Ch.1 - Introduction

* OS Definition, goals
  + Resource managements, CPU cycles, memory, hardware, I/O, etc.
  + Control program, controls execution of any process
* Computer system structure
  + Users
  + Programs
  + OS
  + Hardware
* Kernel
  + Part of OS and always running
  + Provides essential services
* Interrupt
  + Modern OS is interrupt-driven
  + Handling interrupts:
    - Polling system
    - Vectored interrupt system
      * An array that maintains all kinds of interrupts
      * Tells us how to handle the interrupt (?)
      * Maintains the address of the interrupt service routine
  + Can be raised by hardware and software
    - Software: Provides API for users to use system calls
* I/O Structure
  + Synchronous - Blocking, wait until I/O operation is complete
  + Asynchronous - Can do something else while I/O operation is completing
* Storage Hierarchy
  + Closer it is to the CPU, the faster it is, the more volatile
  + The bigger the storage, the cheaper, and slower than CPU
  + CPU can only direct access to main memory
* Caching
  + Storage medium between registers and main memory
  + Faster than main memory, but smaller
  + Used because it’s fast and simplifies expensive calls
    - No need to constantly retrieve the same data over and over again from the slower storage medium
* Multi-process Architecture
  + SMP - Synchro, All process are equal
  + ASMP - Asynchro, Boss/worker or master/slave relation
    - Master allocates jobs
    - Slaves fulfill the jobs
* Dual-mode
  + Why can’t every process run in the same mode?
    - Security
    - Don’t want arbitrary user program to modify kernel/important data
    - Kernel calls are expensive (?)

Ch.2 OS Structures

* User interfaces
  + Command line, bath, graphical
* System services
  + Scheduling, etc.
* System calls
  + APIs so user don’t directly call the system calls
* OS Structures
  + Simple: MSDOS, everything is just one piece of a huge program
    - Makes system vars vulnerable if code is malicious
  + Break system into multiple layers where each layers can only talk to and provide services to its neighbor layers directly
    - Ideal situation, but difficult to create in practice because sometimes one module/layer may need services from a layer other than its neighbor
* Microkernel
  + Eliminate unnecessary system services to make the system smaller and easier to maintain
    - Communication will bring a lot of overhead bc kernel is responsible for communication stuff (?)
* Modules
  + No need to restart kernel/reload kernel because modules allow you do dynamically modify kernel

Ch.3 Processes

* Definition
* State and state transitions
  + Understand the transitions, sometimes they’re not arbitrary
* Process image in memory
  + Each process will have its own memory space
    - Program section, text section (code), data section (global vars), stack (temporary data), heap (dynamic allocated memory space)
* Process control block
  + Defines everything about a process
    - PID - Uinque
    - Global data
    - Program counter
    - Open files
    - Stack information
    - Register information
* Type of process
  + I/O Bound
  + CPU Bound
  + It’s critical to create a combo of both types
    - If only made CPU process, then CPU is very busy
    - If only I/O processes made, then CPU is idle and wasting resources
* Context switch
  + If want to switch process, have to save current process info
  + Not doing anything productive, just pure overhead, bc it doesn’t execute instructions
* Long/Medium/short-term scheduling
* Creating processes
  + Parent process creates child process
  + Each process has unique PID
    - Child can inherit all or most of parent process information
* Inter-process communication
  + Shared memory
    - Involves less OS involvement
    - OS only involved when generating shared reading, then all reading/writing is done on process level
    - Very cheap bc no OS involvement
  + Message Passing
    - Lots of system calls, OS is involved throughout the procedure
  + Which one is better is dependent on the actual need
  + Distributed computing uses message-passing because it better fills the need

Ch.4 Threads

* Definition
  + Unit of execution, similar to process
  + One process can have multiple threads
  + Each thread has its own program counter
* User-level
  + Supported by the thread level
* Kernel-level
  + Supported by the OS
* Multithreading models
  + One-to-one
    - One kernel thread for each user thread
    - If one users level thread is blocked, it doesn’t affect other user level threads
  + One-to-many
    - Multiple user threads with one kernel threads
    - However, only one user thread can be ran at a time (?)
    - If one user level thread is blocked, they’re all blocked
  + Many-to-many
    - Allows parallelism
    - If one user is blocked, can communicate with another kernel thread
* Thread libraries
  + UNIX, POSIX, Java

Ch.5 Process Synchronization

* Race condition
  + Based on the timing and which process starts first, we may get different results bc of shared variable
* Critical section
  + Section of shared memory that’s modified by one process at a time through Mutual Exclusion (software based solution) or hardware based solutions (different to understand and expensive, usually want OS to provide ME services)
* OS providing ME Services
  + Semaphores, Mutex Locks, etc.
* Monitor
  + Very similar to semaphores, but much easier to use