**操作系统中的并发问题**

**一、并发**

操作系统中的并发指的是两个或两个以上的事件在同一时间间隔内发生。它是进程在时间上的交错，从而呈现出在同时执行的现象。 它不同于并行性，它提供了真正的同时执行。

并发性有三个概念：原子性、可见性、有序性问题。

原子性是指，一个或多个操作要么全部执行且执行过程中不被打断，要么就都不执行。

可见性是指，多个线程同时访问一个数据时，如果一个线程修改了该数据，其他线程应该立刻能看到修改的值。

有序性是指，程序执行的顺序按照代码的先后顺序执行。

1. **产生的问题**

系统中的多道程序并发执行，虽然能改善资源利用率、提高系统吞吐量，但是也会导致进程对系统资源的无序争夺造成混乱。经典并发问题如：生产者-消费者问题、哲学家进餐问题、读者-写者问题，下面就写这三个问题的解决方法。

1. **生产者-消费者问题**

两个线程，一个做“生产者”，另一个做“消费者”。生产者会生成数据放到缓冲区中个，消费者会消耗缓冲区的的数据。但是消费者不能消费空的缓冲区，生产者不能将数据放入满的的缓冲区。

为了实现这些，就需要保持线程间的同步，一个线程生产或消费完之后其他线程方能竞争CPU，这就需要对资源加锁，从而保证某时刻只有一个线程在访问该资源。

**伪代码：**

令items代表缓冲区已经使用过的资源，space代表可用资源数。mutex表示互斥锁,start和end分别表示第一个和最后一个资源的位置。

//生产者

Producer{

int start=0,end=0;

items it[n]={0};//用于存放资源

while(true)

{

wait(space);//等待缓冲区有空闲位置的信号

wait(mutex);//等待没有其他线程在占用缓冲区的信号

it[start]=item;//将item存入it的start位置

start=(start+1)%n;//编号循环增加，范围从1到n

signal(mutex);//释放占用

signal(items);//当前缓冲区资源数

}

}

//消费者

consumer

{

while(true)

{

wait(items);//等待有资源可以使用

wait(mutex);//等待缓冲区不被占用

item=it[end];//取出end位置的资源

end=(end+1)%n;//编号循环增加，范围从1到n

signal(mutex);//释放占用

signal(space);//当前缓冲区空闲位置

}

}

**c++代码：**

#include <iostream>

#include <queue>

#include <thread>

#include <mutex>

#include <condition\_variable>

using namespace std;

mutex mut;

condition\_variable produce, consume; // 条件变量是一种同步机制，要和mutex以及lock一起使用

const int maxSize = 20;

queue<int> que;

void consumer()

{

while (1)

{

unique\_lock<mutex> lck(mut);

while (que.size() == 0)

consume.wait(lck); //condition\_variable.wait()锁至满足while条件不满足

cout << "消费者 " << this\_thread::get\_id() << ": ";

que.pop();

cout << que.size() <<endl;

produce.notify\_all();

lck.unlock();

}

}

void producer(int id)

{

while (true)

{

unique\_lock<mutex> lck(mut);

while (que.size() == maxSize)

produce.wait(lck);

cout << "-> 生产者 " << this\_thread::get\_id() << ": ";

que.push(id);

cout << que.size() <<endl;

consume.notify\_all();

lck.unlock();

}

}

int main()

{

thread consumers[2], producers[2];

for (int i = 0; i < 2; ++i)

{

consumers[i] = thread(consumer);

producers[i] = thread(producer, i + 1); //thread：第一个参数是task任务，第二个参数是task函数的参数

}

for (int i = 0; i < 2; ++i)

{

producers[i].join();

consumers[i].join();

}

return 0;

}

1. **哲学家进餐问题**

五个哲学家分别坐在一张圆桌的五张椅子上，桌上有五只碗和五只筷子。只有在哲学家同时拿到左右手的两只筷子时才能进餐。筷子就相当于临界资源，只允许一个哲学家单独占用，互斥访问。解决方法有三种：

1. **最多4个哲学家同时拿左边筷子**

semaphore chopstick[5]={1,1,1,1,1};

semaphore count=4; //最多4个哲学家可以进来

void philosopher(int i)

{

while(true)

{

think();

wait(count); //请求进入房间进餐 当count为0时 不能允许哲学家再进来了

wait(chopstick[i]); //请求左手边的筷子

wait(chopstick[(i+1)%5]); //请求右手边的筷子

eat();

signal(chopstick[i]); //释放左手边的筷子

signal(chopstick[(i+1)%5]); //释放右手边的筷子

signal(count); //离开饭桌释放信号量

}

}

1. **哲学家左右两边的筷子都没人使用时，才可以拿起筷子**

semaphore mutex = 1; //需判断两根筷子是否可用 semaphore chopstick[5]={1,1,1,1,1};

void philosopher(int i)

{

while(true)

{

think();

wait(mutex); // 保护信号量

wait(chopstick[(i+1)%5]); // 请求右手边的筷子

wait(chopstick[i]); // 请求左手边的筷子

signal(mutex); // 释放保护信号量

eat();

signal(chopstick[(i+1)%5]); // 释放右手边的筷子

signal(chopstick[i]); // 释放左手边的筷子

}

}

1. **奇数位上的哲学家先拿左边筷子再拿右边筷子，偶数位哲学家先拿右边筷子再拿左边筷子。**

semaphore chopstick[5]={1,1,1,1,1};

void philosopher(int i)

{

while(true)

{

think();

if(i%2)//奇数哲学家，先左后右。

{

wait (chopstick[i]) ; //请求左手边的筷子

wait (chopstick[(i + 1)%5]) ; //请求右手边筷子

eat();

signal (chopstick[i]) ;//释放左手边的筷子

signal (chopstick[(i + 1)%5]) ;//释放右边筷子

}

else //偶数哲学家，先右后左

{

wait (chopstick[(i + 1)%5]) ;/请求右手边筷子

wait (chopstick[i]) ;//请求左手边的筷子

eat();

signal (chopstick[(i + 1)%5]) ;//释放右边筷子

signal (chopstick[i]) ;//释放左手边的筷子

}

}

}

1. **读者-写者问题**

当两个或两个以上的读进程同时访问共享数据时，不会产生副作用，但是当某个写进程同时和其他读进程或写进程访问共享数据时，则可能会导致数据不一致而引发混乱。

因此要保证写进程必须和其他进程互斥的访问共享对象，即允许多个读者同时访问或者只允许一个写者单独访问共享数据。

smaphore rMutex=1,wMutex=1;//读写互斥信号量

int readcount=0;//读进程数目

Reader{

while(true){

wait(rMutex);//等待可读信号

if(!readcount)//若没有读进程

wait(wMutex);//等待读写互斥信号释放

readcount++;//读进程数量加一

signal(rMutex);//释放读进程信号

Read();

readcount--;

if(readcount==0)//当没有读进程的时候，释放可写信号

signal(wMutex);

}

}

Writer{

while(true){

wait(wMutex);//等待可写信号

Write();//写

signal(wMutex);//释放写进程占用资源

}

}