COMP 3430 – Operating Systems

Course Description

Calendar entry

(Lab Required) Operating systems, their design, implementation, and usage. Prerequisites: one of COMP 2140 or COMP 2061; and COMP 2280 or ECE 3610. COMP 2160 is recommended.

General Course Description

This course builds on the simple execution models used in second year through the introduction of the system software needed to manage multiple applications running on a single computer at the same time.

In this course we focus on the application of data structures and algorithms to the problem of *managing* all the resources introduced in second year: CPUs, I/O devices, and memory. Clever management of resources also gives us the opportunity to introduce and make use of new programming models that allow us to write code that works on multiple problems at the time and code that interacts with other running applications. In fact, we'll introduce the programming models as motivation for learning how an operating system does what it does.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Implement and manipulate data structures such as stacks, queues, and trees.
- Design, implement, and build complex applications requiring multiple modules with well-defined interfaces.
- Perform well-defined and structured tests on code.
- Use a variety of standard development tools such as those that automate builds or allow for the inspection of active program state.
- Write code that directly manipulates memory through pointers and address arithmetic.
- Show how a memory address is interpreted to access a word of memory given a particular memory organization.
- Differentiate between user and system code and implement a basic interrupt service routine.

Course Goals

By the end of this course students will:

- See why operating systems have grown in complexity to become one of the most important pieces of software in the world today.
- Analyze the algorithm and data structure choices made to implement resource management in the form of CPU scheduling, memory partitioning/virtualization, and the file system abstraction.
- Experience the design and implementation of an operating system as a case study in the design, development, and evolution of a large and complex software development project.
- Learn how to implement software that makes use of core operating system functionality through concurrent shared memory and message passing programming models.
- Gain a deeper understanding of the separation between system and user code.
- Gain a better understanding of what an operating system provides and how best to exploit that functionality in the code they write.

Learning Outcomes

Processes and Threads

Students should be able to:

1. Compare and contrast processes, threads, and applications.
2. Describe the lifecycle of a process.
3. Describe the lifecycle of the Unix operating system through the lens of processes.
4. Describe the operating system data structures (et al) needed to manage a process and those needed to manage a thread.
5. Write code that uses system calls to create and manage processes and threads.

Synchronization

Students should be able to:

1. Identify the critical section(s) requiring mutually exclusive access in a piece of code that will be run concurrently using threads or processes.
2. Identify correct and incorrect implementations that attempt to protect a critical section.
3. Identify code protecting a critical section that would result in deadlock and propose solutions that avoid and/or prevent said deadlock.
4. Write code that uses pthread locks and condition variables to implement a semaphore.
5. Protect a critical section using semaphores.

6. Use atomic hardware instructions to implement a lock.

Inter- process/thread Communication

Students should be able to:

1. Write code where threads communicate and coordinate activity through a shared memory buffer.
2. Write code where processes communicate through message passing via signals, pipes, and FIFOs.
3. Describe how the operating system supports and implements shared memory, signals, pipes, and FIFOs.
4. Compare and contrast the shared memory and message passing programming models and their relationship as seen through the lens of synchronization.

CPU Scheduling

Students should be able to:

1. Compare and contrast preemptive and non-preemptive (cooperative) multitasking.
2. Compare and contrast different scheduling policies.
3. Evaluate the performance of scheduling policies.
4. Describe a scheduling algorithm used by a modern operating system.
5. Write code that implements and analyzes a scheduling policy.

Memory Management

Students should be able to:

1. Discuss the benefits of partitioning memory.
2. Explain the operating system data structures and algorithms used to manage a partitioning strategy such as paging.
3. Explain how a process' entire address space can be provided/supported via the concept of virtual memory.
4. Translate virtual addresses into physical addresses.
5. Describe how the operating system works with the underlying hardware to manage paging and virtual memory.
6. Compare and contrast free space management policies.

File Systems

Students should be able to:

1. Describe the data structures used to represent files and directories as seen in Unix.
2. Show how common file operations are performed in terms of manipulating a file system's data structures.
3. Explain how files are managed as part of a process' active state.

4. Compare and contrast traditional file systems with modern approaches such as log structured and journaling file systems.