", 4)==0){
183         show_pcb_help();
184     }
185
186     else if (strcmp(args, "suspended", 9)==0){
187         showSuspended();
188     }
189
190     // Handles any other input
191     else{
192         showPCB(args);
193     }
194 }
195 }
```

```
1 #ifndef ALARM_H
2 #define ALARM_H
3
4 /**
5  * @brief Struct that stores data relating to the alarm
6  * @struct stores Seconds, Minutes, Hours, and Message
7 */
8 typedef struct {
9     int hour;
10    int minute;
11    int second;
12    char message[100];
13 } AlarmData;
14
15 /**
16  * @brief Create an alarm with at the specified HH:MM:SS with specified message
17 */
18 void alarm(void);
19
20 /**
21  * @brief Creates a new pcb with alarm data
22  * @param data Passes relevant data such as time (Hours, minutes, seconds) and alarm
23  * message
24 */
25 void alarm_create(AlarmData* data);
26
27 /**
28  * @brief Main handler for the alarm command.
29  * @param args The argument string passed after 'alarm'
30 */
31 void alarm_command(const char* args);
32
33 /**
34  * @brief Prints help information related to the alarm command.
35 */
36 void alarm_help(void);
37
38#endif
```

```

1 #include "clock.h"
2 #include <string.h>
3 #include <sys_req.h>
4 #include <stdint.h>
5 #include <mpx/interrupts.h>
6 #include <mpx/io.h>
7 #include "stdlib.h"
8 #include "itoa.h"
9 #include "alarm.h"
10 #include "clock.h"
11 #include <pcb.h>
12 #include <processes.h>
13 #include <loadR3.h>
14 #include "sys_req.h"
15 #include "sys_call.h"
16 #include "memory.h"
17
18
19 void alarm(void) {
20     struct pcb* self = sys_get_current_process();
21     //error checking
22     if (!self) {
23         // Shouldn't happen when properly dispatched, but guard anyway
24         sys_req(EXIT);
25         return;
26     }
27     AlarmData* data = (AlarmData*) self->args;
28
29     while (1) {
30         rtc_time_t current_time;
31         get_time(&current_time);
32
33         if ((current_time.hour > data->hour) ||
34             (current_time.hour == data->hour && current_time.minute > data->minute)
35         ||
36             (current_time.hour == data->hour && current_time.minute == data->minute
37             && current_time.second >= data->second)) {
38                 sys_req(WRITE, COM1, data->message, strlen(data->message));
39                 sys_req(WRITE, COM1, "\r\n", 2);
40                 // free data before exiting so it doesn't leak or become dangling
41                 sys_free_mem(data);
42                 self->args = NULL; // avoid dangling pointer
43                 break;
44             }
45             sys_req(IDLE);
46         }
47         sys_req(EXIT);
48 }
49
50 void alarm_create(AlarmData* data){
51     // Allocate memory for alarm data
52     AlarmData* newData = (AlarmData*) sys_alloc_mem(sizeof(AlarmData));
53     if (!newData) {
54         sys_req(WRITE, COM1, "\033[31mError: Memory allocation failed.\033[0m\r\n",
55     );
56         return;
57     }
58     // Copy fields
59     newData->hour = data->hour;

```

```

58     newData->minute = data->minute;
59     newData->second = data->second;
60     strncpy(newData->message, data->message, sizeof(newData->message) - 1);
61     newData->message[sizeof(newData->message) - 1] = '\0';
62
63     // Generate unique name
64     static int alarm_counter = 1;
65     char name[16];
66     my_strcpy(name, "alarm");
67     char numbuf[8];
68     itoa(alarm_counter++, numbuf);
69     my_strcat(name, numbuf);
70
71     // Create and queue PCB
72     struct pcb* newProc = pcb_setup(name, CLASS_USER, 1, alarm);
73     //error checking
74     if (!newProc) {
75         sys_free_mem(data);
76         const char *err = "\033[31mError: Failed to setup alarm
process.\033[0m\r\n";
77         sys_req(WRITE, COM1, err, strlen(err));
78         return;
79     }
80     newProc->args = newData;
81     pcb_insert(newProc);
82
83     char* mes = "\033[32mAlarm set successfully.\033[0m\r\n";
84
85     sys_req(WRITE, COM1, mes, strlen(mes));
86 }
87
88 void alarm_command(const char* args){
89     if (args == NULL || *args == '\0'){
90         sys_req(WRITE, COM1, "\033[31mError: Please ensure a name, class, and
priority are given\033[0m\r\n", 69);
91         alarm_help();
92         return;
93     }
94
95     // Checks if argument is help
96     else if (strcmp(args, "help")==0){
97         alarm_help();
98         return;
99     }
100    char time_str[9]; //to hold time substring of args
101    const char* time_ptr = strncpy(time_str, args, 8);
102    const char* message = args + 9; // skip "HH:MM:SS "
103
104
105    //error test message
106    if (strlen(message) > 100 || message == NULL || *message == '\0') {
107        const char *err = "\033[31mMust include valid message\033[0m\r\n";
108        sys_req(WRITE, COM1, err, strlen(err));
109        alarm_help();
110        return;
111    }
112    // Expected format: HH:MM:SS
113    if (strlen(time_ptr) != 8 || time_ptr[2] != ':' || time_ptr[5] != ':') {
114        const char *err = "\033[31mInvalid format. Use set time
<HH:MM:SS>\033[0m\r\n";

```

```

115     sys_req(WRITE, COM1, err, strlen(err));
116     alarm_help();
117     return;
118 }
119
120 char buf[3];
121 buf[2] = '\0';
122
123 buf[0] = time_ptr[0]; buf[1] = time_ptr[1];
124 int hour = atoi(buf);
125
126 buf[0] = time_ptr[3]; buf[1] = time_ptr[4];
127 int minute = atoi(buf);
128
129 buf[0] = time_ptr[6]; buf[1] = time_ptr[7];
130 int second = atoi(buf);
131
132 if (hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 || second >
59) {
133     const char *err = "\033[31mInvalid time values.\033[0m\r\n";
134     sys_req(WRITE, COM1, err, strlen(err));
135     alarm_help();
136     return;
137 }
138 AlarmData* data = (AlarmData*)sys_alloc_mem(sizeof(AlarmData));
139 data->hour=hour;
140 data->minute=minute;
141 data->second=second;
142 strncpy(data->message, message, sizeof(data->message) - 1);
143 data->message[sizeof(data->message) - 1] = '\0';//ensure null termination
144 alarm_create(data);
145 }
146
147 void alarm_help(void){
148     const char *helpMsg =
149         "\r\n\033[33malarm\033[0m [ \033[36mhelp\033[0m|<\033[36mtime\033[0m>
[<\033[36mmessage\033[0m>]\r\n"
150         " \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"
151         " \033[33malarm\033[0m \033[36mhelp\033[0m                                     prints this
message.\r\n"
152         "\r\n";
153     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
154 }
155

```

```

1 #ifndef CLOCK_H
2 #define CLOCK_H
3
4 #include <string.h>
5 #include <sys_req.h>
6 #include <stdint.h>
7 #include <mpx/interrupts.h>
8 #include <mpx/io.h>
9
10 /**
11  * @file clock.h
12  * @brief Handles accesses to the Real Time Clock (RTC)
13  */
14
15 /**
16  * @brief Struct that stores data relating to the time
17  * @struct stores Seconds, Minutes, and Hours
18  */
19 typedef struct {
20     uint8_t second;
21     uint8_t minute;
22     uint8_t hour;
23 } rtc_time_t;
24
25 /**
26  * @brief Struct that stores data relating to the date
27  * @struct stores Day, Month, and Year
28  */
29 typedef struct {
30     uint8_t day;
31     uint8_t month;
32     uint8_t year;    // Last two digits of the year
33 } rtc_date_t;
34
35
36 /**
37  * @brief Accesses the RTC and stores the current time into specified struct
38  * @param time Stores the current Seconds, Minutes, and Hours
39  */
40 void get_time(rtc_time_t *time);
41
42 /**
43  * @brief Stores the specified time into associated RTC registers
44  * @param time Passes user defined Seconds, Minutes, and Hours
45  */
46 void set_time(const rtc_time_t *time);
47
48 /**
49  * @brief Accesses the RTC and stores the current date into specified struct
50  * @param date Stores the current Day, Month, and Year
51  */
52 void get_date(rtc_date_t *date);
53
54 /**
55  * @brief Stores the specified date into associated RTC registers
56  * @param date Passes user defined Day, Month, and Year
57  */
58 void set_date(const rtc_date_t *date);
59
60 /**

```

```
61 * @brief Prints the specified time in the form HH:MM:SS
62 * @param time Passes the Seconds, Minutes, and Hours
63 */
64 void print_time(rtc_time_t *time);
65
66 /**
67 * @brief Prints the specified date in the form MM/DD/YY
68 * @param time Passes the Day, Month, and Year
69 */
70 void print_date(const rtc_date_t *date);
71
72 /**
73 * @brief Prints help information related to the clock command.
74 */
75 void clock_help(void);
76
77 /**
78 * @brief Main handler for the clock command.
79 * @param args The argument string passed after 'clock'
80 */
81 void clock_command(const char *args);
82
83 //---- Helper Functions ----//
84
85 /**
86 * @brief Concatenates src to the end of dest
87 * @param dest The argument string to be appended to
88 * @param src The argument string to append to
89 */
90 void my_strcat(char *dest, const char *src);
91
92 /**
93 * @brief Copies src to dest
94 * @param dest Stores a copy of src
95 * @param src Copied to dest
96 */
97 void my strcpy(char *dest, const char *src);
98
99 /**
100 * @brief Writes data to an RTC register
101 * @param reg Address of specified RTC register
102 * @param value The byte to write to the RTC register
103 */
104 void rtc_write(uint8_t reg, uint8_t value);
105
106 /**
107 * @brief Reads data from an RTC register
108 * @param reg Address of specified RTC register
109 */
110 uint8_t rtc_read(uint8_t reg);
111
112 /**
113 * @brief Converts an integer from binary to Binary Coded Decimal (BCD)
114 * @param value The value to convert
115 * @returns The specified value as a BCD
116 */
117 uint8_t bin_to_bcd(uint8_t value);
118
119 /**
120 * @brief Converts an integer from Binary Coded Decimal (BCD) to binary
```

```
121 * @param value The value to convert
122 * @returns The specified value in binary
123 */
124 uint8_t bcd_to_bin(uint8_t value);
125
126 /**
127 * @brief Converts the standard RTC timezone (UTC) to EST
128 */
129 void tz_correction(void);
130
131 #endif
132
```

```

1 #include "clock.h"
2 #include <string.h>
3 #include <sys_req.h>
4 #include <stdint.h>
5 #include <mpx/interrupts.h>
6 #include <mpx/io.h>
7 #include "stdlib.h"
8 #include "itoa.h"
9
10#define CMOS_ADDRESS 0x70
11#define CMOS_DATA      0x71
12
13 // RTC register indexes
14#define REG_SECONDS    0x00
15#define REG_MINUTES    0x02
16#define REG_HOURS      0x04
17#define REG_DAY         0x07
18#define REG_MONTH       0x08
19#define REG_YEAR        0x09
20#define REG_STATUS_B    0x0B
21
22 // --- Helpers ---
23 uint8_t bcd_to_bin(uint8_t value) {
24     return ((value >> 4) * 10) + (value & 0x0F);
25 }
26
27 uint8_t bin_to_bcd(uint8_t value) {
28     return ((value / 10) << 4) | (value % 10);
29 }
30
31 uint8_t rtc_read(uint8_t reg) {
32     outb(CMOS_ADDRESS, reg);
33     return inb(CMOS_DATA);
34 }
35
36 void rtc_write(uint8_t reg, uint8_t value) {
37     outb(CMOS_ADDRESS, reg);
38     outb(CMOS_DATA, value);
39 }
40
41 void my_strcpy(char *dest, const char *src) {
42     while (*src) {
43         *dest++ = *src++;
44     }
45     *dest = '\0';
46 }
47
48 void my_strcat(char *dest, const char *src) {
49     while (*dest) dest++;
50     while (*src) {
51         *dest++ = *src++;
52     }
53     *dest = '\0';
54 }
55
56 void clock_command(const char *args){
57     if (args == NULL || *args == '\0') {
58         rtc_time_t t;
59         rtc_date_t d;
60         get_time(&t);

```

```

61     get_date(&d);
62     print_date(&d);
63     print_time(&t);
64 }
65 else if (strcmp(args, "get time") == 0) {
66     rtc_time_t t;
67     get_time(&t);
68     print_time(&t);
69 }
70 else if (strcmp(args, "get date") == 0) {
71     rtc_date_t d;
72     get_date(&d);
73     print_date(&d);
74 }
75 else if (strncmp(args, "set time ", 9) == 0) {
76     const char *val = args + 9; // skip "set time "
77
78     // Expected format: HH:MM:SS
79     if (strlen(val) != 8 || val[2] != ':' || val[5] != ':') {
80         const char *err = "\033[31mInvalid format. Use set time
<HH:MM:SS>\033[0m\r\n";
81         sys_req(WRITE, COM1, err, strlen(err));
82         clock_help();
83         return;
84     }
85
86     char buf[3];
87     buf[2] = '\0';
88
89     buf[0] = val[0]; buf[1] = val[1];
90     int hour = atoi(buf);
91
92     buf[0] = val[3]; buf[1] = val[4];
93     int minute = atoi(buf);
94
95     buf[0] = val[6]; buf[1] = val[7];
96     int second = atoi(buf);
97
98     if (hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 ||
99 second > 59) {
100         const char *err = "\033[31mInvalid time values.\033[0m\r\n";
101         sys_req(WRITE, COM1, err, strlen(err));
102         clock_help();
103         return;
104     }
105
106     rtc_time_t t = { second, minute, hour };
107     set_time(&t);
108     //print_time(&t);
109 }
110 else if (strncmp(args, "set date ", 9) == 0) {
111     const char *val = args + 9; // skip "set date "
112
113     // Expected format: DD/MM/YY
114     if (strlen(val) != 8 || val[2] != '/' || val[5] != '/') {
115         const char *err = "\033[31mInvalid format. Use set date
<MM/DD/YY>\033[0m\r\n";
116         sys_req(WRITE, COM1, err, strlen(err));
117         clock_help();
118         return;

```

```

118     }
119
120     char buf[3];
121     buf[2] = '\0';
122
123     buf[0] = val[0]; buf[1] = val[1];
124     int month = atoi(buf);
125
126     buf[0] = val[3]; buf[1] = val[4];
127     int day = atoi(buf);
128
129     buf[0] = val[6]; buf[1] = val[7];
130     int year = atoi(buf);
131
132     if (day < 1 || day > 31 || month < 1 || month > 12) {
133         const char *err = "\033[31mInvalid date values.\033[0m\r\n";
134         sys_req(WRITE, COM1, err, strlen(err));
135         clock_help();
136         return;
137     }
138
139     rtc_date_t d = { day, month, year };
140     set_date(&d);
141     //print_date(&d);
142 }
143 else if (strcmp(args, "help") == 0) {
144     clock_help();
145 }
146 else {
147     const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
148     sys_req(WRITE, COM1, argMsg, strlen(argMsg));
149     clock_help();
150 }
151 }
152
153 void print_time(rtc_time_t *time){
154     char buffer[100];
155     char num[4];
156
157     //time->hour = (time->hour + 24 - 4) % 24;
158     my_strcpy(buffer, "\r\nTime: ");
159
160     if (time->hour < 10) my_strcat(buffer, "0");
161     itoa(time->hour, num);
162     my_strcat(buffer, num);
163     my_strcat(buffer, ":");
164
165     if (time->minute < 10) my_strcat(buffer, "0");
166     itoa(time->minute, num);
167     my_strcat(buffer, num);
168     my_strcat(buffer, ":");
169
170     if (time->second < 10) my_strcat(buffer, "0");
171     itoa(time->second, num);
172     my_strcat(buffer, num);
173     my_strcat(buffer, "\r\n\r\n");
174
175     size_t len = strlen(buffer);
176 }
```

```

177     sys_req(WRITE, COM1, buffer, len);
178 }
179
180 void print_date(const rtc_date_t *date){
181     char buffer[100];
182     char num[4];
183
184     my_strcpy(buffer, "\r\nDate: ");
185
186     if (date->month < 10) my_strcat(buffer, "0");
187     itoa(date->month, num);
188     my_strcat(buffer, num);
189     my_strcat(buffer, "/");
190
191     if (date->day < 10) my_strcat(buffer, "0");
192     itoa(date->day, num);
193     my_strcat(buffer, num);
194     my_strcat(buffer, "/");
195
196     if (date->year < 10) my_strcat(buffer, "0");
197     itoa(date->year, num);
198     my_strcat(buffer, num);
199     my_strcat(buffer, "\r\n\r\n");
200
201     size_t len = strlen(buffer);
202
203     sys_req(WRITE, COM1, buffer, len);
204 }
205
206 void get_time(rtc_time_t *time) {
207     uint8_t regB = rtc_read(REG_STATUS_B);
208
209     time->second = rtc_read(REG_SECONDS);
210     time->minute = rtc_read(REG_MINUTES);
211     time->hour   = rtc_read(REG_HOURS);
212
213     // Convert from BCD if needed
214     if (!(regB & 0x04)) {
215         time->second = bcd_to_bin(time->second);
216         time->minute = bcd_to_bin(time->minute);
217         time->hour   = bcd_to_bin(time->hour);
218     }
219 }
220
221 void set_time(const rtc_time_t *time) {
222     uint8_t regB = rtc_read(REG_STATUS_B);
223
224     cli();
225     if (!(regB & 0x04)) { // BCD mode
226         rtc_write(REG_SECONDS, bin_to_bcd(time->second));
227         rtc_write(REG_MINUTES, bin_to_bcd(time->minute));
228         rtc_write(REG_HOURS,  bin_to_bcd(time->hour));
229     } else { // Binary mode
230         rtc_write(REG_SECONDS, time->second);
231         rtc_write(REG_MINUTES, time->minute);
232         rtc_write(REG_HOURS,  time->hour);
233     }
234     sti();
235 }
236

```

```

237 void get_date(rtc_date_t *date) {
238     uint8_t regB = rtc_read(REG_STATUS_B);
239
240     date->day    = rtc_read(REG_DAY);
241     date->month  = rtc_read(REG_MONTH);
242     date->year   = rtc_read(REG_YEAR);
243
244     // Convert from BCD if needed
245     if (!(regB & 0x04)) {
246         date->day    = bcd_to_bin(date->day);
247         date->month  = bcd_to_bin(date->month);
248         date->year   = bcd_to_bin(date->year);
249     }
250 }
251
252 void set_date(const rtc_date_t *date) {
253     uint8_t regB = rtc_read(REG_STATUS_B);
254
255     cli();
256     if (!(regB & 0x04)) { // BCD mode
257         rtc_write(REG_DAY, bin_to_bcd(date->day));
258         rtc_write(REG_MONTH, bin_to_bcd(date->month));
259         rtc_write(REG_YEAR, bin_to_bcd(date->year));
260     } else { // Binary mode
261         rtc_write(REG_DAY, date->day);
262         rtc_write(REG_MONTH, date->month);
263         rtc_write(REG_YEAR, date->year);
264     }
265     sti();
266 }
267
268 void clock_help(void) {
269     const char *helpMsg =
270         "\r\n\033[33mclock\033[0m
271         [\033[36mget\033[0m|\033[36mset\033[0m|\033[36mhelp\033[0m]  [\033[36mtime\033[0m
272         <\033[36mHH:MM:SS\033[0m>|\033[36mdate\033[0m <\033[36mMM/DD/YY\033[0m>]\r\n"
273         "  \033[33mclock\033[0m                                prints the current date and
274         time.\r\n"
275         "  \033[33mclock\033[0m \033[36mget\033[0m \033[36mtime\033[0m
276         prints the current time as: hour:minute:second.\r\n"
277         "  \033[33mclock\033[0m \033[36mget\033[0m \033[36mdate\033[0m
278         prints the current date as: month/day/year.\r\n"
279         "  \033[33mclock\033[0m \033[36mset\033[0m \033[36mtime\033[0m
280         <\033[36mHH:MM:SS\033[0m>      sets the current time to: hour:minute:second.\r\n"
281         "  \033[33mclock\033[0m \033[36mset\033[0m \033[36mdate\033[0m
282         <\033[36mMM/DD/YY\033[0m>      sets the current date to: month/day/year.\r\n"
283         "  \033[33mclock\033[0m \033[36mhelp\033[0m                                prints this
284         message\r\n"
285         "\r\n";
286         sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
287     }
288
289 void tz_correction(void){
290     rtc_time_t t;
291     rtc_date_t d;
292     get_time(&t);
293     get_date(&d);
294
295     // Determine if subtracting 4 hours goes to previous day
296     int new_hour = t.hour - 4;

```

```
289     if (new_hour < 0) {
290         new_hour += 24;
291
292         // decrement day
293         d.day--;
294
295         // handle month/year wraparound
296         if (d.day < 1) {
297             d.month--;
298             if (d.month < 1) {
299                 d.month = 12;
300                 d.year--;
301             }
302
303             // set day to last day of new month
304             switch (d.month) {
305                 case 1: case 3: case 5: case 7: case 8: case 10: case 12:
306                     d.day = 31; break;
307                 case 4: case 6: case 9: case 11:
308                     d.day = 30; break;
309                 case 2:
310                     d.day = (d.year % 4 == 0) ? 29 : 28;
311                     break;
312             }
313         }
314     }
315
316     t.hour = new_hour;
317
318     // commit corrected time and date
319     set_time(&t);
320     set_date(&d);
321 }
322 }
```

```
1 #ifndef EXIT_H
2 #define EXIT_H
3 /**
4 * @file exit.h
5 * @author Caleb Edwards
6 * @brief Header file for the exit command used in the command handler.
7 * Exits the terminal when called and confirmed by the user.
8 */
9
10 void exit_help(void);
11
12 /**
13 * @brief Begins the shutdown process when the user
14 * types 'exit' in the terminal. Confirmation by typing
15 * 'Y' or 'n' is then required to completely exit.
16 * @param arg_counter Counts the number of arguments input.
17 * @param arg_vector Stores the arguments.
18 * @return int return 1 to confirm exit and 0 to return to terminal.
19 */
20 int exit_command(const char *args);
21
22#endif
23
```

```

1 #include <sys_req.h>
2 #include <string.h>
3 #include "comhand.h"
4 #include "exit.h"
5 #include <sys_req.h>
6
7 void exit_help(void) {
8     const char *helpMsg =
9         "\r\n\033[33mexit\033[0m [\033[36mhelp\033[0m|\033[36mforce\033[0m]\r\n"
10    " \033[33mexit\033[0m           prompts for confirmation before shutting
down\r\n"
11    " \033[33mexit\033[0m \033[36mhelp\033[0m      displays this message\r\n"
12    " \033[33mexit\033[0m \033[36mforce\033[0m      shuts down without
confirmation\r\n"
13    "\r\n";
14     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
15 }
16
17 int exit_command(const char *args) {
18     if (args == NULL || *args == '\0') {
19         // exit confirmation
20         const char *confirmationMsg = "\r\nExit? (Y/n): \r\n";
21         sys_req(WRITE, COM1, confirmationMsg, strlen(confirmationMsg));
22
23         // exit confirmation logic
24         while (1) {
25             char confirmation[2] = {0};
26             int nreadExit = sys_req(READ, COM1, confirmation, sizeof(confirmation));
27             confirmation[nreadExit] = '\0';
28
29             if (confirmation[0] == 'y' || confirmation[0] == 'Y') {
30                 const char *exitMsg = "\r\nExiting...\r\n";
31                 sys_req(WRITE, COM1, exitMsg, strlen(exitMsg));
32                 return 1; // tell comhand to exit
33
34             } else if (confirmation[0] == 'n' || confirmation[0] == 'N') {
35                 const char *returnMsg = "\r\nReturning...\r\n";
36                 sys_req(WRITE, COM1, returnMsg, strlen(returnMsg));
37                 return 0;
38             } else {
39                 const char *invalidMsg = "\r\nInvalid input. Please type 'Y' or
'n'.\r\n";
40                 sys_req(WRITE, COM1, invalidMsg, strlen(invalidMsg));
41             }
42         }
43     }
44     else {
45         if (strcmp(args, "help") == 0) {
46             exit_help();
47             return 0;
48         }
49     else if (strcmp(args, "force") == 0 || strcmp(args, "f") == 0) {
50         return 1;
51
52     }
53     else {
54         const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
55         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
56         exit_help();

```

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

```
1 #ifndef HELP_H
2 #define HELP_H
3 /**
4 * @file help.h
5 * @author Caleb Edwards
6 * @brief Header for the help command used in command handler.
7 * Used to list the commands available to the user.
8 */
9
10 void help_message(void);
11
12 /**
13 * @brief Prints all commands available into the terminal
14 * when the user types 'help' in the input.
15 */
16 void help_command(const char *args);
17
18 /**
19 * @brief Prints all commands individual help functions.
20 */
21 void help_verbose(void);
22 #endif
23
```

```

1 #include <sys_req.h>
2 #include <string.h>
3 #include "help.h"
4 #include "exit.h"
5 #include "version.h"
6 #include "clock.h"
7 #include "setPriority.h"
8 #include "showPCB.h"
9 #include "ready.h"
10 #include "init.h"
11 #include "block.h"
12 #include "yield.h"
13 #include "loadR3.h"
14 #include "mcb/allocate.h"
15 #include "mcb/free.h"
16 #include "mcb/show.h"
17 #include "alarm.h"
18
19
20
21 void help_message(void) {
22     // command help
23     const char *helpMsg =
24         "List of Commands [arguments]:\r\n\r\n"
25         "\033[33mhelp\033[0m           [\033[36mverbose\033[0m]\r\n"
26         "\033[33mversion\033[0m
27         [\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
28         "\033[33mexit\033[0m
29         [\033[36mhelp\033[0m|\033[36mforce\033[0m]\r\n"
30         "\033[33mclock\033[0m
31         [\033[36mget\033[0m|\033[36mset\033[0m|\033[36mhelp\033[0m]  [<\033[36mdate\033[0m|<\033[36mtime\033[0m]|\r\n"
32         "\033[33mcreate\033[0m          [<\033[36mname\033[0m|\033[36mhelp\033[0m]
33         [<\033[36mclass\033[0m]  [<\033[36mpriority\033[0m]|\r\n"
34         "\033[33mdelete\033[0m
35         [<\033[36mname\033[0m|\033[36mhelp\033[0m]\r\n"
36         "\033[33malarm\033[0m          [\033[36mhelp\033[0m|<\033[36mtime\033[0m]
37         [<\033[36mmessage\033[0m]|\r\n"
38         "\033[33mshow pcb\033[0m
39         [<\033[36mname\033[0m|\033[36mready\033[0m|\033[36mblocked\033[0m|\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
40         "\033[33mpriority set\033[0m  [<\033[36mname\033[0m|\033[36mhelp\033[0m]
41         [<\033[36mpriority\033[0m]|\r\n"
42         "\033[33msuspend\033[0m
43         [<\033[36mname\033[0m|\033[36mhelp\033[0m]\r\n"
44         "\033[33mresume\033[0m
45         [<\033[36mname\033[0m|\033[36mhelp\033[0m]\r\n"
46         "\033[33mblock\033[0m
47         [<\033[36mname\033[0m|\033[36mhelp\033[0m]\r\n"
48         "\033[33munblock\033[0m
49         [<\033[36mname\033[0m|\033[36mhelp\033[0m]\r\n"
50         "\033[33myield\033[0m          [\033[36mhelp\033[0m]\r\n"
51         "\033[33mload\033[0m
52         [<\033[36mname\033[0m|\033[36mhelp\033[0m|\033[36msuspended\033[0m]\r\n"
53         "\033[33mallocate\033[0m
54         [<\033[36mbytes\033[0m|\033[36mhelp\033[0m]\r\n"
55         "\033[33mfree\033[0m
56         [<\033[36maddress\033[0m|\033[36mhelp\033[0m]\r\n"
57         "\033[33mshow mcb\033[0m
58         [\033[36mallocated\033[0m|\033[36mfree\033[0m]

```

```

<\033[36maddress\033[0m>| \033[36mhelp\033[0m]\r\n"
43     "\033[33mclear\033[0m\r\n\r\n"
44 "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";
45     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
46 }
47
48 void help_help(void) {
49     const char *helpMsg =
50         "List of Commands [arguments]:\r\n\r\n"
51         "\033[33mhelp\033[0m [\033[36mverbose\033[0m]\r\n"
52         "  \033[33mhelp\033[0m           prints a basic help message\r\n"
53         "  \033[33mhelp\033[0m \033[36mverbose\033[0m   prints this
message.\r\n\r\n";
54     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
55 }
56
57 void help_verbose(void) {
58     help_help();
59     exit_help();
60     version_help();
61     clock_help();
62     create_help();
63     delete_help();
64     alarm_help();
65     show_pcb_help();
66     set_priority_help();
67     suspend_help();
68     resume_help();
69     block_help();
70     unblock_help();
71     yield_help();
72     load_help();
73     allocate_help();
74     free_help();
75     show_mcb_help();
76
77     const char *clearMsg =
78         "\r\n";
79     sys_req(WRITE, COM1, clearMsg, strlen(clearMsg));
80
81     const char *docMsg =
82         "\033[33mclear\033[0m clears the terminal\r\n";
83     sys_req(WRITE, COM1, docMsg, strlen(docMsg));
84 }
85
86 void help_command(const char *args) {
87
88     if (args == NULL || *args == '\0') {
89         help_message();
90     }
91     else if (strcmp(args, "help") == 0) {
92         help_message();
93     }
94     else if (strcmp(args, "verbose") == 0 || strcmp(args, "v") == 0 || strcmp(args,
"all") == 0 || strcmp(args, "a") == 0) {
95         help_verbose();
96     }
97     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
```

```
1 #ifndef LOADR3_H
2 #define LOADR3_H
3
4 /**
5  * @file loadR3.h
6  * @brief Handles loading the R3 processes into the system
7  */
8
9 /**
10 * @brief Displays a help message for loading the processes
11 */
12 void load_help(void);
13
14 /**
15 * @brief Loads all 5 of the R3 processes
16 */
17 void loadR3(void);
18
19
20 /**
21 * @brief Loads all 5 of the processes, but in a suspended state
22 */
23 void loadR3_suspended(void);
24
25 /**
26 * @brief Loads a specific R3 process
27 * @param name The name of a specific process to load
28 */
29 void loadProcess(const char* name);
30
31 /**
32 * @brief Handles the selection of which load function to call
33 * @param args Argument for selecting which load function, for example "help"
34 */
35 void load_command(const char* args);
36
37#endif
38
```

```

1 #include <pcb.h>
2 #include <processes.h>
3 #include <loadR3.h>
4 #include "sys_req.h"
5 #include "string.h"
6 #include "ready.h"
7
8 // Displays help message
9 void load_help(void){
10     char *helpMsg = "\r\n\033[33mload\033[0m
[<\033[36mname\033[0m>]\033[36mhelp\033[0m|\033[36msuspended\033[0m]\r\n"
11         " \033[33mload\033[0m           loads all 5 premade processes\r\n"
12         " \033[33mload \033[0m<\033[36mname\033[0m>      loads a specific process,
13 name is like 'proc1'\r\n"
14         " \033[33mload \033[36msuspended \033[0mloads all 5 processes as
15 suspended\r\n"
16         " \033[33mload \033[36mhelp \033[0m      prints this message\r\n"
17         "\r\n";
18     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
19 }
20
21 // Loads all R3 processes
22 void loadR3(void){
23     // Struct used for setup, reused for each process
24     struct pcb* newProc = pcb_setup("proc1", CLASS_USER, 1, proc1);
25     pcb_insert(newProc);
26     newProc = pcb_setup("proc2", CLASS_USER, 1, proc2);
27     pcb_insert(newProc);
28     newProc = pcb_setup("proc3", CLASS_USER, 1, proc3);
29     pcb_insert(newProc);
30     newProc = pcb_setup("proc4", CLASS_USER, 1, proc4);
31     pcb_insert(newProc);
32     newProc = pcb_setup("proc5", CLASS_USER, 1, proc5);
33     pcb_insert(newProc);
34     char *msg = "Processes have been loaded\r\n";
35     sys_req(WRITE, COM1, msg, strlen(msg));
36 }
37
38 // Loads and suspends all processes
39 void loadR3_suspended(void){
40     loadR3();
41     suspend_pcb("proc1");
42     suspend_pcb("proc2");
43     suspend_pcb("proc3");
44     suspend_pcb("proc4");
45     suspend_pcb("proc5");
46 }
47
48 // Loads a specific process
49 void loadProcess(const char* name){
50     // Struct for a named process
51     struct pcb* newProc;
52     char *msg = "Process has been loaded\r\n";
53     if (strcmp(name, "proc1")==0){
54         newProc = pcb_setup("proc1", CLASS_USER, 1, proc1);
55         pcb_insert(newProc);
56         sys_req(WRITE, COM1, msg, strlen(msg));
57     }
58     else if (strcmp(name, "proc2")==0){
59         newProc = pcb_setup("proc2", CLASS_USER, 1, proc2);
60     }

```

```

58     pcb_insert(newProc);
59     sys_req(WRITE, COM1, msg, strlen(msg));
60 }
61 else if (strcmp(name, "proc3")==0){
62     newProc = pcb_setup("proc3", CLASS_USER, 1, proc3);
63     pcb_insert(newProc);
64     sys_req(WRITE, COM1, msg, strlen(msg));
65 }
66 else if (strcmp(name, "proc4")==0){
67     newProc = pcb_setup("proc4", CLASS_USER, 1, proc4);
68     pcb_insert(newProc);
69     sys_req(WRITE, COM1, msg, strlen(msg));
70 }
71 else if (strcmp(name, "proc5")==0){
72     newProc = pcb_setup("proc5", CLASS_USER, 1, proc5);
73     pcb_insert(newProc);
74     sys_req(WRITE, COM1, msg, strlen(msg));
75 }
76 else{
77     char *errorMsg = "\033[31mError: Please enter a valid process name, like
'proc1'\033[0m\r\n";
78     sys_req(WRITE, COM1, errorMsg, strlen(errorMsg));
79     load_help();
80 }
81 }
82
83 void load_command(const char* args){
84     if (args==NULL || *args == '\0'){
85         loadR3();
86     }
87     else if (strncmp(args, "suspended", 9)==0){
88         loadR3_suspended();
89     }
90     else if (strncmp(args, "help", 4)==0){
91         load_help();
92     }
93     else{
94         loadProcess(args);
95     }
96 }
97

```

```
1 #ifndef VERSION_H
2 #define VERSION_H
3
4 #ifndef GIT_DATE
5 #define GIT_DATE "unknown"
6 #endif
7
8 #ifndef GIT_HASH
9 #define GIT_HASH "unknown"
10#endif
11
12 #ifndef GIT_DIRTY
13 #define GIT_DIRTY "unknown"
14#endif
15
16 /**
17 * @file version.h
18 * @brief Displays the current version of MacaroniOS.
19 */
20
21
22 /**
23 * @brief Prints help information related to the version command.
24 */
25 void version_help(void);
26
27 /**
28 * @brief Displays the latest version.
29 */
30 void version_latest(void);
31
32 /**
33 * @brief Displays the past and present versions.
34 */
35 void version_history(void);
36
37 /**
38 * @brief Main handler for the version command.
39 * @param args The argument string passed after 'version'
40 */
41 void version_command(const char *args);
42
43#endif
44
```

```

1 #include <sys_req.h>
2 #include <string.h>
3 #include "version.h"
4
5 void version_help(void) {
6     const char *helpMsg =
7         "\r\n\033[33mversion\033[0m [\033[36mall\033[0m|\033[36mhelp\033[0m]\r\n"
8         " \033[33mversion\033[0m      prints information about the current
9 version as: R<x> <date> <hash> (<status>)\r\n"
10        " \033[33mversion\033[0m \033[36mall\033[0m      prints information about all
11 version history as: R<x> <date> <hash>\r\n"
12        " \033[33mversion\033[0m \033[36mhelp\033[0m      prints this message\r\n"
13        "\r\n";
14     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
15 }
16
17 void version_latest(void) {
18     char buffer[128];
19     char *write_ptr = buffer;
20     const char *read_ptr;
21
22     // main version
23     *write_ptr++ = 'R';
24     *write_ptr++ = '6';
25     *write_ptr++ = ' ';
26
27     // git metadata
28     read_ptr = GIT_DATE;
29     while (*read_ptr) {
30         *write_ptr++ = *read_ptr++;
31     }
32     *write_ptr++ = ' ';
33
34     read_ptr = GIT_HASH;
35     while (*read_ptr) {
36         *write_ptr++ = *read_ptr++;
37     }
38     *write_ptr++ = '(';
39     read_ptr = GIT_DIRTY;
40     while (*read_ptr) {
41         *write_ptr++ = *read_ptr++;
42     }
43     *write_ptr++ = ')';
44
45     *write_ptr++ = '\r';
46     *write_ptr++ = '\n';
47     *write_ptr = '\0';
48
49     // print out
50     sys_req(WRITE, COM1, buffer, write_ptr - buffer);
51 }
52
53 void version_history(void) {
54     const char *historyMsg =
55         "R5 2025-11-20 431691d\r\n"
56         "R4 2025-10-31 266935a\r\n"
57         "R3 2025-10-20 9b0243f\r\n"
58         "R2 2025-10-03 15c9229\r\n"

```

```
59     "R1 2025-09-12 956ee13\r\n"
60     "R0 2025-08-24 6738747\r\n";
61 sys_req(WRITE, COM1, historyMsg, strlen(historyMsg));
62 }
63
64 void version_command(const char *args) {
65     if (args == NULL || *args == '\0') {
66         version_latest();
67     }
68     else if (strcmp(args, "all") == 0) {
69         version_latest();
70         version_history();
71     }
72     else if (strcmp(args, "help") == 0) {
73         version_help();
74     }
75     else {
76         const char *argMsg = "\033[31mInvalid argument. Please try
again.\033[0m\r\n";
77         sys_req(WRITE, COM1, argMsg, strlen(argMsg));
78         version_help();
79     }
80 }
81
```

```
1 #ifndef YIELD_H
2 #define YIELD_H
3
4 /**
5  * @file yield.h
6  * @brief Adds yield functionality so that a process can be set to IDLE
7  */
8
9 /**
10 * @brief Yields the current process
11 */
12 void yield(void);
13
14 /**
15 * @brief Prints a help message for the yield function
16 */
17 void yield_help(void);
18
19 /**
20 * @brief Handles arguments given in the command handler for yield
21 * @param args Arguments for the yield command
22 */
23 void yield_command(const char* args);
24
25 #endif
26
```

```
1 #include "sys_req.h"
2 #include <string.h>
3
4 void yield(void){
5     sys_req(IDLE);
6 }
7
8 void yield_help(void){
9     const char *helpMsg = "\r\n\033[33myield\033[0m [ \033[36mhelp\033[0m]\r\n"
10    " \033[33myield\033[0m           yields the currently running process\r\n"
11    " \033[33myield \033[36mhelp   \033[0mprints this message\r\n"
12    "\r\n";
13     sys_req(WRITE, COM1, helpMsg, strlen(helpMsg));
14 }
15
16 void yield_command(const char* args){
17     if (args == NULL || *args=='\0'){
18         yield();
19     }
20     else if (strcmp(args, "help")==0){
21         yield_help();
22     }
23     else{
24
25     }
26 }
27 }
```