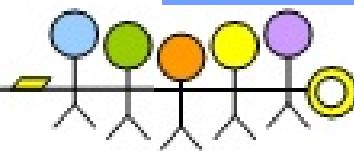


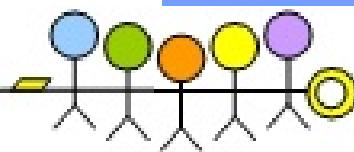
# Tornado &VxWorks

## 培训



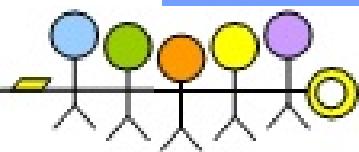
# 实时系统概念

- 实时系统是对外来事件在限定时间内能做出反应的系统。
- 指标
  - 响应时间 Response Time
  - 生存时间 Survival Time
  - 吞吐量 Throughput



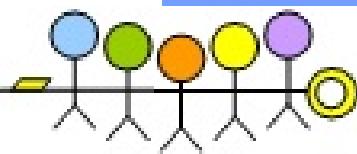
# 实时系统与普通系统

- 在实时计算中，系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间
- 对于实时系统来说最重要的要求就是实时操作系统必须有满足在一个事先定义好的时间限制中对外部或内部的事件进行响应和处理的能力
- 此外作为实时操作系统还需要有效的中断处理能力来处理异步事件和高效的I/O能力来处理有严格时间限制的数据收发应用



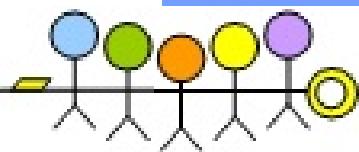
# 实时系统分类

- 根据不同的分类方法可以分为几种。
  - 方法一是分为周期性的和非周期性的 (periodic和aperiodic)
  - 方法二是分为硬实时和软实时 (hard real\_time和soft real\_time)
  - 专用系统和开放系统
  - 集中式系统和分布式系统



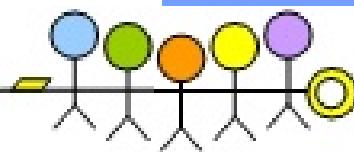
# 实时多任务操作系统与分时多任务操作系统

- 分时操作系统，软件的执行在时间上的要求，并不严格，时间上的错误，一般不会造成灾难性的后果。
- 实时操作系统，主要任务是对事件进行实时的处理，虽然事件可能在无法预知的时刻到达，但是软件上必须在事件发生时能够在严格的时限内作出响应（系统响应时间），即使是在尖峰负荷下，也应如此，系统时间响应的超时就意味着致命的失败。另外，实时操作系统的重要特点是具有系统的可确定性，即系统能对运行情况的最好和最坏等的情况能做出精确的估计。



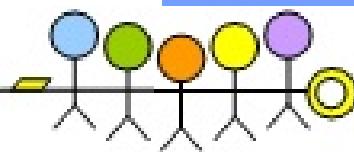
# 实时操作系统中的重要概念

- 系统响应时间(System response time )  
系统发出处理要求到系统给出应答信号的时间。
- 任务换道时间(Context-switching time)  
是任务之间切换而使用的时间。
- 中断延迟(Interrupt latency )  
是计算机接收到中断信号到操作系统作出响应，并完成换道转入中断服务程序的时间。



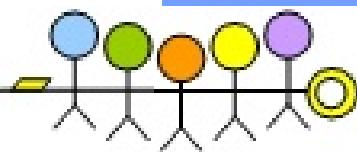
## 实时操作系统应具有如下的功能

- 任务管理（多任务和基于优先级的任务调度）
- 任务间同步和通信（信号量和共享内存等）
- 存储器优化管理（含ROM的管理）
- 实时时钟服务
- 中断管理服务



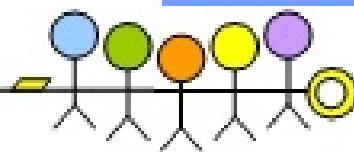
# 硬实时、软实时(一)

- 硬实时要求在规定的时间内必须完成操作，这是在操作系统设计时保证的
- 软实时则没有那么严，只要按照任务的优先级，尽可能快地完成操作即可
- 对于软实时系统基于优先级调度的调度算法可以满足要求，提供高速的响应和大的系统吞吐率；而对于硬实时系统则完成 timely response 是必须的。这两类系统的区别在于调度算法。
- 实时操作系统是保证在一定时间限制内完成特定功能的操作系统。例如，可以为确保生产线上的机器人能获取某个物体而设计一个操作系统。在“硬”实时操作系统中，如果不能在允许时间内完成使物体可达的计算，操作系统将因错误结束。在“软”实时操作系统中，生产线仍然能继续工作，但产品的输出会因产品不能在允许时间内到达而减慢，这使机器人有短暂的不生产现象。



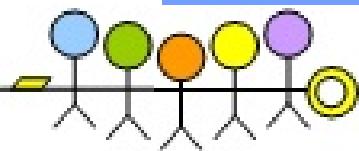
# 硬实时、软实时(二)

- 软实时的RTOS一般应用在消费类电子产品,如手持电脑、个人数字助理(PDA)和机顶盒等消费电子类。WinCE。
- 硬实时的RTOS一般应用于通信、控制和航空航天等实时性强和可靠性高的领域。通信行业使用PSOS、VxWorks、VRTX,航天、航空使用VRTX、VxWorks,工业PC 控制使用QNX。



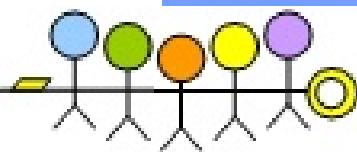
# 实时系统的体系结构设计

- 实时系统的体系结构必须满足
  - 1.高运算速度
  - 2.高速的中断处理
  - 3.高的I/O吞吐率
  - 4.合理的处理器和I/O设备的拓扑连接
  - 5.高速可靠的和有时间约束的通信
  - 6.体系结构支持的出错处理
  - 7.体系结构支持的调度
  - 8.体系结构支持的操作系统
  - 9.体系结构支持的实时语言特性。
  - 10.系统的稳定性和容错也非常 important
  - 11.还要考虑到实时的分布式应用。



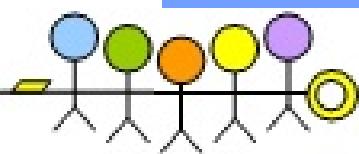
# 实时进程调度算法(一)

- 静态的周期性调度
  - 这种调度算法的基本思想是将处理器的时间分为"帧"。
- FIFO
  - 也就是将系统中所有的任务组织成一个队列。先到先服务
- 优先级队列算法
  - 一种算法从FIFO发展而来。给每个任务设定优先级，然后在FIFO中按照优先级排列。这种算法保证了高优先级的任务的完成，但是对于低优先级的任务很可能无法满足时间的正确性。而且对低优先级的任务来说等待的时间是无法预知的。
- 以上的调度算法都是独占的
  - 即任务运行时，不允许别的任务抢先。完成一个任务后才能完成下一个



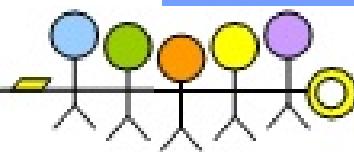
# 实时进程调度算法(二)

- Rate Monatomic/Pacing算法
  - 此算法是基于静态优先级调度协议的方法。此算法给系统中每个任务设置一个静态的优先级。这个优先级的设定是在计算任务的周期性和任务需要满足的deadline的时间的长短的基础上完成的。周期越短，deadline越紧迫，优先级越高。
- Deadline Driven算法
  - Deadline Driven算法提供动态的优先级。因为此算法根据任务满足deadline的紧迫性来修改任务的优先级，以保证最紧迫的任务能够及时完成。
- Priority Ceiling算法
  - 这种算法用于抢先式多任务的实时操作系统。该算法的基本思想是在系统中使用优先级驱动的可抢先的调度算法。也就是系统首先调度高优先级的任务运行。低优先级的任务在高优先级的任务运行时不能抢先；CPU由高优先级进程独占。



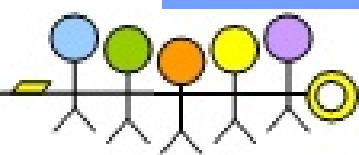
# 嵌入式系统概述

- 嵌入式系统 ( Embedded Systems ) 是指以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。是将应用程序和操作系统与计算机硬件集成在一起的系统



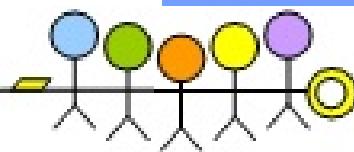
# 嵌入式硬件

- 嵌入式硬件包括处理器 / 微处理器、存储器及外设器件和 I / O 端口、图形控制器等
- 特点：
  - 1) 对实时多任务有很强的支持能力，能完成多任务并且有较短的中断响应时间，从而使内部的代码和实时内核心的执行时间减少到最低限度。
  - 2) 具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断。
  - 3) 可扩展的处理器结构，以能最迅速地开展出满足应用的最高性能的嵌入式微处理器。
  - 4) 嵌入式微处理器必须功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此，如需要功耗只有 mW 甚至  $\mu$ W 级。



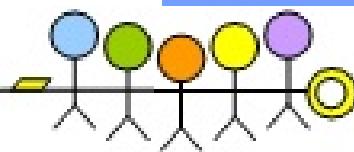
# 嵌入式系统发展趋势

- 嵌入式应用软件的开发需要强大的开发工具和操作系统的支持。
- 联网成为必然趋势
- 支持小型电子设备实现小尺寸、微功耗和低成本
- 提供精巧的多媒体人机界面



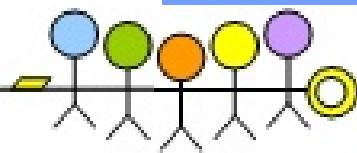
# 实时系统内存管理

- 预先分配内存。
  - 在系统构造或编译时为每个任务指定其使用的内存空间。这种方法对于硬实时系统来说是很合适的。而且嵌入式实时操作系统很多都是在ROM中运行，仅仅只有需要变化的数据才放在RAM中。这种系统在组成上无疑是静态的。
- 虚拟内存
  - 但必须给实时任务提供方法，以便将实时任务“锁”进内存，也就是系统在管理虚拟内存时，不将“锁”住的内存块换出物理内存。

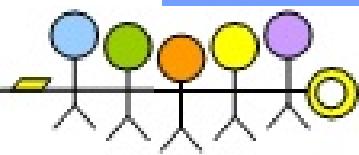


# 嵌入式系统和实时系统

- 嵌入式系统经常被误解为就是实时性系统。其实，多数嵌入式系统并不需要实时性
- Linux是嵌入式操作系统，并非实时操作系统。
- Vxwork、pSOS、Nucleus和Windowss CE 是嵌入式实时操作系统

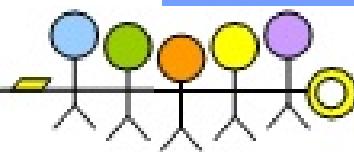


# 一、实时多任务

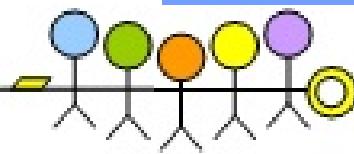
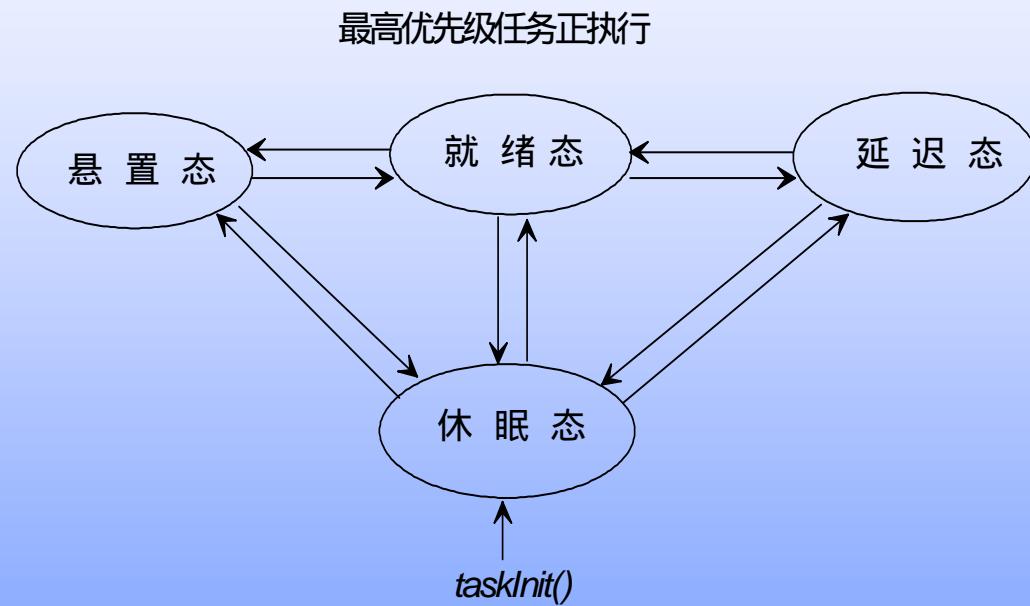


# 任务状态

- 实时系统的一个任务可有多种状态，其中最基本的状态有四种：
  - 就绪态：任务只等待系统分配CPU资源；
  - 悬置态：任务需等待某些不可利用的资源而被阻塞；
  - 休眠态：如果系统不需要某一个任务工作，则这个任务处于休眠状态；
  - 延迟态：任务被延迟时所处状态；

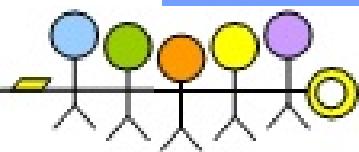


# 任务状态迁移



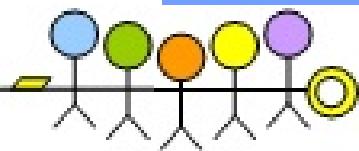
# 任务状态迁移函数（一）

- 就绪态 ----> 悬置态  
*semTake() / msgQReceive()*
- 就绪态 ----> 延迟态  
*taskDelay()*
- 就绪态 ----> 休眠态  
*taskSuspend()*
- 悬置态 ----> 就绪态  
*semGive() / msgQSend()*
- 悬置态 ----> 休眠态  
*taskSuspend()*



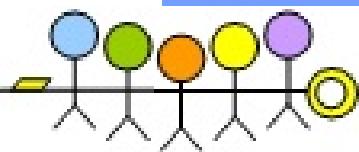
# 任务状态迁移函数（二）

- 延迟态 ----> 就绪态  
`expired delay`
- 延迟态 ----> 休眠态  
`taskSuspend()`
- 休眠态 ----> 就绪态  
`taskResume() / taskActivate()`
- 休眠态 ----> 悬置态  
`taskResume()`
- 休眠态 ----> 延迟态  
`taskResume()`



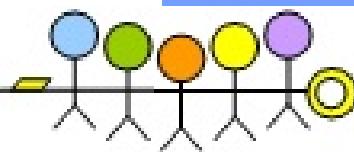
# 多任务内核

- 进行任务管理
  - 任务是竞争系统资源的最小运行单元。任务可以使用或等待CPU、I/O设备及内存空间等系统资源，并独立于其它任务，与它们一起并发运行（宏观上如此）。VxWorks内核使任务能快速共享系统的绝大部分资源，同时有独立的上下文来控制个别线程的执行。
- VxWorks实时内核Wind提供了基本的多任务环境，系统内核根据某一调度策略让它们交替运行。
- 系统调度器使用**任务控制块**的数据结构（简记为TCB）来管理任务调度功能。

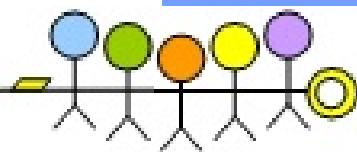
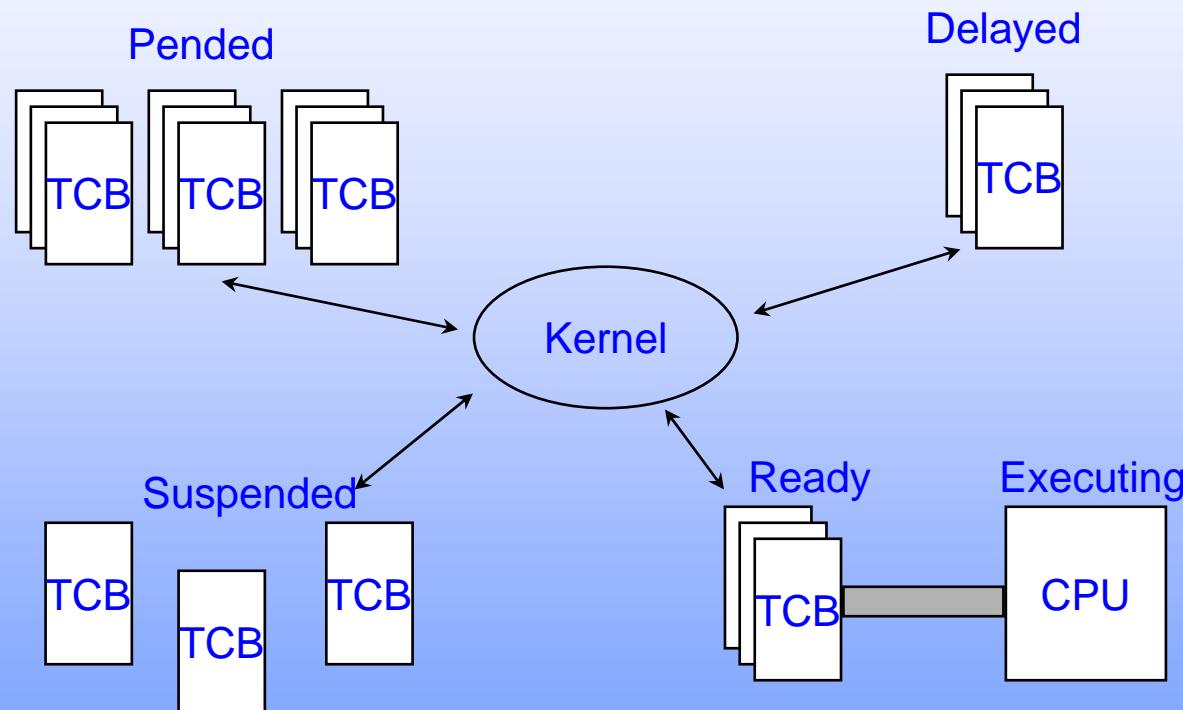


# 任务控制快(TCB)

- 任务控制块用来描述一个任务，每一任务都与一个TCB关联。
- 任务控制块里面包含了：
  - 当前状态、优先级、要等待的事件或资源、任务程序码的起始地址、初始堆栈指针
  - 任务的“上下文”(context)。任务的上下文就是当一个执行中的任务被停止时，所要保存的所有信息。通常，上下文就是计算机当前的状态，也即各个寄存器的内容。
- VxWorks中，内存地址空间不是任务上下文的一部分。所有的代码运行在同一地址空间。如每一任务需各自的内存空间，需可选产品VxVMI的支持。

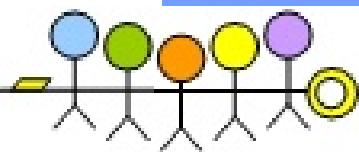


# 内核管理示意图



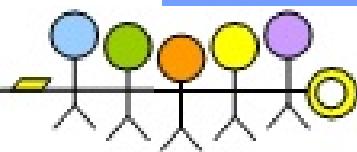
# 任务上下文切换

- 当前运行的任务的上下文被存入TCB
- 将要被执行的任务的上下文从它的TCB中取出，放入各个寄存器中。
- 上下文切换必须很快速的

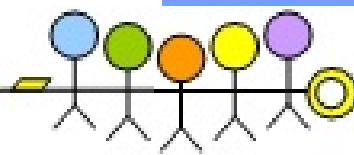
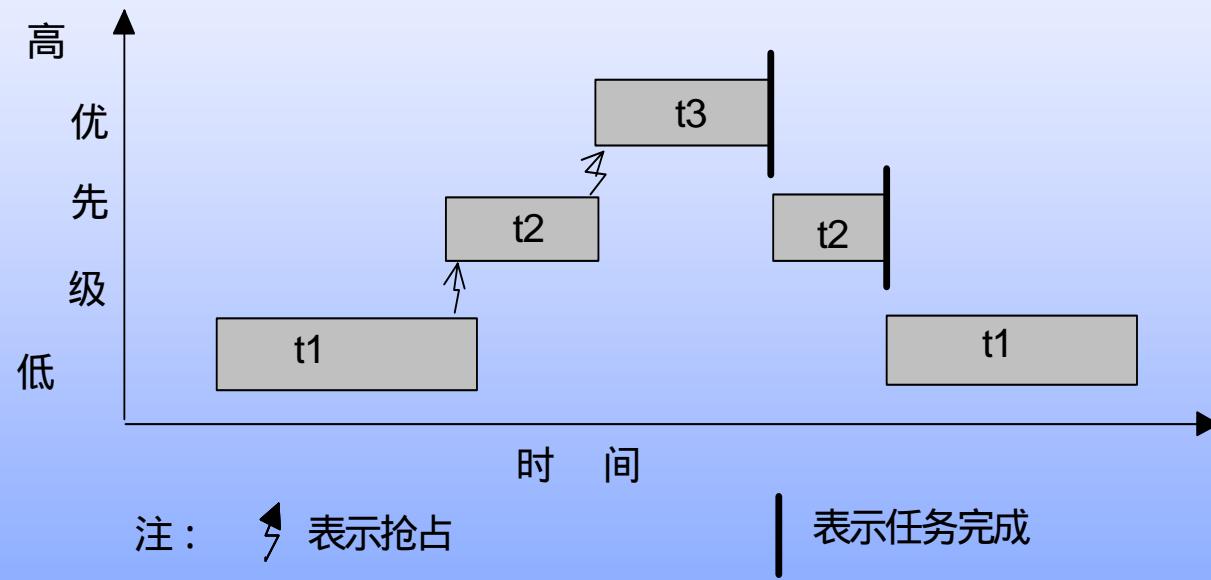


# 优先级调度

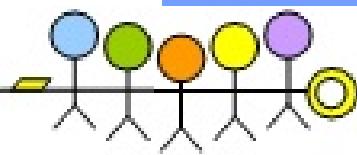
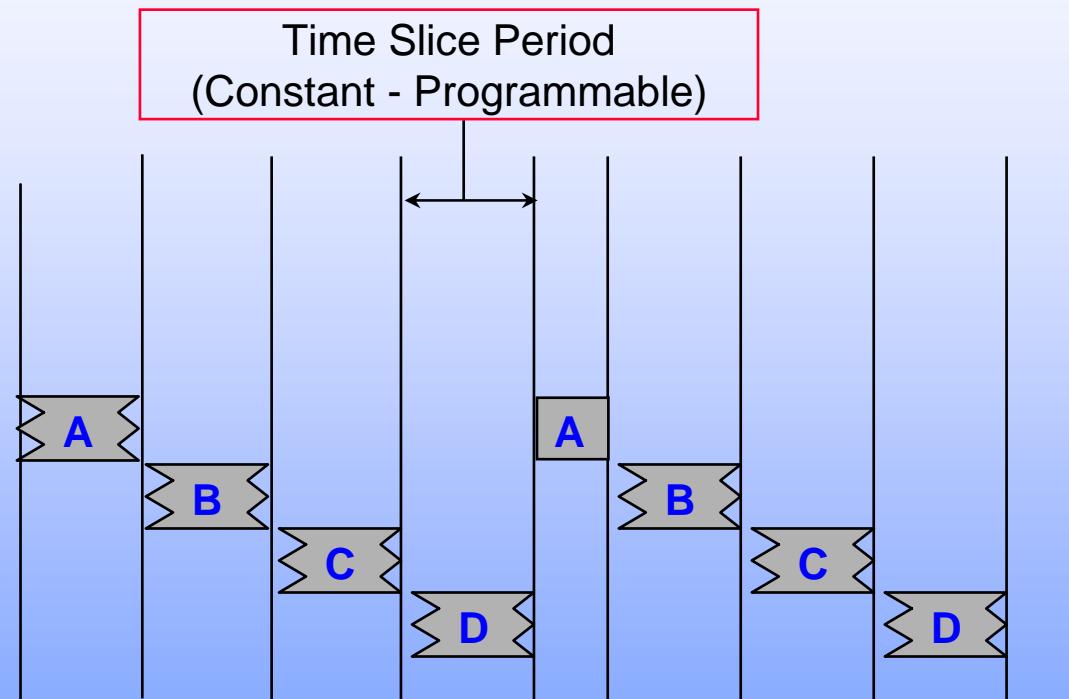
- 基于优先级的抢占式调度法作为缺省策略
  - 这种调度方法为每个任务指定不同的优先级。没有处于悬置或休眠态的最高优先级任务将一直运行下去。
  - 当更高优先级的任务由就绪态进入运行时，系统内核立即保存当前任务的上下文，切换到更高优先级的任务。
  - 当有内核调用或有中断到来时，内核重新调度
- 同时提供了时间片轮转调度



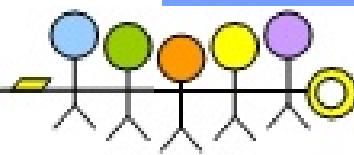
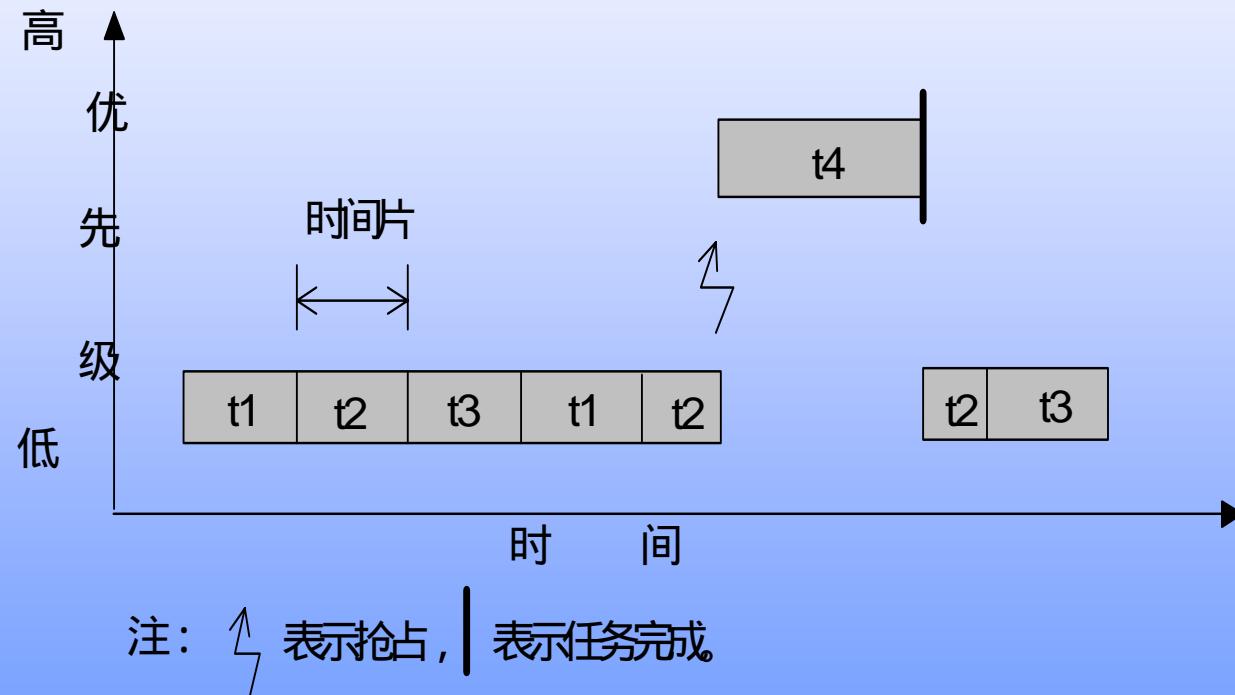
# 基于优先级的抢占式调度



# 时间片轮转调度

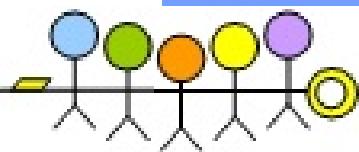


# 混合调度策略



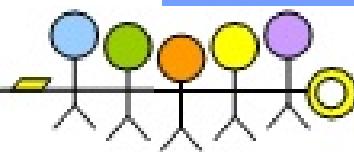
# 内核时间片

- 时间片的长度可由系统调用  
`KernelTimeSlice(ticks)`  
通过输入参数值来指定。  
当 `ticks` 为 0 时，时间片调度被关闭
- 基于优先级的抢占式调度可以发生在任何时候，时间片轮转调度只能在相同优先级的任务间每隔 `ticks` 发生一次。
- 在 VxWorks 系统中，可以调用函数 `kernelTimeSlice` 来使用时间片轮转调度。



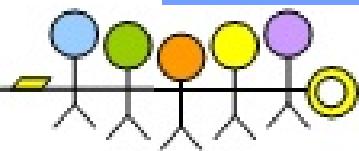
# VxWorks任务特性

- 所有的代码运行在同一地址空间。
- 任务能快速共享系统的绝大部分资源，同时有自己独立的上下文
- 所有的任务都运行在特权模式下



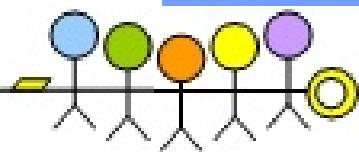
# 共享代码和重入（一）

- VxWorks提倡单个子程序或子程序库被多个不同的任务调用。例如printf。一个被多个任务调用的单个拷贝称为共享代码。
- VxWorks动态链接功能很容易实现代码共享。
- 共享代码必须是可重入的。
- VxWorks的I/O和驱动程序是可重入的。但是要求应用小心设计。对于缓冲 ( buffer)I/O,VxWorks推荐使用文件指针。



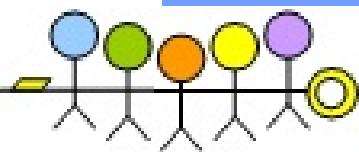
# 共享代码和重入（二）

- 大部分VxWorks程序使用下面的重入机制
  - 动态堆栈变量
    - 如果程序仅仅是纯代码，除了自己的动态堆栈变量外没有自己的数据，除了调用者以参数传进的数据外，没有其他数据。任务只在自己的堆栈内进行操作。
  - 由信号量保护的全局或静态变量
    - VxWorks的一些库封装对公共数据的访问。需要借用互斥机制
  - 任务变量
    - 一些程序可能会被多个任务同时调用，这些任务可能要求全局变量和静态变量有不同的值。这种情况下，VxWorks提供了所谓任务变量的机制，这种机制允许在任务上下文中增加4字节的变量，因此每次上下文交换时，改变量的值被保存。
    - 这种机制由taskVarLib库中的函数来提供。



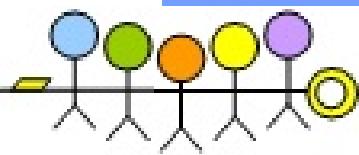
# 抢占禁止

- Wind内核可通过调用 `taskLock()` 和 `taskUnlock()` 来使调度器起作用和失效。当一个任务调用 `taskLock()` 使调度器失效，任务运行时没有基于优先级的抢占发生。然而，如果任务被阻塞或是悬置时，调度器从就绪队列中取出最高优先级的任务运行。当设置抢占禁止的任务解除阻塞，再次开始运行时，抢占又被禁止。这种抢占禁止防止任务的切换，但对中断处理不起作用。



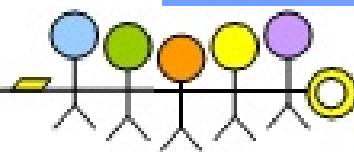
# POSIX调度

- schedPxDLib库提供了POSIX 1003.1b调度函数
- POSIX和Wind调度差异
  - POSIX调度算法应用在进程到进程基础之上的，而Wind调度算法则用于整个系统基础之上的，所有任务即可以使用轮转调度，也可以使用基于优先级的抢占调度。
  - POSIX的优先级编号方案与Wind的方案相反。POSIX中，优先数越高，优先级越高；Wind方案相反。为了解决这种冲突，用户需要通过将默认的全局变量posixPriorityNumbering的设置改委FALSE。
  - 使用POSIX调度程序，需要在配置VxWorks时，包含INCLUDE\_POSIX\_SCHED宏定义，系统将自动包含POSIX调度程序。



# VxWorks怎样满足实时性需求

- 多任务特性
- 基于抢占式多任务调度
- 快速的任务上下文切换
- 快速确定的系统响应
- 高效的任务间通讯机制



# Real-Time Multitasking

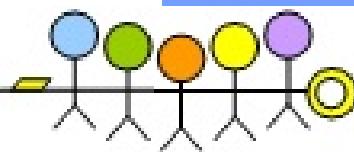
Introduction

Task Basics

Task Control

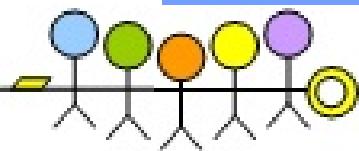
Error Status

System Tasks

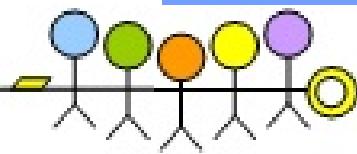
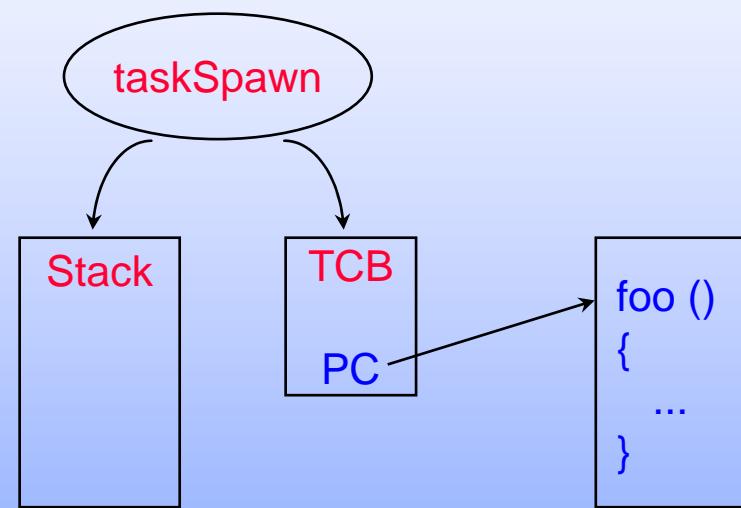


# Overview

- Low level routines to create and manipulate tasks are found in **taskLib**.
- A VxWorks task consists of:
  - A **stack** (for local storage of automatic variables and parameters passed to routines).
  - A **TCB** (for OS control)
- Do not confuse executable *code* with the task(s) which execute it.
  - Code is downloaded before tasks are spawned.
  - Several tasks can execute the same code (e.g., `printf()`).



# Creating a Task

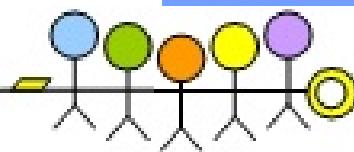


# Creating a Task

```
int taskSpawn (name, priority, options,  
stackSize, entryPt, arg1, ..., arg10)
```

name	Task name, if NULL gives a default name.
priority	Task priority, 0-255.
options	Task options e.g. VX_UNBREAKABLE.
stackSize	Size of stack to be allocated in bytes.
entryPt	Address of code to start executing (initial PC)
arg1, ..., arg10	Up to 10 arguments to entry point routine.

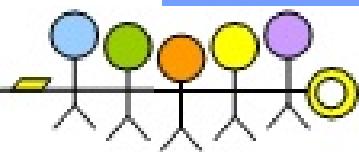
- Returns a **task id** or **ERROR** if unsuccessful.



# Task ID's

- Assigned by kernel when task is created.
- Unique to each task.
- Efficient 32-bit handle for task.
- May be reused after task exists.
- A task id of zero refers to task making call (self).
- Relevant **taskLib** routines:

<b>taskIdSelf()</b>	Get ID of calling task.
<b>taskIdListGet()</b>	Fill array with ID's of all existing tasks.
<b>taskIdVerify()</b>	Verify a task ID is valid.



# Task Names

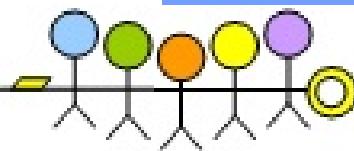
- Provided for human convenience.
  - Typically used only from the shell (during development).
  - Within programs, use task Ids.
- By convention, start with a **t**.
  - Promotes interpretation as a task name.
  - Default is an ascending integer following a t.
- Doesn't have to be unique (but usually is).
- Relevant taskLib routines.

**taskName()**

Get name from tid.

**taskNameTid()**

Get tid from task name.

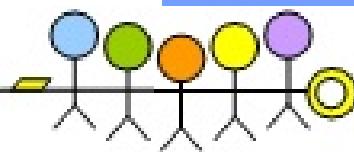


# Task Priorities

- Range from 0 (highest) to 255 (lowest).
- No hard rules on how to set priorities. There are two (often contradictory) “rules of thumb”:
  - More important = higher priority.
  - Shorter deadline = higher priority.
- Can manipulate priorities dynamically with:

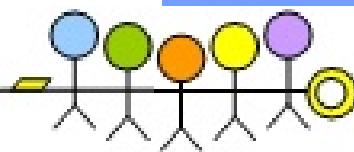
`taskPriorityGet (tid, &priority)`

`taskPrioritySet (tid, priority)`



# Task Stacks

- Allocated from memory pool when task is created.
- Fixed size after creation.
- The kernel reserves some space from the stack, making the stack space actually available slightly less than the stack space requested.
- Exceeding stack size (“stack crash”) causes unpredictable system behavior.

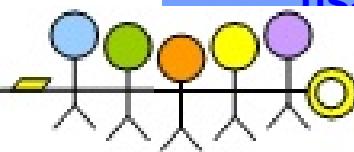
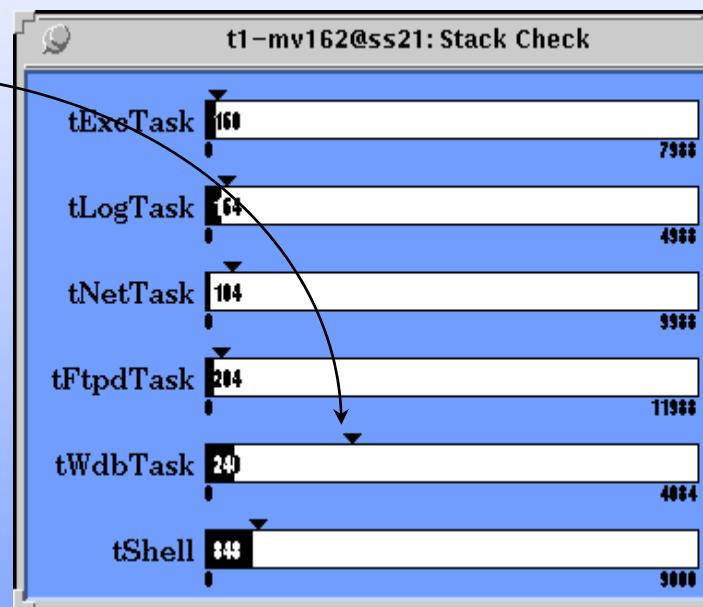


# Stack Overflows

- To check for a stack overflow use the Browser.
  - Press the check-stack button:
  - Examine the high-water mark indicator in the stack display window.

High-water Mark  
Indicator (UNIX)

Note: In the PC Browser Stack Check Window, the high water makr triangles are not present. Instead, the number displayed *inside* each stack bar is that stack's high water mark. The filled portion of the bar indicates the current stack usage graphically.



# Task Options

- Can be bitwise or'ed together when the task is created.

VX\_FP\_TASK

Add floating point support.

VX\_NO\_STACK\_FILL

Don't fill stack with 0xee's.

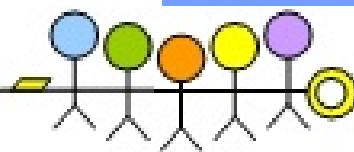
VX\_UNBREAKABLE

Disable breakpoints.

VX DEALLOC\_STACK

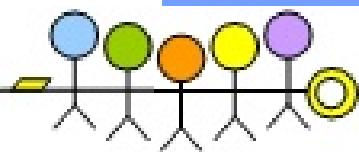
Deallocate stack and TCB  
when task exists (automatically  
set for you).

- Use **taskOptionsGet()** to inquire about a task's options.
- Use **taskOptionsSet()** to unset VX DEALLOC\_STACK.



## Task Creation

- During time critical code, task creation can be unacceptably time consuming.
- To reduce creation time, a task can be spawned with the `VX_NO_STACK_FILL` option bit set.
- Alternatively, spawn a task at system start-up, which blocks immediately, and waits to be made ready when needed.



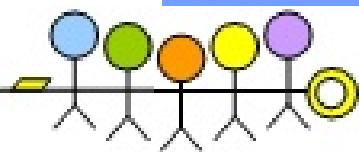
# Task Deletion

## **taskDelete(tid)**

- Deletes the specified task.
- Dealлокates the TCB and stack.

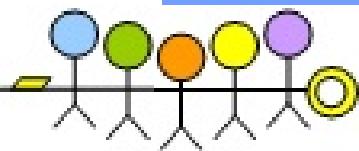
## **exit (code)**

- Analogous to a **taskDelete()** of self.
- code parameter gets stored in the TCB field *exitCode*.
- TCB may be examined for post-mortem debugging by
  - Unsetting the `VX DEALLOC STACK` option or,
  - Using a delete hook.



# Resource Reclamation

- Contrary to the philosophy of sharing system resources among all tasks.
- Can be an expensive process, which must be the application's responsibility.
- TCB and stack are the only resources automatically reclaimed.
- Tasks are responsible for cleaning up after themselves.
  - Deallocating memory.
  - Releasing locks to system resources.
  - Closing files which are open.
  - Deleting child / client tasks when parent / server exits.



# Real-Time Multitasking

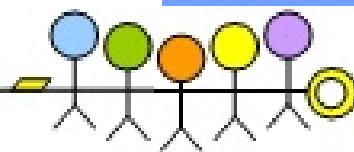
Introduction

Task Basics

Task Control

Error Status

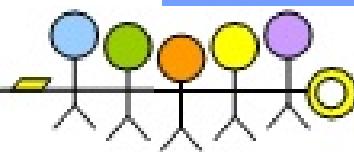
System Tasks



# Task Restart

## taskRestart (tid)

- Task is terminated and respawned with original arguments and tid.
- Usually used for error recovery.



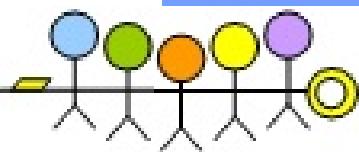
# Task Suspend / Resume

## taskSuspend (tid)

- Makes task ineligible to execute.
- Can be added to pended or delayed state.

## taskResume (tid)

- Removes suspension.
- Usually used for debugging and development purposes.



# Task Delay

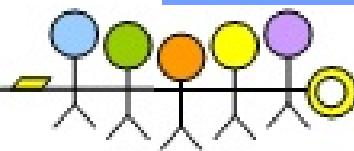
- To delay a task for a specified number of system clock ticks.

**STATUS taskDelay (ticks)**

- To poll every 1/7 second:

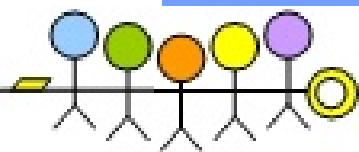
```
FOREVER
{
    taskDelay (sysClkRateGet() / 7);
    ...
}
```

- Accurate only if clock rate is a multiple of seven ticks / seconds.
- Can suffer from “drift.”
- Use **sysClkRateSet()** to change the clock rate.



# Reentrancy and Task Variables

- If tasks access the same global or static variables, the resource can become corrupted (called a *race condition*).
- Possible Solutions:
  - Use only stack variables in applications.
  - Protect the resource with a semaphore.
  - Use task variables to make the variable private to a task
- Task Variables cause a 32-bit value to be saved and restored on context switchs, like a register.
- Caveat: task variables increase context switch times.
- See the **taskVarLib** manual pages for details.

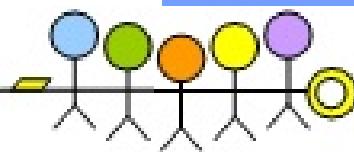


# Task Hooks

- User-defined code can be executed on every context switch, at task creation, or at task deletion:

**taskSwitchHookAdd ()**  
**taskCreateHookAdd ()**  
**taskDeleteHookAdd ()**

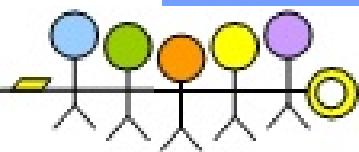
- VxWorks uses a switch hook to implement task variables.
- See manual pages on **taskHookLib** for details.



# Task Information

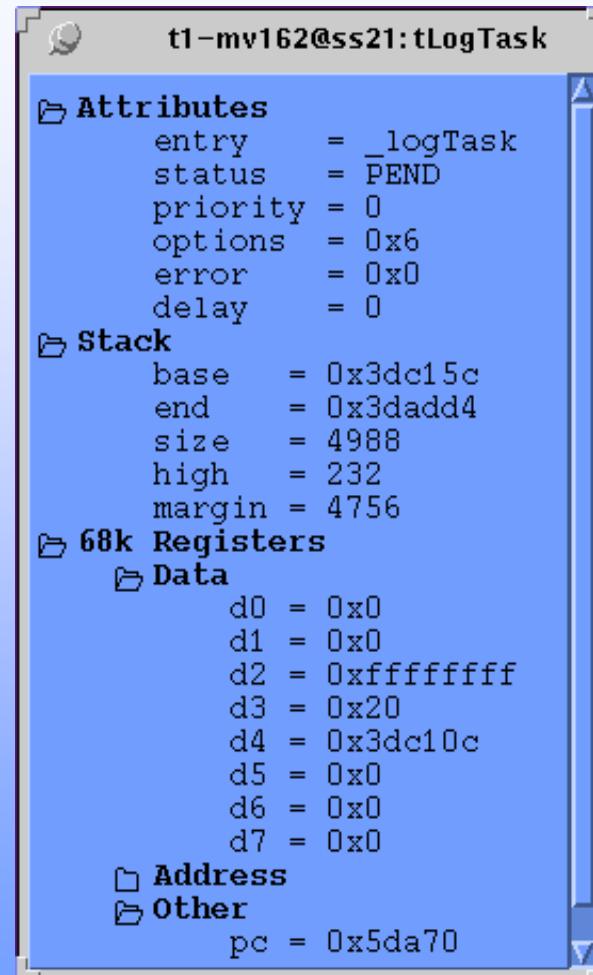
## ti (taskNameOrId)

- Like `i()`, but also displays:
  - Stack information
  - Task options
  - CPU registers
  - FPU registers (if the VX\_FP\_TASK option bit is set)
- Can also use `show ()`:  
`-> show (tNetTask, 1)`



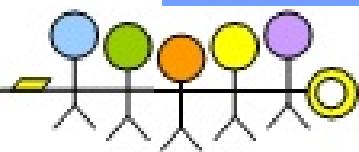
# Task Browser

- To obtain information about a specific task, click on the task's summary line in the main target browser.



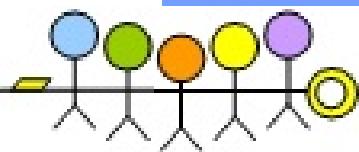
t1-mv162@ss21:tLogTask

- Attributes
  - entry = \_logTask
  - status = PEND
  - priority = 0
  - options = 0x6
  - error = 0x0
  - delay = 0
- Stack
  - base = 0x3dc15c
  - end = 0x3dadd4
  - size = 4988
  - high = 232
  - margin = 4756
- 68k Registers
  - Data
    - d0 = 0x0
    - d1 = 0x0
    - d2 = 0xffffffff
    - d3 = 0x20
    - d4 = 0x3dc10c
    - d5 = 0x0
    - d6 = 0x0
    - d7 = 0x0
  - Address
  - Other
    - pc = 0x5da70



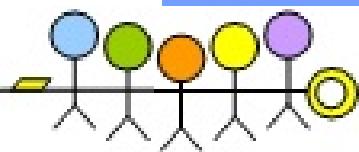
# What is POSIX ?

- Originally, an IEEE committee convened to create a standard interface to UNIX for:
  - Increased portability.
  - Convenience.
- VxWors supports almost all of the 1003.1b POSIX Real-time Extensions.
- The POSIX real-time extensions are based on implicit assumptions about the UNIX process model which do not always hold in VxWorks. In VxWorks,
  - Context switch times are very fast.
  - Text, data, and bss are stored in a common, global address space.



# What does VxWorks support ?

<u>Library</u>	<u>Description</u>
aioPxLib	Asynchronous I/O
semPxLib	POSIX Semaphores
mqPxLib	POSIX Message Queues
mmanPxLib	POSIX Memory Management
schedPxLib	POSIX Scheduler Interface
sigLib	POSIX Signals
timerLib, clockLib	POSIX Timer/Clock Interface
dirLib	File/Directory Information



# Real-Time Multitasking

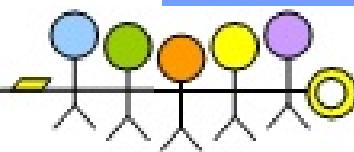
Introduction

Task Basics

Task Control

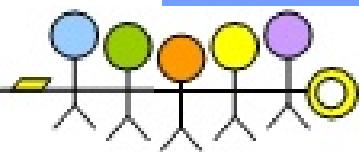
Error Status

System Tasks

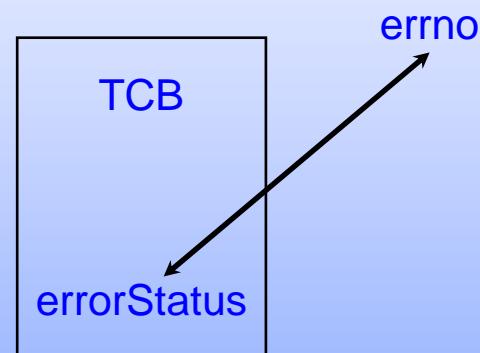


## Error Status

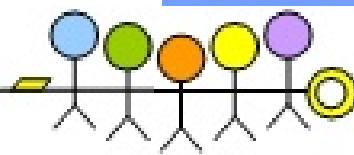
- VxWorks里使用一个全局整型变量**errno**来描述错误信息
  - 程序执行过程中我们可以设置并调用一些函数例程来检测错误信息，并针对错误信息设置相应的错误号
  - 然后调用一些函数例程检测错误号，当程序执行异常时可以根据错误号发现相应的错误



# Errno and Context Switches



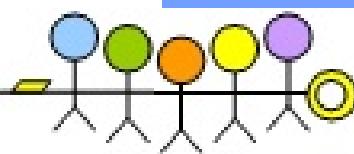
At each context switch, the kernel saves and restores the value of *errno*.



# Setting Errno

- Lowest level routine to detect an error sets *errno* and returns ERROR:

```
STATUS myRoutine ()  
{  
    ...  
    if (myNumFlurbishes >= MAX_FLURBISH)  
    {  
        errno = s_myLib_TOO_MANY_FLURBISHES;  
        return (ERROR);  
    }  
    ...  
    pMem=malloc(sizeof(myStruct));  
    if ( pMem== NULL)  
    {  
        /* malloc() sets errno - don't redefine it */  
        return (ERROR);  
    }  
    ...  
}
```

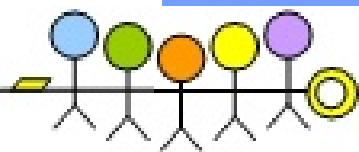


# Examining Errno

- Examine *errno* to find out why a routine failed.

```
if (reactorOk() == ERROR)
{
    switch (errno)
    {
        case S_rctorLib_TEMP_DANGER_ZONE:
            startShutDown ();
            break;
        case S_rctorLib_TEMP_CRITICAL_ZONE:
            logMsg ("Run!");
            break;
        case S_rctorLib_LEAK_POSSIBLE:
            checkVessel ();
            break;
        default: startEmergProc ();
    }
}
```

- *errno* is only valid after an error occurs.

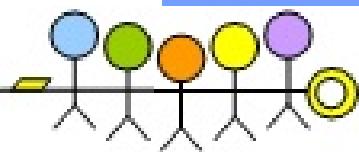


# Interpreting Errno

- VxWorks uses the 32-bit value *errno* as follows:

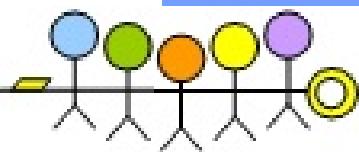


- Module numbers are defined in **vwModNum.h**.
- Each module defines its own error numbers in its header file.
- For example, an *errno* of 0x110001 would be:
  - Module number 0x11 (define in **vwModNum.h** to be **memLib**) and
  - Error number 0x01 (defined in **memLib.h** to be “not enough memory”).



# Error Messages

- VxWorks uses an error symbol table (*statSymTbl*) to convert error numbers to error messages.
- To print the error message for the current task's error status to **STD\_ERR**:  
`-> perror ("darn")  
darn: S_netDrv_NO SUCH_FILE_OR_DIR`
- To print the error message associated with an error number to the WindSh console :  
`-> printErrno (0x110001)  
S_memLib_NOT_ENOUGH_MEMORY`



# User-Defined Error Codes

To get *perror()* to print your error messages:

1. Modify **vwModNum.h** to reserve a module number:

```
#define M_myLib           (501 << 16)
```

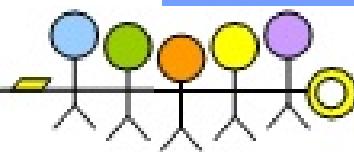
2. Define error macros in your header file (which must be in the **wind/target/h** directory):

```
#define S_myLib_BAD_STUFF (M_myLib | 1)
```

3. Rebuild the system error table:

- Go to the **wind/target/src/usr** directory.
- remove **statTbl.c**
- Execute the makefile.

4. Rebuild VxWorks with the new error table built in.



# Real-Time Multitasking

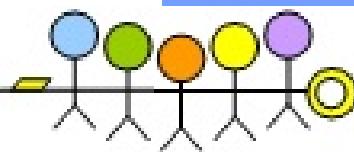
Introduction

Task Basics

Task Control

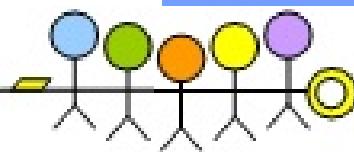
Error Status

System Tasks



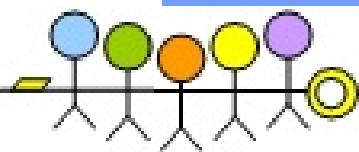
# System Tasks

<u>Task Name</u>	<u>Priority</u>	<u>Function</u>
<i>tUsrRoot</i> system,	0	Initial task. Configures the spawns the shell, then exits.
<i>tLogTask</i>	0	Message logging.
<i>tExecTask</i>	0	Exception handling.
<i>tWdbTask</i>	3	WDB running agent.
<i>tNetTask</i>	50	Task-level network functions.
<i>tFtpdTask</i>	55	FTP server.



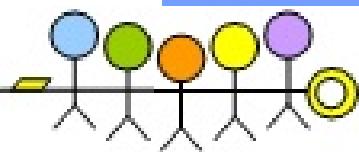
# Summary

- Real-time multitasking requirements:
  - Preemptive priority-based scheduler.
  - Low overhead.
- Task properties stored in task's TCB.
  - OS control information (priority, stack size, options, state, ...).
  - CPU context (PC, SP, CPU registers, ...).
- `taskSpawn()` lets you specify intrinsic properties :
  - Priority
  - Stack size
  - Options

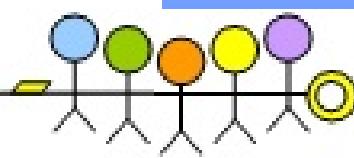


# Summary

- Task manipulation routines in **taskLib**:
  - taskSpawn / taskDelete
  - taskDelay
  - taskSuspend / taskResume
- Task information
  - ti / show
  - Task Browser
- Additional task context :
  - errno
  - Task variables
  - Task hooks

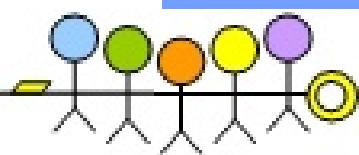


# Semaphores



# 概述

- 信号量是最早出现的用来解决进程同步与互斥问题的机制，包括一个称为信号量的变量及对它进行的两个原语操作。
- 信号量作为任务间同步和互斥的机制，是快速和高效的，它们除了被应用在开发设计过程中外，还被广泛地应用在 VxWorks 高层应用系统中。
  - 二进制信号量
  - 计数信号量
  - 互斥信号量
  - POSIX 信号量



# 信号量的类型定义

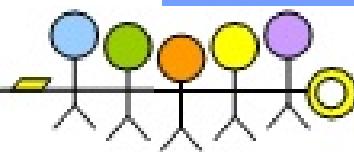
- 每个信号量至少须记录两个信息：信号量的值和等待该信号量的进程队列。它的类型定义如下：（用类PASCAL语言表述）

```
semaphore = record  
    value: integer;  
    queue: ^PCB;  
end;
```

其中PCB是进程控制块，是操作系统为每个进程建立的数据结构。

s.value>=0时，s.queue为空；

s.value<0时，s.value的绝对值为s.queue中等待进程的个数；

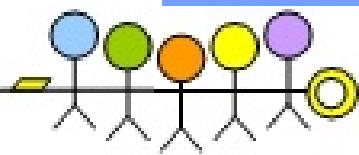


# PV原语（一）

- 对一个信号量变量可以进行两种原语操作：p操作和v操作，定义如下：

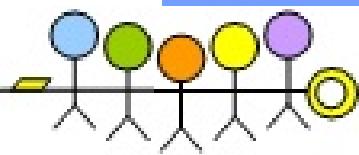
```
procedure p(var s:samephore);
{
    s.value=s.value-1;
    if (s.value<0) asleep(s.queue);
}

procedure v(var s:samephore);
{
    s.value=s.value+1;
    if (s.value<=0) wakeup(s.queue);
}
```



# PV原语（二）

- 其中用到两个标准过程：  
asleep(s.queue);执行此操作的进程的PCB进入s.queue尾部，进程变成等待状态  
wakeup(s.queue);将s.queue头进程唤醒插入就绪队列  
s.value初值为1时，可以用来实现进程的互斥。  
p操作和v操作是不可中断的程序段，称为原语。如果将信号量看作共享变量，则pv操作为其临界区，多个进程不能同时执行，一般用硬件方法保证。一个信号量只能置一次初值，以后只能对之进行p操作或v操作。
- 信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点。

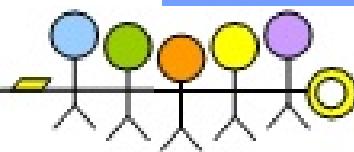


# Semaphores

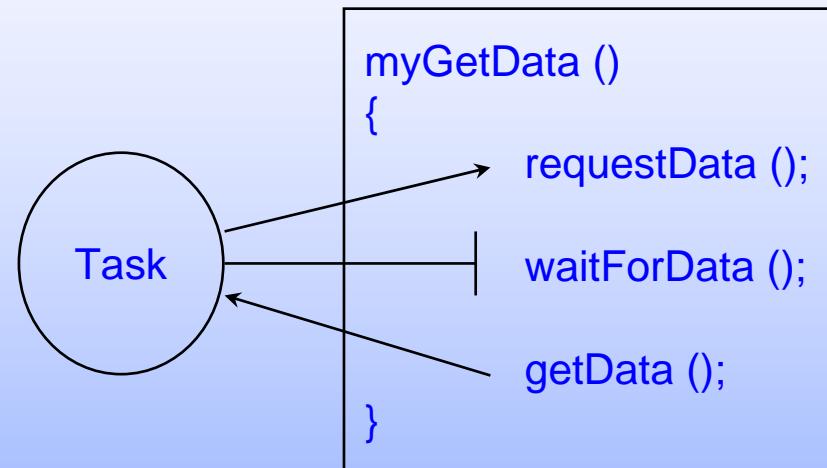
Overview

Binary Semaphores and Synchronization

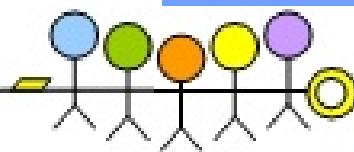
Mutual Exclusion



# The synchronization Problem

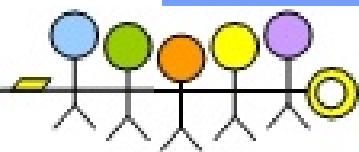


- Task may need to wait for an event to occur.
- Busy waiting (i.e., polling) is inefficient.
- Pending until the event occurs is better.



# The Synchronization Solution

- Create a binary semaphore for the event.
- Binary semaphores exist in one of two states:
  - Full (event has occurred).
  - Empty (event has not occurred).
- Task waiting for the event calls **semTake()**, and blocks until semaphore is given.
- Task or interrupt service routine detecting the event calls **semGive()**, which unblocks the waiting task.



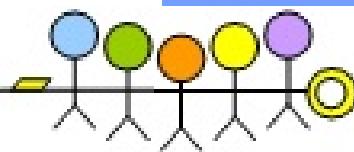
# Binary Semaphores

**SEM\_ID semBCreate (options, intialState)**

options      Sepcify queue type (**SEM\_Q\_PRIORITY** or  
**SEM\_Q\_FIFO**) for tasks pended on this  
semaphore.

initialState      Initialize semaphore to be available  
(**SEM\_FULL**) or unavailable (**SEM\_EMPTY**).

- Semaphores used for synchronization are typically initialized to **SEM\_EMPTY** (event has not occurred).
- Returns a **SEM\_ID**, or **NULL** on error.



# Taking a Semaphore

STATUS **semTake** (semId, timeout)

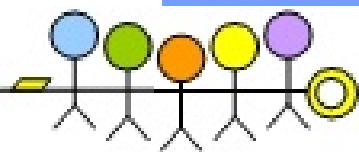
semId

The SEM\_ID returned from **semBCreate()**.

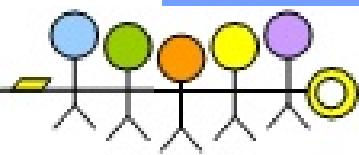
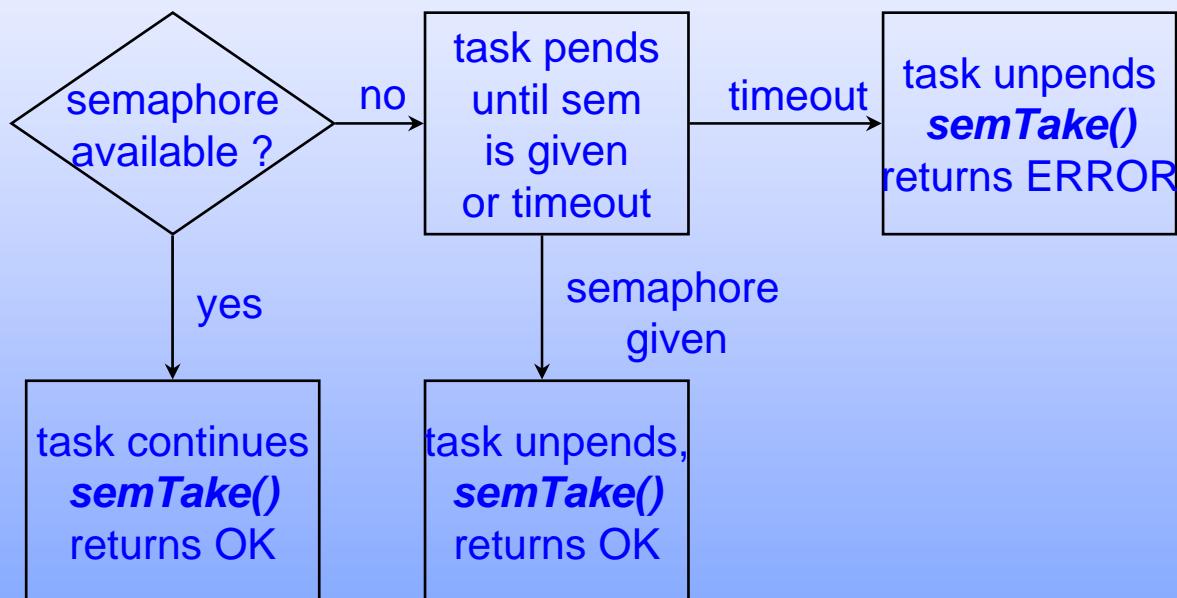
timeout

Maximum time to wait for semaphore. Value can be clock ticks, WAIT\_FOREVER, or NO\_WAIT.

- Can pend the task until either
  - Semaphore is given or
  - Timeout expires.
- Semaphore left **unavailable**.
- Returns OK if successful, ERROR on timeout (or invalid *semId*).



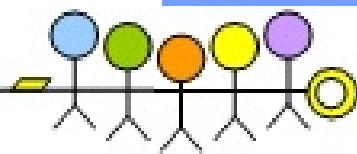
# Taking a Binary Semaphore



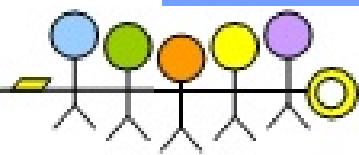
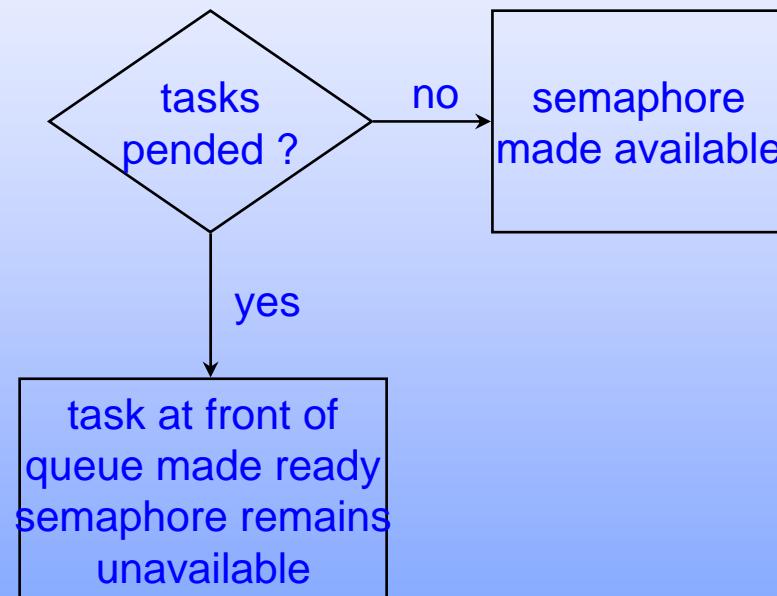
# Giving a Semaphores

STATUS **semGive** (*semId*)

- Unblocks a task waiting for *semId*.
- If no task is waiting, make *semId* available.
- Returns OK, or ERROR if *semId* is invalid.



# Giving a Binary Semaphore



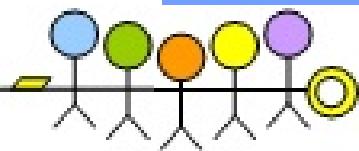
# Information Leakage

- Fast event occurrences can cause lost information.
- Suppose a VxWorks task (priority=100) is executing the following code, with semId initially unavailable:

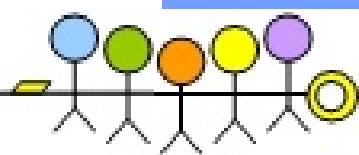
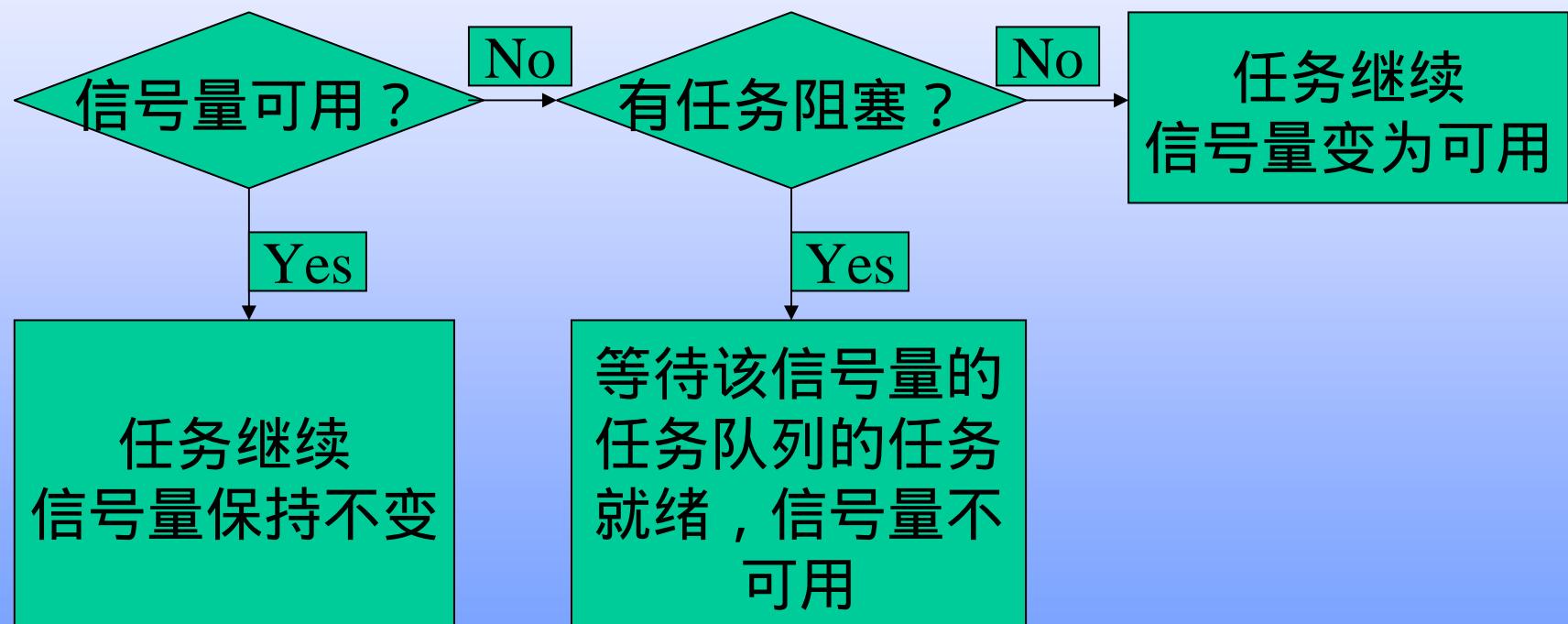
```
FOREVER
{
    semTake (semId, WAIT_FOREVER);
    printf ("Got the semaphore\n");
}
```

What would happen in the scenarios below ?

1. -> repeat (1, semGive, semId);
2. -> repeat (2, semGive, semId);
3. -> repeat (3, semGive, semId);



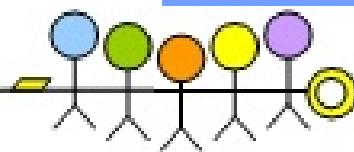
# 释放信号量图示



# Synchronizing Multiple Tasks

STATUS **semFlush** (semId)

- Unblocks all tasks waiting for semaphore.
- Does not affect the state of a semaphore.
- Useful for synchronizing actions of multiple tasks.

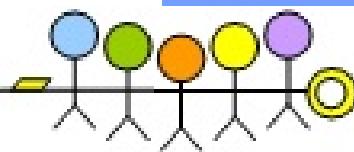


# Semaphores

Overview

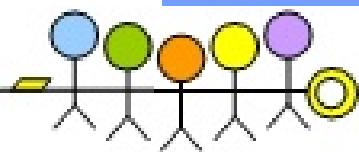
Binary Semaphores and Synchronization

Mutual Exclusion



# Mutual Exclusion Problem

- Some resources may be left inconsistently if accessed by more than one task simultaneously.
  - Shared data structures.
  - Shared files.
  - Shared hardware devices.
- Must obtain exclusive access to such a resource before using it.
- If exclusive access is not obtained, then the order in which tasks execute affects correctness.
  - We say a *race condition* exists.
  - Very difficult to detect during testing.
- Mutual exclusion cannot be compromised by priority.



# Race Condition Example

```
tTask1                                tTask2
  ┌────────────────────────────────────────┐
  | 1   char myBuf(BU_SIZE);           /* store data here
  | 2 data */ int myBufIndex = -1;      /* Index of last
  | 3
  | 4     myBufPut (char ch)
  | 5     {
  | 6         myBufIndex++;
  | 7         myBuf [myBufIndex] = ch;
  | 8     }
```

myBufIndex++

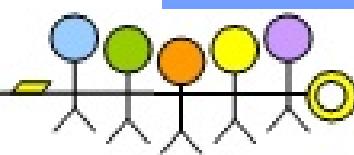
myBuf [my

BufIndex] = 'b'

myBufIndex++

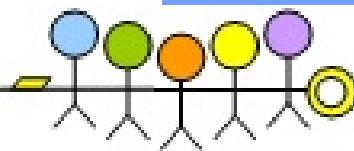
myBuf [myBufIndex] = 'a'

.....



## Solution Overview

- Create a mutual exclusion semaphore to guard the resource.
- Call ***semTake()*** before accessing the resource; call ***semGive()*** when done.
  - ***semTake()*** will block until the semaphore (and hence the resource) becomes available.
  - ***semGive()*** releases the semaphore (and hence access to the resource).



# Creating Mutual Exclusion Semaphores

SEM\_ID **semMCreate** (options)

- *options* can be :

queue specification

**SEM\_Q\_FIFO** or  
**SEM\_Q\_PRIORITY**

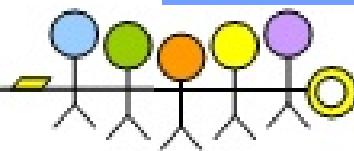
deletion safety

**SEM\_DELETE\_SAFE**

priority inheritance

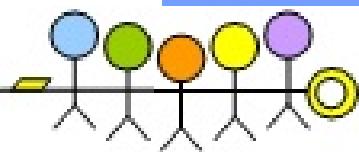
**SEM\_INVERSION\_SAFE**

- Initial state of semaphore is **available**.

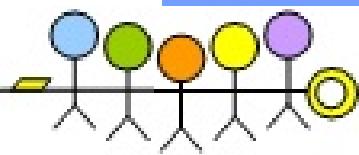
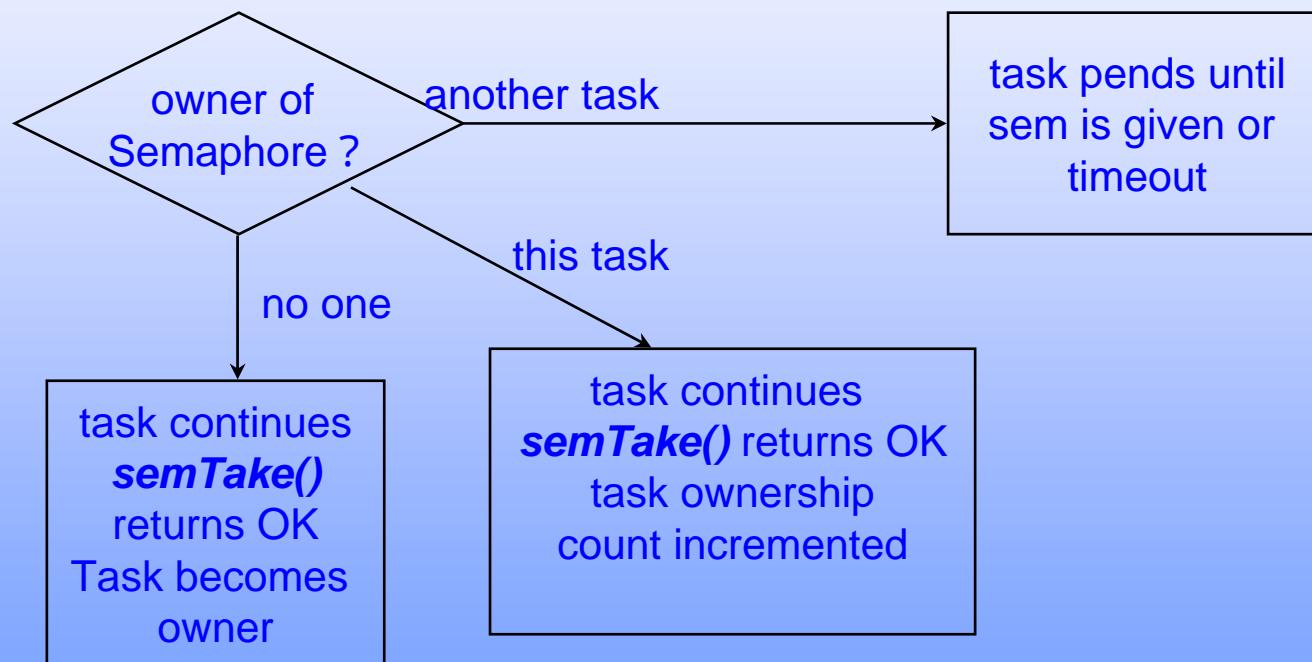


# Mutex Ownership

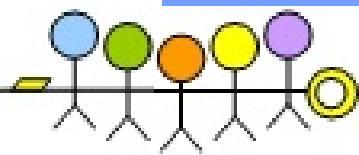
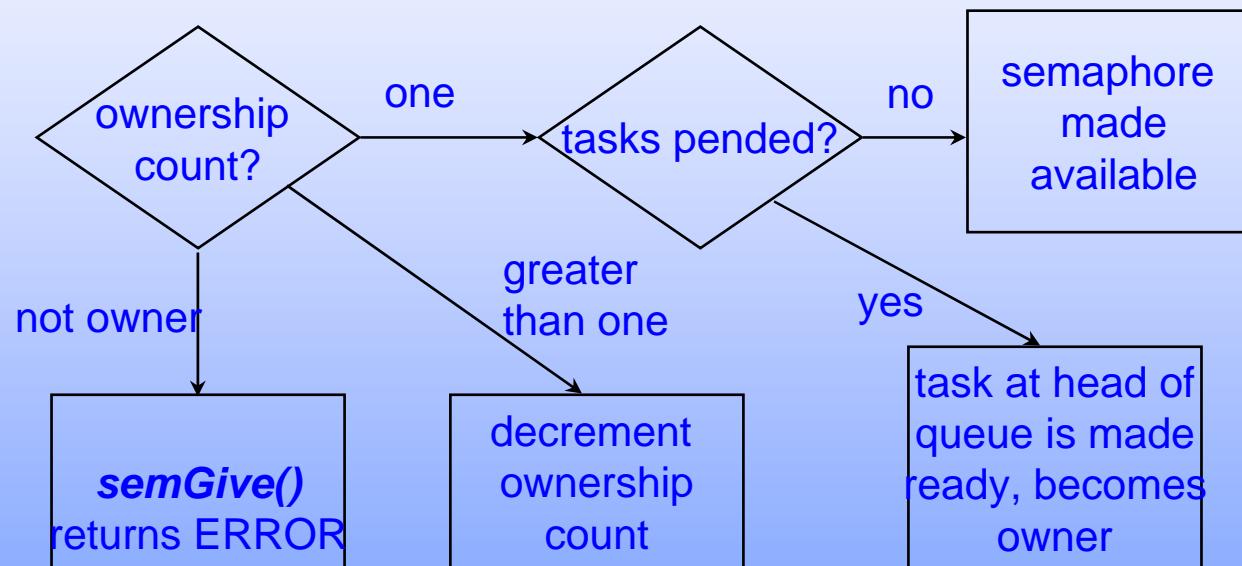
- A task which takes a mutex semaphore “owns” it, so that no other task can give this semaphore.
- Mutex semaphores can be taken recursively.
  - The task which owns the semaphore may take it more than once.
  - Must be given same number of times as taken before it will be released.
- Mutual exclusion semaphores cannot be used in an interrupt service routine.



# Taking a Mutex Semaphore

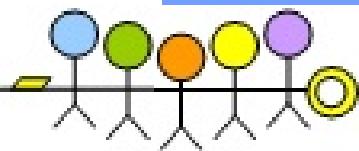


# Giving a Mutex Semaphore



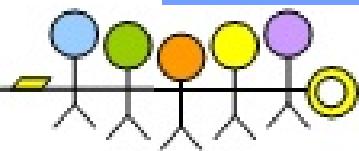
# Code Example - Solution

```
1 #include "vxWorks.h"
2 #include " semLib.h"
3
4 LOCAL char myBuf[BUF_SIZE]; /* Store data here */
5 LOCAL int myBufIndex = -1; /* Index of last data */
6 LOCAL SEM_ID mySemId;
7
8 void myBufInit ()
9 {
10     mySemId = semNCreate (SEM_Q_PRIORITY |
11                         SEM_INVERSION_SAFE |
12                         SEM_DELETE_SAFE );
13 }
14
15 void myBufPut (char ch)
16 {
17     semTake(mySemId, WAIT_FOREVER);
18     myBufIndex++;
19     myBuf[myBufIndex] = ch;
20     semGive (mySemId);
21 }
```



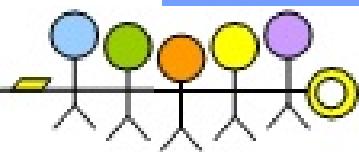
## Deletion Safety

- Deleting a task which owns a semaphore can be catastrophic.
  - data structures left inconsistent.
  - semaphore left permanently unavailable.
- The deletion safety option prevents a task from being deleted while it owns the semaphore.
- Enabled for mutex semaphores by specifying the SEM\_DELETE\_SAFE option during ***semMCreate( )***.

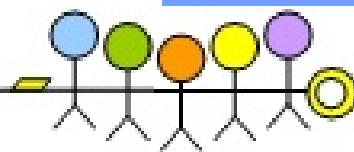
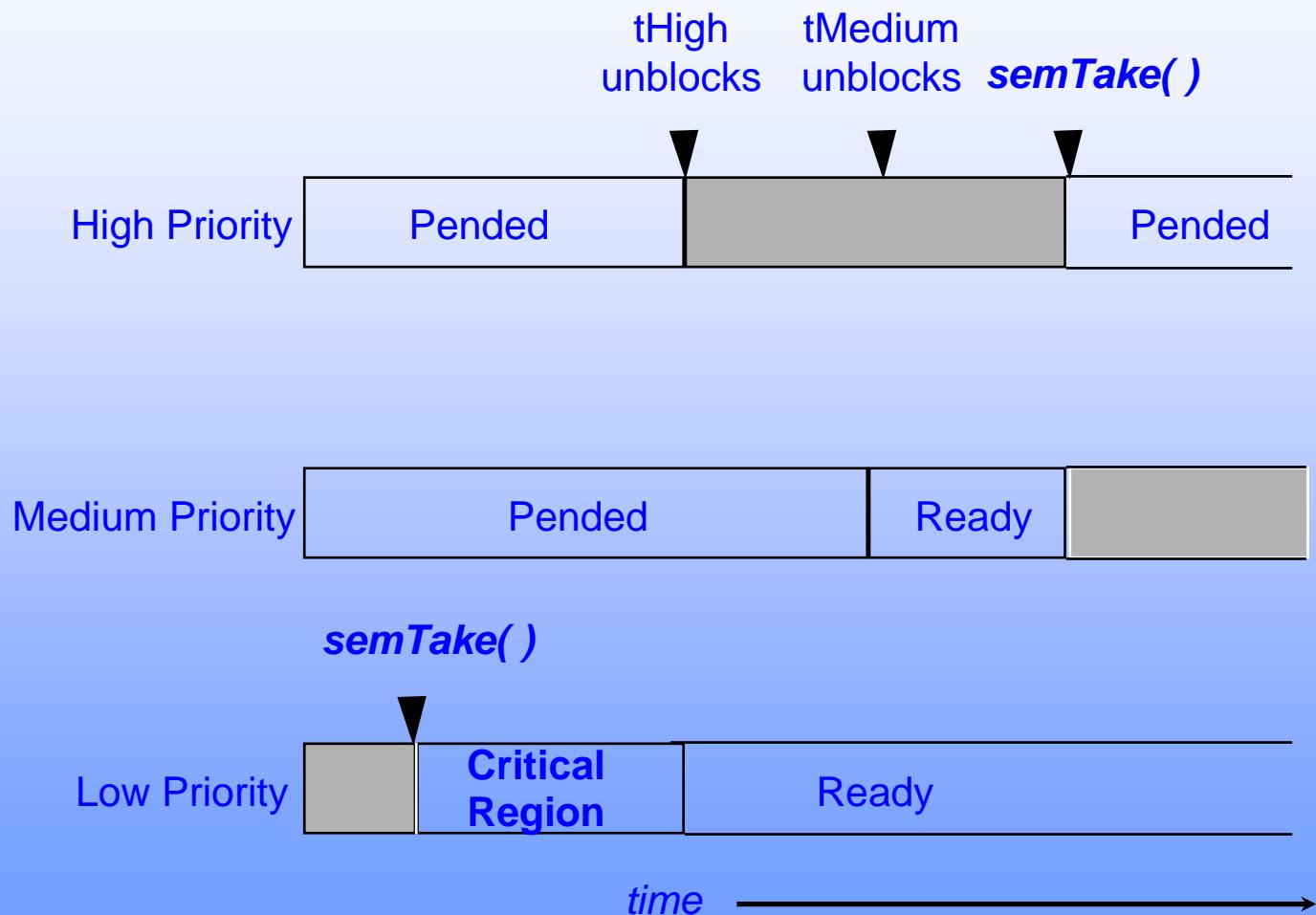


# 安全删除

- wind内核提供防止任务被意外删除的机制。通常，一个执行在临界区或访问临界资源的任务要被特别保护。
- 我们设想下面的情况：一个任务获得一些数据结构的互斥访问权，当它正在临界区内执行时被另一个任务删除。由于任务无法完成对临界区的操作，该数据结构可能还处于被破坏或不一致的状态。而且，假想任务没有机会释放该资源，那麼现在其他任何任务现在就不能获得该资源，资源被冻结了。
- 任何要删除或终止一个设定了删除保护的任务的任务将被阻塞。当被保护的任务完成临界区操作以后，它将取消删除保护以使自己可以被删除，从而解阻塞删除任务。

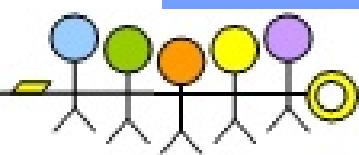


# Priority Inversion



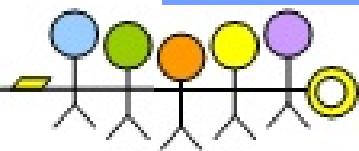
# 优先级逆转/优先级继承

- 优先级逆转发生在一个高优先级的任务被强制等待一段不确定的时间以便一个较低优先级的任务完成执行。
- T1，T2和T3分别是高、中、低优先级的任务。T3通过拥有信号量而获得相关的资源。当T1抢占T3，为竞争使用该资源而请求相同的信号量的时候，它被阻塞。如果我们假设T1仅被阻塞到T3使用完该资源为止，情况并不是很糟。毕竟资源是不可被抢占的。然而，低优先级的任务并不能避免被中优先级的任务抢占，一个抢占的任务如T2将阻止T3完成对资源的操作。这种情况可能会持续阻塞T1等待一段不可确定的时间。这种情况成为优先级逆转，因为尽管系统是基于优先级的调度，但却使一个高优先级的任务等待一个低优先级的任务完成执行。
- 互斥信号量有一个选项允许实现优先级继承的算法。优先级继承通过在T1被阻塞期间提升T3的优先级到T1解决了优先级逆转引起的问题。这防止了T3，间接地防止T1，被T2抢占。通俗地说，优先级继承协议使一个拥有资源的任务以等待该资源的任务中优先级最高的任务的优先级执行。当执行完成，任务释放该资源并返回到它正常的或标准的优先级。因此，继承优先级的任务避免了任何中间优先级的任务抢占。

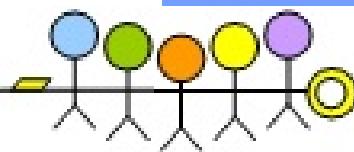
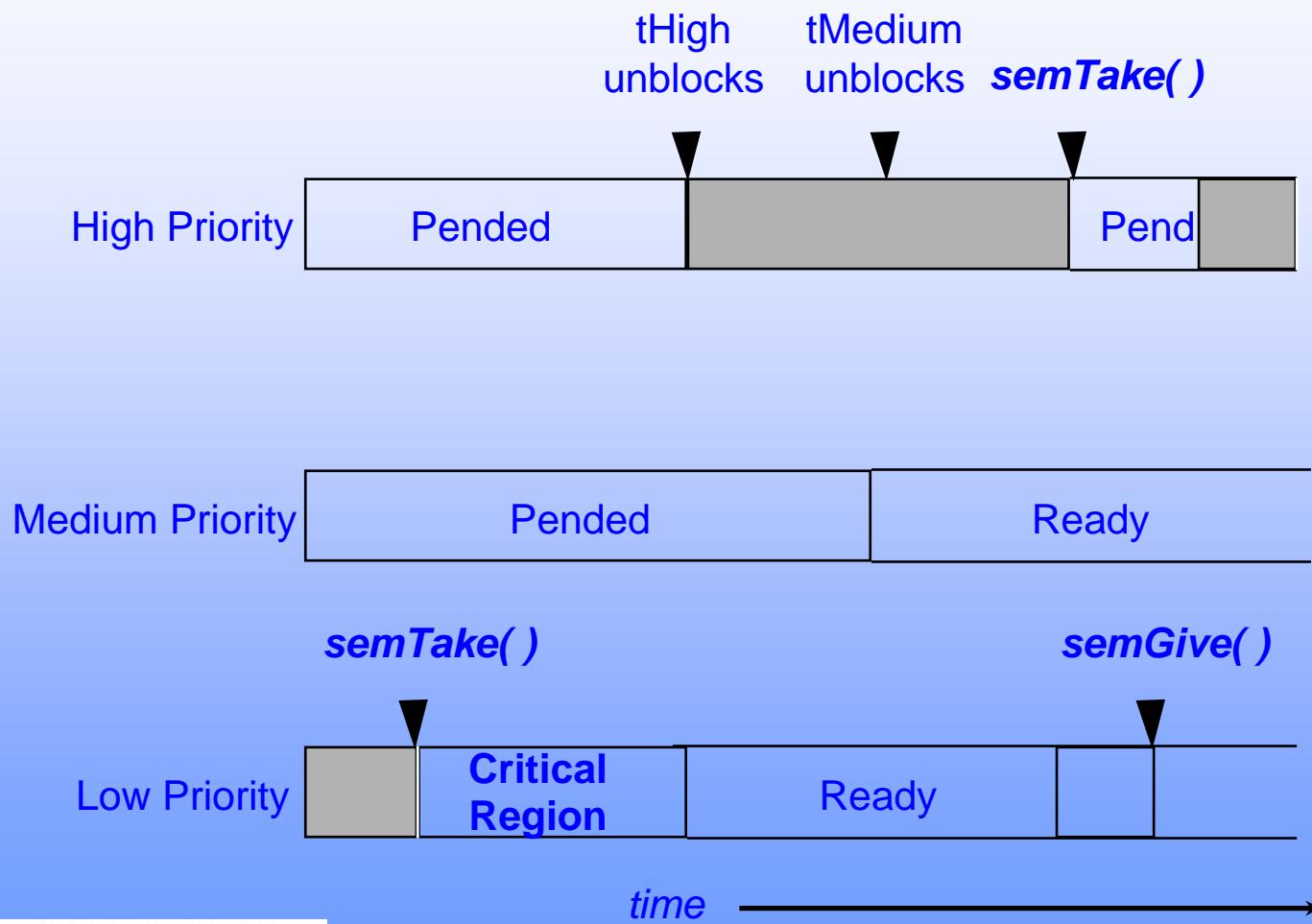


# Priority Inheritance

- Priority inheritance algorithm solves priority inversion problem.
- Task owning a mutex semaphore is elevated to priority of highest priority task waiting for that semaphore.
- Enabled on mutex semaphore by specifying the **SEM\_INVERSION\_SAFE** option during ***semMCreate( )***.
- Must also specify **SEM\_Q\_PRIORITY** (**SEM\_Q\_FIFO** is incompatible with **SEM\_INVERSION\_SAFE**).

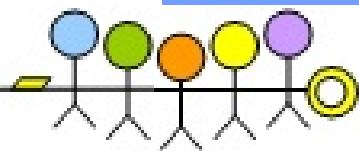


# Priority Inversion Safety

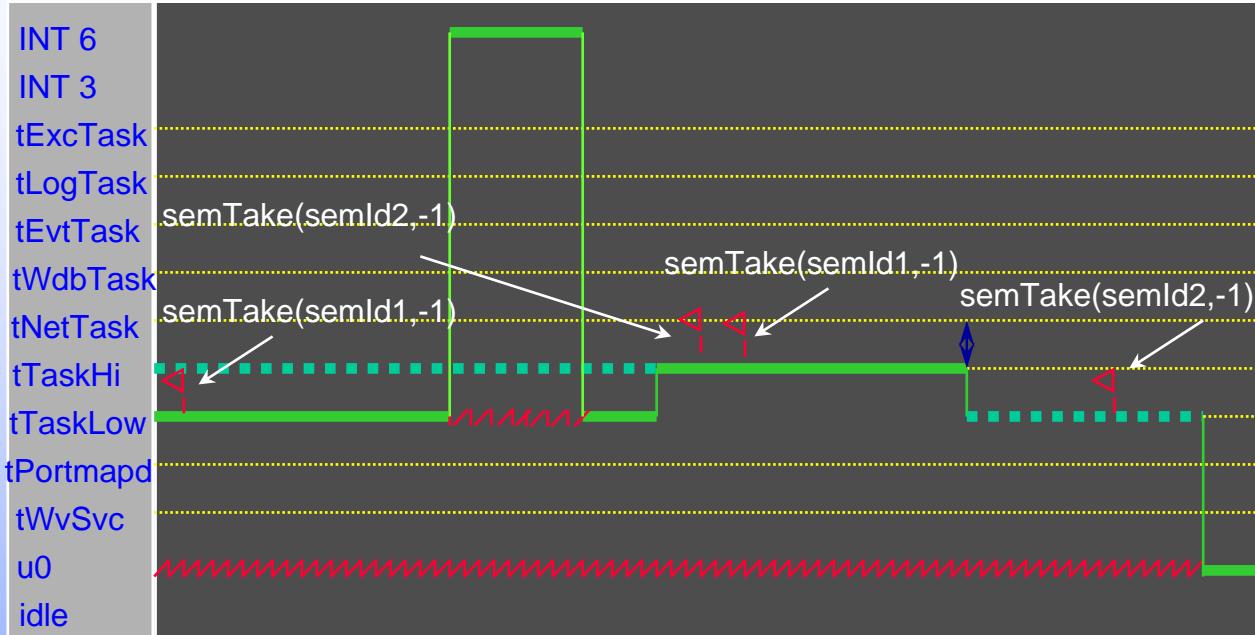


# Avoiding Mistakes

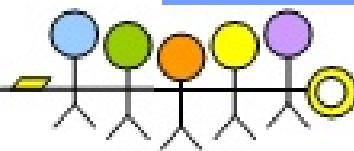
- It is easy to misuse mutex semaphores, since you must protect *all* accesses to the resource.
- To prevent such a mistake
  - Write a library of routines to access the resource.
  - Initialization routine creates the semaphore.
  - Routines in this library obtain exclusive access by calling **semGive( )** and **semTake( )**.
  - All uses of the resource are through this library.



## Caveat - Deadlocks

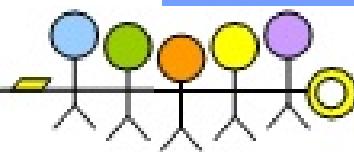


- A deadlock is a race condition associated with the taking of multiple mutex semaphores.
- Very difficult to detect during testing.



## Other Caveats

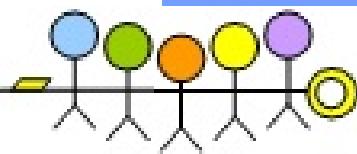
- Mutual exclusion semaphores can not be used at interrupt time.  
This issue will be discussed later in the chapter.
- Keep the critical region (code between **semTake( )** and **semGive( )**) short.



## Common Routines

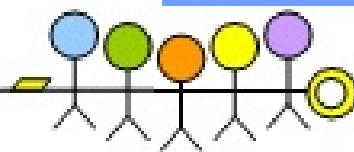
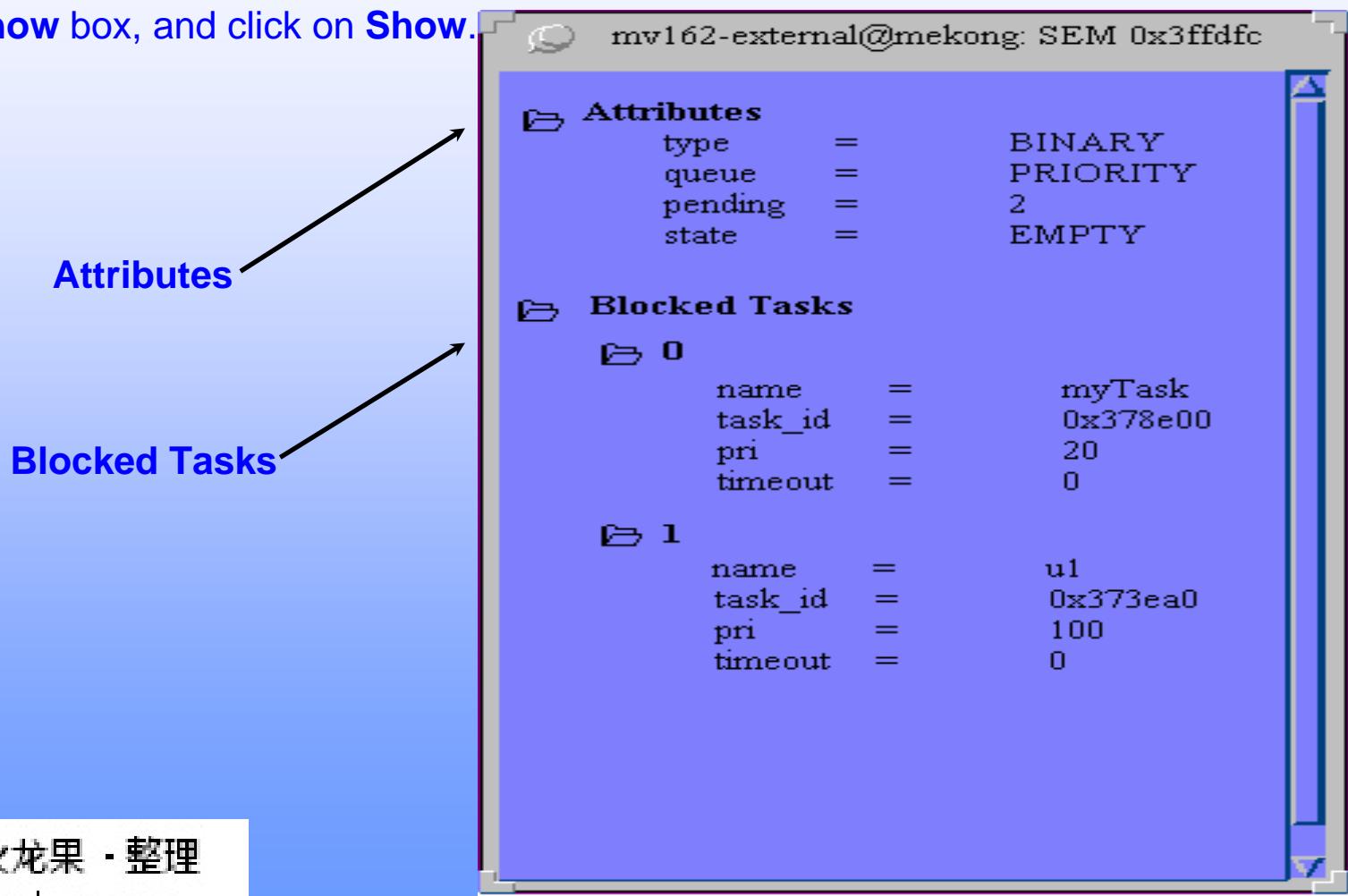
- Additional semaphore routines :

<i>semDelete( )</i>	Destroy the semaphore. <b>semTake( )</b> calls for all tasks pended on the semaphore return ERROR.
<i>show( )</i>	Display semaphore information.



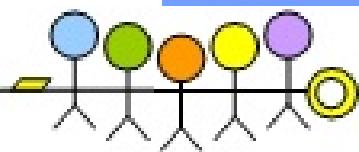
# Semaphore Browser

- To inspect the properties of a specific semaphore insert the semaphore ID in the Browser's **Show** box, and click on **Show**.



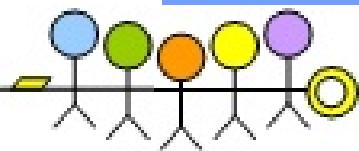
## Locking out Preemption

- When doing something quick frequently, it is preferable to lock the scheduler instead of using a Mutex semaphore.
- Call ***taskLock( )*** to disable scheduler.
- Call ***taskUnLock( )*** to reenable scheduler.
- Does **not** disable interrupts.
- If the task blocks, the scheduler is reenabled.
- Prevents all other tasks from running, not just the tasks contending for the resource.



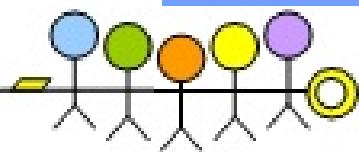
## ISR's and Mutual Exclusion

- ISR's can't use Mutex semaphores.
- Task sharing a resource with an ISR may need to disable interrupts.
- To disable / re-enable interrupts :  
`int intLock( )`  
`void intUnLock(lockKey)`  
*lockKey* is return value from *intLock( )*.
- Keep interrupt lock-out time short (e.g., long enough to set a flag) !
- Does not disable scheduler.

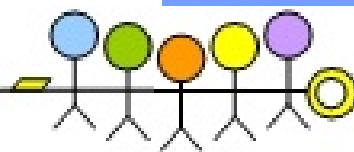
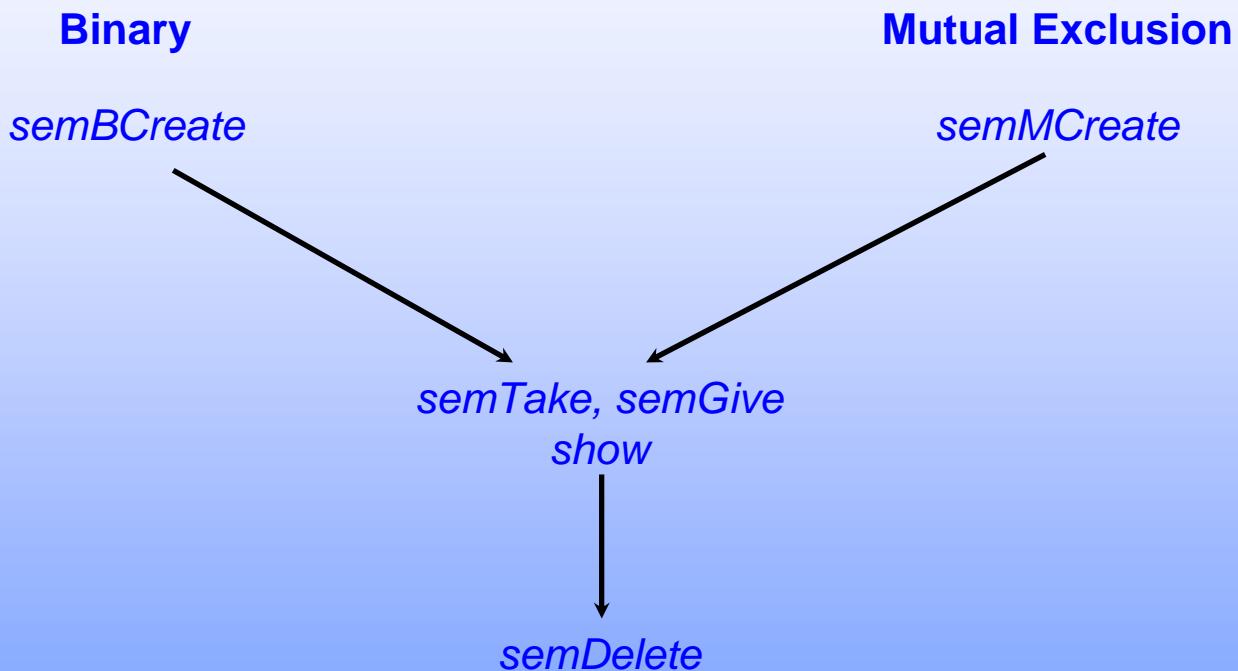


# 计数器信号量

- 计数器信号量是实现任务同步的和互斥的另一种方法。计数器信号量除了象二进制信号量那样工作外，他还保持了对信号量释放次数的跟踪。信号量每次释放，计数器加一，每次获取，计数器减一。当计数器减到0，试图获取该信号量的任务被阻塞。
- 正如二进制信号量，当信号量释放时，如果有任务阻塞在该信号量阻塞队列上，那么任务解除阻塞。然而，不象二进制信号量，如果信号量释放时，没有任务阻塞在该信号量阻塞队列上，那么计数器加一。
- 计数器信号量适用于保护拥有多个拷贝的资源。

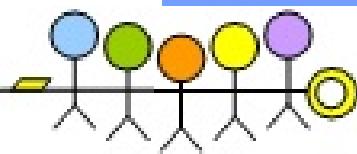


# Summary



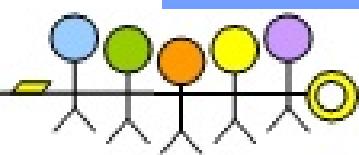
# Summary

- Binary semaphores allow tasks to pend until some event occurs.
  - Create a binary semaphore for the given event.
  - Tasks waiting for the event blocks on a **semTake( )**.
  - Task or ISR detecting the event calls **semGive( )** or **semFlush( )**.
- Caveat : if the event repeats too quickly, information may be lost



# Summary

- Mutual Exclusion Semaphores are appropriate for obtaining access to a resource.
  - Create a mutual exclusion semaphore to guard the resource.
  - Before accessing the resource, call **semTake()**.
  - To release the resource, call **semGive()**.
- Mutex semaphores have owners.
- Caveats :
  - Keep critical regions short.
  - Make all accesses to the resource through a library of routines.
  - Can't be used at interrupt time.
  - Deadlocks.



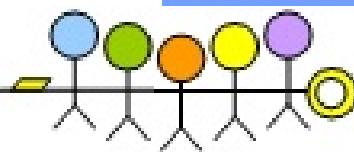
# Summary

- ***taskLock( ) / taskUnLock( ) :***

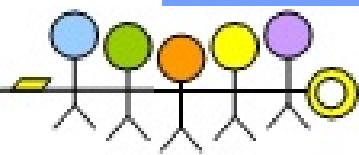
- Prevents other tasks from running.
- Use when doing something quick frequently.
- Caveat : keep critical region short.

- ***intLock( ) / intUnLock( ) :***

- Disable interrupts.
- Use to protect resources used by tasks and interrupt service routines.
- Caveat : keep critical region short.



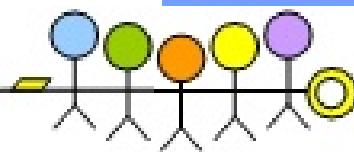
**Now  
Have a rest**



# Chapter

# 7

# Intertask Communication



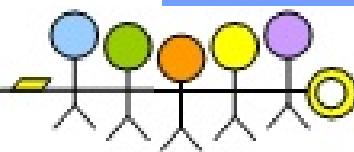
# Intertask Communication

Introduction

Shared Memory

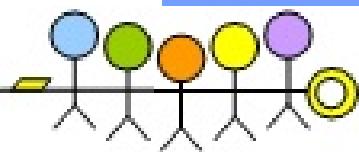
Message Queues

Pipes

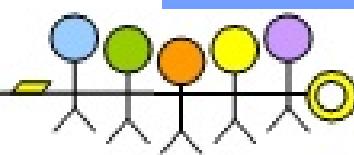
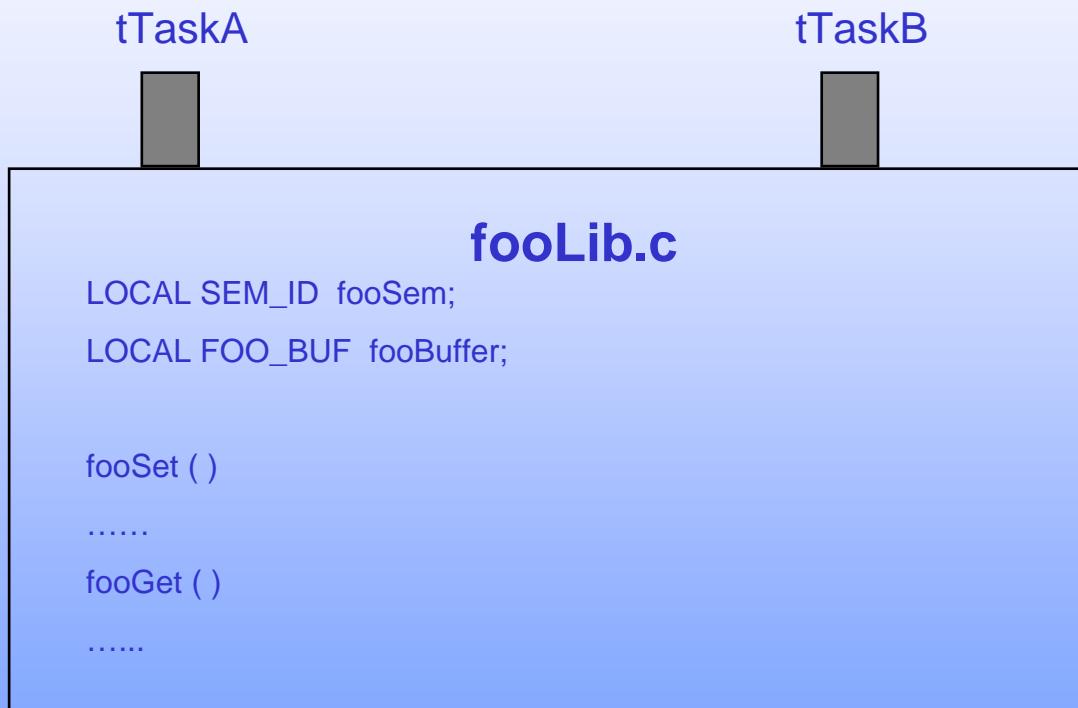


# Overview

- Multitasking systems need communication between tasks.
- Intertask communication made up of three components :
  - Data / information being shared.
  - Mechanism to inform task that data is available to read or write.
  - Mechanism to prevent tasks from interfering with each other (e.g., if there are two writers).
- Two common methods :
  - Shared memory.
  - Message passing.



# Shared Memory

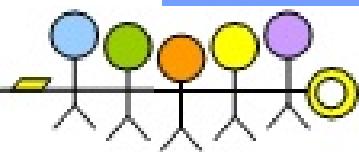


# Message Passing Queues

- VxWorks pipes and message queues are used for passing messages between tasks.
- Both pipes and message queues provide :



- FIFO buffer of messages
- Synchronization
- Mutual Exclusion
- More robust than shared data.
- Can be used from task to task, or from ISR to task.



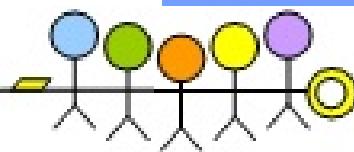
# Intertask Communication

Introduction

Shared Memory

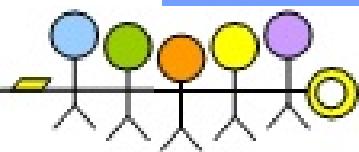
Message Queues

Pipes



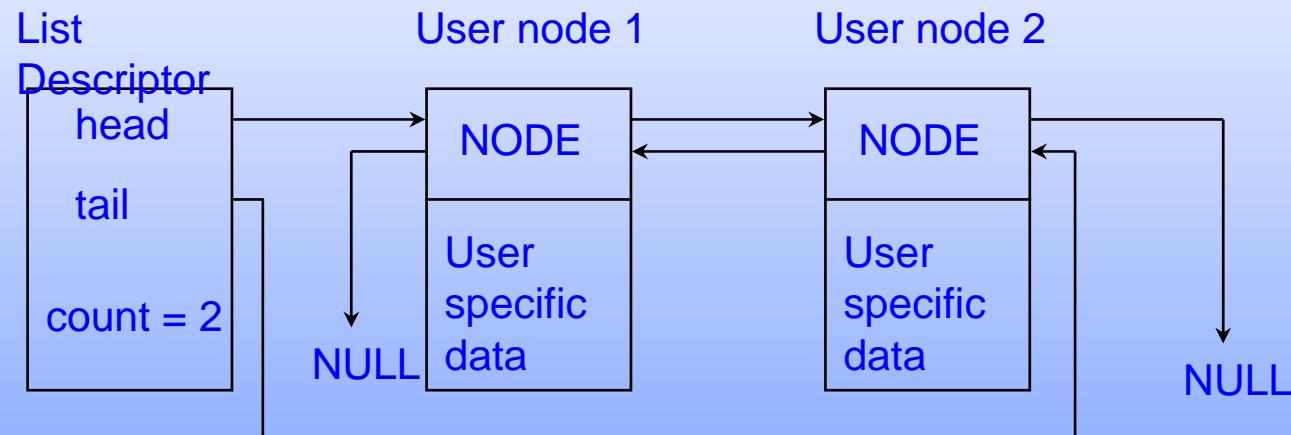
# Overview

- All tasks reside in a common address space.
- User-defined data structures may be used for Intertask communication :
  - Write a library of routines to access these global or static data-structures.
  - All tasks which use these routines manipulate the same physical memory.
  - Semaphores may be used to provide mutual exclusion and synchronization.
- VxWorks provides libraries to manipulate common data structures such as linked lists and ring buffers.

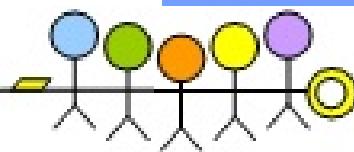


# Linked Lists

- **IstLib** contains routines to manipulate doubly linked lists.

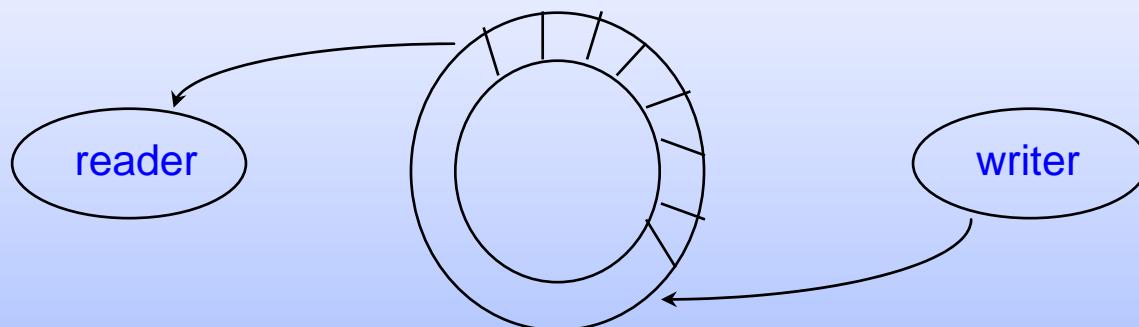


- Mutual exclusion and synchronization are *not* built-in.

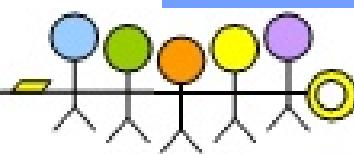


# Ring Buffers

- **rngLib** contains routines to manipulate ring buffers (FIFO data streams)



- Mutual exclusion is not required if there is only one reader and one writer. Otherwise, user must provide.
- Synchronization is **not** built-in.



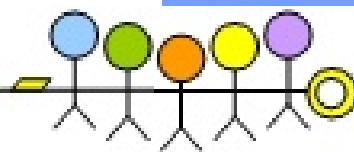
# Intertask Communication

Introduction

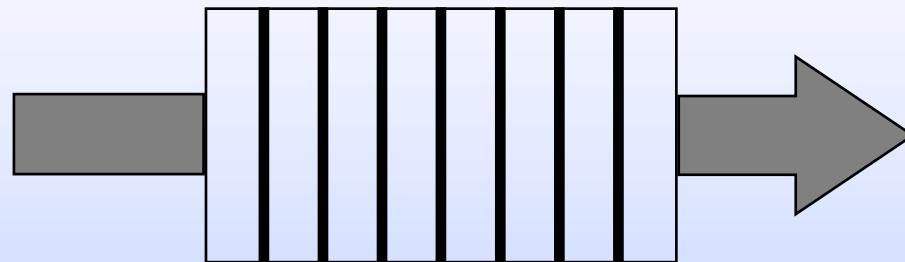
Shared Memory

Message Queues

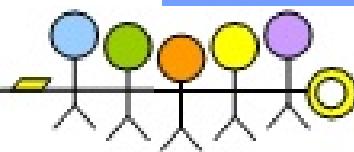
Pipes



# Message Queues



- Used for intertask communication within one CPU.
- FIFO buffer of variable length messages.
- Task control is built-in :
  - Synchronization.
  - Mutual Exclusion.



# Creating a Message Queue

`MSG_Q_ID msgQCreate (maxMsgs, maxMsgLength,  
options)`

`maxMsgs`  
queue.

Maximum number of messages on the

`maxMsgLength`  
the

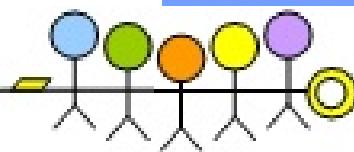
Maximum size in bytes of a message on  
queue.

`options`  
**(MSG\_Q\_FIFO**

Queue type for pended tasks

or **MSG\_Q\_PRIORITY**)

- Returns an id used to reference this message queue or **NULL** on error.



# Sending Messages

STATUS msgQSend (msgQId, buffer, nBytes, timeout,  
priority)

msgQId **MSG\_Q\_ID** returned by **msgQCreate( )**.

buffer Address of data to put on queue.

nBytes Number of bytes to put on queue.

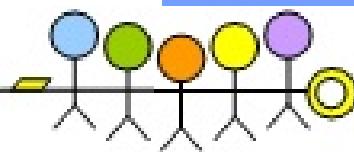
timeout Maximum time to wait (if queue is full).

Values

can be tick count, **WAIT\_FOREVER** or  
**NO\_WAIT**.

priority “Priority“ of message to put on queue. If  
**MSG\_PRI\_URGENT**, message put at  
queue. If **MSG\_PRI\_NORMAL** message  
end of queue.

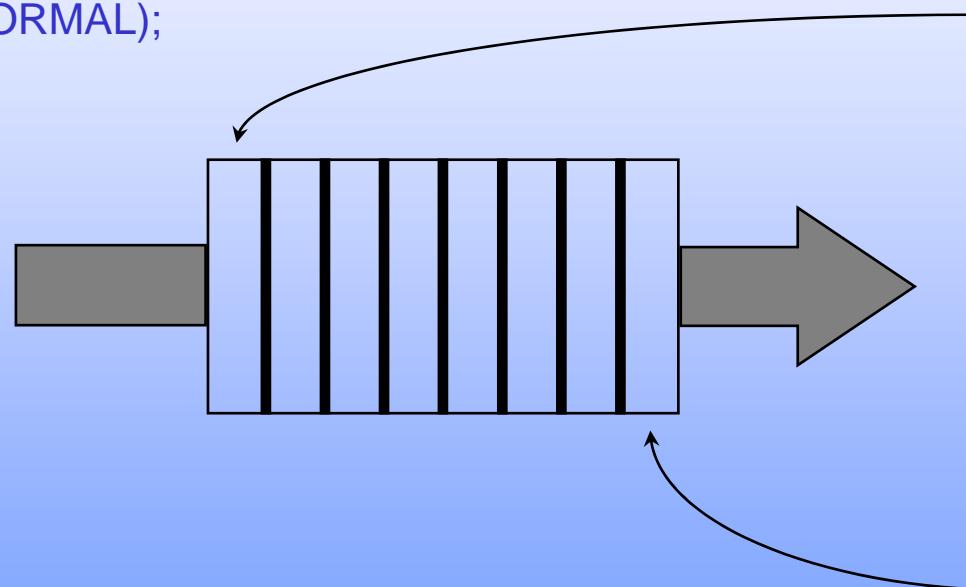
head of  
put at



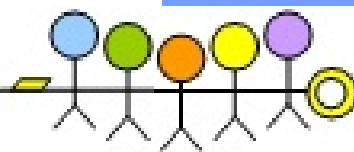
## Message Sending Examples

```
char buf[BUFSIZE];
```

```
status = msgQSend (msgQId, buf, sizeof(buf), WAIT_FOREVER,  
MSG_PRI_NORMAL);
```



```
status = msgQSend (msgQId, buf, sizeof(buf), NO_WAIT, MSG_PRI_URGENT);
```

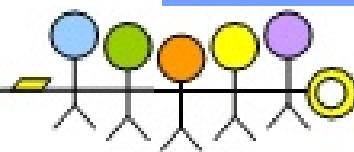


# Receiving Messages

```
int msgQReceive (msgQId, buffer, maxNBytes,timeout)
```

msgQId	Returned from <b>msgQCreate()</b> .
buffer	Address to store message.
MaxNBytes	Maximum size of message to read from queue.
timeout	Maximum time to wait (if no message available). Values can be clock ticks, <b>WAIT_FOREVER</b> or <b>NO_WAIT</b> .

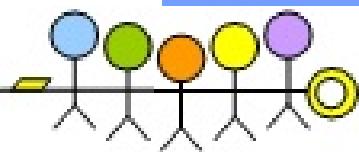
- Returns number of bytes read on success, **ERROR** on timeout or invalid *msgQId*.
- Unread bytes in a message are lost.



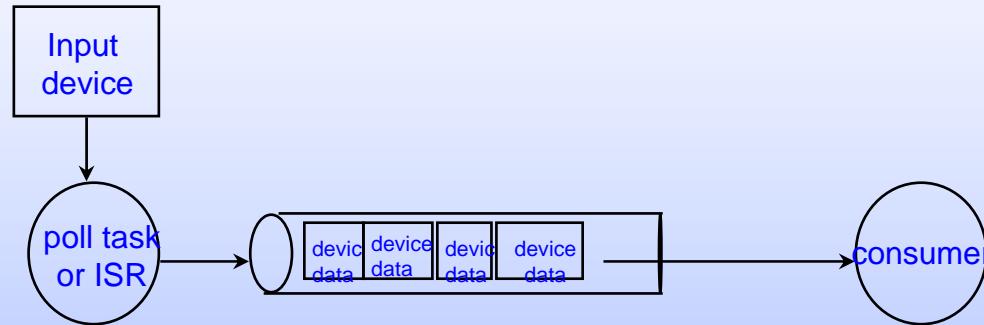
# Deleting a Message Queue

STATUS msgQDelete (msgQId)

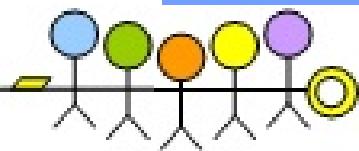
- Deletes message queue.
- Tasks pended on queue will be unpended; their *msgQSend( )* or *msgQReceive( )* calls return **ERROR**. These tasks' *errno* values will be set to **S\_objLib\_OBJ\_DELETED**.



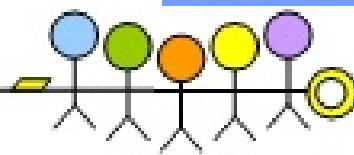
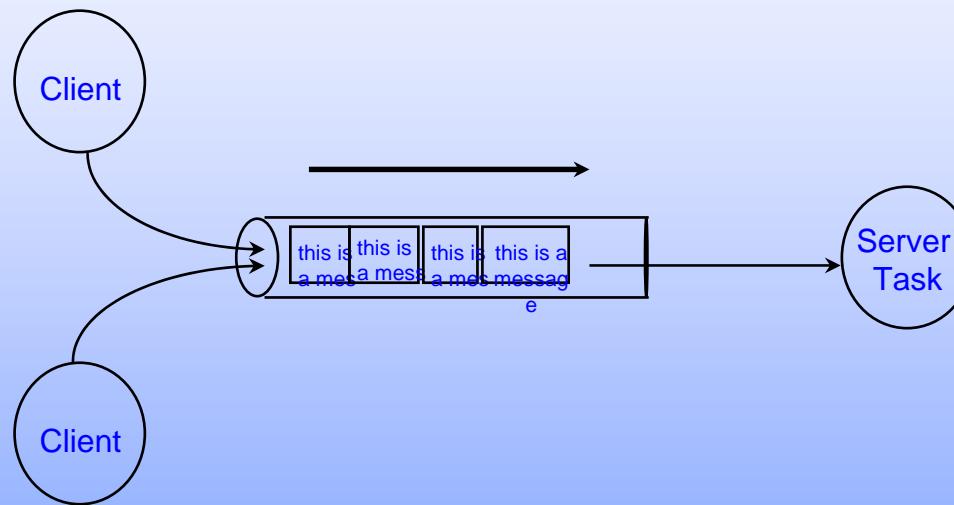
# Gathering Data with Message Queues



- To capture data quickly for future examination :
  - Have a poll task or ISR place device data in a message queue.
  - Have a lower priority task read data from this queue for processing.



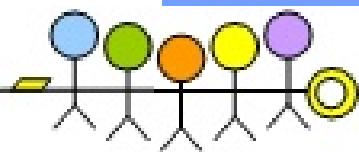
# Client-Server model with Message Queues



## Client-Server Variations

How would the previous code example change if :

- The client needs a reply from the server ?
- We wish to simultaneously service several requests ?  
(we may wish to do this if there are multiple clients, and this service requires blocking while I / O completes).



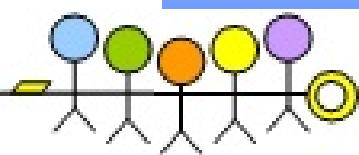
## Message - Queue Browser

- To examine a message queue, enter the message queue ID in the Browser's **Show** box, and click on **Show**.

mv152-external@mekong: Mempart Ox3ff988		
Attributes		
options	=	PRIORITY
maxMsgs	=	10
maxLength	=	100
sendTimeouts	=	0
recvTimeouts	=	0
Receivers Blocked		
Senders Blocked		
Messages Queued		
0		
address	=	0x3ffdb4
length	=	0x7
value	=	68 65 66 6f 0a
*hello..		
1		
address	=	0x3ffd48
length	=	0x4
value	=	68 69 0a 0d
*hi..		

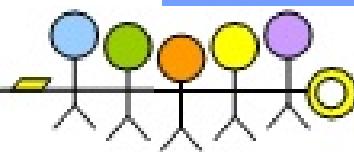
# POSIX消息队列

- POSIX消息对列由mqPxLib库提供，除了POSIX消息队列提供命名队列和消息具有一定范围的优先级之外，提供的这些函数与Wind消息队列类似。
- 在使用POSIX消息队列前，系统初始化代码必须调用mqPxLibInit()进行初始化。也可以在配置VxWorks时进行配置。
- 通知任务一个消息队列在等待
  - 任务可以调用mq\_notify()函数要求系统当有一个消息进入一个空的消息队列时通知它。这样可以避免任务阻塞或轮询等待一个消息。
  - mq\_notify()以消息进入空队列时系统发给任务的信号(signal)作为参数。这种机制只对当前状态为空的消息队列有效，如果消息队列中已有可用消息，当有更多消息到达时，通知不会发生。



# POSIX和Wind消息队列比较

特征	Wind消息	POSIX消息对列
消息优先级	1	32
阻塞的任务队列	FIFO或基于优先级	基于优先级
不带超时的接收	可选	不可用
任务通知	不可用	可选
关闭/解链语法	无	有



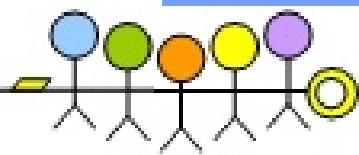
# Intertask Communication

Introduction

Shared Memory

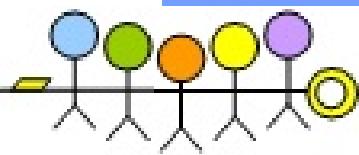
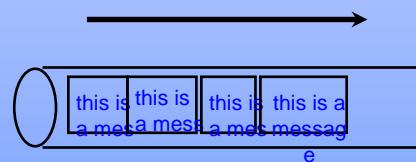
Message Queues

Pipes



# Pipes

- Virtual I/O device managed by **pipeDrv**.
- Built on top of message queues.
- Standard I/O system interface (read / write).
- Similar to named pipes in UNIX. (**UNIX Host**)



# Creating a Pipe

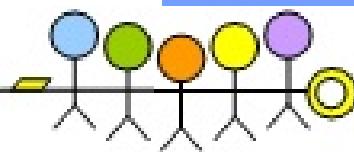
STATUS pipeDevCreate (name, nMessages, nBytes)

**name** Name of pipe device, by convention use  
“/pipe/*yourName*”.

**nMessages** Maximum number of messages in the  
pipe.

**nBytes** Maximum size in bytes of each  
message.

- Returns **OK** on success, otherwise **ERROR**.



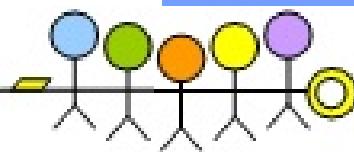
## Example Pipe Creation

```
-->pipeDevCreate ("/pipe/myPipe",10,100)
```

```
    value = 0 = 0x0
```

```
-->devs
```

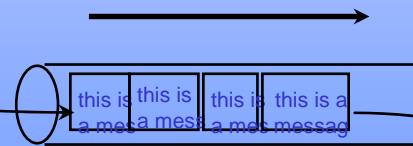
drv	name
0	/null
1	/tyCo/0
1	/tyCo/1
4	columbia:
2	/pipe/myPipe



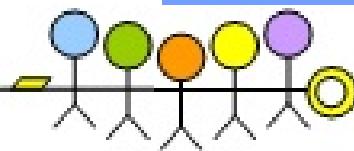
# Reading and Writing to a Pipe

- To access an existing pipe, first open it with `open( )`.
- To read from the pipe use `read( )`.
- To write to the pipe use `write( )`.

```
fd = open (“/pipe/myPipe”, O_RDWR, 0);  
write (fd, msg, len);
```

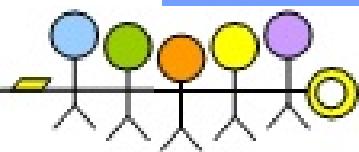


```
read (fd, msg,  
len);
```



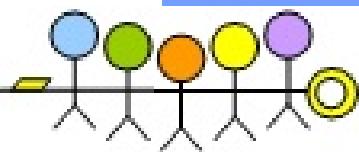
# Message Queues vs. Pipes

- Message Queue advantages :
  - Timeouts capability.
  - Message prioritization.
  - Faster.
  - *show( ).*
  - Can be deleted.
- Pipe advantages :
  - Use standard I / O interface i.e., *open( )*, *close( )*, *read( )*, *write( )*, etc.
  - Can perform redirection via *ioTaskStdSet( )* (see I/O chapter)
  - File descriptor can be used in *select( )*.



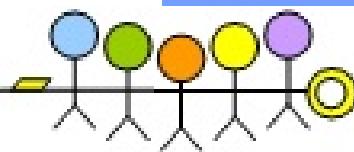
# Summary

- Shared Memory
  - Often used in conjunction with semaphores.
  - **IstLib** and **rngLib** can help.
- Message queues :  
*msgQCreate( )*  
*msgQSend( )*  
*msgQReceive( )*
- Pipes  
*pipeDevCreate( )*  
Access pipe via file descriptor returned from **open( )**.  
Use **write( ) / read( )** to send / receive messages from a pipe.

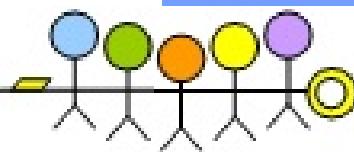
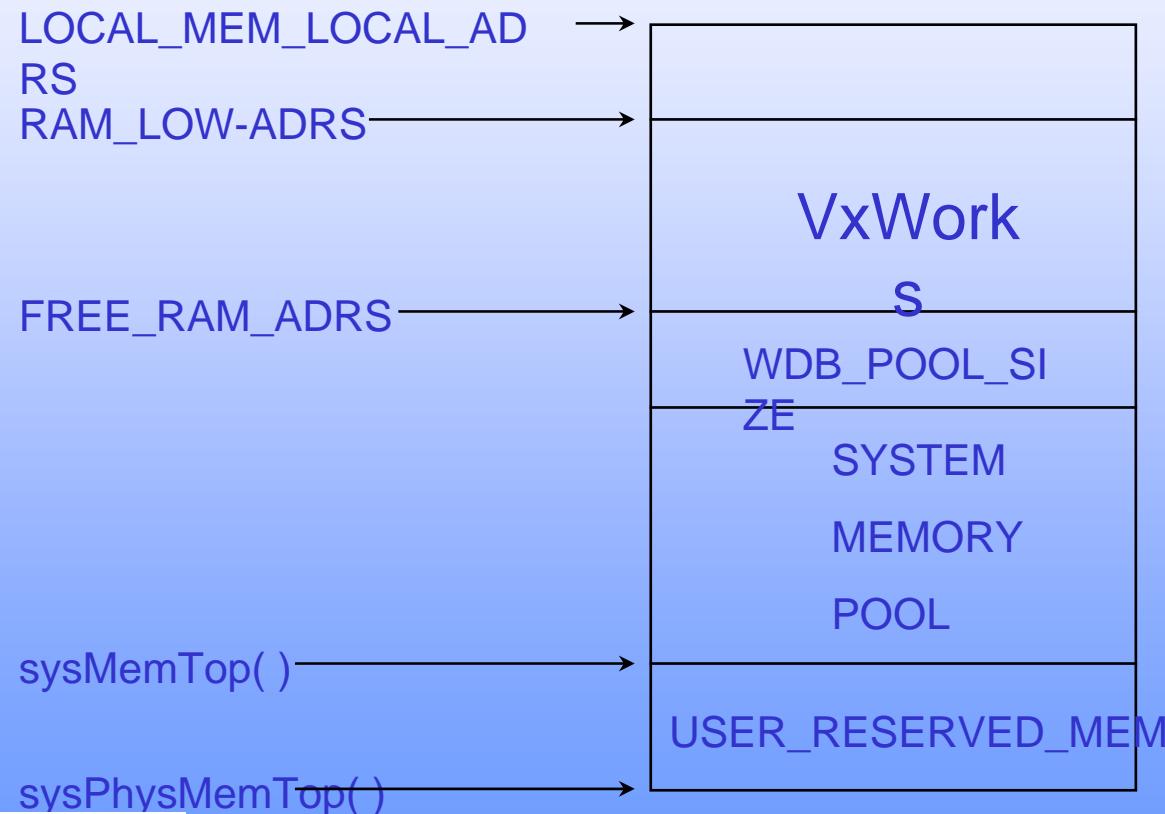


# Chapter 8

# Memory

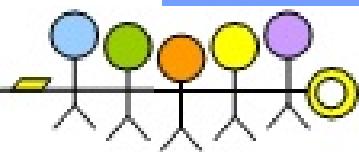


# Memory Layout



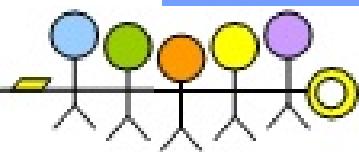
# Target Server Memory Pool

- A pool of memory on the target reserved for use by the Tornado tools :
  - Dynamic loading of object modules.
  - Spawning task from WindSh or CrossWind.
  - Creation of variables from WindSh.
- The target server manages the pool, keeping overhead such as block lists on the host.
- The initial size of the target server memory pool is configured by **WDB\_POOL\_SIZE**.  
The default is 1 / 16 of **sysMemTop( ) - FREE\_RAM\_ADRS**.
- Additional memory is silently allocated from the system memory pool, if needed.



# System Memory Pool

- Used for dynamic memory allocation in programs :
  - *malloc( ).*
  - Creating tasks (stack and TCB).
  - VxWorks memory requests.
- Initialized at system startup.
  - Can modify **USER\_RESERVED\_MEM** to reserve memory for application-specific use.
  - May need to modify **sysPhysMemTop( )** (or just **LOCAL\_MEM\_SIZE**) when adding memory to your board.
- To add off board memory :  
*void memAddToPool (pPool, poolSize)*  
*pPool* must be the *local* address of the memory.



# Allocating / Releasing Memory

- To dynamically allocate memory :

**void \*malloc (nBytes)**

Returns a pointer to the newly allocated memory or **NULL** on error.

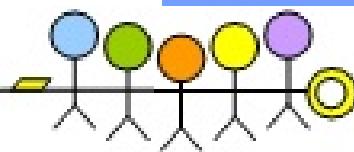
- Uses first-fit algorithm.

- Free memory is stored in a linked list.
  - Some (small) overhead for each **malloc( )**.

- To release allocated memory :

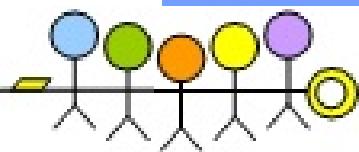
**void free (ptr)**

Adjacent blocks are coalesced.



# Debugging Options

- Default *malloc( )* debugging : If request too large, log an error message.
- Default *free( )* debugging :
  - Check block for consistency.
  - If corrupted: suspend task, log error message.
- Can change default debugging options with :  
**void memOptionsSet (options)**
- Options can be :
  - + **MEM\_ALLOC\_ERROR\_LOG\_FLAG**
  - **MEM\_ALLOC\_ERROR\_SUSPEND\_FLAG**
  - + **MEM\_BLOCK\_CHECK**
  - + **MEM\_BLOCK\_ERROR\_LOG\_FLAG**
  - + **MEM\_BLOCK\_ERROR\_SUSPEND\_FLAG**

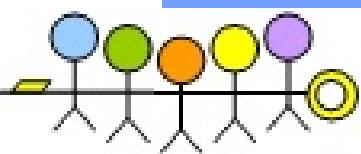


# Examining Memory

- Use the browser.
- Enter the memory partition ID in the **Show** box.

```
mv152-external@mekong: Mempart Ox
Total
  bytes   =      3870840
Allocated
  blocks  =        86
  bytes   =      467224
Free
  blocks  =         7
  bytes   =      3403584
Cummulative
  blocks  =        88
  bytes   =      467832
Free List
  0
    addr     =      0x3fd8f8
    size     =       9604
  1
    addr     =      0x4ef90
    size     =      3355432
```

Free List →

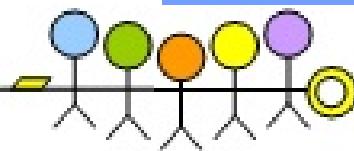


# Additional System Memory Management Routines

***void \*calloc(nElems, size )*** Allocate zeroed memory for an array.

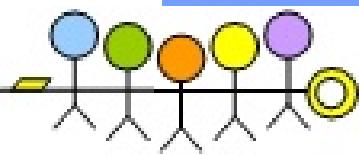
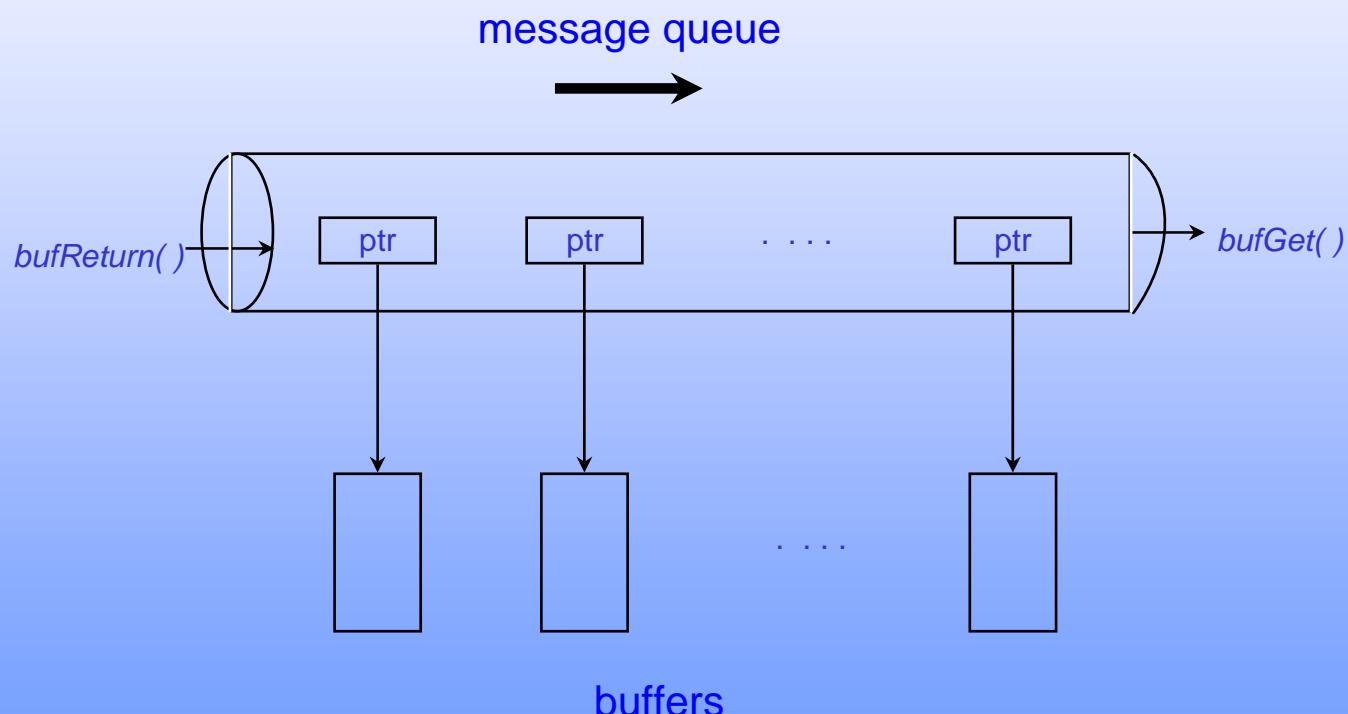
***void \*realloc(ptr,newSize)*** Resize an allocated block. The block may be moved.

***int memFindMax( )*** Returns the size of the largest free block in system memory.



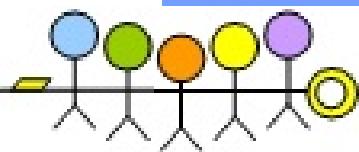
# Fine Tuning

- For fast, deterministic allocation of fixed size buffers, use message queues instead of `malloc( )`.



## Generic Partition Manager

- VxWorks provides low level routines to create and manipulate alternate memory pools.
- High level routines like *malloc( )* and *free( )* call these lower level routines, specifying the system memory pool.
- Application may use alternate memory partitions to reduce fragmentation.
- Applications may use alternate memory partitions to manage memory with different properties.



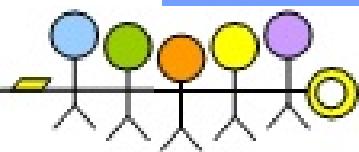
# Creating a memory Partition

PART\_ID memPartCreate (pPool, size)

pPool            Pointer to memory for this  
partition

size            Size of memory partition in  
bytes.

- Returns a partition id (**PART\_ID**) or **NULL** or error.
- The memory for this partition (*pPool*) may be taken from :
  - A separate memory board.
  - A block allocated from the system memory partition.
  - The top of the CPU board's RAM.



# Managing Memory Partitions

- System partition management routines call routines listed below, specifying the **PART\_ID** as *memSysPartId*.

## Generic

*memPartAlloc( )*

*memPartFree( )*

*memPartShow( )*

*memPartAddToPool( )*

*memPartOptionsSet( )*

*memPartRealloc( )*

*memPartFindMax( )*

## System Memory Pool

*malloc( )*

*free( )*

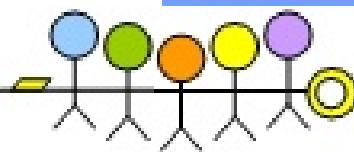
*memShow( )*

*memAddToPool( )*

*memOptionsSet( )*

*realloc( )*

*memFindMax( )*



## Example Creating a Memory Partition

```
-->partId=memPartCreate(pMemory, 100000)
```

new symbol “partId” added to symbl table.

```
partId = 0x23ff318 : value = 37745448 = 0x23ff328 = partId +  
0x10
```

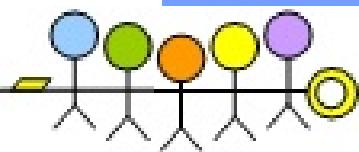
```
-->ptr=memPartAlloc(partId, 200)
```

new symbol “ptr” added to symbl table.

```
ptr = 0x23ff2ec : value = 37652632 = 0x23e8898
```

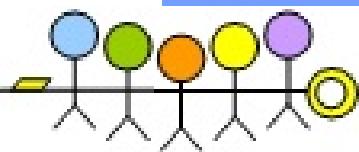
```
-->show partId
```

	status	bytes	blocks	ave	block	max	block
current							
free	99776	1		99776		99776	
alloc	208	1		208		-	
cumulative							
alloc	208	1		208		-	



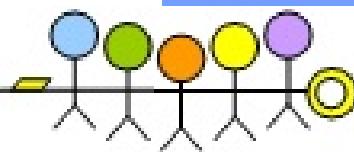
# Summary

- Standard C routines used for dynamic memory allocation.
- To configure the system memory pool :
  - Modify *sysPhysMemTop( )*
  - Modify **USER\_RESERVED\_RAM**
  - Modify *memAddToPool( )*
- For fast, deterministic allocation of fixed size buffers, use message queues instead of *malloc( )*.
- Create separate memory partition for off-board memory, or to help reduce fragmentation.



# Chapter 9

# Exceptions, Interrupts and Timers

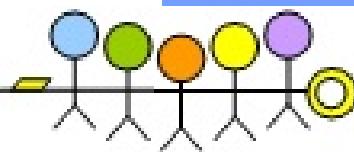


# Exceptions, Interrupts and Timers

Exception Handling and Signals

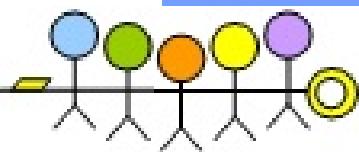
Interrupt Service Routines

Timers



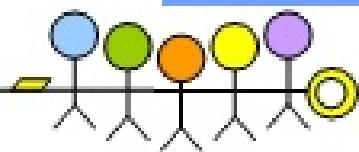
# 信号(Signals) ( - )

- VxWorks支持软件信号功能。信号可以异步改变任务控制流。任何任务和ISR都可以向指定的任务发信号。获得信号的任务立即挂起当前的执行，在下次调度它运行时转而执行指定的信号处理程序。
- 信号处理程序在信号接收任务的 上下文中执行，使用该任务的堆栈。在任务阻塞时，信号处理程序仍可被唤醒
- 信号机制适合于错误和异常处理。
- 通常，信号处理程序可以作为中断处理程序看待。任何可能导致调用程序阻塞的函数均不能在信号处理程序中调用。
- Wind内核支持两种类型的信号接口
  - UNIX BSD风格的信号
  - POSIX兼容的信号



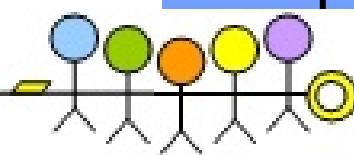
# 信号(Signals) (二)

- 信号在很多方面跟硬件中断相似。基本信号接口提供了31个不同的信号。调用sigvec()或sigaction()可为信号指定一个信号处理程序。这与调用intConnect()为中断指定一个中断处理程序(ISR)相似。可以调用kill()将信号发送给任务，这类似于于中断发生。函数sigsetmask()和sigblock或sigprocmask()可以用来象屏蔽中断那样屏蔽信号。
- 使用sigInit()初始化信号函数库，使得基本信号函数可用



# 基本信号接口函数

POSIX 1003.1b兼容调用	UNIX BSD兼容调用	说明
signal()	signal()	指定信号的处理程序
kill()	kill()	向任务发送一个信号
raise()	N/A	发信号给自身
sigaction()	sigvec()	检测或设置信号处理程序
sigsuspend()	pause()	挂起任务直到信号提交
sigpending()	N/A	取回一组提交阻塞的信号
..	..	设置信号屏蔽
..	..	..

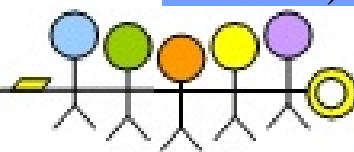


# 几个函数原型

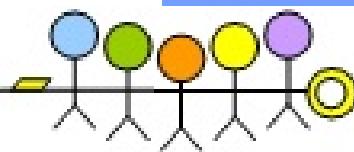
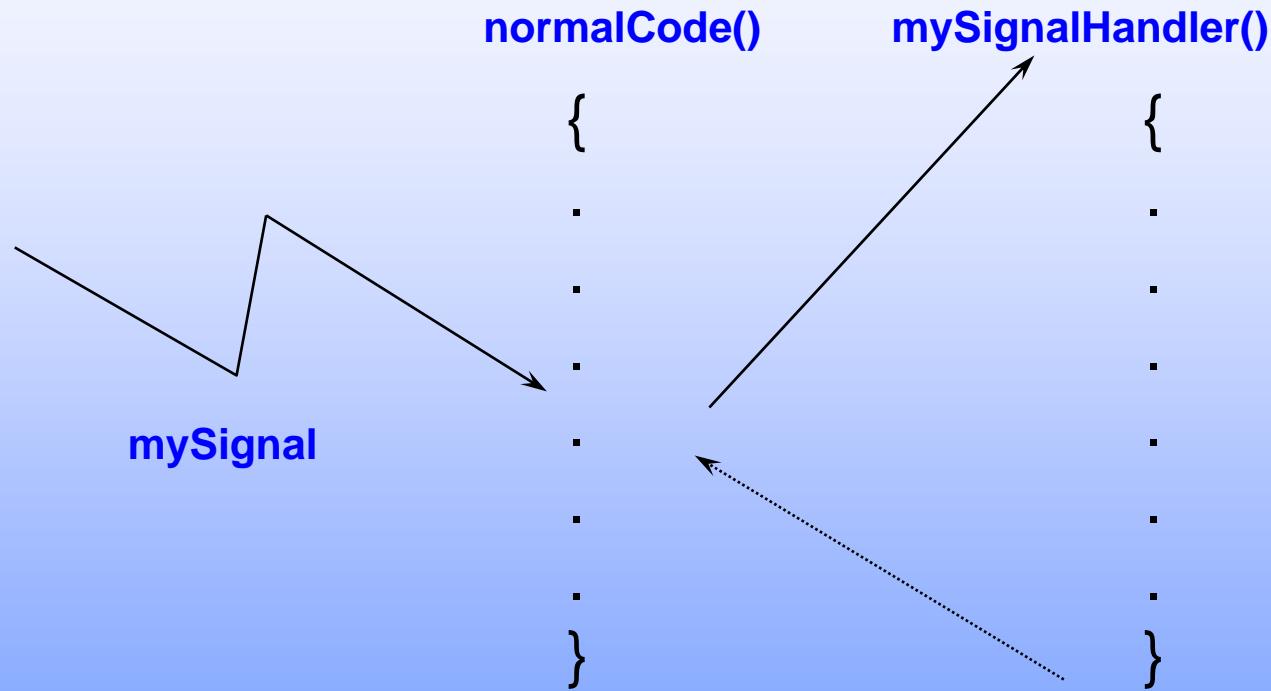
int sigvec 检测或设置信号处理程序  
( int sig, /\*于处理程序相联系的信号\*/  
  const struct sigvec \*pVec, /\*新的处理程序信息\*/  
  struct sigvec \*pOvec /\*旧的处理程序信息\*/  
)

int kill 向任务发送一个信号  
( int tid, /\*接收信号的任务号\*/  
  int signo /\*发送给任务的信号\*/  
)

sigqueue 提供与kill()等价的功能  
( int tid, /\*接收信号的任务\*/  
  int signo, /\*发送给任务的信号\*/  
  const union sigval value /\*隋信号发送的值\*/  
)

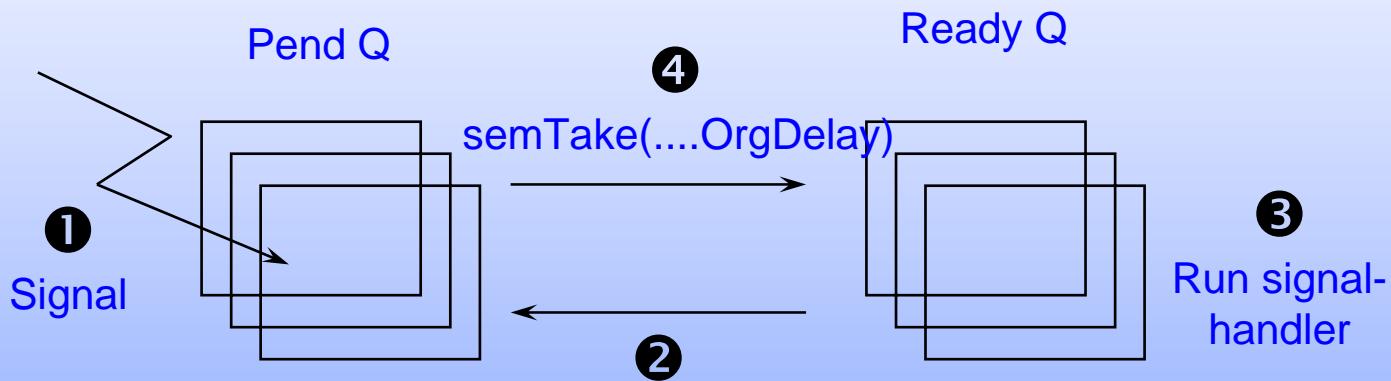


# Signals



# UNIX: UNIX vs. VxWorks Signals

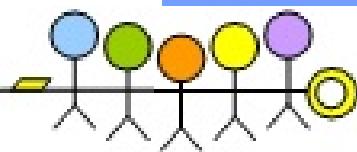
- Signal is ignored if no handler is installed
- “Automatic function restarting “



- Can install a handler to catch **SIGKILL**.
- No **SIGCHLD**, **SIGPIPE**, or **SIGURG**.
- **taskDelay()** sets `errno = EINTR` and returns **ERROR** if interrupted by a signal.

## Caveats

- Signals are **not** recommended for general intertask communication. A signal :
  - May be handled at too high a priority if it arrives during a priority inheritance.
  - Disrupts a task's normal execution order.(It is better to create two tasks, than multiplex processing in one task via signals.)
  - Can cause reentrancy problems between a task running its signal handler and the same task running its normal code.
  - Can be used to tell a task to shut itself down.
- **SigLib** contains both POSIX and BSD UNIX interfaces. Do not mix them.



# Registering a Signal Handler

- To register a signal handler

**signal ( signo.handler )**

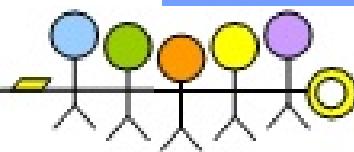
signo              Signal number.

handler              Routine to invoke when signal arrives  
(or **SIG\_IGN** to ignore signal).

Returns the previously installed signal handler, or **SIG\_ERR**.

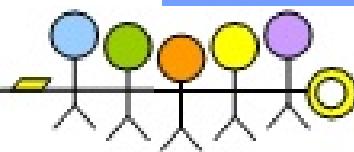
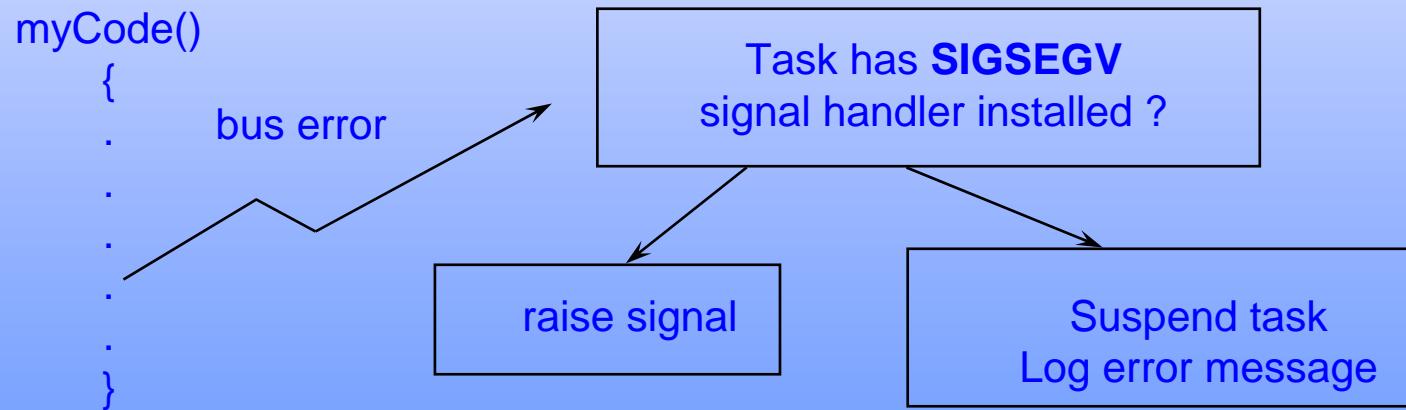
- The signal handler should be declared as :

**void sigHandler (int sig) ; /\* signal number \*/**



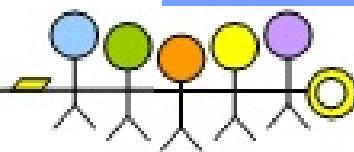
# Signals and Exceptions

- Hardware exceptions include bus error, address error, divide by zero, floating point overflow, etc.
- Some signals correspond to exceptions (e.g., **SIGSEGV** corresponds to a bus error on a 68K; **SIGFPE** corresponds to various arithmetic exceptions).



# The Signal Handler

- If an exception signal handler returns :
  - The offending task will be suspended.
  - A message will be logged to the console.
  
- Exception signal handlers typically call :
  - `exit( )` to terminate the task, or
  - `taskRestart( )` to restart the task, or
  - `longjmp( )` to resume execution at location saved by `setjmp( )`

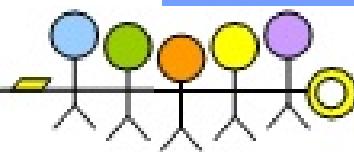


# Exceptions, Interrupts and Timers

Exception Handling and Signals

Interrupt Service Routines

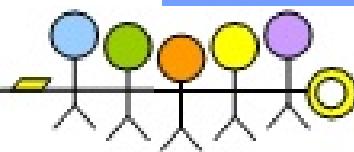
Timers



# 中断

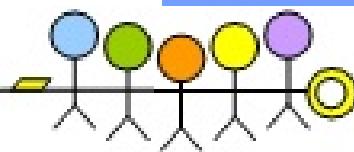
- 硬件中断处理是实时系统设计的最重要、最关键的问题。为了获得尽可能快的中断响应时间，VxWorks的中断处理程序运行在特定的上下文中(在所有任务上下文之外)。因此，中断处理不会涉及任何任务上下文的交换。
- VxWorks提供函数intConnect()，该函数允许将指定的C函数与任意中断相联系。
- STATUS intConnect(  
    VOIDFUNCPTR \*vector, /\*要联系的中断向量\*/  
    VOIDFUNCPTR \*routine, /\*中断发生时要调用的函数\*/  
    int               parameter /\*传递给中断处理函数的参数\*/  
)

该函数将指定的C函数routine与指定的中断向量vector相联系，函数的地址存储在这个中断向量里。

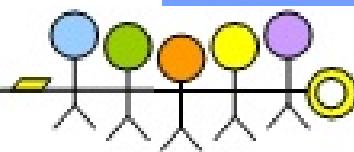
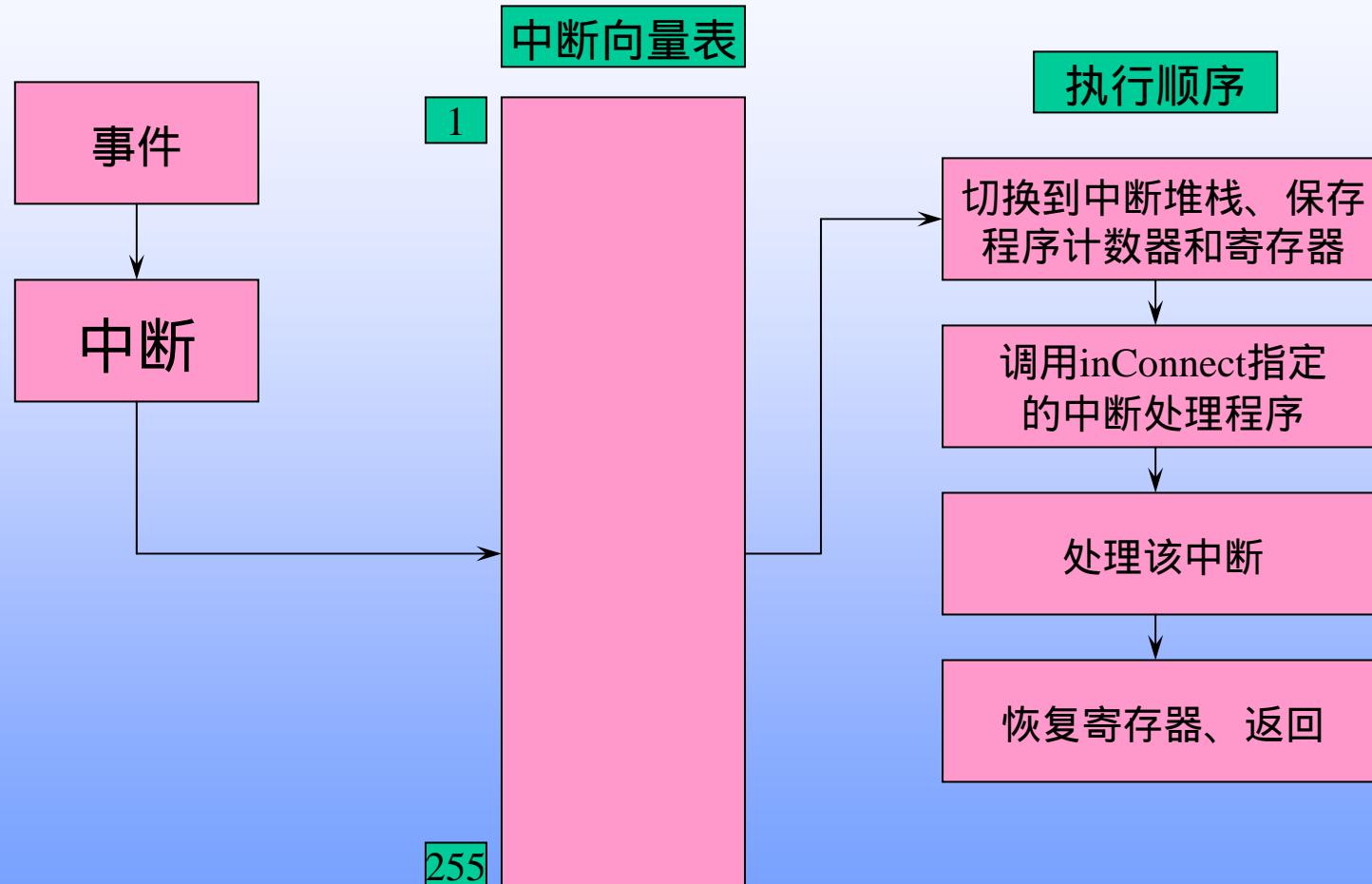


# 中断函数

调用	描述
intConnect()	设置中断处理程序
intContext()	如果是在中断级调用，返回真
intCount()	得到当前的中断嵌套深度
intLevelSet()	设置处理器中断屏蔽级
intLock()	禁止中断
intUnlock()	重新允许中断
intVecBaseSet()	设置向量基地址
intVecBaseGet()	得到向量基地址
intVecSet()	设置一个异常向量
intVecGet()	得到一个异常向量

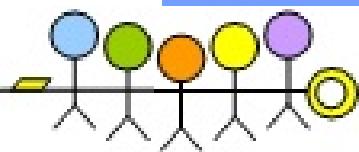


# 中断处理过程



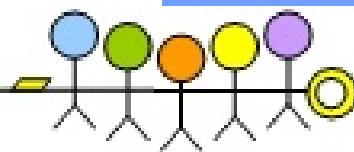
# 中断堆栈

- 如果体系结构允许，所有的ISRs使用的中断堆栈。堆栈的定位和初始化由系统在启动时根据指定的配置参数完成。堆栈必须足够大，以保证能够处理系统最坏情形下的中断嵌套。
- 体系结构不允许使用一个特定的堆栈。在这种结构中，ISRs使用中断任务的堆栈。对于这种结构的目标机，应用必须创建足够大的堆栈空间。
- 开发过程中，可以调用checkStack()函数察看一个任务堆栈的使用或整个系统堆栈的使用情况。



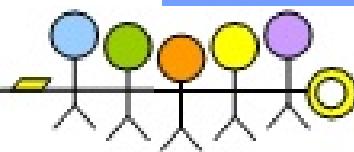
# ISR的特殊限制

- 基本约束：必须不能调用可能引起调用阻塞的函数。
- 在中断服务程序中不能试图获取一个信号量，因为信号量可能不可用。
- 中断服务程序里面不能使用malloc和free，因为他们都需要获取一个信号量。
- 中断服务程序也不能通过VxWorks驱动执行I/O操作。多数设备驱动由于可能需要等待设备而引起调用者阻塞，因此需要任务上下文交换。
- VxWorks支持纪录功能，任务可以向系统输出平台打印文本信息。  
logMsg()



# Interrupts

- Interrupts allow devices to notify the CPU that some event has occurred.
- An user-defined routine can be installed to execute when an interrupt arrives.
- This routine runs at interrupt time. It is *not* a task.
- On-board timers are a common source of interrupts. Using them requires understanding interrupts.



# Device Drivers

- Use interrupts for asynchronous I/O
- Are beyond the scope of this course
- For more information:

**intArchLib**

To install user defined ISR's.

**Board Support Package**

Board specific interrupt handling.

**Programmers Guide**

Architecture specific interrupt info.

**Tornado User's Guide**

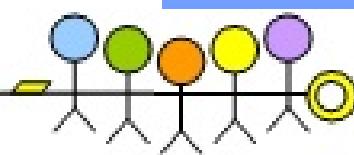
System mode debugging info.

**BSP Porting Kit**

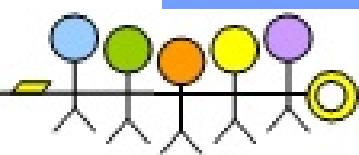
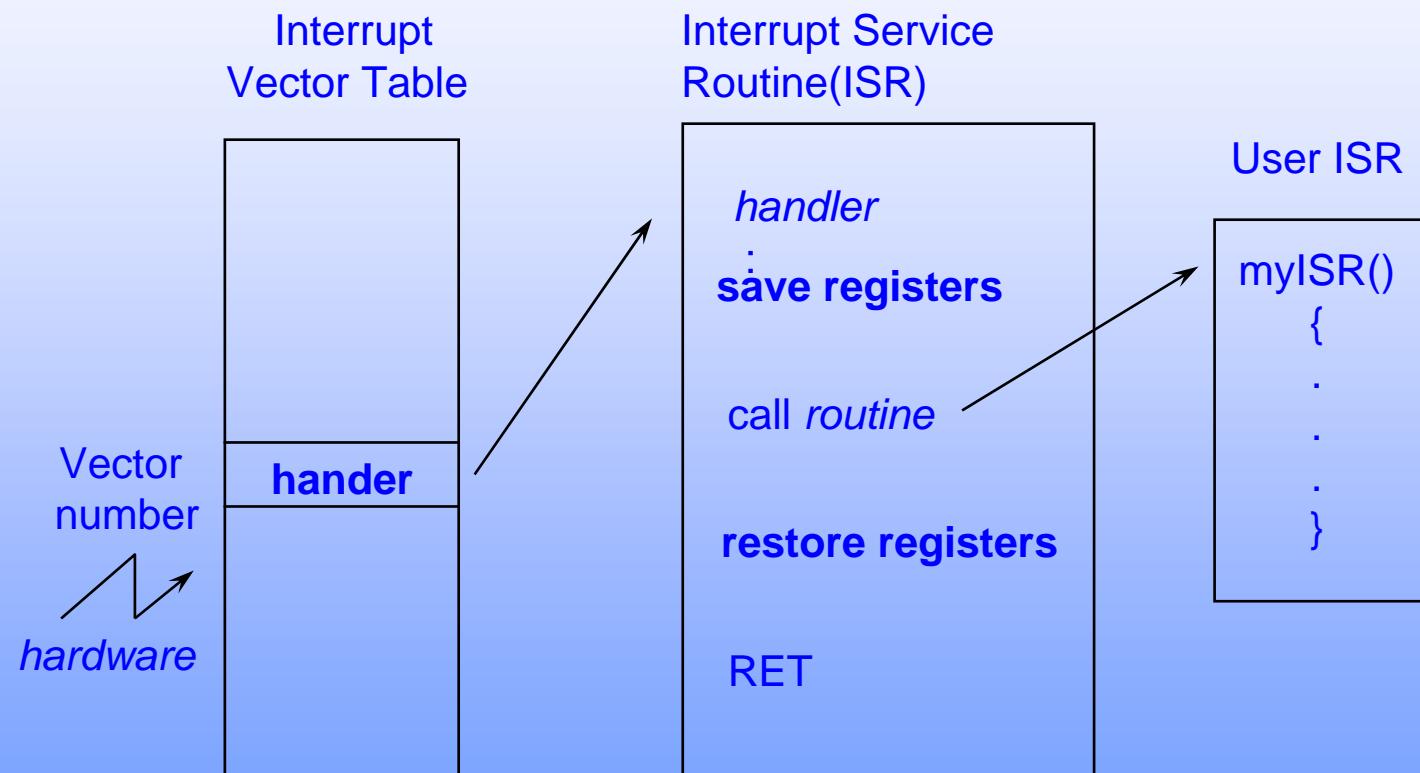
Optional product for writing BSP's.

**VxWorks Device Driver Workshop**

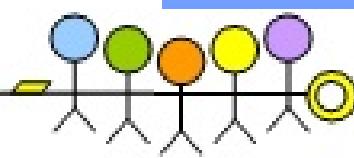
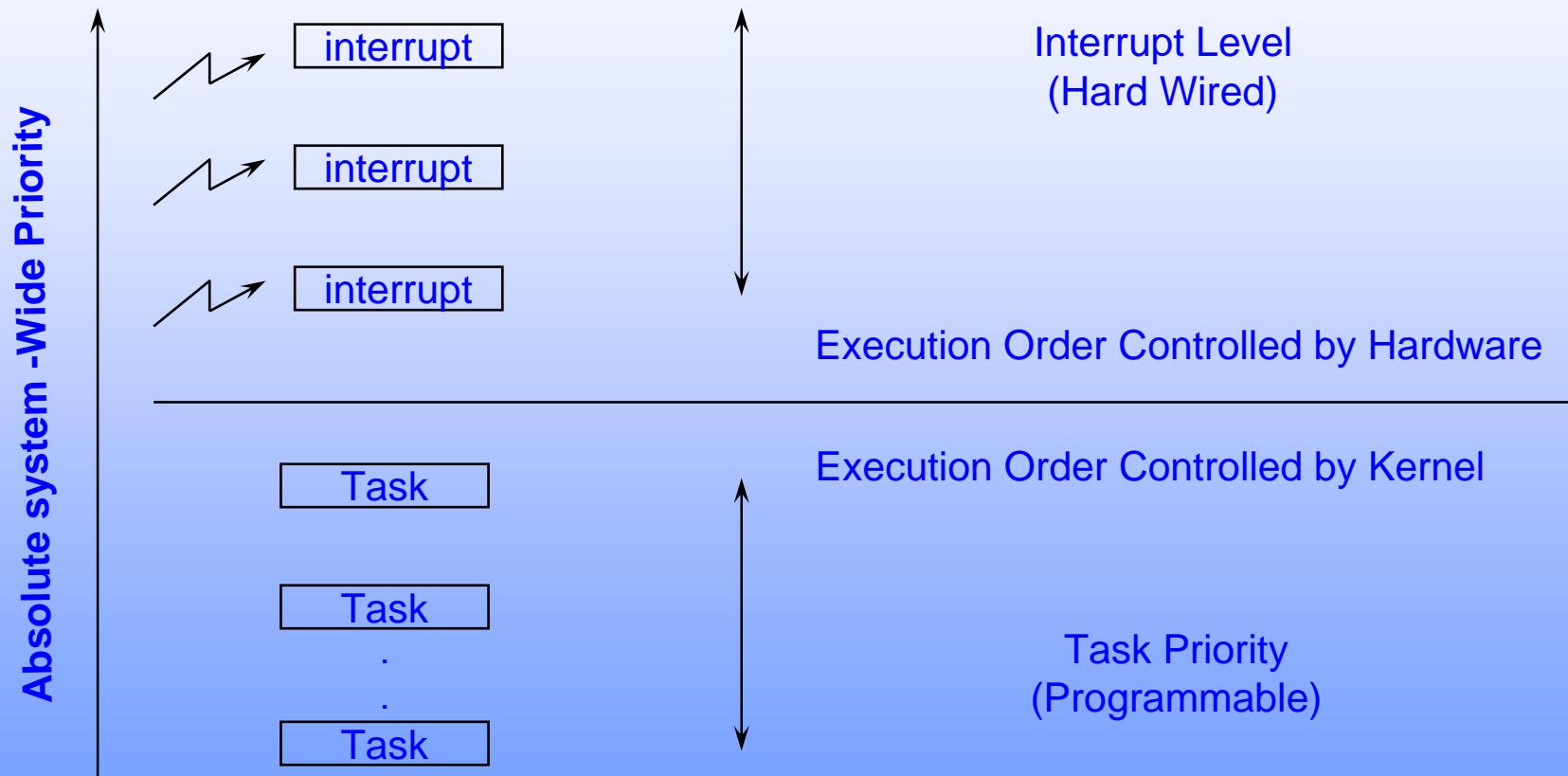
Write VMEbus and VxWorks standard device drivers.



# Handling an interrupt

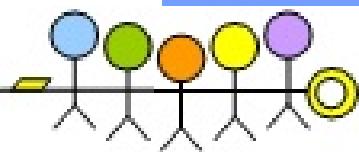
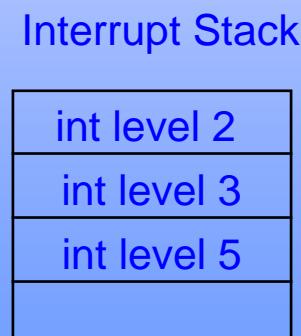


# Interrupts and Priorities



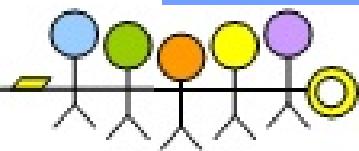
# Interrupt Stack

- Most architectures use a single dedicated interrupt stack
- Interrupt stack is allocated at system start-up.
- The Interrupt stack size is controlled by the macro `INT_STACK_SIZE` in `config All.h`
- Must be large enough for worst-case nesting



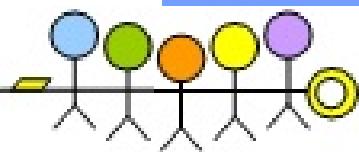
# ISR Restrictions

- No tasks can run until ISR has completed
- ISRs are restricted from using some VxWorks facilities.  
In particular they can't block :
  - Can't call ***semTake( )***.
  - Can't call ***malloc( )*** (uses semaphores).
  - Can't call I/O system routines (e.g.,***printf( )***).
- The *Programmer's guide* gives a list of routines which are callable at interrupt time



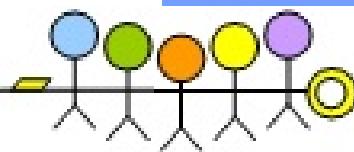
# ISR Guidelines

- Keep your ISRs short, because ISRs :
  - Delay lower and equal priority interrupts.
  - Delay all tasks
  - Can be hard to debug
- Avoid using floating-point operations in an ISR.
  - They may be slow.
  - User must call *fppSave( )* and *fppRestore( )*.
- Try to off-load as much work as possible to some task:
  - Work which is longer in duration.
  - Work which is less critical.



## Typical ISR

- Read and writes memory-mapped I/O registers.
- Communicates information to a task by :
  - Writing to memory
  - Making non-blocking writes to a message queue.
  - Giving a binary semaphore.



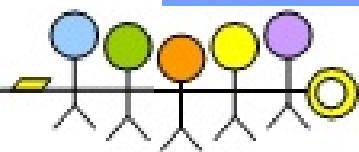
# Debugging ISR's

- To log diagnostic information to the console at interrupt time.

```
logMsg("foo = %d\n", foo, 0, 0, 0, 0, 0, 0);
```

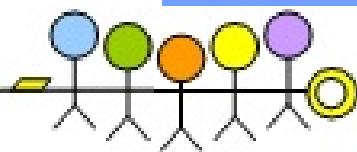
Sends a request to *tLogTask* to do a *printf( )* for us

- Similar to *printf( )*, with the following caveats :
  - Arguments must be 4 bytes.
  - Format string plus 6 additional arguments.
- Use a debugging strategy which provides system level debugging.
  - WDB agent
  - Emulator



## Exceptions at Interrupts Time

- Causes a trap to the boot ROMs.
- Logged messages will not be printed.
- Boot ROM program will display an exception description on reboot.
- An exception occurring in an ISR will generate a warm reboot.
- Can use **sprintf( )** to print diagnostic information to memory not overwritten on reboot, if necessary.

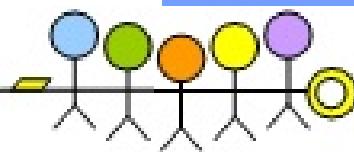


# Exceptions, Interrupts and Timers

Exception Handling and Signals

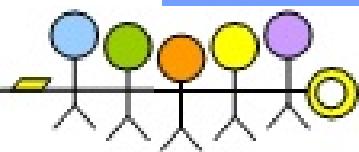
Interrupt Service Routines

Timers



# Timers

- On board timers interrupts the CPU periodically.
- Allows user-defined routines to be executed at periodic intervals which is useful for :
  - Polling hardware.
  - checking for system errors.
  - Aborting an untimely operation.
- VxWorks supplies a generic interface to manipulate two timers :
  - System clock.
  - Auxiliary clock ( if available).



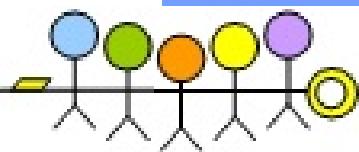
# System Clock

- System clock ISR performs book-keeping:
  - Increments the tick count (use **tickGet()** to examine the count ).
  - Updates the delatys and timeouts.
  - Checks for round robin rescheduling.
- These operations may cause a reschedule.
- Default clock rate is 60Hz.

**sysClkRateSet (freq)** Sets the clock rate .

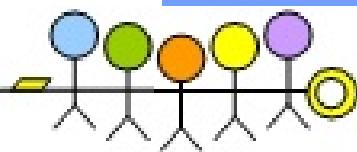
**int sysClkRateGet()** Returns the clock rate

- **sysClkRateSet( )** should only be called at systems startup.



# Watchdog Timers

- User interface to the system clock.
- Allows a C routine to be connected to a specified time delay.
- Upon expiration of delay, connected routine runs.
  - As part of system clock ISR
  - Subject to ISR restriction.



# Creating watchdog Timers

- To create a watchdog timer :

**WDOG\_ID wdCreate()**

Returns watchdog id, or **NULL** on error.

- To start (or restart) a watchdog timer :

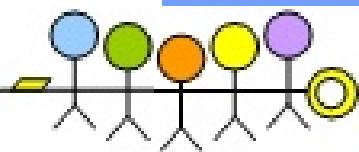
**Status wdSDstart (wdId,delay,pRoutine,parameter)**

**wdId** Watchdog id, returned from **wdCreate()**.

**delay** Number of ticks to delay

**pRoutine** Routine to call when delay has expired

**parameter** Argument to pass to routine

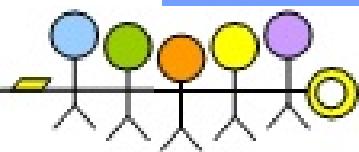


# Using Watchdog

- To use watchdogs for periodic code execution :

```
wdId = wdCreate();
wdStart ( wdId, DELAY_PERIOD, myWdTsr, 0 )
void myWdIsr ( int param)
{
    doit(param);
    wdStart ( wdId, DELAY_PERIOD, mywdIsr, 0 );
}
```

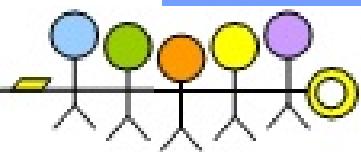
- The ***doit()*** routine might :
  - Poll some hardware device.
  - Unblock some task.
  - Check if system errors are present.



# Missed Deadlines

To recover from a missed deadline :

```
WDOG_ID wdId;
void foo(void)
{
    wdId = wdCreate( );
    /* Must finish each cycle in under 10
seconds */
    FOREVER
    {
        wdStart ( wdId, Delay_10_SEC, fooISR, 0 );
        fooDoWork ( );
    }
}
void fooISR (int param)
{
    /* Handle missed deadline */
    .....
}
```



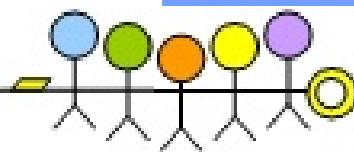
# Stopping Watchdogs

- To cancel a previously started watchdog :

**STATUS wdCancel ( wdId)**

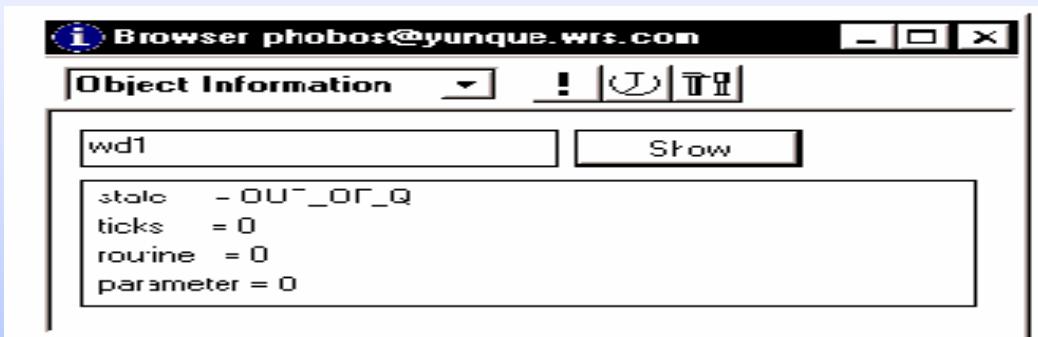
- To deallocate a watchdog timer ( and cancel any previous start) :

**STATUS wdDelete ( wdId)**

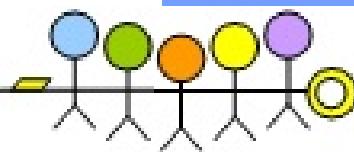
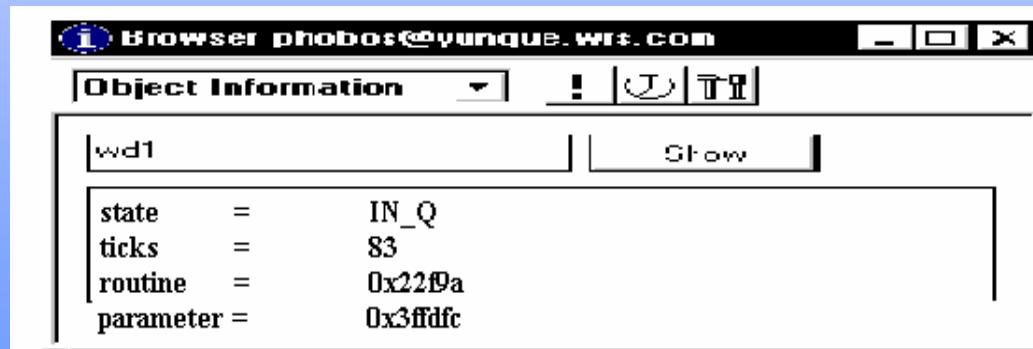


## Watchdog Browser

- After creating, but prior to activating the watchdog, the Browser provides minimal information :

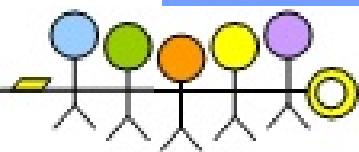


- After activating the watchdog, more useful information is provided :



# Polling Issues

- Can poll at task time or interrupt time.
  - Interrupt time polling is more reliable .
  - Task time polling has a smaller impact on the rest of the system.
- To poll at task time, two options :
  - *taskDelay( )* - fastest, but may “drift”.
  - *wdStart( ) + semGive( )* - more robust.

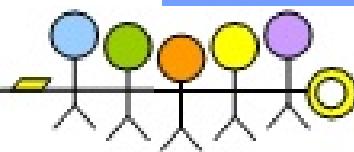


## Polling Caveats

- The following code is accurate only if the system clock rate is a multiple of 15hz :

```
void myWdISR ()  
{  
    pollMyDevice() ;  
    wdStart (myWdId, sysClkRateGet )/15, 0) ;  
}
```

- Do not set the system clock rate too hiogh because there is OS overhead in each clock tick.
- Use auxillary clock to poll at high speeds.



# Auxillary Clock

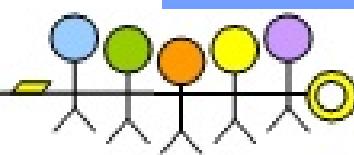
- For high speed polling, use the auxillary clock.
- Precludes using **spy**, which also uses the auxillary clock.
- Some routines to manipulate auxillary clock :

***sysAuxClkConnect( )*** Connect ISR to Aux clock

***sysAuxClkRateSet( )*** Set Aux clock rate.

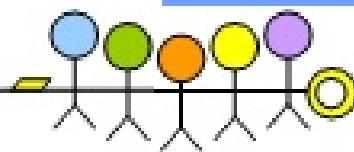
***sysAuxClkEnable( )*** Start Aux clock .

***sysAuxClkDisable( )*** Stop Aux clock



# Summary

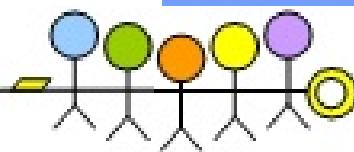
- Using signals for exception handling :
  - *signal( )*
  - *exit( )*
  - *taskRestart( )*
  - *longjmp()*
- Interrupt Service routines have a limited context :
  - No blocking.
  - No I/O system calls



# Summary

- Polling with timers :

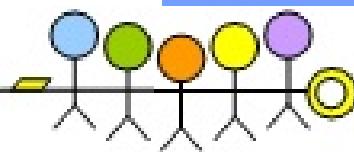
	low speed	high speed
Interrupt time	wd timer	aux clock
task time	taskDelay, or wd timer + semGive	



# Chapter

# 10

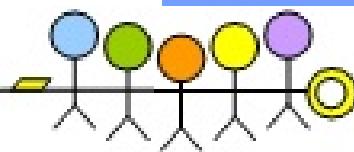
# I/O System



# Chapter

# 10

# I/O System



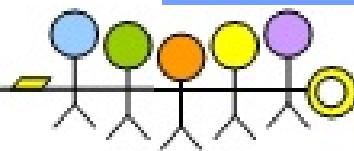
# I/O System

Introduction

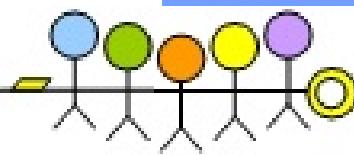
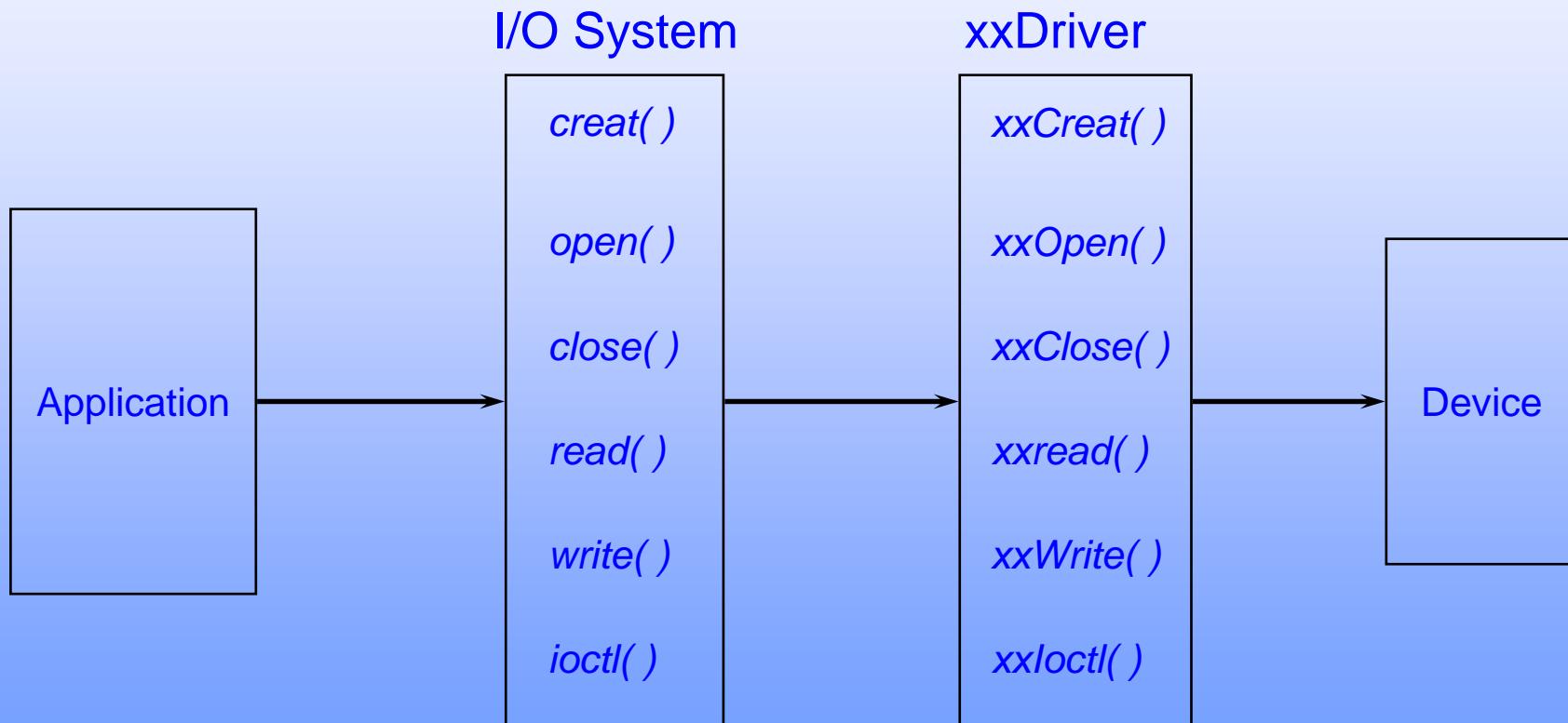
Basic I/O

Select( )

Standard I/O

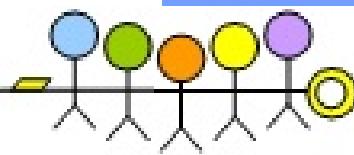


# I/O System Interface



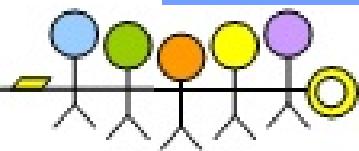
## I/O System Benefits

- Simple interface
- Standard (portable) interface
- **Select( )**
- I/O redirection



## Driver Installation

- A device driver must be installed in the I/O system.
- Then VxWorks can call driver-specific routines when generic routines are called: e.g. calling `xxRead( )` when `read( )` is called on an XX device.
- Installation done automatically for VxWorks drivers included in VxWorks image.
  - `ttyDrv( )`.
  - `pipeDrv( )`.
- Must call drivers's `xxDrv( )` routine before using a third party driver.



# Device Creation

- Each device driver has a device creation routine:

Pipes

Created at your request with  
*pipeDevCreate( )*.

Serial devices

Created automatically at system  
start up (via *ttyDevCreate( )*).

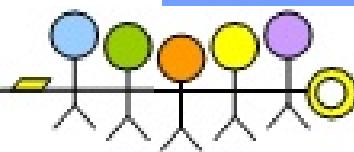
Remote file systems See *Network Basics* Chapter

Local file systems

See *Local file Systems* chapter.

Third-party devices *xxDevCreate( )*.

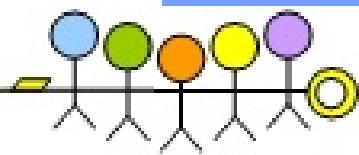
- These routines each initialize instances of a particular device type.



# Devices

-> devs

drv	name
0	/null
1	/tyCo/0
1	/tyCo/1
4	columbia:
2	/pipe/dream
2	/pipe/cleaner
5	/vio



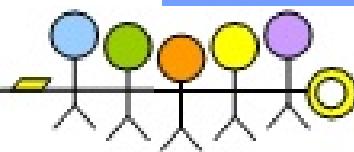
# I/O System

Introduction

Basic I/O

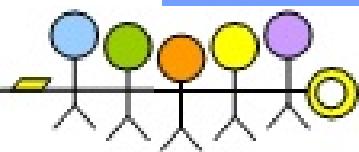
Select( )

Standard I/O



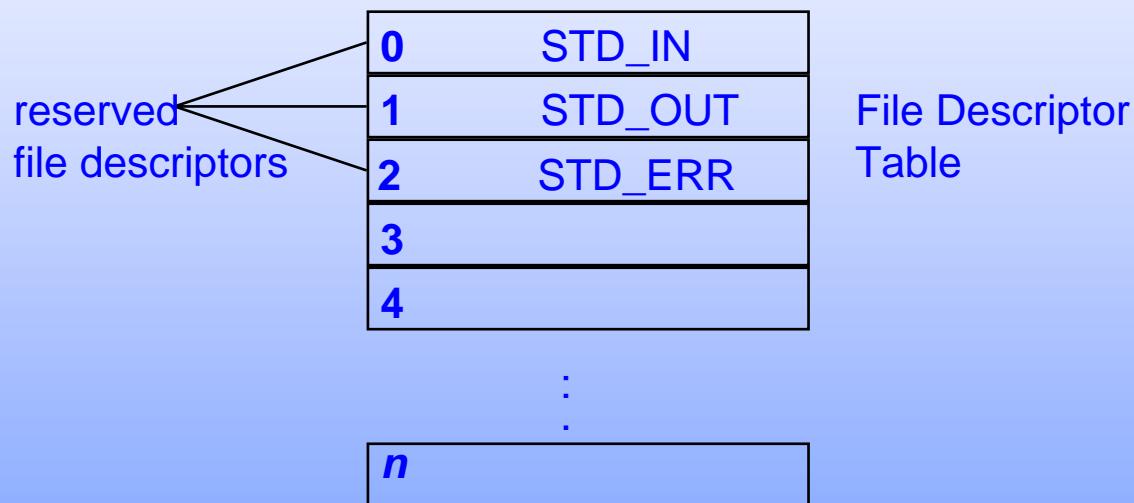
# File Descriptors

- Integer that identifies a *file* (file or device).
- Assigned by *open( )* or *creat( )*.
- Used by *read( )*, *write( )*, *ioctl( )* and *close( )* to specify file.
- File descriptor table is global
- Table size defined by the symbolic constant **NUM\_FILES** (default of 50 is specified in **configAll.h**).

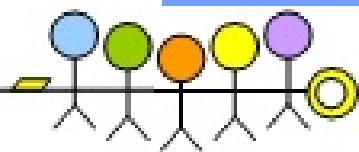


# Standard Input, Standard Output, and Standard Error

- The first three file descriptors are predefined by the system and are never reassigned by the system.



- These three file descriptors (0-2) are never returned by *creat()* or *open()*.



# Standard Input, Output, and Error

- By default, these are directed to the console at system startup.
- Global assignments may be changed using:

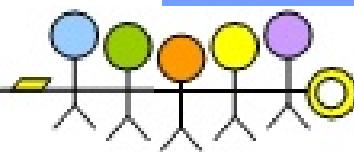
**Void ioGlobalStdSet(stdFd, newFd)**

**Example:**

```
fd = open ("columbia:/vx/errLog", O_RDWR, 0);
ioGlobalStdSet ( STD_ERR, fd);
```

- Redirect I / O on a per task basis with :

**void iotaskStdSet (taskId, stdFd, newFd)**



# Accessing Files

## `int open (name, flags, mode)`

`name` Name of file to open.

`flags` Specifies type of access:

`O_RDONLY` Open file for reading only.

`O_WRONLY` Open file for writing only.

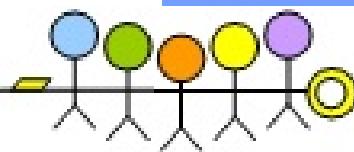
`O_RDWR` Open for reading and writing.

`O_TRUNC` Truncate file on open.

`O_CREAT` Create file if it doesn't exist

`mode` Permissions used when creating a file in an NFS system.

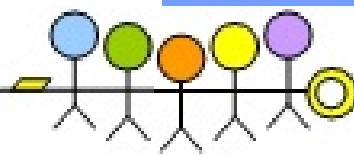
- Returns file descriptor on success, otherwise **ERROR**
- *creat( )* routine similar to *open( )*, but for new files on a file system device.



# Examining the File Descriptor Table

-> iosFdShow

fd	name	drv
3	/tyCo/0	1
4	(socket)	3
10	/pipe/dream	2
12	/pipe/cleaner	2

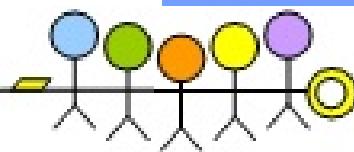


# Closing Files

## STATUS **close(fd)**

fd                  File descriptor returned from *open( )* or *creat( )*

- Flushes buffers.
- Frees up resources associated with file descriptor.
- Tasks must explicitly close files when they are no longer needed.
  - File descriptor table is fixed size.
  - VxWorks does not close files on task deletion.



## Read / Write

**int read (fd, buffer, nBytes)**

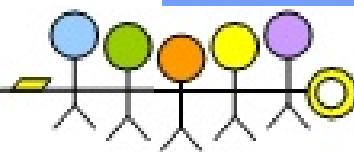
**int write (fd, buffer, nBytes)**

**fd** File descriptor returned by *open()* or *creat()*.

**buffer** Address of buffer. Will be filled on a read,  
copied to device on write.

**nBytes** Maximum number of bytes to read / write.

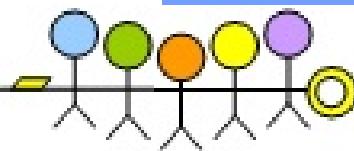
- Returns the number of bytes read / written or **ERROR** if unsuccessful.



# ioctl

## int ioctl (fd, command, arg)

- |         |   |
|---------|---|
| fd      | File descriptor returned from <code>open()</code> or <code>creat()</code> .   |
| command | Integer identifying some driver specific command.<br>Symbolic constants typically used.   |
| arg     | Type and value depends on command. <ul style="list-style-type: none"><li>• Driver manual pages lists valid <code>ioctl</code> commands for that driver.</li></ul> |



## ioctl() Examples

- To set the baud rate of a serial device:

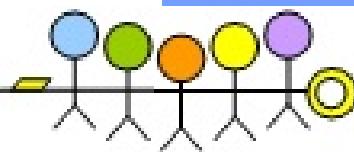
```
status = ioctl (fd, FIOBAUDRATE, baudRate);
```

- To find the number of messages in a pipe:

```
status = ioctl (fd, FIONMSGS, &nMsgs);
```

- To get / set your current file offset:

```
status = ioctl ( fd, FIOSEEK, newOffset);  
position = ioctl ( fd, FIOWHERE, 0);
```

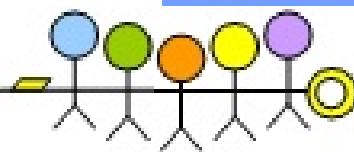


# Setting Options for a Serial Device

```
status = ioctl (fd, FIOSETOPTIONS, option);
```

Symbolic constants for options may be OR'ed:

1	OPT_ECHO	Echo of input.
2	OPT_CRMOD	Convert \n to CR-LF on output.
4	OPT_TANDEM	Implement ^S / ^Q flow control.
8	OPT_7_BIT	Strip parity bit from 8 bit input.
16	OPT_MON_TRAP	Reboot on ^X.
32	OPT_ABORT	Restart <i>target</i> shell on ^C.
64	OPT_LINE	Allow line editing before \n.
<hr/>		
0	OPT_RAW	All options off ( raw mode).
127	OPT_TERMINAL	All options on.



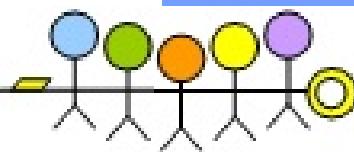
# I/O System

Introduction

Basic I/O

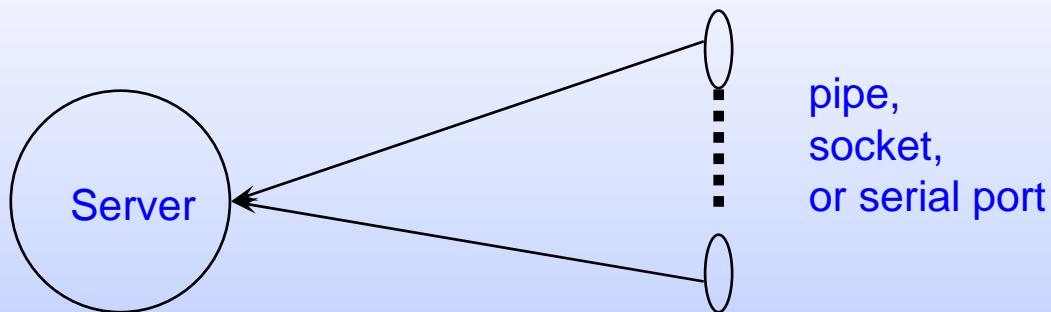
Select( )

Standard I/O

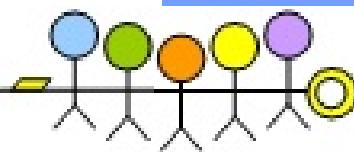


# Overview

***select( )***



- ***select( )*** allows a task to wait for activity on set of file descriptors.
- Requires driver support:
  - VxWorks pipes, sockets and serial device drivers support ***select( )***.
  - Third party drivers may also support ***select( )***.
- Also used to pend with a timeout.



# select()

- select调用及宏FD\_CLR、FD\_ISSET、FD\_SET、FD\_ZERO用于I/O多路复用。
- int select(

```
    int          width,           /*number of bits to examine from 0*/
    fd_set      *pReadFds,        /*read fds*/
    fd_set      *pWriteFds,       /*write fds*/
    fd_set      *pExceptFds,      /*exception fds unsupported*/
    struct timeval *pTimeout     /*max time to wait ,NULL=forever*/
)
```

pReadFds 监测是否有字符可以从某个描述符读入

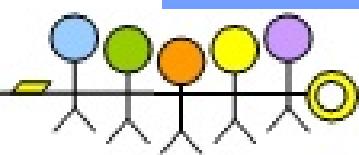
pWriteFds 监测是否某个描述符准备好了可以立即写入

pExceptFds 监测是否某个描述符有异常出现

FD\_SET(fd,&fdset) 从一个集合中添加一个描述符

FD\_CLR(fd,&fdset) 从一个集合中删除一个描述符

FD\_ZERO(&fdset) 清空一个集合



## struct fd\_set

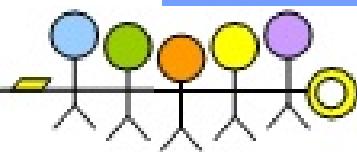
- used by `select()` to specify file descriptors.
- Conceptually an array of bits, with bit N corresponding to file descriptor N.
- Manipulated via a collection of macros:

`FD_SET(fd, &fdSet)` Sets the bit which corresponds to *fd*.

`FD_CLR(fd, &fdSet)` Clears the bit which corresponds to *fd*.

`FD_ISSET(fd, &fdSet)` Returns **TRUE** if the *fd* bit is set, else **FALSE**.

`FD_ZERO(&fdSet)` Clears all bits.

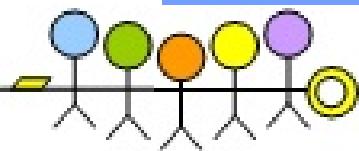


# Select

**int select(width, pReadFds, pWriteFds, pExceptFds, pTimeOut)**

width	Number of bits to examine in <i>pReadFds</i> and <i>pWriteFds</i> .
<i>pReadFds</i>	<i>struct fd_set</i> pointer for the file descriptor we wish to read.
<i>pWriteFds</i>	<i>struct fd_set</i> pointer for the file descriptor we wish to write.
<i>pExceptFds</i>	Not implemented.
<i>pTimeOut.</i>	Pointer to a <i>struct timeval</i> , or <b>NULL</b> to wait forever.

- Returns number of active devices, or **ERROR**.



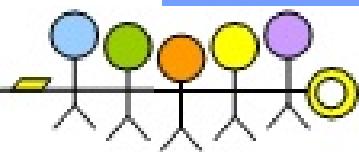
# I/O System

Introduction

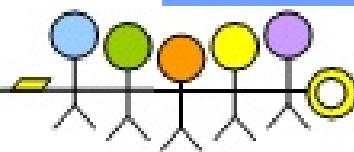
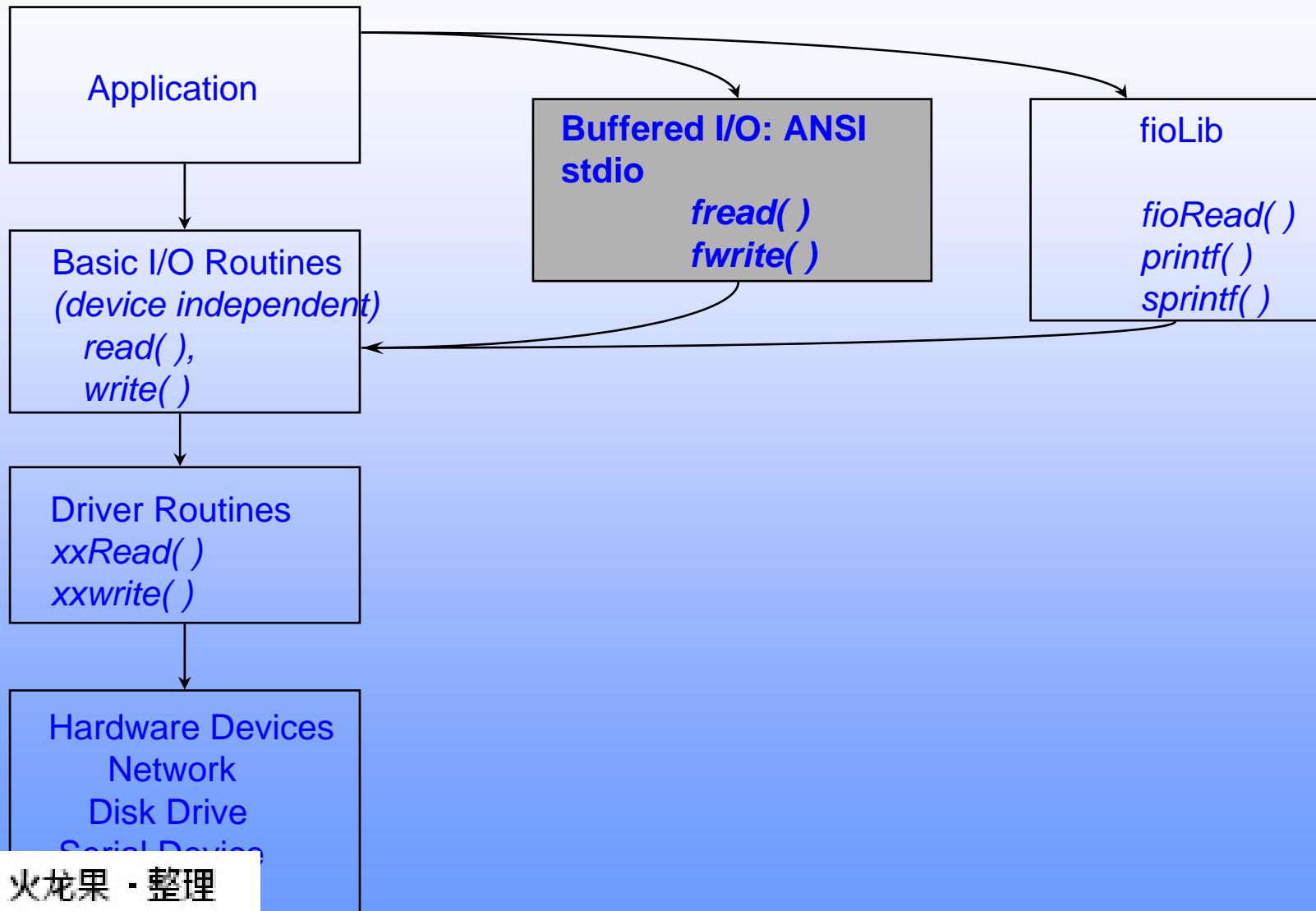
Basic I/O

Select( )

Standard I/O



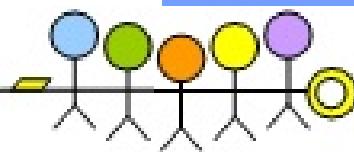
# Standard I/O



# Standard I/O Functions

- **ansiStdio** functions include:

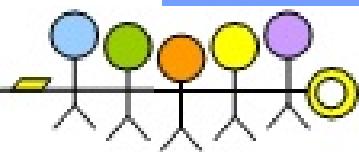
<i>fopen( )</i>	opens a stream to a device or file.
<i>fclose( )</i>	closes a stream.
<i>fread( )</i>	reads data into an array.
<i>fwrite( )</i>	writes data to a stream.
<i>getc( )</i>	gets next character from a stream.
<i>putc( )</i>	puts a character to a stream.
<i>ungetc( )</i>	returns character to input stream.



# File Pointers

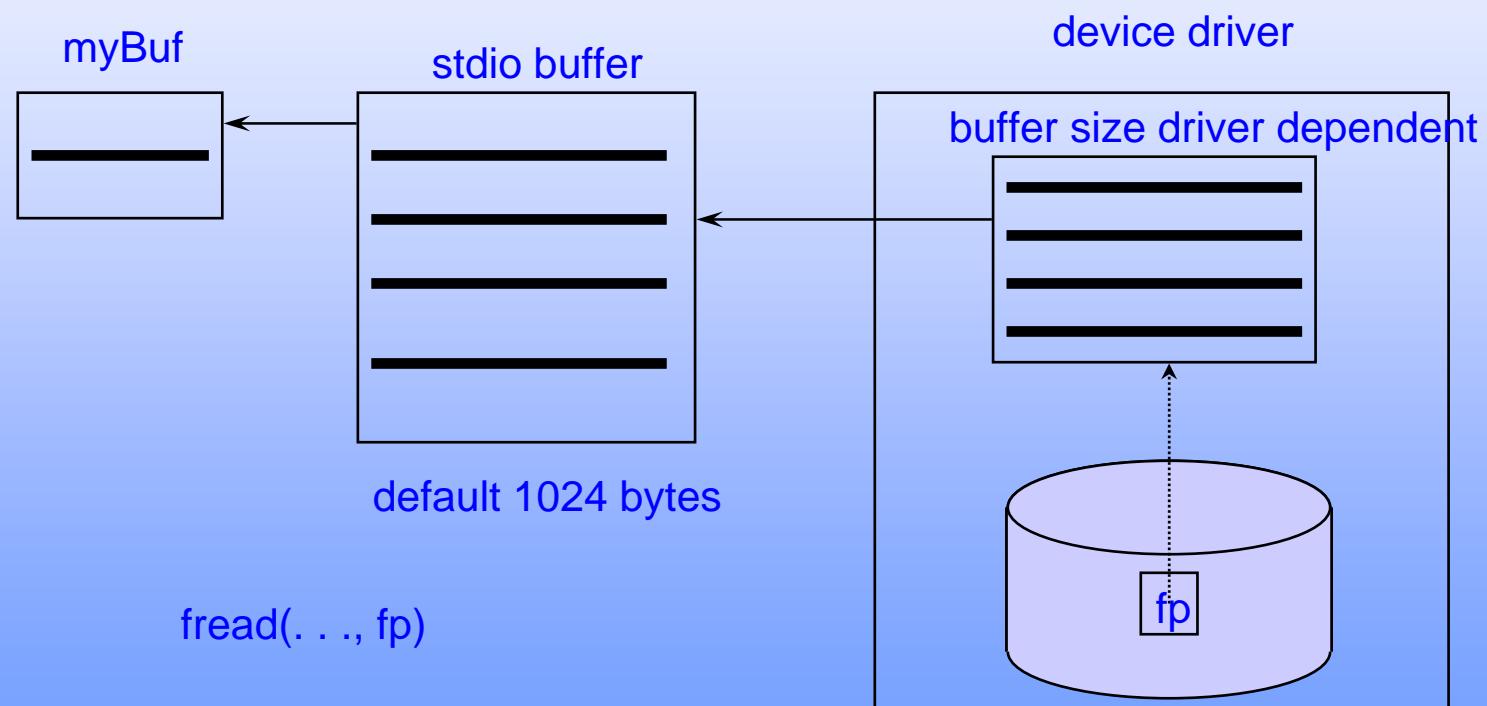
- **ansiStdio** routines use *file pointers* (pointers to FILE data structure) instead of file descriptors. The FILE data structure, typedef'ed in **stdio.h**, contains :
  - The underlying file descriptor.
  - Pointers, etc., for managing the file buffers.
- *stdin*, *stdout*, and *stderr* are file pointers created using file descriptors 0,1, and 2.

File Descriptors (vxWorks.h)	File Pointers (stdio.h)
STD_IN	stdin
STD_OUT	stdout
STD_ERR	stderr

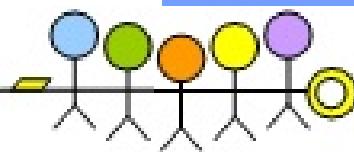


# Standard I/O Buffers

Private buffers minimize driver access:



`fread(..., fp)`



# Standard I/O Library Caveats

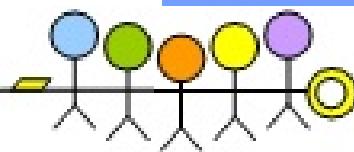
- Stdio buffers **not** protected with semaphores. Two tasks should **not** use the same *fp* at the same time.
- stdio library contains routines and macros.
  - Breakpoints cannot be set on macros.
  - Routines useful for access from the shell.
  - Macros override routines when including **stdio.h**.
- Symbolic constants in **configAll.h** (default):

**INCLUDE\_STDIO**

Initialize library.

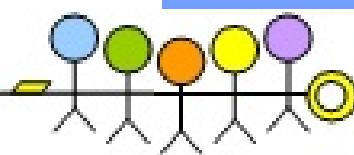
**INCLUDE\_ANSI\_STDIO**

Link *all* **ansiStdio**  
routines into VxWorks.

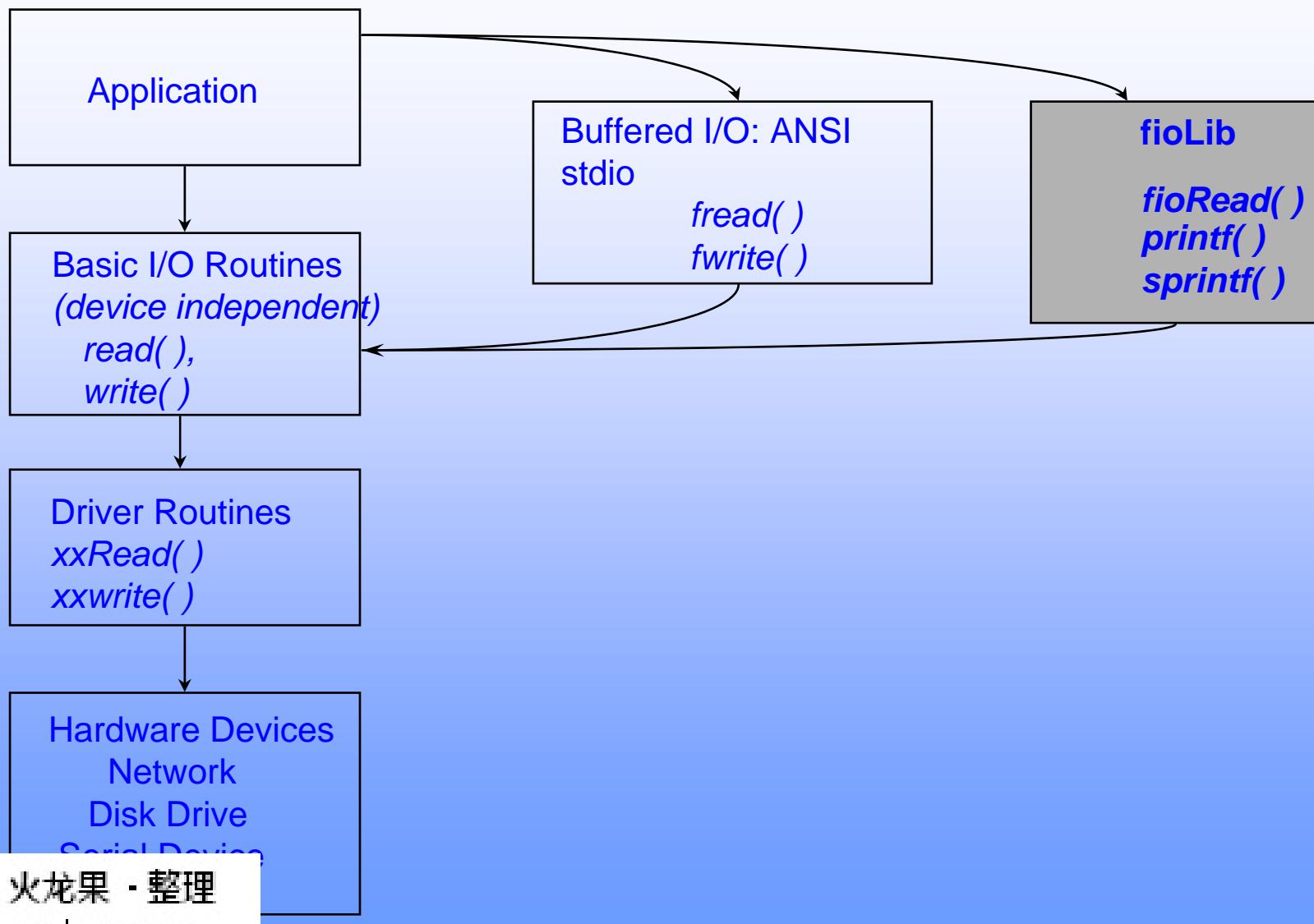


## Other ANSI Libraries

Manual Page	Header File	Description	Examples
ansiCtype	ctype.h	Character testing and conversion.	<i>isdigit( )</i> <i>isalpha( )</i> <i>toupper( )</i>
ansiString	string.h	String manipulation	<i>strlen( )</i> <i>strcat( )</i> <i>strcpy( )</i>
ansiStdarg	stdarg.h	Variable number of arguments	<i>va_start( )</i> <i>var_arg( )</i> <i>va_end( )</i>
ansiStdlib	stdlib.h	Standard conversion routines	<i>atoi( )</i> <i>atof( )</i> <i>rand( )</i>

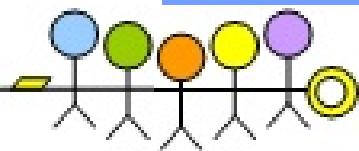


# Formatted I/O



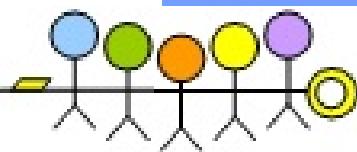
# Formatted I/O

- **fioLib** contains routines for non-bufferd formatted I/O.
- Includes *printf()* and *sprintf()* which are normally part of **stdio**.
- **fioLib** allows **ansiStdio** to be excluded without losing functionality.



# Buffered Vs. Non Buffered I/O

Destination	Buffered I/O (ansiStdio)	Unbuffered I/O (fioLib)
stdout	<code>fprintf(stdout, ...)</code>	<code>printf(...)</code>
stderr	<code>fprintf(stderr, ...)</code>	<code>printErr(...)</code>
other	<code>fprintf(fp, ...)</code>	<code>fdprintf(fd, ...)</code>



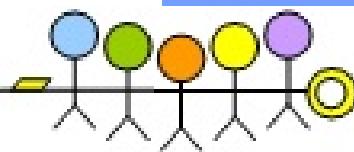
# UNIX and VxWorks - I/O Comparison

- Similarities:

- Source code compatible..
- Devices have names like files.
- Concept of current directory.
- Concept of *stdin*, *stdout*, *stderr*.

- Differences:

- Speed.
- File descriptors in UNIX are local to a process; in VxWorks they are global.
- VxWorks uses only a subset of UNIX *open( )* flags.
- *printf( )* is not buffered.



# Summary

- Basic I/O routines:

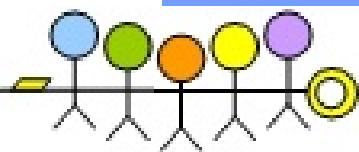
<i>open( )</i>	<i>close( )</i>
<i>read( )</i>	<i>write( )</i>
<i>ioctl( )</i>	<i>creat( )</i>

- Buffered I/O (**ansiStdio**) built on top of basic I/O using file pointers:

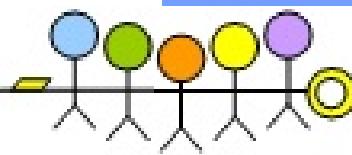
<i>fopen( )</i>	<i>fclose( )</i>
<i>fread( )</i>	<i>fwrite( )</i>
<i>fprintf( )</i>	

- Formatted Non-buffered I/O (**fioLib**):

<i>printf( )</i>	<i>sprintf( )</i>
<i>fdprintf( )</i>	

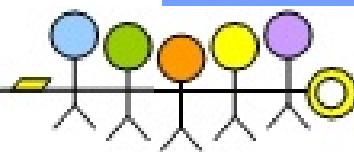


Now  
Have a rest



# Chapter 11

# Local File Systems



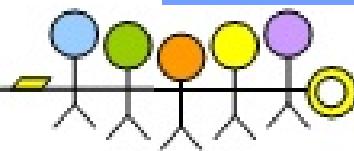
# Local File systems

Overview

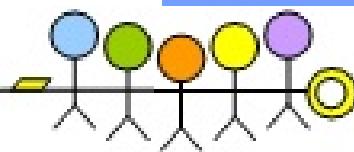
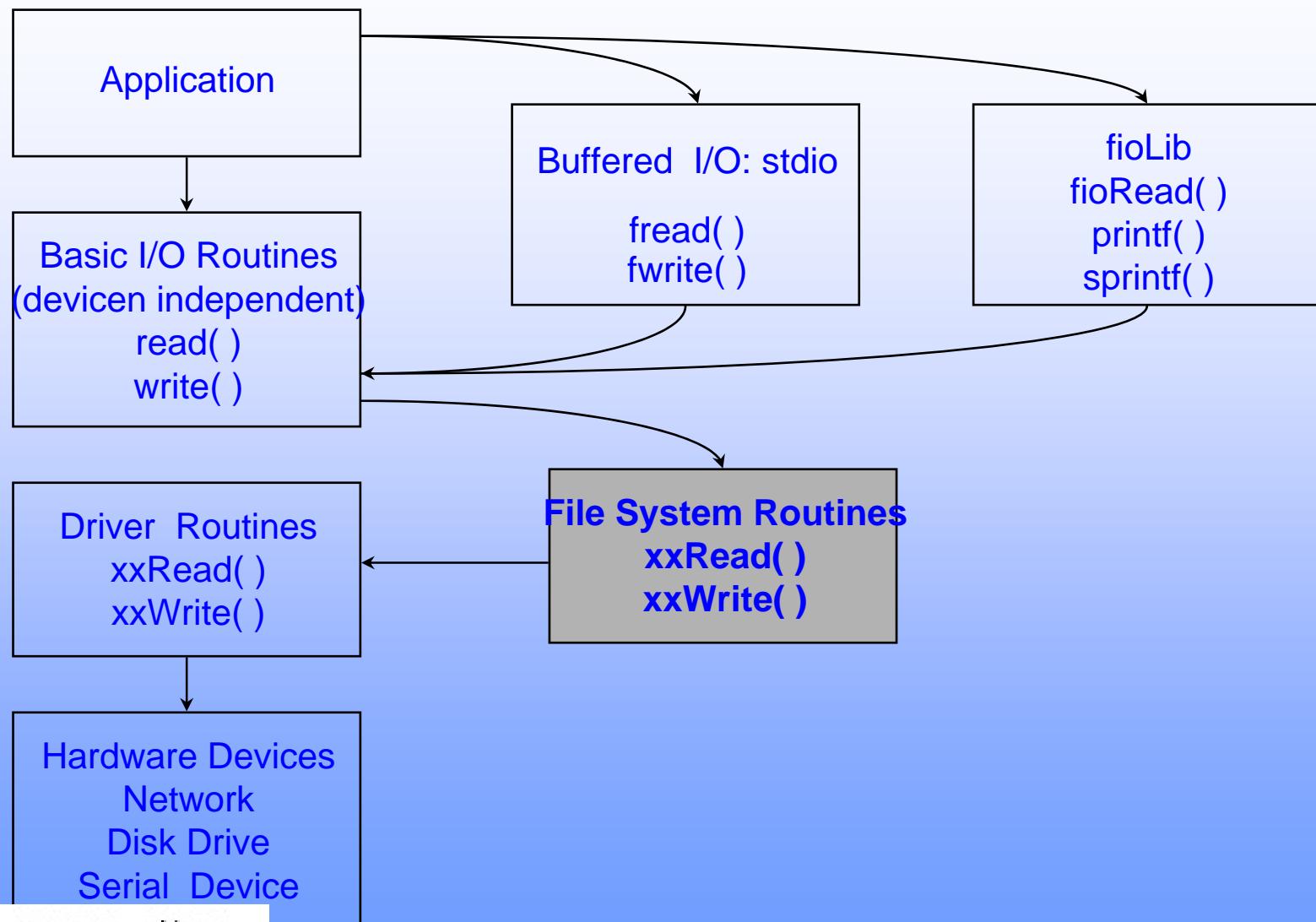
DOS File System

Raw File System

SCSI Devices



# File Systems



# Local File Systems

- Available block device drivers :

ramDrv      Makes memory look like a disk

scsiLib      Supports SCSI random-access devices.

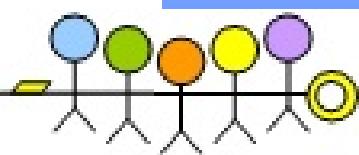
xxDrv      Third Party block drivers.

- Available file system libraries :

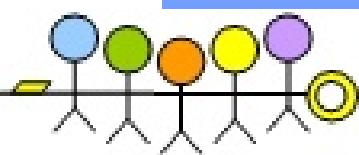
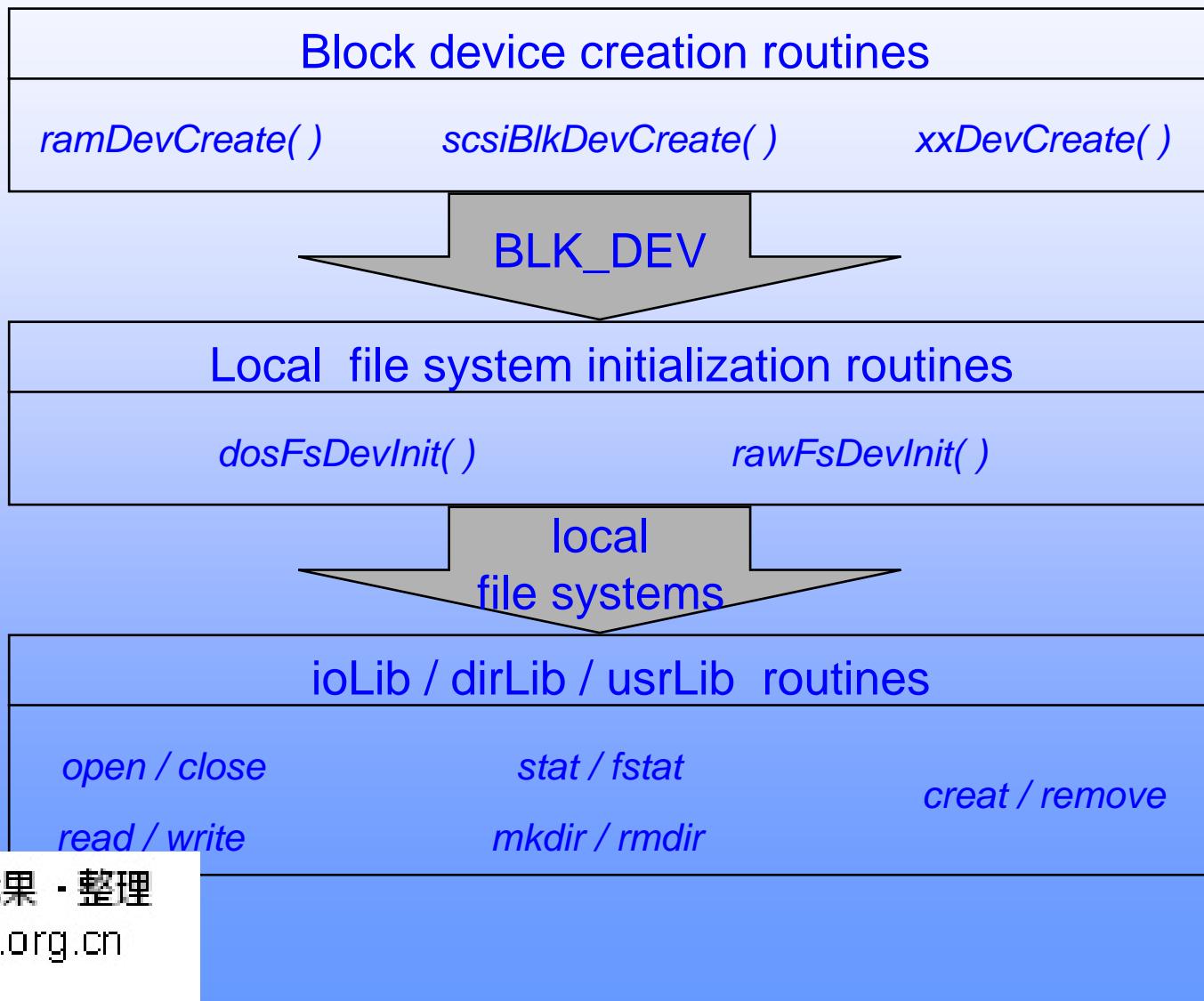
dosFsLib      MS-DOS compatible.

rawFsLib      Supports disk as a single file.

rt11FsLib      For backward compatibility.



# Initialization and Use



# Creating a RAM Disk

```
BLK_DEV * ramDevCreate ( ramAddr , bytesPerBlk ,  
                         bytesPerTrack , nBlocks , blkOffset )
```

ramAddr      Memory location of ram disk ( 0 = *malloc( )* ).

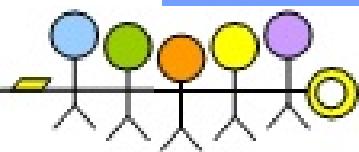
bytesPerBlk    Number of bytes per block.

blksPerTrack   Number of blocks per track.

nBlocks        Size of disk in blocks.

blkOffset      Number of blocks to skip. Typically zero.

- Returns a pointer to a **BLK\_DEV** structure describing the RAM disk, or **NULL** on error.
- Define **INCLUDE\_RAMDRV** to include **ramDrv** in your vxWorks image.

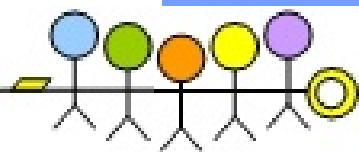


## A Simple Example

```
-> pBlkDev = ramDevCreate(0,512,400,400,0)
-> dosFsMkfs ("/RAM1" , pBlkDev)

/* Create and write to a file. Flush to RAM disk */
-> fd = creat("/RAM1/myfile",2)
-> writeBuf = "This is a string.\n"
-> write(fd,writeBuf,strlen(writeBuf)+1)
-> close(fd)

/* Open file and read contents. */
-> readBuf = (char *) malloc(100)
-> fd = open("/RAM1/myfile",2)
-> read(fd , readBuf , 100)
-> printf readBuf
This is a string.
```



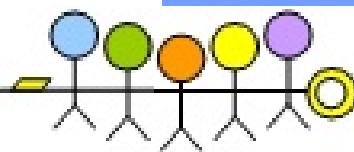
# Local File systems

Overview

DOS File System

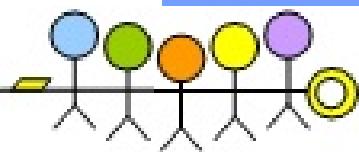
Raw File System

SCSI Devices



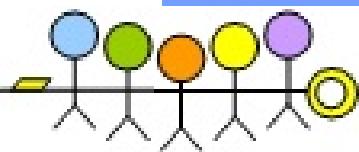
# DOS File System

- Hierarchical file system
- Restrictive file names.
- Compatible with MS-DOS file systems up to release 6.2.
- Disks are interchangeable with those created on PCs using MS-DOS ( if configured properly).



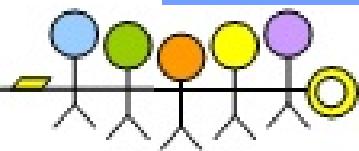
## Configuring dosFs

- Define **INCLUDE\_DOSFS**, in **configAll.h** , ( excluded by default ).
- Maximum number of dosFs files that can be open at a time with **NUM\_DOSFS\_FILES** ( default of 20 defined in **configAll.h** ).
- Specify Volume Configuration information in a **DOS\_VOL\_CONFIG** data structure :
  - Disk layout.
  - VxWorks options.



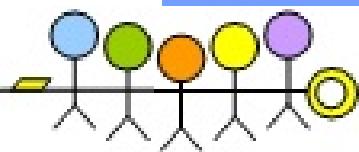
## Caveat : Data Integrity

- To improve performance, FAT and directory changes are not flushed to disk immediately.
- To flush FAT and directory changes (before removing media) , use ***dosFsVolUnmount( )***.
- Alternatively , set options in the *dosvc\_options* field of a partition's **DOS\_VOL\_CONFIG** structure :
  - **DOS\_OPT\_AUTOSYNC** Flush FAT and directory changes.
  - **DOS\_OPT\_CHANGENOWARN** Remount dosFs on ***open()*** or ***creat()***.
  - Setting these options will degrade performance.
- Flush file changes with, ***ioct(fd, FIOSYNC, 0)***.



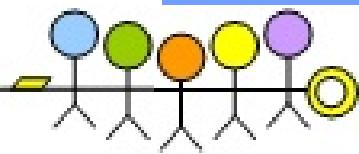
## Caveat : DOS file names

- By default, standard 8 + 3 DOS file names are used.
- Enable UNIX - style file names by setting  
**DOS\_OPT\_LONGNAMES** option bit in  
**DOS\_VOL\_CONFIG** data structure.
- UNIX - style file names are not PC compatible.



## Caveat : PC Compatibility

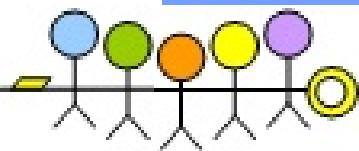
- A disk initialized on a PC will work with VxWorks.
- For PC compatibility, a disk initialized with VxWorks
  - Must use correct media byte.
  - Do not enable UNIX - style file names enabled with **DOS\_OPT\_LONGNAMES**.



# Configuration : New File System

To create a new DOS filesystem with custom configuration parameters :

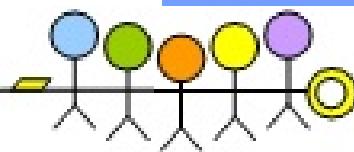
1. pBlkDev = xxDevCreate(...);
2. Initialize **DOS\_VOL\_CONFIG** structure.
3. ***dosFsDevInit***( "/DOS" , pBlkDev , pConfig);
4. fd = open( "/DOS" , O\_WRONLY , 0 );
5. ioctl ( fd , FIODISKFORMAT , 0); /\* if necessary \*/
6. ioctl ( fd , FIODISKINIT , 0);
7. close( fd );



## Configuration : New Filesystem

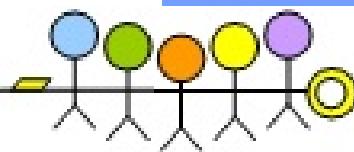
To create a new DOS filesystem with default configuration parameters :

1. pBlkDev = xxDevCreate(...);
2. ***dosFsMkfs***( "/DOS", pBlkDev);



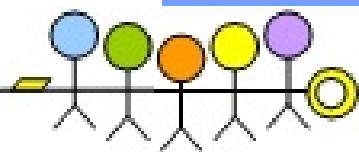
# Configuration : Using an Existing File System

- To remount an existing file system :
  1. pBlkDev = xxDevCreate(...);
  2. pDesc = dosFsDevInit( "/DOS", pBlkDev , NULL );
  3. dosFsVolOptionsSet (pDesc , options); /\* if needed \*/
- Typically , file system are configured in startup code.



# Contiguous File Support

- To pre-allocate contiguous disk space for a file :  
`ioctl (fd , FIOCONTIG,numBytes )`  
Must be called before anything is written to file.
- Example :  
`fd = creat( "/dos1/myDir/myFile" , O_RDWR ) ;  
ioctl ( fd , FIOCONTIG , 0x10000 ) ;`
- To obtain the largest contiguous block available , set  
*numBytes* to **CONTIG\_MAX**.
- Pre-allocated space may be reclaimed with :  
`ioctl (fd , FIOTRUNC , newLength)`



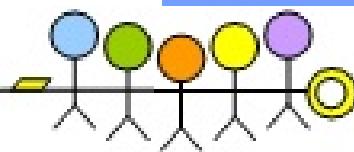
# Local File systems

Overview

DOS File System

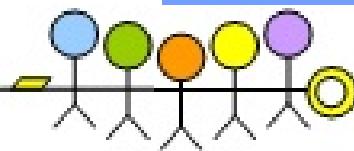
Raw File System

SCSI Devices



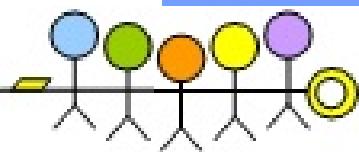
# Introduction to the Raw File System

- Wind River Systems defined.
- Fastest , least structured file system.
- Handles device (partition) as one file.
  - Analogous to the UNIX raw interface to a block device.
  - No Directories or other structure on disk.



# Configuring the Raw File System

- Define **INCLUDE\_RAWFS** in **configAll.h** (excluded by default ).
- Maximum number of rawFs files that can be open at a time controlled by **NUM\_RAWFS\_FILES** ( default of 5 is defined in **configAll.h.**)



# Raw FS Initialization

- To initialize a raw device :

**raw FsDevInit ( devName , pBlkDev )**

devName      Name of raw file system device being created.

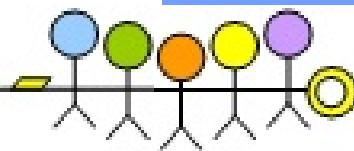
pBlkDev      Pointer to the **BLK\_DEV** returned from  
**xxDevCreate( ).**

Returns NULL on error.

- To low-level format the disk :

- Open a file descriptor on the rawFs device.
- Call the **FIODISKFORMAT** ioctl.

- With no structure on the disk to initialize, there is no **diskInit( )** routine.



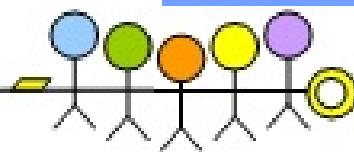
# Local File systems

Overview

DOS File System

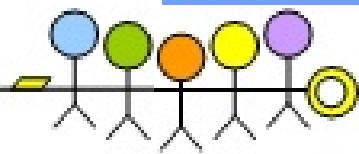
Raw File System

SCSI Devices

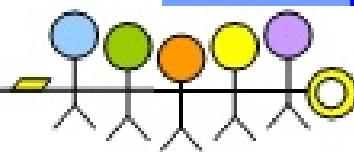
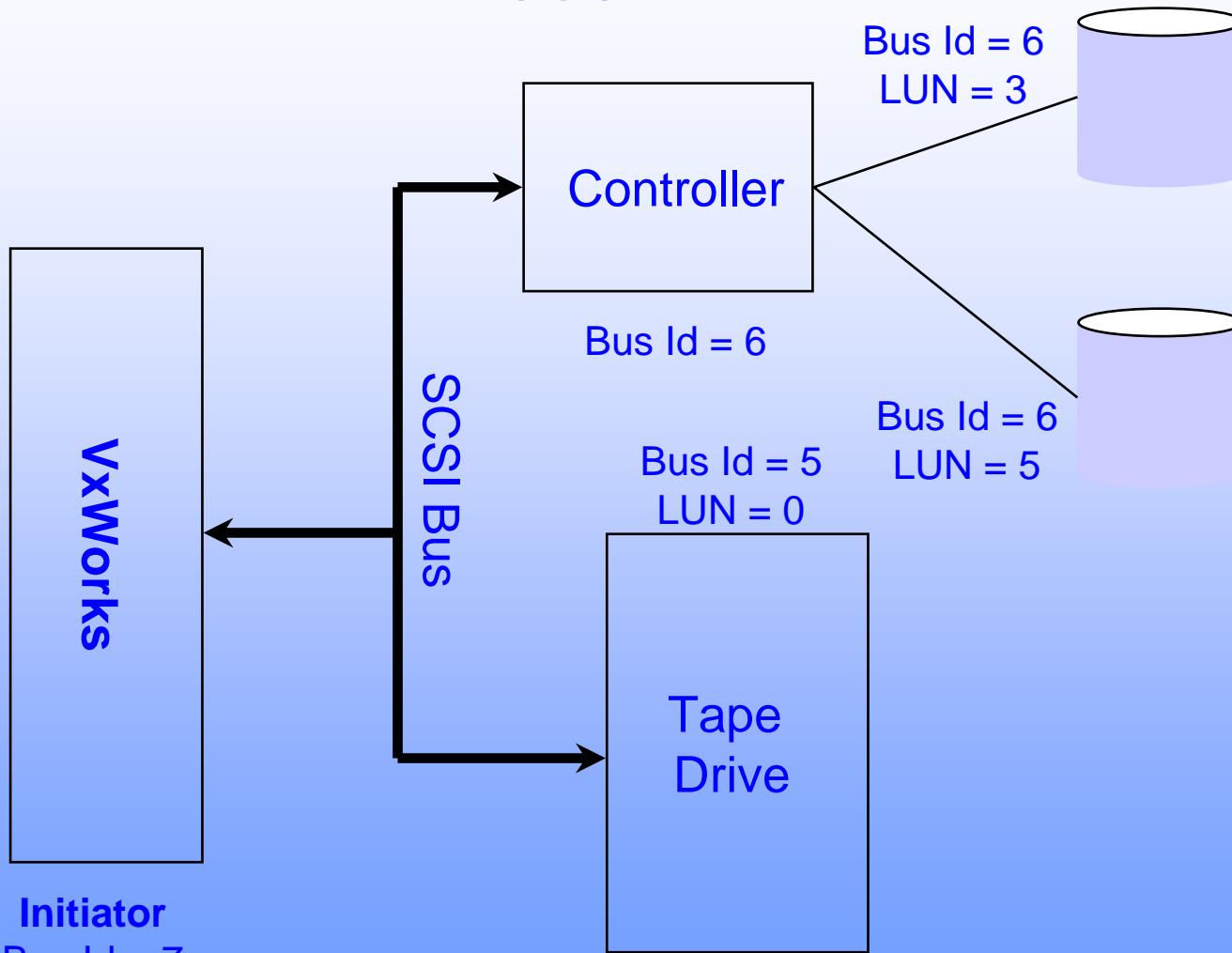


# SCSI Overview

- SCSI - Small Computer System Interface.
- ANSI standard providing device independence.
  - All devices “speak the same language.”
  - Less driver writing is required.
- Support for SCSI-1 and SCSI-2 devices.
- Support for general and block drives is supplied by **scsiLib**.
- Support for sequential devices is provided by **scsiSeqLib**.

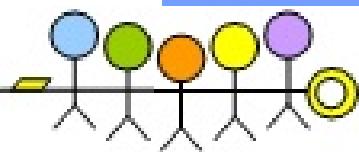


# SCSI Bus



## SCSI -1 Restrictions

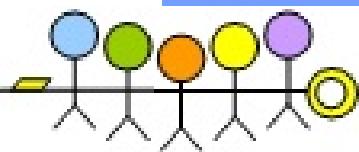
- VxWorks must be the sole initiator.
- Only **SCSI I** random-access block devices conforming to the Common Command Set is supported.
- Disconnect / reconnect is not supported.
- Synchronous transfers are not supported.
- Drivers for non-random-access devices ( e.g., a tape drivse ) must be written by the user :
  - **scsiLib** provides routines to put commonly used SCSI commands on the bus..
  - **scsilioctl ( )** allows arbitrary SCSI commands to be put on the bus.



# SCSI -1 Support configuration

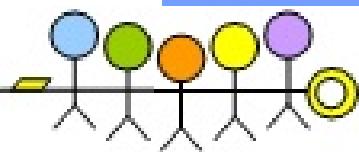
- To configure VxWorks for SCSI-1 support , selectively define the appropriate macro(s) by moving them outside the **# if FALSE** block in *wind/target/config/bspName/config.h* :

```
# if FALSE
# define INCLUDE_SCSI
# define INCLUDE_SCSI_BOOT
# define INCLUDE_SCSI_DMA /* if supported by BSP */
# define INCLUDE_DOSFS
# define INCLUDE_TAPEFS /* SCSI -2 only */
# define SYS_SCSI_CONFIG
# endif
...
```



## SCSI -2 Features

- Supports all mandatory features of the SCSI-2 protocol , and many optional features.
- SCSI - 2 features not supported by SCSI-1 :
  - Multiple Initiators.
  - Disconnect / reconnect.
  - Mandatory direct access commands.
  - Mandatory sequential access commands.
  - Identify message.
  - Synchronous data transfer.
  - Fast SCSI
  - Wide SCSI
  - Tagged command queuing.

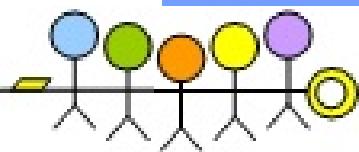


## SCSI -2 Support configuration

- To include SCSI-2 support , the macro **INCLUDE\_SCSI2** (as well as **INCLUDE\_SCSI**) must be defined in **wind/target/config/target/config.h**
- **INCLUDE\_SCSI2** must be defined before the include control line for the BSP header file ***bspName.h***

```
# define INCLUDE_SCSI2  
# include "configAll.h"  
# include "bspName.h"
```

- Access to SCSI-2 devices :
  - DOS or RAW file systems on direct-access devices.
  - Tape File System on sequential-access devices.



# Initializing SCSI Block Devices

Include SCSI-2 Support in VxWorks

```
#define INCLUDE_SCSI2
```

pSysScsiCtrl

Initialization of SCSI Peripheral

```
scsiPhysDevcreate(pSysScsiCtrl,busId,LUN,...)
```

pScsiPhysDev

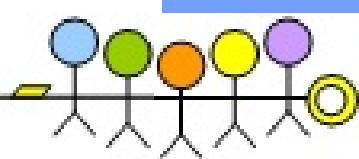
Initialize Logical Partition

```
scsiBlkDevcreate( )
```

pBlkDev

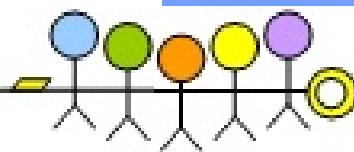
Initialize File System

```
dosFsDevInit( )
```



# Configuration of SCSI Devices

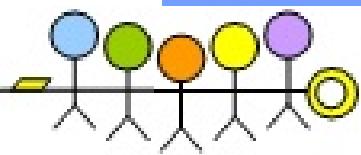
- When support for SCSI is enabled, the system initialization code will :
  - Initialize SCSI controller.
  - Configure SCSI peripherals.
- Peripheral configuration call chain :
  - `usrRoot( ) => usrScsiConfig( ) => sysScsiConfig( )`
  - `sysScsiConfig( )` is called only if `SYS_SCSI_CONFIG` is defined.
- Configuring specific peripherals requires writing the `sysScsiConfig( )` function.
- The `sysScsiConfig( )` function should be placed in `sysScsi.c` (preferably) or `sysLib.c` in the BSP directory. Do not modify `wind/target/src/config/usrScsi.c`



# Configuration : Hard Disks

```
0  /* configure Winchester at busId=2 , LUN=0 */  
1  if((pSpd20=  
2      scsiPhysDevCreate(pSysScsiCtrl,2,0,0,NONE,0,0,0))  
3      == (SCSI_PHYS_DEV *) NULL)  
4  {  
5      SCSI_DEBUG_MSG("usrScsiConfig:"\  
6          "scsiPhysDevCreate failed. \n");  
7      return ERROR;  
8  }  
9  /* create block devices */  
10 if((pSbd0=scsiBlkDevcreate (pSpd20,0x10000,0)==  
11     NULL) ||  
12     ((pSbd1=scsiBlkDevcreate (pSpd20,0x10000,0x10000))  
13     == NULL)  
14     return (ERROR);  
15 /* Configure file systems on partitions */  
16 if((dosFsDevInit ("/sd0/",pSbd0,NULL)== NULL) ||  
17     (rawFsDevInit ("/sd1/",pSbd1)==NULL)  
18     return (ERROR);  
19 /* Continue with other devices... */
```

The diagram shows four arrows pointing from specific parameters in the code to labels on the right. The first arrow points from the value '2' in the line 'scsiPhysDevCreate(pSysScsiCtrl, **2,0,0,NONE,0,0,0**)' to the label 'Bus Id'. The second arrow points from the value '0' in the same line to the label 'LUN'. The third arrow points from the value '0x10000' in the line 'scsiBlkDevcreate (pSpd20, **0x10000,0**)' to the label 'Size'. The fourth arrow points from the value '0x10000' in the line 'scsiBlkDevcreate (pSpd20,0x10000, **0x10000**)' to the label 'Offset'.



# Initializing SCSI Sequential Devices

Include SCSI-2 Support in VxWorks

```
#define INCLUDE_SCSI2
```

pSysScsiCtrl

Initialization of SCSI Peripheral

```
scsiPhysDevcreate(pSysScsiCtrl,busId,LUN,...)
```

pScsiPhysDev

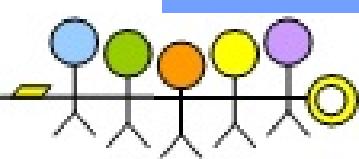
Initialize Logical Partition

```
scsiSeqDevcreate( )
```

pSeqDev

Initialize File System

```
tapeFsDevInit( )
```

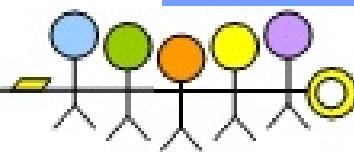


# Sequential Devices

**SEQ\_DEV \* scsiSeqDevCreate (pScsiPhysDev)**

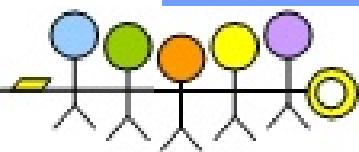
pScsiPhysDev      pointer to a SCSI\_PHYS\_DEV structure  
returned by *scsiPhysDevCreate()*

- Returns a pointer to a **SEQ\_DEV** structure describing the sequential device , or **NULL** on error.
- Only need to define **INCLUDE\_SCSI** and **INCLUDE\_SCSI2** in **config.h**.
- Sequential devices allow blocks of data to be read and written sequentially.



# The Tape File System

- To access a sequential device create a tape file system.
- Can be configured for fixed block size transfers or variable size block transfers.
- No directory structure. No file names used.
- The Sequential device is accessed through the standard I/O interface.
- Define **INCLUDE\_TAPEFS** in **config.h**.



# Tape File System Initialization

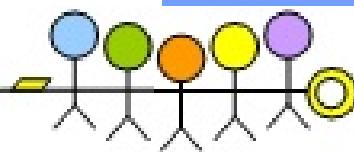
```
TAPE_VOL_DESC * tapeFsDevInit (devName,  
    pSeqDev , pTapeConfig )
```

**devName** Volume name of tape file system device being created.

**pSeqDev**      Pointer to a **SEQ\_DEV** returned from sequential device creation routine.

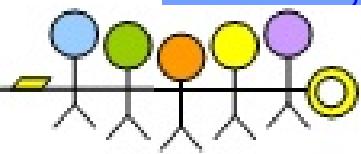
**pTapeConfig** Pointer to a tape configuration structure **TAPE\_CONFIG**.

- **TAPE\_VOL\_DESC** used to reference the file system.



# Configuration : Sequential Devices

```
1  /* configure Exabyte 8mm tape drive at busId=4 , LUN=0 */
2  if((pSpd40=scsiPhysDevCreate
3      (pSysScsiCtrl,4,0,0,NONE,0,0,0))
4      == (SCSI_PHYS_DEV *) NULL)
5  {
6      SCSI_DEBUG_MSG("usrScsiConfig:\"
7          "scsiPhysDevCreate failed. \n");
8      return (ERROR);
9  }
10     /* configure the sequential device for this physical device */
11     if(((pSd0=scsiSeqDevcreate (pSpd40))
12         == (SEQ_DEV *)NULL)
13     {
14         SCSI_DEBUG_MSG("usrScsiConfig:\"
15             "scsiSeqDevCreate failed. \n");
16         return (ERROR);
17     }
18     /* initialize a tapeFs device */
19     ....
20     if(tapeFsDevInit ("/tape1/",pSd0,pTapeConfig)== NULL)
21         return (ERROR);
```

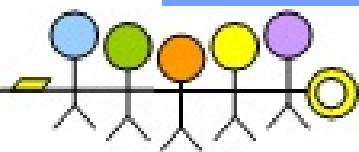


# Debugging Tools

-> **scsiShow**

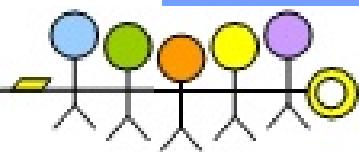
ID	LUN	VendorId	ProductId	Rev.	Type	Blocks	BlkSize	scsiPhysDe
--	--	-----	-----	-----	----	-----	-----	-----
5	0	ARCHIVE	VIPER	150 21247-005	1R	0	0	0X003CF654

- When first starting out , define **SCSI\_AUTO\_CONFIG** in **wind/target/config/target/config.h**.
  - Calls **scsiPhysDevCreate( )** for each bus Id and LUN.
  - Calls **scsiShow( )** when done.
- If you don't see all your peripherals attached :
  - Check cable connections.
  - Check SCSI bus terminations.
  - Make sure your peripheral supports the SCSI protocol being used.
  - In **sysScsiConfig( )**, set the global **scsiDebug = TRUE**.



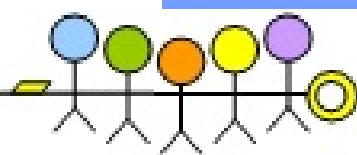
# Summary

- DOS and RAW file systems for :
  - RAM disk.
  - SCSI devices.
  - Custom block devices.
- SCSI-1 device support for :
  - Direct-access devices.
- SCSI-2 device support for :
  - Direct-access devices.
  - Sequential-access devices.
  - Tape file system.



# Chapter 12

# Network Basics



# Network Basics

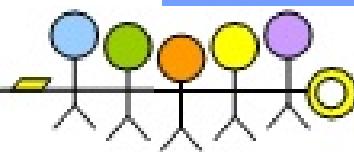
Introduction

Configuring The Network

Remote Login

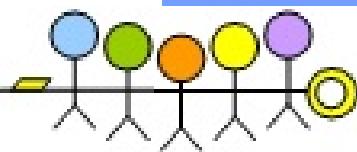
Remote Command Execution

Remote File Access

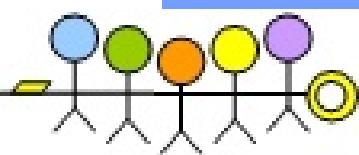
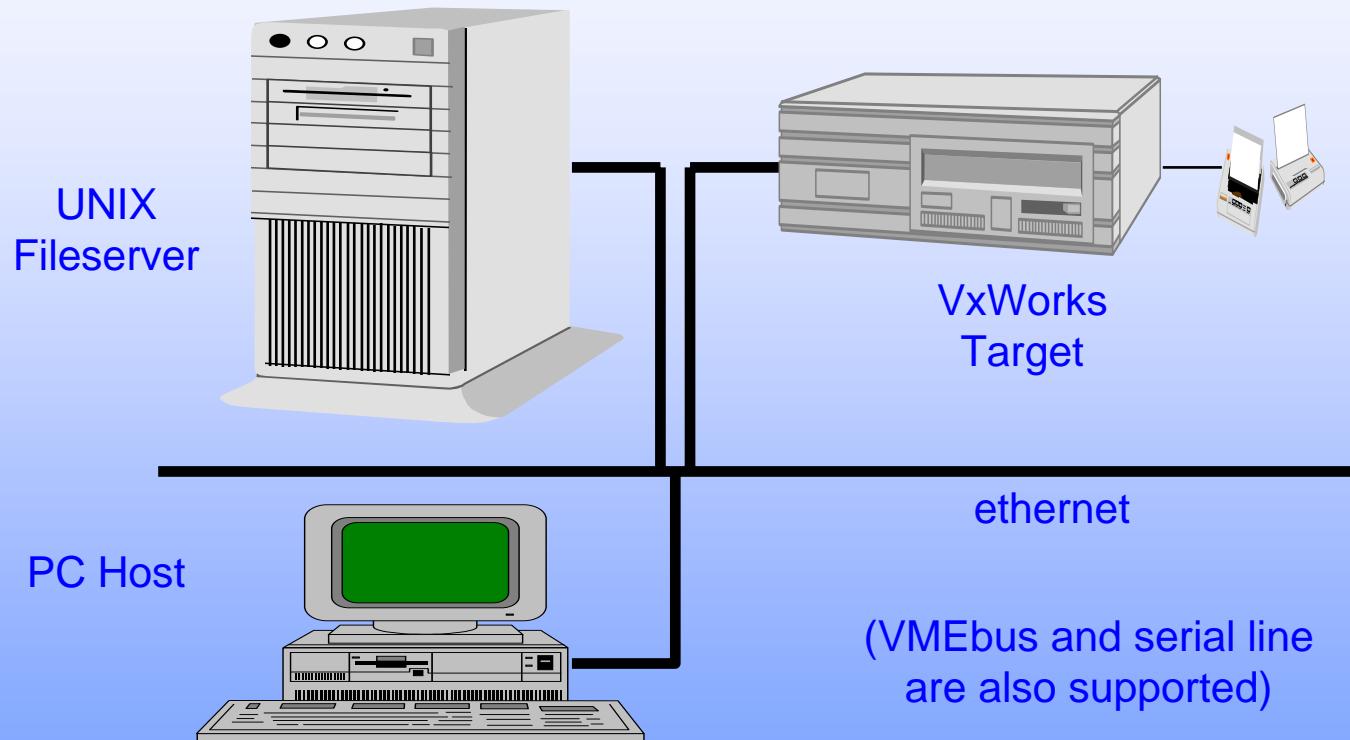


# Network Protocols

- Machines on a network must agree to exchange data in some standard way.
- Internet protocol suite (also called TCP / IP) provides system independent protocols.
- VxWorks implementation of TCP / IP protocol suite is based on 4.3BSD “Tahoe release”.



# Simple Network



# Terminology

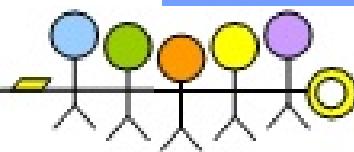
- Each node on the network is a “host”. Not to be confused with VxWorks host / target relationship.

- *Internet* is an ambiguous word :

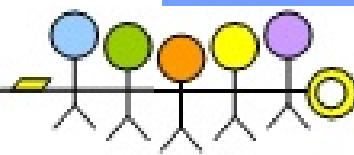
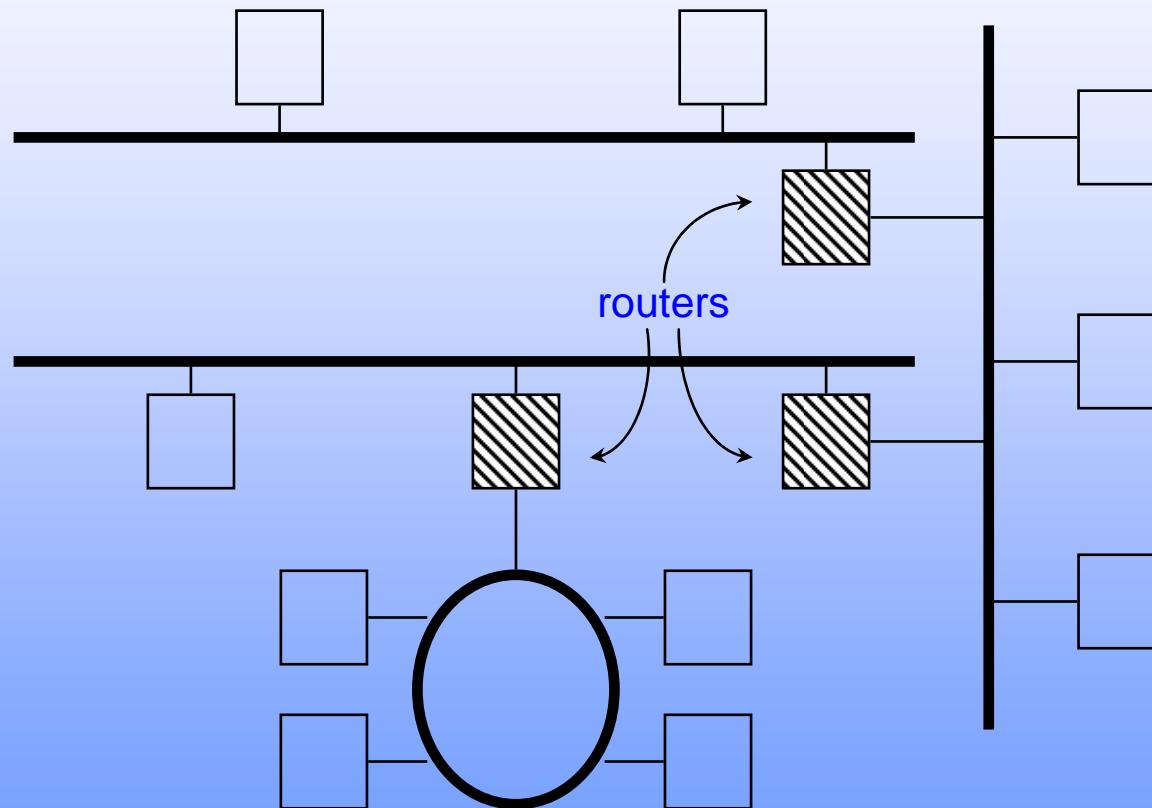
Internet Protocol Suite : A collection of protocols for system independent network communication.

An Internet : A collection of connected networks.

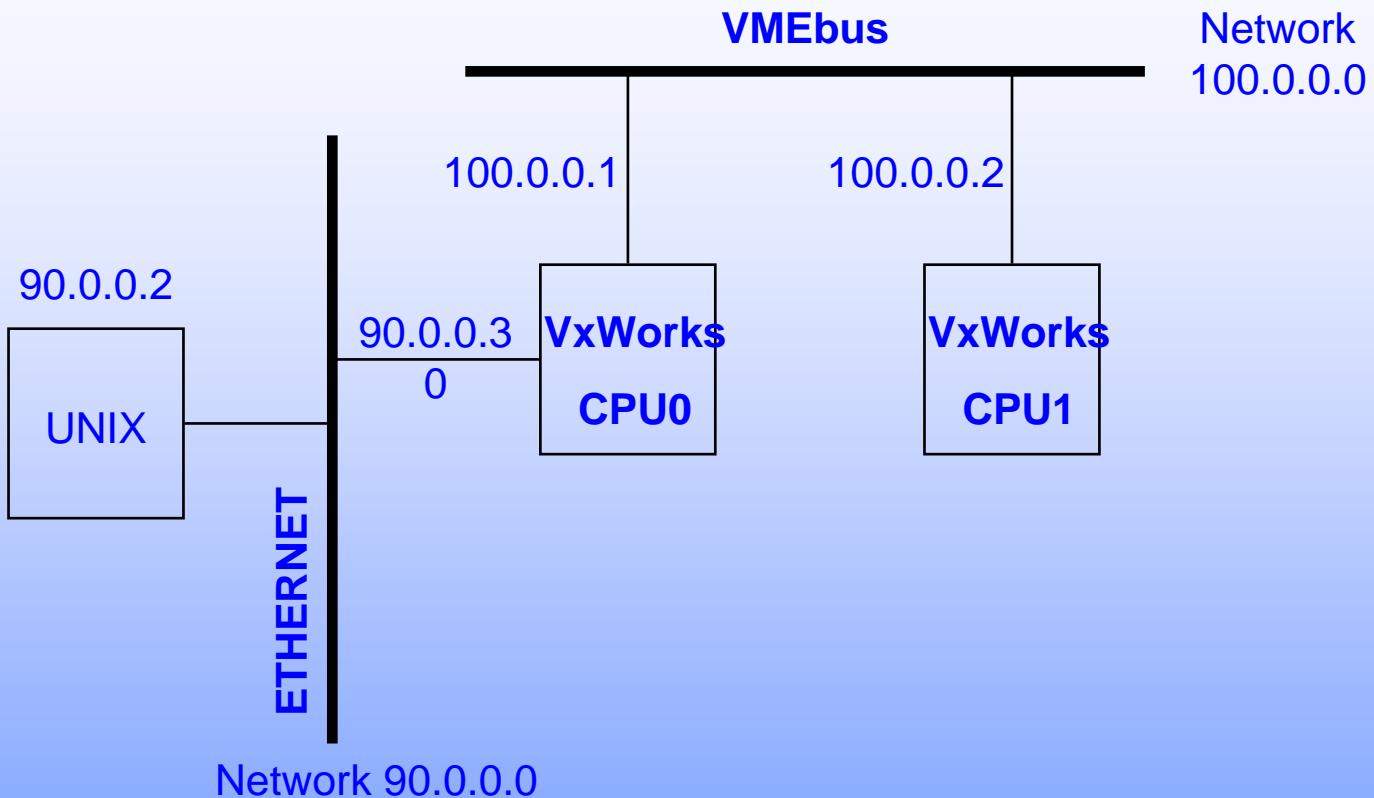
The Internet : The global internet.



# Complex Network (Internet)



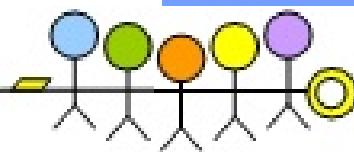
# Shared Memory Network Overview



- Details in appendix and *Programmers Guide*.

# Network Services

- VxWorks network services include :
  - Remote file access
  - Remote login
  - Remote command execution
- User can build other network services as needed. Writing network applications will be discussed in the next chapter.

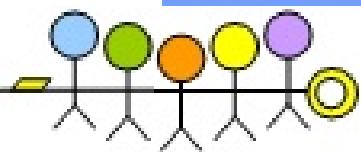


# Network Non-Determinism

TCP / IP is non-deterministic because

of :

- Ethernet collisions
- VMEbus spin locks
- Dropped packets
- Timeouts and retransmissions



# Networking

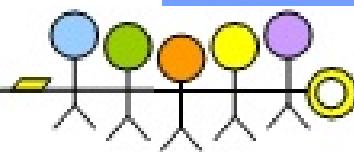
Introduction

Configuring The Network

Remote Login

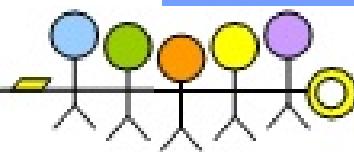
Remote Command Execution

Remote File Access



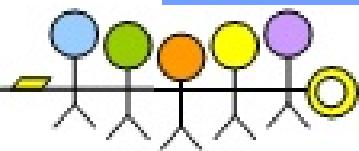
# Routing

- If client and server are on different networks, data passing between them must be routed.
- Internet routing algorithm :  
*if (destination on a directly attached network)  
    send data to destination  
else  
    use routing table to find correct router  
    and send data to router.*



# Internet Addresses

- Internet addresses written in “dot” notation  
90.0.0.70
- Internally stored as 4 byte integer  
0x5a000046
- This value encodes both a network number and a host number.
  - Analogous to a phone number having an area code and a local portion.
  - Two nodes are on the same network if they have the same network number.
  - Makes routing easier.



# Internet Address Classes

## Class

A

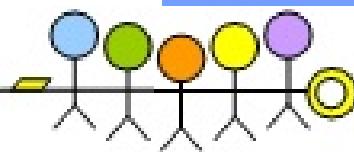
0	network 7 bits	host 24 bits
---	-------------------	-----------------

B

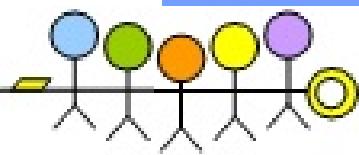
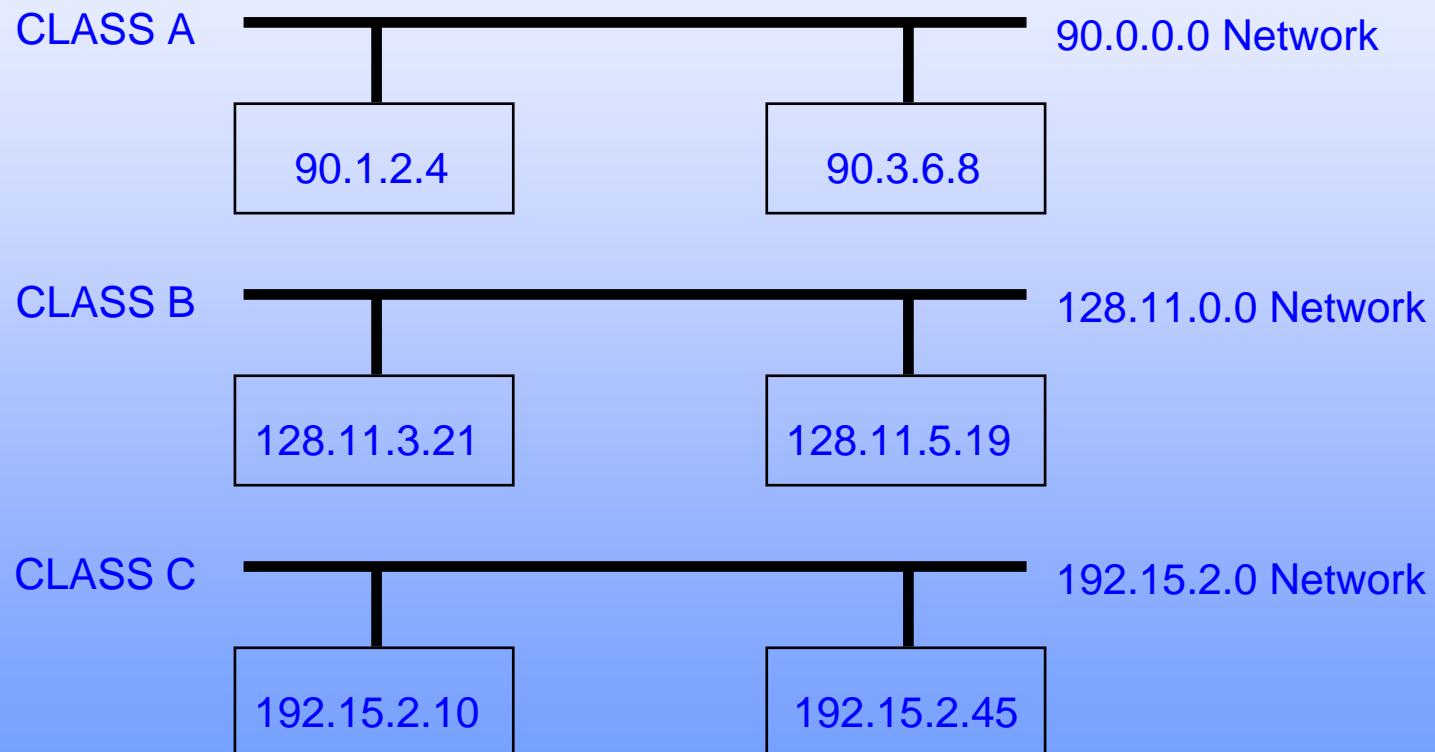
1	0	network 14 bits	host 16 bits
---	---	--------------------	-----------------

C

1	1	0	network 21 bits	host 8 bits
---	---	---	--------------------	----------------



# Internet Address Examples



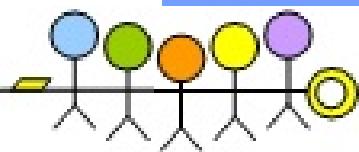
# Assigning Internet Addresses

## Network Prefixes

- Must be unique for each network on an internet.
- Assigned by administrator of that internet.
  - In an isolated environment, there is no restriction on network prefix.
  - If connected to the Internet, network prefix must be assigned by the InterNIC (Internet Network Information Center)

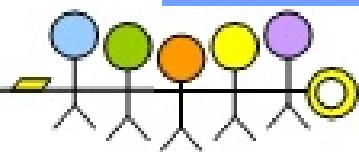
## Host Numbers

- Must be unique for each host on a network
- Assigned by administrator of that network.



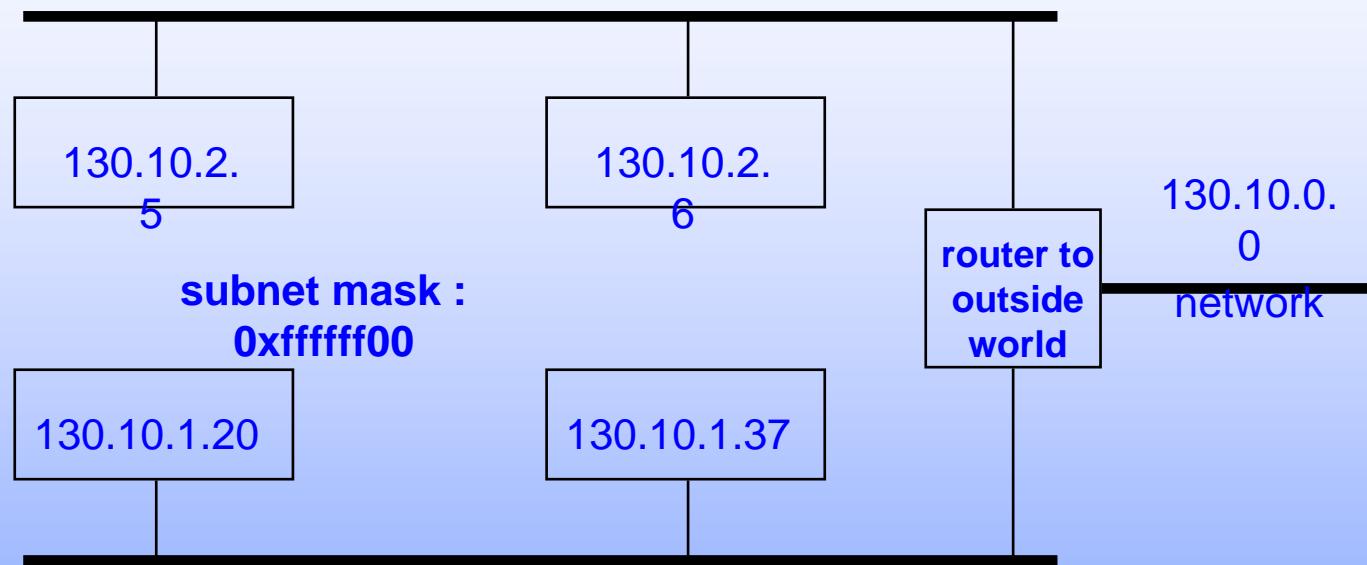
# Subnets

- At a large site, many physical networks are needed :
  - To improve network performance
  - To make administration easier.
- Adding a new network normally requires obtaining a new network prefix.
- A new network number may not be available.
- *Subnetting* allows a site to add new networks transparently to the rest of the internet.



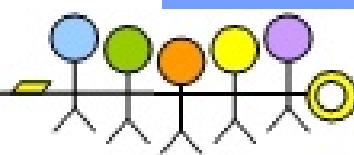
# Subnet Example

130.10.2.0 subnet



130.10.1.0 subnet

VxWorks target on 130.10.1.0/24 subnet must specify:  
inet on ethernet (e) : 130.10.1.20:ffffffff00



## inetLib

Routines for manipulating internet addresses :

*inet\_addr( )*

Converts dot notation internet address to an integer.

*inet\_lnaof( )  
address.*

Returns the host portion of an internet

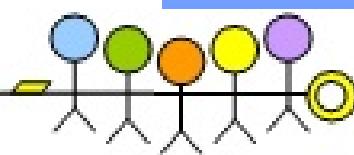
*inet\_netof( )*

Returns the network portion of an internet address.

*inet\_netof\_string( )  
string.*

Obtains network portion of address as a  
Knows about interface subnet masks.

Converts internet address to ASCII dot



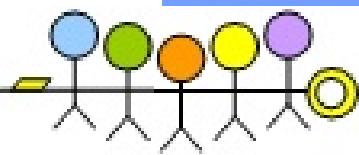
# Host Names

- To associate a name with an internet address :

**hostAdd (hostName,  
hostAddr)**

- To display host name table use :

**hostShow ()**



## Example

-->hostAdd “styx”, “147.11.1.80”

value = 0 = 0x0

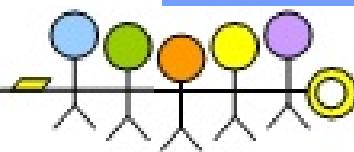
-->hostAdd “nelson”, “147.11.1.80”

value = 0 = 0x0

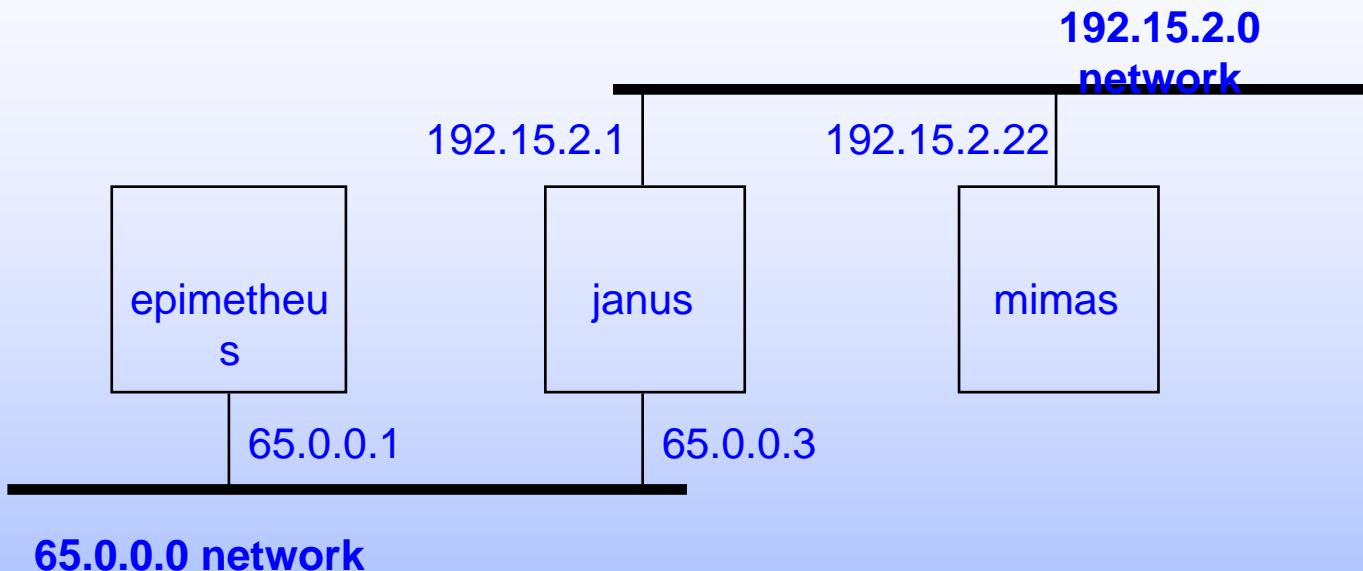
-->hostShow

hostname	inet	address	aliases
localhost		127.0.0.1	
ohio		147.11.1.81	
styx		147.11.1.80	nelson

value = 0 = 0x0



# Routing Tables

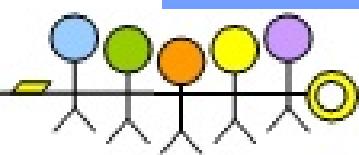


Host A Routing Table

Network	Gateway
192.15.2.0	65.0.0.3

Host C Routing Table

Network	Gateway
65.0.0.0	192.15.2.1

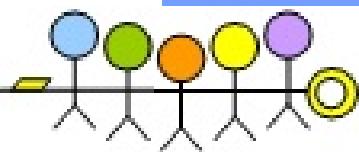


# Adding Routes in VxWorks

- To add a route use :  
**route (destination, gateway)**

Can use a host name or an internet address.

- Destination can be :
  - A specific host :  
**routeAdd ("65.0.0.4","192.15.2.7")**
  - A network :  
**routeAdd ("65.0.0.0","gateHost")**
  - Anywhere else ( default route ):  
**routeAdd ("0","192.15.2.1")**



# A Debugging Tool

```
-->routeAdd "150.39.0.0", "147.11.54.254"
```

```
value = 0 = 0x0
```

```
-->routeShow
```

```
ROUTE NET TABLE
```

destination	Interface	gateway	flags	Refcnt	Use
150.39.0.0	enp0	147.11.54.254	3	0	0
147.11.54.0	enp0	147.11.54.170	1	1	52

```
ROUTE HOST TABLE
```

destination	Interface	gateway	flags	Refcnt	Use
127.0.0.1		127.0.0.1	5	0	0

# Testing Routes

- To check if a host is reachable :

STATUS ping (host, nPackets, options)

-->**ping “columbia”, 1**

columbia is alive

value = 0 = 0x0

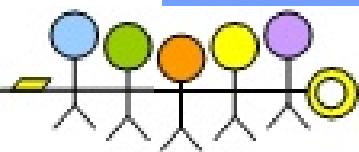
-->**ping “90.0.0.70”, 1**

no answer from 90.0.0.70

value = -1 = 0xffffffff = \_end + 0xffff91c4f

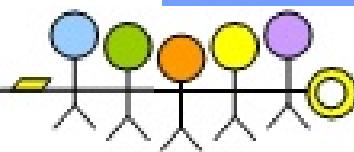
- If *ping( )* fails, check :

- Routing information.
- Hardware connections.



# Host Configuration

- Associate host names with internet addresses in host table.
- Set up routing tables so that packets will be routed correctly.
- Speak with your network administrator!
- For more information, see :
  - *TCP / IP Network Administration*, O'Reilly & Associates.  
**(UNIX Host)**
  - *Networking personal Computers with TCP / IP*, O'Reilly & Associates. **(PC Host)**
  - *Managing NFS and NIS*, O'Reilly & Associates.  
**(UNIX Host)**
  - Your network software's documentation.



# Network Services

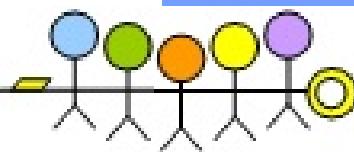
Introduction

Configuring The Network

Remote Login

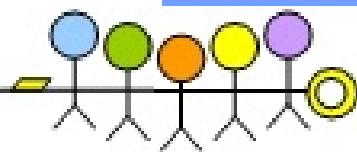
Remote Command Execution

Remote File Access



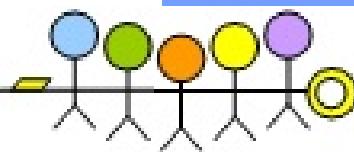
# Remote Login and VxWorks

- Two protocols :
  - rlogin (**UNIX Host**)
  - telnet (internet protocol)
- Gives access paths to VxWorks applications for sites without Tornado interface.
- Security can be installed. Restricts access to VxWorks by user name and password:
  - Define **INCLUDE\_SECURITY**
  - Modify **LOGIN\_USER\_NAME** and **LOGIN\_PASSWORD**
  - Add additional users with *loginUserAdd( )*



# Remote Login and Support Facilities

- Support routines must be linked into VxWorks.
  - **INCLUDE\_RLOGIN** rlogin server and client.
  - **INCLUDE\_TELNET** telnet server.
- Remote login also requires the target resident shell.
  - **INCLUDE\_SHELL**
- Additional tools to be used in conjunction with remote login facilities can also be linked into VxWorks :
  - Target resident symbol table.
  - Target resident show routines.
  - Target resident module loader / unloader.



# Network Basics

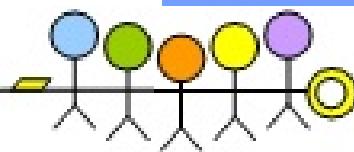
Introduction

Configuring The Network

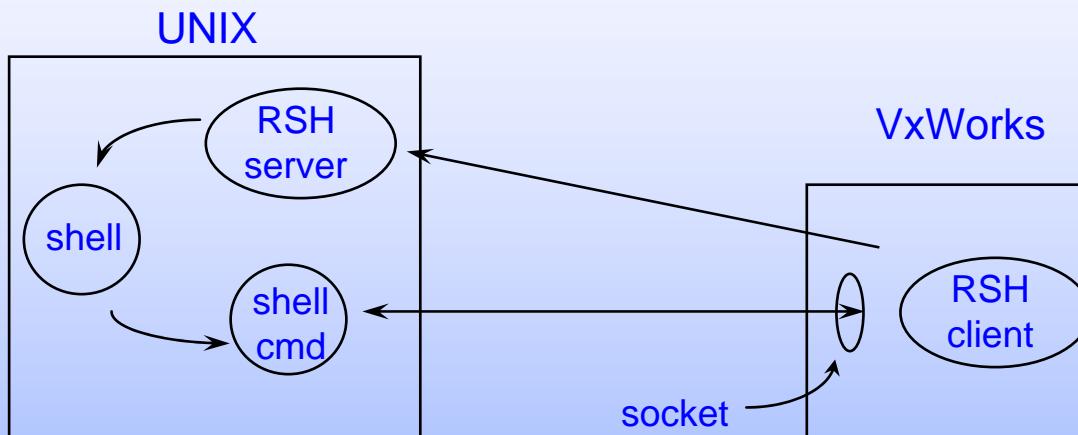
Remote Login

Remote Command Execution

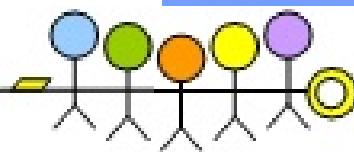
Remote File Access



# Executing Remote Commands



- VxWorks programs can invoke remote shell (RSH) commands on a UNIX host
- A file descriptor called a **socket** is created. Can **read( )** from this socket to get command output.

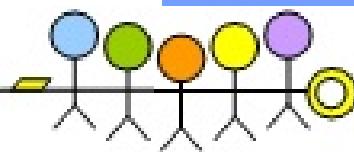


# UNIX : Remote Command Execution

int rcmd (host, remotePort, localUser, remoteUser, cmd, pFd2)

host	Host name or inet number.
remotePort typically	Remote port number to connect to, 514(shell).
localUser	Name of local user.
remoteUser	User name on remote host.
cmd	Shell command string to execute.
pFd2 returned	If non-zero, a socket for <b>STD_ERR</b> is through this pointer.

- Returns a socket file descriptor or **ERROR**.



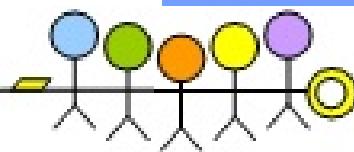
## UNIX : rcmd Example

```
-->unixDate = calloc (300,1)
unixDate = 0x23ff264 : value = 37744912 = 0x23ff110
```

```
-->aFd = rcmd("ohio",514,"debbie","debbie","date",0)
newsymbol "aFd" added to symbol table.
aFd = 0x23fefa0 : value = 4 = 0x4
```

```
-->read (aFd, unixDate, 300)
value = 29 = 0x1d
```

```
-->printf ("%s \n", unixDate)
Mon Nov 18 12:25:45 PST 1991
value = 30 = 0x1e
-->close (aFd)
```



# Network Basics

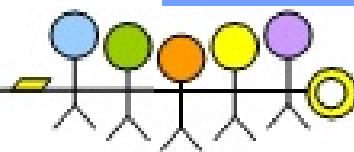
Introduction

Configuring The Network

Remote Login

Remote Command Execution

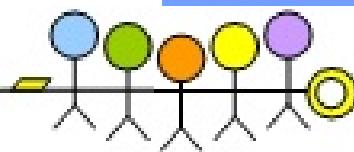
Remote File Access



## Remote File Access

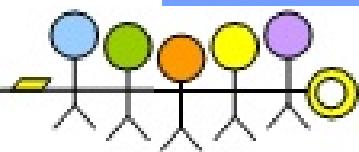
VxWorks comes with two drivers which allow access to files  
on  
remote machines.

- nfsDrv
- netDrv



# NFS

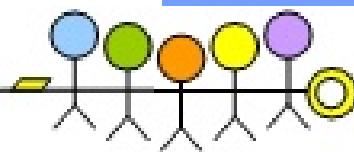
- Network File System (NFS) was developed by Sun Microsystems
- Allows efficient access to files. NFS transfers and buffers files in pieces (usually 8 Kbytes).
- Remote file systems are *mounted*; then accessed as if they were local file systems.
- VxWorks provides NFS client and server.
- NFS client excluded by default. To include, define **INCLUDE\_NFS** in **ConfigAll.h**



# NFS Overview

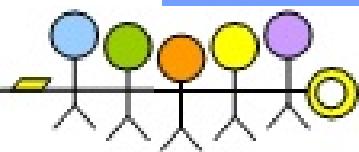
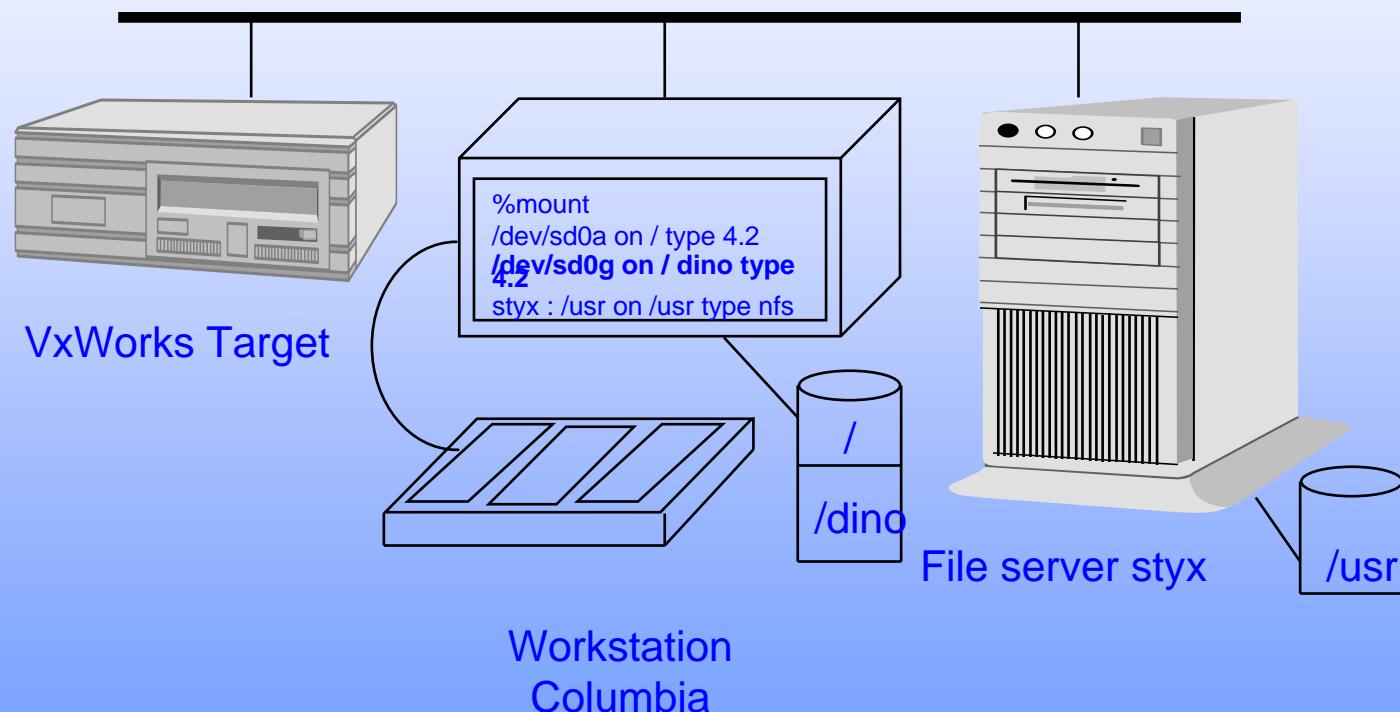
To access files on a remote machine using NFS :

1. Export server's file system :
  - Make a local file system available to remote host with appropriate access permissions.
2. Mount the file system on the client.
3. Set the client's authentication parameters.



# 1. Exporting the File System

- NFS servers only export local file system



## 2. Mounting NFS Filesystems

STATUS nfsMount (host, fileSystem, localName)

host Name of remote host.

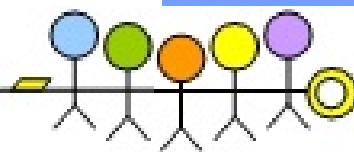
fileSystem  
(must be exported by remote host).

localName Local device name for file system. If  
NULL,

- Example :

-->nfsMount (“columbia”, “/dino”, “/dinoNfs”)

-->fd = open (“/dinoNfs/comics/myFile”, 2)



# Examining NFS Filesystems

To display all mounted NFS file systems :

-->**nfsDevShow**

device name

/usrNfs

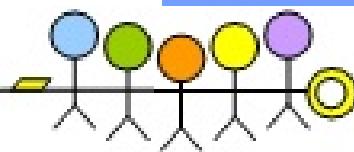
/dinoNfs

value = 0 = 0x0

file system

styx : /usr

columbia : / dino

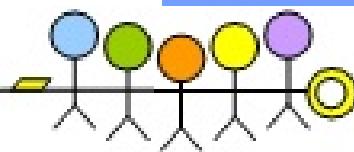


## UNIX : 3. NFS Authentication

- For UNIX NFS servers, once file system is mounted client can access files as if they were local after authentications are set.
- File access depends on :
  - User id.
  - Group id.
- Default id's specified in **configAll.h** :

```
#define NFS_USER_ID      2001  
#define NFS_GROUP_ID     100
```
- To find your user and group ids on UNIX host :

```
%id  
uid = 219(marc) gid=700(training)
```



# UNIX : NFS Authentication

`void nfsAuthUnixSet (hostName, uid, gid, ngids, aup_gids)`

hostName

Remote host name.

uid

User id on *hostName*.

gid

Group id on *hostName*.

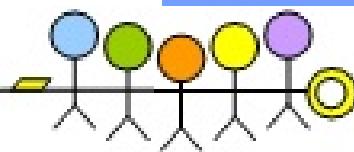
ngids

Number of gids listed in *aup\_gids*.

aup\_gids

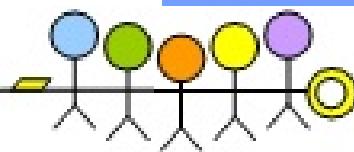
Array of additional group id's.

- Sets authentication programmatically.
- To see current authentication parameters, use :  
`void nfsAuthUnixShow( )`



## netDrv

- **netDrv** allows access to remote files :
  - Entire file read into target memory on ***open()***.
  - ***read()*** / ***write()*** act on copy of file in memory.
  - File not written to remote host until ***close()***.
- Can't access large files (constrained by available memory).
- ***ioctl (fd, FIOSYNC, 0)*** is not supported
- **dirLib** routines ***opendir / readdir*** do not work on **netDrv** directories.  
***stat / fstat*** only partially implemented.



# Creating network Devices

STATUS netDevCreate (devName, host, protocol)

devName Local name of device to create. By convention, ends with ':'.

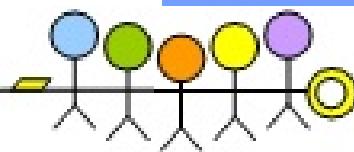
host Name of remote machine (from **hostAdd( )**).

previous  
protocol Protocol to transfer files to / from VxWorks (0=RSH or 1=FTP).

- Example :

```
-->netDevCreate ("ohio:", "ohio", 1)
```

```
-->fd = open ("ohio:/u/teamN", 2)
```

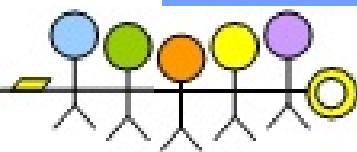


# Permissions

- FTP file access requires a name and a password :  
--> **iam “team8” , “team8Password”**
- RSH file access permission requires : (**UNIX host**)
  - A user name to be sent on the target.  
--> **iam “team8”**
  - Target must be trusted to verify user name without a password.

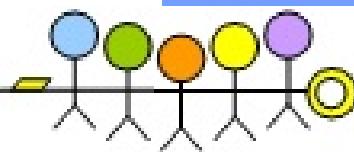
```
~team8/.rhosts  
vx8
```

- *rcmd* and *rlogin* to UNIX also use the RSH protocol. (**UNIX host**)



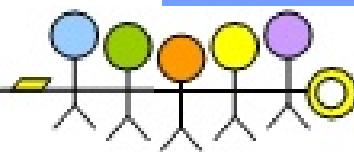
# UNIX : netDrv Protocol Comparison

- Advantages of FTP over RSH :
  - FTP is part of the TCP / IP protocol suite, so is available on all networked hosts.
  - FTP is faster on *open / close*; since there is no shell delay.
  - FTP server supplied with VxWorks.
  
- Advantages of RSH over FTP :
  - FTP clear text password goes over the network on each *open( )* and *close( )*.



## netDrv vs. nfsDrv

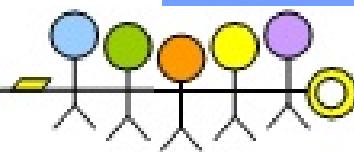
- Advantages of **nfsDrv** over **netDrv** :
  - Reads and writes only needed parts of the file.
  - Can access arbitrarily large files.
  - **open( )** and **close( )** much faster.
  - Can use **dirLib** routines.
  - Can flush file changes with **FIOSYNC**.
  
- Advantages of **netDrv** over **nfsDrv** :
  - All networked hosts have an FTP server.
  - Easier to configure.
  - **read( )** / **write( )** slightly faster.



# Accessing Files on VxWorks

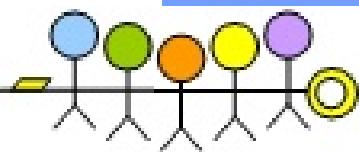
Requires a file server :

- VxWorks supplies FTP and NFS servers.
- VxWorks does not supply RSH server.



# NFS Server Overview

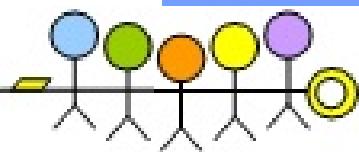
- Allows remote hosts to access a target's file system using NFS protocol.
- Only DOS file system is supported.
- Define INCLUDE\_NFS\_SERVER in **configAll.h**
- NFS server configuration :
  1. Initialize DOS file system with NFS server support.
  2. Inform NFS server that the file system is exportable.
  3. Mount file system on remote host.



# 1. Creating A Mountable File System

- Enable the **DOS\_OPT\_EXPORT** configuration option, before initializing a DOS file system to support remote access.
- Example :

```
/* Configure Block Device */  
pBlkDev = xxDevCreate(...);  
  
/* Configure File System with NFS Support */  
dosFsDevInitOptionsSet (DOS_OPT_EXPOT);  
dosFsDevInit ("fsName, pBlkDev, NULL);
```



## 2. Exporting A File System

To allow remote systems access to local file system :

nfsExport (name, fsId, rdOnly, notUsed)

name                          File system name.

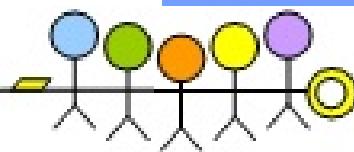
fsId                          NFS export Id number - Use 0 for  
NFS to                      create an ID.

rdOnly                        TRUE = mountable read only.  
                              FALSE = mountable read write.

notUsed                      Set to zero - reserved for future use.

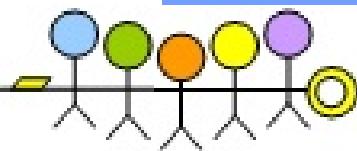
- Example :

```
nfsExport ("/fsName", 0, FALSE, 0);
```



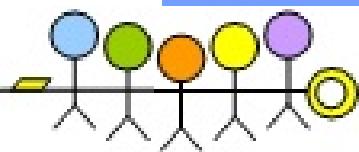
### 3. Accessing File System From A Remote Host

- To mount the VxWorks file system on a remote host, use :  
**mount -F nfs node:fileSystem directory**
- Typically, superuser (root) privilege is required on UNIX systems to mount a file system. (**UNIX** host)
- Example :  
**mount -F nfs vxtarget: /fsName /mountPoint**
- Once mounted, files may be accessed as if they were on a local file system.



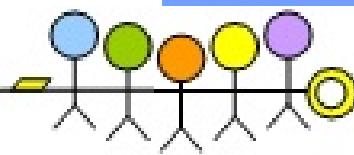
## Caveats

- By default, no authentication is performed
- Only DOS file system supported.
- Standard DOS limitations apply:
  - File naming restrictions.
  - No symbolic links.
  - No directory renaming.

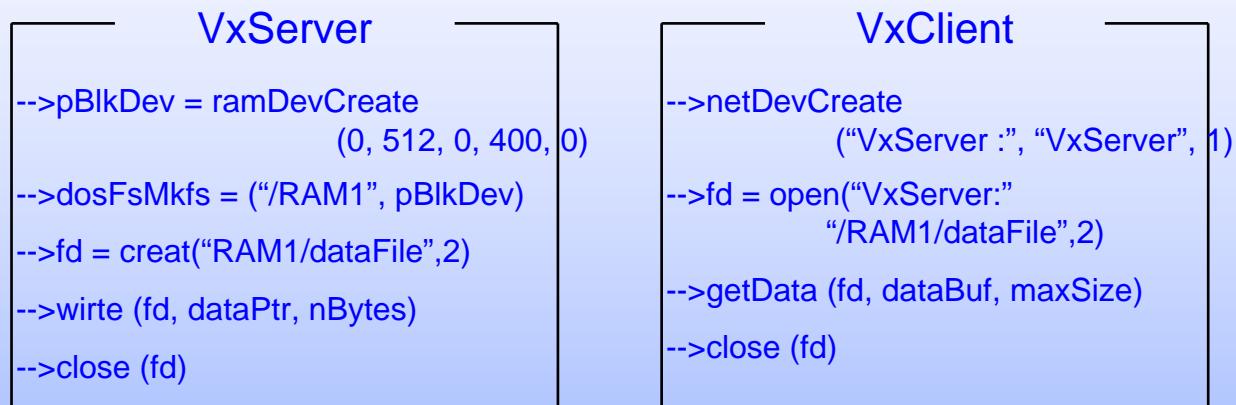


# VxWorks FTP Server

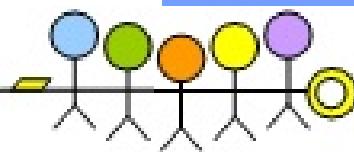
- Define the macro **INCLUDE\_FTP\_SERVER** in **configAll.h** or **config.h**.
- When target is rebooted, should see *tFtpdTask* (server daemon) running.



## FTP Example - VxWorks to VxWorks



- Can also use FTP to boot one VxWorks target from another.



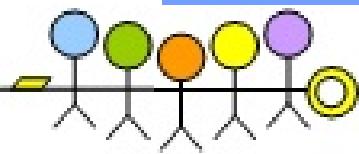
# FTP Example - VxWorks to UNIX

VxServer

```
-->pBlkDev = ramDevCreate  
          (0, 512, 0, 400, 0)  
-->dosFsMkfs = ("/RAM1", pBlkDev)  
-->fd = creat("RAM1/aFile",2)  
-->write (fd, dataPtr, nBytes)  
-->close (fd)
```

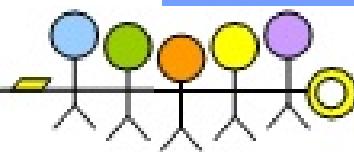
UNIX Client

```
% FTP VxServer  
connected to VxServer.  
Name (VxServer : debbie) :  
Password :  
ftp> cd /RAM1  
ftp> get dataFile  
local : dataFile remote : dataFile  
137 bytes received in 0.02 seconds  
(6.7Kbytes /s)  
ftp> quit  
% ls  
dataFile
```



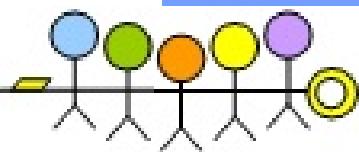
## NFS vs. FTP Server

- NFS advantages :
  - Efficient file access.
  - Configurable number of servers pre-spawned.
  
- FTP advantages :
  - Can access file systems other than DOS.
  - All network hosts have FTP client.



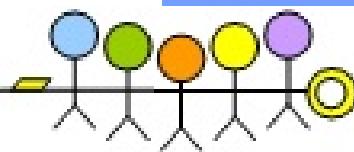
# Summary

- Internet Protocol for node-to-node routing.
- Configuring the network :
  - VxWorks : ***routeAdd( ), hostAdd( )***.
  - Configure host's host routing tables.
- *rlogin* (**UNIX** host) and *telnet* support.
- *rcmd ( )* (**UNIX** host)
- **nfsDrv** mounts remote file systems with ***nfsMount( )***
  - Reads and writes portion of remote file.
  - File system must be exported by remote host.
  - File access requires uid and gid.



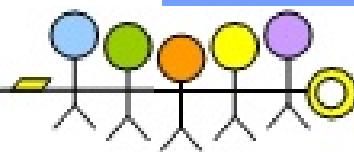
# Summary

- **netDrv** devices created with *netDevCreate()*.
  - Entire file is read into VxWorks on *open()*, and not updated until *close()*.
  - Uses either RSH (**UNIX** host) or FTP for file transfer.
  - RSH requires name and entry in **.rhosts**. (**UNIX** Host)
  - FTP requires name and password.
  
- FTP and NFS servers allows local VxWorks files to be accessed by a remote machine.



# Chapter 13

# Network Programming



# Network Programming

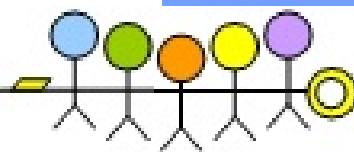
Introduction

Sockets

UDP Sockets Programming

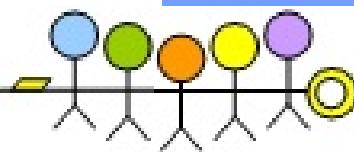
TCP Sockets Programming

RPC

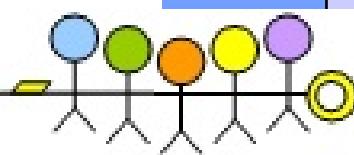
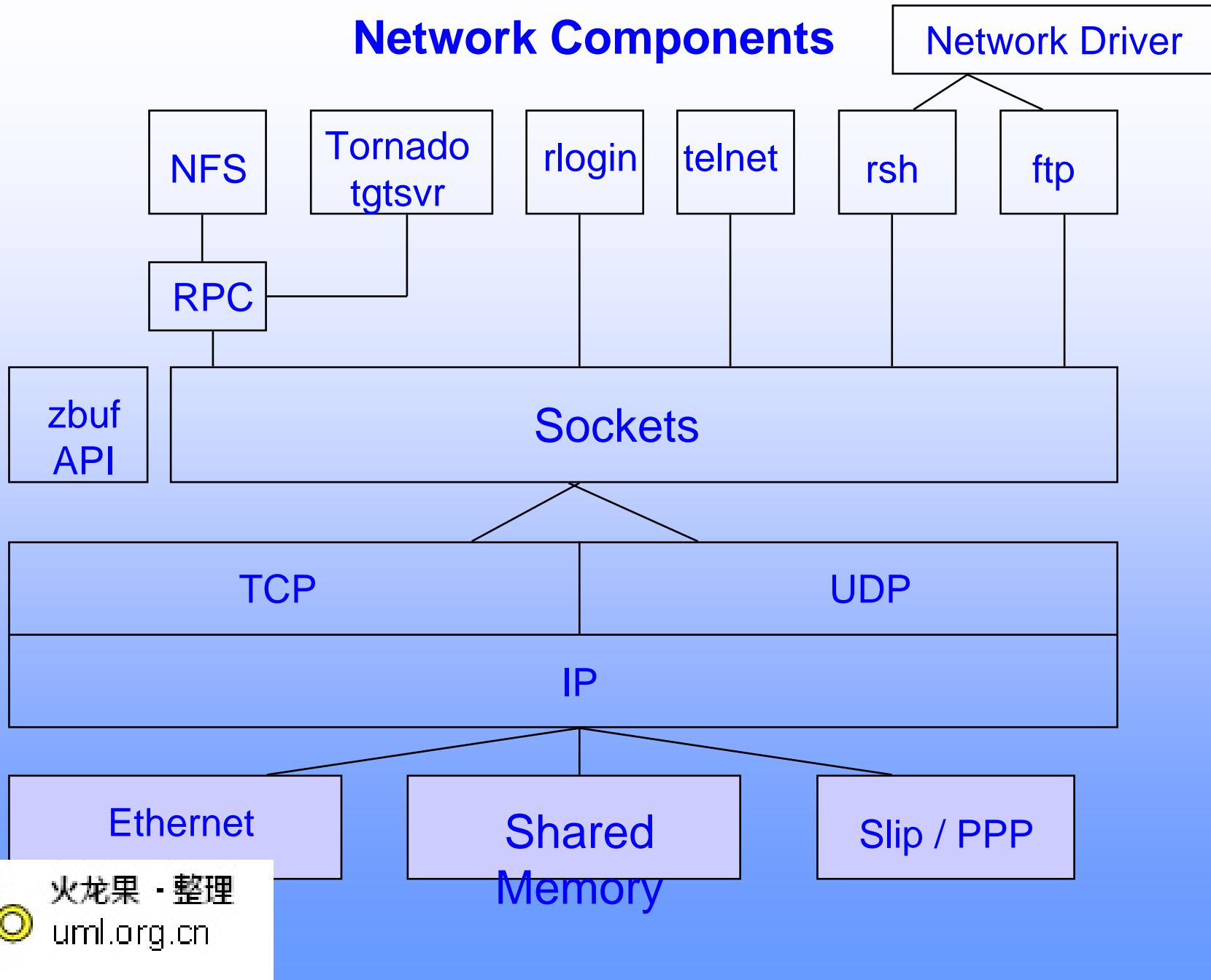


# VxWorks Networking

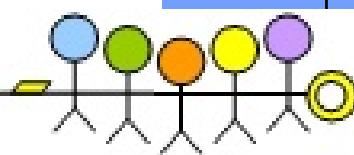
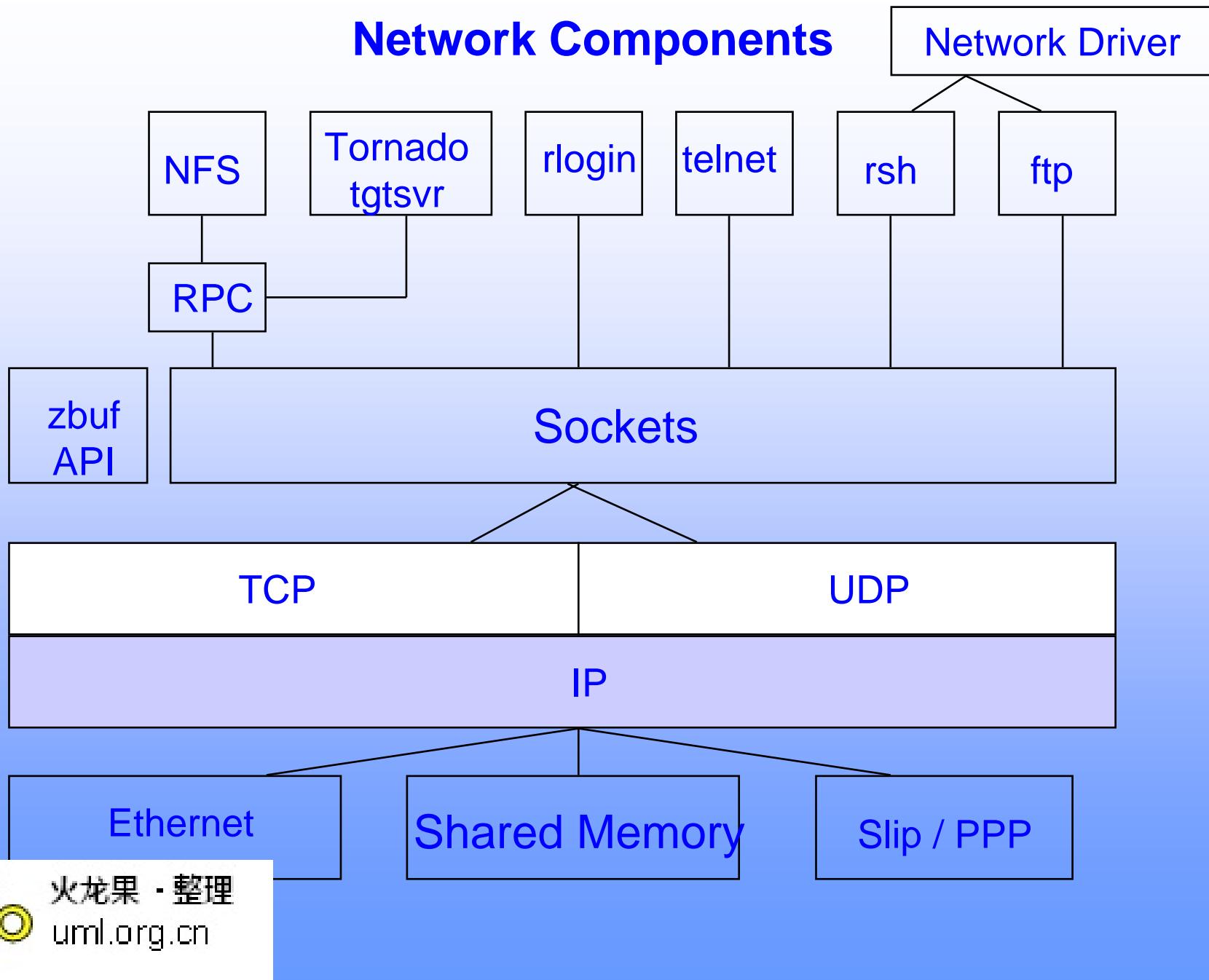
- Network programming allows users to :
  - Build services
  - Create distributed applications
- VxWorks network programming tools :
  - Berkeley sockets
  - zbuf socket API
  - Sun RPC ( Remote Procedure Call )



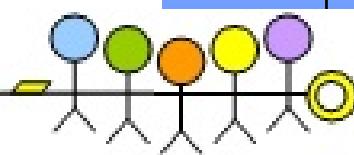
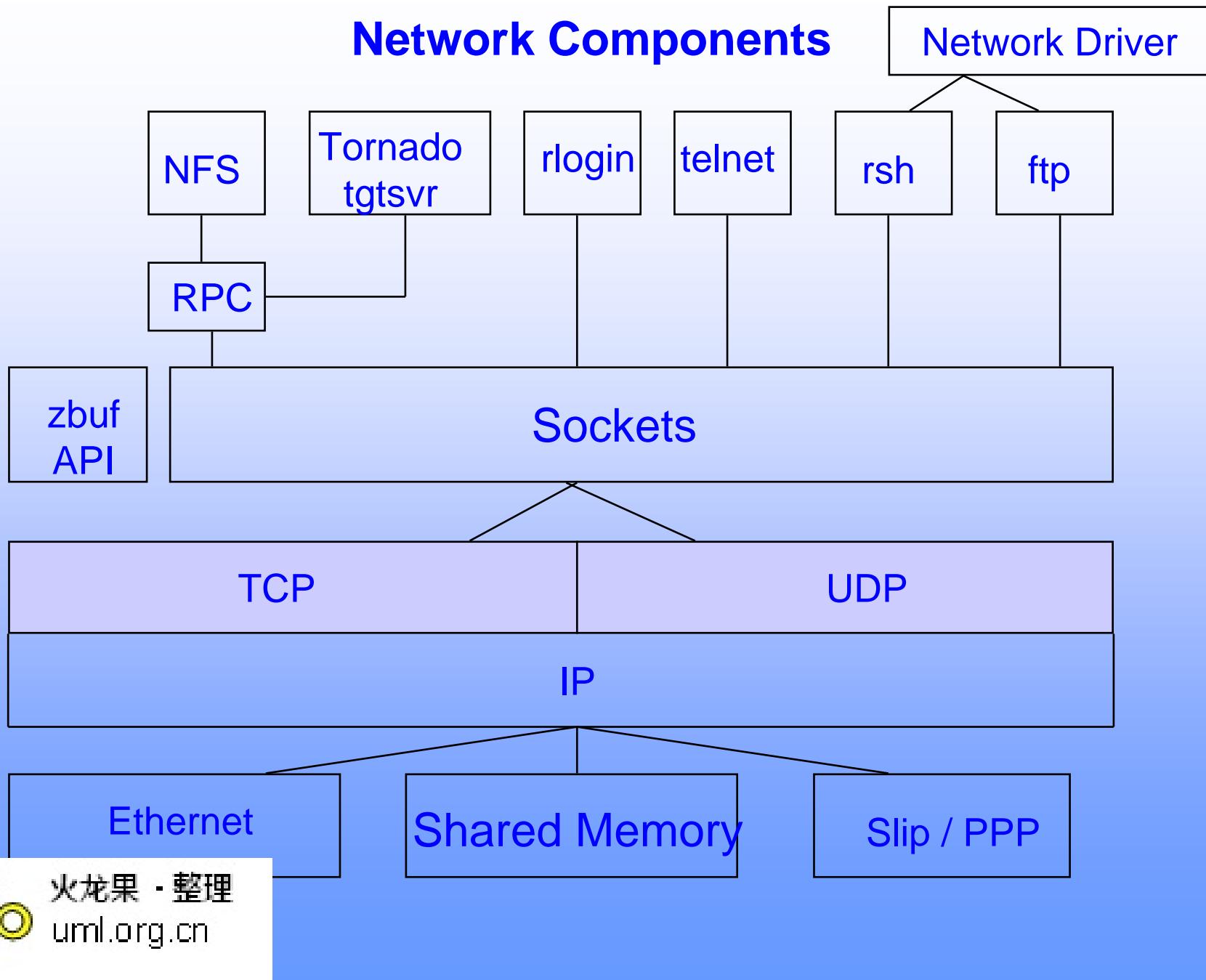
# Network Components



# Network Components

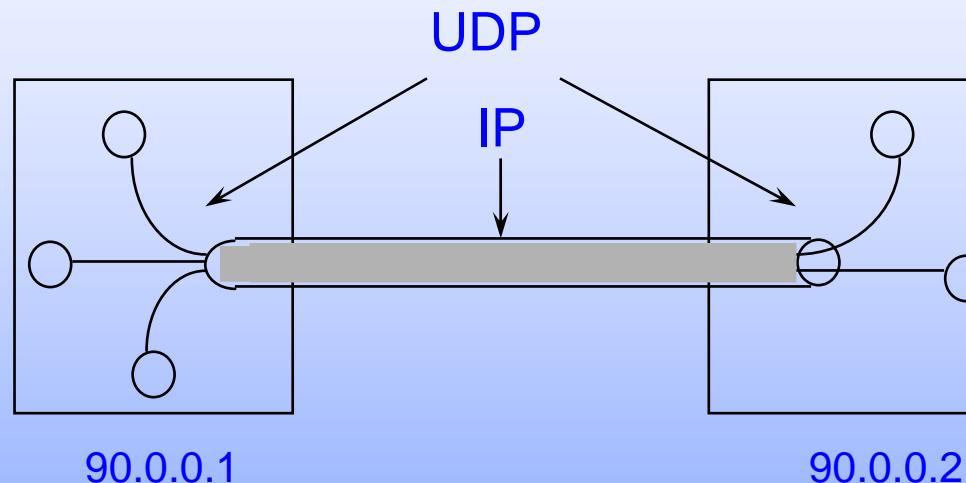


# Network Components

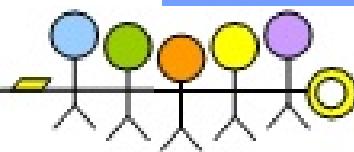


# Ports

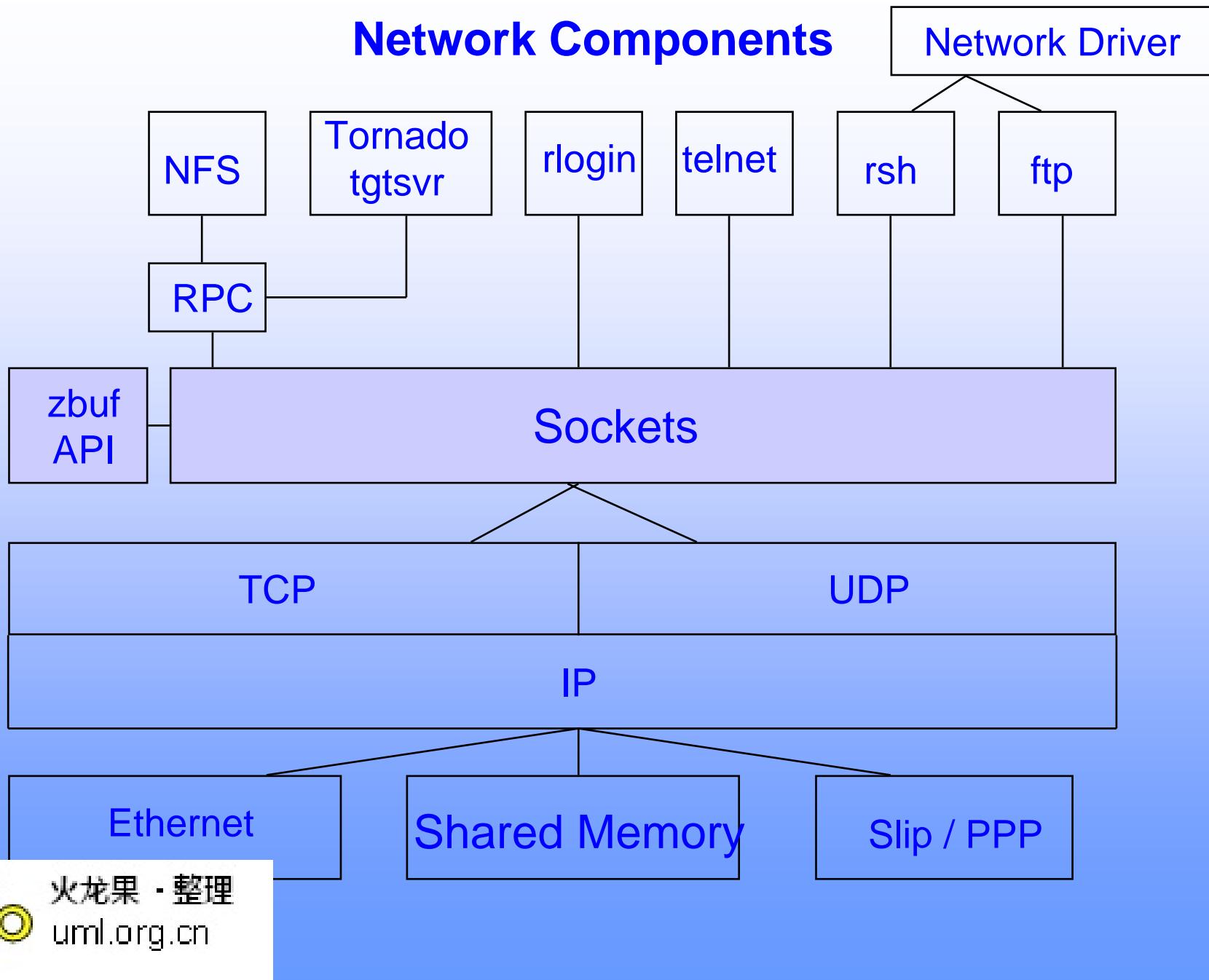
- Abstract destination point within a node.



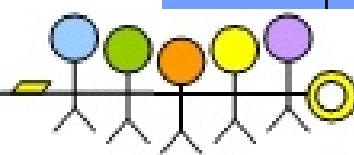
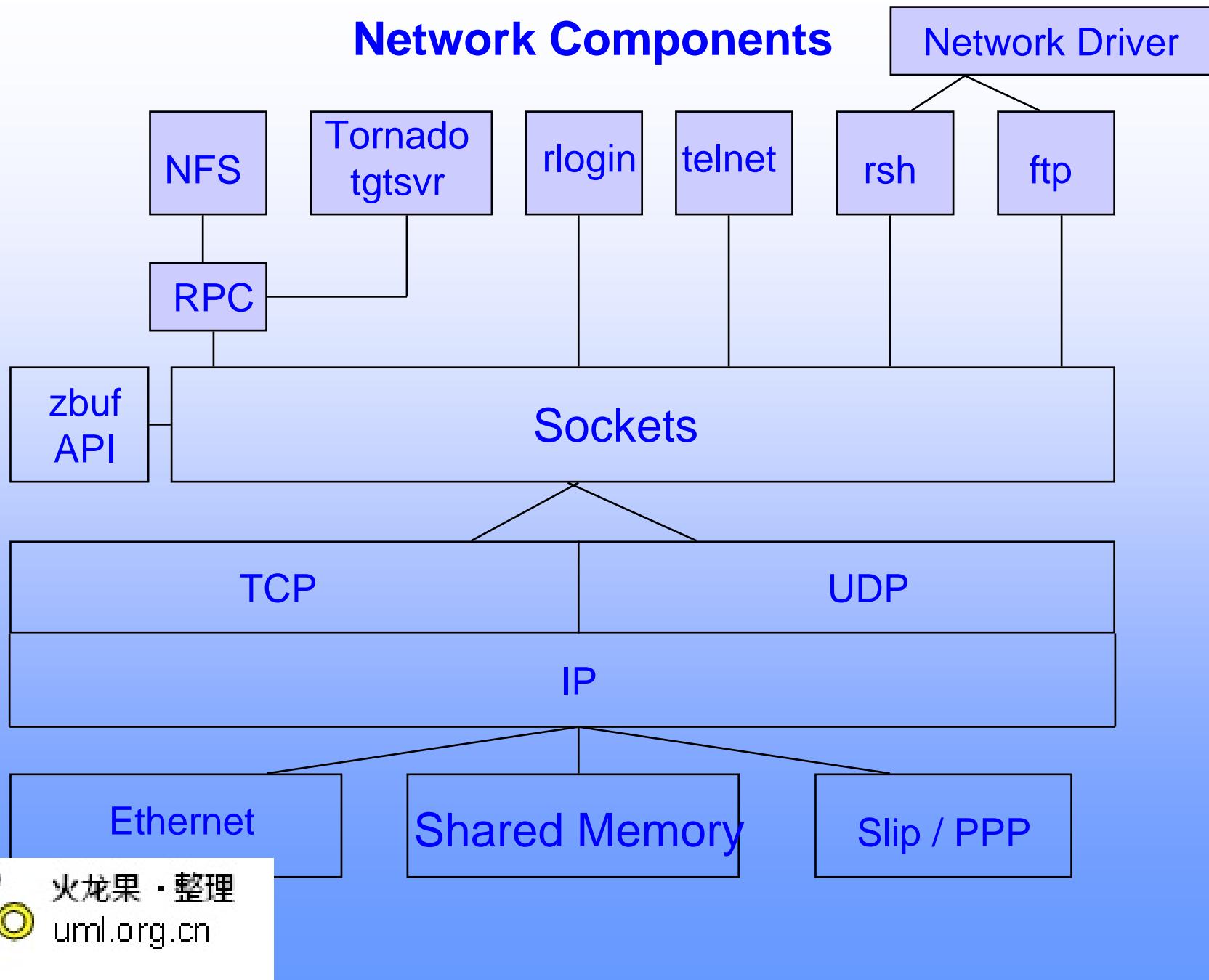
- TCP/ UDP intertask communication.
  - Data is sent by writing to a remote port.
  - Data is received by reading from a local port.



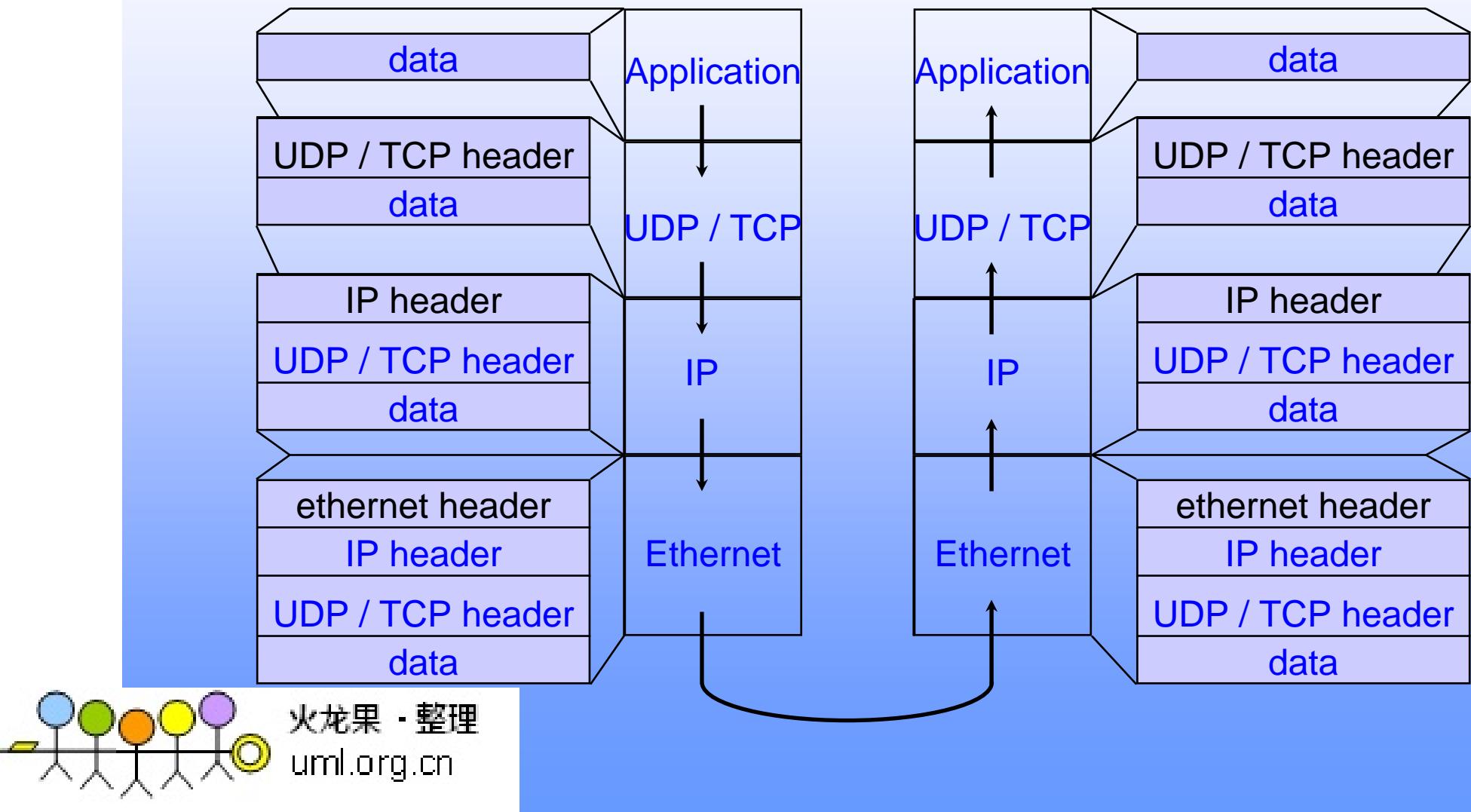
# Network Components



# Network Components



# Packet Encapsulation



# Network Programming

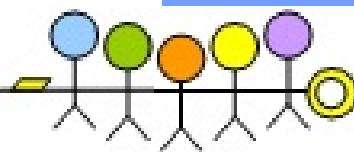
Introduction

Sockets

UDP Socket Programming

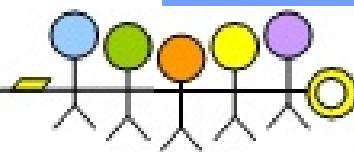
TCP Socket Programming

RPC



# Socket Overview

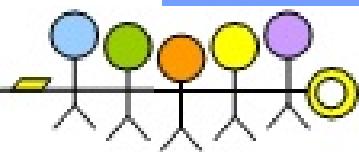
- Programmatic interface to internet protocols.
- Protocol specified when socket created ( e.g., UDP or TCP ).
- Server binds its socket to a well known port.
- Client's port number dynamically assigned by the system.



# Ports

- Socket address consists of :
  - An internet address ( node socket is on ).
  - A port number ( destination within that node )
- Port identified by a short integer :

0-1023 services	Reserved for system (e.g.,rlogin,telnet, etc).
1024-5000	Dynamically allocated
>5000	User defined
- Unique to each machine and protocol



# Socket Address

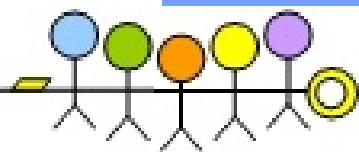
- Generic socket address :

```
struct sockaddr
{
    u_short      sa_family; /* address family */
    char        sa_data[14]; /*protocol specific address data */
};
```

- Generic address structure used by Internet Protocol :

```
struct sockaddr_in
{
    short       sin_family; /* AF_INET */
    u_short     sin_port;   /* port number */
    struct in_addr sin_addr; /*internet address */
    char        sin_zero[8]; /* padding, must be zeroed out */
};
```

VxWorks supports only internet sockets.



# Network Byte Ordering

- Fields in the *struct sockaddr\_in* must be put in network byte order ( big - endian)
- Macros to convert long / short integers between the host and the network byte ordering :

*htonl( )*

host to network long

*htons( )*

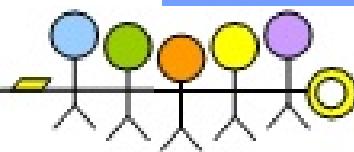
host to network short

*ntohl( )*

network to host long

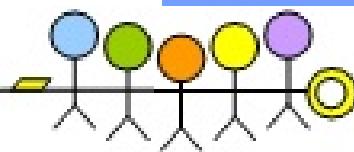
*ntohs( )*

network to host short



## Caveat - User Data

- To send data in a system independent way :
  1. Sender converts data from its system-dependent format to some standard format.
  2. Receiver converts data from the standard format to its system-dependent format.
- The standard format used must handle :
  - Any standard data types used ( e.g., int, short, float, etc.).
  - Data structure alignment.
- One such facility, XDR, will be discussed in the RPC section of this chapter.



# Creating a Socket

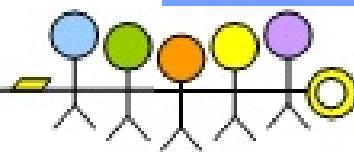
`int socket ( domain , type , protocol)`

domain      Must be **PF\_INET**

type      Typically **SOCK\_DGRAM** (UDP) or **SOCK\_STREAM** (TCP)

protocol      Socket protocol ( typically 0 ).

- Opens a socket ( analogous to **open()** for files)
- Returns a socket file descriptor or **ERROR**



# Binding a Socket

- To bind a socket to a well known address :

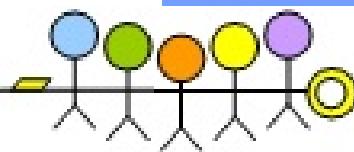
## STATUS bind ( **sockFd** , **pAdrs** , **adsLen** )

**sockFd**      Socket descriptor returned from **socket( )**

**pAdrs**      Pointer to a **struct sockaddr** to which to bind this socket

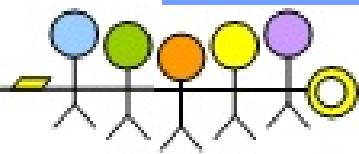
**adsLen**      **sizeof ( struct sockaddr ).**

- Typically only called by server.



## Example Server Stub

```
1 #define PORT_NUM ( USHORT ) 5001
2 struct sockaddr_in myAddr;
3 int mySocket ;
4 ...
5 mySocket = socket ( PF_INET, SOCK_DGRAM,0 );
6 if ( mySocket == ERROR)
7     return (ERROR) ;
8
9 bzero (&myAddr, sizeof (struct sockaddr_in));
10 myAddr.sin_family      = AF_INET;
11 myAddr.sin_port        = htons (PORT_NUM) ;
12 myAddr.sin_addr.s_addr = INADDR_ANY ;
13
14 if ( bind ( mySocket, (struct sockaddr *) &myAddr,
15             sizeof (myAddr)) == ERROR )
16 {
17     close (mySocket) ;
18     return (ERROR);
19 }
```



# Network Programming

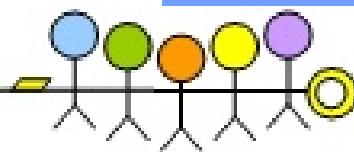
Introduction

Sockets

UDP Socket Programming

TCP Socket Programming

RPC



# UDP Socket Overview



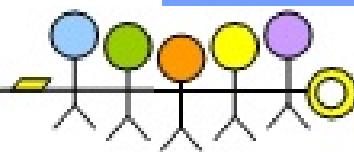
```
fds = socket ( PF_INET,  
              SOCK_DGRAM, 0 )  
bind ( fds , (struct sockaddr *)&saddr,  
           sizeof(saddr))  
loop  
    recvfrom(fds , ...) /* wait for request */
```

```
.....  
sendto (...)      /* perform service */  
                /* Send reply     */
```

```
fds = socket ( PF_INET,  
              SOCK_DGRAM, 0 )
```

```
sendto (...) /* Send request to  
server */
```

```
select(...) /* wait for reply with  
timeout */
```



## Sending Data on UDP Sockets

```
int sendto ( sockFd , pBuf , bufLen , flags ,  
            pDestAdrs , destLen)
```

sockFd      Socket to send data from.

pBuf      Address of data to send.

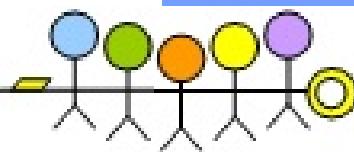
bufLen      Length of data in bytes.

flags      Special actions to be taken.

pDestAdrs      Pointer to *struct sockaddr* containing  
                  destination address.

destLen      *sizeof (struct sockaddr)*.

- Returns the number of bytes sent or **ERROR**.



# Receiving Data on UDP Sockets

```
int recvfrom (sockFd, pBuf, buflen, flags, pFromAdrs,  
              pFromLen)
```

sockFd      Socket to received data from.

pBuf      Buffor to hold incoming data.

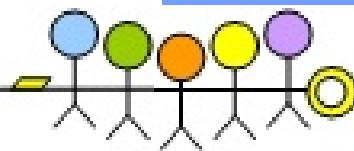
buflen      Maximum number of bytes to read.

flags      Flags for special handling of data.

pFromAdrs      Pointer to **struct sockaddr**. Routine supplies internet address of sender.

pFromLen      Pointer to integer. Must be initialized to sizeof (struct sockaddr).

- Blocks until data available to receive.
- Returns number of bytes received or **ERROR**.



# Network Programming

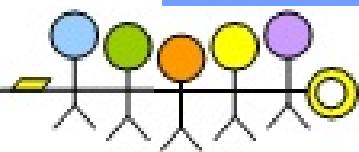
Introduction

Sockets

UDP Socket Programming

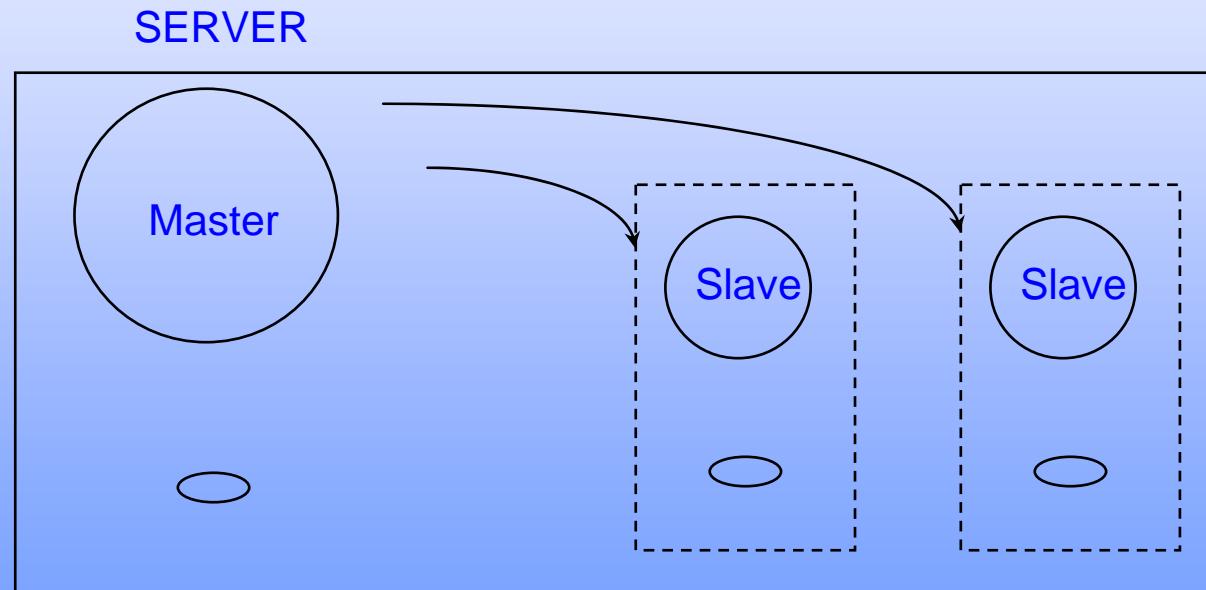
TCP Socket Programming

RPC



# TCP Socket Overview

- TCP is connection based (like making a phone call)
- *Concurrent servers* are often implemented.

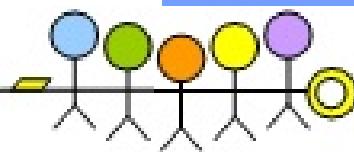


Clients send connection

to Master's socket

New socket created dynamically for

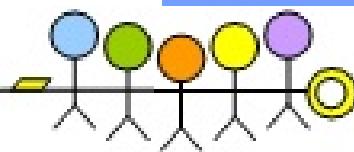
each connection, serviced by work task.



# TCP Server Overview

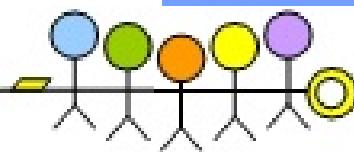
```
1  /* master server */
2  masterFd = socket (PF_INET, SOCK_STREAM, 0)
3  /* fill in server's sockaddr struct */
4  bind (...) /* bind to well-known port */
5  listen (...) /* configure request queue */
6  FOREVER
7  {
8      clientFd = accept (masterFd, ...)
9      taskSpawn (... , slaveSrv, clientFd, ...)
10     }

1  /* slave server */
2  slaveSrv (clientFd, ...)
3  {
4      read (clientFd, ...) /* read request */
5      serviceClient ()
6      write (clientFd, ...) /* send reply */
7      close (clientFd)
8 }
```



# TCP Client Overview

```
1  /* TCP Client */
2  fd = socket (PF_INET, SOCK_STREAM, 0)
3
4  /* fill in sockaddr with server's address */
5
6  connect (fd, ...) /* request service */
7
8  write (fd, ...)    /* send request */
9  read (fd, ...)     /* read reply */
10
11 close (fd)        /* terminate connection */
```



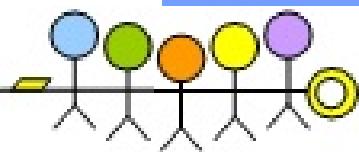
# Server Initialization

- Before accepting connections, server must :
  - Create a socket (**socket( )**)
  - Bind the socket to a well known address (**bind( )**)
  - Establish a connection request queue :

## STATUS listen (sockFd , queueLen)

SockFd      Socket descriptor returned from **socket( )**.

queueLen      Nominal length of connection request queue.



# Accepting Connections

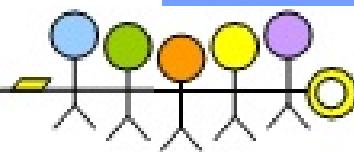
`int accept (sockFd , pAdrs , pAdrsLen)`

`sockFd` Servers socket (returned from `socket( )`).

`pAdrs` Pointer to a `struct sockaddr` through which the client's address is returned.

`pAdrsLen` Pointer to length of address.

- Blocks until connection request occurs.
- Returns new socket file descriptor (connected to the client) or `ERROR`.
- Original socket, `sockFd`, is unconnected and ready to accept other connection requests.



# Requesting Connections

- To connect to the server, the client calls:

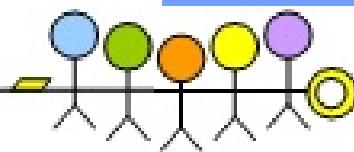
STATUS connect (sockFd, pAdrs, adsLen)

sockFd Client's socket descriptor.

pAdrs Pointer to server's socket address.

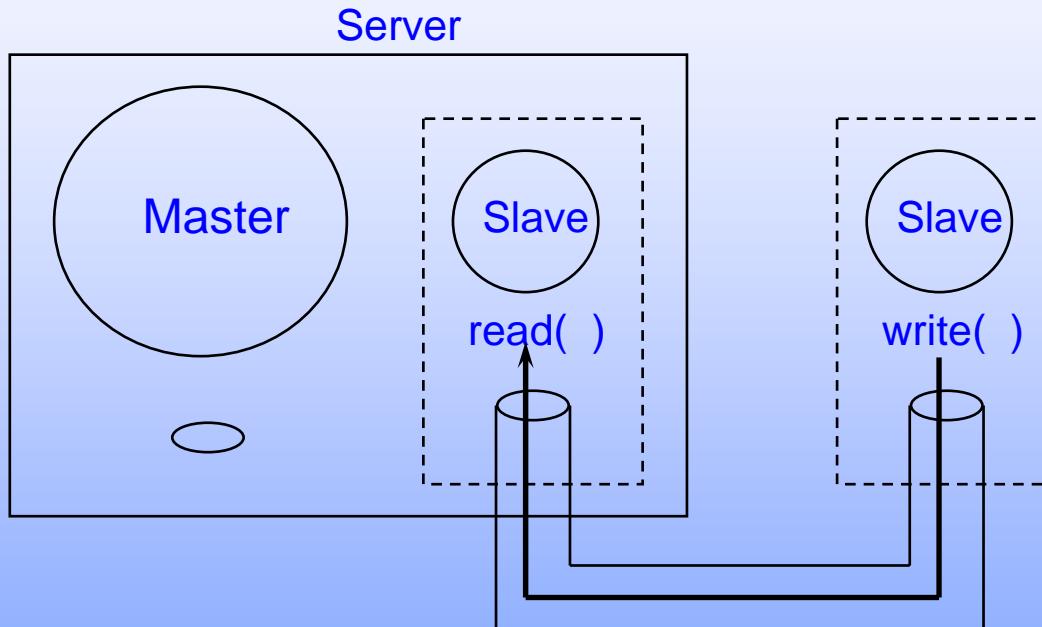
adsLen sizeof (struct sockaddr)

- Blocks until connection is established or timeout.
- Returns **ERROR** on timeout or if no server is bound to *pAdrs*

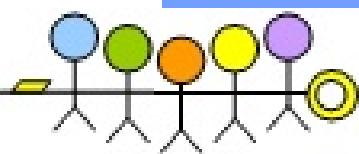


# Exchanging Data

- `read( ) / write( )` may be used to exchange data:

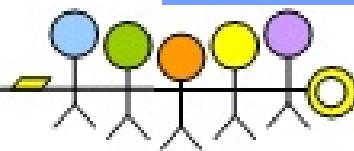


- Caveat : TCP is stream oriented.
  - `write( )` may write only part of message if I / O is nonblocking.
  - `read( )` may read more or less than one message.



## Cleaning up a Stream Socket

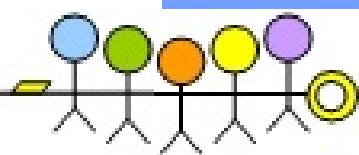
- When done using a socket, ***close( )*** it :
  - Freed resources associated with socket.
  - Attempts to deliver any remaining data.
  - Next ***read( )*** from peer socket will return 0.
- Can also use ***shutdown( )*** to terminate output, while still receiving data from peer socket.



# Setting Socket Options

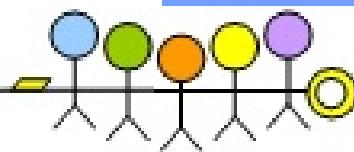
- Options can be enabled on a per socket basis, including :
  - Don't delay *write( )*'s to coalesce small TCP packets.
  - Enable UDP broadcasts.
  - Linger on *close( )* until data is sent.
  - *bind( )* to an address already in use.
  - Change the size of the send / receive buffers.
- Consult UNIX man pages on *setsockopt( )* for details.
- To make a socket non-blocking :

```
int val = 1; /* Set to 0 for blocking I / O */  
ioctl (sock, FIONBIO, &val);
```



## zbuf Socket API

- Improves application performance by minimizing data copies through buffer loaning.
- Application must manage buffers.
- zbuf application can communicate with a standard socket application.
- Supports TCP and UDP protocols.
- See **zbufLib** and **zbufSockLib** for details.
- Proprietary API.



# Network Programming

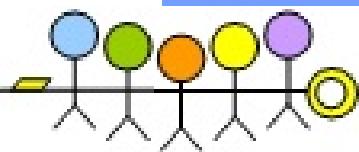
Introduction

Sockets

UDP Socket Programming

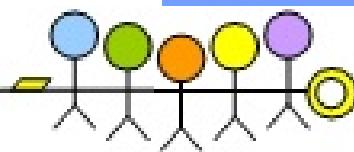
TCP Socket Programming

RPC

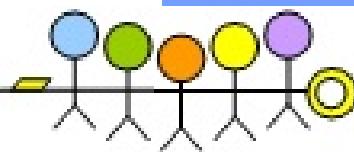
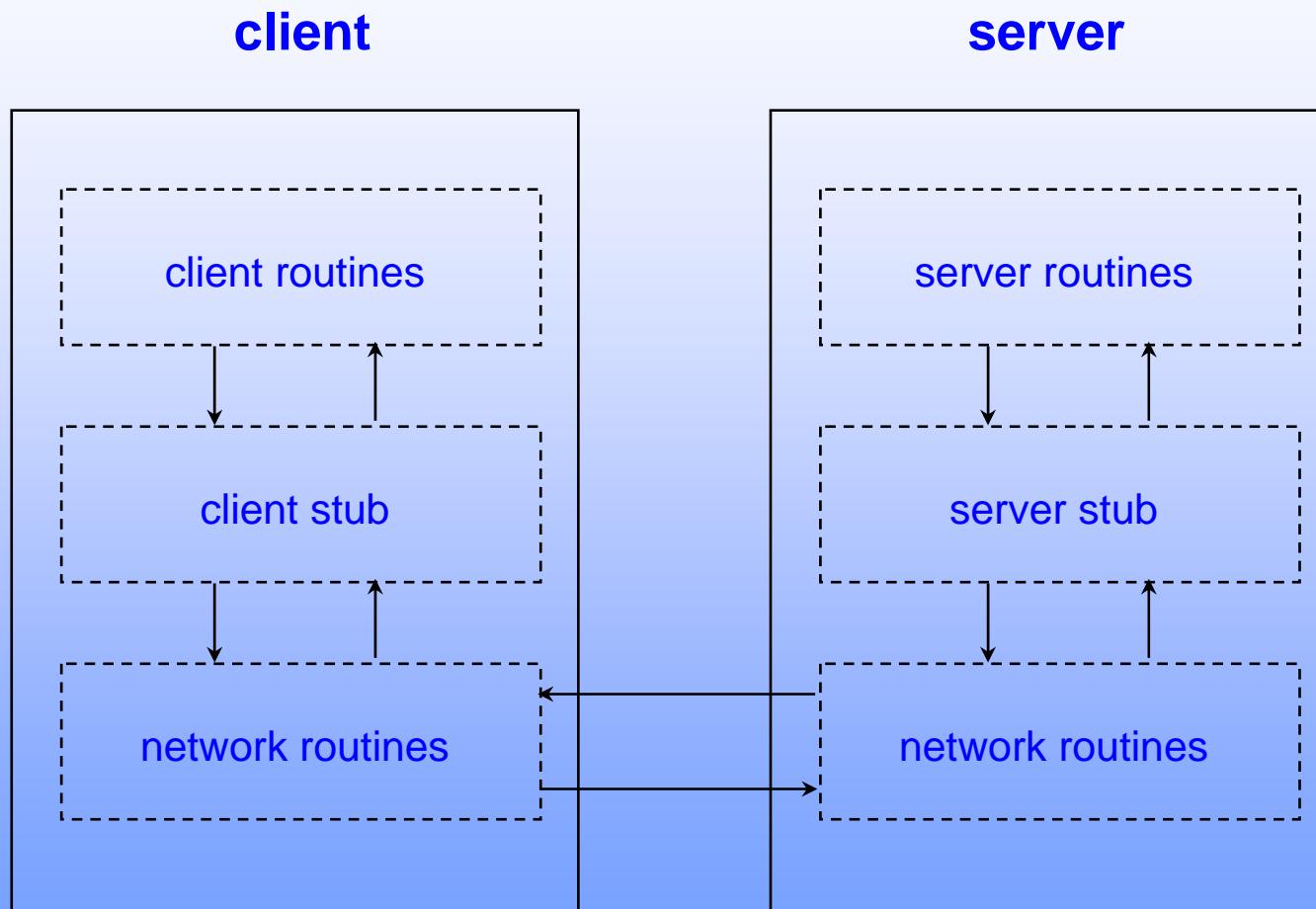


# Overview

- RPC (Remote Procedure Call) provides a standard way to invoke procedures on a remote machine.
- For more information about RPC :
  - Appendix
  - TCP / IP *Illustrated Volume I* (Stevens).
  - *Power Programming with RPC* (O'Reilly & Associates).
  - Documentation and source code can be found in  
**wind/target/unsupported/rpc4.0**



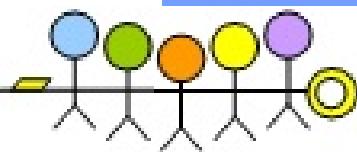
# RPC Client - Server Model



## VxWorks and rpcgen

- **rpcgen** is a RPC protocol compiler.
- From a specification of the remote procedures, **rpcgen** creates :
  - A client stub.
  - A server stub.
  - The XDR routines for packing / unpacking data structures. Not created if all parameters / return\_values are standard data types.
  - A header file for inclusion by client and server.
- Each VxWorks task accessing RPC calls using code produced by **rpcgen** must first initialize access.

### **STATUS rpcTaskInit( )**



# Summary

- Transport layer network Protocols:

TCP      Stream oriented, reliable port-to-port communication.

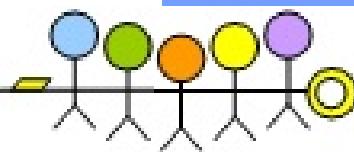
UDP      Packet oriented, non-reliable port-to-port communication.

- Sockets as the interface to network protocols:

- UDP transport protocol.
- TCP transport protocol.
- Configurable socket options.

- zbuf socket API.

- Client / server programming strategies for distributed applications.

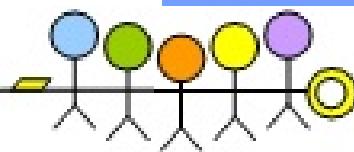


# Chapter 14

# Reconfiguring VxWorks

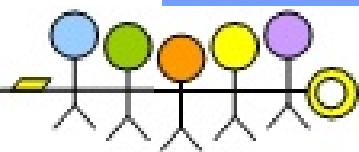
# Chapter 14

# Reconfiguring VxWorks



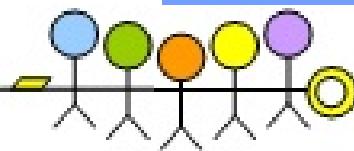
# Overview

- After development is completed, you may need to:
  - Exclude unneeded VxWorks facilities.
  - Link application code into VxWorks.
  - Modify VxWorks's start-up code to spawn application tasks.
- Final application may be:
  - Downloaded from host over a network.
  - Linked with VxWorks and put into ROM.
  - Loaded from a local disk.



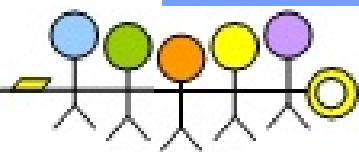
## Typical Steps

1. Exclude unneeded facilities from VxWorks.
2. Modify system start up code to spawn a task to initialize your application.
3. Modify the target's **Makefile** to link your application code with VxWorks.
4. Run **make** to produce either a downloadable or a ROMable application.



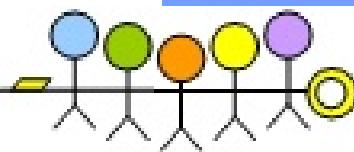
# 1. Scaling VxWorks

- To build a VxWorks image which includes a facility, the facility's corresponding macro(s) must be defined in either:
  - **wind/target/config/all/configAll.h** (generic base configuration
    - avoid modifying this file if possible).
  - **wind/target/config/bspName/config.h** (target specific configuration).
- Alternatively, **WindConfig** can be used to select or unselect VxWorks facilities.
- When the new VxWorks image is built and booted, it will initialize the facilities selected.



# Configuration Dependencies

- Some optional facilities depend on others (e.g., the NFS facility requires RPC).
- `wind/target/src/config/usrDepend.c` checks these dependencies and includes all required facilities.
- As a result of dependency checking, some facilities which apparently were excluded may actually be included in the final system image.

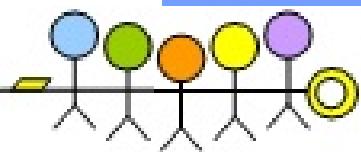


## 2. Modifying System Start-up Code

- ***usrRoot( )*** in ***usrConfig.c***:

- Executed by first VxWorks task.
- Initializes system facilities.
- Start user application if macros defined appropriately :

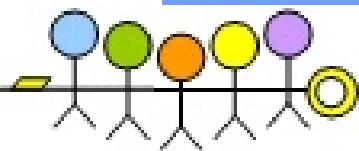
```
void usrRoot( ..... )
{
.....
#endif INCLUDE_USER_APPL
    /* Start the user's application. */
USER_APPL_INIT;      /* must be valid C statement
                         /* or block. */
#endif
}
```



## Example **USER\_APPL\_INIT**

- Define **INCLUDE\_USER\_APPL**
- Define **USER\_APPL\_INIT** to be a statement or block which starts your application. You may make necessary declarations within the block :

```
/* mv162/config.h - Motorola MVME162 header */  
.....  
#define INCLUDE_USER_APPL  
#define USER_APPL_INIT  
{\  
    extern void myApp( void ); \  
    taskSpawn ("tMyApp", 60, 0, 30000, \  
              (FUNCPTR) myApp, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
              0); \  
}  
.....
```

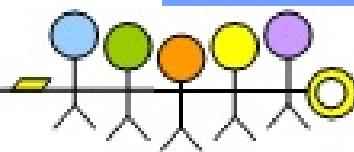


### 3. Linking An Application with VxWorks

Modify the **Makefile** in the target directory:

```
MACH_EXTRA = myCode.o  
.  
.  
myCode.o: . . .  
$(make) -f Makefile_myCode  
.
```

- Alternatively, specify your application's modules on the command line with **ADDED\_MODULES**.  
% make -f Makefile\_mycode  
% make ADDED\_MODULES = muCode.o vxWorks\_rom



## 4. Building VxWorks

- Use makefile in target directory to rebuild VxWorks or boot ROMs:

- Go to the target directory.
  - **make object**

- Common makefile objects:

**vxWorks**

VxWorks image (default).

**vxWorks\_rom .hex**

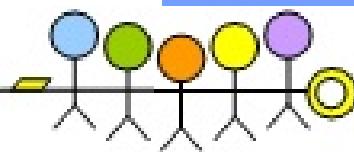
ROM-able VxWorks image

**vxWorks\_res\_rom\_nosym**

ROM-resident standalone  
version of VxWorks without  
symbol table. Starts fast, uses  
little RAM

**bootrom .hex**

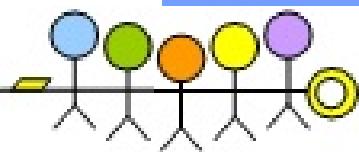
Boot ROM code.



# ROMable Objects

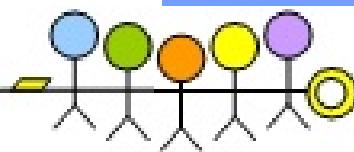
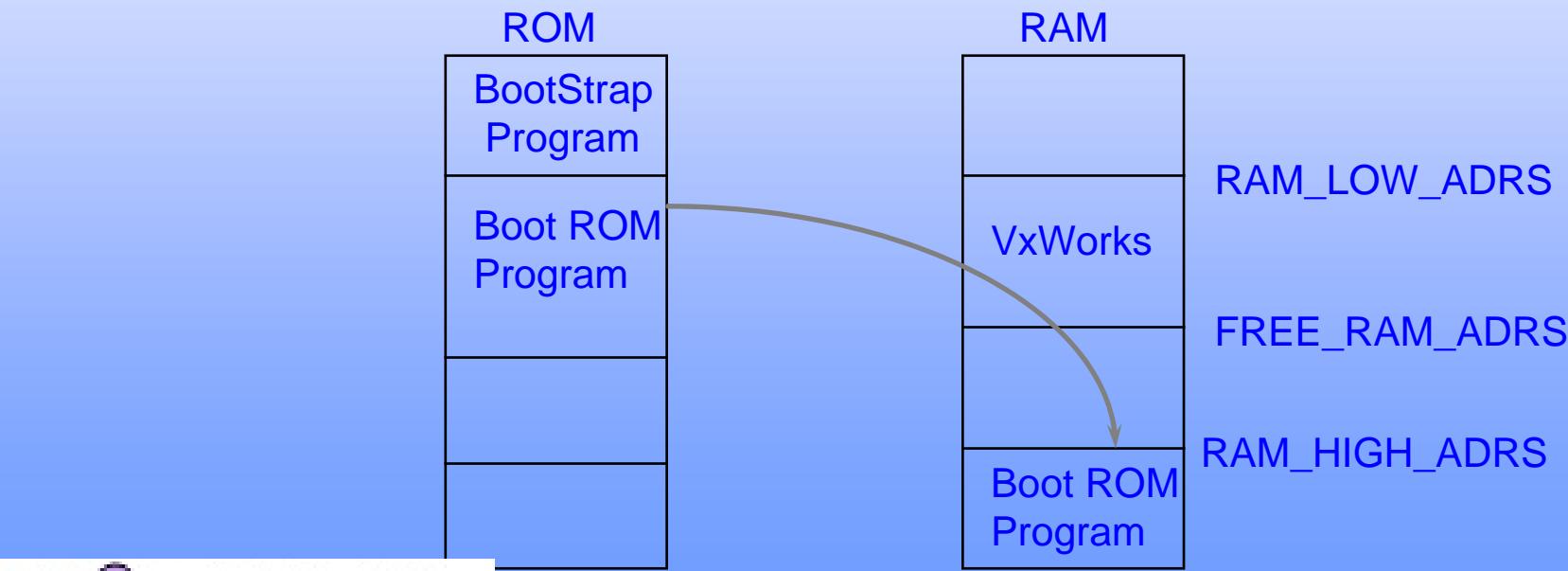
- ROMable objects copy themselves into RAM before running.
- Most of the ROMable VxWorks objects are compressed. An exception to this is **vxWorks\_rom**
- If using large capacity ROM's, may need to modify the constant **ROM\_SIZE** in both:
  - **wind/target/config/target/config.h** .
  - **wind/target/config/target/Makefile**.

Make will fail if the object is larger than the **ROM\_SIZE**



# Downloadable Objects

- ROM boot code copied into RAM at **RAM\_LOW\_ADDRS**, bootstrap part decompresses rest to **RAM\_HIGH\_ADDRS**.
- VxWorks system image loaded at **RAM\_LOW\_ADDRS** (typically 0x1000).

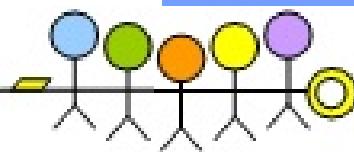


## Rebuilding Boot ROM's

- If system image is large (>570 Kbytes, typically), downloading will overwrite the booting code, failing while printing:

Loading . . . 400316 + 28744 + 23852

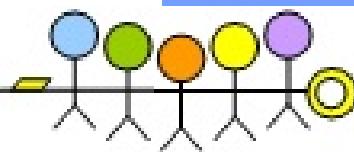
- May need change **RAM\_HIGH\_ADRS** in:
  - **wind/target/config/target/config.h** .
  - **wind/target/config/target/Makefile**.
- New ROM's are also required to boot off of a local disk



# Standard make Targets

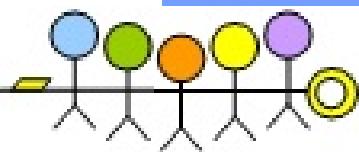
TYPES OF IMAGES	VxWorks Tornado	VxWorks Standalone	Boot Program
ROMable compressed	—	vxWorks.st_rom	bootrom
ROMable uncompressed	vxWorks_rom	—	bootrom_uncmp
ROM resident	vxWorks.res_rom_nosym	vxWorks.res_rom	bootrom_res
Downloadable uncompressed	vxWorks	vxWorks.st	—

- Not all BSP's will support all of these images. Some BSP's support additional images
- Standalone VxWorks has a target shell built-in symbol table.
- target/make/rules.bsp** has the make rules for these images.



## Additional Configuration

- Many other configuration changes are possible :
  - Change the default boot line in **config.h**
  - Change boot code in **bootConfig.c**.
  - Modify I/O system parameters in **ConfigAll.h**
- For details, see the *Configuration* chapter in the Programmer's Guide.



# ROM-based VxWorks Start-up

## 1. *\_romInit* in **config/bspName/romInit.s**

- Minimal initialization : mask interrupts, disable caches, set initial initial stack, initialize DRAM access.

## 2. *romStart( )* in **config/all/bootinit.c**

- Copy text / data segment(s) to RAM, clear other RAM.  
Decompress if necessary.

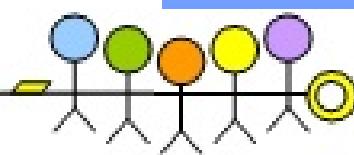
## 3. *usrInit( )* in **bootconfig.c** or **usrConfig.c**

- Do pre-kernel hardware & software initialization; start kernel by calling ***kernellInit( )***.

## 4. *usrRoot( )* in **bootconfig.c** or **usrConfig.c**

- Initialize facilities configured into VxWorks; start boot program or user application

## 5. Boot program or user application.



# Downloaded VxWorks Start-up

1. Boot program loads VxWorks over network, from local disk, etc. Call :

- Minimal initialization : mask interrupts, disable caches, set initial initial stack, initialize DRAM access.

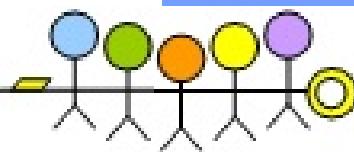
2. `_sysInit( )` in **config/bspName/sysALib.s**

- Similar to `_romInit`: mask interrupts, disable caches, set initial stack pointer. Then call:

3. `usrInit( )` in **config/all/usrConfig.c**

4. `usrRoot( )` in **config/all/usrConfig.c**

5. User application.



# Summary

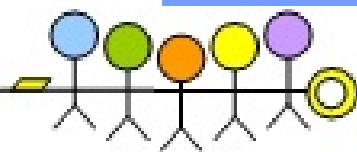
1. Modify **config.h** to exclude unneeded facilities.
2. Define **INCLUDE\_USER\_APPL** and **USER\_APPL\_INIT** to make **usrRoot()** spawn your tasks.
3. Modify the **Makefile** to link your application into the VxWorks system image

```
MACH_EXTRA = myCode.o
```

```
...
```

4. Remake the system image. New image can be booted or ROM-ed.

```
% make vxWorks  
% make vxWorks_rom.hex  
% make bootrom.hex
```



OVER

Thank you very much

Good luck

