June 11, 2018

# Extension of CPE 454 Operating System

Joseph Arhar <jarhar@calpoly.edu> <josepharhar@gmail.com>

Table of Contents

**1. Abstract** **2**
- Overview 2
- Outline 2

**2. Introduction** **3**
- Features from CPE 454 3
- Anticipated Design Decisions 3
- Implemented Features 3

**3. Key Design Decisions** **4**
- Virtual Address Space 4
- User Program Library and New System Calls 5
  - Implemented POSIX Library Functions 5
    - Semaphores 5
    - IPC 5
    - Process Management 5
    - Directory Scanning 6
- Network Stack 6
  - PCI Driver 6
  - Intel E1000 Driver 6
  - Ethernet Driver 6
  - IP Driver 6
  - TCP Driver 7
  - User Program Interface 7
- Preemptive Multitasking 7
  - Programmable Interval Timer (PIT) 7
  - Context Switching 7
- Design Constraints 8

**4. Future Work** **8**

**5. Reflections** **8**
- Optimizations 8
- Security 9

**6. Conclusion** **9**

# 1. Abstract

## Overview

This project extended on the operating system I wrote in CPE 454 by adding additional features on top of the existing implementation. In order to implement them, I researched operating system design patterns and hardware details. I used wiki.osdev.org for most research, just like I did in CPE 454. The source code for the project is at https://github.com/josepharhar/jos.

## Outline

This outline contains a list of features I planned on implementing in the operating system. The first list contains features which I planned on completing within the first quarter of senior project. The second list contains stretch goals which I attempted in the second quarter. I planned on completing at least one of the stretch goals.

**Goals:**

- POSIX-like Interprocess Communication (IPC)
- POSIX-like Shell
- POSIX-like Semaphores
- Preemptive Multitasking

**Stretch Goals:**

- TCP Networking with POSIX-like sockets
- POSIX-like Process Management
    - Process Tree
    - Signals
- FAT32 File Writing
- Running on bare metal
- Microkernel Architecture

# 2. Introduction

## Features from CPE 454

I implemented many features in the operating system during CPE 454, some of which I had to go back to and rewrite during the course of the senior project. Here is a list of the CPE 454 features present in the operating system:

- Interrupt handling
- Virtual addressing and swappable page tables
- Kernel threads
- VGA console output
- Keyboard input
- Physical frame allocator, virtual page allocator, and variable size memory allocator (kmalloc)
- FAT32 filesystem reading
- System calls
- User processes loaded from ELF formatted executables
- `fork()` for user processes

## Anticipated Design Decisions

Most of the new features to implement are more focused on user processes compared to the work I did in CPE 454, so I anticipated a more fleshed out design for user processes. This includes a better build system for user processes and more state stored in process contexts. Many more system calls will also be added, but I continued to use the same system call pattern I started in CPE 454.

## Implemented Features

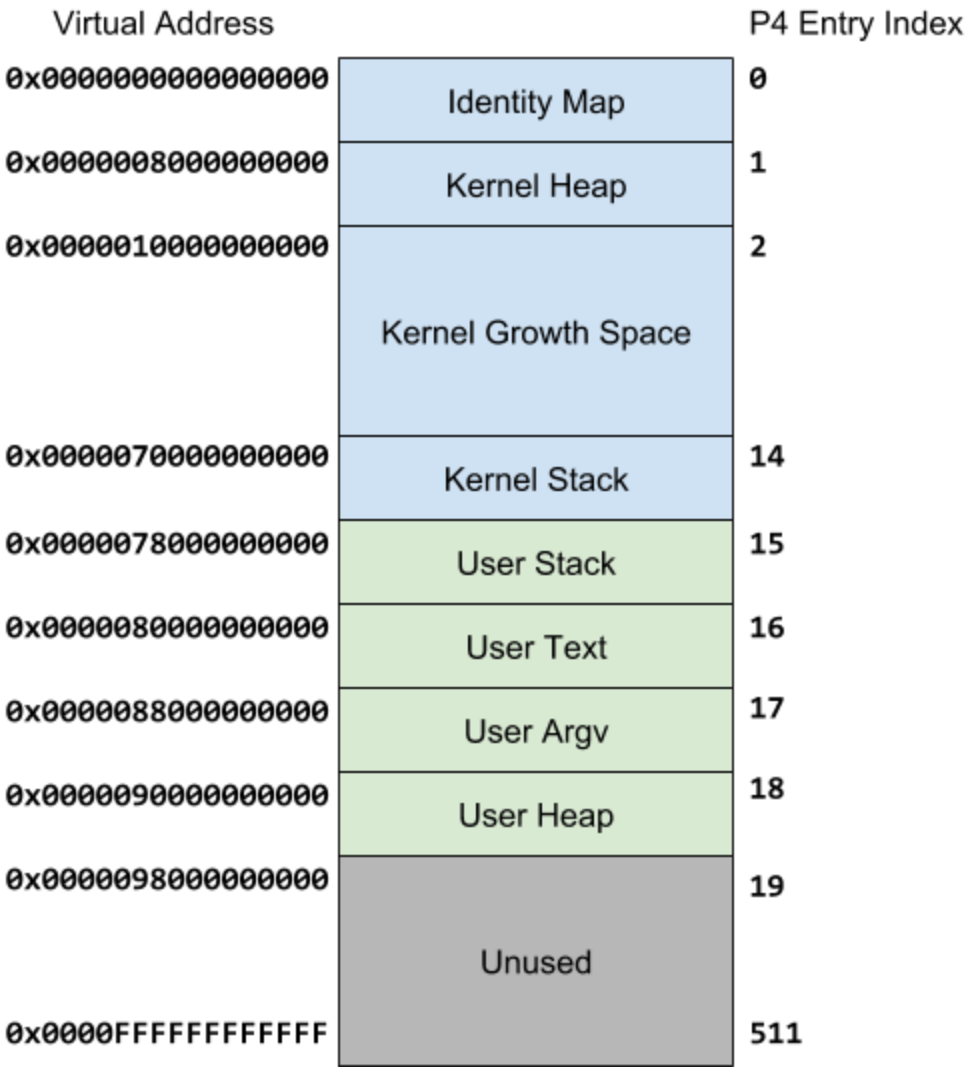
I succeeded in implementing all of my base goals and one stretch goal:

- IPC
- Preemptive multitasking
- Semaphores
- Shell
- TCP sockets

# 3. Key Design Decisions

## Virtual Address Space

The virtual address space I designed originated from Dr. Bellardo's virtual address space design from CPE 454. The additions I designed were mostly for user processes. I decided to separate each section by giving it a unique P4 index in its virtual address which gives each section more than enough virtual addresses and makes it easy to identify which section any virtual address belongs to in my page fault handler. There was a tradeoff I was not aware of when I put the user text at such a high virtual address: support for 32 bit executables and otherwise straightforward linking of user processes. In order to link user processes to start at such a high address, I had to use the `-mcmodel=large` gcc flag and disable debug information. Here is a diagram of the virtual address space, where only green addresses are user process accessible.



# User Program Library and New System Calls

In order to implement the shell, IPC, and the network stack, I needed to design an interface for user processes to communicate with the kernel through system calls. I decided to model almost all of these interfaces on the POSIX standard library. Although I could have come up with my own independently which could have been easier for me to implement or were more efficient, I decided to go with POSIX so that I could write cross-platform userspace tests of the library which could run on my OS and on Linux. Although I never ended up running my tests on Linux, I still had the advantage of being familiar with the library from prior systems programming experience.

## Implemented POSIX Library Functions

### Semaphores

When implementing semaphores, I had to choose between implementing POSIX named semaphores or POSIX memory-based semaphores. Named semaphores are accessed by using a unique semaphore name, so multiple processes can open the same semaphore by calling the library function with the same string. Memory-based semaphores are accessed by multiple processes by sharing the same region of memory which the semaphore is stored in. I decided to implement named semaphores because of the lack of shared memory support which would make it hard or impossible for separate processes to use a semaphore.
Here is the list of functions I implemented in `src/shared/semaphore.h`:

- `void sem_open(sem_t* semaphore, const char* name);`
- `int sem_post(sem_t* semaphore);`
- `int sem_wait(sem_t* semaphore);`

### IPC

I decided to implement `pipe()` for IPC because it provides a very simple interface for IPC.
IPC functions from `src/shared/unistd.h`:

- `int pipe(int pipefd[2]);`
- `int write(int fd, const void* buffer, int size);`
- `int read(int fd, void* buffer, int size);`

### Process Management

In order to implement the shell, I needed to add command line arguments for launching programs and I needed a way for the shell to wait for the child processes it runs, so I added these functions in `src/shared/unistd.h`:

- `int execv(const char* path, char* const argv[]);`
- `pid_t wait();`

### Directory Scanning

As part of having a functional shell, I decided to write an `ls` program which could list the entries in a directory on the filesystem. Referring again to the POSIX standards, I implemented the POSIX `dirent.h` in `src/shared/dirent.h`:

- `DIR* opendir(const char* name);`
- `dirent* readdir(DIR* dir);`
- `int closedir(DIR* dir);`

# Network Stack

## PCI Driver

The virtual network adapter provided by qemu, the virtual machine host my operating system runs on, is an Intel E1000 adapter attached on the PCI bus. Since it is on the PCI bus, I had to use the PCI interface to scan for the E1000 adapter, properly initialize it, and communicate with it. I heavily relied on https://wiki.osdev.org/PCI to learn how to use the x86 PCI interface. Some of the initialization required for me to use the E1000 adapter included enabling bus mastering and enabling direct memory access. With direct memory access, the E1000 adapter can write directly to physical memory regions which are selected by the BIOS.

## Intel E1000 Driver

The virtual E1000 adapter has many registers which have to be configured in order to send and receive packets. In fact, according to this Intel manual for the E1000, there is 128KB of memory for registers! I used https://wiki.osdev.org/Intel_Ethernet_i217 as an example to configure the E1000. In addition to using this register configuration, I had to use my physical address allocator to provide the E1000 physical memory locations to write and read packets and enable interrupts on the PIC line it is connected to. At this point, I can send and receive packets with ethernet headers on the network!

## Ethernet Driver

The ethernet driver handles ARP requests and responses and maintains an ARP cache to resolve IP addresses to MAC addresses. When an IP address is not in the table, it blocks outgoing packets to that IP address and sends an ARP request for that IP. When it gets a response, all of the blocked packets get sent to the appropriate MAC address.

## IP Driver

The IP driver is pretty simple, it just adds IP headers and passes off incoming packets to the TCP driver.

### TCP Driver

The TCP driver is based off another TCP implementation I wrote for CPE 465, Advanced Networks. I made it mostly by reverse engineering example TCP packet traces, and its functionality leaves much to be desired. However, it does work for making basic HTTP requests. I would have liked to complete this driver enough to be able to accept incoming TCP connections and host a web server, but that would have taken much more time.

### User Program Interface

Using TCP sockets is the only non-POSIX compliant interface I implemented because I found the POSIX interface for opening sockets overly complicated for my use case. Instead, I made one method called `socket()` which takes an IPv4 address and a port, and returns a file descriptor which can immediately be used to read or write from the socket using the POSIX IPC interface.

## Preemptive Multitasking

Implementing preemptive multitasking involved two parts: configuring the Programmable Interval Timer (PIT) and replacing the cooperative multitasking with interrupt driven context switching.

### Programmable Interval Timer (PIT)

The PIT has a several registers accessible from I/O ports containing a lot of configuration. For preemptive multitasking, I was just concerned with getting some form of recurring interrupt to switch process contexts with rather than having high precision to measure time with so I didn't modify any of the PIT registers. All I did was clear the bit on the PIC corresponding to the PIT to allow the PIT to send interrupts, and I started getting interrupts on PIC line 0. Choosing this method allowed me to implement preemptive multitasking quickly, but going through PIT configuration would be helpful if I were to implement timing functionality into my operating system later.

### Context Switching

Changing the context switching from cooperative to preemptive was simple. I replaced the process yielding function and corresponding system call with a preempt function which does the same thing, except it is called by the PIT interrupt handler to make processes get forced to switch contexts every time the PIT sends an interrupt, which with the default PIT configuration ended up being a couple times per second.

## Design Constraints

While implementing the new features and rewriting some of the functionality from CPE 454, I went back on some design choices to make things more simple and easy to write. First, I was having many issues with debugging broken system calls because of system calls interrupting themselves and making nested system calls. In order to avoid these issues, I masked the interrupt flag for system call interrupts to make them non-interruptible and I replaced the nesting pattern with a continuation passing pattern, where a system call will register a callback based on what to do after receiving an interrupt and the interrupt handler will call the callback to let the system call handler continue doing its work. I also made little or no attempt to minimize the use of memory copying, which made the kernel (especially the network stack) much slower. Were it not for these design constraints, I don't think I would have been able to implement all of my goals in time.

# 4. Future Work

There's a virtually limitless number of features I could add to this operating system. In addition to the stretch goals which I didn't implement, including FAT32 filesystem writing, richer process management, running on bare metal, and making the kernel a microkernel, there is a plethora of work to do to make it more like a usable kernel like Linux or Minix. Getting full POSIX compliance would be phenomenal because then I could compile all POSIX dependent packages. I could even run it for personal development use! In addition to full POSIX compliance, I could shoot for wider hardware support and maybe even implement SMP support. With that and further optimizations in the kernel like avoiding memory copies, it could be fast enough to compete with other kernels.

# 5. Reflections

## Optimizations

As I mentioned in the previous sections, I used memcpy generously to make things easier when I could have figured out how to pass buffers around more efficiently. In addition, I also generously swapped page tables when changing process contexts or sending buffers to, from, and between processes.

## Security

One thing I noticed when writing almost all of my system call handlers was that I was blindly accepting input from user processes like pointers and using them without making sure they are within the process's address space. This along with blindly doing pointer arithmetic on data from incoming packets are huge security vulnerabilities which are unacceptable in real kernels. If I went back I would be safer about pointer usage.

# 6. Conclusion

I had a great time working on this senior project. Getting to dig in and implement many things I've always wondered about in operating systems has given me a much better understanding of how they work and of systems programming in general. I don't think I'll have a hard time getting around similar terminology like "threads" and "processes" or "pages" and "frames" ever again now that I truly know what the underlying meaning is. Writing the network stack was wonderful because I got to see how PCI works and what working with PCI and network devices is like, in addition to actually implementing drivers for layers 2 through 4 of the OSI model. I also got to see what it's like to implement operating system features against a standard like POSIX. However, I feel like I didn't learn quite as much as I did in CPE 454 when I first started working on the operating system because in CPE 454 I had Dr. Bellardo's lectures and curriculum to speed up the learning curve. I would recommend this project and CPE 454 to anyone who is interested in going deeper with systems programming.