

library computing CPU mainframe task web apps desktop program applications OS user interface allocation file system resources storage real-time security command processor memory web server process code update

operating system operating system

河南大学

# 操作系统

计算机学院

instruction multi-tasking software hardware computer graphical interface CUI management interrupt input output device driver multi-user game console supercomputer services microcomputer RAM user job control networking virtual memory phone version

library computing CPU mainframe task web apps desktop program applications OS user interface allocation file system resources storage real-time security command processor memory web server process code kernel update



operating system operating system

instruction multi-tasking software hardware computer graphical interface CUI management interrupt input output device driver multi-user game console supercomputer services microcomputer RAM user job control networking virtual memory phone version

# 张 帆

## 教授



**13839965397**



**zhangfan@henu.edu.cn**



**计算机学院 412 房间**



## 第 2 章

# 进程的描述与控制



# 大纲

**1** 2.7 线程的基本概念

**2** 2.8 线程的实现

**3** 2.4 进程同步



## 1 2.7 线程的基本概念

## 2 2.8 线程的实现

## 3 2.4 进程同步



# 线程的引入

- 线程是近年来操作系统领域出现的一个非常重要的技术，其引入进一步提高了程序并发执行的程度，从而进一步提高了资源的利用率和系统的吞吐量。
- 进程的两个基本属性
  - 资源的拥有者
  - 独立调度单位
- 进程并发执行的时空开销
  - 创建进程
  - 撤销进程
  - 进程切换：对进程进行上下文切换时，要保护当前进程的 CPU 环境，设置新进程的 CPU 环境。

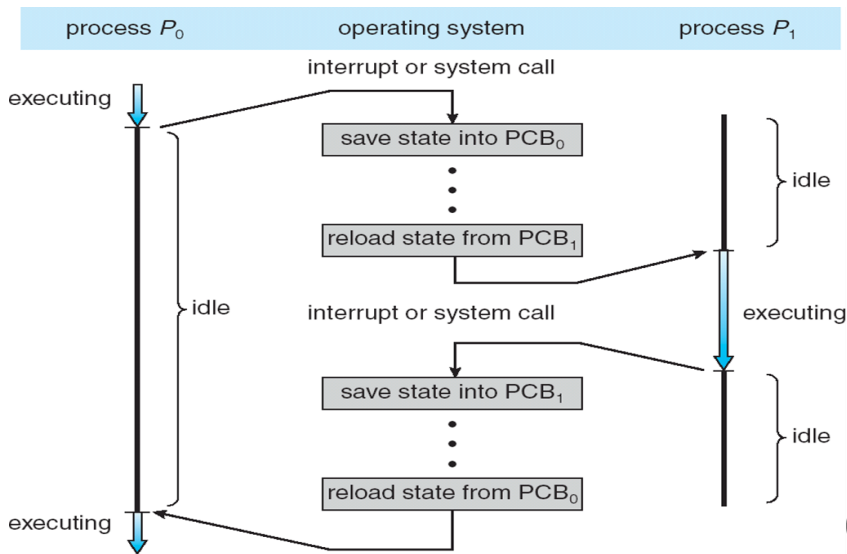


# 上下文切换

- **上下文切换** (Context Switching) : 在多任务系统中, 上下文切换是指 CPU 的控制权由运行任务转移到另外一个就绪任务时所发生的事件。
- **任务上下文**(Task Context) : 任务上下文是指任务运行的环境。例如程序计数器、堆栈指针、通用寄存器的内容。任务上下文的内容依赖于具体的 CPU 。
- 在 UNIX 系统中, 进程的上下文由 3 部分组成
  - 用户级上下文: 程序、数据、用户堆栈以及共享存储区
  - 系统级上下文: 进程控制块、内存管理信息
  - 寄存器上下文: 程序计数器 PC 、程序状态寄存器 PS 、栈指针 SP , 通用寄存器



# 线程的引入





# 线程的引入

- 进程的数量不宜过多，进程间切换不宜过频繁。
- 线程的引入
  - 目的：减少并发执行时的时空开销。因为进程的创建、撤消、切换较费时空，它既是调度单位，又是资源拥有者。
  - 线程是系统独立调度和分派的基本单位，基本上不拥有系统资源，只需要少量的资源（指令指针 IP，寄存器，栈），但可以共享其所属进程所拥有的全部资源。



# 进程与线程的比较



## 1 调度单位

- 引入线程后，线程是处理机调度的基本单位，进程是资源分配的基本单位，而不再是一个可执行的实体。
- 在同一进程中线程的切换不会引起进程的切换，但从一个进程中的线程切换到另一个进程中的线程时，将会引起进程的切换。

## 2 并行性

- 引入线程后，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行。
- 多个线程会争夺处理机，在不同的状态之间进行转换。线程也是一个动态的概念，也有一个从创建到消亡的生命过程，具有动态性。



# 进程与线程的比较

## 3 资源分配

- 进程是资源分配的单位，一般线程自己不拥有系统资源，但可以访问其隶属进程的资源。
- 同一进程中的所有线程都具有相同的地址空间（进程的地址空间）。

## 4 独立性

- 同进程的不同线程间的独立性要比不同进程间的独立性低得多。
- 多个线程共享进程的内存地址空间和资源。



# 进程与线程的比较

## 5 系统开销

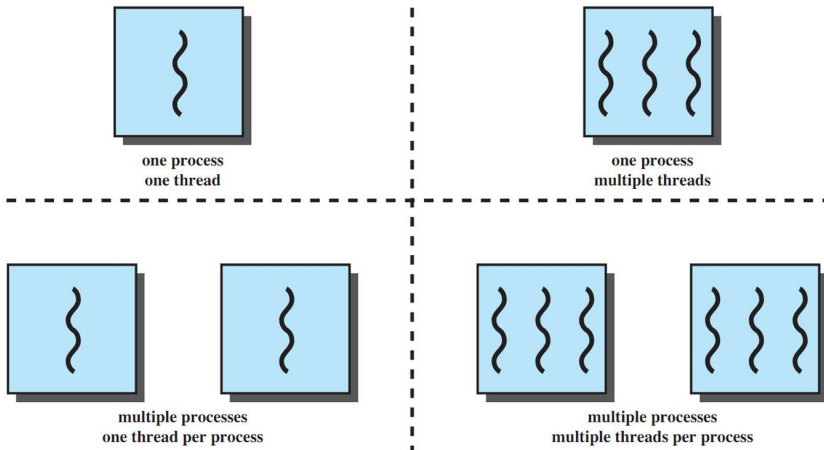
- 创建、撤销一个新线程系统开销小。
- 两个线程间的切换系统开销小。
- 在一些 OS 中，同一进程内的线程之间切换、同步、相互通信无须内核干预。

## 6 支持多处理机系统

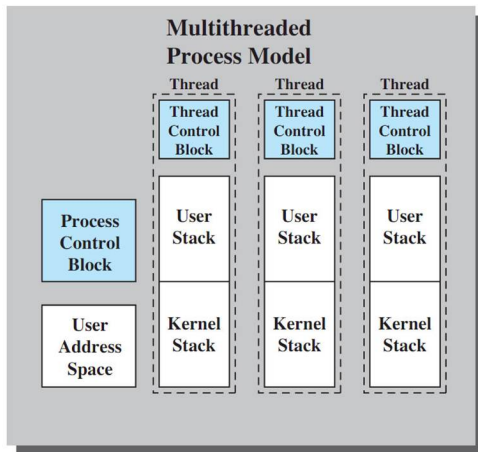
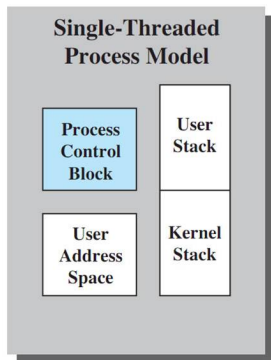
- 同进程的不同线程可以分配到多个处理机上执行，加快了进程的完成。
- 现代 OS 全部支持多线程。



# 进程与线程的存在方式



# 进程与线程的存在方式



# 进程与线程的比较

	进程	线程
引入目的	能并发执行，提高资源的利用率和系统吞吐量。	提高并发执行的程度，进一步提高资源的利用率和系统吞吐量。
并发性	较低	较高
基本属性（调度）	资源拥有的基本单位→进程 独立调度/分派基本单位→进程	资源拥有的基本单位→进程 独立调度/分派基本单位→线程
基本状态	就绪、执行、等待	就绪、执行、等待
拥有资源	资源拥有的基本单位—进程	资源拥有的基本单位—进程
系统开销	创建/撤消/切换时空开销较大	创建/撤消/切换时空开销较小
系统操作	创建、撤消、切换	创建、撤消、切换
存在标志	进程控制块PCB	PCB、线程控制块TCB
关系	单进程单线程、单进程多线程、多进程单线程、多进程多线程	



## 线程运行的三个状态

- 线程的运行状态：如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。
  - 1 执行状态：线程正获得处理机而运行
  - 2 就绪状态：具备了除 CPU 外的所有资源
  - 3 阻塞状态：线程处于暂停执行时的状态
- 线程有挂起状态吗？





## 线程运行的三个状态

- 线程的运行状态：如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。
  - 1 执行状态：线程正获得处理机而运行
  - 2 就绪状态：具备了除 CPU 外的所有资源
  - 3 阻塞状态：线程处于暂停执行时的状态
- 线程有挂起状态吗？

挂起是进程级的概念

一个进程被挂起，它的所有线程也必被挂起



## 线程控制块 TCB

- **线程控制块 TCB**：标志线程存在的数据结构，其中包含对线程管理所需要的全部信息。
  - 线程标识符
  - 寄存器状态：程序计数器和堆栈指针中的内容
  - 堆栈：保存局部变量和返回地址
  - 运行状态：执行、就绪、阻塞
  - 优先级
  - 线程专有存储器：保存线程自己的局部变量
  - 信号屏蔽

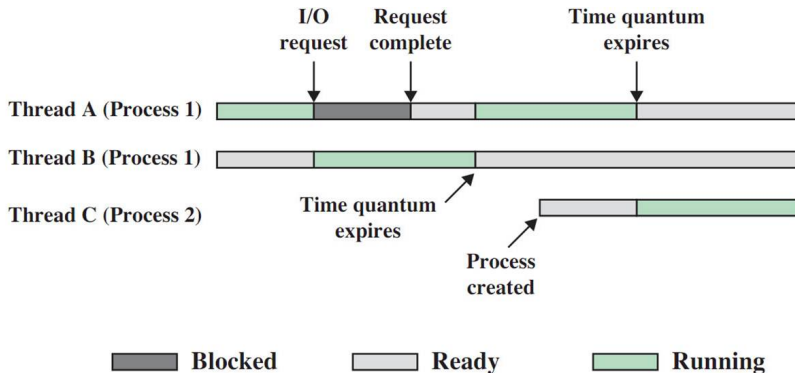


## 多线程中的进程

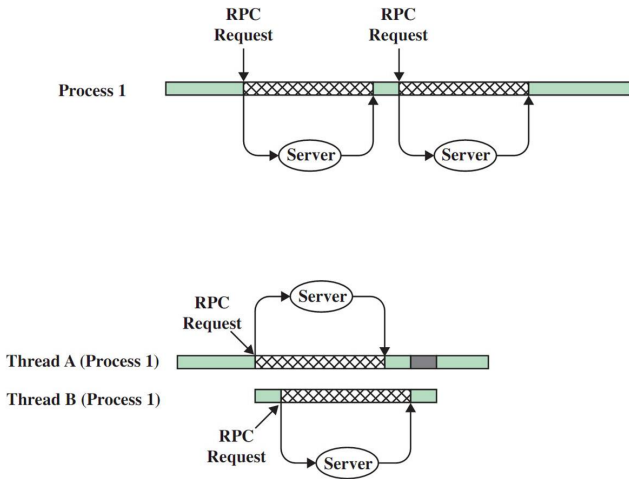
- 拥有系统资源的基本单位。
- 可包括多个线程。
- 不再是一个可执行的实体。所谓进程处于执行状态，实际上是指该进程中的某线程正在执行。
  - 把某个进程挂起时，该进程中的所有线程也都被挂起。
  - 把某进程激活时，属于该进程的所有线程也都被激活。



# 多线程示例



# 多线程示例



上图：单线程的远程过程调用（RPC）。下图：多线程的 RPC。



## 多线程的应用例子

- 局域网中只有一个文件服务器，在一段时间内需要处理多个文件请求。为每一个请求创建一个线程。
- 一个进程由几个独立部分组成，且不需要顺序执行，则每个部分可以线程方式实现。当一个线程因 I/O 阻塞时，可切换到同一进程的另一个线程。
- 前台和后台操作：如电子表格中，一个线程读取用户输入，另一个线程执行用户指令并更新表格。
- 异步处理：如文字处理中，创建一个线程，其任务是周期性进行备份。



## 1 2.7 线程的基本概念

## 2 2.8 线程的实现

## 3 2.4 进程同步



## 线程的实现方式

- 对于通常的进程，无论是系统进程还是用户进程，进程的创建、撤消以及要求由系统设备完成的 I/O 操作，都是利用系统调用而进入 OS 内核，再由内核中的相应处理程序予以完成。
- 进程的切换同样是在内核的支持下实现的。
- 不论是系统进程还是用户进程，不论是进程还是线程，都必须直接或间接得到 OS 内核的支持。

系统调用是内核提供的一组函数，是应用程序和操作系统内核之间的功能接口。





# 线程的实现方式

## ■ 内核支持线程 KST

- 内核支持线程由操作系统直接支持，在内核空间中执行线程的创建、调度和管理。
- 优点：支持多处理器；支持用户进程中的多线程；某个线程阻塞，其他的线程还可继续执行；**内核线程间切换速度比较快。**



# 线程的实现方式

## ■ 用户级线程 ULT

- 用户线程指不需要内核支持而在用户空间中实现的线程。
- 内核并不知道用户级的线程，所有线程的创建和调度都是在用户空间内进行的，而无需内核的干预。
- 对于用户级线程其调度仍是以进程为单位。
- 优点：同一进程内的线程切换不需要转换到内核空间，调度算法是进程专用的，与操作系统平台无关。
- 缺点：①系统调度**阻塞问题**：当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都会被阻塞。②不能充分利用多处理器。



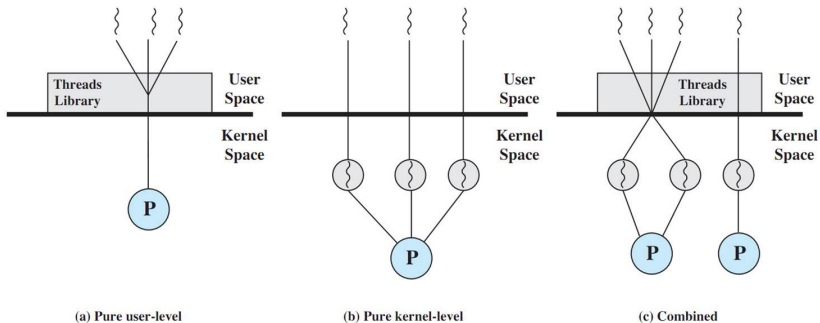
# 线程的实现方式

## ■ 组合方式

- ULT/KST 组合方式线程系统
- 内核支持多个内核支持线程的建立、调度和管理，同时允许用户应用程序建立、调度和管理用户级线程。
- 多对一模型：阻塞问题
- 一对一模型：系统开销大
- 多对多模型



# 线程的实现方式



{ User-level thread      (wavy circle) Kernel-level thread      (P) Process



# 线程的实现

- 内核支持线程的实现
  - 直接利用系统调用进行线程控制。
  - 内核支持线程的创建、撤消调度和切换调度和切换与进程类似。(开销比进程小得多)
- 用户级线程的实现
  - 运行时系统
  - 轻型进程



# 线程的实现

## ■ 运行时系统

- 运行时系统是用于管理和控制线程的函数（过程）的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数以及实现线程调度的函数等。
- 运行时系统使用户级线程与内核无关。
- 运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。
- 当线程需要系统资源时，将该要求传送给运行时系统，由后者通过相应的 **系统调用** 来获得系统资源。



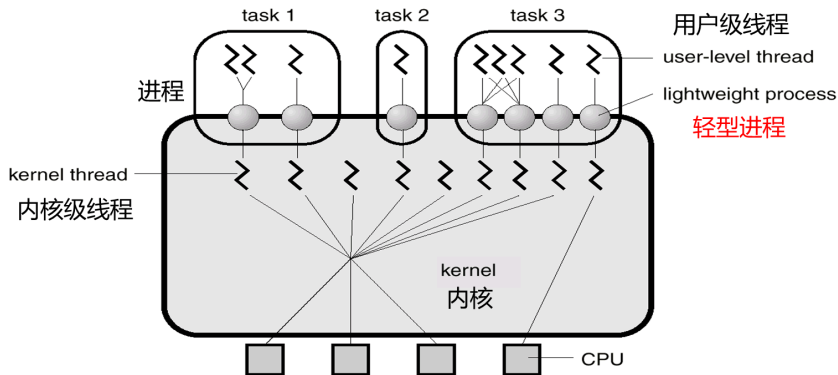
# 线程的实现

## ■ 轻型进程 (LWP)

- LWP 是一种由内核支持的用户线程（内核控制线程）。
- 每个 LWP 由一个内核线程支持，LWP 通过系统调用来获得内核提供的服务。
- 当一个用户级线程运行时，只要将它连接到一个 LWP 上，便具有了内核支持线程的所有属性。
- LWP 把用户线程和内核线程绑定到一起，内核看到的是多个 LWP，而看不到用户级线程，实现了在内核与用户级线程之间的隔离，从而使用户级线程与内核无关。
- 内核线程阻塞时，与之相连的 LWP 都会阻塞，连接到这些 LWP 的用户级线程也会阻塞。



# 线程的实现





# 内核级和用户级线程的比较

## ■ 调度和切换速度

- 内核级线程切换类似于进程切换，但速度比进程快。
- 用户级线程切换通常发生在同一用户进程的诸线程间，无需进入内核，更快。

## ■ 系统调用

- 用户级线程，一个线程进行系统调用，整个进程阻塞。
- 内核级线程，一线程进行系统调用，阻塞该线程，进程仍可运行。

## ■ 执行时间

- 用户级线程以进程为单位平均分配时间，对线程间并发执行并不有利。
- 内核级线程以线程为单位平均分配时间。



# 内核级和用户级线程的比较

	用户级线程ULT	核心级线程KLT
控制	线程库	内核
调度单位	进程	线程
切换速度	同一进程诸线程间，切换由线程库完成，速度较快	由内核完成，速度较慢
系统调用行为	内核看作是整个用户进程的行为	内核只看作该线程的行为
阻塞	用户进程	线程
优点	线程切换不调用内核，切换速度更快；调度算法可由应用程序决定	适合多处理器，可同时调度同一进程的多个线程，速度较快；阻塞在线程一级
缺点	阻塞在用户进程一级	



# 线程的创建与终止

- 线程的创建
  - 应用程序启动时通常只有一个线程（初始化线程），主要功能是利用系统调用创建新线程。
- 线程的终止
  - 线程被中止后并不立即释放它所占有的资源。
  - 已被终止但尚未释放资源的线程，仍可以被其它线程所调用，重新恢复运行。



## 课堂练习

- 1 在下面的叙述中，不正确的是（ ）。
  - A. 一个进程可创建一个或多个线程
  - B. 一个线程可创建一个或多个线程
  - C. 一个线程可创建一个或多个进程
  - D. 一个进程可创建一个或多个进程
- 2 一个运行的进程用完了分配给它的时间片后的状态变为（ ）。
  - A. 阻塞    B. 就绪    C. 运行    D. 挂起
- 3 进程依靠（ ）从阻塞状态过渡到就绪状态。
  - A. 程序员的命令      B. 合作进程的唤醒
  - C. 等待下一个时间片到来    D. 系统服务



# 课堂练习

- 1 在下面的叙述中，不正确的是（ ）。
- A. 一个进程可创建一个或多个线程
  - B. 一个线程可创建一个或多个线程
  - C. 一个线程可创建一个或多个进程
  - D. 一个进程可创建一个或多个进程
- 2 一个运行的进程用完了分配给它的时间片后的状态变为（ ）。
- A. 阻塞
  - B. 就绪
  - C. 运行
  - D. 挂起
- 3 进程依靠（ ）从阻塞状态过渡到就绪状态。
- A. 程序员的命令
  - B. 合作进程的唤醒
  - C. 等待下一个时间片到来
  - D. 系统服务

1C 2B 3B



## 课堂练习

- 4 下面关于线程的叙述中，正确的是（ ）。
- A. 不论是系统支持线程还是用户级线程，其切换都需要内核的支持。
  - B. 线程是资源的分配单位，进程是调度和分配的单位。
  - C. 不管系统中是否有线程，进程是拥有资源的独立单位。
  - D. 在引入线程的系统中，进程仍是资源分配和调度分派的基本单位。
- 5 为使进程由活动就绪变为静止就绪，使用（ ）原语？
- A. Wakeup    B. Active    C. Suspend    D. Block



## 课堂练习

- 4 下面关于线程的叙述中，正确的是（ ）。
- A. 不论是系统支持线程还是用户级线程，其切换都需要内核的支持。
  - B. 线程是资源的分配单位，进程是调度和分配的单位。
  - C. 不管系统中是否有线程，进程是拥有资源的独立单位。
  - D. 在引入线程的系统中，进程仍是资源分配和调度分派的基本单位。
- 5 为使进程由活动就绪变为静止就绪，使用（ ）原语？
- A. Wakeup    B. Active    C. Suspend    D. Block

4C 5C



## 课堂练习

- 6 在进程状态转换时，下列（ ）转换是不可能发生的。
- A. 就绪态 → 运行态    B. 运行态 → 就绪态
  - C. 运行态 → 阻塞态    D. 阻塞态 → 运行态
- 7 在下面的叙述中，正确的是（ ）。
- A. 引入线程后，处理机只在线程间切换。
  - B. 引入线程后，处理机仍在进程间切换。
  - C. 线程的切换，不会引起进程的切换。
  - D. 线程的切换，可能引起进程的切换。





## 课堂练习

- 6 在进程状态转换时，下列（ ）转换是不可能发生的。
- A. 就绪态 → 运行态    B. 运行态 → 就绪态
  - C. 运行态 → 阻塞态    D. 阻塞态 → 运行态
- 7 在下面的叙述中，正确的是（ ）。
- A. 引入线程后，处理机只在线程间切换。
  - B. 引入线程后，处理机仍在进程间切换。
  - C. 线程的切换，不会引起进程的切换。
  - D. 线程的切换，可能引起进程的切换。

6D    7D



## 1 2.7 线程的基本概念

## 2 2.8 线程的实现

## 3 2.4 进程同步



# 进程同步

- **进程同步** 是指对多个相关进程在执行次序上进行协调，它的目的是使系统中诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。
- 用来实现同步的机制称为 **同步机制**。如：信号量机制；管程机制。



## 两种形式的制约关系

- 系统中诸进程之间在逻辑上存在着两种制约关系

### 1 直接制约关系（进程同步）：

为完成同一个任务的诸进程间，因需要协调它们的工作而相互等待、相互交换信息所产生的直接制约关系。

### 2 间接制约关系（进程互斥）：

进程共享独占型资源而必须互斥执行的间接制约关系。



# 两种形式的制约关系

## ■ 同步与互斥的比较

同 步	互 斥
进程-进程	进程-资源-进程
时间次序上受到某种限制	竞争某一资源时不允许进程同时工作
相互清楚对方的存在及作用，交换信息	不一定清楚其它进程情况
往往指有几个进程共同完成一个任务	往往指多个任务、多个进程间相互制约
例：生产与消费之间，作者与读者之间	例：过独木桥



# 临界资源

- **临界资源**(Critical Resource) 是一次只允许一个进程使用的资源，如打印机、绘图机、变量、数据等。
- 进程之间采取 **互斥方式** 实现对临界资源的共享，从而实现并行程序的封闭性。
- 引起不可再现性是因为对临界资源没有进行互斥访问。
- 在每个进程中，访问临界资源的那一段代码称为**临界区** (Critical Section )，简称 CS 区。



# 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**

```
R1=X;  
R1=R1+1;  
X=R1;
```



**B:**

```
R2=X;  
R2=R2+1;  
X=R2;
```



# 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**

```
R1=X;  
R1=R1+1;  
X=R1;
```



**B:**

```
R2=X;  
R2=R2+1;  
X=R2;
```

A 与 B 均对 X 加 1，即 X 加 2。





# 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**                    **R1=X;**  
                         **R1=R1+1;**  
                         **X=R1;**

← 临界区



**B:**                    **R2=X;**  
                         **R2=R2+1;**  
                         **X=R2;**

← 临界区

A 与 B 均对 X 加 1，即 X 加 2。



# 临界资源

如果按另一顺序对变量进行修改

<b>A:</b>	<b>R1=X;</b>
<b>B:</b>	<b>R2=X;</b>
<b>A:</b>	<b>R1=R1+1;</b>
	<b>X=R1;</b>
<b>B:</b>	<b>R2=R2+1;</b>
	<b>X=R2;</b>



# 临界资源

如果按另一顺序对变量进行修改

```
A:          R1=X;  
B:          R2=X;  
A:          R1=R1+1;  
            X=R1;  
B:          R2=R2+1;  
            X=R2;
```

结果 X 只加了 1



# 临界资源

如果按另一顺序对变量进行修改

A:	R1=X;
B:	R2=X;
A:	R1=R1+1;
	X=R1;
B:	R2=R2+1;
	X=R2;

结果 X 只加了 1

**产生错误的原因：**不加控制地访问共享变量 X  
对临界区需要进行保护（互斥访问）



# 临界区

- 为了保证临界资源的正确使用，可以把 **临界资源的访问过程** 分成以下几部分：

- **进入区**：增加在临界区前面的一段代码，用于检查欲访问的临界资源此刻是否被访问。
- **临界区**：进程访问临界资源的那段代码。
- **退出区**：增加在临界区后面的一段代码，用于将临界资源的访问标志恢复为未被访问标志。
- **剩余区**：进程中除了进入区、临界区及退出区之外的其余代码。

进入区

临界区

退出区

剩余区



## 同步机制应遵循的规则

- 1 空闲让进**：当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效地利用临界资源。
- 2 忙则等待**：当已有进程进入临界区时，表明临界资源正在被访问，因而其他试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。
- 3 有限等待**：对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，以免陷入 **死等** 状态。
- 4 让权等待**：当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入 **忙等**。



## 解决临界区（互斥）问题的几类方法

- 软件方法：用编程解决。
  - Dekker 算法
  - Peterson 算法
- 硬件方法：用硬件指令解决。
- **信号量及 P-V 操作**
- 管程



## 解决临界区（互斥）问题的几类方法

- 软件方法：用编程解决。
  - Dekker 算法
  - Peterson 算法
- 硬件方法：用硬件指令解决。
- **信号量及 P-V 操作**
- 管程

同步机制





## 软件方法：Dekker 算法的初步设想

定义全局变量  $turn$ ，如果  $turn=0$ ：P0 可以进入 CS；  
如果  $turn=1$ ：P1 可以进入 CS。

```
int turn; /*共享的全局变量*/
```

**P0**

...

```
while (turn!=0) do no_op;
```

```
<CS>
```

```
turn=1;
```

...

**P1**

...

```
while (turn!=1) do no_op
```

```
<CS>
```

```
turn=0;
```

...



## 软件方法：Dekker 算法的初步设想

定义全局变量  $turn$ ，如果  $turn=0$ ：P0 可以进入 CS；  
如果  $turn=1$ ：P1 可以进入 CS。

**int**  $turn$ ; /\*共享的全局变量\*/

**P0**

...

**while** ( $turn \neq 0$ ) **do**  $no\_op$ ;

**<CS>**

**turn=1;**

...

忙等  
busy waiting

**P1**

...

**while** ( $turn \neq 1$ ) **do**  $no\_op$

**<CS>**

**turn=0;**

...



# Dekker 算法：初步设想

## ■ 出现的问题

- 进程强制交替进入临界区，容易造成资源利用不充分。
- 当  $turn=0$  时，即使此时 CS 空闲，P1 也必须等待 P0 进入 CS 执行、退出后才能进入 CS，**不符合空闲让进的原则**
- 进程不能进入自己的临界区时，没有立即释放处理机，陷入忙等，**不符合让权等待的原则**。
- 任何进程在 CS 区内、CS 区外失败，其它进程将可能因为等待使用 CS 而无法向前推进。



# Dekker 算法：改进一

使用全局共享数组 flag 标志 CS 状态：

flag[0] 或 flag[1] = true: 表示 P0 或 P1 占用 CS

flag[0] 或 flag[1] = false: 表示 CS 空闲

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

```
flag[0]=flag[1]=false;
```

**P0**

**P1**

...

...

```
while (flag[1]) do no_op; while (flag[0]) do no_op;
```

```
flag[0]=true;
```

```
flag[1]=true;
```

```
<CS>
```

```
<CS>
```

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...



# Dekker 算法：改进一

## ■ 出现的问题

- 进程在 CS 内失败且相应的  $\text{flag}=\text{ture}$ ，则其它进程永久阻塞。
- **不能实现互斥！** P0 和 P1 可能同时进入临界区。当 P0 执行  $\text{while}(\text{flag}[1])$  并通过以后，在执行  $\text{flag}[0]=\text{true}$  之前，P1 执行  $\text{while}(\text{flag}[0])$ ，这样两个进程同时进入了临界区。



# Dekker 算法：改进一

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

```
flag[0]=flag[1]=false;
```

**P0**

**P1**

...

...

```
while (flag[1]) do no_op;① while (flag[0]) do no_op;②
```

```
flag[0]=true;③
```

```
flag[1]=ture;④
```

<CS>

<CS>

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...

按图中①②③④的次序执行，P0 和 P1 都可进入 CS 区  
不能实现互斥。



# Dekker 算法：改进二，改为先置标志位

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

```
flag[0]=flag[1]=false;
```

**P0**

...

```
flag[0]=true;
```

```
while (flag[1]) do no_op;
```

```
<CS>
```

```
flag[0]=false;
```

...

**P1**

...

```
flag[1]=ture;
```

```
while (flag[0]) do no_op;
```

```
<CS>
```

```
flag[1]=false;
```

...



## Dekker 算法：改进二

- 出现的问题
  - 不能实现空闲让进，有限等待。P0 和 P1 可能都进入不了临界区。
  - 当 P0 执行了  $\text{flag}[0] = \text{true}$  后，P1 执行了  $\text{flag}[1] = \text{true}$ ，这样两个进程都无法进入临界区（阻塞）。





# Dekker 算法：改进二

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
flag[0]=flag[1]=false;
```

**P0**

...

**flag[0]=true; ①**

**while (flag[1]) do no\_op; ③**

**<CS>**

**flag[0]=false;**

...

**P1**

...

**flag[1]=ture; ②**

**while (flag[0]) do no\_op; ④**

**<CS>**

**flag[1]=false;**

...

按图中①②③④的次序执行，都无法进入各自的 CS 区。



# 成功的 Dekker 算法

同时使用flag和turn

```
Process P0    begin
                flag[0]:=true;
                while (flag[1])
                { if turn=1 then
                  begin
                    flag[0]:=false;
                    while (turn=1) do no-op;
                    flag[0]:=true;
                  end    };
                临界区;
                turn = 1;
                flag[0]:=false;
            end;
```

While循环体避免  
“改进2”中的死锁



# Peterson 算法

- 代码更简洁
- 设两个全局共享变量：flag[0]、flag[1]，表示临界区状态及哪个进程正在占用 CS。
- 设一个全局共享变量 turn：表示能进入 CS 的进程序号。



# Peterson 算法

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

```
flag[0]=flag[1]=false;
```

**P0**

...

```
flag[0]=true;
```

```
turn=1;
```

```
while (flag[1] and turn==1)
```

```
do no_op;
```

```
<CS>
```

```
flag[0]=false;
```

...

**P1**

...

```
flag[1]=ture;
```

```
turn=0;
```

```
while (flag[0] and turn==0)
```

```
do no_op;
```

```
<CS>
```

```
flag[1]=false;
```

...



# Peterson 算法

- 在进入区先修改后检查
  - 检查对方 flag：如果不在临界区则自己进入（空闲让进）。
  - 否则再检查 turn：turn 保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入（先到先得）。
- 当  $\text{flag}[1]=\text{false}$  或  $\text{turn} = 0$ ，即当进程 1 没有要求进入 CS，或仅允许进程 0 进入 CS 时，P0 进入 CS。



## 硬件方法

- 关中断指令
- TestAndSet 指令
- SWAP 指令



## 中断禁用 (关中断, Interrupt Disabling)

- 如果进程访问临界资源时 (执行临界区代码) 不被中断, 就可以利用它来保证互斥地访问。
- 方法: 使用**关中断**、**开中断**原语。
- 过程
  - 关中断**
  - 临界区
  - 开中断**
  - 剩余区
- 存在问题
  - 代价高: 限制了处理器交替执行各进程的能力。
  - 不能用于多处理器结构: 关中断不能保证互斥。



# TestAndSet 指令 (TS 指令) (测试并设置)

## TS指令定义(逻辑)

```

Boolean TestAndSet (int i)
{
    if (i==0)
    {
        i=1;
        return true;
    }
    else
        return false;
}

```

## 使用TestAndSet实现互斥

```

int lock;           //lock取0或1
...                 //0开, 1锁
While(!TestAndSet(lock));
临界区;             //此时lock=1
lock=0;              //开锁
剩余区
...

```

- TS指令管理临界区时，把一个临界区与一个变量lock相连，由于变量lock代表了临界资源的状态，可把它看成一把锁。
- TS指令自动整体执行，不响应任何中断，故可实现进程互斥。





# Swap 指令 (交换)

Swap指令：

```
void Swap (int a, int b)
{
    int temp=a;
    a = b;
    b=temp ;
}
```

使用Swap指令实现互斥

- 每个临界资源设置一个公共变量lock，初值为0(开锁)。
- 进程要使用临界资源时首先把私有变量key置为1。

key=1;

```
do{
    Swap (lock, key);
} while (key);
```

临界区  
lock=0;  
剩余区

0为false  
非0为true



# 互斥与同步解决方法

## ■ 软件方法

- 实现比较复杂，需要较高的编程技巧。

## ■ 硬件方法

- 不能实现让权等待。
- 可能出现死锁。



# 互斥与同步解决方法

- 软件方法

- 实现比较复杂，需要较高的编程技巧。

- 硬件方法

- 不能实现让权等待。
  - 可能出现死锁。

有效解决进程同步问题的方法

**信号量机制**



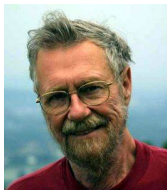
# 信号量机制

- **信号量机制**是荷兰科学家 E. W. Dijkstra 在 1965 年提出的一种同步机制，也称为**P、V 操作**。
- **信号量**
  - 用于表示资源数目或请求使用某一资源的进程个数的整型量。
- 1 整型信号量
- 2 记录型信号量
- 3 AND 信号量集
- 4 一般信号量集



# Dijkstra

- Dijkstra(1930~2002), 荷兰计算机科学家, 计算机先驱之一, 1972 年第七位图灵奖获得者。
- ALGOL 语言的主要贡献者, 提出了结构化程序设计结构, 曾经提出“goto 有害论”, 提出了信号量和 PV 原语, 解决了“哲学家聚餐”问题。
- Dijkstra 的创新思想包括: 结构编程、堆栈、矢量、信号量、同步进程和死锁。



# 信号量机制

- 信号量只能通过**初始化**和**两个标准的原语**（P，V 操作）来访问。
- **初始化**：指定一个整数值，表示空闲资源总数。
- **P 操作也称为 wait 操作，V 操作也称为 signal 操作。**
- 信号量是比锁更高级的资源抽象方式。
- **注意**：P、V 操作应作为一个整体实施，不允许分割。



## 整型信号量

- **整型信号量**：非负整数，用于表示资源数目。除了初始化外，只能通过两个原子操作 wait 和 signal (P, V) 来访问。
- wait 和 signal 操作描述：

```
wait(S): while S ≤ 0 do no-op      // 测试有无可用资源
          S := S - 1;              // 可用资源数减一
signal(S): S := S + 1;
```

- 主要问题：只要  $S \leq 0$ ，wait 操作就不断地测试（忙等），因而未做到“让权等待”。



## 记录型信号量

- 为了解决“让权等待”问题，需要引入阻塞队列，信号量值可以取负值——**记录型信号量**。
- 设置一个代表资源数目的整型变量 value（资源信号量）。
- 设置一链表指针 L 用于链接所有等待的进程。



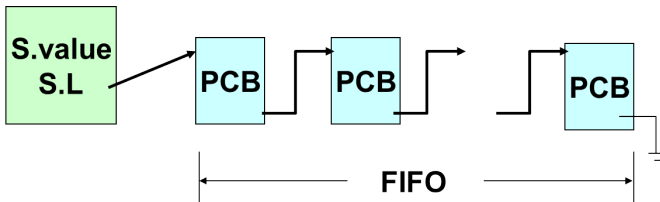


# 记录型信号量的数据结构

```

Type semaphore=record
    value: integer;
    L: list of process;
end
  
```

- 1 S.value: 信号量值
- 2 S.L: 进程等待队列



## s.value 的物理含义

- 执行一次  $P(s)$  操作，意味着进程请求分配该类资源一个单位的资源。
- 执行一次  $V(s)$  操作意味着进程释放相应资源一个单位的资源。当值小于等于 0 时，表明有进程被阻塞，需要唤醒。
- 在记录型信号量机制中：
  - $S.value > 0$ ：表示系统中某类资源当前可用的数目。
  - $S.value \leq 0$ ：表示该类资源已分配完。若有进程请求该类资源，则被阻塞，其绝对值表示该信号量链表中因申请该资源而被阻塞进程的数目。

例如：10 个进程，5 台打印机。



# 记录型信号量

记录型信号量 wait 和 signal 操作描述：


```
wait(S): S.value:=S.value-1;  
        if S.value<0 then block(S.L); // 让权等待  
signal(S): S.value:=S.value+1;  
          if S.value≤0 then wakeup(S.L);
```

- 1  $S.value > 0$ : 表示系统中某类资源当前可用的数目。
- 2  $S.value \leq 0$ : 其绝对值表示该信号量链表中因申请该资源而被阻塞进程的数目。



## P 操作过程描述

```
P(S) {  
    lockout interrupt ; //关中断  
    S=S-1 ;             //可用资源数减1  
    if(S<0) {  
        status= "blocked" ; //状态=阻塞  
        Insert(Q); //插入相应阻塞队列Q中  
        unlock interrupt;    //开中断  
        Scheduler;          //进程调度  
    }  
    else unlock interrupt ;  //开中断  
}
```



## V 操作过程描述

```
V(S) {  
    lockout interrupt;           //关中断  
    S=S+1;  
    If( S≤0) {  
        A = Remove(Q);  
        //从相应阻塞队列Q中取出队首进程A  
        status(A) = "ready" ; //A状态 = 就绪  
        Insert(A, RL);          //A插入就绪队列RL  
        length(RL) = length(RL)+1;  
        //就绪队列RL长度加1  
    }  
    unlock interrupt             //开中断  
}
```



## AND 型信号量

- 上述的进程互斥问题，是针对各进程之间只共享一个临界资源而言的。
- 在有些应用场合，是一个进程需要先获得两个或更多的共享资源后方能执行，但这种情况可能发生死锁。
- 假定现有两个进程 A 和 B，他们都要求访问共享数据 D 和 E。可为这两个数据分别设置用于互斥的信号量 Dmutex 和 Emutex，并令它们的初值都是 1。

process A	process B
wait(Dmutex);	wait(Emutex);
wait(Emutex);	wait(Dmutex);



## AND 型信号量

- 若进程 A 和 B 按下述次序交替执行 wait 操作：

process A: wait(Dmutex); 于是 Dmutex=0

process B: wait(Emutex); 于是 Emutex=0

process A: wait(Emutex); 于是 Emutex=-1 **A 阻塞**

process B: wait(Dmutex); 于是 Dmutex=-1 **B 阻塞**

- 最后，进程 A 和 B 处于僵持状态。在无外力作用下，两者都将无法从僵持状态中解脱出来。我们称此时的进程 A 和 B 已进入**死锁**状态。

为避免死锁，可以采用**AND 型信号量**。



## AND 型信号量

- AND 型信号量的基本思想
  - 将进程在整个运行过程中所需要的**所有临界资源**，**一次性全部分配给进程**，待进程使用完后再一起释放。
  - 只要有一个资源未能分配给进程，其它所有可能分配的资源也不分配给该进程。从而可避免死锁发生。
- AND 型信号量集 P 原语为 Swait(Simultaneous Wait)，V 原语为 Ssignal(Simultaneous Signal)。
- 在 Swait 时，各个信号量的次序并不重要，虽然会影响进程归入哪个阻塞队列，但是因为是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。





# AND 型信号量

```

1  Swait(S1; S2; ...; Sn)    // P 原语;
2  {
3      if (S1>=1 && S2>=1 && ... && Sn>=1)
4          {                      // 满足全部资源要求才进行减 1 操作
5              for (i=1; i<=n; i++)
6                  Si--;
7          }
8      else
9          {调度进程进入第一个小于
10         1 的信号量的等待队列 Si.L;
11         }
12     }

```



# 一般信号量集

- **一般信号量集**是同时需要多种资源、每种占用的数目不同、且可分配资源还存在一个临界值时的信号量处理。
- 一般信号量集的基本思路就是在 AND 型信号量集的基础上扩充，在一次原语操作中完成所有的资源申请。
- 进程对信号量  $S_i$  的测试值为  $t_i$ （表示信号量的判断条件，要求当资源数量低于  $t_i$  时，便不予分配），占用值为  $d_i$ （表示资源的申请量，即  $S_i = S_i - d_i$  和  $S_i = S_i + d_i$ ）。
- 一般信号集的特点
  - 一次可分配多个某种临界资源，不需执行多次 P 操作。
  - 每次分配前都测试该种资源数目是否大于测试值。



# 一般信号量集

```
Swait(S1, t1, d1; ...; Sn, tn, dn);
Ssignal(S1, d1; ...; Sn, dn);    // 释放时不必考虑 t i
```

- 一般信号量集可以用于各种情况的资源分配和释放。下面是几种特殊的情况：
  - Swait(S, d, d): 表示每次申请d个资源，当资源数量少于d个时，便不予分配。
  - Swait(S, 1, 1): 蜕化为一般的记录型信号量 ( $S > 1$  时) 或互斥信号量 ( $S = 1$ )。
  - Swait(S, 1, 0): 可作为一个可控开关 ( $S \geq 1$  时，允许多个进程进入某特定区； $S = 0$  时禁止任何进程进入)。



# 一般信号量集

$S_i$ : 可用资源数     $t_i$ : 阈值     $d_i$ : 申请资源数

```

1  Swait( $S_1$ ,  $t_1$ ,  $d_1$ ; ...,  $S_n$ ,  $t_n$ ,  $d_n$ )
2  if  $S_1 \geq t_1$  and ... and  $S_n \geq t_n$  then
3      for  $i:=1$  to  $n$  do
4           $S_i := S_i - d_i$ ;
5      endfor
6  else
7      Place the executing process in the waiting queue.
8  endif
9
10 Ssignal( $S_1$ ,  $d_1$ ; ...,  $S_n$ ,  $d_n$ )
11 for  $i:=1$  to  $n$  do
12      $S_i := S_i + d_i$ ;
13     Remove the process waiting in the queue associated with  $S_i$ .
14 endfor;
  
```



## 课堂练习

- 1 若 P、V 操作的信号量 S 初值为 2，当前值为 -1，则表示有（ ）个阻塞进程。  
■ A.0 个    B.1 个    C.2 个    D.3 个
- 2 若有三个进程共享一个程序段，且每次最多允许两个进程进入该程序段，则信号量的初值应置为（ ）。  
■ A.3    B.1    C.2    D.0
- 3 如果有 4 个进程共享同一程序段，每次允许 3 个进程进入该程序段，若用 PV 操作作为同步机制，则信号量的取值范围是（ ）。  
■ A. 4 3 2 1 -1    B. 2 1 0 -1 -2  
■ C. 3 2 1 0 -1    D. 2 1 0 -2 -3



## 课堂练习

- 1 若 P、V 操作的信号量 S 初值为 2，当前值为 -1，则表示有（ ）个阻塞进程。
- A.0 个    B.1 个    C.2 个    D.3 个
- 2 若有三个进程共享一个程序段，且每次最多允许两个进程进入该程序段，则信号量的初值应置为（ ）。
- A.3    B.1    C.2    D.0
- 3 如果有 4 个进程共享同一程序段，每次允许 3 个进程进入该程序段，若用 PV 操作作为同步机制，则信号量的取值范围是（ ）。
- A. 4 3 2 1 -1    B. 2 1 0 -1 -2
- C. 3 2 1 0 -1    D. 2 1 0 -2 -3

1B    2C    3C



## 课堂练习

- 4 有  $m$  个进程共享同一临界资源（每次只允许一个进程访问该临界资源），若使用信号量机制实现对临界资源的互斥访问，则信号量值的变化范围是（ ）。
- 5 设系统有  $n$  ( $n > 2$ ) 个进程，且当前不在执行进程调度程序，试考虑下述 4 种情况，不可能发生的是（ ）。
- A. 没有运行进程，有 2 个就绪进程， $n-2$  个进程阻塞
  - B. 有 1 个运行进程，没有就绪进程， $n-1$  个进程阻塞
  - C. 有 1 个运行进程，有 1 个就绪进程， $n-2$  个进程阻塞
  - D. 有 1 个运行进程， $n-1$  个就绪进程，没有进程阻塞



## 课堂练习

- 4 有  $m$  个进程共享同一临界资源（每次只允许一个进程访问该临界资源），若使用信号量机制实现对临界资源的互斥访问，则信号量值的变化范围是（ ）。
- 5 设系统有  $n$  ( $n > 2$ ) 个进程，且当前不在执行进程调度程序，试考虑下述 4 种情况，不可能发生的是（ ）。
- A. 没有运行进程，有 2 个就绪进程， $n-2$  个进程阻塞
  - B. 有 1 个运行进程，没有就绪进程， $n-1$  个进程阻塞
  - C. 有 1 个运行进程，有 1 个就绪进程， $n-2$  个进程阻塞
  - D. 有 1 个运行进程， $n-1$  个就绪进程，没有进程阻塞

41 至  $-(m-1)$  5A





## 课堂练习

6 对于两个并发进程，设互斥信号量为 mutex。当 mutex=0 时，则（ ）。

- A. 表示没有进程进入临界区
- B. 表示有一个进程进入临界区
- C. 表示有一个进程进入临界区，另一个进程等待进入
- D. 表示有两个进程进入临界区



## 课堂练习

- 6 对于两个并发进程，设互斥信号量为 mutex。当 mutex=0 时，则（ ）。
- A. 表示没有进程进入临界区
  - B. 表示有一个进程进入临界区
  - C. 表示有一个进程进入临界区，另一个进程等待进入
  - D. 表示有两个进程进入临界区

6B

A:mutex=1

B:mutex=0

C:mutex=-1

D: 错误



# THE END

*School of Computer & Information Engineering*

*Henan University*

*Kaifeng, Henan Province*

*475001*

*China*

