Technology Tarena 达内科技

Unix系统高级编程

IPC

DAY12

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	XSI的IPC对象
	10:30 ~ 11:20	共享内存
	11:30 ~ 12:20	消息队列
下午	14:00 ~ 14:50	基于消息队列的本地银行
	15:00 ~ 15:50	
	16:00 ~ 16:50	信号量与IPC命令
	17:00 ~ 17:30	总结和答疑

XSI的IPC对象


XSI的IPC对象
IPC对象的标识符和键
IPC对象的会合方式
IPC对象的编程接口
IPC对象的权限结构
IPC对象的优缺点
IPC对象的标识符和键
IPC对象的会合方式
IPC对象的编程接口
IPC对象的权限结构
IPC对象的优缺点

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

知识讲解

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

知识讲解

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

知识讲解

IPC对象的编程接口

IPC对象的编程接口

- IPC对象的创建与获取
  - 共享内存
    int shmget (key_t key, size_t size, int shmflg);
    消息队列
    int msgget (key_t key, int msgflg);
    信号量集
    int semget (key_t key, int nsems, int semflg);
  - 以IPC_PRIVATE为键创建新IPC对象总能获得成功
    创建标志取0，无则失败，有则获取
    创建标志含IPC_CREAT，无则创建，有则获取
    创建标志含IPC_CREAT和IPC_EXCL，无则创建，有则失败

知识讲解

IPC对象的编程接口（续1）

- IPC对象的销毁与控制
  - 共享内存

    int shmctl (int shmid, int cmd, struct shmid_ds* buf);

    消息队列

    int msgctl (int msqid, int cmd, struct msqid_ds* buf);

    信号量集

    int semctl (int semid, int semnum, int cmd,
    union semun arg);
  - cmd可取以下值

    IPC_STAT - 获取IPC对象属性

    IPC_SET - 设置IPC对象属性

    IPC_RMID - 删除IPC对象

IPC对象的权限结构

IPC对象的权限结构

- XSI IPC为每个IPC对象都设置了一个ipc_perm结构，该结构描述了IPC对象的拥有者和权限
  -
```
struct ipc_perm {
        key_t          __key; // 键
        uid_t          uid;   // 拥有者用户ID
        gid_t          gid;   // 拥有者组ID
        uid_t          cuid;  // 创建者用户ID
        gid_t          cgid;  // 创建者组ID
        unsigned short mode;  // 权限
        unsigned short __seq; // 序号
};
```
- 不同实现的ipc_perm结构不完全相同，但以上所列是最基本的形式，具体的结构定义可以查阅<ipc.h>头文件

知识讲解

IPC对象的权限结构（续1）

- ipc_perm结构中各成员在IPC对象被创建时即获得初值，此后可以通过shmctl、msgctl、semctl等函数获取或修改，但是只有uid、gid和mode三个成员可以修改，而且修改进程的用户必须是该IPC对象的创建者或超级用户
- ipc_perm结构的mode成员，表示拥有者用户、拥有者组和其它用户对该IPC对象的读、写(注意没有执行)权限，分别用从高到低的三位八进制数表示，每位八进制数用4和2中的一个或位或组成，4表示读权限，2表示写权限
  - 0666：任何用户都可读可写该对象
  - 0644：任何用户都可读该对象，但只有拥有者用户可写
  - 0444：任何用户都只能读该对象，除非是超级用户

知识讲解

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

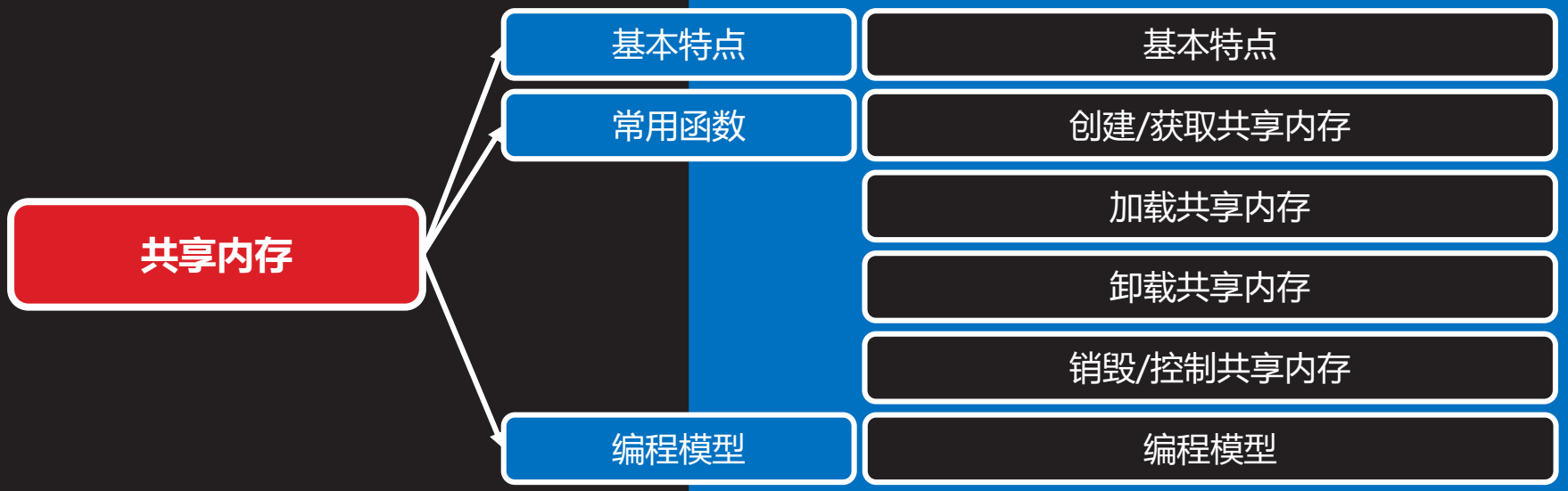
知识讲解

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

知识讲解

共享内存


共享内存
基本特点
常用函数
编程模型
基本特点
创建/获取共享内存
加载共享内存
卸载共享内存
销毁/控制共享内存
编程模型

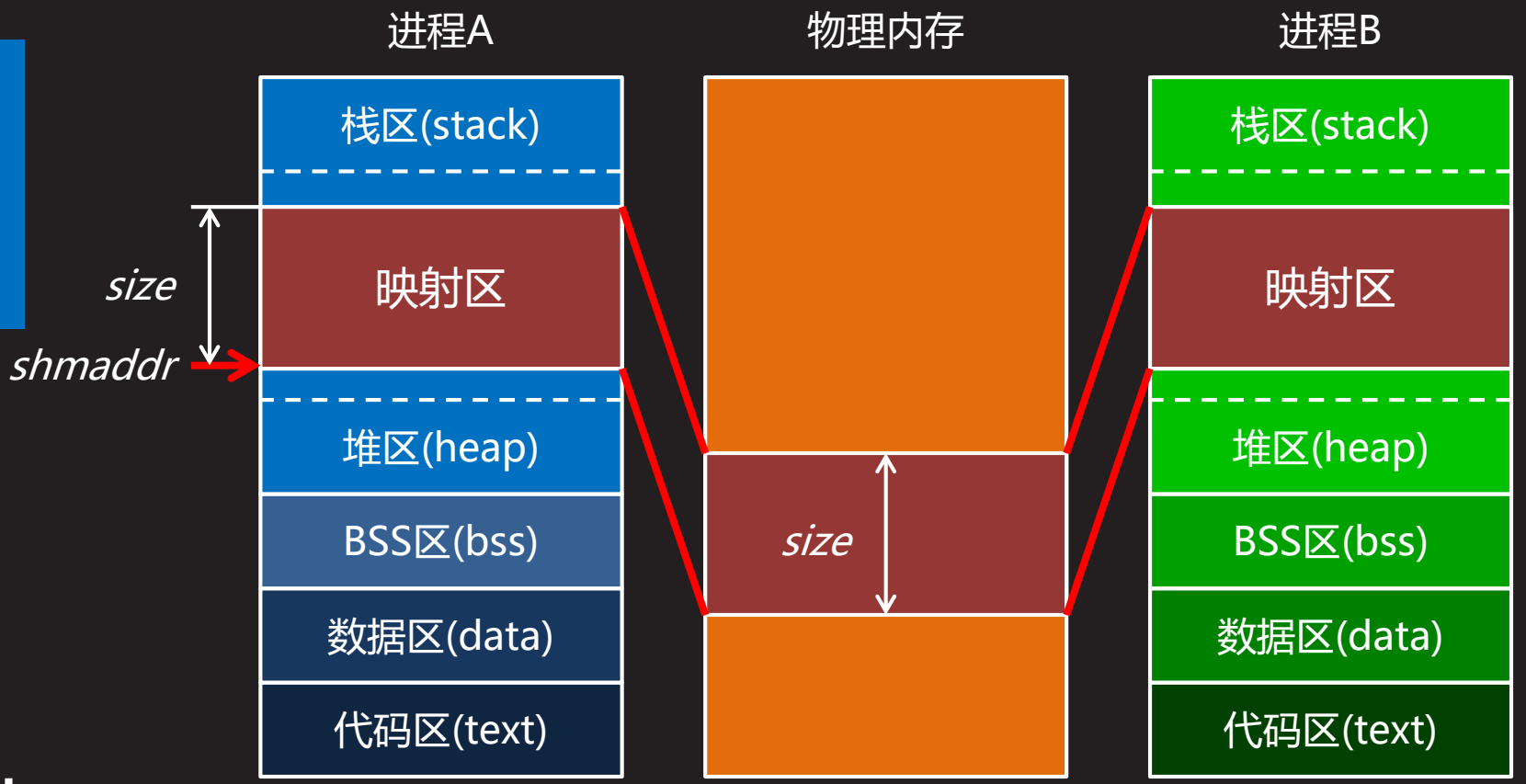
Technology Tarena 达内科技

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

知识讲解

基本特点（续2）

- 系统内核为每个共享内存对象维护一个属性结构
  - struct shmid_ds {

```
    struct ipc_perm shm_perm;    // 拥有者和权限
    size_t          shm_segsz;   // 大小(以字节为单位)
    time_t          shm_atime;   // 最后加载时间
    time_t          shm_dtime;   // 最后卸载时间
    time_t          shm_ctime;   // 最后改变时间
    pid_t           shm_cpid;    // 创建进程PID
    pid_t           shm_lpid;    // 最后加载/卸载进程PID
    shmatt_t        shm_nattch;  // 当前加载计数
    ...
};
```


Technology Tarena 达内科技

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

知识讲解

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); }

知识讲解

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

知识讲解

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

知识讲解

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

知识讲解

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(shmid_ds::shm_nattch)减一
- 例如
  - if (shmdt (shmaddr) == -1) {
    perror ("shmdt"); exit (EXIT_FAILURE); }

销毁/控制共享内存

- 销毁或控制共享内存

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

int shmctl (int shmid, int cmd, struct shmid_ds* buf);

                                成功返回0，失败返回-1
```

  - *shmid*：共享内存标识符

知识讲解

销毁/控制共享内存（续1）

- 销毁或控制共享内存
  - ***cmd***：控制命令，可取以下值

    `IPC_STAT` - 获取共享内存的属性，通过***buf***参数输出

    `IPC_SET` - 设置共享内存的属性，通过***buf***参数输入，仅以下三个属性可以设置

    ```
    shmid_ds::shm_perm.uid   // 拥有者用户ID
    shmid_ds::shm_perm.gid   // 拥有者组ID
    shmid_ds::shm_perm.mode  // 权限
    ```

    `IPC_RMID` - 销毁共享内存。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0时，共享内存才会真的被销毁
  - ***buf***：shmid_ds类型的共享内存属性结构

知识讲解

销毁/控制共享内存（续2）

- 例如
  - struct shmid_ds shm;
    if (shmctl (shmid, IPC_STAT, &shm) == -1) {
        perror ("shmctl"); exit (EXIT_FAILURE); }
    shm.shm_perm.mode = 0600;
    if (shmctl (shmid, IPC_SET, &shm) == -1) {
        perror ("shmctl"); exit (EXIT_FAILURE); }
    if (shmctl (shmid, IPC_RMID, NULL) == -1) {
        perror ("shmctl"); exit (EXIT_FAILURE); }

知识讲解

Technology Tarena 达内科技

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	——	——

知识讲解

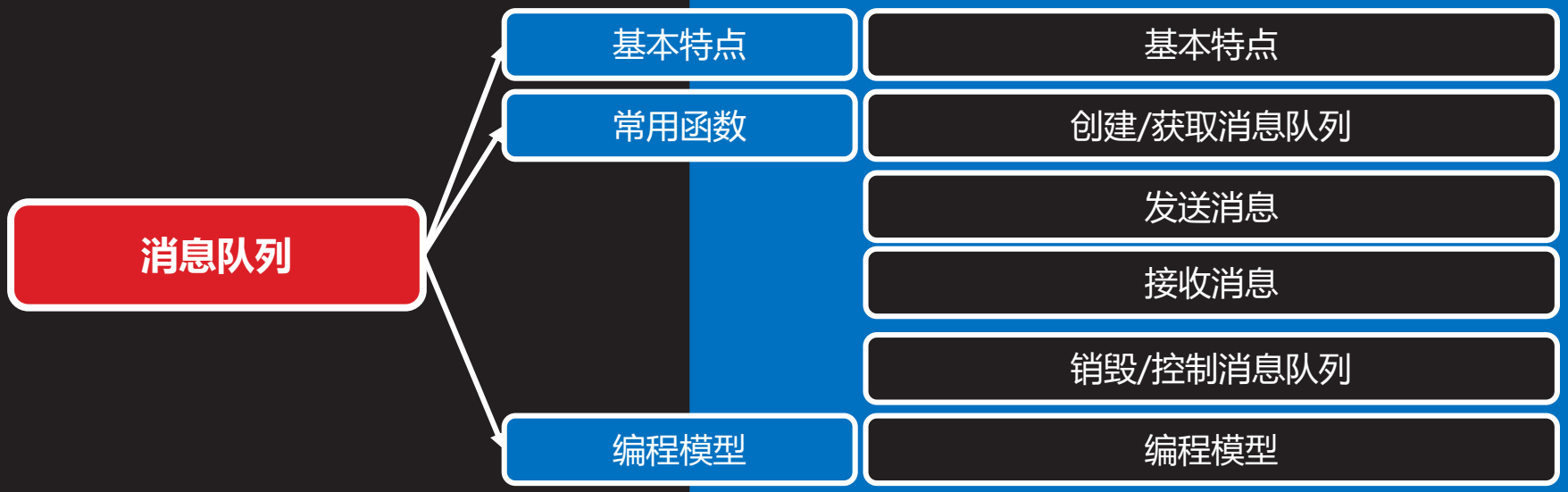
基于共享内存的进程间通信

【参见：wshm.c、rshm.c】

课堂练习

- 基于共享内存的进程间通信

消息队列

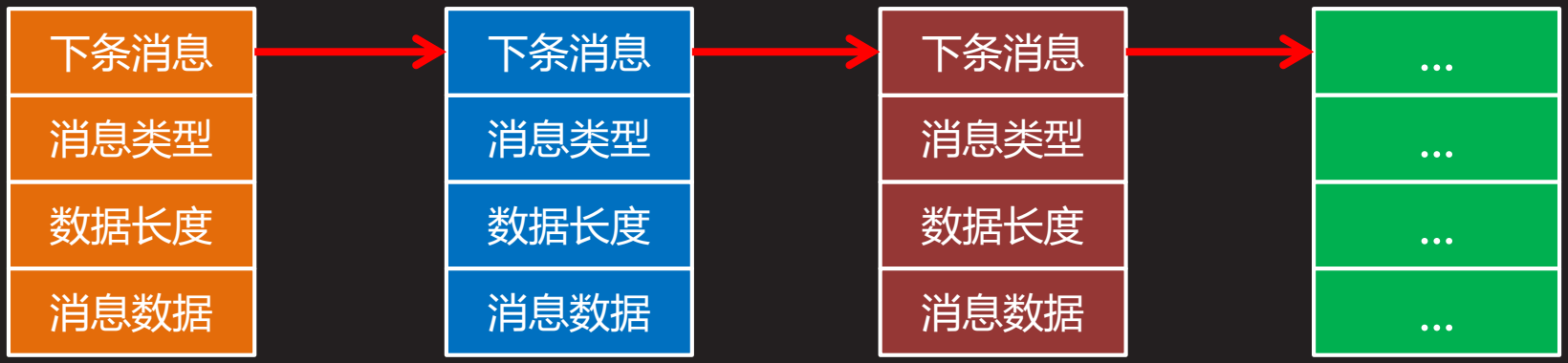


Technology Tarena 达内科技

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

知识讲解

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
  - 可发送消息字节数$_{max}$：8192
  - 队列数据长度之和$_{max}$：16384
  - 系统中消息队列数$_{max}$：16
  - 系统中消息数总和$_{max}$：262144

知识讲解

基本特点（续3）

- 系统内核为每个消息队列对象维护一个属性结构
  - struct msqid_ds {

```
    struct ipc_perm msg_perm;     // 拥有者和权限
    time_t          msg_stime;    // 最后发送时间
    time_t          msg_rtime;    // 最后接收时间
    time_t          msg_ctime;    // 最后改变时间
    unsigned long   __msg_cbytes; // 队列字节数
    msgqnum_t       msg_qnum;     // 队列消息数
    msglen_t        msg_qbytes;   // 队列最大字节数
    pid_t           msg_lspid;    // 最后发送进程PID
    pid_t           msg_lrpid;    // 最后接收进程PID
    ...
};
```

Technology Tarena 达内科技

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

知识讲解

创建/获取消息队列（续1）

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

知识讲解

发送消息

- 发送消息

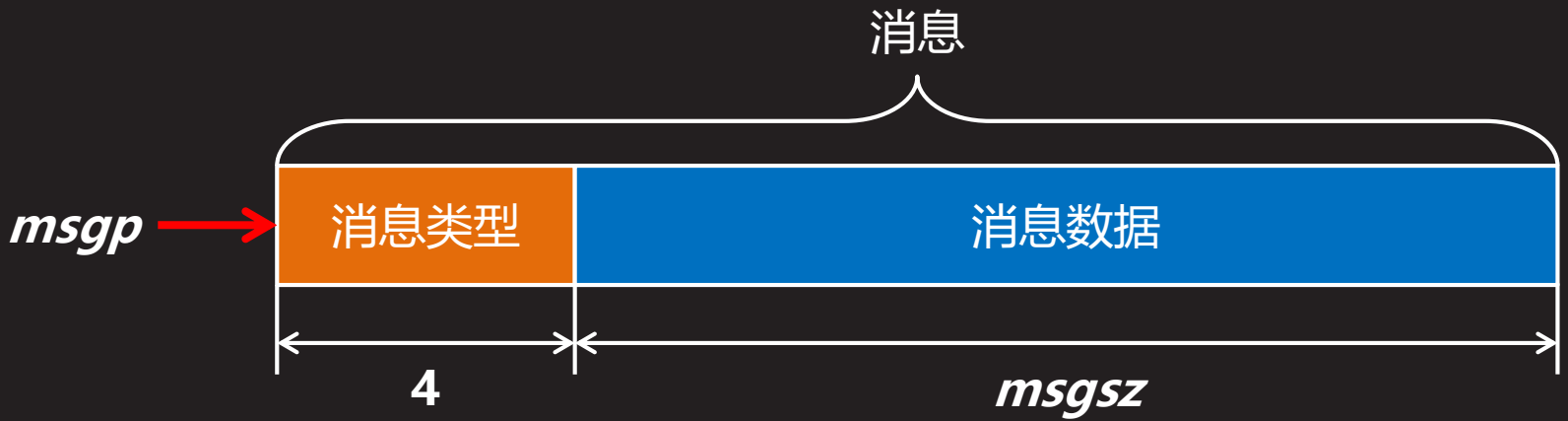
```
#include <sys/msg.h>

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

成功返回0，失败返回-1

- ***msqid***：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 mtype;
    char mtext[1024];
    } msgbuf = {1234};
    gets (msgbuf.mtext);
    if (msgsnd (msqid, &msgbuf, (strlen (msgbuf.mtext)
    + 1) * sizeof (msgbuf.mtext[0]), 0) == -1) {
    perror ("msgsnd"); exit (EXIT_FAILURE); }

接收消息

- 接收消息

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

ssize_t msgrcv (int msqid, 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

- *msqid*：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

知识讲解

接收消息（续1）

- 接收消息
  - ***msgtyp***：消息类型，可取以下值

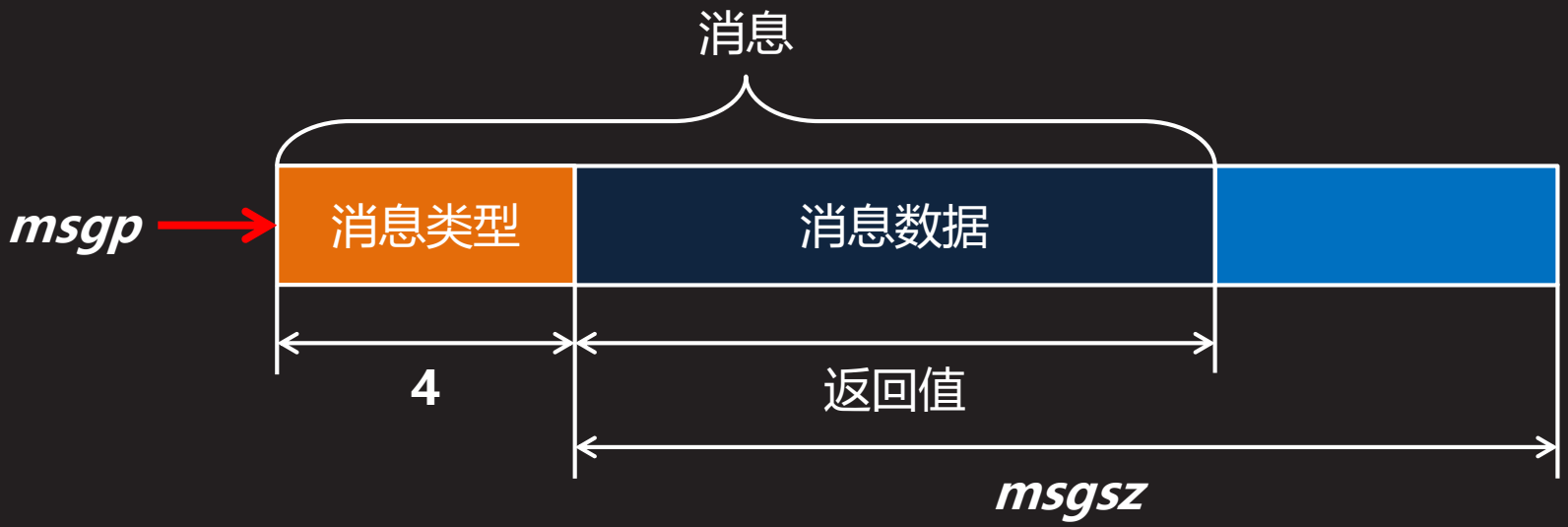
    0 - 提取消息队列的第一条消息

    >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

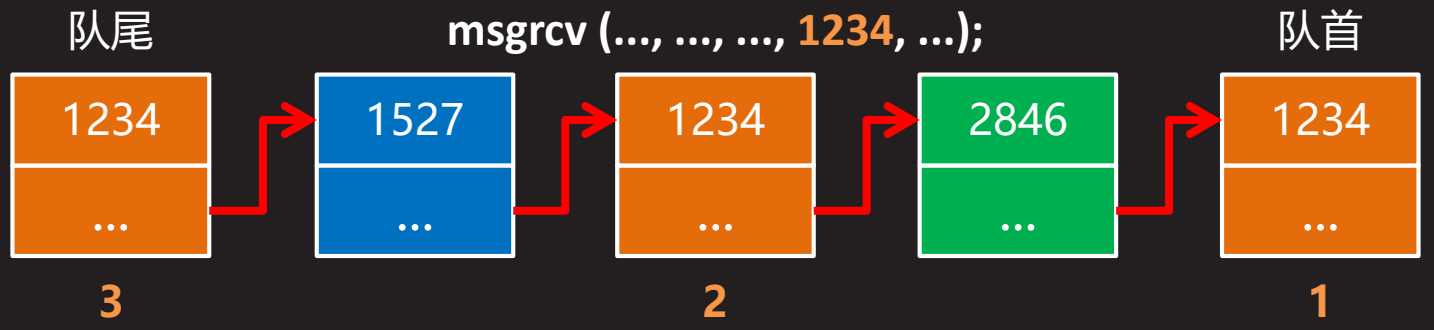


知识讲解

接收消息（续3）

知识讲解

- 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

知识讲解

接收消息（续5）

- 例如
  - struct {
    long mtype;
    char mtext[1024];
    } msgbuf = {};
    ssize_t msgsz = msgrcv (msqid, &msgbuf,
    sizeof (msgbuf.mtext) - sizeof (msgbuf.mtext[0]),
    1234, MSG_NOERROR);
    if (msgsz == -1) {
    perror ("msgrcv");
    exit (EXIT_FAILURE); }
    printf ("长度：%d，内容： %s\n",
    msgsz, msgbuf.mtext);

销毁/控制消息队列

- 销毁或控制消息队列

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

int msgctl (int msqid, int cmd, struct msqid_ds* buf);

                                成功返回0，失败返回-1
```

  - *msqid*：消息队列标识符

知识讲解

销毁/控制消息队列（续1）

Technology Tarena 达内科技

- 销毁或控制消息队列
  - ***cmd***：控制命令，可取以下值

    `IPC_STAT` - 获取消息队列的属性，通过***buf***参数输出

    `IPC_SET` - 设置消息队列的属性，通过***buf***参数输入，仅以下四个属性可以设置

    ```
    msqid_ds::msg_perm.uid   // 拥有者用户ID
    msqid_ds::msg_perm.gid   // 拥有者组ID
    msqid_ds::msg_perm.mode  // 权限
    msqid_ds::msg_qbytes     // 队列最大字节数
    ```

    `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***：msqid_ds类型的消息队列属性结构

知识讲解

销毁/控制消息队列（续2）

- 例如
  - struct msqid_ds msq;

```
if (msgctl (msqid, IPC_STAT, &msq) == -1) {
    perror ("msgctl"); exit (EXIT_FAILURE); }
msq.msg_perm.mode = 0600;
if (msgctl (msqid, IPC_SET, &msq) == -1) {
    perror ("msgctl"); exit (EXIT_FAILURE); }
if (msgctl (msqid, IPC_RMID, NULL) == -1) {
    perror ("msgctl"); exit (EXIT_FAILURE); }
```


Technology Tarena 达内科技

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	——	——

知识讲解

基于消息队列的进程间通信

【参见：wmsq.c、rmsq.c】

课堂练习

- 基于消息队列的进程间通信

基于消息队列的本地银行

Technology Tarena 达内科技

需求分析

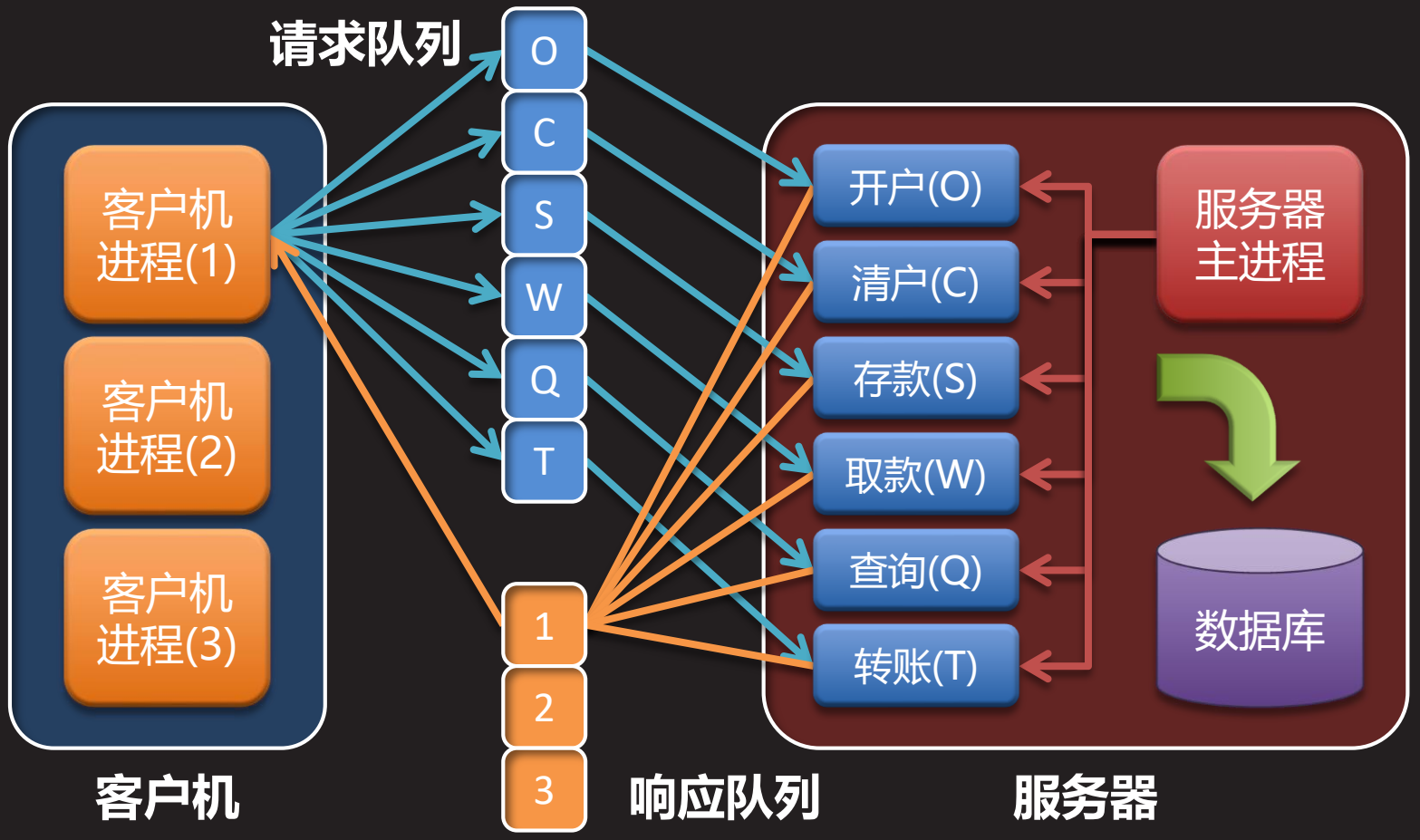
需求分析

- 基于消息队列的本地银行，包括客户机和服务器两个组成部分。其中客户机负责提供用户界面，接收用户的输入，并向其提供输出，服务器负责后台业务逻辑的处理
- 每一种银行业务皆被实现为一个独立的进程，谓之业务服务。服务器主进程负责消息队列的创建和各业务服务进程的启动与终止。增减业务只需要增减业务服务即可
- 客户机与各业务服务之间通过消息队列在同一台机器上进行通信，故谓之本地银行。消息队列分为请求队列和响应队列，前者负责传输客户机发往各业务服务的业务请求，后者负责传输各业务服务返回客户机的处理结果
- 业务数据需要在服务器端持久化，采用文件系统数据库

Technology Tarena 达内科技

概要设计

概要设计



- 请求队列以业务标识作为消息类型以区分不同的业务
- 响应队列以进程标识作为消息类型以区分不同的客户机

Technology Tarena 达内科技

开发计划

开发计划

1. 编写公共头文件：bank.h
2. 实现数据存储模块：dao.h、dao.c
3. 实现各业务服务模块：open.c、close.c、save.c、withdraw.c、query.c、transfer.c
4. 实现服务器主进程模块：server.c
5. 实现客户机模块：client.c
6. 编写构建脚本：makefile
7. 链接、运行、测试

知识讲解

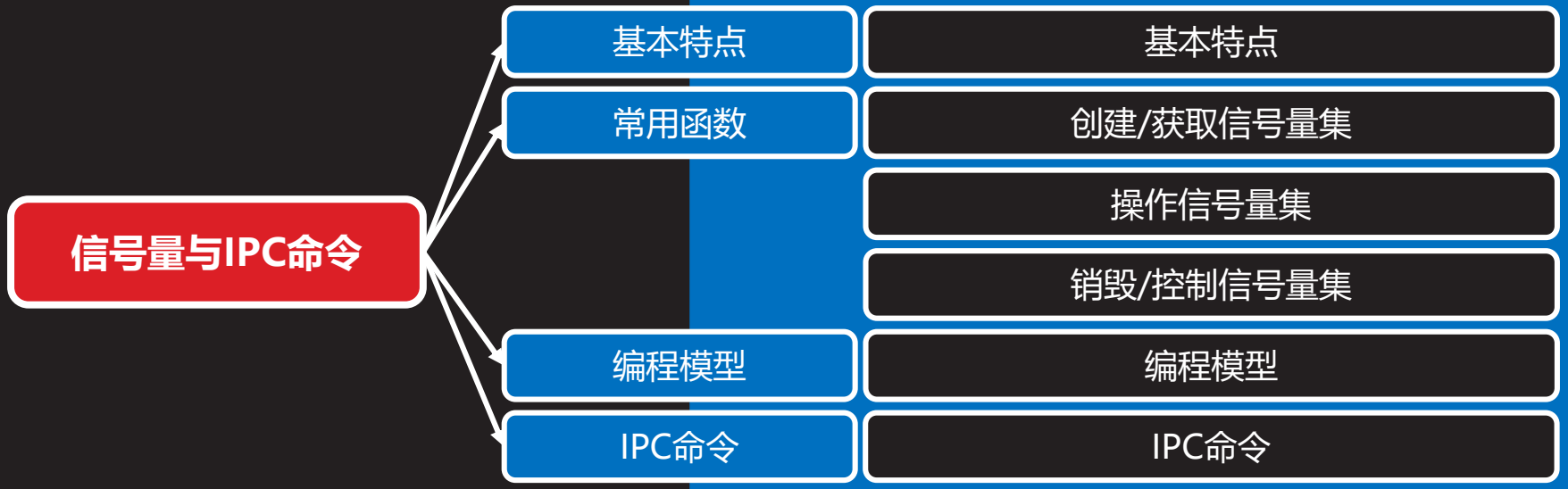
基于消息队列的本地银行

【参见：bank/】

课堂练习

- 基于消息队列的本地银行

信号量与IPC命令


信号量与IPC命令
基本特点
常用函数
编程模型
IPC命令
基本特点
创建/获取信号量集
操作信号量集
销毁/控制信号量集
编程模型
IPC命令

Technology Tarena 达内科技

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

知识讲解

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

知识讲解

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的信号量比起通常意义上的信号量要复杂一些
  - 信号量并非被定义为一个简单的非负整数，相反必须把一个或多个信号量放在一起，组成一个信号量集来使用
  - 信号量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，信号量也是系统级对象。如果某进程在其正常或异常终止前，没有恢复信号量里的资源计数，也没有通过函数或命令删除该信号量对象，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

基本特点（续3）

- 系统内核为每个信号量集对象维护一个属性结构
  - struct semid_ds {

```
    struct ipc_perm sem_perm;   // 拥有者和权限
    time_t          sem_otime;  // 最后操作时间
    time_t          sem_ctime;  // 最后改变时间
    unsigned short  sem_nsems;  // 信号量个数
    ...
};
```


Technology Tarena 达内科技

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

知识讲解

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);
    }

知识讲解

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);
```

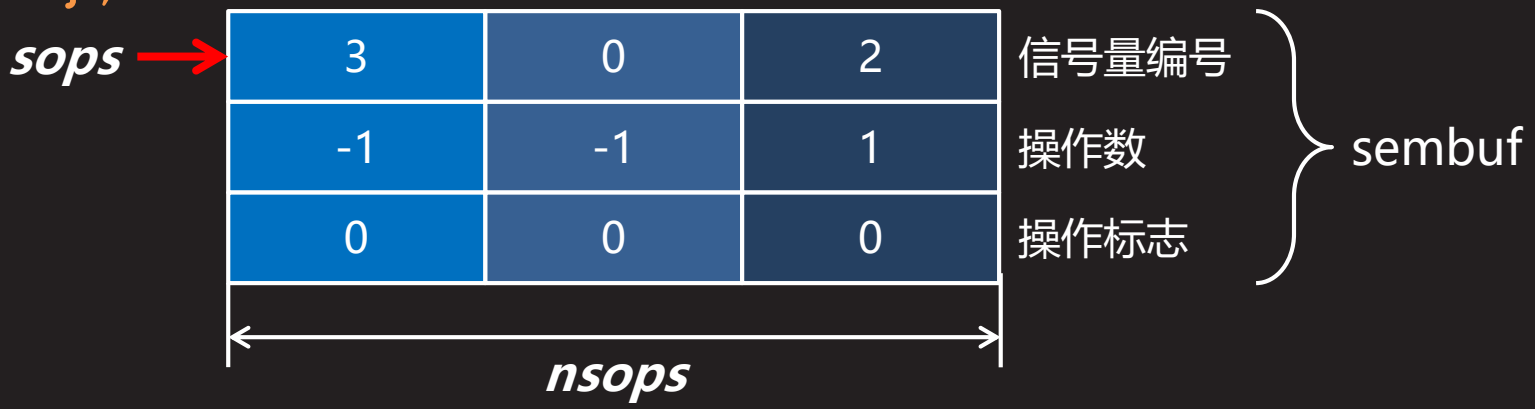
成功返回0，失败返回-1

  - *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

知识讲解

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

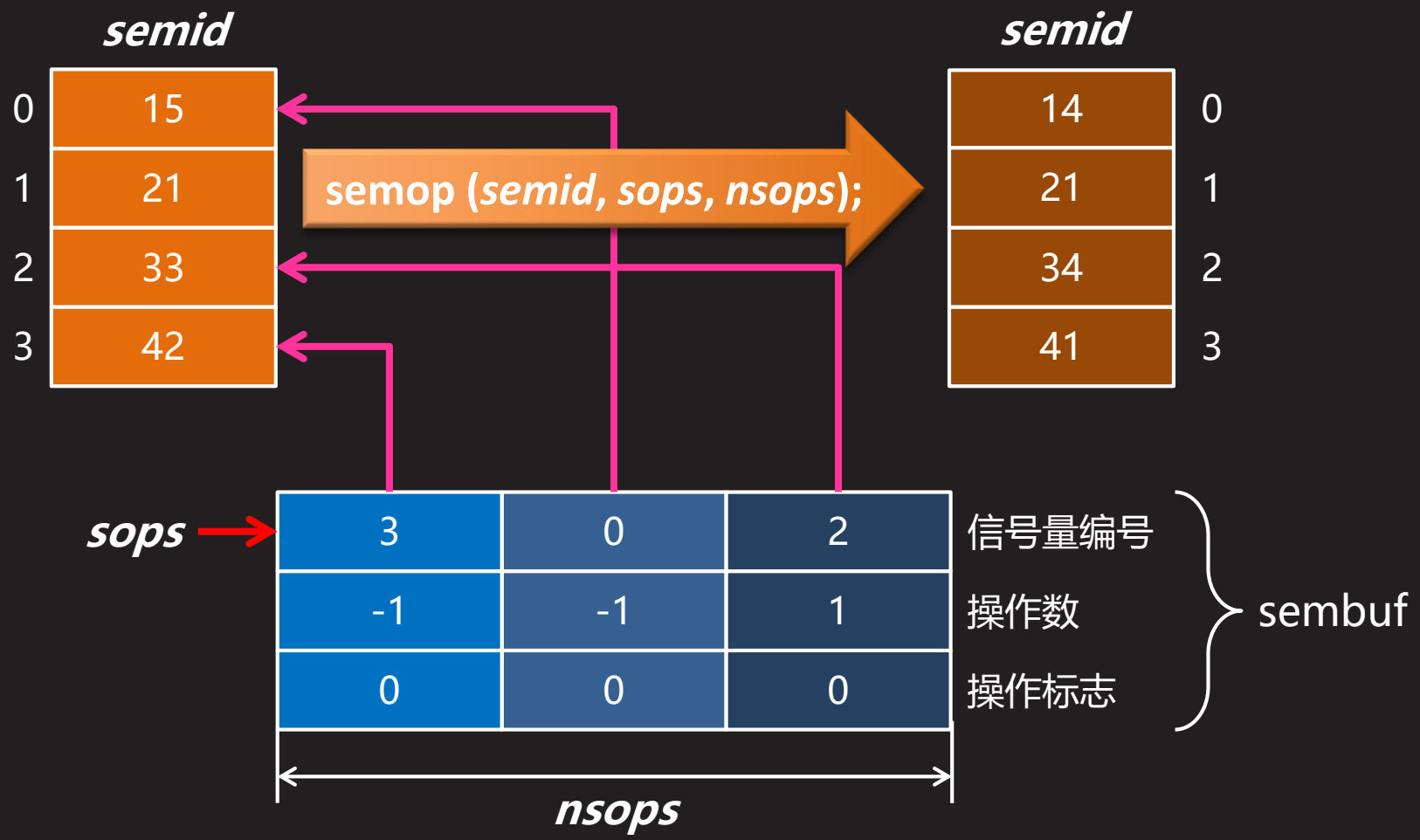
知识讲解

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);
    }

知识讲解

销毁/控制信号量集

- 销毁或控制信号量集

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

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

成功返回0（*cmd*取某些值时存在例外），失败返回-1

  - ***semid***：信号量集标识符
  - ***semnum***：信号量编号。只有针对信号量集中某个具体信号量的操作，才需要此参数；针对整个信号量集的操作，此参数将被忽略，置0即可
  - ***cmd***：控制命令

- 该函数的复杂性在于因控制命令***cmd***参数取值的不同，函数参数的个数、类型以及返回值的意义也会有所不同

知识讲解

销毁/控制信号量集（续1）

- 获取信号量集的属性
  - int semctl (int semid, 0, IPC_STAT, struct semid_ds* buf);
- 设置信号量集的属性
  - int semctl (int semid, 0, IPC_SET, struct semid_ds* buf);
  - 仅以下四个属性可以设置

```
semid_ds::sem_perm.uid  // 拥有者用户ID
semid_ds::sem_perm.gid  // 拥有者组ID
semid_ds::sem_perm.mode // 权限
```

- 删除信号量集
  - int semctl (int semid, 0, 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

知识讲解

销毁/控制信号量集（续2）

- 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);
- 获取信号量集的内核参数
  - int semctl (int semid, 0, IPC_INFO, struct seminfo* buf);

知识讲解

销毁/控制信号量集（续3）

- 例如
  - unsigned short array[] = {15, 21, 33, 42};
    if (semctl (semid, 0, SETALL, array) == -1) {
        perror ("semctl");
        exit (EXIT_FAILURE); }
  - int val = semctl (semid, semnum, GETVAL);
    if (val == -1) {
        perror ("semctl");
        exit (EXIT_FAILURE); }
    printf ("%d[%d]=%d\n", semid, semnum, val);
  - if (semctl (semid, 0, IPC_RMID) == -1) {
        perror ("semctl");
        exit (EXIT_FAILURE); }

Technology Tarena 达内科技

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	——	——

知识讲解

基于信号量的进程间通信

【参见：csem.c、gsem.c】

课堂练习

- 基于信号量的进程间通信

Technology Tarena 达内科技

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`=semphore)
  - 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>

知识讲解

Technology Tarena 达内科技

总结和答疑