CS 450 Module R1
Basic Interface Development

West Virginia University

Spring 2023

Goals

- Design a user interface
- Become familiar with MPX-specific functions
- Directly interface with hardware components

Background Knowledge: MPX Specific Functions

- In addition to the handful of normal standard library functions, MPX has several system-specific functions you will need to become familiar with
- First is the primitive dynamic memory management declared in `<memory.h>`
- Next is the user-to-kernel interface declared in `<sys_req.h>`
- Finally, there are macros for low-level I/O in `<mpx/io.h>`
- There are others, but those don't become relevant until later modules

`void *sys_alloc_mem(size_t size)`

- The dynamic memory allocator for MPX
- Remember that there is no standard library, so you don't have `malloc()`
- In a later module, you will implement callback functions that modify how this function works, but require no changes to existing code to use
- `size` is the number of bytes to allocate
- Returns a pointer to the newly allocated memory, or `NULL` on error

`int sys_free_mem(void *ptr)`

- The function used to return memory allocated by `sys_alloc_mem()` to the heap manager
- Does not actually free any memory as written – you will add that functionality via callbacks in a later module
- `ptr` is a pointer (returned from a previous call to `sys_alloc_mem()`) to free
- Returns 0 on success, non-zero on error

`int sys_req(int op_code, ...)`

- Simulates a system call to the kernel
- Probably the most important function the support library
- First parameter (`op_code`) determines the kernel action requested:
  - `READ` – read input from a serial port
  - `WRITE` – write contents of a buffer to a serial port
  - `IDLE` – (beginning in module 3) suspend the calling process
  - `EXIT` – (beginning in module 3) terminate the calling process
- Some operations require additional parameters
- Returns a negative value on error, otherwise based on `op_code`

`int sys_req(READ, device dev, char *buf, size_t len)`

- You will implement this functionality
- Reads data from a serial port
- `dev` must be one of the constants from `<mpx/device.h>`
- `buf` points to a buffer that data will be read into
- `len` is the size of the buffer/maximum number of bytes to read
- Returns the number of bytes read, or a negative value on error

`int sys_req(WRITE, device dev, const char *buf, size_t len)`

- Writes data to a serial port
- Code in `user` **must** call this instead of calling `serial_write()` directly
- `dev` must be one of the constants from `<mpx/device.h>`
- `buf` points to a buffer containing data to be output
- `len` is the number of bytes to output
- Returns the number of bytes written, or a negative value on error

`unsigned char inb(int port)`

- Reads a single byte from an I/O device
- `port` is the I/O address to read from
- Returns the byte read from the device

`void outb(int port, unsigned char data)`

- Writes a single byte to an I/O device
- `port` is the I/O address to write to
- `data` is the byte to write

Requirements

- Each module has a number of requirements, split between *functions* and *commands*

Requirements – Functions

- Must be implemented as distinct functions as described in the slides
- Need to have the specified name, return type, and parameters
- Typically part of the kernel, so should be placed in the `kernel` directory
- Must **not** directly produce any output (unless specified to do so)
- Allow a theoretical outside developer to build on your MPX code base

Requirements – Commands

- Implementation is flexible based on your interface
- It may make sense to combine some commands based on parameters or menu options
- Operate in user-space, so should be placed in the `user` directory
- Commands are how the user will interact with the system

I/O in MPX

- MPX does not support a standard keyboard or virtual terminal device
- Instead, all communication is done via serial port (COM1)
- In Module R1, you must implement the `serial_poll()` function in `kernel/serial.c`
    - This functions *polls* the serial port for data and stores it, one character at a time, into a user provided buffer until a new-line is encountered or the buffer is full
    - How this works is described in the following slides

Polling for Serial Input

- The Intel 8250 UART contains multiple I/O ports used to set options and get information – see `kernel/serial.c` for examples
- Data is read one character at a time from the device's base register (same address as the device) using the `inb()` macro
- When data is available, the least significant bit in the Line Status Register (`device + LSR`) is set
- To poll this, you may do something like:

```
while (/* buffer is not full*/) {
  if (/* LSR indicates data is available*/) {
    char c = inb(device);     /* read one byte */
    /* update the user buffer or otherwise handle the data */
  }
}
```

Requirement – `int serial_poll(device dev, char *buffer, size_t len)`

- `dev` is the device to read from
- `buffer` is the user-provided buffer
- `len` is the size of the user-provided buffer
- Returns the number of bytes read from the device, or a negative number on error

Polling Implementation requirements

- Each character must be either stored into the user-provided buffer OR undergo special processing for special keys
- The `serial_poll()` function must properly handle: alphanumerics (a-z, A-Z, and 0-9), space, backspace, delete, arrow keys (up, down, left, and right), carriage returns (`\r`) and new-lines (`\n`)
- The codes generated by your terminal program may not match the ASCII-defined values (especially common for delete and backspace) – **your code needs to handle the ASCII values**

Polling Implementation Notes

- Not all keys generate a single byte
- How "special processing" occurs depends on the user interface your group chooses (e.g. command line or menu based)
- Serial lines do not echo their input, so for a command line interface, you will need to call `serial_write()` or `outb()` for timely user feedback

The Command Handler

- This it the primary way users will interact with your operating system
- You only have serial access, so text-based is logical
- Examples
  - UNIX-style: `$ time 12:45:00`
  - Menu-driven:

```
Welcome to MPX. Please select an option.
1) Help
2) Set Time
3) Get Time
Enter choice: 2
```

- It does not have to be one of these (or even text based)
- MAKE IT USER FRIENDLY!!!

Basic Command Handler Structure

```c
void comhand(void)
{
  for (;;) {
    char buf[100] = { 0 };
    int nread = sys_req(READ, COM1, buf, sizeof(buf));
    /* the following simply echoes the input */
    sys_req(WRITE, COM1, buf, nread);
    /* process the command */
    if (/* command was shutdown and shutdown is confirmed*/) {
      return;
    }
  }
}
```

Command Handler – Questions to Ask

- How are arguments passed to functions implementing commands?
- How are optional arguments handled?
- How are extraneous arguments handle?
- How are incorrect types of arguments handled (e.g. user inputs a string when an integer is expected)?
- Be creative and consistent

Version Command

- Prints the current version of MPX and the compilation date
- Takes no parameters
- You must update this command for each module
- Examples:
  - Module based: R1, R2, R3, etc.
  - Major/minor: 1.0, 2.0, 3.0, etc.
  - Convergent: 3.14, 3.141, 3.1415, etc.

Help Command

- Provide usage instruction for all commands (similar to UNIX `man` pages)
- Takes no parameters
    - Optionally, allow a single (optional) parameter to provide help for only one command
- Since there is no filesystem (possibly in Module R6), strings will need to be hard-coded

Required Command – Shutdown

- Exit the command handler loop
    - Execution will return to `kmain()` and the system will halt
- MUST ask for confirmation

Required Commands – Get/Set Date, Get/Set Time

- Refer to the Intel document "Accessing the Real Time Clock Registers and the NMI Enable Bit: A Study in I/O Locations 0x70-0x77" on eCampus
- To read the clock:
  - Write an index to the RTC index register (0x70) with `outb()`
  - Read the result from the RTC data register (0x71) using `inb()`
- To write to the clock:
  - Disable interrupts (`cli()`)
  - Write an index to the RTC index register (0x70) with `outb()`
  - Write a new value for that index the RTC data register (0x71) using `outb()`
  - Enable interrupts (`sti()`)
- A potential trap (no pun intended): The RTC stores data in Binary Coded Decimal (BCD)

Documentation

- Documentation must be updated and submitted as part of every module
  - Must be placed in the `doc` directory at the top of your repository – links to online sources (Google Docs, Wikis, etc.) will **not** be accepted
- Three required documents:
  - User's Manual
  - Programmer's Manual
  - Contributions
- Must be well formatted and attractive
  - Must be PDF or HTML – plain text (including unprocessed Markdown) will **not** be accepted
- No templates will be provided for manuals

User’s Manual

- Explain every command available to the user
- Do not include information not necessary to use the system
- Must contain at least as much information as the Help Command, should contain more details as well as examples
- It should allow someone unfamiliar with your system and access to a freshly booted MPX to execute and understand all possible commands

Programmer's Manual

- Must cover every function and data type (`struct`, `enum`, etc.) you create
- Function documentation needs to describe what the function does, the meaning and valid values of parameters, and the range of return values
- Type documentation needs to describe the purpose of the type and, if applicable, the meaning of each member
- Use of Doxygen is allowed, or manually written
- It should **not** resemble comments copied and pasted from header files
- It should allow a developer with a fresh copy of your repository to be able to use your functions and types to extend the system without needing to read your source code

Contributions

- Easiest points you'll earn
- Describes each member's contribution to the project throughout the semester
    - Git history may be used to confirm this
- Document contributions at the function level

Contributions Example

Supercute	R1	R2	R3/R4	R5	R6

Hello Kitty	gettime()
	version()
	shutdown()
	serial_poll()				

My Melody	settime()
	itoa()
	Programmer's Manual				

Kuromi	getdate()
	sprintf()
	commhand()
	Users's Manual				

Badtz Maru	setdate()
	help()
	strcpy()
	Users's Manual				

Common Errors – Undeclared Function

```
clang -std=c18 --target=i386-elf -Wall -Wextra -Werror -ffreestanding -g -Iinclude
-c -o kernel/kmain.o kernel/kmain.c
kernel/kmain.c:125:2: error: call to undeclared function 'commhand'; ISO C99 and later do not support implicit function declarations [-Werror,-Wimplicit-function-declaration]
        commhand();
        ^

1 error generated.
make: *** [<builtin>: kernel/kmain.o] Error 1
```

- Problem: You are calling a function without an available prototype
- Solution: Add the function prototype to an appropriate header and `#include` that header

Common Errors – Undefined Reference

```
i686-linux-gnu-ld -melf_i386 -znoexecstack -T kernel/link.ld -o kernel.bin kernel/boot.o kernel/irq.o kernel/gdt.o kernel/interrupts.o kernel/kmain.o kernel/panic.o kernel/serial.o kernel/vm.o lib/ctype.o lib/stdlib.o lib/string.o user/system.o
i686-linux-gnu-ld: i686-linux-gnu-ld: DWARF error: invalid or unhandled FORM value: 0x25
kernel/kmain.o: in function 'kmain':
kmain.c:(.text+0x10a): undefined reference to 'commhand'
make: *** [Makefile:48: kernel.bin] Error 1
```

- Problem: You are calling a function that isn't being linked into your MPX
- Solution: Add the *object* (`.o`) file that corresponds to that function's source (`.c`) file to the `Makefile`

Final Notes

- Start early and *ask questions* if you have *any* difficulty
- Use `sys_alloc_mem()` and `sys_free_mem()` for dynamic memory
    - Any other form of heap management will result in failure
    - Minimize dynamic memory until implementing `free()` in Module R5
- Remember there is no `printf()` (unless you write it)
- Use `sys_req()` for input and output
    - Direct calls to other I/O related functions from your module code will result in failure
- Use GDB for debugging – see tutorial on eCampus