操作系统
第四讲

王宇英
wangyy@nwpu.edu.cn

Review

用户与操作系统的接口

作业控制级接口—作业管理

程序级接口—系统调用

第三章

进程管理

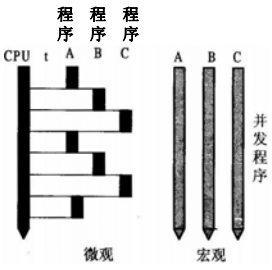
进程的概念

进程概念的引入

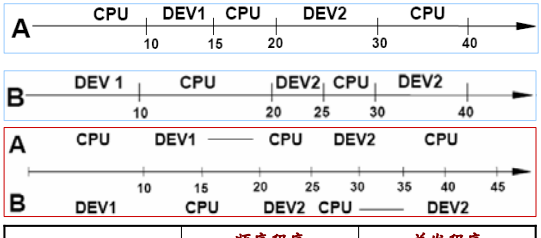
进程的表示和状态转换

3.1 进程概念的引入

- 顺序程序
  -Uniprogramming
  - 顺序性
  - 封闭性
  - 可再现性
- 并发程序
  -Multiprogramming



并发 vs. 顺序



	顺序程序	并发程序
CPU利用率	40/80=50%	40/45=89%
DEV1利用率	15/80=18.75%	15/45=33%
DEV2利用率	25/80=31.25%	25/45=56%

并发的特点

- 充分利用系统资源
  - CPU和外设可以并行起来，利用率都增加
- 平均响应时间短
  - 并发(二者<=45秒) vs. 顺序(其中之一=80秒)
- 系统吞吐量大
  - 并发(两任务/45秒) vs. 顺序(两任务/80秒)
- 失去封闭性
- 程序与机器执行程序的活动不再一一对应
- 并发程序间相互制约

与并发有关的错误

一飞机订票系统，两个终端，运行T1、T2进程

T1 :	T2:
...	...
read(x);	read(x);
if x>=1 then	if x>=1 then
x:=x-1;	x:=x-1;
write(x);	write(x);
...	...

多道程序系统

- 程序并发执行：允许多个程序同时进入内存并运行，提高了系统效率
- 资源共享：各程序因资源竞争或并行程序间需要相互协同而引起的相互关系；
- 程序概念不确切
  - 程序本身完全是一个静态的概念
  - 程序概念已不能反映系统中的并行特性

进程的概念

a program in execution

- **进程**
  - 程序在处理机上执行时所发生的活动（Dijkstra）
  - 是一个容器，该容器用以聚集相关资源（A. S. Tanenbaum）
  - 是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位

程序与进程之间的区别

- 程序是静态的，进程是动态的
- 进程与程序的组成不同，进程＝程序＋数据＋PCB
- 进程的存在是暂时的，程序的存在是永久的
- 一个程序可以对应多个进程，一个进程可以包含多个程序

进程的特征

- **动态性**：进程是程序的一次执行，有着“创建”、“活动”、“暂停”、“撤消”等过程，具有一定的生命期，是动态地产生、变化和消亡的。
- **并发性**：进程之间的动作在时间上可以重叠，即系统中有若干进程都已经“开始”但又没有“结果”，称这些进程为并发进程。
- **独立性**：进程是系统调度和资源分配的独立单位，它具有相对独立的功能，拥有自己独立的进程控制块PCB。
- **异步性**：各个并发进程按照各自独立的、不可预知的速度向前推进。
- **相互制约性**：并发进程之间具有直接或间接的关系，在运行过程中需要进行必要的交互（同步、互斥和数据通信等），以完成特定的任务。

如何用进程来实现并发？

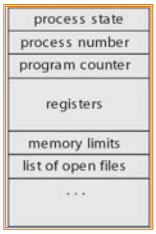
- 将CPU时间分割后给不同的进程
  - 任何时刻只有一个进程是运行的
  - 不同的进程可分配不同的CPU时间
- 将其它资源加以限制后分配给不同的进程
  - 如内存映射: Give each process their own address space
  - 内核/用户模式: through system calls

3.2 进程的表示和状态转换

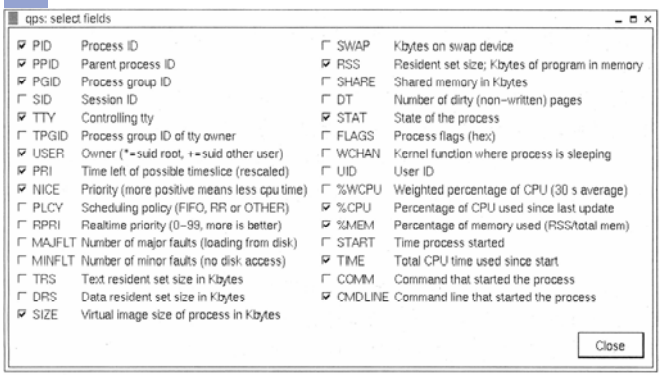
- 进程控制块PCB（Process Control Block）
  - 进程的组成： program+data+PCB
  - 系统为了管理进程设置的一个专门的数据结构，用来记录进程的外部特征，描述进程的变化过程
- PCB是系统感知进程存在的唯一标志，进程与PCB是一一对应的
  - 包含了进程的描述信息和控制信息，
  - 是进程的动态特征的集中反映，
  - 系统根据PCB而感知某一进程的存在

PCB的内容

- 进程描述信息：
  - 进程标识符(process ID)，唯一，通常是一个整数
  - 进程名，通常基于可执行文件名（不唯一）
  - 用户标识符(user ID)；进程组关系
- 进程控制信息：
  - 当前状态
  - 优先级(priority)
  - 代码执行入口地址
  - 程序的外存地址
  - 运行统计信息（执行时间、页面调度）
  - 进程间同步和通信；阻塞原因
- 所拥有的资源和使用情况：
  - 虚拟地址空间的现状
  - 打开文件列表
- CPU现场保护结构：寄存器值

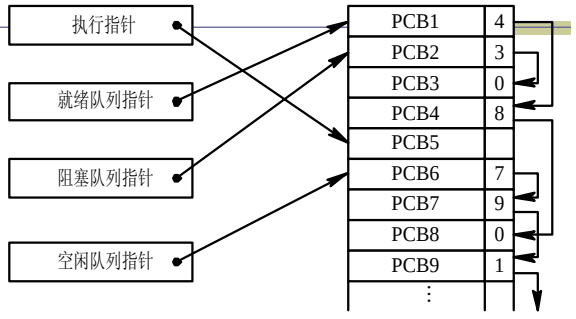


Linux中的进程描述参数

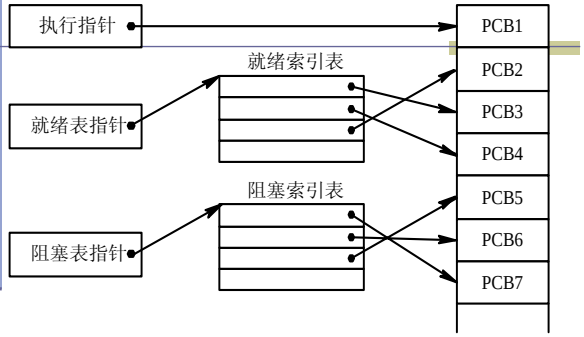


PCB表组织方式

- PCB表：
  - 系统把所有PCB组织在一起，并把它们放在内存的固定区域，就构成了PCB表
  - PCB表的大小决定了系统中最多可同时存在的进程个数，称为系统的并发度
- 组织方式：
  - 链表：同一状态的进程其PCB成一链表，多个状态对应多个不同的链表：就绪链表、阻塞链表
  - 索引表：同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表：就绪索引表、阻塞索引表

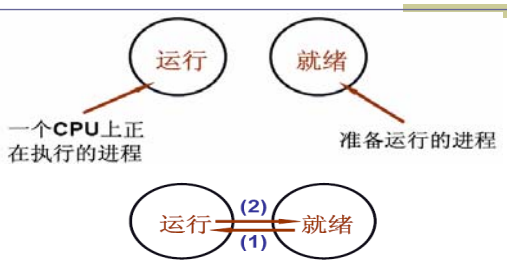


PCB链表队列



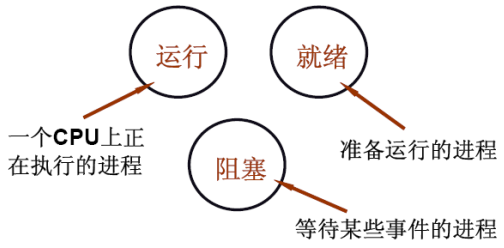
按索引方式组织PCB

进程的状态



- (1) 就绪—运行：该进程被配遣(Dispatch)
- (2) 运行—就绪：运行被中断，如时间片用完

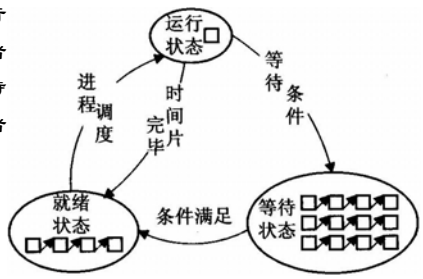
- 引入阻塞态：由于某种原因(等待设备)不能运行



- 最基本的进程状态有三种：
- 运行状态（Running），进程占有CPU，并在CPU上运行；
- 就绪状态（Ready），一个进程已经具备运行条件，但由于无CPU暂时不能运行的状态（当调度给其CPU时，立即可以运行）；
- 阻塞状态（Blocked），也称为等待状态，进程因等待某种事件的发生而暂时不能运行的状态（即使CPU空闲，该进程也不可运行）

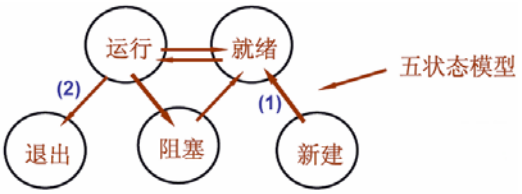
进程三状态转换

- 就绪—运行
- 运行—就绪
- 运行—等待
- 等待—就绪

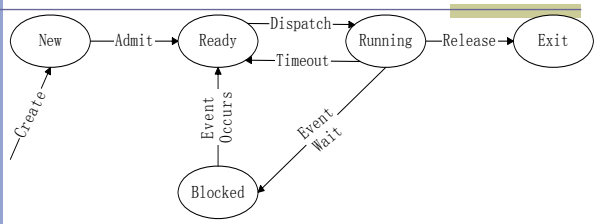


进程五状态转换

- 引入新建态和退出态：初始化和信息收集
  - 新建—就绪：允许进入
  - 运行—退出：执行完



五状态进程转换图



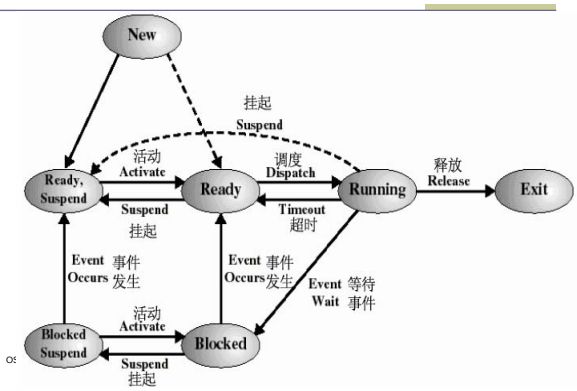
- 创建新进程Create
- 提交（收容）Admit
- 调度运行(Dispatch)
- 释放(Release)
- 超时（Timeout）
- 事件等待（Event Wait）
- 事件出现（Event Occurs）

挂起进程模型

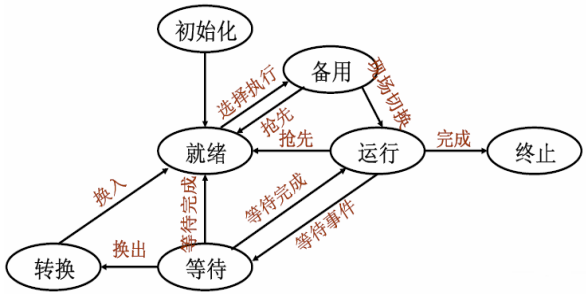
- 挂起（Suspend）：一些低优先级进程可能等待较长时间而被对换至外存，为运行进程提供足够内存。
  - 阻塞挂起（Blocked, suspend）：进程在外存并等待某事件的出现；
  - 就绪挂起（Ready, suspend）：进程在外存，但只要进入内存，即可运行；

挂起（Suspend）：把一个进程从内存转到外存
激活（Activate）：把一个进程从外存转到内存

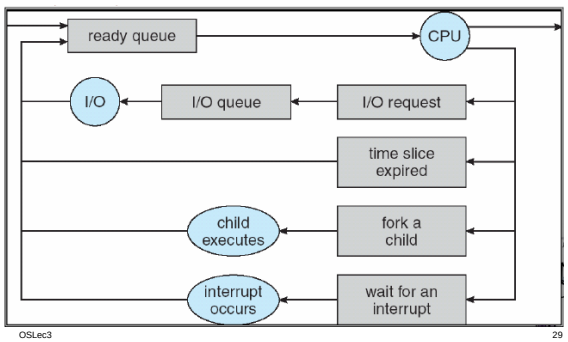
七状态进程转换图



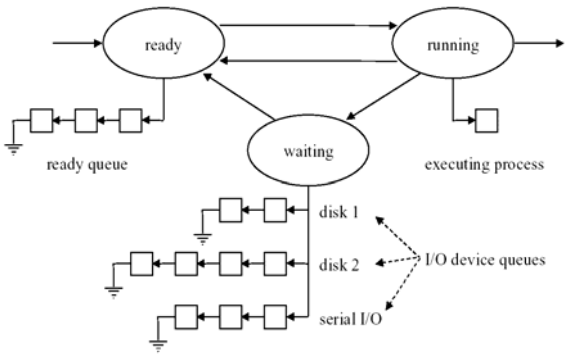
Windows的进程状态转换



状态切换的实现



Linked List



操作系统
第五讲

王宇英
wangyy@nwpu.edu.cn

Review

进程概念的引入

进程的表示和状态转换

Today

进程的控制

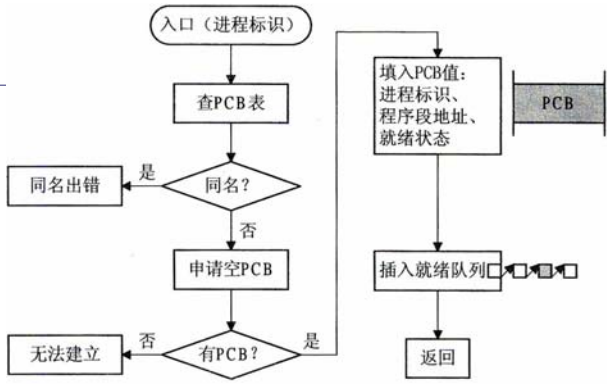
处理机调度的基本概念

3.3 进程的控制

- 创建、撤消进程以及完成进程各状态之间的转换，由具有特定功能的原语完成
- “原语”是由若干条机器指令构成、完成一种特定功能的程序段；这段程序在执行期间不允许被分割，必须一次执行完。
- 进程控制原语：
  - 进程创建原语
  - 进程状态转换原语
  - 进程撤消原语

进程的创建

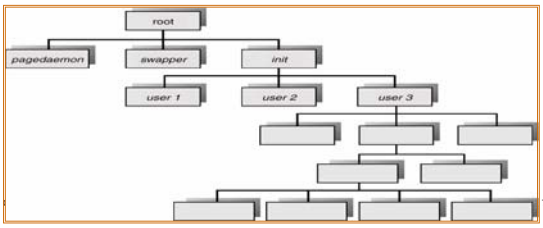
- 何时创建：
  - 用户登录、作业调度、提供服务、应用请求
- 进程创建的基本过程：
  - 首先从空闲的PCB集合中申请一个新的PCB，同时获得该进程的内部标识；
  - 然后向该PCB中填写各种参数；
  - 把该进程的状态设置成就绪状态，并将该PCB插入到就绪队列中。



进程创建原语执行过程

继承家族树

- 资源分配严格，子进程只能继承父进程所拥有的资源，便于管理；
- 系统可根据需要赋予进程不同的控制权，并可以把一个任务分解成若干个进程来完成，具有较好的灵活性；
- 树形结构层次清晰，关系明确。



Linux Example

- fork system call creates new process
- exec system call used after a fork to replace the process' memory space with a new program

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
    }
    else if (pid == 0) { /* child process */
          execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
          /* parent will wait for the child
    to complete */
          wait(NULL);
          printf("Child Complete");
          exit(0);
    }
}
```


进程的状态转换（1）

- 进程挂起原语
  - 检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞。
- 进程激活原语
  - 先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，便将之改为活动就绪；若为静止阻塞便将之改为活动阻塞。

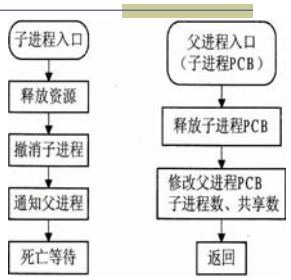
进程的状态转换（2）

- 进程阻塞/等待原语
  - 找到相应进程的PCB；
  - 如果该进程为执行状态，则保护其现场，将其状态改变为等待状态，停止运行，并把该PCB插入到相应的等待队列中去；
- 进程唤醒原语
  - 在等待队列中找到相应进程的PCB，将其从等待队列中移出；
  - 置为就绪状态，然后把该PCB插入就绪队列；
  - 等待调度程序调度。

进程的撤销

- 引起进程终止(Termination of Process)的事件
  - 正常结束
  - 异常结束
  - 外界干预
- 撤消进程的两种策略：
  - 撤消指定进程
  - 撤消该进程及其所有子孙进程

- 进程终止原语
  - 找到相应进程的PCB；
  - 若进程正处于执行状态，则立即停止，设置重新调度标志；
  - 撤消属于该进程的所有"子孙"进程；
  - 释放被撤消进程所有资源；
  - 释放进程的PCB；
  - 若调度标志为真，则进行重新调度



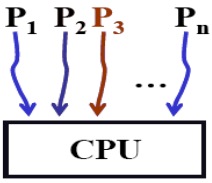
(a)子进程撤销　（b）父进程撤销

Linux中的进程控制原语

- 在Linux系统中，进程控制的原语有：
  - 进程建立fork、
  - 进程监控ps、
  - 进程优先级的确定nice、
  - 进程等待lock、
  - 进程唤醒wakeup、
  - 进程终止kill等。
  - 。。。。。。

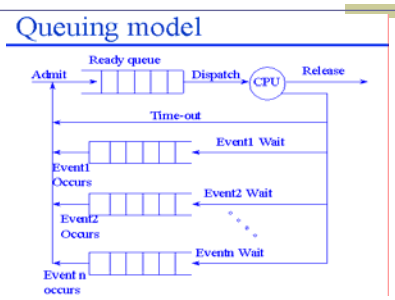
3.4 进程调度 (Scheduling)

- 要解决的问题
  - WHAT：按什么原则分配CPU
    —进程调度算法
  - WHEN：何时分配CPU
    —进程调度的时机
  - HOW：如何分配CPU
    —进程调度方式




```
interrupt 当前进程;
保存state to PCB;
获得next 进程;
装载state from PCB;
jmp新进程;
```

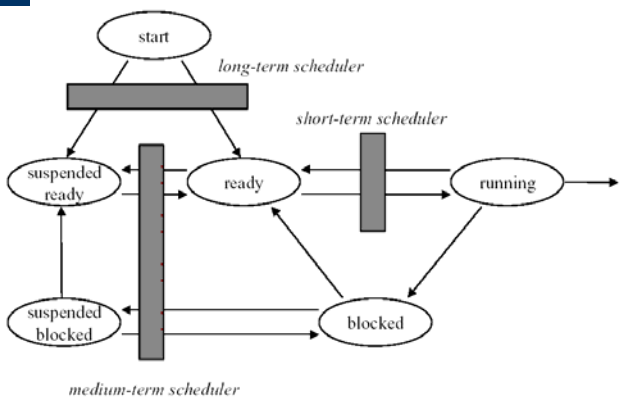

3.4.1 处理机调度的基本概念

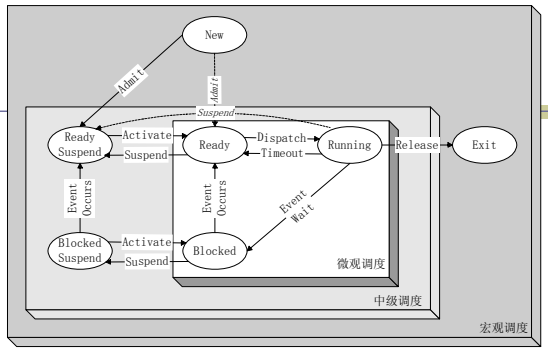


- 目标：高CPU的利用率、大吞吐量、快响应时间

（1）调度的类型

- 高级调度
  - 也称为作业调度或宏观调度，从用户工作流程的角度，一次提交的若干个流程，其中每个程序按照进程调度。时间上通常是分钟、小时或天。New－>Ready suspend, Running －>Exit
- 中级调度
  - 涉及进程在内外存间的交换，从存储器资源管理的角度来看，把进程的部分或全部换出到外存上，将当前进程所需部分换入到内存。 Ready <－>Ready suspend, Blocked <－>Blocked suspend
- 低级调度
  - 也称进程调度、微观调度，从处理机资源分配的角度来看，处理机需要经常选择就绪进程或线程进入运行状态。Ready <－>Running





- 进程调度的任务是控制协调进程对CPU的竞争，即按一定的调度算法从就绪队列中选中一个进程，把CPU的使用权交给被选中的进程
- 微观调度，低级调度，Ready <－>Running

(2) 调度的性能准则

- 面向用户的调度性能准则
  - 周转时间：作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待等。
  - 平均周转时间 $T = \frac{(\sum_{i=1}^{n} T_i)}{n}$
  - 平均带权周转时间 $w = \frac{(\sum_{i=1}^{n} w_i)}{n} = \frac{(\sum_{i=1}^{n} \frac{T_i}{t_{Ri}})}{n}$

n —— 作业流中的作业数

T_i —— 第i个作业周转时间

t_{Ri} —— 作业i的运行时间

带权周转时间：作业周转时间与作用实际运行时间的比

- 响应时间：用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间。
- 优先级：可以使关键任务达到更好的指标。
- 面向系统的调度性能准则
  - 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 处理机利用率：——大中型主机
  - 各种设备的均衡利用：如CPU繁忙的作业和I/O繁忙的作业搭配——大中型主机
- 调度算法本身的调度性能准则
  - 易于实现
  - 执行开销比

调度算法设计目标

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

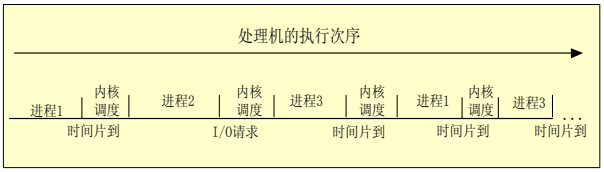
Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

(3) 进程调度的时机

- 现运行进程完成任务正常结束或因出现错误异常结束；
- 时间片到（按时间片运行）；
- 进程提出I/O请求——阻塞，调新进程；
- 执行原语操作而信号量不足被阻塞；
- 具有更高优先级的进程进入就绪队列，要求使用处理机（可剥夺调度）。



(4) 进程调度的方式

- 非剥夺调度(nonpreemptive scheduling)
  - 某一进程被调度运行后，除非由于它自身的原因不能运行，否则一直运行下去。
- 剥夺调度(preemptive scheduling)
  - 当有比正在运行的进程优先级更高的进程就绪时，系统可强行剥夺正在运行进程的CPU，提供给具有更高优先级的进程使用。

What you need to do?

- 复习课本3.4节的内容
- 作业：P91页，6，7

See you next time!

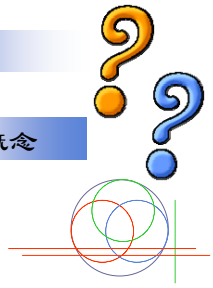
操作系统
第六讲

王宇英
wangyy@nwpu.edu.cn

Review

进程的控制

处理机调度的基本概念



Today

进程调度算法

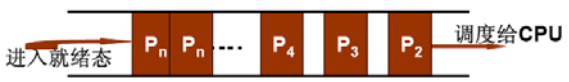
实时调度

3.4.2 进程调度算法

- 先来先服务
- 短作业优先
- 时间片轮转
- 基于优先级的调度算法
- 多级队列算法
- 多级反馈队列调度

先来先服务
FCFS, First Come First Service

- 按照进程进入就绪队列的先后次序分派CPU；
- 特点：
  - 有利于长作业，不利于短作业
  - 有利于CPU繁忙的作业，不利于I/O繁忙的作业
  - 实现简单



- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

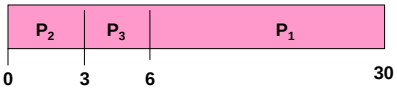
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart（甘特图） for the schedule is:

- 等待时间： P_1 = 0; P_2 = 24; P_3 = 27
- 平均等待时间: (0 + 24 + 27)/3 = 17
- ***Convoy effect***: short process behind long process

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The Gantt chart for the schedule is:



- 等待时间： $P_1 = 6; P_2 = 0, P_3 = 3$
- 平均等待时间： (6 + 0 + 3)/3 = 3

最短作业优先
SJF, Shortest Job First

- 对预计执行时间短的进程优先分派处理机
- 最短作业优先(SJF)
  - 优先运行最短的作业
- 最短剩余时间优先(Shortest Remaining Time First，SRTF)
  - 可抢占的SJF
  - 如果一个具有更短运行时间的作业就绪时，抢占CPU

Example of SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Non-Preemptive---SJF

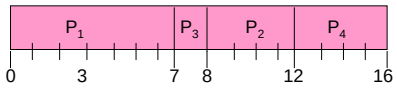
Preemptive----SRTF

Please give the Waiting Time of each process

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

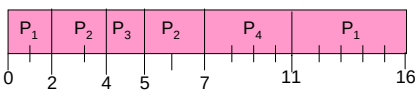


- 平均等待时间 = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SRTF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- 平均等待时间 = (9 + 1 + 0 +2)/4 = 3

SJF的特点

- 优点：
  - 提高系统的吞吐量；
  - SJF对于给定的进程集合，平均周转时间最小
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行；
  - 需要预知作业的未来！

调度算法 \ 作业情况	进程名	A	B	C	D	E	平 均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS (a)	完成时间	Try your best!					
	周转时间						
	带权周转时间						
SJF (b)	完成时间						
	周转时间						
	带权周转时间						

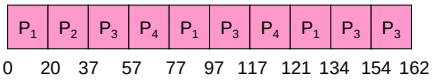
时间片轮转
RR, Round Robin

- 基本思路：
  - 通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；
- 执行过程：
  - 将系统中所有的就绪进程按照FCFS原则，排成一个队列。
  - 每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
  - 在一个时间片结束时，发生时钟中断。调度程序暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
  - 进程可以未使用完一个时间片，就出让CPU（如阻塞）。

Example: RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- 等待时间：

P1=57+24=81，P2=20，P3=37+40+17=94，P4=57+40=97

- 平均等待时间：

时间片长度的确定

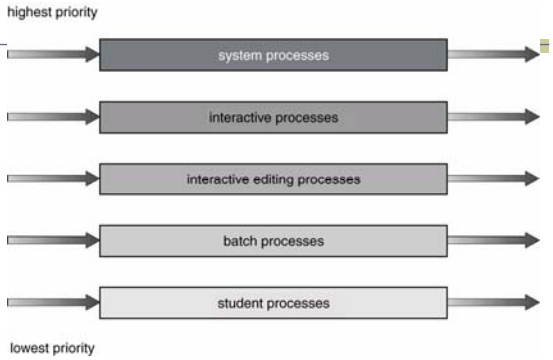
- 时间片长度变化的影响
  - 过长->退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
  - 过短->用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长。
- 对响应时间的要求：
  - T(响应时间)=N(进程数目)*q(时间片)
- 时间片长度的影响因素：
  - 就绪进程的数目：当响应时间一定时，数目越多，时间片越小
  - 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则使响应时间延长。

基于优先级的调度算法
Priority Scheduling

- 基本思想：
  - 系统为每个进程设置一个优先级，把所有的就绪进程按优先级从大到小排序，调度时从就绪队列中选择优先级最高的进程投入运行。
- 剥夺方式：
  - 非剥夺（抢占）的优先级调度法
  - 可剥夺（抢占）的优先级调度法

优先级的类型

- 静态优先级：创建进程时就确定，直到进程终止前都不改变。通常是一个整数。依据：
  - 进程类型（系统进程优先级较高）
  - 对资源的需求（对CPU和内存需求较少的进程优先级较高）
  - 用户要求（紧迫程度和付费多少）
- 动态优先级：在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：
  - 在就绪队列中等待时间延长则优先级提高，使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
  - 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

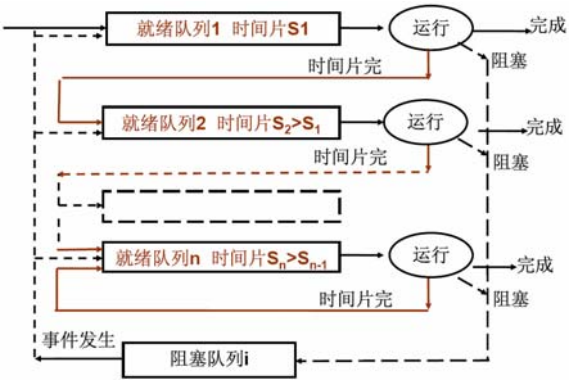


多级队列算法
Multiple-level Queue

- 本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；
- 基本思想：
  - 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
  - 每个作业固定归入一个队列。
  - 各队列不同处理：不同队列可有不同的优先级、时间片长度、调度策略等。

多级反馈队列调度
Multiple-level Queue

- 是时间片轮转算法和优先级算法的综合
- 基本思想：
  - 设置多个就绪队列，分别赋予不同的优先级，每个队列时间片的长度不同，规定优先级越低则时间片越长；
  - 新进程进入内存后，先投入队列1的末尾按FCFS算法调度；若一个时间片未能执行完，则降低优先级投入到队列2的末尾，同样按FCFS算法调度；如此下去，直到降低到最后的队列，按"时间片轮转"算法调度直到完成；
  - 仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。
- 组间可剥夺，组内不可剥夺。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程。



- 特点：
  - 为提高系统吞吐量和缩短平均周转时间而照顾短进程—短作业优先
  - 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程—每次I/O返回时提高优先级，时间片用完时降低优先级
  - 时间片的变化使得运算型进程将获得较长的时间片执行，减少调度次数
  - 不必估计进程的执行时间，动态调节优先级

- 不同的环境需要不同的调度算法
  - 批处理系统
    - FCFS
    - Shortest job first
    - Shortest remaining time first
  - 交互式系统
    - Round robin
    - Priority Scheduling
    - Round Robin with Multiple Feedback （CTSS）
    - Lottery scheduling
  - 实时系统
    - 硬实时：必须满足绝对的截止时间
    - 软实时：不希望错失截止时间，但可以容忍

3.4.3 实时调度

- 实时系统（real-time system）
  - 能够实现在指定或者确定的时间内完成系统功能和对外部或内部、同步或异步时间做出响应的系统
  - 在实时计算中，系统的正确性不仅仅依赖于计算的逻辑结果，而且依赖于结果产生的时间。
- 实时任务（real-time task）
  - 周期性实时任务
  - 非周期性实时任务
  - 硬实时任务
  - 软实时任务

实现实时调度的基本条件

- 1. 提供必要的信息
  - 就绪时间
  - 开始截止时间和完成截止时间
  - 处理时间
  - 资源要求
  - 优先级
- 2. 可调度的实时系统(Schedulable real-time system)
  - 给定m个周期性事件(periodic events)
  - 事件i的周期为P_i，需要的处理时间为 C_i
  - 可调度的标准(schedulable)

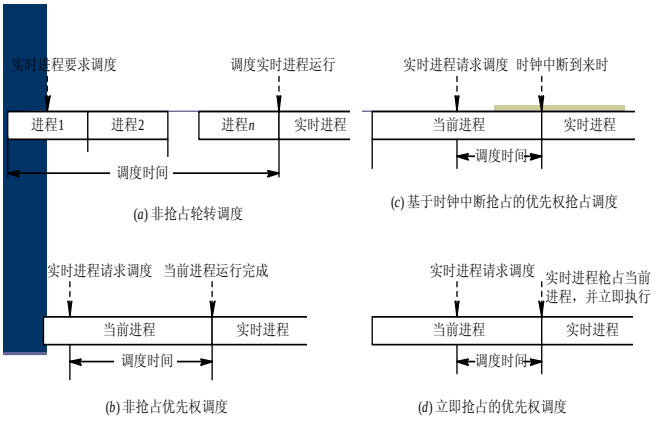
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

- 不可调度的例子：
  - 系统中有6个硬实时任务，周期时间都是 50 ms，每次的处理时间为 10 ms。
- 解决的方法：提高系统的处理能力。途径有二：
  - 采用单处理机系统，但须增强其处理能力，以显著地减少对每一个任务的处理时间；
  - 采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

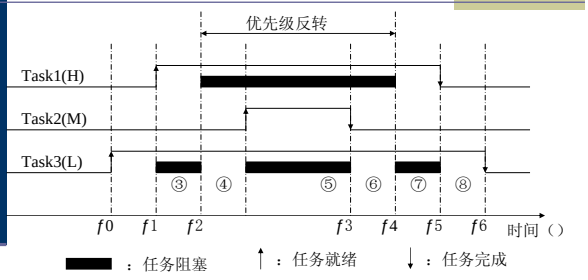
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le N$$

实现实时调度的基本条件

- 3. 采用抢占式调度机制
  - 当一个优先权更高的任务到达时，允许将当前任务暂时挂起，而令高优先权任务立即投入运行，可满足该实时任务对截止时间的要求。
- 4. 具有快速切换机制
  - 对外部中断的快速响应能力。
  - 快速的任务分派能力。

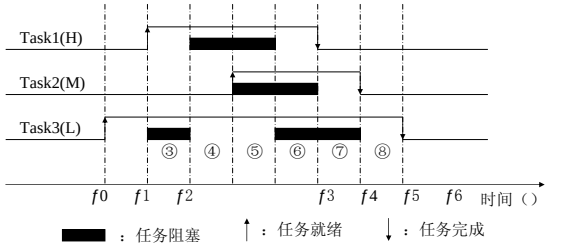


- 优先级反转：高优先级的进程必须等待低优先级的进程完成



优先级反转示意图

- 优先级继承：所有使用到高优先级进程所需资源的进程，继承高优先级直到用完竞争资源，再回到原来的优先级。



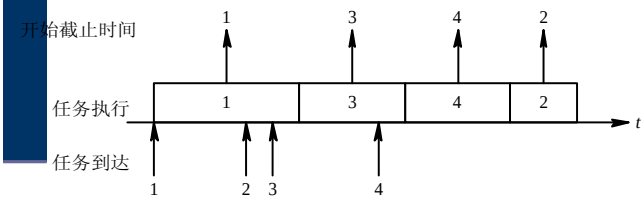
采用优先级继承消除优先级反转

常用实时调度算法

- 静态表调度算法（Static table-driven scheduling）
  - 适用于周期性的实时应用。通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。这种方法的特点是有效但不灵活。
- 静态优先级调度算法（Static priority-driven scheduling）
  - 把通用的优先级调度算法用于实时系统，但优先级的确定是通过静态分析（运行时间、到达频率）完成的。

常用实时调度算法(2)

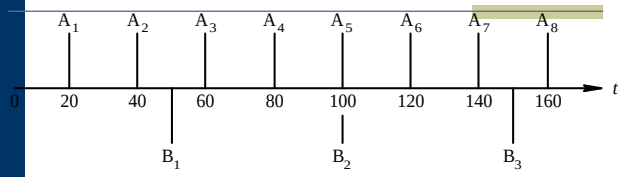
1. 最早截止期优先EDF（Earliest Deadline First）算法



EDF算法用于非抢占调度方式

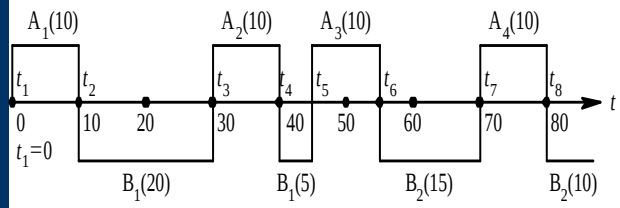
常用的几种实时调度算法(2)

2. 最低松弛度优先，LLF（Least Laxity First）
   - 松弛度：任务必须完成的时刻-此任务需要运行的时间
   - 根据任务紧急（或松弛）的程度，来确定任务的优先级；
   - 系统中有一个按松弛度排序的实时任务就绪队列，松弛度最低的任务排在队列最前面，调度程序总是选择就绪队列中的队首任务执行；
   - 主要用于可抢占调度方式中。



A和B任务每次必须完成的时间

假如在一个实时系统中，有两个周期性实时任务A和B，任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。



利用LLF算法进行调度的情况

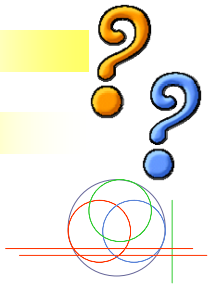
操作系统
第七讲

王宇英
wangyy@nwpu.edu.cn

Review

进程调度算法

实时调度



3.5 线程及其管理

线程的引入

进程和线程的比较

操作系统对线程的实现

线程举例

3.5.1 线程的引入

- 进程：资源分配单位和CPU调度单位。
  - 进程是拥有自己资源的单元体。
  - 进程是被调度分派在处理器上运行的单元体。
- 缺点：时间空间开销大，限制并发度的提高
- 将程序块的执行从进程中分离出来
  - 一个进程中的多个程序块可共享地址空间
  - 程序块执行时需要单独的：
    - 执行状态：寄存器内容、栈、局部内存
    - 指令指针：PC
  - 程序块执行时不需要单独的：
    - 进程资源：诸如打开文件指针、页表等内容
- 多线程：将一程序分为多个并发的活动(程序块)

- 进程= 线程+ 资源集
  - 进程是资源的拥有者
    - 虚拟地址空间、资源文件、I/O设备等资源
  - 线程是程序的一个个执行轨迹——可并发
- 线程：作为CPU调度单位，而进程只作为其他资源的分配单位。
  - 线程只拥有必不可少的资源，如：线程状态、寄存器和堆栈
  - 同样具有就绪、阻塞和执行三种基本状态

线程的概念

- 线程是进程内一个相对独立的、可调度的执行单元
  - 进程中的一个运行实体，是一个CPU调度单位
  - 资源的拥有者还是进程

将原来进程的两个属性分开处理

- 多线程机制
  - 一个进程可以有多个线程，这些线程共享进程资源，驻留在相同的地址空间，共享数据和文件。
  - 一个线程修改了一个数据项，其他线程可以读取和使用此结果数据。一个线程打开并读一个文件时同一进程中的其他线程也可以同时读此文件。
  - 这些线程运行在同一进程的相同的地址空间内。

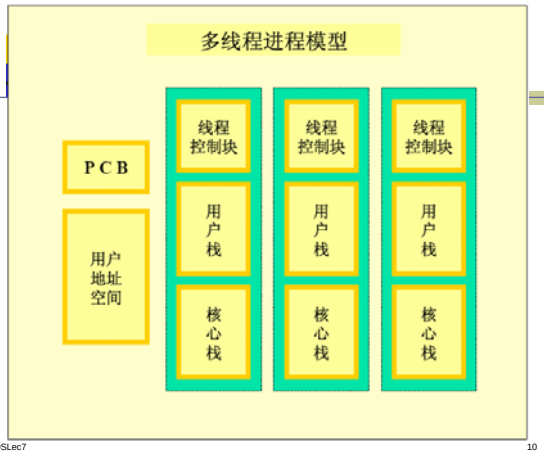
3.5.2 进程和线程的比较

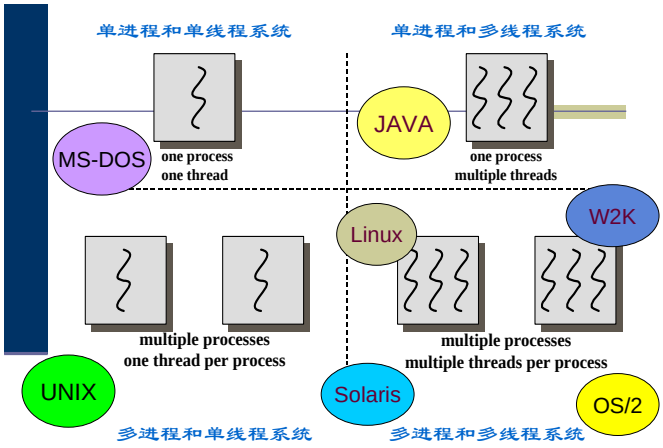
- 调度：线程上下文切换比进程上下文切换快；
  - 线程的创建时间比进程短；终止时间比进程短；
  - 同进程内的线程切换时间比进程短；
- 拥有资源：进程间相互独立，同一进程的各线程间资源共享—— 某进程内的线程在其他进程不可见。
  - 由于同进程内线程间共享内存和文件资源，可直接进行通信不需通过内核；
- 系统开销：线程减小并发执行的时间和空间开销；
- 并发性：在系统中建立更多的线程来提高并发程度。

多线程OS中的进程

- 多线程OS中的进程有以下属性：
  - 作为系统资源分配的单位。
  - 可包括多个线程。
  - 进程不是一个可执行的实体。
- 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信—— 需要同步和互斥手段的辅助以保证数据的一致性

资源所有权（Process）
调度的实体（Thread）





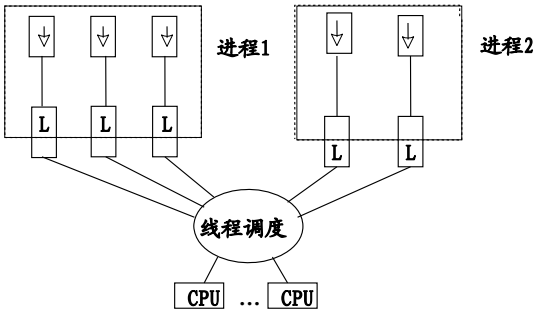
3.5.3 OS对线程的实现方式

- 核心级线程（内核线程，kernel-level thread）
  - 由操作系统内核进行管理。操作系统内核给应用程序提供相应的系统调用和应用程序接口API，使用户程序可以创建、执行、撤消线程。
- 用户级线程（用户线程，User-level thread）
  - 管理过程全部由用户程序完成，操作系统内核心只对进程进行管理。

内核线程KLT

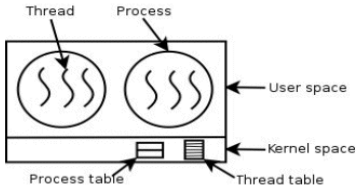
- 依赖于OS核心，由内核的内部需求进行创建和撤销
  - 内核维护进程和线程的上下文信息；
  - 线程切换由内核完成；
  - 一个线程发起系统调用而阻塞，不会影响其他线程的运行；
  - 时间片分配给线程，所以多线程的进程获得更多CPU时间；
  - Windows NT和OS/2支持内核线程。

- 内核级线程



内核线程优点和缺点

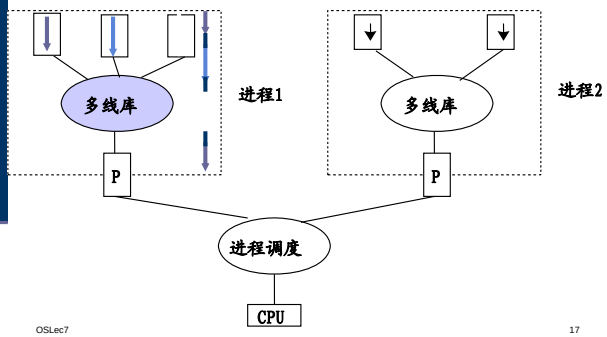
- 优点：
  - 对多处理器，核心可以同时调度同一进程的多个线程
  - 阻塞是在线程一级完成
  - 核心例程是多线程的
- 缺点：
  - 在同一进程内的线程切换调用内核，导致速度下降



用户线程ULT

- 用户级线程仅存在于用户空间中，线程的创建、撤消、线程之间的同步与通信等功能，都无须内核来实现。
  - 用户线程的维护由应用进程完成（通过线程库，用户空间，一组管理线程的过程）；
  - 内核不了解用户线程的存在；
  - 用户线程切换不需要内核特权；
  - 用户线程调度算法可针对应用进行优化；
- 线程库
  - 创建、撤消线程；在线程之间传递消息和数据；调度线程执行；保护和恢复线程上下文

- 用户级线程

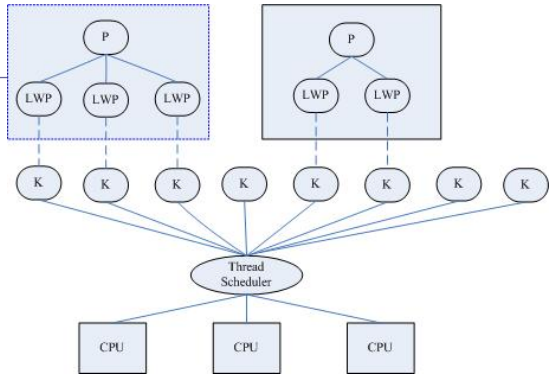


用户线程优点和缺点

- 优点：
  - 线程切换不调用核心
  - 调度是应用程序特定的：可以选择适合的算法
  - ULT可运行在任何操作系统上（只需线程库）
- 缺点：
  - 大多数系统调用是阻塞的，因此核心阻塞进程，故进程中所有线程将被阻塞
  - 核心只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上

用户级线程的实现

- 运行时系统(Runtime System)
  - 用于管理和控制线程的函数(过程)的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数以及实现线程调度的函数等。
- 内核控制线程，又称为轻量级进程LWP(Light Weight Process)
  - 每一个进程都可拥有多个LWP，可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只要将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。



利用轻型进程作为中间系统

线程的状态

- 状态参数： 在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。
  - ① 寄存器状态② 堆栈③ 线程运行状态④ 优先级⑤ 线程专有存储器⑥ 信号屏蔽
- 线程运行状态：各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。
  - ① 执行状态② 就绪状态③ 阻塞状态

线程的创建和终止

- 创建：
  - 应用程序在启动时，通常仅有一个线程在执行——初始化线程
  - 初始化线程可根据需要再去创建若干个线程
  - 线程创建函数执行完后，将返回一个线程标识符供以后使用
- 终止
  - 线程完成了自己的工作后自愿退出
  - 线程在运行中出现错误或由于某种原因而被其它线程强行终止

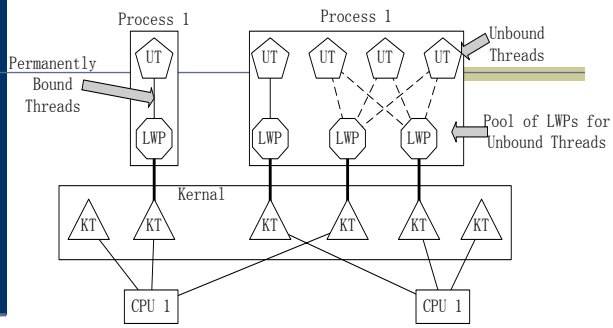
- 线程操作在SPARC上执行时间的比较(ms)

操　作	用户级线程	核心级线程	进　程
创建	52	350	1700
使用信号量同步	66	390	200

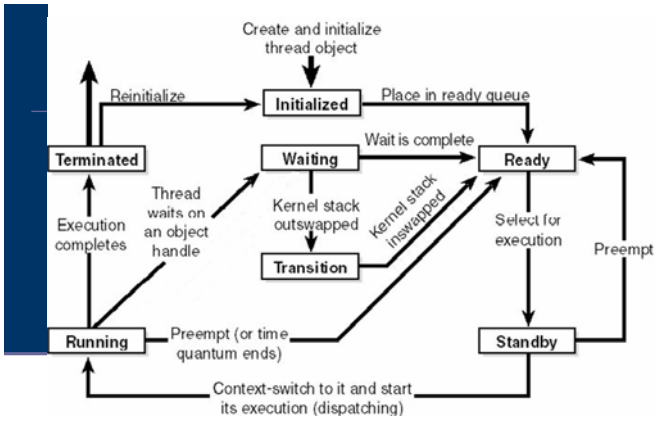
3.5.4 线程举例

- SUN Solaris
  - Solaris支持内核线程(Kernel threads)、轻权进程(Lightweight Processes)和用户线程(User Level Threads)。一个进程可有大量用户线程；大量用户线程复用少量的轻权进程，不同的轻权进程分别对应不同的内核线程。

- 用户级线程在使用系统调用时（如文件读写），需要“捆绑(bound)”在一个LWP上。
  - 永久捆绑：一个LWP固定被一个用户级线程占用，该LWP移到LWP池之外
  - 临时捆绑：从LWP池中临时分配一个未被占用的LWP
- 在使用系统调用时，如果所有LWP已被其他用户级线程所占用（捆绑），则该线程阻塞直到有可用的LWP——例如6个用户级线程，而LWP池中有4个LWP
- 如果LWP执行系统调用时阻塞（如read()调用），则当前捆绑在LWP上的用户级线程也阻塞。



用户线程、轻权进程和核心线程的关系



Windows 2000线程状态

作业、程序、进程和线程的比较

- 作业
  - 用户向计算机提交任务的任务实体
- 程序
  - 一组有序的指令集合
- 进程
  - 系统分配资源的基本单位
- 线程
  - 处理机调度的基本单位

What you need to do?

- 复习课本3.5节的内容
- 作业：在使用线程的系统中，若使用用户级线程，是每个线程一个堆栈还是每个进程一个堆栈？如果是使用内核级线程情况又如何？请给出解释。

See you next time!

操作系统
第八讲

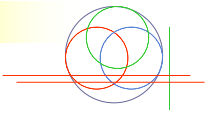
王宇英
wangyy@nwpu.edu.cn

Review

- 线程的引入
- 进程和线程的比较
- 操作系统对线程的实现
- 线程举例

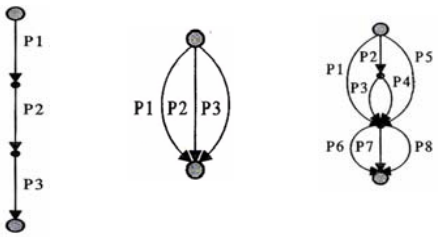
3.6 进程间通讯

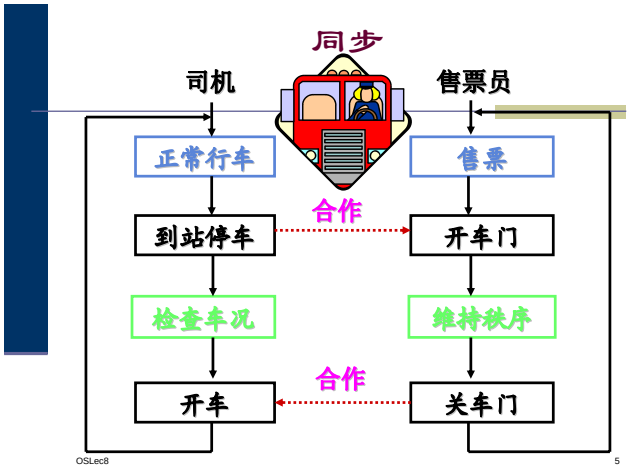
- 进程同步和互斥
- 信号量和PV原语操作
- 经典进程同步问题
- 进程通信



3.6.1 进程同步和互斥

- 进程间同步：指进程之间的一种协调配合关系，它表现在对进程执行顺序的规定上。





例：计算进程和打印进程的同步关系.

- 设消息名bufempty表示buf空，初始化 bufempty =true，设消息名buffull表示buf满，Buffull=false.

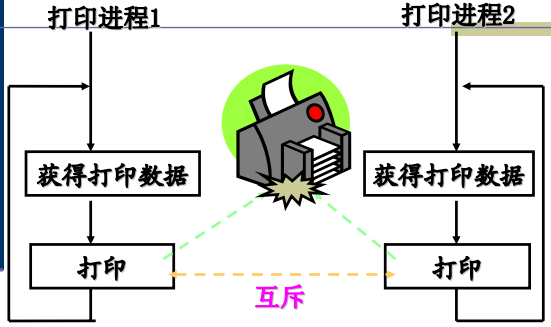
Pc:
```
while(true){
    计算
    wait(bufempty)
    buf←计算结果
    bufempty ← false
    signal(buffull)}
```

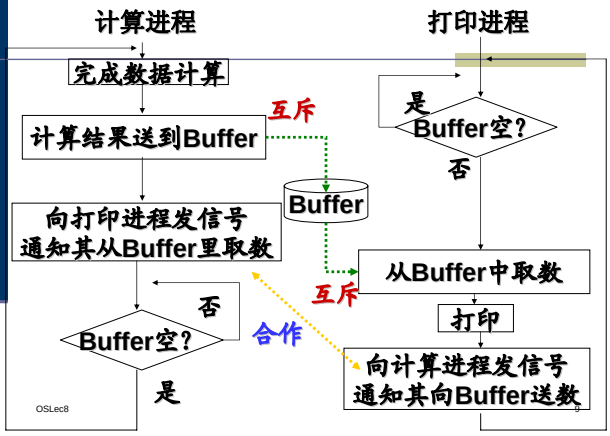
Pp:
```
while(true){
    wait(buffull)
    打印Buf中的数据
    清除Buf中的数据
    buffull ←false
    signal(bufempty)}
```

进程间的互斥

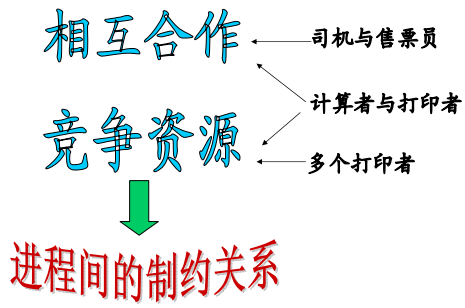
- 进程互斥：两个或两个以上的进程由于不能同时使用同一资源，只能一个进程使用完了另一个进程才能使用的现象。
- 互斥关系也是一种协调关系，从广义上讲它也属于同步关系的范畴。

互斥



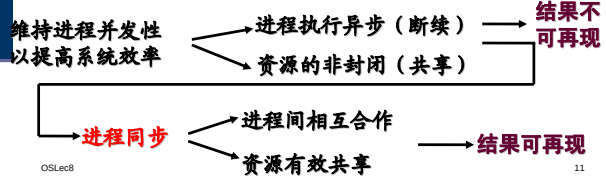


- 进程同步时面临的两种主要关系



进程间的制约关系

- 直接制约关系：进行协作——等待来自其他进程的信息，“同步”
- 间接制约关系：进行竞争——独占分配到的部分或全部共享资源，“互斥”
- 进程间这种相互依赖又相互制约，相互合作又相互竞争的关系表现为同步和互斥两个方面。
- 同步和互斥机制的主要任务：



临界资源与临界区

- 临界资源(critical resource)：一次仅允许一个进程访问的资源。
- 临界区(critical section)：临界段，在每个程序中，访问临界资源的那段程序。
- 对临界段的设计有如下原则：
  - 每次至多只允许一个进程处于临界段中。
  - 对于请求进入临界段的多个进程，在有限时间内只让一个进入。
  - 进程只应在临界段中停留有限时间。

进程A和进程B在各自的执行过程中，都需要使用变量M作为其中间变量，它们的程序如下：

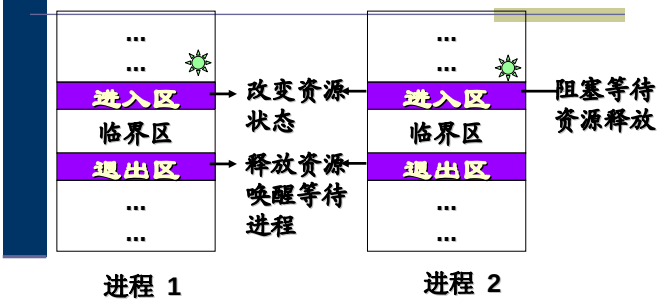
进程A:	进程B:
X：=1;	K：=3;
Y：=2;	L：=4;
M：=X;	M：=K;
X：=Y;	K：= L;
Y：=M;	L：=M;
PRINT （"A: ", X,Y）;	PRINT （"B: ", K,L）;


```
进程A:   X：=1;
进程B:   K：=3;
进程A:   Y：=2;
进程B:   L：=4;
进程A:   M：=X;
进程B:   M：=K;
进程A:   X：=Y;
进程B:   K：=L;
进程A:   Y：=M;
进程B:   L：=M;
进程A:   PRINT （"A: ", X,Y）;
进程B:   PRINT （"B: ", K,L）;
```

临界区的访问过程

- 临界区(critical section)：进程中访问临界资源的一段代码。
- 进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应"正在访问临界区"标志
- 退出区(exit section)：用于将"正在访问临界区"标志清除。
- 剩余区(remainder section)：代码中的其余部分。

Entry section
Critical section
Exit section
Remainder section



同步机制应遵循的准则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

解决进程互斥的方法

- (1) 软件方法
- (2) 硬件方法
- (3) 利用操作系统原语

（1）软件方法解决互斥

- 特点
  - 无需硬件、OS和程序设计语言的支持
  - 处理开销大，容易出错
- 学习的意义
  - 更好地理解并发处理的复杂性

Dekker算法

- 尝试1：单标志
- 设整型变量turn，用于指示被允许进入临界区的进程的编号，即若turn=0，表示进程Pi可进入。turn =1表示进程Pj可进入。

进程Pi：

```
Repeat
  While turn<>i do no_op;

  Critical section
  turn:=j;

  Other code
Until false;
```


算法1的问题

该算法可确保每次只允许一个进程进入临界区。但它强制两个进程轮流进入。如当Pi退出时将turn置为1，以便 Pj能进入，但Pj暂不需要进入，而这时Pi又需要进入时，它无法进入。这不能保证准则1。

- 尝试2：双标志、先检查

设var flag:array[0..1] of boolean，若flag[i]=true，表示进程Pi正在临界区内。flag[i]=false表示进程Pi不在临界区内。若flag[j]=true，表示进程Pj正在临界区内。flag[j]=false表示进程Pj不在临界区内。

Pi进程：

```
Repeat
  While flag[ j ] do no_op;

  flag[i]:=true;
  Critical section
  flag[i]:=false;

  Other code
Until false;
```


算法2的问题

该算法可确保准则1。但又出现新问题。当pi和pj都未进入时，它们各自的访问标志都为false。如果pi和pj几乎同时要求进入，它们都发现对方的标志为false，于是都进入了。这不能保证准则2。

```
while (flag[j]);          <a>
flag[i] = TRUE;           <b>
  critical section
flag[i] = FALSE;
  remainder section
```


- 尝试3：双标志、后检查
- 在算法3 中，设var Flag:array[0..1] of boolean，若flag[i]=true，表示进程Pi希望进入临界区内。若flag[j]=true，表示进程Pj希望进入临界区。

Pi进程：

```
Repeat
  flag[i]:=true;
  While flag[j] do no_op;

  Critical section
  flag[I]:=false;

  Other code
Until false;
```

算法3的问题

该算法可确保准则2。但又出现新问题。它可能造成谁也不能进入。如，当pi和pj几乎同时都要进入，分别把自己的标志置为true，都立即检查对方的标志，发现对方为true. 最终谁也不能进入。 这不能保证准则1。

```
flag[i] = TRUE;          <b>
while (flag[j]);         <a>
  critical section
flag[i] = FALSE;
  remainder section
```


算法4（ Peterson算法）

- 组合算法1、3，为每一进程设标志位flag[i]，当flag[i]=true时，表示进程pi要求进入，或正在临界区中执行。此外再设一个变量turn，用于指示允许进入的进程编号。
- 先修改、后检查、后修改者等待：
- 进程为了进入，先置flag[i]=true，并置turn为j，表示应轮到pj进入。接着再判断flag[j] and turn=j的条件是否满足。若不满足则可进入。或者说，当flag[j]=false或者turn=i时，进程可以进入。前者表示pj未要求进入，后者表示仅允许pi进入。

算法4

```
Repeat
  flag[i]:=true;
  turn:=j;
  While (flag[ j] and turn=j) do
  no_op;

  Critical section
  flag[i]:=false;

  Other code
Until false
```

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
  critical section
flag[i] = FALSE;
  remainder section
```


软件解法的缺点

1. 忙等待
2. 实现过于复杂，需要较高的编程技巧

(2) 硬件方法解决进程互斥

- 1. 中断禁用（关中断，Interrupt Disabling）
  - 如果进程访问临界资源时(执行临界区代码)不被中断，就可以保证互斥访问
  - 途径：使用关中断原语、开中断原语
  - 过程：
    关中断原语；
    临界区
    开中断原语
    其余部分
  - 存在问题
    - 代价高：限制了处理器交替执行各进程的能力
    - 不能用于多处理器结构：关中断不能保证互斥

硬件方法解决进程互斥

- 2.专门的机器指令
  - 专门的机器指令，用于保证两个动作的原子性，如在一个指令周期中实现测试和修改
    - TestandSet指令
    - Swap指令

Test-and-Set指令实现互斥

1、Test-and-Set指令

```
Function TS(var lock:boolean):boolean;
Begin
  TS:=lock;
  Lock:=true;
End;
```

其中，有lock有两种状态：

当lock=false时，表示该资源空闲；

当lock=true 时，表示该资源正在被使用。

2、利用TS指令实现进程互斥

为每个临界资源设置一个全局布尔变量lock，并赋初值false，表示资源空闲。在进入区利用TS进行检查：有进程在临界区时，重复检查；直到其它进程退出时，检查通过；

```
repeat
  while TS(lock) do skip;
  critical section
  lock:=false;
  Other code
Until false;
```

```
while TS(&lock);
  critical section
lock = FALSE;
  remainder section
```


swap指令实现进程互斥

1、swap指令，又称交换指令，X86中称为XCHG指令。

```
Procedure swap(var a,b:boolean);
Var temp:boolean;
Begin
  Temp:=a;    A:=b;    B:=temp;
End;
```


2、利用swap实现进程互斥

为每一临界资源设置一个全局布尔变量lock，其初值为false，在每个进程中有局部布尔变量key。

```
Repeat
  key:=true;
  Repeat    Swap(lock,key);    Until key=false;
  Critical section
  lock:=false;
  Other code
Until false;
```


- 硬件方法的优点
  - 适用于任意数目的进程
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 等待要耗费CPU时间，不能实现"让权等待"
  - 可能"饥饿"：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
  - 可能死锁

(3) 用原语实现进程互斥

- 锁即操作系统中的一个标志位，0表示资源可用，1表示资源已被占用。用户程序不能对锁直接操作，必须通过操作系统提供的上锁和开锁原语来操作。
- 通常锁用w表示，上锁开锁原语分别用lock(w)、unlock(w)来表示。

上锁和开锁原语

- 上锁原语lock(w)可描述为：

L：if(w==1) goto L
　　else w=1；

- 开锁原语unlock(w)可描述为：

w=0；

用原语实现进程互斥

```
pa ( )                    pb ( )
{                         {
    ⋮                         ⋮
    lock (w);                 lock (w);
    csa;                      csb;
    unlock (w);               unlock (w);
    ⋮                         ⋮
}                         }
```


进程的交互关系：可以按照相互感知的程度来分类

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知(完全不了解其它进程的存在)	竞争	一个进程的操作对其他进程的结果无影响	互斥，死锁（可释放的资源），饥饿
间接感知(双方都与第三方交互，如共享资源)	通过共享进行协作	一个进程的结果依赖于从其他进程获得的信息	互斥，死锁（可释放的资源），饥饿，数据一致性
直接感知(双方直接交互，如通信)	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息	死锁，饥饿

互斥，指多个进程不能同时使用同一个资源；
死锁，指多个进程互不相让，都得不到足够的资源；
饥饿，指一个进程一直得不到资源（其他进程可能轮流占用资源）

What you need to do?

- 复习、预习课本3.6节的内容
- 作业：P91页10,11题

See you next time!

操作系统
第九讲

王宇英
wangyy@nwpu.edu.cn

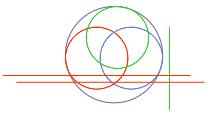
Review

- 进程同步和互斥
- 临界资源及其访问过程
- 同步机制应遵循的准则
- 解决进程互斥的方法

本次内容

信号量和PV原语操作

经典进程同步问题



3.6.2 信号量(semaphore)和P、V原语

- 信号量(Semaphores)是一个generalized lock，1965年由荷兰学者Dijkstra提出，是一种卓有成效的解决进程同步与互斥的机制。
- 是UNIX主要使用的同步原语
- 信号量是一个数据结构

由整型变量v、指针变量q构成

```
信号量定义：
struct semaphore
{
        int value;
        pointer_PCB queue;
}
信号量声明：
 semaphore s;
```


信号量的说明

- 信号量的物理意义：
  - S>0表示有S个资源可用
  - S=0表示无资源可用
  - S<0则| S |表示S等待队列中的进程个数
  - 进程等待队列 s.queue是阻塞在该信号量的各个进程
- 信号量的使用：
  - 初始化指定一个非负整数，表示空闲资源总数
  - 若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数
  - 信号量的值只能被P、V原语操作进行改变

P、V操作

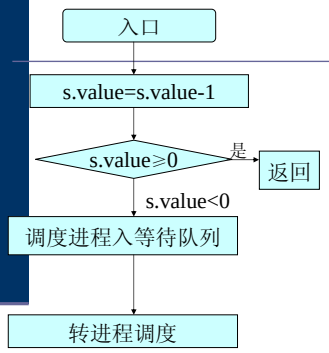
- P是荷兰语test(proberen)
- P(s):表示申请一个资源，等待信号量变为正，如果为正就减1
  - 对“sem”执行减
  - If sem< 0, 则阻塞进程，调用wait()
- V是荷兰语increment(verhogen)
- V(s):表示释放一个资源，将信号量加1，并唤醒等待的线程
  - 对“sem”执行加1操作
  - If 其上有阻塞进程，则调用signal()

P原语wait(s)

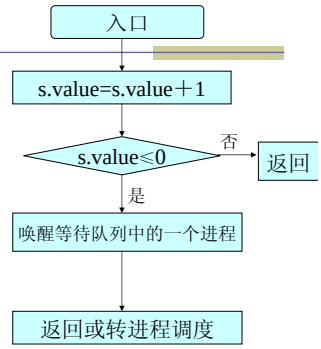
```
P(s)
{
 s.value = s.value -1 ;   //表示申请一个资源;
 if (s.value < 0)         //表示没有空闲资源;

 {
    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;
 }
}
```


V原语signal(s)

```
V(s)
{
 s.value = s.value +1;       //表示释放一个资源;
 if (s.value < = 0)          //表示有进程处于阻塞状态;
 {
   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
   }
}
```

P原语操作功能流程图



V原语操作功能流程图

对信号量的操作

- 操作系统对信号量只能通过初始化和两个标准的原语来访问，对信号量的操作只有三种原子操作
  - 初始化：通常将信号量的值初始化为非负整数
  - P操作(wait操作)
    - 使信号量的值减1(申请一个单位的资源 (s.value--))
    - 如果使信号量的值变成负数，则执行P操作的进程被阻塞(当s.value < 0时，资源已分配完毕，进程自己阻塞在S的队列上----让权等待)
  - V操作(signal操作)
    - 使信号量的值加1(释放一个单位资源 (s.value++))
    - 如果信号量的值不是正数，则使一个因执行P操作被阻塞的进程解除阻塞(若s.value <= 0，则唤醒一个等待进程)

利用信号量实现互斥

```
P(mutex);
  critical section
V(mutex);
  remainder section
```

- 在互斥问题中，对信号量mutex必须设置一次初值
- 在每个进程中将临界区代码置于P和V原语之间,P、V原语操作应该分别紧靠临界区的头部和尾部，从而提高进程的并发度
- Mutex的取值为：1,0,-1,-2,…,-(n-1)
- P、V操作必须成对出现，而且它们同处于同一个进程中

例1：设进程A和进程B，它们都要求进入临界段CS，下面的设计就可以满足进程的互斥要求：

```
信号量  S=1;          * 定义信号量并确定初值*/
进程A:
    ⋮
   P (S) ;
   CS1;
   V (S) ;
    ⋮
进程B:
    ⋮
   P (S) ;
   CS2;
   V (S) ;
    ⋮
```

例2: 设有M个进程都要以独享的方式用到某一种资源，且一次只申请一个资源，该种资源的数目为N。

信号量　S＝N;

进程 Pi:

⋮

P（S）;

CSi;

V（S）;

⋮

利用信号量实现同步

例3: 设有进程A和B，要求进程A的输出结果成为进程B的输入信号，也就是说进程B必须在进程A执行完毕后才能执行。

信号量　S＝0;

进程A:

⋮

V（S）;

⋮

进程B:

⋮

P（S）;

⋮

例4: 设有进程A、B、C，要求进程A、C先于进程B运行（见图）。实现方法如下:

信号量　S1＝0, S2＝0;

进程A:　进程C:

⋮　⋮

V（S1）;　V（S2）;

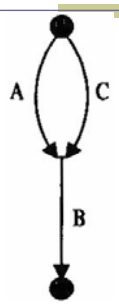
⋮　⋮

进程B:　⋮

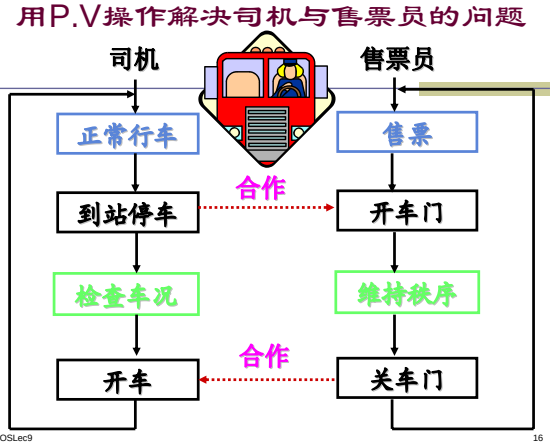
P（S1）;

P（S2）;

⋮



用P.V操作解决司机与售票员的问题



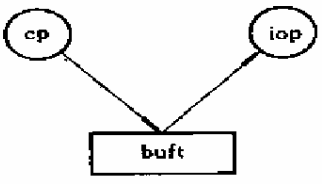
解

- 设有两个信号量S1，S2，初值均为0。

```
司机进程:                售票员进程:
while(1)                 while(1)
{                        {
  P(S1)                      关门
  启动车辆                   V(S1)
  正常驾驶                   售票
  到站停车                   P(S2)
  V(S2)                      开门
}...                     }...
```


共享缓冲区的进程的同步

- 设某计算进程C P和打印进程ＩＯＰ共用一个单缓冲区，ＣＰ进程负责不断地计算数据并送入缓冲区Ｔ中，ＩＯＰ进程负责不断地从缓冲区Ｔ中取出数据去打印。



解

- 为此设有两个信号量Sa=0，Sb=1，Sa表示缓冲区中有无数据，Sb表示缓冲区中有无空位置。

```
cp (  )                          iop (  )
{ while (计算未完成)             { while (打印工作未完成)
   {                                {
    得到一个计算结果;                  p (Sa);
    p (Sb);                          从缓冲区中取一数;
    将数送到缓冲区中;                  v (Sb);
    v (Sa);                          从打印机上输出;
   }                                 }
}                                }
```

【思考题】

- 桌上有一空盘，最多允许存放一只水果。爸爸可向盘中放一个苹果或放一个桔子，儿子专等吃盘中的桔子，女儿专等吃苹果。
  试用P、V操作实现爸爸、儿子、女儿三个并发进程的同步。

提示：设置一个信号量表示可否向盘中放水果，一个信号量表示可否取桔子，一个信号量表示可否取苹果。

解

设置三个信号量S,So,Sa，初值分别为1，0，0。分别表示可否向盘中放水果，可否取桔子，可否取苹果。

```
Father()              Son()               Daughter()
{ while(1)            { while(1)          { while(1)
  { p(S);               { p(So)             { p(Sa)
    将水果放入盘中;       取桔子              取苹果
    if(是桔子)v(So);      v(S);               v(S);
    else v(Sa);           吃桔子;             吃苹果;
  }                     }                   }
 }                    }                   }
```


P、V操作的优缺点

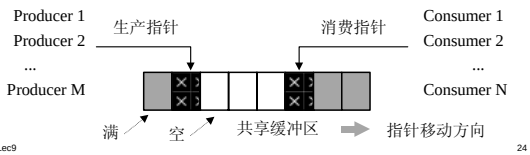
- 优点：
  - 简单，而且表达能力强（用P、V操作可解决任何同步互斥问题）
- 缺点：
  - 不够安全，P、V操作使用不当会出现死锁；
  - 遇到复杂同步互斥问题时实现复杂

3.6.3 经典进程同步问题

- 生产者－消费者问题
  the Producer-Consumer Problem
- 读者/写者问题
  Readers and Writers Problem
- 哲学家进餐问题
  the Dining Philosophers Problem

3.6.3.1 生产者/消费者问题

- 问题描述：若干进程通过有限的共享缓冲区交换数据。其中，
  - “生产者”进程不断写入；
  - “消费者”进程不断读出；
  - 共享缓冲区共有N个；
  - 任何时刻只能有一个进程可对共享缓冲区进行操作。



问题分析

- 为解决生产者消费者问题，应该设两个同步信号量，一个说明空缓冲区的数目，用S1表示，初值为有界缓冲区的大小N，另一个说明已用缓冲区的数目，用S2表示，初值为 0 。
- 由于在此问题中有M个生产者和N个消费者，它们在执行生产活动和消费活动中要对有界缓冲区进行操作。由于有界缓冲区是一个临界资源，必须互斥使用，所以，另外还需要设置一个互斥信号量mutex，其初值为1。

Producer	Consumer
P(empty); P(mutex); //进入区	P(full); P(mutex); //进入区
one unit --> buffer;	one unit <-- buffer;
V(mutex); V(full); //退出区	V(mutex); V(empty); //退出区

- 问题的解：设信号量
  - full是"满"数目，初值为0，
  - empty是"空"数目，初值为N。实际上，full和empty是同一个含义：full + empty == N
  - mutex用于访问缓冲区时的互斥，初值是1

```
P:
  i = 0;
  while (1)
  {
     生产产品;
     P(empty);
     P(mutex);
      往Buffer [i]放产品;
      i = (i+1) % n;
     V(mutex);
     V(full);
  };
```

```
Q:
  j = 0;
  while (1)
  {
    P(full);
    P(mutex);
      从Buffer[j]取产品;
      j = (j+1) % n;
    V(mutex);
    V(empty);
    消费产品;
  };
```

分析P操作的顺序很重要

- 假定执行顺序如下

Producer：	Consumer：	
P(empty);	P(mutex);	
P(mutex);	P(full);	//进入区
one unit-->buf;	one unit<--buf;	
V(mutex);	V(mutex);	
V(full);	V(empty);	//退出区

分析：当full=0, mutex = 1时，执行顺序：

Consumer.P(mutex) ; Consumer.P(full);

// C阻塞，等待Producer 发出的full信号

Producer.P(empty) ; Producer.P(mutex) ;

// P 阻塞，等待Consumer发出的empty信号

→ 发生死锁

信号量及P、V操作讨论

- P.V操作必须成对出现，有一个P操作就一定有一个V操作
  - 当为互斥操作时，它们同处于同一进程
  - 当为同步操作时，则不在同一进程中出现
- 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
- 两个V操作的顺序无关紧要
- 二元信号量可用于互斥，一般信号量可用于同步

【思考题】

- 有一个仓库，可以存放A和B两种产品，要求：
  - （1）每次只能存入一种产品（A或B）
  - （2）－N＜A产品数量－B产品数量＜M。
- 其中，N和M是正整数。试用P、V操作描述产品A与B的入库过程。

- 提示：设两个同步信号量Sa、Sb
  - Sa表示允许A产品比B产品多入库的数量
  - Sb表示允许B产品比A产品多入库的数量

Answer：

设两个信号量Sa、Sb，初值分别为M-1，N-1

Sa表示允许A产品比B产品多入库的数量

Sb表示允许B产品比A产品多入库的数量

设互斥信号量mutex，初值为1。

```
A产品入库进程：
  i = 0;
  while (1)
  {
     生产产品;
     P(Sa);
     P(mutex);
       A产品入库
     V(mutex);
     V(Sb);
  };
```

```
B产品入库进程：
  j = 0;
  while (1)
  {
    P(Sb);
    P(mutex);
      B产品入库
    V(mutex);
    V(Sa);
    消费产品;
  };
```

3.6.3.2 读者/写者问题

- 问题描述：对共享资源的读写操作，有两组并发进程——读者和写者,共享一组数据区
- 要求：
  - 允许多个读者同时执行读操作
  - 不允许读者、写者同时操作
  - 不允许多个写者同时操作

“读－写”互斥，“写－写”互斥，“读－读”允许

问题分析：第一类，读者优先

如果读者来：

1）无读者、写者，新读者可以读
2）有写者等，但有其它读者正在读，新读者也可以读
3）有写者写，新读者等

如果写者来：

1）无读者，新写者可以写
2）有读者，新写者等待
3）有其它写者，新写者等待

第一类读者写者问题的解法

- 设有两个信号量w=1，mutex=1
- 另设一个全局变量readcount =0
- w用于读者和写者、写者和写者之间的互斥
- readcount表示正在读的读者数目
- mutex用于对readcount 这个临界资源的互斥访问

读者：

```
while (1)
{
   P(mutex);
     readcount ++;
     if (readcount==1) P (w);
   V(mutex);
     读
   P(mutex);
     readcount --;
     if (readcount==0) V(w);
   V(mutex);
};
```

写者：

```
while (1)
{
     P(w);
      写
     V(w);
};
```


【思考题】写优先

- 修改以上读者写者问题的算法，使之对写者优先，即一旦有写者到达，后续的读者必须等待，无论是否有读者在读。
- 提示：增加一个信号量，用于在写者到达后封锁后续的读者

解：增加一个信号量S，初值为1

读者：

```
while (1)
{  P(s);
   P(mutex);
     readcount ++;
     if (readcount==1) P (w);
   V(mutex);
   V(s);
     读
   P(mutex);
     readcount --;
     if (readcount==0) V(w);
   V(mutex);
};
```

写者：

```
while (1)
{
     P(s);
     P(w);
      写
     V(w);
     V(s);
};
```

3.6.3.3 哲学家就餐问题

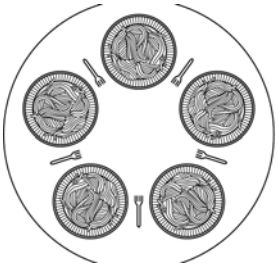
- 问题描述：（由Dijkstra首先提出并解决）
  - 5个哲学家围绕一张圆桌而坐，
  - 桌子上放着5支筷子，每两个哲学家之间放一支；
  - 哲学家的动作包括思考和进餐，
  - 进餐时需要同时拿起他左边和右边的两支筷子，
  - 思考时则同时将两支筷子放回原处。
- 问题：如何保证哲学家们的动作有序进行？如：
  - 不出现相邻者同时要求进餐；
  - 不出现有人永远拿不到筷子；

解

设fork[5]为5 个信号量，初值为均1

Philosopheri：

```
while (1)
{
   思考；
   P(fork[i]);
   P(fork[(i+1) % 5]);
   进食；
   V(fork[i]);
   V(fork[(i+1) % 5]);
}
```



分析

以上解法会出现死锁。为防止死锁发生：

- 最多允许4个哲学家同时坐在桌子周围
- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子 (√)
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之


```
#define N 5                         /* number of philosophers */

void philosopher(int i)             /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                   /* philosopher is thinking */
        take_fork(i);               /* take left fork */
        take_fork((i+1) % N);       /* take right fork; % is modulo operator */
        eat( );                     /* yum-yum, spaghetti */
        put_fork(i);                /* put left fork back on the table */
        put_fork((i+1) % N);        /* put right fork back on the table */
    }
}
```

A nonsolution to the dining philosophers problem


```
#define N           5               /* number of philosophers */
#define LEFT        (i+N-1)%N       /* number of i's left neighbor */
#define RIGHT       (i+1)%N         /* number of i's right neighbor */
#define THINKING    0               /* philosopher is thinking */
#define HUNGRY      1               /* philosopher is trying to get forks */
#define EATING      2               /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1)


```
void take_forks(int i)              /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Solution to dining philosophers problem (part 2)

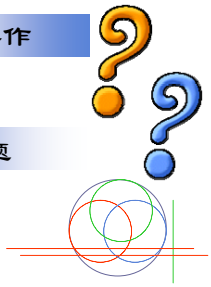
操作系统
第十讲

王宇英
wangyy@nwpu.edu.cn

Review

信号量和PV原语操作

经典进程同步问题



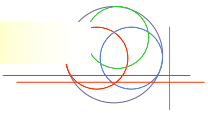
3.6.3 进程间通信
IPC, INTER-PROCESS COMMUNICATION

进程间通信类型

消息缓冲通信的实现

信箱通信的实现

管道通信的实现



基本概念

- 进程通信是指进程之间可直接以较高的效率传递较多数据的信息交换方式。
- 低级通信：只能传递状态和控制信息等，进行简单的信号交换。
  - 优点：速度快
  - 缺点：
    - 传送信息量小：效率低
    - 编程复杂：用户直接实现通信的细节，易出错。
- P、V操作实现的是进程之间的低级通讯
- 高级通信：高效、大量的数据传递。

高级通信的特征

- 通信链路(communication link)：
  - 点对点/多点/广播
  - 单向/双向
  - 有容量（链路带缓冲区）/无容量（发送方和接收方需自备缓冲区）
- 数据格式：
  - 字节流(byte stream)：各次发送之间的分界，在接收时不被保留，没有格式；
  - 报文(datagram/message)：各次发送之间的分界，在接收时被保留，通常有格式，定长/不定长报文，可靠/不可靠报文。
- 收发操作的同步方式
  - 发送阻塞和不阻塞
  - 接收阻塞和不阻塞
  - 由事件驱动收发：在允许发送或有数据可读时，才做发送和接收操作

3.6.3.1 进程通信类型

- 共享存储器系统
  - 通过数据、数据区的共享，写入与读出达到通信的目的
- 消息传递系统
  - 直接通信方式：消息缓冲
    - 采用进程的消息缓冲队列
    - 消息发送者将消息直接放在接收者的消息缓冲队列
  - 间接通信方式：邮箱通信
    - 利用中间者——信箱、邮局来传递信件。
    - 发送进程将消息发送到信箱中，接收进程从信箱中取出消息
- 管道通信 (共享文件方式)
  - 用以连接读、写进程的共享文件

1. 共享存储器系统 Shared-Memory System

- 基于共享数据结构的通信方式
  - 诸进程公用某些数据结构，进程通过它们交换信息。如生产者-消费者问题中的有界缓冲区。
- 基于共享存储区的通信方式
  - 高级通信，在存储器中划出一块共享存储区，进程在通信前，向系统申请共享存储区中的一个分区，并指定该分区的关键字，若系统已经给其它进程分配了这样的分区，则将该分区的描述符返回给申请者。接着，申请者把获得的共享存储分区连接到本进程上，此后可读写该分区。

以上两种方式的同步互斥都要由进程自己负责。

UNIX的共享存储区

- 创建或打开共享存储区(shmget)：依据用户给出的整数值key，创建新区或打开现有区，返回一个共享存储区ID。
- 连接共享存储区(shmat)：连接共享存储区到本进程的地址空间，可以指定虚拟地址或由系统分配，返回共享存储区首地址。父进程已连接的共享存储区可被fork创建的子进程继承。
- 拆除共享存储区连接(shmdt)：拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl)：对共享存储区进行控制。如：共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)；

2. 消息传递系统 Message passing system

- 进程间的数据交换以消息为单位，程序员利用系统的通信原语实现通信。
- 消息传递系统可分为：
  - 直接通信：发送进程直接把消息发送给接收者，并将它挂在接收进程的消息缓冲队列上。接收进程从消息缓冲队列中取得消息。也称为消息缓冲通信
  - 间接通信：发送进程将消息发送到某种中间实体中（信箱），接收进程从中取得消息。也称信箱通信。在网络中称为电子邮件系统。
- 两种方式的主要区别？
  - 前者需要两进程都存在，后者不需要。

3. 管道通信 Pipe

- 所谓管道，是指用于连接一个读进程和一个写进程的文件，称pipe文件。向管道提供输入的进程（称写进程），以字符流的形式将大量数据送入管道，而接受管道输出的进程（读进程）可从管道中接收数据。该方式首创于UNIX，它能传送大量数据，被广泛采用。

字符流方式写入读出
先进先出顺序

3.6.3.2 消息缓冲通信的实现

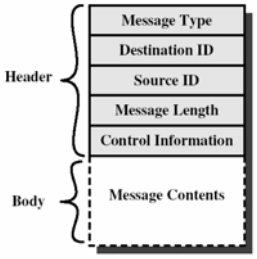
- 用消息传递的方式(传递数据块)达到通信的目的

1)系统管理空白缓冲池
2)发送者向系统申请空白缓冲区
3)发送者将填有消息的缓冲区挂到接收者的消息队列
4)接收者在适当时候从消息队列中读取数据
5)系统提供消息通信原语：
- send_message( ) ,
- receive_message( );

每个接收者有自己的消息队列

消息

- 消息：一段文本。消息格式设计与应用环境和要求有关
  - 固定长度消息：可以减小处理和存储的开销
  - 基于文件的：传送大量的数据
  - 可变长度消息：灵活
- 消息的一般格式
  - 消息头：源标识、目的标识、长度域、类型域、控制域
  - 消息体



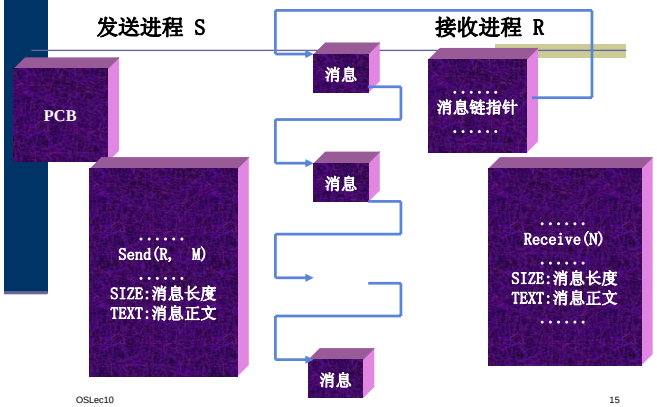
消息缓冲区结构

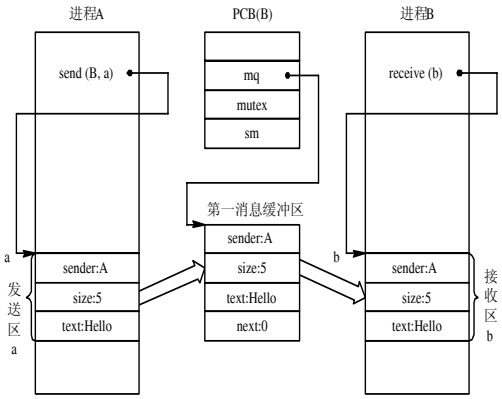
```
type message Buffer=record
  sender ;      //发送者ID
  size   ;      //消息长度
  text   ;      //消息正文
  next   ;      //消息队列指针
end
```


PCB中有关通信的数据项

```
type processcontrol block=record
  ...
  mq;       //消息队列队首指针
  mutex;    //消息队列互斥信号量
  sm; //消息队列资源信号量
  ...
end
```

当进程向一个满队列发送消息时，它将被挂起；
当进程从一个空队列读取时也会被挂起。





消息缓冲通信

用P、V操作来实现Send原语

```
send（R,M）
begin
  在OS中分配M.size大小的缓冲区t;
  将M中的内容复制到t;
  得到进程R的PCB的指针q;
  P(q.mutex);
      将t挂到队列q.mq队尾；
  V(q.mutex);
  V（q.sm);
end
```


用P、V操作来实现Receive原语

```
Receive(N)
begin
  得到本进程PCB的指针q；
  P(q.sm);
  P(q.mutex);
    从q.mq队首取下一个缓冲区t；
  V(q.mutex);
  将t的内容复制到N，并释放t
end
```

```
procedure send(receiver, a)
  begin
    getbuf(a.size,i);                    //根据a.size申请缓冲区；
    i.sender：= a.sender;   // 将发送区a中的信息复制到缓冲区;
    i.size：  = a.size;
    i.text：  =  a.text;
    i.next：  = 0;
   getid(PCB set, receiver.j);       // 获得接收进程内部标识符；
   wait(j.mutex);
   insert(j.mq, i);                  // 将消息缓冲区插入消息队列；
   signal(j.mutex);
   signal(j.sm);
end
```



```
procedure receive(b)
  begin
   j：  =internal name;       //j为接收进程内部的标识符；
   wait(j.sm);
   wait(j.mutex);
   remove(j.mq, i);           //将消息队列中第一个消息移出；
   signal(j.mutex);
   b.sender：  =i.sender;     //将缓冲区i中的信息复制到接收区b;
   b.size：  =i.size;
   b.text：  =i.text;
 end
```


3.6.3.3 信箱通信的实现

- 信箱的通讯过程

- 信箱的数据结构

```
type mailbox Buffer=record
    boxname ;        //信箱名
    boxsize  ;       //信箱大小
    mesnum  ;        //已存信件数
    fromnum ;        //空的格子数
    。。。。。。
end
```

信箱头（信箱名、口令）				
信格	信格	信格	信格	信格

信箱的使用

- 信箱的创建和撤消
- 消息的发送和接收

Send(mailbox，message)：将一个消息发送到指定信箱；
Receive(mailbox，message)：从指定信箱中接收一个消息；

- 信箱使用规则
  - 若发送信件时信箱已满，则发送进程被置为“等信箱”状态，直到信箱有空时才被唤醒
  - 若取信件时信箱中无信，则接收进程被置为“等信件”状态，直到有信件时才被唤醒

Send实现

- send（MailBox，M）：把信件M送到指定的信箱MailBox中
- 步骤：
  - 查找指定信箱MailBox；
  - 若信箱未满，则把信件M送入信箱且唤醒“等信件”者；
  - 若信箱已满置发送信件进程为“等信箱”状态；

Receive实现

- receive（MailBox，X）：从指定信箱MailBox中取出一封信，存放到指定的地址X中
- 步骤：
  - 查找指定信箱MailBox；
  - 若信箱中有信，则取出一封信存于X中且唤醒“等信箱”者；
  - 若信箱中无信件则置接收信件进程“等信件”状态；

信箱的分类

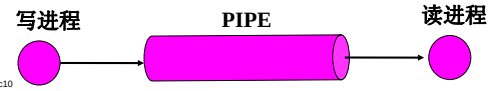
- 私用信箱
  - 用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。
- 公用信箱
  - 由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。
- 共享信箱
  - 由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

消息传递方式－类型

- 直接通信：必须明确命名通信的接收者或发送者，原语send和receive的定义如下：
  - Send(P,message):发送消息到进程P
  - Receive(Q,message):接收来自进程Q的消息。
- 间接通信：消息通过邮箱或端口来发送和接收。原语定义如下：
  - Send（A,message）；
  - receive（A,message）；

3.6.3.4 管道通信的实现

- 管道：是所有Unix都提供的一种IPC机制
  - 一个进程将数据写入管道，另一个进程从管道中读取数据
  - 是连接接收进程和发送进程的共享文件
    - (1) 互斥访问
    - (2) 写后读，读后写的同步
    - (3) 只有在管道双方都存在时才能通信
  - 无名管道，有名管道



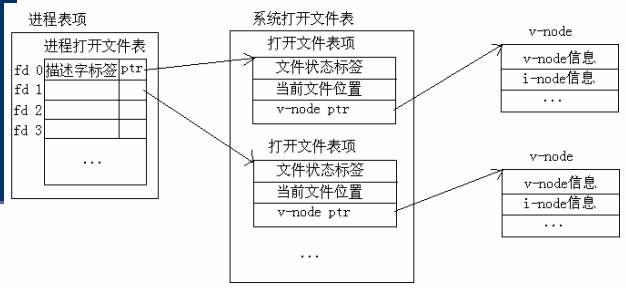
UNIX无名管道

- 通过pipe系统调用创建无名管道，得到两个文件描述符
  - 文件描述符fildes[0]只能从管道读，fildes[1]只能向管道写；
  - 通过系统调用write和read进行管道的写和读；

```
#include <unistd.h>
int pipe(int filedes[2]);
```

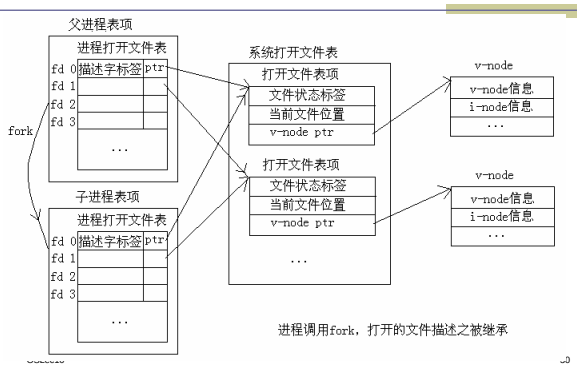
- 使用管道的两种限制：
  - 管道只能在具有亲缘关系的进程之间进行通信（具有公共祖先的进程之间）
    - 通过fork传递管道的描述符
  - 管道通信方式是单向的，进程间双向通信通常需要两个管道；

单个进程打开两个文件



一个进程打开两个文件的内核数据结构

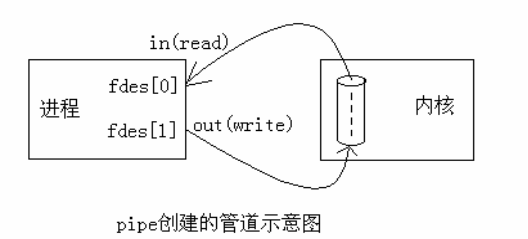
fork之后



进程调用fork，打开的文件描述之被继承

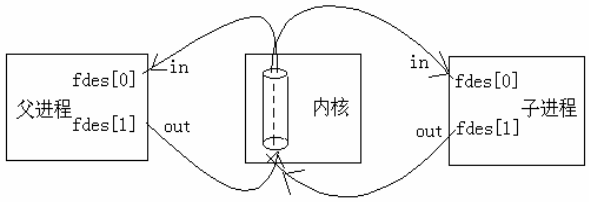
管道通信

- pipe函数成功后，内核打开两个文件描述字fdes[0]，fdes[1]。fdes[0]输入端，fdes[1]为输出端。
- 当进程调用了pipe



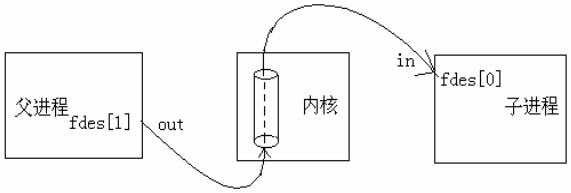
pipe创建的管道示意图

fork被调用后



fork之后共享一个管道的两个进程

两个进程分别关闭一个端



从父进程通往子进程的管道


```
int main(void) {
   pid_t pid;
   int fdes[2];
   if (pipe(fdes) < 0) { perror("pipe"); exit(1); }
   if ((pid = fork() < 0) { perror("fork"); exit(1); }
   if (pid > 0) {
      close(fdes[0]);
      write(fdes[1], "Hmmmmmmmmmmm", 12);
                  /* 1 2 3  4 5 6 7 8  9 0 1 2 */
   }
   else {
      char buf[4096]; ssize_t n;
      close(fdes[1])
      n = read(fdes[0], buf, 4096);
      if (n >= 0) { buf[n] = '\0'; printf("%s\n", buf); }
   }

   return 0;
```

管道破裂

- 如果一个管道的读端已经关闭，进程还继续向写端写数据，如：

```
pipe(fdes);
close(fdes[0]);
write(fdes[1], "Let me die", 10);
```

则进程会收到一个SIGPIPE信号，表示管道破裂。默认动作为结束进程。

- 读一个写端已经关闭的管道则read返回0。

UNIX有名管道

- FIFO，有名管道
  - 特殊的文件类型：
    1，先入先出
    2，类似管道，在文件系统中不存在数据块，而是与一块内核缓冲区相关联
    3，有名字，FIFO的名字包含在系统的目录树结构中，可以按名访问
  - open，close，read，write等普通文件操作

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Example

进程A1、A2，。。。An1通过m个缓冲区向进程B1、B2、。。。Bn2不断发送消息。发送和接收工作遵循下列规则：

（1） 每个发送进程一次发送一个消息，写入一个缓冲区，缓冲区大小等于消息长度

（2） 对每个消息，B1，B2，Bn2都须各接收一次，读入各自的数据区内

（3） m个缓冲区都满时，发送进程等待，没有可读消息时，接收进程等待。

试用P、V操作组织正确的发送和接收工作。

Hint……

- 每个缓冲区只要写一次但要读n2次，因此，可以看成n2组缓冲区，每个发送者要同时写n2个缓冲区，而每个接收者只要读它自己的缓冲区：
- Sin[n2]=m；
- Sout[n2]=0;

解：先将问题简化

- 设缓冲区的大小为1
- 有一个发送进程A1，有二个接收进程B1、B2
- 设有信号量Sin[1]、Sin[2] 初值为1
- 设有信号量Sout[1]、Sout[2] 初值为0

```
A1:
  while (1)
  {
     P(Sin[1]);
     P(Sin[2]);
       将数据放入缓冲区
     V(Sout[1]);
     V(Sout[2]);
  }
```

```
Bi:
  while (1)
  {
     P(Sout[i]);
        从缓冲区取数
     V(Sin[i]);
}
```


向目标前进一步

- 设缓冲区的大小为m
- 有一个发送进程A1
- 有二个接收进程B1、B2

- 设有信号量Sin[1]、Sin[2] 初值为m
- 设有信号量Sout[1]、Sout[2] 初值为0


```
A1:
  while (1)
  {
     P(Sin[1]);
     P(Sin[2]);
     P(mutex);
       将数据放入缓冲区
     V(mutex);
     V(Sout[1]);
     V(Sout[2]);
  }
```

```
Bi:
  while (1)
  {
     P(Sout[i]);
     P(mutex);
        从缓冲区取数
     V(mutex);
     V(Sin[i]);
};
```


到达目标

- 设缓冲区的大小为m
- 有n1个发送进程A1….An1
- 有n2个接收进程B1…Bn2

- 设有n2个信号量Sin[n2] 初值均为m
- 设有n2个信号量Sout[n2] 初值均为0

```
Aj:
while (1)
{
    for(i=1;i<=n2;i++)
     P(Sin[i]);
     P(mutex);
        将数据放入缓冲区
     V(mutex);
     for(i=1;i<=n2;i++)
     V(Sout[2]);
}
```

```
Bi:
   while (1)
   {
     P(Sout[i]);
      P(mutex);
         从缓冲区取数
      V(mutex);
      V(Sin[i]);
   };
```


利用直接通信原语解决生产者-消费者问题

```
repeat
      …
  produce an item in nextp;
      …
  send(consumer, nextp);
 until false;

 repeat
  receive(producer, nextc);
      …
  consume the item in nextc;
until false;
```


What you need to do?

- 复习课本3. 6节的内容
- 课后作业：习题16、33

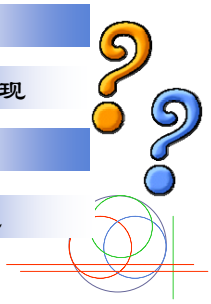
See you next time!

操作系统
第十一讲

王宇英
wangyy@nwpu.edu.cn

Review

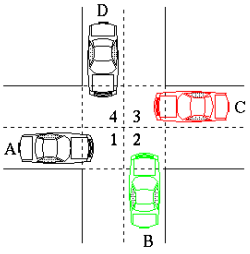
- 进程间通信类型
- 消息缓冲通信的实现
- 信箱通信的实现
- 管道通信的实现

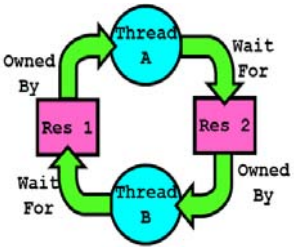


3.7 死锁（Deadlock）

- 死锁现象
- 产生死锁的原因和必要条件
- 处理死锁的基本方法
  - 死锁的预防
  - 死锁的避免
  - 死锁的检测
  - 死锁的解除

3.7.1 死锁现象





由于等待资源进程形成循环

Deadlock

- 死锁的规范定义

一组进程中，每个进程都无限等待被该组进程中另一进程所占有的资源，因而永远无法得到资源，这种现象称为进程死锁，这一组进程就称为死锁进程。

如果死锁发生，会浪费大量系统资源，甚至导致系统崩溃。

系统模型

- 系统拥有一定数量的资源，分布在若干竞争进程之间。
- 资源：
  - 物理资源：内存、CPU、I/O设备（打印机和磁带机等）
  - 逻辑资源：文件、信号量等
- 资源分成多种类型，每种类型有相同数量的实例。
- 正常操作模式下，进程按如下顺序使用资源
  - 申请（获得资源或者等待）－使用－释放
  - 并发环境下。。。

可重用资源与可消费资源

- 可重用资源
  - 一次只能供一个进程安全地使用，且不会由于使用而耗尽
  - 例子：处理器，I/O通道，主存和辅存，设备，文件、数据库、信号量等数据结构
- 可消费资源
  - 可以创建并且可以销毁的资源
  - 数目没有限制，当一个进程得到一个可消费资源时，这个资源就不再存在了
  - 例子：中断，消息，I/O缓冲区中的信息

两个进程竞争可重用资源死锁的例子

Process P		Process Q	
Step	Action	Step	Action
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

- 两个进程：一个访问磁盘文件D，一个访问磁带设备T
- 如果执行序列为：P0、P1、q0、q1、P2、q2，则发生死锁

涉及可消费资源死锁的例子

P1:	*P2*:
…	…
receive(P2);	receive(P1);
…	…
send(P2, M1);	send(P1, M2);
…	…

◆每个进程试图从另一个进程接收消息，然后再给它发送一条消息。

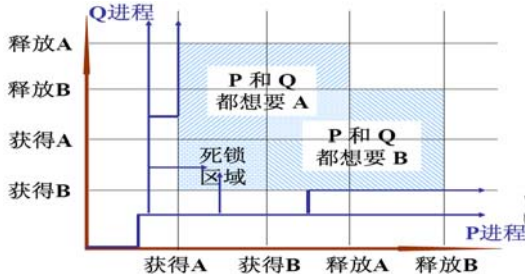
◆如果receive阻塞（接收进程被阻塞直到收到消息），则可能发生死锁。

死锁 vs. 饥饿
Deadlock vs. Starvation

- 饥饿，进程长时间等待处理器而得不到运行
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock ⇒Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

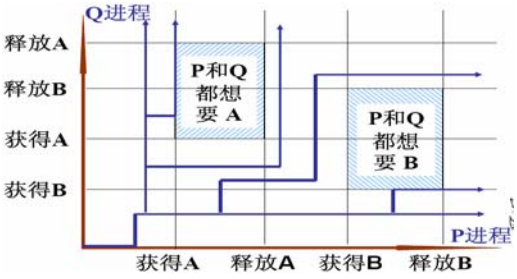
死锁可能发生

- P进程：获得A …获得B…释放A…释放B…
- Q进程：获得B …获得A…释放B…释放A…



死锁不发生

- P进程：获得A …释放A…获得B …释放B…
- Q进程：获得B …获得A…释放B…释放A…



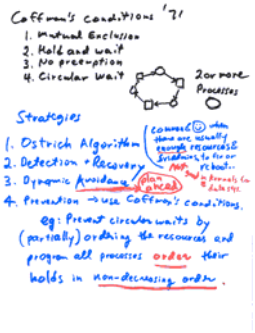
3.7.2 产生死锁的原因和必要条件

- 死锁指进程处于等待状态，且等待事件永远不会发生。
  - 参与死锁的进程最少是两个；
  - 参与死锁的进程至少有两个已经占有资源；
  - 参与死锁的所有进程都在等待资源；
  - 参与死锁的进程是当前系统中所有进程的子集。
- 产生死锁的原因：
  - 资源不足导致的资源竞争：多个进程所共享的资源不足，引起它们对资源的竞争而产生死锁。
  - 并发执行的顺序不当。进程运行过程中，请求和释放资源的顺序不当，而导致进程死锁. 如P, V操作的顺序不当

产生死锁的四个必要条件

- 互斥(mutual exclusion)
- 占有等待 (hold and wait)
- 不可剥夺 (no preemption)
- 环路等待(circular wait)

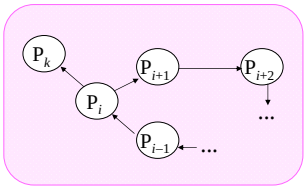
G. E. Coffman



四个必要条件

- 互斥使用(Mutual exclusion)
  - 一个资源只能给一个进程使用
- 请求保持(Hold and wait，占有且等待)
  - 非一次性授予：动态申请，动态分配
- 不可剥夺(No preemption)
  - 申请者不能抢夺占有者的资源，只能自愿释放
- 循环等待(Circular wait)
  - 进程组形成一个占有和请求的环路

- 系统中有两台输出设备，Pi+1 占有一台，Pk 占有另一台，且 k {0,1,…,n}。虽然系统有一个循环等循圈，但Pk 不在圈内。若Pk释放了输出设备，则可打破循环等待圈。因此循环等待只是死锁的必要条件。



3.7.3 处理死锁的基本方法

- 忽略此问题不做任何实际处理 (鸵鸟策略)
- 不允许出现死锁
  - 死锁预防 (deadlock prevention)
  - 死锁避免 (deadlock avoidance)
- 允许系统出现死锁后排除之
  - 死锁检测 (deadlock detection)
  - 死锁恢复 (deadlock recovery)

死锁处理方法

- 预防死锁：通过限制如何申请资源的方法来确保至少有一个条件不成立。
- 避免死锁：根据有关进程申请资源和使用资源的额外信息，确定对一个申请，进程是否应该等待
- 检测死锁和恢复：通过算法来检测并恢复
- 忽视此问题：认为死锁不可能在系统内发生。

3.7.4 死锁的预防

- 定义：在系统设计时确定资源分配算法，保证不发生死锁
- 死锁的四个必要条件记做C1，C2，C3，C4，死锁记做D，则有逻辑公式：D→ C1∧C2∧ C3∧ C4。
  推导得：¬ C1∨¬ C2∨¬ C3∨¬ C4→¬D
- 具体的做法：破坏产生死锁的四个必要条件之一
  - 互斥条件 ✗
  - 请求和保持条件（占有和等待）
  - 不可剥夺条件
  - 环路等待条件

破坏请求和保持条件

- 方法一：要求每个进程在运行前必须一次性申请它所要求的所有资源，
- 方法二：进程提出申请资源前必须释放已占有的一切资源。
- 优点：简单、易于实现、安全
- 缺点：
  - 一个进程可能被阻塞很长时间，等待资源，发生饥饿
  - 资源严重浪费，进程延迟运行；

破坏不可剥夺条件

- 方法一：OS可以剥夺一个进程占有的资源，分配给其他进程。
- 方法二：一个已经保持了某些资源的进程，当它再提出新的资源请求而不能立即得到满足时，必须释放它已经保持的所有资源，待以后需要时再重新申请
- 适用条件
  - 资源的状态可以很容易地保存和恢复，如CPU寄存器、内存空间，不能适用于打印机、磁带机
- 缺点
  - 实现复杂、代价大，反复申请/释放资源，系统开销大，降低系统吞吐量

破坏环路等待条件

- 采用资源有序分配法：
  - 把系统中所有资源编号，进程在申请资源时必须严格按资源编号的递增次序进行，否则操作系统不予分配

$$r_1, \quad r_2, \quad \ldots, \quad r_i \qquad F(r_i) = i$$
$$\downarrow \quad \downarrow \qquad \downarrow$$
$$1 \quad 2 \qquad i$$

- 例：哲学家就餐问题
- 问题：
  - 此方法要求资源类型序号相对稳定，不便于添加新类型的设备.
  - 易造成资源浪费，类型序号的安排只能考虑一般作业的情况，限制用户简单、自主地编程
  - 限制进程对资源的请求；资源的排序占用系统开销；

死锁的预防

- **Prevention**
  - **破坏互斥条件：适用于磁盘**
  - **破坏占有等待条件：静态分配策略**
  - **破坏非剥夺条件：适用于CPU、内存**
  - **破坏循环等待条件：层次分配策略**

3.7.5 死锁的避免

- 不需象死锁预防那样，事先采取限制措施破坏产生死锁的必要条件；在资源的动态分配过程中，采用某种策略防止系统进入不安全状态，从而避免发生死锁
  - 如果一个新的进程的资源请求会导致死锁，则拒绝启动这个进程
  - 如果满足一个进程新提出的一项资源请求会导致死锁，则拒绝分配资源给这个进程
- 在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

安全状态与不安全状态

- 安全系统：指系统能按某种进程顺序(P1, P2, …, Pn)来为每个进程Pi分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。（称〈P1, P2, …, Pn〉序列为安全序列）。
- 安全状态：如果存在一个由系统中所有进程构成的安全序列P_1，…P_n，则系统处于安全状态
- 不安全状态：不存在一个安全序列。
- 不安全状态不一定导致死锁，只是很可能死锁。

安全序列

一个进程序列{P_1，…，P_n}是安全的，如果对于每一个进程P_i（1≤i≤n），它以后尚需要的资源量不超过系统当前剩余资源量与所有进程P_j（j < i）当前占有资源量之和，系统处于安全状态

- 安全序列可以不唯一！

安全状态一定是没有死锁发生的
不安全状态：不存在一个安全序列。不一定导致死锁

安全状态举例

- 假定系统中有三个进程P1、P2和P3，共有12台磁带机。进程P1总共要求10台磁带机，P2和P3分别要求4台和9台。假设在T0时刻，进程P1、P2和P3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进程	最大需求	已分配	可用
P_1	10	5	3
P_2	4	2	
P_3	9	2	

死锁 → 四个必要条件
¬(4个必要条件)→¬(死锁)
4个必要条件 ↛ 死锁（不一定）

死锁预防是严格破坏4个必要条件之一，一定不出现死锁；而死锁的避免是不那么严格地限制死锁必要条件的存在，其目的是提高系统的资源利用率。万一当死锁有可能出现时，就小心避免这种情况的发生。

银行家算法

例：假设有三个进程P、Q、R，系统只有某类资源共10个，而三个进程合计申请资源数为20个。

目前的分配情况如下：

进程	已占资源个数Loan	需申请资源个数Claim	need = loan + claim
P	4	4	8
Q	2	1	3
R	2	7	9
合计	8	12	20

而系统　8 + 2 = 10

↑ 已分配 Loan　↑ 剩 Cash　↑ 共有 Capital

此后P、R再申请资源就不能分配了。因为现在只剩下2个资源，不能满足它们的最大要求 (P:4, R:7)，如果将剩下2个分配给P或R，则会产生死锁。

	已占	还申					
P	5	3	或	5	3	或	4	4
Q	2	1		2	1		2	1
R	3	6		3	6		4	5

然而将2个资源分给Q (只需一个)则

P	4	4		*P*	4	4
Q	3	0	⇒Q可运行结束、释放			系统资源回收为4个
R	2	7		*R*	2	7

此后可将4个资源分组P…

即 $Q \rightarrow P \rightarrow R$

* 单一种类资源引出的银行家算法的基本思想也同样适用于多种类的资源情况：

	r_1	r_2	r_3	r_4
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

资源已分配情况

r_1	r_2	r_3	r_4
4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

最大需求

还需(剩余需求)矩阵

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix}$$

总共资源数 r = (6, 3, 4, 2)

已分资源数 s = (5, 3, 2, 2)

余下资源数 t = (1, 0, 2, 0)

(1) 寻找剩余矩阵的某一未标记的行*x*，使得它的每一个元素都不大于向量*t* 的对应元素，如果找不到转(4)。

(2) 对于找到的行*x*，表示可以满足它，标记此进程，并将它占有的资源加到向量*t*。

(3) 重复上述步骤，转(1)。

(4) 如果所有进程都已标记，则状态是安全的，否则为不安全。

What you need to do?

- 复习课本3. 7节的内容
- 课后作业：习题17、18

See you next time!

操作系统
第十二讲

王宇英
wangyy@nwpu.edu.cn

Review

- 产生死锁的原因和必要条件
- 处理死锁的基本方法
- 死锁的预防
- 死锁的避免

Today we focus on …

- 死锁的避免——银行家算法
- 死锁的检测
- 死锁的解除

3.7.5 死锁的避免

- 对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源
- 安全状态：
  - 如果存在一个由系统中所有进程构成的安全序列P1，…，Pn，则系统处于安全状态
  - 安全状态一定是没有死锁发生的
- 安全序列P1，…Pn
  - 进程序列P1，…Pn是安全的iff对于每个进程Pi(1≤i≤n)，Pi需要资源≤剩余资源+分配给Pj(1≤j<i)资源
  - 不安全状态可能导致死锁，但不一定是死锁状态

银行家算法

- 基本思想
  - 银行家拥有一笔周转资金
  - 客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款
  - 银行家应谨慎的贷款，防止出现坏帐
- 用银行家算法避免死锁
  - 操作系统（银行家）
  - 操作系统管理的资源(周转资金)
  - 进程（要求贷款的客户）

银行家算法

- 数据结构：
  - n：系统中进程总数；m：资源类总数
  - Available[1..m]表示每种资源的剩余数量
  - Max[1..n,1..m] 进程对每种资源的最大需求
  - Allocation[1..n,1..m]表示当前给每个进程分配的各种资源数量
  - Need[1..n,1..m] =Max-Allocation表示当前每个进程还需分配的各种资源数
  - Request[1..m]表示进程Pi提出的资源申请

Example

可利用资源向量Available：

A	B	C
5	2	3

最大需求矩阵Max：

	A	B	C
P1	5	2	2
P2	2	1	1
P3	4	2	3
P4	3	1	2

银行家算法

- 针对进程Pi提出资源申请Request：
  - 若Request≤Need[i],继续;否则错误返回
  - 若Request≤Available,继续;否则进程等待
  - 系统试图分配资源：
    - Available=Available-Request;
    - Allocation[i]=Allocation[i]+Request;
    - Need[i]=Need[i]-Request；
  - 对该状态进行安全性检测，若不安全则不分配

安全性检测

(1) 设置两个向量：

① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，初始化Work：=Available;

② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。初始化时设置所以Finish［i］：=false; 当有足够资源分配给进程时，再令Finish［i］：=true。

(2) 从进程集合中找到一个能满足下述条件的进程：

① Finish［i］=false;

② Need［i,j］≤Work［j］；若找到，执行步骤(3)，否则，执行步骤(4)。

(3) 当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］：=Work［i］+Allocation［i,j］;

Finish［i］：=true;

go to step 2;

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

银行家算法—举例

- 假定系统有三个进程P1，P2，P3，共有12台磁带机。进程P1总共要求10台磁带机，P2和P3分别要求4台和9台。设在T0时刻进程P1，P2，P3已分别获得5，2，2台，尚有3台空余未分。

	最大需求	已分配	尚需	可用
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

3台空闲 → 可使P2结束 → P2运行，并释放资源 → 5台空闲 → 可使P1结束 → P1运行，并释放资源 → 10台空闲 → 可使P3结束 → P3

称 P2，P1，P3 为安全序列

银行家算法—举例

- 条件同前例，如果：P3提出申请2台资源，问是否可以满足要求？

假设法

解：先假设能够满足要求且分配资源，则系统状态如下：

	最大需求	已分配	尚需	可用
P1	10	5	5	1
P2	4	2	2	
P3	9	4	5	

只剩余 1 台资源，不能使任何一个进程执行结束，因而不存在安全序列，所以系统不安全，因此拒绝分配资源，撤消开始的假设。

一个安全性计算的实例

问：T0时刻是否为安全状态？

T_0时刻资源分配表

进程 \ 资源情况	Claim			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	3	3	2
P_1	3	2	2	2	0	0	1	2	2			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

进程 \ 资源情况	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	3	2	1	2	2	2	0	0	5	3	2	True
P_3	5	3	2	0	1	1	2	1	1	7	4	3	True
P_4	7	4	3	4	3	1	0	0	2	7	4	5	True
P_2	7	4	5	6	0	0	3	0	2	10	4	7	True
P_0	10	4	7	7	4	3	0	1	0	10	5	7	True

- P1发出请求Request(1,0,2)，执行银行家算法

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0	1	0	7	4	3	3	3	2
p1	3	2	2	2	0	0	1	2	2	(2	3	0)
				(3	0	2)	(0	2	0)			
p2	9	0	2	3	0	2	6	0	0			
p3	2	2	2	2	1	1	0	1	1			
p4	4	3	3	0	0	2	4	3	1			

	Work	Need	Alloc	Work+alloc	Finish
	A B C	A B C	A B C	A B C	
p1	2 3 0	0 2 0	3 0 2	5 3 2	true
p3	5 3 2	0 1 1	2 1 1	7 4 3	true
p4	7 4 3	4 3 1	0 0 2	7 4 5	true
p0	7 4 5	7 4 3	0 1 0	7 5 5	true
p2	7 5 5	6 0 0	3 0 2	10 5 7	true

P1申请资源(1,0,2)时安全性检查(安全)

P1请求资源之后：

P4发出请求Request(3,3,0), 执行银行家算法

Available=2 3 0

不能通过算法第2步（ Request[i]≤Available ），所以P4等待。

P0请求资源：Request（0，2，0），执行银行家算法

	Allocation	Need	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	

为P0分配（0，2，0）后的情况（不安全）

- **练习：有三类资源A(17)、B(5)、C(20)。有5个进程P1—P5。T0时刻系统状态如下：**

	最大需求	已分配
P1	5 5 9	2 1 2
P2	5 3 6	4 0 2
P3	4 0 11	4 0 5
P4	4 2 5	2 0 4
P5	4 2 4	3 1 4

问(1)、T0时刻是否为安全状态，若是，给出安全系列。

(2)、T0时刻，P2: Request(0,3,4)，能否分配，为什么？

(3)、在(2)的基础上P4：Request(2,0,1)，能否分配，为什么？

(4)、在(3)的基础上P1：Request(0,2,0)，能否分配，为什么？

解

	最大资源需求M			已分配资源数量U			尚需资源N		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	2	4	3	1	4	1	1	0

剩余向量A = （2，3，3）

1. **是否安全？**

(2, 3, 3) → P4 (4, 3, 7) → P5 (7, 4, 11) → P1 (9, 5, 13) → P2 (13, 5, 15) → P3

(2, 0, 4)　(3, 1, 4)　(2, 1, 2)　(4, 0, 2)

安全序列：p4, p5, p1, p2, p3

思考： 安全序列是否唯一？

解：

	最大资源需求M			已分配资源数量U			尚需资源N		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	2	4	3	1	4	1	1	0

剩余向量A =（2，3，3）

2. 在T0时刻若进程P2请求资源（0，3，4），是否能实施资源分配？为什么？

解：R（0，3，4）> A（2，3，3）

所以不能满足，不能分配

解

	最大资源需求M			已分配资源数量U			尚需资源N		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	2	4	3	1	4	1	1	0

剩余向量A =（2，3，3）

③ 在②的基础上，若进程P4请求资源（2，0，1），是否能实施资源分配？为什么？

解：因为R（2，0，1）<=N4(2, 0, 1) 且 < A（2，3，3）

假设可以满足，则N4=(0, 0, 0), A=（0，3，2），在此基础上，

(0, 3, 2) → P4 → (4, 3, 7) → P5 → (7, 4, 11) → P1 → (9, 5, 13) → P2 → (13, 5, 15) → P3

所以是安全的，因此可以实施分配

(4) P1：Request(0,2,0)

	Allocation	Need	Available
P1	2 3 2	3 2 7	0 1 2
P2	4 0 2	1 3 4	
P3	4 0 5	0 0 6	
P4	4 0 5	0 2 0	
P5	3 1 4	1 1 0	

0 1 2 已不能满足任何进程的需要，不能分配

Summary of Banker's Algorithm

- 优点
  - 比死锁预防限制少
  - 无死锁检测方法中的资源剥夺，进程重启
- 缺点
  - 必须事先声明每个进程请求的最大资源
  - 考虑的进程必须是无关的，也就是说，它们执行的顺序没有任何同步要求的限制
  - 进程的数量保持固定不变，且分配的资源数目必须是固定的
  - 在占有资源时，进程不能退出

3.7.6 死锁的检测

- 允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生
- 一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行
- 检测时机：
  - 当进程等待时检测死锁（缺点是系统的开销大）
  - 定时检测
  - 系统资源利用率下降时检测死锁

死锁检测算法

- 局部变量
  - Work[1..m]用来表示当前可分配的资源
  - Finish[1..n]用来标记进程是否处于死锁
- 初始化
  - Work= Available
  - for i = 1,2, …, n, if Allocation[i]≠0, then Finish[i] = false;else, Finish[i] = true

- 死锁检测
  - 1. 寻找*i*:Finish[*i*]==false&& Need[*i*]≤Work
    if no such *i*, goto3
  - 2. Work = Work+ Allocation[*i*]
    Finish[*i*] = true; goto1
  - 3. for *i*(1≤*i*≤*n*) if (Finish[*i*] == false),
    then return *deadlock* = *deadlock*+ {*i* }

死锁进程组

死锁检测实例

- 进程：*P*0…*P*4; 资源：A(数量为7),*B*(2),*C*(6)
- 当前状态：

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- 序列<*P*0,*P*2,*P*3,*P*1,*P*4>使得所有Finish[i]=true

死锁检测实例

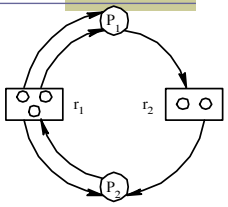
- 进程：*P*0…*P*4; 资源：A(7),*B*(2),*C*(6)
- *P*2再申请一个资源*C*:

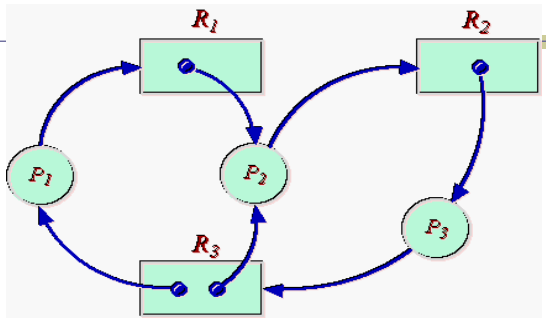
	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- 死锁进程组为<*P*1,*P*2,*P*3,*P*4>

资源分配图
Resource Allocation Graph

- 二元组G=（V，E）
  - V：结点集，分为P，R两部分
    - P={p1,p2,…,pn}
    - R={r1,r2,…,rm}
  - E：边的集合，其元素为有序二元组
    - (pi,rj)或(rj,pi)
- 表示法
  - 资源类（资源的不同类型），用方框表示
  - 资源实例（每个资源类中），用方框中的黑圆点（圈）表示
  - 进程，用圆圈中加进程名表示
- 分配边：资源实例→进程的一条有向边
- 申请边：进程→资源类的一条有向边





例：系统有两台宽行打印机和一台图形显示器，进程P_1请求一台宽打，则有资源分配图1。进程P_1分配到一台宽打，并请示一台图形显示器，进程P_2已分到一台图显，并请求一台宽打，则有其分配图2。

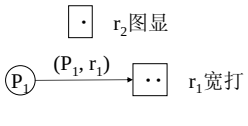


图1

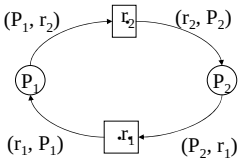


图2

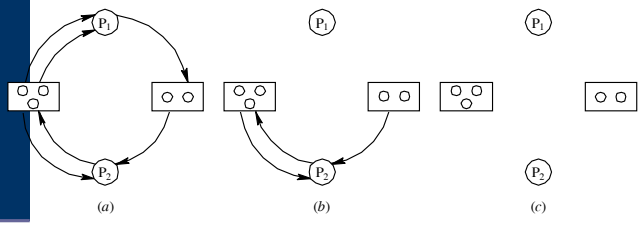
死锁定理

- 如果资源分配图中没有环路，则系统中没有死锁，如果图中存在环路则可能存在死锁。
- 如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件。

资源分配图的化简

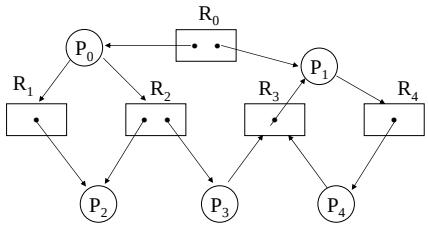
- 1）找一个非孤立进程结点且只有分配边，去掉分配边，将其变为孤立结点
- 2）再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边
- 3）重复以上步骤，若所有进程成为孤立结点，称该图是可完全简化的，否则称该图是不可完全简化的。

- 一个给定的进程-资源图的全部化简序列导致同一不可化简图。
- 死锁状态的充分条件：当且仅当资源分配图是不可完全简化的。

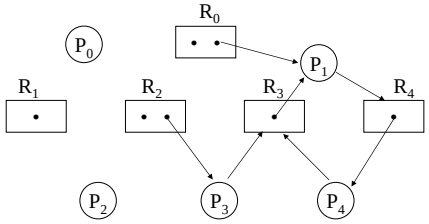


资源分配图的简化

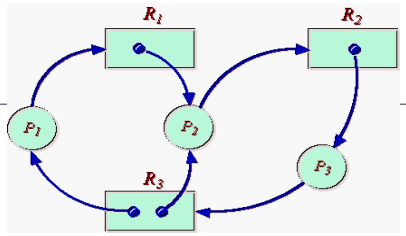
分配图化简：



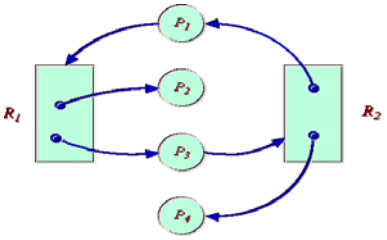
解：化简得：



有一个循环等待的圈，死锁。



有环有死锁



有环无死锁

3.7.7 死锁的解除
Deadlock Recovery

- 以最小的代价恢复系统的运行。方法如下：
  - 1）重新启动
  - 2）撤消进程
  - 3）剥夺资源
  - 4）进程回退

- 方法2：进程终止
  - 终止所有的死锁进程－OS中常用方法
  - 一次只终止一个进程直到取消死锁循环为止－基于某种最小代价原则。
- 选择原则
  - 已消耗CPU时间最少
  - 到目前为止产生的输出量最少
  - 预计剩余的时间最长
  - 目前为止分配的资源总量最少
  - 优先级最低
- 终止进程需要做很多工作

- 方法3：资源抢占：逐步从进程中强占资源给其它进程使用，直到死锁环被打破为止。考虑如下问题：
  - 选择一个牺牲品：抢占哪些资源和哪个进程，确定抢占顺序以使代价最小。
  - 饥饿：确保资源不会总是从同一个进程中被抢占
- 方法4：回滚：把每个死锁进程备份到前面定义的某些检查点，并且重新启动所有进程－需要系统构造重新运行和重新启动机制

What you need to do?

- 复习课本3.7节的内容
- 课后作业：习题19、22、29、31、34

See you next time!