澳門城市大學
Universidade da Cidade de Macau
City University of Macau

作業系統概論
07-進程互斥

主講人｜ 澳門城市大學
City University of Macau

眭相傑 助理教授
Sui Xiangjie Assistant Professor

大綱

1. 順序與並發進程
2. 進程互斥
   - 軟件實現方法
   - 硬件實現方法

大綱

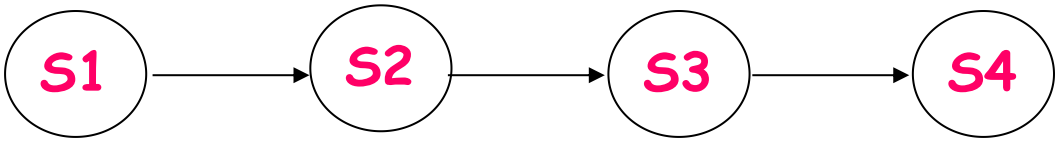
1. 順序與並發進程
2. 進程互斥
   - 軟件實現方法
   - 硬件實現方法

7-1 順序與並發進程

➢ 程序的順序執行

- 內部順序性：對於一個進程來說，它的所有指令是按序執行的

S1：a:=x+y
S2：b:=a-z
S3：c:=a+b
S4：d:=c+5

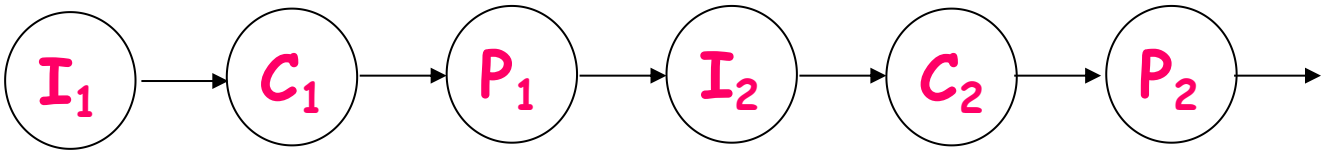


7-1 順序與並發進程

➢ 程序的順序執行

- **外部順序性**：對於多個進程來說，所有進程的活動是依次執行的

輸入(I)、計算(C)、打印(P)三個活動構成的進程，每個進程的內部活動是順序的，即Ii→Ci→Pi，多個進程的活動也是順序的



7-1 順序與並發進程

➢ 順序程序特性

- 順序性：處理機嚴格按照指令次序依次執行；
- 封閉性：程序在執行過程中獨占系統中的全部資源，運行環境只與其自身動作有關，不受其它程序及外界因素影響；
- 可再現性：程序的執行結果與執行速度無關，只與初始條件有關

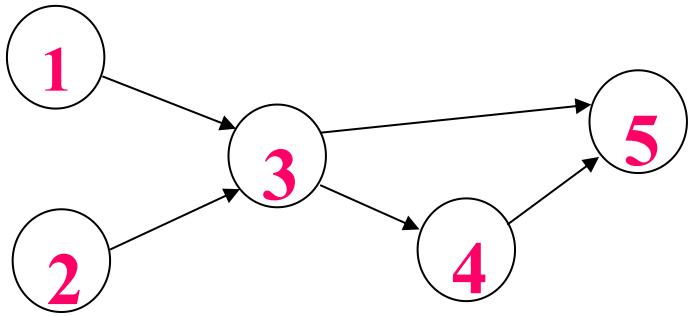
給定相同的初始條件，程序的任意多次執行一定得到相同的執行結果

7-1 順序與並發進程

➢ 程序的並發執行

- 內部並發性：一個程序內部的並發性

S1：a:=x+2；
S2：b:=y+4；
S3：c:=a+b；
S4：d:=c+6；
S5：e:=c-d；

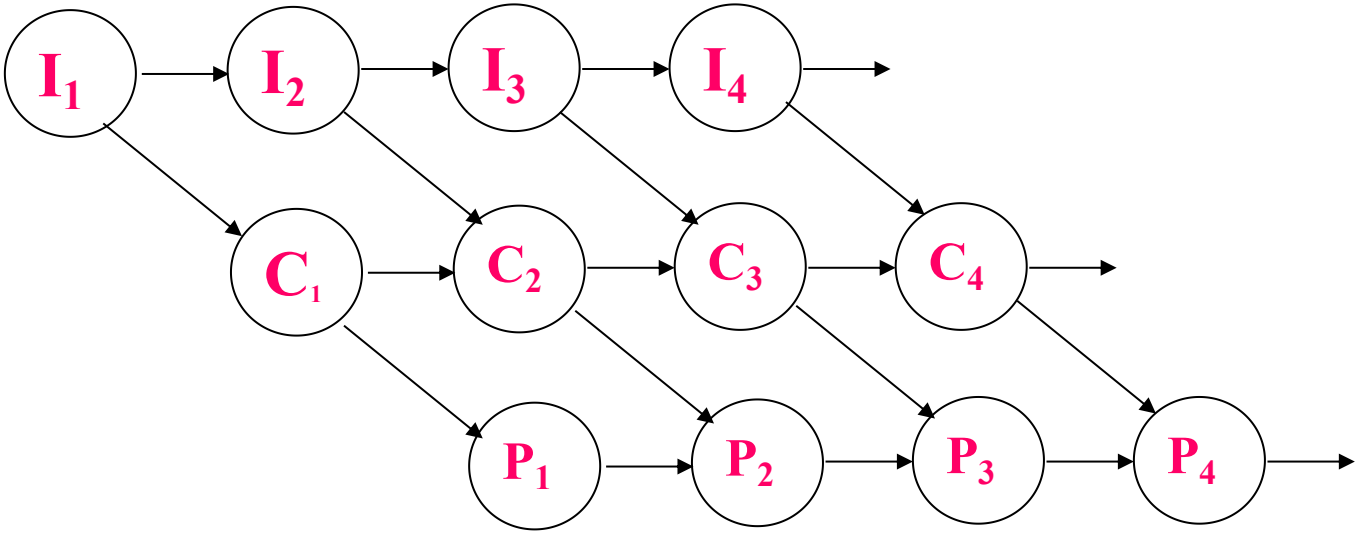


7-1 順序與並發進程

➢ 程序的並發執行

◘ 內部並發性：多個程序活動的重疊執行

I2與C1並發執行
I3、C2與P1並發執行



7-1 順序與並發進程

➢ 並發程序特性

- 交叉性：程序並發對應某一種交叉，不同的交叉可能導致不同的計算結果
- 非封閉性：一個進程的運行環境可能被其它進程所改變，從而相互影響
- 不可再現性：由於交叉的隨機性，並發程序的多次執行可能對應不同的交叉

操作系統應保證只產生導致正確結果的交叉，去除可能導致不正確結果的交叉

7-1 順序與並發進程

➢ 不可再現性的例子

進程**P**累積計數，進程**Q**將結果打印出來，打印出的數是累計數的和。

進程P
A1: N=0
A2: N=:N+1
A3: GOTO A2

進程Q
B1: PRINT（N）
B2: N:=0
B3: GOTO B1

兩進程並發執行時有許多可能的交叉

Case 1:
P - 5 loops, Q - 1 loop,
P - 3 loops, Q - 1 loop,

輸出結果是5，3

Case 2:
P - 6 loops, Q - 1 loop,
P - 2 loops, Q - 1 loop,

輸出結果是6，2

Case 3：Q執行完B1後被中斷，P對N執行加1操作，然後Q執行B2，在這種情況下P的累計數將被丟失

7-1 順序與並發進程

➢ 程序的並發執行的條件

在失去封閉性的條件下，保持可再現性。

读集	写集
• R(pi)={a1,a2,…,am} 表示程序pi所需读取的所有变量的集合	• W(pi)={b1,b2,…,bn} 表示程序pi所需改变的所有变量的集合

例如，若有兩條語句c=a+b和v=c-1，則

讀集為　R(c:=a+b)={a,b}，R(v:=c-1)={c}

寫集為　W(c:=a+b)={c}，W(v:=c-1)={v}

7-1 順序與並發進程

➢ 程序的並發執行的條件

- 若兩個程序**p1**，**p2**滿足如下條件，則能夠保持可再現性，可以並發執行

$$R(p1) \cap W(p2) \cup$$
$$R(p2) \cap W(p1) \cup$$
$$W(p1) \cap W(p2) = \Phi$$

即：**p1**，**p2**的讀集和寫集之間、寫集與寫集之間不能有交集

注：該條件1966年由Bernstein首先提出，稱為Bernstein條件

7-1 順序與並發進程

➢ 程序的並發執行的條件

- 有如下四條語句:
  - S1: a:=x+y
  - S2: b:=z+1
  - S3: c:=a-b
  - S4: w:=c+1

讀集
R(S1)={x,y}，R(S2)={z}，
R(S3)={a,b}，R(S4)={c}

寫集
W(S1)={a}，W(S2)={b}，
W(S3)={c}，W(S4)={w}

S1和S2可以並發執行
$R(S1)\cap W(S2)\cup R(S2)\cap W(S1)\cup W(S1)\cap W(S2)=\Phi$

S1 和S3不能並發執行
$W(S1)\cap R(S3)=\{a\}$

S2和S3不能並發執行
$W(S2)\cap R(S3)=\{b\}$

S3和S4不能並發執行
$W(S3)\cap R(S4)=\{c\}$

7-1 順序與並發進程

➢ 與時間有關的錯誤

- 例：圖書借閱系統　（x：某種書冊數，設當前x=1）

```
終端1:                          終端2:
CYCLE                           CYCLE
   等待借書者;                     等待借書者;
   IF x>=1 Then  <-- 1             IF x>=1 Then  <-- 2
      Begin                           Begin
        x:=x-1;                         x:=x-1;
        借書  <-- 3                     借書  <-- 4
      End                             End
   Else 無書                       Else 無書
End                             End
```

- 若P1執行到①處時被中斷， P2執行到②處時被中斷
- 此時P1和P2都判斷有書，將同一本書借給了兩位讀者，發生了錯誤

7-1 順序與並發進程

➢ 與時間有關的錯誤

- 例：圖書借閱系統　（x：某種書冊數，設當前x=1）

終端1：

```
CYCLE
   等待借書者;
   IF x>=1 Then ←(1)
      Begin
        x:=x-1;
        借書 ←(3)
      End
   Else 無書
```

終端2：

```
CYCLE
   等待借書者;
   IF x>=1 Then ←(2)
      Begin
        x:=x-1;
        借書 ←(4)
      End
   Else 無書
```

- 上述錯誤並不是一定發生的，它與進程的推進速度有關
  - 若P1執行到③處而不是①處時被中斷，則不會發生上述錯誤
- 上述錯誤發生與否與進程P1和P2的推進速度有關，速度是時間的函數，這種錯誤稱為與時間有關的錯誤

7-1 順序與並發進程

➢ 與時間有關的錯誤

- 例：圖書借閱系統　（x：某種書冊數，設當前x=1）

```
終端1:                          終端2:
CYCLE                           CYCLE
   等待借書者;                     等待借書者;
   IF x>=1 Then  <-- 1             IF x>=1 Then  <-- 2
      Begin                           Begin
        x:=x-1;                         x:=x-1;
        借書  <-- 3                     借書  <-- 4
      End                             End
   Else 無書                       Else 無書
End                             End
```

- 一個進程對x的操作僅做了一部分，另一個進程中途插入使得變量x處於一種不確定的狀態，失去了變量x的數據完整性。
  - 錯誤原因之1：進程執行交叉（interleave）；
  - 錯誤原因之2：涉及公共變量(x)。

7-1 順序與並發進程

➢ 與時間有關的錯誤

飛機訂票系統

- 訂票是隨機的，若幹旅客同時訂票，都要訪問同一個**Aj**
- 當**Aj>=1**時認為有餘票，執行減**1**操作後將餘數送回**Aj**
- 實際上**Aj**只做了一次減法，即將同一張票買給了若幹個旅客

```
begin
  按旅客订票要求找到Aj;
  Ri := Aj
  if Ri >=1 then
  begin
    Ri := Ri-1;
    Aj := Ri;
    输出一张票;
  end
  else
    输出“已经售完票”;
end;
```

大綱

1 順序與並發進程

2 進程互斥

- 軟件實現方法

- 硬件實現方法

7-2 進程互斥

➢ 進程互斥定義

兩個或兩個以上的進程，不能同時進入關於同一組共享變量的臨界區域，否則可能發生與時間有關的錯誤，這種現象被稱作進程互斥。

進程互斥是進程之間發生的一種間接性相互作用，這種相互作用是運行進程“感覺不到”的。

意味著互斥是由操作系統實現的

7-2 進程互斥

➢ 與互斥相關的定義

- 共享變量（Shared Variable）
  - 多個進程都需要訪問的變量。
- 臨界區域（Critical Region）
  - 訪問共享變量的程序段。
- 臨界資源（Critical Resource）
  - 一次僅允許一個進程使用的資源

7-2 進程互斥

➢ 進程互斥

- 前面的例子各進程的臨界區：

例：圖書借閱系統　（x：某種書冊數，設當前x=1）

終端1：

```
CYCLE
    等待借書者;
    IF x>=1 Then
       Begin
          x:=x-1;
          借書
       End
    Else 無書
End
```

終端2：

```
CYCLE
    等待借書者;
    IF x>=1 Then
       Begin
          x:=x-1;
          借書
       End
    Else 無書
End
```

7-2 進程互斥

➢ 進程互斥

- 前面的例子各進程的臨界區：

```
begin
   按旅客订票要求找到Aj;
   Ri := Aj
   if Ri >=1 then
   begin
      Ri := Ri-1;
      Aj := Ri;
      输出一张票;
   end
   else
      输出“已经售完票”;
end;
```

7-2 進程互斥

➢ 進程互斥的實現

- 實現進程互斥，可采用軟件或硬件的方法
- Framework 框架

Repeat

entry section　進入區

critical section　臨界區

exit section　退出區

remainder section　餘下部分

Until false

7-2 進程互斥

➢ 進程互斥

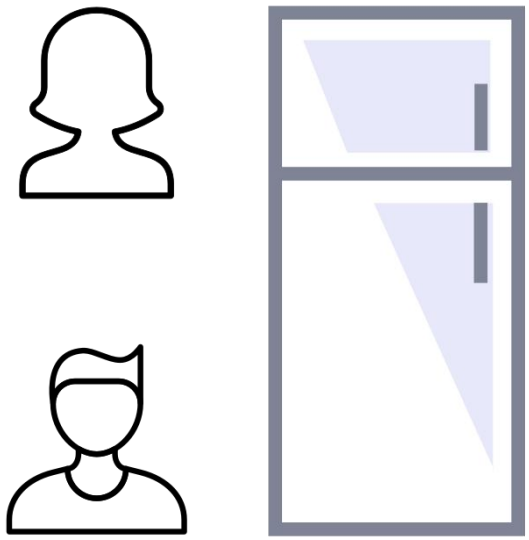
- 為保證正確性和公平性，實現進程互斥應滿足三個原則
  - mutual exclusion (互斥性)：一次只允許一個進程進入關於同一組公共變量的臨界區；
  - Progress (進展性)：當臨界區空閑時，競爭進入臨界區的多個進程在有限時間之內確定下一個進入臨界區的進程；
  - bounded waiting (有限等待)：一個想要進入臨界區的進程在等待有限個進程進入並離開臨界區後獲得進入臨界區的機會

大綱

1. 順序與並發進程
2. 進程互斥
   - 軟件實現方法
   - 硬件實現方法

7-2 進程互斥

➢ 進程互斥的軟件實現方法



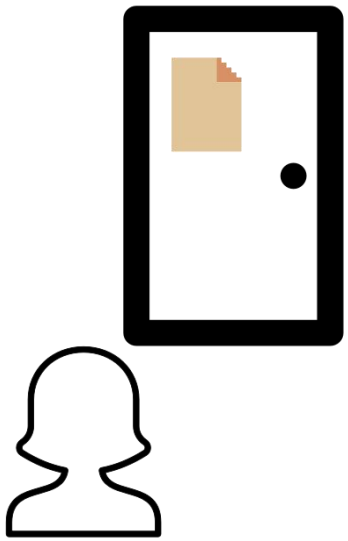
故事背景：
夫妻吵架，不言語交流。

約定：
如果打開冰箱看到沒有面包，那麼就立刻去買面包放進冰箱。

如何能正常補充面包，同時又不多買？

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法一



- ◆ 同一時間只能一個人使用（查看）冰箱，這樣就不會同時看到沒有面包。
- ◆ 使用紙條貼對方門上，告訴對方我正在使用（查看）冰箱，結束後撕掉

此時：

- 紙條 -> 共享資源
- 冰箱 -> 臨界資源
- 查看冰箱的操作 -> 臨界區

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法一

- 用標志位flag[i]標識進程i是否在臨界區內執行

```
var flag :array[2] of boolean; # 紙條
repeat
    while flag[j] do no_op;
    flag[i]:=true; # 貼紙條
    臨界區; /*進程Pi的臨界區代碼段
    flag[i]:=false; # 撕紙條
    剩餘區; /*進程Pi的其它代碼
until false;
```

如果自己門上有對方貼的紙條，就等待

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法一

```
var flag :array[2] of boolean; # 紙條
```

進程i代碼

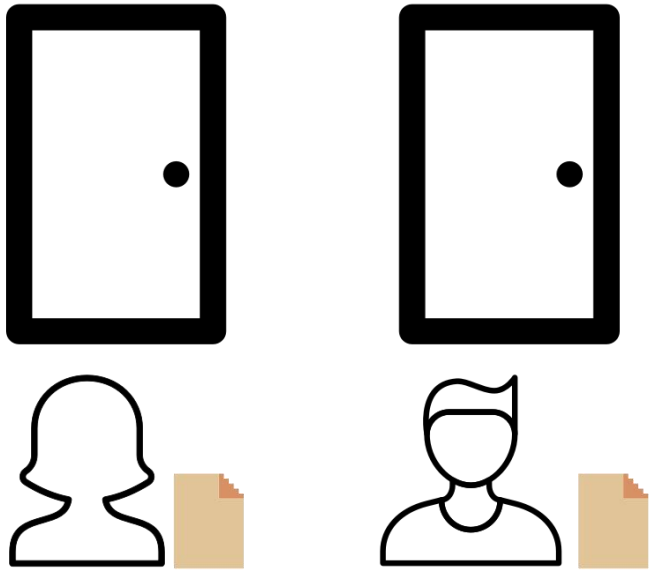
```
repeat
    while flag[j] do no_op;
    flag[i]:=true;
    臨界區；/*進程Pi的臨界區代碼段
    flag[i]:=false;
    剩餘區；/*進程Pi的其它代碼
until false;
```

進程j代碼

```
repeat
    while flag[i] do no_op;
    flag[j]:=true;
    臨界區；/*進程Pj的臨界區代碼段
    flag[j]:=false;
    剩餘區；/*進程Pj的其它代碼
until false;
```

7-2 進程互斥

- 進程互斥的軟件實現方法：解法一



問題：貼紙條的操作是有“時延”的，兩個人可能同時去對方房門貼紙條

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法一

進程i代碼

```
repeat
    while flag[j] do no_op
    flag[i]:=true;
    臨界區; /*進程Pi的臨界區代碼段
    flag[i]:=false;
    剩餘區; /*進程Pi的其它代碼
until false;
```

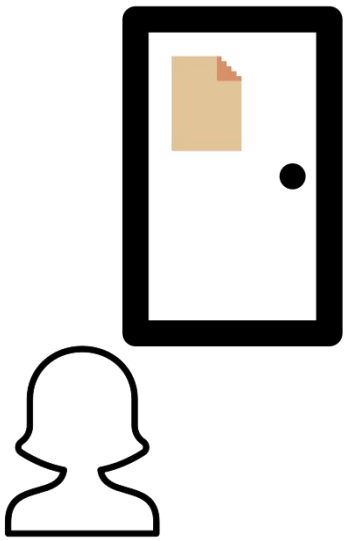
進程j代碼

```
repeat
    while flag[i] do no_op
    flag[j]:=true;
    臨界區; /*進程Pj的臨界區代碼段
    flag[j]:=false;
    剩餘區; /*進程Pj的其它代碼
until false;
```

若兩進程最初flag均為false，則同時進入臨界區，違反了互斥性的原則

7-2 進程互斥

- 進程互斥的軟件實現方法：解法二



- 使用紙條貼“自己門上”告訴對方我想要使用（查看）冰箱了
- 再去查看對方門上是否有紙條，沒有則使用冰箱

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法二

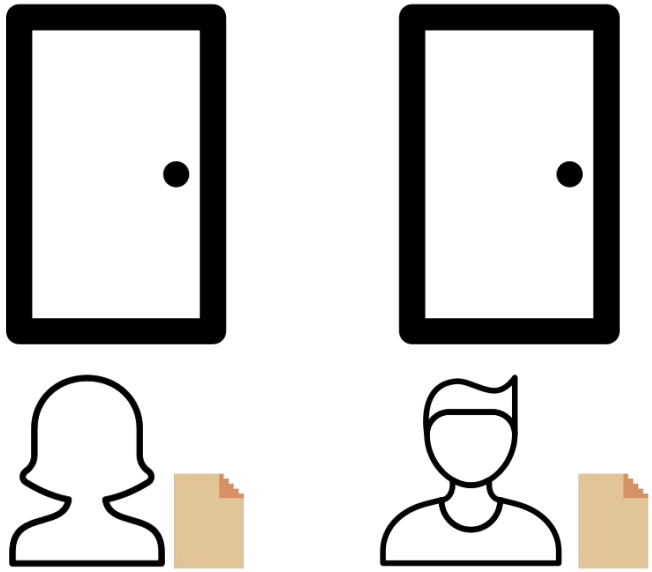
- 用flag[i]表示進程i想要進入臨界區

```
var flag :array[2] of boolean; # 紙條
repeat
    flag[i]:=true;
    while flag[j] do no_op
        臨界區;      /*進程Pi的臨界區代碼段
    flag[i]:=false;
        剩餘區;      /*進程Pi的其它代碼
until false;
```

先申明“想進入”臨界區，再查看對方門上有沒有紙條

7-2 進程互斥

- 進程互斥的軟件實現方法：解法二



問題：兩個人可能同時在自己房門上貼紙條，造成互相等待

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法二

進程i代碼

```
repeat
    flag[i]:=true;
    while flag[j] do no_op
    臨界區;      /*進
    flag[i]:=false;
    剩餘區;      /*進
until false;
```

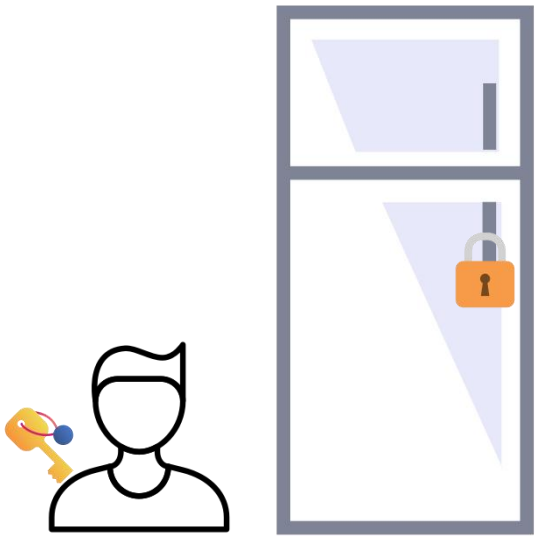
進程j代碼

```
repeat
    flag[j]:=true;
    while flag[i] do no_op
                 /*進程Pj的臨界區代碼段
                 lse;
                 /*進程Pj的其它代碼
until false;
```

若兩進程flag同時為true，則互相謙讓，違反了進展性、有限等待的原則

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法三



- ◆ 給冰箱上鎖，若冰箱裏沒有面包，就把鑰匙帶走（不讓對方查看冰箱），然後去買面包
- ◆ 放面包後，把鑰匙給對方

此時：

- ● 鑰匙只能被一個人獲取，且只能輪流获取

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法三

```
var turn = 0;
turn = i; # 指定鑰匙先放誰這
```

進程i代碼

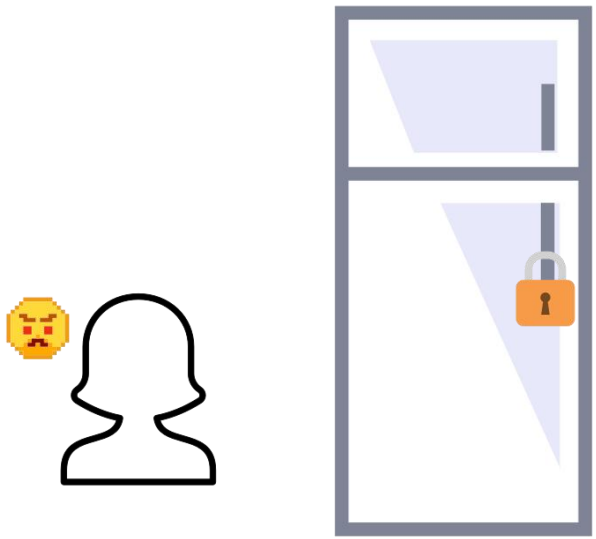
```
repeat
    while turn != i do no_op;
    臨界區;
    turn:=j; # 將鑰匙讓給j
    剩餘區;
until false;
```

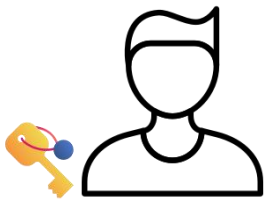
進程j代碼

```
repeat
    while turn != j do no_op;
    臨界區;
    turn:=i; # 將臨界區讓給i
    剩餘區;
until false;
```

7-2 進程互斥

- 進程互斥的軟件實現方法：解法三





問題：開啟冰箱時機與主觀意願無關，收到鑰匙後就得開啟冰箱，再送回鑰匙給對方。屬於強制性交替使用。

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法三

```
var turn = 0;
turn = i; # 指定鑰匙先放誰這
```

進程i代碼

```
repeat
    while turn != i do no_op;
    臨界區;
    turn:=j; # 將臨界區讓給j
    剩餘區;
until
```

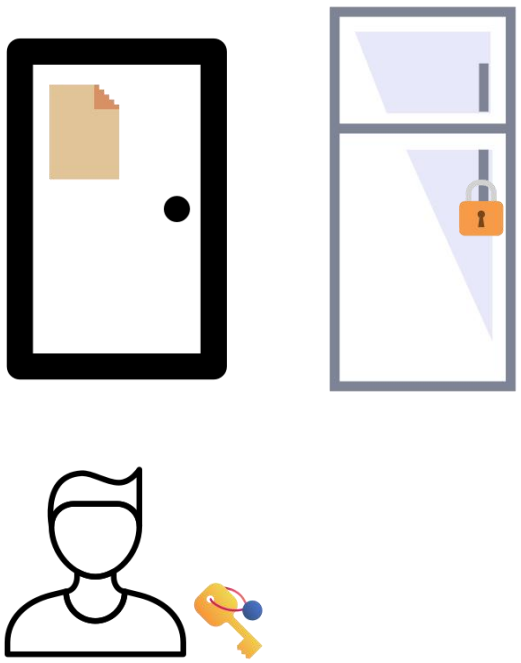
進程j代碼

```
repeat
    while turn != j do no_op;
    臨界區;
    turn:=i; # 將臨界區讓給i
    剩餘區;
```

強制兩個進程交替使用臨界區，將導致某個進程無論是否想要進入臨界區，都必須等待另一個進程退出臨界區才能繼續運行。

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法四



結合紙條、鑰匙兩個策略（Dekker算法，1965年，第一個正確方案）。

- 要使用冰箱時給自己門上貼紙條
- 如果都貼了紙條，有鑰匙的才能開冰箱，沒有鑰匙的先撕掉紙條
- 使用完冰箱後，交鑰匙給對方

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法四

```
var flag:array[2] of boolean; # 指示誰貼了紙條(想進入臨界區)
turn = i; # 指示誰有鑰匙(輪到誰進入臨界區)
```

進程i代碼

```
repeat
  flag[i]=true;   # 貼紙條
  while flag[j] do  # 如果對方門上有紙條
     if (turn==j){ # 如果鑰匙在對方那
        flag[i]=false; # 撕掉自己紙條
        while turn==j do no_op; # 等待鑰匙
        flag[i]=true; # 拿到鑰匙後貼紙條 }
  臨界區 # 開冰箱
  turn=j; # 鑰匙給對方
  flag[i]=false; # 撕掉自己紙條
  其餘代碼
until false;
```

讓對方能正常執行下去

進程j代碼

```
repeat
  flag[j]=true;
  while flag[i] do
     if (turn==i){
        flag[j]=false;
        while turn==i do no_op;
        flag[j]=true;}
  臨界區
  turn=i;
  flag[j]=false;
  其餘代碼
until false;
```

turn = j; # 設鑰匙最初在進程j這裏

進程i代碼

```
repeat
  flag[i]=true; ①
  while flag[j] do
     if (turn==j){
        flag[i]=false;
        while turn==j do no_op; ③
        flag[i]=true;}
  臨界區 ⑥
  turn=j;
  flag[i]=false; ⑦
  其餘代碼
until false;
```

進程j代碼

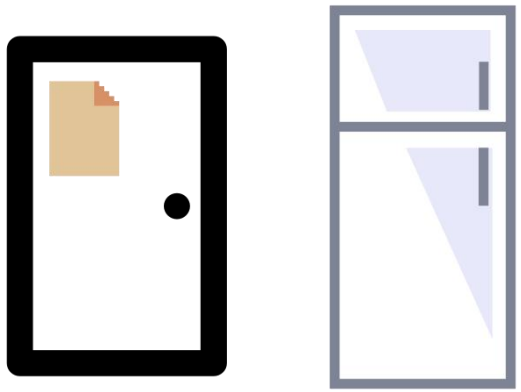
```
repeat
  flag[j]=true; ②
  while flag[i] do
     if (turn==i){
        flag[j]=false;
        while turn==i do no_op;
        flag[j]=true;}
  臨界區 ④
  turn=i;
  flag[j]=false; ⑤
  其餘代碼
until false;
```

➢ 該算法是否能滿足需求？

時刻	1	2	3	4	5	6	7	8
進程i	flag[i]:=true		flag[i]=false While等待			進入臨界區	turn=i flag[j]=false	
進程j		flag[j]:=true		進入臨界區	turn=i flag[j]=false			

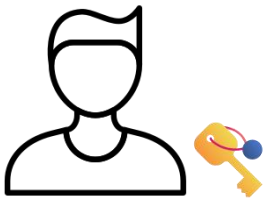
7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法五



更簡便的，1981年G.L.Peterson提出

- 要使用冰箱時給自己門上貼紙條
- 把鑰匙給對方
- 如果對方“貼紙條”且“有鑰匙”，則等待；否則開冰箱（冰箱無鎖）



鑰匙此時的作用相當於“保險”，若對方有開冰箱意圖，則謙讓

7-2 進程互斥

➢ 進程互斥的軟件實現方法：解法五

```
var flag:array[2] of boolean; # 指示誰貼了紙條(想進入臨界區)
turn = i; # 指示誰有鑰匙(輪到誰進入臨界區)
```

進程i代碼

```
repeat
  flag[i]:=true; # 貼紙條
  turn:=j; # 把鑰匙給對方
  while (flag[j] and turn=j) do no_op;
  臨界區;
  flag[i]:=false;
  剩餘區;
until false;
```

進程j代碼

```
repeat
  flag[j]:=true;
  turn:=i;
  while (flag[i] and turn=i) do no_op;
  臨界區;
  flag[j]:=false;
  剩餘區;
until false;
```

```
var flag:array[2] of boolean; # 指示誰想進入臨界區
turn = i; # 指定誰進入臨界區
```

進程i代碼

```
repeat
  flag[i]:=true; # 貼紙條
  turn:=j; # 把鑰匙給對方 ①
  while (flag[j] and turn=j) do no_op;③
  臨界區; ⑤
  flag[i]:=false; ⑦
  剩餘區;
until false;
```

進程j代碼

```
repeat
  flag[j]:=true; ②
  turn:=i; ④
  while (flag[i] and turn=i) do no_op;⑥
  臨界區; ⑧
  flag[j]:=false;
  剩餘區;
until false;
```

➢ 該算法是否能滿足需求？

時刻	1	2	3	4	5	6	7	8
進程i	flag[i]:=true turn:=j		While等待		進入臨界區		flag[i]:=false	
進程j		flag[j]:=true		turn:=i		While等待		進入臨界區

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程

- 基本思想：顧客在面包店中購買面包時的排隊原理
  - 顧客進入面包店前，首先抓一個號
  - 按號碼由小到大的次序依次進入面包店購買面包
  - 面包店發放的號碼由小到大，但是兩個或兩個以上的顧客有可能得到相同的號碼（所抓號碼不需要互斥）
  - 如果多個顧客抓到相同的號碼，按照顧客名字的字典次序進行排序（假定顧客沒有重名）

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程

- **Bakery算法（面包店算法）**
  - 進入臨界區之前，進程接收一個數字
  - 得到的數字最小的進入臨界區
  - 如果進程Pi和Pj收到相同的數字，那麼如果i< j，Pi先進入臨界區，否則Pj先進入臨界區
  - 編號方案總是按照枚舉的增加順序生成數字

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

- 算法需要兩個數據結構:

```
int choosing[n];
int number[n];
```

- 前者表示進程是否正在抓號，正在抓號的進程choosing[i]為1，否則為0，初值均為0
- 後者記錄各進程所抓到的號碼
  - number[i]為0，則Pi沒有抓號
  - number[i]不為0，其正整數值為進程Pi所抓到的號碼，初值均為0

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

- 定義
  - (a,b) < (c,d):

    如果 a<c or (a=c and b<d)
  - max(a_0,…,a_{n-1}) :

    其值為一正整數k, 對於所有i, 0≤i≤n-1, k≥a_i

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

```
do{
    choosing[i]=1;  # 進程i准備抓號
    number[i]=max{number[0],number[1],...,number[n-1]}+1; # 抓號
    choosing[i]=0;  # 抓號完成
    for(j=0;j< n;j++){
        while (choosing[j])  no op; # 若進程j在抓號則等待
        while ((number[j]!=0)&&(number[j],j)< (number[i],i)) no_op;
        # 進程j有號，且優先級比進程i高，則等待進程j離開
    };
    臨界區
    number[i]=0; # 進程i離開
    其餘部分
}while(1);
```

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

```
do{
    choosing[i]=1;  # 進程i准備抓號
    number[i]=max{number[0],number[1],...,number[n-1]}+1; # 抓號
    choosing[i]=0;  # 抓號完成
    for(j=0; j< n; j++){
        while (choosing[j])  no op; # 若進程j在抓號則等待
        while ((number[j]!=0)&&(number[j],j)< (number[i],i)) no_op;
        # 進程j有號，且優先級比進程i高，則等待進程j離開
    };
    臨界區
    number[i]=0; # 進程i離開
    其餘部分
}while(1);
```

choosing的作用：
“軟互斥”，減少不同進程拿同一個號的概率

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

```
do{
    choosing[i]=1;  # 進程i准備抓號
    number[i]=max{number[0],number[1],...,number[n-1]}+1; # 抓號
    choosing[i]=0;  # 抓號完成
    for(j=0;j< n;j++){
        while (choosing[j])  no op; # 若進程j在抓號則等待
        while ((number[j]!=0)&&(number[j],j)< (number[i],i)) no_op;
        # 進程j有號，且優先級比進程i高，則等待進程j離開
    };
    臨界區
    number[i]=0; # 進程i離開
    其餘部分
}while(1);
```

number的作用：
“硬互斥”，強制進程按約定的次序進入臨界區

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

- 互斥性：
  - 若Pi在臨界區中，Pk剛抓到號碼，則有 (number[i], i)< (number[k], k)
  - 如Pi正在臨界區中，Pk試圖進入該臨界區，則它在 while ((number[i]!=0)&&(number[i], i)< (number[k], k)) 循環處等待，直到Pi離開該臨界區。

7-2 進程互斥

➢ 進程互斥的軟件實現方法：多進程Bakery算法

- 有限等待性：競爭進程按FIFO的次序進入臨界區，進程個數有限，進程不會無限期地等待進入臨界區

- 進展性：當多個進程競爭進入臨界區時，下述條件之一成立：
  - 一個進程抓到最小號碼：獲准進入臨界區
  - 若幹進程抓到最小號碼：編號最小獲准進入臨界區

小結

➢ 進程互斥的軟件實現方法

- 解法一：引入了紙條通信的方法，但未保證互斥性
- 解法二：改進了紙條使用方法，但未保證進展性、有限等待
- 解法三：引入了“鎖”機制實現互斥，但未考慮程序主觀意願

✓ 解法四（Dekker）：同時采用紙條和鎖，第一個正確方案

✓ 解法五（Peterson）：更簡潔的Dekker方案

✓ Bakery算法：受面包店排隊啟發的多進程互斥方案

大綱

1 順序與並發進程

2 進程互斥

- 軟件實現方法

- 硬件實現方法

7-2 進程互斥

➢ 進程互斥的硬件實現方法

思考：為什麼需要實現互斥?

答：程序的並發性導致程序每次運行結果可能不一致，為保證結果一致性，需實現對共享資源訪問的時候，保證互斥性。

7-2 進程互斥

➢ 進程互斥的硬件實現方法

思考：如何實現程序的並發性?

答：中斷，CPU響應中斷後，操作系統進行現場保護及上下文切換。

7-2 進程互斥

➢ 進程互斥的硬件實現方法——禁用硬件中斷

進程進入臨界區前執行“關中斷”指令，離開臨界區時執行“開中斷”

```
do{
    關中斷
    臨界區
    開中斷
    其餘部分
}while(1)
```

7-2 進程互斥

➢ 進程互斥的硬件實現方法——禁用硬件中斷

◘ 優點:

- 實現效率高
- 沒有忙式等待問題（While）
- 簡單易行

◘ 缺點:

- 開關中斷只在單CPU系統中有效
- 影響並發性，甚至可能導致操作系統奔潰，因此需謹慎使用

7-2 進程互斥

- 進程互斥的硬件實現方法——基於原子操作

  為了執行某些重要指令時，不會被中斷打斷，硬件提供了一些“原子操作”

- 原子操作：指不會被進程/線程調度機制打斷的操作
  - 例如中斷禁用
- 操作體系基於原子操作提供了更高級的編程抽象
  - 例如鎖、信號量

7-2 進程互斥

➢ 原子操作

☐ Test and Set

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;  # 從內存中讀取值
    *target = TRUE;  # 將內存值設為1/True
    return rv;  # 返回該值（*target）是否為1/True
}
```

7-2 進程互斥

➢ 原子操作

□ Exchange

```
void Exchange (boolean *a, boolean *b)
{
    boolean temp = *a;  # 從內存中讀取值
    *a = *b;
    *b = temp;  # 交換值
}
```

7-2 進程互斥

- “鎖”機制

```
class Lock {
    int value = 0;
}
```

```
Lock::Acquire() {
    while(test_and_set(value));
}
```

```
Lock::Release() {
    value = 0;
}
```

如果鎖被釋放(空閑)，那麼test_and_set讀取0並將值設置為1 → 鎖被設置為忙並且需要等待完成

如果鎖處於忙狀態，那麼test_and_set讀取1並將值設置為1 → 不改變鎖的狀態並且需要循環等待

7-2 進程互斥

- “鎖”機制

```
class Lock {
    int value = 0;
}
```

```
Lock::Acquire() {
    while(test_and_set(value));
}
```

```
Lock::Release() {
    value = 0;
}
```

```
do{
    紙條部分
    Lock. Acquire()
    臨界區
    Lock. Release()
    其餘部分
}while(1);
```

7-2 進程互斥

- “鎖”機制

```
Lock::Acquire() {
    while(test_and_set(value));
}

Lock::Release() {
    value = 0;
}
```

→ 忙等待，無意義消耗cpu時間

7-2 進程互斥

- “鎖”機制的改進——結合調度算法

```
class Lock {
    int value = 0;
    WaitQueue q;
}
```

如果鎖處於忙狀態，那麼將申請鎖的進程加入等待隊列，采用調度算法暫時移出cpu

```
Lock::Acquire() {
    while(test_and_set(value)){
        add this TCB/PCB to q;
        schedule();
    }
}
```

```
Lock::Release() {
    value = 0;
    remove 1 TCB/PCB t from q;
    wakeup(t)
}
```

7-2 進程互斥

- “鎖”機制的改進——結合調度算法

```
class Lock {
    int value = 0;
    WaitQueue q;
}
```

釋放鎖的同時，喚醒一個處於等待隊列中的進程

```
Lock::Acquire() {
    while(test_and_set(value)){
        add this TCB/PCB to q;
        schedule();
    }
}
```

```
Lock::Release() {
    value = 0;
    remove 1 TCB/PCB t from q;
    wakeup(t)
}
```

7-2 進程互斥

- 利用Exchange操作實現“鎖”

```
do{
      紙條部分
      key = 1;
      while (key == 1) do exchange(*lock,*key);
      臨界區
      lock = 0;
    其餘部分
}while(1);
```

小結

➢ 進程互斥的硬件實現方法——禁用硬件中斷

- 通過禁用中斷的方式，禁止程序的並發性，使訪問臨界區時，始終只有一個程序運行。
- 只適用於單cpu，且需謹慎使用禁用中斷。

➢ 進程互斥的硬件實現方法——基於原子操作

- test_and_set指令和exchange指令是原子的，不可中斷的。
- test_and_set將內存中一個單元的內容取出，再送一個新值。
- exchange交換內存兩個單元的內容。

感謝觀賞

Thank you for listening.

主講人｜ 澳門城市大學

City University of Macau

眭相傑 助理教授

Sui Xiangjie Assistant Professor