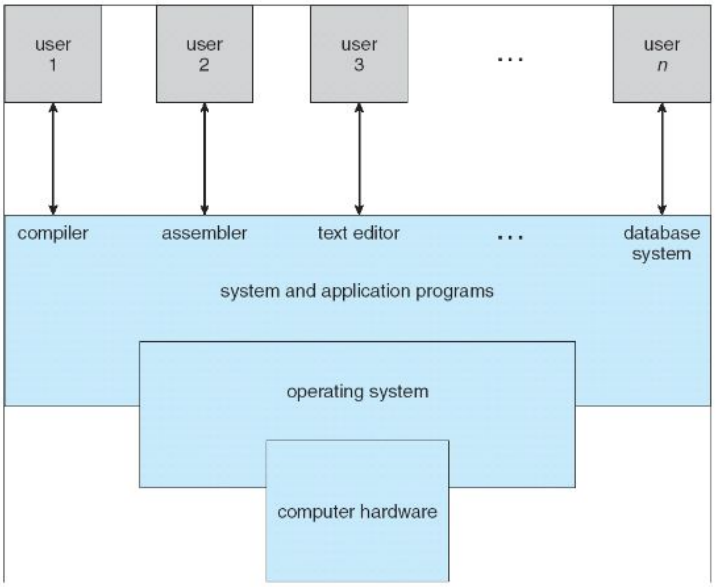
Week 1

Operating System Overview

- Computer system can be divided into four components
  - **Hardware** - provides basic computing resources (CPU, memory, I/O devices)
  - **Operating System** - Controls and coordinates use of hardware among various applications and users
  - **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users (word processors, compilers, web browsers)
  - **Users** - People, machines, and other computers



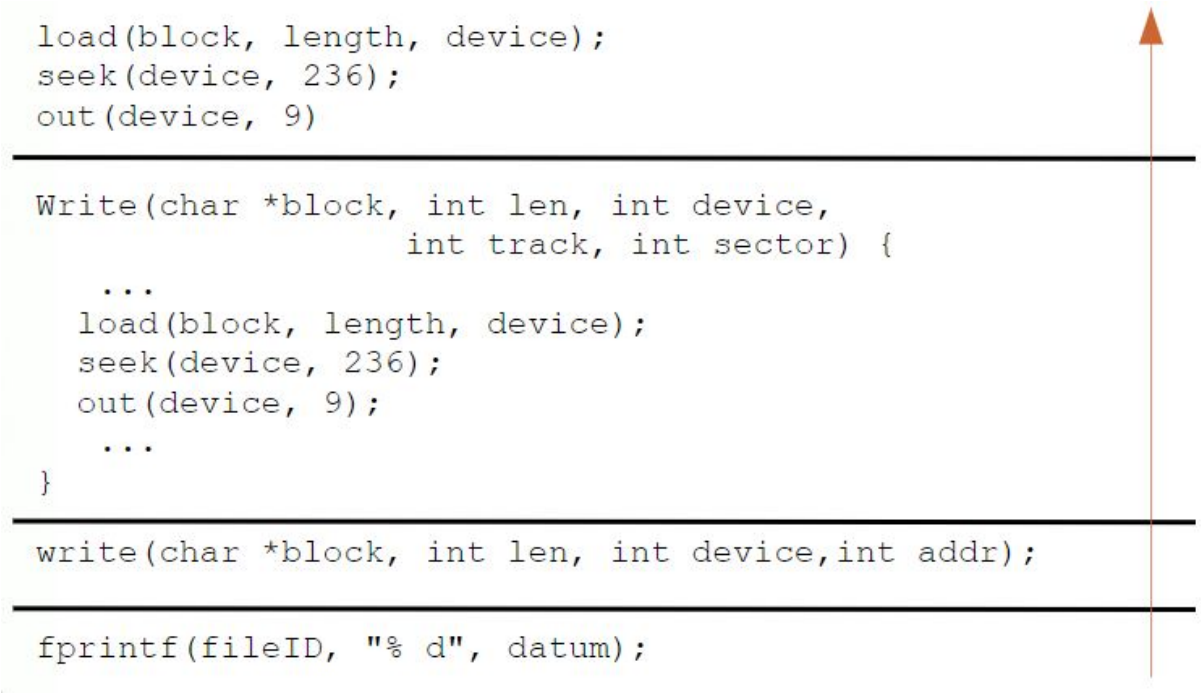
Operating System

- An OS is a resource allocator
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- An OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer

Requirements:

- Provide resource abstractions
  - Convenience of use
  - Process abstraction of CPU / memory
    - Address space
    - Thread abstraction of CPU within address space

    - Resource: "*Anything a process can request that can block the process if it is unavailable*"
    - File abstraction of secondary storage use
- Manage resource sharing
    - Efficient operation of the computer system
    - Time/space-multiplexing
    - Exclusive use of a resource
    - Isolation
    - Managed sharing



Functions in C (and most programming languages) are just higher level abstractions upon higher level abstractions. In this example, a call to fprintf gets translated into a call to write, which gets translated into a call to write with the track and sector in the parameters, which gets translated to load/seek/out, which then gets translated to assembly language.

These higher-level abstractions allow for isolation and managed sharing of a resource (in this case memory), by only allowing the kernel to use the lower-level functions, and requiring the user to use the higher-level functions. For example, restricting the user to using fprintf, and only allowing the kernel to use the write functions.

Resource Sharing

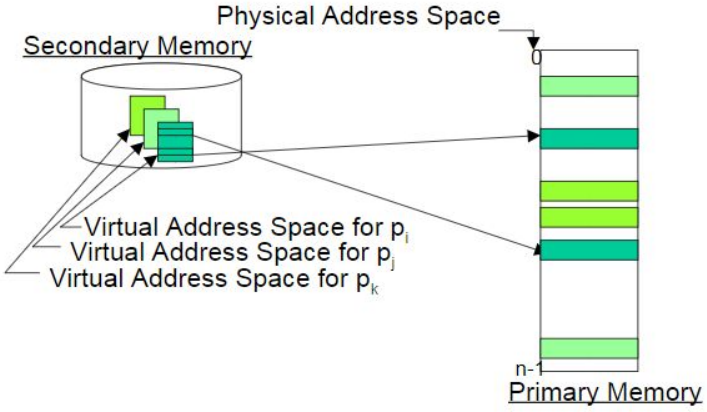
Multiplexing - a method by which multiple analog or digital signals are combined into one signal over a **shared** medium.

- Space- vs time-multiplexed sharing
    - Priority based on memory vs queue time?
- To control sharing, an OS must be able to isolate resources

- OS usually provides a mechanism to isolate, then selectively allow sharing
    - How to isolate resources?
    - How to be sure that sharing is acceptable?
- Transparent resource sharing vs Explicit sharing?
- **FILL THESE IN WITH ANSWERS**

Space Multiplexing: Virtual Memory

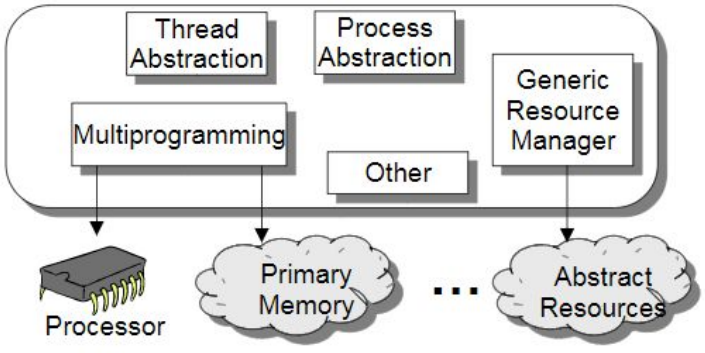
- Physical memory is “split” across multiple applications



More diagrams in 01_A_OS_Overview that I don’t understand (slide 11-12)

Process & Resource Management

- **Process manager** allows users to share the machine by providing multiple execution contexts and scheduling the processor
- **Resource manager** allocates resources to processes when they are requested and keeps track of resources



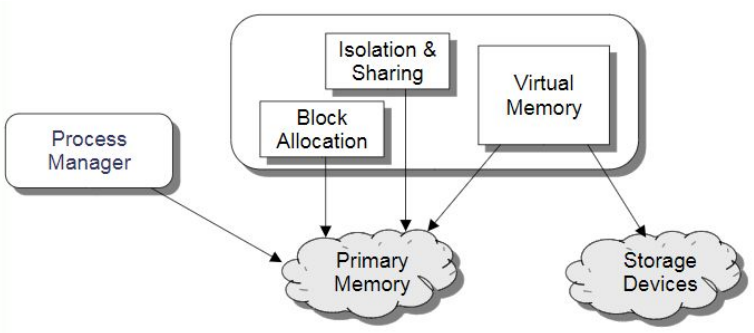
Process Manager

- 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
    - Initialization data

- Typically system has many processes running concurrently on one or more CPUs.
    - Concurrency by multiplexing the CPUs among the processes/threads
- Concurrency vs Parallel
    - Concurrent - Threads will run asynchronously (not in any particular order)
        - Example: Two instances of a project would run concurrently
    - Parallel - Threads will run uniquely at the same time
        - Example: Two separate copies of a QNX project would run in parallel
    - Threads logically appear to be executing in parallel although they may be physically executing on a uniprocessor (a single processor)
    - Every application *thinks* it has its own CPU, but it's not quite like that

Memory Management

- The memory manager cooperates with the process manager to administrate the allocation and use of primary memory.

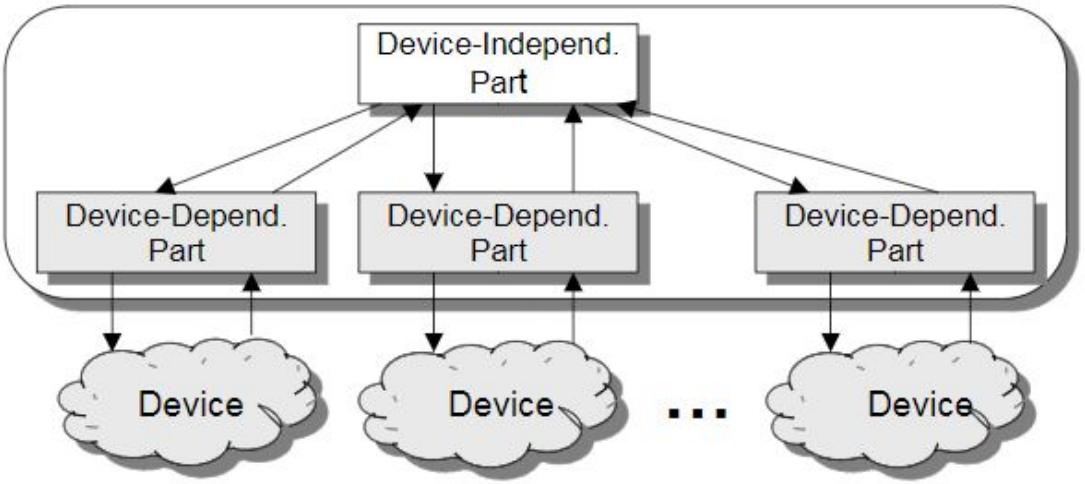


File / Storage Management

- OS provided uniform, logical view of information storage
    - Abstracts physical properties to logical storage unit - **file**
    - Each medium is controlled by device (i.e., disk drive, tape drive)
- File-System management
    - Files usually organized into directories
    - Access control on most systems to determine who can access what
    - OS activities include
        - Creating and deleting files and directories
        - Primitives to manipulate files and directories
        - Mapping files onto secondary storage
        - Backup files onto stable (non-volatile) storage media

Device Management

- Device management is the part of the OS responsible for directly manipulating the hardware devices.
- It provides the first level of abstraction of these resources that will be used by applications to perform I/O



OS Design and Implementation

- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- User goals and System goals
    - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
    - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Important principle to separate:
  Policy: What will be done?
  Mechanism: How to do it?

OS Design Constraints

- **Performance**. The OS should be as efficient as possible in its use of the resources.
    - The OS is an overhead function⇒ should not use too much of machine's resources
    - Minimum functionality is to implement abstractions
    - Additional functionality must be traded off against performance
- **Exclusive use of resources**. The OS must provide resource isolation. Protection and Security
    - Multiprogramming ⇒ resource sharing
    - Therefore, need software-controlled resource isolation
- **Correctness & Maintainability**
    - Security depends on correct operation of software ⇒ **trusted** vs **untrusted** software
    - Maintainability relates to ability of software to be changed
- **Commercial factors**
    - ???

Implementation Mechanism

- **Processor modes** - Hardware is used to distinguish between OS or user execution.
- **Kernels** - Core of the OS designed as a trusted software that support the correct operation of all other software.
- **Method of invoking system service** - How user processes request services from the OS.

Processor Modes

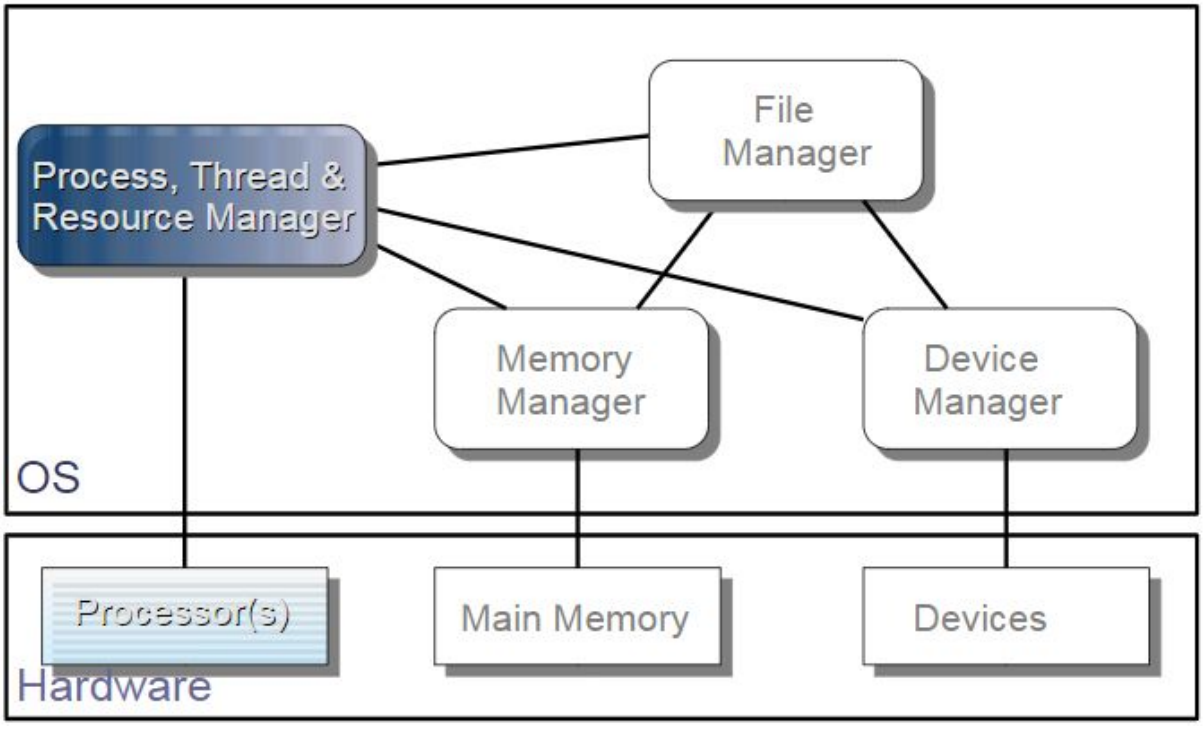
- Mode bit: **Supervisor** or **User** mode
- **Supervisor mode**
  - Can execute all machine instructions
  - Can reference all memory locations
- **User mode**
  - Can only execute a subset of instructions
  - Can only reference a subset of memory location

Kernels

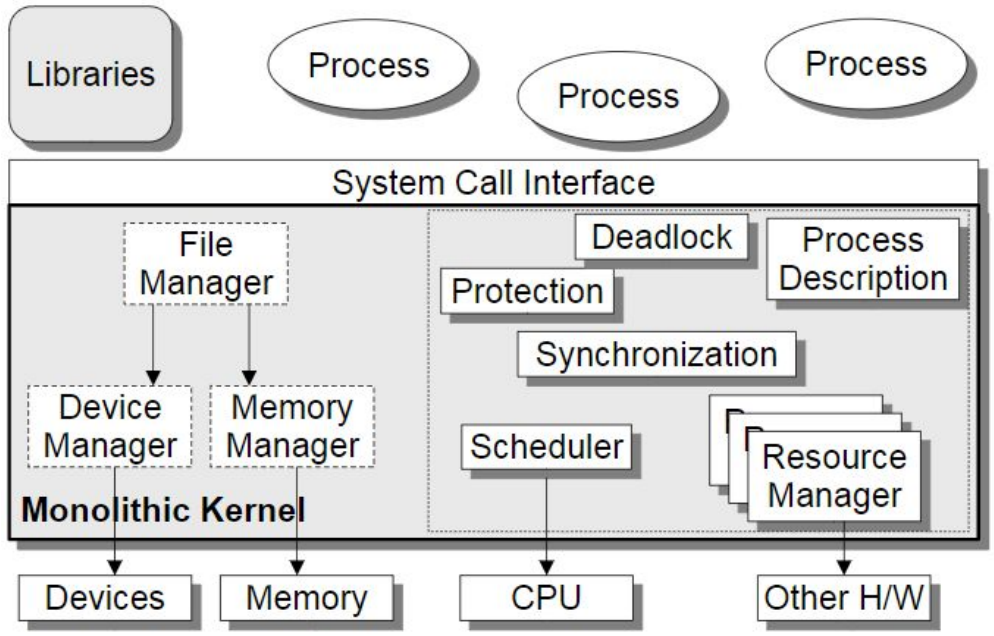
- The part of the OS critical to correct operation (trusted software)
- Executes in supervisor mode
- The *trap* instruction is used to switch from user to supervisor mode, entering the OS

Processes, Threads and Resources

OS Basic Organization



Linux / UNIX Organization



Process

- A process is a program in execution.
- A process is the unit of work in a system.
- A process is composed of the following elements:
    - An **address space** to refer memory location
    - A **program** to define the behaviour of the process
    - The **data** used by the process
    - The **resources** required for execution
    - A **process identifier** to have a unique reference

Process Manager Responsibilities

1. Define & implement the essential characteristics of processes and threads
    a. **Process descriptor** - Data structures to preserve the state of the execution
    b. **Address space** - memory locations that can be referenced by the process
    c. **Context switch** - Algorithms to define the behavior
2. Tools to create/destroy/manipulate processes & threads
3. Manage the resources used by the processes / threads
4. Tools to time-multiplex the CPU
5. Tools to allow threads to synchronization the operation with one another
6. Mechanisms to handle deadlock
7. Mechanisms to handle protection

Process Descriptors (1a)

- The data structure where the OS keeps all information it needs to manage the process (also known as the **Process Control Block - PCB**)
- **Process state**, current process state
- **Program counter**, indicates the address of the next instruction to be executed by this process
- **CPU registers**, accumulators, index registers, stack pointers, general purpose register, etc.
- **Scheduling information**, process priority, pointer to scheduling queue, etc.
- **Address space**, a description of the address space
- ... much more

Address Space (1b)

- Collection of memory addresses that a process can reference
    - Some parts are built into the environment
        - Files
        - System services
    - Some parts are imported at run time
        - Mailboxes
        - Network connections
    - Memory addresses are created at compile (and run) time
- In a 32-bit arch. the address space is 2^32 or 4 GB

Context Switch (1c)

- Procedure of **saving** all registers of one process and then **reloading** all register with the values for another process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

2 Process Creation / Termination

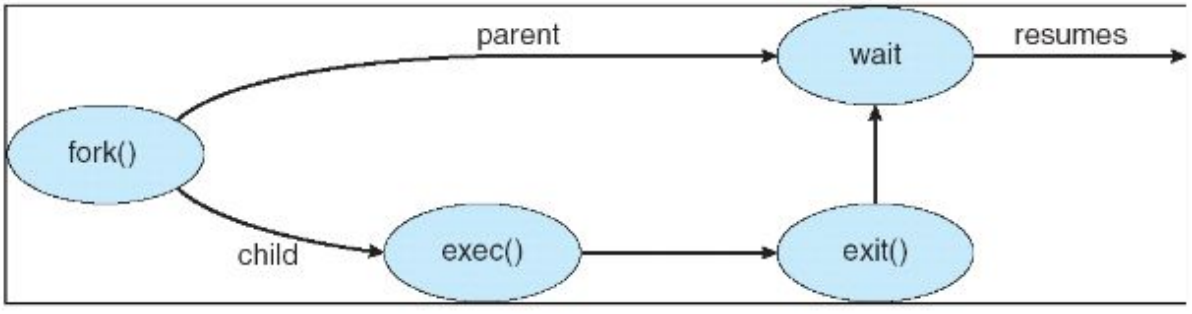
- Creation
    - New program executing
    - Interactive logon
    - Need for the OS to provide a service (for example printing for a user process)
    - Spawned by another process
- Termination
    - Normal completion
    - Memory unavailable
    - Bounds violation
    - Protection error
    - And so on...

Process Creation

- To create a new process, the process manager has to:
  - Assign a unique process identification to the new process and add a new entry in the primary process table.
  - Allocate space for the process
  - Initialize the process control block
  - Link the process to the scheduling queues
  - Create or expand any other data structures need in the OS
- Parent process create children processes, which, in turn, create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Termination

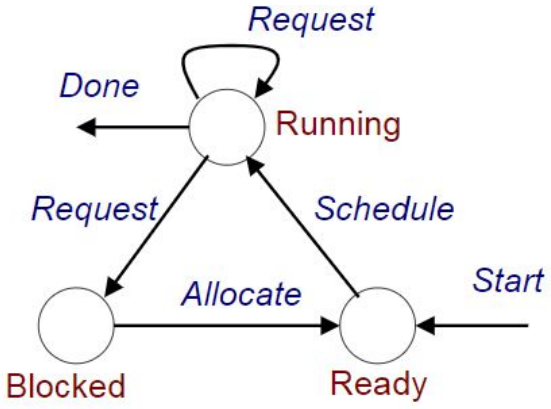
- Process executes last statement and asks the OS to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination



State

- **State**. Representation of the action in which the process is currently engaged.
- **State diagram**. Set of states and transitions between states. It represents the operating system's characterization of a process in terms of how it is managed.
- **State transition**. Action that causes the state to change

Simple State Diagram



Week 2

Intro To QNX

- QNX is a company.
- QNX is a development platform.
- QNX is local.
- QNX is a subsidiary of Research in Motion BlackBerry!

The QNX Development Platform

- It consists of the following:
    - The Momentics Tool Suite
    - Neutrino RTOS
    - An incredible amount of documentation and help manuals

QNX Momentics Tool Suite

- Momentics is the IDE provided by QNX to facilitate software development for the Neutrino RTOS.
- Enables development, debugging, profiling, tracing and diagnostics
- Runs on a “development host”. There are 3 choices of development hosts: Linux, Windows and Mac OS

QNX Neutrino RTOS

- Neutrino is the name given to the QNX RTOS – Real-Time Operating System.
- Neutrino is a microkernel architecture.
- One of the most reliable, highly trusted and flexible operating systems

What is an RTOS?

- An operating system (OS) intended to serve real-time application requests.
- It must be able to process data as it comes in, typically without buffering delays.
- Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.
- A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is **jitter**.
- **Hard real-time OS**
    - less jitter
    - Can *usually* meet a deadline
- **Soft real-time OS**
    - more jitter
    - Can meet a deadline *deterministically*
- An RTOS has an advanced algorithm for scheduling.
    - Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications.
    - Key factors in a real-time OS are
        - minimal interrupt latency and
        - minimal thread switching latency
    - A real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

Monolithic Kernels vs Microkernels

- Monolithic Kernel
    - Entire OS and it's services run in the same process thread
    - Share the same memory space
- MicroKernels
    - Only the bare minimum of the OS is in the kernel process.
    - The rest reside in external servers that run in the user space.

Advantages

- Monolithic Kernels
    - Less software so should be faster
    - One piece of code so should be smaller in source and binary forms
    - Less code typically translates to less bugs
- Microkernel
    - Easier to maintain
    - Patches can be developed in separate instances and swapped over to production
    - Better persistence – if an instance of a server is corrupted, another can be swapped into place.

Disadvantages

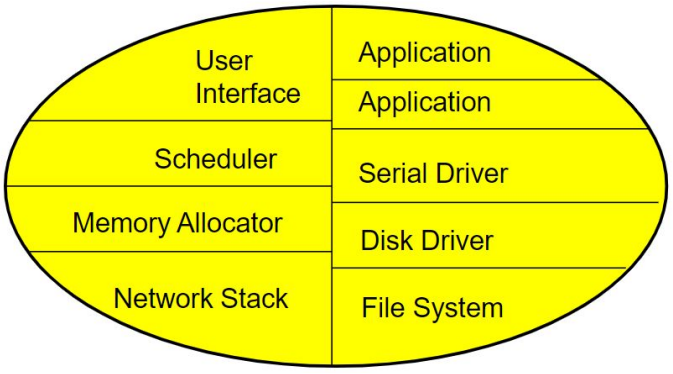
- Monolithic
  - Developing in the kernel space is difficult and requires frequent reboots of the system
  - Small bugs have larger, more pervasive side effects
  - Kernels become large and hard to maintain
  - One bug to rule them all – one can bring down the entire system
  - Not portable, need to be rewritten for each architecture
  - Difficult to integrate the different modules
- Microkernel
  - Larger memory footprint
  - More software needed for interfacing servers with core kernel
  - Process management can be more complicated

- Who wins? Microkernel.

Week 3

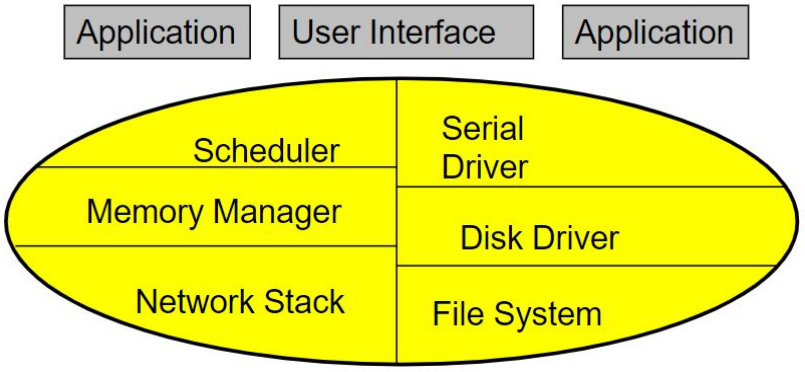
QNX Neutrino Architecture

In a traditional Real-Time Executive:



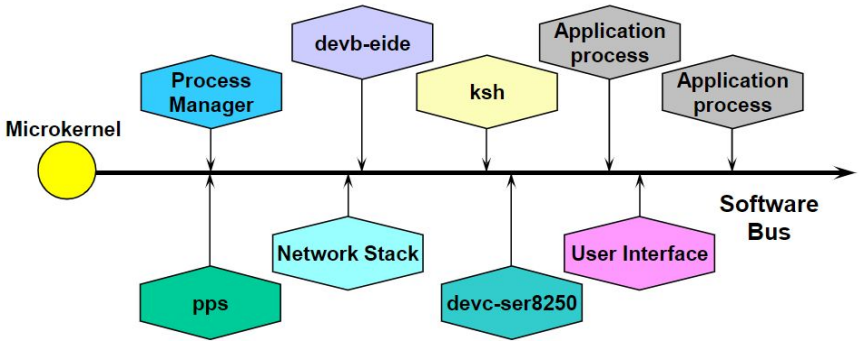
- All modules share the same address space and are, effectively, one big program

In a traditional Monolithic kernel OS:



- The kernel contains the OS kernel functionality and all drivers, so driver development is complex and debugging can be painful.
- Applications are processes in protected memory space, so the kernel is protected from applications and applications are protected from each other.

In the QNX Neutrino OS:



- The OS consists of the microkernel (or just “kernel”) and a set of cooperating processes.
- The processes are separate from the kernel so if something goes wrong in a process it would not affect the kernel.
- The OS processes and your processes cooperate using interprocess communication. Together, the OS and your processes make up one seamless system.
- There are a large variety of types of interprocess communication

Examples of processes are:

- Disk Drivers
  - devb-eide, devb-aha2
- Network Stack
  - Io-pkt
- Character Drivers
  - devc-ser8250, devc-con
- GUI components
  - Screen
- Bus managers
  - pci-server, io-usb-otg
- System daemons
  - cron, inetd, mqueue, qconn

Trade-Offs:

- Benefits:
  - resilience and reliability
  - ease of configuration and reconfiguration
  - ease of debugging
  - ease of development
  - Scalability
- Costs:
  - system overhead

- more context switches
- more copies of data

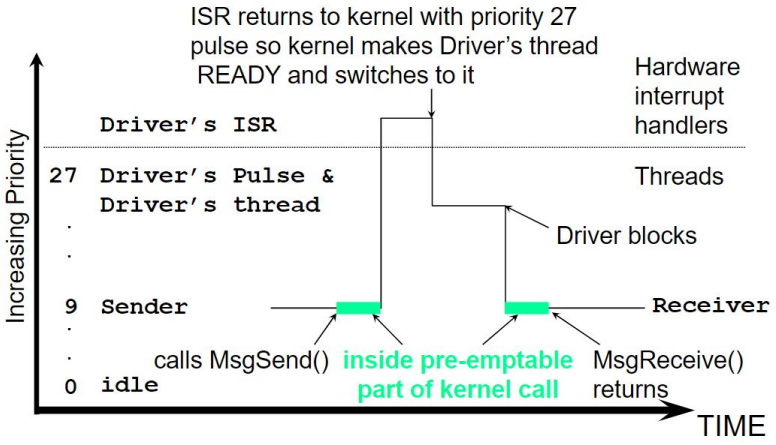
Processes and threads:

- processes are your "building blocks" components of a system
  - visible to each other
  - communicate with each other
- threads are the implementation detail
  - hidden inside processes

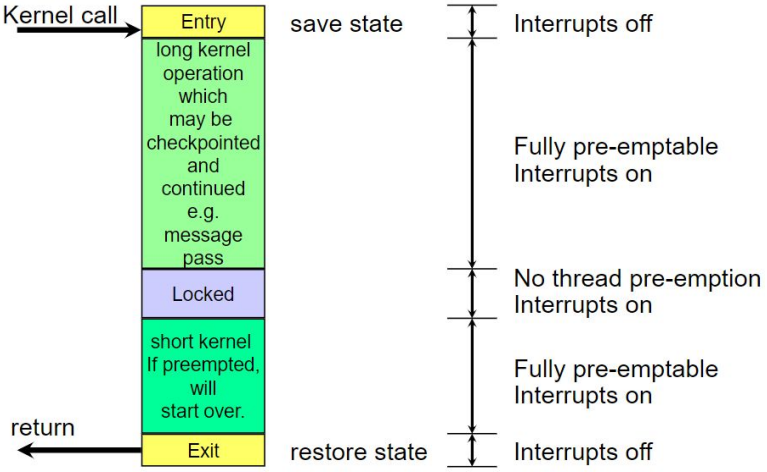
The Kernel is special:

- it is the glue that holds the system together
- programs deal with the kernel by using special library routines, called “kernel calls”, that execute code in the kernel
- most of the other sub-systems, including user applications, communicate with each other using the message passing provided by the kernel through kernel calls

- Kernel calls:
  - this means you'll be executing code in the kernel for the duration of the call
  - Kernels are **preemptable**
    - *Preemptable* means a resource can be taken away from its current owner/place and be given back later.



Kernel operations



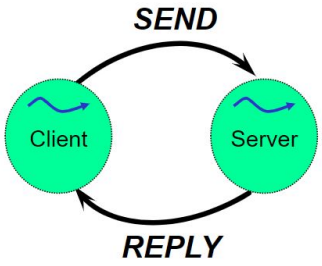
So what does "Preemption" mean?

- More trade-offs:
  - benefit: reduce latency
    - respond to new events faster
    - shorter interrupt latency, scheduling latency
  - cost: throughput
    - takes more time to restart an interrupted kernel call
    - take more time to save current state & restart a pre-empted message pass

The forms of IPC (Inter-Process Communication) provided by the kernel:

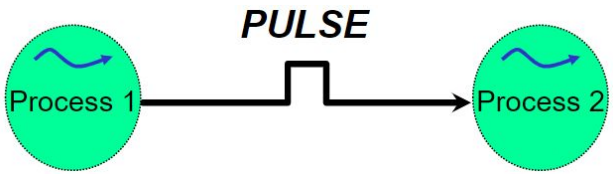
- Messages
  - exchanging information between processes
- Pulses
  - delivering notification to a process
- Signals
  - interrupting a process and making it do something different (usually termination)

Native QNX Neutrino Messages:



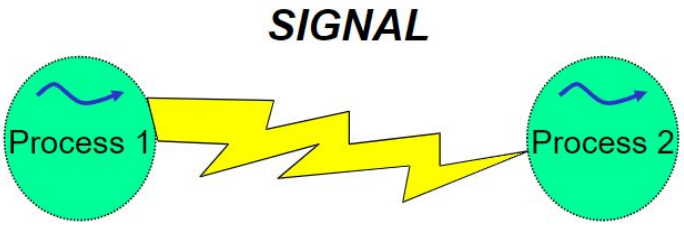
Native QNX Neutrino Pulses:

- used for event notification: "something happened"



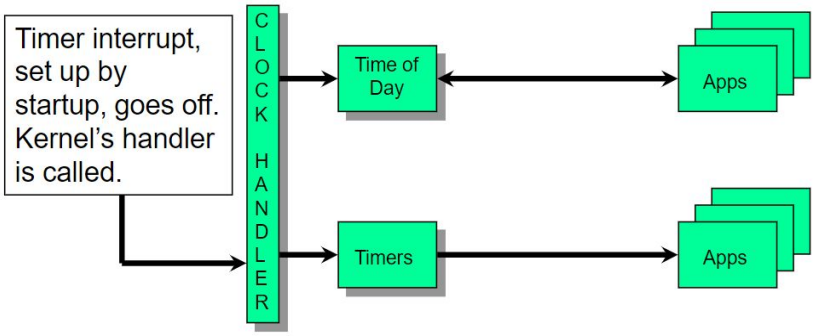
POSIX Signals:

- interrupt another process



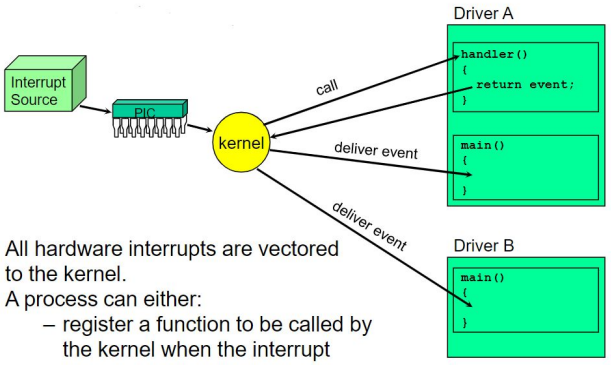
Thread synchronization methods:
Mutex - mutually exclude threads
Condvar - wait for a change
Semaphore - wait on a counter
Join - synchronize to termination of a thread

QNX Neutrinos Concept of Time



Interrupt Handling

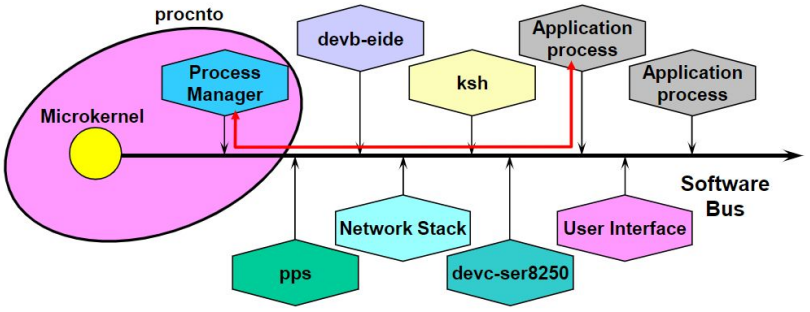
- All hardware interrupts are vectored to the kernel.
- A process can either:
  - register a function to be called by the kernel when the interrupt happens
  - request notification that the interrupt has happened



Kernel

- can be thought of as a library
    - no processing loop, no **while(1)**
- only runs if invoked by:
    - kernel call
    - Interrupt
    - processor fault/exception e.g. illegal instruction, invalid address

Communication with Process Manager



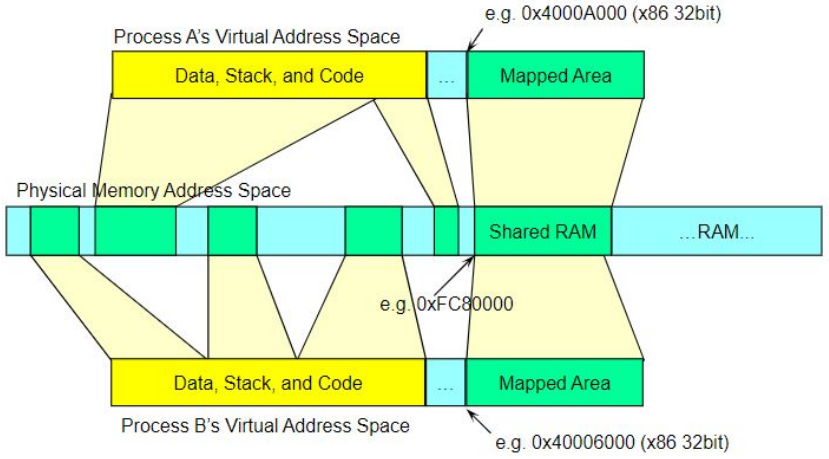
- **procnto** is QNX
    - proc for the process manager
    - nto for the Neutrino microkernel
    - they share address space, but behave differently
- process manager is reached using messages

- The Process Manager provides:
    - packaging of groups of threads together into processes
    - memory protection, address space management including shared memory for IPC
    - pathname management
    - process creation and termination
        - spawn / exec / fork
        - loads ELF executables
    - system information
    - an idle thread that uses CPU no-one else wants
        - or threads in multi-core
- We use a virtual address model:
    - all process run in their own virtual address space
    - all pointers/addresses you use will be virtual addresses, not physical
        - to access physical (hardware) addresses, you must create a virtual mapping
        - all processes share the underlying physical address space
    - the system process/kernel (**procnto**) have an address space that doesn't overlap user processes
        - this makes kernel calls cheaper
    - on 64-bit architectures all virtual address spaces are 512G in size
    - on 32-bit architectures, it varies on platform:

- x86 user space is 0 -3.5G, system 3.5G –4G
- ARM user space is 0 -2G, system 2G –4G

- QNX 7 supports running with 32-or 64-bit address spaces
  - If running with 64-bit address spaces, many data types have changed
    - Pointers, long, size_t, and ssize_t are 64-bit
      - time_t changes from unsigned 32-bit to signed 64-bit

Virtual addresses map to physical addresses:



- Pointers that you deal with contain virtual addresses, not physical

- When QNX Neutrino starts up, the entire pathname space is owned by **procnto**
- Any requests for file or device pathname resolution are handled by **procnto**
- **procnto** allows resource managers to adopt a portion of the pathname space

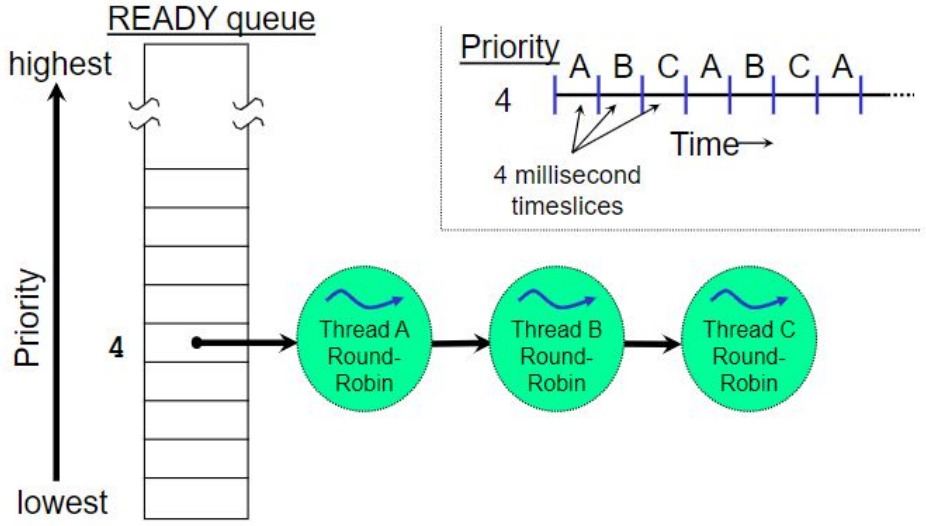
Scheduling

- Threads have two basic states:
  - Blocked
    - waiting for something to happen
    - there are lots of different blocked states depending on what they are waiting for, e.g.:
      - **REPLY** blocked is waiting for an IPC reply
      - **MUTEX** blocked is waiting for a mutex
      - **RECEIVE** blocked is waiting to get a message
  - Runnable
    - capable of using the CPU
    - two main ready states
      - **RUNNING** actually using the CPU
      - **READY** waiting while someone else is running
- All Threads have a priority
  - the priority range is 0 (low) to 255 (high)
  - priority matters for ready threads only
  - the kernel always picks the highest priority **READY** thread to be the one that actually uses the CPU (fully pre-emptive)

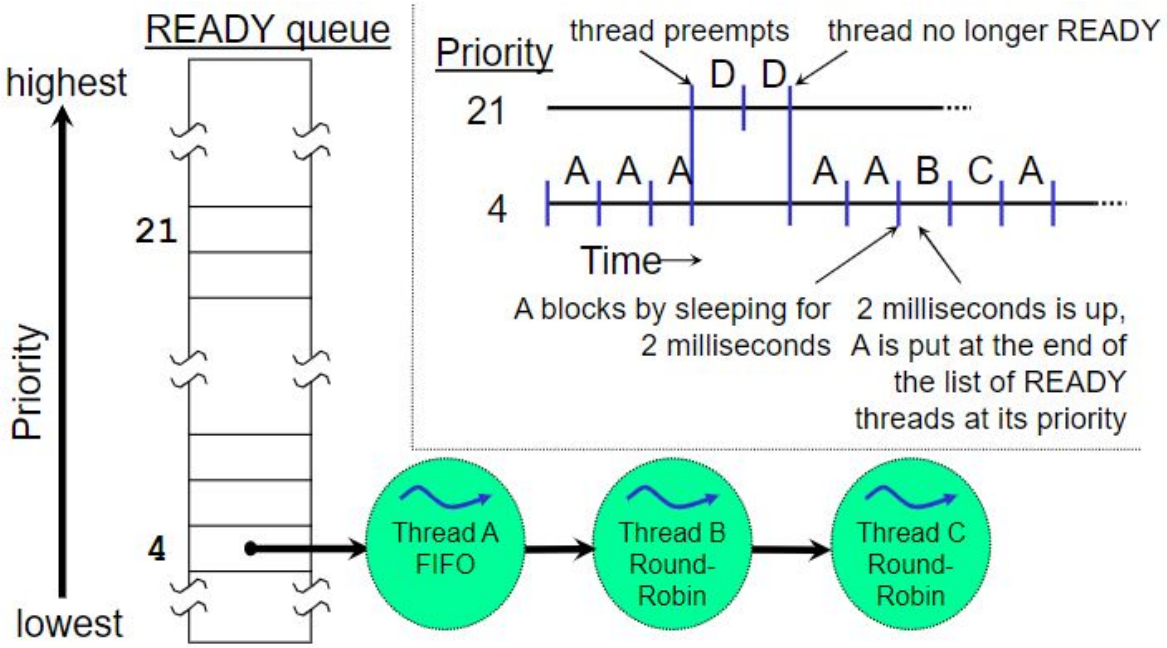
- the thread's state becomes **RUNNING**
- blocked threads don't even get considered

- most threads spend most of their time blocked
  - that is how CPU is shared between threads

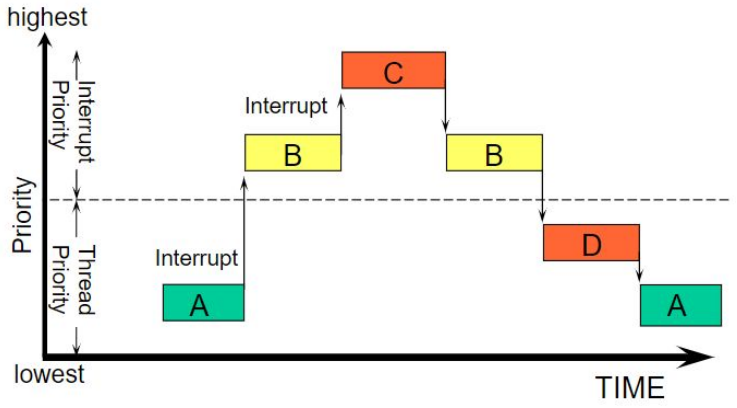
Round-robin



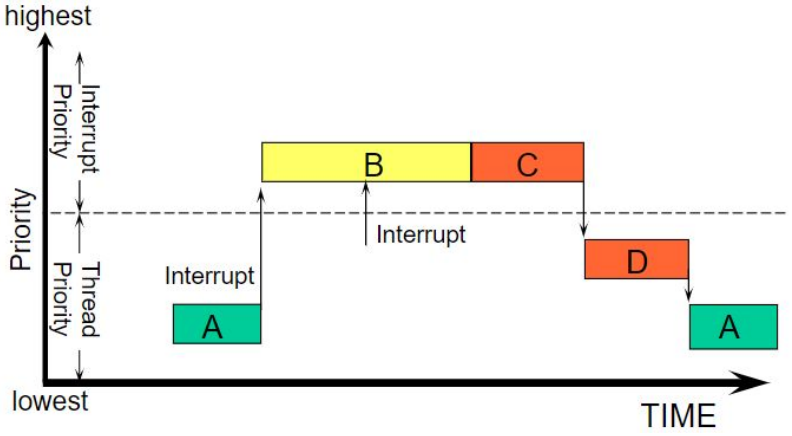
FIFO



Interrupt Scheduling (preemptive)

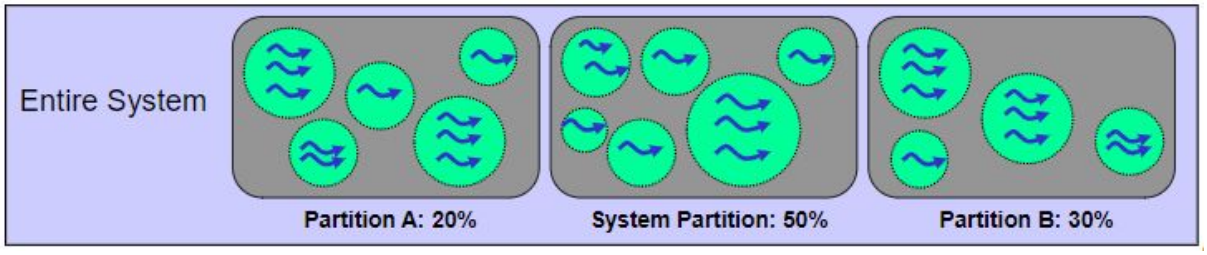


Interrupt Scheduling (non-preemptive)



Adaptive Partitioning

- System designer:
  - creates scheduling partitions
  - decides which partition processes/threads go into
    - child processes/threads go into parent's partition by default
  - specifies minimum % CPU usage for each partition



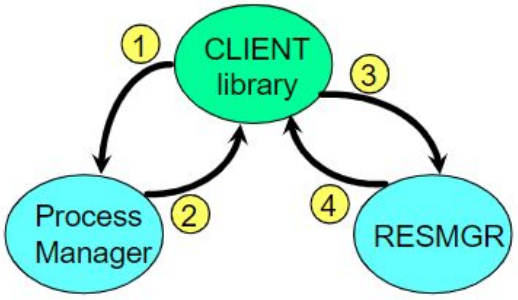
- Scheduling is 'Adaptive':
  - if CPU time is not needed by a partition, it can go to another one
  - if system is < 100% loaded:
    - scheduling works as it does without adaptive partitioning
    - CPU time goes to highest priority thread in system

    - threads that have strict real-time requirements can be designated as having a ‘critical priority’
        - e.g. interrupt handling threads
        - all threads that are at or above the defined ‘critical priority’ are considered to be critical
        - a critical budget (in milliseconds) must be specified (default of 0) in addition to the partition’s ‘regular’ budget
        - critical threads only deduct from their critical budget when the partition’s ‘regular’ budget has been exhausted
- Multicore:
    - multicore is a single chip that has two or more tightly coupled CPUs
    - you don't have to write special code
    - on a multicore system, threads of different priorities or multiple FIFO threads of the same priority may execute at the same time

What is a resource manager?

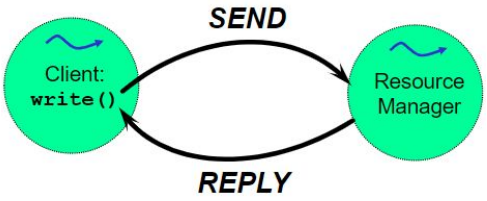
- a program that looks like it is extending the operating system by:
    - creating and managing a name in the pathname space
    - providing a POSIX interface for clients (e.g. open(), read(), write(), ...)
- can be associated with hardware (such as a serial port, or disk drive)
- or can be a purely software entity (such as **mqueue**, the POSIX queue manager)

Interactions



1. Client’s library (open()) sends a “query” message
2. Process Manager replies with who is responsible
3. Client’s library establishes a connection to the specified resource manager and sends an open message
4. Resource manager responds with status (pass/fail)

Further communication is message passing directly to the resource manager:

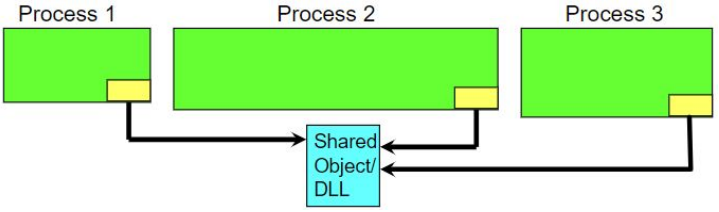


- Other notes:
    - this setup allows for a lot of powerful solutions

        - debug “OS” drivers with a high-level (symbolic) debugger
        - distribute drivers across a QNX network
        - export access to your custom driver with a network file system such as NFS or CIFS
        - provide resiliency or redundancy of OS services
    - QSS supplies a library that provides a lot of useful code to minimise the work needed to write one
- Many standard functions in the library are built on kernel calls
    - usually this is a thin layer, that may just change the format of arguments
    - It’s recommended to use the standard calls
- QNX, instead become a message pass
- they build a message then call MsgSend() passing it to a server

Shared Objects

- one copy used (shared) by all programs using library
- sometimes called **DLL**s (Dynamic-Link Library)
    - shared objects and DLLs use the same architecture to solve different problems



- most system services are delivered by a process
- if you want the service, you run the process
- services can be dynamically configured/removed as needed

Security

- QNX uses Unix style permissions
- Controlling system privileges:
    - traditionally controlled on a root/non-root basis
    - QNX 7 has added many new security features

Conclusion

- QNX Neutrino is a microkernel architecture OS
- most OS services are delivered by cooperating processes
- processes own resources and threads run code
- QNX Neutrino does preemptive scheduling
    - only READY threads are schedulable, blocked threads are not

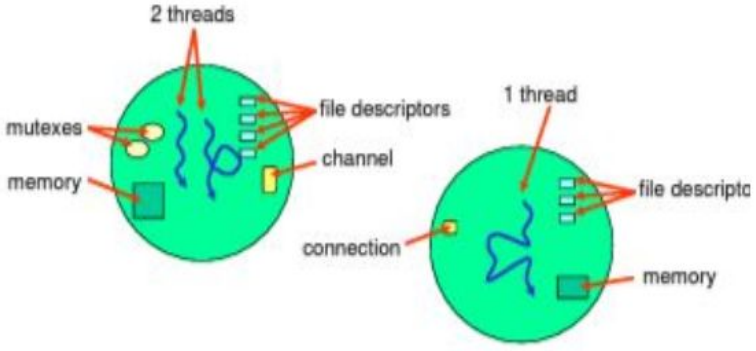
Week 4

Processes Threads Signals

- When a Program is loaded into memory it creates a Process.
    - Each Process is given a unique Process Identification (PID) Number when it is started by the Kernel.
    - Each Process has its own private memory stack.
    - Each Process can have resource descriptors (i.e. file descriptors) to access many different resources.
        - Memory
        - Open Files
        - Timers
        - Synchronization Objects
    - Resources within one Process are not accessible by other Processes

Processes contain Threads.

- A Thread is a single path of instructions to execute.
- Processes must have at least one Thread of execution.
- All Threads within a Process share the same Memory stack and resource descriptors.
- Threads can have unique attributes.
    - Scheduling priority and algorithm
    - Register set
    - Signals
    - Memory stack
- Bottom Line: **Threads run code, Processes own resources.**



Creating Processes

- 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.
    - Continues execution after the fork() function call within both Processes.
    - New “Child” Process inherits all the same resources as the “Parent” Process.
        - File descriptors
        - Threads
- When fork() is called…

        - The “Parent” Process will receive the PID of the newly created Process as the return value from fork().
        - The “Child” Process will receive ‘0’ from the call to fork().
        - If an error occurs ‘-1’ will be returned and the global errno will be set to the appropriate error.
- When a Process is created it is always forked off from a currently running Process.
    - This means every Process is a “Child” of some other Process.
    - When “Child” Processes die they send their return values to their “Parent”.
- Fork is **not** supported within Multi-Threaded Processes

Zombies

- If a “Parent” dies before a “Child” dies the “Child” will have no Process to return to. This creates a zombie Process.
    - Zombies are Processes that are no longer active with the majority of their resources freed but an entry in the Process Table persists.

Detecting Process Termination

- In many cases a “Parent” will need to know when one of its “Child” Processes has died. There are several ways this can be done.
    - 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.
    - Receive a “Child” termination Signal.
        - When “Child” Processes terminate they send a SIGCHLD signal to the “Parent” Process

Signals

- Signals are a non-blocking form of Inter-Process Communication.
    - Signals are asynchronous
    - Signals can be received at any time.
    - Signals are POSIX compliant
- When a Process receives a Signal it has several options on how to handle it.
    - Ignore the Signal.
    - Mask the Signal
    - Terminate the Process (default behaviour for many Signals).
    - Use a Signal Handler function to process the Signal.
- Some common signals:
    - SIGUSR1 & SIGUSR2 - User defined signals
    - SIGINT - Sent when you sent ctrl + c to a Process
    - SIGSTOP - Set to pause Process execution
    - SIGKILL - Terminate a Process by any means necessary
    - SIGTERM - Terminate a Process normally
- Note: You can **not** catch SIGSTOP or SIGKILL Signals

To Deliver a Signal From Code

- Use kill to deliver a Signal.
    - Int kill (pid_t pid, int sig);
        - Pid–The PID of the Process you want to Signal.
        - Sig –The Signal you want to send
        - Returns 0 on success or -1 on failure.
- Use sigqueueto queue a Signal for a Process.
    - Int sigqueue(pid_t pid, int sig, const union sigval value);
        - Pid–The PID of the Process you want to Signal.
        - Sig –The Signal you want to send.
        - Value –A value to send with the Signal.

To Deliver a Signal From Command Line

- If you want to send a Signal to a Process from the command line you can use the kill shell command. Alternatively you can send the SIGINT Signal to a Process by pressing CTRL + C within the running shell.
- The command is
  Kill –s <signal> <pid>
- Example
  #kill –s SIGUSR1 12413

- **NOT SURE IF THIS IS NECESSARY. REMAINDER OF SLIDES OF SIMILAR NATURE ARE IN SLIDES 14-16**

Signal Example

```
extern void handler();
struct sigaction act;
sigset_t set;

sigemptyset( &set );
sigaddset( &set, SIGUSR1 );
sigaddset( &set, SIGUSR2 );

/*
 * Define a handler for SIGUSR1 such that when
 * entered both SIGUSR1 and SIGUSR2 are masked.
 */
act.sa_flags = 0;
act.sa_mask = set;
act.sa_handler = &handler;
sigaction( SIGUSR1, &act, NULL );

kill( getpid(), SIGUSR1 );

/* Program will terminate with a SIGUSR2 */
return EXIT_SUCCESS;
}
```

Handler Example

```
void handler( signo )
  {
    static int first = 1;

    if( first ) {
      first = 0;
      kill( getpid(), SIGUSR1 );  /* Prove signal masked */
      kill( getpid(), SIGUSR2 );  /* Prove signal masked */
    }
  }

/*
 * - SIGUSR1 is set from main(), handler() is called.
 * - SIGUSR1 and SIGUSR2 are set from handler().
 * - however, signals are masked until we return to main().
 * - returning to main() unmasks SIGUSR1 and SIGUSR2.
 * - pending SIGUSR1 now occurs, handler() is called.
 * - pending SIGUSR2 now occurs. Since we don't have
 *   a handler for SIGUSR2, we are killed.
 */
```

Other Details

- To mask signals for each thread instead of for the entire process:
    - use pthread_sigmask().

Important Notes

- Signal handlers are designed to be small and fast. You should always limit the amount of work you do within the handler.
- Some functions are not safe to call in Signal Handlers. This will be mentioned in their documentation.
- If a Thread is blocked and receives a Signal it will not be blocked once it leaves the Signal Handler.

Threads

- When a program is loaded into memory it becomes a Process. Every Process contains at least one Thread.
- Threads are single path of execution.
    - Threads are the smallest sequence of instructions that can be scheduled within an Operating system.
    - Threads allow a Process to accomplish more then one task simultaneously.
- After you have started your thread, you should clean up the data structures you used in the thread setup process

Thread Synchronization

- How do threads manage who gets access to which resource at which time?
    - Thread Synchronization
- Examples of when you would use these.
    - Restricting access to shared memory.
    - Notifying when a memory region is available to read from.
- Maybe now information on this?

Mutex

- A method for controlling access to a shared resource which should have mutual exclusion
- A mutex has two states (locked or unlocked)
- If a thread tries to lock a mutex, but discovers that it is already locked, the tread will block until the mutex unlocks (i.e. another thread unlocks it)
- A mutex which controls a shared region should be at the beginning of that region where it is easy to find
- Try to keep a mutex locked for as short a time as possible.
- Remember other Threads all block while waiting on a Mutex. If these Threads are critical you could “Mutex Lock” your Program

Semaphores

- Semaphores a very similar to a mutex
    - The key difference is a semaphore allows more than one thread to gain access to it at a time
- A semaphore is essentially an integer value that is decremented whenever a thread gets access to it
- Once the value hits 0, the next thread to try and gain access to the semaphore will either block or return an error
- Threads **Post** and **Wait** on a semaphore
    - Post increments the semaphores value
    - Wait decrements the semaphores value
- Semaphores are “asynchronous safe”, meaning you can access them in a signal handler.
- Designed to work between processes
    - Can even span across a network on Neutrino
- Named vs Unnamed
    - Unnamed are designed to work more efficiently within a program.
    - Names can span across processes, however a names semaphore requires a kernel lookup

Common Usage

- The standard method for controlling access to a shared memory region is through semaphores
- A semaphore which controls a shared region should be at the beginning of that region where it is easy to find.

Things to Remember

- When using an unnamed semaphore, a post/wait call will *access?* The kernel directly
- When using a named semaphore, messages are sent to the queue service who then makes the kernel call
- Unnamed semaphores will be faster but names semaphores allow you to use them across processes.

Week 5

Inter-Process Communication (IPC)

- Used to determine how processes communicate with each other
- Why do processes need to communicate? Long answer:
    - Data-Flow requirements
        - eg. a system retrieves data from some measuring instrument and analyzes it - if one process retrieves the data, and another process analyzes it, the first process must transmit that data to the second.
    - Synchronization of process steps
        - eg. an assembly line at a bottling plant - one process controls the conveyor belt and a another fills the bottles, then these two processes must be able to synchronize themselves; the conveyor must be motionless when the filling begins and the filling must be finished when the conveyor starts moving again
    - On-demand requests in client-server
        - eg. In client-server architecture one process (the server) services requests from other processes (the clients); the client must communicate its needs to the server, and the server must communicate the results to the correct client.
- Why do processes need to communicate? Short answer:
    - Synchronize processes
    - Communicate data
- Two methods of IPC:
    - Blocking vs non-blocking
    - Synchronous vs asynchronous

Blocking vs Non-Blocking

- Blocking methods cause *transmitting* process to block until *receiving* process handles transmitted data
- Non-blocking methods do not cause the transmitting process to block

Synchronous vs Asynchronous

- Sync: Receiving process can only handle the transmitted data *at a certain point* in its code
- Async: Receiving process can handle the transmitted data *at any point* in its code

Message Passing

- 3 core functions:
    - MsgSend()
    - MsgReceive()
    - MsgReply()
- 2 supporting functions:

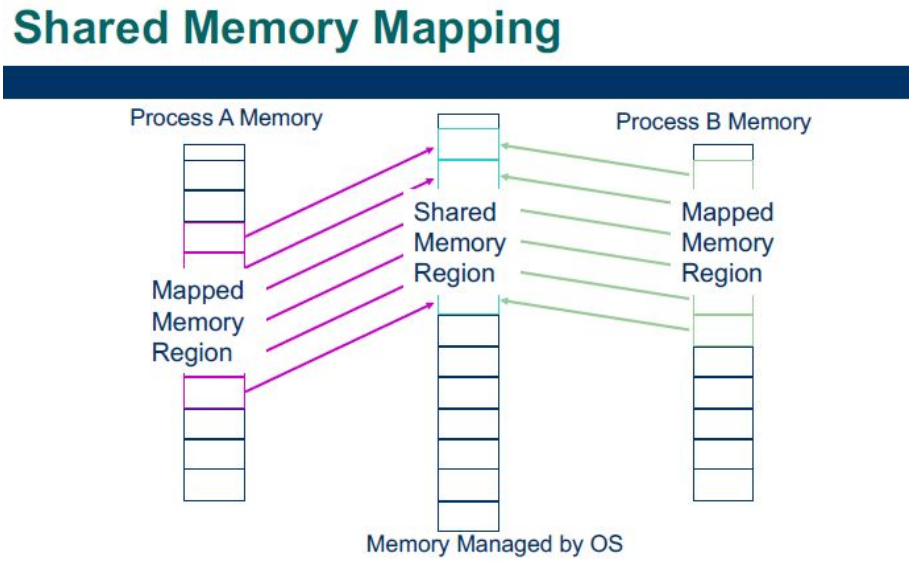
    - ChannelCreate()
    - ConnectAttach()
- Message passing is a **blocking** form of IPC

Shared Memory

- Shared memory allows multiple processes to share the same physical memory. One of the fastest forms of IPC
    - Used when two or more processes need fast access to common data
- Shared memory is **synchronous** and **non-blocking**
- Using shared memory *eliminates the copying of data* commonly found in other forms of IPC
- Using shared memory introduces a number of *synchronization issues* and techniques

How Shared Memory Works

- Shared memory is **NOT fault tolerant**
    - To solve:
        i. Process A and process B map a piece of memory
        ii. OS maps a piece of memory and the two processes relay through it
        iii. If Process A dies, process B will survive because it is mapped to **OS memory** and **not** directly to A



- Shared memory should be **controlled by a semaphore (or mutex)** to ensure the same memory isn't being overwritten during reading