Yalnix OS

CSCI 460 Final Project Technical Document

November 15, 2020

Team members:

Christine Johnson, j81n325
Wes Robbins, d77k915
Austin Hull, q82b632
Joseph Icopini, w93m572

Introduction

In our project, we proposed to write a mini operating system called Yalnix that is based on the Yalnix specification sheet. Yalnix will be specifically written for the virtual DCS 58 computer. We intend for our operating system to implement primary operating system functionalities including facilitating virtual memory, having user and kernel space, traps, syscalls, and reading in user programs. We chose to leave off the optional terminal I/O functionality from our Yalnix implementation.

Background

The majority of research for this project was done by reading and referring to the provided Yalnix spec document. The *Operating Systems: Internals and Design Principles* textbook was an additional resource that helped explain OS design in a broader sense. When a problem came up that could not be easily solved by referring to the document, meetings were scheduled with Professor Peters to further discuss the issue. Group meetings were held twice a week on Tuesdays and Thursdays, and additional meetings were held when necessary.

The code for this project is written entirely in C and requires a high level of experience and knowledge of the language in order to get a better understanding of making an operating system. The code runs on a Linux virtual machine and requires VirtualBox and Vagrant to be installed.

Approach

Our team approached the project by first overviewing the specification sheet and then dividing different parts of the operating system among team members. The project specification and our textbook provided most of the information we needed to make progress. If a member was having difficulties with a functionality, they could talk it over with other members during the group meetings. If the group as a whole still had difficulties, we could meet with Travis, who provided helpful information to us throughout the process.

Accomplished

High-level list of functionalities completed:

- Virtual memory set up
- Successful boot
- Idle Process runs
- Kernel Context Copy and Switch functions implemented
- User Program can by loaded
- Primary Traps implemented
- System calls implemented

Functionality Overview

When booting the Yalnix executable is mapped directly into a page table to allow for virtual memory to be enabled. Page tables are set up for kernel and user spaces. Physical registers are written to facilitate the boot process. These include registers that hold addresses for the page tables and the interrupt vector table. Once Yalnix is booted and virtual memory is enabled, a default function called ‘idle’ is set up to run if there are no other processes to run. This is formalized into a process by being stored in a process control block(PCB).

Yalnix then loads in the first program (‘init’ if not specified otherwise by command line argument) and adds it to the list of running processes. Before loading in a process a new PCB is created to store all necessary information. When loading in a user process the different text, data, and stack sections are identified and mapped into the page tables held in the PCB. Additionally, the stack pointer and program counter are identified and updated in the PCB so the program starts executing in the right place. The last step for a new process is copying the kernel context into the PCB and then switching to that kernel context. This is done by calling KernelContextSwitch which in turn calls KCCopy or KCSwitch which are both functions that have been implemented to respectively copy kernel and switch kernel context. This is the last step in the boot function KernelStart().

Yalnix begins running in user space by executing the code loaded in from the original user program. Now that Yalnix is running in user space it can only access the kernel through interrupts. There are many interrupts that can be triggered, each having a distinct and important role. When an interrupt is triggered it is handled by running the corresponding trap handler whose address was stored in the interrupt vector table.

Trap handlers were used to handle interrupts, exceptions, and traps for the system. The trap handlers for TRAP_KERNEL, TRAP_CLOCK, TRAP_ILLEGAL, TRAP_MEMORY, and TRAP_MATH have been

implemented. Specific functions for the traps will be run based upon what code was sent in with the UserContext. These codes are located in the "yalnix.h" file. An additional handler has been implemented for the undefined traps TRAP_TTY_RECEIVE, TRAP_TTY_TRANSMIT, and TRAP_DISK, that prints out a notice that the trap is undefined. TRAP_KERNEL executes a requested syscall based on the UserContext code, such as calling the syscall Exec() for the code YALNIX_EXEC (0x2). For any syscall requiring arguments, the handler sends in arguments to the syscalls from the contents of the eight general-purpose CPU registers at the time the trap was called, found in the UserContext "regs[8]" fields starting at "reg[0]". The return value from the given syscall is then stored in the "reg[0]" field. TRAP_CLOCK is called from the machine's hardware clock and performs a context switch to the next runnable process when possible. TRAP_MEMORY results as an exception for when a memory access is not allowed. The UserContext code will determine whether functionality for YALNIX_MAPPERR or YALNIX_ACCERR is performed. If the code calls for YALNIX_MAPPERR, the system is requesting to enlarge the memory allocated to the stack and if more space is available the memory will be increased by the function. Otherwise for YALNIX_ACCERR, the system violated the page protection given for the corresponding page table entry, and the currently running user process is exited. TRAP_ILLEGAL occurs as an exception when the system tries to execute an illegal instruction. The trap handler then exits the currently running user process. TRAP_ MATH occurs as an exception from any arithmetic error made during the execution by the current user process. The trap handler then exits the currently running user process.

The following system calls were implemented: GetPid(), which returns the process identifier of the calling process; Brk(), which allocates or deallocates memory frames to a given address; Exit(), which ends the current process; and Delay(), which will delay a process for a certain number of clock ticks. Fork() and Exec() were partially implemented, but have not achieved complete functionality for these calls.

Difficulties

Throughout the project, our group experienced many difficulties related to implementing an operating system. Many of these difficulties revolved around the fact that coding and debugging an operating system is different in many ways than coding a normal C program. There are more things to account for than just whether your code is running correctly. While working on Yalnix our code would break because of improper context switches or stale TLB mappings. One concept that was particularly difficult to understand had to do with copying a process. In order to copy a process, new frames need to be allocated for the old process to be copied into. These frames will be mapped by a new page table in the new process. However, you can not be in user spaces at once (i.e accessing frames that are not mapped to the current page table), so in order to copy the process, the newly allocated frames first need to be mapped to the current page table and then copied. Once that is done these frames can be unmapped from the current page table and mapped to the new page table. This same concept applies to copying the kernel stack.

Working within the Yalnix framework of included files and other aspects of the virtualized environment provided some difficulties. Since these files were only available within the virtual machine running in Terminal, navigating included files involved running large amounts of terminals, each with files opened, or extensive use of cd, ls, and less to inspect these files. Given the amount of information provided in these framework files that we needed to make use of in order to write our code, this slowed the process down and made it more complicated to find the information we need. For example, function declarations and macro variables were scattered throughout several different directories, so recalling which we needed

to use involved a good amount of time spent searching. We also encountered some issues in which an imported header file would not resolve undeclared reference errors, such as working with the userland calls for syscalls.

A specific function for aborting the current Yalnix user process was not created, and the syscall KernelExit() was used in its place for TRAP_ILLEGAL, TRAP_MEMORY, and TRAP_MATH. In the future, a separate function would be written for aborting the current process to be called in the trap handlers.

One of the files included in the project was a file called "template," which gives a template to make the function LoadProgram. The template handles opening and writing to files as well as the calculations for finding the correct amount of space required for loading the program into the user stack. Integrating the pre-written code with the code that we made beforehand gave some problems with the largest being where and how to read the program into our memory structure. After this was completed, there was one line of code that didn't get fully implemented and caused strange errors in other functions, and was a hard bug to track down.

Lessons Learned

There were takeaways from doing this project including a stronger technical understanding of the operating system along with others that will be useful outside of this project. Cooperating on the code for this project via Github helped give a better understanding of how to use the Github website and command line tools for collaboration rather than as a solo programmer. Given that these skills have not been explicitly taught in the MSU Computer Science curriculum, this training is extremely valuable as we enter the workforce, where collaboration is paramount. Another lesson learned was with regard to the organization of .c and .h files and using include to organize the use of multiple files efficiently and in a way that doesn't create circular references.

Some specific takeaways about operating system functionality that we had not fully understood before this project include:
Once the kernel is finished booting and running in user mode it only accesses the kernel through interrupts and handlers.
Physical frames can not be written to if they are not mapped to the current page table.
All user programs are executables that are loaded into the operating system.

Summary

In summary, our Yalnix implementation serves as a basic operating system. The system is able to boot and be exited with various defined traps and syscalls run user programs. While implementing the Yalnix operating system took a lot of dedicated time and research, it was an excellent introduction into the more technical details of building systems alongside other programmers and with pre-existing structures.

References

Peters, T., Smith, S., Johnson, D., & Salem, A. (n.d.). *The Yalnix Project* [Scholarly project]. In *The Yalnix*

Project. Retrieved from https://www.traviswpeters.com/cs460-2020-fall/yalnix/yalnix2020.pdf

Stallings, W. (2018). *Operating systems: Internals and Design Principles*. Harlow: Pearson.