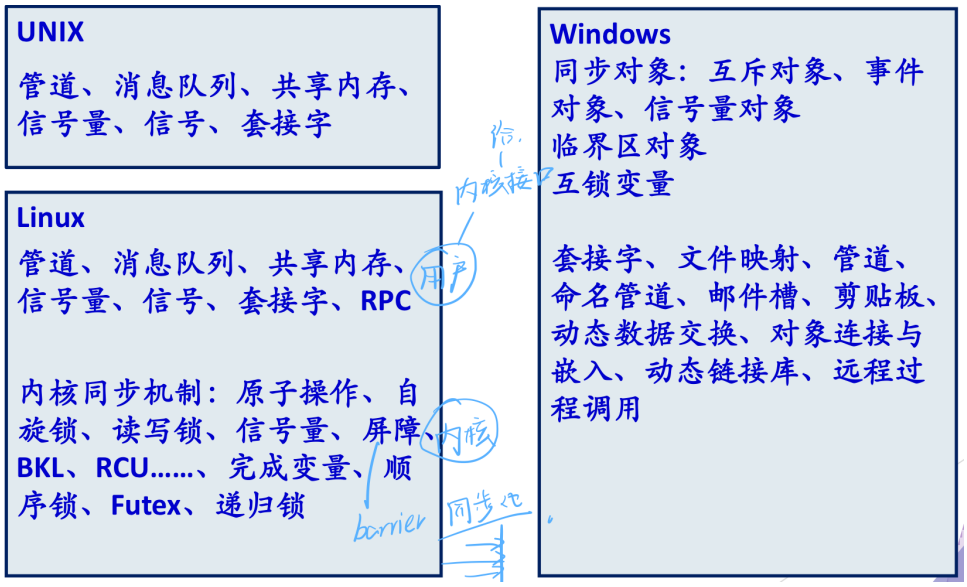
Lecture11 进程间通信

Linux 的 IPC 机制列表

1. 原子操作
2. 管道
3. 信号
4. 套接字
5. 信号量
6. 消息队列
7. 共享内存
8. 自旋锁
9. 读写锁
10. 屏障
11. 完成变量
12. 顺序锁
13. RCU 机制
14. Futex
15. 递归锁

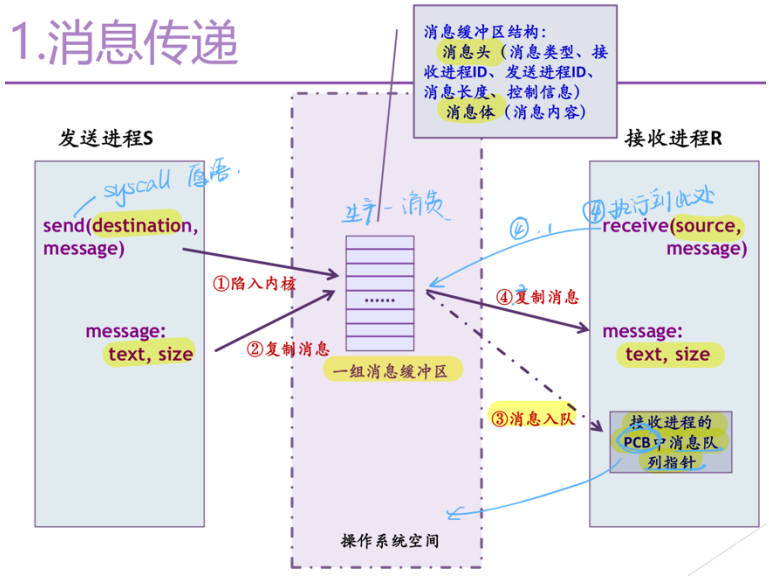
要求：
（1）了解概念并说明其应用；
（2）理解在Linux中的实现；
（3）应用举例（包括代码实现）

进程间同步/通信实例



进程间通信（IPC，Inter-Process Communication）

1. 基本思想
   1) 为什么需要通信机制？
      a) 信号量及管程的不足
      b) 多处理器情况下原语失效
   2) 进程通信机制
      a) 消息传递 send & receive 原语
      b) 适用于分布式系统、基于共享内存的多处理机系统、单处理机系统
      c) 可以解决进程间的同步问题、通信问题（支持大量消息的传递）
2. 基本通信方式
   1) 消息传递
      a) 基本机制



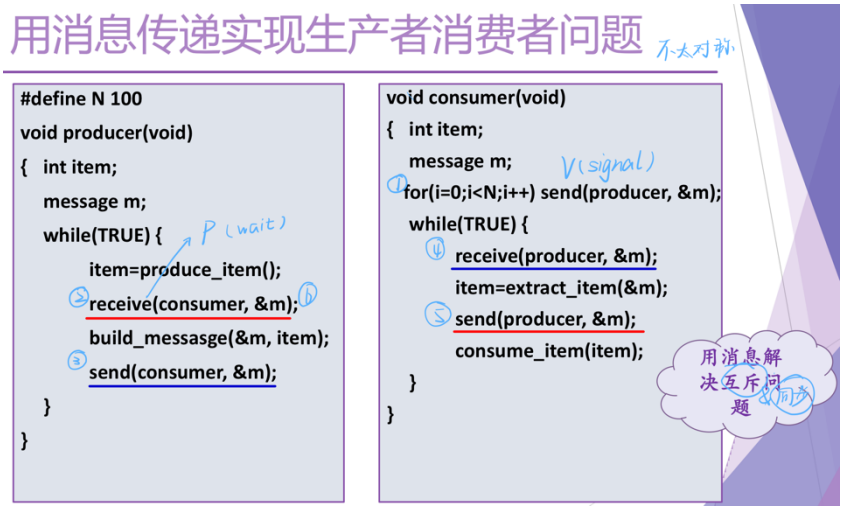
      b) 用 P、V 操作实现 Send 原语

```
Send（destination, message）
Begin
  根据destination找接收进程;
  如果未找到，出错返回;

  申请空缓冲区P(buf-empty);
    P(mutex1);
        取空缓冲区;
    V(mutex1);

  把消息从message处复制到空缓冲区;
```

```
  P(mutex2);
        把消息缓冲区挂到接收进程的消息队列;
  V(mutex2);
  V(buf-full);
End.

buf-empty初值为n
buf-full初值为0
mutex1初值为1
mutex2初值为1
```

思考题：receive 原语的实现

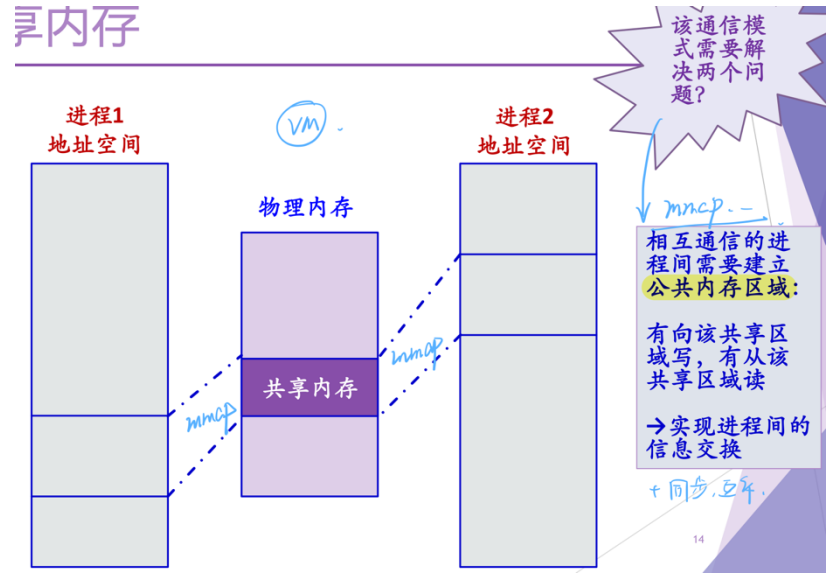
c) 用消息传递解决生产者消费者问题



d) 实现消息传递机制需要考虑的其他问题

- 缓冲形式：信箱、无缓冲、无限缓冲
- 消息丢失（如在网络和分布式系统中）、身份识别、效率

2) 共享内存



3) 管道（pipe）

a) 简介

- 利用一个缓冲传输介质——内存或文件连接两个相互通信的进程

- ✓ 字符流方式写入读出
- ✓ 先进先出顺序
- ✓ 管道通信机制必须提供的协调能力（互斥、同步以及能够判断对方进程是否存在）

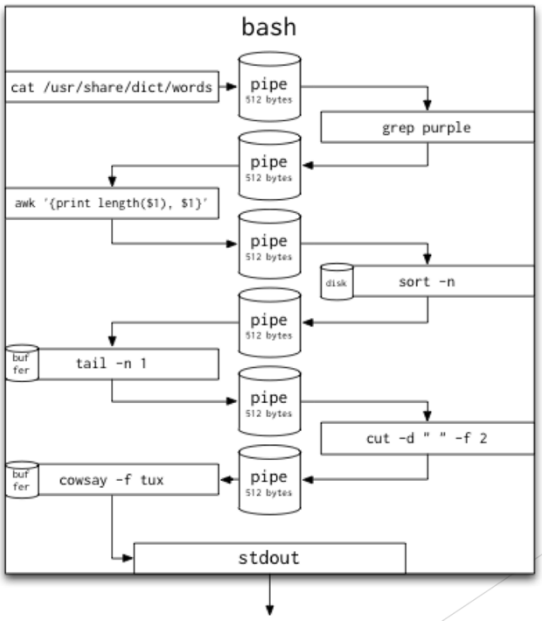
b) 设计目标

高内聚、低耦合；它以一种“链式模型”来串接不同的程序或者不同的组件，让它们组成一条工作流；给定一个完整的输入，经过各个组件的先后协同处理，得到唯一的最终输出

c) 应用

```
cat /usr/share/dict/words |     # Read in the system's
dictionary.
grep purple |                   # Find words containing
'purple'
awk '{print length($1), $1}' |  # Count the letters in each
word
sort -n |                       # Sort lines ("${length} ${word}")
tail -n 1 |                     # Take the last line of the
input
cut -d " " -f 2 |               # Take the second part of each
line
cowsay -f tux                   # Put the resulting word into
Tux's
                                # mouth
```

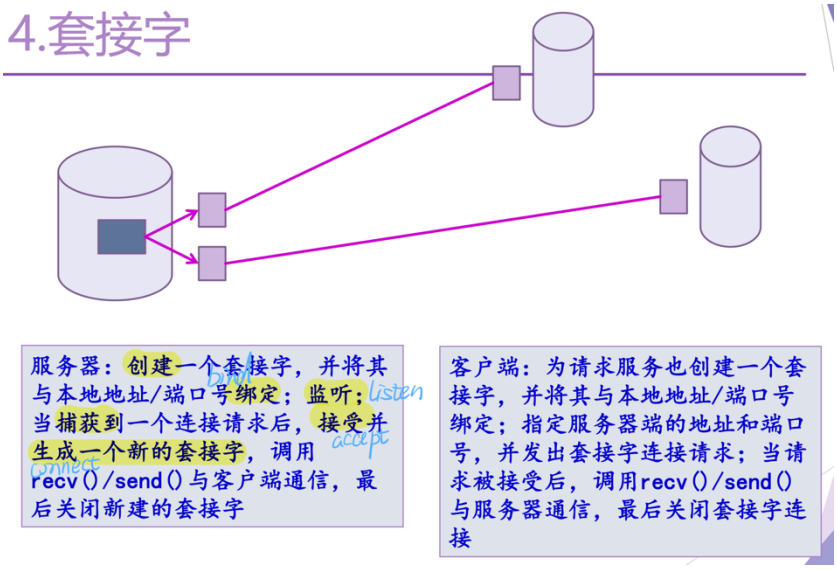
```
 ___________
< unimpurpled >
 -----------
  \
   \
       .-----.
      |o_o |
      | :_/  |
     //    \ \
    (|      | )
   /'\_    _/`\
   \__)=(__/
```



d) 无名管道和命名管道

	无名管道	命名管道
服务对象	进程及其子孙进程	任意关系的进程
实现机制	逻辑上：管道文件 物理上：利用高速缓冲区，与外设无关	逻辑上：管道文件 物理上：依赖于文件系统实现，作为特殊文件
永久性	创建在内存中 临时存在（没有磁盘映像）	存在于文件系统中 可供以后使用（有一个磁盘i-节点）
名称	无文件名，用文件描述符存取	有文件名
主要操作	pipe() write() read()	mkfifo fopen() write() read()

4) 套接字



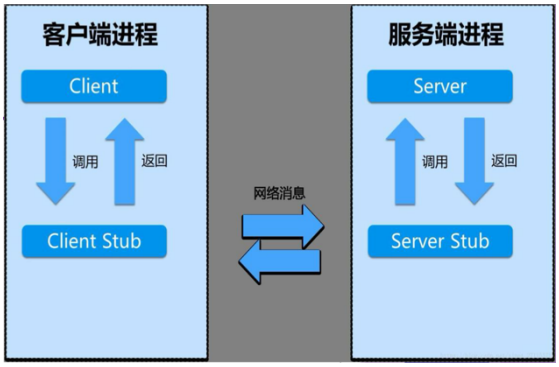
5) 远程过程调用(RPC)

a) 简介

- 定义
  - ✓ RPC（Remote Procedure Call）是一种技术思想，屏蔽网络编程细节，像调用本地方法一样调用远程方法

- 应用

  ✓ 应用访问量增加和业务增加时，单机无法承受，可以根据不同的业务拆分成互不关联的应用，分别部署在不同的机器上，应用与应用相互调用，此时需要用到 RPC。解耦服务、扩展性强、部署灵活，主要解决分布式系统中，服务与服务之间的调用问题



b) 一般过程

- 客户端 Client 通过本地调用的方式，调用远程接口服务
- 客户端存根 Client Stub 接收到调用后，将调用信息对象进行序列化，组装成网络传输的二进制消息体
- 客户端 Client 通过 Socket 将消息发送到远程服务端
- 服务端存根 Server Stub 收到消息后，对网络信息对象进行反序列化解码
- 服务端存根 Server Stub 根据解码结果，调用服务端本地的接口服务
- 本地接口服务执行，并将处理结果返回给服务端存根 Server Stub
- 服务端存根 Server Stub 将返回结果对象进行序列化，组装成消息体
- 服务端 Server 再通过 Socket 将消息发送到客户端
- 客户端存根 Client Stub 收到结果消息后，对网络信息对象进行序列化解码
- 客户端 Client 拿到最终接口处理结果

c) 具体应用

- 电商系统

  将电商系统拆分成用户服务、商品服务、优惠券服务、支付服务、

订单服务、物流服务、售后服务等，这些服务之间都相互调用，内部调用使用 RPC，同时每个服务都可以独立部署，独立上线

- 车载以太网

  SOME/IP(Scalable service-Oriented MiddlewarE over IP)是基于 IP 的可扩展的面向服务的中间件，2011 年由 BMW 集团的 Dr. Lars Völker 设计，是一种面向服务的车载以太网通信协议，采用 Client/Server 通信架构，其中 Server 是服务提供者，Client 是服务消费者。根据服务接口类型，使用远程过程调用（Remote Procedure Call）机制，实现控制器之间的服务调用，通过数据序列化和反序化（Serialization/Deserialization）使数据在网络中传输，通过服务发现 SD（Service Discovery）机制来实现服务的动态配置

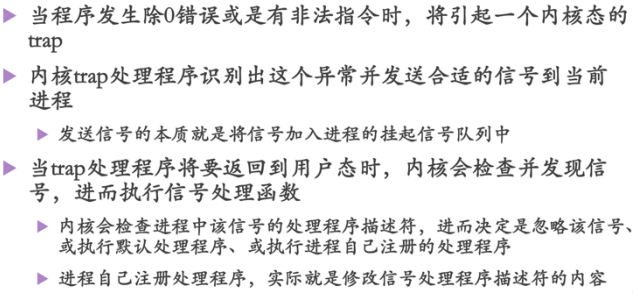
6) 信号（signal）

a) 概念

- 软中断信号简称为信号，用来通知进程发生了异步事件
- 信号本质上是在软件层次上对中断机制的一种模拟
  - ✓ 进程之间可以互相通过系统调用发送软中断信号
  - ✓ 内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件

b) 信号的产生

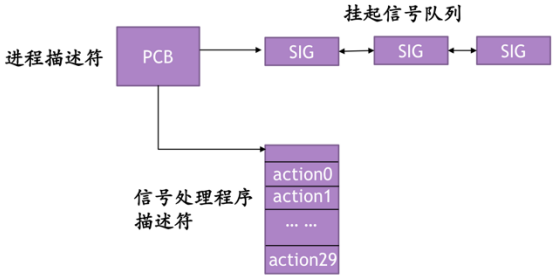
- 异常
  - ✓ 当一个进程出现异常（比如试图执行一个非法指令，除 0，浮点溢出等），内核通过向进程发送一个信号来通知进程异常的发生
  - ✓ 例子



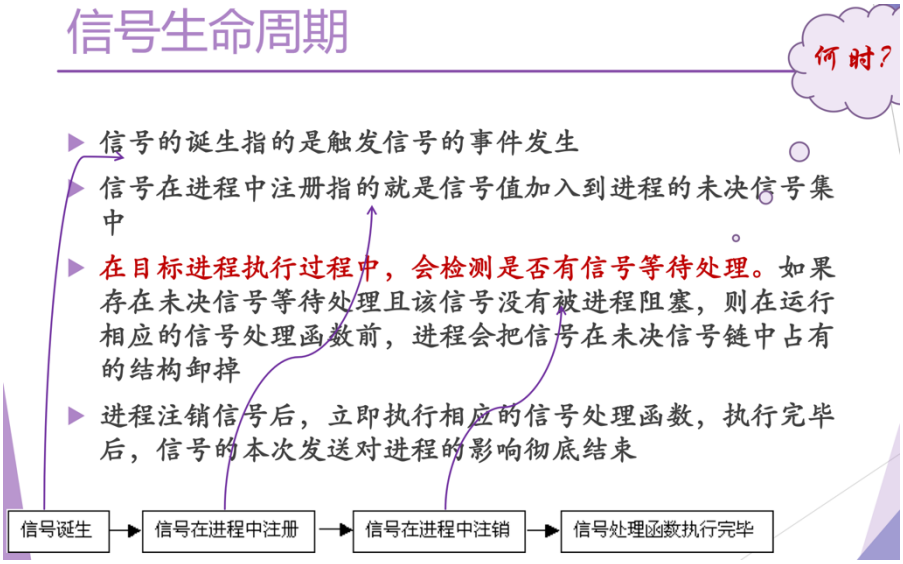
- 其他进程
  - ✓ 一个进程可以通过 kill 或是 sigsend 系统调用向另一个进程或一个进程组发送信号。一个进程也可以向自身发送信号
- 作业控制
  - ✓ 发送信号给那些想要读或写终端的后台进程。比如 shell 使用信号来管理前台和后台进程
- 通知
  - ✓ 一个进程也许要求能被通知某些事件的发生。比如设备已经就绪等待 I/O 操作
- 闹钟
  - ✓ 定时器产生的信号，由内核发送给进程
- ……

c) 信号的处理

- 收到信号的进程对各种信号有三类处理方法
  - ✓ 对于需要处理的信号，进程可以指定处理函数，由该函数来处理
  - ✓ 忽略某个信号，对该信号不做任何处理，就像未发生过一样
  - ✓ 对该信号的处理保留系统的默认处理方式
- 检测并响应信号的时机
  - ✓ 进程由于系统调用、中断或异常进入内核，当该进程从内核返回用户空间前
  - ✓ 例子：进程被调度程序选中，由于检测到信号从而提前返回到用户空间
- 关键数据结构



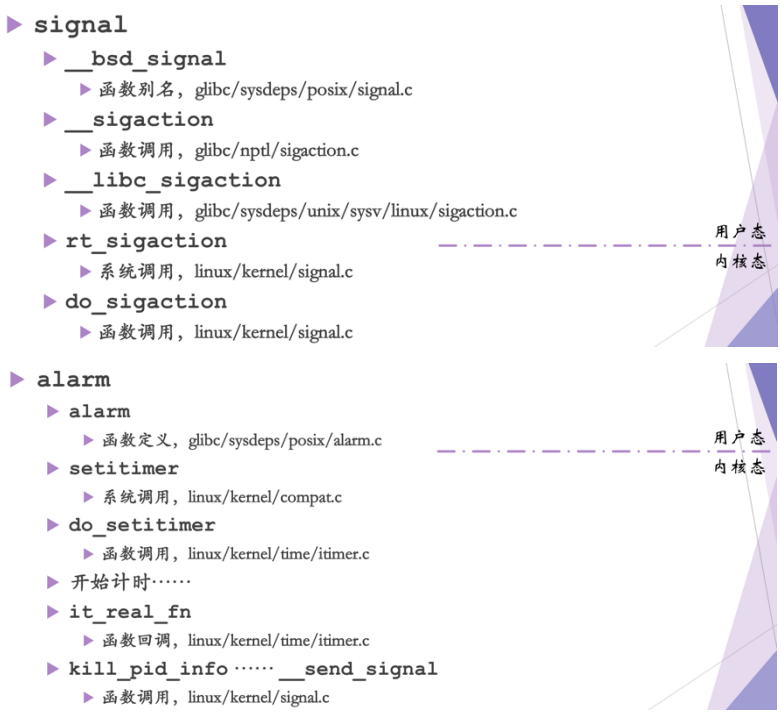
d) 信号生命周期




```c
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("BOOM!\n");
    exit(0);
  }
}
```

```c
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

e) Linux 信号的实现



- 信号处理（x86）
  - `prepare_exit_to_usermode`, `exit_to_usermode_loop`
    - 系统调用返回，linux/arch/x86/entry/common.c
  - `do_signal`, `handle_signal`
    - 处理信号并设置栈帧，linux/arch/x86/kernel/signal.c
  - `iret`, `sysexit`, `sysret`指令
    - 返回用户态
  - 执行signal handler
  - `rt_sigreturn`
    - 系统调用

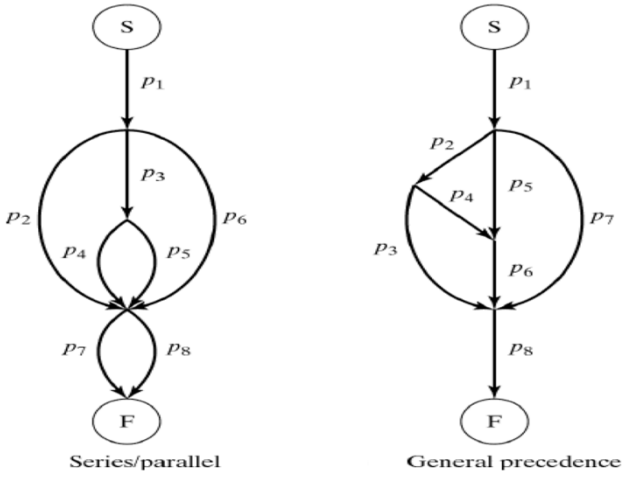
经典 IPC 问题

经典问题列表

1. 纯同步问题
2. 另类PV操作问题
3. 食品供应问题
4. 三峡大坝问题
5. 狒狒过峡谷问题
6. 睡眠理发师问题
7. 第二类读者写者问题
8. 复杂的消息缓冲问题
9. 利用信号量解决资源管理问题

同步问题解题要求：
1. 分析问题的关键点（同步互斥关系）；
2. 给出问题的解法（代码或伪代码）并说明。

1. 纯同步问题



2. 另类 PV 操作问题

P(s):

s.count --;

if(s<0) then

将本进程插入相应队列末尾等待

V(s):

s.count ++;

if(s<=0) then

从相应等待队列末尾唤醒一个进程，将其插入就绪队列

1) 这样定义 PV 操作是否有问题
2) 用这样的 PV 操作实现 N 个进程竞争使用某一共享变量的互斥机制
3) 对 b 的解法，有无效率更高的方法

3. 食品供应问题

某商店有两种食品 A 和 B， 最大数量各为 m 个。该商店将 A、B 两种食品搭配出售，每次各取一个。为避免食品变质，遵循先到食品先出售的原则。有两个食品公司分别不断地供应 A、B 两种食品(每次一个)。为保证正常销售，当某种食品的数量比另一种的数量超过 k(k<m)个时，暂停对数量大的食品进货。试用 P、V 操作解决上述问题中的同步和互斥关系。

4. 三峡大坝问题

由于水面高度不同,有 160~175 米的落差,所以三峡大坝有五级船闸,T1~T5。由上游驶来的船需经由各级船闸到下游;由下游驶来的船需经由各级船闸到上游。假设只能允许单方向通行（此假设与实际情况不符，实际为双向各五级船闸)。试用 P、V 操作正确解决三峡大坝船闸调度问题。

5. 狒狒过峡谷问题

1) 一个主修人类学、辅修计算机科学的学生参加了一个研究课题，调查是否可以教会非洲狒狒理解死锁。他找到一处很深的峡谷，在上边固定了一根横跨峡谷的绳索，这样狒狒就可以攀住绳索越过峡谷。同一时刻，只要朝着相同的方向就可以有几只狒狒通过。但如果向东和向西的狒狒同时攀在绳索上那么会产生死锁（狒狒会被卡在中间），由于它们无法在绳索上从另一只的背上翻过去。如果一只狒狒想越过峡谷，它必须看当前是否有别的狒狒正在逆向通行。利用信号量编写一个避免死锁的程序来解决该问题。不考虑连续东行的狒狒会使得西行的狒狒无限制地等待的情况。

2) 重复上一个习题，但此次要避免饥饿。当一只想向东去的狒狒来到绳索跟前，但发现有别的狒狒正在向西越过峡谷时，它会一直等到绳索可用为止。但在至少有一只狒狒向东越过峡谷之前，不允许再有狒狒开始从东向西过峡谷。

6. 睡眠理发师问题

1) 理发店里有一位理发师,一把理发椅和N把供等候理发的顾客坐的椅子。如果没有顾客，则理发师便在理发椅上睡觉；当一个顾客到来时，他必须先唤醒理发师。如果顾客到来时理发师正在理发，则如果有空椅子，可坐下来等，否则离开。试用P、V操作解决睡眠理发师问题。

2) 思考：N个理发师的解决方案

7. 第二类读者写者问题（写者优先）

试用信号量及 P、V 操作解决写者优先问题，要求：

a. 多个读者可以同时进行读

b. 写者必须互斥(只允许一个写者写，也不能读者写者同时进行)

c. 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。

8. 复杂的消息缓冲问题

消息缓冲区为 k 个，有 m 个发送进程，n 个接收进程，每个接收进程对发送来的消息都必须取一次。试用 P、V 操作解决发送进程和接受进程之间的正确通信问题。

9. 利用信号量解决资源管理问题

考虑具有如下特征的共享资源：当使用该资源的进程小于 3 个时，新申请资源的进程可以立刻获得资源；当三个资源都被占用后，只有当前使用资源的三个进程都释放资源后，其他申请资源的进程才能够获得资源。

1) 由于需要使用计数器来记录有多少进程正在使用资源和等待资源，而这些计数器自身也需要互斥执行修改动作的共享资源，所以可以采用如下的程序：

```
1 semaphore mutex = 1, block = 0;
2 int active = 0, waiting = 0;
3 boolean must_wait = false;
4
5  P(mutex);
6     if(must_wait) {
7          ++waiting;
8         V(mutex);
9         P(block);
10        P(mutex);
11        --waiting;
12 }
13    ++active;
14    must_wait = active == 3;
15    V(mutex);
16
17 /* 临界区：对获得的资源进行操作
   */
18
19 P(mutex);
20 --active;
21 if(active == 0) {
22   int n;
23   if (waiting < 3) n = waiting;
24   else n = 3;
25   while( n > 0 ) {
26        V(block);
27        --n;
28    }
29      must_wait = false;
30 }
31    V(mutex);
```

这个程序看起来没有问题：所有对共享数据的访问均被临界区所保护，进程在临界区中执行时不会自己阻塞，新进程在有三个资源使用者存在时不能使用共享资源，最后一个离开的使用者会唤醒最多 3 个等待着的进程

a) 这个程序仍不正确，解释其出错的位置；

b) 假如将第六行的 if 语句更换为 while 语句，是否解决了上面的问题？有什么难点仍然存在？

2) 现在考虑第一问的正确解法。解释下面这个程序的工作方式，为什么这种工作方式是正确的？

a) 这个程序不能完全避免新到达的进程插到已有等待进程前得到资源，但是至少使这种问题的发生减少了。给出一个例子

b) 这个程序是一个使用信号量实现并发问题的通用解法样例，这种解法被称作“I'll Do it for You”（由释放者为申请者修改计数器）模式。解释这种模式。

```
1 semaphore mutex = 1, block = 0;
2 int active = 0, waiting = 0;
3 boolean must_wait = false;
4
5 P(mutex);
6   if(must_wait) {
7     ++waiting;
8     V(mutex);
9     P(block);
10 } else {
11  ++active;
12   must_wait = active == 3;
13   V(mutex);
14 }
15
16 /* 临界区：对获得的资源进行操作 */
17
18   P(mutex);
19   --active;
20   if(active == 0) {
21   int n;
22   if (waiting < 3) n = waiting;
23   else n = 3;
24   waiting -= n;
25   active = n;
26   while( n > 0 ) {
27     V(block);
28     --n;
29    }
30   must_wait = active == 3;
31 }
32   V(mutex);
```

3) 现在考虑上一问的另一个正确解法。解释这个程序的工作方式，为什么这种工作方式是正确的？

a) 这个方法在可以同时唤醒进程个数上是否和上一题的解法有所不同？为什么？

b) 这个程序是一个使用信号量实现并发问题的通用解法样例，这种解法被称作“Pass the Baton”（接力棒传递）模式。解释这种模式。

```
1 semaphore mutex = 1, block = 0;
2 int active = 0, waiting = 0;
3 boolean must_wait = false;
4
5 P(mutex);
6 if(must_wait) {
7     ++waiting;
8     V(mutex);
9     P(block);
10    --waiting;
11 }
12 ++active;
13 must_wait = active == 3;
14 if(waiting > 0 && !must_wait)
15    V(block);
16
17 else  V(mutex);
18
19 /* 临界区：对获得的资源进行操作 */
20
21  P(mutex);
22  --active;
23  if(active == 0)
24      must_wait = false;
25     if(waiting > 0
&& !must_wait)
26   V(block);
27
28 else V(mutex);
```