

# 嵌入式系统实验

## 第4-2节 VxWorks任务间通信

夏海轮

北京邮电大学 信息与通信工程学院



火龙果·整理  
uml.org.cn

# 本节主要内容



1. 共享存储器（Shared Memory）
2. 互斥操作（Mutual Exclusion）
3. 信号量（Semaphore）
4. 消息队列（Message Queue）
5. 管道（Pipe）



# 1. 共享存储器



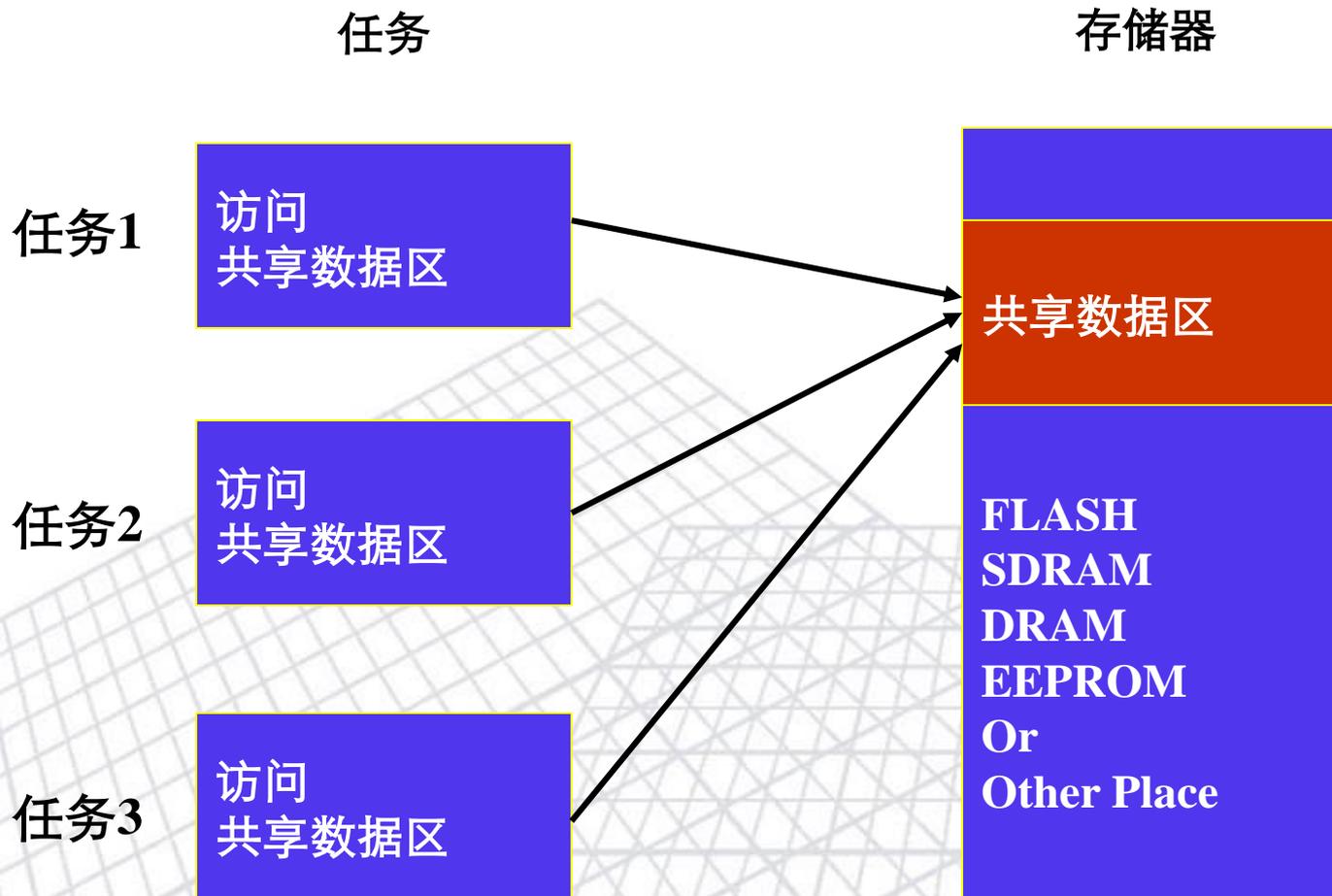
## ◆ 基本原理

## ◆ 链表 (List)

## ◆ 环 (Ring)



# 原理框图



# 编程实例



tTaskA



二进制  
信号量



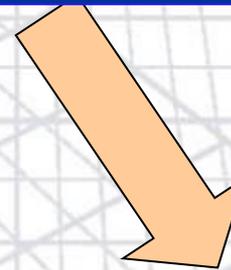
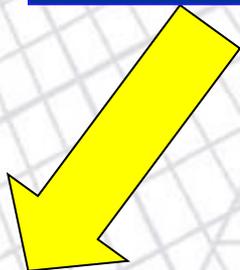
tTaskB

**fooLib.c**

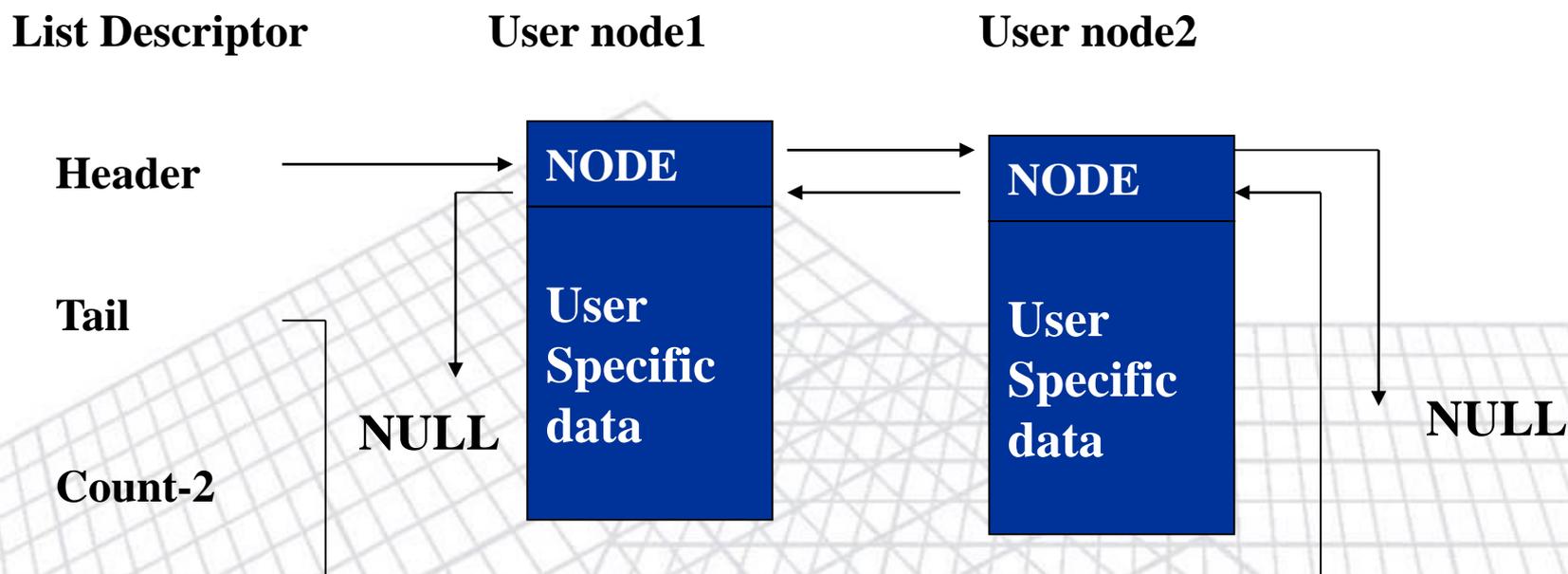
```
LOCAL SEM_ID fooBinSemId, fooMutexId;  
LOCAL FOO_BUF fooBuffer;  
fooSet();  
...  
fooGet();  
...
```

共享内存

互斥信号量



## ◆ lstLib库提供对双向链表进行操作的函数



- lstLib库中对双向链表进行操作的函数不提供数据的互斥和同步

# 链表库: IstLib



## ◆ 头文件

- target\h\lstLib.h    host\include\lstLib.h

## ◆ 链表标识

- LIST \* pList

## ◆ 链表节点

- NODE \* pNode

```
typedef struct node            /* Node of a linked list. */  
{  
  struct node *next;            /* Points at the next node in the list */  
  struct node *previous;       /* Points at the previous node in the list */  
} NODE;  
  
typedef struct                 /* Header for a linked list. */  
{  
  NODE node;                    /* Header list node */  
  int count;                   /* Number of nodes in list */  
} LIST;
```



# 链表库函数



- ◆ [lstLibInit](#)( ) - initializes lstLib module
- ◆ [lstInit](#)( ) - initialize a list descriptor
- ◆ [lstAdd](#)( ) - add a node to the end of a list
- ◆ [lstConcat](#)( ) - concatenate two lists
- ◆ [lstCount](#)( ) - report the number of nodes in a list
- ◆ [lstDelete](#)( ) - delete a specified node from a list
- ◆ [lstExtract](#)( ) - extract a sublist from a list
- ◆ [lstFirst](#)( ) - find first node in list
- ◆ [lstGet](#)( ) - delete and return the first node from a list
- ◆ [lstInsert](#)( ) - insert a node in a list after a specified node
- ◆ [lstLast](#)( ) - find the last node in a list
- ◆ [lstNext](#)( ) - find the next node in a list
- ◆ [lstNth](#)( ) - find the Nth node in a list
- ◆ [lstPrevious](#)( ) - find the previous node in a list
- ◆ [lstNStep](#)( ) - find a list node *nStep* steps away from a specified node
- ◆ [lstFind](#)( ) - find a node in a list
- ◆ [lstFree](#)( ) - free up a list

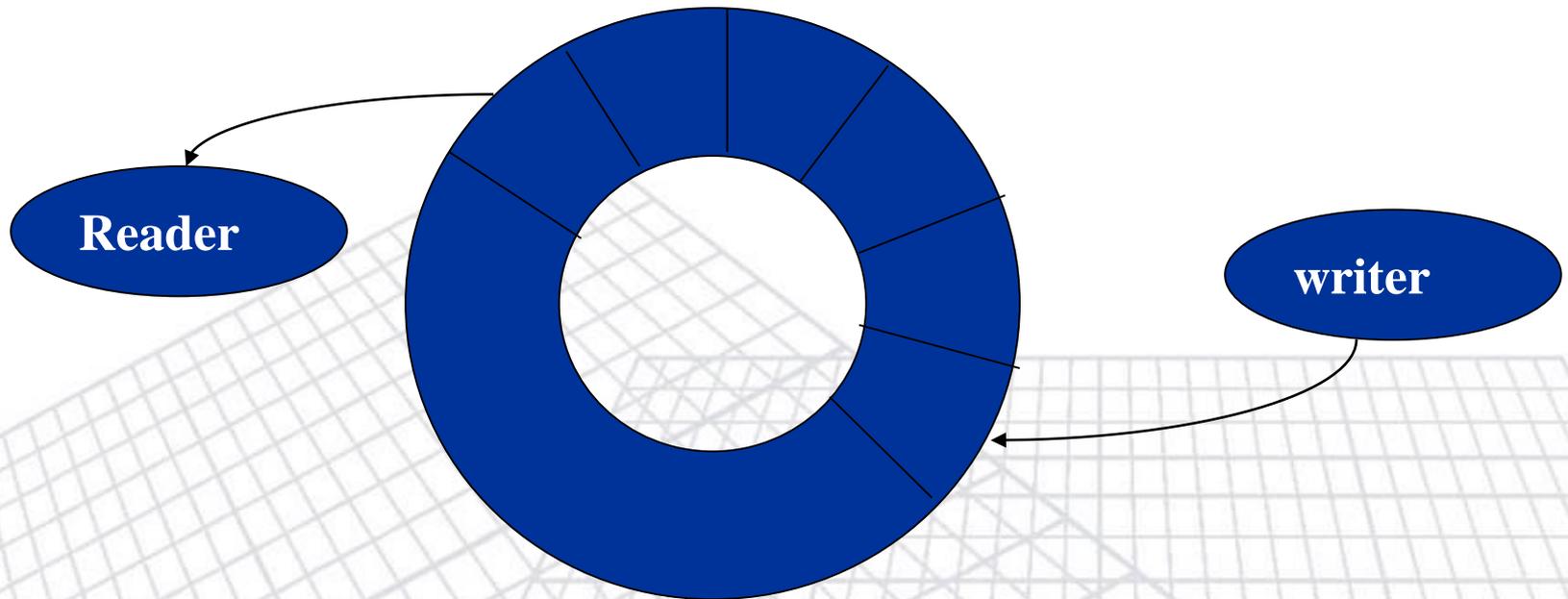


LIST List;

```
typedef struct INFO /* Define student information Structure */
{
    _UINT32 no;          /* Index of the student */
    _UINT08 name[11];   /* Name , no more than 10 chars */
    _UINT08 mobile[12]; /* Mobile phone, no more than 11 chars */
    _UINT08 gender;     /* Gender , 1 male , 0 female */
    _UINT08 age;        /* Age */
}tINFO; /* Student information structure */
```

```
typedef struct STUDENT
{
    NODE node;          /* Node of the List */
    tINFO info;         /* student information */
}tSTUDENT; /* Student List structure */
```

- rngLib中包含对环形缓存（FIFO数据流）的操作



- ◆ 如果只有一个reader和一个writer，就不需要互斥机制，否则用户要自己提供互斥。
- ◆ 没有同步机制

# 环库: rngLib



## ◆ 头文件

- **target\h\lstLib.h**

## ◆ 环标识

- RING\_ID ringId,

## ◆ 环缓存

- char \* buffer, int nbytes

```
typedef struct          /* RING - ring buffer */
{
  int pToBuf;           /* offset from start of buffer where to write next */
  int pFromBuf;        /* offset from start of buffer where to read next */
  int bufSize;         /* size of ring in bytes */
  char *buf;           /* pointer to start of buffer */
} RING;
```



# 环库函数



- ◆ [rngCreate](#)( ) - create an empty ring buffer
- ◆ [rngDelete](#)( ) - delete a ring buffer
- ◆ [rngFlush](#)( ) - make a ring buffer empty
- ◆ [rngBufGet](#)( ) - get characters from a ring buffer
- ◆ [rngBufPut](#)( ) - put bytes into a ring buffer
- ◆ [rngIsEmpty](#)( ) - test if a ring buffer is empty
- ◆ [rngIsFull](#)( ) - test if a ring buffer is full (no more room)
- ◆ [rngFreeBytes](#)( ) - determine the number of free bytes in a ring buffer
- ◆ [rngNBytes](#)( ) - determine the number of bytes in a ring buffer
- ◆ [rngPutAhead](#)( ) - put a byte ahead in a ring buffer without moving ring pointers
- ◆ [rngMoveAhead](#)( ) - advance a ring pointer by n bytes



# 共享缓存的特点



- ◆所有的任务都在同一个地址空间
- ◆任务间的通信可能协议用户自定义的数据结构
  - 编写一个函数库来实现对这些全局变量或静态变量的操作
  - 所有使用这个函数库中函数的任务实际上是对同一个物理空间进行操作
  - 信号量可以用来提供互斥和同步
- ◆VxWorks提供函数库对公共数据如环形缓存和链表的操作



## 2. 互斥操作



◆ 中断锁

◆ 任务锁

◆ 互斥信号量（下一节）



# 中断锁的使用



## ◆ intLock()/intUnlock()

- 禁止中断
- 用于保护被任务或中断使用的资源
- 注意要保持critical region短

**funcA ()**

**{**

**int lock = intLock();**

**·**

**· /\* critical region of code that cannot be interrupted \*/**

**·**

**intUnlock (lock);**

**}**



# 任务锁的使用



## ◆ taskLock()/taskUnlock()

- 禁止所有其它任务执行
- 当非常频繁地做某事时使用
- 注意要保持critical region短

```
funcA ()
```

```
{
```

```
    taskLock ();
```

```
    .
```

```
    . /* critical region of code that cannot  
      be interrupted */
```

```
    .
```

```
    taskUnlock ();
```

```
}
```



# 任务的禁止抢占



- ◆ 当要频繁地做某事时，最好是使用禁止抢占，而不是使用互斥信号量。
- ◆ 调用**taskLock()**来禁止抢占
- ◆ 调用**taskUnlock()**重新使能抢占
- ◆ 任务的禁止抢占并不禁止中断
- ◆ 如果任务被锁定，则抢占被重新使能。当任务重新开始执行时，抢占又被锁定
- ◆ 禁止抢占是禁止所有其它任务运行，而不仅仅是对资源进行竞争的任务



# 3. 信号量



- ◆ 概述
- ◆ 二进制信号量 (binary semaphores)
- ◆ 互斥信号量 (mutual exclusion semaphores)
- ◆ 计数信号量 (counting semaphores)
- ◆ 共享存储器信号量 (shared memory semaphores)

## ◆ 什么是信号量

- 是Vxworks操作系统提供的一种同步和互斥操作机制。实现任务间的信息传递。

## ◆ VxWorks提供三种类型的信号量

- 二进制信号量：最快和常用的信号量，提供阻塞方式，用于实现同步或互斥
- 互斥信号量：用于实现互斥问题的特殊的二进制信号量，解决具有互斥、优先级继承、删除安全和递归等情况
- 计数信号量：类似于二进制信号量，记录信号量被释放的次数。适合于一个资源的多个实例需要保护的情况

```
myGetData()
```

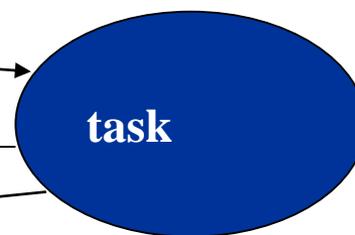
```
{
```

```
  requestData();
```

```
  waitForData();
```

```
  getData();
```

```
}
```



- ◆ 任务可能会等待一个事件的发生
- ◆ 一直等待（**polling**）的效率非常低
- ◆ 阻塞一直到事件发生会更好

# 同步问题的解决



- ◆ 为事件创建一个信号量
- ◆ 二进制信号量有下面两种状态
  - Full (事件已经发生)
  - Empty (事件没有发生)
- ◆ 等待事件的任务调用**semTake()**, 并一直阻塞到得到信号量
- ◆ 检测到事件的任务或中断调用**semGive()**; 解锁了等待事件的任务





- ◆ **semLib** - general library
- ◆ **semBLib** - binary semaphores
- ◆ **semCLib** - counting semaphores
- ◆ **semMLib** - mutual exclusion semaphores
- ◆ **semSmLib** - shared memory semaphores

# 信号量的创建



- `SEM_ID semBCreate(int options, SEM_B_STATE initialState )`
- `SEM_ID semMCreate(int options)`
- `SEM_ID semCCreate(int options, int initialCount)`
- `SEM_ID semBSmCreate( int options, SEM_B_STATE initialState)`
- `SEM_ID semCSmCreate( int options, int initialCount)`

返回值：SEM\_ID



## ◆ options

- SEM\_Q\_PRIORITY (0x1)
- SEM\_Q\_FIFO (0x0)
- SEM\_DELETE\_SAFE (0x4) (互斥信号量)
- SEM\_INVERSION\_SAFE (0x8) (互斥信号量)
- SEM\_EVENTSEND\_ERR\_NOTIFY (0x10)

## ◆ initialState

- SEM\_FULL (1)
- SEM\_EMPTY (0)

## ◆ initialCount

## ◆ timeout

- WAIT\_FOREVER (-1)
- NO\_WAIT (0)
- *value*



# 基本函数



- ◆ [semGive\( \)](#) - give a semaphore
- ◆ [semTake\( \)](#) - take a semaphore
- ◆ [semFlush\( \)](#) - unblock every task pended on a semaphore
- ◆ [semDelete\( \)](#) - delete a semaphore
- ◆ [semMGiveForce\( \)](#) - give a mutual-exclusion semaphore without restrictions





- ◆ `SEM_ID semBCreate( options, initialState)`
  - Options 为阻塞在该信号量的任务规定排队的类型 (`SEM_Q_PRIORITY`或`SEM_Q_FIFO`)
  - initialState 初始化信号量为可用 (`SEM_FULL`) 或不可用 (`SEM_EMPTY`)
- ◆ 用于同步的信号量通常初始化为不可用 (`SEM_EMPTY`) (事件没有发生)
- ◆ 返回`SEM_ID`, 出错返回`NULL`

# 获得信号量的流程



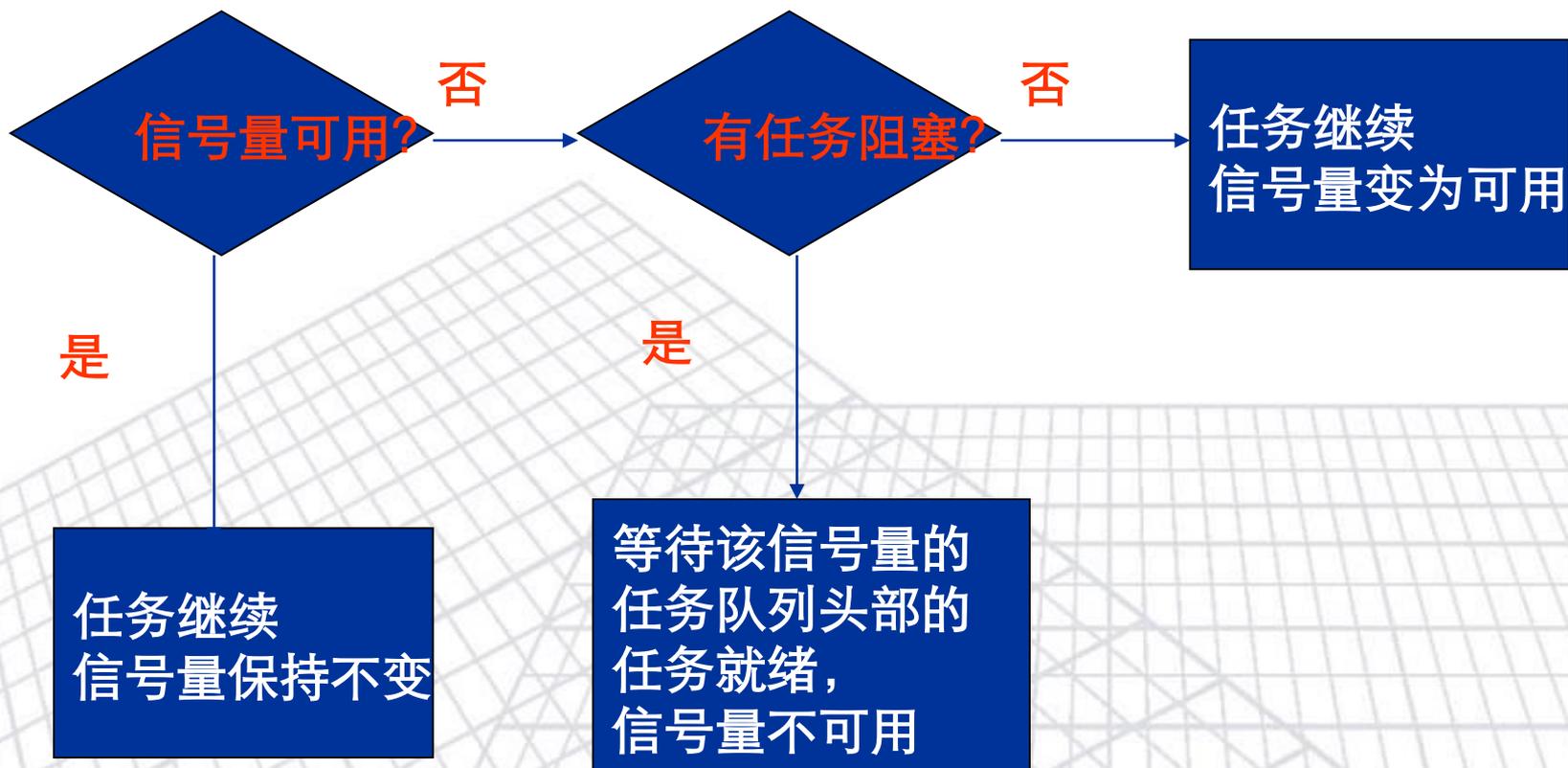
# 释放信号量



- ◆ STATUS semGive (semId)
- ◆ 解锁等待semId信号量的任务
- ◆ 如果没有任务等待，则将信号量变为可用
- ◆ 成功返回OK，如果semId无效返回ERROR



# 释放信号量的流程



# 多个任务之间的同步



- ◆ **STATUS semFlush (semId)**
- ◆ 解除阻塞所有等待该信号量的任务
- ◆ 不影响信号量的状态
- ◆ 对多个任务之间的同步非常有用



# 互斥问题



- ◆ 如果被多个任务同时操作，某些资源可能会变得不确定
  - 共享事件结构
  - 共享文件
  - 共享硬件设备
- ◆ 在使用这些资源时，必须获得对资源操作得唯一权限
- ◆ 如果没有获得对资源操作得唯一权限，那么任务执行得顺序将影响结果得正确性
- ◆ 互斥操作不能受优先级的影响

# 资源竞争实例



tTask1

tTask2

```
Char    myBuf [ BUF_SIZE ]; /*store data here*/
Int     myBufIndex = -1; /*index of last data*/
myBufPut ( char ch)
{
    myBufIndex++;
    myBuf [myBufIndex] = ch;
}
```

myBufIndex ++;  
myBuf[myBufIndex]='b';

myBufIndex ++;  
myBuf[myBufIndex]='a';





- ◆ 创建一个互斥信号量来保护资源
- ◆ 在对资源进行操作前调用 `semTake()` ; 操作完成后, 调用 `semGive()`
  - `semTake()` 将锁定, 直到信号量 (也就是被保护的资源) 成为可用
  - `semGive()` 释放信号量 (资源就可用了)

# 创建互斥信号量



◆ SEM\_ID semMCreate (options)

◆ Options可以是

- 排队队列的性质 SEM\_Q\_FIFO或SEM\_Q\_PRIORITY
- 安全删除 SEM\_DELETE\_SAFE
- 优先级继承 SEM\_IVERSION\_SAFE

◆ 信号量的初始状态为可用

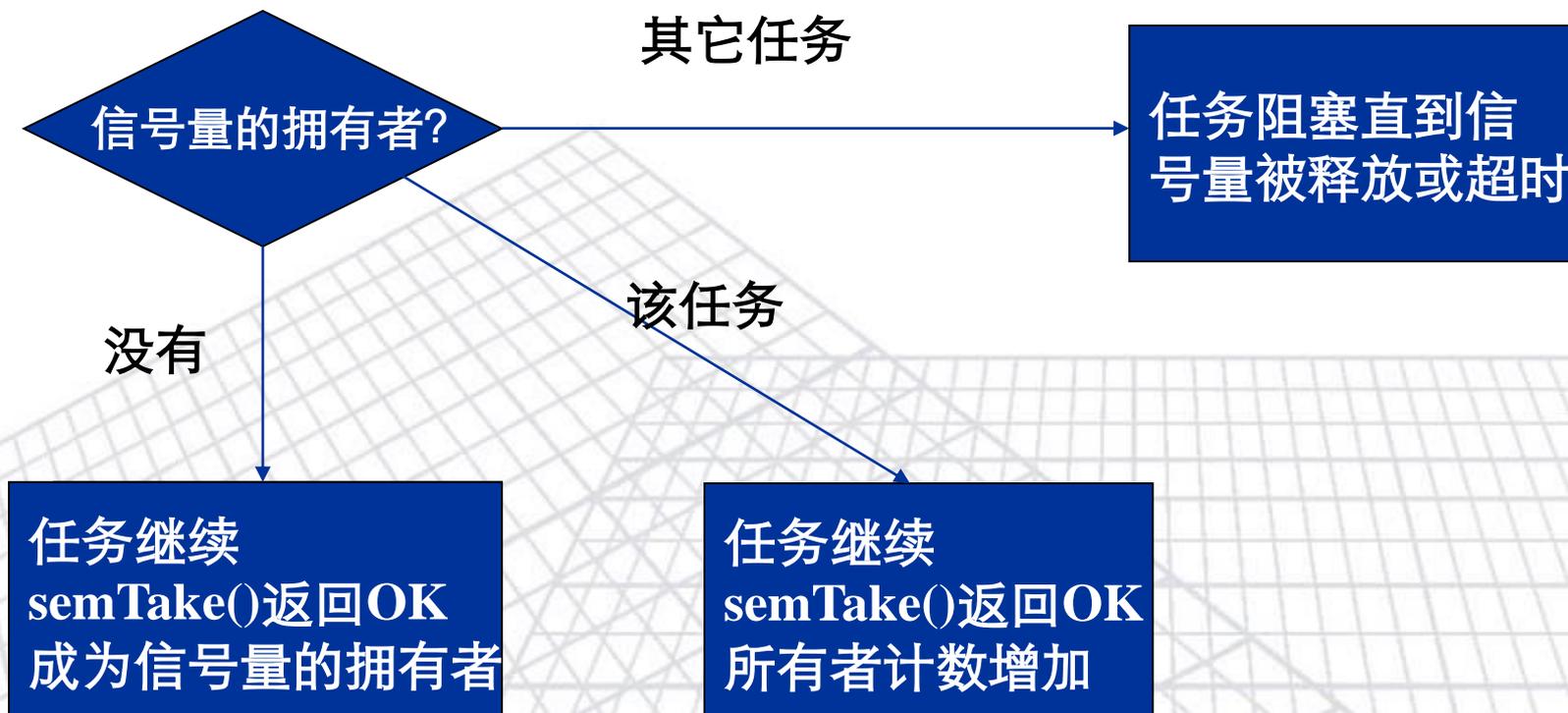




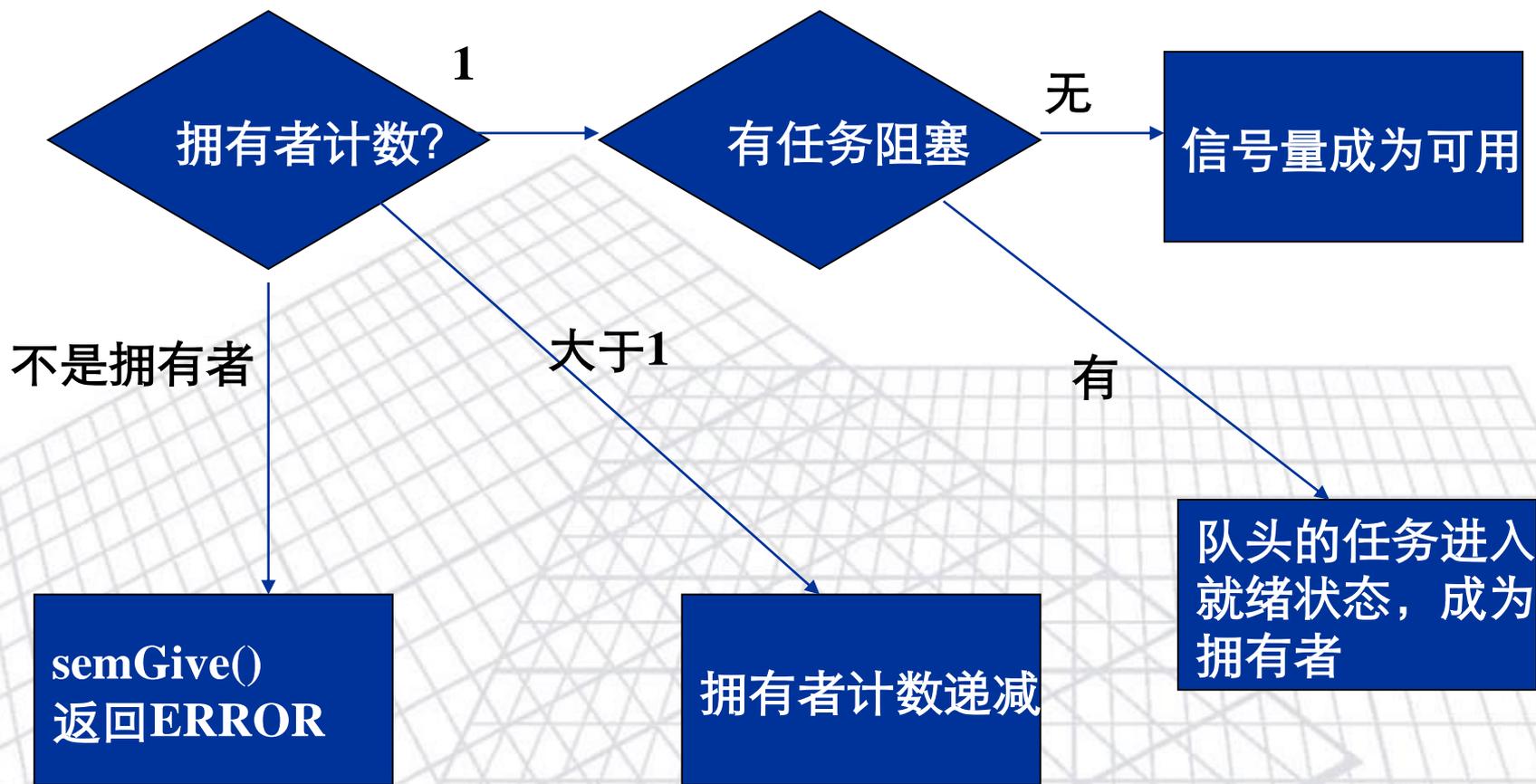
- ◆ 一个获得信号量的任务就拥有它，所以别的任务是不能释放它的
- ◆ 互斥信号量可以被递归获取
  - 拥有信号量的任务可以多次获取它
  - 一个被递归n次的信号量必须被释放与获取相同的n次，才能被真正释放。
- ◆ 中断服务例程中不能使用互斥信号量



# 获取互斥信号量



# 释放一个互斥信号量



# 编程举例



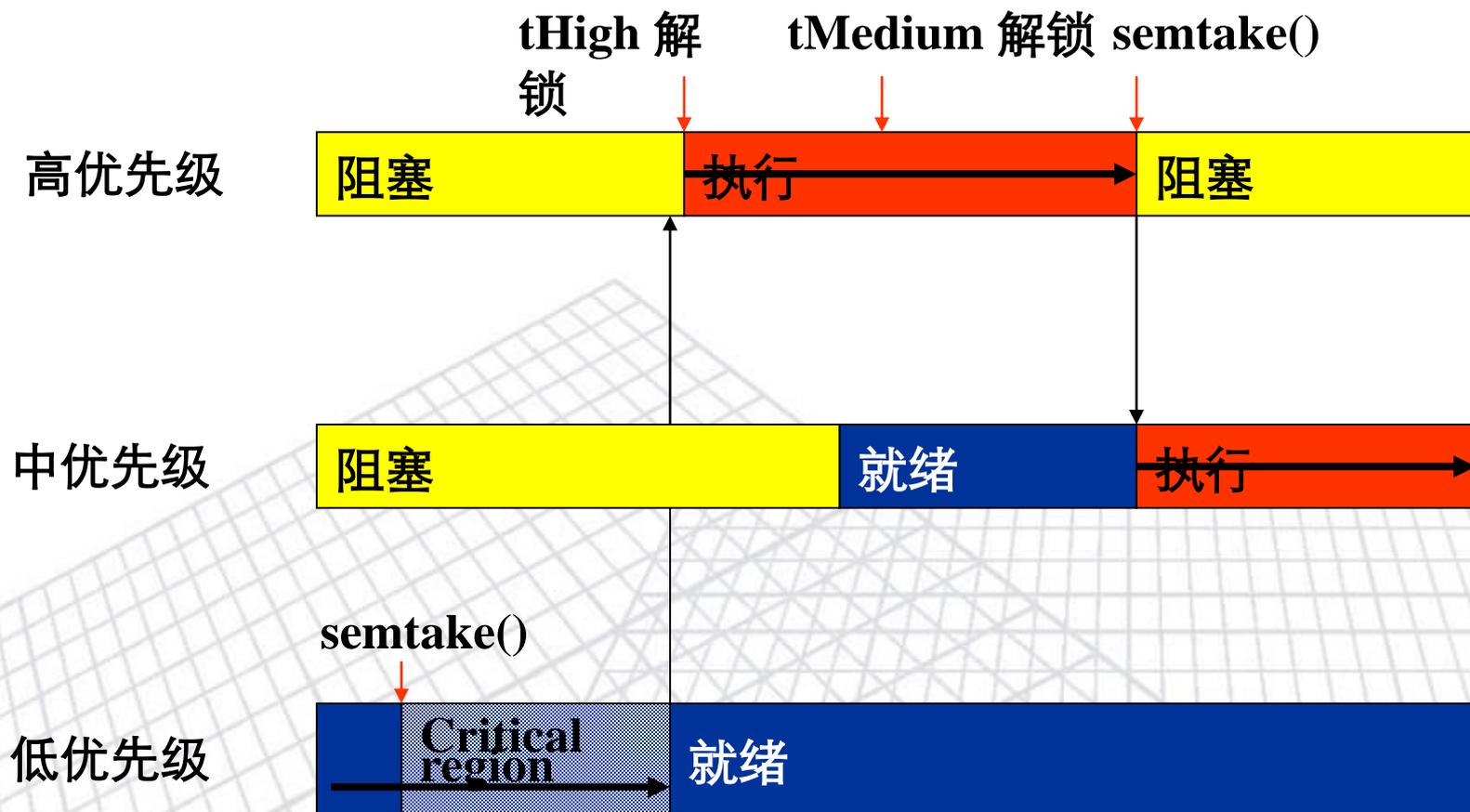
```
#include "vxWorks.h"
#include "semLib.h"
LOCAL char muBuf [BUF_SIZE];
LOCAL int myBufIndex = -1;
LOCAL SEM_ID mySemId;
Void myBufInit()
{ mySemId = semMCreate ( SEM_Q_PRIORITY | SEM_INVERSION_SAFE | SEM_DELETE_SAFE );
}
Void myBufPut(char ch)
{ semTake ( mySemId, WAIT_FOREVER);
  myBufIndex ++;
  myBuf [myBufIndex] = ch;
  semGive (mySemId );
}
```





- ◆ 删除一个拥有信号量的任务是非常危险的
  - 数据结构中的值会变得不可靠
  - 信号量变得永远不可用
- ◆ 删除安全选项防止一个拥有信号量的任务被删除
- ◆ 互斥信号量通过在信号量创建时设置 `SEM_DELETE_SAFE` 选项来使能删除安全

# 优先级倒置



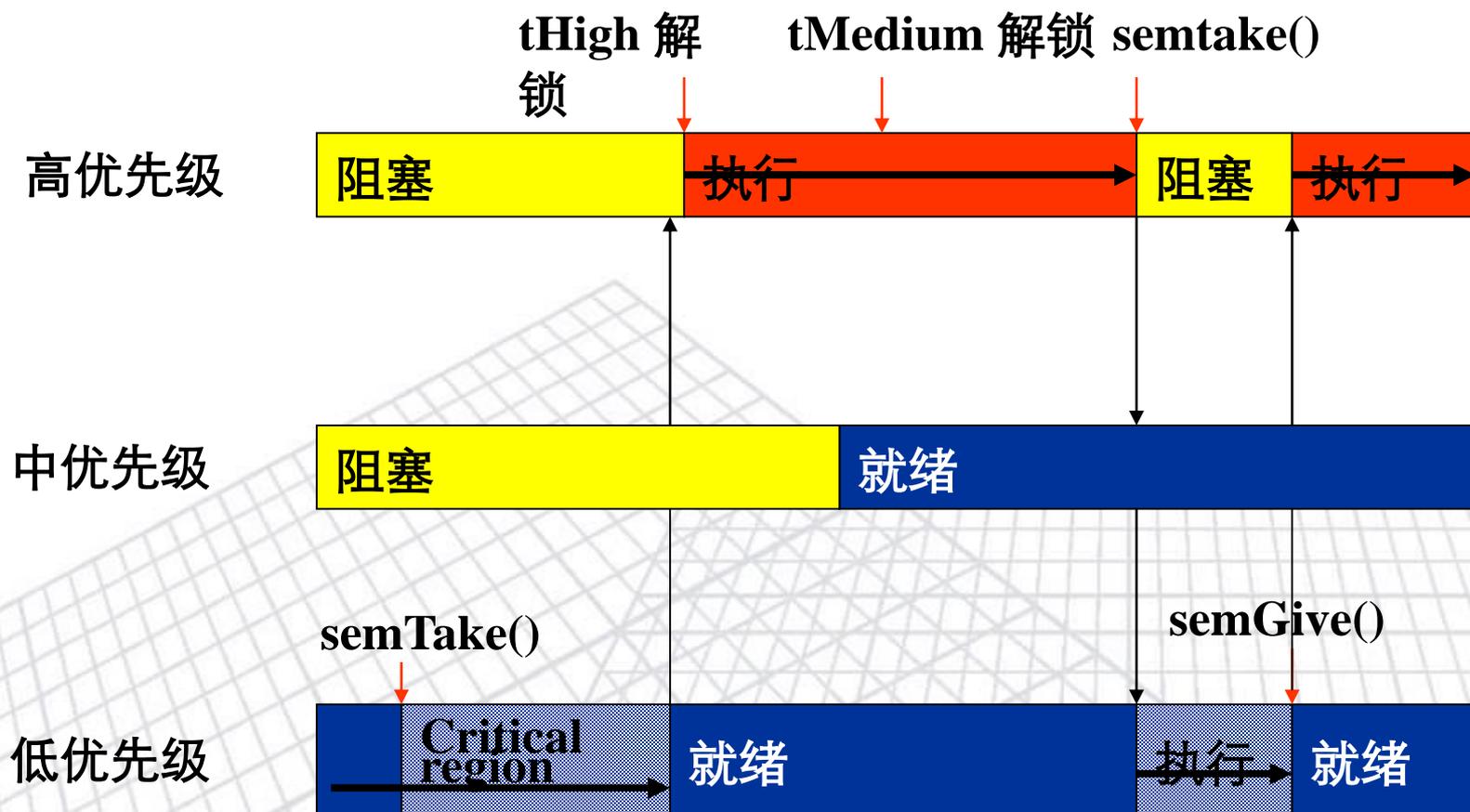
# 优先级继承



- ◆ 优先级继承解决了优先级倒置的问题
- ◆ 拥有该信号量的任务的优先级被提高到等待该信号量的所有任务中优先级最高的任务的优先级
- ◆ 优先级继承在信号量创建`semMCreate`时，通过设置`SEM_IVERSION_SAFE`来实现
- ◆ 同时还必须设置`SEM_Q_PRIORITY`



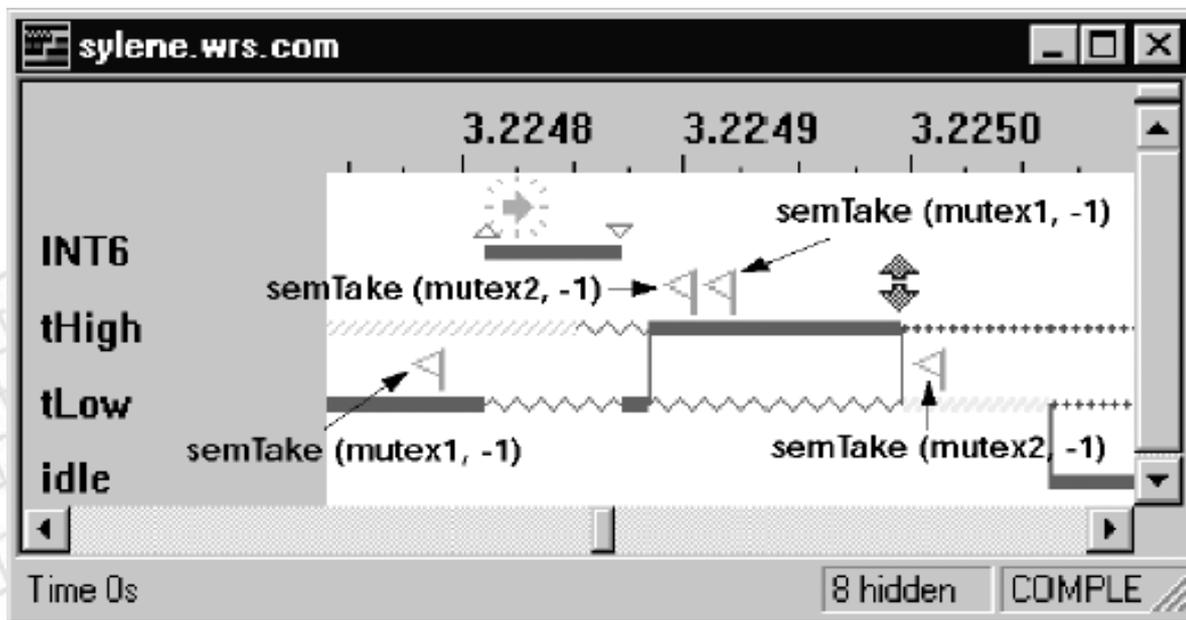
# 优先级继承



# 使用互斥信号量要注意的问题



- ◆ 死锁发生在获取对多个共享资源进行竞争
- ◆ 死锁难以检查



# 其它需要注意的问题



- ◆ 使用一个互斥信号量来保护要同时访问的多个资源。
- ◆ 对资源进行顺序操作，如在上例中所有的任务应该首先获取**mutex1**，然后在都获取信号量**mutex2**。
- ◆ 互斥信号量不能用于中断。
- ◆ 尽量保持**Critical Region**（**semTake()**和**semGive()**之间的程序）尽可能短。



# 计数器信号量



## ◆ 计数器信号量的执行实例：

- 用于保护多个资源副本。例如，3个可用设备可以用计数器为3的信号量来保护。保证设备同时可供3个不同用户使用。

---

### Semaphore Call    Count after Call    Resulting Behavior

---

<code>semCCreate( )</code>	3	Semaphore initialized with an initial count of 3.
<code>semTake( )</code>	2	Semaphore taken.
<code>semTake( )</code>	1	Semaphore taken.
<code>semTake( )</code>	0	Semaphore taken.
<code>semTake( )</code>	0	Task blocks waiting for semaphore to be available.
<code>semGive( )</code>	0	Task waiting is given semaphore.
<code>semGive( )</code>	1	No task waiting for semaphore; count incremented.

---



# 信号量事件



- ◆ 将任务注册到某个信号量上，在信号量变得可用时，向注册任务发送一个事件。
- ◆ 一个任务可以注册多个信号量；
- ◆ 一个信号量只能对应一个任务；
- ◆ 相关函数：
  - semEvStart():
  - semEvStop



# 信号量的公共函数



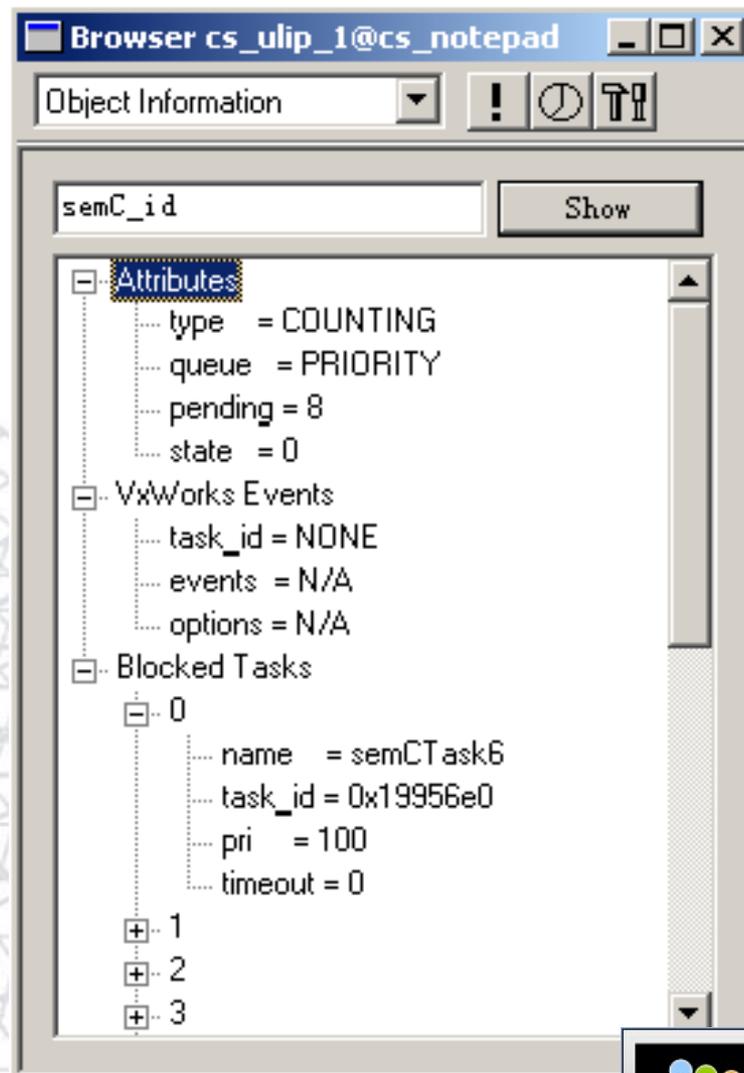
- ◆ **semDelete ()** : 删除信号量, 所有阻塞在该信号量的任务从**semTake ()** 上返回**ERROR**。
- ◆ **Show ()** : 显示信号量的信息。



# 信号量查看



- ◆ 如果想要观察一个指定信号量的信息，将信号量的ID号输入到Browser的Show方框内，并点击show按钮。



# 中断和互斥信号量



- ◆ **ISR**不能使用互斥信号量
- ◆ 如果任务和中断共享某种资源，就需要禁止该中断
- ◆ 可以使用**intLock()**和**intUnlock()**来禁止和恢复中断
- ◆ 使禁止中断的时间尽量短
- ◆ 在任务级调用内核函数会恢复中断





## ▪ POSIX semPxlLib library

– #include "semaphore.h"

- **semPxlLibInit( )** Initializes the POSIX semaphore library (non-POSIX).
- **sem\_init( )** Initializes an unnamed semaphore.A\_B
- **sem\_destroy( )** Destroys an unnamed semaphore.
- **sem\_open( )** Initializes/opens a named semaphore.
- **sem\_close( )** Closes a named semaphore.
- **sem\_unlink( )** Removes a named semaphore.
- **sem\_wait( )** Lock a semaphore.
- **sem\_trywait( )** Lock a semaphore only if it is not already locked.
- **sem\_post( )**Unlock a semaphore.
- **sem\_getvalue( )**Get the value of a semaphore



# 总结1



二进制信号量

`semBCreate()`

互斥信号量

`semMCreate()`

计数器信号量

`semMCreate()`

`semTake(), semGive(),  
show()`

`semDelete()`



- ◆ 互斥信号量用于获得对共享资源得唯一访问权
  - 为要保护得资源创建一个互斥信号量
  - 在对资源操作前，调用semTake（）
  - 为了释放资源，调用semGive（）
- ◆ 互斥信号量具有所有者
- ◆ 注意事项
  - 保持critical region尽量短
  - 通过函数库来进行对资源的操作
  - 不能在中断中使用
  - 会产生死锁



# 4. 消息队列



- ◆ 多任务系统需要在任务之间进行通信
- ◆ 任务间的通信由三个部分组成
  - 共享的数据或信息
  - 通知任务这些共享的数据可以读或写
  - 防止任务之间相互干扰的机制（如果有两个任务要写共享内存）
- ◆ 任务间的通信通常使用两种机制
  - 共享内存
  - 消息传递



# 消息传送队列



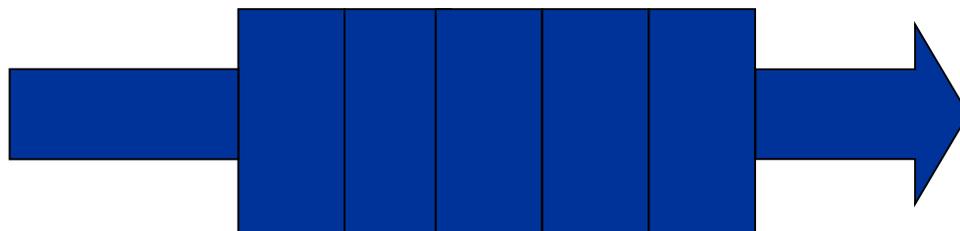
- ◆ VxWorks的管道和消息队列用于在任务之间传递消息
- ◆ 管道和消息队列都提供
  - 一个消息的FIFO缓存
  - 同步
  - 互斥
- ◆ 比共享数据更健壮
- ◆ 可以用于任务之间，也可以用于ISR和任务间的通信

# 消息队列



- ◆消息队列的创建
- ◆发送和接收消息
- ◆消息队列的应用





- ◆ 用于在一个**CPU**上任务间的通信
- ◆ 消息长度可变的先进先出（**FIFO**）的缓存
- ◆ 内含有任务控制
  - 互斥
  - 同步

# 消息队列库



- ◆ Wind消息队列库
  - msgQLib.h
- ◆ POSIX消息队列库
  - **mqPxLib**库
  - mqueue.h

# Wind消息队列函数



- ◆ **msgQCreate**( ) - create and initialize a message queue
- ◆ **msgQDelete**( ) - delete a message queue
- ◆ **msgQSend**( ) - send a message to a message queue
- ◆ **msgQReceive**( ) - receive a message from a message queue
- ◆ **msgQNumMsgs**( ) - get the number of messages queued to a message queue

返回值：MSG\_Q\_ID

# POSIX消息队列函数



- ◆ [mqPxLibInit](#)( ) - initialize the POSIX message queue library
- ◆ [mq\\_open](#)( ) - open a message queue (POSIX)
- ◆ [mq\\_receive](#)( ) - receive a message from a message queue (POSIX)
- ◆ [mq\\_send](#)( ) - send a message to a message queue (POSIX)
- ◆ [mq\\_close](#)( ) - close a message queue (POSIX)
- ◆ [mq\\_unlink](#)( ) - remove a message queue (POSIX)
- ◆ [mq\\_notify](#)( ) - notify a task that a message is available on a queue (POSIX)
- ◆ [mq\\_setattr](#)( ) - set message queue attributes (POSIX)
- ◆ [mq\\_getattr](#)( ) - get message queue attributes (POSIX)

# Wind vs POSIX msgQ



特性	Wind msgQ	POSIX msgQ
消息优先级	1	32
阻塞方式	FIFO or Priority	Priority
接收超时	可选	不可选
队列属性设置	无	有
关闭/Unlink	无	有
消息ID	变量	变量 or 字符串



# 创建一个消息队列



◆ `MSG_Q_ID msgQCreate (maxMsgs, maxMsgLength, options)`

*maxMsgs* 能够排队的最大消息数

*maxMsgLength* 消息的最大字节数

*Options* 消息队列的类型 (`MSG_Q_FIFO`, `MSG_Q_PRIORITY`)

◆ 返回指向消息队列的id号，如果出错，则返回 `NULL`





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

*msgQId* 由msgQCreate返回的MSG\_Q\_ID

*Buffer* 放到队列上的数据的地址

*nBytes* 放到队列上的字节数

*Timeout* 最长的等待时间（如果队列满），可以是tick数，或WAIT\_FOREVE, NO\_WAIT

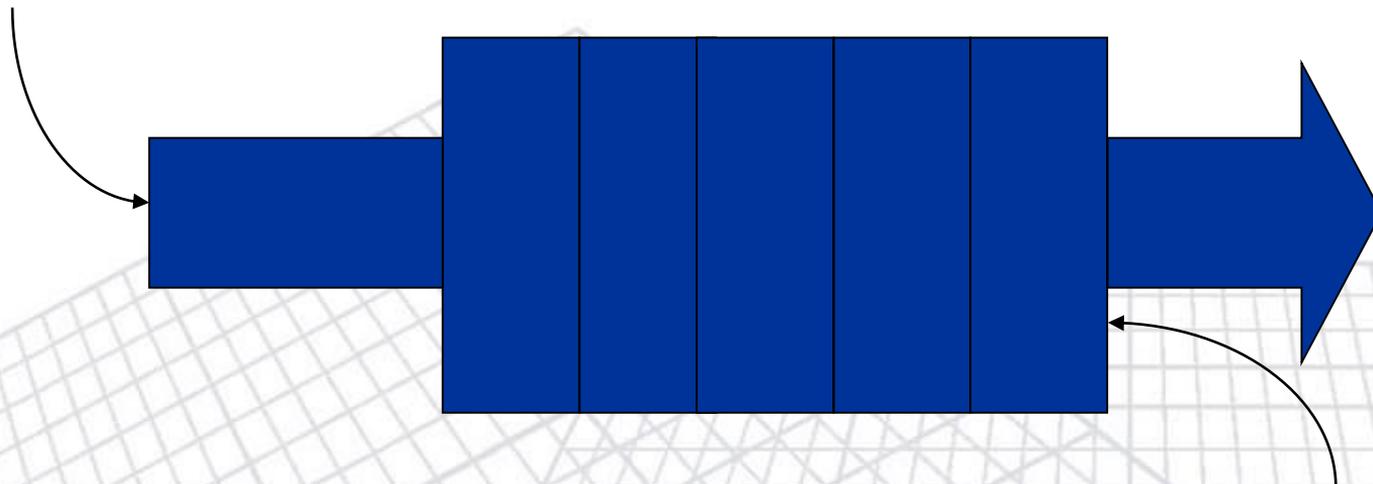
*Priority* 放到队列中的消息的优先级。如果是MSG\_PRI\_URGENT（1），则将消息放到队列的头部，如果是MSG\_PRI\_NORMAL（0），放到队列尾。

# 消息发送举例



```
Char    buf[BUFSIZE];
```

```
STATUS msgQsend ( msgQId, buf, sizeof(buf), WAIT_FOREVER,  
MSG_PRI_NORMAL)
```



```
Char    buf[BUFSIZE];
```

```
STATUS msgQsend ( msgQId, buf, sizeof(buf), NO_WAIT,  
MSG_PRI_URGENT)
```





- ◆ `int msgQReceive (msgQId, buffer, maxNBytes, timeout)`
  - msgQId* 由msgQCreate返回的MSG\_Q\_ID
  - Buffer* 存放数据的地址
  - maxNBytes* 从队列中读到的消息的最大长度
  - Timeout* 最长的等待时间（如果没有可用的消息），可以是tick数，或WAIT\_FOREVE, NO\_WAIT
- ◆ 成功时返回成功读取的字节数，超时或msgQId不正确，返回ERROR
- ◆ 消息中没有读的字节被扔掉

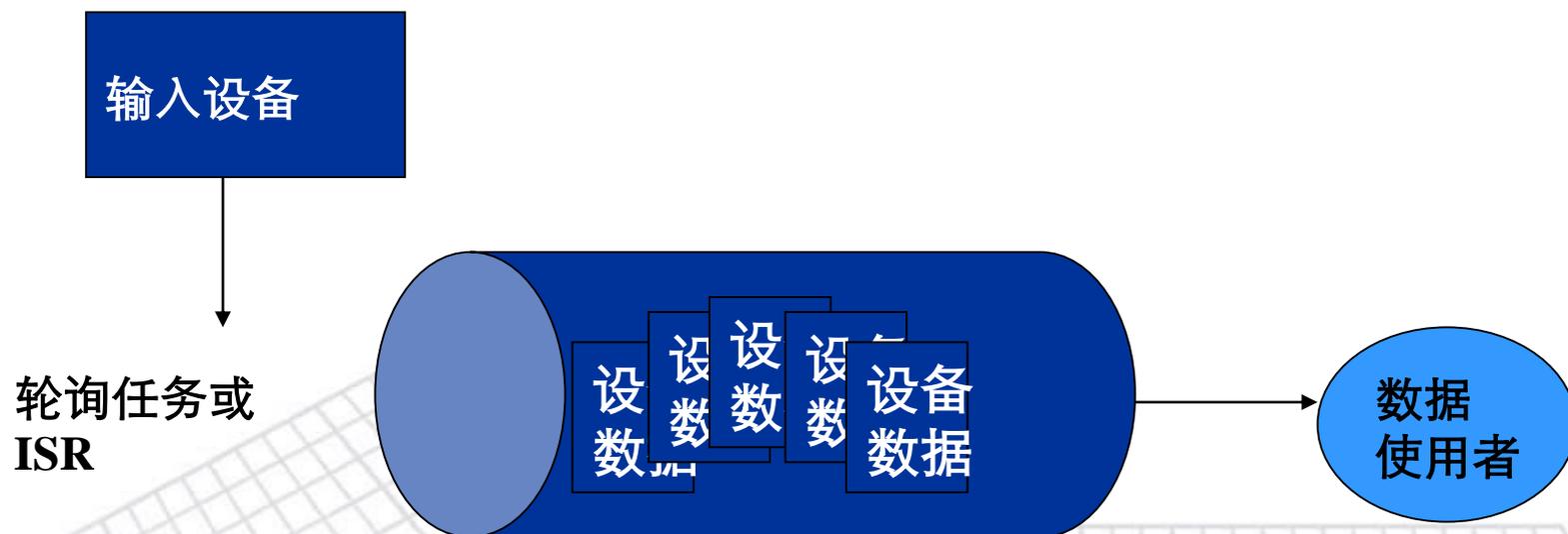
# 删除消息



- ◆ **STATUS msgQDelete (msgQId)**
- ◆ 删除消息队列。
- ◆ 被该队列所阻塞的任务被解锁，它们对该消息ID所进行的msgQCreate和msgQSend的调用，将返回**ERROR**，**errno**为 **S\_objLib\_OBJ\_DELETED**。



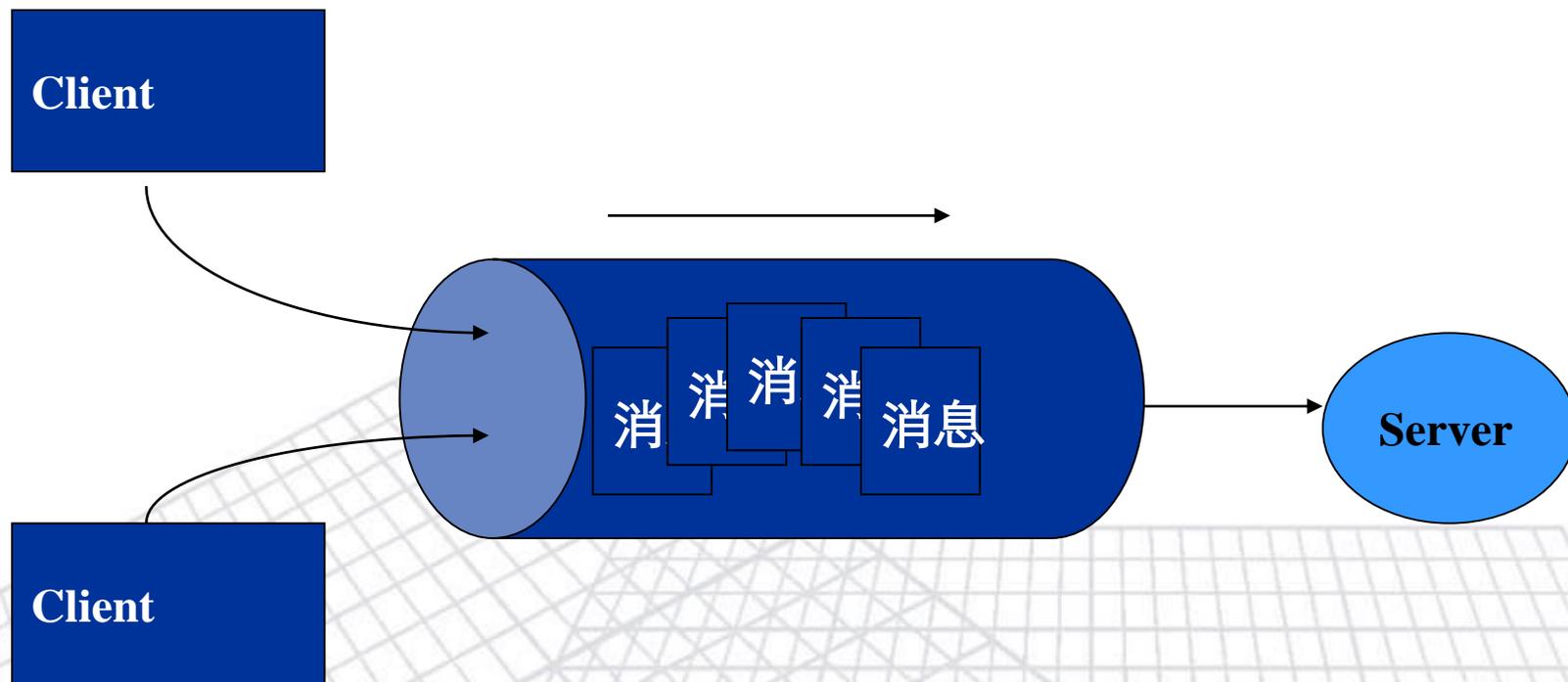
# 用消息队列得到数据



- ◆ 为了很快地获得数据供以后使用
  - 让一个轮询任务或ISR将数据放入消息队列
  - 让低优先级地任务能够从消息队列中读出数据进行处理



# 客户—服务器模型 (Client/Server)

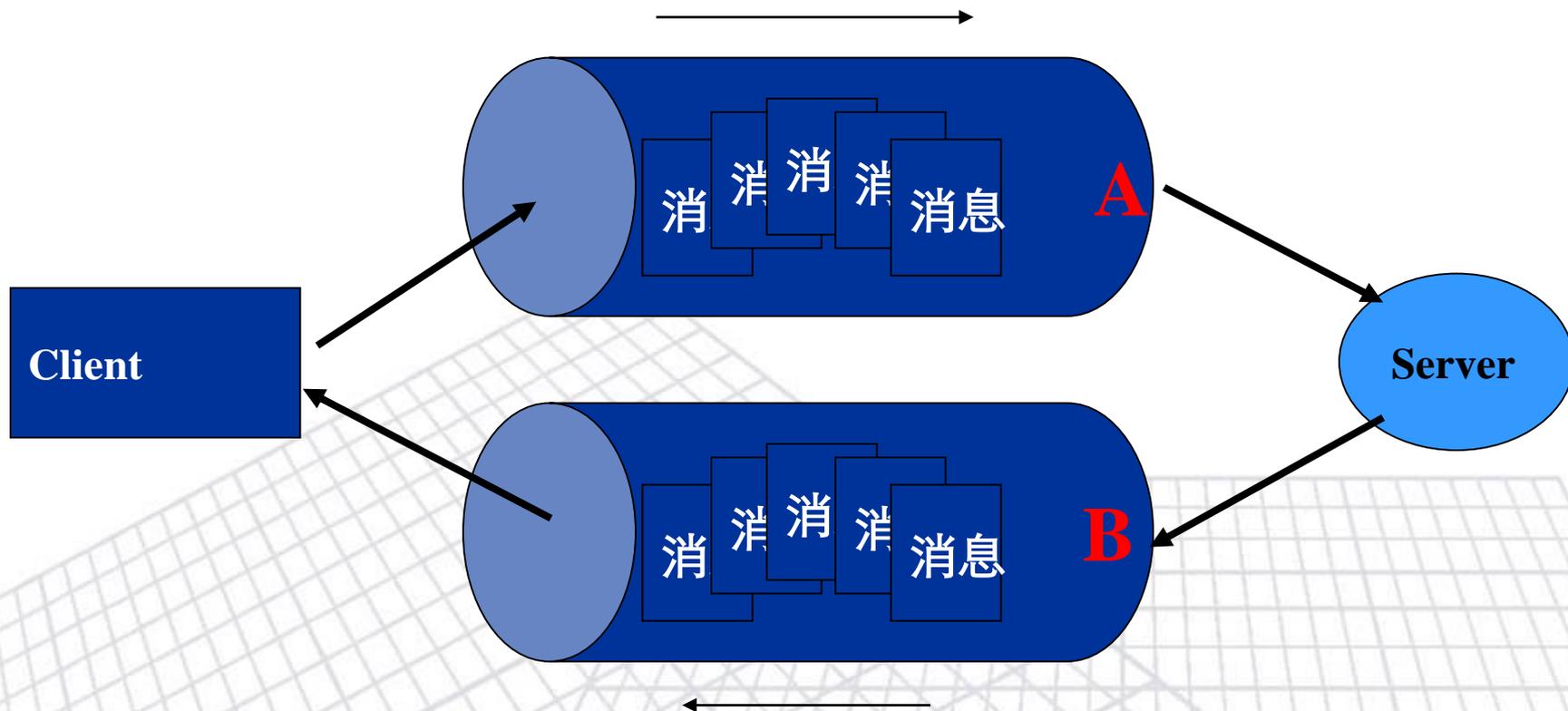


## ◆ 客户—服务器模型

- 客户向服务器任务发出请求 (将请求放入消息队列)
- 服务器任务对这些请求提供服务
- 有些服务器还对请求做出响应



# 使用消息队列的双工通信



## ◆ 双工通信模型

- 客户向服务器任务发出请求（将请求放入消息队列A）
- 服务器任务对这些请求提供服务
- 服务器对请求做出响应，回复给客户（将回复放入消息队列B）



# 5. 管道



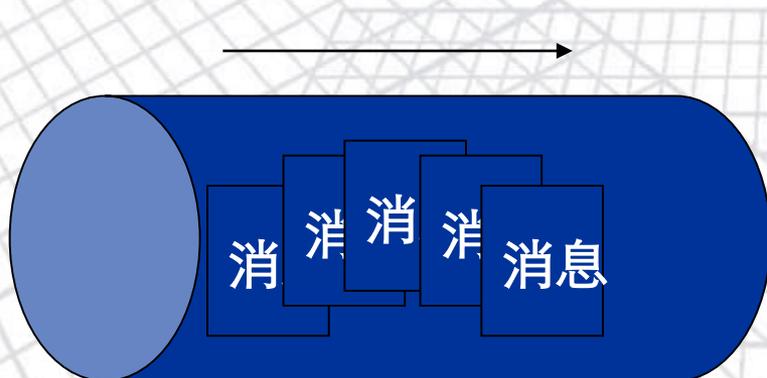
- ◆ 创建一个管道 (pipe)
- ◆ 从管道中读
- ◆ 向管道中写



# 管道



- ◆是由pipeDrv所控制的一个虚拟的I/O设备。
- ◆构筑于消息队列之上
- ◆具有标准的I/O接口（读和写）
- ◆和UNIX中的具名管道相似



# 创建一个管道



## ◆ STATUS pipeDevCreate (name, nMessages, nBytes)

*Name*      管道设备的名称，通常使用 “/pipe/yourName”

*nMessages*      管道中能够排列的最大消息数

*nBytes*      每个消息的最长字节数

## ◆ 创建成功返回OK，否则返回ERROR

# 管道创建的例子



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

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

```
->devs
```

drv	name
0	/null
1	tyCO/0
1	tyCO/1
4	Columbia:
2	/pipe/myPipe



# 管道的读和写



- ◆ 为了能够进入一个已存在的管道，首先使用 `open()` 将它打开
- ◆ 从管道中读，使用 `read ()`
- ◆ 向管道中写，使用 `write ()`

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



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



# 消息队列和管道的对比



## ◆消息队列的优点

- 具有超时功能
- 消息有优先级
- 更快
- 可以被删除

## ◆管道的优点

- 使用标准的I/O接口，如`read()`，`write()`，`open()`，`close()`等
- 可以通过`io taskStdSet ()`重定向
- 在`select ()`中可以使用文件描述符进行监听

