1 Lab 14

Date: Nov 21, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to introduce to you to different methods used for doing input/output (I/O) on computer systems. After completing this lab, you should be familiar with the following topics:

- The problems with *blocking I/O*.
- Using *polling* to check whether an I/O device is ready.
- The use of *interrupts* so that programs can be interrupted only when an I/O device is ready.

You will also gain some experience using a primitive operating system where a program error may crash the entire system and require a reboot.

1.2 Background

Assume that a program running on a computer needs to read input from the user via the keyboard. An obvious strategy is to have the program wait until the user has typed a key. However, a back-of-the-envelope analysis easily shows that such a strategy would be a waste of resources:

Assume that the user is a fast typist capable of typing 4 characters per second and assume that a modern CPU can execute at 1000 MIPS ("Million Instructions Per Second"). So in the time spent waiting for the user to type one character, the CPU could have executed 250 million instructions. Such a situation is obviously a tremendous waste of resources.

There are two main solutions to this problem:

Polling Instead of waiting for the user to type a key, the computer continues execution (executing the current program or another program). But it periodically checks or **polls** the keyboard to see whether the user has typed a key. If that is the case, then it read the key and proceeds as per the original program.

Interrupts The computer continues executing the current program or another program. When the user has typed a key, the keyboard uses special hardware to **interrupt** the program. The interrupt results in execution of an *interrupt-service routine* which will read the keyboard. Things are setup so that the original program receives the value of the key so that it can continue execution.

1.3 Needing Dosbox

A problem with experimenting with I/O on current computer systems is that all I/O in modern computer systems is under the control of the operating system and not directly accessible to a non-kernel programmer. It may be possible to reboot a present day computer with a simple kernel which allows access to the I/O but such a kernel would probably also have an extremely primitive development environment. A happy medium is to use a virtual machine emulator running within a modern development environment like Linux.

There are many such emulators available for Linux: two which come to mind are `virtualbox` and `dosbox`. For this lab, `dosbox` was chosen because it already comes with a suitable primitive operating system (an emulation of MS-DOS) while making it easy to continue using the Linux development environment.

To ensure that hardware from different PC manufacturers could be accessed in a uniform manner, early PC's came with a *Basic Input/Output System* or BIOS. The MSDOS operating accessed I/O devices using the hardware abstractions provided by this BIOS layer. In modern OS's, the BIOS is only used during initialization.

1.4 Starting up

Use the startup directions from the earlier labs to create a `work/lab14` directory. Make sure that your `lab14` directory contains a copy of the files directory. Unlike other labs, instead of making a `script` of your Linux session, show the TA the changes you make in the provided files.

Change directory into your `files` directory and start `dosbox` in the background:

```
$ cd files
$ dosbox &
```

You should see the `dosbox` program start up in a separate window. Within a couple of seconds, you will see a msdos prompt `Z:\>`.

[In the rest of this lab, commands typed to a Linux shell are shown prefixed with a `$` prompt, whereas commands typed in to `dosbox` are prefixed with either `C:\>` or `Z:\>`.]

One advantage of `dosbox` is that it is very easy to share files with the Linux host. You can simply mount and access your current Linux directory as *drive c* using the following commands:

```
Z:\>mount c .
Z:\>c:
C:\>dir
```

The first command mounts the current Linux directory as the `c:` drive; the second command sets `c:` as the current drive and the last `dir` command produces a directory listing of the current directory.

Note that any edits that you make in your Linux files are reflected immediately immediately on `dosbox` but new files do not show up. To access new files simply unmount `c` using `mount -u` and remount:

```
C:\>mount -u c
Z:\>mount c .
Z:\>c:
```

During the course of this lab, it is very likely that you will crash `dosbox` (it will either become unresponsive or the window will close itself). In that case, simply restart it and remount your Linux directory.

In this lab we will be playing with a simple program which does the equivalent of the following pseudo-code (with minor variations in the different exercises):

```
COUNT = 100;
while (true) {
  char c = getchar();
  if (c == 'q') break;
  for (int i = 0; i < COUNT; i++) {
    putchar(c);
    c = get-char-if-char-available();
  }
}
exit(0);
```

The different exercises will explore ways of implementing `get-char-if-char-available()`.

1.5 Fully Blocking I/O

In this exercise, we do not make any attempt to implement `get-char-if-¬ char-available()`. Consequently, we read a character, output it `COUNT` times and then repeat until a `'q'` input character terminates the program.

The program is implemented in assembly language in <./files/block.s>. Take a look at this file (<./files/block.s>). The following points are worth noting:

- The `.code16` directive sets things up to use the 16-bit subset of the `x86` instruction set similar to the original 8086 instruction set. Specifically, all registers are 16-bits. So both the the 64-bit `rxx` registers and the 32-bit `exx` registers are unavailable.
- With a 16-bit architecture, addresses are retricted to 16-bits which would result in a address space of 64-KB. However, the architecture uses segmented addressing to allow addressing of upto 1 MB of memory. Hence a full memory address consists of a *segment*`:`*offset* pair. Fortunately, we use a program executable format which is restricted to using a single 64 KB segment; hence we can ignore segmentation since all segment registers point to the start of the single segment (note this is similar to why we can ignore segment registers in x86-32 and x86-64 except that there they are pointing to a large multi-GB segment!).
- The program is written in a style typical of manual assembly language with ad-hoc use of registers. The program makes heavy use of global variables like `inChar` (which holds the character read by `getchar()` and written by `putchar()` for reasons which will become clearer in the following exercises).
- The program accesses DOS services using `int 0x21` instructions.
- The program uses a software delay loop (the program simply delays a certain amount by spinning in a loop executing useless instructions).

You should be able to understand the program reasonably well.

Now build the program under Linux:

```
$ make block.com
```

This runs the GNU assembler to assemble `block.s` followed by a couple of other programs to massage the executable into a `.com` format. Additionally, the assembler also creates an assembler listing in `block.lst`.

The `.com` format is an extremely simple format which merely contains the binary instructions and data which will be loaded into memory without any headers, etc.

You can look at the bytes in `block.com`:

```
$ od -x block.com
```

Compare the output with assembled code in `block.lst`. You should see that it is identical.

Now run the program in `dosbox`:

```
C:\>block
```

When you type a character, it will be repeatedly echoed a fixed number of times; this continues until you type a `'q'` character to quit.

Notice that if you type a character while the last character is being repeated in the inner loop, the new character you typed is ignored until the start of the next iteration of the outer loop.

To make sure you understand the program make the following minor modifications:

1. Reduce the count of characters echoed on each iteration of the inner loop by half.
2. Reduce the delay between consecutive repeated characters by approximately half.

Reassemble and test as above.

1.6 Polled I/O

Look at program poll.s. Notice that within the inner loop it calls a function `checkKey()` which sets global variable `hasKey` and then if `hasKey` is set, the inner loops refreshes the current character using `getchar()`. This corresponds to the following pseudo-code:

```
COUNT = 100;
char inChar;
while (true) {
  char inChar = getchar();
  if (inChar == 'q') break;
  for (int i = 0; i < COUNT; i++) {
    bool hasKey = checkKey();
    if (hasKey) {
      inChar = getchar();
      hasKey = false;
    }
    putchar(inChar);
  }
}
exit(0);
```

So on each iteration of the inner loop, the program is polling the keyboard using `checkKey()` and reading the keyboard only if `hasKey` is true.

Assemble and run this program as before:

```
$ make poll.com
C:\>poll
```

Notice that it is much more responsive to the keyboard. Specifically, if you type a key while it is in the inner loop the output of the inner loop reflects the newly entered key immediately.

There is one deficiency in the program: the `'q'` key works as a quit key only outside the inner loop; in fact, if you type a `q` into the inner loop it merely echoes it and does not quit the program.

Modify the program to fix this deficiency. The simple fix will simply be a copy-and-paste job violating DRY. A better fix would refactor to avoid violating DRY.

1.7 Interrupt Driven I/O

The program for this exercise in key-int.s has pseudo-code very similar to that of the last exercise:

```
COUNT = 100;
char inChar;
bool hasKey = false;
while (true) {
  char inChar = getchar();
  if (inChar == 'q') break;
  for (int i = 0; i < COUNT; i++) {
    if (hasKey) {
      inChar = getchar();
        hasKey = false;
    }
    putchar(inChar);
  }
}
exit(0);
```

Note that the previous exercise assigned the results of `checkKey()` to `hasKey` but in the above code there is no assignment of `true` to `hasKey` (except for the initializer). So how does `hasKey` ever get assigned `true`.

The answer is that it is done **asynchronously** in an *interrupt service routine* which is run asynchronously when a key is input. Look at the code in key-int.s:

- At the start of the `main` program, there is a call to `setupHandler()`. What that does is use DOS services to wrap the keyboard interrupt handler. Specifically, it saves the current addresses of the handler for `KEY_INT` (the keyboard interrupt) and `CHK_INT` (a BIOS interrupt) in global variables `intAddr` and `chkAddr` respectively. Then it replaces the address of the keyboard interrupt handler with the address of the wrapping routine intHandler.

- After the above replacement, whenever a keyboard interrupt occurs our routine intHandler() will be called. It's code is somewhat unusual as it is called as an interrupt handler which will always be called with the flags and return address on the stack and must return using a special iret instruction which takes care of popping of the additional flags word from the stack (in addition to taking care of the normal return).
- The code for intHandler() wraps the original interrupt handler by calling it via the saved address. Note that the call simulates an interrupt by pushing the flags on the stack before the call; since the wrapped routine is an interrupt handler it will pop those flags off the stack using the iret instruction.

  It then uses the BIOS routine to check if a key has really been pressed and if one has, then it sets the hasKey variable. It does this with interrupts disabled (instruction cli) and reenables when done (instruction sti).
- Before the program exits, it restores the original interrupt handler using resetHandler().

[Note: this program is rather brittle and it took a lot of effort to make it work. Even now, the program runs successfully the first time you run it, but if you try to run it a second time it hangs dosbox. It can be run again only after restarting dosbox. So obviously it is not doing a good job of cleaning up after itself.]

Assemble and run the program in the usual way:

```
$ make key-int.com
C:\>key-int
```

Unlike the above pseudo-code and the assembly code for poll.s, the code to reset the hasChar flag has been moved to the end of the getchar() routine. Why would it not work if the reset was within the inner loop of the main program as in <./files/poll.s> poll.s?

1.8 References

- Lukan, Dejan, *Logging Keystrokes with MSDOS*, *Part 1* and *Part 2*.
- *DOS INT 21h - DOS Function Codes*.
- Dosbox