

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: ^POB;
end;
```

其中 POB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。

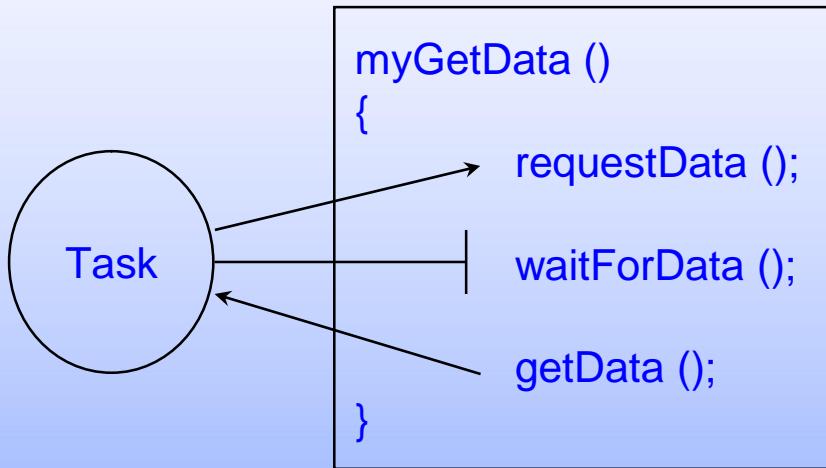
Semaphores

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.

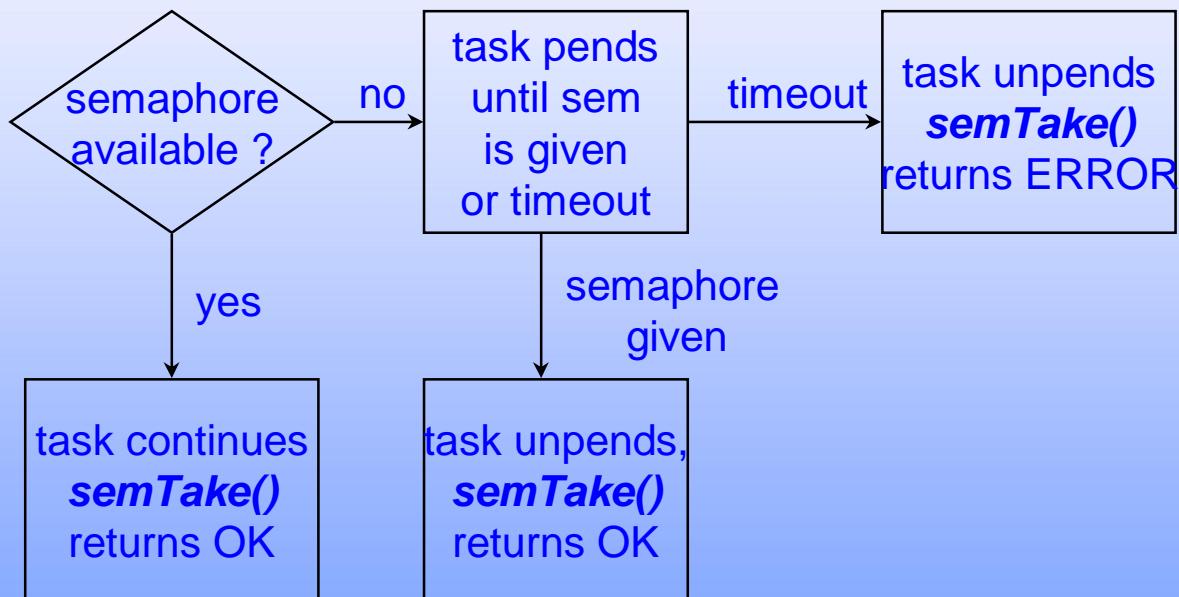
Taking a Semaphore

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

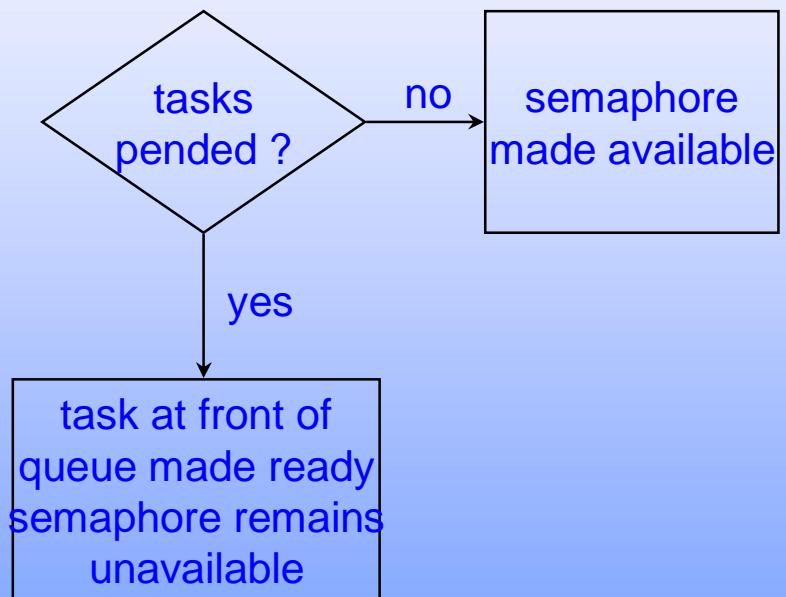


Giving a Semaphores

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

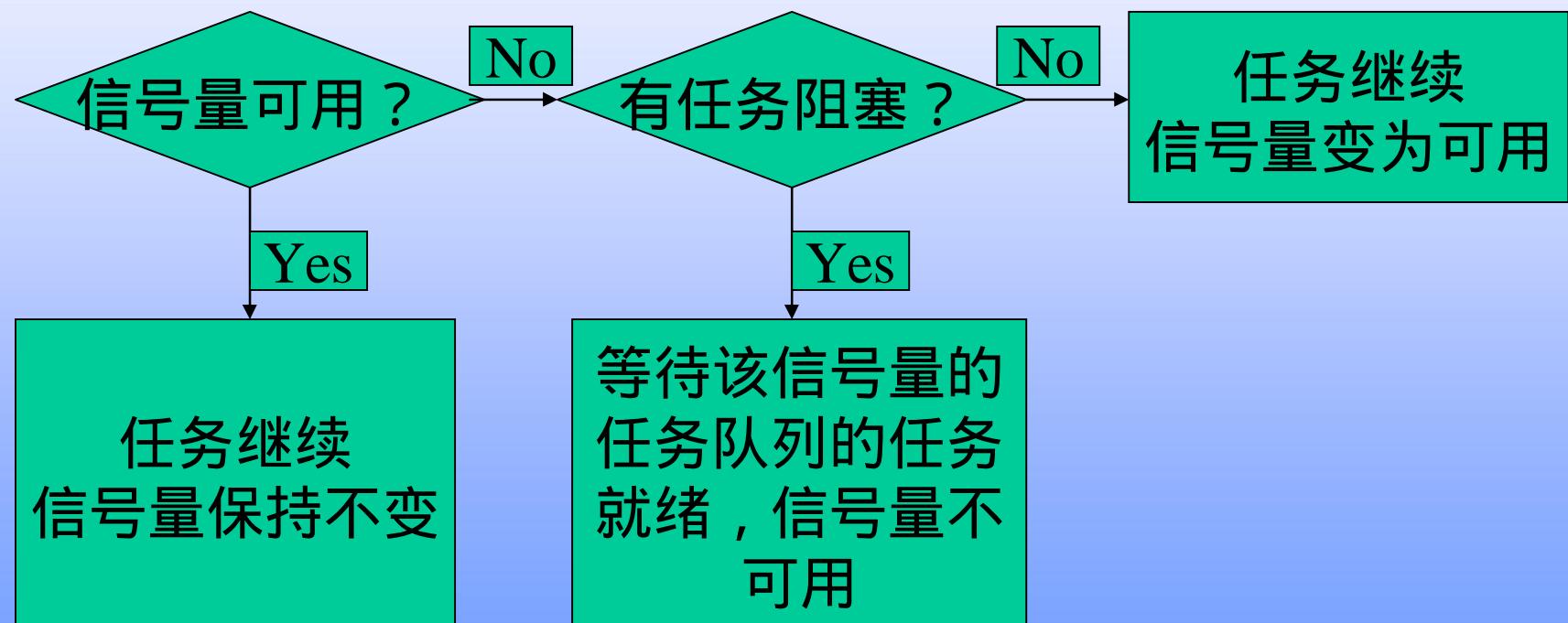
- Fast event occurrences can cause lost information.
- Suppose a VxWorks task (priority=100) is executing the following code, with semId initially unavailable:

```
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

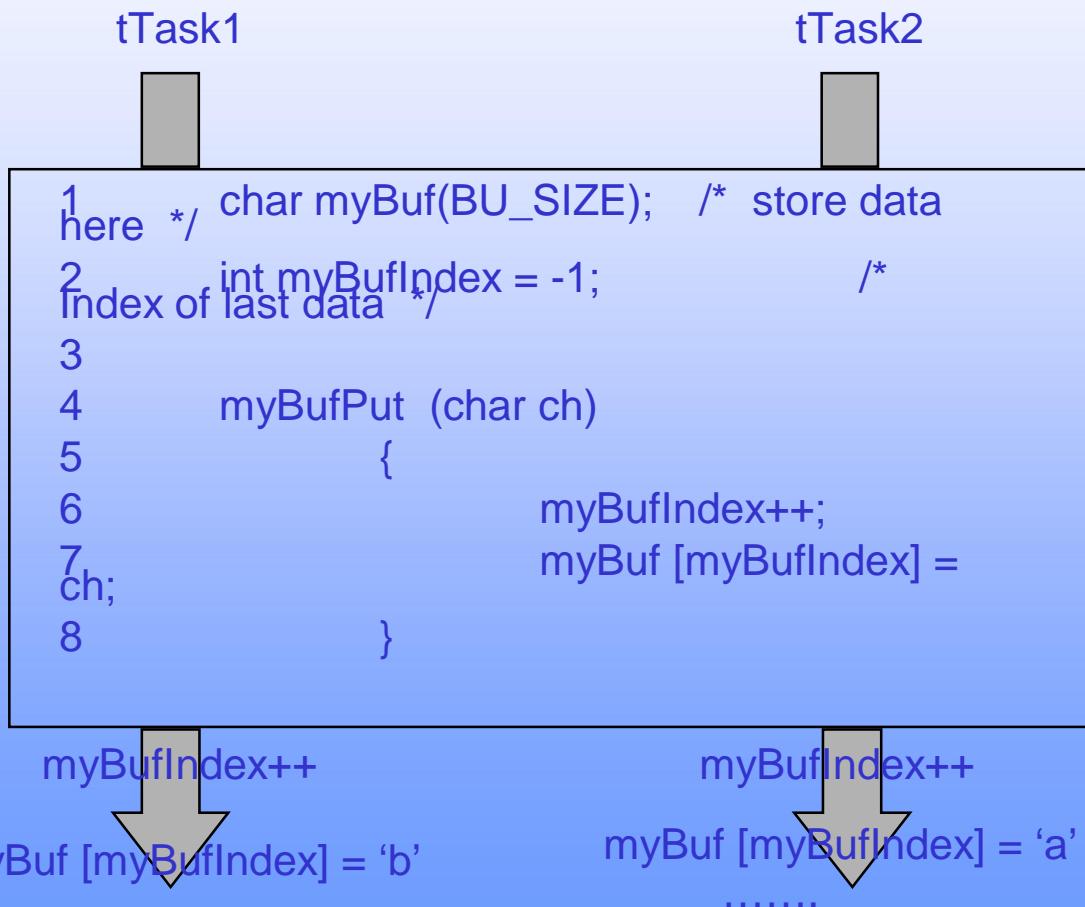
Binary Semaphores and Synchronization

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



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

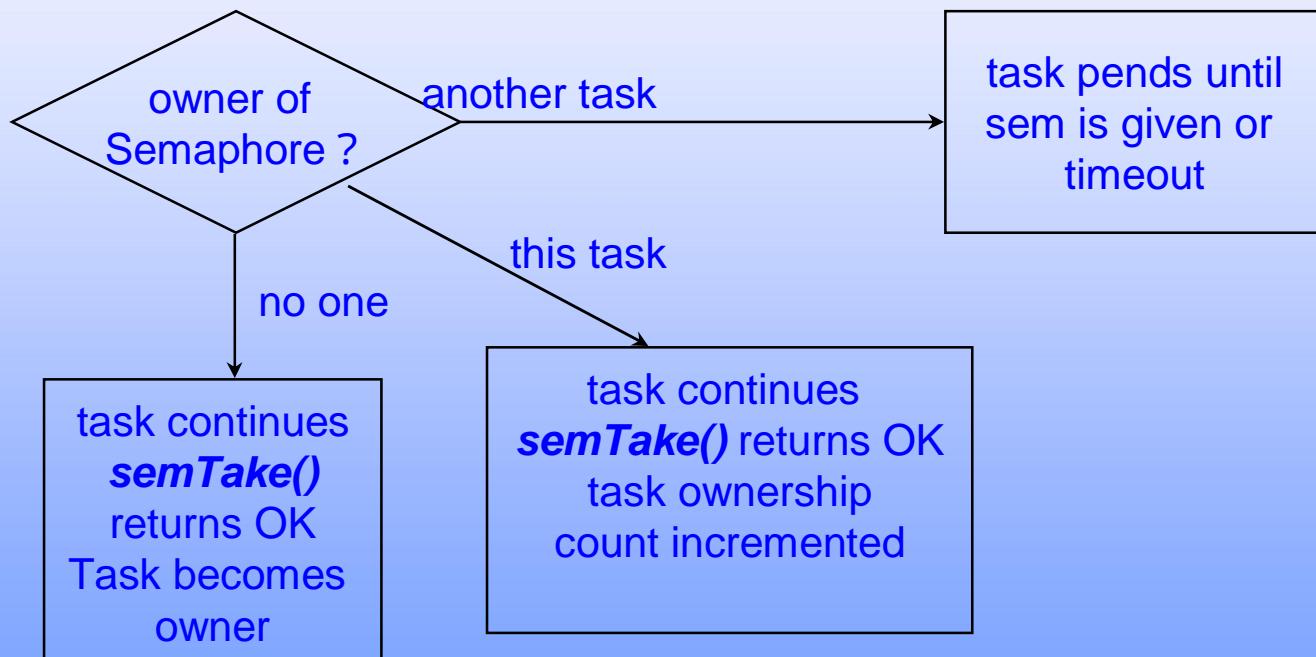
SEM_INVERSION_SAFE

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

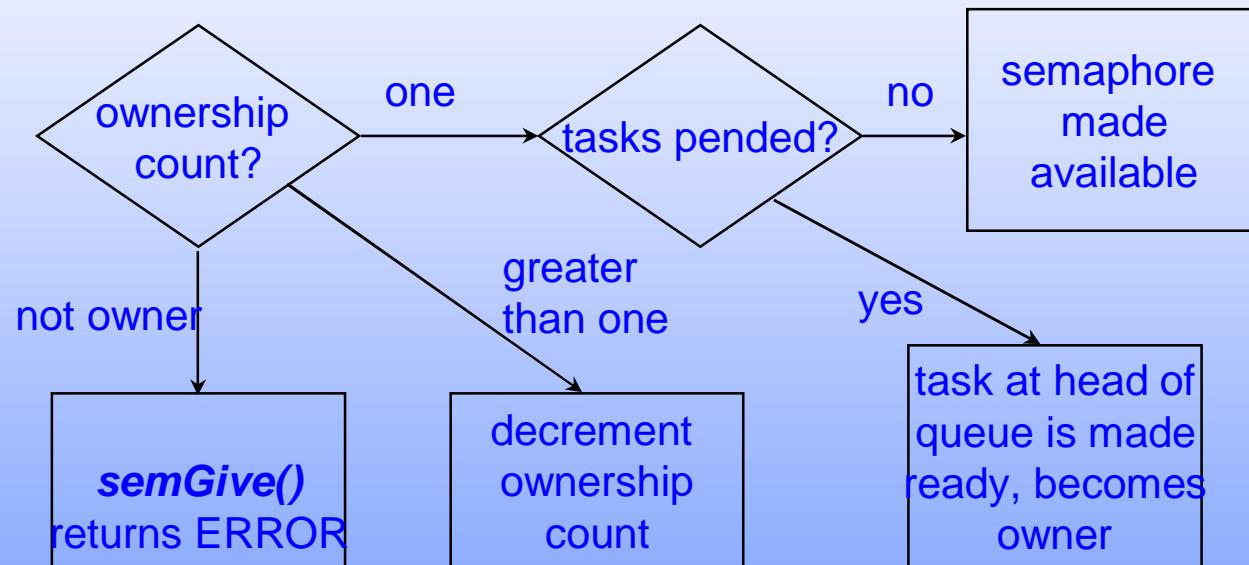
Mutex Ownership

- 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
11         (SEM_Q_PRIORITY |
12             SEM_INVERSION_SAFE |
13             SEM_DELETE_SAFE );
14
15 void myBufPut (char ch)
16 {
17     semTake(mySemId, WAIT_FOREVER);
18     myBufIndex++;
```

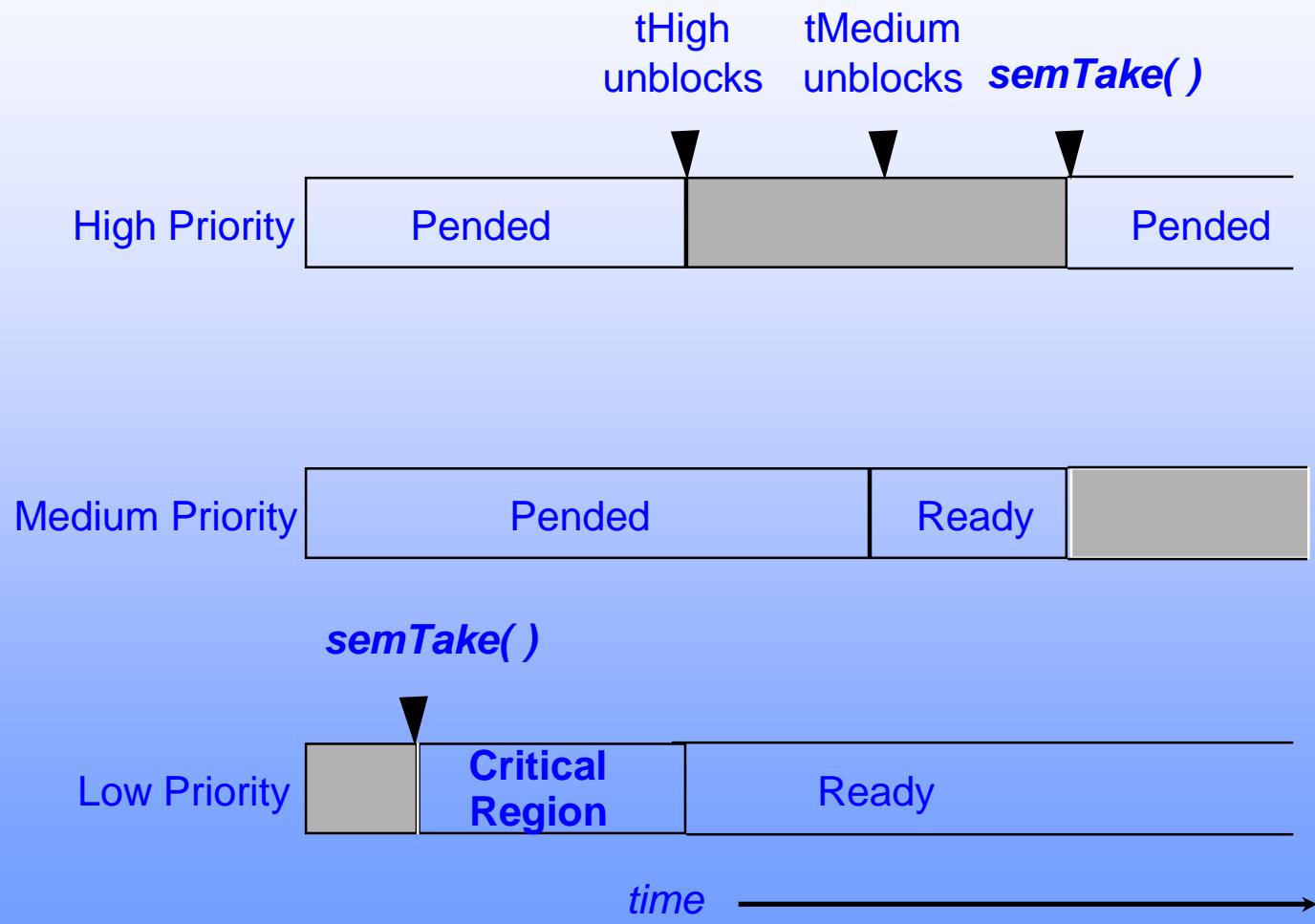
Deletion Safety

- 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



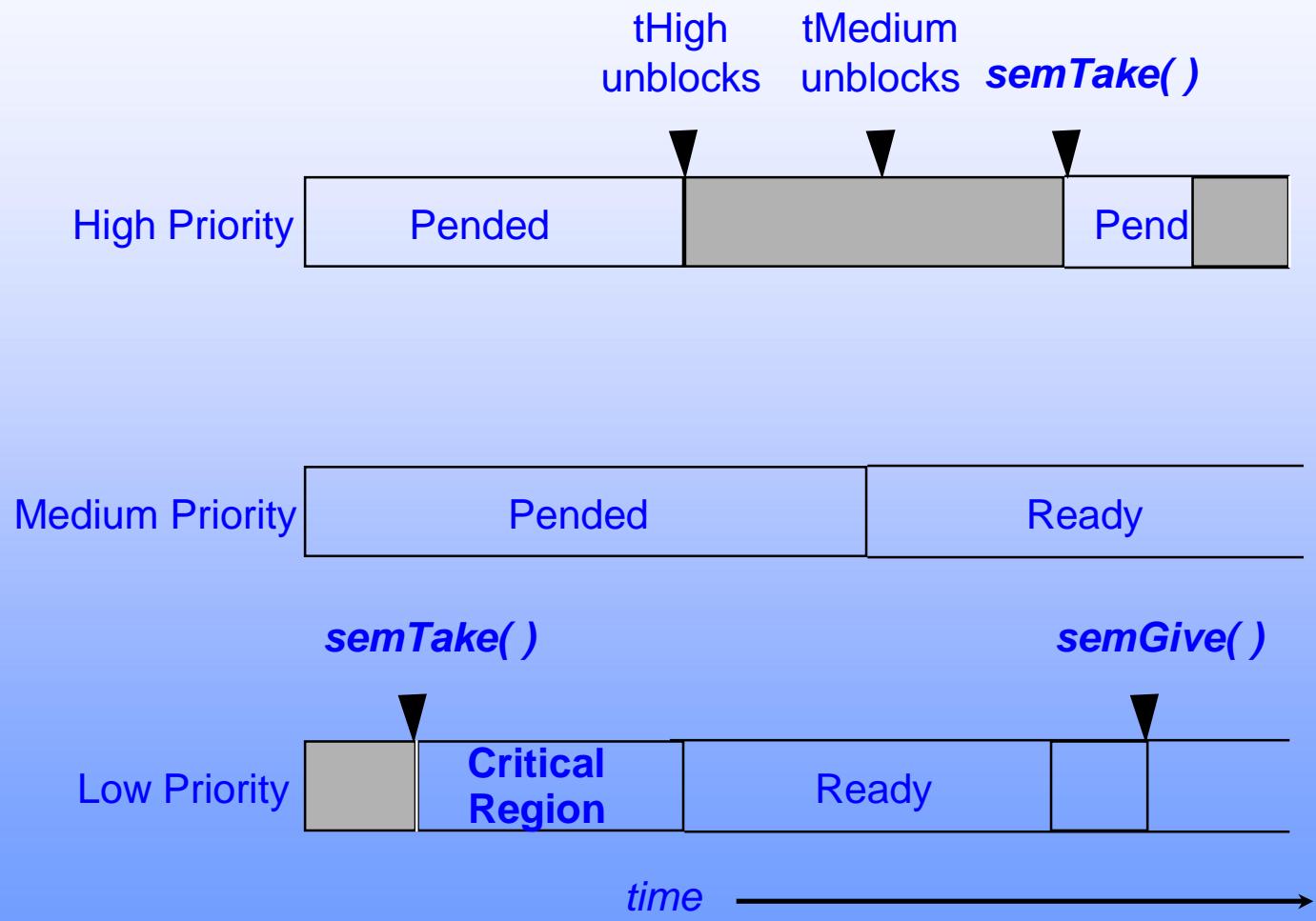
优先级逆转/优先级继承

- 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被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**).

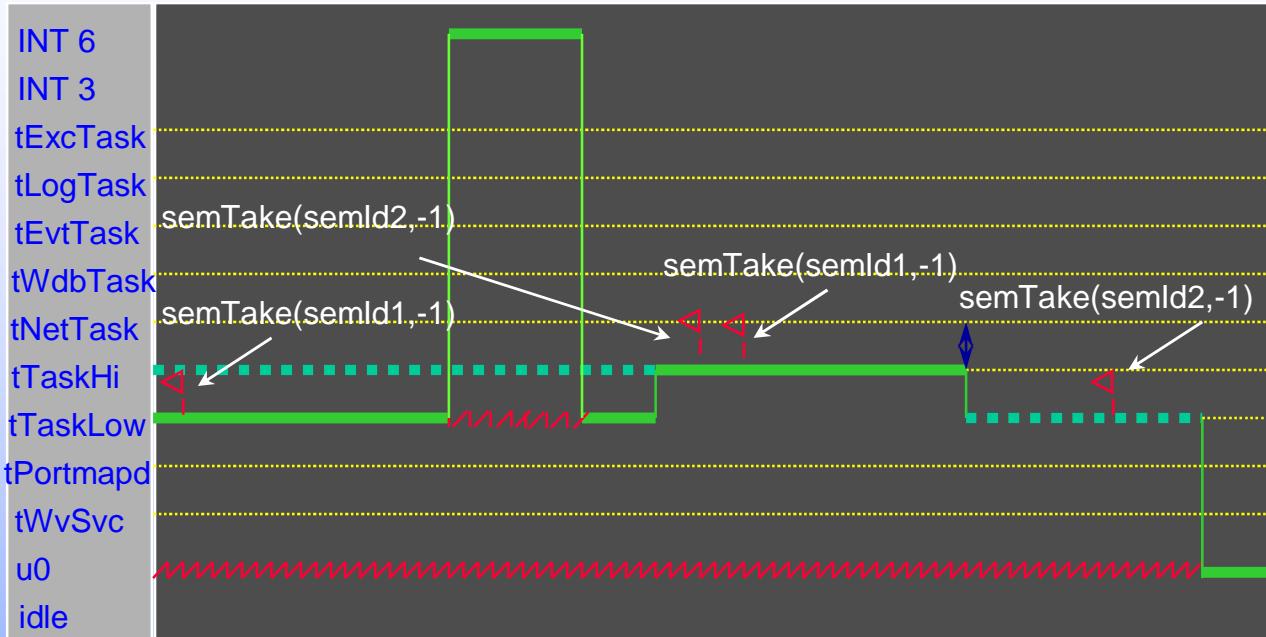
Priority Inversion Safety



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.

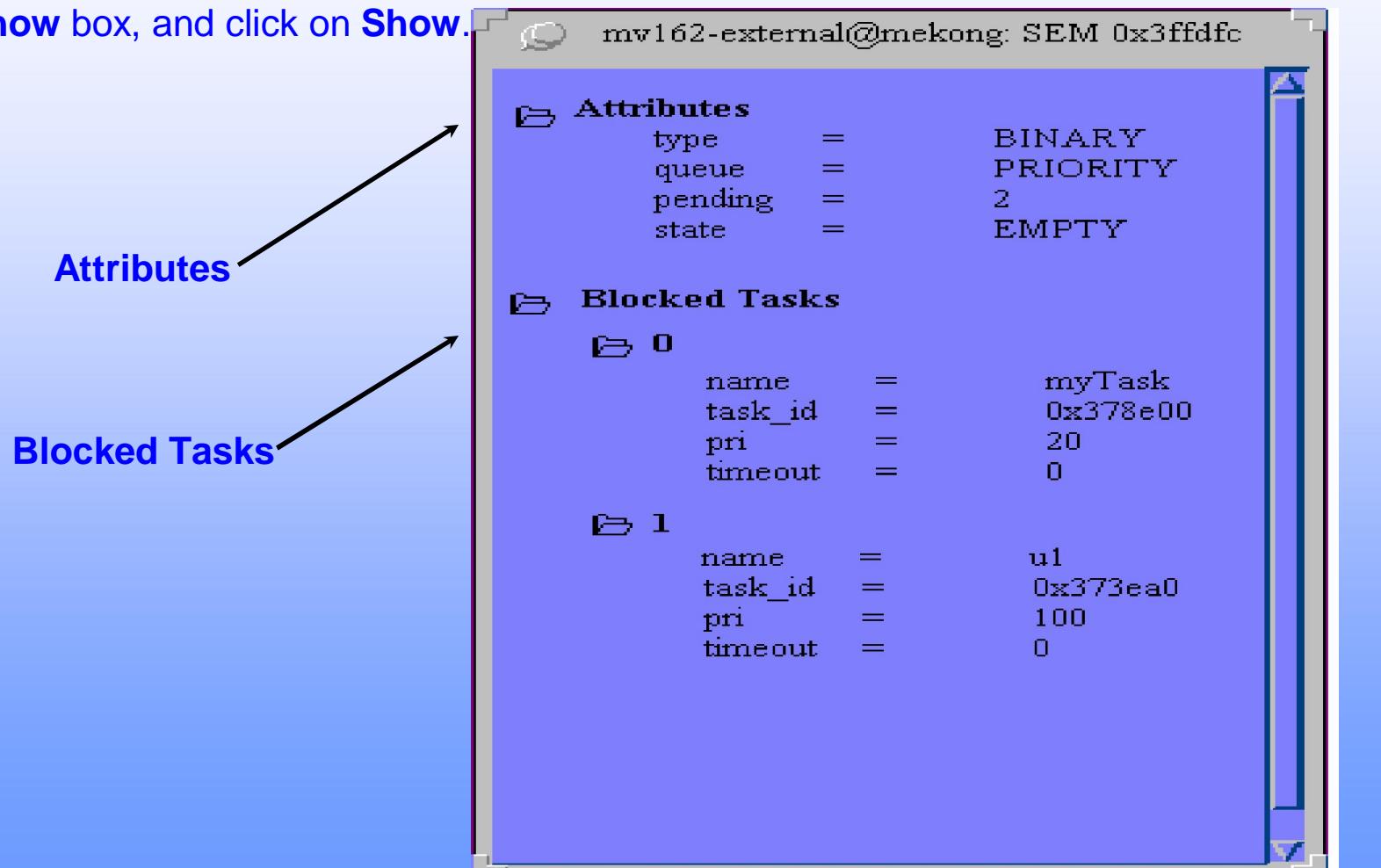
Common Routines

- Additional semaphore routines :

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

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

Binary

semBCreate

Mutual Exclusion

semMCreate

semTake, semGive

show

semDelete

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

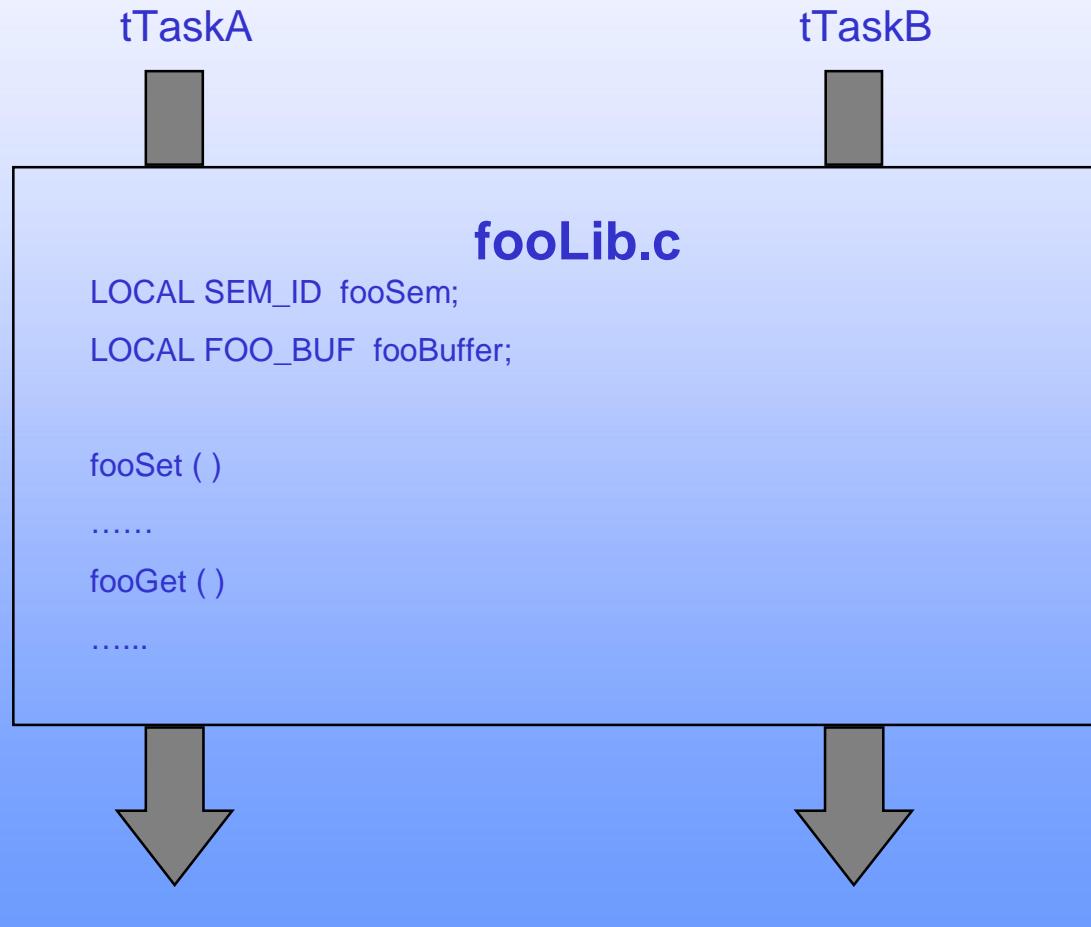
Message Queues

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

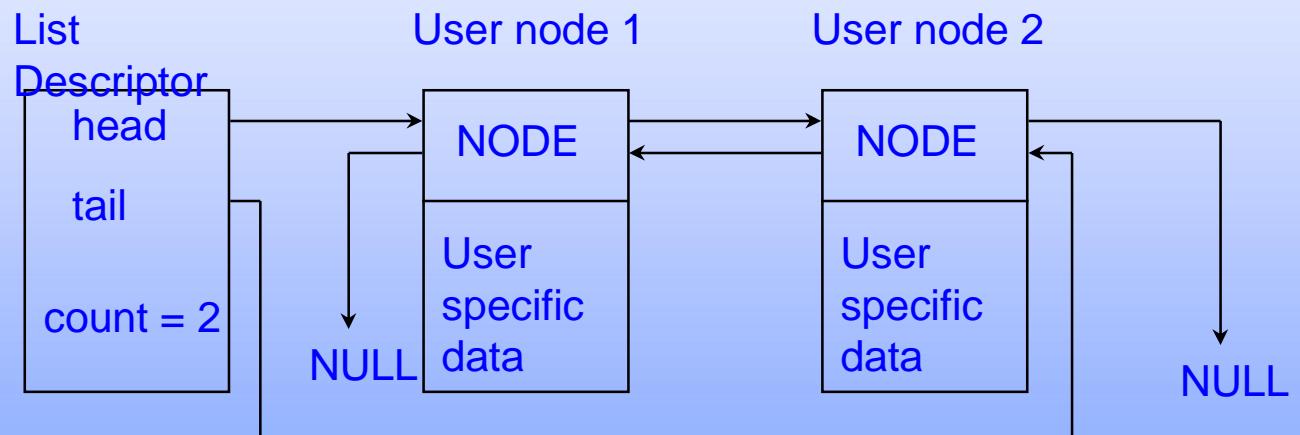
Pipes

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

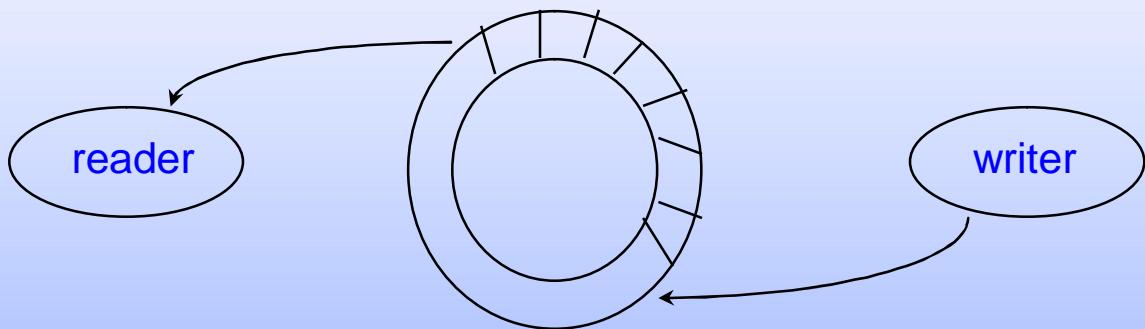
- **IstLib** contains routines to manipulate doubly linked lists.



- 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

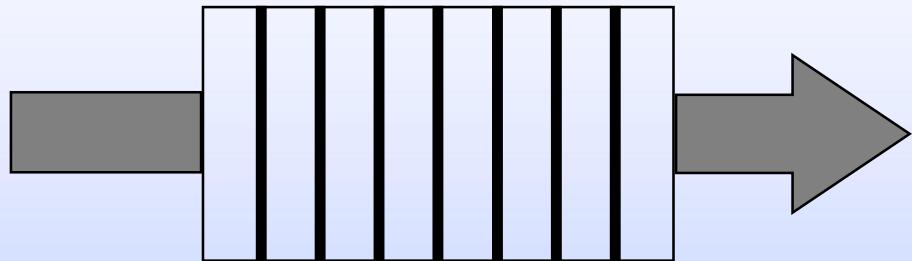
Introduction

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` Maximum number of messages on the queue.

`maxMsgLength` Maximum size in bytes of a message on the queue.

`options` Queue type for pended tasks
`(MSG_Q_FIFO`

or `MSG_Q_PRIORITY`)

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

Sending Messages

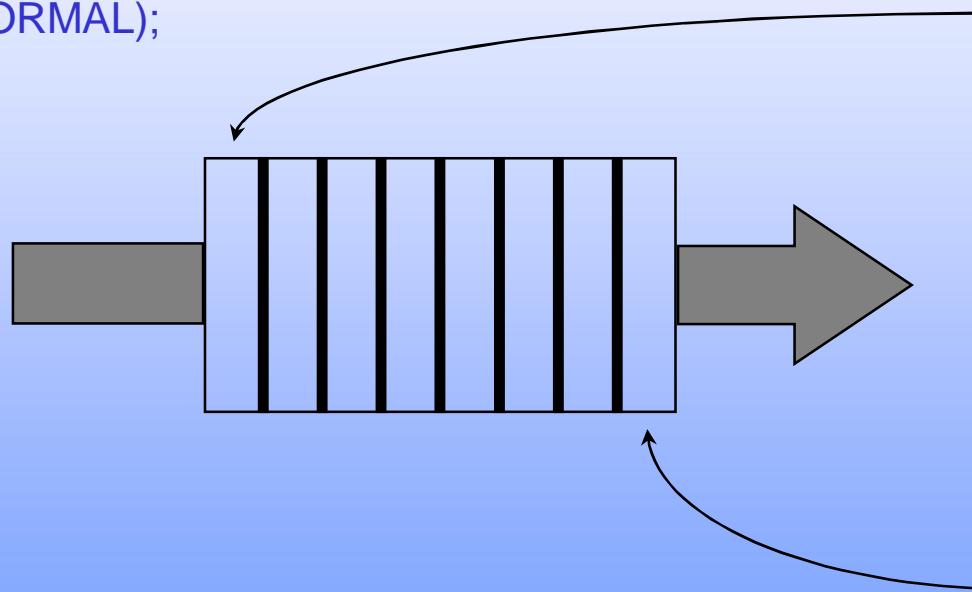
STATUS msgQSend (msgQId, buffer, nBytes, timeout,
priority)

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 <i>msgQCreate()</i> .
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, WAIT_FOREVER or NO_WAIT .

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

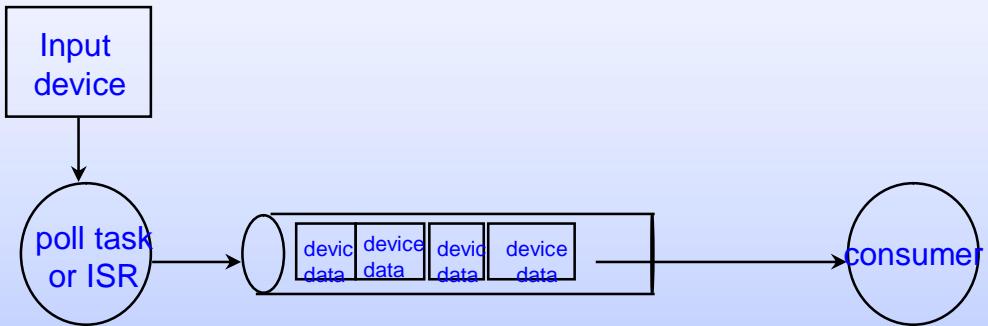
Deleting a Message Queue

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**.

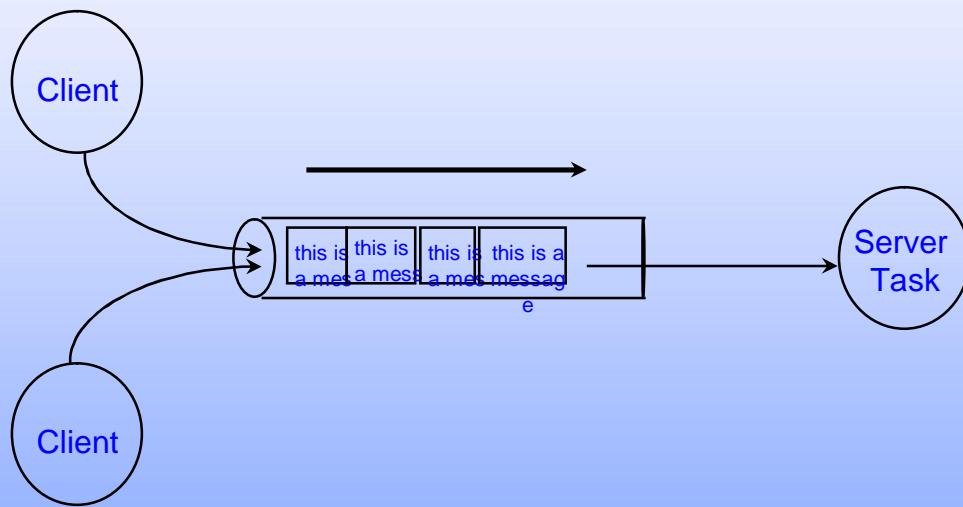
Gathering Data with Message Queues



- 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.

[GO TO APPENDIX A3](#)

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

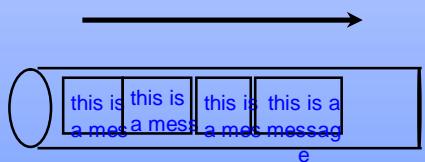
Shared Memory

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

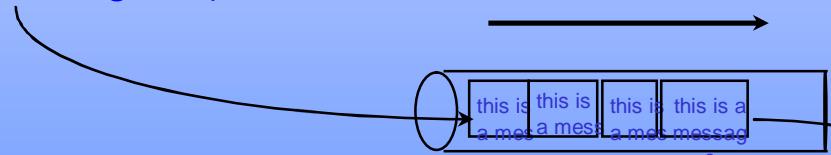
-->devs

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.