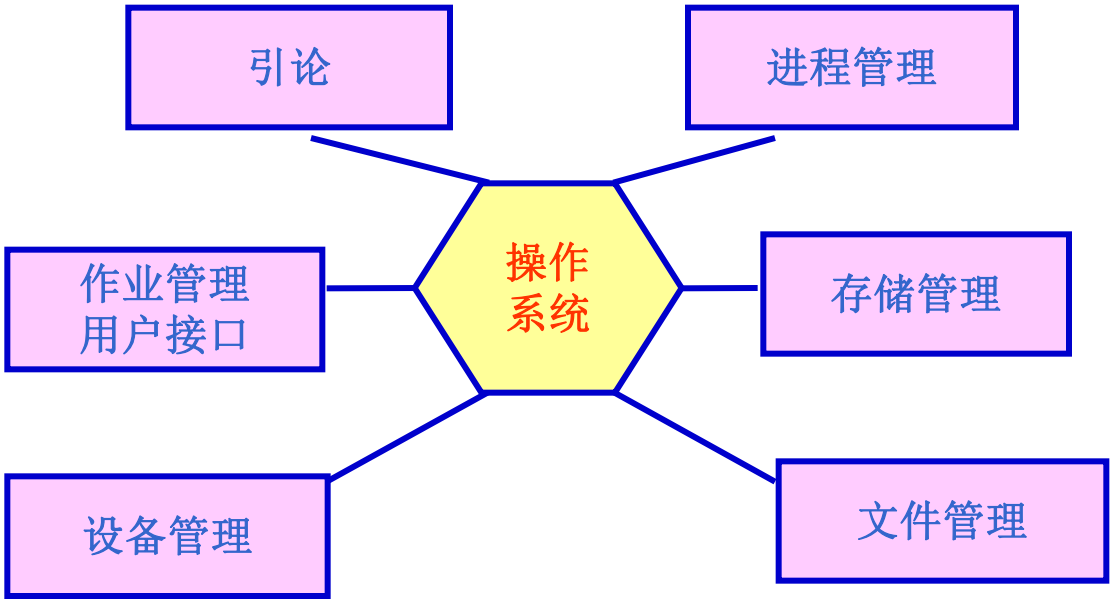
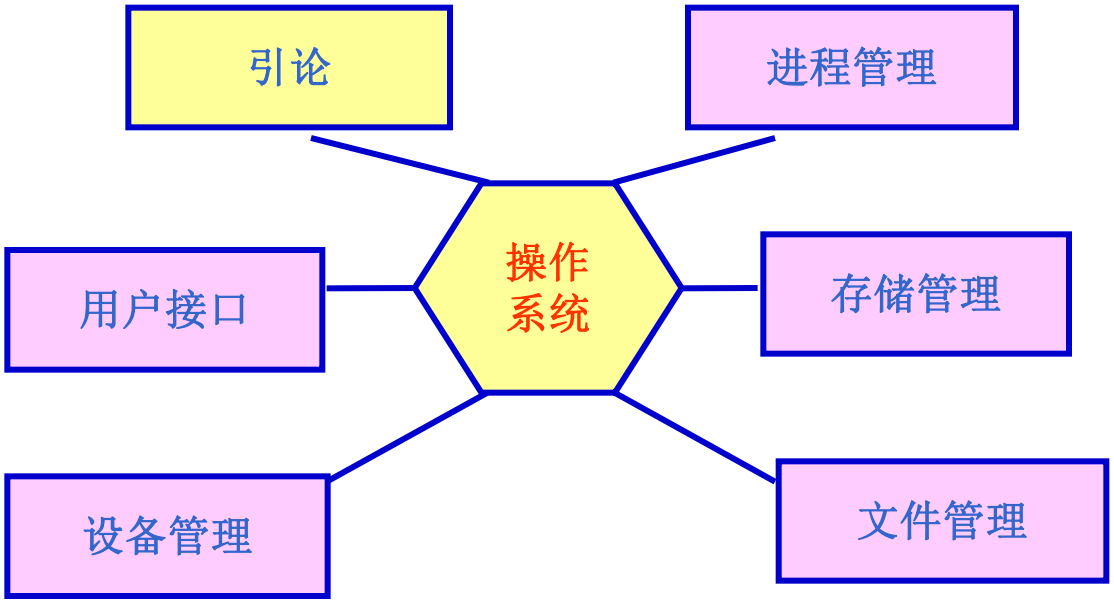
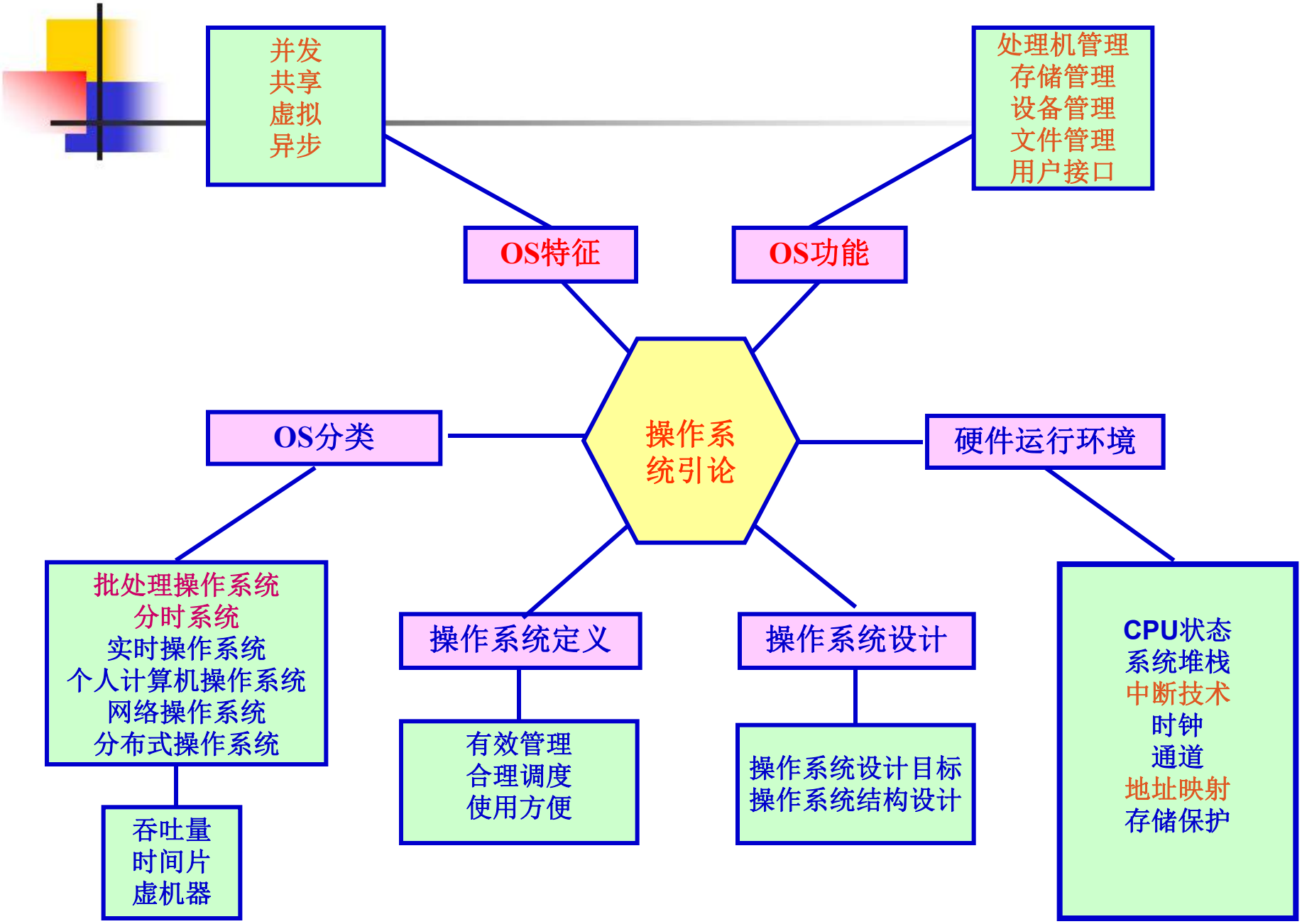
操作系统课程复习


引论
进程管理
操作
系统
作业管理
用户接口
存储管理
设备管理
文件管理


引论
进程管理
操作
系统
用户接口
存储管理
设备管理
文件管理

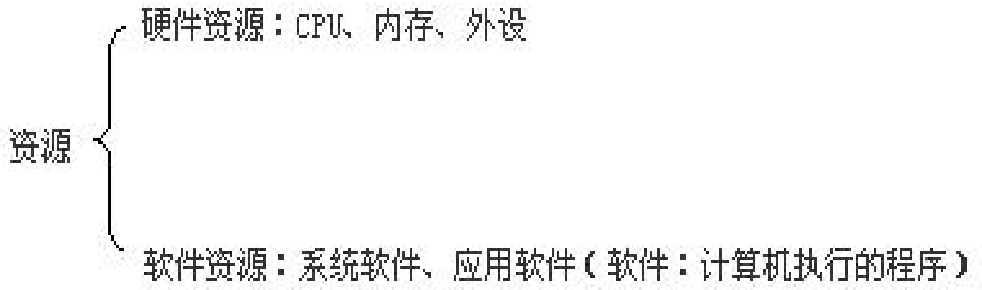

操作系统引论
OS特征
并发
共享
虚拟
异步
OS功能
处理机管理
存储管理
设备管理
文件管理
用户接口
OS分类
批处理操作系统
分时系统
实时操作系统
个人计算机操作系统
网络操作系统
分布式操作系统
吞吐量
时间片
虚机器
操作系统定义
有效管理
合理调度
使用方便
操作系统设计
操作系统设计目标
操作系统结构设计
硬件运行环境
CPU状态
系统堆栈
中断技术
时钟
通道
地址映射
存储保护

主要知识点

- 什么是操作系统?
- 操作系统包括哪些功能?
- 操作系统的地位
- 操作系统的基本特征?
- 操作系统的主要类型有哪些? 各有特点?

主要知识点

- 什么是操作系统?
- 操作系统：是控制和管理计算机系统内各种硬件和软件资源、有效地组织多道程序运行的系统软件（或程序集合），是用户与计算机之间的接口。
- 1) OS是什么：是系统软件（一整套程序组成，如UNIX由上千个模块组成）
- 2) 管什么：控制和管理系统资源（记录和调度）



主要知识点

操作系统的主要功能?

操作系统的功能：存储器管理、处理机管理、设备管理、文件管理和用户接口管理 。

1) 存储器管理：内存分配，地址映射，内存保护和内存扩充

2) 处理机管理：作业和进程调度，进程控制和进程通信

3) 设备管理：缓冲区管理，设备分配，设备驱动和设备无关性

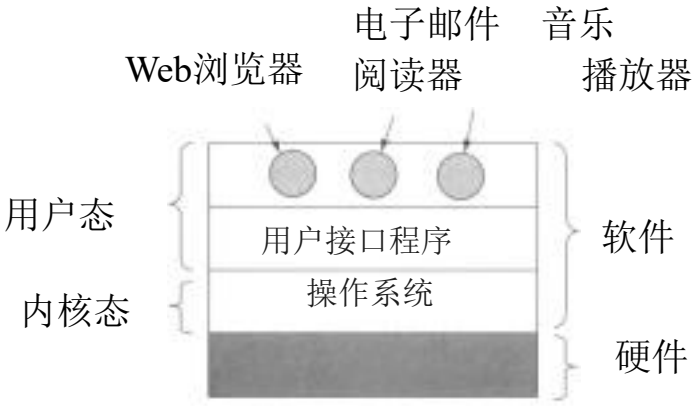
4) 文件管理：文件存储空间的管理，文件操作的一般管理，目录管理，文件的读写管理和存取控制

5) 用户接口：命令界面/图形界面和系统调用接口

主要知识点

操作系统的地位?

操作系统是裸机之上的第一层软件，是建立其他所有软件的基础。它是整个系统的控制管理中心，既管硬件，又管软件，它为其它软件提供运行环境。



主要知识点

- 核心模式和用户模式 （管态和目态）
- 核心模式一般指操作系统管理程序运行的状态，具有较高的特权级别。
- 用户模式一般指用户程序运行时的状态，具有较低的特权级别。
- 当处理器处于管态时全部指令（包括特权指令）可以执行，可使用所有资源，并具有改变处理器状态的能力。当处理器处于用户模式时，就只能执行非特权指令。特权级别不同，可运行指令集合也不同。特权级别越高，可以运行指令集合越大。高特权级别对应的可运行指令集合包含低特权级的。核心模式到用户模式的唯一途径是通过中断。

主要知识点

操作系统的基本特征？

并发，共享和异步性。

1) 并发：并发性是指两个或多个活动在同一给定的时间间隔中进行。

并发与并行的区别：并行性指两个或多个事件在同一时刻发生，并发性指两个或多个事件在同一时间间隔发生。

2) 共享：共享是指计算机系统中的资源被多个任务所共用。

3) 异步性：每个程序什么时候执行，向前推进速度快慢，是由执行的现场所决定。但同一程序在相同的初始数据下，无论何时运行都应获得同样的结果。

主要知识点

操作系统的主要类型？

多道批处理系统、分时系统、实时系统、个人机系统、网络系统和分布式系统

1) 多道批处理系统

- 特点：多道、成批
- 优点：资源利用率高、系统吞吐量大
- 缺点：等待时间长、没有交互能力

2) 分时系统

- 分时：指若干并发程序对CPU时间的共享。它是通过系统软件实现的。共享的时间单位称为时间片。
- 特征：
  - 同时性：若干用户可同时上机使用计算机系统
  - 交互性：用户能方便地与系统进行人--机对话
  - 独立性：系统中各用户可以彼此独立地操作，互不干扰或破坏
  - 及时性：用户能在很短时间内得到系统的响应
- 优点：
  - 响应快，界面友好
  - 多用户，便于普及
  - 便于资源共享

主要知识点

操作系统的主要类型？

3) 实时系统

- 实时系统：响应时间很快，可以在毫秒甚至微秒级立即处理
- 典型应用形式：过程控制系统、信息查询系统、事务处理系统

4) 个人机系统

(1) 单用户操作系统

- 单用户操作系统特征：
  - 个人使用：整个系统由一个人操纵，使用方便。
  - 界面友好：人机交互的方式，图形界面。
  - 管理方便：根据用户自己的使用要求，方便的对系统进行管理。
  - 适于普及：满足一般的工作需求，价格低廉。

主要知识点

操作系统的主要类型？

(2) 多用户操作系统多：代表是UNIX，具有更强大的功能和更多优点。

① 网络操作系统

- 计算机网络 = 计算机技术+通信技术
- 特征：分布性、自治性、互连性、可见性
- 功能
  - 本机+网络操作系统：本地OS之上覆盖了网络OS，可以是同构的也可以是异构的。
  - 功能：实现网络通信、资源共享和保护、提供网络服务和网络接口等

② 分布式操作系统

- 定义：运行在不具有共享内存的多台计算机上，但用户眼里却像是一台计算机。（分布式系统无本地操作系统运行在各个机器上）
- 特征：分布式处理、模块化结构、利用信息通信、实施整体控制
- 特点：透明性、灵活性、可靠性、高性能、可扩充性

测试题

- 1.以下有关操作系统的叙述中，哪一个是不正确的？
- A. 操作系统管理系统中的各种资源
- B. 操作系统为用户提供的良好的界面
- C. 操作系统就是资源的管理者和仲裁者
- D. 操作系统是计算机系统中的一个应用软件
- **D**
- 2.分时操作系统的主要特点是_____。
- A. 个人独占机器资源　　B. 自动控制作业运行
- C. 高可靠性和安全性　　D. 多个用户共享计算机资源
- **D**

测试题

- 3.操作系统具有进程管理,存储管理,文件管理和设备管理的功能,下列有关描述中,哪一项是不正确的?
- A. 进程管理主要是对程序进行管理
- B. 存储管理主要管理内存资源
- C. 文件管理可以有效的支持对文件的操作，解决文件共享、保密和保护问题
- D. 设备管理是指计算机系统中除了CPU和内存以外的所有输入输出设备的管理
- A
- 4.下列哪一个不是操作系统的主要特征?
- A. 并发性　B. 共享性　C. 灵活性　D. 随机性
- C

测试题

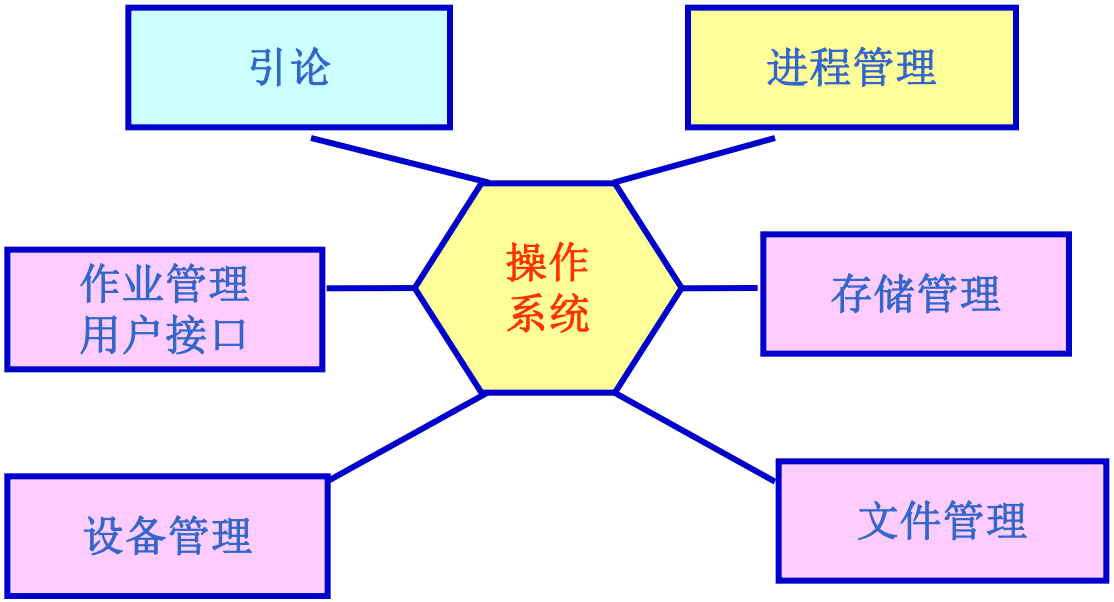
- 5.用户与操作系统打交道的手段称为_____。
- A. 命令输入　B. 广义指令　C. 通信　D. 用户接口
- D

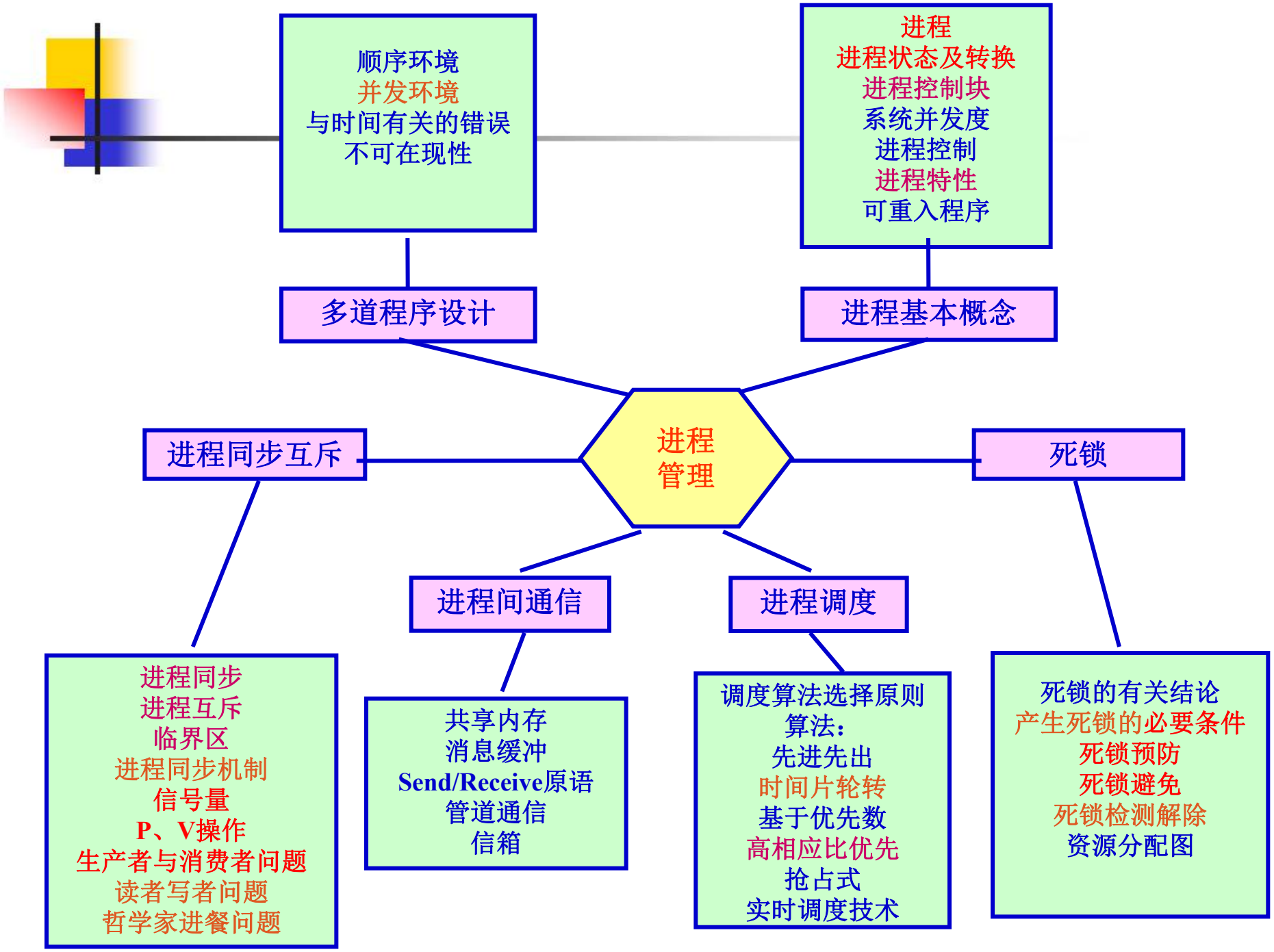
- 6. 从用户的观点看，操作系统是_____。
- A. 用户与计算机之间的接口
- B. 控制和管理计算机资源的软件
- C. 合理地组织计算机工作流程的软件
- D. 由若干层次的程序按一定的结构组成的有机体
- A

测试题

- 7.操作系统提供给程序员的接口是_____。
- A. 进程　B. 系统调用　C. 库函数　D. B和C
- B
- （系统调用是用户或其他系统程序获得操作系统服务的唯一途径）

- 8.计算机的操作系统是一种_____。
- A. 应用软件　B. 系统软件　C. 工具软件　D. 字表处理软件
- B


引论
进程管理
操作
系统
作业管理
用户接口
存储管理
设备管理
文件管理


顺序环境
并发环境
与时间有关的错误
不可在现性
进程
进程状态及转换
进程控制块
系统并发度
进程控制
进程特性
可重入程序
多道程序设计
进程基本概念
进程
管理
进程同步互斥
死锁
进程间通信
进程调度
进程同步
进程互斥
临界区
进程同步机制
信号量
P、V操作
生产者与消费者问题
读者写者问题
哲学家进餐问题
共享内存
消息缓冲
Send/Receive原语
管道通信
信箱
调度算法选择原则
算法：
先进先出
时间片轮转
基于优先数
高相应比优先
抢占式
实时调度技术
死锁的有关结论
产生死锁的必要条件
死锁预防
死锁避免
死锁检测解除
资源分配图

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

程序顺序执行与并发执行比较

顺序执行	并发执行
程序顺序执行	间断执行，多个程序各自在“走走停停”种进行
程序具有封闭性	程序失去封闭性
独享资源	共享资源
具有可在现性	失去可再现性
	有直接和简接的相互制约

主要知识点

多道程序设计概念及其优点

- 多道程序设计：是在一台计算机上同时运行两个或更多个程序。
- 多道程序设计的特点：多个程序共享系统资源、多个程序并发执行
- 多道程序设计的优点：提高资源利用率、增加系统吞吐量

主要知识点

什么是进程，进程与程序的区别和关系

- 进程的引入

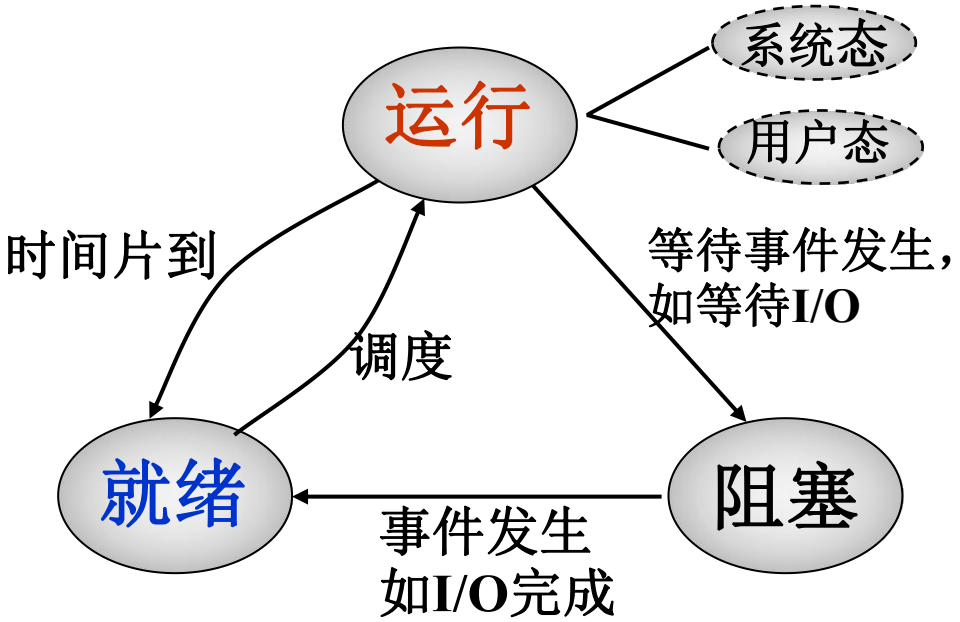
由于多道程序的特点，程序具有了并行、制约和动态的特征，就使得原来程序的概念已难以刻划和反映系统中的情况了。

- 进程：程序在并发环境下的执行过程。
- 进程与程序的主要区别：
  - 程序是永存的，进程是暂时的
  - 程序是静态的观念，进程是动态的观念
  - 进程由三部分组成：程序+数据+进程控制块（描述进程活动情况的数据结构）
  - 进程和程序不是一一对应的
    - 一个程序可对应多个进程即多个进程可执行同一程序
    - 一个进程可以执行一个或几个程序
- 进程特征：动态性、并发性、调度性、异步性、结构性

主要知识点

进程的基本状态及其转换

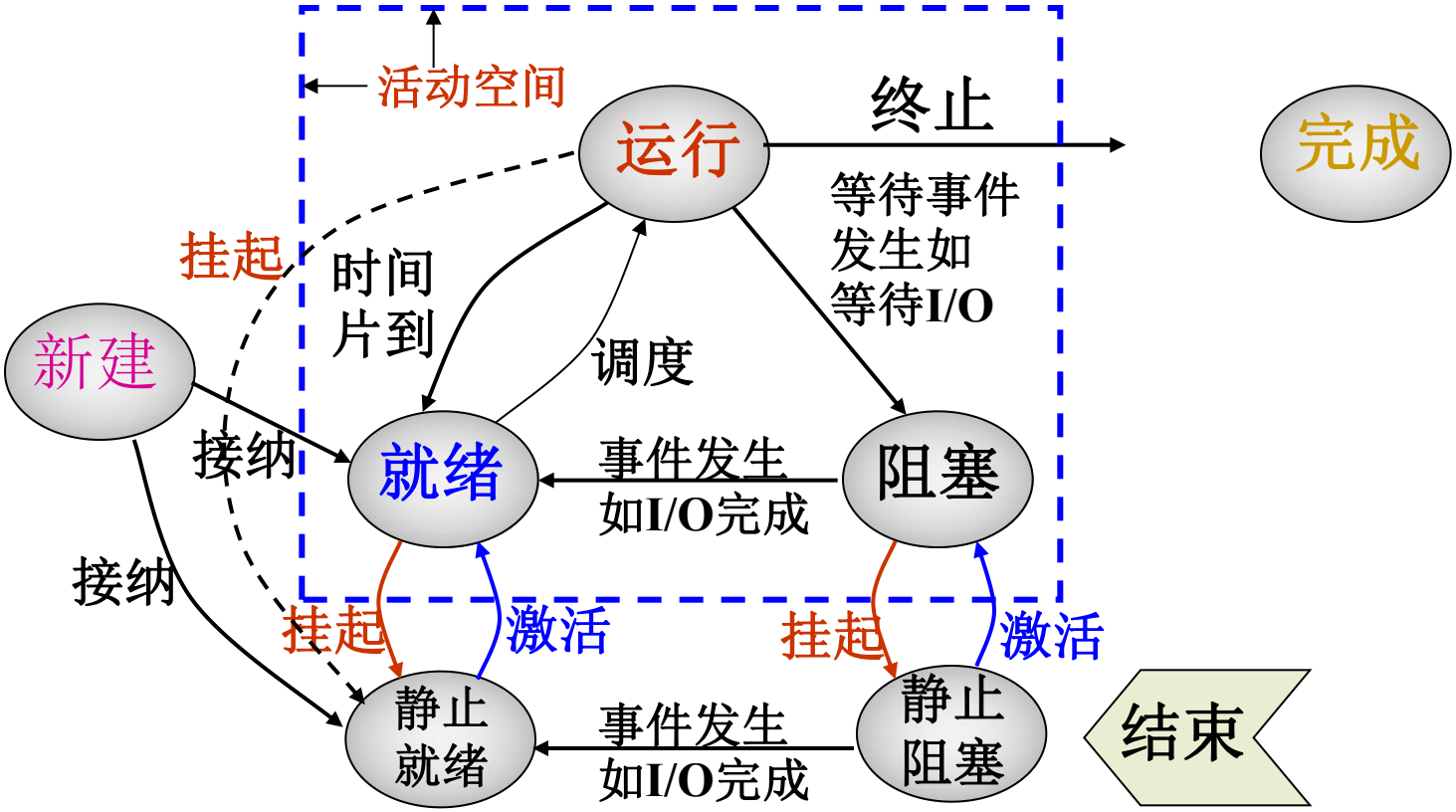
- 进程基本状态
  - 运行态（Running）：进程正在占用CPU；
  - 就绪态（Ready）：进程具备运行条件，但尚未占用CPU；
  - 阻塞态（Blocked）：进程由于等待某一事件不能享用CPU。
- 进程状态的转换
  - 就绪态->运行态
  - 运行态->就绪态
  - 运行态->阻塞态
  - 阻塞态->就绪态



进程三状态及转换图

主要知识点

- 进程的挂起状态



具有挂起状态的进程状态转换图

主要知识点

进程由哪些部分组成，进程控制块的作用及组织方式

- **进程的组成：由程序、数据集合和PCB三部分组成。**
- **进程控制块的作用：进程控制块是进程组成中最关键的部分。**
  - **每个进程有唯一的PCB。**
  - **操作系统根据PCB对进程实施控制和管理。**
  - **进程的动态、并发等特征是利用PCB表现出来的。**
  - **PCB是进程存在的唯一标志。**
- 进程控制块的组织方式：线性队列，链接表、索引表

主要知识点

进程控制

- 创建进程API

WinExec

```
UINT WinExec
(
 LPCSTR
lpCmdLine,
UINT
uCmdShow
 );
```

ShellExecute

```
 HINSTANCE
ShellExecute(
HWND hwnd,
LPCTSTR pOperation,
"edit","explore","open",
"find","print","NULL"
LPCTSTR lpFile,
LPCTSTR lpParameters,
LPCTSTR lpDirectory,
INT nShowCmd
);
```

CreateProcess

```
 BOOL CreateProcess(
LPCTSTR pApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBU,
TES lpProcessAttributes,
LPSECURITY_ATTRIBUTES
lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION,
lpProcessInformation
);
```

主要知识点

进程控制

- 打开，结束进程API

```
HANDLE OpenProcess(
DWORD dwDesiredAccess,
BOOL bInheritHandle,
DWORD dwProcessId)

BOOL TerminateProcess(
HANDLE hProcess,
UINT uExitCode);
```

测试题

- 选择题：
- 1. 当前运行的进程（　　），将引发系统进行进程调度重新分配CPU。
- A.执行了一条转移指令
- B.要求增加主存空间，经系统调用银行家算法进行测算认为是安全的
- C.执行了一条I/O指令（阻塞）
- D.执行程序期间发生了I/O完成中断

- C

测试题

- 2.下面所述步骤中，_____不是创建进程所必需的。
- A. 由调度程序为进程分配CPU　　B. 建立一个进程控制块
- C. 为进程分配内存　　D. 将进程控制块链入就绪队列
- A
- 3.分配到必要的资源并获得处理机时的进程状态是_____。
- A. 就绪状态　B. 执行状态　C. 阻塞状态　D. 撤销状态
- B

测试题

- 4.下面对进程的描述中，错误的是_____。
- A. 进程是动态的概念　　B. 进程执行需要处理机
- C. 进程是有生命期的　　D. 进程是指令的集合
- D

- 5.操作系统中，若进程从执行状态转换为就绪状态，则表示_____。
- A. 时间片到　　B. 进程被调度程序选中　　C. 等待某一事件　　D. 等待的事件发生
- A

测试题

- 6.一个进程被唤醒意味着_____。
- A. 该进程重新占有了CPU
- B. 它的优先权变为最大
- C. 其PCB移至等待队列队首
- D. 进程变为就绪状态

- **D**

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

处理机调度

- **调度：选出待分派的作业或进程**
- 处理机**调度**：是把进程指定到一个处理机中执行
- **三级调度：**
  - **高级调度（作业调度）**
  - **中级调度（内存对换）**
  - **低级调度（进程调度）**

主要知识点

进程调度方式

进程调度基本方式可分为非抢占和抢占方式：

- **非抢占方式**

进程被选中就一直运行下去（不会因为时钟中断而被迫让出CPU），直至完成工作、自愿放弃CPU、或因某事件而被阻塞才把CPU让出给其它进程。

- **抢占方式**

抢占方式发生的情况可为：新进程到达、出现中断且将阻塞进程转变为就绪进程、以及用完规定的时间片等。

主要知识点

评价调度算法的指标

- 调度性能评价准则：CPU利用率、吞吐量、周转时间、就绪等待时间和响应时间
- 吞吐量：单位时间内CPU完成作业的数量
- 周转时间：
  - 周转时间=完成时刻－提交时刻
  - 平均周转时间=周转时间／n
  - 带权周转时间=周转时间／实际运行时间
  - 平均带权周转时间=带权周转时间／n

主要知识点

调度算法

- 先来先服务调度算法
- 短作业优先调度算法
- 最高响应比调度算法
- 最短剩余时间优先调度算法
- 轮转调度算法
- 优先级调度算法
- 多级队列调度算法

主要知识点

- 先来先服务（**FCFS First come first served**）

也称为先进先出（FIFO），或严格排队方式。

- **对于作业调度**，该算法就是从后备作业队列中（按进入的时间顺序排队）选择队首一个或几个作业，调入内存，创建进程，放入就绪队列。
- **对于进程调度**，该算法就是从就绪队列中选择一个最先进入队列的进程，将CPU分配于它。

先来先服务（FCFS）

一般的情况，设有五个作业，见下表

一般作业类型的FCFS 的调度性能

作业	到达时间 T_{in}	服务时间 T_r	开始时间 T_s	结束时间 T_c	周转时间 T	带权周转时间W
A	0	3	0	3	T_A=3	W_A=1
B	2	6	3	9	T_B=7	W_B=1.17
C	4	4	9	13	T_C=9	W_C=2.25
D	6	5	13	18	T_D=12	W_D=2.40.
E	8	2	18	20	T_E=12	W_E=6
平均					$\overline{T}$ =8.60	$\overline{W}$= 2.56

同样，看到作业E的不利情况。

先来先服务（FCFS）

有四个作业（或进程），他们相应的时间见下表：

比较**极端作业**类型的FCFS 的调度性能

作业	到达时间 T_{in}	服务时间 T_r	开始时间 T_S	结束时间 T_c	周转时间T	带权周转时间W
A	0	1	0	1	$T_A=1$	$W_A=1$
B	1	100	1	101	$T_B=100$	$W_B=1$
C	2	1	101	102	$T_C=100$	$W_C=100$
D	3	100	102	202	$T_D=199$	$W_D=1.99$
平均					$\overline{T}=100$	$\overline{W}=26$

问题：C的周转时间是所需要处理时间的100倍！　作业D的周转时间近乎是C的两倍，但它的带权周转时间却低于2.0。

先来先服务（FCFS）

FCFS 算法优缺点

- 有利于长作业（进程）而不利于短作业（进程）
- 有利于CPU繁忙型作业（进程）而不利于I/O繁忙型作业（进程）

主要知识点

- 短作业优先 (SJF Shortest job first）

该算法从就绪队列中选出下一个“CPU执行期最短”的进程，将CPU分配于它。

短作业/进程优先（SJF）

下一步

降低对长作业有利的一种方法就是短作业优先策略，见下表：

表 SJF 的调度性能

作业	到达时间 T_{in}	服务时间 T_r	开始时间 T_s	结束时间 T_c	周转时间 T	带权周转时间W
A	0	3	0	3	$T_A=3$	$W_A=1$
B	2	6	3	9	$T_B=7$	$W_B=1.17$
C	4	4	11	15	$T_C=11$	$W_C=2.75$
D	6	5	15	20	$T_D=14$	$W_D=2.80$
E	8	2	9	11	$T_E=3$	$W_E=1.50$
平均	周转时间T=结束时间T_c-到达时间T_{in}				$\overline{T}=7.60$	$\overline{W}=1.84$

带权周转时间W=周转时间T/服务时间T_r

A→B→E→C→D

结束

短作业/进程优先（SJF）

SJF对短作业有利，明显的作业E提前接受了服务，并且整体性能也得到了提高；SJF的问题：

- SJF需要事先知道或至少需要估计每个作业所需的处理机时间。
- 只要不断的有短作业进入系统，就有可能使长作业长期得不到运行而“饿死”。
- SJF 偏向短作业，不利于分时系统（由于不可抢占性）。

主要知识点

- 最高响应比（HRP）
  - FCFS 强调的在系统的等待时间。
  - SJF　强调运行的时间；由此，考虑下面比值：

$$R = \frac{\text{响应时间} + \text{运行时间}}{\text{运行时间}}$$

式子R 既考虑了在系统的等待时间，又考虑了作业自身所需的运行时间，综合了FCFS与SJP各自特点。在进行进程调度时，从中**选择响应比高者**的进程投入运行。

最高响应比（HRP）

$R_D=[(9-6)+5]/5=2.2$

$R_B=[(9-6)+5]/5=1.6$

下一步

$R_E=[(9-8)+2]/2=1.5$

表 HRP 的调度性能

作业	到达时间 T_{in}	服务时间 T_r	开始时间 T_s	结束时间 T_c	周转时间 T	带权周转时间W
A	0	3	0	3	$T_A=3$	$W_A=1$
B	2	6	3	9	$T_B=7$	$W_B=1.17$
C	4	4	9	13	$T_C=9$	$W_C=2.25$
D	6	5	15	20	$T_D=14$	$W_D=2.80$
E	8	2	13	15	$T_E=7$	$W_E=3.50$
平均	带权周转时间W=周转时间T/服务时间T_r=3/3=1 结束时间T_c-到达时间T_{in}=3-0=3				$\overline{T}=8.00$	$\overline{W}=2.14$

A→ B →C→E→ D

结束

主要知识点

- 最短剩余时间（**SRT Shortest remaining time**）

► SRT是针对 SJF 增加了强占机制的一种调度算法，它总是选择预期剩余时间最短的进程。只要新进程就绪，且有更短的剩余时间，调度程序就可能抢占当前正在运行的进程。

► SRT不象FCFS偏向长进程，也不象轮转法（下个算法）产生额外的中断，从而减少了开销。

► 必须记录过去的服务时间，从而增加了开销。

► 从周转时间来看，SRT 比SJF 有更好的性能。

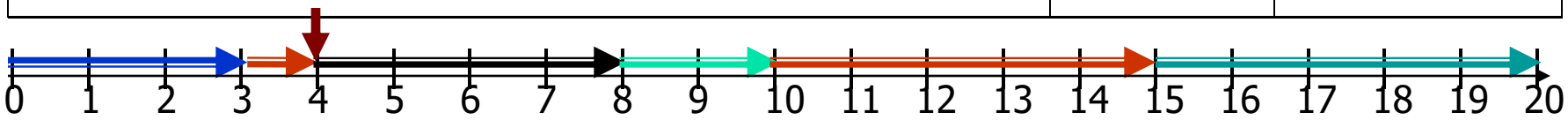
最短剩余时间（SRT）

下一步

表 SRT 的调度性能

A→B→C→E→B→D

作业	到达时间 T_{in}	服务时间 T_r	开始时间 T_s	结束时间 T_c	周转时间 T	带权周转时间W
A	0	3	0	3	T_A=3	W_A=1
B	2	5	3	15	T_B=13	W_B=2.17
C	4	1	4	8	T_C=4	W_C=1.00
D	6	5	15	20	T_D=14	W_D=2.80
E	8	2	8	10	T_E=2	W_E=1.00
平均	周转时间T=结束时间T_c-到达时间T_{in}=3-0=3				$\overline{T}$=7.20	$\overline{W}$=1.59



B剩余时间=6-1=5;

C剩余时间=4-0=4;

结束

不同调度算法对同一个 作业/进程的性能分析:

作业	到达时间T_{in}	服务时间T_r	从平均周转时间及其平均带权周转时间来看，**SRT** 好于前面的任何一个算法。	
A	0	3		
B	2	6		
C	4	4		
D	6	5		
E	8	2	$\overline{T}$	$\overline{W}$
FCFS			8.60	2.56
SJF			7.60	1.84
HRP			8.00	2.14
SRT			7.20	1.59

主要知识点

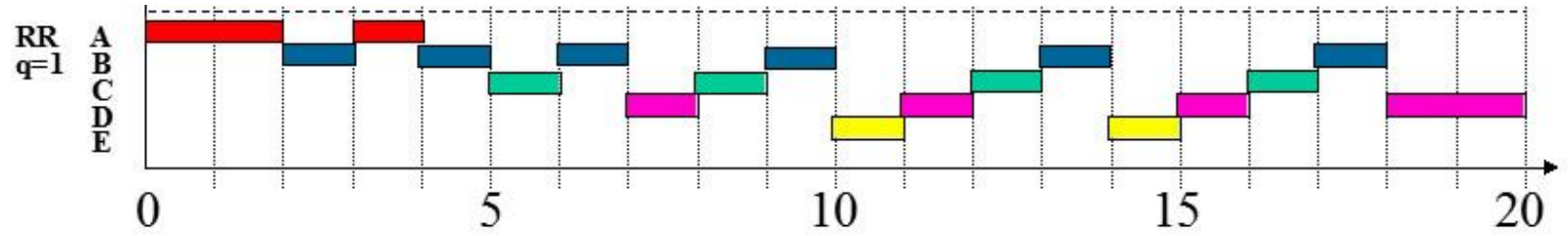
轮转（RR——Round Robin）

- 轮转调度算法是一种基于时钟的抢占策略，以时钟中断正在运行的进程，当中断发生时，当前进程被放到就绪队列中，然后再根据先来先服务算法选择下一个进程来运行。它主要用于分时系统的进程调度。
- 时间片轮转算法的关键问题是时间片的长度，若太短，频繁的进行进程切换，会导致系统开销很大。若太长，又退化为先来先服务算法了。(抢占)
- 时间片轮转算法就是在先来先服务的基础上加入时间片而已。

轮转（RR——Round Robin）

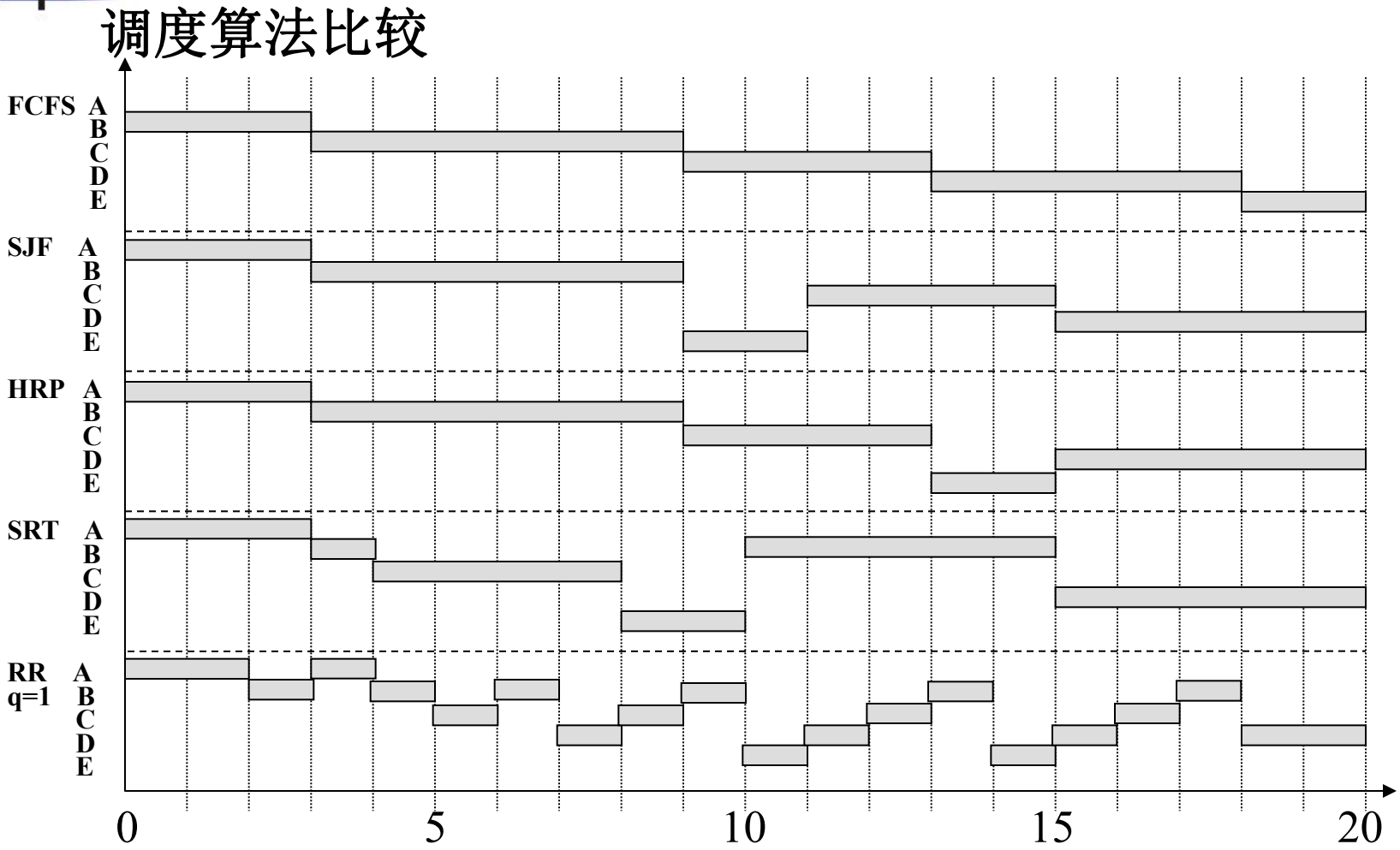
表 RR 的调度性能

作业	到达时间T_{in}	服务时间T_r	开始时间T_s	结束时间T_c	周转时间 T	带权周转时间W
A	0	3	0	4	T_A=4	W_A=1.33
B	2	6	2	18	T_B=16	W_B=2.67
C	4	4	5	17	T_C=13	W_C=3.25
D	6	5	7	20	T_D=14	W_D=2.80
E	8	2	10	15	T_E=7	W_E=3.50
平均					$\overline{T}$ =10.80	$\overline{W}$ =2.71



轮转调度算法

主要知识点



各种调度算法的比较

主要知识点

- 优先级调度算法

在时间片轮转的基础上，赋予各个进程不同的优先级，调度时依据各个进程的优先级高低进行调度。所以该算法的主要问题在于如何确定进程的优先级。

主要知识点

- 多级队列调度算法

把就绪进程放入若干个队列中，在每个队列中采用不同的调度算法，同时在每个队列之间也要进行调度。

测试题

- 选择题：
- 1.分时系统中进程调度算法通常采用_____。
- A. 响应比高者优先　　B. 时间片轮转法　　C. 先来先服务　　D. 短作业优先
- B
- 2. 下列进程调度算法中，综合考虑进程等待时间和执行时间的是（）。
- A 时间片轮转调度算法　　B 短进程优先调度算法
- C 先来先服务调度算法　　D 高响应比优先调度算法
- D

测试题

- 3为了照顾紧迫型作业，应采用（　）。
- A.先来服务调度算法　　B.短作业优先调度算法
- C.时间片轮转调度算法　　D.优先权调度算法
- D

- 4采用时间片轮转法分配CPU时，当处于运行状态的进程用完一个时间片后，它的状态是（）。
- A 阻塞　B 运行　C 就绪　D 消亡
- C

测试题

- 问答题：
- 1什么是常用调度算法的评价指标？
- 答：CPU利用率，吞吐量，周转时间（平均周转时间和带权周转时间），响应时间，系统开销
- 周转时间指的是从作业提交到外存就绪队列到作业执行完毕的时间间隔，就绪等待时间是每个作业在就绪队列所花的时间，响应时间是提交第一个请求到产生第一个响应的时间。
- 周转时间=就绪等待时间（等待时间）+运行时间

测试题

- 有5个任务A，B，C，D，E，它们几乎同时到达，预计它们的运行时间为10，6，2，4，8min。其优先级分别为3，5，2，1和4，这里5为最高优先级。对于下列每一种调度算法，计算其平均进程周转时间（进程切换开销可不考虑）。
- (1) 先来先服务（按A，B，C，D，E）算法。
- (2) 优先级调度算法。
- (3) 时间片轮转算法。（时间片为2分钟）

测试题

- (1) 采用先来先服务（FCFS）调度算法时，5个任务在系统中的执行顺序、完成时间及周转时间如下表所示

执行次序	运行时间	等待时间	周转时间
A	10	0	10
B	6	10	16
C	2	16	18
D	4	18	22
E	8	22	30

- 根据表中的计算结果，5个进程的平均周转时间T为：
- T=（10+16+18+22+30）/5=19.2min

测试题

- (2) 采用最高优先级调度（HPF）算法时，5个任务在系统中的执行顺序、完成时间及周转时间如下表所示：

执行次序	运行时间	等待时间	周转时间
B	6	0	6
E	8	6	14
A	10	14	24
C	2	24	26
D	4	26	30

- 它们的平均周转时间为：
- T=（6+14+24+26+30）/5= 20min

测试题

- (3) 如果系统采用时间片轮转（RR）算法，令时间片为2分钟，5个任务轮流执行的情况为：
- 第1轮：（A，B，C，D，E）
- 第2轮：（A，B，D，E）
- 第3轮：（A，B，E）
- 第4轮：（A，E）
- 第5轮：（A）
- 显然，5个进程的周转时间为：T1=30min、T2=22min、 T3=6min、T4=16min、T5=28min。它们的平均周转时间T为：
- T=（30+22+6+16+28）/5=20.4min

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

- **进程通信**：指的是并发进程之间相互交换信息，这种信息交换的量可大可小。
- **目的**：
  - **数据传输**：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
  - **共享数据**：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
  - **通知事件**：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
  - **资源共享**：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
  - **进程控制**：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

主要知识点

进程通信方法

- 管道
- 消息方法
- 共享内存
- 剪贴板方法
- 套接字方法

管道

- 管道：
- *匿名管道：*
  - *没有名字，在使用它们时不需要知道其名字。*
  - *允许进程与另一个与其有共同祖先的进程之间通信。*
- *有名管道：*
  - *在使用前必须知道其名字。*
  - *可以用于任何两个进程之间通信。*

匿名管道

- 主要目的是作为父进程与子进程、子进程之间通讯的联结通路。
- 在处理控制台问题时，匿名管道很有用。
- 匿名管道用得最多的功能就是重定向子进程的标准输入和标准输出。父进程可以是一个控制台或者是图形程序，而子进程必须是控制台应用程序。
- 控制台应用程序有三个用于输入输出的标准句柄，它们是标准输入、标准输出和标准错误句柄。
- *标准输入用于从控制台读或取信息。*
- *标准输出用于往控制台写或打印信息。*
- *标准错误用于汇报输出不能重定向的错误。*

匿名管道

- 通过调用 CreatePipe来创建一个匿名管道，它的原型为：
- BOOL CreatePipe(PHANDLE hReadPipe, PHANDLE hWritePipe, PSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize);
- ***pReadPipe*** *双字指针变量，指向管道读端的句柄。*
- ***pWritePipe*** *双字指针变量，指向管道写端的句柄*
- ***pPipeAttributes*** *双字指针变量，指向**SECURITY_ATTRIBUTES** 结构，其用于决定读写句柄是否可以被子进程继承*
- ***nSize*** *建议管道留给用户使用的缓冲区的大小，这仅仅是个建议值，可以用 **NULL** 来使用缺省值*
- 如果函数调用成功返回值为非零，否则为零。成功调用之后，就会得到两个句柄，一个指向管道的读出端，另一个指向管道的写入端。

有名管道

- 通过调用 CreateNamedPipe来创建一个有名管道，它的原型为：
- HANDLE CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode, DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize, DWORD nInBufferSize, DWORD nDefaultTimeOut, LPSECURITY_ATTRIBUTES lpSecurityAttributes);
- 如果函数调用成功返回值为管道句柄。

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

- 互斥与同步的概念
- 临界区
- 使用临界区的原则
- 信号量PV 机制
- 信号量 类型
- PV操作解决同步问题

进程间两种形式的制约关系

- 直接制约关系：进程间的相互联系是有意识的安排的。
- 间接制约关系：进程间要通过某种中介发生联系，是无意识安排的。

进程的同步（直接制约）

- 一般来说，一个进程相对另一个进程的运行速度是不确定的。也就是说，进程之间是在异步环境下运行的，每个进程都以各自独立的、不可预知的速度向运行的终点推进。

- 但是， 相互合作的几个进程需要在某些确定点上协调其工作。一个进程到达了这些点后，除非另一进程已完成了某些操作，否则就不得不停下来等待这些操作的结束。

- 所谓进程同步是指多个相互合作的进程，在一些关键点上可能需要互相等待或互相交换信息，这种相互制约关系称 为进程同步。

进程的同步（直接制约）

例：

司机　P1	售票员　P2
while (true)	while (true)
{	{
启动车辆;	关门;
正常运行;	售票;
到站停车;	开门;
}	}

进程的互斥（间接制约）

- 由于各进程要求共享资源，而有些资源需要互斥使用，因此各进程间竞争使用这些资源，进程的这种关系为进程的互斥。
- 其实互斥是进程同步的一种特殊情况，互斥也是为了达到让进程之间协调推进的目的。

临界区

1、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。

2、临界区（互斥区）：critical section

在进程中涉及到临界资源的程序段叫临界区

多个进程的临界区称为相关临界区

使用临界区的原则

- **空闲让进：**当无进程在互斥区时，任何有权使用互斥区的进程可进入
- **忙则等待：**不允许两个以上的进程同时进入互斥区
- **有限等待：**任何进入互斥区的要求应在有限的时间内得到满足
- **让权等待：**处于等待状态的进程应放弃占用CPU，以使其他进程有机会得到CPU的使用权

信号量的定义

信号量是一种十分有效的进程同步工具。

它的定义如下：

```
Struct  semaphore
{
Int value;
Pointer_PCB queue;
}
Semaphore S;
```

除初始化以外，对信号量的访问只能通过两个原子操作：wait和signal。 最初，这被称为 P操作(for wait;)和V操作(for signal;)。

对一信号量执行 P（down）操作，是检查其值是否大于0。若该值大于0，则将其值减1（即用掉一个保存的唤醒信号）并继续；若该值为0，则进程将睡眠，而且此时down操作并未结束。

信号量分类

- 整型信号量
- 记录型信号量
- AND型信号量
- 信号量集

整型信号量

定义为一个整型量，由两个标准原子操作wait(S)（P操作）和signal(S)（V操作）来访问。

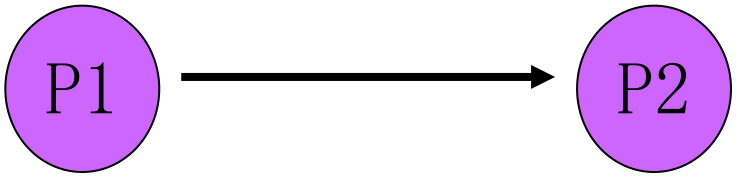
P(S): while $S \leq 0$ do no-op;

S:=S-1;

V(S): S:=S+1;

信号量的应用

- **利用信号量实现前驱关系（同步例子）**



设置一个信号量S，其初值为0,

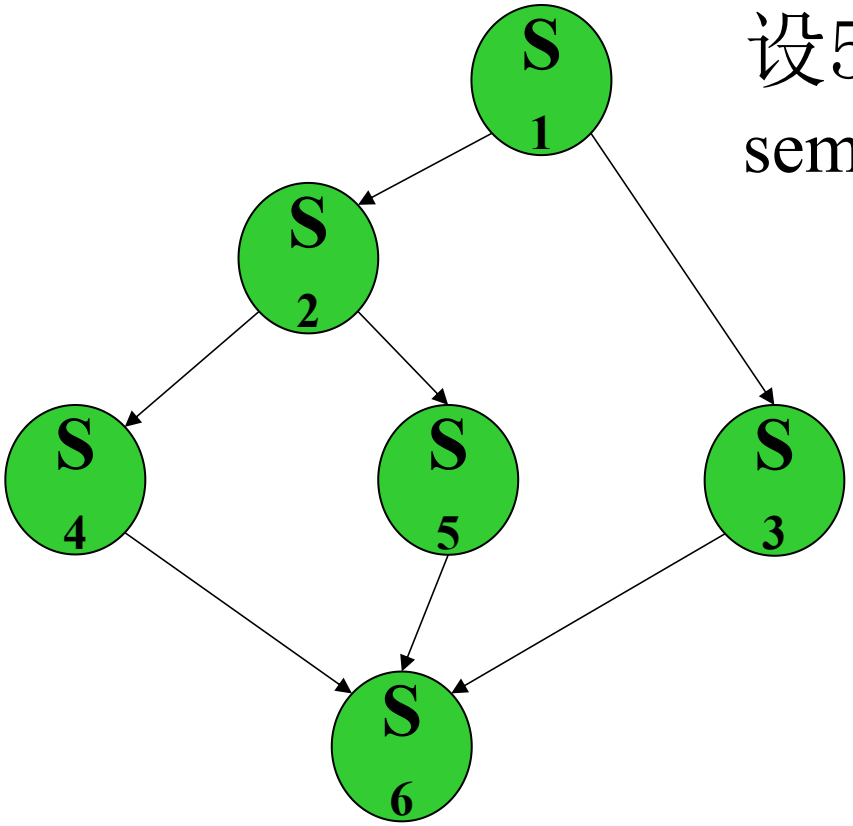
```
P1;
V(S);
```

```
P(S);
P2;
```

如此即可实现先执行P1，再执行P2

信号量的应用

- **利用信号量实现前驱关系**



设5个信号量

semaphore f1=f2=f3=f4=f5=O；

含义：分别表示进程S1、S2、S3、S4、S5是否执行完成。

信号量的应用

- 各个进程的结构为：

void S_1()
{
……
V(f1);
V(f1);
}
	void S_2()
	{
	P(f1);
	……
	V(f2);
	V(f2); }
		void S_3()
		{
		P(f1);
		……
		V(f3);
		}

void S_4()
{
P(f2);
……
V(f4);
}
	void S_5()
	{
	P(f2);
	……
	V(f5);
	}
		void S_6()
		{
		P(f3);
		P(f4);
		P(f5);
		…… }

利用信号量实现进程互斥

- mutex初值=1

Process1:

```
while (true)
 {
   P(mutex);
   critical section;
   V(mutex);
   remainder  section;
 }
```

Process2:

```
while (true)
 {
   P(mutex);
   critical section;
   V(mutex);
   remainder  section;
 }
```

记录型信号量

- 在整型信号量机制中的**wait**操作，只要是信号量**S<=0,**就会不断测试。因此，该机制并未遵循“让权等待”准则，而是使进程处于“忙等”状态。记录型信号量机制则是一种不存在“忙等”的进程同步机制。
- 但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一个临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量**value**外，还增加一个进程链表指针**L**，用于链接上述的所有等待进程。
- 记录型信号量是由于它采用了记录型的数据结构而得名的。

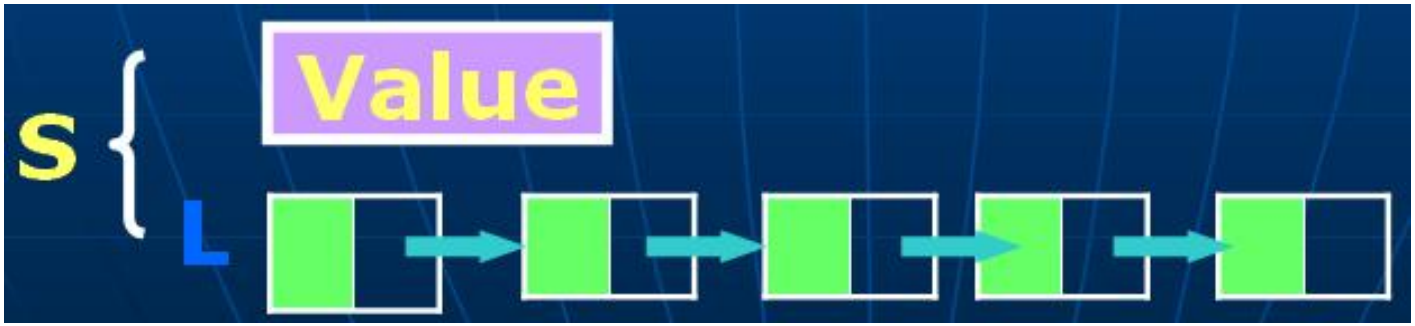
记录型信号量

- 数据结构：

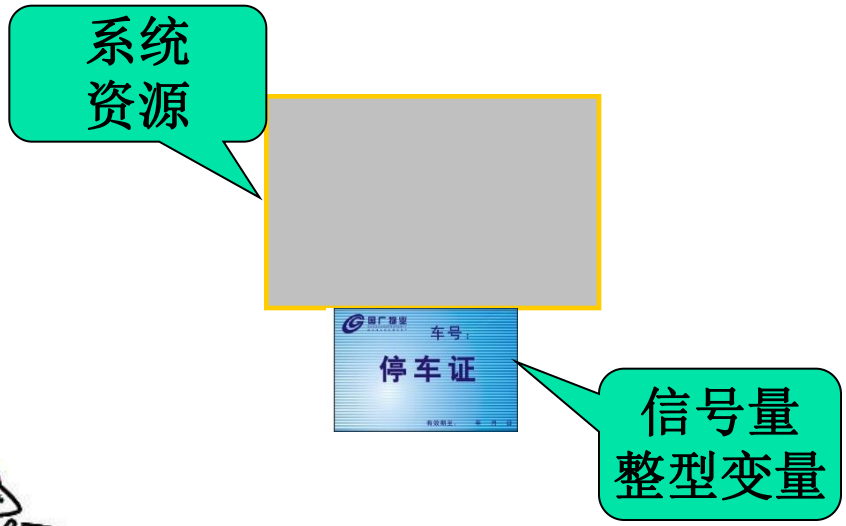
```
type semaphore=record
        value: integer;
        L: list of process;
        end
```

初值为资源信号量的数目。

链表L用于链接所有等待的进程。



场景模拟



1. P操作

2. V操作

3. S.value值的含义

P操作

```
P(s)
{
     s.value = s.value -1 ;
     if (s.value < 0)
    {
        该进程状态置为等待状态
         将该进程的PCB插入相应的等待队列末尾s.queue;
    }
}
```

P操作

S.value=3

① S.value:= S.value-1= 2

② S.value=1

③ S.value=0

④ S.value=-1

等待

⑤ S.value=-2

L

V操作

S.value=-2

① S.value:= S.value+1=-1

② S.value= 0

③ S.value= 1

④ S.value= 2

⑤ S.value= 3

等待

L

问题. S.value值的含义

若信号量**S.value**的初值为**2**，当前值为**-1**，则表示有（　）个进程等待。

A、0　B、1　C、2　D、3

S.value数值含义

S.value=3

S.Value=2

S.value=1

S.value>0:系统中可利用的资源数量

S.value数值含义

S.value=0

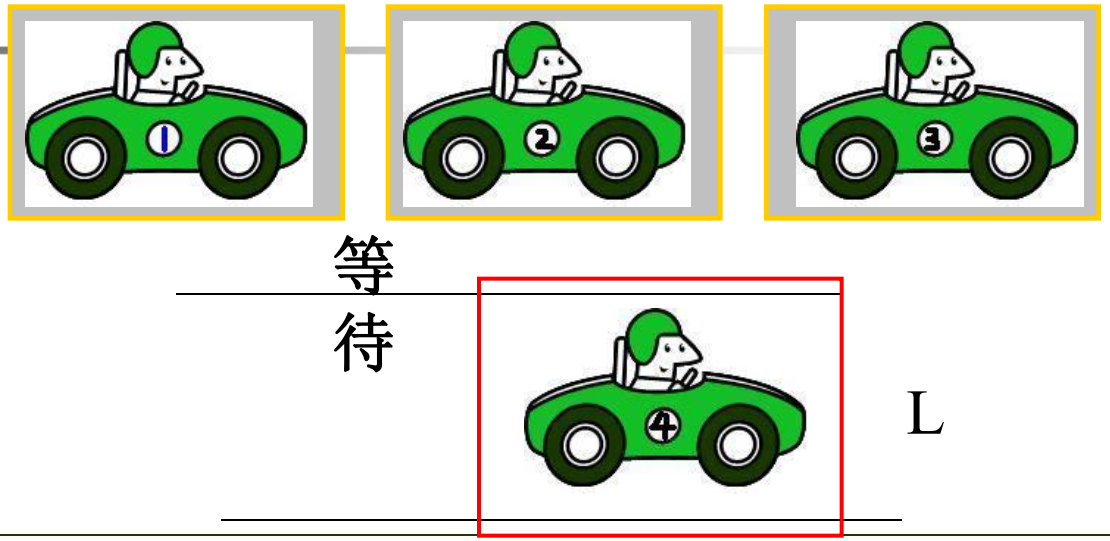
等待

L

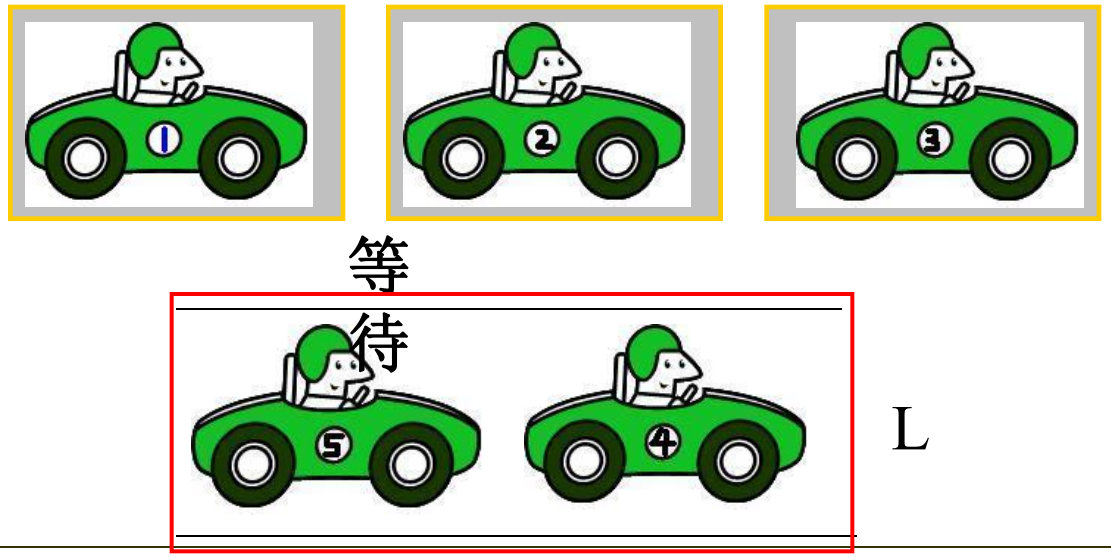
- **S.value=0:资源恰好分配完毕**

S.value数值含义

S.value= -1



S.value= -2



S.value<0:其绝对值表示在该信号量链表中已阻塞进程的数目。

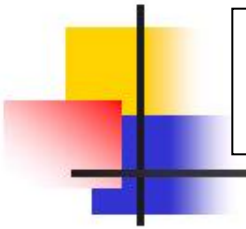
S.value值的含义

若信号量**S.value**的初值为**2**，当前值为**-1**，则表示有（B）个进程等待。

A、0　B、1　C、2　D、3

信号量分类

- 整型信号量
- 记录型信号量
- **AND型信号量**
- 信号量集



经典同步问题

- 生产者—消费者问题
- 读者-写者问题
- 哲学家进餐

生产者—消费者问题

- **使用信号量的方法解决生产者—消费者问题**

```
#define N 100                          /* 缓冲区中的槽数目 */
typedef int semaphore;                 /* 信号量是一种特殊的整型数据 */
semaphore mutex=1                      /* 控制对临界区的访问 */
semaphore empty=N                      /* 计数缓冲区的空槽数目 */
semaphore full =0                      /* 计数缓冲区的满槽数目 */

void producer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        item=produce_item();           /* 产生放在缓冲区中的一些数据 */
        p(empty);                      /* 将空槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        insert_item(item);             /* 将新数据项放到缓冲区中 */
        v(mutex);                      /* 离开临界区 */
        v(full);                       /* 将满槽的数目加1 */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        p(full);                       /* 将满槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        item=remove_item();            /* 从缓冲区中取出数据项 */
        v(mutex);                      /* 离开临界区 */
        v(empty);                      /* 将空槽数目加1 */
        consume_item(item);            /* 处理数据项 */
    }
}
```

- 该解决方案使用了三个信号量：
- **full**，用来记录充满的缓冲槽数目；
- **empty**，记录空的缓冲槽总数；
- **mutex**，用来确保生产者和消费者不会同时访问缓冲区。
- full的初值为0，empty的初值为缓冲区中槽的数目，mutex初值为1。
- 供两个或多个进程使用的信号量，其初值为1，保证同时只有一个进程可以进入临界区，称作二元信号量。
- 如果每个进程在进入临界区前都执行一个P操作，并在刚刚退出时执行一个V操作，就能够实现互斥。

哲学家进餐问题

```
#define N                5          /* 哲学家数目 */
#define LEFT             (i+N-1)%N  /* i 的左邻居编号 */
#define RIGHT            (i+1)%N    /* i 的右邻居编号 */
#define THINKING         0          /* 哲学家在思考 */
#define HUNGRY           1          /* 哲学家试图拿起叉子 */
#define EATING           2          /* 哲学家进餐 */
typedef int semaphore;              /* 信号量是一种特殊的整型数据 */
int state[N]                        /* 数组用来跟踪记录每位哲学家的状态 */
semaphore mutex=1                   /* 临界区的互斥 */
semaphore s[N];                     /* 每个哲学家一个信号量 */

void philosopher(int i)             /* i：哲学家编号，从0到N-1 */
{
  while (TRUE){
    think();                        /* 哲学家在思考 */
    take_forks(process);            /* 拿起叉子 */
    eat();                          /* 进食 */
    put_forks(process);             /* 将叉子放回桌上 */
  }
}
void take_forks(int i)              /* i：哲学家编号，从0到N-1 */
 {
   P(mutex);                        /* 进入临界区 */
   state[i] = HUNGRY;               /* 记录哲学家i处于饥饿的状态 */
   test(i);                         /* 尝试获取2把叉子 */
   V(mutex);                        /* 离开临界区 */
   P(s[i]);                         /* 如果得不到需要的叉子则阻塞 */
 }
void put_forks(int i)               /* i：哲学家编号，从0到N-1 */
{
 P(mutex);                          /* 进入临界区 */
 state[i]= THINKING;                /* 哲学家已经就餐完毕 */
 test(LEFT(i));                     /* 检查左边的邻居现在可以吃吗 */
 test(RIGHT(i));                    /* 检查右边的邻居现在可以吃吗 */
 V(mutex); }                        /* 离开临界区 */
 }
```

```
void test(int i)
{
if ( state[i] == HUNGRY
&&state[LEFT(i)] != EATING &&
state[RIGHT(i)] != EATING )
{
   state[i] = EATING;
   V(s[i]);
 }
 }
```

PV操作注意事项

PV操作实现进程互斥

- 一般模型是：

进程P1	进程P2	进程Pn
……	……	……
P（S）；	P（S）；	P（S）；
临界区；	临界区；	临界区；
V（S）；	V（S）；	V（S）；
……	……	……

- 其中信号量S用于互斥，初值为1。　使用PV操作实现进程互斥时应该注意的是：

① 每个程序中用户实现互斥的P、V操作必须成对出现，先做P操作，进临界区，后做V操作，出临界区。若有多个分支，要认真检查其成对性。

② P、V操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。

③ 互斥信号量的初值一般为1。

PV操作注意事项

- **使用信号量的方法解决生产者—消费者问题**

```
#define N 100                          /* 缓冲区中的槽数目 */
typedef int semaphore;                 /* 信号量是一种特殊的整型数据 */
semaphore mutex=1                      /* 控制对临界区的访问 */
semaphore empty=N                      /* 计数缓冲区的空槽数目 */
semaphore full =0                      /* 计数缓冲区的满槽数目 */

void producer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        item=produce_item();           /* 产生放在缓冲区中的一些数据 */
        p(empty);                      /* 将空槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        insert_item(item);             /* 将新数据项放到缓冲区中 */
        v(mutex);                      /* 离开临界区 */
        v(full);                       /* 将满槽的数目加1 */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        p(full);                       /* 将满槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        item=remove_item();            /* 从缓冲区中取出数据项 */
        v(mutex);                      /* 离开临界区 */
        v(empty);                      /* 将空槽数目加1 */
        consume_item(item);            /* 处理数据项 */
    }
}
```

- 该解决方案使用了三个信号量：
- **full**，用来记录充满的缓冲槽数目；
- **empty**，记录空的缓冲槽总数；
- **mutex**，用来确保生产者和消费者不会同时访问缓冲区。
- full的初值为0，empty的初值为缓冲区中槽的数目，mutex初值为1。
- 供两个或多个进程使用的信号量，其初值为1，保证同时只有一个进程可以进入临界区，称作二元信号量。
- 如果每个进程在进入临界区前都执行一个P操作，并在刚刚退出时执行一个V操作，就能够实现互斥。

PV操作注意事项

PV操作实现进程同步

- 一般模型是：
- 用一个信号量与一个消息联系起来，当信号量的值为0时，表示期望的消息尚未产生；当信号量的值非0时，表示期望的消息已经存在。用PV操作实现进程同步时，调用P操作测试消息是否到达，调用V操作发送消息。

- 使用PV操作实现进程同步时应该注意：

① 分析进程间的制约关系，确定信号量种类。在保持进程间有正确的同步关系情况下，哪个进程先执行，哪些进程后执行，彼此间通过什么资源（信号量）进行协调，从而明确要设置哪些信号量。

② 信号量的初值与相应资源的数量有关，也与P、V操作在程序代码中出现的位置有关。

③ 同一信号量的P、V操作要成对出现，但它们分别在不同的进程代码中。

PV操作注意事项

- **使用信号量的方法解决生产者—消费者问题**

```
#defind N 100                          /* 缓冲区中的槽数目 */
typedef int semaphore;                 /* 信号量是一种特殊的整型数据 */
semaphore mutex=1                      /* 控制对临界区的访问 */
semaphore empty=N                      /* 计数缓冲区的空槽数目 */
semaphore full =0                      /* 计数缓冲区的满槽数目 */

void producer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        item=produce_item();           /* 产生放在缓冲区中的一些数据 */
        p(empty);                      /* 将空槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        insert_item(item);             /* 将新数据项放到缓冲区中 */
        v(mutex);                      /* 离开临界区 */
        v(full);                       /* 将满槽的数目加1 */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        p(full);                       /* 将满槽数目减1 */
        p(mutex);                      /* 进入临界区 */
        item=remove_item();            /* 从缓冲区中取出数据项 */
        v(mutex);                      /* 离开临界区 */
        v(empty);                      /* 将空槽数目加1 */
        consume_item(item);            /* 处理数据项 */
    }
}
```

- 该解决方案使用了三个信号量：
- **full**，用来记录充满的缓冲槽数目；
- **empty**，记录空的缓冲槽总数；
- **mutex**，用来确保生产者和消费者不会同时访问缓冲区。
- full的初值为0，empty的初值为缓冲区中槽的数目，mutex初值为1。
- 供两个或多个进程使用的信号量，其初值为1，保证同时只有一个进程可以进入临界区，称作二元信号量。
- 如果每个进程在进入临界区前都执行一个P操作，并在刚刚退出时执行一个V操作，就能够实现互斥。

PV操作注意事项

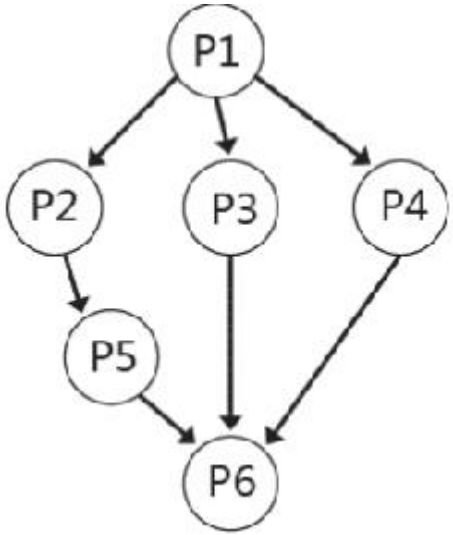
- 该问题中，无论是生产者进程还是消费者进程，**V操作的次序无关紧要**
- 但**P操作的次序不能随意交换**，否则可能造成死锁。例如，若将生产者进程中的P(empty)与P(mutex)的次序交换，则在一定条件下就会出现死锁现象。

PV操作注意事项

- 各进程须先检查自己对应的资源数，确信有可用资源后再申请对整个缓冲区的互斥操作；
- 否则，先申请对整个缓冲区的互斥操(p(mutext))，后申请自己对应的缓冲块资源(p(empty))，就可能死锁。
- 出现死锁的条件是，申请到对整个缓冲区的互斥操作后，才发现自己对应的缓冲块没有可用资源，导致挂起，这时已不能放弃对缓冲区的占用，于是导致死锁。
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，一个同步P操作与一个互斥P操作在一起时，**同步P操作在互斥P操作前。**

练习题

【例 3】P1、P2、P3、P4、P5、P6 为一组合作进程，其前趋图如图所示，试用 P、 V 操作完成这六个进程的同步。



练习题

图中说明任务启动后 P1 先执行，当其结束后 P2、P3、P4 可以开始执行，P2 完成后 P5 可以开始执行，仅当 P3、P4、P5 都执行完后，P6 才能开始执行。为了确保这一执行顺序， 设 5 个同步信号量 f1、f2、f3、f4、f5 分别表示进程 P1、P2、P3、P4、P5 是否执行完成， 其初值均为 0。这六个进程的同步描述如下：

```
Semaphore
f1=f2=f3=f4=f5=0;
Main()
{
Cobegin
P1();
P2();
P3();
P4();
P5();
Coend
}
```

```
P1()
{
...
V(f1);
V(f1);
V(f1);
}
```

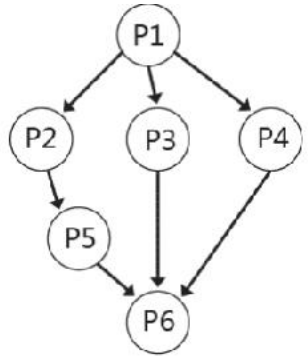
```
P2()
{
P(f1);
...
V(f2);
}
```

```
P3()
{
P(f1);
...
V(f3);
}
```

```
P4()
{
P(f1);
...
V(f4);
}
```

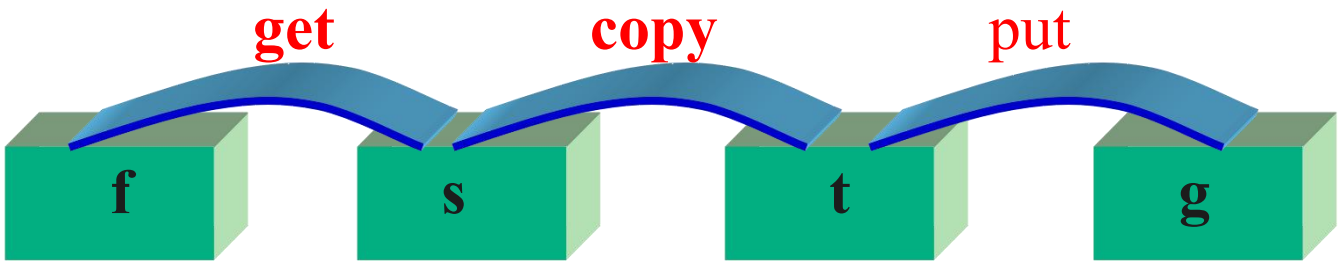
```
P5()
{
P(f2);
...
V(f5);
}
```

```
P6()
{
P(f3);
P(f4);
P(f5);
...
}
```



1、用P.V操作解决下面的同步问题

有3个进程：get，copy和put，它们对4个存储区域f、s、t和g进行操作：
其中：f有取之不尽的数据可以get；g有用之不完的空间可以put，s和t则只有一个存储空间。



1,2,3,4,...,m　　(…　　)
2,3,4,...,m　　1　　1　　(1…　　)
3,4,...,m　　2　　2　　(1,2…　　)

要解决的同步问题：

Get不能向“满”的S中放；
Copy不能从“空”的S中取；不能向“满”的T中放；
Put不能从“空”的T中取

（同步）信号量：

S_Empty，T_Empty，　{初值为1}

S_Full，　T_Full；　　{初值为0}

Get进程：

```
Begin
  Repeat
    P(S_Empty)
    T_get_S();
    V(S_Full);
  Until false;
End
```

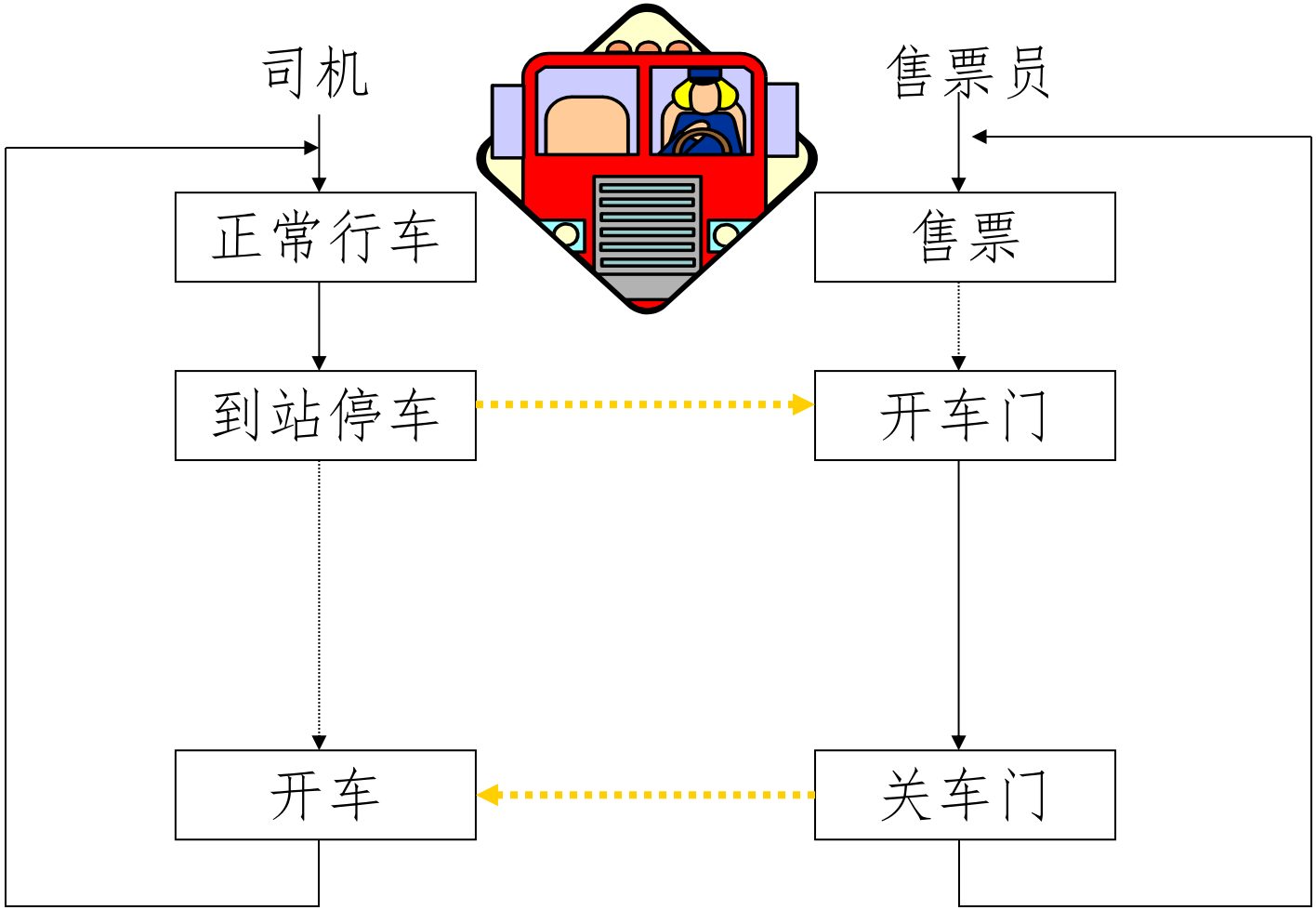
Copy进程：

```
Begin
  Repeat
    P(S_Full);
    P(T_Empty);
    S_copy_T( );
    V(T_Full);
    V(S_Empty);
  Until false;
End
```

Put进程：

```
Begin
  Repeat
     P(T_Full);
     T_put_G( );
     V(T_Empty);
  Until false;
End
```

2、重新研究司机和售票员问题，分别写出司机和售票员进程，从而实现该问题的同步



Var door，stop：semaphore：1，0

司机进程：

```
Begin
 Repeat
      P(door);
      行驶;
       …
      停车;
      V(stop);
 Until false;
End
```

乘务员进程：

```
Begin
 Repeat
      P(stop);
      开门;
      关门;
      V（door）;
      售票;
 Until false;
End
```

3、桌上有一空盘，允许存放一只水果。爸爸可向盘中放苹果，也可向盘中放桔子，儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一只水果供吃者取用，请用P、V原语实现爸爸、儿子、女儿三个并发进程的同步。

分析：

本题中，爸爸、儿子、女儿共用一个盘子，盘中一次只能放一个水果。

当盘子为空时，爸爸可将一个水果放入果盘中。

若放入果盘中的是桔子，则允许儿子吃，女儿必须等待；

若放入果盘中的是苹果，则允许女儿吃，儿子必须等待。

本题实际上是生产者-消费者问题的一种变形：

这里，生产者放入缓冲区的产品有两类，消费者也有两类，每类消费者只消费其中固定的一类产品

设置三个信号量:S、So、Sa，

S：表示盘子是否为空，其初值为1;
So：表示盘中是否有桔子，其初值为0;
Sa：表示盘中是否有苹果，其初值为0。

```
Father进程:
while(1)
{
        P(S);
        将水果放入盘中;
        if（放入的是桔子）V(So);
        else  V(Sa);
}
```

```
Son进程:
 while(1)
{
        P(So);
        从盘中取出桔子;
        V(S);
        吃桔子;
       }
}
```

```
Daughter进程:
 while(1)
{
        P(Sa);
        从盘中取出苹果;
        V(S);
        吃苹果;
}
```

具体的规范格式如下（以苹果桔子题目为例：）

解：设置三个信号量S、So、Sa，信号量S表示盘子是否为空，其初值为1；信号量So表示盘中是否有桔子，其初值为0；信号量Sa表示盘中是否有苹果，其初值为0。同步描述如下：

```
int S=1;
int Sa=0;
int So=0;
    main()
    {cobegin
        father();     /*父亲进程*/
        son();        /*儿子进程*/
        daughter();   /*女儿进程*/
     coend
   }

son()
   {   while(1)
       { P(So);
         从盘中取出桔子;
         V(S);
         吃桔子;
        }
   }
```

```
father()
{       while(1)
        {  P(S);
           将水果放入盘中;
           if（放入的是桔子）V(So);
           else  V(Sa);
        }
}

daughter()
{       while(1)
        {  P(Sa);
           从盘中取出苹果;
           V(S);
           吃苹果;
        }
}
```

小结：同步问题解法

分析问题中涉及的进程；
分析问题中的同步关系（竞争，合作）；
参照所竞争的资源设置信号量，并赋予初值；
写出各个进程的描述；
检查每个进程的描述，看是否会出现死锁现象并改正之。

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

- 线程的基本概念及组成

线程，是进程内一个相对独立的进程流或控制流，是处理机分配的实体。线程有自己的执行堆栈，程序计数器，通用寄存器组和状态标记，当然少不了线程控制块。同一个进程的多个线程共享该进程的全部资源。

- 引入线程的好处：

充分提高了操作系统的共享性，并发性，提高了机器的响应速度，提高了多处理机体系结构的利用率。

主要知识点

- Windows 多线程编程方法
- **HANDLE CreateThread(**

  PSECURITY_ATTRIBUTES psa,

  DWORD cbStack,

  PTHREAD_START_ROUTINE pStartAddr,

  PVOID pvParam,

  DWORD fdwCreate,

  PDWORD pdwThreadId);
- DWORD SuspendThread(HANDLE hThread)
- DWORD ResumeThread(HANDLE hThread);
- BOOL TerminateThread(HANDLE hThread,DWORD dwExitCode);

主要知识点

线程间同步

1. 临界区
2. 互斥量
3. 事件
4. 信号量

主要知识点

- 进程概念与进程控制
- 进程调度
- 进程通信
- 进程同步
- 线程
- 死锁

主要知识点

- 死锁的基本概念
- 死锁产生的四个必要条件
- 死锁的处理方法
  - 死锁的预防
  - 死锁的避免-银行家算法
  - 死锁检测-资源分配图
  - 死锁的解除

死锁的基本概念

- 死锁的定义
- 所谓死锁，是由于多个进程竞争资源而引起的一种僵局，若无外力作用，则这些进程永远不能向前推进。在一组进程中，每个进程都在等待被该组进程中其他进程占用的资源，却又无法等到该资源，这种现象称为进程死锁。这组进程称为死锁进程。
- 进程死锁两大根本原因：
- 1）竞争系统资源　2）进程的推进顺序不当

死锁产生的四个必要条件

- **互斥**：在一段时间内，一个资源只能由一个进程独占使用。如果另一个进程请求这个资源，那么该进程必须等待这个资源被释放。
- **持有并等待**：允许一个进程在持有已占有资源的条件下等待另一个已经被其他进程占有的资源。
- **不可抢占**：进程所占有的资源只能由他自己释放，不能被别人强行抢占。
- **循环等待**：存在一组等待进程{P0, P1, …, Pn}，必须：P0 等待 P1 持有的资源，P1 等待 P2持有的资源，…，Pn-1等待Pn 持有的资源，而 Pn 等待 P0 持有的资源。

死锁的处理方法

- 预防死锁：破坏死锁的四个必要条件中的一个或几个来预防死锁的发生。
- 避免死锁：在资源的动态分配过程中使用某种方法防止系统进入不安全状态。
- 检测死锁和解除死锁：允许系统进入死锁状态，然后检测并恢复。

死锁的预防

—— 破坏死锁的四个必要条件（常针对条件2, 3, 4）

（1）破坏互斥条件（不可行）

即允许多个进程同时访问资源。但由于资源本身固有特性限制，有的资源根本不能同时访问，只能互斥访问，所以不可能用破坏互斥条件来预防死锁。

死锁的预防

（2）破坏请求和保持条件

可采用预先静态分配方法，即要求进程在运行之前一次申请它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦运行后，这些资源全归其占有，同时它也不再提出其它资源要求，这样可以保证系统不会发生死锁。

此方法虽简单安全，但降低了资源利用率，同时必须预知进程所需要的全部资源。

死锁的预防

（3）破坏不可剥夺条件

一个已经获得某些资源的进程，若又请求新的资源时不能得到满足，则它必须释放出已获得的所有资源，以后需要资源时再请求。即一个进程已获得的资源在运行过程中可被剥夺。从而破坏了“不剥夺”条件。

这种方法实现较复杂，会增加系统开销，降低系统吞吐量。

死锁的预防

（4）破坏环路条件

可采用有序资源分配方法，即将系统中的所有资源都按类型赋予一个编号，要求每一个进程均严格按照编号递增的次序来请求资源，同类资源一次申请完。也就是，只要进程提出请求资源Ri，则在以后的请求中，只能请求Ri后面的资源，这样不会出现几个进程请求资源而形成环路。

该方法虽提高了资源的利用率，但编号难，加重进程负担及因使用资源顺序与申请顺序不同而造成资源浪费。

避免死锁的算法-银行家算法

- 它的模型基于一个小城镇的银行家。假定一个银行家拥有资金，数量为Σ，被N个客户共享。银行家对客户提出下列约束条件：

① 每个客户必须预先说明自已所要求的最大资金量；

② 每个客户每次提出部分资金量申请各获得分配；

③ 如果银行满足了客户对资金的最大需求量，那么，客户在资金动作后，应在有限时间内全部归还银行。

- 只要每个客户遵守上述约束，银行家将保证做到：若一个客户所要求的最大资金量不超过Σ，则银行一定接纳该客户，并可处理他的资金需求；银行在收到一个客户的资金申请时，可能因资金不足而让客户等待，但保证在有限时间内让客户获得资金。
- 在银行家算法中，客户可看做进程，资金可看做资源，银行家可看做操作系统。

银行家算法

- 银行家金庸先生共有四个客户需要申请贷款，每个客户被授予一定数量的贷款额度(单位1万美金)。
- 金庸先生很有经济头脑，他知道，不可能所有客户同时需要最大贷款额，所有他保留10个单位而不是22个单位的资金来为客户服务。

客户	已贷款	最大需求
张无忌	0	6
令狐冲	0	5
乔峰	0	4
杨过	0	7

图A　　空闲：10

银行家算法

- 各位大侠们都有自己的生意，在某些时候需要贷款。在某一时刻，具体贷款情况如图**B**。这个状态安全吗？
- 这个状态是安全的，由于保留了**2**个单位资金，金庸能够拖延除了乔峰之外的其他客户的请求. 先让乔峰把生意完成，然后还款所贷的**4**个单位的资金。有了这**4**个单位的资金，金庸就可以给杨过或令狐冲贷款，以此类推。

客户	已贷款	最大需求
张无忌	1	6
令狐冲	1	5
乔峰	2	4
杨过	4	7

图B　　空闲：**2**

银行家算法

- 这一天，令狐冲来再要求贷款1万美金。可否贷给他？
- 如果贷款给令狐大侠，状态如图D。
- 该状态是不安全的。如果忽然所有的大侠都请求最大限额贷款。而金庸无法满足任何一个，就会产生死锁，可能导致各位大侠资金链断裂，全部破产。
- 不安全状态并不一定引起死锁，由于各个大侠不一定需要最大贷款额度，但金庸老先生不敢报侥幸心理。

客户	已贷款	最大需求
张无忌	1	6
令狐冲	1	5
乔峰	2	4
杨过	4	7

图C　　空闲：2

客户	已贷款	最大需求
张无忌	1	6
令狐冲	2	5
乔峰	2	4
杨过	4	7

图D　　空闲：1

银行家算法

- 银行家算法就是对每一个请求进行检查，检查如果满足这一请求是否会达到安全状态。
- 若是，就满足该请求。
- 若否，就推迟对这一请求的满足。

—— 单资源的银行家算法

可扩展到多资源的银行家算法

避免死锁的算法-银行家算法

为实现银行家算法，须设置若干数据结构。假定系统中有n个进程（P1，P2，...，Pn），m类资源（R1，R2，...，Rm），银行家算法中使用的数据结构如下：

◆可利用资源向量：available[j]=k，资源Rj类有k个可用

◆最大需求矩阵：Max[i,j]=k,进程Pi最大请求k个Rj类资源

◆分配矩阵：Allocation[i,j]=k,进程Pi分配到k个Rj类资源

◆需求矩阵：Need[i,j]=k,进程Pi还需要k个Rj类资源

三个矩阵的关系：

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$

银行家算法描述—资源分配算法（1）

设$Request_i$是进程Pi的请求向量，设$Request_i[j]=k$，表示进程Pi请求分配Rj类资源k个。当进程Pi 发出资源请求后，系统按如下步骤进行检查：

(1) 如$Request_i[j] \leq Need[i,j]$, 转(2)；否则出错，因为进程申请资源量超过它声明的最大量。

(2) 如$Request_i[j] \leq Available[j]$, 转(3)；否则表示资源不够，需等待。

银行家算法描述—资源分配算法（2）

(3) 系统试分配资源给进程Pi，并作如下修改：

Available[j]:= Available[j]- $Request_i$[j]

Allocation[i,j]:= Allocation[i,j]+ $Request_i$[j]

Need[i,j]:= Need[i,j]- $Request_i$[j]

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，则正式进行分配，否则恢复原状态，让进程Pi等待。

银行家算法描述—安全性检查算法

为了进行安全性检查，需要定义如下**数据结构**：

- int Work[m]　工作变量，记录可用资源。开始时， Work:= Available
- boolean Finish[n]　工作变量，记录进程是否执行完。开始时， Finish[i]=false;当有足够资源分配给进程Pi时，令Finish[i]=true。

银行家算法描述—安全性检查算法

安全性检查算法：

(1) Work := Available

Finish[i] := false

(2) 寻找满足如下条件的进程Pi

Finish [i] = false 并且 Need[i,j] ≤ Work

如果找到，转(3)，否则转(4)

(3) 当进程Pi获得资源后，可顺利执行完，并释放分配给它的资源，故执行：

Work := Work + Allocation

Finish[i] := true

转 (2).

(4) 若所有进程的Finish [i] = true，则表示系统处于安全状态，否则处于不安全状态。

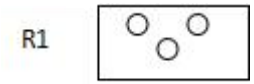
死锁的检测

资源分配图

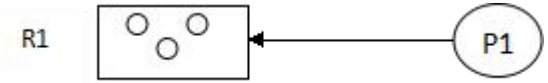
如果资源分配图中不存在环路，则系统中不存在死锁；反之，如果资源分配图中存在环路，则系统中可能存在死锁，也可能不存在死锁。

资源分配图化简方法

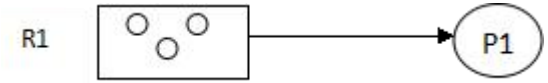
表示：进程p1



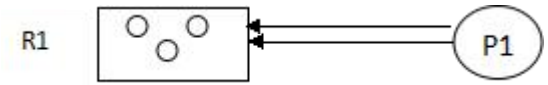
表示：有3个R1类资源



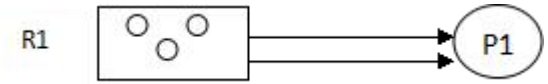
表示：进程p1申请一个R1类资源



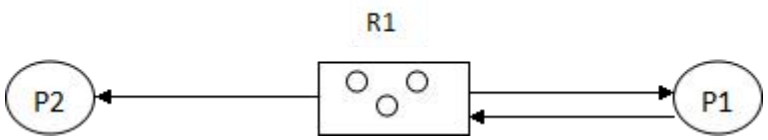
表示：系统分配一个R1类资源给进程p1，此时，系统还剩下2个R1类资源



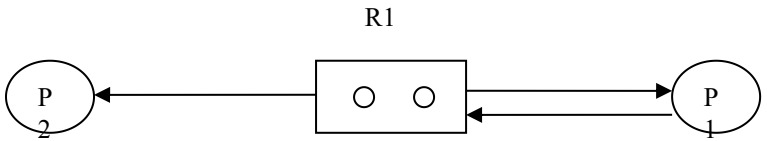
表示：进程p1申请2个R1类资源



表示：系统分配2个R1类资源给进程p1，此时，系统还剩下1个R1类资源



表示：系统分配一个R1资源给进程p2，然后又分配一个R1类资源给进程p1，最后进程p1收到一个R1类资源后又继续申请1个R1类资源，此时，还剩下一个R1类资源可以分配给P1，但还没分配给P1。（注意：图中P1的申请是还没得到响应的，不要以为R1指向P1的那个箭头是响应P1的申请，而分配了资源给P1）



表示：系统分配一个R1资源给进程p2，然后又分配一个R1类资源给进程p1，最后进程p1收到一个R1类资源后又继续申请1个R1类资源，此时，系统已经没有R1类资源可以分配给进程P1了，于是p1进程受到阻塞。

化简资源分配图

方法步骤：

①先看系统还剩下多少资源没分配，再看有哪些进程是不阻塞（“不阻塞”即：系统有足够的空闲资源分配给它）的

②接着把不阻塞的进程的所有边都去掉，形成一个孤立的点，再把系统分配给这个进程的资源回收回来

③这样，系统剩余的空闲资源便多了起来，接着又去看看剩下的进程有哪些是不阻塞的，然后又把它们逐个变成孤立的点。

④最后，所有的资源和进程都变成孤立的点。这样的图就叫做“可完全简化”。

如果一个图可完全简化，则不会产生死锁；如果一个图不可完全简化（即：图中还有“边”存在），则会产生死锁。这就是“死锁定理”

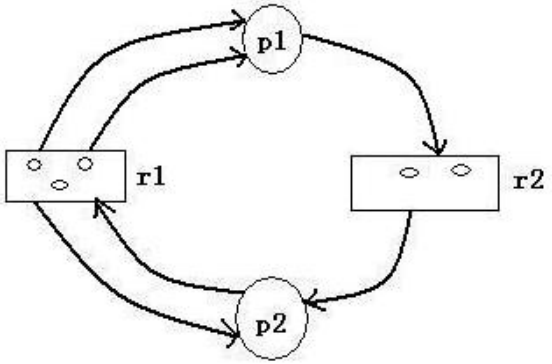


图1

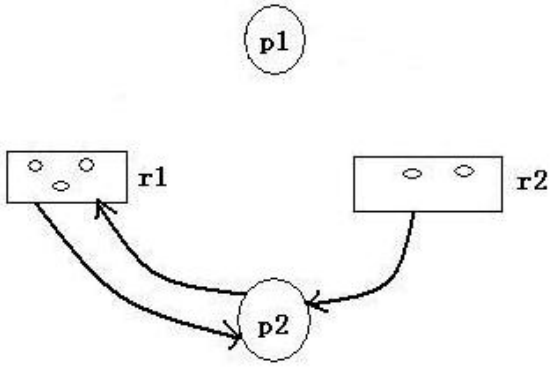


图2

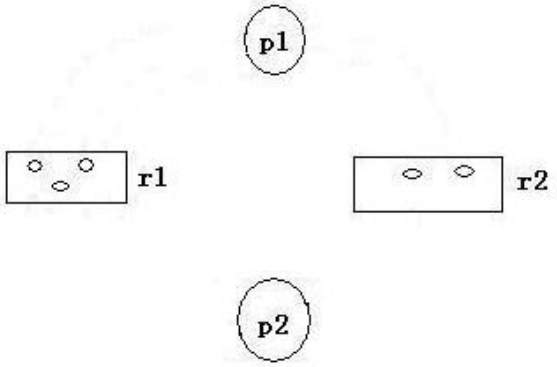


图3

第一步：先看R1资源，它有三个箭头是向外的，因此它一共给进程分配了3个资源，此时，R1没有空闲的资源剩余。

第二步：再看R2资源，它有一个箭头是向外的，因此它一共给进程分配了1个资源，此时，R2还剩余一个空闲的资源没分配。

第三步：看完资源，再来看进程，先看进程P2，它只申请一个R1资源，但此时R1资源已经用光了，所以，进程P2进入阻塞状态，因此，进程P2暂时不能化成孤立的点。

第四步：再看进程P1，它只申请一个R2资源，此时，系统还剩余一个R2资源没分配，因此，可以满足P1的申请。这样，进程P1便得到了它的全部所需资源，所以它不会进入阻塞状态，可以一直运行，等它运行完后，我们再把它的所有的资源释放。相当于：可以把P1的所有的边去掉，变成一个孤立的点，如图2所示。

第五步：进程P1运行完后，释放其所占有的资源（2个R1资源和1个R2资源），系统回收这些资源后，空闲的资源便变成2个R1资源和1个R2资源，由于进程P2一直在申请一个R1资源，所以此时，系统能满足它的申请。这样，进程P2便得到了它的全部所需资源，所以它不会进入阻塞状态，可以一直运行，等它运行完后，我们再把它的所有的资源释放。相当于：可以把P2的所有的边都去掉，化成一个孤立的点，变成图3

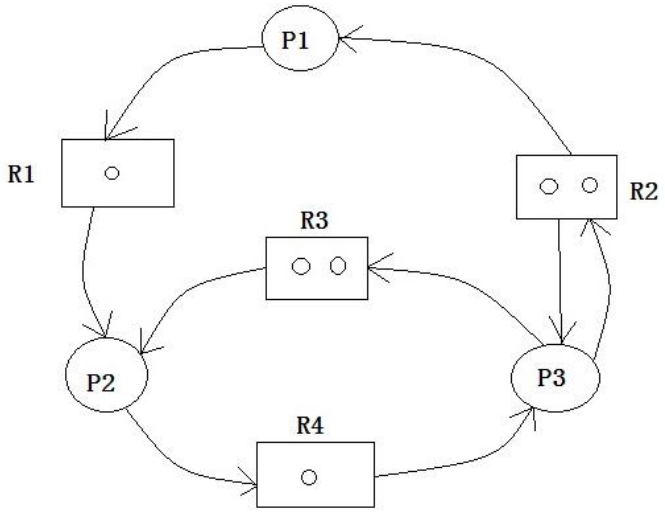
由于这个资源分配图可完全简化，因此，不会产生死锁。

第一步：先看R1资源，它有1个箭头是向外的，因此它一共给进程分配了1个资源，此时，R1没有空闲的资源剩余。

第二步：再看R2资源，它有2个箭头是向外的，因此它一共给进程分配了2个资源，此时，R2没有空闲的资源剩余。

第三步：再看R3资源，它有1个箭头是向外的，因此它一共给进程分配了1个资源，此时，R3还剩余一个空闲的资源没分配。

第四步：再看R4资源，它有1个箭头是向外的，因此它一共给进程分配了1个资源，此时，R4没有空闲的资源剩余。



第五步：从上面4步可以看出，整个系统只剩下R3一个空闲资源没分配。

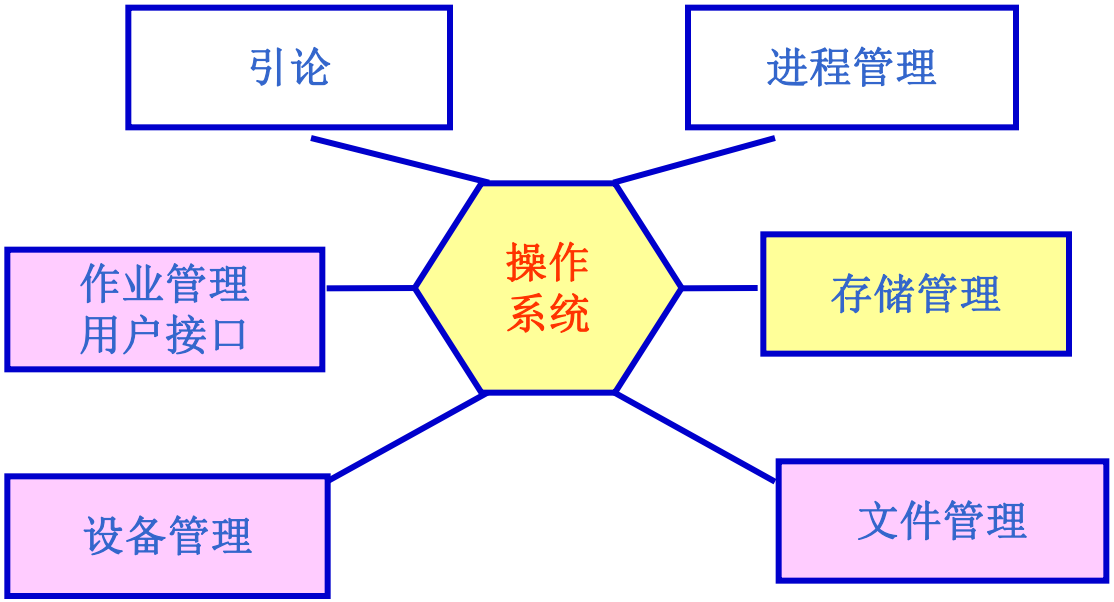
第六步：看完资源，再来看进程，先看进程P1，它只申请一个R1资源，但此时R1资源已经用光了，所以，进程P1进入阻塞状态，因此，进程P1暂时不能化成孤立的点。

第七步：再看进程P2，它只申请一个R4资源，但此时R4资源已经用光了，所以，进程P2进入阻塞状态，因此，进程P2暂时不能化成孤立的点。

第八步：再看进程P3，它申请一个R2资源和一个R3资源，但此时R2资源已经用光了，所以，进程P3进入阻塞状态，因此，进程P3暂时不能化成孤立的点。

第九步：从第六步至第八步可以看出，3个进程都不能化成孤立的点。

因此，此图不可完全化简，也就是说，这个图会产生死锁。


引论
进程管理
操作
系统
作业管理
用户接口
存储管理
设备管理
文件管理