CS230/330: Project 1
Hello Yalnix

Handed out: Thursday, January 11, 2001
Due: Friday, January 19, 2001, 11:59 p.m.

A physicist, an engineer, and a computer scientist were discussing the nature of God. Surely a physicist, said the physicist, because early in the Creation, God made Light; and you know, Maxwell's equations, the dual nature of electro-magnetic waves, the relativist consequences... An Engineer!, said the engineer, because before making Light, God split the Chaos into Land and Water; it takes a hell of an engineer to handle that big amount of mud, and orderly separation of solids from liquids... The computer scientist shouted: And the Chaos, where do you think it was coming from, hmm?

– Anonymous (from the GNU autoconf manual)

1 Assignment

In this assignment, you are going to start with nothing and develop the initial pieces of the Yalnix kernel. To do this, you must produce a kernel that initializes all of its interrupt and trap vectors, recognizes system calls, and runs an idle process. In addition to this, you are required to work out an initial high-level design for your kernel. This project does not require a lot of programming, but it does require a reasonable understanding of the overall project. Make sure you have read the Yalnix kernel handout a few times before beginning this assignment.

2 Project Groups

Before doing anything else, you need to do the following:

1. Form a group of up to three people (two for graduate students) and let us know who is in your group and what name you want the group to be given (up to eight letters). We will use this name to create a Unix group for your project. If you don't know what a Unix group is and how it pertains to file security, please refer to the information in the man pages for `chmod,` `chgrp,` and `newgrp`.

2. We will create a directory and repository for use by your group. After this directory has been created, modify your `.cshrc` file (or other login files) to set the `CVSROOT` environment variable to your group project repository. Please see Ian myself if you have any trouble setting this up.

3. Create a project directory "yalnix" in your group directory and check it into CVS.

4. Copy the file /yalnix/sample/Makefile.template to the yalnix project and rename it to Makefile.

You are now ready to begin.

3 Starting Out

1. Write a kernel.c file that defines a rudimentary kernel main() function. Your first main function should look something like this:

```
#include "hardware.h"

int main(char *args[], cpu_context_frame *frame) {
   WriteRegister(REG_CONSOLE,'!');
   Halt();
}
```

2. Compile your two-line kernel into 'yalnix' using the supplied Makefile. Now, boot your kernel by typing 'yboot'. If all is well, you will see a single '!' printed and your kernel will exit. Congratulations, you have just written the world's smallest and most useless operating system–well, besides DOS.

3. Now, it's time to write a few functions to help you debug your kernel. These functions are going to produce various types of output on the hardware console. First, implement a function kputchar(int c) that writes a single character to the console port like this:

```
void kputchar(int c) {
    while (ReadRegister(REG_CONSOLE));   /* Console is busy, wait. */
    WriteRegister(REG_CONSOLE, c);
}
```

   Now, write a collection of functions to do the following:

   - kprints(char *s). This function prints a normal string to the console using your kputchar() function.
   - kprintd(int d). Print a decimal number to the console.
   - kprinth(unsigned int h). Print an unsigned number in hexadecimal to the console.

4. Modify your kernel so that it prints "Hello Yalnix" using your kprints() function. Recompile your kernel and run to make sure it works.

5. Modify your kernel so that it reads the amount of physical memory from the REG_PHYSM register and prints out the value using your kprintd() function.

6. Modify your kernel so that it prints the value of the cpu_context_frame * parameter as a hexadecimal number using your kprinth() function.

7. At this point, the output of your kernel should look something like this:

```
Hello Yalnix
memory = 1048576
frame  = 0xffbdd858
```

8. Remove the call to Halt().

9. Write dummy trap (interrupt) handling functions that simply prints out a message indicating the type of interrupt and the current value of the program counter. Modify kernel main() to initialize the exception vector to point at these dummy routines. Note: don't forget to load the address of the exception vector into one of the special machine registers. The output of each interrupt handler should appear something like this:

```
TRAP_CLOCK : pc = 0x3265d8
TRAP_MEMORY : pc = 0x3265d8
... etc ...
```

   Keep in mind that this output needs to be produced by your kprint functions defined earlier.

10. Write a simple in-kernel idle process that prints a message stating that it is executing each time around the idle loop. It should look something like this:

```
void KernelIdleLoop() {
     while(1) {
          kprints("Kernel idle.\n");
          Pause();
     }
}
```

11. Get yalnix to jump to this function after you exit kernel main. To do this, you will need to set the program counter (PC) field in the context structure (passed to the main routine) to point to your main function. In addition, you will need to set the stack pointer (SP) field to a suitable location (you will need to experiment). After setting these values, simply return from main. Now, run your kernel. You should see the timer interrupt calling your exception handler every second, and your idle loop running. If you're feeling lucky, go to the Yalnix display window and move the mouse around with

the button down. You should see a whole bunch of interrupts being generated in your kernel in addition to the clock interrupt. Congratulations, you have just created the first Yalnix process!

12. Implement dummy handling functions for all of the system calls required in the project (FORK, EXEC, etc...). These handler functions should do nothing more than print the name of the system call and return. Modify the trap handler for `TRAP_KERNEL` to invoke the appropriate system call handler. You should also check for invalid/unknown system calls and print an appropriate error message. Note : the system call numbers can be found in the file `/yalnix/include/yalnixcall.h`.

13. To test all of your trap vectors and system calls, you can run your kernel by booting it using "yboot -p". This option pings all of your kernel's interrupt vectors and system calls to make sure you have covered all of the entry points to the kernel. If you have done everything correctly, your kernel will print a series of messages for each trap and system call.

14. For fun, see if you can get the pen and display devices to do anything. In your `TrapPen` interrupt handler, read the value of the `REG_PEVTR` register to find out where the user touched the screen. Then write a command to the `REG_DCTLR` register to plot a single pixel on the display. Now, hold down the mouse button and move the mouse around in the Yalnix window. You should see a bunch of points being drawn.

4 Organizing your work

Once you have everything in the first part working, your next step is to complete the following steps:

1. Go through the kernel project handout and make a list of all major subsystems and subcomponents that will form the implementation of your kernel. In your list, you should list all of the system calls and traps that make up each subcomponent.

2. Next, break up your solution to the first part into a collection of files that correspond to the subsystems in the previous step. For example, instead of having everything in `kernel.c`, you should have files such as 'process.c' for process management system calls, 'serial.c' for interrupt handlers pertaining to the serial device, and so forth.

3. Once your have broken up your kernel into files, create a header file for *each* of your kernel subcomponents (e.g., `process.h`, `serial.h`, etc...). Within each header file, provide the C prototypes for all of the functions that you are going to include in the associated `.c` file. You should do this by making declarations such as the following:

   - `extern void TrapMathHandler(cpu_context_frame *);`

   As a general rule, students have found it easier to declare all trap handlers and system calls with exactly the same prototype. For example:

```
extern void TrapMemory(cpu_context_frame *);
extern void SysRFork(cpu_context_frame *);
extern void SysExec(cpu_context_frame *);
... etc ...
```

Please note that inside the kernel, the function prototypes and names of the system calls do not have to match the functions that user programs will use to access system calls. This is because user programs do not directly link against your kernel functions. Instead they issue a trap which is handled through your `TRAP_KERNEL` handler function.

4. Modify each `.c` file so that it includes its corresponding `.h` file.
5. Modify your Makefile and any other related functions so that your kernel compiles and works exactly as it did when it was all in one big file.
6. In your README file, write a short listing of *every* file in your kernel along with a short description of the system calls, traps, and additional functionality that you plan to implement.

5 Testing

A reference kernel is available in `/yalnix/kernel/yalnix` You can run this kernel to view the desired behavior of what you will be implementing (note : you must run the reference kernel from your own directory). The correct output of the kernel ping test should appear something like this:

```
% yboot -p
HW: Initializing memory... 1048576 bytes
KernelBoot: Loading 'yalnix'
Starting the kernel. 100286c
Hello Yalnix
frame = 0xffbdd858
Initializing traps.
**** HARDWARE WARNING **** VIRTUAL MEMORY NOT ENABLED

*** Pinging the interrupt vectors...
TRAP_CLOCK   : pc = 0x0100282c
TRAP_ILLEGAL : pc = 0x0100282c
TRAP_MEMORY  : pc = 0x0100282c
TRAP_MATH    : pc = 0x0100282c
TRAP_SERIAL : pc = 0x0100282c
TRAP_PEN    : pc = 0x0100282c
TRAP_FLASH  : pc = 0x0100282c
TRAP_POWER  : pc = 0x0100282c
```

```
*** Testing system calls...
TRAP_KERNEL  : pc = 0x0100282c
SysRFork, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysExec, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysExit, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysWait, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysBrk, frame =  = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysDelay, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysGetPid, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysYield, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSuspend, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysResume, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSerialRead, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSerialWrite, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysPenRead, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysDisplayWrite, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaCreate, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaSignal, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaWait, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaDestroy, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaDestroy, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSend, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysReceive, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysConsoleWrite, frame =  = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
UNDEFINED SYSTEM CALL
```

```
TRAP_KERNEL  : pc = 0x0100282c
UNDEFINED SYSTEM CALL
0 Kernel Pages in use.

** Machine Halted **
4 seconds elapsed.
%
```

6 Handin procedure

Your project must be handed in via CVS as follows:

- Make sure all of the files needed to recreate your kernel (source code, header files, and makefiles) have been added to a CVS project named 'yalnix.' You will need to use the `cvs add` command to do this.
- Add your README file to the repository. Your README file should contain all of the names and email addresses of your team members. In addition, it should include the short writeup previously described.
- Make sure you commit all of your files to the repository using `cvs commit`.
- Test your handin by going to a temporary directory and trying it out:

  ```
  % cvs checkout yalnix
  ...
  % cd yalnix
  % make
  % ./yalnix
  ```

- Tag your handin as 'project1' by typing '`cvs tag project1 yalnix`' in the directory that contains your 'yalnix' project directory (e.g., you should not type this while inside your yalnix project directory itself).
- You're done. We will check out your project by doing a '`cvs checkout -r project1 yalnix`' on your repository.

7 Grading

This part of the project will be graded on correctness as well as your proposed organization of the kernel. The following grading criteria will be used:

- Printing the "Hello Yalnix" message (15%).

- Initializing the interrupt/trap vector correctly and printing the correct interrupt messages in the exception handler (15%).
- Initializing the idle process and having it print the correct messages (15%).
- Modifying the `TRAP_KERNEL` handler to invoke functions for all of the required system calls and printing the correct system call messages (15%).
- The organization of your files and identification of kernel subsystems (40%).
- **If we cannot compile your kernel, you will receive NO CREDIT!**

8 Implementation Hints

- The C prototype for all of your trap handlers is as follows:

  ```
  void InterruptHandler(cpu_context_frame *);
  ```

- The interrupt vector should be created as an array of pointers to handler functions. The C syntax for doing this is as follows:

  ```
  void (*ivector[TRAP_VECTOR_SIZE])(cpu_context_frame *);
  ```

  Alternative, you can also do it with typedefs like this:

  ```
  typedef void (*handlerfunction)(cpu_context_frame *);
  handlerfunction ivector[TRAP_VECTOR_SIZE];
  ```

  Please refer to the K&R C programming for gory details on function pointers.

- To debug your kernel after a crash, there are a couple of things you can do. First, you can type '/yalnix/bin/ydb' to produce a simple stack trace. Alternatively, you can run gdb on the core file. To do this, you must run gdb as '`gdb yalnix core`'.
- Start early. You can probably finish this project in a couple of hours if you are really on top of everything. However, you probably don't want to wait until the last minute.