ECE 469 Spring 2017 Laboratory #1

Due Thursday, January 26, 23:59 EST

IMPORTANT: You should work *alone* on this lab. Do not work in teams!

The purpose of this lab is to get you started with DLXOS, the mini-OS used for all course laboratory assignments, and to teach you how system calls are implemented - the understanding of which is essential for completing all future assignments. Specifically, we will cover the following three topics:

- Setting up DLXOS
- Context switching, stack frames, and implementing a trap
- Implementing a new system call, `Getpid()`

Laboratory assignments are structured as much as possible such that you should be able to finish the assignment even if some of the pieces that you have built do not work.

DLXOS

The projects for this course require you to build and experiment with a real operating system. We will be using a very simple, but functional, operating system as the base to which you will make modifications and add features. It is named DLXOS and was written at the University of Maryland Baltimore County and the University of California, Santa Cruz. It is based on the DLX instruction set architecture described by Hennessy & Patterson. Over the course of the semester, your job will be to improve the functionality and performance of DLXOS.

The home page for DLXOS is located here. Note that the DLX Simulator contains changes to the basic DLX architecture that are related to operating system support. For more information on the DLX instruction set architecture, please refer to the following: DLX Instruction Set.

Getting Started

The introduction below will refer to various source files and executables associated with DLXOS. You should download these files according to the following instructions, and refer to them throughout the rest of this introduction. Note that the "$" is a Linux command prompt. You should run all these commands from your home directory on an ECN Linux machine. IMPORTANT: shay.ecn.purdue.edu is NOT and ECN Linux machine. It is a SunOS machine, and therefore none of the DLX tools will run on shay. To determine the operating system of a machine, simply run the `uname` program.

You can use the Linux machines in EE 206 and EE 207. To remotely login to a Linux machine please ssh into ecegrid.ecn.purdue.edu (ssh [your-account]@ecegrid.ecn.purdue.edu) or type ecegrid.ecn.purdue.edu in your browser and use the machine remotely through the browser.

- Create a directory for all your EE469 labs:

```
$ mkdir ~/ee469
$ cd ~/ee469
```

- Copy the lab 1 code to your ee469 directory:

```
$ cp ~ee469/labs_2017/Labs/lab1.tar.gz .
```

- Untar the tarball into a lab1 directory:

```
$ tar xzvf lab1.tar.gz
```

- Explore the directory structure:

```
$ ls -R
```

The directory structure will look like this:

```
lab1/
     apps/
          user programs go here
     bin/
         put any executable DLX files here
     include/
             header files for user programs are here
             os/
                header files for the operating system are here
     lib/
         precompiled object files for user programs (i.e. usertraps.o) go here
     os/
        source code for the DLXOS operating system is here
```

DLXOS Introduction

There are several key components involved in the DLXOS system, and understanding the differences between these components is important to understanding the lab assignments. These components are:

1. **DLX Hardware Simulator**:
   Any operating system must have hardware on which to run. Testing a newly written operating system on a real processor is extremely difficult. This is because you only have oscilloscopes and LED's as debugging tools on real hardware: no printf, gdb, etc. In fact, there are no guarantees that the processor itself is correct. To help alleviate these problems, operating system developers typically test their code on simulated hardware called virtual machines. A virtual machine is simply a program that responds to all instructions from an operating system in the same way that a real processor would.

   The DLX hardware simulator can be run from the command-line in Linux like this: (note that the $ is the Linux command prompt)

   simulator running

```
     $ dlxsim -x <compiled operating system file> -a <arguments to operating system>
```

   It is important to realize that while you run DLXOS on top of the DLX simulator, which runs as a user program in Linux, all of the code that you write will run exactly the same as if DLXOS were running on bare hardware. That is indeed the point of using a simulator. The simulator runs in userland for convenience - multiple students can run their version of DLXOS at the same time on the same physical machine. This same approach is often used in industry - it's generally a good idea to test out system code in a simulated environment before running it on potentially flaky hardware. In real life, you are not allowed to throw out a running machine and ask for a CPU with different features to make your code easier to write. Anyone that has dealt with the x86 architecture is painfully aware of this fact.

2. **DLX Operating System**:

Since it is beyond the scope of this course to write an operating system completely from scratch, we are providing you with a set of basic skeleton source files that can be compiled into a DLX-based operating system. This operating system will be referred to as DLXOS. You will be editing these source files throughout this course as you implement new features. The initial set of source files are as follows:

C Source Files:

- memory.c: contains functions dealing with paging and other memory managment tasks.
- process.c: contains functions dealing with process maintenance and switching.
- misc.c: contains miscellaneous helpful functions such as string manipulation functions.
- queue.c: contains functions related to queue management.
- synch.c: contains functions related to synchronization primitives such as semaphores.
- sysproc.c: contains functions used primarily for tesing the other operating system functions.
- traps.c: contains functions that respond to traps.
- filesys.c: contains functions for reading and writing to a filesystem.

Assembly Source Files:

- dlxos.s: contains low-level operating system code such as the bootloader.
- osend.s: a bookkeeping file listing the last address of the operating system.
- trap_random.s: contains the trap subroutines to generate random numbers.
- usertraps.s: contains the trap subroutines that are available to user programs.

3. **GNU Compiler Suite and Binutils for DLX Instruction Set**:

Since it is simplest to write an operating system in a high-level language like C, it is necessary to have a compiler that can create executable code in the given instruction set. In order to compile either assembly or C code into executable DLX bytecode, the standard GNU compiler suite and binutils (gcc, as, ld, ar, ranlib, etc.) were ported to compile for DLX. **This compiler is used to compile both the operating system and any user programs that will be run by the operating system.** The primary difference in compiling the operating system vs. compiling user programs is that the operating system bytecode must have bootloader code as the first instruction. This is handled by explicitly calling the assembler with the "-i" flag.

Any C source file can be compiled using the DLX GCC compiler in the standard way that Linux programs are compiled. This means that you should first compile each source file individually into an object file, and then link all the object files together into an executable. The only noticeable difference with the DLX version of GCC is that, for reasons known only to the people who wrote DLXOS, the object and executable files from the gcc-dlx compiler are text assembly files rather than bytecode. You must explicitly call the assembler to create a file capable of running on the DLX hardware simulator. This caveat can make debugging your programs simpler if you have a knack for reading assembly code.

The following command will compile the source file foo.c into an object file foo.o:

```
$ gcc-dlx -mtraps -O3 -Wall -c -o foo.o foo.c
```

compile source file to object

enable traps

Note that the "-mtraps" flag is a machine-specific flag for GCC that enables traps in DLX. Also, the "-O3" option is a standard option which defines the type of optimization that the compiler should attempt (in this case we are telling it to optimize both for program size and execution speed). The "-c" flag instructs the C compiler to stop before linking, causing it to output object code.

Also note that since all header files are in the ../include/ and ../include/os directories, rather than located in the same directory as the source files, you will need to tell GCC where to look for header files, otherwise nothing will compile. This is done with the -I<path> flag like this:

```
$ gcc-dlx -mtraps -O3 -Wall -I../include -I../include/os -c -o foo.o foo.c
```

Note that you should NOT have -I../include/os when compiling user programs, as they should not have access to any operating system functions.

Similarly, the command to compile an assembly source file (bar.s) into object code is:

```
$ gcc-dlx -mtraps -O3 -Wall -c -o bar.o bar.s
```

Once all source files have been compiled into object files, they can be linked into a DLX executable program with this command:

```
$ gcc-dlx -mtraps -O3 -Wall -o <name_of_executable.dlx> <list of object files>
```

Once you have compiled the "executable" file with gcc-dlx, you then need to pass it through the assembler to create loadable bytecode. Note that this version of the assembler command is only used to compile the operating system: the command for user programs will be given in the next section.

```
$ dlxasm -i _osinit -l <name_of_output_list_file_to_create.lst>
<name_of_os_executable.dlx>
```

This command creates two files: a list file (a more detailed sort of assembly file), and the executable DLX bytecode, which will be the same name as your executable file, but with the ".obj" extension added. The "-i _osinit" tells the assembler that the _osinit function (found in dlxos.s) should be the entry point of the operating system (as opposed to the main() function).

4. **User programs**:

The purpose of an operating system is to enable developers to write user-space programs that can run on shared hardware. You can write C programs for DLXOS in much the same way you write C programs for any other operating system. Here is an example C program for reference:

```
#include "usertraps.h"
void main (int x) {
  Printf("Hello World!\n");
}
```

Note some differences:

- There is no stdio.h to go with libc. Instead, there is a header file listing function declarations for user-space traps within the operating system.
- The main function returns a void and takes one integer argument.
- All user-space trap function names (like Printf) start with capital letters by convention. This convention was chosen in order to easily distinguish them from os-space traps.

The procedure for compiling your user programs is almost identical to the procedure for compiling the operating system, which is why it can sometimes be very difficult to understand what code should be in the operating system and what code should be in a user program.

To compile a user program, you first compile any source files into object files. The following command will compile the source file userprog.c into the object file userprog.o from the apps/ directory:

```
$ gcc-dlx -mtraps -O3 -Wall -I../include -c -o userprog.o userprog.c
```

The meanings of all the compiler flags are listed in the previous section. Note that we are NOT including the os header files directory (../include/os) in the search path for header files because user programs should not have access to operating system internal functions.

Next, link all the object files into one file:

```
$ gcc-dlx -mtraps -O3 -Wall -o userprog.dlx <list of object files>
```

Note that the list of object files should include "usertraps.o" generated while compiling the operating system, in addition to "userprog.o".

Finally, assemble the single file into executable DLX bytecode, and generate the list file for debugging:

```
$ dlxasm -l userprog.lst userprog.dlx
```

This command will create two files: the list file for debugging, and the executable bytecode file name userprog.dlx.obj. The assembler automatically names the executable file the same name as the input DLX file with the ".obj" extension added.

Note that this command differs from the command used to compile the operating system in that we don't have to explicitly tell it where the first instrcution is: it will automatically use your main function as the entry point.

For your convenience, we have included makefiles in the project folder that does all the compiling and assembling jobs for you. To compile the os, all you need to do is to run following commands:

```
$ cd ~/ee469/lab1/os
$ make
```

If successfully compiled, the executable bytecode of the os will be `~/ee469/lab1/os/work/os.dlx.obj`.

Similarly, you can compile your user program (suppose it is `~/ee469/lab1/apps/userprog.c`) by running following commands:

```
$ cd ~/ee469/lab1/apps
$ make
```

If successfully compiled, the executable bytecode of the userproj will be `~/ee469/lab1/apps/work/userprog.dlx.obj`.

In order to run your user program inside the DLXOS operating system (which is running on the DLX hardware simulator), you must pass the name of your compiled executable file as a command-line argument to the operating system. Recall from the "DLX Hardware Simulator" section above that your operating system (for simplicity, assume it is named "os.dlx.obj") can be run with the following command:

```
$ dlxsim -x os.dlx.obj -a <arguments to operating system>
```

If you read the operating system source code, specifically the main function (found in process.c), you can find the available arguments to the operating system. You are welcome to add any you would like. Of the existing arguments, the "-D" and "-u" flags will be of the most use to you. The "-u" flag is needed to run a user-space program. You can use it like this to run your program (assuming the compiled program is `~/ee469/lab1/apps/work/userprog.dlx.obj` and the compiled os is `~/ee469/lab1/os/work/os.dlx.obj`):

```
$ dlxsim -x ~/ee469/lab1/os/work/os.dlx.obj -a -u
~/ee469/lab1/apps/work/userprog.dlx.obj
```

User Traps in DLXOS

Since one of the primary tasks in this lab is to implement your own user trap within DLXOS, it is important to understand how traps are implemented in DLXOS. You only need to read a few files:

- **include/os/traps.h:** specific numbers (called "vectors") are #defined here for each trap. This number is passed to the "trap" assembly command to tell the hardware which trap should be run.

- **os/usertraps.s:** each assembly procedure listed here is a user-space trap. The convention is that the name of the function in C is the part of the procedure name after the "_". For instance, consider these lines from userprog.s:

```
.proc _Printf
.global _Printf
_Printf:
        trap    #0x201
        jr      r31
        nop
.endproc _Printf
```

  This defines a function for any user C programs called "Printf" (Note the capital "P"!). This function will execute the "trap" assembly instruction, passing it the number 0x201. If you look in include/os/traps.h, you will see that the constant TRAP_PRINTF is defined as 0x201. The trap instruction transfers control to the _intrhandler procedure written in dlxos.s.

- **os/traps.c:** the "dointerrupt" function in this file is called by _intrhandler (from dlxos.s) whenever an interrupt occurs. If you look inside the dointerrupt function, you will see a switch statement where all the trap vector constants are checked against the "cause" variable. This variable contains the number that was passed to the trap instruction. If you scroll down to the "TRAP_PRINTF" case, you will see what happens when a user program calls the Printf function. Note also that any traps which return values use the "ProcessSetResult" function to return the value through the trap to the user program. Read through other trap handlers in the switch statement to figure out how user parameters to the trap functions are popped and processed. At this point we are in kernel mode.

- **include/usertraps.h:** this file simply contains function declarations for all procedures defined in os/usertraps.s. This file is only used to prevent the compiler from warning about implicitly defining functions.

That's all there is to traps in DLXOS. To implement your own trap, simply modify the proper locations in those files.

DLXOS System Stack Tutorial

At this point you know everything necessary to implement all your code for lab 1. This section discusses background material that is vital to your understanding of how various key functions operate in a real operating system. While this information is not necessary for completing lab 1, it may show up on any future exams. You will therefore be responsible for learning this material.

Recall from the course lectures that when an interrupt occurs, the system is asked to stop whatever it is currently doing, do something else for awhile, and then return to what it was originally doing. This sounds easy enough, but there is no way to tell the original function that it was interrupted. Think about this in your own code for a minute. How would you write code if you needed to check for interrupts all the time? Consider a simple HelloWorld program:

```
#include <stdio.h>
int main() {
  printf("Hello World!\n");
  return 1;
}
```

A first cut at modifying this to check for interrupts would look like this:

```
#include <stdio.h>
int main() {
  checkforinterrupt();
  printf("Hello World!\n");
  checkforinterrupt();
  return 1;
}
```

You will learn later in class why even that wouldn't work due to synchronization problems. Also, what if the checkforinterrupt() function gets interrupted? It just wouldn't work. Notice that even this simple solution has effectively almost doubled the size of the program. Therefore, since you can't write programs that are aware that they have been interrupted, you must instead make interrupts happen in their own "sandbox." This means that when the system returns from the interrupt, all of the original process's memory should look the same as when the interrupt occured.

To make this possible, recall from the course lectures that any interrupt service routine begins with saving various data prior to servicing the interrupt, and then restores those data after the interrupt is complete before returning control to the original function.

The data that need to be saved include:

- The address of the last instruction (otherwise known as the "return address"), which enables the system to resume where it left off. This is equivalent to saving the value of the Program Counter (PC).
- any processor registers that will be changed by the interrupt service routine during execution.

It turns out that the best data structure to store this information is a stack. This enables you to think of the entry into the interrupt service routine as "pushing" informtion onto the system stack, and the exit of the interrupt service routine as "popping" information back of the top in the reverse order that it was pushed. This algorithm allows for interrupts to interrupt other interrupts.

This is the same way that a standard C program deals with function calls: all information that is "local" to a function is stored on a function call stack in individual stack frames, and the stack frames are pushed and popped as functions are called and return. The stack structure in C programs enables recursive functions, since local variables and function parameters are stored on the stack.

Take look at how DLX pushes and pops registers when one of your user traps is called. This code is located in os/dlxos.s, in the _intrhandler and _intrreturn function. The _intrhandler function is called directly by the DLX hardware when an interrupt occurs. The hardware has saved the last PC address in register 31 (r31). The _intrhandler then determines if the current program is a user program or a kernel program, and saves the current stack pointer accordingly. It then pushes all the registers (r1-31 and f0-30) onto the stack, saves some more information about the interrupt on the stack, and finally calls the C function "dointerrupt" that is written in traps.c.

If you look at dointerrupt, you'll notice that the last thing it does is call the "intrreturn" function, rather than actually calling the standard C "return" statement. This is because all the registers that were previously pushed onto the stack must be popped back off. If this is not done, then the stack will be corrupted and your computer could lock up. Or explode.

Open dlxos.s again and look at the _intrreturn function. You'll see that it basically does the same thing as _intrhandler, except in the opposite order (popping instead of pusing).

You must read the following pdf, which discusses memory layout - in particular the stack - in addition to the C calling convention. This will help to clarify some of the assembly code you saw in _intrhandler and _interreturn.

Note that you will not have to modify _intrhandler or _intrreturn in order to complete this lab. The writers of the operating system were nice enough to do the hard work for you. You now should be able to answer the question, "In usertraps.s, what does the 'jr r31' instruction do?"

Assignment

Setting up the simulator and compiling DLXOS

1. **Directory Setup:**

You should have already setup your directory structure on an ECN Linux machine following the instructions given earlier in this lab document. These files must be given 700 permissions to prevent other students from having access to them. If you do not know what "700" permissions means, a quick "man chmod" or Google search for "chmod" will tell you what you need to know.

2. **Path setup:**

   When you type "gcc-dlx", "dlxasm", or "dlxsim", the Linux shell needs to know where to look to find these programs. This is accomplished via the "PATH" environment variable. You must add the path to these programs to your PATH.

   If you are using CSH as your shell, this can be accomplished with the following command:

   ```
   $ set path = (/home/shay/a/ee469/labs/common/dlxos/bin $path)
   ```

   If you are using BASH as your shell, this is accomplished with the following command:

   ```
   $ export PATH="/home/shay/a/ee469/labs/common/dlxos/bin:$PATH"
   ```

   You can find out the type of your shell by the following command:

   ```
   $ echo $SHELL
   ```

   Unfortunately, this modification to your path will be lost when you logout, since environment variables are loaded on login. To keep from having to retype that command each time you login, you can add that command to the file that gets run each time you login. In CSH, this file is ~/.cshrc. In Bash, this file is ~/.bashrc. You can add it like this:

   ```
   $ echo "set path = (/home/shay/a/ee469/labs/common/dlxos/bin $path)" >> ~/.cshrc
   ```

   Note that this simply writes that command to the .cshrc file: it does not run it until you login again. If you want to run it immediately, you can run all the commands in ~/.cshrc with the "source" command like this:

   ```
   $ source ~/.cshrc
   ```

   Voila! Now your environment will be setup correctly each time you login.

3. **Implement a new user trap:**

   You must implement a new user trap called "Getpid" which returns an integer representing the ID of the current process. The ID of a process in DLXOS can be defined as simply the process control block's index in the array of all process control blocks. You can easily find this number through simple pointer arithmetic involving the currentPCB and pcbs pointers in process.c. The user trap should have the following prototype:

   ```
   unsigned int Getpid();
   ```

   You must write the following function in os/process.c (and put its declaration in include/os/process.h) and use it in traps.c in order to maintain the logistical structure of DLXOS (i.e. functions dealing with processes go in process.c):

```
unsigned int GetCurrentPid();
```

The general procedure for adding a new trap to DLXOS is shown below:

1. Create a new trap vector in include/os/traps.h. Note that new user trap vectors should be greater than 0x430.
2. Specify the trap procedure "_Getpid" in os/usertraps.s.
3. Put the function prototype for your trap procedure in include/usertraps.h.
4. Write the function that will perform the action associated with your new trap (in this case, the GetCurrentPid() function).
5. Write the actual trap handler code in the dointerrupt function in traps.c to call your new function.

4. **Modify apps/userprog.c to test your new trap.**

Turning in your solution

You should make sure that your `lab1/` directory contains all of the files needed to build your project source code. You should include a README file that explains:

- how to build your solution,
- as anything unusual about your solution that the TA should know,
- and **a list of any external sources referenced while working on your solution**.
  DO NOT INCLUDE OBJECT FILES in your submission! When you are ready to submit your solution, change to the directory containing the `lab1` directory and execute the following command:

  ```
  turnin -c ee469 -p lab1-Y lab1
  ```

  where `Y` stands for your lab session, so it will be lab1-1 or lab1-2 or lab1-3 depending on which lab session you are in.
  Wednesday, 2:30pm = 1, Friday, 11:30am = 2, Friday, 2:30pm = 3.