CS230/330: Project 1
Getting Started on the Yalnix Kernel

Handed out: Friday April 7, 2000
Due: Friday, April 14, 2000, 11:59 p.m.

1 Assignment

The purpose of this project is to 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 Getting Started

To start out, 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 Dustin or 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`.
5. Write a `kernel.c` file that defines a rudimentary `KernelStart()` function that simply prints "Hello Yalnix" and then calls `exit()`. In addition, write a function `SetKernelBrk(caddr_t addr)` that prints the message "SetKernelBrk not implemented yet" and returns. Modify the `Makefile` to create the yalnix executable from this file. Run "yalnix" to test your kernel. If it works, the "Hello Yalnix" message should appear.
6. Remove the call to `exit()` from `KernelStart`. Insert a call to `BogusInitMemory(pmem_size)` at the beginning of `KernelStart`, passing in the physical memory size that is passed

to `KernelStart`. `BogusInitMemory` is a temporary hack to set up basic memory mappings for the kernel (this call will be removed later). Don't forget to include a prototype for it like this:

- `extern void BogusInitMemory(int)`

7. 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 `KernelStart` 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 = 3265d8
TRAP_MEMORY : pc = 3265d8
... etc ...
```

8. 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) {
          write(1,"Idling indeed\n",14);
          Pause();
     }
}
```

9. Get yalnix to jump to this function after you exit `KernelStart`. To do this, you will need to set the program counter (PC) field in the context structure (passed to the `KernelStart` routine) to point to your `KernelIdleLoop` 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 KernelStart. Now, 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!

10. 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`.

11. To test all of your trap vectors and system calls, you can run your kernel using the special "yalnixping" command as follows (note: this command should only be used for this step of the assignment):

    ```
    % /yalnix/bin/yalnixping
    ```

    This will start yalnix and "ping" your kernel with all of the required interrupts, traps, and system calls. If you have done everything correctly, your kernel will print a series of messages for each trap and system call. At this point you can relax–you have resolved all of the possible entry points to your kernel.

3 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(struct cpu_context_frame *);`

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.

4 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:

```
% /yalnix/bin/yalnixping
Hello Yalnix
BogusInitMemory: pmem_size = 4194304 (00400000).
&end 0002b200, &etext 00016750, sbrk(0) 00032000.
pmem_size: 4194304 (00400000).  Num pages: 512.
Making first 12 pages executable.
Loading initial kernel page table.

*** Pinging the interrupt vectors...
TRAP_CLOCK: pc = 12fe0
TRAP_ILLEGAL: pc = 12fe0
TRAP_MEMORY: pc = 12fe0
TRAP_MATH: pc = 12fe0
TRAP_SERIAL_RECEIVE: pc = 12fe0
TRAP_SERIAL_TRANSMIT: pc = 12fe0
TRAP_PEN_EVENT: pc = 12fe0
TRAP_FLASH_ACCESS: pc = 12fe0

*** Testing system calls...
TRAP_KERNEL: pc = 12fe0
Fork
TRAP_KERNEL: pc = 12fe0
Exec
TRAP_KERNEL: pc = 12fe0
Exit
TRAP_KERNEL: pc = 12fe0
Wait
TRAP_KERNEL: pc = 12fe0
Brk
TRAP_KERNEL: pc = 12fe0
Delay
TRAP_KERNEL: pc = 12fe0
GetPid
TRAP_KERNEL: pc = 12fe0
Fcntl
TRAP_KERNEL: pc = 12fe0
Sync
TRAP_KERNEL: pc = 12fe0
Open
TRAP_KERNEL: pc = 12fe0
```

```
Pipe
TRAP_KERNEL: pc = 12fe0
Close
TRAP_KERNEL: pc = 12fe0
Read
TRAP_KERNEL: pc = 12fe0
Write
TRAP_KERNEL: pc = 12fe0
Seek
TRAP_KERNEL: pc = 12fe0
Unlink
TRAP_KERNEL: pc = 12fe0
Dup2
TRAP_KERNEL: pc = 12fe0
Unknown system call 49
TRAP_KERNEL: pc = 12fe0
Unknown system call 2000
512 Kernel Pages in use.

** Machine Halted **
%
```

5 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.

6 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!**

7 Implementation Hints

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

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

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

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

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

- Start early. You can probably finish this project in an hour or so if you are really on top of everything. However, you probably don't want to wait until the last minute.