CS314 - Operating Systems Project 1 (Fall '23)

1 Administrivia

This is the introductory project. Everyone should work independently and turn in their own work. Do not work in groups yet.

The purpose of this project is to get started with DLXOS, the mini-OS used in course projects. You will learn how system calls are implemented, the understanding of which is essential for the remaining projects. The following topics are covered by this project:

- Setting up DLXOS
- Context switch, stack frame, trap implementation tutorials
- Debugging with print statements
- Implementation of the `GetPID()` system call

2 DLXOS

The base for all projects in this course is a very simple, but functional, operating system called DLXOS. The system was written at the University of Maryland Baltimore County and the University of California, Santa Cruz, and is based on the DLX instruction set architecture described by the Hennessy and Patterson textbook. Over the course of the semester, your job will be to improve the functionality and performance of DLXOS.

As much as possible, the assignments are structured so that you will be able to finish the project even if not all of the pieces you have built are working.

It's important to realize that while you run DLXOS on top of this simulation as a user program on UNIX, all of the code you write is exactly the same as if DLXOS were running on bare hardware. The simulator runs as a user program for convenience: multiple students can run DLXOS at the same time on the same physical machine. These same reasons apply in industry, since it's usually a good idea to test out system code in a simulated environment before running it on potentially flaky hardware. You do not typically have the option of throwing out a running machine to be replaced with a CPU with different features before your code will work.

2.1 Useful DLXOS pages

The DLX simulator homepage contains changes to the basic DLX architecture that are related to operating system support. For more information on DLX instruction set, please refer to the following DLX instruction set.

3 Setting Up DLXOS

Do as shown in your home directory (do this on os.cs.siue.edu)

```
mkdir -p ~/cs314/
cp ~/Public/cs314/project1.tgz ~/cs314
cd ~/cs314/
tar xvzf project1.tgz
```

Depending on which shell you are running (`echo $SHELL` to find out), you need to set your path variable accordingly. If done correctly, this will be the only time you'll have to do this. Below are instructions for csh, tcsh, and bash (bash is preferred).

3.1 bash

If using bash, add the line below at the end of your `~/.bashrc` file to permanently add the path to the DLX compiler and assembler:

```
PATH=${PATH}:/home/<your username>/Public/cs314/dlxtools/bin
export PATH
```

Execute the following to have the shell re-read your `~/.bashrc` config file (you only have to do this here since you changed .bashrc, which will automatically be read each time you login, i.e. start a new shell instance):

```
source ~/.bashrc
```

3.2 csh

If using csh, add the line below at the end of your `~/.cshrc` to add the path to the DLX compiler and assembler:

```
set path=(/home/<your username>/Public/cs314/dlxtools/bin $path)
```

Execute the following to have the shell re-read your `~/.cshrc` config file:

```
source ~/.cshrc
```

3.3 tcsh

If using tcsh, add the line below at the end of your `~/.tcshrc` to add the path to the DLX compiler and assembler:

```
setenv PATH ".:/home/<your username>/Public/cs314/dlxtools/bin:${PATH}"
```

Execute the following to have the shell re-read your `~/.cshrc` config file:

```
source ~/.tcshrc
```

4 Compiling

To compile, type `make` in your `~/cs314/src` directory.

To compile the user program, type `make userprog` in your `~/cs314/src` directory. The `Makefile` is set up to build both the OS and user program `userprog` by default.

5 Running

To run the user program in DLXOS, assuming you're in the `src` directory, you'll run:

```
cd ../execs
dlxsim -x os.dlx.obj -a -u userprog.dlx.obj
```

You should see "Hello there" printed somewhere.

Note: DLXOS seems to leave your terminal in an abnormal state after the above program executes (you don't see what you're typing, but it's there). If this happens, execute `reset`. You will not see it as you type in, but things should be back to normal after it executes.

Look at the `Makefile` to see how `userprog.dlx.obj` is made.

6 Debugging

Unfortunately, using the debugger is nigh useless here, because of the simulation environment nestling DLXOS. You will have to rely on printed statements. To print stuff from the user programs use the `Printf` procedure, as in the provided `userprog.c`, you will not be able to use standard libraries in DLXOS user programs.

To print from the kernel, you may use printf as usual, BUT there is a lovely `dbprintf` macro defined in dlxos.h that you should take a look at. Find usage examples just about anywhere.

From dlxos.h

```
// dbprintf() is a VERY useful macro.  It gets used as follows:
// dbprintf ('x', "This prints %d and %x\n", val1, val2);
// This will print the expected string only if the debugging options contain
// the letter 'x'.  This allows users to turn on different debugging
// statements at different times by using different letters.  For example,
// process debugging statements could use 'p', and memory 'm'.
Specifying
// a '+' in the debugging string will turn on all debugging printfs.
#define dbprintf(flag, format, args...)                 \
    if ((dindex(debugstr,flag)!=(char *)0) ||   \
        (dindex(debugstr,'+')!=(char *)0)) {    \
        printf (format, ## args);               \
    }
```

Got that? Basically, use dbprintf to write debugging printfs that display conditional upon being specified as a parameter when you start DLXOS. So... from the execs directory, assuming everything compiled correctly:

```
dlxsim -x os.dlx.obj -a -D p -u userprog.dlx.obj
```

This invocation will run userprog.dlx.obj as usual, but will also cause all dbprintfs containing the 'p' flag (used in process-related kernel functions) to be printed.

```
dlxsim -x os.dlx.obj -a -D mp -u userprog.dlx.obj
```

This invocation will run userprog.dlx.obj as usual, but will also cause all dbprintfs containing either the 'p' or 'm' flag (used in process- and memory-related kernel functions, respectively) to be printed.

```
dlxsim -x os.dlx.obj -a -D + -u userprog.dlx.obj
```

This invocation will run userprog.dlx.obj as usual, but will also cause all encountered dbprintf statements to print. This is fairly voluminous, so I suggest you consider an unused flag for use with any changes related to the project, so that you only see the dbprintf output you're actually interested in. Try it and save yourself some headaches down the road.

7 Major Source Components

In your `~/cs314/src` directory, you will find the following important files:

- `process.c`: Process management routines and the main routine which creates the initial process.
- `sync.c`: Synchronization routines.
- `memory.c`: Memory management routines.
- `filesys.c`: The simulator's file system's basic routines.
- `traps.c`: Low-level trap stuff.
- `sysproc.c`: System process definitions (such as the initialize system process).
- `queue.c`: Basic routines for implementing a queue (may be needed in your assignments).

8 Traps

To understand how traps work in DLXOS, you must familiarize yourself with the following files: `process.h dlxos.s usertraps.s process.c`.

If you are unfamiliar with the DLX instruction set architecture, find an online tutorial. Google is very helpful here. Search for "DLX ISA" or "DLX ISA tutorial". Or, you can simply look at the code that is provided. We won't be digging into the instruction set.

The procedure for adding a new trap to DLXOS is as follows (**fork() is only used as an example, read the assignment carefully because you're not implementing fork()**;

1. Find a new trap vector. If you open `traps.h`, you'll see lots of trap vectors. To create one of your own, you'd add something like

   ```
   #define TRAP_PROCESS_FORK       0x430
   ```

2. User trap interfaces are in `usertraps.s`. **To add one for the above example**, you'd need to add:

```
.proc _fork
   .global _fork
   _fork:
       trap    #0x430    ; the vector you defined above
       nop
       jr      r31
       nop
   .endproc _fork
```

Note that this is **not what you're implementing, because GetPID() does something completely different**.

3. Now, once the user calls `fork()`, the architecture simulator will execute the trap instruction and transfer control to `_intrhandler` in `dlxos.s` (read this handler to understand the major steps of pushing stacks, etc).

4. Finally, you'd create a trap handler for `fork()`. Read `dointerrupt()` in `traps.c` to figure out how user parameters are popped and processed. At this point, we are in kernel mode.

9 Assignment

Modify `usertraps.s` to provide a user trap interface for `GetPID`.

For example, GetPID should look something like this:

```
.proc _Getpid
.global _Getpid
_Getpid:
        trap    select_some_unused_value
        jr      r31
        nop
.endproc _Getpid
```

Implement `GetPID` support in the kernel.

Finally, make sure that this works as your `userprog.c`:

```
main()
{
   int pid;
   pid = GetPID();
   Printf("The process pid is %d\n", pid);
}
```

Hint: Besides `usertraps.s`, You may end up modifying `process.c`, `process.h`, `traps.c`, `traps.h`, and of course `userprog.c`. However, if you write much more than 20 lines of code, you may be overthinking it...

Second hint: Look at the end of process.c:`ProcessFork` for a "clue" about how you might compute the PID. Keep in mind that DLXOS keeps track of the currently executing process with the currentPCB pointer.

Third hint: Look at ProcessSchedule in process.c, and specifically consider how often this function will be invoked.

Last hint: To return a value from the kernel to the user program, see process.c:`ProcessSetResult`, which ends up getting used extensively by traps.c:`dointerrupt`.

10 What to Turn In

In design.txt write up how you arrived at a solution. Describe the sequence of events from the time the getpid system call is made by the user program, until the user program returns the correct value. If you were not successful in producing the solution, write what should have happened. For completeness, you should describe the control flow of your solution (i.e., the functions called and their respective input/output). **No points will be awarded without a completed design.txt!!!**

Turn in your entire project directory, including the design.txt, as a single archive. If you followed the instructions above, you should have the following directories:

```
~/cs314/execs
~/cs314/src
```

So, if you were me, you might tar and compress it as follows:

```
cd ~
tar cvzf project1_icrk.tgz ~/cs314
```

But, you're not me, so you wouldn't put icrk in that filename. Use your own username to name your file.

Now, if you were me and you had to upload this file and you didn't have a copy on your local machine, you might scp the thing to your local machine first:

```
scp icrk@os.cs.siue.edu:~/project1_icrk.tgz .
```

Which would put the remote file in the directory that you were in when you ran the command. If you're running Linux or cygwin that might work. Otherwise, if you're stuck in Windows-land, PuTTY includes PSCP (an scp client) that I'm sure you can use somehow to get your file.