02 进程的描述与控制

> *在早期未配置OS的系统和单道批处理系统中，程序的执行方式是顺序执行，即在内存中仅装入一道用户程序，它独占系统中的所有资源，只有在一个用户程序执行完成后，才允许装入另一个程序并执行。这种方式浪费资源、系统运行效率低。*

2.1 引入

2.1.1 前趋图(Precedence Graph)

> *用于描述进程之间执行的先后顺序和并发执行情况，是一个有向无循环图，记DAG(Directed Acyclic Graph)*

构成：

- 结点：进程、程序段或一条语句
  - 初始结点(Initial Node)
  - 终止结点(Final Node)
- 有向边：两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)
- 重量：程序量或执行时间。

如：(Pi，Pj)∈→ 或 Pi→Pj

表示在Pj开始执行之前Pi必须完成。此时Pi是Pj的 直接前趋 ，而Pj是Pi的 直接后继 。

前趋图中不允许有循环。（会产生不可能实现的前趋关系）

2.1.2 程序顺序执行

> *通常，一个应用程序由若干个程序段组成，每一个程序段完成特定的功能，它们在执行时，都需要按照某种先后次序顺序执行，仅当*

> *前一程序段执行完后，才运行后一程序段。*

程序顺序执行时的特征：

- 顺序性
  指处理机严格地按照程序所规定的顺序执行，即每一操作必须在下一个操作开始之前结束
- 封闭性
  指程序在封闭的环境下运行，即程序运行时独占全机资源，资源的状态(除初始状态外)只有本程序才能改变它，程序一旦开始执行，其执行结果不受外界因素影响。
- 可再现性
  指只要程序执行时的环境和初始条件相同，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“停停走走”地执行，都可获得相同的结果。程序顺序执行时的这种特性，为程序员检测和校正程序的错误带来了很大的方便。

在引入了程序间的并发执行功能后，虽然提高了系统的吞吐量和资源利用率，但由于它们共享系统资源，以及它们为完成同一项任务而相互合作，致使在这些并发执行的程序之间必将形成相互制约的关系，由此会给程序并发执行带来新的特征。

2.2 进程的描述

2.2.1 进程的定义与特征

1.进程的定义

> *在多道程序环境下，程序的执行属于并发执行，此时它们将失去其封闭性，并具有间断性，以及其运行结果不可再现性的特征。由此，决定了通常的程序是不能参与并发执行的，否则，程序的运行也就失去了意义。为了能使程序并发执行，并且可以对并发执行的程序加以描述和控制，引入了“进程”的概念。*

对于进程的定义，从不同的角度可以有不同的定义，其中较典型的定义有：

进程是程序的一次执行。
进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

2.进程的特征

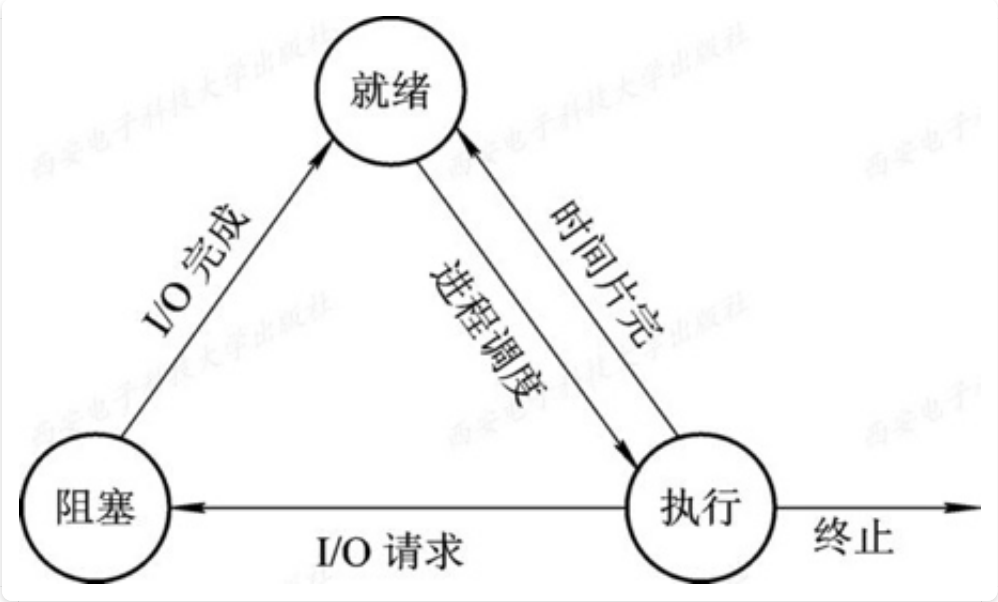
除了进程具有程序所没有的PCB结构外，还具有下面一些特征：

- 动态性
- 并发性
- 独立性
- 异步性

2.2.2 进程的基本状态及转换

1.进程的三种基本状态及转换

1. 就绪状态(Ready)
2. 执行状态(Running)
3. 阻塞状态(Block)



2.创建状态和终止状态

（1）创建状态

> *进程是由创建而产生。*

1. 由进程申请一个空白PCB，并向PCB中填写用于控制和管理进程的信

息；
2. 为该进程分配运行时所必须的资源；
3. 把该进程转入就绪状态并插入就绪队列之中。

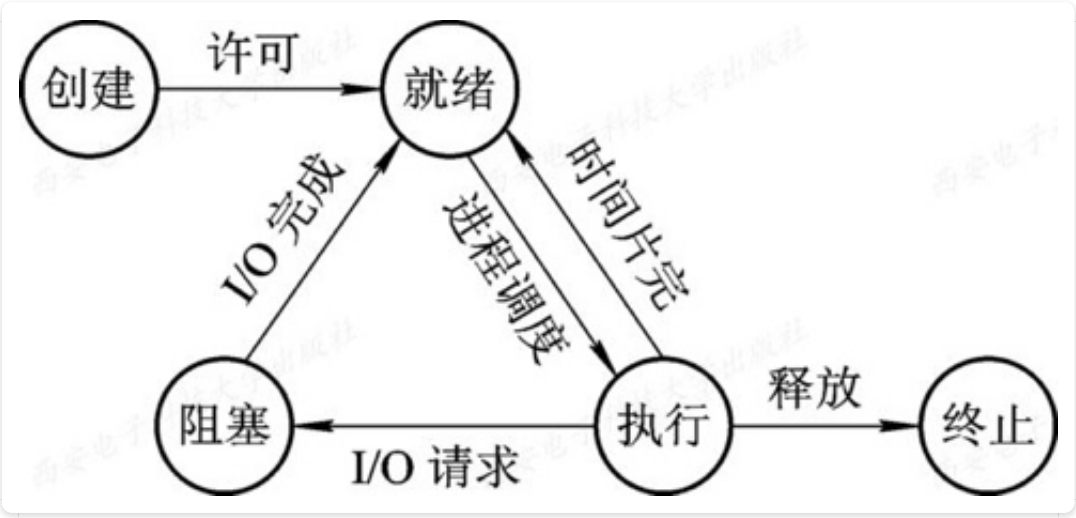
如果进程所需资源尚不能得到满足（比如系统尚无足够的内存使进程无法装入其中），此时创建工作尚未完成，进程不能被调度运行，于是把此时进程所处的状态称为创建状态。

（2）终止状态

1. 等待操作系统进行善后处理，
2. 最后将其PCB清零，并将PCB空间返还系统。

当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，它将进入终止状态。进入终止态的进程以后不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其他进程收集。一旦其他进程完成了对其信息的提取之后，操作系统将删除该进程，即将其PCB清零，并将该空白PCB返还系统。

3.进程的五种基本状态及转换



4.挂起操作和进程状态的转换

引入挂起操作的原因，是基于系统和用户的如下需要：

- 终端用户的需要
- 父进程请求
- 负荷调节的需要
- 操作系统的需要

（1）引入挂起操作后三个进程状态的转换

在引入挂起原语Suspend和激活原语Active后，在它们的作用下，进程将可能发生以下几种状态的转换：

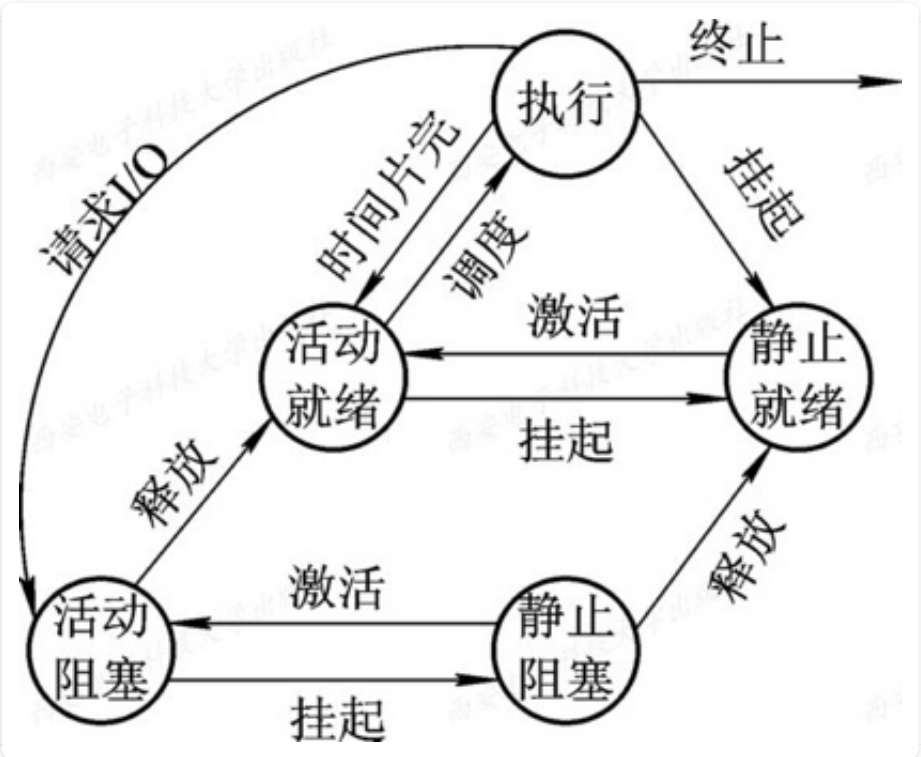
- 活动就绪→静止就绪
- 活动阻塞→静止阻塞
- 静止就绪→活动就绪
- 静止阻塞→活动阻塞

（2）引入挂起操作后五个进程状态的转换

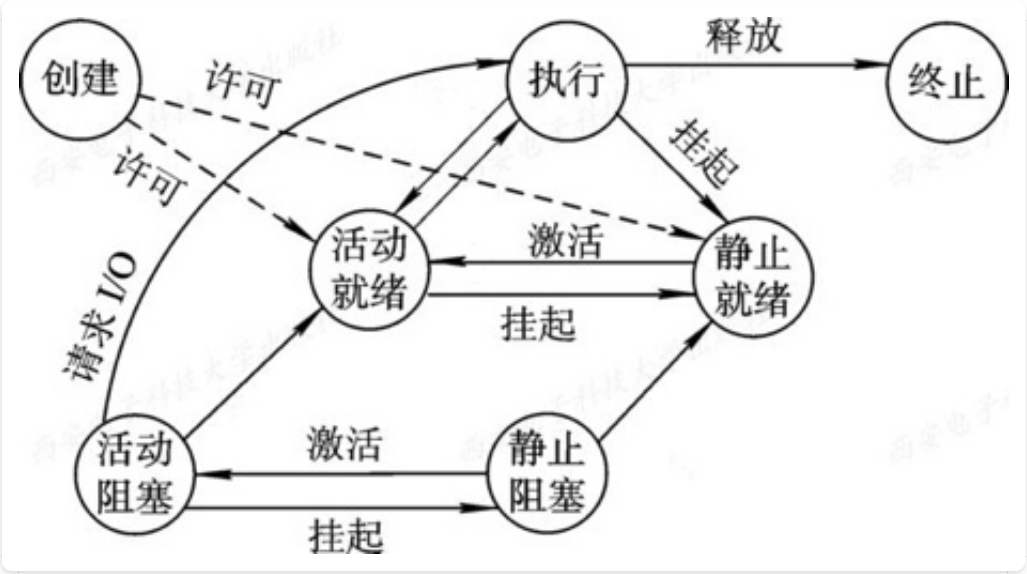
引进创建和终止状态后，在进程状态转换时，与进程五状态转换相比较，要增加考虑下面的几种情况：

1. NULL→创建
2. 创建→活动就绪
3. 创建→静止就绪
4. 执行→终止

具有挂起状态的进程状态图:



具有创建、终止和挂起状态的进程状态图:

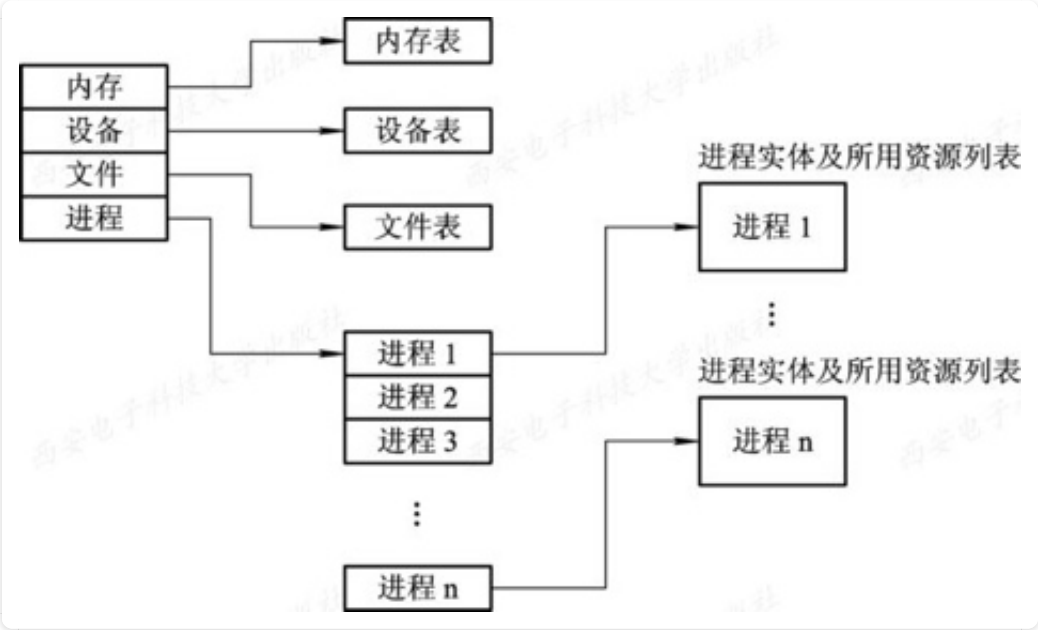


2.2.3 进程管理中的数据结构

> *在计算机系统中，对于每个资源和每个进程都设置了一个数据结构用于表征其实体，称之为资源信息表或进程信息表。*
>
> *其中包含了资源或进程的标识、描述、状态等信息以及一批指针。通过这些指针，可以将同类资源或进程的信息表，或者同一进程所占用的资源信息表分类链接成不同的队列，便于操作系统进行查找。*

- 内存表
- 设备表
- 文件表
- 进程表（进程控制块PCB）

操作系统控制表的一般结构:



1.PCB的作用

- 作为独立运行基本单位的标志
- 能实现间断性运行方式
- 提供进程管理所需要的信息
- 提供进程调度所需要的信息
- 实现与其它进程的同步与通信

2.PCB中的信息

（1）进程标识符

进程标识符用于唯一地标识一个进程。通常有两种标识符：

1. 外部标识符
2. 内部标识符

（2）处理机状态（处理机的上下文）

主要是由处理机的各种寄存器中的内容组成的。

（3）进程调度信息

- 进程状态 指明进程的当前状态，它是作为进程调度和对换时的依据；
- 进程优先级 用于描述进程使用处理机的优先级别的一个整数；
- 进程调度所需的其它信息 它们与所采用的进程调度算法有关（如进程已

等待CPU的时间总和、进程已执行的时间总和等）；
- 事件 进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

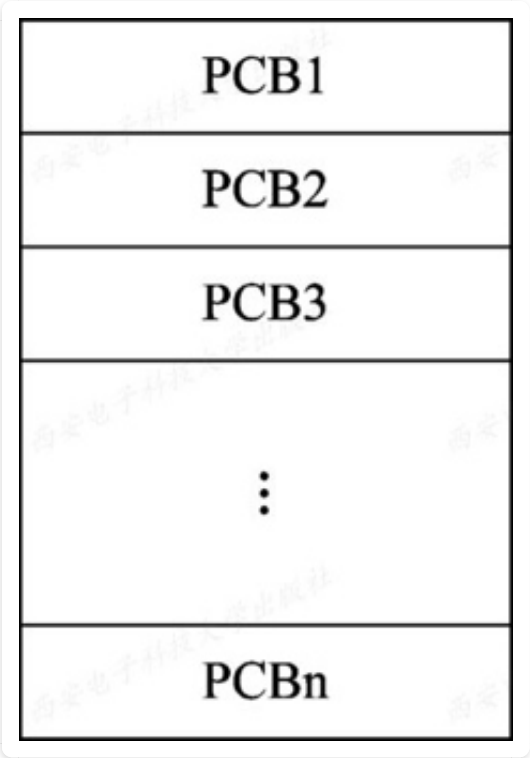
（4）进程控制信息

- 程序和数据的地址
  进程实体中的程序和数据的内存或外存地(首)址，以便再调度到该进程执行时，能从PCB中找到其程序和数据；
- 进程同步和通信机制
  这是实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中；
- 资源清单
  在该清单中列出了进程在运行期间所需的全部资源(除CPU以外)，另外还有一张已分配到该进程的资源的清单；
- 链接指针
  它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。

3.进程控制块的组织方式

（1）线性方式

> *将系统中所有的PCB都组织在一张线性表中，* *将该表的首址存放在内存的一个专用区域中* 。

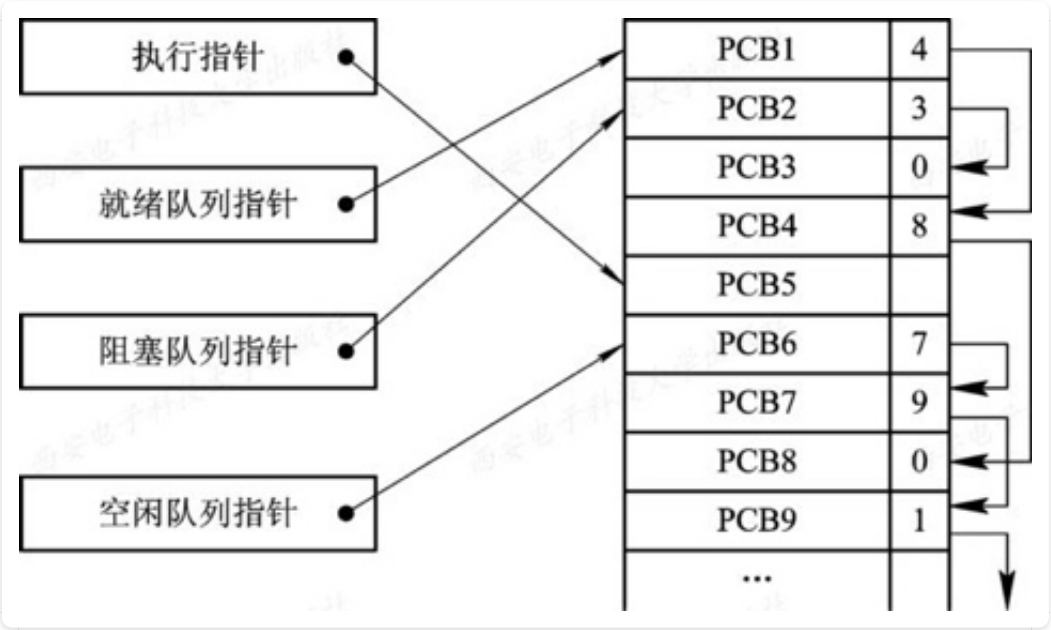


该方式实现简单、开销小，但每次查找时都需要扫描整张表，因此适合进程数目不多的系统。

（2）链接方式

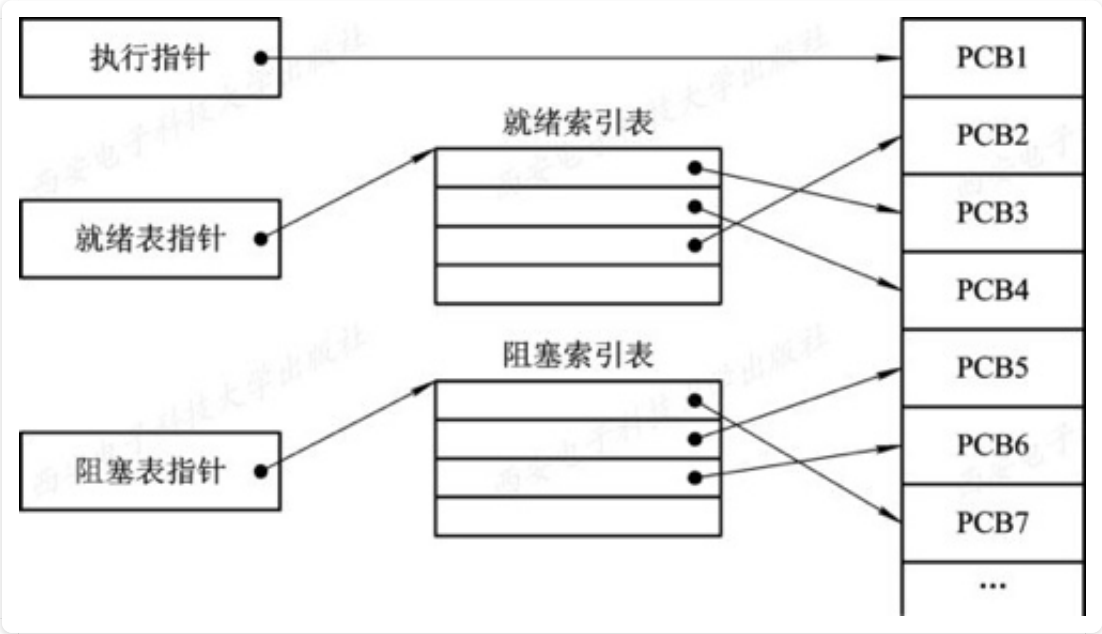
> *把具有 相同状态 进程的PCB分别通过PCB中的链接字链接成一个队列。这样，可以形成就绪队列、若干个阻塞队列和空白队列等。*

对就绪队列而言，往往按进程的优先级将PCB从高到低进行排列，将优先级高的进程PCB排在队列的前面。同样，也可把处于阻塞状态进程的PCB根据其阻塞原因的不同，排成多个阻塞队列，如等待I/O操作完成的队列和等待分配内存的队列等。



（3）索引方式

> *系统根据所有进程状态的不同，建立几张索引表，例如，就绪索引表、阻塞索引表等，并把各索引表在内存的首地址记录在内存的一些专用单元中。在每个索引表的表目中，记录具有相应状态的某个PCB在PCB表中的地址。*



2.3 进程的控制

> *进程控制是进程管理中最基本的功能。*
> *进程控制一般是由OS的内核中的原语来实现的。*

2.3.1 操作系统内核

1.支撑功能

- 中断处理
- 时钟管理
- 原语操作

2.资源管理功能

- 进程管理
- 存储器管理
- 设备管理

2.3.2 进程的创建(Creation of Process)

> *在OS中，进程可有层次结构：一个进程可创建另一个进程（父进程、子进程、孙进程……）*
> *如在UNIX中，进程与其子孙进程共同组成一个进程家族(组)。*

> *为了形象地描述一个进程的家族关系而引入了 进程图(Process Graph)（用于描述进程间关系的一棵有向树）。*

1.引起创建进程的事件

为使程序之间能并发运行，应先为它们分别创建进程。
导致一个进程去创建另一个进程的典型事件有四类：

1. 用户登录
2. 作业调度
3. 提供服务
4. 应用请求

2.进程的创建过程

每当出现了创建新进程的请求后，OS便调用进程 创建原语Creat

1. 申请空白PCB，为新进程申请获得唯一的数字标识符，并从PCB集合中索取一个空白PCB；
2. 为新进程分配其运行所需的资源，包括各种物理和逻辑资源，如内存、文件、I/O设备和CPU时间等；
3. 初始化PCB；
4. 如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

2.3.3 进程的终止(Termination of Process)

1.引起进程终止的事件

- 正常结束
- 异常结束
- 外界干预

2.进程的终止过程

如果系统中发生了要求终止进程的某事件，OS便调用进程 终止原语

1. 根据被终止进程的标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态；
2. 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度；

3. 若该进程还有子孙进程，还应将其所有子孙进程也都予以终止，以防它们成为不可控的进程；
4. 将被终止进程所拥有的全部资源或者归还给其父进程，或者归还给系统；
5. 将被终止进程(PCB)从所在队列(或链表)中移出，等待其它程序来搜集信息。 ```

2.3.4 进程的阻塞与唤醒(Block and Wakeup of Process)

1.引起进程阻塞和唤醒的事件

- 向系统请求共享资源失败
- 等待某种操作的完成
- 新数据尚未到达
- 等待新任务的到达

2. 进程阻塞过程

正在执行的进程，如果发生了上述某事件，进程便通过调用阻塞原语block将自己阻塞。可见，阻塞是进程自身的一种主动行为。

进入block过程后，由于该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将PCB插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞队列。最后，转调度程序进行重新调度，将处理机分配给另一就绪进程，并进行切换，亦即，保留被阻塞进程的处理机状态，按新进程的PCB中的处理机状态设置CPU的环境。

3. 进程唤醒过程

当被阻塞进程所期待的事件发生时，比如它所启动的I/O操作已完成，或其所期待的数据已经到达，则由有关进程(比如提供数据的进程)调用唤醒原语wakeup，将等待该事件的进程唤醒。wakeup执行的过程是：首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。

2.3.5 进程的挂起与激活

1.进程的挂起

2.进程的激活过程

2.4 进程同步

> *在OS中引入进程后，一方面可以使系统中的多道程序并发执行，这不仅能有效地改善资源利用率，还可显著地提高系统的吞吐量，但另一方面却使系统变得更加复杂。如果不能采取有效的措施，对多个进程的运行进行妥善的管理，**必然会因为这些进程对系统资源的无序争夺给系统造成混乱**。致使每次处理的结果存在着**不确定性**，即显现出其**不可再现性**。*

2.4.1 进程同步的基本概念

1.两种形式的制约关系

1. 间接相互制约关系
2. 直接相互制约关系 ### 2.临界资源(Critical Resource) 许多硬件资源如打印机、 磁带机等，都属于临界资源，诸进程间应采取互斥方式，实现对这种资源的共享。 ### 3.临界区(critical section) 由前所述可知，不论是硬件临界资源还是软件临界资源，多个进程必须互斥地对它进行访问。人们把在每个进程中访问临界资源的那段代码称为临界区(critical section)。 ### 4.同步机制应遵循的规则 为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：
3. 空闲让进
4. 忙则等待
5. 有限等待
6. 让权等待

2.4.2 同步机制

1.硬件同步机制

> *虽然可以利用软件方法解决诸进程互斥进入临界区的问题，但有一定难度，并且存在很大的局限性，因而现在已很少采用。相应地，目前许多计算机已提供了一些特殊的硬件指令，允许对一个字中的内容进行检测和修正，或者是对两个字的内容进行交换等。可利用*

> *这些特殊的指令来解决临界区问题。*

（1）关中断

> *在进入锁测试之前关闭中断，直到完成锁测试并上锁之后才能打开（实现互斥的最简单的方法之一）。*

这样，进程在临界区执行期间，计算机系统不响应中断，从而不会引发调度，也就不会发生进程或线程切换。由此，保证了对锁的测试和关锁操作的连续性和完整性，有效地保证了互斥。

缺点：

1. 滥用关中断权力可能导致严重后果
2. 关中断时间过长会限制处理器交叉执行程序的能力
3. 关中断方法也不适用于多CPU系统，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。

（2）利用Test-and-Set指令实现互斥

（3）利用Swap指令实现进程互斥

在Intel 80x86中又称为XCHG指令，用于交换两个字的内容。

2.信号量机制

（1）整型信号量

最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量S，它与一般整型量不同，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) `wait(S)` 和 `signal(S)` 来访问。很长时间以来，这两个操作一直被分别称为 `P、V操作` 。

（2）记录型信号量

> *在整型信号量机制中的wait操作，只要是信号量S≤0，就会不断地测试。*

因此，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。记录型信号量机制则是一种不存在“忙等”现象的进程同步机制。但在采取了“让

权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表指针list，用于链接上述的所有等待进程。

（3）AND型信号量

> *前面所述的进程互斥问题针对的是多个并发进程仅共享一个临界资源的情况。*

在有些应用场合，是一个进程往往需要获得两个或更多的共享资源后方能执行其任务。假定现有两个进程A和B，它们都要求访问共享数据D和E，当然，共享数据都应作为临界资源。

（4）信号量集

> *wait(S)或signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。*

当一次需要N个单位时，便要进行N次wait(S)操作，这显然是低效的，甚至会增加死锁的概率。此外，在有些情况下，为确保系统的安全性，当所申请的资源数量低于某一下限值时，还必须进行管制，不予以分配。因此，当进程申请某类临界资源时，在每次分配之前，都必须测试资源的数量，判断是否大于可分配的下限值，决定是否予以分配。

（5）信号量的应用

i.实现进程互斥

为使多个进程能互斥地访问某临界资源，只需为该资源设置一互斥信号量 `mutex` ，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

ii.实现前趋关系

还可利用信号量来描述程序或语句之间的前趋关系。设有两个并发执行的进程P1和P2。P1中有语句S1；P2中有语句S2。我们希望在S1执行后再执行S2。为实现这种前趋关系，只需使进程P1和P2共享一个 `公用信号量S` ，并赋予其初值为0，将signal(S)操作放在语句S1后面，而在S2语句前面插入wait(S)操作，即

在进程P1中，用S1；signal(S)；

在进程P2中，用wait(S)；S2；

由于S被初始化为0，这样，若P2先执行必定阻塞，只有在进程P1执行完S1；signal(S)；操作后使S增为1时，P2进程方能成功执行语句S2。

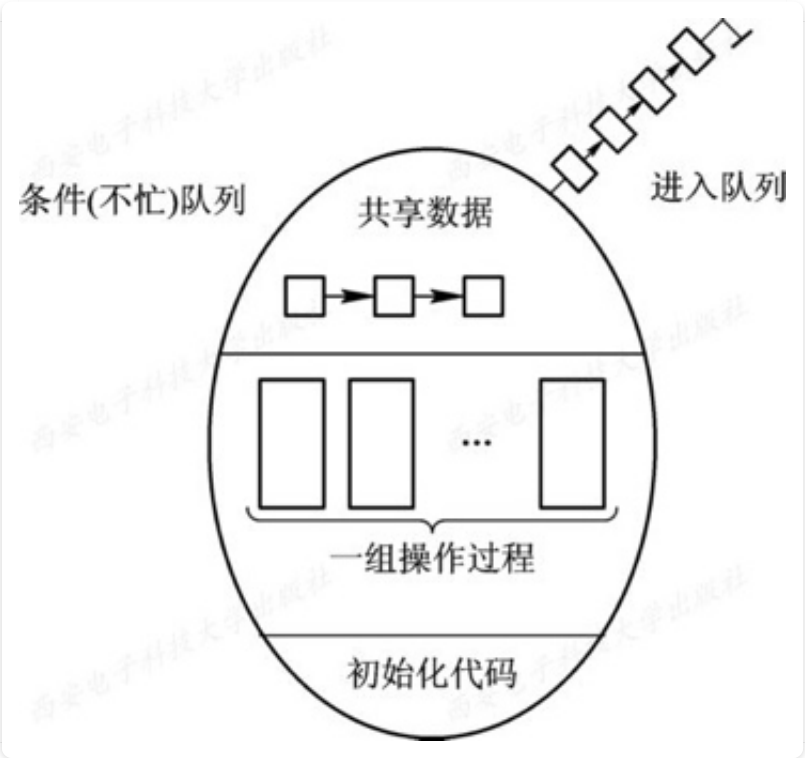
2.4.3 管程机制

1.管程的定义

> *系统中的各种硬件资源和软件资源均可用数据结构抽象地描述其资源特性，即用少量信息和对该资源所执行的操作来表征该资源，而忽略它们的内部结构和实现细节。*

由上述的定义可知，管程由四部分组成：

1. 管程的名称
2. 局部于管程的共享数据结构说明
3. 对该数据结构进行操作的一组过程
4. 对局部于管程的共享数据设置初始值的语句



2.条件变量

> *在利用管程实现进程同步时，必须设置同步工具*

如两个同步操作原语wait和signal。当某进程通过管程请求获得临界资源而未能满足时，管程便调用wait原语使该进程等待，并将其排在等待队列上，如图所示。仅当另一进程访问完成并释放该资源之后，管程才又调用signal原语，唤醒等待队列中的队首进程。

2.5 经典进程的同步问题

2.5.1 生产者-消费者问题

1.利用记录型信号量解决生产者-消费者问题

假定在生产者和消费者之间的**公用缓冲池**中具有n个缓冲区，这时可利用互斥信号量 `mutex` 实现诸进程对缓冲池的互斥使用；利用信号量 `empty` 和 `full` 分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。

2.利用AND信号量解决生产者-消费者问题

对于生产者–消费者问题，也可利用AND信号量来解决，即
用Swait(empty，mutex)来代替wait(empty)和wait(mutex)；

用Ssignal(mutex，full)来代替signal(mutex)和signal(full)；

用Swait(full，mutex)代替wait(full)和wait(mutex)；

以及用Ssignal(mutex，empty)代替Signal(mutex)和Signal(empty)。

3.利用管程解决生产者-消费者问题

在利用管程方法来解决生产者–消费者问题时，首先便是为它们建立一个管程，并命名procducerconsumer，或简称为PC。其中包括两个过程：

1. put(x)过程
2. get(x)过程　　对于条件变量 `notfull` 和 `notempty` ，分别有两个过程 `cwait` 和 `csignal` 对它们进行操作：
3. cwait(condition)过程：当管程被一个进程占用时，其他进程调用该过程

时阻塞，并挂在条件condition的队列上。

4. csignal(condition)过程：唤醒在cwait执行后阻塞在条件condition队列上的进程，如果这样的进程不止一个，则选择其中一个实施唤醒操作；如果队列为空，则无操作而返回。

2.5.2 哲学家进餐问题

1.利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。

2.利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。

2.5.3 读者-写者问题

1.利用记录型信号量解决读者-写者问题

为实现Reader与Writer进程间在读或写时的互斥而设置了一个互斥信号量Wmutex。另外，再设置一个整型变量Readcount表示正在读的进程数目。由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当Readcount=0，表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。若wait(Wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。

2.利用信号量集机制解决读者-写者问题

这里的读者—写者问题，与前面的略有不同，它增加了一个限制，即最多只允许RN个读者同时读。为此，又引入了一个信号量L，并赋予其初值为RN，通过执行wait(L, 1, 1)操作来控制读者的数目，每当有一个读者进入时，就要先执行wait(L, 1, 1)操作，使L的值减1。当有RN个读者进入读后，L便减为0，第RN + 1个读者要进入读时，必然会因wait(L, 1, 1)操作失败而阻塞。

2.6 进程通信

> *进程通信是指进程之间的信息交换。*
> *由于进程的互斥与同步，需要在进程间交换一定的信息，故不少学者将它们也归为进程通信，但只能把它们称为低级进程通信。*

以信号量机制为例来说明，它们之所以低级的原因在于：

- 效率低 生产者每次只能向缓冲池投放一个产品(消息)，消费者每次只能从缓冲区中取得一个消息；
- 通信对用户不透明 OS只为进程之间的通信提供了共享存储器。

在进程之间要传送大量数据时，应当利用OS提供的高级通信工具，该工具最主要的特点是：

- 使用方便 OS隐藏了实现进程通信的具体细节，向用户提供了一组用于实现高级通信的命令(原语)，用户可方便地直接利用它实现进程之间的通信。或者说，通信过程对用户是透明的。这样就大大减少了通信程序编制上的复杂性。
- 高效地传送大量数据 用户可直接利用高级通信命令(原语)高效地传送大量的数据。

2.6.1 进程通信的类型

1.共享存储器系统(Shared-Memory System)

1. 基于共享数据结构的通信方式
2. 基于共享存储区的通信方式

2.管道(pipe)通信系统

> *所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。*

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

1. 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。
2. 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待，直到读(输出)进程取走数据后再把它唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后才将之唤醒。
3. 确定对方是否存在，只有确定了对方已存在时才能进行通信。

3.消息传递系统(Message passing system)

> *在该机制中，进程不必借助任何共享存储区或数据结构，而是以格式化的消息 (message)为单位，将通信的数据封装在消息中，并利用操作系统提供的一组通信命令(原语)，在进程间进行消息传递，完成进程间的数据交换。*
> *基于消息传递系统的通信方式属于**高级通信方式**，因其实现方式的不同，可进一步分成两类：*

1. 直接通信方式
2. 间接通信方式

4.客户机-服务器系统(Client-Server system)

（1）套接字(Socket)

> *套接字起源于20世纪70年代加州大学伯克利分校版本的UNIX(即BSD Unix)，是UNIX操作系统下的网络通信接口。一开始，套接字被设计用在同一台主机上多个应用程序之间的通信(即进程间的通信)，主要是为了解决多对进程同时通信时端口和物理线路的多路复用问题。随着计算机网络技术的发展以及UNIX操作系统的广泛使用，套接字已逐渐成为最流行的网络通信程序接口之一。*

（2）远程过程调用和远程方法调用

> *远程过程(函数)调用RPC(Remote Procedure Call)，是一个通信协议，用于通过网络连接的系统。该协议允许运行于一台主机(本地)系统上的进程调用另一台主机(远程)系统上的进程，而对程序员表现为常规的过程调用，无需额外地为此编程。如果涉及的软件采用面向对象编程，那么远程过程调用亦可称做远程方法调用。*

远程过程调用的主要步骤是：

1. 本地过程调用者以一般方式调用远程过程在本地关联的客户存根，传递相应的参数，然后将控制权转移给客户存根；
2. 客户存根执行，完成包括过程名和调用参数等信息的消息建立，将控制权转移给本地客户进程；
3. 本地客户进程完成与服务器的消息传递，将消息发送到远程服务器进程；
4. 远程服务器进程接收消息后转入执行，并根据其中的远程过程名找到对应的服务器存根，将消息转给该存根；
5. 该服务器存根接到消息后，由阻塞状态转入执行状态，拆开消息从中取出过程调用的参数，然后以一般方式调用服务器上关联的过程；
6. 在服务器端的远程过程运行完毕后，将结果返回给与之关联的服务器存根；
7. 该服务器存根获得控制权运行，将结果打包为消息，并将控制权转移给远程服务器进程；
8. 远程服务器进程将消息发送回客户端；
9. 本地客户进程接收到消息后，根据其中的过程名将消息存入关联的客户存根，再将控制权转移给客户存根；
10. 客户存根从消息中取出结果，返回给本地调用者进程，并完成控制权的转移。

2.6.2 消息传递通信的实现方式

1.直接消息传递系统

> *在直接消息传递系统中采用直接通信方式，即发送进程利用OS所提供的发送命令(原语)，直接把消息发送给目标进程。*

（1）直接通信原语

- 对称寻址方式
- 非对称寻址方式

（2）消息的格式

在消息传递系统中所传递的消息，必须具有一定的消息格式。在单机系统环境中，由于发送进程和接收进程处于同一台机器中，有着相同的环境，所以消息的格式比较简单，可采用比较短的定长消息格式，以减少对消息的处理和存储

开销。该方式可用于办公自动化系统中，为用户提供快速的便笺式通信。但这种方式对于需要发送较长消息的用户是不方便的。为此，可采用变长的消息格式，即进程所发送消息的长度是可变的。对于变长消息，系统无论在处理方面还是存储方面，都可能会付出更多的开销，但其优点在于方便了用户。

（3）进程的同步方式

在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(或接收)或者阻塞。

（4）通信链路

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路。第一种方式是：由发送进程在通信之前用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路，在链路使用完后拆除链路。

2.6.3 直接消息传递系统实例

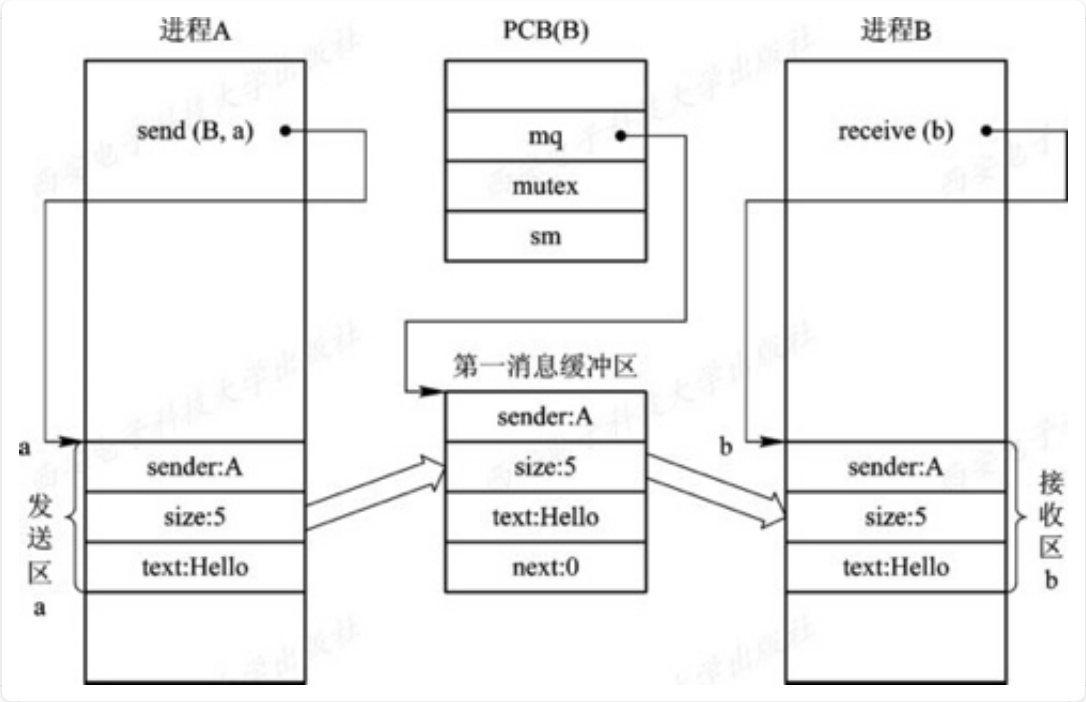
> *消息缓冲队列通信机制首先由美国的Hansan提出，并在RC4000系统上实现，后来被广泛应用于本地进程之间的通信中。在这种通信机制中，发送进程利用Send原语将消息直接发送给接收进程；接收进程则利用Receive原语接收消息。*

1.消息缓冲队列通信机制中的数据结构

- 消息缓冲区
- PCB中有关通信的数据项

2.发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间设置一发送区a，如下图所示，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的内部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后都要执行wait和signal操作。



3.接收原语

接收进程调用接收原语receive(b)，从自己的消息缓冲队列mq中摘下第一个消息缓冲区i，并将其中的数据复制到以b为首址的指定消息接收区内。

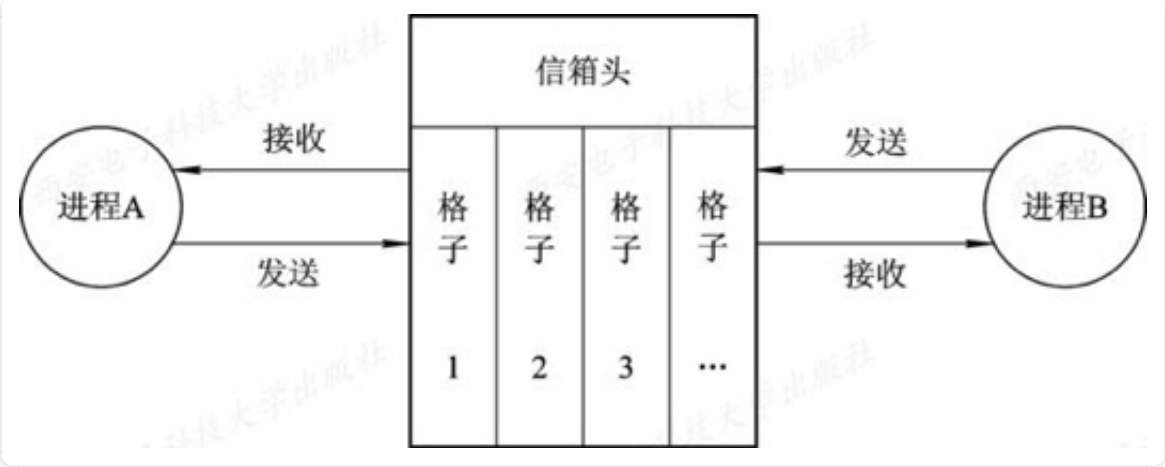
2.信箱通信

（1）信箱的结构

> *信箱定义为一种数据结构。*

在逻辑上，可以将其分为两个部分：

1. 信箱头
2. 信箱体



（2）信箱通信原语

系统为邮箱通信提供了若干条原语，分别用于：

- 邮箱的创建和撤消
- 消息的发送和接收

（3）信箱的类型

> *邮箱可由操作系统创建，也可由用户进程创建，创建者是邮箱的拥有者。*

据此，可把邮箱分为以下三类：

- 私用邮箱
- 公用邮箱
- 共享邮箱

2.7 线程(Threads)

引入线程的优点：

- 使多个程序能并发执行，以提高资源利用率和系统吞吐量。
- 减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性。

回顾进程的两个基本属性：

1. 进程是一个可拥有资源的独立单位。一个进程要能独立运行，它必须拥有一定的资源，包括用于存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；
2. 进程同时又是一个可独立调度和分派的基本单位，一个进程要能独立运行，它还必须是一个可独立调度和分派的基本单位。每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，也可以根据其PCB中的信息，对进程进行调度，还可将断点信息保存在其PCB中。反之，再利用进程PCB中的信息来恢复进程运行的现场。正是由于进程有这两个基本属性，才使进程成为一个能独立运行的基本单位，从而也就构成了进程并发执行的基础。

程序并发执行所需付出的时空开销，系统必须进行以下的一系列操作：

1. 创建进程，系统在创建一个进程时，必须为它分配其所必需的、除处理

机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB；
2. 撤消进程，系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤消PCB；
3. 进程切换，对进程进行上下文切换时，需要保留当前进程的CPU环境，设置新选中进程的CPU环境，因而须花费不少的处理机时间。

有不少研究操作系统的学者们想到，要设法将进程的上述两个属性分开，由OS分开处理，亦即并不把作为调度和分派的基本单位也同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之施以频繁的切换。正是在这种思想的指导下，形成了线程的概念。

2.7.1 线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销
6. 支持多处理机系统

2.7.2 线程的状态和线程控制块

1.线程运行的三个状态

(1)执行状态

表示线程已获得处理机而正在运行；

(2)就绪状态

指线程已具备了各种执行条件，只须再获得CPU便可立即执行；

(3)阻塞状态

指线程在执行中因某事件受阻而处于暂停状态。（如键盘读入数据）。

2.线程控制块TCB

> *所有用于控制和管理线程的信息记录在线程控制块中。*

3.多线程OS中的进程属性

OS支持在一个进程中的多个线程能并发执行，但此时的进程就不再作为一个执行的实体。

多线程OS中的进程有以下属性：

- 进程是一个可拥有资源的基本单位
- 多个线程可并发执行
- 进程已不是可执行的实体　　## 2.7.3 线程的实现 ### 1.实现方式 >线程已在许多系统中实现，但各系统的实现方式并不完全相同。在有的系统中，特别是一些数据库管理系统，如infomix所实现的是用户级线程；而另一些系统(如Macintosh和OS/2操作系统)所实现的是内核支持线程；还有一些系统如Solaris操作系统，则同时实现了这两种类型的线程。

内核支持线程KST(Kernel Supported Threads)

> *在OS中的所有进程，无论是系统进程还是用户进程，都是在操作系统内核的支持下运行的，是与内核紧密相关的。而内核支持线程KST同样也是在内核的支持下运行的，它们的创建、阻塞、撤消和切换等，也都是在内核空间实现的。为了对内核线程进行控制和管理，在内核空间也为每一个内核线程设置了一个线程控制块，内核根据该控制块而感知某线程的存在，并对其加以控制。当前大多数OS都支持内核支持线程。*

四个主要优点：

1. 在多处理器系统中，内核能够同时调度同一进程中的多个线程并行执行
2. 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程
3. 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小
4. 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率

用户级线程ULT(User Level Threads)

> *用户级线程是在用户空间中实现的。对线程的创建、 撤消、同步与通信等功能，都无需内核的支持，即用户级线程是与内核无关的。*

> *在一个系统中的用户级线程的数目可以达到数百个至数千个。由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无需内核的帮助，因而内核完全不知道用户级线程的存在。*

优点：

1. 线程切换不需要转换到内核空间
2. 调度算法可以是进程专用的
3. 用户级线程的实现与OS平台无关，因为对于线程管理的代码是属于用户程序的一部分，所有的应用程序都可以对之进行共享。

缺点：

1. 系统调用的阻塞问题。在基于进程机制的OS中，大多数系统调用将使进程阻塞，因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且，进程内的所有线程会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。
2. 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点，内核每次分配给一个进程的仅有一个CPU，因此，进程中仅有一个线程能执行，在该线程放弃CPU之前，其它线程只能等待。

组合方式

> *在组合方式ULT/KST线程系统中，内核支持多个内核支持线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。*

多线程模型

2.具体实现

内核支持线程的实现

在仅设置了内核支持线程的OS中，一种可能的线程控制方法是，系统在创建一个新进程时，便为它分配一个任务数据区PTDA(Per Task Data Area)，其中包括若干个线程控制块TCB空间，如图2–19所示。

图2–19　任务数据区空间

用户级线程的实现

运行时系统(Runtime System)

> *实质上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。正因为有这些函数，才能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。*

内核控制线程

这种线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。LWP也可以共享进程所拥有的资源。LWP可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只须将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。这种线程实现方式就是组合方式。

图2–20　利用轻型进程作为中间系统

3.线程的创建和终止

线程的创建

应用程序在启动时，通常仅有一个线程在执行，人们把线程称为“初始化线程”，它的主要功能是用于创建新线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程的创建函数执行完后，将返回一个线程标识符供以后使用。

线程的终止

当一个线程完成了自己的任务(工作)后，或是线程在运行中出现异常情况而须被强行终止时，由终止线程通过调用相应的函数(或系统调用)对它执行终止操作。但有些线程(主要是系统线程)，它们一旦被建立起来之后，便一直运行下去而不被终止。在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。