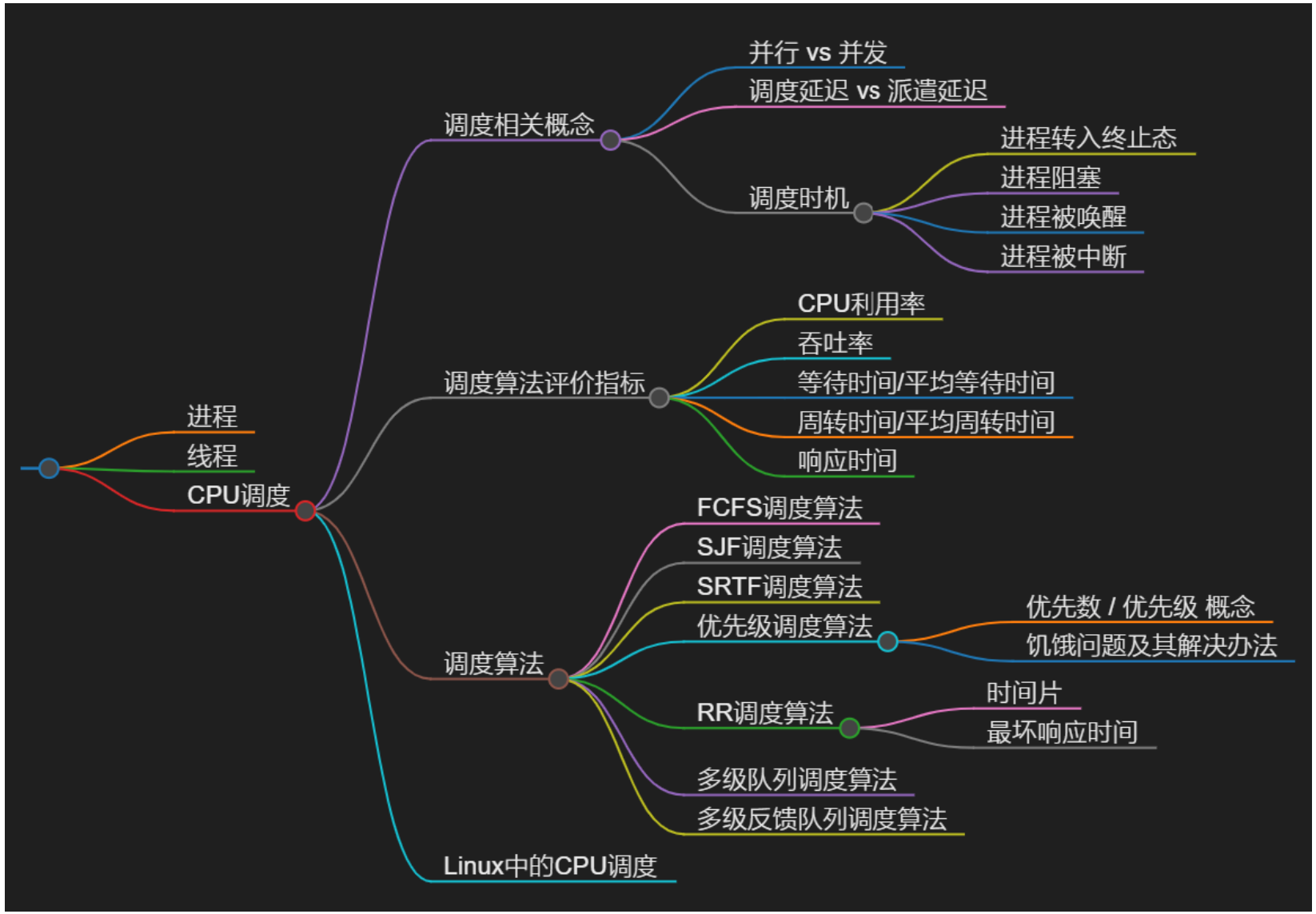
操作系统

L08 进程同步1

胡燕
大连理工大学 软件学院

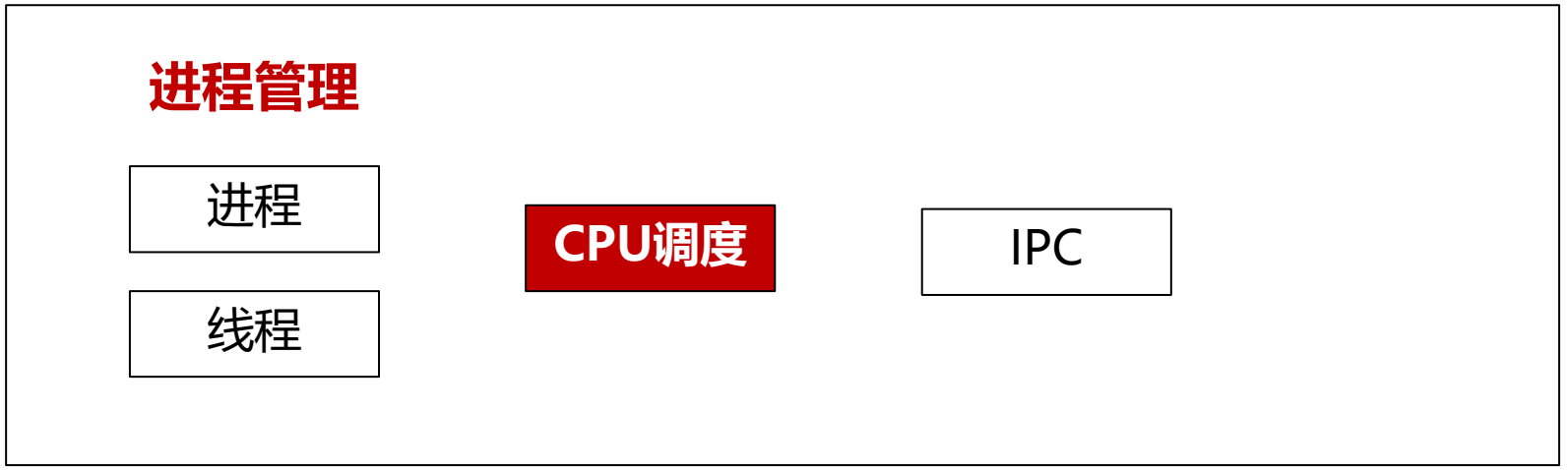
CPU调度回顾

- **CPU调度**



引子

- **进程管理**



OS的事件驱动视角：系统的运转的核心为一串中断事件+对中断事件的响应

当中断处理过程中，出现合理的调度时机，则调度器介入，进行任务切换

任务异步执行是常态，效率高

投票　最多可选1项

设置

系统为（　），则进程P一旦被唤醒就能够投入运行。

A　分时系统，进程P的优先级最高

B　抢占调度方式，就绪队列上的所有进程的优先级皆比P的低

C　就绪队列为空队列

D　抢占调度方式，P的优先级高于当前运行的进程

A.系统采取轮转调度，当前运行进程可能继续运行
B.进程P可能 比正在运行进程的优先级低
C.存在正在运行的进程，那么P可能仍要进入就绪队列等待被调度
D.正确答案，抢占式系统中，P的优先级大于当前运行进程，也就一定比就绪的进程优先级高

提交

投票　最多可选1项

设置

为照顾紧迫型任务，应采用（　）。

A　先来先服务调度算法

B　短作业优先调度算法

C　优先级调度算法

D　时间片轮转调度算法

提交

投票 最多可选1项

设置

（　）优先权在创建进程时确定，确定之后在整个进程运行期间不再改变。

A 先来先服务

B 静态

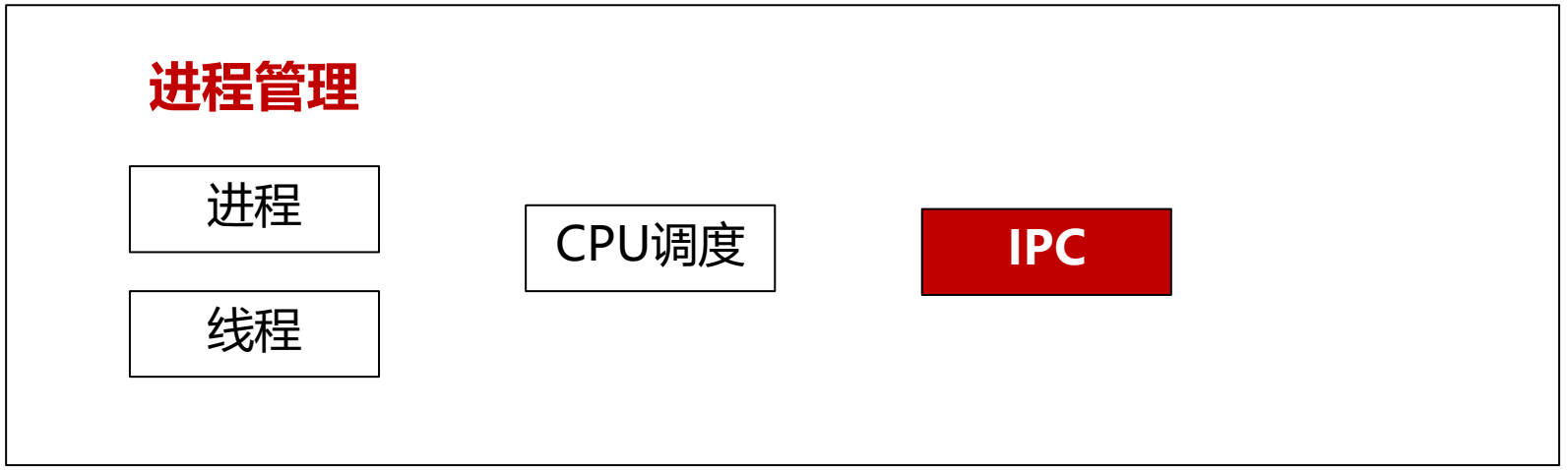
C 动态

D 短作业

提交

引子

- **进程管理**



IPC： 一个大规模任务的完成需要一个团队中的个体各展所长。
为了完成任务，在做好自己的分工事情的同时，还要做好沟通。

现实中的问题 => 操作系统中的多任务协同完成复杂任务

进程协作

- **Cooperation**

为何需要进程协作

- **现代OS的典型特点：Multitasking**

- 进程（任务）在独立地址空间中各自运行
- 不同进程异步执行

进程间存在协作需求 => Synchronization

两个任务需要协作：工作+看娃

为何需要进程协作

- **异步为主，同步为辅**

- 当某个进程进行到特定点（如baby睡醒），需要博得另外一个进程关注时，系统必须提供必要的协助

进程异步执行是为了效率，进程同步是为了将多个相关进程联系在一起，保证系统环境正确、和谐

直接协作 v.s. 间接协作

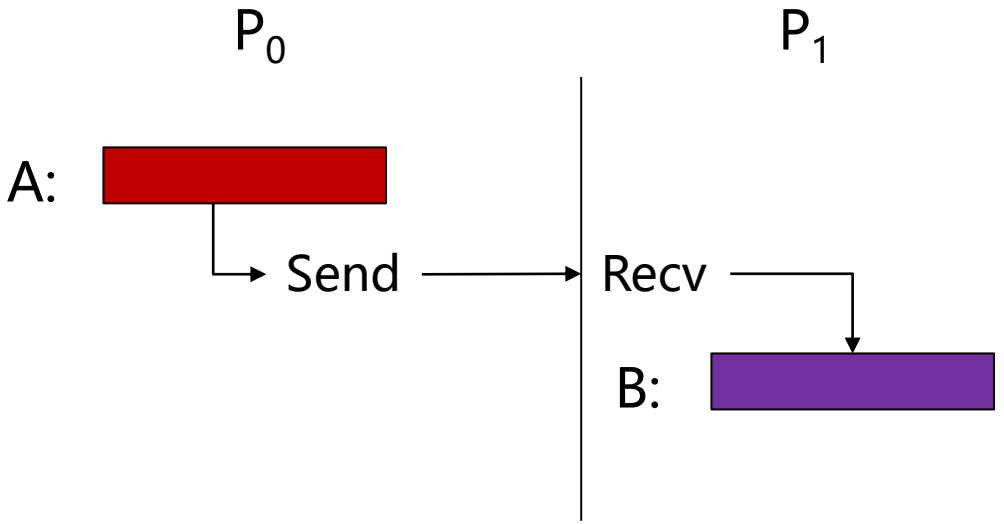
- **进程协作可分为两大类型：直接协作和间接协作**

直接协作（Direct Cooperation）

例如：直接消息发送与接收

进程P_0与进程P_1存在直接协作关系：
- 当P_0完成A点执行后，会发送一个消息通知；
- 该通知由进程P_1执行到位置B时接收

如果进程P_1率先执行到位置B，在未获得消息之前，该进程处于阻塞状态
待到进程P_0执行到位置A后，消息得以发送后，进程P_1被唤醒进入就绪态



进程直接协作示例

司机 P_1

```
While(true)
{
    启动车辆;
    正常运行;
    到站停车;
}
```

售票员 P_2

```
While(true)
{
    关门;
    售票;
    开门;
}
```

主观题 10分

设置

试分析，司机与售票员之间的协作关系，并用箭头在合适的位置加以表示。

司机 P_1

```
While(true)
{
    启动车辆;
    正常运行;
    到站停车;
}
```

售票员 P_2

```
While(true)
{
    关门;
    售票;
    开门;
}
```

作答

直接协作 v.s. 间接协作

- **进程协作可分为两大类型：直接协作和间接协作**

直接协作（Direct Cooperation）

间接协作（Indirect Cooperation）

进程之间间接协作

- 多个进程竞争使用共享数据（资源）
- 间接作用不会强制合作进程之间遵循特定的先后顺序
- 间接合作进程必须保证对共享数据的一致性访问

直接协作 v.s. 间接协作

- 进程间接协作：场景示例

多人竞争使用独木桥通行权

直接协作 v.s. 间接协作

- 进程之间间接作用示例：共享变量

```
P1:
  if(x≥100){
      x-=100;
  }
```

```
P2:
  if(x≥100){
      x-=100;
  }
```

x为共享变量，初值=100

借记卡账户取100块钱逻辑：
账面余额大于等于100，则取出，余额减去100
若账面余额不足，则取款失败

从执行逻辑上，应该要求：
每个进程对x值的判断，以及对x的减值操作一气呵成

直接协作 v.s. 间接协作

- 进程之间间接作用示例：共享变量

P_1:
```
if(x≥100){
    x-=100;
}
```

P_2:
```
if(x≥100){
    x-=100;
}
```

x为共享变量，初值=100

进程P1
if(x>=100);
进程P1
x-=100;
进程P2
if(x>=100)
进程P2
x-=100;

两个进程串行执行，结果没有问题

直接协作 v.s. 间接协作

- 进程之间间接作用示例：共享变量

P_1:
```
if(x≥100){
    x-=100;
}
```

P_2:
```
if(x≥100){
    x-=100;
}
```

x为共享变量，初值=100

问题是：现代操作系统中，为了整体执行效率的提升，采取的是进程并发执行模式

直接协作 v.s. 间接协作

- 进程之间间接作用示例：共享变量

P_1:
```
if(x≥100){
    x-=100;
}
```

P_2:
```
if(x≥100){
    x-=100;
}
```

x为共享变量，初值=100

进程P1
if(x>=100);
进程P2
if(x>=100)
进程P1
x-=100;
进程P2
x-=100;

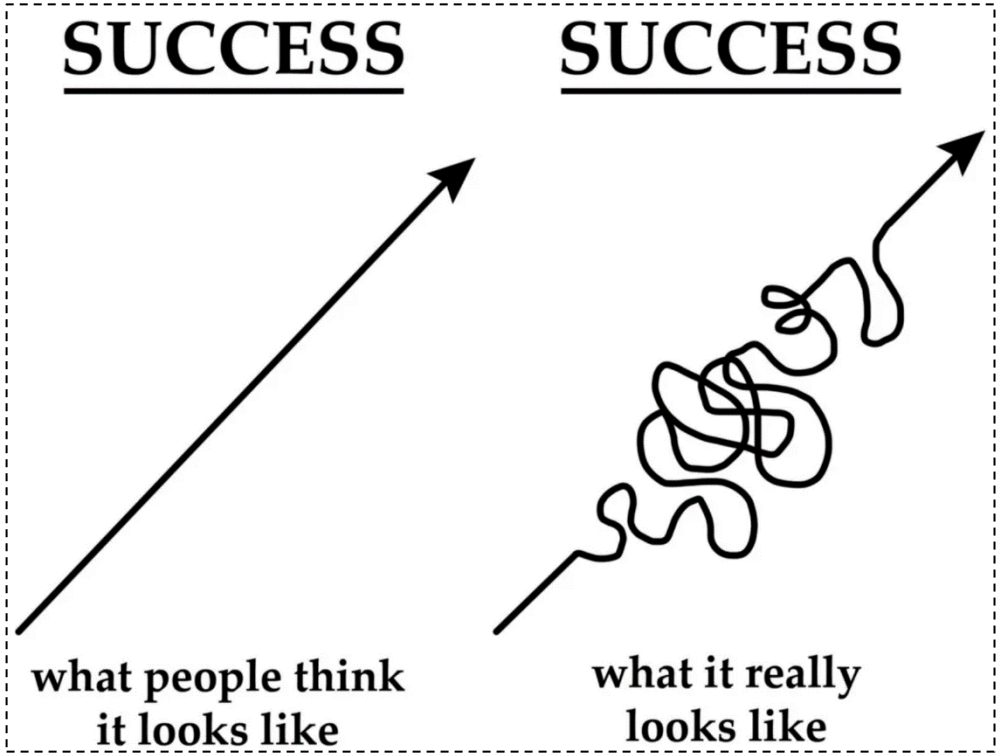
并发执行模式下的一种可能执行顺序

导致什么结果?

直接协作 v.s. 间接协作

进程间接协作目标

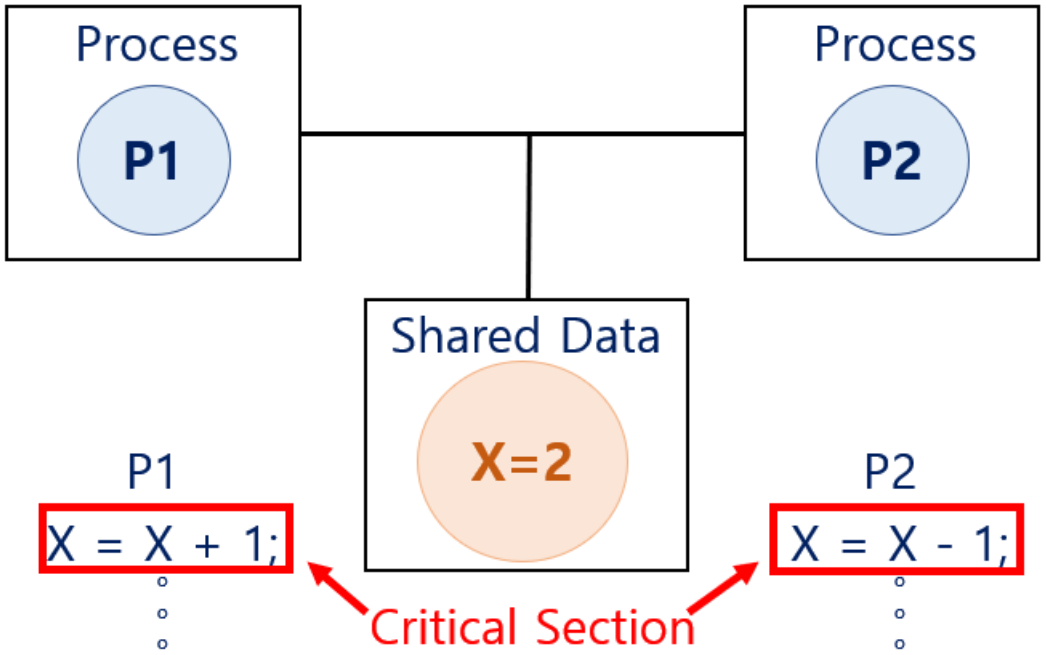
操作系统中，任务间的间接协作关系需要设立专门的机制，以保证结果的一致性



须保证执行过程乱中有序

什么是临界区问题

- **临界区问题：对进程间接协作问题进行建模**
  - 多个进程并发访问共享变量（代表共享资源）
  - 访问共享变量的代码区域，称为临界区（Critical Section）



什么是临界区问题

- **临界区问题的性质**
  - Only one thread can be inside critical section
  - Others attempting to enter Critical Section must wait until thread that’ s inside Critical section leaves it.

临界区要求以互斥方式访问

什么是临界区问题

- **临界资源**
  - 被多个并发进程竞争使用的共享变量（共享资源）
  - 例如：
    - 被P1和P2并发访问的共享变量x
    - 限制同一时期只能1个方向通行的独木桥
    - 一次仅能一人使用的试衣间

临界区问题的解决思路

- **通过保证临界区被以互斥的方式访问，从而保证并发进程计算结果的一致性**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

- 不应对并发进程作任何访问顺序的约束
- 通过在临界区进入和退出的位置进行控制，从而保证结果一致性

问题是：进入区和退出区应该做什么，才能保证临界区问题被正确处理

临界区问题的解决思路

- **通过保证临界区被以互斥的方式访问，从而保证并发进程计算结果的一致性**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

互斥要求：
- **互斥访问**：同一时刻仅有一个进程能进入CS
- **空闲让进**（Progress）
- **有限等待**：不能让某个进程无休止等待

思考途径

一、 两进程解法-尝试1

二、 两进程解法-尝试2

三、 Peterson算法

四、 多进程软件解法

从两进程问题，开始探索问题本质

两进程简化问题

两进程P0,P1

Try1: 使用一个共享变量turn，初值为0(或1)

```
do{
   <进入区>
   <临界区>
    <退出区>

    <剩余区>
}while(1);
```

i=0或1
j=1-i

进程Pi：

```
do{
   while(turn!=i);
    <临界区>
   turn=j;

    <剩余区>
}while(1);
```

问题：如果P0循环体执行速度快于P1，可能会出现P0处于turn=1，但是P1实质是在剩余区
（P0空闲不让进）

妥不妥?

两进程简化问题

两进程P0,P1

进程Pi：

```
do{
    while(turn!=i);
    <临界区>
    turn=j;

    <剩余区>
}while(1);
```

单个共享变量turn显得不够用

i=0或1
j=1-i

Try2：用共享布尔数组flag[2]来替代共享变量turn

进程Pi：

```
do{
    flag[i]=true;
    While(flag[j]);
    <临界区>
    Flag[i]=false;

    <剩余区>
}while(1);
```

问题：存在空闲不让进的问题

存在一种并发执行模式，使得flag[0],flag[1]均被设为true，而导致进程P0，P1陷入相互循环等待的问题

两进程简化问题

两进程P0,P1

Try2解决方案

进程Pi：

```
do{
    flag[i]=true;
    while(flag[j]);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

i=0或1

j=1-i

Try3： 布尔数组flag[2]与turn同时使用(Peterson算法)

进程Pi：

```
do{
    flag[i]=true;
    turn=j;
    while(flag[j]
        &&turn==j);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

Peterson算法是两进程临界区问题的正确解法

Lamport’s Bakery Algorithm

涉及n个进程的临界区问题求解 （n>2）

```
do{
    choosing[i]=true;
    number[i]=max(number[0],...,number[n-1])+1;
    choosing[i]=false;

    for(j=0;j<n;j++){
        while(choosing[j]);
        while((number[j]!=0) &&(number[j],j)<(number[i],i));
    }

    临界区

    number[i]=0;

    剩余区
}while(1);
```

取时间戳

如果进程Pj在领取时间戳，则等待

如果取得号比Pi小的进程，亦等待

当number[j]<number[i], 结果为true;
当number[j]==number[i]，且j<i，结果为true

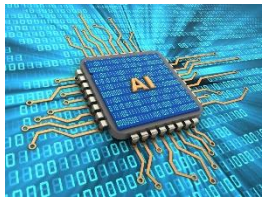
问题：临界区进入区的执行代价太高

为何要引入硬件方案

Hardware

Software

当纯软件算法无法达到效率要求时，需要软硬协同设计



为何要引入硬件方案

面临的挑战：临界区问题的进入区保护，若单纯使用软件方案，代价巨大

重新思考临界区保护的问题本质

若有一个进程已进入临界区，则不再允许其他进程进入临界区

现实生活中的解决思路：进程进入临界区后，给该临界区加上锁

关键词：LOCK

互斥的硬件解法思路

临界区加锁解决思路

```
lock = 0
```

进程A

```
while(lock != 0)
    NULL;
lock = 1;
Critical_section();
lock = 0;
```

进程B

```
while(lock != 0)
    NULL;
lock = 1;
Critical_section();
lock = 0;
```

核心问题：时间窗

先判断lock是否是开着的（对应临界资源空闲，临界区无人进入的状态）

实施加锁操作，锁定临界区（设置lock为1）

软件加锁方案时间窗问题示意

初始锁状态：lock = 0

```
进程A
while(lock != 0);
```
→ 发生进程切换A->B

```
进程B
while(lock != 0);
```
→ 发生进程切换B->A

```
进程A
lock = 1;
critical_section();
```

```
进程B
lock = 1;
critical_section();
```

结果：进程A和B同时进入临界区
（违背临界区的互斥要求）

互斥的硬件解法思路

硬件加锁目标：封锁时间窗

```
While(lock!=0);

Lock=1;
```

X ← 其他任务的指令

将要实现加锁的这两个操作整合，做成一个原子操作

执行期间不会被中断的指令序列，称为原子操作（Atomic Operation）

互斥的硬件解法思路

- 最终实现思路归结为：提供TestAndSet指令
  - 使用TestAndSet来处理上述临界区问题
  - TestAndSet(lock)指令语义：
    - lock=0, 则将lock置1，返回0
    - lock=1,则直接返回1

- TestAndSet实例：x86上的xchg指令

互斥的硬件解法思路

临界区通过基于TestAndSet原子操作实现的锁对象保护，方式如下

```
do{
    while(TestAndSet(lock));
    临界区
    lock = false;
    剩余区;
}while(1);
```

满足临界区条件(1)和(2):
-互斥
-空闲让进

不满足临界区条件（3）：
非有限等待

请尝试思考给出解决方案

互斥的硬件解法思路

- **终于实现**

“躲进厕所锁上门，我把全世界的人都所在外面”

互斥的硬件解法思路

- 引入互斥硬件解法的意义
- 互斥硬件解法的思路
- 基于加锁操作的互斥实现

Summary

小结：

- 进程同步概念
- 临界区问题
- 互斥软件解法
- 互斥硬件解法

讨论

临界区代码

临界区代码的特点是什么?

临界区代码是否可被中断?

互斥-TestAndSet指令

基于TestAndSet指令的互斥实现

```
//布尔型共享变量 lock 表示当前临界区是否被加锁
//true 表示已加锁，false 表示未加锁
bool TestAndSet (bool *lock){
  bool old;
  old = *lock;  //old用来存放lock 原来的值
  *lock = true; //无论之前是否已加锁，都将lock设为true
  return old;   //返回lock原来的值
}
```

TestAndSet指令语义

```
//以下是使用 TSL 指令实现互斥的算法逻辑
while (TestAndSet (&lock)); //"上锁"并"检查"
临界区代码段...
lock = false;       //"解锁"
剩余区代码段...
```

基于TestAndSet的互斥实现

互斥-XCHG指令

基于Swap指令的互斥实现

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
  bool temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Swap指令语义

?

基于Swap指令的互斥实现

互斥-XCHG指令

基于Swap指令的互斥实现

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
  bool temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Swap指令语义

```
//以下是用 Swap 指令实现互斥的算法逻辑
//lock 表示当前临界区是否被加锁
bool old = true;
while (old == true)
  Swap (&lock, &old);
临界区代码段...
lock = false;
剩余区代码段...
```

基于Swap指令的互斥实现

进程直接制约：生活中的例子

- 接力赛
- 产品的供应链
- 生物的食物链

进程间接制约：生活中的例子

- 上课时教室中同学占用的座位
- 试衣间的使用
- 十字路口的交通

谢 谢!
Thank you!