Input/Output

Week 11 – Lab

I/O Devices

- **Block devices:** store information in fixed size blocks
  - Commands include open(), read(), write(), seek()
- **Character devices:** deliver or accept stream of characters
  - Commands include get(), put()

I/O Devices

- In Unix, all I/O devices are represented as files
  - **$ /dev**

    contains special device files for all devices
  - **$ /proc**

    virtual file system

Major and Minor numbers

- Major number identifies driver associated with the device
- Minor number is used by the kernel to determine which device is being referred to

  - **$ cd /dev**
  - **$ ls -l**

Accessing I/O ports

- *ioperm()* sets the port access permission bits for the calling thread for *num* bits starting from port address *from*. If *turn_on* is nonzero, then permission for the specified bits is enabled; otherwise it is disabled

```
int ioperm(unsigned long from,
          unsigned long num, int turn_on);
```

- *iopl()* changes the I/O privilege *level* of the calling process, as specified by the two least significant bits in level. In addition to granting unrestricted I/O port access, running at a higher I/O privilege level also allows the process to disable interrupts. This will probably crash the system, and is not recommended

```
int iopl(int level);
```

Accessing I/O ports

```
 inw(x)
outw(value, x)
```

- This family of functions is used to do low-level port input and output. The *out** functions do port output, the *in** functions do port input; the b-suffix functions are byte-width and the w-suffix functions word-width; the _p-suffix functions pause until the I/O completes
- They are primarily designed for internal kernel use, but can be used from user space

Example

```
/* Get access to the ports */
if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

/* Set the data signals (D0-7) of the port to all low (0) */
outb(0, BASEPORT);

/* Sleep for a while (100 ms) */
usleep(100000);

/* Read from the status port (BASE+1) and show the result */
printf("status: %d\n", inb(BASEPORT + 1));

/* We don't need the ports anymore */
if (ioperm(BASEPORT, 3, 0)) { perror("ioperm"); exit(1);}

exit(0);
```

http://www.tldp.org/HOWTO/IO-Port-Programming-9.html

Memory Mapped I/O

- The *mmap()* system call creates a new memory mapping in the calling process' virtual address space.
- Once a file is mapped, its contents can be accessed by operations on the bytes in the corresponding memory region

```
void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);
  int munmap(void *addr, size_t length);
```

Exercise 1

- Create a file *ex1.txt* with a random string in it. Write a C program (*ex1.c*) that changes the string in *ex1.txt* to “This is a nice day” by using mmap()
- Hints:
  - Open the file in O_RDWR mode
  - Use *stat()* or *fstat()* to get the size of the file

Buffered I/O

- **Full buffering:**
  - Invokes read() or write() system call when the buffer becomes full
- **Line buffering:**
  - On output, data is written when a newline character is inserted into the stream or when the buffer is full, what so ever happens first. On Input, the buffer is filled till the next newline character when an input operation is requested and buffer is empty
- **No buffering:**
  - Directly translate every library call into a read() or write() system call

Buffered I/O

```
int setvbuf(FILE *stream, char *buf,
            int mode, size_t size);
```

- The setvbuf() function may be used on any open stream to change its buffer. The mode argument must be one of the following three macros:
  - **_IONBF** unbuffered
  - **_IOLBF** line buffered
  - **_IOFBF** fully buffered

Buffered I/O

- What is the default buffering type of standard streams (stdout, stdin, stderr)?

Exercise 2

- Write a C program (*ex2.c*) using line buffer. Write your code according to the instructions:
- Each of the 5 characters of “Hello” string should be put in separate printf(). Add a 1 sec sleep after every printf()
- Output should be:
  A 5 sec wait and then “Hello” printed instantaneously

Exercise 3

- Run the following two examples of code and explain the difference in output. Save your answer in *ex3.txt* file

Program 1	Program 2

`#include <stdio.h>`
`#include <unistd.h>`
 	
`int main(void) {`
` printf("Hello");`
` fork();`
` printf("\n");`
` return 0;`
`}`
	`#include <stdio.h>`
	`#include <unistd.h>`
	
	`int main(void) {`
	` printf("Hello\n");`
	` fork();`
	` printf("\n");`
	` return 0;`
	`}`

memcpy()

```
void *memcpy(void *dest,
             const void *src,
             size_t n)
```

- The *memcpy()* function copies *n* bytes from memory area *src* to memory area *dest*
- Parameters:
  *void *dest*: a pointer to the destination array where the content is to be copied.
  *void *src*: a pointer to the source of data to be copied
  n: number of bytes to copy

Exercise 4

- Write a C program (*ex4.c*) to copy the content of *ex1.txt* to *ex1.memcpy.txt* using memory mappings.

For more knowledge

DMA

- DMA-TO-DEVICE: DMA from the main memory to the device
- DMA-FROM-DEVICE: DMA from the device to the main memory
- DMA-BIDIRECTIONAL: DMA from the device to main memory or from the main memory to device

DMA Example (1)

- The actual form of DMA operations on the PCI bus is very dependent on the device being driven. Thus, this example does not apply to any real device; instead, it is part of a hypothetical driver called dad (DMA Acquisition Device). A driver for this device might define a transfer function like this http://gauss.ececs.uc.edu/Courses/c4029/code/dma/dad.c

DMA Example (2)

```
int dad_transfer(struct dad_dev *dev, int write,
                 void *buffer, size_t count) {
 dma_addr_t bus_addr;
 /* Map the buffer for DMA */
 dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
 dev->dma_size = count;
 bus_addr = dma_map_single(&dev->pci_dev->dev, buffer,
                       count, dev->dma_dir);
 dev->dma_addr = bus_addr;

 /* Set up the device */
 writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
 writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
 writel(dev->registers.addr, cpu_to_le32(bus_addr));
 writel(dev->registers.len, cpu_to_le32(count));

 /* Start the operation */
 writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
 return 0;
}
```

DMA Example (3)

- The above function maps the buffer to be transferred and starts the device operation. The other half of the job must be done in the interrupt service routine, which looks something like this:

- pt_regs = intel registers

DMA Example (4)

```
void dad_interrupt(int irq, void *dev_id,
            struct pt_regs *regs) {
 struct dad_dev *dev = (struct dad_dev *) dev_id;

 /* Make sure it's really our device interrupting */
 /* Unmap the DMA buffer */
 /* After this call, reads by the CPU to the buffer are
  guaranteed to see whatever the device wrote there*/

 dma_unmap_single(dev->pci_dev->dev,
     dev->dma_addr,
     dev->dma_size,
     dev->dma_dir);
 /* Only now is it safe to access the buffer,
    copy to user, etc. */
 /*...*/
}
```