Operating Systems Theory

Chapter Zero(Preparation)

Download Putty

* ssh or download putty to access Linux server
* <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

Linux server

* Linux server: CS3060.tc.uvu.edu
* Linux userID:J\_10905862
* Linux password:vfwyyxUS

Uploading a file

* scp <files-to-transfer> J\_123456@cs3060.tc.uvu.edu:<directory-on-server>
* scp hello.c J\_123456@cs3060.tc.uvu.edu:homework/assn1
* Use scp or download WinScp to transfer file from Linux to laptop

Downloading a file

* scp J\_123456@cs3060.tc.uvu.edu:<remote-path-of-file-to-download> <local-directory-to-copy-into>
* scp J\_123456@cs3060.tc.uvu.edu:homework/assn1.tar.gz .

VPN

* Need a vpn, search campusvpn.uvu.edu

# Chapter One (Introduction)

Computer System

* Hardware
* Operating system
* Application programs
* Users

About an Operating System

* High level language tells the operating system what to do
* An operating system acts as an intermediary between the user of a computer and the computer hardware
* Execute user programs and make solving user problems easier
* Has system programs and application programs
* Resource allocator
  + Manages all resources
  + Decides between conflictng requests requests for efficient and fair resource use
* Control program
  + Controls execution of programs to prevent errors and improper use of the computer

Kernal

* most essential functionality of operating system
* Not include web browser, editor
* Book covers basic functionalities
* User’s cannot communicate with kernel
* “The one program running at all times on the computer

Shell

* Translate command to operating system
* Command interpreter
* sh and bash
* bash: is the default shell set for our program
  + is color coordinated

Harware

* CPU
* Memory
* I/O devices

Application Programs

* define the ways in which the system resources are used to solve the computing problems of the users
* Word processors, video games, web browsers

Users

* People
* Machines
* Other computers

Editor

* vim
  + Type: ‘vim’
  + To start coding type ‘i’
* Used to run ‘C’ in Linux

Bootstrap Program

* Loaded at power-up or reboot
* Typically stored in ROM or EPROM. generally known as firmware
* Initializes all aspects of system
* Loads operating system kernel and starts execution

Main Memory

* Only large storage media that the CPU can access directly
* RAM
* Typically volatile

Secondary Memory

* Extension of main memory that provides large nonvolatile storage capacity

Hard Disks

* Rigid metal or glass platters covered with magnetic recording material
* Disk surface is logically divided into tracks, which are subdivided into sectors
* The disk controller determines the logical interaction between the device and the computer

Solid-State Disks

* Faster than hard disks, nonvolatile
* Various technologies
* Becoming more popular

Caching

* Information in use copied from slower to faster storage temporarily
* Checked first to determine if information is here
  + If it is, information used directly from the cache
  + If not, data copied to cache and used there

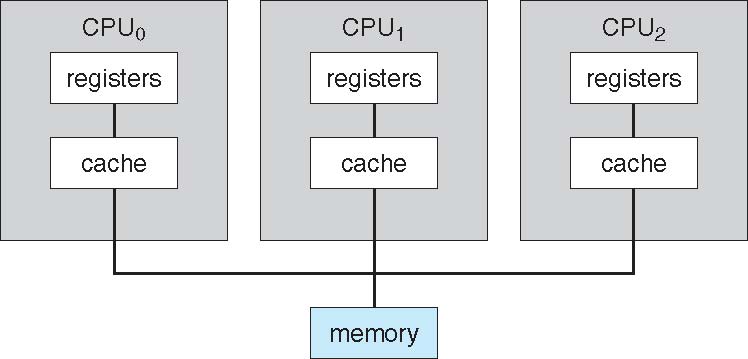
Storage-Device Hierarchy

Diagram

Description automatically generated

Multiprocessor Systems

* Increased throughput
* Economy of scale
* Increased reliabiltiy
  + Graceful degradation or fault tolerance
* Asymmetric Multiprocessing
  + Each processor is assigned a specific task
* Symmetric Multiprocessing
  + Each processor performs all tasks
* Multiple processors with their own registers and caches that all share one main memory



Clustered Systems

* Like multiprocessor systems, but multiple systems working together
* Sharing storage via a storage-area network (SAN)
* Provides a high-availability service which survivers failures
* Asymmetric clustering
  + Has one machien in hot-standby mode
* Symmetric clustering
  + Hass multiple nodes running applications, monitoring each other
* High-performance computing (HPC)
  + Applications must be written to use parallelization
* Multiple computers with their own systems all sharing on storage area network

1.08.pdf

Process

* Program in execution
* Unit of work within the system
* Program is passive entity, process is an active entity
* Needs resources to accomplish its task

MultiThread

* One program counter per thread

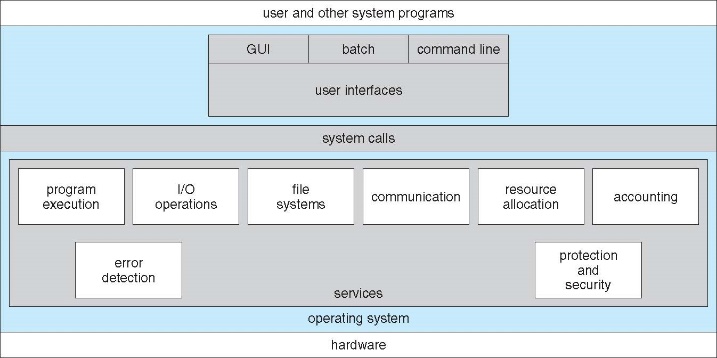
Chapter Two (Operating-System Structures)

User Interface

* GUI
* Command-Line Interpretor
* Batch
  + put command in file and run the file

Operating System Services

* User interface
* Program execution
* I/O operations
* File-system manipulation
* Communications
* Error dectection
* Accounting
* Protection and securit
* Resource allocation
* Protection and security



System Calls

* Needs to be included in a system program to be able to run
* Programming interface to services provided by the OS

System Program

* Runs system calls to interact with the operating system
* Parameters pushed onto the stack by the program and popped off the stack by the operating system
  + Does not limit the number or length of parameters being passed

Types of System Calls

* Process control
* File management
* Device management
* Information maintenance
* Communications
* Protection

Most common API’s

* Win32 API for Windows
* POSIX API for Linux and Mac OS
* Java API for the Java virtual machine (JVM)

User Goals

* Convenient for use
* Easy to learn
* Reliable
* Safe
* Fast

System Goals

* Easy to design
* Easy to implement and maintain
* Flexible
* Reliable
* Error-free
* Efficient

Mickrokernel

* Moves as much from the kernel into user space
* Communication takes place between user modules using message passing
* Easier to extend
* Easier to port the OS to new architectures
* More reliable
* More secure
* Performance overhead of user space to kernel space communication

Loadable Kernel Modules

* Object-oriented
* Each core component separate
* Each talks to the others over known interfaces
* Each is loadable as needed within the kernel

Core dump

* File capturing of the memory of a failing process
* Generated by a failed application

Performance Tuning

* Remove bottlenecks

# Chapter Three

Process Parts

* Text section
  + The program code
* Program counter
  + Processor registers
* Stack
  + Contains temporary data
    - Function parameters
    - Return addresses
    - Local variables
* Data section
  + Contains global variables
* Heap
  + Contains memory dynamically allocated during run time

Process state

* describing one thing
* Every state can have queues
* states
  + new
    - OS allocates memory for the program
  + ready
    - wait in the queue
    - ready to run
  + running
    - program running
    - can only be here for limited time before forced to go back to ready
  + terminate
    - release resources
  + waiting
    - program waiting
    - release CPU

Diagram

Description automatically generated

Process Control Block

* Information associated with each process
* Process state
  + New, running, waiting, etc.
* Program counter
  + Location of instruction to next executre
* CPU registers
  + Contents of all process-centric registers
* CPU scheduling information
  + Priorities
  + Scheduling queue pointers
* Memory
  + Management information
  + Memory allocated to the process
* I/O status information
  + I/O devices allocated to process
  + List of open files

Process scheduler

* Selects among available processes for next execution on CPU
* Maintains scheduling queues of processes
  + Job queue
    - set of all processes in the system
  + Ready queue
    - Set of all processes residing in main memory, ready and waiting to execute
  + Device queues
    - Set of processes waiting for an I/O device
* Processes migrate among the various queues

Short-Term Scheduler

* Selects which process should be executed next and allocates CPU
* Invoked frequently
* Sometimes the only scheduler in a system

Long-Term Scheduler

* Selects which processes should be brought into the ready queue
* Invoked infrequently (seconds, minutes)
* Controls degree of multiprogramming

Medium-Term Scheduler

* Can be added if degree of multiple programming needs to decrease
* Remove process from memory, store on disk, bring back in from disk to continue execution: swapping

I/O Bound Process

* Spends more time doing I/O than computations, many short CPU bursts

CPU-Bound Process

* Spends more time doing computations, few very long CPU bursts

Zombie

* Child process has no parent waiting (did not invoke wait())

Orphan

* Child process has the parent terminated without invoking wait()

Independent Process

* Cannot affect or be affected by the execution of another process

Coorperating Process

* Can affect or be affected by the execution of another process
* Information sharing
* Computation speed-up
* Modularity
* Convenience

Undbounded Buffer

* Places no practical limit on size of the buffer

Bounded Buffer

* Assumes that there is a fixed buffer size
* We like bounded buffers

Shared Memory

* An area of memory shared among the processes that wish to communicate
* The communication is under the control of the users processes not the operating system
* Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

3_12.pdf

Socket

* An endpoint for communication
* Concatenation of IP address and port

Port

* A number included at start of message packet to differentiate network servieces on a host
* xxx.xx.xx.x:yyy, the :yyy is the port number

# Chapter Four (Threads)

Parallelism

* Implies a system can perform more than on task simultaneously

Concurrency

* Supports more than one task making progress
* Single processor / core, scheduler providing concurrency

Data Parallelism

* Distributes subsets of the same data across multiple cores, same operation on each

Task Parallelism

* Distributing threads across cores, each thread performing unique operations

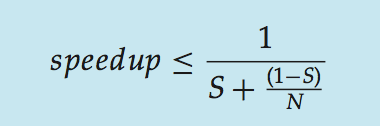
Mulitcore Programming

* CPUs have corse as well as hardware threads
* Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Single and Multithreaded Processes

4_01.pdf

Amdahl’s Law

* Idetifies performance gains from adding additional cores to an application that has both serial and parallel components
* S is serial portion
* N processing cores
* 
* That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
* As N approaches infinity, speedup approaches 1 / S

Three Primary Thread Libraries

* POSIX Pthreads
* Windows threads
* Java threads

Multithreading Models

* Many-to-One
  + Many user-level threads mapped to a single kernel thread
  + One thread blocking causes all to block
  + Multiple threads may not run parallel on multicore system beacause only one may be in kernel at a time
  + Few systems use this model
* One-to-One
  + Each user-level thread maps to kernel thread
  + Creating a user-level thread creates a kernel thread
  + More concurrency than many-to-one
  + Number of threads per process sometimes restricted due to overhead
  + Examples
    - Windows
    - Linux
* Many-to-Many
  + Allows many user level threads to be mapped to many kernel threads
  + Allows the operating system to create a sufficient number of kernel threads

Thread Libraries

* Provides programmer with API for creating and managing threads

Pthreads

* Specifies how to use threads
* Specification not implementation
* API specifies behavior of the thread library, implementation is up to development of the library

Implicit Threading

* Creation and management of threads done by compilers and run-time libraries rather than programmers

Thread Methods

* Thread Pools
* OpenMP
* Grand Central Dispatch

Thread Pools

* Create a number of threads in a pool where they await work
* Usually faster to service a request with an existing thread than create a new thread
* Allows the nmber of threads in the application to be bound to the size of the pool
* Separating task to be performed from mechanics of creating task allows different strategies for running task

OpenMP

* Set of compiler directives and an a API for C, C++, FORTRAN
* Provides support for parallel programming in shared-memory environments
* Identifies parallel regions – blocks of code that can run in parallel

Gradn Central Dispatch

* Manages most of the details of threading
* Allows identification of parallel sections
* Block is in “^{ }” - ˆ{ printf("I am a block"); }
* Blocks placed in dispatch queue

Serial Dispatch Queues

* Blocks removed in FIFO order, queue is per process, called main queue
* Programmers can create additional serial queues within program

Concurrent Dispatch Queues

* Removed in FIFO order but several may be removed at a time
* Three system wide queues with priorities low, default, high

Signal Handling

* Signals are used in UNIX systems to notify a process that a particular event has occurred
* Is used to process signals
* Has a default handler that kernel runs when handling signal
* User-defined signal handler can override default handler

Thread-Local Storage

* Allows each thread to have its own copy of data
* Useful when you do not have control over the thread creation process
* Unique to each thread

# Chapter Five (Process Synchronization)

Producer

* Table

  Description automatically generated

Consumer

* Text

  Description automatically generated

Race Condition

* Graphical user interface, text, application

  Description automatically generated

Critical Section Problem

* Each process has a critcal section segment of code
  + Process may be changin common variables, updating table, writing file, etc
  + When one process in critical section, no other may be in its critical section
* Critical section problem is to design protocol to solve this
* Each process must ask permisssion to enter section in entry section, may follow critical section with exit section, then remainder section

Critical Section

* Graphical user interface, text, application

  Description automatically generated

Algorithm for Process Pi

* Graphical user interface, text, application

  Description automatically generated

Solution to Critical-Section Problem

* Mutual Exclusion
  + If process Pi is executing in its critical section, then on other processes can be executing in their critical sections
* Progress
  + If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefiitely
* Bounded Waiting
  + A bound must exist on the number of times that other processes are allowed to enter their critical sectinos after a process has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
    - No assumumption concerning relative speed of the n processes

Peterson’s Solution

* Assume that load and store machne-language instructions are atomic; that is, cannot be interrupted
* The two processes share two variables
  + int turn;
  + Boolean flag[2]
* The variable turn indicates whose turn it is to enter the critical section
* The flag array is usedto indicate if a process is ready to enter the critical section flag[i[ = true implies that process Pi is ready
* Table

  Description automatically generated

Atomic

* Non-Interruptible
* Done in hardware

Simplifed Solution to Critical-Section Problem

* Graphical user interface, text, application

  Description automatically generated

Test and Set Instruction

* Text, application, email

  Description automatically generated

Test and Set Solution

* Text

  Description automatically generated with medium confidence

Compare and Swap Instruction

* Graphical user interface, text, application, email

  Description automatically generated

Compare and Swap Solution

* Text

  Description automatically generated with medium confidence

Mutex Locks

* Software tool to solve critical section problem
* mutex lock
* protect a critical section by first acquire() a lock then release() the lock
* requires busy waiting
  + called spinlock
* Graphical user interface, text, application

  Description automatically generated

Semaphore

* Synchronization tool that provides more sophisticated ways for process to synchronize their activites
* Accessed via two atomic operations
  + wait() and signal()
* Text, letter

  Description automatically generated

Sempaphore Usage

* Counting semaphore
  + integer value can range over an unrestricted area
* Binary semaphore
  + integer value can range only between 0 and 1
  + same as mutex lock
* Graphical user interface, text, application, email

  Description automatically generated

Semaphore Implementation

* No two processes can execute wait() and signal() on the same semaphore at the same time
* Block
  + place the process invoking the operation on the appropriate wating queue
* Wakeup
  + Remove one of processes in the waiting queue and place it in ready queue
* Text, letter

  Description automatically generated

Deadlock

* Two or more processes are waiting indefinitely for an even that can be caused by only one of the waiting processes
* Table

  Description automatically generated with medium confidence

Starvation

* Indefinite blocking
* A process may never be removed from the semaphore quese in which it is suspended

Priority Inversion

* Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  + solved via priority0inheritance protocol

Bounded-Buffer Problem

* n buffers, each can hold on item
* Semaphore mutex initialized to value 1
* Semaphore full initialized to the value 0
* Semaphore empty initialized to the value n

Readers-Writers Problem

* A data set is shared among a number of concurrent processes
  + Readers only read the data set; they do not perform any updates
  + Writers can both read and write
* Problem
  + allow multiple readers to read at the same time
  + Only one single writer can access the shared data at the same time
* Shared data
  + Data set
  + Semaphore rw\_mutex initialized to 1
  + Semaphore mutex initialized to 1
  + Integer read\_count initialized to 0

Dining-Philosophers Problem

* A screenshot of a computer

  Description automatically generated with low confidence

# Chapter Six

First Come First Serve

* Non-Premeptive
* Do not sort

Shortest Job First

* Sort
* Only sort ready queue
  + Do not kick process out of CPU

Shortest Remaining Job First

* Always take the shorter option
* Sort CPU and ready queue
  + Kick process out of CPU if necessary
* Premeptive

Round Robin

* Has a quantum
  + Have limited time to complete burst before the process has to leave the CPU
* Either jump to quantum or jump to arrival time or remaining burst time, whichever is smaller
  + Never jump larger than quantum time
* No need to sort

Premptive Algorithm

* If a process has started let it finish

Quantum Sizes

* To small
  + Processes are in waiting queue for long time
  + Very inefficient
* To big
  + Will become FCFS

Gantt Chart

* Jump the clock, do not tick
* Always take the sortest

# Chapter Seven

Deadlock Conditions

* All must be active to cause Deadlock
  + Mutual exclusion
  + Hold and wait
    - Every process must be requesting a resource while already being assigned a resource
  + No preemption
  + Circular wait
    - A subset of the processes can form deadlock, not every process needs be involved
    - Cycle in resource-allocation graph
* If graph contains no cycles → no deadlock
* If graph contains a cycle →
  + If only one instance per reousrce type, then deadlock
  + If several instance per resource type, possibility of deadlock

Resource-Allocation Graph

* Process = circle
* Resource = rectangle
  + Dots inside rectangle = instances
* Deadlock
  + There is a cycle

Request Edge

* a process is requesting a resource

Assignment Edge

* Resource is assigned to a process

Bankers Algorithm

# Chapter Nine

Use of a stack to record most recent page references

* Have a reference string
* Have a stack
* The bottom of the page is the least recently used in the reference string
* Traverse the reference string
* Push each number on the string onto the stack, when you come across the duplicate remove the first instance from the stack before pushing the new instance onto the stack

Reference bit

* on if the page is being referenced

Modify bit

* is on if the page has been modified

Valid/Invalid bit

* on if the page is in main memory

Best Victim Criteria

* Reference bit off and not modified (0,0) – Ideal #1
* Reference bit off and modified (0,1) – Good but must be saved before kicking out #2/#3
* Reference bit on and modify bit off – Good bu currently being used #2/#3
* Refernece bit on and modify bit on – Bad currently being used and must be saved #4

Thrashing

* Degree of multiprogramming
* CPU utilization
* At a certain point the CPU utilzation decreases significantly
  + eventually each process will only get one frame
  + lots of page fault
* The amount of programs you can have in the main memory to maximize the CPU utilization without causing it to drop significantly

Locality memory-reference pattern

* data from many users to determine which pages are referenced most often and when
* not perfect each user is different
* can be used to determine what pages to load ahead of time

Mass Storage

* Many new devices coming
* Currently used device
  + Moving-head disk mechanism
  + Has several plates
    - Cylinder that rotates
      * has tracks
        + has sectors
        + hard disk composed of sectors
  + Arms with read-write head
  + Arm assembly
  + Hard-drive

Disk Scheduling

Shortest SeekTime First

* run processes with the leas distance between them
* can lead to starvation

Scan

* split the queue into two sections
* run all of section one and then run all of section two
* travel time longer than SSTF but improves fairness

C-Scan

* Scan from one direction only
* Do multiple passes
* is more fair than normal scan

C-Look

* don’t go to the edges

RAID Levels

* Redundent array of independent disks
* When you create a database you want to have some redundency correction stuff
* 0 – no redundency
* 1 – copy of all the data
  + cost is double
* 2 – Include parity bit

# Chapter Eleven (File System Interface)

File Attributes

* Name
* Identifier
* Type
* Location
* Size
* Protections
* Time, data, user identification

File Operations

* File is an abstract data type
  + just data on a hard disk
  + 0s and 1s
  + difference between images and text are just the file attributes, or how the 0s and 1s are interpreted
* Create
* Write
* Read
* Repostion within a file
* Delete
* Tuncate
* Open
* Close

File Extensions

* system reads file extensions to know how to interpret the 0s and 1s

File Structure

* None
  + Sequence of words, bytes
* Simple record strucure
* Complex Structures

Access Methods

* Sequential Access
  + Read next
  + write next
  + reset
* Direct Access
  + file is fixed length
  + read n
  + write n

# Chapter Twelve

Layered File System

* Application programs

Logical file system

File-organization module

basic file system

I/O control

Devices

File Systems Layers

* Device drivers
* Basic fiel system
* File organization module
  + Handles translation from logical to physical
  + User doesn’t need to know where things are physically stored
* Logical file system
* Information node (inode)
  + FCB (File Control Block)
  + Hold data

Virtual File Systems

* More convienent for the user
* Store files on the internet
* User doesn’t care where the data is stored

Virtual File System Interface

* connects all of the file systems that the user uses
* unifies them so that the user doesn’t have to know where data is stored

In-Memory File system structures

* Need to have directory structure on your hard disk
* Need FCB for each file
* File structure related to FCB
* In the kernel you need to know about the file structure to operate
* Read information and hold a data structure to operate on

Contiguous Allocation

* When putting data on hard disk you can put in in a contiguous way
* <file name, start, length>

FAT

* File Allocation Table
* FAT table
* <file name, start, end>
* does not need to continuously stored
* Each block has information about where the next block is located
* Slower to search but less memory wasted

Indexed Allocation

* <file name, index block>
* index block has information about each block
* can have several level fo index to speed up the search

# Final Exam Study

Contiguous, FAT, Index Strategy

* advantages and disadvantages

Interrupt Vector

* reduce the need for a single interrupt
* make it easy to find the correct location to handle the interrupt
* is a table

Linux stuff

* (-): switch
* (ls): list
* (Lighter green or x): executable
* (rw): read write
* (./): run something
* (-u): force
* (-m): send message
* (man \_\_\_): gives you a description of a command
* cd: change directory
* cd /: root folder
* cd bin: has list of commands
* (.): current directory
* (..): parent directory
* mkdir: makes a dir
* echo: either display something to the terminal or to a file
  + ‘content’ > file/content
* mv:
  + can be used to move to same location and change the file name
* (-rf): recursive force
* rm: delete a file
* gcc “*name\_program\_to\_compile.c”* -o “*name\_of\_executable”*
* chmod
  + Change mode
  + u+x
    - user + executable
* gdb
  + debugger
  + upload main.cc Makefile
* pwd
  + print name of current/working directory
* date
  + print or set the system date and time
* >
  + used to redirect output to another file
* screen
  + create a second window that you can toggle between
  + Ctrl-a, Ctrl-c
    - copy current screen
  + Ctrl-a, Ctrl-a
    - switch between windows
  + Ctrl-a, Ctrl-d
    - detatch window
* downloading file from linux to windows
  + scp J\_123456@cs3060.tc.uvu.edu:homework/assn1.tar.gz .
* Zip a file
  + tar -czf assn1.tar.gz assn1

# C/C++ stuff

* cin
  + only reads up to: the first whitespace
  + overflow output results in: a random number
  + can be used to read an integer into an integer value
* #
  + directive
  + use code that has already been written
* .h
  + header file
  + can define function header/API/interfaces
  + “*header*“
    - one you made
  + <header >
    - standard library, already made
  + Default include
    - #include <stdio.h>
    - #include <stdlib.h>
* main
  + function
  + driver
  + startinig point
  + usually used to call functions and API
* argv
  + value
  + array main argument
* argc
  + count
  + size of argv array
* \*\*
  + pointer to a pointer
  + same as \**variable*[]
* \*
  + pointer
  + can be used to access the value of a \*\*variable
* pointer
* int 0 counts as false
* printf
  + take two arguments
    - %\_ represents the variable type
    - what it is ou want to print
* :wq
  + exit VIM
* Array
  + Has a pointer pre-made pointing at the address of position 0
  + If you create a pointer to the Array then that variable will be pointing to the address of the array
  + You need to create a pointer of a pointer (\*\*) in order to access the individual items in the array by using the (\*) version of the pointer of a pointer variable

# Things to do for next time

# Important notes from class

Section Five – bounded-waiting mutual exclusion with test\_and\_set

* split up code into critical section and remainder section
  + split up critical section into entry section and exit section
* True = lock, False = unlock
* waiting[i] and key are shared/global variables
* test\_and\_set
  + keep testing lock until it turns false
* waiting[i] = I want to go to the critical section
* mod = come back to the start
* i and j are indexes in P
  + n-1 are total number of indexes in P
* Program sections

/\* Entry Section \*/

/\* Critical Section \*/

/\* Exit Section \*/

/\* Remainder Section \*/

Section Five – Mutex Locks

* High level languages can call atomic integers

Section Five - Semaphore

* S = the amount of instances of a resource available
  + S— whenever a resource is being used
  + S++ whenver a resource is done being used

Section Five – Waiting

* instead of busy waiting, just call the waiting system call