**Programmer’s Manual (MPX OS)**

1. **commhand():-**
   1. **Inputs:** none
   2. **Outputs:** none
   3. **Description:** Within Kernel, this function is called, which in return runs an endless loop to continuously ask the user for commands. This calls sys\_req(READ,…) which goes to polling for collecting inputs. It then uses strtok() to separate the commands into individual strings separated by spaces. Each command is processed and then the corresponding steps / lines of code are implemented.
   4. **Variables:**  
      int size: size of the command buffer  
      int sizeBuff: size of the time/date buffers  
      char buffer[]: the main command buffer  
      int dateBuff[] / int timeBuff[]: the buffers to store current date and time values.  
      char \*token: pointer to the first “word” in command buffer before a space  
      const char split[]: the character that splits the command buffer
2. **get\_Time(int \*pointer):-**
   1. **inputs:** a pointer to the time buffer
   2. **outputs:** none
   3. **macros:** 0x00 defined as Sec, 0x02 defined as Min and 0x04 defined as Hr.
   4. **Description:** Grabs the hours, minutes and seconds values from their respective RTC registers and fills them in the buffer given by the pointer. Because the values in the registers are stored in BCD form, they are first converted into binary form and then stored in the buffer one digit at a time.
3. **set\_Time(int hour, int minutes, int seconds):**-
   1. **Inputs:** int hours, int minutes, int seconds
   2. **Outputs:** none
   3. **Description:** Changes the current time of the machine to the provided values. This is achieved by converting each input into BCD and then storing each input to its respective RTC register. This requires that the interrupts be disabled first before performing any changes. Once the values are stored, the interrupts are enabled again.
   4. **Variables:**   
      int s\_time: the BCD results of each input that is used to store the information to the registers.
4. **getdate(int \*ptr):**-
   1. **Inputs:** int \*pointer
   2. **Outputs:** none
   3. **Description:** Grabs the year, month and day values from the RTC registers and stores them in the buffer. Similar to get\_Time(), this function takes the values that are in BCD and converts them to regular binary, where the buffer is then filled one digit at a time.
   4. **Variables:**  
      int I / int j: the int values that would split the BCD value in half, each being converted to binary to represent the actual value to be stored.
5. **setdate(int year, int month, int day):**-
   1. **Inputs:** int year, int month, int day
   2. **Outputs:** none
   3. **Description:** Changes the current date of the machine to the provided values. Similar to set\_Time(), this converts each input into BCD form before entering the values in the registers, after disabling the interrupts. Once it is done, the interrupts are enabled again.
6. **help():**-
   1. **Inputs:** none
   2. **Outputs:** none
   3. **Description:** Once invoked in Terminal, it simply displays available commands. The messages inform the user of what commands he/she is able to perform and what syntax or parameters are needed.
7. **version():**-
   1. **Inputs:** none
   2. **Outputs:** none
   3. **Description:** Once invoked, it will print a message informing the user of the version and release date of this OS.
8. **Polling(char\* buffer, int count):**-
   1. **Inputs:** char\* buffer, int count
   2. **Outputs:** number of characters entered (count)
   3. **Description:** Continuously checks for available data entered from the serial port by checking the status register’s least significant bit (register COM1+5). When data is available, it will be taken from the data register and be stored in the buffer, while increasing count for each alphanumerical letter inserted into the buffer. This builds the user input from the terminal into the buffer while also printing each character onto the screen. Polling also takes into consideration special keystrokes and performs special actions depending on what keys were pressed. It is called from commhand() whenever a task has been performed.
   4. **Variables:**  
      int i: used for indexing of the buffer.  
      int counter: an int data type storing the size of the buffer.
9. **kmain():**-
   1. **Inputs:** none
   2. **Outputs:** none
   3. **Description:** Performs all the initialization steps needed before MPX is ready for the user. Starts by initializing the serial port and setting the correct module to use. Then it will initialize the descriptor tables and interrupt vector tables. Lastly, it will initialize paging before invoking commhand() for processing user commands until shutdown.
10. **print(char\* str, int size):-**
    1. **Inputs:** char\* str, int size
    2. **Outputs:** none
    3. **Description:** invokes an instance of sys\_req(WRITE,..) by using ‘str’ as the string to be printed and ‘size’ as the size of the string to be printed.
11. **itoa(int number, char\* str):**-
    1. **Inputs:** int number, char\* str
    2. **Outputs:** String representation of the number given
    3. **Description:** Creates an empty string and takes the number given. Performs a series of steps to convert the numbers into their equivalent ASCII codes (characters) and puts them in the string in reverse order. Using flip(), which in return invokes swap(), flips the string to the correct orientation before returning the string representation of the number.
    4. **Variables:**int isNegative: a flag to indicate whether the given number is positive or negative.  
       int temp: Stores the absolute value of the number (after indicating that its negative or positive)  
       int i: index of the string
12. **flip(char\* str, int length):-**
    1. **inputs:** char\* str, int length
    2. **outputs:** A flipped version of the provided string
    3. **description:** Given a string and its length, uses two indices at each end of the string. Traversing to the middle, each character at the index is switched with the other index’s character with the help of swap(). This results in a flipped version of the string. Once finished, it will return the flipped string.
    4. **Variables:**  
       int i/j: index variables.
13. **Swap(char \*a, char \*b):**-
    1. **Inputs:** char\* a/b
    2. **Outputs:** none
    3. **Description:** switches the contents of the two pointers with one another. Using a temp variable first, performs a number of switching to prevent loss of data.
    4. **Variables:**  
       char temp: temporarily stores a pointer’s content aside while switching is in progress.
14. **allocatePCB():**-
    1. **Inputs:** none
    2. **Outputs:** pcb pointer to new PCB.
    3. **Description:** uses sys\_alloc\_mem() to dynamically allocate enough memory for a new PCB. Also performs some initialization steps and gives some default values.
15. **setupPCB(char\* newname, unsigned char newClass, int newPriority):**-
    1. **Inputs:** char\* newname, unsigned char newClass, int newPriority
    2. **Outputs:** pcb pointer to new PCB.
    3. **Description:** invokes an instance of allocatePCB() and assigns it to a pointer. Using the given name, class and priority, it will perform error checking to these values before assigning them to the newly created PCB.
16. **freePCB(pcb\* name):**-
    1. **Inputs:** pcb\* name
    2. **Outputs:** char\* cast of sys\_free\_mem()
    3. **Description:** dynamically deallocates the memory assigned for the PCB. This is used when trying to permanently delete the PCB from memory.
17. **findPCB(char\* name):**-
    1. **Inputs:** char\* name
    2. **Output:** pointer to a PCB of the given name, or NULL
    3. **Description:** Using the given name, it will look through all queues for any PCB with a name matching what’s provided. When found, it will return a pointer to that PCB. If it fails to find a PCB from any queue, it will return NULL.
    4. **Variables:**  
       pcb\* ptr: the pointer to be returned when finding the PCB.
18. **insertPCB(pcb\* pntr):**-
    1. **Inputs:** pcb\* pntr
    2. **Outputs:** none
    3. **Description:** Given a pcb pointer, it will look at its priority, state and status. Then traverse through the correct queue and connect the pcb to the surrounding PCB’s in either priority or FIFO base.
19. **removePCB(pcb\* pntr):**-
    1. **Inputs:** pcb\* pntr
    2. **Outputs:** int indicating success or error
    3. **Description:** Given the pcb pointer, it will adjust connections to its next and previous pcbs and ensures that the queue remains intact after disconnecting the PCB.
20. **showPCB(pcb\* pntr):**-
    1. **Inputs:** pcb\* pntr
    2. **Outputs:** none
    3. **Description:** Given a pcb pointer, it will print to the terminal screen any relevant information about the PCB, such as name, class, state, status and priority.
21. **showqueue(char\* queue):**-
    1. **Inputs:** char\* queue
    2. **Outputs:** none
    3. **Description:** based on what keyword is provided in the string, it will traverse through the appropriate queue and invokes showPCB() for each individual PCB. This allows for printing an entire queue easily.
22. **Sys\_call(context\* registers):**-
    1. **Inputs:** context\* registers
    2. **Outputs:** context of a process for execution (either new or current process)
    3. **Description:** Main function used for allowing context switching between different processes. Uses a global variable “cop” representing the currently running process. This function is called whenever a process voluntarily suspends or completely ends execution, triggering an interrupt. After context of previous process is saved, this function replaces the currently running process with the next process that is sitting at the top of the ready queue. Based on whether the process is idling or exiting, sys\_call() either reinserts the process back in the ready queue or frees the process’ memory.
    4. **Variables:  
       context\* registers:** a struct representing a multiple of registers that a process uses to store its information (i.e. context).  
       **struct queue\* ready:** a pointer reference to the ready queue for the function’s use.  
       **pcb\* cop:** A global pcb variable simulating the currently running process.  
       **context\* old\_context:** A global variable representing the context of the previously running process.
23. **returnQueue():-**
    1. **Inputs:** none
    2. **Outputs:** pointer reference to the ready queue’s head
    3. **Description:** Mainly used by sys\_call() as a method for accessing the ready queue that would usually be inaccessible due to it residing in a different file.
24. **Loadr\_pcb(char\* name, unsinged char class, int status, int priority, u32int func):-**
    1. **Inputs:** char\* name, unsinged char class, int status, int priority, u32int func
    2. **Outputs:** none
    3. **Description:** Used to load a function into memory as a process in the ready queue. It starts by creating a new PCB using setupPCB() and the accompanying initial variables. Initializes the PCB’s registers to store information about itself in and sets each register the appropriate value. After register initialization the newly created process is inserted into the ready queue based on priority using insertPCB(). This PCB performs the tasks of the associated function given by the func variable.
    4. **Variables:**  
       pcb\* new\_pcb: The new PCB to be created and initialized to represent the function.  
       context\* cp: The new PCB’s register values (context).
25. **Void infinite():-**
    1. **Inputs:** none
    2. **Outputs:** none
    3. **Description:** Used to continuously print a message before triggering an interrupt to context switch. Mainly used by loadr\_pcb() to create a process that does this task.
26. **createAlarm(char\* message, int time):**-
    1. **Inputs:** char\* message, int time
    2. **Outputs:** none
    3. **Description:** Used to create a new alarm to be stored in the alarm queue. Each alarm consists of a message to print and an int indicating the amount of time in seconds before the alarm message should be printed. It starts by storing the alarm’s creation time and the alarm’s delay time (how long to wait before printing) and then inserts the alarm in the alarm queue.
27. **checkTime():**-
    1. **Inputs:** none
    2. **Outputs: none**
    3. **Description:** Used to keep track of all created alarms and removing any alarms that have went off. Begins by measuring the current time and compares it with every alarm’s creation time added with the delay time. If the current time was larger than the alarm’s creation and delay times, then the alarm’s message is printed and the alarm is removed from the queue. This checks all the alarms in the queue in each iteration to be able to print multiple alarms at once.
28. **getCurrentTime():**-
    1. **Inputs:** none
    2. **Outputs:** int value of the current time of this function’s execution
    3. **Description:** Used mainly by the alarm functions to compare creation time of alarms and the “checking” time to determine if an alarm message should be printed or not. This uses get\_Time() to get a string representation of the current time and converts this string to an int value that sums all seconds, minutes and hours in the string in seconds’ format. This function essentially returns the total amount of seconds that had passed from the start of the day to the time of calling the function.
29. **Struct cmcb (typedef: CMCB):-**
    1. **Int type:** Values indicating either free (0) or allocated (1)
    2. **U32int address:** The address of the CMCB block in memory
    3. **Int size:** Value indicating the size the block is occupying
    4. **Struct cmcb\* next:** CMCB pointer to the next CMCB in the list
    5. **Struct cmcb\* prev:** CMCB pointer to the previous CMCB in the list.

1. **Struct lmcb (typedef: LMCB):-**
   1. **Int type:** Value indicating either free (0) or allocated (1)
   2. **Int size:** Value indicating the size the block is occupying.
2. **Struct list:**-
   1. **CMCB\* head:** A pointer to the head of the memory list.
   2. **CMCB\* tail:** A pointer to the tail of the memory list.
3. **Init\_heap(int size):**-
   1. **Inputs:** int size
   2. **Outputs:** address pointer to the heap’s head
   3. **Description:** Used to initially allocate memory for the heap based on the size provided. Using kmalloc(), the allocated memory is assigned to the heap’s head and that will then be used by CMCBs to create a list of memory blocks. This also initializes the memory list by setting the head and tail of the list to point to the first CMCB in the heap. Before returning, the head CMCB is made sure to be pointing to NULL in both directions, and then print a confirmation message at the end.
4. **allocateMemory(int size):**-
   1. **Inputs:** int size
   2. **Outputs:** address value of allocated memory block
   3. **Description:** Used to allocate an appropriate number of bytes for a process to occupy. Going through the list of memory blocks, its checked whether its free or not. If it is free, its then checked if its size is sufficient for the process. It will then determine if the found block is large enough to be divided to an allocated block and a remainder (free) block to be used by other allocation requests. If the memory block should be divided, then a new CMCB is created for that free block and the connections for previous and next pointers are adjusted to allow this CMCB to be placed correctly in the list. At the end of allocation, the address of the block is retuned. If the allocation was not successful, it will return ‘0’ instead.
5. **freeMemory(u32int address):-**
   1. **Inputs:** u32int address
   2. **Outputs:** none
   3. **Description:** This is called when a process has finished and is no longer needed to occupy the allocated block. Using the address of the block, the memory list is traversed to look up the given address value. When a matching address is found, the block is checked if it is already free or not, in which case the function would print a message indicating that the block is already free and returning. If the block is not free, then its type is changed to free and then the previous and next blocks are checked to see if they are also free. If free blocks are adjacent to each other in this scenario, they are combined together to create a larger free block for future allocations. This is done by removing the CMCBs that occupy the middle areas and adjusting the connections of the new block’s CMCB. The new CMCB’s size is also updated to show the size of combining the blocks.
6. **isEmpty():**-
   1. **Inputs:** none
   2. **Outputs:** int value indicating either True or False
   3. **Description:** Used to check if the heap is currently empty or if allocated blocks exist. This is performed by creating a CMCB pointer starting at the head of the block list. As the pointer traverses through the list, each block’s type is checked and if a block’s type is found to be not free, it will return False to indicate that an allocated block exists. When the pointer traverses through the entire list and finishes, it means that no allocated blocks were found, and so the function returns True.
7. **showList(int printType):**-
   1. **Inputs:** int printType
   2. **Outputs:** none
   3. **Description:** Used to print the information for each block of memory of the given type. This starts by creating a CMCB pointer to the head of the memory list and traverse though it. For each block that matches the given type, the block’s address and size (contained within the block’s CMCB) are printed to the screen. Each block print is followed by a dividing line that is printed to separate multiple blocks from each other. The address and size of the blocks are stored in a string using itoa() and then used in the print function. This loop is executed until the end of the list is reached.