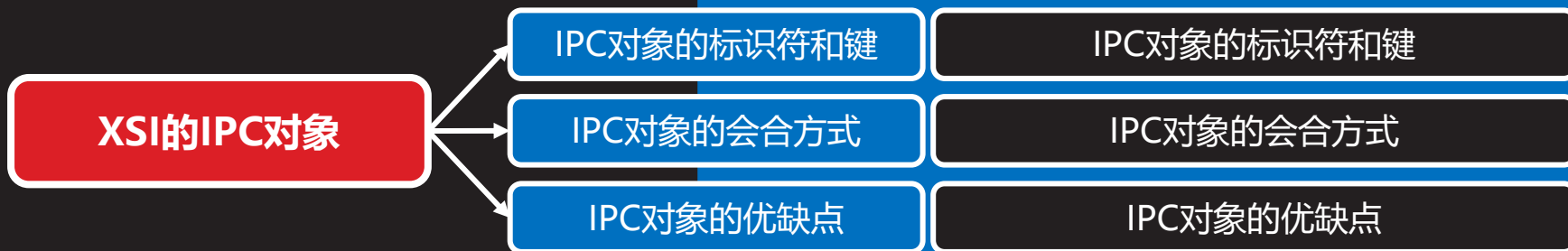


Unix系统高级编程

XSI的IPC对象

Unit22

XSI的IPC对象



IPC对象的标识符和键



IPC对象的标识符和键

- 共享内存、消息队列和信号量，这三种IPC一般被合称为XSI IPC，它们之间有很多相似之处
- XSI IPC源自System V的IPC功能，而后者又源于1970年一个名为Columbus UNIX的AT&T内部版本
- XSI IPC没有使用文件系统名字空间，而是构造了它们自己的名字空间，时至今日仍为此而饱受诟病
- 为了实现进程之间的数据交换，系统内核会为参与通信的诸方维护一个内核对象(类似一个结构体变量)，记录和通信有关的各种配置参数和运行时信息，谓之IPC对象
- 系统中的每个IPC对象都有唯一的，非负整数形式的标识符，所有与IPC相关的操作，都需要提供IPC对象标识符



IPC对象的标识符和键（续1）

- 与文件描述符不同，IPC对象标识符不是个小整数。当一个IPC对象被创建，以后又被销毁时，与该类型对象相关的标识符会持续加1，直至达到一个整型数的最大正值，然后又回转到0
- 标识符是IPC对象的内部名。为了使多个合作进程能够在同一个IPC对象上会合，需要提供一个外部名方案。为此使用了键，每个IPC对象都与一个键相关联，于是键就被作为该对象的外部名
- 无论何时，创建或者获取一个IPC对象都必须指定一个键。键的数据类型为key_t，在<sys/types.h>头文件中被定义为int。系统内核负责维护键与标识符的对应关系



IPC对象的会合方式



IPC对象的会合方式

- 服务器进程可以用IPC_PRIVATE宏(通常被定义为0)作为键创建一个新的IPC对象, 将返回的标识符放在某处, 例如一个文件中, 以方便客户机取用。IPC_PRIVATE宏可以保证所得到的IPC对象一定是新建的, 而不是现有的
 - IPC对象标识符可以被fork函数产生的子进程直接引用, 也可以作为命令行参数或者环境变量的一部分被exec函数传递给新创建的进程, 这样就避免了读写文件之类的开销
- 将键作为宏或者外部变量定义在一个公共头文件中。服务器和客户机都包含该头文件, 服务器用这个键创建IPC对象, 而客户机用这个键获取服务器所创建的IPC对象
 - 键可能已经与某个现有的IPC对象结合, 服务器在创建IPC对象时必须处理这种情况, 比如删除现有的对象后再重试



IPC对象的会合方式 (续1)

- 服务器和客户机用一对约定好的路径和项目ID(0-255), 通过ftok函数合成一个键, 用于创建和获取IPC对象

```
#include <sys/ipc.h>
```

```
key_t ftok (const char* pathname, int proj_id);
```

成功返回可用于创建或获取IPC对象的键, 失败返回-1

- *pathname*: 一个真实存在的路径名
- *proj_id*: 项目ID, 仅低8位有效, 取0到255之间的数



IPC对象的会合方式 (续2)

- `ftok`函数用`pathname`参数调用`stat`函数, 将其输出的`stat`结构体中的`st_dev`(设备ID)和`st_ino`(i节点号)成员与`proj_id`参数组合来生成键
 - 参与生成键的是设备ID和i节点号, 而不是`pathname`参数字符串本身。假设当前路径为`/home/tarena/unixc`, 则`ftok ("/home/tarena/unixc", 123);`
与
`ftok (".", 123);`
所返回的键是完全一样的
 - 设备ID和i节点号都至少是整型字长的数据, 而键也是整型字长, 再加上一个字节项目ID, 在合成键的过程中难免会丢失一部分信息。因此有时候明明提供的是不同的路径, 该函数返回的键却是一样的



IPC对象的优缺点

IPC对象的优缺点

- IPC对象的优点是不言而喻的
 - 共享内存可以在不将数据往复拷贝于参与通信的各个进程之间的前提下，令这些进程通过各自的虚拟内存地址访问相同的物理内存单元
 - 消息队列提供了一种可在进程之间以可靠的、带有流量控制的、面向记录并支持按类型过滤的方式传输数据的途径
 - 信号量很好地解决了由无限的用户分享有限的资源所带来的竞争和冲突问题

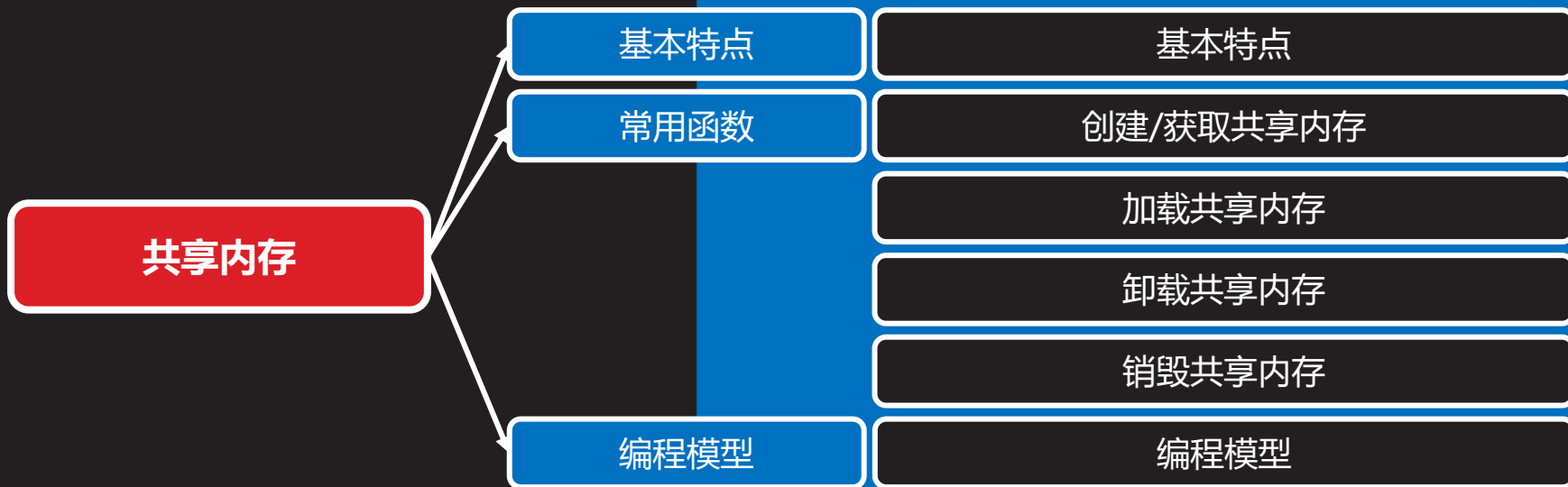


IPC对象的优缺点（续1）

- IPC对象的缺点同样不容忽视
 - IPC对象都是系统级而非进程级的对象，而且也没有所谓的引用计数。即使所有曾使用它们的进程都终止了，只要不通过函数或者命令显式地销毁它们，它们连同它们所缓存的数据将永远存留于系统中，直至关机或重启
 - IPC对象在文件系统中没有名字，不能用ls查看，也不能用rm删除，更不能用chmod修改其权限，因此它们需要通过一套专门的命令，如ipcs、ipcrm等来操作
 - IPC对象都没有文件描述符，因此一切针对文件描述符的系统调用，如open、close、read、write，甚至select和poll等等，对IPC对象都无能为力



共享内存



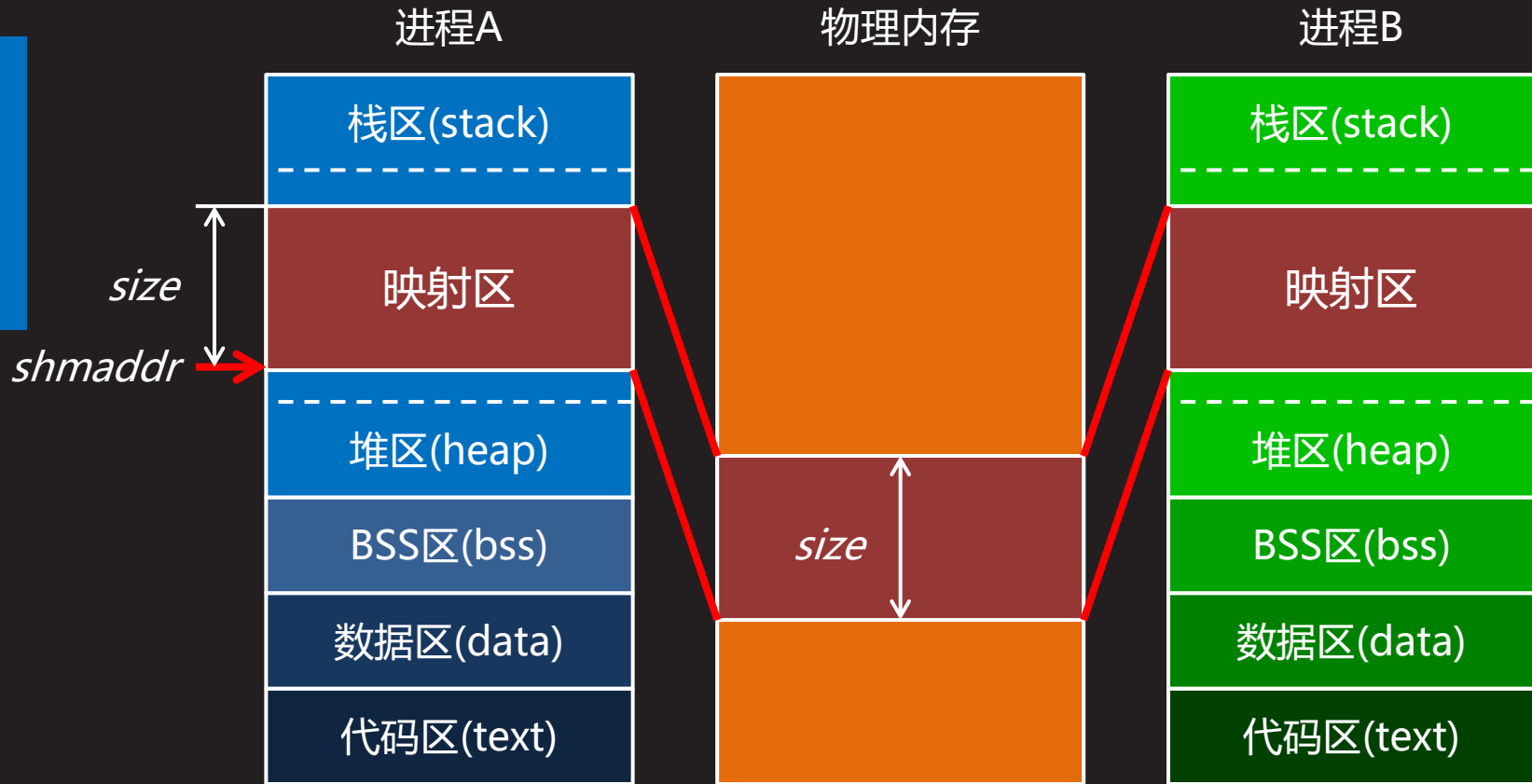
基本特点



基本特点

- 两个或者更多进程，共享同一块由系统内核负责维护的内存区域，其地址空间通常被映射到堆和栈之间

知识讲解



基本特点 (续1)

- 多个进程通过共享内存通信，所传输的数据通过各个进程的虚拟内存被直接反映到同一块物理内存中，这就避免了在不同进程之间来回复制数据的开销。因此，基于共享内存的进程间通信，是速度最快的进程间通信方式
- 共享内存本身缺乏足够的同步机制，这就需要程序员编写额外的代码来实现。例如服务器进程正在把数据写入共享内存，在这个写入过程完成之前，客户机进程就不能读取该共享内存中的数据。为了建立进程之间的这种同步，可能需要借助于其它的进程间通信机制，如信号或者信号量等，甚至文件锁，而这无疑会增加系统开销



常用函数



创建/获取共享内存

- 创建新的或获取已有的共享内存

```
#include <sys/shm.h>
```

```
int shmget (key_t key, size_t size, int shmflg);
```

成功返回共享内存标识符，失败返回-1

- **key**: 共享内存键
- **size**: 共享内存大小(以字节为单位)，自动向上圆整至页(4096)的整数倍。若欲创建新的共享内存，必须指定**size**参数；若只为获取已有的共享内存，**size**参数可取0



创建/获取共享内存 (续1)

- 创建新的或获取已有的共享内存
 - *shmflg*: 创建标志, 可取以下值
 - 0 - 获取, 不存在即失败
 - IPC_CREAT - 创建, 不存在即创建, 已存在即获取
 - IPC_EXCL - 排斥, 已存在即失败

- 例如

- int shmid = shmget (key, 4096,
0644 | IPC_CREAT | IPC_EXCL);
if (shmid == -1) {
 perror ("shmget"); exit (EXIT_FAILURE); }
- int shmid = shmget (key, 0, 0);
if (shmid == -1) {
 perror ("shmget"); exit (EXIT_FAILURE); }

加载共享内存

- 加载共享内存

```
#include <sys/shm.h>
```

```
void *shmat (int shmid, const void* shmaddr,  
            int shmflg);
```

成功返回共享内存起始地址，失败返回-1

- *shmid*: 共享内存标识符
- *shmaddr*: 指定映射地址，可置NULL，由系统自动选择



加载共享内存 (续1)

- 加载共享内存
 - *shmflg*: 加载标志, 可取以下值
 - 0 - 以读写方式使用共享内存
 - SHM_RDONLY - 以只读方式使用共享内存
 - SHM_RND - 只在*shmaddr*参数非NULL时起作用, 表示对该参数自动向下圆整至页(4096)的整数倍
- *shmat*函数负责将给定共享内存映射到调用进程的虚拟内存空间, 返回映射区的起始地址, 同时将系统内核中共享内存对象的加载计数加一
- 调用进程在获得*shmat*函数返回的共享内存起始地址以后, 就可以象访问普通内存一样访问共享内存中的数据



加载共享内存 (续2)

- 例如

- `void* shmaddr = shmat (shmaddr, NULL, 0);`
if (`shmaddr == (void*)-1`) {
 `perror ("shmat"); exit (EXIT_FAILURE); }`
 `strcpy (shmaddr, "Hello, Shared Memory !");`
- `void* shmaddr = shmat (shmaddr, NULL, 0);`
if (`shmaddr == (void*)-1`) {
 `perror ("shmat"); exit (EXIT_FAILURE); }`
 `printf ("%s\n", (char*)shmaddr);`



卸载共享内存

- 卸载共享内存

```
#include <sys/shm.h>
```

```
int shmdt (const void* shmaddr);
```

成功返回0，失败返回-1

- *shmaddr*: 共享内存起始地址
- shmdt函数负责从调用进程的虚拟内存中解除*shmaddr*所指向的映射区到共享内存的映射，同时将系统内核中共享内存对象的加载计数减一
- 例如
 - if (**shmdt** (shmaddr) == -1) {
 perror ("shmdt"); exit (EXIT_FAILURE); }



销毁共享内存

- 销毁共享内存

```
#include <sys/shm.h>
```

```
int shmctl (int shmid, int cmd, struct shm_id* buf);
```

成功返回0，失败返回-1

- *shmid*: 共享内存标识符
- *cmd*: 控制命令，取IPC_RMID
其实并非真的销毁，而只是做一个销毁标记，禁止任何进程对该共享内存形成新的加载，但已有的加载依然保留。只有当其使用者们纷纷卸载，直至其加载计数降为零时，共享内存才会真的被销毁
- *buf*: 置NULL即可



编程模型



编程模型

- 基于共享内存实现进程间通信的编程模型

步骤	进程A	函数	进程B	步骤
1	创建共享内存	shmget	获取共享内存	1
2	加载共享内存	shmat	加载共享内存	2
3	使用共享内存	strcpy printf ...	使用共享内存	3
4	卸载共享内存	shmdt	卸载共享内存	4
5	销毁共享内存	shmctl	——	——



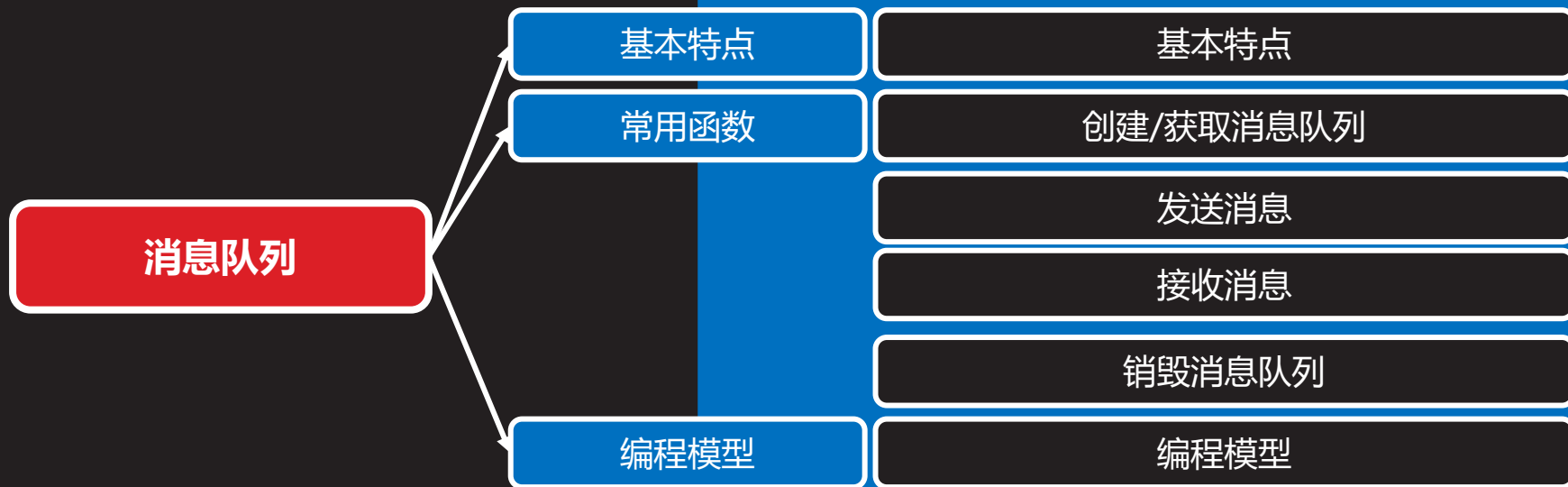
基于共享内存的进程间通信

【参见：wshm.c、rshm.c】

- 基于共享内存的进程间通信



消息队列

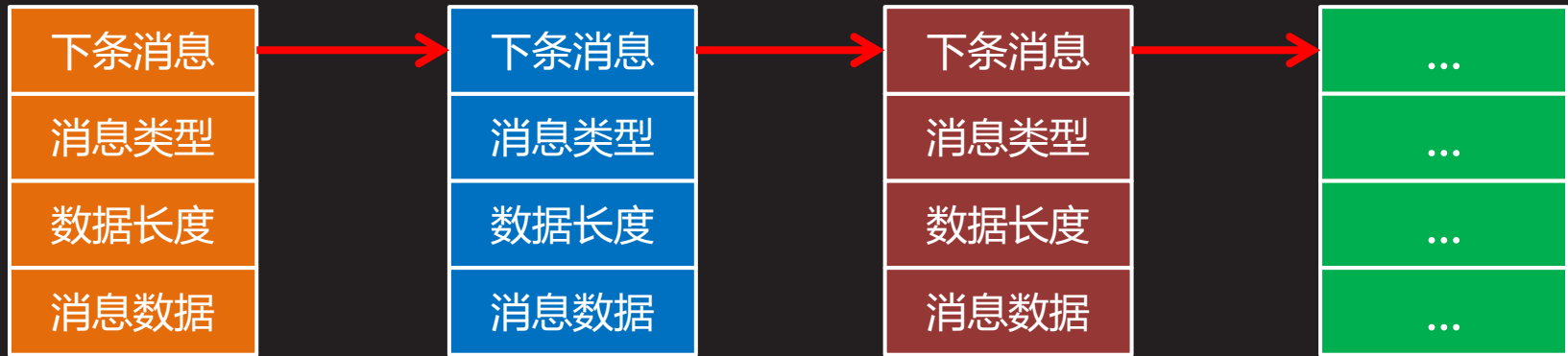


基本特点



基本特点

- 消息队列是一个由系统内核负责存储和管理，并通过消息队列标识符引用的消息链表



- 可以通过msgget函数创建一个新的消息队列或获取一个已有的消息队列。可以通过msgsnd函数向消息队列的尾端追加消息，所追加的消息除了包含消息数据以外，还包含消息类型和数据长度(以字节为单位)。可以通过msgrcv函数从消息队列中提取消息，但不一定非按先进先出的顺序提取，也可以按消息的类型提取

基本特点 (续1)

- 相较于其它几种IPC机制，消息队列具有明显的优势
 - 流量控制
 - 如果系统资源(内存)短缺或者接收消息的进程来不及处理更多的消息，则发送消息的进程会在系统内核的控制下进入睡眠状态，待条件满足后再被内核唤醒，继续之前的发送过程
 - 面向记录
 - 每个消息都是完整的信息单元，发送端是一个消息一个消息地发，接收端也是一个消息一个消息地收，而不象管道那样收发两端所面对的都是字节流，彼此间没有结构上的一致性
 - 类型过滤
 - 先进先出是队列的固有特征，但消息队列还支持按类型提取消息的做法，这就比严格先进先出的管道具有更大的灵活性
 - 天然同步
 - 消息队列本身就具备同步机制，空队列不可读，满队列不可写，不发则不收，无需象共享内存那样编写额外的同步代码



基本特点 (续2)

- 不同系统对消息队列的限制是不一样的，以Linux为例
 - 单个消息最大字节数：8192
 - 单个队列最大字节数：16384
 - 系统最大消息队列数：16
 - 系统最大消息字节数：262144 ($=16 \times 16384$)



常用函数



创建/获取消息队列

- 创建新的或获取已有的消息队列

```
#include <sys/msg.h>
```

```
int msgget (key_t key, int msgflg);
```

成功返回消息队列标识符，失败返回-1

- **key**: 消息队列键
- **msgflg**: 创建标志，可取以下值
 - 0 - 获取，不存在即失败
 - IPC_CREAT - 创建，不存在即创建，已存在即获取
 - IPC_EXCL - 排斥，已存在即失败



创建/获取消息队列 (续1)

- 例如
 - `int msgid = msgget (key, 0644 | IPC_CREAT | IPC_EXCL);`
`if (msgid == -1) {`
`perror ("msgget");`
`exit (EXIT_FAILURE);`
`}`
 - `int msgid = msgget (key, 0);`
`if (msgid == -1) {`
`perror ("msgget");`
`exit (EXIT_FAILURE);`
`}`



发送消息

- 发送消息

```
#include <sys/msg.h>
```

```
int msgsnd (int msgid, const void* msgp, size_t msgsz,  
            int msgflg);
```

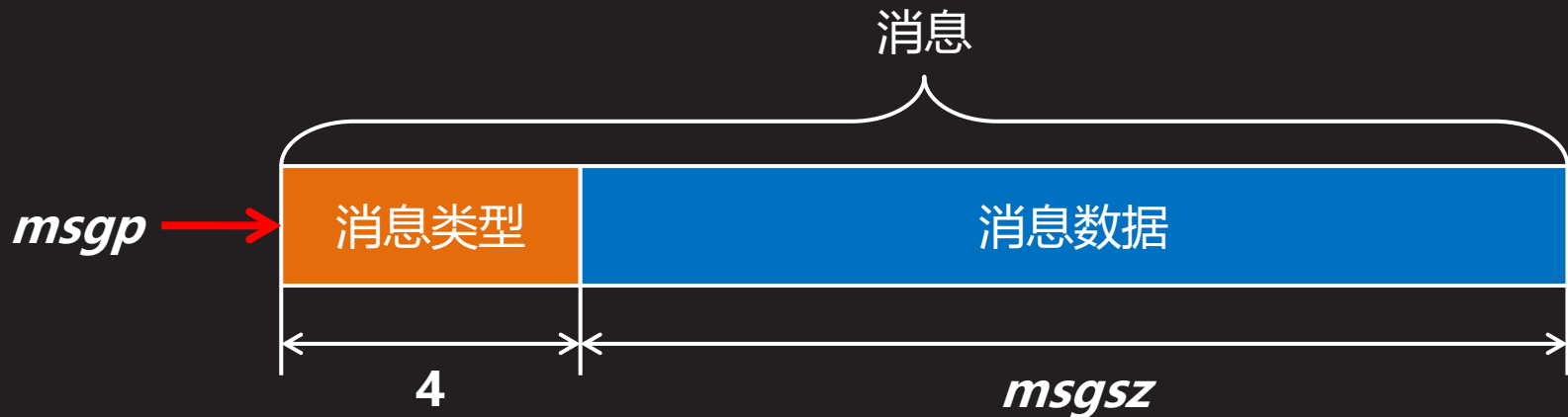
成功返回0，失败返回-1

- *msgid*: 消息队列标识符
- *msgp*: 指向一个包含消息类型和消息数据的内存块。该内存块的前4个字节必须是一个大于0的整数，代表消息类型，其后紧跟消息数据。消息数据长度用*msgsz*参数表示
- *msgsz*: 期望发送消息数据(不含消息类型)的字节数
- *msgflg*: 发送标志，一般取0即可



发送消息 (续1)

- 注意msgsnd函数的*msgp*参数所指向的内存块中包含消息类型，其值必须大于0，但该函数的*msgsz*参数所表示的期望发送字节数中却不包含消息类型所占的4个字节



- 如果系统内核中的消息未达上限，则msgsnd函数会将欲发送消息加入指定的消息队列并立即返回0，否则该函数会阻塞，直到系统内核允许加入新消息为止(比如有消息因被接收而离开消息队列)

发送消息 (续2)

- 若 *msgflg* 参数中包含 `IPC_NOWAIT` 位, 则 `msgsnd` 函数在系统内核中的消息已达上限的情况下不会阻塞, 而是返回 -1, 并置 `errno` 为 `EAGAIN`

- 例如

```
– struct {
    long type;
    char data[1024];
} msg = {1234};
fgets (msg.data, sizeof (msg.data) /
    sizeof (msg.data[0]), stdin);
if (msgsnd (msgid, &msg, strlen (msg.data) *
    sizeof (msg.data[0]), 0) == -1) {
    perror ("msgsnd"); exit (EXIT_FAILURE); }
```



接收消息

- 接收消息

```
#include <sys/msg.h>
```

```
ssize_t msgrcv (int msgid, void* msgp, size_t msgsz,  
               long msgtyp, int msgflg);
```

成功返回所接收消息数据的字节数，失败返回-1

- *msgid*: 消息队列标识符
- *msgp*: 指向一块包含消息类型(4字节)和消息数据的内存
- *msgsz*: 期望接收消息数据(不含消息类型)的字节数



接收消息 (续1)

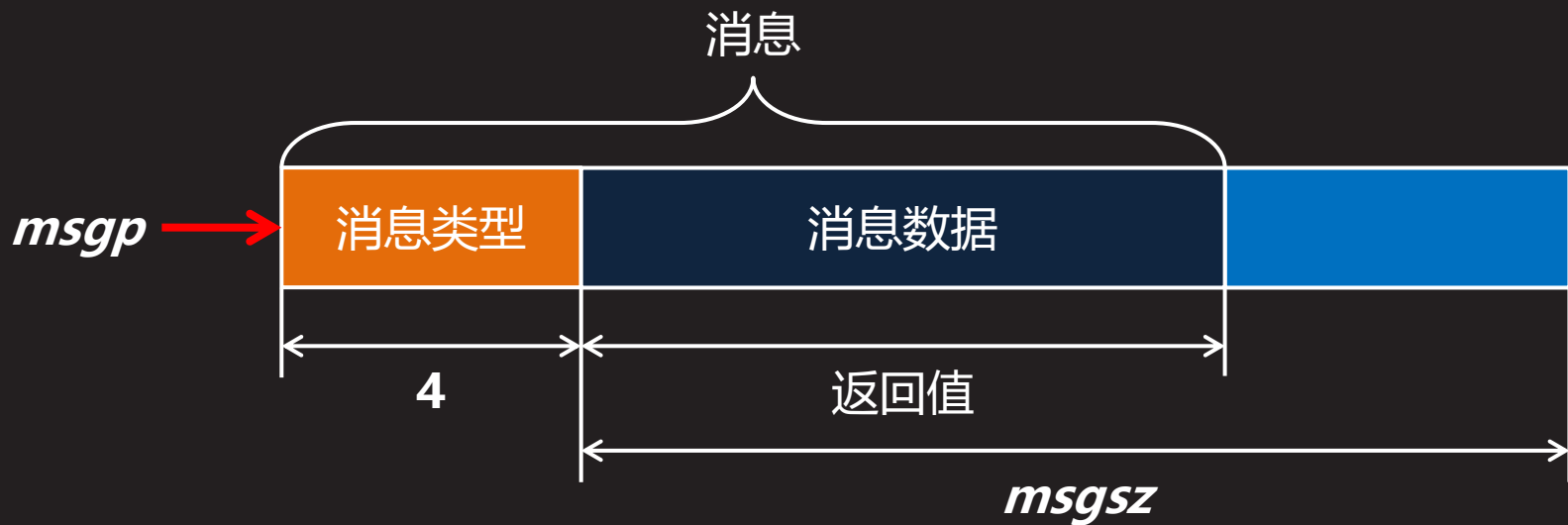
- 接收消息
 - *msgtyp*: 消息类型, 可取以下值
 - 0 - 提取消息队列的第一条消息
 - >0 - 若*msgflg*参数不包含MSG_EXCEPT位, 则提取消息队列的第一条类型为*msgtyp*的消息; 若*msgflg*参数包含MSG_EXCEPT位, 则提取消息队列的第一条类型不为*msgtyp*的消息
 - <0 - 提取消息队列中类型小于等于*msgtyp*的绝对值的消息, 类型越小的消息越被优先提取
 - *msgflg*: 接收标志, 一般取0即可



接收消息 (续2)

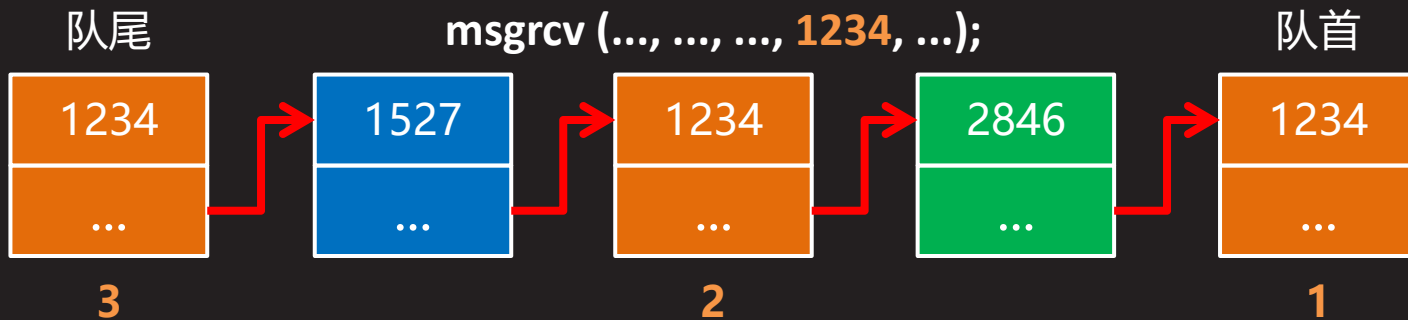
- 注意msgrcv函数的*msgp*参数所指向的内存块中包含消息类型，其值由该函数输出，但该函数的*msgsz*参数所表示的期望接收字节数以及该函数所返回的实际接收字节数中都不包含消息类型所占的4个字节

知识讲解



接收消息 (续3)

- 若存在与 *msgtyp* 参数匹配的消息，但其数据长度大于 *msgsz* 参数，且 *msgflg* 参数包含 MSG_NOERROR 位，则只截取该消息数据的前 *msgsz* 字节返回，剩余部分直接丢弃；但如果 *msgflg* 参数不包含 MSG_NOERROR 位，则不处理该消息，直接返回 -1，并置 *errno* 为 E2BIG
- *msgrcv* 函数根据 *msgtyp* 参数对消息队列中的消息有选择地接收，只有满足条件的消息才会被复制到应用程序缓冲区并从内核中删除。如果满足 *msgtyp* 条件的消息不只一条，则按照先进先出的规则提取



接收消息 (续4)

- 若消息队列中有可接收消息，则msgrcv函数会将该消息移出消息队列，并立即返回所接收到的消息数据的字节数，表示接收成功，否则此函数会阻塞，直到消息队列中有可接收消息为止
- 若 *msgflg* 参数包含IPC_NOWAIT位，则msgrcv函数在消息队列中没有可接收消息的情况下不会阻塞，而是返回-1，并置errno为ENOMSG



接收消息 (续5)

- 例如

```
– struct {
    long type;
    char data[1024];
} msg = {};
ssize_t msgsz = msgrcv (msgid, &msg, sizeof (
    msg.data) - sizeof (msg.data[0]), 1234,
    MSG_NOERROR);
if (msgsz == -1) {
    perror ("msgrcv");
    exit (EXIT_FAILURE);
}
printf ("长度: %d, 内容: %s\n", msgsz, msg.data);
```



销毁消息队列

- 销毁消息队列

```
#include <sys/msg.h>
```

```
int msgctl (int msgid, int cmd, struct msqid_ds* buf);
```

成功返回0，失败返回-1

- *msgid*: 消息队列标识符
- *cmd*: 控制命令，取IPC_RMID
立即删除消息队列，所有处于阻塞状态的对该消息队列的msgsnd和msgrcv函数调用，都会立即返回失败，且errno为EIDRM
- *buf*: 置NULL即可



编程模型



编程模型

- 基于消息队列实现进程间通信的编程模型

步骤	进程A	函数	进程B	步骤
1	创建消息队列	msgget	获取消息队列	1
2	发送接收消息	msgsnd msgrcv	发送接收消息	2
3	销毁消息队列	msgctl	——	——



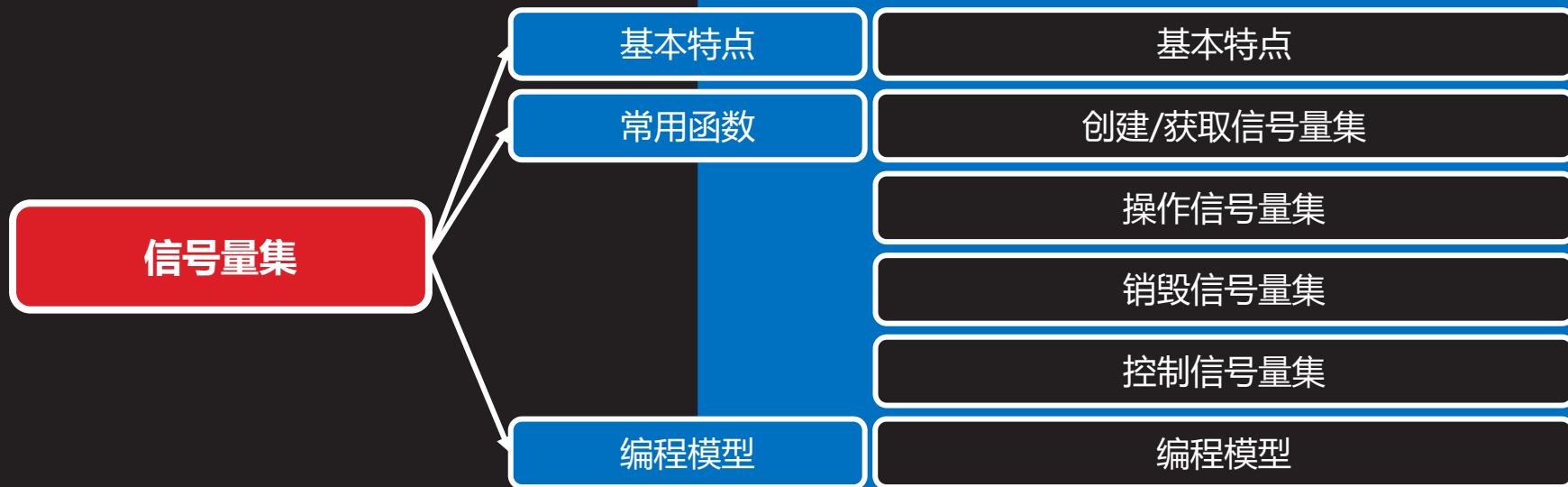
基于消息队列的进程间通信

【参见：wmsg.c、rmsg.c】

- 基于消息队列的进程间通信



信号量集



基本特点



基本特点

- 信号量与其它几种IPC机制(管道、共享内存和消息队列)都不一样，它的目的不是在进程之间搭建数据流通的桥梁，而是提供一个可为多个进程共同访问计数器，实时跟踪可用资源的数量，以解决多个用户分享有限资源时的竞争与冲突问题



基本特点 (续1)

- 为了获得共享资源，进程需按以下步骤执行
 1. 测试控制该资源的信号量
 2. 若信号量的值大于0，说明还有可分配资源，则进程获得该资源，并将信号量的值减1，表示可分配资源少了一个
 3. 若信号量的值等于0，说明没有可分配资源，则进程进入睡眠状态，直到信号量的值再度大于0，这时会有一个正在睡眠中等待该资源的进程被系统内核唤醒，它将返回执行步骤1，而其它进程则继续在睡眠中等待
 4. 当进程不再使用所获得的资源时，应将控制该资源的信号量的值加1，表示可分配资源多了一个。此时那些正在睡眠中等待该资源的进程中的一个将被系统内核唤醒



基本特点 (续2)

- 从有关信号量的操作步骤不难看出，对信号量所做的测试和加减操作都必须是原子化的，用户空间的全局变量显然无法胜任，因此信号量通常被实现在系统内核之中
- System V的信号量集比单个信号量要复杂一些
 - 信号量集并非一个简单的非负整数，而是将一个或多个信号量放在一起，组成一个信号量的集合来使用
 - 信号量集的创建和初始化必须被分作两步而不能在一个原子操作中完成
 - 与其它几种XSI IPC对象一样，信号量集也是系统级对象。如果某进程在其终止之前，没有恢复信号量集里的资源计数，也没有通过函数或命令销毁该信号量集对象，那么这种状态将一直保持下去，并对以后运行的进程构成影响



常用函数



创建/获取信号量集

- 创建新的或获取已有的信号量集

```
#include <sys/sem.h>
```

```
int semget (key_t key, int nsems, int semflg);
```

成功返回信号量集标识符，失败返回-1

- *key*: 信号量集键
- *nsems*: 信号量个数
- *semflg*: 创建标志，可取以下值
 - 0 - 获取，不存在即失败
 - IPC_CREAT - 创建，不存在即创建，已存在即获取
 - IPC_EXCL - 排斥，已存在即失败



创建/获取信号量集 (续1)

- 例如

```
– int semid = semget (key, 4,  
    0644 | IPC_CREAT | IPC_EXCL);  
if (semid == -1) {  
    perror ("semget");  
    exit (EXIT_FAILURE);  
}  
  
– int semid = semget (key, 0, 0);  
if (semid == -1) {  
    perror ("semget");  
    exit (EXIT_FAILURE);  
}
```



操作信号量集

- 操作信号量集

```
#include <sys/sem.h>
```

```
int semop (int semid, struct sembuf* sops,  
           unsigned nsops);
```

成功返回0，失败返回-1

- *semid*: 信号量集标识符
- *sops*: 操作结构体数组
- *nsops*: 操作结构体数组长度

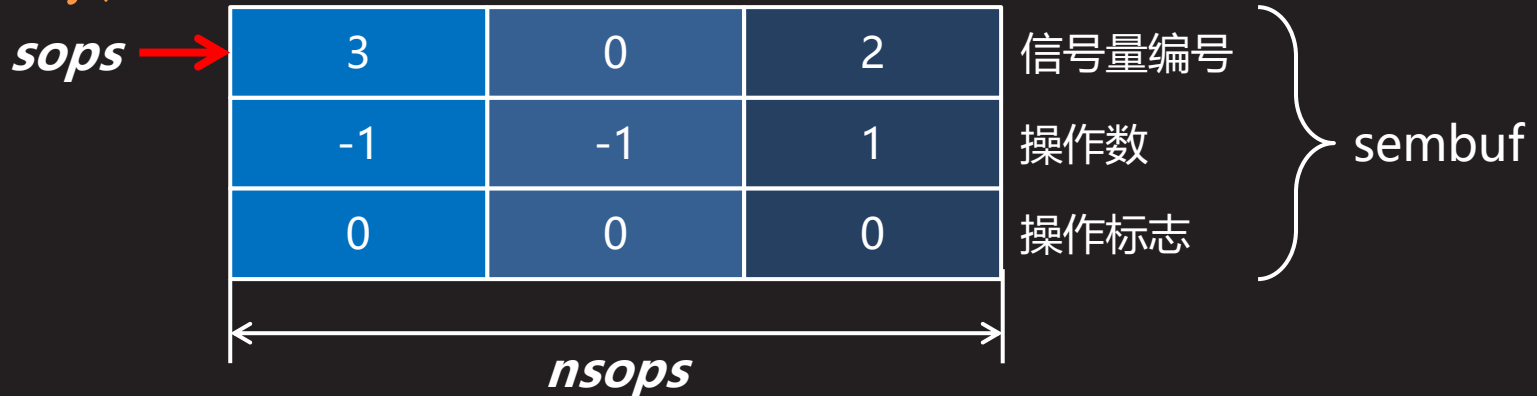


操作信号量集 (续1)

- 作为semop函数的参数, *sops*和*nsops*所表示的数组由若干操作结构体组成, 操作结构体的类型为sembuf

```

- struct sembuf {
    unsigned short sem_num; // 信号量编号
    short          sem_op;  // 操作数
    short          sem_flg; // 操作标志
};
    
```



- 该结构体数组中的每个元素通过其信号量编号成员与信号量集中的一个特定的信号量对应, 表示对该信号量的操作



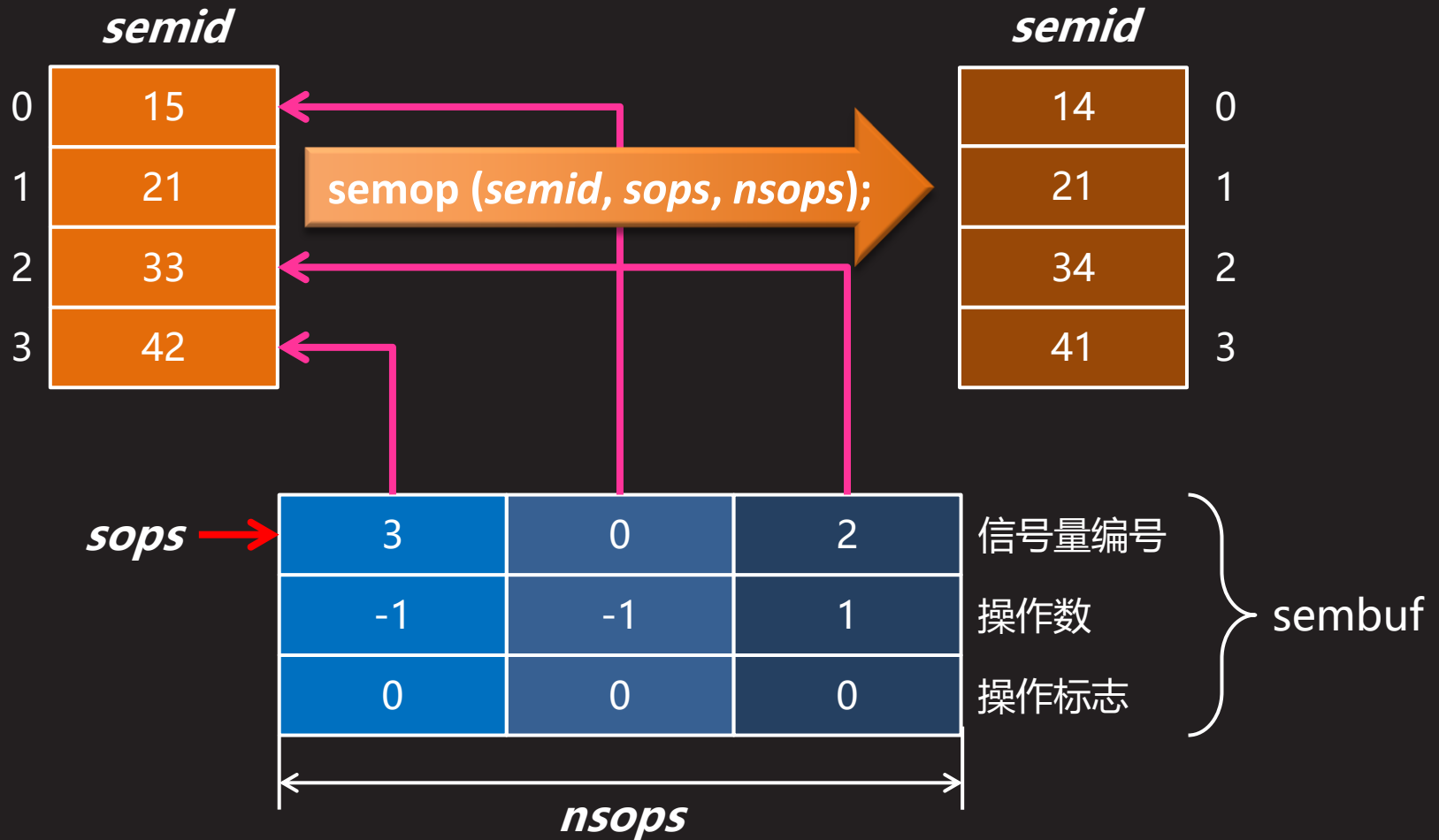
操作信号量集 (续2)

- semop函数对 *sops* 指向的包含 *nsops* 个元素的操作结构体数组中的每个元素执行如下操作
 - 若 *sem_op* 大于0, 则将其加到 *semid* 信号量集第 *sem_num* 号信号量的值上, 以表示对资源的释放
 - 若 *sem_op* 小于0, 则从 *semid* 信号量集第 *sem_num* 号信号量的值中减去其绝对值, 以表示对资源的获取; 如果不够减(信号量的值不能为负), 则此函数会阻塞, 直到够减为止, 以表示对资源的等待; 但如果 *sem_flg* 包含 *IPC_NOWAIT* 位, 则即使不够减也不会阻塞, 而是返回-1, 并置 *errno* 为 *EAGAIN*, 以便在等待资源时还可做其它处理
 - 若 *sem_op* 等于0, 则直到 *semid* 信号量集第 *sem_num* 号信号量的值为0时才返回, 除非 *sem_flg* 含 *IPC_NOWAIT* 位



操作信号量集 (续3)

知识讲解



操作信号量集 (续4)

- 例如

```
– struct sembuf sops[] = { {3, -1, 0}, {0, -1, 0}, {2, 1, 0} };  
  if (semop (semid, &sops,  
           sizeof (sops) / sizeof (sops[0])) == -1) {  
      perror ("semop");  
      exit (EXIT_FAILURE);  
  }
```



销毁信号量集

- 销毁信号量集

```
#include <sys/sem.h>
```

```
int semctl (int semid, int semnum, int cmd, ...);
```

成功返回0，失败返回-1

- *semid*: 信号量集标识符
- *semnum*: 取0即可
- *cmd*: 取IPC_RMID

立即删除信号量集，所有处于阻塞状态的对该信号量集的semop函数调用，都会立即返回失败，且errno为EIDRM



控制信号量集

- 获取信号量集中每个信号量的值
 - `int semctl (int semid, 0, GETALL, unsigned short* array);`
- 设置信号量集中每个信号量的值
 - `int semctl (int semid, 0, SETALL, unsigned short* array);`
- 获取信号量集中指定信号量的值
 - `int semctl (int semid, int semnum, GETVAL);`
 - 成功返回semid信号量集中第semnum号信号量的值
- 设置信号量集中指定信号量的值
 - `int semctl (int semid, int semnum, SETVAL, int val);`



编程模型



编程模型

- 基于信号量实现进程间通信的编程模型

步骤	进程A	函数	进程B	步骤
1	创建信号量集	semget	获取信号量集	1
2	初始信号量集	semctl	——	——
3	操作信号量集	semop	操作信号量集	2
4	销毁信号量集	semctl	——	——



基于信号量的进程间通信

【参见：csem.c、gsem.c】

- 基于信号量的进程间通信



IPC命令

IPC命令

IPC命令

IPC命令

IPC命令



IPC命令

- 查看IPC对象

- 查看共享内存: **ipcs -m** (**m**=memory)
- 查看消息队列: **ipcs -q** (**q**=queue)
- 查看信号量集: **ipcs -s** (**s**=semaphore)
- 查看所有对象: **ipcs -a** (**a**=all)

- 删除IPC对象

- 删除共享内存: **ipcrm -m** <共享内存标识符>
- 删除消息队列: **ipcrm -q** <消息队列标识符>
- 删除信号量集: **ipcrm -s** <信号量集标识符>



总结和答疑

