Group 11 - Programmer's Manual

kernel/kmain.c

As of R4, kmain.c loads two processes. Idle (from modules/mpx_supt.c) and comhand, which are priority 9 and 1 respectively. Comhand will call *sysreq(IDLE, ..)* every 1 second (see polling).

Serial.c

`int *polling(char *buffer, int *count)`

brief: reads data one byte at a time from the serial port
param: buffer – stores the data read into this char array
param: count – max amount of bytes buffer can store
return: int* - number of characters read from the serial port

Suspends execution of the system to let the user to enter ascii characters into the buffer. The current contents of the buffer are printed to the screen along with the pipe symbol "|" to indicate the position of the cursor. Control keys such as the arrows, home, end, backspace, and delete keys move the cursor and delete characters adjacent to it. The user pressing enter will end the clear the line, print the contents of the buffer, print a new line, and exit the polling function.

Calls *sysreq(IDLE..)* every 1 second based on the internal clock to allow other functions to execute in the background.

Functions used to assist in polling:

`const char cursor_symbol = '|'`

Changing this variable will change the symbol that represents the cursor.

`char digit_to_char(int a)`

Returns the char corresponding to integer `a` which may be 0-9.

`void print_int(int a)`

Prints integer *a* to console

`void clear_line(int count)`

Print carriage return and print *count+4* spaces to clear current line on terminal.

`void string_insert(char* buffer, int buff_len, int* count, int index, char c)`

Inserts character *c* at given *index*, shifting everything after *index* to the right.

`void string_remove(char* buffer, int* count, int index)`

Removes the character from the buffer string located at *index*, shifting characters after *index* to the left. Decreases **count* by one.

`void string_print_cursor(char* buffer, int count, int cursor_index)`

Prints the content of *buffer* with the cursor inserted at *cursor_index*

versionAndShutdown.c

`int version();`

Invoked by command '*version*'.

Prints *const char** containing version and compile date. Returns *0.*

`int shutdown();`

Invoked by command '*shutdown'*.

Prompts the user to confirm with 'y\n'. Returns 1 if system is to shut down.

Terminates comhand loop.

getSetDateTime.c

`void getDate();`

Invoked by command '*getdate*'.
Retrieves century, year, month, and day from the RTC, converts them from BCD, and then prints them to console.

`void getTime();`

Invoked by command '*gettime*'.
Retrieves hour, minute, and second from the RTC, converts them from BCD, and then prints them to console.

`void setDate(char* str);`

Invoked by the command '*setdate <yyyy-mm-dd>*'.
Accepts a string of the form "*yyyy-mm-dd*", extracts the century, year, etc. converts them to BCD and stores them in the RTC.
Note that setDate() does NOT detect malformed or invalid input. It will write whatever is given into the RTC.

`void setTime(char* str);`

Invoked by the command '*settime <hh:mm:ss>*'.
Accepts a string of the form "*hh:mm:ss*", extracts the hour, minute, second, converts

them to BCD and stores them in the RTC.
Note that setTime() does NOT detect malformed or invalid input. It will write whatever is given into the RTC.

`unsigned char int_to_bcd(int num)`

Converts *num* into BCD and returns it in BCD form. This function does not properly handle integers out of the range of 1 byte in BCD. For example, *num=100* would need to return 12 bits, 4 bits for each digit, but the return type only hold 8 bits. In this case the BCD for 00 will be returned.

`int bcd_to_int(char bcd)`

Converts a char encoded in BCD to an integer and returns it. *bcd=0b10010000* will return 90.

commandHandler.c

`int comHand()`

Continually loops, prompting the user to enter a command. Stores the user input into a variable *char* buffer*. The *buffer* array scanned for the first space character and then separates the string before the space into the *command* and string following as the *argument*. This space is changed to NULL to make comparison easier. If there is no space then argument is set to NULL. The *command* section of the buffer is then compared to the names of existing commands, if a match is found then the corresponding command is executed and if it accepts arguments, *argument* is passed to it. If *command* is not a valid command name an error message is printed. *comHand()* only terminates when *shutdown* is called.

help.c

help.h contains several *#define* macros each containing a *const char** associated with a command. This is done so that a variable *char* help_strings[]* may be explicitly declared to contain each of the strings. When adding a new command, a new macro should be created in *help.h*, inserted into *help_strings*, and then added to the if-else chain in *help().*

`void help(char* arguments);`

Invoked by command '*help [arguments]'.*
Prints the help statement associated with the command stored in the *arguments* string. If *arguments* does not hold the name of a valid command or is *NULL*, then *help()* will print *help_strings[]*.

`void print_all_help()`

Prints all of the strings in *char* help_strings[].*

`char* help_strings[]`

Meant to contain all of the strings that describe usage for a command.

PCB.c

PCB.c contains several #define macros associated with either a PCB's state, or various array sizes. The queues for each process state are also initialized in this file.

Struct PCB

variables:

char name[PCB_max_name_len + 1]

PCB_max_name_len = 16

int priority

ranges 0-9

char process_class

user process : 1

system process : 2

char execution_state

blocked : 3

ready : 4

running : 5

char dispatching_state

suspended : 6

not suspended :7

unsigned char stack[PCB_stack_size]

PCB_stack_size = 1024

unsigned char mem2[128]

int stack_bottom

int stack_top

Struct context

variables:

segment registers

reg uint32_t gs, fs, es, ds

status control registers

reg uint32_t eip, cs, eflags

general purpose registers

reg uint32_t eax, ebx, ecx, edx, esi, edi, ebp, esp

PCB* allocate_pcb();

brief: allocates memory for a new PCB
return: pointer to created PCB struct

uses sys_alloc_mem() to allocate memory for a new PCB, initializing variables it can, and returns a pointer to the created PCB.

char free_pcb(PCB* pcb);

brief: frees the memory associated with the passed PCB pointer
param: pointer to a PCB
return: code indicating success or error

Uses sys_free_mem() to free all memory associated with a PCB, including the stack.

PCB* setup_pcb(char* name, char process_class, int priority);

brief: Allocates a new PCB and initializes it with provided data
param: string name of the PCB
param: char representing the process_class
param: its priority as an int
return: pointer to created PCB struct

Uses the allocate_pcb() method to allocate a new pcb and then initialize the PCB with the provided data. In case of an error, it returns null.

PCB* find_pcb(char* name);

brief: searches for the process with the provided name
param: string name of the PCB to find
return: pointer to PCB struct

Searches all queues for a PCB with the provided name. Returns null if it cannot be found.

void insert_pcb(PCB* pcb);

brief: Inserts a PCB into a queue
param: pointer to PCB

Inserts a PCB into the appropriate queue based on its state and priority.

char remove_pcb(PCB* pcb);

brief: removes a PCB
param: pointer to PCB
return: success or error code

Finds the PCB indicated, then removes it from the queue.

PCB* dequeue_next_pcb(queue q);**

Removes and returns the first PCB in the passed queue.

Queue.c

queue.c is specifically used by pcb.h. It contains basic methods for queues.

qn* create_q_node(void* pcb);

brief: creates a new node using the provided pcb pointer.

return: pointer of new node.

Allocates memory for the new node by using sys_alloc_mem and sets the “next” and “previous” pointers to NULL. It also uses the points to the provided pcb pointer.

queue* create_queue()

brief: creates a new queue.

return: pointer of new queue.

Creates a new queue by allocating memory using sys_alloc_memand sets the “front” and “end” pointers to NULL. The returns a pointer to the new queue.

void free_qn(qn* qn_)

Frees the node specified with the node pointer using sys_free_mem.

void free_queue(queue* q, void* pcb)

Frees the queue specified with the queue pointer using sys_free_mem.

void enqueue(queue* q, void* pcb)

Adds the specified pcb to the specified queue.

void dequeue(queue* q)

Changes the “front” pointer to the “next” pointer, frees the memory associated with the node, and returns a pointer to the dequeued node.

Show.c

Contains functions related to printing out data associated with processes.

void showPCB(PCB* pcb);

Prints the name, priority, class, state, and dispatching state of the indicated PCB.

void showReadyProcesses();

Prints the name, priority, class, state, and dispatching state of all ready processes.

void showBlockedProcesses();

Prints the name, priority, class, state, and dispatching state of all blocked processes.

void showAllProcesses();

Prints the name, priority, class, state, and dispatching state of all processes.

PCBCommands.c

void suspendPCB(char* name);

Suspends the indicated PCB.

void resumePCB(char* name);

Resumes the execution of the indicated PCB.

void setPriority(char* name, int priority);

Sets the priority of the indicated PCB.

alarm.c

void alarm(int sec, int minute, int hour, char* mess);

Checks if the current time is greater than the time the alarm is set to, if so, it prints the message to the screen, if not, it idles.

void setupAlarm(char* time, char* mess);

Initializes the alarm process with the provided time and message,

time entered in the format hh:mm:ss. Passes args to alarm() via the *mem* struct member of PCB, which sits below the stack.

infinite.c

void infinite();

Prints a message to the console indicating it is running. Default priority is 5.

void setup_infinite();

Sets up a PCB for the infinite process.

loadr3.c

void loadr3();

Sets up a PCB for each function in the procsr3 file.

void initializeRegs(PCB* pcb, void func);

Sets the registers of *pcb*'s context in preparation for it to execute *func*.

procsr3.c

void proc<1-5>();

Prints to screen when the process is loaded, and if it runs after terminating.

kernel/core/irq.s

sys_call_isr

Performs context switching. Pushes all x86 registers onto the stack followed by ds, es, fs, gs and esp. Calls sys_call, moves the stack, and then pops the register values for the new context.

serial_io_isr

Calls serial port interrupt.

kernel/core/syscall.h

extern PCB* cop;

Defines the currently operating process.

sys_call

Performs context switching (called by sys_call_isr). Queues the current pcb after updating its context. It then dequeues the next process and sets it to the current operating process. It then returns the context to the isr, and the new process's registers are loaded into the processor.

void serial_port_interrupt

Checks if the port is open then reads the interrupt ID register and determines the cause of the interrupt. After that it will call the appropriate second-level handler.

void read_interrupt

Reads a character from the input register and stores it into the ring buffer if the current status is not reading. It will then store the character to the ring buffer and then the input buffer if the status is reading. Otherwise if the transfer completes, the status will be set to idle, the event flag will be sent and the count value will be returned.

modules/mod5/mcb.h

struct MCB

variables:

size_t size

the size of blocks in bytes

char allocated

is either 0 or 1 indicating if the block is allocated

struct MCB* next

pointer to the next memory block

struct MCB* prev

pointer to the previous memory block

modules/mod5/initializeHeap.c

initializeHeap(size_t size)

Mallocs a MCB plus the size input. The MCB is initialized to be free and to have the start address of the usable memory. The headList points to the newly created MCB and the tailList is set to the headList.

modules/mod5/allocate_mem.c

uint32_t allocate_mem(uint32_t len);

Allocates memory in the heap with the size indicated, returning a uint32_t corresponding to the start address of the data allocated.

modules/mod5/free_mem.c

int free_mem(void* block);

frees the memory in the block that is tied to the specified pointer. Returns a 1 for success and a 0 for failure.

modules/mod5/showmem.c

void print_hex(uint32_t)

Prints the hex value associated with the provided value

char digit_to_char2(int a)

Converts the provided int to a char

void print_int2(int a)

Prints integer to console

show_MCB(MCB* _mcb)

Prints all information associated with each mcb that has been created including: mcb address, start address, end address, size, allocated status, and a pointer to the next and previous mcb.

void show_free_mem()

Shows all of the free MCBs

void show_allocated_mem()

Shows all of the allocated MCBs

void show_all_mem()

Shows all of the MCBs

modules/mod5/user_comms.c

void_user_alloc(char* str)

Allocates memory with str length, if enough space exists in the heap manager

void_user_free(char* hex)

Frees memory at hex, if an allocated MCB points to that start address

modules/mod6/DCB.h

struct DCB

variables:

int status

Monitors the status of the DCB. Can be 0 (for IDLE), 1 (for READING), or 2 (for Writing)

int open

Reflects if the DCB is open or not. Can be 0 (for OPEN) or 1 (for CLOSE)

int* event_f

pointer to the event flag

int chars_processed

This number represents how many characters have been processed by the DCB.

int to_be_processed

This number represents how many characters still need to be processed by the DCB.

char* buffer

Pointer to the buffer in the IOCB.

char ring_buffer

int ring_buf_full

Determines if the ring buffer is full or not.

int ring_buf_read_index

Index that is used to read the ring buffer

int ring_buf_write_index

Index that is used to write to the ring buffer

int com_open(void)

Checks parameters and initializes the DCB.

int com_close(void)

Checks the port, clears the open indicator in DCB, then disables the proper level in the PIC mask register.

int com_read(char *buf_p, int *count_p)

Validates parameters, initializes the buffer variables, clears the event flag, copies characters from the ring buffer to the specified buffer, and returns the count

int com_write(char *buf_p, int *count_p)

Validates parameters, checks the port, installs buffer pointer and counters to the DCB, clears the event flag, then puts the first character from the buffer and stores it to the output register.

modules/mod6/io_scheduler.c

Struct io_scheduler

Variables

queue* com1_queue

pointer to queue that stores IO operations for COM1

PCB* active_pcb

pointer to the active Process Control Block

DCB* com1

pointer to the Device Control Block used for COM1

int* event_ptr

pointer to the event flag

int event_flag

number that is used by the IO scheduler to determine events

void initialize_io_scheduler()

Allocates memory and calls com_open

int enqueue_iocb(IOCB* iocb)

Validates the passed IOCB and then adds it to the correct queue

PCB* process_next_iocb

Initiates the next IOCB transfer from the queue.

int* get_event_flag()

Returns the event flag

modules/mod6/IOCB.h

Struct IOCB

Variables

PCB* pcb

PCB struct from modules/data-structures/PCB.h

param* params

struct from modules/mpx_supt.h