**Chapter 1**

Four Components of a Computer System

* Computer Hardware
* Operating System
* System and Application Programs
* Users

Operating System is

* Resource Allocator - Decides between conflicting requests for efficient and fair resource use
* Control Program - controls execution of programs
* Interrupt Driven

Controller- When the controller finished reading and writing back to the memory, it generates an interrupt.

***DMA- Direct Memory Access , (pulling data from controller)***

instruct controller

shift to memory

cpu interrupts

Almost all controllers are DMA enabled

Also allows devices to write between each other

Used for high-speed I/O devices able to transmit information at close to memory speeds Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention Only one interrupt is generated per block, rather than the one interrupt per byte

***Kernel*** is the one program running at all times on the computer

Kernel occurs after the bios, only the drives in the operating system understand how to handle the devices. **Kernel will handle interrupts**

Process inside of Kernel is kept track by data structures. **Data structures in the kernel are called Process Control Block**. Kernel needs **PCBs to keep track of processes**.

In User Mode - You can only use a subset of hardware instructions. if you try to use a privileged instruction while in user mode, you will receive an error.

In privilege mode- access to hardware and kernel

Traps- Turns the processor into privilege mode

Storage devices is divided into the blocks for the kernel to perform it tasks.

***Bootstrap Program***

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

Computer Operation flow

We use bus and not a switch because the bus only allows one output at a time. Only one transfer at a time. Switch allows multiple devices to talk to each other.

* I/O devices and the CPU can execute concurrently
* CPU moves data from/to main memory to/from local buffers
* I/O is from the device to local buffer of controller
* Device controller informs CPU that it has finished its operation by causing an ***interrupt***

***Interrupts***

When interrupt occurs must save the current context. If another interrupt occurs interrupt priorities come into play. Only a limited amount of interrupts may occur. Processor can a start queue to keep track of interrupts. Queue must be limited also. When a queue becomes full you lose an interrupt. An interrupt should be handled very very fast. Interrupt handlers need to be very very small.

Software Interrupt - an instruction that causes an interrupt when it is executed, used to enter operating system, handling errors. Trap is a software interrupt.

Call vs Interrupt- Call is cheaper because you don't have to save all of the registers before executing an interrupt. Call needs some kind of protection because it can directly touch the hardware.

Processor- before the cpu services an interrupt, the cpu saves the context . Program Counter and registers allows you to start where you left off when you experience an interrupt. To service an interrupt by interrupt handling routine. You need to have an interrupt for each device.

*Interrupt Vector* - contains the addresses of all the service routines

*Trap/ Exception -*software-generated interrupt

Interrupt architecture must save the address of the interrupted instruction

***Interrupt Handling / Vectors***

Vector- Must have a vector in memory that has a pointer to each interrupt handler. Must initialize vector when you start, by the kernel

Vector has pointers to functions which are used as the interrupt handlers which are called when an interrupt occurs.

The operating system preserves the state of the CPU by storing registers and the program counter

***Memory***

Dram- dynamic ram, transistors used as capacitors.. continuously keep refreshing to allow the dram to keep memory, transistors leak

Sram- Costs more money, runs faster, does not need to be continuously refreshed

Flash- Can’t read and write as much to a regular disk, expensive, If you always read no problem, however writing to flash memory wears out

Cache- When you read and the item is already cached it is called a Cache hit.

Benefits of Virtual Memory; increase in memory, allows execution of processes not completely in memory

If you create a processes you must destroy/delete it to conserve memory.

***System Call -*** A system call is a call from user-level code into kernel-level code; it is usually implemented via a TRAP (software interrupt) under the control of the interrupt vector which is generally created by the OS during boot time.

\*fastest way to pass parameters into system call is through CPU registers

Main memory – only large storage media that the CPU can access directly

Random access Typically volatile

Secondary storage – extension of main memory that provides large nonvolatile storage capacity Hard disks – 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

***Five system calls used for process management***

* fork(): creates a copy of the invoking process.
* exit(): terminates the invoking process.
* wait(): called by a parent process to synchronize with the exit of a child process and collect its exit status.
* exec(): invoked by a process to change the code that it is currently executing.
* kill(): sends signals to a process.
* (Others: getpid(), getppid())

***Multiprocessors***

Advantages include:

1. Increased throughput

2. Economy of scale

3. Increased reliability – graceful degradation or fault tolerance

Two types:

1. Asymmetric Multiprocessing – each processor is assigned a specie task.

2. Symmetric Multiprocessing – each processor performs all tasks

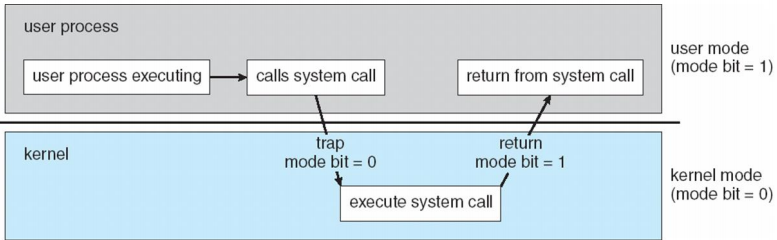
***Multiprogramming (Batch system) - needed for efficiency***

* Single user cannot keep CPU and I/O devices busy at all times
* Multiprogramming organizes jobs (code and data) so CPU always has one to execute
* A subset of total jobs in system is kept in memory
* One job selected and run via job scheduling
* When it has to wait (for I/O for example), OS switches to another job

***Timesharing (multitasking)*** - is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing

\*The operating system loads multiple programs in memory and switches the CPU among them in a frequent manner so as to give them their impression of having a dedicated CPU;

* Response time should be < 1 second
* Each user has at least one program executing in memory process
* If several jobs ready to run at the same time CPU scheduling
* If processes don’t fit in memory, swapping moves them in and out to run
* Virtual memory allows execution of processes not completely in memory



***Process Management***

* A **PROCESS** is a program in execution. It is a unit of work within the system. Program is a passive entity, process is an active entity.
* Process needs resources to accomplish its task; CPU, memory, I/O, files, and Initialization data

***Threading***

* Single-threaded process has one program counter specifying location of next instruction to execute
* Process executes instructions sequentially, one at a time, until completion
* Multi-threaded process has one program counter per thread
* Typically system has many processes, some user, some operating system running concurrently on one or more CPUs

***Memory Management***

* To execute a program all (or part) of the instructions must be in memory
* All (or part) of the data that is needed by the program must be in memory.
* Keeping track of which parts of memory are currently being used and by whom
* Allocating and deallocating memory space as needed

***I/O Subsystem***

* Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
* General device-driver interface
* Drivers for specific hardware devices

***Bitmap*** - – string of n binary digits representing the status of n items

***Computing Environments***

* Compute-server system provides an interface to client to request services (i.e., database)
* File-server system provides interface for clients to store and retrieve files

**Chapter 2**

PCB is in Kernel Memory

We pass the parameters through registers. Because of a lot of parameters we start using on the stack   
Parameters are pushed onto the stack by the program and popped off the stack by the operating system.  
Kernel could locate the memory and look for the parameters while passing.

\*\*Process, used to grab a program and execute the program in memory, resulting in the program becoming a process

**Operating System services**

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

**CLI** or command interpreter allows direct command entry-Shells

GUI-user friendly desktop--icons, programs, actions

GUI+CLI interface such as Mac OSX Aqua-Unix and Linux have CLI with optional GUI

**System Calls**

* Programming interface
* Written in C/C++
* Accessed by programs via high level API
* 3 common api-WIN32, POSIX, JAVA API

System call interface invokes the intended system call in the OS kernel and returns status of the system

Number associated with each system call

Most Details of OS hidden from programmer by API-Managed by run time support library-Set of functions built into compiler libraries

**Three General Methods for System Call Parameter Passing**

* Simplest
* Parameters stores
* Parameters placed
* Block and Stack methods

**\*\*Types of System Calls**

* Processes control
  + Create, terminate, load, execute, wait, allocate, dump error
* File Management
  + Create, delete, open, close, read, write, get, set file/attributes
* Device management
  + request , release, read, write, reposition
* Information maintenance
  + Get or set: time, system data, process file, device attributes
* Communications
  + Create, delete, send, receive, transfer status, attach and detach remote devices
* Protection
  + Control access to resources, get and set permissions, allow or deny users

**Ex-MS DOS**

* Single tasking
* Shell invoked on boot
* Single memory space
* Load program into memory overwriting all but the kernel

**Free BSD-Linux Variant**

* Multitakins
* Invoke user choice of shell
* Shell executed fork
* code=0-no error--code>0-error code

**System Program**s

* File manipulations
* Status information
* Programming language support
* Program loading and execution
* Communications
* Background services
  + services , subsystems, daemons
  + Daemons are background services, they run in the user space, daemon cuts communication with console with file scriptures
* Application programs
  + Don’t pertain to system, run by user, not part of OS

**Operating System Design and Implementation**

* Not Solvable
* Internal structure can vary between OS
* Deisng by definiting goals and specs
* Affected by choice of hardware
* User Goal
  + Operating system should be convenient to use easy to learn, reliable, safe and fast
  + System goals easy to design, implement, maintain, flexible, reliable, error-free and efficient
  + Policy: what will be done
  + Mechanism: how to do it
  + Separation of policy allows flexibility if policy decision are to be changed later

**Implementation**

* Old system assembly, new system C/C++
* Mix of languages
* High level easier to port to other hardware but is slower

Emulation allows os to be run on non-native hardware

**Operating system structure**

* Ms Dos-simple
  + Not divided into moduled-not will separated
* unix - complex-two parts
  + Systems programs
  + The kernel
    - Consists of everything below the system-call interface
    - Provides the file system, mem managment-large number of functions for one level
* Layered-an abstractions
* Mircokernal-Mach

**Microkerel**

* Moves kernel into user user space
* Message passing
* Easier to extend, port, more reliable and secure
* Performance overhead of user space into kernel space communication

**Modules**

* Loadable kernel modules
* Uses OOP approach
* Each core is separate, talks over interface, loadable within kernel

**Hybrid Systems**

* Modern OS are not one model
* Apple OSX hybrid layered Aqua UI plus COCAO programming environment

**Loadable models==Kernal Extensions**

**Operating System Debugging**

* Finding and fixing bugs
* OS generate log files
* Failed application-core dump
* OS failure-crash dump
* Trace listing of activities, recording of analysis
* Profiling periodic sampling of instruction pointer to look for trends

Kernighan’s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”  
Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site

SYSGEN program obtains information concerning the specific configuration of the hardware system-Used to build system-specific compiled kernel or system-tuned-Can general more efficient code than one general kernel

**System Boot**

* Power initialized leads to execution starts at a fixed memory location
* Firmware ROM used to hold initial boot code
* Operating system must be made available to hardware
* bootstrap loader-Small piece of code run on startup stored in ROM or EEPROM also locates the kernel, loads it into memory, and starts it
* Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk
* Common bootstrap loader, GRUB, allows selection of kernel from multiple disks,versions, kernel options
* Kernel loads and system is then running

**Chapter 3**

***Process***

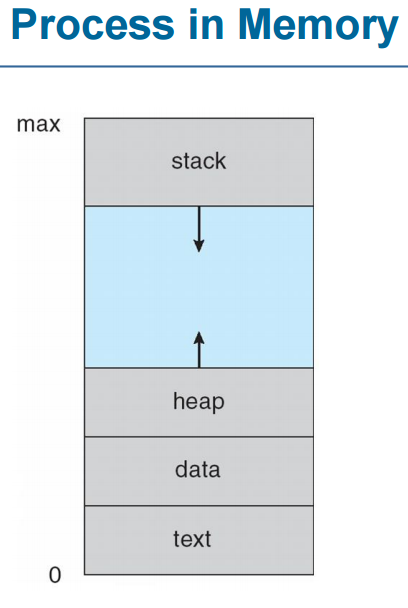
- a program in execution; process execution must progress in sequential fashion

-Stack containing temporary data Function parameters, return addresses, local variables

-Data section containing global variables

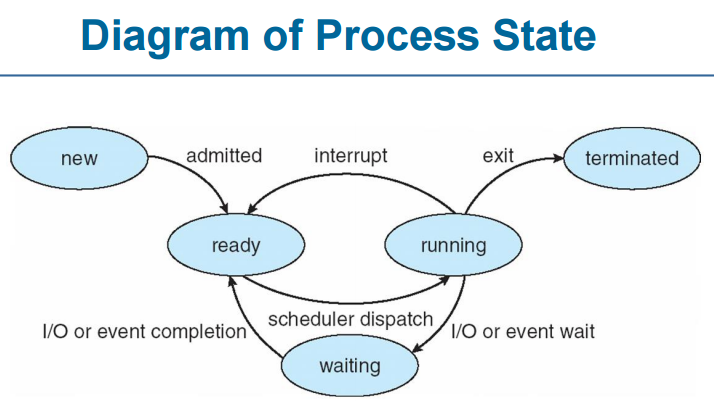
-Heap containing memory dynamically allocated during run time

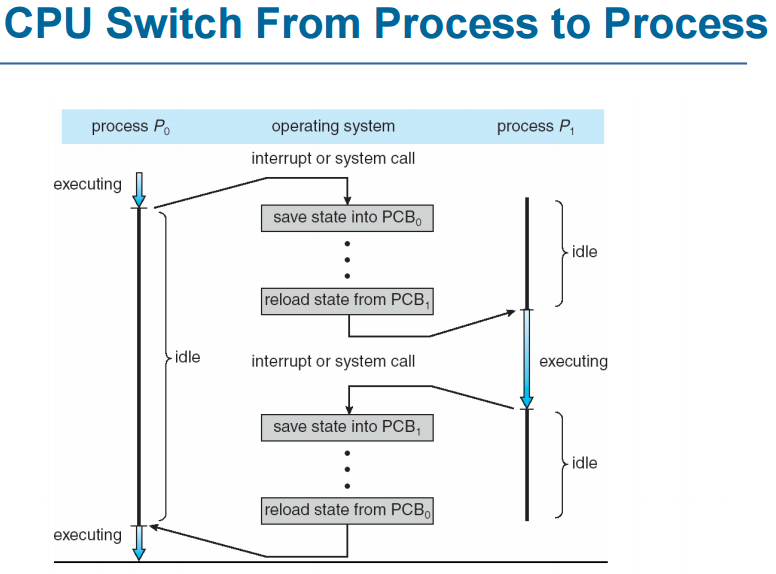
Program is passive entity stored on disk (executable file), process is active



***Process State***

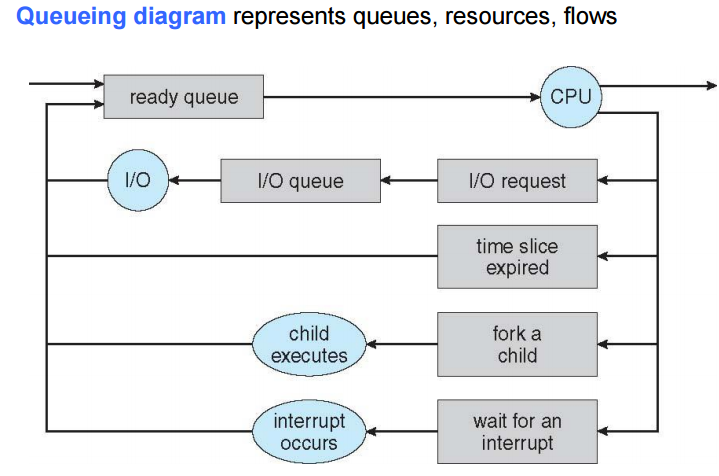
* As a process executes, it changes state
* new: The process is being created
* running: Instructions are being executed
* waiting: The process is waiting for some event to occur
* ready: The process is waiting to be assigned to a processor
* terminated: The process has finished execution





***Process Scheduling***

* 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



***Schedulers***

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

Medium-term scheduler can be added if degree of multiple programming needs to decrease

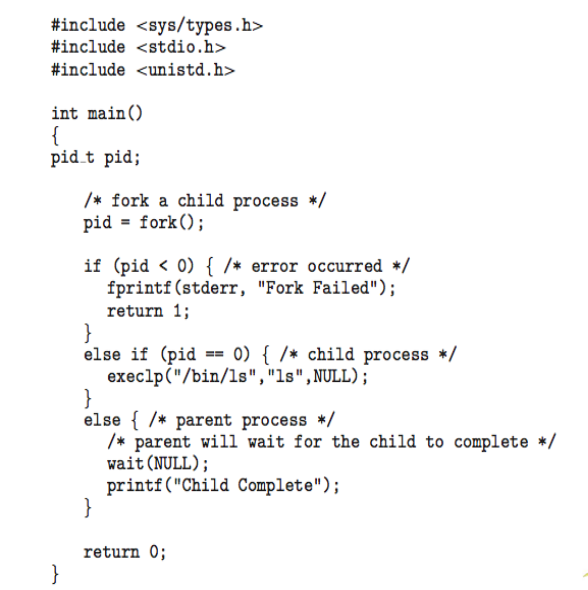
Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

***Context Switch***

* When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
* Context of a process represented in the PCB
* Context-switch time is overhead; the system does no useful work while switching

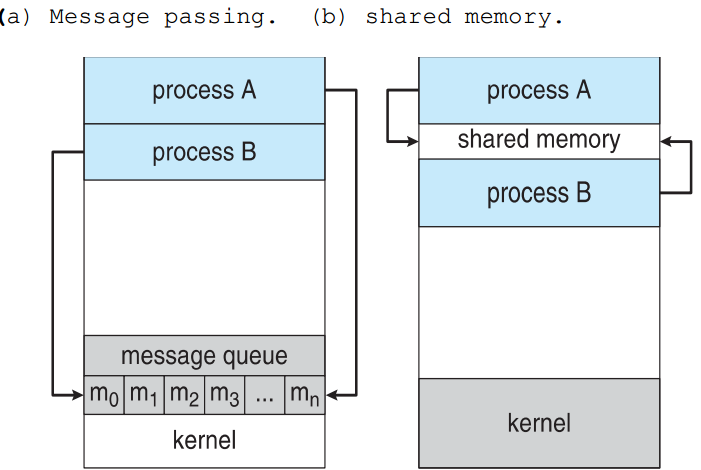
***Process Creation***

* Parent process create children processes, which, in turn create other processes, forming a tree of processes
* Generally, process identified and managed via a process identifier (pid)
* Resource sharing options Parent and children share all resources Children share subset of parent’ s resources Parent and child share no resources
* Execution options Parent and children execute concurrently Parent waits until children terminate



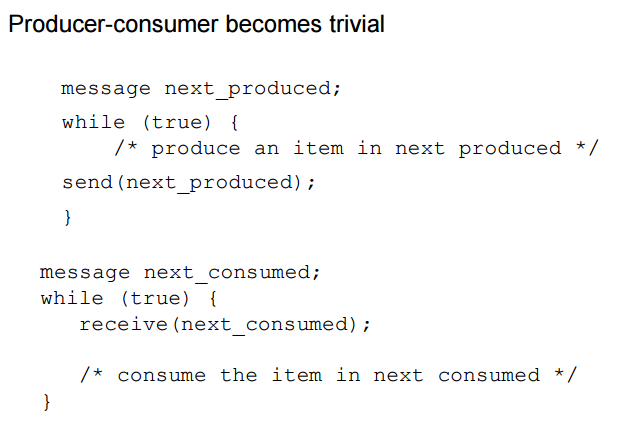
***Process Termination***

* If no parent waiting (did not invoke wait()) process is a zombie
* If parent terminated without invoking wait , process is an orphan
* parent[i]=wait(NULL); // Kills child zombie process
* parent[i]=fork(); // Creates a child process



***Synchronization***

* Blocking is considered synchronous Blocking send -- the sender is blocked until the message is received Blocking receive -- the receiver is blocked until a message is available
* Non-blocking is considered asynchronous Non-blocking send -- the sender sends the message and continue Non-blocking receive -- the receiver receives: A valid message, or Null message
* If both send and receive are blocking, we have a ***rendezvous***



Send and receive are flexible, for example four options if mailbox full: Wait indefinitely Wait at most n milliseconds Return immediately Temporarily cache a message

***Sockets***

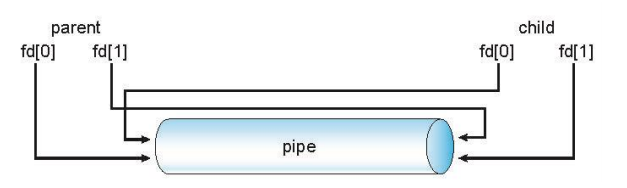
* A socket is defined as an endpoint for communication
* The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
* Three types of sockets Connection-oriented (TCP) Connectionless (UDP) MulticastSocket class– data can be sent to multiple recipients

***Pipes***

Acts as a conduit allowing two processes to communicate

*Ordinary Pipes*

* Unidirectional
* Ordinary Pipes allow communication in standard producer-consumer style Producer writes to one end (the write-end of the pipe) Consumer reads from the other end (the read-end of the pipe)
* Require parent-child relationship between communicating processes



*Named Pipes*

Named Pipes are more powerful than ordinary pipes Communication is bidirectional No parent-child relationship is necessary between the communicating processes Several processes can use the named pipe for communication Provided on both UNIX and Windows systems

**Chapter 4**

Benefits

* Responsiveness
* Resource Sharing
* Economy
* Scalability

Multicore/Multiprocessor

* Dividing Activities
* Balance
* Data splitting
* Data dependency
* Test and debugging

Parallelism- implies a system can perform more than one task simultaneously

* Data-distributes subsets of the same data across cores, same operation on each
* Task-distributing threads across

Concurrency- supports more than one task making progress

More threads means more architectural support for threading

CPU’s have cores and well as hardware threads

Amdahl’s Law-identifies performance gains from adding additional cores

User threads-managment done by user level library

* POSIX Pthreads
* Windows threads
* Java threads

Kernel threads

* Linux, windows, mac osx

Models

Many To One

* Many user threads to single kernel thread
* Multiple threads may not run in parallel
* GNU, Solaris

One to one

* Each user level thread map to kernel thread. Each user level

Many to Many

* Allows many user level threads to be mapped to many kernel threads
* windows

Two level model= allows user thread to be bound to kernel thread

Thread library-API and managing threads

* Library entirely in user space
* Kernel level library supported by OS

Pthreads-provided by user level or kernel level-API-specification not implementation

* Common in UNIX OS

Java Threads

* Managed by JVM
* Implemented using the threads model

Implicit Threading

* Increase number of threads
* Creation and management of threads done by compiler

Three Implicit Methods

* Thread Pools
  + Create number of threads in a pool where they await work
  + Slightly faster to service
  + Windows api
* OpenMP
  + Support for parallel programing
  + Shared memory environment
* Grand Central Dispatch
  + Block in dispatch queue
  + Apple tech max osx
  + Allows parallel sections
* Two types of GCD
  + Serial- clock removed in FIFO order,queue per process, main queue
  + Concurrent- removed in FIFO order, can be removed many at a time, three priorities low, default, high

Issues

* Schematics of fork and exec
  + Does fork duplication
  + Exec works normal
* Signal handling
* A signal handler is used to process signals  
  1. Signal is generated by particular event  
  2. Signal is delivered to a process  
  3. Signal is handled by one of two signal handlers:  
  1. default  
  2. user-defined
* Every signal has default handler that kernel runs when handling signal
* User-defined signal handler can override default
* For single-threaded, signal delivered to process

Where should a signal be delivered for multi-threaded? Deliver the signal to the thread to which the signal applies Deliver the signal to every thread in the process Deliver the signal to certain threads in the process Assign a specific thread to receive all signals for the process

* Thread cancellation of target thread
  + Asynchronous or deferred
  + Terminating before it has finished
  + Thread to be canceled before it has finished
  + Asynchronous
    - Terminates the target thread immediately
  + Deferred cancelled
    - Allows the target thread to periodically check if it should be cancelled
  + Cancellation point/cleanup handler
* Thread local storage
  + TLS thread local storage allows thread to have its own copy of data
* Scheduler activations
  + LWP lightweight process
  + Upcalls
  + Upcall handler

**Diagrams**

