6.1 并发进程

6.1.1 并发程序设计

并发程序设计: 把一个具体问题求解设计成若干个可同时执行的程序模块的方法

并发程序设计方式的特性

并行性: 多个进程在多道程序系统中并发执行或在多处理器系统中并行执行

提高了计算效率

共享性: 多个进程共享软件资源

交往性: 多个进程并发执行时存在制约，增加了程序设计的难度

就是这个交往性导致了并发程序设计中关于进程互斥与同步的问题

6.1.2 并发进程的制约关系

无关的并发进程：一组并发进程分别在不同的变量集合上运行，一个进程的执行与其他并发进程的进展无关

交往的并发进程：一组并发进程共享某些变量，一个进程的执行可能影响其他并发进程的结果

进程互斥: 并发进程之间因相互争夺独占性资源而产生的竞争制约关系

进程同步: 并发进程之间为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系

同步的关键在于等待，例如计算进程必须在输入进程输入结束之后，才能对输入结果进行处理

与时间相关的两类执行错误：结果错误、永远等待

6.2 临界区

6.2.1 临界区概念

临界资源：临界资源: 互斥共享变量所代表的资源，即一次只能被一个进程使用的资源

临界区：并发过程中与互斥共享变量相关的程序段

如果两个进程同时停留在相关的临界区内，就会出现进程互斥或同步问题

两个进程的临界区有相同的临界资源，就是相关的临界区，必须互斥进入

两个临界区不相关，进入则没有限制

临界区的描述：

确定临界资源：shared <variable>

确定临界区：region <variable> do <statement\_list>

临界区管理的三个要求：

一次至多允许一个进程停留在相关的临界区

一个进程不能无限止地停留在临界区内

一个进程不能无限止地等待进入临界区（避免饿死）

临界区的嵌套使用：如果要对临界区进行嵌套使用，则必须规定各临界区资源的优先级，否则就会出现死锁情况

6.2.2 临界区两个错误尝试

错误想法: 利用一个标志量来标记进程是否在临界区中，并与 while 一同使用

尝试一

看上去没有问题，但是会导致两个进程同时进入临界区：

P1 刚执行完 while，发生中断，P2 执行

P2 刚进入临界区，发生中断，P1 执行

P1 进入临界区，此时两个进程同时进入临界区

尝试二

两个进程会陷入死锁

P1 刚执行完 inside1 的赋值，被中断，P2 执行

P2 执行完 inside2 的赋值后，P1 与 P2 陷入死锁

成功方法：peterson算法

6.2.3 临界区管理实现的硬件方式

关中断、测试并设置指令（TS）、对换指令

关中断：仅适用于系统内部程序的临界区操作手段，而且要保证指令长度短小精悍

如果把开关中断权限开放给用户，可能会发生滥用，即造成巨大破坏性

6.3 PV 操作（信号量）

为了改进 TS 或开关中断方式，引入 PV 操作

信号量

核心数据结构: 等待进程队列

信号量声明: 资源报到，建立队列

申请资源原语: 若申请不到，调用进程入队等待

归还资源原语: 若队列中有等待进程，需释放

信号量撤销: 资源注销，撤销队列

每个信号量建立一个等待进程队列、每个信号量对应一个整数值：

正值表示资源可复用次数、0 值表示无资源且无进程等待、负值表示资源等待队列中进程个数

PV 操作

procedure P(semaphore:s) {

s = s - 1; // 信号量减 1

if (s < 0) W(s); // 若信号量小于 0，则进程进入等待队列

}

procedure V(semaphore:s) {

s = s + 1; // 信号量加 1

if (s <= 0) R(s); // 若信号量小于等于 0，则释放等待队列中一个进程

}

可以利用 PV 操作来解决进程互斥问题

PV 操作解决机票问题

int A[m];

Semaphore s;

s = 1;

cobegin

process Pi {

Li: 按旅客订票要去找到 A[j]

P(s);

if ( A[j] >= 1 ) {

A[j]--;

V(s);

// 输出一张票

}

else {

V(s);

// 输出票已售完

}

goto Li;

}

coend;

注意上述代码中，P 操作与 V 操作在执行路径上一一匹配除此之外，上述代码还有一个问题，即只有相同航班的票数才是相关的临界资源，因此用一个信号量处理全部机票会影响进程并发度，下述代码为修改后的代码

int A[m];

Semaphore s[m];

for (int j = 0; j < m; j++) s[j] = 1;

cobegin

process Pi {

L1: 按旅客订票要去找到 A[j]

P(s[j]);

if ( A[j] >= 1 ) {

A[j]--;

V(s);

// 输出一张票

}

else {

V(s);

// 输出票已售完

}

goto Li;

}

6.3.2 进程同步

进程同步: 并发进程为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系

一个进程的执行等待来自于其他进程的消息

生产者消费者问题概述

生产者、消费者共享缓冲区

缓冲区有空位时，生产者可放入产品，否则等待

缓冲区有产品时，消费者可取出产品，否则等待

种类：

1生产，1消费，1缓冲

1生产，1消费，N缓冲

缓冲区扩大了，因此需要加上缓冲区首尾指针

N生产，N消费，N缓冲

由于生产者和消费者数量增加，因此缓冲区的首尾指针也需要保证互斥性

注意点，先同步后互斥，否则会死锁如果是先互斥再同步的话，会出现生产者把缓冲区生产满了，此时先拿锁，然后进入等待队列，而消费者拿不到锁也没办法消费，导致死锁

6.4 管程

6.4.1 管程概念

概念提出：管程本质是对信号量的类封装，简化操作

管程试图抽象相关并发进程对共享变量访问，以提供一个友善的并发程序设计开发环境

管程是由若干公共变量及其说明和所有访问这些变量的过程所组成

管程把分散在各个进程中互斥地访问公共变量的那些临界区集中起来管理，管程的局部变量只能由该管程的过程存取

进程只能互斥地调用管程中的过程

管程的条件变量

条件变量: 当调用管程过程的进程无法运行时，用于阻塞进程的信号量

同步原语 wait: 当一个管程过程发现无法继续时，它在某些条件变量上执行 wait，引起进程阻塞

同步原语 signal: 用于释放在条件变量上阻塞的进程

执行模型

注意，管程中一共有 3 个等待进程队列由于进程只能互斥地调用管程中的过程，因此有了互斥调用管程过程队列

又因为当前在管程中的进程一旦调用 signal，则会有两个进程在管程中，所以又有一个更高优先级的互斥调用队列

signal 问题：当使用 signal 释放一个等待进程时，可能出现两个进程同时停留在管程内，此时有三种解决方法

执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件（霍尔管程）

被释放进程等待，直到执行 signal 的进程退出管程或等待另一个条件

规定管程中的 signal 操作时过程体的最后一个操作（汉森法，增加程序设计难度）

6.4.2 霍尔管程

霍尔管程：使用 signal 释放一个等待进程时，霍尔管程让执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件

霍尔管程基于 PV 操作原语实现

Wait 和 Signal 可以是程序过程，不需要扩展 OS 内核

信号量：

两个互斥信号量

if (IM.next\_count > 0) V(IM.next);

else V(IM.mutex);

条件变量：

x\_sem: semaphore; // 与资源相关的信号量

x\_count: integer; // 在 x\_sem 上等待的进程数

Wait：

void Wait(semaphore, x\_count, IM){

x\_count++; // 等待数加上自己

if(IM.next\_count > 0) V(IM.next);

else V(IM.mutex);

P(x\_sem); // 挂起自己

x\_count--; // 等待数减去自己

}

Signal：

void Signal(semaphore, x\_count, IM){

if(x\_count > 0){

IM.next\_count++; // 进程队列加入自己

V(x\_sem); // 信号量操作

P(IM.next); // 将自己阻塞住，先执行被 signal 的进程

IM.next\_count--; // 进程队列去除自己

}

}

6.4.3 哲学家问题

没讲，没看

6.5 进程通信

6.5.1 进程通信概念

交往进程通过信号量操作实现进程互斥和同步，这是一种低级通信方式

进程有时还需要交换更多的信息（如何把数据传送给另一个进程），可以引入高级通信方式 —— 进程通信机制，实现进程间用信件来交换信息

进程通信扩充了并发进程的数据共享

6.5.2 进程直接通信

发送或接收信件的进程指出信件发给谁，或从谁那里接收信件

send(P, 信件): 把信件发送给进程 P

receive(Q, 信件): 从进程 Q 接收信件

6.5.3 进程间接通信

间接通信概念

发送或者接收信件通过一个信箱来进行，该信箱有唯一标识符

多个进程共享一个信箱

send(A, 信件): 把信件传送到信箱 A

receive(A, 信件): 从信箱 A 接收信件

信箱

信箱是存放信件的存储区域，每个信箱可以分成信箱特征和信箱体两部分

信箱特征: 信箱容量、信件格式、指针

信箱体: 分成若干个区，每个区可容纳一封信

发送信件原语流程

若指定的信箱未满，则把信件送入信箱中指针所指示的位置，释放等待该信箱中信件的等待者

否则，发送信件者被置成等待信箱的状态

接收信件原语流程

若指定信箱中有信件，则取出一封信，释放等待信箱的等待者

否则，接收信件者被置成等待信箱中信件的状态

6.5.4 基于字节流的通信规约

多个进程使用一个共享的消息缓冲区（管道、多路转接器、套接字）

一些进程往消息缓冲区中写入字符流

一些进程从消息缓冲区中读出字符流

信息交换单位基于字符流，长度任意

6.5.5 基于RPC的高级通信规约

采用客户/服务器计算模式

服务器进程提供一系列过程/服务，供客户进程调用

客户进程通过调用服务器进程提供的过程/服务获得服务

考虑到客户计算机和服务器计算机的硬件异构型，外部数据表示 XDR 被引入来转换每台计算机的特殊数据格式为标准数据格式

6.6 死锁

这里需要主要一下死锁防止和死锁避免的区别，下面给出一段英文描述

关键点在于死锁防止是从根本上解决死锁问题，而死锁避免是人为地尽可能避开，还是有出意外的可能

6.5.1 死锁的定义

一组进程处于死锁状态是指，每一个进程都在等待被另一个进程所占有的、不能抢占的资源

存在n个进程 P 1 , P 2 , . . . , P n ，进程 P i （1<=i<=n）因为申请不到资源 R i 而处于等待状态，而 R i又被 P i + 1 （1<=i<=n-1）占有，显然，这 n个进程的等待状态永远不能结束，这 n个进程就处于死锁状态

6.5.2 死锁产生的四个必要条件

互斥条件: 进程应互斥使用资源，任一时刻一个资源仅为一个进程独占

占有和等待条件: 一个进程请求资源得不到满足而等待时，不释放已占有的资源

不剥夺条件: 任一进程不能从另一进程那里抢夺资源

循环等待条件: 存在一个循环等待链，每一个进程分别等待它前一个进程所持有的资源

6.5.3 死锁的防止

独占型资源改为共享（破坏条件1）

独占变共享，可以破坏条件 1，但这对许多资源往往是不能做到的

剥夺式调度（破坏条件3）

剥夺式调度目前只适用于对主存资源和处理器资源的分配，而不适用于所有资源

静态分配（预分配 - 破坏条件2）

具体方法: 一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足之后才开始执行

缺点: 可能会出现一种情况进程执行时，资源 A 只使用了一小段时间，但在整个执行过程中，资源 A 始终被占用，造成资源利用率的低效

层次分配（破坏条件4）

具体方法: 将资源分成多个层次一个进程得到某一层的一个资源后，它只能再申请在较高层的资源，当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源当一个进程获得了某一层的一个资源后，它想再申请该层中的另一个资源，那么必须先释放该层中的已占资源

缺点: 低层次资源使用效率依然较低

6.5.4 死锁的避免

当不能防止死锁的产生时，如果能掌握并发进程中与每个进程有关的资源申请情况，仍然可以避免死锁的发生

只需在为申请者分配资源前先测试系统状态，若把资源分配给申请者会产生死锁的话，则拒绝分配，否则接收申请，为它分配资源

银行家算法: 借钱给有偿还能力的客户

建立多个数据结构：最大需求表(下称为需求表1)，分配表，可用向量

需求表1-分配表 = 现需求表(下称为需求表2)

将现在可用数据置为工作向量

（1）进程提出资源请求，操作系统将请求与需求表2以及可用向量相比较

（2）如果符合要求，则为其分配相应的请求资源，进程得到满足之后释放资源，此时更新可用资源

（3）如果不满足请求则让进程进入等待队列，直到可用资源可以满足进程的需求，此时再将资源分配给相应进程并唤醒

（4）重复此步骤可以得到一个安全序列或者得出系统不安全的结论

6.5.5 死锁的检测

死锁的检测对资源的分配不加限制，但系统定时运行一个 “死锁检测” 程序，判断系统内是否已出现死锁，若检测到死锁则设法加以解除

检测方法

设置两张表格来记录进程使用资源的情况

等待资源表: 记录每个被阻塞进程等待的资源

占用资源表: 记录每个进程占有的资源

可以根据上述两张表构建出一张死锁的有向图，若 P i等待资源 r k ，且 r k 被进程 P j 占用，则 P i 和 P j 具有 “等待占用关系”，记为 W ( P i , P j ) )

可以构造一个二维矩阵，运行传递必包算法，如果对角线为 1，则存在环，出现死锁也可以用拓扑排序的方法，O(n) 判环

for(int k = 1; k <= n; k++)

for(int i = 1; i <= n; i++)

for(int j = 1; j <= n; j++)

b[i][j] = b[i][j] | (b[i][k] & b[k][j]);

死锁后的解决方法

可以采用重新启动进程执行的办法，恢复工作应包含重启动一个或全部进程，以及从哪一点开始重启动

全部卷入死锁从头开始启动，但这样的代价是相当大的

在进程执行过程中定时设置校验点，从校验点开始重新执行

中止一个卷入死锁的进程，以后重执行