Week 1

1. Microkernel and Momentics

Momentics is run on a “development host”. There are 3 choices of development hosts: Linux, Windows and Mac OS.

Neutrino is a microkernel architecture

2. RTOS

What is an RTOS? ...
l A real-time operating system (RTOS) is an operating system (OS) intended to serve real- time application requests.

3. Monolithic Kernel v/s Microkernels

Monolithic Kernels vs Microkernels
In a monolithic kernel architecture, the entire OS and it's services run in the same process thread, and share the same memory space.

In a microkernel, only the bare minimum of the OS is in the kernel process. The rest, including device drivers and file access managers reside in external servers that run in the user space.

Microkernel Advantages
Easier to maintain
Better persistence – if an instance of a server is corrupted, another can be swapped into place.

Week 2

When a Program is loaded into memory it creates a Process.
- Each Process can have resource descriptors (i.e. file descriptors) to access many different resources.
    - Memory
    - Open Files
    - Timers
    - Synchronization Objects

Processes contain Threads.
- A Thread is a single path of instructions to execute.
- Processes must have at least one Thread of execution.
- Threads can have unique attributes.
    - Scheduling priority and algorithm
    - Register set
    - Signals
    - Memory stack

▪ Bottom Line: Threads run code, Processes own resources.

● The most basic way to create a new Process is to use the fork()
function.
▪ The fork() function creates a duplicate of the currently running Process

– Call one of the wait() functions.
● Wait() – wait for a child process to terminate.
● Waitid() – wait for a child process to change state.
● Waitpid() – wait for a specific child process to stop or terminate.

Signals are a non-blocking form of Inter-Process Communication.

– void(*sa_sigaction)(int signo, siginfo_t* info, void* other) – This is the
signal handler function for queuing signals.

The main difference between signal and interrupt is that signal is an event that is triggered by the CPU or the software that runs on the CPU while an interrupt is an event that is triggered by an external component other than the CPU

Interrupt is delivered by I/O devices, timers (i.e. for time out

Interrupts are first sent to the Kernel.
▪ Your programs can register to receive these interrupts once the Kernel receives them.

A function which services Interrupts is called an Interrupt Service Routine(ISR)

To handle Interrupts your program must have I/O privileges

ThreadCtl allows you to request changes to a thread.

struct sigevent - This structure defines what should happen when an event occurs. This structure is used for timers and interrupts.

… InterruptAttach() - automatically enables the interrupt

InterruptUnmask() re-enables a hardware interrupt

InterruptMask() disables a hardware interrupt

InterruptDetach() - Allows you to detach an Interrupt event

If you want to block and wait for an Interrupt you can call InterruptWait

Week 3

When a program is loaded into memory it becomes a Process. Every Process contains at least one Thread.

❏ Threads are single path of execution.

How do you create Threads in your own Programs?

#include <pthread.h>

pthread_create (pthread_t *tid, pthread_attr_t *attr, void *(*func) (void *), void *arg);

Return values.

▪ eOK – Success

▪ eAGAIN – Insufficient system resources to create the Thread.

▪ eFAULT – An error occurred trying to access the buffers or the function being used for the new Thread.

▪ eINVAL – Invalid Thread attribute in the attr object.

pthread_attr_t object allows us to set the default behaviour for a Thread when it is created.

If you created a thread and left it joinable (you didn't set it as detached in the attributes) then you can wait for it to die and find out why.

▪ The pthread_join function allows any thread to wait on any other thread and read its return status.

We call programs with more then one thread running at once a "concurrent program".

Any code that access a shared resource is called a "Critical Section"

θ Synchronization is done by using very special objects provided by the kernel.

- θ Mutual Exclusion known as a Mutex.
- θ Semaphore
- θ Conditional Variables

Mutex

λ A mutex has two states (locked or unlocked).

λ If a thread tries to lock a mutex, but discovers that it is already locked, the thread will block until the mutex unlocks (i.e. another thread unlocks it)

◦ The basic mutex functions are:

◦ pthread_mutex_init() - initializes a mutex

◦ pthread_mutex_lock() - locks a mutex, if the mutex is already locked, then the thread blocks until it has acquired the mutex

◦ pthread_mutex_trylock() - attemps to lock a mutex, it is already locked, this function will fail but the thread will not block

◦ pthread_mutex_unlock() - unlocks a mutex

◦ pthread_mutex_destroy() - releases a mutex when you are finished with it

Semaphores
–The key difference is a semaphore allows more then one thread to gain access to it at a time.

λ Threads “post” and “wait” on a semaphore.
– A post will increase a semaphore’s value by one.
– A wait will decrement a semaphore by one.

come in two flavours. Named or unnamed.

The basic semaphore functions are:
sem_init() - initialize an unnamed semaphore
sem_open() - create or access a named semaphore
sem_post() - increment a named or unnamed semaphore
sem_wait() - decrement a named or unnamed semaphore; if the semaphore = 0, the calling process blocks until another process does a sem_post
sem_trywait() - decrement a named or unnamed semaphore; if the semaphore = 0, the function call fails rather than blocking
sem_destroy() - destroy an unnamed semaphore
sem_close() - close a named semaphore
sem_unlink() - destroy a named semaphore