CS 450 Module R6
Interrupt Driven I/O

West Virginia University

Spring 2023

Goals

Goals

- Implement a basic interrupt-drive serial port driver
  - Enable more efficient communication between your system and attached serial devices (e.g. terminal, another computer)
  - Handle external interrupt events
- Integrate this with all previous modules into a complete MPX

Background

Continuous Dispatch

- R4 **must** be working for R6 to work
- All programs must be treated as processes
- Command Handler must be running as a process
- The Idle process must be running
  - This ensures there is *always* at least one process in the ready, non-blocked state

Device Control

- You'll be implementing low level control of I/O devices
- Device Type:
  - Character – Communication is one byte at a time, sequentially
  - Block – Communication is via blocks of bytes, may be randomly accessible
- Data Transfer:
  - Serial – Bits a transferred one by one
  - Parallel – Multiple bits transferred simultaneously
- Synchronization:
  - Synchronous – Device is assumed ready after a fixed time
  - Asynchronous – Device indicates readiness by some signal

Device Registers

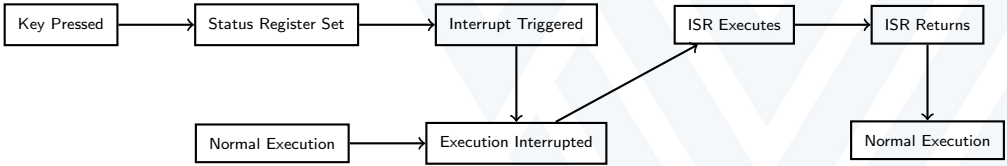
- Data Registers – Used to transfer data to/from devices
  - Example: Receive Buffer/Transmitter Holding to read or write one byte
- Status Registers – Indicate the condition of devices
  - Example: Line Status Register to test whether data is available
- Control Registers – Manipulate configuration of devices
  - Example: Line Control Register to set data speed/baud rate

Programmed I/O

- The x86 architecture uses special instructions for I/O
  - Devices are assigned I/O ports from 0 to 65,535
  - Read data with `in` instructions
  - Write data with `out` instructions
  - Only EAX, AX, and AL are used for transfer
- MPX provides macros in `<mpx/io.h>`
  - `char inb(uint16_t port)` – Reads one byte from `port`
  - `void outb(uint16_t port, char data)` – Writes `data` to `port`
- Seen in R1 implementation of `serial_poll()`

Interrupts

- The preferred method for managing I/O devices
- Device generates an interrupt on a change in status



Handling Interrupts

- Interrupts can happen at any time
- To avoid critical errors, interrupts can be strategically disabled
  - Individual devices can be directed to stop generating interrupts via device-specific outputs
  - Some interrupts can be individually disabled by calls to the Programmable Interrupt Controller (PIC)
  - All interrupts can be disabled via `cli()`
- Interrupt Service Routines (ISRs) can **not** call other routines that may generate interrupts

`void idt_install(int vector, void (*isr)(void*));`

- Declared in `<mpx/interrupts.h>`
- Installs an ISR into the Interrupt Descriptor Table
- `vector` is the interrupt vector that will cause the ISR to execute
- `isr` is the address of the ISR

Safely Handling Interrupts

- Interrupt Service Routines CAN **NOT** call other routines that may generate interrupts
- Debugging-by-`printf()` will **NOT** work in an I/O interrupt handler
  - The output itself is likely to generate an interrupt
  - That interrupt causes more output, which causes another interrupt, which causes more output, ...
- To avoid an ISR interrupting itself, it should disable interrupts while performing its work

Useful Structures – Device Control Block (DCB)

- Maintains information about properties and status of OS resources
- Allocation status (is the resource in use?)
- Current operation (read, write, etc.)
- Event Flag (is there an event that needs to be handled?)
- Internal ring buffer and descriptors (size, beginning and end of valid data)

Useful Structures – I/O Control Block (IOCB)

- Record information about a specific transfer
- May be wise to have a separate queue of IOCBs for each device
- Associated with a process (PCB)
- Associated with a device (DCB)
- Type of operation (read, write)
- Buffer
- Buffer size
- Also known as an I/O Descriptor

Managing DCBs and IOCBs

- Each device should have its own DCB
- Each DCB should have a queue of IOCBs
  - Additional I/O requests may be submitted before the current one completes
  - Each queue entry describes a single I/O request
- Requests for I/O are passed to an I/O scheduler
  - If device is busy, enqueue the request in FIFO order
  - If device is available, immediately begin processing the request

Handling I/O requests

- Process is transitioned to the BLOCKED state
  - When a process becomes blocked, you should switch context to the next ready process just as though an explicit `IDLE` request were made
- Each byte transfer will generate a hardware interrupt
  - This means every byte *read from* **or** *written to* a serial device
- Your device driver ISR checks whether more data needs to be transferred
  - If yes, process the next byte
  - If no, set the DCB Event Flag to ready; if there are pending requests, begin the next

Programmable Interrupt Controller (PIC)

- The PIC is necessary because the 8086 had only one signal wire
- When an interrupt is generated, allows the CPU to determine which
- PIC command register is at I/O port 0x20
  - Your device driver ISR will need to write the End of Interrupt command (0x20) to this register when it completes
- PIC interrupt mask I/O port 0x21, with one bit per hardware interrupt
  - Bits set to 0 indicate interrupts that are enabled
  - Bits set to 1 indicate interrupts that are disabled (masked)

```
1    cli();
2    int mask = inb(0x21);
3    mask |= (1<<7);    // Mask (disable) hardware IRQ 8
4    outb(0x21, mask);
5    sti();
```

Serial Port Registers

- Physically implemented in a Universal Asynchronous Receiver-Transmitter (UART)
- Registers change depending on whether they are being read or written and whether DLAB (bit 7 of the LCR) is set

I/O port	Read (DLAB=0)	Write (DLAB=0)	Read (DLAB=1)	Write (DLAB=1)
base	RBR receiver buffer	THR transmitter holding	DLL divisor latch LSB	DLL divisor latch LSB
base+1	IER interrupt enable	IER interrupt enable	DLM divisor latch MSB	DLM divisor latch MSB
base+2	IIR interrupt identification	FCR FIFO control	IIR interrupt identification	FCR FIFO control
base+3	LCR line control	LCR line control	LCR line control	LCR line control
base+4	MCR modem control	MCR modem control	MCR modem control	MCR modem control
base+5	LSR line status	factory test	LSR line status	factory test
base+6	MSR modem status	not used	MSR modem status	not used
base+7	SCR scratch	SCR scratch	SCR scratch	SCR scratch

- Good reference:
  `https://www.lammertbies.nl/comm/info/serial-uart`

Serial Port Addresses and IRQs

Device	Base I/O Address	Hardware IRQ	Interrupt Vector
COM1	0x3F8	4	0x24
COM2	0x2F8	3	0x23
COM3	0x3E8	4	0x24
COM4	0x2E8	3	0x23

Kernel Functions

Updates to `sys_call()` 1/2

- Update your `sys_call()` to handle `EAX` being `READ` or `WRITE`
- The device will be in `EBX`, buffer in `ECX`, and buffer size in `EDX`
- Check to see whether the requested device is currently in use
- If not, you can directly call the appropriate driver function (i.e. `serial_read()` or `serial_write()`)
- Otherwise, the request must be scheduled by your I/O scheduler (see below)

Updates to `sys_call()` 2/2

- When scheduling I/O operations, process must be moved to BLOCKED state
- Dispatch a new process as though the requested operation was `IDLE`
- Each byte transferred will generate an interrupt
  - Device driver ISR will be invoked
  - If the operation is complete, set the Event Flag in the appropriate DCB
- Each time `sys_call()` is called, check for any available Event Flags
  - For any that are set, perform the required completion sequence

I/O Scheduler

- Processes input and output requests
- Examine and validate system call parameters
  - Operation must be `READ` or `WRITE`
  - Device must have an associated DCB
  - Buffer must not be `NULL`
  - Size must be non-zero
- Check status of requested device
  - If available, begin processing request immediately
  - Otherwise, add an IOCB to the appropriate queue
- Return to `sys_call()`, which dispatches the next process

`int serial_open(device dev, int speed);`

- Initialize the serial port associated with the `dev` parameter
- Initialize the associated DCB
- Install the appropriate ISR
- Set the port speed to the `speed` parameter
- Set other line characteristics (8 data bits, no parity, 1 stop bit)
- Enable all necessary interrupts
- Return an integer indicating success or failure

```
int serial_close(device dev);
```

- End a serial port session
- Clear open indicator of associated DCB
- Disable associated interrupt in PIC mask
- Return an integer indicating success or failure

`int serial_read(device dev, char *buf, size_t len);`

- Begin reading input from the serial device `dev`
- Reads data into the buffer at `buf`
- Reads no more than `len` bytes
- If there are characters in the ring buffer, read until `buf` is full or no more characters in the ring buffer
- If `buf` is full, return
- Otherwise, change device states to reading and notify ISR to place characters in the the associated buffer instead of the ring buffer

```
int serial_write(device dev, char *buf, size_t
len);
```

- Begins writing data to the serial port `dev`
- Write the first character from `buf` to the output register
- Enable write interrupts
- Let the ISR do the rest

Assembly Routine `serial_isr`

- As in R3, it's easiest to do most of the work in C
- The assembly can be as simple as calling your C function and then returning with `iret`

```
void serial_interrupt(void);
```

- Disable interrupts
- Obtain the correct DCB
- Read the Interrupt ID Register to determine the exact type of interrupt
- Based on the type (input, output), pass the handling to a second function
- Issue EOI to the PIC
- Reenable interrupts

```
void serial_input_interrupt(struct dcb *dcb);
```

- Read the character from the device
- If the DCB state is reading, store the character in the appropriate IOCB buffer
  - If the buffer is now full, or input was new-line, signal completion
- Otherwise, attempt to store the character in the ring buffer

`void serial_output_interrupt(struct dcb *dcb);`

- If the DCB state is writing, check for additional characters in the appropriate IOCB buffer
- If there is additional data, write the next character to the device
- If you have just written the last character from the IOCB, disable write interrupts and signal completion

User Commands

Remove Command: Block PCB

- Remove the Block PCB command completely

Remove Command: Unblock PCB

- Remove the Unblock PCB command completely

Final Notes

Final Notes

- Remember to update your documentation and Version and Help commands
- Debugging will be tricky due to interrupts – GDB will be difficult, but still **much** more reliable than debug-by-`printf()`
- No changes should be necessary to previous modules – keep any changes you do need to an absolute minimum*

Final Final Notes

- This is the most complex/tightly coupled module, so start early
- Start early
- Start *early*
- Start **early**
- No, really, **START EARLY** - start a new branch *today* so you can begin initial work in parallel to finishing R5