

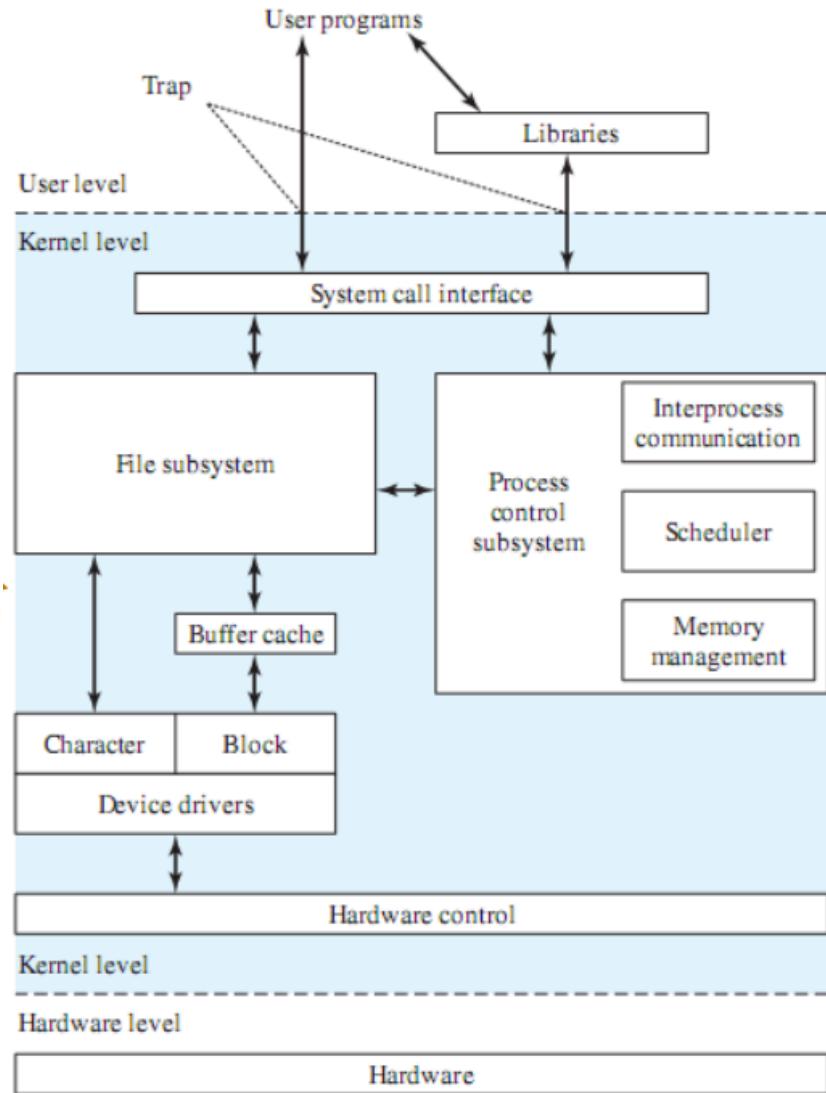
# 高级大数据解析

Module 3: 深入理解Spark的运行机制  
(2) 进程的同步、通信与死锁

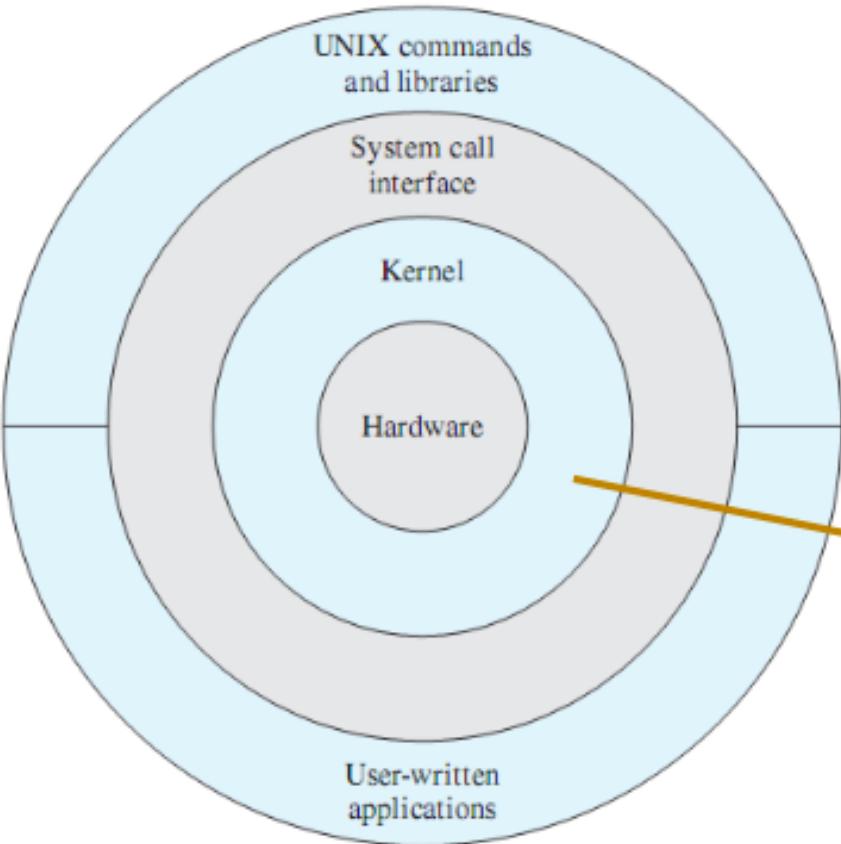
复习一下处理器管理

# UNIX 操作系统的体系结构

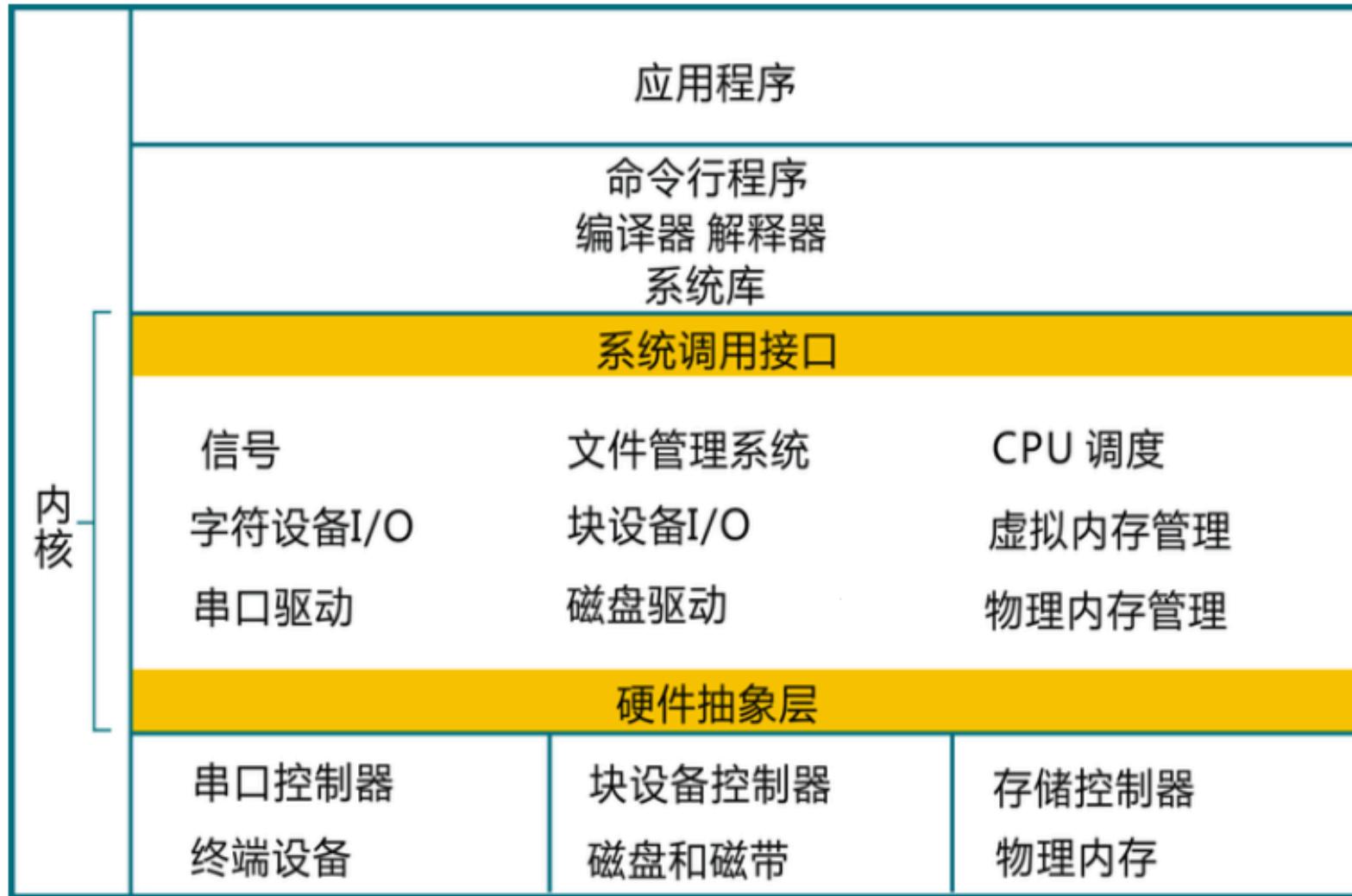
## 内核结构



## 层次结构

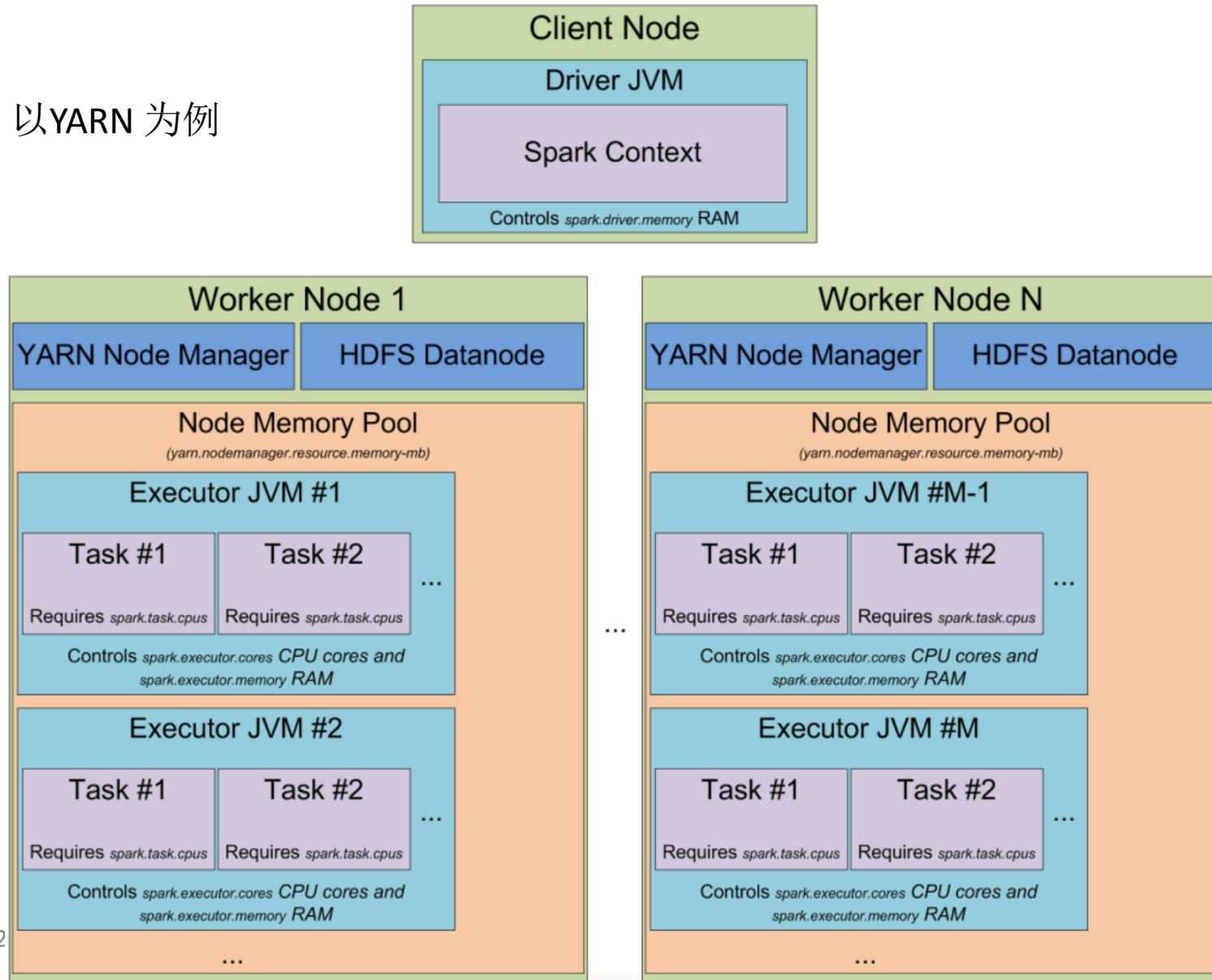


# Linux操作系统内核



# Spark 集群模式JVM分配

以YARN为例



## 操作系统做了什么? (1/4)

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    puts("hello world");
    return 0;
}
```

## 操作系统做了什么? (3/4)

- 执行程序的第一条指令，发生[缺页异常](#)
- 操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行程序
- **helloworld**程序执行**puts**函数（系统调用），在显示器上写一字符串
- 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

## 操作系统做了什么? (2/4)

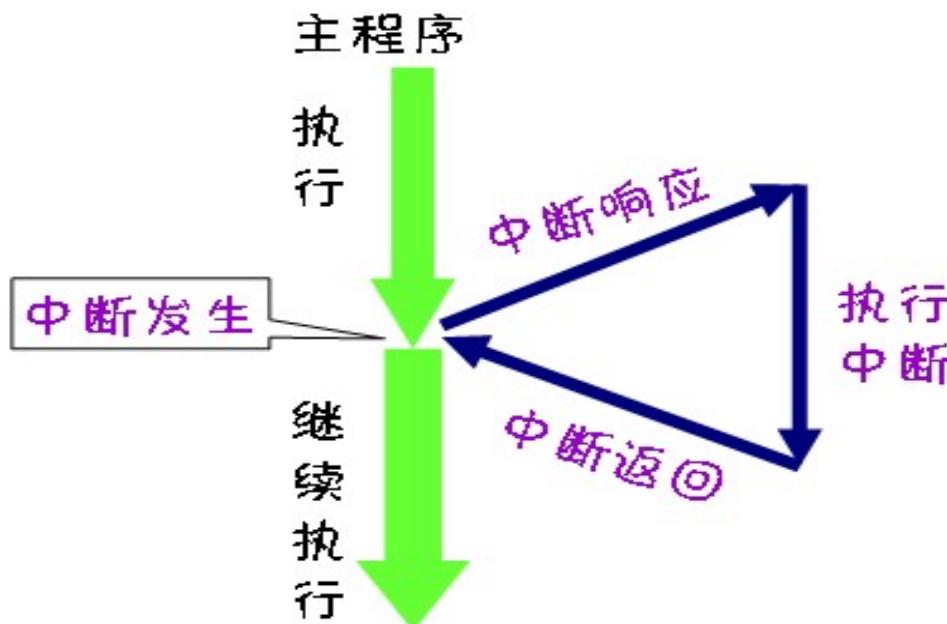
- 用户告诉操作系统执行程序([如何告知?](#))
- 操作系统：找到程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址 ([文件格式?](#))
- 操作系统：创建一个新的进程，并将可执行文件映射到该进程结构，表示由该进程执行程序
- 操作系统：为程序设置CPU上下文环境，并跳到程序开始处 ([假设调度程序选中hello程序](#))

## 操作系统做了什么? (4/4)

- 操作系统：控制设备的进程告诉设备的窗口系统它要显示字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
- 视频硬件将像素转换成显示器可接收的一组控制/数据信号
- 显示器解释信号，激发液晶屏
- **OK! ! !** 我们在屏幕上看到了“**hello world**”

## 2.2 处理器的中断技术

- 中断是指程序执行过程中，遇到急需处理的事件时，暂时中止CPU上现行程序的运行，转去执行相应的事件处理程序，待处理完成后返回原程序被中断处或调度其他程序执行的过程。



# 为什么要中断

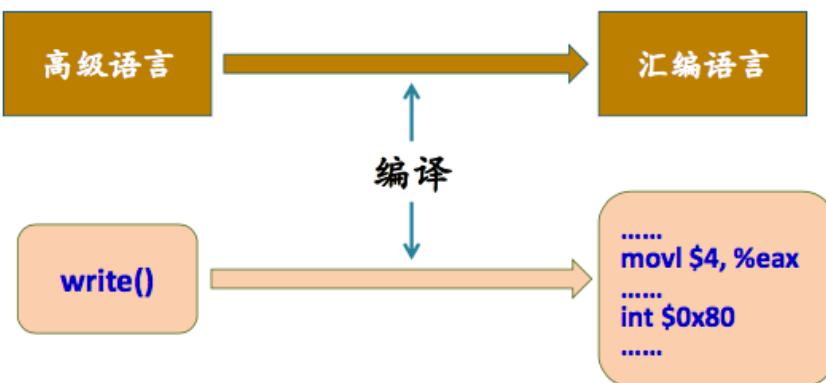
---

- 请求系统服务——系统调用
- 实现并发工作
- 处理突发事件
- 满足实时要求
- . . .

中断对于操作系统的重要性 就像汽车发动机、  
飞机引擎

- →→ 操作系统是由“中断驱动”或“事件驱动”的

## 系统调用举例(1/3)



## 系统调用举例(3/3)

```
1. .section .data
2. output:
3.   .ascii "Hello!\n"
4. output_end:
5.   .equ len, output_end - output

6. .section .text
7. .globl _start
8. _start:
9.   movl $4, %eax
10.  movl $1, %ebx
11.  movl $output, %ecx
12.  movl $len, %edx
13.  int $0x80
14. end:
15.   movl $1, %eax
16.   movl $0, %ebx
17.   int $0x80
```

汇编语  
言视角

# eax存放系统调用号

# 引发一次系统调用

# !这个系统调用的作用?

## 系统调用举例(2/3)

```
#include <unistd.h>
int main(){
    char string[7] = {'H','e','l','l','o','!','\n'};
    write(1, string, 7);
    return 0;
}
```

输出结果: Hello!

高级语  
言视角

## 系统调用的执行过程

当CPU执行到特殊的陷入指令时：

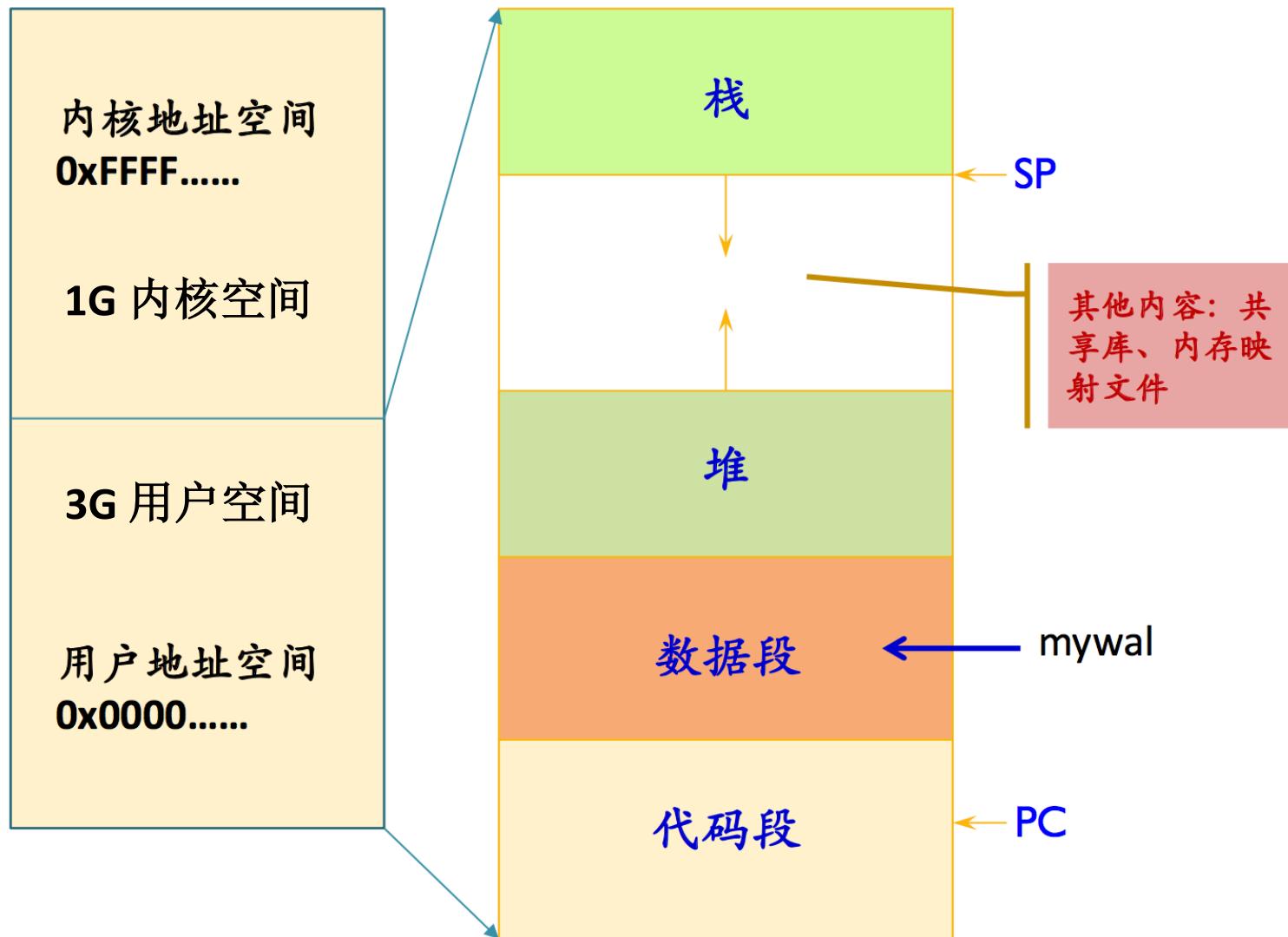
- **中断/异常机制**: 硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序
- **系统调用总入口程序**: 保存现场；将参数保存在内核堆栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- **执行系统调用例程**
- **恢复现场，返回用户程序**

# 进程的定义和性质

---

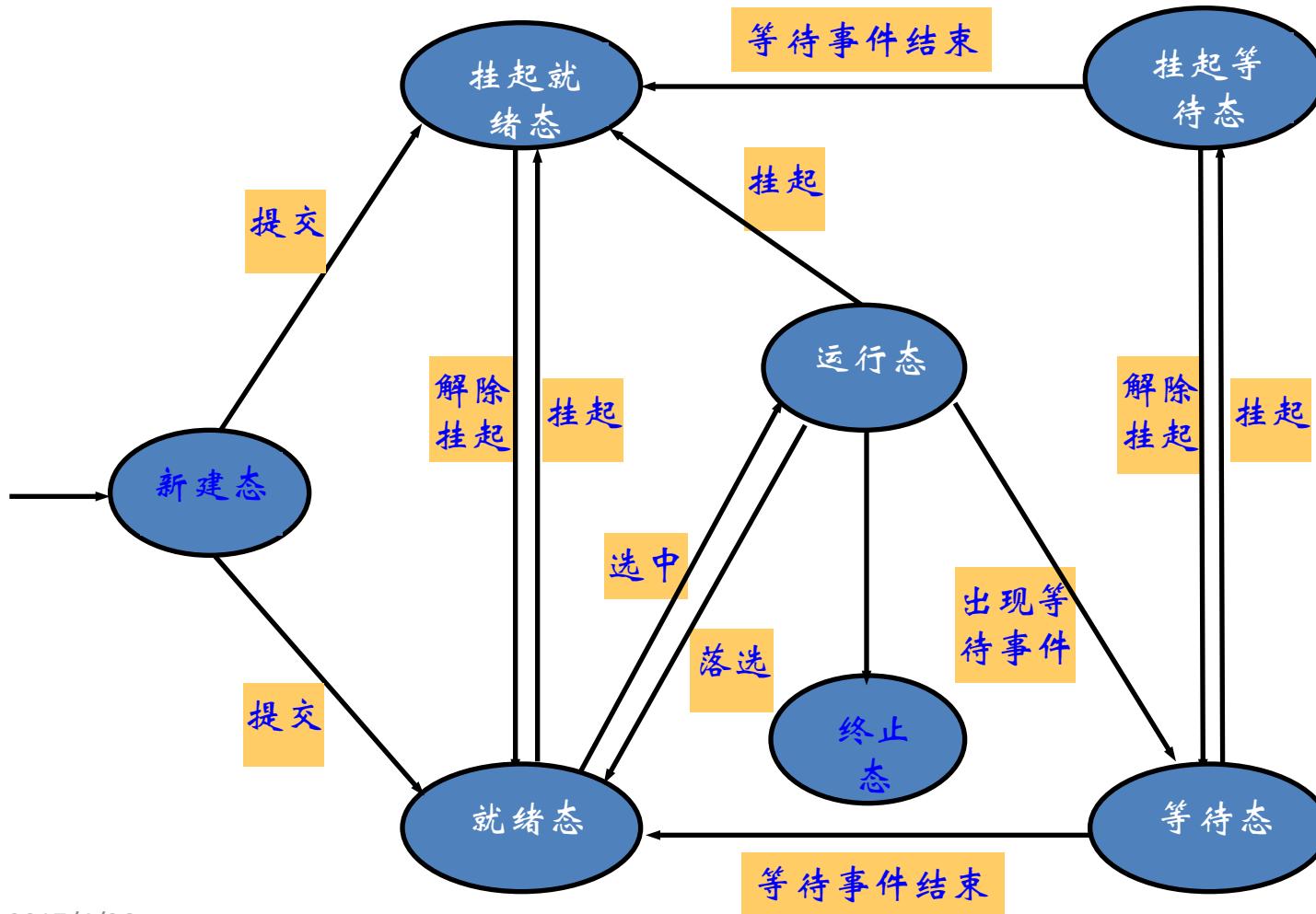
- 进程是可并发执行的程序在某个数据集合上的一次计算活动，也是操作系统进行资源分配和保护的基本单位。
- 进程是一个既能用来共享资源，又能描述程序并发执行过程的一个基本单位。

# 操作系统给每个进程都分配了一个地址空间



# 进程的挂起

- 进程可“交换”到外存，解决资源不足

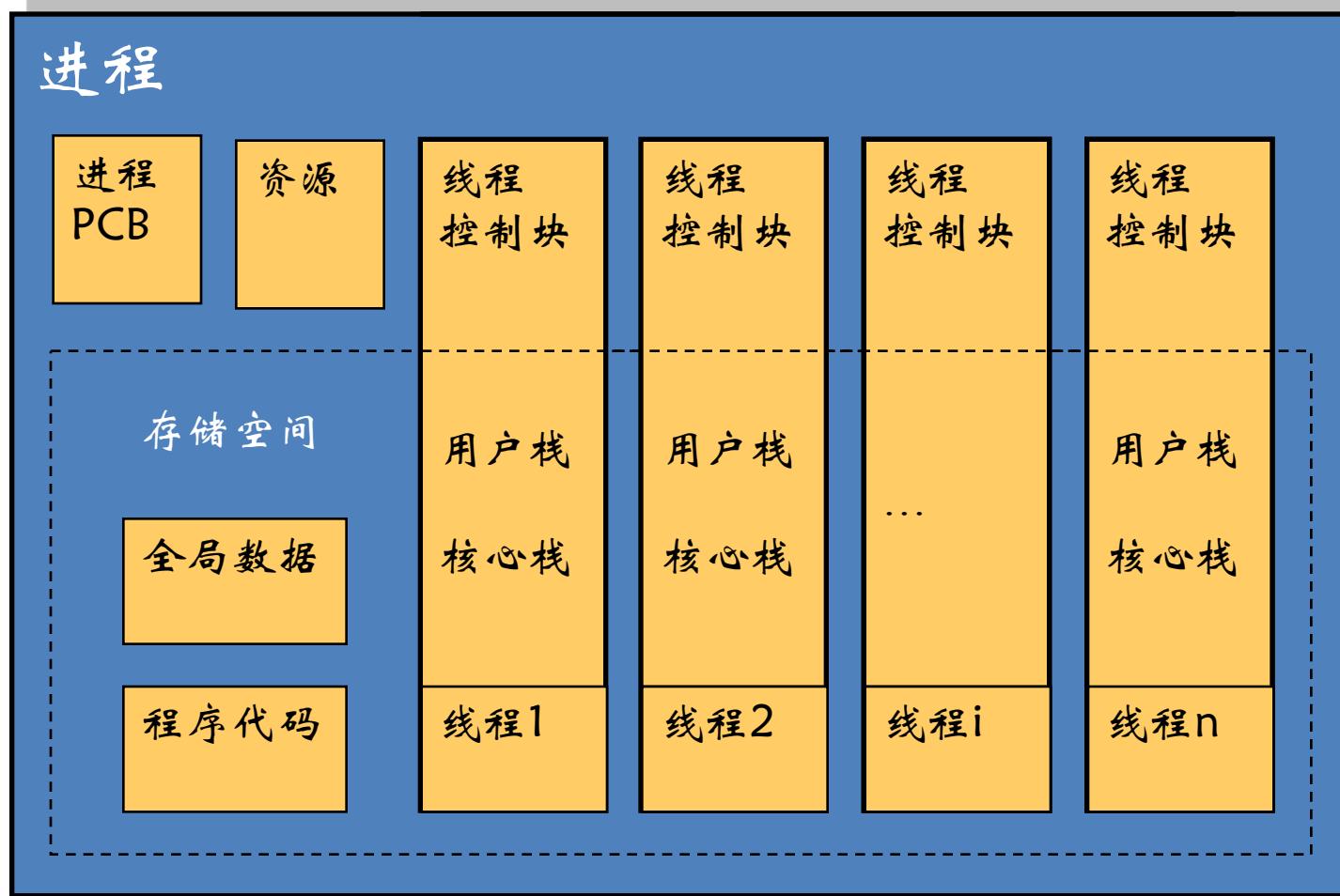


# 引入线程的动机

---

- 引入进程的动机
  - 使多个程序可以并发执行，以改善资源使用率和提高系统效率
- 引入线程的动机
  - 减少进程并发执行时所付出的时空开销，使得并发粒度更细、并发性更好
- 解决思路
  - 分离进程的两项功能：“独立分配资源”与“被调度分派执行”
  - 进程：资源分配单位，无需频繁切换
  - 线程：调度单位，体量小，频繁切换

# 多线程环境中的进程与线程



# 常用的调度算法

---

- 先来先服务(FCFS-First Come First Serve)
- 最短作业优先(SJF-Shortest Job First)
- 最短剩余时间优先(SRT-Shortest Remaining Time Next)
- 最高响应比优先(HRRN-Highest Response Ratio Next)

# Spark集群对应用的调度策略

---

Spark集群的调度分为

- 应用间调度
  - 调度策略1: 资源静态分区
  - 调度策略2: 动态共享CPU cores
  - 调度策略3: 动态资源申请
- 应用内调度: 同一个应用内的不同jobs
  - 默认采用FIFO的调度策略
  - 也可以被配置为"fair sharing"

# Job Scheduling

- [Overview](#)
- [Scheduling Across Applications](#)
  - [Dynamic Resource Allocation](#)
    - [Configuration and Setup](#)
    - [Resource Allocation Policy](#)
      - [Request Policy](#)
      - [Remove Policy](#)
    - [Graceful Decommission of Executors](#)
  - [Scheduling Within an Application](#)
    - [Fair Scheduler Pools](#)
    - [Default Behavior of Pools](#)
    - [Configuring Pool Properties](#)

## Overview

Spark has several facilities for scheduling resources between computations. First, recall that, as described in the [cluster mode overview](#), each Spark application (instance of `SparkContext`) runs an independent set of executor processes. The cluster managers that Spark runs on provide facilities for [scheduling across applications](#). Second, *within* each Spark application, multiple “jobs” (Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network. Spark includes a [fair scheduler](#) to schedule resources within each `SparkContext`.

## Scheduling Across Applications

When running on a cluster, each Spark application gets an independent set of executor JVMs that only run tasks and store data for that application. If multiple users need to share your cluster, there are different options to manage allocation, depending on the cluster manager.

The simplest option, available on all cluster managers, is *static partitioning* of resources. With this approach, each application is given a maximum amount of resources it can use, and holds onto them for its whole duration. This is the approach used in Spark’s [standalone](#) and [YARN](#) modes, as well as the [coarse-grained Mesos mode](#). Resource allocation can be configured as follows, based on the cluster type:

- **Standalone mode:** By default, applications submitted to the standalone mode cluster will run in FIFO (first-in-first-out) order, and each application will try to use all available nodes. You can limit the number of nodes an application uses by setting the `spark.cores.max` configuration property in it, or change the default for applications that don’t set this setting through `spark.deploy.defaultCores`. Finally, in addition to controlling cores, each application’s `spark.executor.memory` setting controls its memory use.
- **Mesos:** To use static partitioning on Mesos, set the `spark.mesos.coarse` configuration property to `true`, and optionally set `spark.cores.max` to limit each application’s resource share as in the standalone mode. You should also set `spark.executor.memory` to control the executor memory.
- **YARN:** The `--num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster (`spark.executor.instances` as configuration property), while `--executor-memory` (`spark.executor.memory` configuration property) and `--executor-cores` (`spark.executor.cores` configuration property) control the resources per executor. For more information, see the [YARN Spark Properties](#).

# 应用间调度 (1)

---

- 调度策略1: 资源静态分区

- 资源静态分区是指整个集群的资源被预先划分为多个partitions，资源分配时的最小粒度是一个静态的partition。
  - 根据应用对资源的申请需求为其分配静态的partition(s)是Spark支持的最简单的调度策略。
  - 若Spark集群配置了static partitioning的调度策略，则它对提交的多个应用间默认采用FIFO顺序进行调度，
  - 每个获得执行机会的应用在运行期间可占用整个集群的资源，这样做明显不友好，所以应用提交者通常需要通过设置spark.cores.max来控制其占用的core/memory资源。

# 应用间调度 (2)

---

- 调度策略2: 动态共享CPU cores
  - 若Spark集群采用Mesos模式，可以采用该策略；
  - 每个应用仍拥有各自独立的cores/memory，但当应用申请资源后并未使用时（即分配给应用的资源当前闲置），其它应用的计算任务可能会被调度器分配到这些闲置资源上。
  - 当提交给集群的应用有很多是非活跃应用时（即它们并非时刻占用集群资源），这种调度策略能很大程度上提升集群资源利用效率。
  - 风险是：若某个应用从非活跃状态转变为活跃状态时，且它提交时申请的资源当前恰好被调度给其它应用，则它无法立即获得执行的机会。

# 应用间调度 (3)

---

- 动态资源申请
  - 它允许根据应用的workload动态调整其所需的集群资源。
  - 若应用暂时不需要它之前申请的资源，则它可以先归还给集群，
  - 当它需要时，可以重新向集群申请。当Spark集群被多个应用共享时，这种按需分配的策略显然也是非常有优势的。

# 应用内调度 (1)

---

在应用内部，每个action (e.g. save, collect) 以及计算这个action结果所需要的一系列tasks 被统称为一个"job"

- FIFO的调度策略

- 每个job被分解为不同的stages;
- 当多个job各自的stage所在的线程同时申请资源时，第1个job的stage优先获得资源；
- 如果job queue头部的job恰好是需要最长执行时间的job时，后面所有的job均得不到执行的机会，这样会导致某些job(s)饿死的情况。

# 应用内调度 (2)

---

- "fair sharing"
  - Spark对不同jobs的stages提交的tasks采用Round Robin的调度方式，如此，所有的jobs均得到公平执行的机会。因此，即使某些short-time jobs本身的提交时间在long jobs之后，它也能获得被执行的机会，从而达到可预期的响应时间
  - 要启用fair sharing调度策略，需要在spark配置文件中将spark.scheduler.mode设置为FAIR。
  - fair sharing调度也支持把不同的jobs聚合到一个pool，不同的pools赋予不同的执行优先级。
  - 这是FIFO和fair sharing两种策略的折衷策略，既能保证jobs之间的优先级，也能保证同一优先级的jobs均能得到公平执行的机会

# How to set?

To enable the fair scheduler, simply set the `spark.scheduler.mode` property to FAIR when configuring a SparkContext:

```
val conf = new SparkConf().setMaster(...).setAppName(...)  
conf.set("spark.scheduler.mode", "FAIR")  
val sc = new SparkContext(conf)
```

Without any intervention, newly submitted jobs go into a *default pool*, but jobs' pools can be set by adding the `spark.scheduler.pool` "local property" to the SparkContext in the thread that's submitting them. This is done as follows:

```
// Assuming sc is your SparkContext variable  
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

After setting this local property, *all* jobs submitted within this thread (by calls in this thread to `RDD.save`, `count`, `collect`, etc) will use this pool name. The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user. If you'd like to clear the pool that a thread is associated with, simply call:

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

# 3.1 并发进程

---

- 顺序程序设计 ( Sequential Programming)
  - 执行的顺序性
  - 环境的封闭性
  - 结果的确定性
  - 过程的可再现性
- 缺点：效率很低
- 引入并发进程：同一个时间段内多个进程/线程同时运行
  - 和并行的区别？
  - 并发的基础是什么？

# 从进程的特征出发

并发是所有问题的基础

并发是操作系统设计的基础

并发

- 程序的执行是间断性的
- 进程的相对执行速度不可预测

共享

- 进程/线程之间的制约性

不确定性

- 并发程序执行的结果与其执行的相对速度有关，是不确定的

# 串行 VS 并发

- 设计一个音乐播放器，一边下载音乐，一边播放
- 串行设计：

```
char buffer[1024];
While(true)
{
    download(buff);
    play(buff);
}
```

下载的时候没法播放，播放的时候没法下载！！！

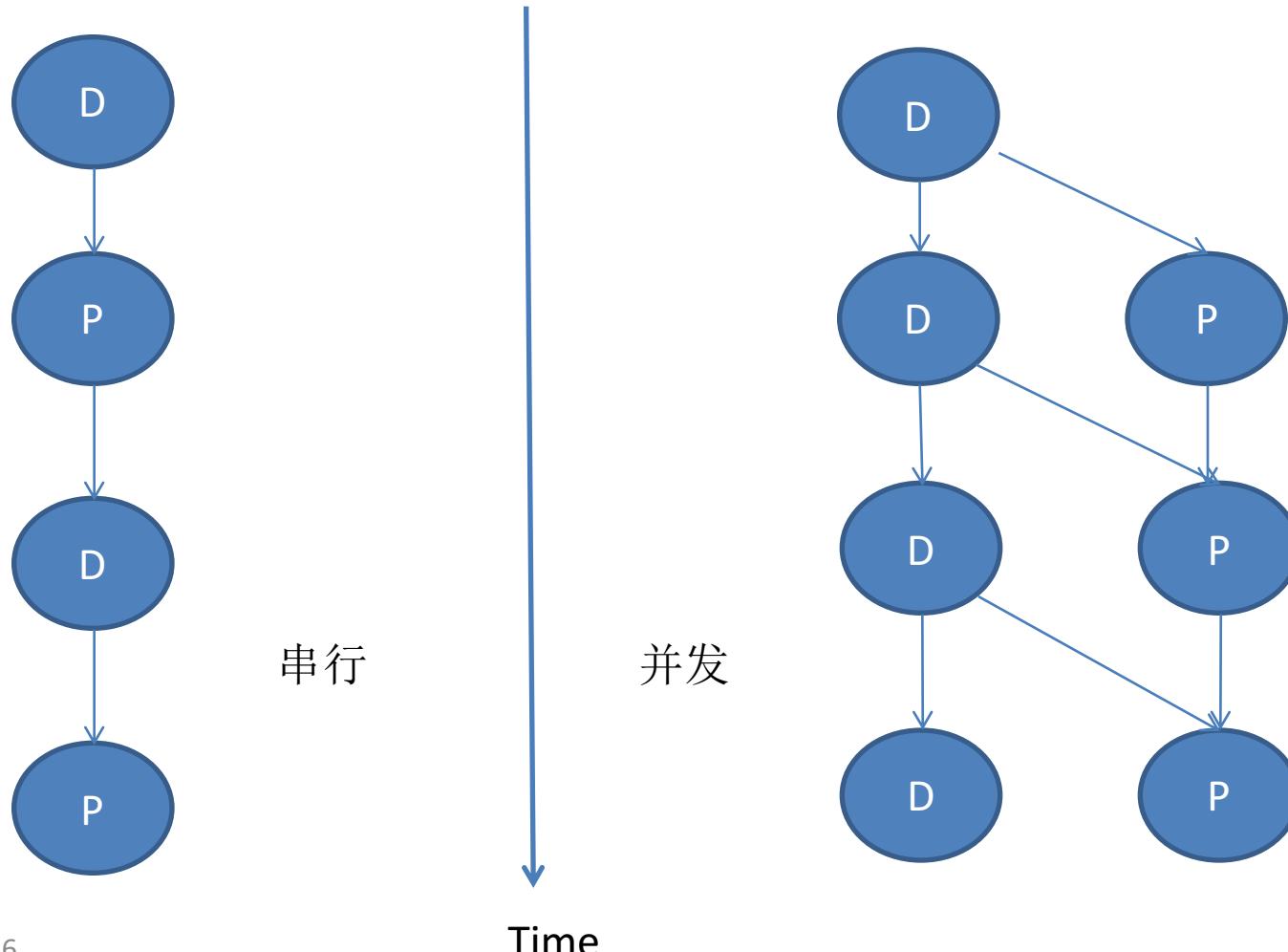
- 并发设计

```
Char buffer[1024];
While(true)
{
    download(buff);
    write(sharedBuff, buff);
}
```

```
Char buffer[1024];
While(true)
{
    read(sharedBuff,buff)
    play(buff);
}
```

# 串行 VS 并发

- 并发大大提高系统效率



# 并发带来的问题

- 数据一致性问题

- 丢失修改

甲售票点读出某航班的机票余额A,设A=16.

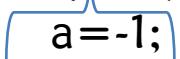
乙售票点读出同一航班的机票余额A,也为16.

甲售票点卖出一张机票,修改余额 $A \leftarrow A - 1$ .所以A为15,把A写回数据库.

乙售票点也卖出一张机票,修改余额 $A \leftarrow A - 1$ .所以A为15,把A写回数据库.

- 不可重复读

甲计算: if( $a > 0$ ) { if( $a > 5$ ) b=1; else b=2;} else b=3;

乙计算:       a=-1;

- 读脏数据

思考: 数据库并发带来的问题?

# 死锁与饥饿

---

- 资源竞争引起死锁与饥饿
- 死锁， deadlock
  - 一组进程都获得部分资源，又要申请彼此占有的资源 → 死锁
  - 死锁条件：
    - 互斥条件、不可剥夺、请求和保持、循环等待
- 饥饿， starvation
  - 进程一直抢占不到资源
  - SJF 调度方法（最短作业优先算法）。

# 并发进程的关系(1)

---

- 完全无关 Bernstein 条件
  - 条件  $R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1) \cup W(P_1) \cap W(P_2) = \emptyset$
  - 并发进程执行与时间无关，不会产生任何错误

$R(p_i) = \{a_1, a_2, \dots, a_n\}$ , 表示程序  $p_i$  在执行期间引用的变量集；  
 $W(p_i) = \{b_1, b_2, \dots, b_m\}$ , 表示程序  $p_i$  在执行期间改变的变量集。
- 竞争
  - 彼此不知道对方的存在
  - 竞争独占性资源
  - 需要互斥
  - 问题：死锁、饥饿

# 并发进程的关系(2)

---

- 共享合作
  - 进程不知道彼此的存在
  - 有共享数据：变量、文件、数据库等
  - 互斥需求
  - 问题：数据一致性问题、死锁、饥饿等
- 通信合作
  - 进程知道彼此的存在
  - 通过通信协作
  - 同步需求
  - 死锁、饥饿

# 进程的同步

## 进程同步：synchronization

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务

具体地说，一个进程运行到某一点时，要求另一伙伴进程为它提供消息，在未获得消息之前，该进程进入阻塞态，获得消息后被唤醒进入就绪态

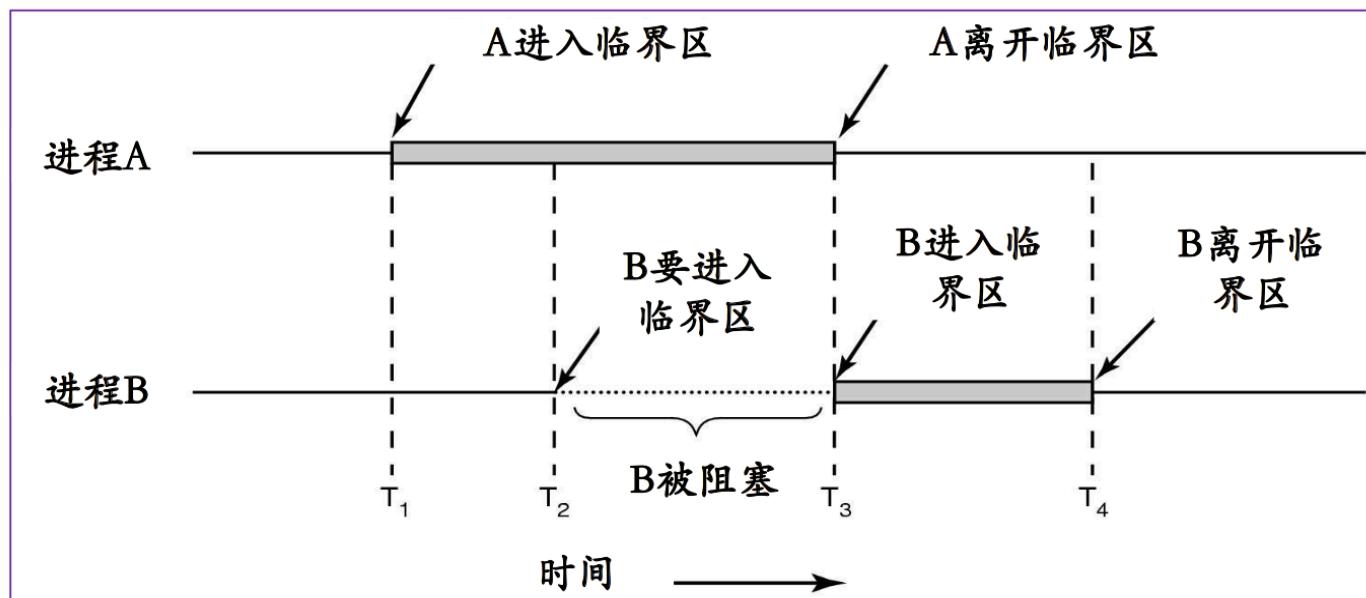
## 3.2 临界区

---

- 进程互斥：
  - 由于各进程要求使用共享资源(变量、文件等)，而这些资源需要排他性使用；
  - 各进程之间竞争使用这些资源；
- 临界资源：critical resource  
系统中某些资源一次只允许一个进程使用，称这样的资源为**临界资源或互斥资源或共享变量**
- 临界区(互斥区)：critical section(region)  
各个进程中对某个临界资源(共享变量)实施操作的程序片段

# 临界区和互斥

- 进程必须以互斥的方式执行临界区
- 若干个进程共享一个变量的相关临界区， 调度三原则：
  - 一次至多有一个进程进入临界区
  - 如果已有进程在临界区， 其他试图进入的进程必须等待
  - 进入临界区的进程需要在有限时间内退出临界区
- 临界区的调度原则总结：互斥使用、有空让进、忙则等待、优先等待、择一而入、算法可行。



# 临界区实现

- 如何设计临界区，从而满足三原则？
- 尝试一（软件方法）

```
bool inside1 = false;           //P1 不在其临界区内
bool inside2 = false;           //P2 不在其临界区内
cobegin
process P1() {
    while(inside2);           //等待
    inside1 = true;
    {临界区};
    inside1 = false;
}
process P2() {
    while(inside1);           //等待
    inside2 = true;
    {临界区};
    inside2 = false;
}
coend
```

讨论的问题：该管理方法是否可以实现临界区管理？ **nope**

# 临界区实现

- 尝试二

```
bool inside1 = false;           //P1 不在其临界区内
bool inside2 = false;           //P2 不在其临界区内
cobegin
process P1() {
    inside1 = true;
    while(inside2);           //等待
    {临界区};
    inside1 = false;
}
process P2() {
    inside2 = true;
    while(inside1);           //等待
    {临界区};
    inside2 = false;
}
coend
```

讨论的问题：该管理方法是否可以实现临界区管理？ **nope**

# 临界区实现-软件算法

- Peterson算法

```
bool inside[2];
inside[0] = false;inside[1] = false;
enum {0,1} turn;
cobegin
process P0() {
    inside[0] = true;
    turn = 1;
    while(inside[1] && turn == 1);
    {临界区};
    inside[0] = false;
}
process P1() {
    inside[1] = true;
    turn = 0;
    while(inside[0] && turn == 0);
    {临界区};
    inside[1] = false;
}
coend
```

# 临界区实现 – 硬件算法

---

- 关中断?
  - 不适用于多处理器
  - 关中断的时间长会影响性能
  - 不适用于多 CPU 系统
- 利用原子指令
  - 测试并建立指令 TS(Test and Set)
  - 对换指令 (Swap):

# 临界区实现 - 硬件算法

- 测试并建立指令

```
bool s = true;  
cobegin  
process Pi() {  
    while(! TS(s)); //上锁  
    {临界区};  
    s = true; //开锁  
}  
coend
```

```
bool TS(bool &x) {  
    if(x) {  
        x = false;  
        return true;  
    }  
    else  
        return false;  
}
```

# 临界区实现 – 硬件算法

- 对换指令

```
bool lock = false;  
cobegin  
process Pi() {  
    bool keyi = true;  
    do {  
        SWAP(keyi, lock);  
        | while(keyi); //上锁  
        | 临界区;  
        SWAP(keyi, lock); //开锁  
    }  
coend
```

```
void SWAP(bool &a, bool &b) {  
    bool temp = a;  
    a = b;  
    b = temp;  
}
```

在 Intel 80x86 中，对换指令称为 XCHG 指令。

# 典型的同步机制

- 信号量及P、V操作
- 管程
- 锁 + 条件变量

# 3.3 信号量与PV操作

---

3.3.1 同步与同步机制

3.3.2 信号量与PV操作

3.3.3 信号量实现互斥

3.3.4 信号量解决五个哲学家吃通心面问题

3.3.5 信号量解决生产者-消费者问题

3.3.6 记录型信号量解决读者-写者问题

3.3.7 记录型信号量解决理发师问题

# 同步和同步机制

---

- 生产者消费者问题(producer-consumer problem)
  - 生产者：不断产生数据的进程
  - 消费者：不断接受使用数据的进程
- 生产者进程与消费者进程需要同步
  - 生产速度大于消费速度：生产者需要等待
  - 消费速度大于生产速度：消费者需要等待
- 进程需要同步
  - 以生产者消费者问题为例学习进程同步机制

# 生产者消费者问题(1)

## • 形式化描述

- n个生产者，m个消费者，通过k个单位的循环缓冲区连接
- Pi和Ci是并发进程，缓冲区不满，Pi就生产产品投入缓冲区，缓冲区不空，Ci就从缓冲区取走并消耗产品

```
process producer(void) {  
    while (true) { //无限循环  
        {produce an item in nextp};//生产一个产品  
        if (counter==k) //缓冲满时，生产者睡眠  
            sleep(producer);  
        buffer[in]=nextp;//将一个产品放入缓冲区  
        in=(in+1)%ok; //指针推进  
        counter++; //缓冲内产品数加1  
        if(counter==1) //缓冲为空，加进一件产品  
            wakeup(consumer);//并唤醒消费者  
    }  
}
```

```
process consumer(void) {  
    while (true) { //无限循环  
        if (counter==0) //缓冲区空，消费者睡眠  
            sleep(consumer);  
        nextc=buffer[out];//取一个产品到nextc  
        out=(out+1)%ok; //指针推进  
        counter--; //取走一个产品，计数减1  
        if(counter==k-1) //缓冲满了，取走一件产品并唤醒  
            wakeup(producer); //醒生产者  
        {consume the item in nextc};//消耗产品  
    }  
}
```

# 生产者消费者问题 (2)

---

- 生产者和消费者通过共享资源协作
- 临界区导致一系列问题
  - 结果的不确定性：数据一致性错误
  - 进程永远等待：死锁
- 使用前述临界区管理方法？
  - 软件算法太过复杂
  - 硬件方法虽然简单，但采用忙等待。。。。

# 信号量与PV操作

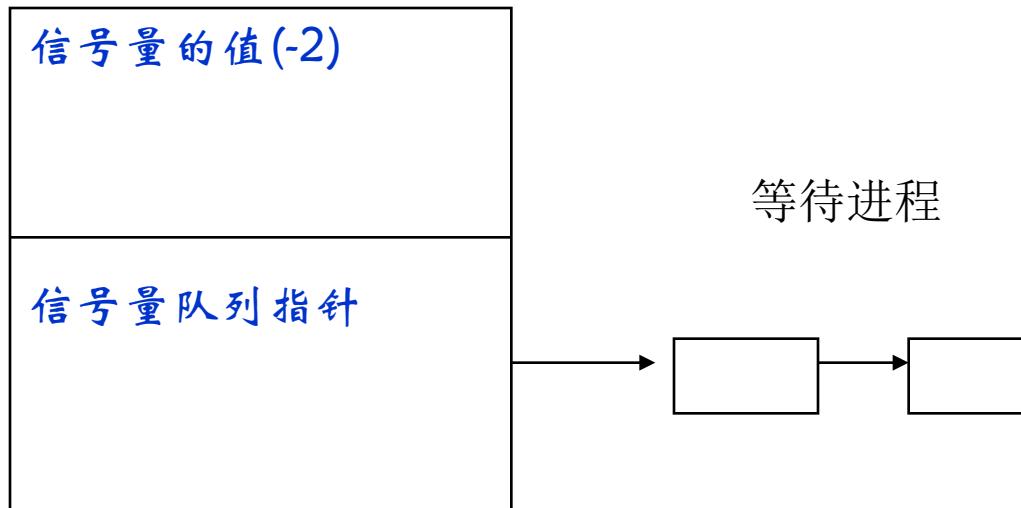
- 1965年E.W.Dijkstra提出了新的同步工具--信号量和P、V操作
- 信号量(semaphore)
  - 旗语
  - 特殊变量，交互进程在关键点上一直等待直到接收到特殊变量值
- P操作原语和V操作原语
  - 对信号量进行特殊操作
  - P：检测，V：增量
  - 对信号量可以实施的操作：初始化、P和V(P、V分别是荷兰语的test(proberen)和increment(verhogen))



```
struc semaphore
{
    int count;
    queueType queue;
}
```

# 信号量

- 信号量的实现



- 信号量的分类
  - 按用途分：公用、私有
  - 按取值分：二元、一般

# 一般信号量

---

- 设 $s$ 为一个信号量
- $P(s)$ : 将信号量 $s$ 减去1，若结果小于0，则调用 $P(s)$ 的进程被置成等待信号量 $s$ 的状态
- $V(s)$ : 将信号量 $s$ 加1，若结果不大于0，则释放一个等待信号量 $s$ 的进程
- $P(s)$ 和 $V(s)$ 都是原语操作

# P、V操作定义

```
P(s)
{
    s.count--;
    if (s.count < 0)
    {
        该进程状态置为阻塞状态;
        将该进程插入相应的等待队列s.queue末尾;
        重新调度;
    }
}
```

down, semWait

```
V(s)
{
}
```

```
s.count++;
if (s.count <= 0)
{
    唤醒相应等待队列s.queue中等待的一个进程;
    改变其状态为就绪态，并将其插入就绪队列;
}

```

```
struct semaphore
{
    int count;
    queueType queue;
}
```

}

up, semSignal

# 一般信号量的推论

---

- 推论1：若信号量 $s$ 为正值，则该值等于在封锁进程之前对信号量 $s$ 可施行的P操作数
- 推论2：若信号量 $s$ 为负值，则其绝对值等于登记排列在该信号量 $s$ 队列之中等待的进程个数
- 推论3：通常，P操作意味着请求一个资源，V操作意味着释放一个资源。在一定条件下，P操作代表挂起进程操作，而V操作代表唤醒被挂起进程的操作

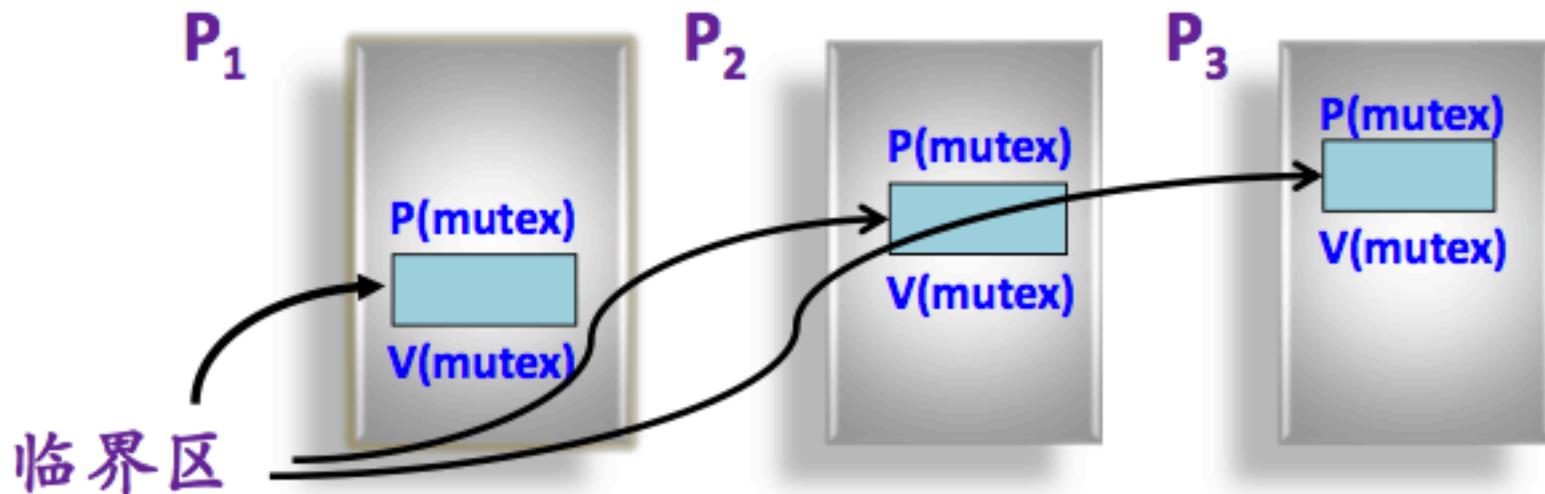
# 信号量实现互斥

---

- semaphore mutex;
- mutex=1;
- cobegin
- process Pi() { //i=1,...,n
- P(mutex);
- {临界区};
- V(mutex);
- }
- coend

# 用PV操作解决进程间互斥问题

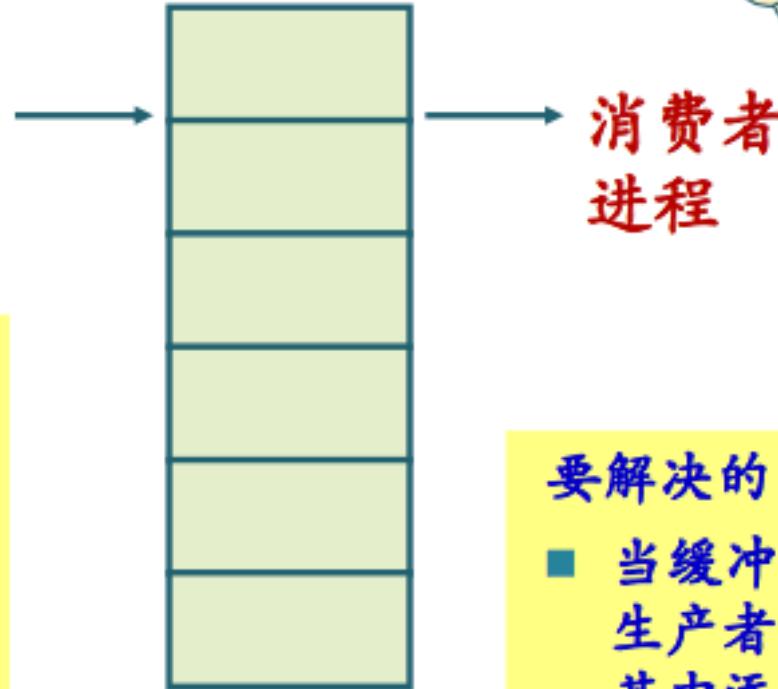
- 分析并发进程的关键活动，划定临界区
- 设置信号量 mutex，初值为1
- 在临界区前实施 P(mutex)
- 在临界区之后实施 V(mutex)



# 生产者/消费者问题

又称为  
有界缓冲  
区问题

生产者  
进程



## 问题描述:

- 一个或多个生产者生产某种类型的数据放置在缓冲区中
- 有消费者从缓冲区中取数据，每次取一项
- 只能有一个生产者或消费者对缓冲区进行操作

## 要解决的问题:

- 当缓冲区已满时，生产者不会继续向其中添加数据；
- 当缓冲区为空时，消费者不会从中移走数据

# 用信号量解决生产者/消费者问题

```
void producer(void)
{ int item;
  while(TRUE) {
    item=produce_item();
    P(&empty);
    P(&mutex);
    insert_item(item);
    V(&mutex)
    V(&full);
  }
}
```

```
void consumer(void)
{ int item;
  while(TRUE) {
    P(&full);
    P(&mutex);
    item=remove_item();
    V(&mutex);
    V(&empty);
    consume_item(item);
  }
}
```

```
#define N 100
typedef int semaphore;
semaphore mutex =1;
semaphore empty =N;
semaphore full = 0;
```

/\* 缓冲区个数 \*/  
/\* 信号量是一种特殊的整型数据 \*/  
/\* 互斥信号量：控制对临界区的访问 \*/  
/\* 空缓冲区个数 \*/  
/\* 满缓冲区个数 \*/

# 用信号量解决更多的同步问题

---

- 用信号量哲学家吃面问题
- 用信号量解决理发师问题
- 用信号量解决读者-写者问题
- 用信号量多个生产者、多个消费者、多个缓冲区

# 管程

---

3.4.1 管程和条件变量

3.4.2 管程的实现

3.4.3 管程解决进程同步问题

# 为什么引入管程

---

- 直接使用信号量很容易出错
  - 生产者消费者例子-p(full)和p(mutex)互换
- 管程， monitor
  - 把分散在各进程中的临界区集中起来进行管理；
  - 防止进程有意或无意的违法同步操作，
  - 编程语言支持

# 为什么引入管程？

问题

- 信号量机制的不足：程序编写困难、效率低

解决

- Brinch Hansen(1973)
- Hoare (1974)

方案

- 在程序设计语言中引入管程成分
- 一种高级同步机制

# 管程的定义

- 是一个特殊的模块
- 有一个名字
- 由关于共享资源的数据结构及在其上操作的一组过程组成

## □ 进程与管程

进程只能通过调用管程中的过程来间接地访问管程中的数据结构

**monitor example**

integer *i*;  
condition *c*;

变量

procedure *producer*( );

.

end;

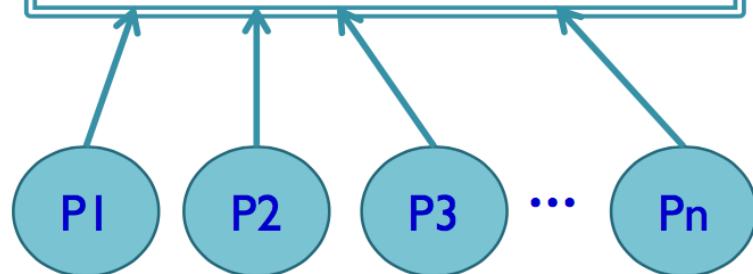
过程

procedure *consumer*( );

.

end;

end monitor;



# 管程的定义和特性

---

- 管程的定义
  - 由局部与自己的若干公共变量和所有访问这些公共变量的过程（临界区）所组成的软件模块
- 管程的特性
  - 共享性
    - 对外接口可以被多个进程共享
  - 安全性
    - 管程的局部变量只能由管程自己访问，管程也不访问任何外部变量
  - 互斥性
    - 一次只让一个进程（线程）进入管程

# 管程要保证什么？

- 作为一种同步机制，管程要解决两个问题 ?
- 互斥
  - 管程是互斥进入的
    - 为了保证管程中数据结构的数据完整性
  - 注意：**管程的互斥性是由编译器负责、保证的
- 同步
  - 管程中设置**条件变量及等待/唤醒操作**以解决同步问题
  - 可以让一个进程或线程等待在条件变量上等待（此时，应先释放管程的使用权），也可以通过发送信号将等待在条件变量上的进程或线程唤醒

# 应用管程时遇到的问题

是否会出现这样一种情况，有多个进程同时在管程中？

场景：

当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权

当后面进入管程的进程执行唤醒操作时（例如，P唤醒Q），管程中便存在两个同时处于活动状态的进程

如何解决？

三种处理方法：

- P等待Q执行 (Hoare) ✓
- Q等待P继续执行 MESA
- 规定唤醒为管程中最后一个可执行的操作 (Hansen 并发pasca)

# 进程通信

---

3.5.1 信号通信机制

3.5.2 管道通信机制

3.5.3 共享主存通信机制

3.5.4 消息传递通信机制

# 进程通信概念

---

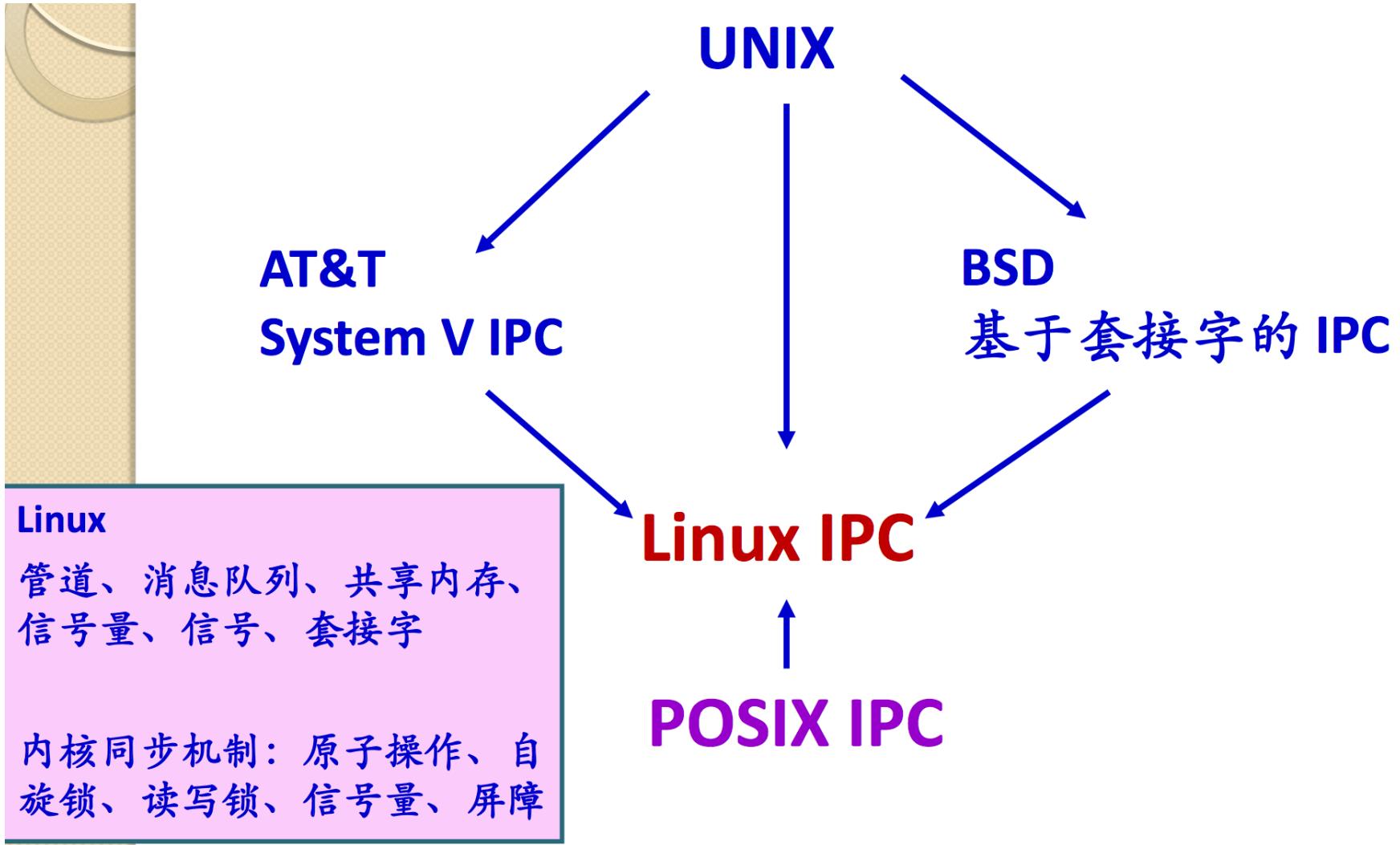
- 并发进程交互的两个基本需求
  - 同步
  - 通信
- 并发进程需要交换信息
  - 少量信息
    - 同步信号
  - 大量信息
- 进程之间互相交换信息的工作称为进程通信IPC (InterProcess Communication)

# IPC方式

---

- 信号(signal)通信机制
  - 发送单个信号、无法传送数据
- 管道(pipeline)通信机制
  - 进程家族之间(非命名管道)
- System V IPC (Bell)
  - 消息传递(message passing)通信机制
  - 信号量(semaphore)通信机制
  - 共享主存(shared memory)通信机制
- 套接字(BSD)
  - 网络通信(TCP/IP)

# Linux的进程通信机制



# 管道通信机制

---

- 管道(pipeline)是连接读写进程的一个特殊文件，允许进程按先进先出方式传送数据，也能使进程同步执行操作
- 发送进程以字符流形式把大量数据送入管道，接收进程从管道中接收数据，所以叫管道通信
- 管道的实质是一个共享文件，基本上可借助于文件系统的机制实现，包括（管道）文件的创建、打开、关闭和读写。

# 3.6 死锁

---

- 3.6.1 死锁产生
- 3.6.2 死锁防止
- 3.6.3 死锁避免
- 3.6.4 死锁检测和解除

# 死锁例子(1)

- 例 1 进程推进顺序不当产生死锁

设系统有打印机、读卡机各一台，被进程  $P$  和  $Q$  共享。两个进程并发执行，按下列次序请求和释放资源：

进程  $P$

请求读卡机  
请求打印机  
释放读卡机  
释放打印机

进程  $Q$

请求打印机  
请求读卡机  
释放读卡机  
释放打印机

# 死锁例子(2)

---

- 例 2 PV 操作使用不当产生死锁
  - 生产者消费者程序  $p(mutex)$ ;  $p(empty)$
  - 如下执行顺序

进程 Q1      进程 Q2

..... .....

- $P(s1); P(s2);$
- $P(s2); P(s1);$
- 使用 r1 和 r2; 使用 r1 和 r2
- $V(s1); V(s2);$
- $V(s2); V(s1);$

# 死锁例子 (3)

---

- 资源分配不当引起死锁

若系统中有 $m$ 个资源被 $n$ 个进程共享，每个进程都要求 $K$ 个资源，而 $m < n \cdot K$ 时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁。

能否举个例子？？？

# 死锁定义

---

- 如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生死锁。
- 例如， $n$ 个进程 $P_1, P_2, \dots, P_n$ ， $P_i$ 因为申请不到资源 $R_j$ 而处于等待状态，而 $R_j$ 又被 $P_{i+1}$ 占有， $P_n$ 欲申请的资源被 $P_1$ 占有，此时这 $n$ 个进程的等待状态永远不能结束，则说这 $n$ 个进程处于死锁状态。

# 形成死锁的四个条件

---

- 四个条件（1971年Coffman总结）
  - 互斥条件：进程互斥使用资源
  - 请求与保持条件：申请新资源时不释放已占有资源
  - 不可剥夺条件：一个进程不能抢夺其他进程占有的资源
  - 循环等待条件：存在一组进程循环等待资源
- 前三个为必要条件、非充分条件
- 前三个+第四个→死锁

# 死锁解决办法

---

- 死锁 防止(Deadlock Prevention)
  - 破坏产生死锁的必要条件，防止死锁发生
  - 强制规则 → 降低进程的并发度
- 死锁 避免(Deadlock Avoidance)
  - 允许前三个条件,避免第四个条件
  - 并发度高
- 死锁 检测和恢复(Deadlock detection & recovery)

# 死锁防止 (1)

---

- 破坏条件1?
  - 使资源可同时访问而不是互斥使用
- 破坏条件2?
  - 静态分配：一个进程先申请所需的所有资源，都满足后再执行
  - 效率低下
- 破坏条件3?
  - 当进程在申请资源获准许的情况下，如能主动释放资源
  - 效率低下
- 破坏条件4?
  - 层次策略、按序分配策略

# 死锁防止 (2)

---

- 层次分配策略
  - 资源被分成多个层次
  - 当进程得到某一层的一个资源后，它只能再申请较高层次的资源
  - 当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源
  - 当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源

# 死锁防止 (3)

---

- 按序分配策略
  - 将资源排序编号:  $r_1, r_2, \dots, r_m$
  - 如果进程不得在占用资源  $r_i (1 \leq i \leq m)$  后再申请  $r_j (j < i)$ 。

# 死锁避免

---

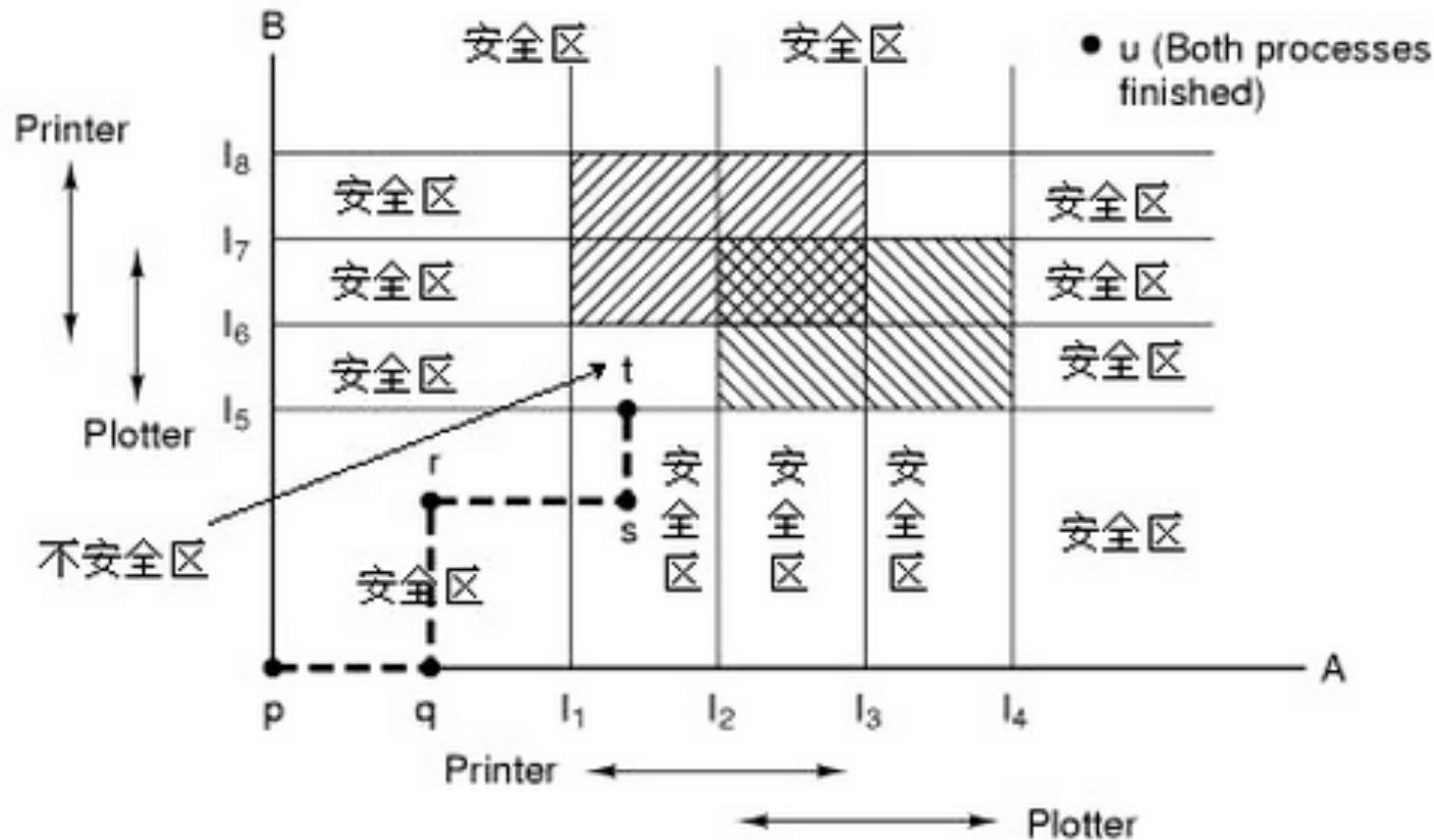
- 主要思想：
  - 通过合理的资源分配算法避免第四个条件
  - 允许前三个条件发生
- 优点：并发度高
- 主要代表：银行家算法
  - Dijkstra老人家的又一经典算法

# 银行家算法

---

- 问题：
  - 银行家拥有一笔资金
  - 每个客户在第一次申请贷款时要声明完成该项目所需的最大资金量
  - 在满足所有贷款要求时，客户应及时归还
  - 银行家在客户申请的贷款数量不超过自己拥有的最大值时，都应尽量满足客户的需要
- 类比
  - 银行家：操作系统
  - 资金：资源
  - 客户：进程
- 银行家算法思想：在一个进程请求资源时，判断一旦分配，系统状态是否安全？

# 进程资源状态图



两个进程的资源轨迹图

# 银行家算法数据结构(1)

---

- 一个系统有n个进程和m种不同类型的资源，  
定义包含以下向量和矩阵的数据结构
  - 系统每类资源总数--该m个元素的向量为系统中  
每类资源的数量  
 $\text{Resource} = (R_1, R_2, \dots, R_m)$
  - 每类资源未分配数量--该m个元素的向量为系统  
中每类资源尚可供分配的数量  
 $\text{Available} = (V_1, V_2, \dots, V_m)$

# 银行家算法数据结构(2)

- 最大需求矩阵--每个进程对每类资源的最大需求量,Cij表示进程Pi需Rj类资源最大数

$$\text{Claim} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$

- 分配矩阵—表示进程当前已分得的资源数,Aij表示进程Pi已分到Rj类资源的个数

$$\text{Allocation} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{21} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

# 一些约束条件

---

$R_i = V_i + \sum A_{ki}$  对  $i=1,..,m, k=1,..,n$ ; 表示所有资源要么已被分配、要么尚可分配

$C_{ki} \leq R_j$  对  $i=1,..,m, k=1,..,n$ ; 表示进程申请资源数不能超过系统拥有的资源总数

$A_{ki} \leq C_{ki}$  对  $i=1,..,m, k=1,..,n$ ; 表示进程申请任何类资源数不能超过声明的最大资源需求数

# 一种简单实现策略

---

- 系统中若要启动一个新进程工作,其对资源  $R_i$  的需求仅当满足下列不等式:

$$R_i \geq C_{(n+1)i} + \sum C_{ki} \text{ 对 } i=1,..,m, k=1,..,n;$$

- 缺点:
  - 进程未必需要立即申请所有资源
  - 降低并发度

# 系统安全性定义

---

- 在时刻 $T_0$ 系统是安全的,仅当存在一个进程序列 $P_1,..,P_n$ ,对进程 $P_k$ 满足公式:

$$C_{ki} - A_{ki} \leq Available_i + \sum A_{ji} \quad j = 1, \dots, k-1; k = 1, \dots, n; i = 1, \dots, m;$$

- 该序列称安全序列。
- 公式左边表示进程 $P_k$ 尚缺少的各类资源；
- $Available_i$ 是 $T_0$ 时刻系统尚可于分配且为 $P_k$ 所想要的那类资源数.

# 实例

- 系统中共有五个进程和A、B、C三类资源
- A类资源共有10个,B类资源共有5个,C类资源共有7个
- 在时刻T<sub>0</sub>,系统目前资源分配情况如下:

| process | Allocation |   |   | Claim |   |   | Available |   |   |
|---------|------------|---|---|-------|---|---|-----------|---|---|
|         | A          | B | C | A     | B | C | A         | B | C |
| p0      | 0          | 1 | 0 | 7     | 5 | 3 | 3         | 3 | 2 |
| p1      | 2          | 0 | 0 | 3     | 2 | 2 |           |   |   |
| p2      | 3          | 0 | 2 | 9     | 0 | 2 |           |   |   |
| P3      | 2          | 1 | 1 | 2     | 2 | 2 |           |   |   |
| p4      | 0          | 0 | 2 | 4     | 3 | 3 |           |   |   |

# 上述状态安全吗？

| 资源<br>进程       | currentavil |   |   | C <sub>ki</sub> -A <sub>ki</sub> |   |   | allocation |   |   | currentavil+allocation |   |   | possible |
|----------------|-------------|---|---|----------------------------------|---|---|------------|---|---|------------------------|---|---|----------|
|                | A           | B | C | A                                | B | C | A          | B | C | A                      | B | C |          |
| P <sub>1</sub> | 3           | 3 | 2 | 1                                | 2 | 2 | 2          | 0 | 0 | 5                      | 3 | 2 | TRUE     |
| P <sub>3</sub> | 5           | 3 | 2 | 0                                | 1 | 1 | 2          | 1 | 1 | 7                      | 4 | 3 | TRUE     |
| P <sub>4</sub> | 7           | 4 | 3 | 4                                | 3 | 1 | 0          | 0 | 2 | 7                      | 4 | 5 | TRUE     |
| P <sub>2</sub> | 7           | 4 | 5 | 6                                | 0 | 0 | 3          | 0 | 2 | 10                     | 4 | 7 | TRUE     |
| P <sub>0</sub> | 10          | 4 | 7 | 7                                | 4 | 3 | 0          | 1 | 0 | 10                     | 5 | 7 | TRUE     |

- {P<sub>1</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>2</sub>, P<sub>0</sub>}能满足安全性条件

# 继续实例(1)

- 进程P1申请资源 $\text{request}_1 = (1,0,2)$ ，可以满足它吗？
- 假设可以

| process        | allocation |   |   | $C_{ki} - A_{ki}$ |   |   | available |   |   |
|----------------|------------|---|---|-------------------|---|---|-----------|---|---|
|                | A          | B | C | A                 | B | C | A         | B | C |
| P <sub>0</sub> | 0          | 1 | 0 | 7                 | 4 | 3 | 2         | 3 | 0 |
| P <sub>1</sub> | 3          | 0 | 2 | 0                 | 2 | 0 |           |   |   |
| P <sub>2</sub> | 3          | 0 | 2 | 6                 | 0 | 0 |           |   |   |
| P <sub>3</sub> | 2          | 1 | 1 | 0                 | 1 | 1 |           |   |   |
| P <sub>4</sub> | 0          | 0 | 2 | 4                 | 3 | 1 |           |   |   |

上述状态安全吗？

# 继续实例(2)

| 资源<br>进程       | currentavil |   |   | C <sub>ki</sub> -A <sub>ki</sub> |   |   | allocation |   |   | currentavil+allocation |   |   | possible |
|----------------|-------------|---|---|----------------------------------|---|---|------------|---|---|------------------------|---|---|----------|
|                | A           | B | C | A                                | B | C | A          | B | C | A                      | B | C |          |
| P <sub>1</sub> | 2           | 3 | 0 | 0                                | 2 | 0 | 3          | 0 | 2 | 5                      | 3 | 2 | TRUE     |
| P <sub>3</sub> | 5           | 3 | 2 | 0                                | 1 | 1 | 2          | 1 | 1 | 7                      | 4 | 3 | TRUE     |
| P <sub>4</sub> | 7           | 4 | 3 | 4                                | 3 | 1 | 0          | 0 | 2 | 7                      | 4 | 5 | TRUE     |
| P <sub>0</sub> | 7           | 4 | 5 | 7                                | 4 | 3 | 0          | 1 | 0 | 7                      | 5 | 5 | TRUE     |
| P <sub>2</sub> | 7           | 5 | 5 | 6                                | 0 | 0 | 3          | 0 | 2 | 10                     | 5 | 7 | TRUE     |

- {P<sub>1</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>2</sub>, P<sub>0</sub>}能满足安全性条件
- 所以P<sub>1</sub>的请求可以满足

# 继续实例(3)

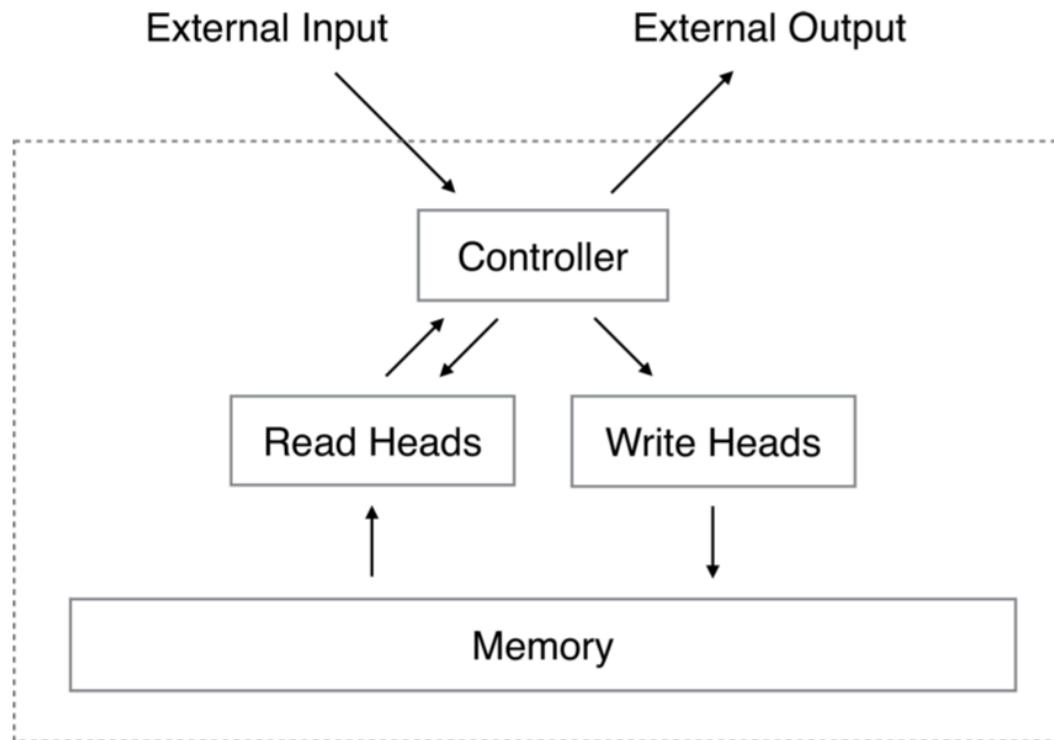
- 系统若处在下面状态中，进程P4请求资源(3,3,0)，由于可用资源不足，申请被系统拒绝；此时，系统能满足进程P0的资源请求(0,2,0)；但可看出系统已处于不安全状态了。

| 资源<br>进程 | allocation |   |   | Cki-Aki |   |   | available |   |   |
|----------|------------|---|---|---------|---|---|-----------|---|---|
|          | A          | B | C | A       | B | C | A         | B | C |
| P0       | 0          | 3 | 0 | 7       | 2 | 3 | 2         | 1 | 0 |
| P1       | 3          | 0 | 2 | 0       | 2 | 0 |           |   |   |
| P2       | 3          | 0 | 2 | 6       | 0 | 0 |           |   |   |
| P3       | 2          | 1 | 1 | 0       | 1 | 1 |           |   |   |
| P4       | 0          | 0 | 2 | 4       | 3 | 1 |           |   |   |

# 银行家算法的基本思想

- 系统中的所有进程进入进程集合，
- 在安全状态下系统收到进程的资源请求后，先把资源试分配给它
- 系统用剩余的可用资源和在的进程集集中找到剩余其他进程，从而保证这个进程集集中全部资源
- 把进程从进程中去掉，回收其占用资源，反反复复执行上述步骤
- 最后，若剩下的进程集合为空，表明本次申请可实施；否则，有进程执行不完，系统处于不安全状态，本次资源分配暂不实施，让申请进程等待。

如果是换一种思维呢？也许一切更美好了



**Figure 1: Neural Turing Machine Architecture.** During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

# Neural Turing Machines

We extend the capabilities of neural networks by coupling them to external memory resources, which they can interact with by attentional processes. The combined system is analogous to a Turing Machine or Von Neumann architecture but is differentiable end-to-end, allowing it to be efficiently trained with gradient descent. Preliminary results demonstrate that *Neural Turing Machines* can infer simple algorithms such as copying, sorting, and associative recall from input and output examples.

# 死锁检测和解除

---

- 基本思想：
  - 系统对资源分配不加限制，允许死锁。
  - 系统定期运行死锁检测程序，一旦发现死锁，将它解除。
- 难点：
  - 何时运行死锁检测算法
  - 太频繁：浪费处理器资源
  - 太稀疏：影响死锁进程效率

# Appendix

# 使用信号量解决生产者消费者问题

---

- 一个生产者、一个消费者共享一个缓冲区
- 一个生产者、一个消费者共享多个缓冲区
- 多个生产者、多个消费者共享多个缓冲区

# 一个生产者、一个消费者、一个缓冲区

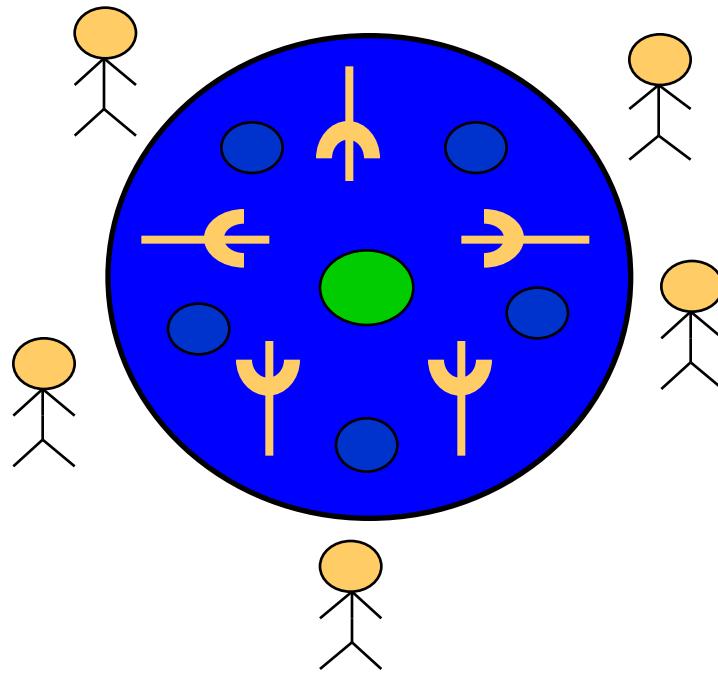
---

- int B;
- semaphore empty; //可以使用的空缓冲区数
- semaphore full; //缓冲区内可以使用的产品数
- empty=1; //缓冲区内允许放入一件产品
- full=0; //缓冲区内没有产品
- cobegin
- process producer(){ process consumer(){
- while(true){           while(true) {
- produce();           P(full);
- P(empty);           take() from B;
- append() to B;      V(empty);
- V(full);            consume();
- }
- }
- }
- coend

# 多个生产者、多个消费者、多个缓冲区

- item B[k];
- semaphore empty; empty=k; //可以使用的空缓冲区数
- semaphore full; full=0; //缓冲区内可以使用的产品数
- semaphore mutex; mutex=1; //互斥信号量
- int in=0; //放入缓冲区指针
- int out=0; //取出缓冲区指针
- cobegin
- process producer\_i (){ process consumer\_j (){
  - while(true) {
  - produce();
  - P(empty);
  - P(mutex);
  - append to B[in];
  - in=(in+1)%k;
  - V(mutex);
  - V(full);
  - }
  - }
- }
- coend

# 哲学家吃面问题



有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。

# 使用信号量解决吃面问题

```
semaphore fork[5];
for (int i=0;i<5;i++)
    fork[i]=1;
```

```
Cobegin
process philosopher_i() {
    //i= 0,1,2,3,4
    while(true) {
        think();
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat();
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
Coend
```

死锁问题：

五个哲学家同时拿起右边的叉子，然后再拿左边的叉子

解决方法：

- 1.至多允许四个哲学家同时吃
- 2.奇数号先取左手边的叉子，偶数号先取右手边的叉子；
- 3.每个哲学家取到手边的两把叉子才吃，否则一把叉子也不取

# 信号量解决读者-写者问题(1)

---

- 有两组并发进程：读者和写者，共享一个文件F，要求：
- 允许多个读者同时执行读操作
- 任一写者在完成写操作之前不允许其它读者或写者工作
- 写者执行写操作前，应让已有的写者和读者全部退出

# 信号量解决读者-写者问题(2)

---

- int readcount=0;//读进程计数
- semaphore writeblock.mutex;
- writeblock=1;mutex=1;
- cobegin
- process reader\_i(){       process writer\_j(){  
•     P(mutex);                      P(writeblock);  
•     readcount++;                  {写文件};  
•     if(readcount==1)              V(writeblock);  
•     P(writeblock);                }  
•     V(mutex);  
•     {读文件};  
•     P(mutex);  
•     readcount--;  
•     if(readcount==0)  
•         V(writeblock);  
•     V(mutex);  
•     }  
• }
- coend

# 信号量解决理发师问题

---

- 理发店理有一位理发师、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，理发师便在理发椅上睡觉
- 一个顾客到来时，它必须叫醒理发师
- 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开

# 信号量解决理发师问题

```
int waiting = 0;           //等候理发的顾客坐的椅子数
int CHAIRS = N;            //为顾客准备的椅子数
semaphore customers, barbers, mutex;
customers = 0; barbers = 0; mutex = 1;
cobegin
process barber() {
    while(true) {
        P(customers);      //判断是否有顾客,若无顾客,理发师睡眠
        P(mutex);           //若有顾客,进入临界区
        waiting -= 1;       //等候顾客数减 1
        V(barbers);         //理发师准备为顾客理发
        V(mutex);           //退出临界区
        cut_hair();          //理发师正在理发(非临界区)
    }
}
process customer_i() {
    P(mutex);              //进入临界区
    if(waiting < CHAIRS) {
        waiting++;          //等候顾客数加 1
        V(customers);        //唤醒理发师
        V(mutex);             //退出临界区
        P(barbers);           //理发师忙,顾客坐着等待
        get_haircut();         //否则,顾客可以理发
    }
    else
        V(mutex);           //入满了,顾客离开
}
coend
```

# 管程详解

# 管程例子

```
static class our_monitor { // 这是一个管程
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // 计数器和索引

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // 如果缓冲区满，则进入休眠
        buffer[hi] = val; // 向缓冲区中插入一个新的数据项
        hi = (hi + 1) % N; // 设置下一个数据项的槽
        count = count + 1; // 缓冲区中的数据项又多了一项
        if (count == 1) notify(); // 如果消费者在休眠，则将其唤醒
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // 如果缓冲区空，进入休眠
        val = buffer[lo]; // 从缓冲区中取出一个数据项
        lo = (lo + 1) % N; // 设置待取数据项的槽
        count = count - 1; // 缓冲区中的数据项数目减少1
        if (count == N - 1) notify(); // 如果生产者在休眠，则将其唤醒
        return val;
    }

    private void go_to_sleep() { try{wait();} catch(InterruptedException exc){}; }
}
```

# 管程的条件变量

---

- 条件变量
  - 管程内的一种数据结构
  - 通过wait()和signal()两个原语操作它
- wait()-挂起调用进程并释放管程，直到另一个进程在该条件变量上执行signal()
- signal()-如果存在其他进程由于对条件变量执行wait()而被挂起，便释放之；如果没有进程在等待，那么，信号不被保存
- 条件变量与P、V操作中信号量的区别？

# 应用管程时遇到的问题

是否会出现这样一种情况，有多个进程同时在管程中？

场景：

当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权

当后面进入管程的进程执行唤醒操作时（例如，P唤醒Q），管程中便存在两个同时处于活动状态的进程

如何解决？

三种处理方法：

- P等待Q执行 (Hoare) ✓
- Q等待P继续执行 MESA
- 规定唤醒为管程中最后一个可执行的操作 (Hansen 并发pasca)

# 管程的难点

---

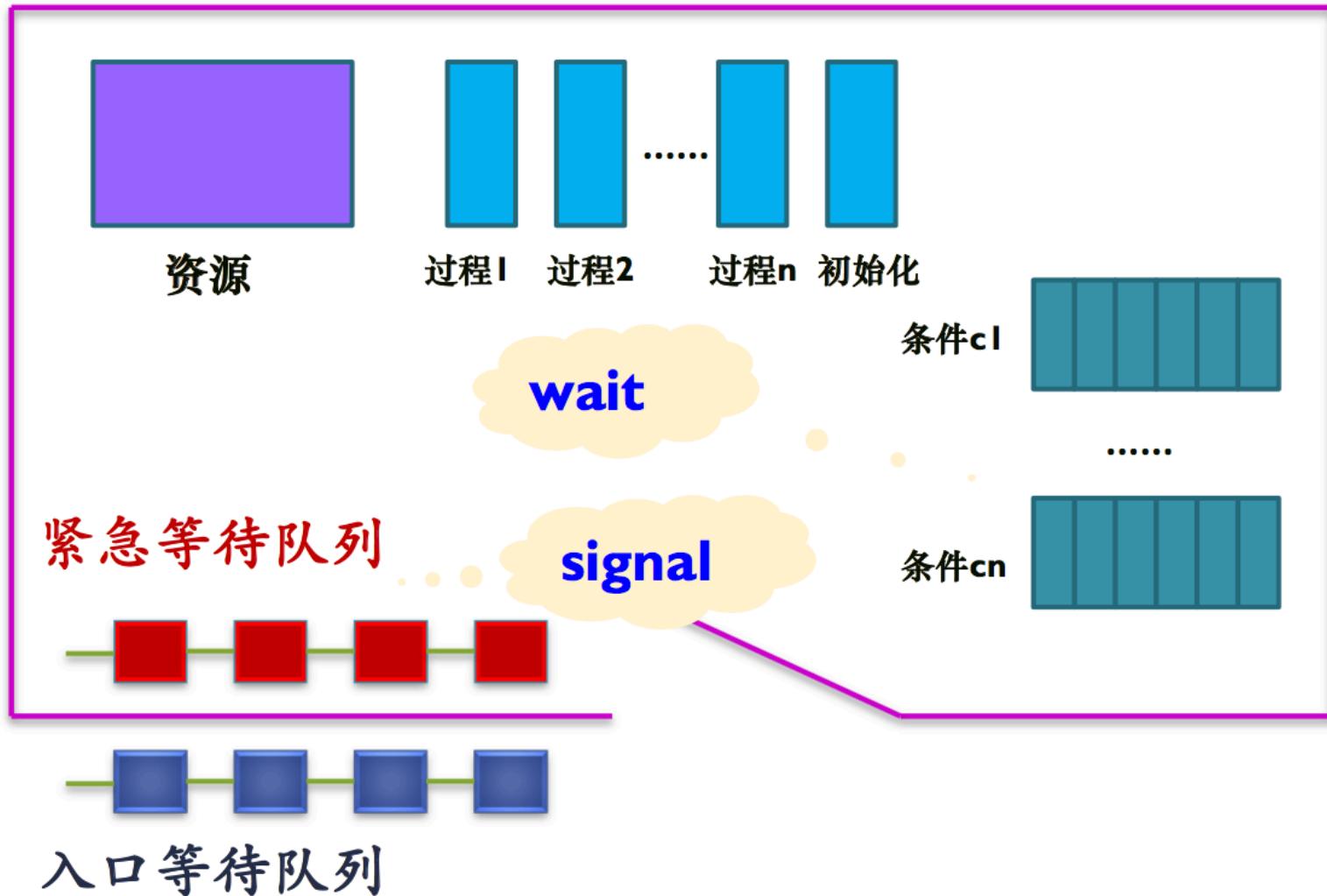
- 使用signal释放等待进程时，出现两个进程同时停留在管程内 → 与管程的互斥性矛盾
- 解决方法：
  - 执行signal的进程等待，直到被释放进程退出管程或等待另一个条件
  - 被释放进程等待，直到执行signal的进程退出管程或等待另一个条件
- 霍尔（Hoare）采用第一种办法
- 汉森（Hansen）选择两者的折衷，规定管程中的过程所执行的signal操作是过程体的最后一个操作

# 管程的实现 – Hoare 方法

---

- 使用P和V操作原语来实现
  - 管程中过程的互斥调用
  - 共享资源互斥使用
- 不要求signal操作是过程体的最后一个操作
- wait和signal操作可被设计成可以中断的过程

# Hoare管程示意图



# Hoare管程数据结构 (1)

---

- Mutex信号量
  - 用于管程中过程的互斥调用
  - 初始值为1
  - 进程调用管程中的任何过程时执行P(mutex)
  - 进程退出管程时应执行V(mutex)开放管程
  - wait操作中也必须执行V(mutex)? ? ?

# Hoare管程数据结构 (2)

---

- next信号量
  - 凡发出signal操作的进程应该用P(next)挂起自己，直到被释放进程退出管程或产生其他等待条件？？？  
保证只有一个进程在管程中！
  - 初始值为0
- next-count计数器
  - 用来记录在next上等待的进程个数
  - 进程在退出管程的过程前检查是否有别的进程在信号量next上等待( $\text{next-count} > 0?$ )，若有，则用V(next)唤醒它

# Hoare管程数据结构 (3)

---

- x-sem信号量
  - 申请资源得不到满足时，执行P(x-sem)挂起
  - 初始值为0
  - 执行signal操作时，应让等待资源的诸进程中的某个进程立即恢复运行，而不让其他进程抢先进入管程，这可以用V(x-sem)来实现
- x-count计数器
  - 记录等待资源的进程数

# Hoare管程 的enter()操作

---

- 每个过程调用之前执行

```
void enter(InterfaceModule &IM) {  
    P(IM.mutex);      //互斥进入管程  
}
```

# Hoare管程的leave()操作

---

- 每个过程调用返回前调用

- void leave(InterfaceModule &IM) {
  - //判有否发出过signal的进程?
  - if(IM.next\_count>0)
    - V(IM.next);
  - //有就释放一个发出过signal的进程
  - else
    - V(IM.mutex); //否则开放管程
  - }

# Hoare管程的wait()操作

---

- void wait(semaphore &x\_sem,int &x\_count,InterfaceModule &IM) {
  - x\_count++; //等资源进程个数加1, x\_count初始化为0
  - if(IM.next\_count>0)//判有否发出过signal的进程
    - V(IM.next); //有就释放一个
  - else
    - V(IM.mutex); //否则开放管程
  - P(x\_sem); //等资源进程阻塞自己, x\_sem初始化为0
  - x\_count--; //醒过来后等资源进程个数减1
- }

# Hoare管程的signal()操作

---

```
void signal(semaphore &x_sem,int  
           &x_count,InterfaceModule &IM) {  
    if(x_count>0) { //有等资源进程吗?  
        IM.next_count++; //当前发出signal的进程马上阻塞在next上  
        V(x_sem); //释放一个等资源的进程  
        P(IM.next); //当前发出signal进程阻塞自己，保证管程中只有1个  
        //进程  
        IM.next_count--; //醒过来后，等待在next上的进程数减1    }  
    • }
```

## 2 管程解决生产者-消费者问题(1)

---

- type producer\_consumer=monitor
- item B[k]; //缓冲区个数
- int in,out; //存取指针
- int count; //缓冲中产品数
- semaphore notfull,notempty; //条件变量
- int notfull\_count,notempty\_count;
- InterfaceModule IM;
- define append,take;
- use enter,leave,wait,signal;

# 管程解决生产者-消费者问题(2)

---

- void append(item x) {
- enter(lM);
- if(count==k) //缓冲已满  
        wait(notfull,notfull\_count,lM);
- B[in]=x;
- in=(in+1)%k;
- count++; //增加一个产品
- signal(notempty,notempty\_count,lM);  
        //唤醒等待消费者
- leave(lM);
- }

# 管程解决生产者-消费者问题(3)

---

- void take(item &x) {
- enter(lM);
- if(count==0)
- wait(notempty,notempty\_count,lM);
- //缓冲已空
- x=B[out];
- out=(out+1)%k;
- count--;                                  //减少一个产品
- signal(notfull,notfull\_count,lM);
- //唤醒等待生产者
- leave(lM);
- }

# 管程解决生产者-消费者问题(4)

```
static class our_monitor { // 这是一个管程
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // 计数器和索引
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // 如果缓冲区满，则进入休眠
        buffer[hi] = val; // 向缓冲区中插入一个新的数据项
        hi = (hi + 1) % N; // 设置下一个数据项的槽
        count = count + 1; // 缓冲区中的数据项又多了一项
        if (count == 1) notify(); // 如果消费者在休眠，则将其唤醒
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // 如果缓冲区空，进入休眠
        val = buffer[lo]; // 从缓冲区中取出一个数据项
        lo = (lo + 1) % N; // 设置待取数据项的槽
        count = count - 1; // 缓冲区中的数据项数目减少1
        if (count == N - 1) notify(); // 如果生产者在休眠，则将其唤醒
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc){}}
}
```

有问题吗？？？

# 用管程解决生产者消费者问题

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert (item: integer);
  begin
    if count == N then wait(full);
    insert_item(item); count++;
    if count == 1 then signal(empty);
  end;

  function remove: integer;
  begin
    if count == 0 then wait(empty);
    remove = remove_item; count--;
    if count==N-1 then signal(full);
  end;

  count:=0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item);
    end
  end;

procedure consumer;
begin
  while true do
    begin
      item=ProducerConsumer.remove;
      consume_item(item);
    end
  end;
```

# Linux 的信号机制

# Linux信号通信机制

---

- 信号机制是一种软中断
- 用户、内核和进程都发送信号
  - 用户：输入ctrl+c，或终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号。
  - 内核：当进程执行出错时，内核检测到事件并给进程发送信号，例如，非法段存取、浮点数溢出、或非法操作码，内核也利用信号通知进程种种特定事件发生。
  - 进程：通过系统调用kill给另一个进程发送信号，一个进程可通过信号与另一个进程通信。

# Linux系统信号分类

---

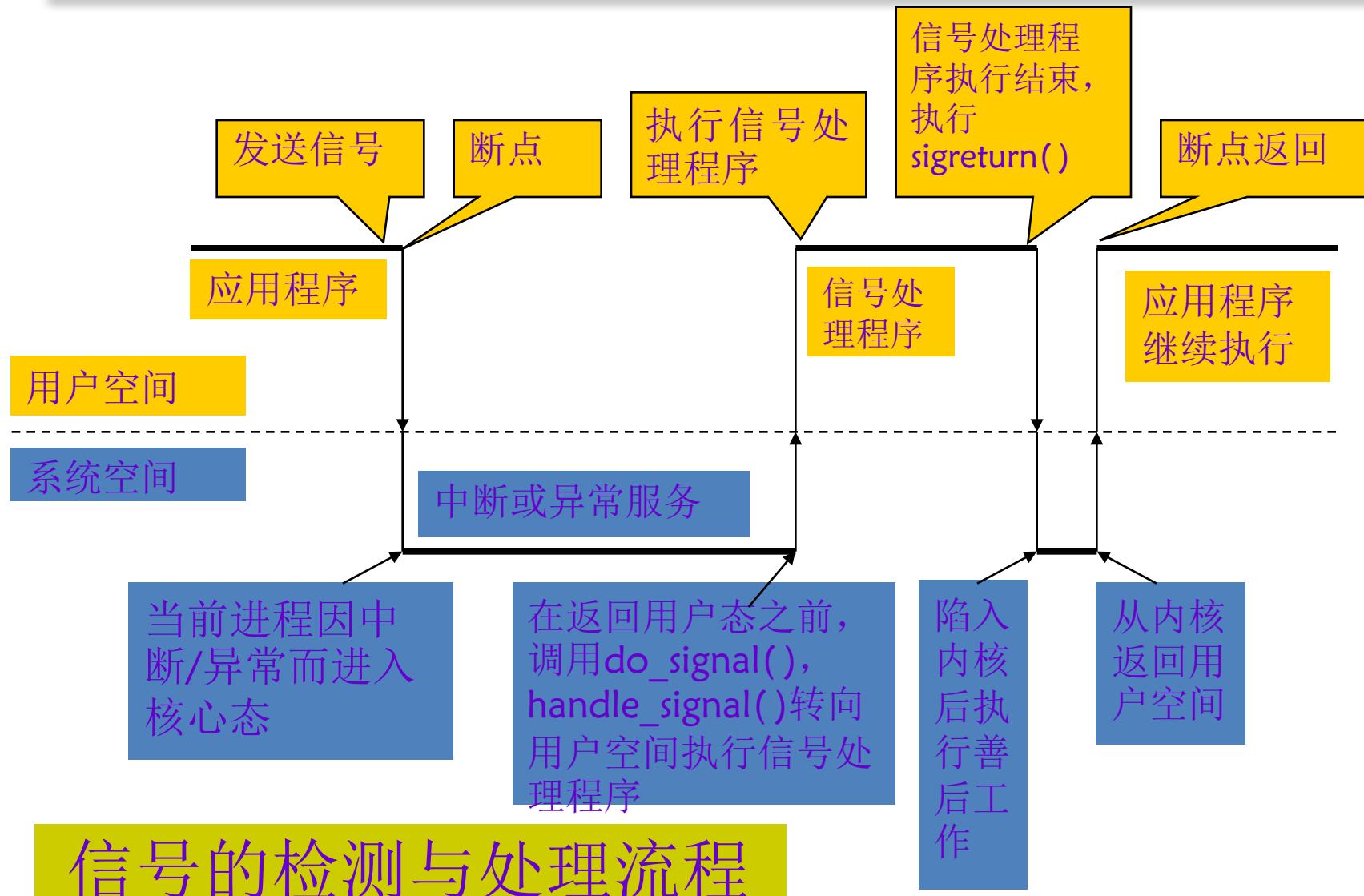
- 与进程终止相关的信号
- 与进程例外事件相关的信号
- 与进程执行系统调用相关的信号
- 与进程终端交互相关的信号
- 用户进程发送的信号
- 跟踪进程执行的信号

# 信号机制的实现(1)

---

- 信号的生命周期
  - 产生 → 传送 → 捕获 → 释放
- task\_struct
  - signal域：保存接收到的信号
  - blocked域：信号屏蔽标记
  - sigaction数组：存储处理程序的入口地址（64个元素）
- 系统调用 kill：发送信号
- Sigaction 系统调用：设定响应函数

# 信号机制的实现(2)



信号的检测与处理流程

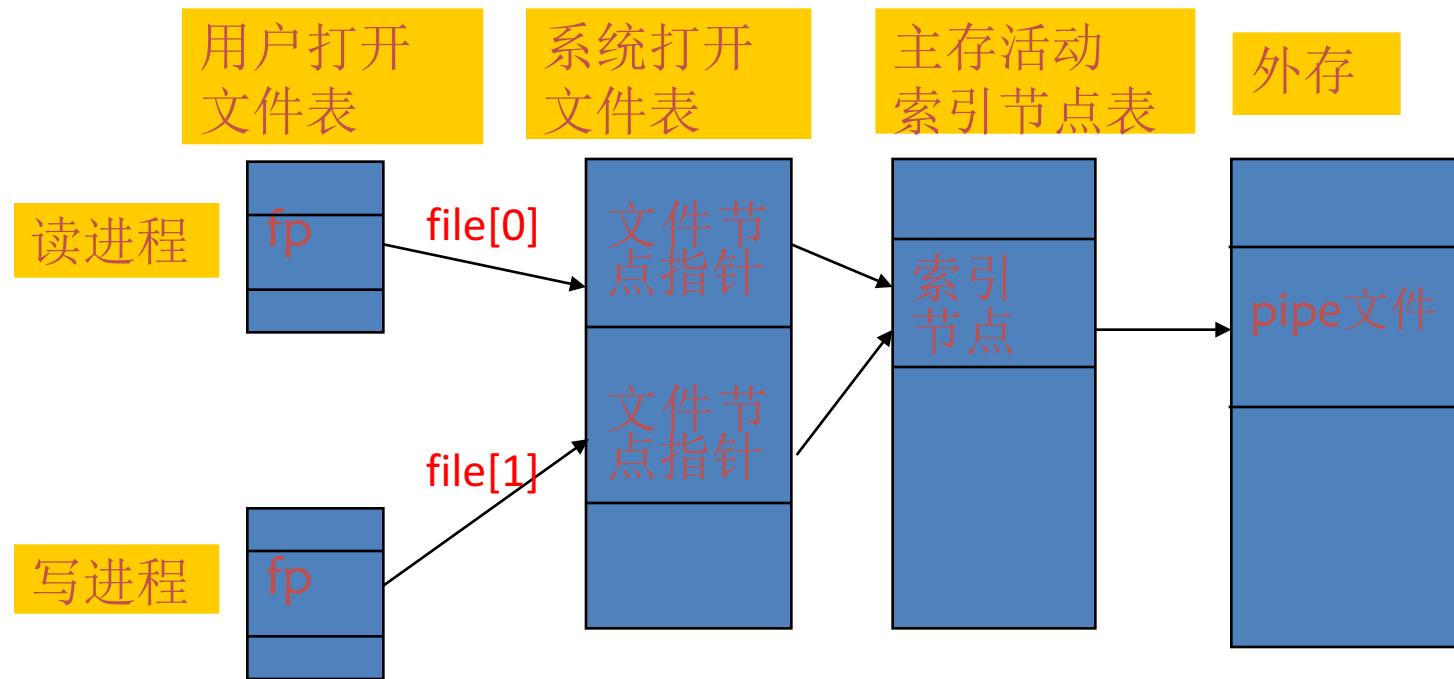
# 其他的通信方式

# 共享文件通信机制

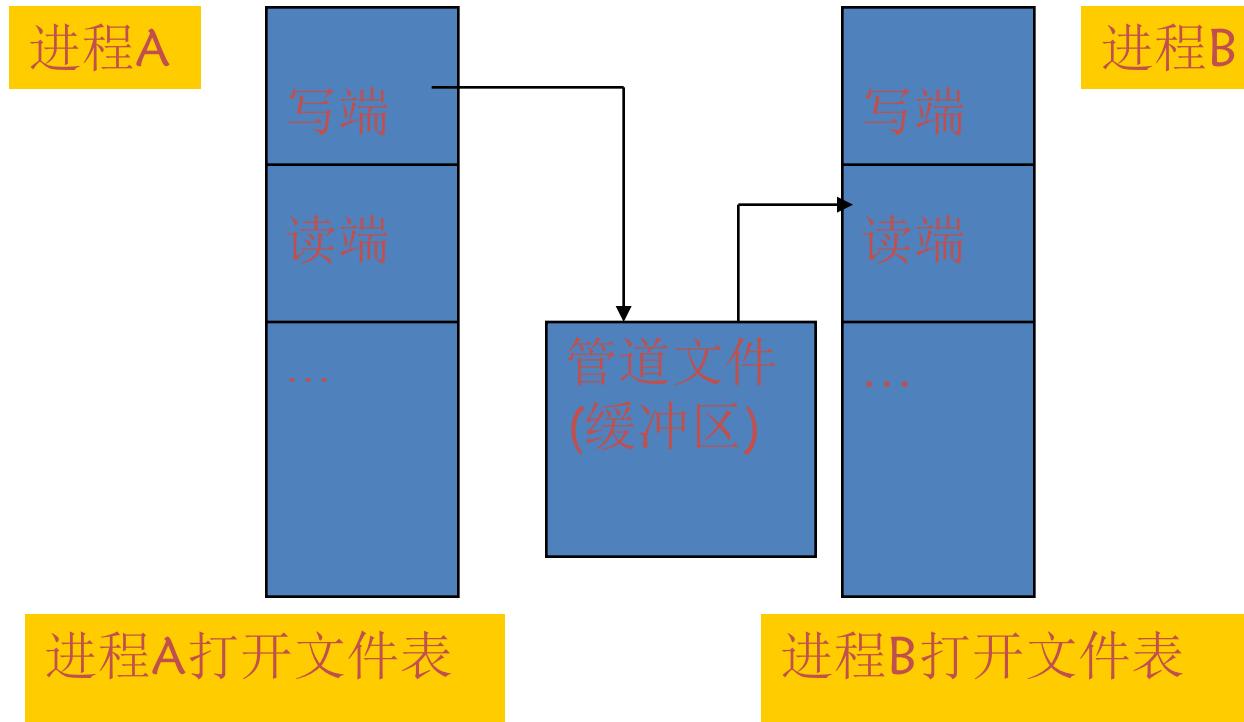
---

- 进程读写管道需要同步
  - 互斥：一个进程正在使用某个管道写入或读出数据时，另一个进程就必须等待(write阻塞、read阻塞)
  - 通信双方需要知道对方是否存在：一方关闭了怎么办
  - 解决生产者消费者问题

# 管道的数据结构

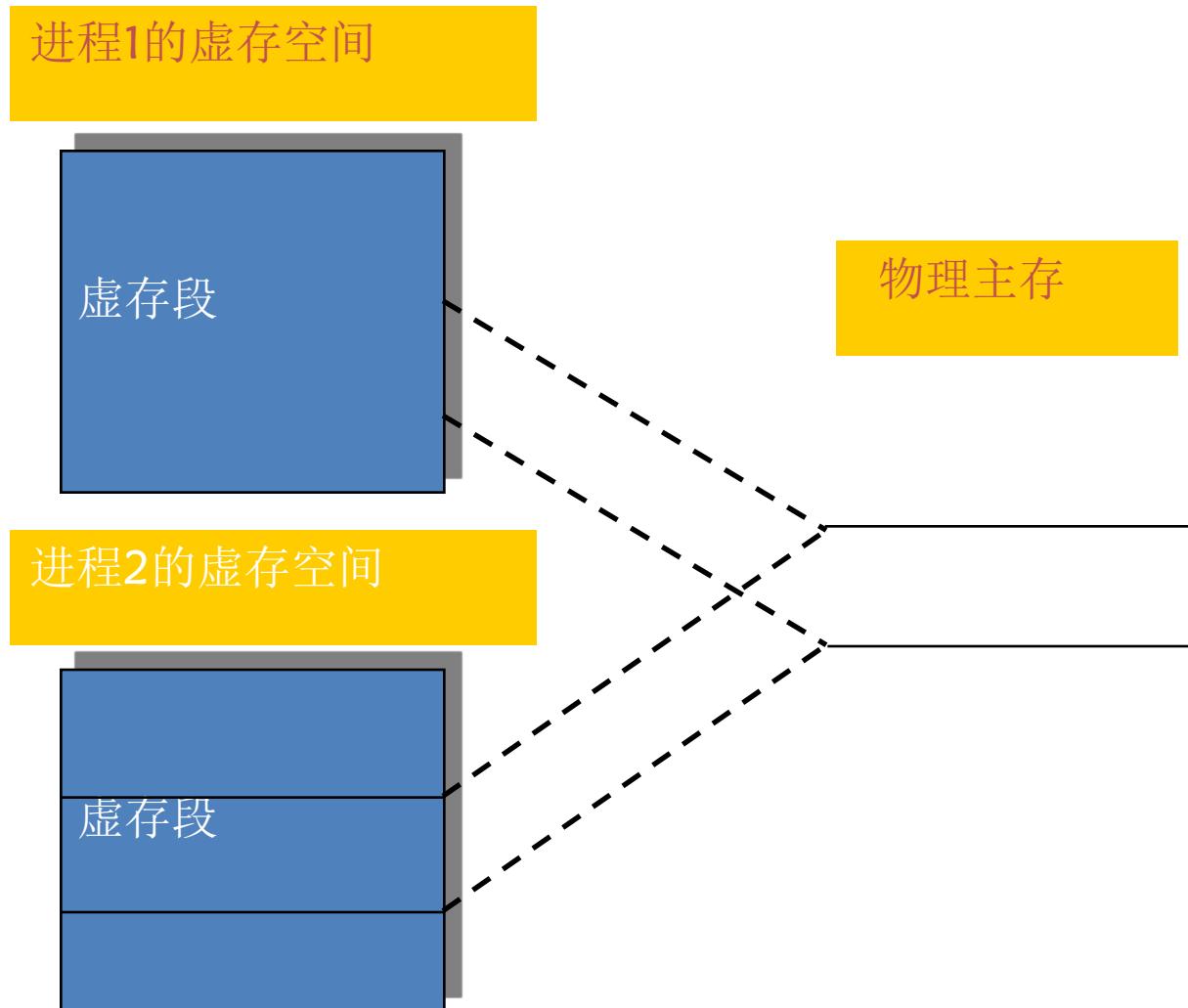


# 父进程与子进程管道通信



父子进程通过  
管道单向通信

# 共享主存通信机制



# Linux与共享存储有关的系统调用

---

- `shmget(key,size,permflags)`
  - 开辟共享内存
- `shmat(shm-id,daddr,shmflags)`
  - 将共享内存映射到进程地址空间
- `shmdt(memptr)`
  - 断开共享内存映射
- `shmctl(shm-id,command,&shm-stat)`
  - 操作共享内存
- 操作共享内存的代码通常被视作临界区

# 消息传递机制

---

- 消息传递机制
  - 一进程可在任何时刻向另一个进程发送消息，
  - 一进程也可在任何时刻向另一个进程请求消息
- 进程地址空间隔离
  - 消息传统无法在用户空间完成
  - 消息传递通过内核完成
- 支持进程间大量交换信息， 使用方便
- 提高进程的同步能力
  - 通过传递消息来进行同步

# 直接通信

---

- 发送或接收消息的进程必须指出信件发给谁或从谁那里接收消息
- 原语send (P, 消息) : 把一个消息发送给进程P
- 原语receive (Q, 消息) : 从进程Q接收一个消息
  - 没有消息→阻塞

# 间接通信

---

- 原语send (A, 信件)：把一封信件（消息）传送到信箱A
- 原语receive (A, 信件)：从信箱A接收一封信件（消息）
- 信箱是存放信件的存储区域，每个信箱可分成信箱特征和信箱体两部分。
- 信箱特征指出信箱容量、信件格式、指针等；信箱体用来存放信件

# 间接通信的实现

---

- 典型的生产者消费者问题
- 发送信件
  - 满则等待
  - 不满则发送，并释放等待接受的进程
- 接收信件
  - 空则等待
  - 非空则接受，并释放等待发送的进程

# 消息传递机制解决临界区互斥问题

---

- `create_mailbox(box);`
- `send(box,null);`
- `void Pi() { //i=1,2,...,n`
- `message msg;`
- `while(true) {`
- `receive(box,msg);`
- `{临界区};`
- `send(box,msg);`
- `}`
- `}`
- `cobegin`
- `Pi();`
- `coend`

# 消息传递机制解决生产者消费者问题

```
int capacity, i ;      //缓冲大小  
creat-mailbox(producer); //创建信箱  
creat-mailbox(consumer);  
for(i=0;i<capacity;i++)  
    send(producer,null); //发送空消息
```

```
void producer_i( ) {           //i=1,...,n  
    message pmsg;  
    while(true) {  
        produce( );           //生产消息  
        receive(producer,null); //等待空消息  
        build( );             //构造一条消息  
        send(consumer,pmsg);  //发送消息  
    }  
}
```

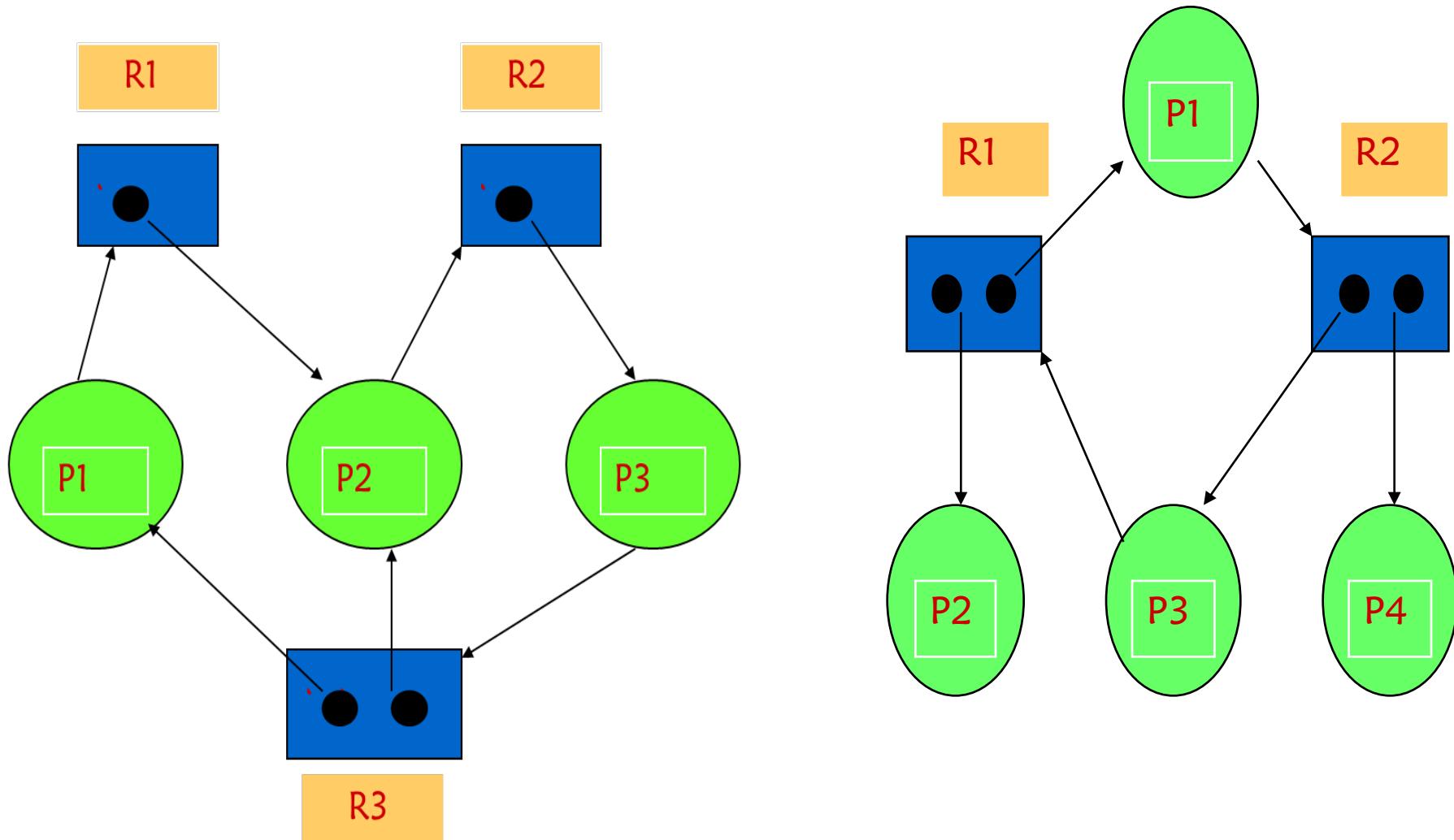
```
void consumer_j( ) {           //j=1,...,m  
    message cmsg;  
    while(true) {  
        receive (consumer,cmsg); //接收消息  
        extract( );            //取消息  
        send(producer,null);   //回送空消息  
        consume(csmg);        //消耗消息  
    }  
}
```

# 死锁检测和解除

---

- 基本思想：
  - 系统对资源分配不加限制，允许死锁。
  - 系统定期运行死锁检测程序，一旦发现死锁，将它解除。
- 难点：
  - 何时运行死锁检测算法
  - 太频繁：浪费处理器资源
  - 太稀疏：影响死锁进程效率

# 进程-资源分配图



# 依赖于资源分配图的死锁检测算法(1)

---

- (1)如果进程-资源分配图中无环路，则此时系统没有发生死锁
- (2)如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁
- (3)如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件

# 依赖于资源分配图的死锁检测算法(2)

---

- 如果能在进程-资源分配图中消去此进程的所有请求边和分配边，则其成为孤立结点
- 如果能使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的
- 系统为死锁状态的充分条件
  - 当且仅当该状态的进程-资源分配图是不可完全简化的

# 死锁的解除

---

- 方法1：重启所有死锁进程
- 方法2：逐个撤销陷于死锁的进程，回收其资源重新分派，直至死锁解除
- 方法3：剥夺陷于死锁的进程占用的资源，但并不撤销它，直至死锁解除
- 方法4：根据系统保存的检查点，让所有进程回退，直到足以解除死锁
- 方法5：如果存在某些未卷入死锁的进程，而随着这些进程执行到结束，有可能释放足够的资源来解除死锁

# Spark应用概念(1)

---

- Application: Spark 的应用程序，用户提交后，Spark为App分配资源，将程序转换并执行，其中Application包含一个Driver program和若干Executor
- SparkContext: Spark 应用程序的入口，负责调度各个运算资源，协调各个 Worker Node 上的 Executor
- Driver Program: 运行Application的main()函数并且创建SparkContext
- RDD Graph: RDD是Spark的核心结构，可以通过一系列算子进行操作（主要有Transformation和Action操作）。当RDD遇到Action算子时，将之前的所有算子形成一个有向无环图（DAG）。再在Spark中转化为Job，提交到集群执行。一个App中可以包含多个Job
- Executor: 是为Application运行在Worker node上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上。每个Application都会申请各自的Executor来处理任务
- Worker Node: 集群中任何可以运行Application代码的节点，运行一个或多个Executor进程

# Spark应用概念(2)

---

- 下面介绍Spark Application运行过程中各个组件的概念：
  - Job：一个RDD Graph触发的作业，往往由Spark Action算子触发，在SparkContext中通过runJob方法向Spark提交Job
  - Stage：每个Job会根据RDD的宽依赖关系被切分很多Stage，每个Stage中包含一组相同的Task，这一组Task也叫TaskSet
  - Task：一个分区对应一个Task，Task执行RDD中对应Stage中包含的算子。Task被封装好后放入Executor的线程池中执行