Operating systems review
Linux commands
- man -> Manual
- crt+c-> interrupts process
- crt+z -> suspends process from foreground to background
- fg -> takes last process you sent to the background
- bg -> runs process in the background
- grep -> searches
- kill -9 -> kills process must enter number
- ps -ef -> processes currently running
  - 2nd number is pid
- l -> pipes output of one command to input of another
- % <job number> -> enables job number as identifier
- exec() -> function that permanently change process
- more -> interface to limit to screen size choices
- -o -> names executable
- g++-> compiler
- make-> does the executable line compile for you
- nice->
- top-> brings system statistics interface

C - specifics
- C - i/o
  - printf("<put what you want on screen here>");
    - Prints to screen
  - fprintf(stderr, "<any string>");
    - Prints to file

Operating systems are designed to be convenient and efficient

1.1 what operating systems do
- Computers Divided into four components
  - Hardware
    - Provide basic computing resources for system
      - (CPU) central processing unit
      - Memory
      - (i/o) Input/output devices
  - Operating system
    - Controls hardware and coordinates its use among the various application programs for the various users
  - Application programs
    - Define ways in which resources are used to solve users computing problems
  - Users

1.1.1 user view
- User view affect how operating system are conceived and work
  - Single person experience
    - Operating system designed for ease of use, performance
    - Not designed for resource utilization
      - Resource utilization
        - How various hardware and software resources are shared
  - Mainframe/minicomputer
    - Multiple users accessing shared resources

1.1.2 system view
- Operating system as resource allocator
  - Manages resources
- Operating system as control program
  - Manages execution of user programs to prevent errors and improper use of computer

1.1.3 defining operating system
- Kernel running at all times
  - System programs associated with operating system but not part of kernel
  - Application programs not associated with operating system

1.2 computer-system organization

1.2.1 computer system organization
- Bootstrap program
  - Stored in ROM and used when computer is booted up
  - Must know how to load the operating system and how to start executing that system
  - Must locate kernel and load it into memory
- Once kernel loaded it can provide services
- Some services provided by system processes or system daemons that are loaded at boot time and run while the kernel is running

1.2.2 storage structure
- Main memory too small to store all needed programs and data permanently
- Main memory is a volatile storage device that loses its content when power is turned off

1.3.1 single processor systems
- One single general-purpose processor with multiple special purposed processors

1.3.2 multiprocessor systems
- Two or more general purposed processors in close communication
  - Three main advantages
    - Increased throughput
      - More work is done in less time
    - Economy of scales
      - Multiple multiprocessor systems cost less than equivalent single processor systems
    - Increased reliability
      - Functions can be distributed among several processors allowing for cpu failures to not cripple the device
- Asymmetric multiprocessing
  - Each processor assigned a specific task
  - Boss processor controls the system
- Symmetric multiprocessing
  - Each processor performs all tasks including operating system functions and user processes.
- Multiprocessing can cause system to change its memory access model from
  - Uniform memory access
    - Access to any RAM from any CPU takes the same amount of time
  - Non-uniform memory access
    - Some parts of memory may take longer to access than other parts creating a performance penalty.

1.3.3 clustered system
- Composed of two or more individual systems joined together
- loosely coupled
- Connected by LAN
- Asymmetric clustering
  - One machine is in
    - Hot-standby mode
      - Machine does nothing but monitor active server, if that server fails the hot standby host becomes the active server
  - While others run the application
- Symmetric clustering
  - Two or more hosts are running applications and are monitoring each other

Chapter 3 processes

3.2 process scheduling
- Process scheduler selects an available process for program execution on the CPU

3.2.1 scheduling queues
- Job queues
  - Queue that process are put when they enter the system
- Ready queue
  - Processes in main memory that are ready and waiting to be executed
  - Generally a linked list
  - Ready queue header contains pointers to first and last PCB in the list
    - Each PCB points to the next PCB
- Device queue
  - List of processes waiting for a particular i/o device
- Queueing diagram
  - Representation of a processes scheduling
    - Process in Ready queue dispatched to the CPU
      - ProcessTerminates
        - Process PCB and Resources deallocated
      - Process forks and waits for child termination
      - Process waits for an interrupt to force it back into ready queue
      - Process places i/o request and is sent to i/o queue
        - Process waits in i/o queue until i/o device ready

a) Message passing and shared memory
- They are interprocess communication, used to exchange information between processes. Message passing is when processes pass each other messages to communicate information. This requires at least two operations, a send(message) and a receive(message). Shared memory is when two cooperating processes have a region of memory that is accessible by both processes. They share information by writing or reading info to that shared region of memory.

b) Synchronous and asynchronous communication
- They are two methods of communicating messages in the message passing system. They are also known as blocking or nonblocking. In synchronous send the sending process is blocked until the receiving process or the receiving mailbox receives the message. In synchronous receive the process is blocked until it receives a message. If both process are employing matching synchronous send and receive then it is known as a rendezvous. In asynchronous send the sending process sends the message and resumes whatever it was doing. In asynchronous receive the receiving process either gets a message or it doesn't, regardless it moves on with its life. Process can mix and match techniques, so a synchronous send can send a message to a asynchronous receive process.

c) Direct and indirect message passing

They are the way that message passing processes refer to each other. Direct message passing is when the process sends messages directly to the process and receives messages directly from other messages with no intermediaries. They establish a single link between each other and there is always only one link between two processes. In indirect messaging the processes send messages to the mailbox or port of its cooperative process. The receiving process will then not receive the message directly but check its mailbox or port for messages. Multiple links to from multiple process may link to a mailbox but two process will only be able to share information if they are both linked to the mailbox.

d) Bounded and zero capacity buffering

They are two types of temporary queues that exist to facilitate message passing. A bounded capacity buffer is a queue with a finite length. It can only receive as many messages as it has space for. Once its capacity is reached the sender must block and stop sending messages and wait for the receiver to empty at least one message from the buffer. A zero capacity buffer has a length of zero and therefor cannot store messages. The sender must block after every message until the receiver has received the message.

e) Fixed-size and variable-size messages

A fixed size message is fixed when the message is always the same size, filled with filler when lacking and split into multiple messages when too large. Variable size messages is when the message can be of a variable size, thus the message must store extra overhead that tells the receiver how large the message is.

f) Local procedure call and remote procedure call

A local procedure call is a type of message passing between processes on the same system that uses ports to establish and maintain communication channels. The client opens a handle to the subsystems connection port. Then the client sends a connection request. The server responds to this by creating two private communication ports and returning the handle of one of them to the client. The client and server use the corresponding port handles to send messages and listen for replies. A remote procedure call is relatively similar but used on networked systems. Basically this allows a process on one system to interact with a server on another system as if it was on the same system. It does this by providing a stub on the client side. This stub packages or marshals the parameters into a form that can be transmitted over the network. There is a similar stub on the server side of the network. The stubs act as intermediaries that resolve the difference for the client and server such that the client and server can interact just as if they would locally through the stub and never know the difference.

3.2.3 schedulers

Scheduler

Selects processes

Long term scheduler (job scheduler)

Selects processes from a pool of backed up processes and loads them into memory for execution

- Less frequent execution once every couple minutes
- Controls degree of multiprogramming
  - Number of processors in memory
  - If degree of multiprogramming is stable than average rate of process creation must be equal to the average departure rate of processes leaving the system
  - Invoked when process leaves the system
    - i/o bound processes
      - Spends most of its time doing i/o
    - CPU bound process
      - Generates i/o infrequently does lots of computations
  - Must select a good mix of both so CPU can be used to full advantage
- Short term scheduler (CPU scheduler)
  - Selects from the ready queue processes that are ready to execute and allocates the CPU to one of them
  - Frequent execution once every 100 milisecond
- Medium term scheduler
  - Swapping
    - Removes a process from memory to reduce degree of multiprogramming and later reintroduces it

3.2.3 context switch

- When an interrupt occurs the OP switches the Operating system to change the CPU from its current task to run a kernel routine
- System must save the context of the process running on the CPU
- Context represented in PCB of process
- State save saves PCB
- State restore resumes
- Context switch when processes are switched

Chapter 4 threads

4.1

- Thread is basic unit of CPU utilization
  - Individual thread properties
    - Thread ID
    - Program counter
    - Register set
    - Stack
  - Common threads to a process properties
    - Code section
    - Data section
    - Operating system resources , open files signals ect...
- Traditional (heavyweight) process
  - Single thread of control
- If process has multiple threads of control it can do multiple things at once

4.1.1 motivation

- Process creation method of multicore programming
  - Time consuming
  - Resource intensive
- More efficient to use multithreaded since they do the same thing
- Critical to (RPC) remote procedure call
  - Server receives message it serves message using a separate thread
  - Allows to service concurrent requests
- Most OS kernels are multithreaded
  - Each thread in kernel operates a different task
    - Managing devices
    - Managing memory
    - Interrupt handling
  - Linux uses a kernel thread for managing the amount of free memory in the system

4.1.2 benefits

- Responsiveness
  - Allows a program to keep running even if part of it is blocked or performing a lengthy operation
  - Useful for UI
- Resource sharing
  - Threads share memory and resources of the process, thus no interprocess communication and applications can have different threads of activity within the same address space
- Economy
  - Allocating memory and resources is costly, simpler to create and context switch threads
  - Creating process much slower, than creating thread
- Scalability
  - Multithreading in multiprocessor architecture awesome because threads can be running in parallel on different processor cores

4.2 multicore programming

- Concurrency
  - Supports more than one task by allowing all tasks to make progress
- Parallelism
  - Performs more than one task simultaneously
- Speedup <= 1/s+(1-s)/n

4.2.1 programming challenges

- Identifying tasks
  - Finding areas in applications that can be divided into separate, concurrent tasks
- Balance
  - Ensure that tasks perform equal work of equal value, a task that is no as worth as much to overall process should not be run in parallel but run concurrently
- Data splitting
  - Data must also be split to run on separate cores
- Data dependency
  - Data accessed by task must be examined for dependencies between tasks, if one task requires data from another they must be synced
- Testing and debugging
  - When programs are running in parallel much more difficult to debug

4.2.2 types of parallelism

- Data parallelism
  - Distributes subsets of the same data across multiple computing cores and performing the same operation on each core
- Task parallelism
  - Distributes tasks across multiple computing cores. Each thread performs a unique operation.

4.3 multithreading models
- User threads
  - Supported above the kernel and managed without kernel support
- Kernel threads
  - Supported and managed directly by operating system

4.3.1 many-to-one model
- Maps many user level threads to one kernel thread.
- Thread management is done by the thread library in user space
- Entire process blocks if a thread makes a blocking system call
- Only one thread can access the kernel at a time, not parallel

4.3.2 one-to-one model
- Maps each user thread to its own kernel thread
- Provides more concurrency than many to one by not having the blocking restriction
- Allows multiple threads to run in parallel
- Large overhead

4.3.3 many-to-many model
- Multiplexes many user threads to a smaller or equal number of kernel level threads
- Number of kernel threads can be specific to either a particular application to a particular machine
- None of the earlier problems
- Two-level model
  - Same as before but also allows a user thread to be bound to a single kernel thread.

4.4 thread libraries
- Provides an API for creating and managing threads
  - Provide a library entirely in user space with no kernel support
    - All code and data structures for the library exist in user space
    - Invoking a function in the library results in a local function call not a system call
  - Implement kernel level library supported directly by the operating system
    - Code and data structures exist in kernel space
    - Invoking a function in the API generally results in a system call to the kernel
- Three main thread libraries
  - POISIX Pthreads
    - Maybe either user level or kernel level
  - Windows
    - Kernel level
  - Java.Pthreads
    - Generally on host api
- On POSIX and windows
  - Global variables shared among all threads
- On Java
  - All data must be shared explicitly
  - Data declared local to a function are stored in a stack
- Asynchronous threading
  - Once parent creates child thread the parent continues, parent and child concurrent
  - Threads act independently of one another and parent need not know when child dies
  - Little data sharing
- Synchronous threading
  - Parent must wait for all children to terminate before it resumes execution
  - Called fork-join strategy
  - Children are concurrent but not parent with children
  - Once thread finished work it terminates and joins parent
  - Significant data sharing between threads

4.4.1 Pthreads

- POSIX standard defining an API for thread creation and management
- It is a specification for thread behavior not an implementation

4.5 thread pools

- Create a number of threads at process startup and place them in a pool
- When requested thread is awakened from thread pool
- If none available requeste waits
- Thread returned to pool after use

- Thread pool benefits
  - Servicing request with an existing thread is faster than creating a new one
  - A thread pool limits the number of threads that exist at any one point
  - Separating the tasks to be performed from the mechanics of creating the task allows us to use different strategies for running the task.

4.5.2 openMP

- Set of compiler directives as as an api for programs written in c, c++, fortran that provides support for parallel programming in shared memory environments
- Identifies parallel regions as blocks of code that may run in parallel
- When #pragman omp parallel encountered
  - Creates as many threads as there are processing cores in the system
  - All threads then simultaneously execute the parallel region

4.5.3 grand central dispatch

- Extension to c, c++, api and runtime libraries
- Parallel sections identified in blocks
- Represented by a ^infront of a pair of braces
- Scheduled on a dispatch queue
  - When it removes a block from queue it assigns the block to an available thread from the thread pool it manages.
  - two types of dispatch queues
    - Serial
      - Removed in FIFO order
      - Once removed must complete execution before another block is removed
      - Each process has its own serial queue (main queue)
    - Concurrent
      - Removed in FIFO order
      - Multiple blocks can be removed at a time
      - Allowing multiple blocks to run in parallel
      - Three system wide concurrent dispatch queues
        - Low priority
        - Default priority
        - High priority

4.6.1 the fork() and exec() system calls

- If exec is not called after fork
  - All threads thread fork
- If exec is called
  - Single thread fork

4.6.2 signal handling

- Signal is used to notify a process that a particular event has happened
- Maybe received either synchronously or asynchronously
- All signals follow same pattern
    - Signal generated for reason
    - Signal delivered to process
    - Signal must be handled
- Synchronous signal
    - Signal is sent to same process that generated it
- Asynchronous
    - Signal is sent to a different process than the one that generated it
- Signal may be handled by two possible handlers
    - Default signal handler
    - User defined signal handler
- Options for multithreaded signals
    - Deliver signal to the thread which the signal applies
    - Deliver signal to every thread in process
    - Deliver signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process

4.6.3 thread cancellation

- Terminating a thread before it has completed
- Thread to be canceled is target thread
    - Asynchronous cancellation
        - One thread immediately terminates the target thread
        - System may not reclaim all resources
    - Deferred cancellation
        - Target thread periodically checks whether it should terminate
        - Target self terminates safely, no resource lost
        - Only cancels when cancelation point reached
        - Cleanup handler invoked, releases all resources

4.6.4 thread-local storage

- Data that is local to the thread
- Similar to static data

4.6.5 scheduler activations
- Lightweight process
  - Data structure between user and kernel threads in many to many and two level
  - Attached to kernel thread
  - When kernel thread locks, LWP locks and so do the user threads its connected to
- Scheduler activation
  - Kernel provides application with set of virtual processors (lwp) and application can schedule user threads onto available virtual processor
  - Upcalls
    - Inform application about certain events
    - Handled by upcall handler
    - Run on LWP
    - When tread is about to block and identifying specific thread
      - Kernel allocates new LWP to to application
      - Application runs upcall handler on LWP
      - Saves state of blocking thread and relinquishes the LWP on which blocking thread is running
      - When event that is blocking thread waiting occurs kernel makes another upcall to the thread library informing it that the previously blocked thread is no eligible to run
      - Kernel either allocates new virtual handler(LWP) or takes one of the user threads and runs the upcall handler on its LWP
      - After marking the unblocked thread as eligible to run application schedules eligible thread to run on a LWP

Linux threads
- Fork() creates copy of parent
- Clone() points to parent

Sfjga'ng'asfng;l'asnfg'lkasng'lkasn'glnas'lkgnas'lkfng'lasng'lksang'anr'gklnasf'lkgnas'flkngl

Chapter 7 osc
Main memory

Base register
- Holds the smallest legal physical memory address

Limit register
- Specifies the size of the range

Input queue
- Processes waiting to be brought into memory for execution

Compile time
- You know at compile time where the process will reside thus absolute code can be generated, if changed must recompile

Load time
- Unknown where process will reside at compile time, thus relocatable code generated

Execution time
- Process can be moved during execution

Logical address
- Address generated by cpu
- Range 0 to max

Physical address
- Real address in memory as seen by memory unit
- Range R+0 to R+max | R = base value

Compile and load time methods generate identical logical and physical addresses
Execution time method generates a virtual address instead of physical address

MMU (memory management unit)
- Does the runtime mapping from virtual to physical addresses

Contiguous memory allocation
- First fit
  - Allocate the first hole that is big enough
  - Searching starts either at beginning of set of holes or at location where previous first-fit search ended
- Best fit
  - Allocates the smallest hole that is big enough
  - Search entire list, unless list is ordered by size
- Worst fit
  - Allocate the largest hole
  - Search entire list unless sorted by size
- When two holes meet they merge into larger hole

Fragmentation
- External fragmentation
  - Enough memory is available memory to satisfy a request but it is stuck between processes
  - Best fit and first fit do this
  - 50 percent rule
    - Up to One third of memory may be unusable cause of external fragmentation
- Internal fragmentation
  - Memory is broken up into fixed size blocks, memory allocated by block size such that a process may be allocated more memory than it needs
  - This is happens because some holes are so small the overhead to keep track of them are more expensive than the hole itself

- The difference between the block size and the size required by the process is the internal fragmentation
- Compaction
  - Solution to external fragmentation
  - Shuffles memory contents so that all free memory is in one large block
  - If relocation is static (assembly or load time) compaction cant be done

- Segmentation
  - Logical address consist of two tuple unit
    - <segment-number, offset>
      - Segment-number
        - Index to segment table
      - Offset
        - Is what the logical address use to be, the referenced location within the segment
      - Two dimensional logically, still mapped to one dimension physically by segment table
      - Segment table
        - Each entry has segment base and segment limit
        - Segment base
          - Contains the starting physical address where the segment resides
        - Segment limit
          - Specifies length of segment