上节课重点：

1. 程序的执行有两种方式：顺序执行和并发执行。
并发执行的条件：达到封闭性和可再现性（正确性的要求）。

2. 进程的描述：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。处理机、存储器和外设等资源的分配和回收的基本单位。
特征：动态性、并发性、独立性、异步性和结构化。

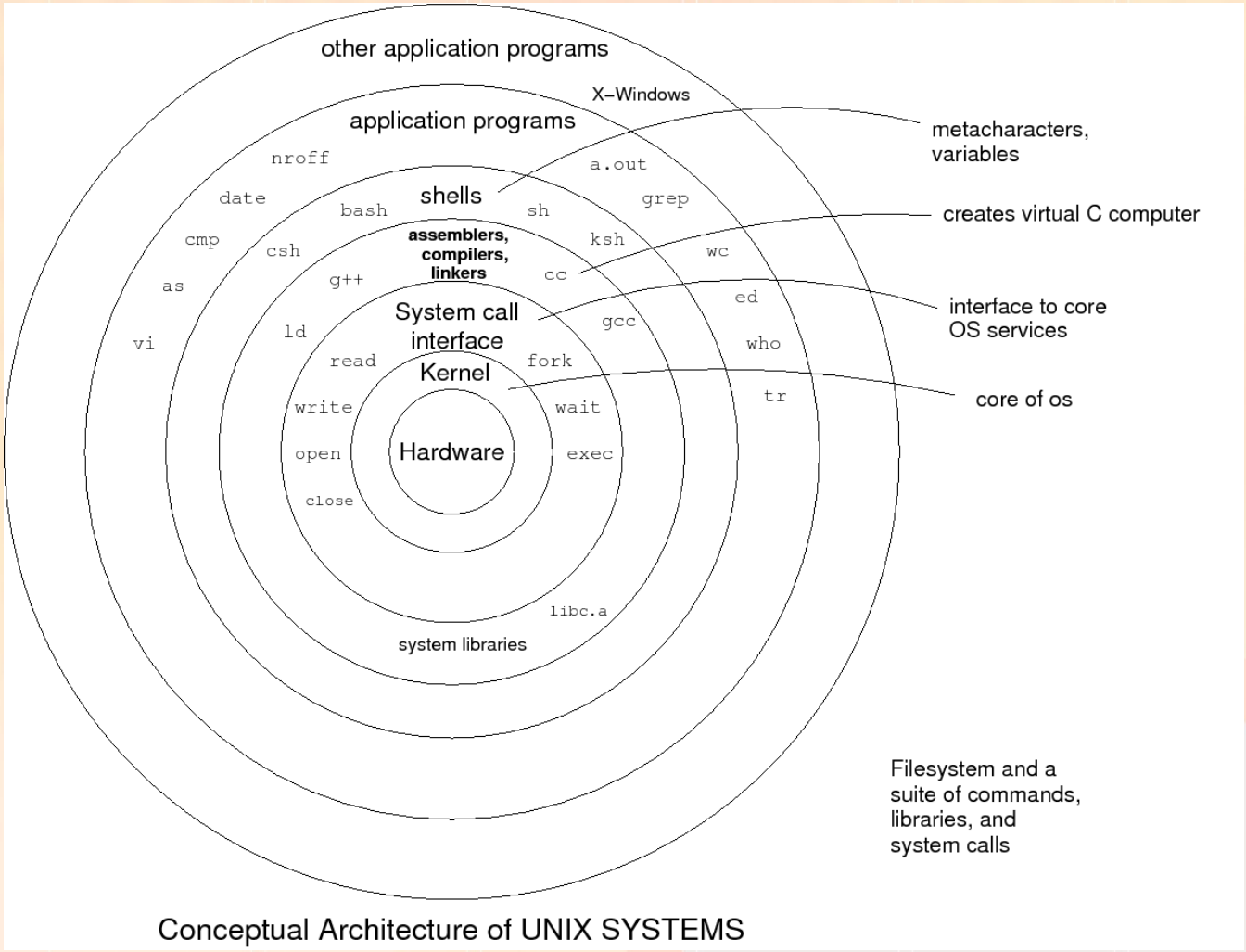
3. 进程控制块（ PCB ）：进程存在的唯一标志。

4. 进程的基本状态：就绪态、执行态、阻塞态。

2.3 进程控制

- 进程管理中最基本功能是进程控制
- 进程的生命周期：
  - 进程创建
  - 进程运行
  - 进程等待
  - 进程唤醒
  - 进程终止
- 进程控制任务：
  - 进程的创建、终止、进程状态的转变等
- 进程控制一般由 OS 内核的原语（ primitive ）来实现。
  - 原语：由若干条指令构成的“原子操作 (atomic operation)” 过程。
  - 许多系统调用是原语。但并不是所有的系统调用都是原语

2.3.1 操作系统内核

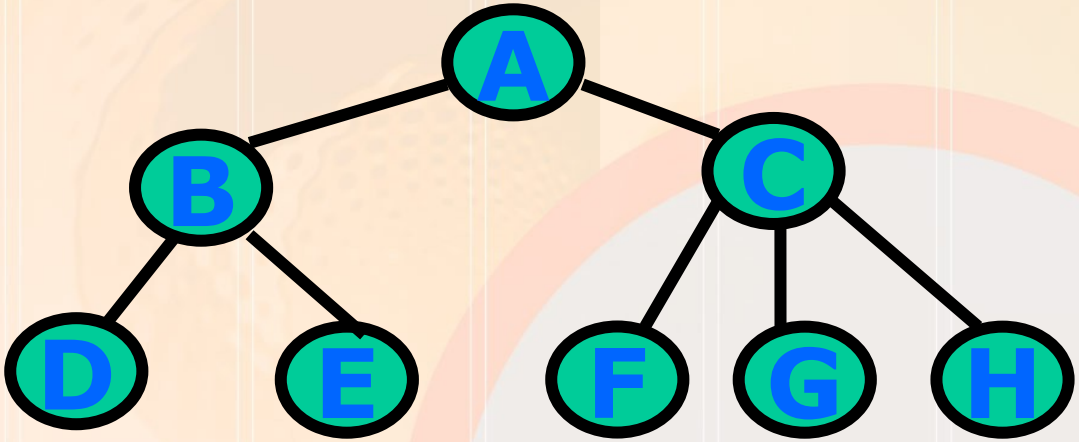


Conceptual Architecture of UNIX SYSTEMS

OS 内核功能：
1. 支撑功能；
2. 资源管理功能。

2.3.2 进程的创建（ Creation of process ）

- 进程图（ Process Graph ）
  - 树状结构，父子进程关系。
  - 子进程可以继承父进程所拥有的资源，如打开文件、缓冲区等。当子进程被撤销应将继承的资源归还给父进程。撤销父进程也必须同时撤销所有的子进程。
  - PCB 中设置了家族关系表项。



2.3.2 进程的创建（ Creation of process ）

定义：操作系统发现要求创建新进程的事件后，调用进程创建原语 Creat() 创建新进程。

引起创建进程的事件

1. **系统初始化**
   - ① 分时系统中用户登录
   - ② 批处理系统中作业调度

   } 由系统内核创建

2. **提供服务**
   - ➢ 用户请求创建进程

3. **应用请求**
   - ➢ 正在运行的进程执行了创建进程的系统调用

   → 由应用程序自身创建

2.3.2 进程的创建（ Creation of process ）

- 原语 CREAT （ ）按下述步骤创建一个新进程：
  - 申请空白 PCB 。
  - 为新进程分配资源：为代码、数据、用户栈分配空间。
  - 初始化 PCB ：初始化标识信息、处理机状态（ PC 、 SP) 、处理机控制信息（进程状态、优先级）。
  - 将新进程插入就绪队列。

- UNIX ：
  - fork 系统调用：创建一个子（新）进程。精确复制父进程
  - exec 系统调用：在 fork 之后执行，用新的代码替换原父进程的代码。

```
void  main (int argc,  char *argv[ ])
{ int  pid;
  pid = fork( );
  if ( pid < 0) {
        fprint ( stderr, “fork failed”);
        exit (-1);
   }
  else  if ( pid == 0) {
            execlp (“ /bin/ls”, “ls”, NULL);
   }
  else {
            wait ( NULL);
            printf (“ child complete”);
            exit (0);
   }
}
```

- **Windows : CreateProcess**

```
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

```
int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
     "C:\\WINDOWS\\system32\\mspaint.exe", // command line
     NULL, // don't inherit process handle
     NULL, // don't inherit thread handle
     FALSE, // disable handle inheritance
     0, // no creation flags
     NULL, // use parent's environment block
     NULL, // use parent's existing directory
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

2.3.3 进程的终止

引起进程终止的事件

- 正常结束 UNIX: exit Windows: ExitProcess
- 异常结束
  - ① 越界错误。②保护错。
  - ③ 非法指令。④特权指令错。
  - ⑤ 运行超时。⑥等待超时
  - ⑦ 算术运算错。⑧ I/O 故障。
- 外界干预 Kill
  - ① 操作员或 os 干预。
  - ② 被父进程终止
  - ③ 父进程终止

- **进程的终止过程**
  - 从 PCB 集合中检索出该进程的 PCB ，从中读出该进程的状态。
  - 若处于执行状态，终止该进程的执行，并置调度标志为真，重新调度。
  - 若有子孙进程，将所有子孙进程终止。
  - 将进程全部资源归还其父进程或系统：释放内外存空间、关闭所有打开文件、释放当前目录、释放共享内存段和各种锁定 lock 。
  - 将其 PCB 从所在队列（或链表）中移出。

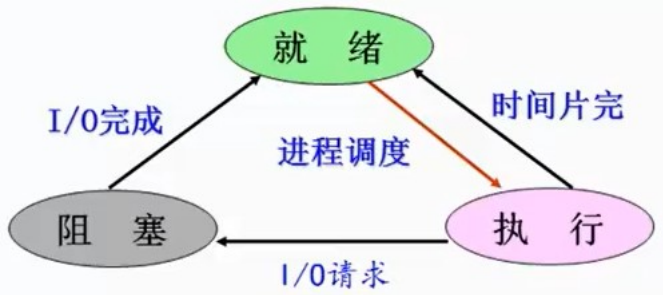
unix 进程终止过程

- 进程执行完最后一条语句，请求操作系统删除进程 (通过执行 exit 系统调用) 。
  - 将子进程运行数据传递给父进程 (通过 wait 系统调用) 。
  - 回收的系统资源由操作系统再另行分配
- 父进程也可以终止子进程的执行 (通过 abort 系统调用) 。原因：
  - 子进程超额使用资源 .
  - 分配给子进程执行的任务不再需要执行 .
  - 父进程退出。
    - 如果父进程终止，操作系统不再允许子进程继续执行。

2.3.4 进程的阻塞与唤醒

引起阻塞和唤醒的事件

- 请求系统服务
- 启动某种操作
- 新数据尚未到
- 无新工作可做

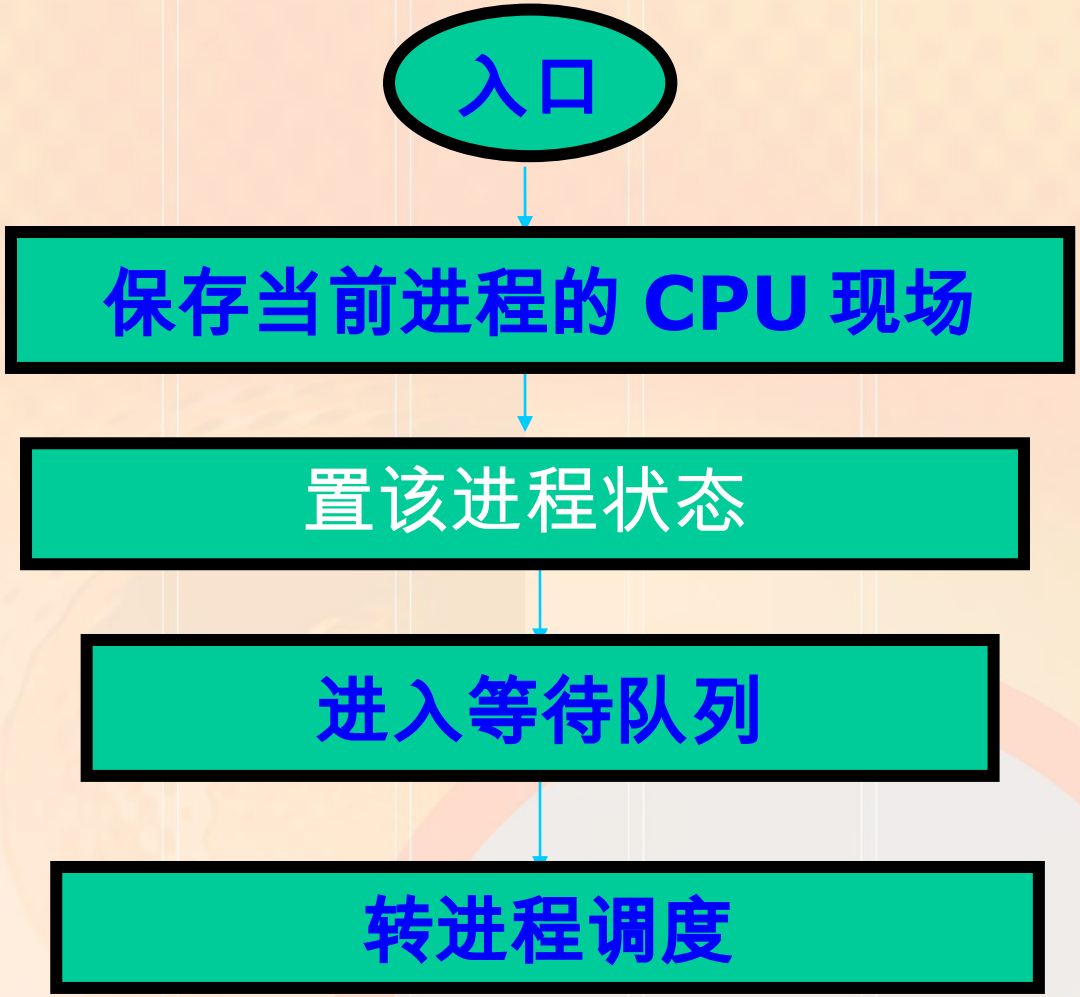


2.3.4 进程的阻塞与唤醒

- **进程阻塞过程**
  - 正在执行的进程，发生上述事件时，**自身调用**有关阻塞原语，进入等待队列。进程的主动性行为。
  - 进程 PCB 中的**状态** 由运行态变为阻塞态。
  - 引起**处理机调度**。
  - UNIX ：
    - sleep 将在指定的时间 seconds 内阻塞本进程。调用格式为：unsigned sleep(unsignedseconds) ，返回值为实际的挂起时间。
    - pause 阻塞本进程以等待信号，接收到信号后恢复执行。当接收到终止进程信号时，该调用不再返回。其调用格式为： int pause(void) 。
    - wait 阻塞本进程以等待子进程的结束，子进程结束时返回。当父进程创建多个子进程且已有子进程退出时，父进程中 wait 函数在第一个子进程结束时返回。

进程阻塞过程

由阻塞原语 BLOCK 完成



- **进程唤醒过程**
  - **唤醒原因：等待的事件到达。**
  - **由阻塞队列转入就绪队列。进程由阻塞态变为就绪态 .**
  - **方法：其他有关进程（例如用完并释放了该 I/O 设备的进程）发送信号到某个或一组进程。**
  - **UNIX: kill**
    - **kill 可发送信号 sig 到某个或一组进程 pid 。其调用格式为： int kill(pid_tpid ， intsig)**

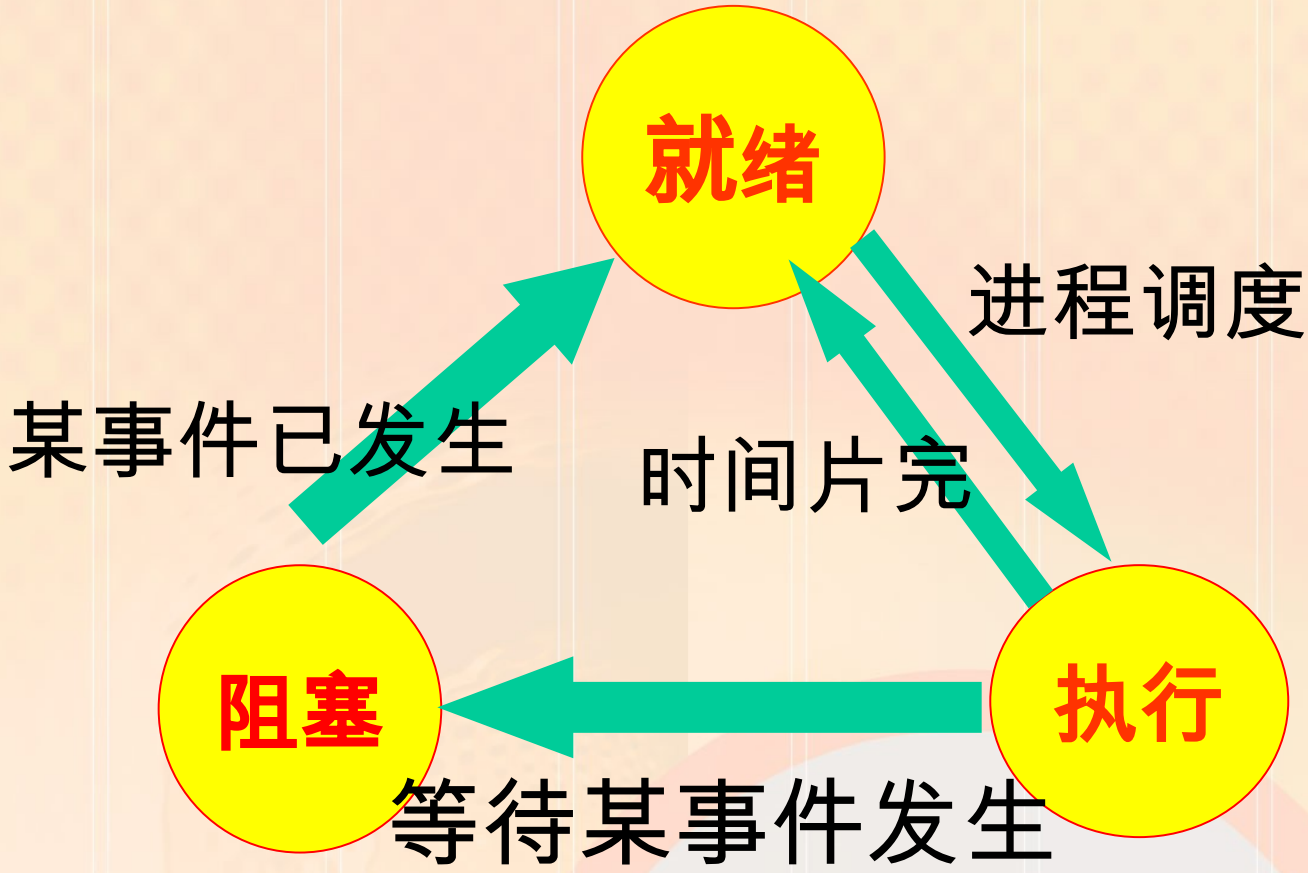
进程唤醒过程

由唤醒原语 WAKEUP 完成

2.3.4 进程的阻塞与唤醒

- **BLOCK 和 WAKEUP 是一队作用相反的原语。**
- **如果在某进程中调用了阻塞原语，则必须在与之相合作的另一进程中或其他相关的进程中，安排唤醒原语，以能唤醒阻塞进程；否则，被阻塞进程将会因不能被唤醒而长久地处于阻塞状态，从而再无机会继续运行。**

进程的状态



2.3.5 进程的挂起与激活

- 挂起引入原因：

（1）**终端用户请求**

- 终端用户发现自己运行的程序有问题，希望暂停自己程序的执行

（2）**父进程请求**

- 父进程希望挂起自己的某个子进程，以便考察和修改该子进程

（3）**负荷调节需要**

- 高优先级的进程要执行，而内存不空，系统将低优先级进程对换至外存。
- 提高处理机效率：就绪进程表为空时，要提交新进程，而此时内存不空，需挂起阻塞态的进程
- 为运行进程提供足够内存（对换及调整负荷）：资源紧张时，暂停某些进程，如：实时任务执行，而内存紧张

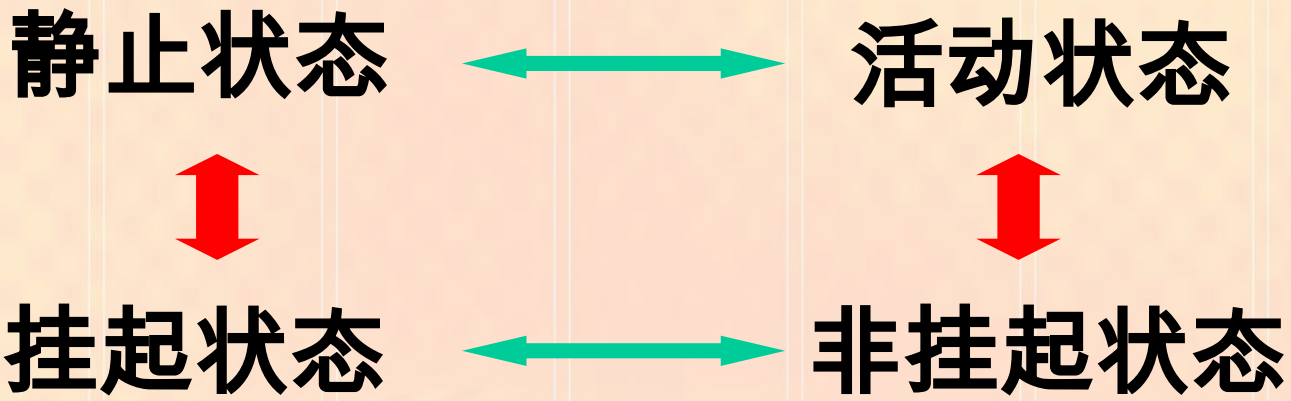
（4）**操作系统的需要**

- 操作系统需要挂起某些进程以便检查运行中的资源使用情况或进行记账。

2.3.5 进程的挂起与激活

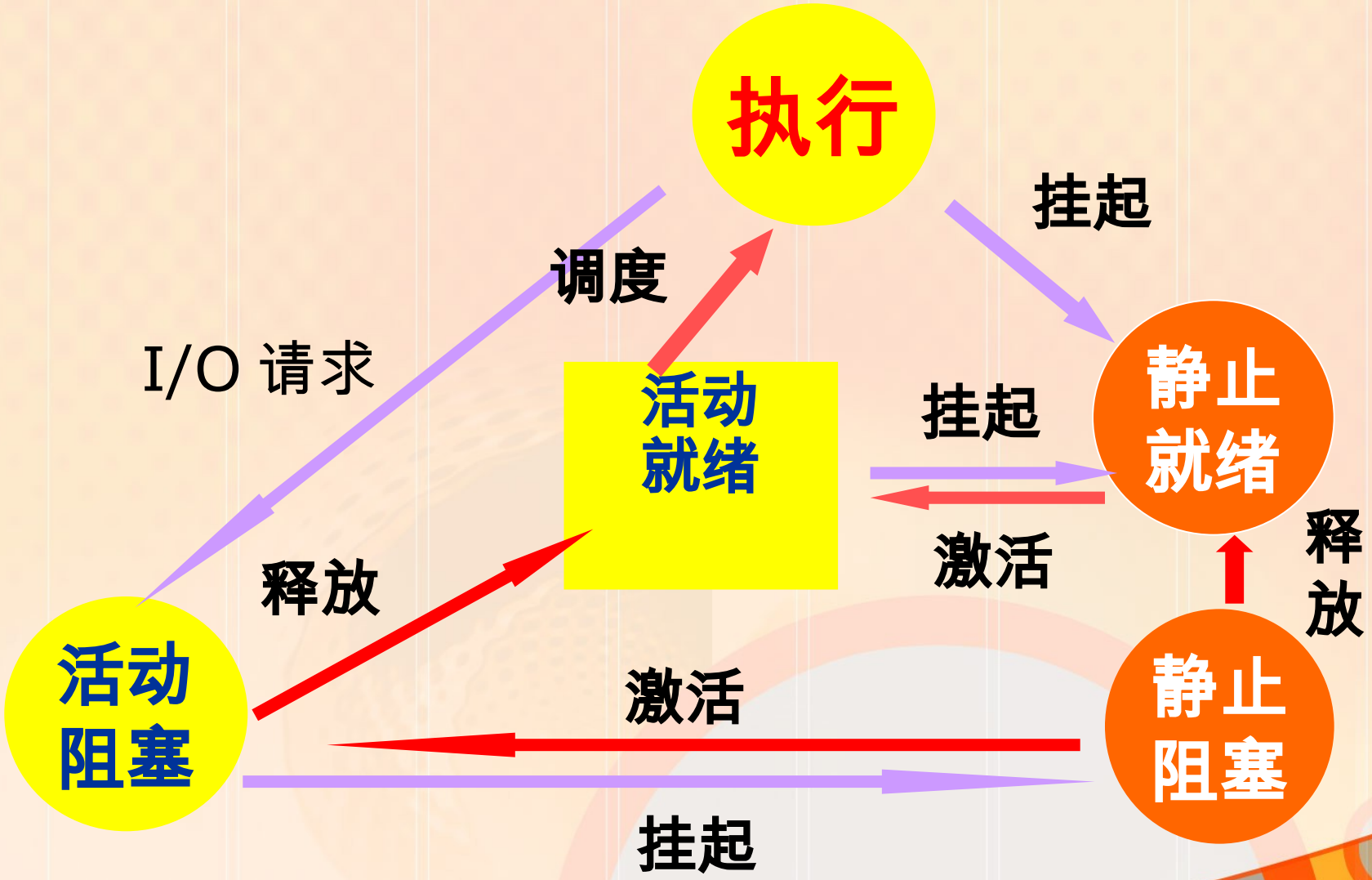
- 进程的**挂起**
  - 目的：合理且充分地**利用系统资源**。进程在挂起状态时，进程没有占用内存空间，仅映像在磁盘上。
  - 执行过程：挂起原语： SUSPEND（ ）
  - 挂起原语的执行过程：
    - 从内存调到外存，改变进程的状态。（ PCB 不调）
    - 若处于**活动就绪状态**，改为**静止就绪**；
    - 若处于**活动阻塞状态**，则改为**静止阻塞**；
    - 若正在执行，则转向调度程序重新调度。
- 进程的**激活**：激活原语 active()
  - 原因：父进程或用户进程请求，或内存已有足够空间
  - 执行过程：从外存调入内存，改变进程的状态
  - 可能也会引起处理机调度

挂起引起的状态转换

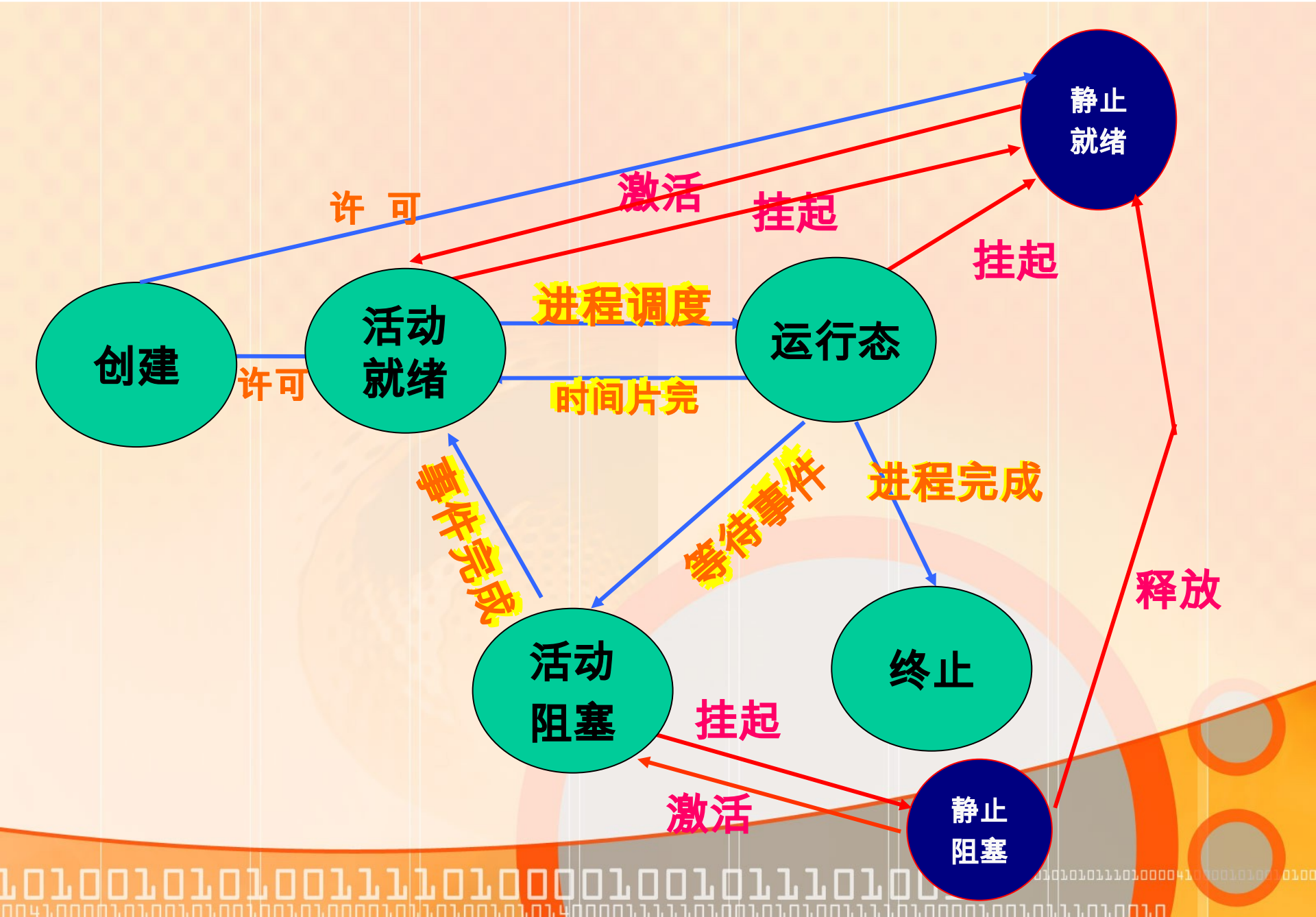


- 引入的新状态
  - 1) 就绪挂起（静止就绪）：进程在外存，但只要进入内存，即可运行；
  - 2) 阻塞挂起状态（静止阻塞）：进程在外存并等待某事件的出现。

有挂起状态的进程状态图



创建状态和终止状态


静止
就绪
许 可
激活
挂起
挂起
创建
许可
活动
就绪
进程调度
运行态
时间片完
事件完成
等待事件
进程完成
释放
活动
阻塞
终止
挂起
激活
静止
阻塞

- **什么是进程同步？**

进程具有异步性的特征。异步性是指，各并发执行的进程以各自独立的、不可预知的速度向前推进。

操作系统要提供“进程同步机制”来解决异步问题。

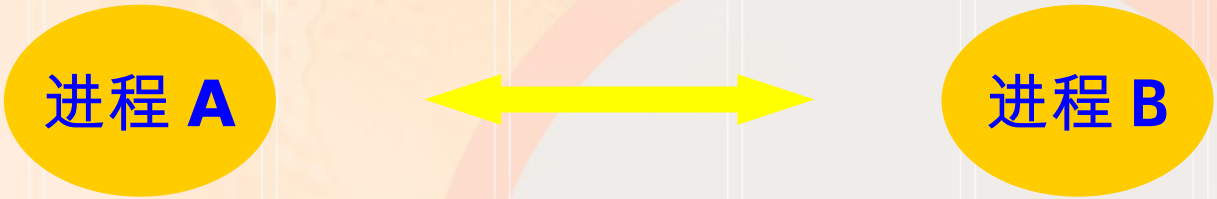
进程 A ：
1. **从缓冲区中读数据**
2. **数据预处理**
3. **将预处理完的数据写回缓冲区**
4. **进行其他任务**

内存缓冲区

进程 B ：
1. **进行准备工作**
2. **从缓冲区中读数据**
3. **数据的后续处理**

- 由于多进程在系统中的并发执行，进程之间存在 2 种制约关系：
  - 间接制约：进程间由于共享某种系统资源，而形成的相互制约。

  - 直接制约：进程间由于相互合作而形成的相互制约



进程的两大关系

进程的“并发”需要“共享”的支持。各个并发执行的进程不可避免地需要共享一些系统资源（比如内存、打印机、摄像头、只读数据、 CPU 多核计算、数据库读操作）。存在两种资源共享方式：

- **互斥**

  由于共享资源所要求的排他性，进程间要相互竞争，以获得这些资源的使用权。

- **同时**

  多个进程中发生的事件存在某种时序关系，必须协同工作、相互配合，以共同完成一项任务。

临界资源：一个时间段内只允许一个进程使用的资源称为临界资源。许多物理设备（摄像机、打印机）都属于临界资源。此外还有很多变量、数据、内存缓冲区等都属于临界资源。

有限缓冲区的生产者与消费者问题

例如：生产者消费者问题（协作进程的范例）

- **生产者进程生产的信息由消费者进程消费。**
  - **无限缓冲区：缓冲区大小无限制。**
  - **有限缓冲区：缓冲区大小是有限且固定的。**
- **有限缓冲区实现时要注意：**
  - **缓冲区中无内容，消费者等待；**
  - **缓冲区满，生产者等待；**
  - **缓冲区是循环的。**

有限缓冲区的生产者与消费者问题

- **生产者与消费者共享的数据**

```
#define BUFFER_SIZE 10 // 定义缓冲区 buffer 大小为 10 ;
typedef struct {
        . . .
} item;      // 在 c 或者 c++ 中定义一个 item 结构 ;
item buffer[BUFFER_SIZE];
int in = 0; //  生产者放数据的指针 ;
int out = 0; // 消费者取数据的指针 ;
int counter = 0; // 指明缓冲区中放入的数据个数。
```

有限缓冲区的生产者与消费者问题

- **生产者进程：**

```
item nextProduced;   // 生产出一个产品

while (1) { …       //  循环过程
        while (counter == BUFFER_SIZE) ;  //  循环等待
      /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;  //  自增 + 取模
        counter++; // 自增
}
```

- **消费者进程：**

```
item nextConsumed;

while (1) {
        while (counter == 0) ;
 /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;  // 自减
}
```

有限缓冲区的生产者与消费者问题

- **语句“counter++” 可能由以下机器（汇编）语言来实现：**

  LOAD Reg1 counter

  INC Reg1

  STORE counter Reg1

- **语句“counter- -” 可能被实现为：**

  LOAD Reg2 counter

  DEC Reg2

  STORE counter Reg2

有限缓冲区的生产者与消费者问题

- 因为生产者和消费者进程并发执行，获得 CPU 、执行到哪条指令都是动态的，以上汇编语言指令可能会交错地执行。
- 交错或不交错、以及如何交错，取决于生产者进程和消费者进程被如何调度等等。

有限缓冲区的生产者与消费者问题

- **假设目前 counter 的值是 5。一种交错执行的情形如下：**
  生产者： LOAD Reg1 counter (***reg1* = *5***)
  生产者： INC Reg1(***reg1=6***)
  消费者： LOAD Reg2 counter (***reg2* = *5***)
  消费者： DEC Reg2(***reg2=4***)
  生产者： STORE counter Reg1 (***counter* = *6***)
  消费者： STORE counter Reg2 (***counter* = *4***)

- **counter 中的值可能是 4、5 或 6，但正确结果应该是 5。**

有限缓冲区的生产者与消费者问题

- **希望上述两进程中的语句：**
  counter++;
  counter--;
  必须是原子操作。
- **原子操作（ atomic operation ）是指一个操作必须无间断的一次完成。**
- **实际上操作往往不是原子的**

进程并发地共享数据导致不一致性

- **进程并发地对共享数据的访问有可能引起数据的不一致性。**
- **为了维持数据的一致性，必须有一种机制来保证协作进程之间按某种正确的次序执行。**

进程并发地共享数据导致不一致性

- 例：有两个并发进程 P1 和 P2 ，共享初值为 1 的变量 x 。 P1 对 x 加 1 ， P2 对 x 减 1 。加 1 和减 1 的指令序列如下：

```
// 加 1                    // 减 1
load  R1, x                load  R2, x
inc    R1                  dec R2
store  x, R1               store x, R2
```

两个操作完成后， x 的值

A) - 1 或 3　　B) 1

C) 0 ， 1 ，或 2　　D) - 1 ， 0 ， 1 或 2

怎样采用互斥方式实现对临界资源的共享？

- 临界资源
  - 临界资源：硬件或软件（如外设、共享代码段、共享数据结构），多个进程在对其进行访问时（关键是进行写入或修改），必须互斥地进行。
  - 有些共享资源可以同时访问，如只读数据。因而不是临界资源。

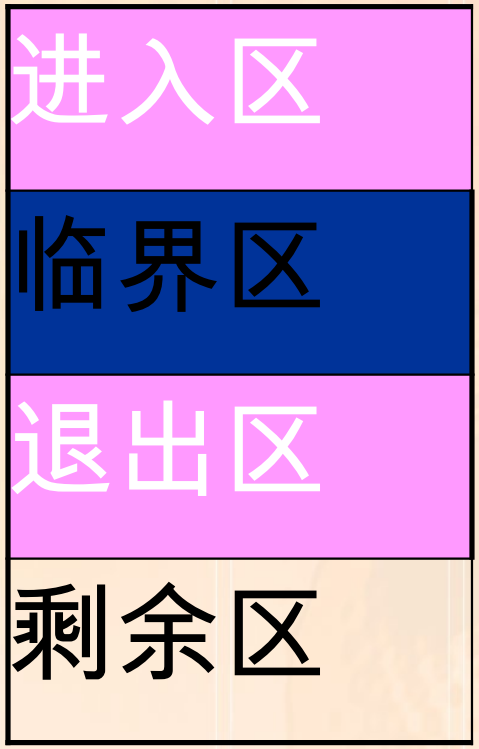
怎样采用互斥方式实现对临界资源的共享？

临界区

- 临界区 (critical section)：在每个进程中，访问临界资源的一段代码。
  - 临界区问题 - - 确保一个进程在执行它的临界区代码时，不允许其他进程再进入他们各自的临界区内执行代码。
- 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)：位于临界区的后面，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。

访问临界区的循环进程描述

repeat



检查临界资源是否能访问

将临界区标志设为未访问

until false;

怎样采用互斥方式实现对临界资源的共享？

对临界资源的互斥访问，可以在逻辑上分为如下四个部分：

```
do{
    entry section;   //  进入区
    critical section;   //  临界区
    exit  section;      //  退出区
    remainder section;     //  剩余区
}   while (true)
```

负责检查是否可以进入临界区，若可进入，则应当设置正在访问临界区的资源标志（“上锁”），阻止其他进程同时进入临界区。

访问临界资源的代码

负责解除正在访问临界资源的标志（“解锁“）

做其他处理

注意：
临界区是进程中访问临界资源的代码段。
进入区和退出区是负责实现互斥的代码段。
临界区可也成为“临界段”。

如果一个进程暂时不能进入临界区，那么该进程是否应该一直占着处理机？该进程有没有可能一直进不了临界区？

怎样采用互斥方式实现对临界资源的共享？

- 当进程处于临界区时，说明进程正在占用处理机，只要不破坏临界资源的使用规则，是不会影响处理及调度的。比如，通常访问临界资源可能是慢速的外设（如打印机），如果在进程访问打印机时，不能处理机调度，那么系统的性能将非常低。
- 不适合处理机调度的情况：
  1. 在处理中断的过程中
  2. 进程在操作系统内核程序临界区中
  3. 其他需要完全屏蔽中断的原子操作过程中

怎样采用互斥方式实现对临界资源的共享？

- 例：若某单处理器多进程系统中，有多个进程处于就绪态，则下列关于处理机调度中的叙述中，错误的是：
  - A) 在进程结束时能进行处理机切换
  - B) 创建新进程后能进行处理机切换
  - C) 在进程处于临界区时，不能进行处理机切换
  - D) 在系统调用完成并返回用户态时能进行处理机切换

怎样采用互斥方式实现对临界资源的共享？

- 为了实现对临界资源的互斥访问，同时保证系统整体性能，同步机制应遵循的准则：
  - **空闲则入**：其他进程均不处于临界区，应允许请求进入临界区的进程进入；
  - **忙则等待**：已有进程处于其临界区，请求进入临界区的进程应等待；
  - **有限等待**：等待进入临界区的进程不能“死等”；
  - **让权等待**：不能进入临界区的进程，应释放 CPU（如转换到阻塞状态）

```
do{
    entry section;   // 进入区
    critical section;   // 临界区
    exit  section;      // 退出区
    remainder section;     // 剩余区
}  while (true)
```

为解决临界区问题的最初尝试

- 只有 2 个进程： P_0 和 P_1
- 进程 P_i （或 P_j） 通常的结构：

```
do {
    进入区( entry section ) ;
   临界区( critical section ) ;
    退出区( exit section ) ;
    剩余区( reminder section ) ;
        } while (1);
```

- 进程可以通过共享某些公共变量来同步他们的活动。

算法 1

- **共享变量：**
  - int turn ；
    初值： turn = i;
  - **如果** turn = i $\Rightarrow$ P_i **能进入他的临界区**

- 进程 P_i　　进程 P_j

```
do {
    while (turn != i) ;
        临界区 ;
    turn = j;
        剩余区 ;
} while (1);
```

```
do {
    while (turn != j) ;
        临界区 ;
    turn = i ;
        剩余区 ;
} while (1);
```

- 可以**确保互斥**，但是不能做到**空闲让进**

算法 2

- **共享变量：**
  - boolean flag[2]**;**
    初值： flag [i] = flag [j] = false.
  - **如果** flag [i] = true $\Rightarrow$ P_i **准备进入临界区**

进程 P_i

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

进程 P_j

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

- **可以确保互斥，但是不能保证有限等待！**

算法 3

- **综合算法 1 和算法 2**
- **flag[i]=true 表示进程 Pi 想进入临界区**
- **turn = i 表示轮到 Pi 进程进入临界区**
- **初始** flag [i] = flag [j] = false

算法 3

Pi 进程：

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

Pj 进程：

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

- **满足上述解决两个进程的临界区问题的三个原则。**
- **只要采用 FCFS 调度算法，即可保证满足有空让进和有限等待**

算法 4: 单标志法

算法思想：两个进程在访问临界区后会把使用临界区的权限转交给另一个进程。也就是每个进程进入临界区的权限只能被另一个进程赋予。

```
int turn = 0; //turn 表示当前允许进入临界区的进程号

P0 进程:                                P1进程:
while (turn != 0);          ①           while (turn != 1);          ⑤ //进入区
critical section;           ②           critical section;           ⑥ //临界区
turn = 1;                   ③           turn = 0;                   ⑦ //退出区
remainder section;          ④           remainder section;          ⑧ //剩余区
```

turn 变量背后的逻辑：表达“谦让”

turn 的初值为 0 ，即刚开始只允许 0 号进程进入临界区。
若 P1 先上处理机运行，则会一直卡在⑤。直到 P1 的时间片用完，发生调度，切换 P0 上处理机运行。
代码①不会卡住 P0 ， P0 可以正常访问临界区，在 P0 访问临界区②期间即时切换回 P1 ， P1 仍然会卡在⑤。
只有 P0 在退出区③将 turn 改为 1 后， P1 才能进入临界区。

因此，该算法可以实现“同一时刻最多只允许一个进程访问临界区”。

算法 4: 单标志法

```
int turn = 0; //turn 表示当前允许进入临界区的进程号
```

turn 变量背后的逻辑：表达“谦让”

```
P0 进程:                          P1进程:
while (turn != 0);        ①       while (turn != 1);        ⑤ //进入区
critical section;         ②       critical section;         ⑥ //临界区
turn = 1;                 ③       turn = 0;                 ⑦ //退出区
remainder section;        ④       remainder section;        ⑧ //剩余区
```

只能按照 P0-->P1-->P0-->P1... 这样轮流访问。这种必须“轮流访问”带来的问题是：如果 turn 的初值为 0 ，刚开始 P1 先上处理机运行，则会一直卡在⑤，此时允许进入临界区的进程是 P0 ，而 P0 一直不访问临界区，那么虽然此时临界区空闲，但是并不允许 P1 访问。

因此，**单标志法存在的主要问题是：违背“空闲让进”原则。**

算法 5 双标志先检查法

算法思想：设置一个布尔型数组 flag[] ，数组中各个元素用来标记各进程想进入临界区的意愿，比如“ flag[0]=true” 意味着 0 号进程 P0 现在想要进入临界区。每个进程在进入临界区之前，先检查当前有没有别的进程想进入临界区，如果没有，则把自身对应的标志 flag[i] 设置为 true ，之后开始访问临界区。

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                        P1 进程:
while (flag[1]);   ①            while (flag[0]);   ⑤   //如果此时 P0 想进入临界区，P1 就一直循环等待
flag[0] = true;    ②            flag[1] = true;    ⑥   //标记为 P1 进程想要进入临界区
critical section;  ③            critical section;  ⑦   //访问临界区
flag[0] = false;   ④            flag[1] = false;   ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;              remainder section;
```

理解背后的含义：“表达意愿”

进入区

若按照①⑤②⑥③⑦ ... 的顺序执行， P0 和 P1 将会同时访问临界区。
因此，双标志先检查法的主要问题是：违反”忙则等待”原则。
原因在于，进入区的“检查”和“上锁”两个处理不是一气呵成的。
“ 检查”后，“上锁”前可能发生进程切换。

算法 6 双标志后检查法

算法思想：双标志先检查法的改版。前一个算法的问题是先“检查”后“上锁”，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题。因此，人们又想到先“上锁”后“检查”的方法，来避免上述问题。

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                          P1 进程:
flag[0] = true;       ①          flag[1] = true;       ⑤   //标记为 P1 进程想要进入临界区
while (flag[1]);      ②          while (flag[0]);      ⑥   //如果 P0 也想进入临界区，则 P1 循环等待
critical section;     ③          critical section;     ⑦   //访问临界区
flag[0] = false;      ④          flag[1] = false;      ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;                remainder section;
```

理解背后的含义：“表达意愿”

进入区

若按照①⑤②⑥ ... 的顺序执行， P0 和 P1 将都无法进入临界区。

因此，双标志后监察法虽然解决了“忙则等待”的问题，但是又违背了“空闲让进”和“有限等待”原则，会因各进程都长期无法访问临界资源而产生“饥饿”现象。

两个进程都想争着进入临界区，但是谁都不让谁，最后谁都无法进入临界区。

算法 7 Peterson 算法

算法思想：结合双标志发、单标志法的思想。如果双方都争着想进入临界区，那可以让进程尝试“孔融让梨”(谦让)。做一个有礼貌的进程。

```
bool flag[2];          //表示进入临界区意愿的数组，初始值都是false
int turn = 0;          //turn 表示优先让哪个进程进入临界区
P0 进程:
flag[0] = true;                    ①
turn = 1;                          ②
while (flag[1] && turn==1);        ③
critical section;                  ④
flag[0] = false;                   ⑤
remainder section;
P1 进程:
flag[1] = true;                    ⑥ //表示自己想进入临界区
turn = 0;                          ⑦ //可以优先让对方进入临界区
while (flag[0] && turn==0);        ⑧ //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;                  ⑨
flag[1] = false;                   ⑩ //访问完临界区，表示自己已经不想访问临界区了
remainder section;
```

背后的含义：“表达意愿”

表达“谦让”

进入区：1. 主动争取；2. 主动谦让；3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”

谁最后说了“客气话”，谁就失去了行动的优先权。

Eg: 过年了，某阿姨给你发压岁钱。

场景一

阿姨：乖，收下阿姨的心意~

你：不用了阿姨，您的心意我领了

阿姨：对阿姨来说你还是个孩子，你就收下吧

结局...

动手推导：

按不同的顺序穿插执行会发生什么？

①②③⑥⑦⑧...

①⑥②③...

①③⑥⑦⑧...

①⑥②⑦⑧...

面包店算法

解决 n 个进程的临界区问题

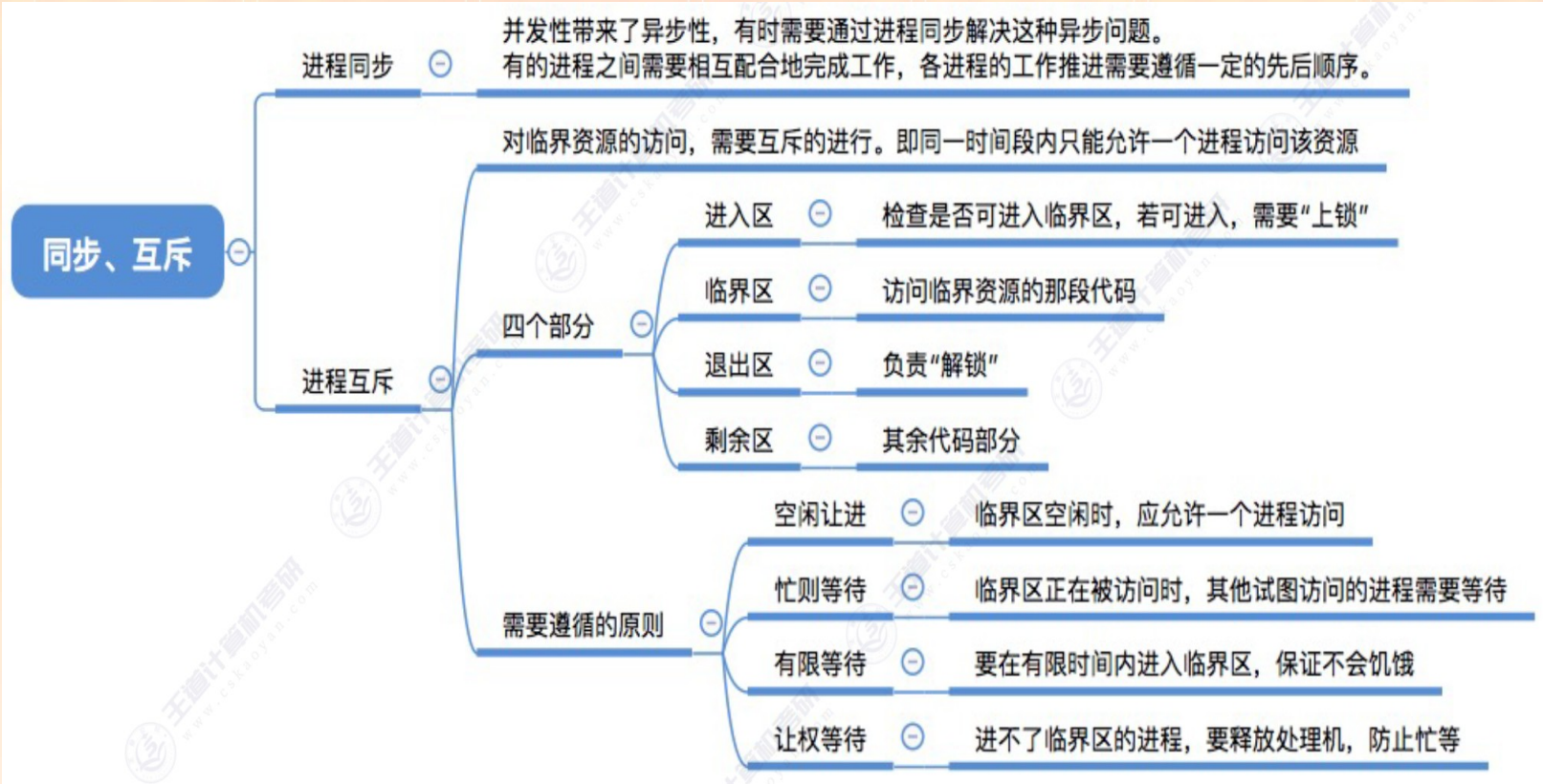
- **在进入临界区之前，进程试图去获取一个排队号码，持有最小号码的进程进入他的临界区。**
- **如果进程 P_i 和 P_j 得到的是同样的号码，如果 $i < j$， 那么进程 P_i 先进入临界区，否则进程 P_j 进入临界区。**
- **系统生成的后一个号码总是大于等于前一个。例如：1,2,3,3,3,3,4,5...**

- 符号 < 表示按字典顺序排序关系（序号#, 进程 id #）
  - $(a,b) < (c,d)$ 如果 $a < c$ 或者 $a = c$ 且 $b < d$
  - $\max(a_0,\ldots, a_{n-1})$ 得到整数 k, 从而 $k \geq a_i$, $i = 0, \ldots, n - 1$
- 共享数据：
  - boolean choosing[n]; // 表示进程是否正在抓号，初值为 false。
    若进程 i 正在抓号，则 choosing[i-1]=true.
  - int number[n]; // 记录进程抓到的号码，初值为 0。
    若 number[i-1]=0 ，则进程 i 没有抓号。

任一个进程 Pi 的代码段如下：

```
do {
    choosing[i] = true;
    number[i] = max ( number[0], number[1], …, number
    [n – 1] )+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {     // 遍历数组
      while (choosing[j]);
      //  判断进程是否抽号，若是则等待该进程抽完号再比较
    while ((number[j]!=0)&&(number[j],j)<(number[i],i));
     //  判断自己的号是否最小，若是则继续直到遍历完成后进入
    临界区；否则等待比自己号小的进程处理完毕。
    }
    临界区
    number[i] = 0;     // 进程处理完后，将自己的号清零
    剩余区
} while (1);
```

进程同步与进程互斥总结



2.4.2 进程互斥的硬件实现方法

- **中断屏蔽方法**

利用“开 / 关中断指令”实现（与原语的实现思想相同，即在某进程开始访问临界区到结束访问临界区为止。都不允许中断，也就不可能发生进程切换。因此，也不可能发生两个进程，同时访问临界区的情况）

```
...
关中断；
临界区；
开中断；
...
```

关中断后即不允许当前进程被中断，也必然不会发生进程切换

直到当前进程访问完临界区，再执行开中断指令，才有可能有别的进程上处理机并访问临界区。

优点：简洁、高效
缺点：不适用于多处理机；只适用于操作系统内核进程，不适用于用户进程（因为开 / 关中断指令只能运行在内核态，这组指令如果能让用户随意使用会很危险）

2.4.2 进程互斥的硬件实现方法

- **TestAndSet 指令**

简称 TS 指令，有的地方也成为 TestAndSetLock 指令，或 TSL 指令。TSL 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用 C 语言描述的逻辑。

```
//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
}
```

```
//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...
```

若刚开始 lock 为 false ，则 TSL 返回的 old 值为 false ， while 循环条件不满足，直接跳过循环，进入临界区。若刚开始 lock 为 true ，则执行 TLS 后 old 返回的值是 true ， while 循环条件满足，会一直循环，直到当前访问临界区的进程在退出区进行"解锁"。

相比软件实现方法， TSL 指令把"上锁"和"检查"操作用硬件的方式变成了一气呵成的原子操作。优点：实现简单，无需像软件实现方法那样严格检查是否有逻辑漏洞；适用于多处理机环境。缺点：不满足"让权等待"原则，暂时无法进入临界区的进程会占用 CPU 并循环执行 TSL 指令，从而导致"忙等"。

2.4.2 进程互斥的硬件实现方法

- **Swap 指令**

有的地方也叫做 Exchange 指令，或简称 XCHG 指令。

Swap 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用 C 语言描述的逻辑：

```
//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;
}
```

```
//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...
```

逻辑上来看 Swap 和 TSL 并无太大区别，都是先记录下此时临界区是否已经被上锁（记录在 old 变量上），再将上锁标记 lock 设置为 true ，最后检查 old ，如果 old 为 false 则说明之前没有别的进程对临界区上锁，则可跳出循环、进入临界区。

优点：实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞；适用于多处理机环境。缺点：不满足“让权等待”原则，暂时无法进入临界区的进程会占用 CPU 并循环执行 Swap 指令，从而导致“忙等”。

2.4.2 进程互斥的硬件实现方法