在程序顺序执行时，具有这样三个特征：①**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。②**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。③**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。

为了使参与并发执行的每个程序（含数据）都能独立地运行，在操作系统中必须为之分配一个专门的数据结构，称为**进程控制块**（Process Control Block，PCB）。系统利用 PCB 来描述进程的基本情况和活动过程，进而控制和管理进程。这样，由**程序段**、**相关的数据段**和 **PCB** 三部分便构成了**进程实体**（又称**进程映像**）。一般情况下，我们把进程实体就简称为进程，例如，所谓**创建进程，实质上是创建进程实体中的 PCB**，而**撤销进程，实质上是撤销进程的 PCB**。

进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

进程和程序是两个截然不同的概念，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：

（1）**动态性**。进程的实质是进程实体的执行过程，因此，动态性就是进程的最基本的特性。动态性还表现在：“它由创建而产生，由调度而执行，由撤销而消亡。”可见，进程实体有一定的生命期，而程序只是一组有序指令的集合，并存放于某种介质上，其本身并不具有活动的含义，因而是静态的。

（2）**并发性**。是指多个进程实体同存于内存中，且能在一段时间内同时运行。引入进程的目的也正是为了使其进程实体能和其它进程实体并发执行。因此，并发性是进程的另一重要特征，同时也成为 OS 的重要特征。而程序（没有建立 PCB）是不能参与并发执行的。

（3）**独立性**。在传统的 OS 中，独立性是指进程实体是一个能独立运行、独立获得资源和独立接受调度的基本单位。凡未建立 PCB 的程序都不能作为一个独立的单位参与运行。

（4）**异步性**，是指进程是按异步方式运行的，即按各自独立的、不可预知的速度向前推进。正是源于此因，才导致了传统意义上的程序若参与并发执行，会产生其结果的不可再现性。为使进程在并发运行时虽具有异步性，但仍能保证进

程并发执行的结果是可再现的，在 OS 中引入了进程的概念，并且配置相应的同步机制。

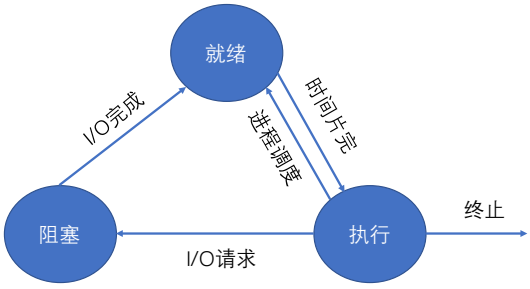
由于多个进程在并发执行时共享系统资源，致使它们在运行过程中呈现间断性的运行规律，所以进程在其生命周期内可能具有多种状态。一般而言，每个进程至少应处于以下三种基本状态之一：

（1）**就绪**（Ready）状态。这是指进程已处于准备好运行的状态，即进程已分配到除 CPU 以外的所有必要资源后，只要再获得 CPU，便可立即执行。如果系统中有许多处于就绪状态的进程，通常将它们按一定的策略排成一个队列，称该队列为就绪队列。

（2）**执行**（Running）状态。这是指进程已获得 CPU，其程序正在执行的状态。对任何一个时刻而言，在单处理机系统中，只有一个进程处于执行状态，而在多处理机系统中，则有多个进程处于执行状态。

（3）**阻塞**（Block）状态。这是指正在执行的进程由于发生某事件（如 I/O 请求、申请缓冲区失败等）暂时无法继续执行时的状态，亦即进程的执行受到阻塞。此时引起进程调度，OS 把处理机分配给另一个就绪进程，而让受阻进程处于暂停状态，一般将这种暂停状态称为阻塞状态，有时也称为等待状态或封锁状态。通常系统将处于阻塞状态的进程也排成一个队列，称该队列为阻塞队列。实际上，在较大的系统中，为了减少队列操作的开销，提高系统效率，根据阻塞原因的不同，会设置多个阻塞队列。

进程在运行过程中会经常发生状态的转换。例如，处于就绪状态的进程，在调度程序为之分配了处理机之后便可执行，相应地，其状态就由就绪态转变为执行态；正在执行的进程（当前进程）如果因分配给它的时间片已完而被剥夺处理机暂停执行时，其状态便由执行转变为就绪；如果因发生某事件，致使当前进程的执行受阻（例如进程访问某临界资源，而该资源正被其它进程访问时），使之无法继续执行，则该进程状态将由执行转变为阻塞。



为了满足进程控制块对数据及操作的完整性要求以及增强管理的灵活性，通

常在系统中又为进程引入了两种常见的状态：创建状态和终止状态。

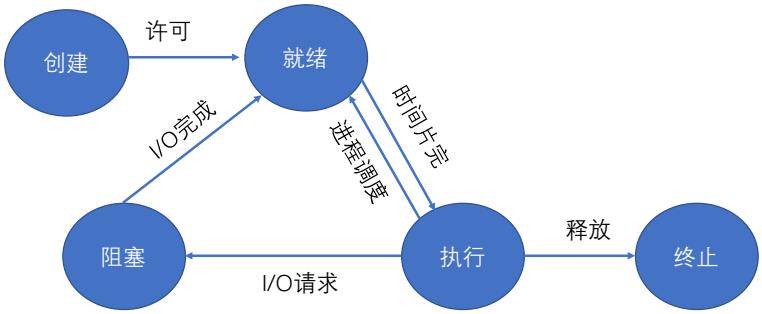
（1）**创建**状态

进程由创建而产生。创建一个进程是个很复杂的过程，一般要通过多个步骤才能完成：如首先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；然后为该进程分配运行时所必须的资源；最后，把该进程转入就绪状态并插入就绪队列之中。但如果进程所需的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，于是把进程所处的状态称为创建状态。

引入创建状态是为了保证**进程的调度必须在创建工作完成后进行**，以确保对进程控制块操作的完整性。同时，创建状态的引入也**增加了管理的灵活性**，OS 可以根据系统性能或主存容量的限制推迟新进程的提交（创建状态）。**对于处于创建状态的进程，当其获得了所需的资源以及对其 PCB 的初始化工作完成后，便可由创建状态转入就绪状态**。

（2）**终止**状态

进程的终止也要通过两个步骤：首先，是**等待操作系统进行善后处理**，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被执行，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。



为了便于系统描述和管理进程的运行，在 OS 的核心为每个进程专门定义了一个数据结构——**进程控制块 PCB**（Process Control Block）。PCB 作为进程实体的一部分，记录了操作系统所需的，用于描述进程的当前情况以及管理进程运行的全部信息，是操作系统中最重要的记录型数据结构。

PCB 的作用是使一个在多道程序环境下不能独立运行的程序（含数据）成为

一个能独立运行的基本单位，一个能与其他进程并发执行的进程。下面对 PCB 的具体作用作进一步的阐述：

（1）**作为独立运行基本单位的标志**。当一个程序（含数据）配置了 PCB 后，就表示它已是一个能在多道程序环境下独立运行的、合法的基本单位，也就具有取得 OS 服务的权利，如打开文件系统中的文件，请求获得系统中的 I/O 设备，以及与其它相关进程进行通信等。因此，当系统创建一个新进程时，就为它建立了一个 PCB。进程结束时又回收其 PCB，进程于是也随之消亡。系统是通过 PCB 感知进程的存在的。事实上，PCB 已成为进程存在于系统中的唯一标志。

（2）能实现间断性运行方式。在多道程序环境下，程序是采用停停走走间断性的运行方式运行的。当进程因阻塞而暂停运行时，它必须保留自己运行时的 CPU 现场信息，再次被调度运行时，还需要恢复其 CPU 现场信息。在有了 PCB 后，系统就可**将 CPU 线程信息保存在被中断进程的 PCB 中，供该进程再次被调度执行时恢复 CPU 现场时使用**。由此，可再次明确，在多道程序环境下，作为传统意义上的静态程序，因其并不具有保护或保存自己运行现场的手段，无法保证其运行结果的可再现性，从而失去运行的意义。

（3）**提供进程管理所需要的信息。当调度程序调度到某进程运行时，只能根据该进程 PCB 中记录的程序和数据在内存或外存中的始址指针，找到相应的程序和数据**；在进程运行过程中，当需要访问文件系统中的文件或 I/O 设备时，也都需要借助于 PCB 中的信息。另外，还可**根据 PCB 中的资源清单了解到该进程所需的全部资源**等。可见，在进程的整个生命期中，操作系统总是根据 PCB 实施对进程的控制和管理。

（4）**提供进程调度所需要的信息**。只有处于就绪状态的进程才能被调度执行，而在 PCB 中就提供了进程处于何种状态的信息。如果进程处于就绪状态，系统便将它插入到进程就绪队列中，等待着调度程序的调度；另外在进行调度时往往还需要了解进程的其他信息，如在优先级调度算法中，就需要知道进程的优先级。在有些较为公平的调度算法中，还需要知道进程的等待时间和已执行的时间等。

（5）**实现与其它进程的同步与通信**。进程同步机制是用于实现诸进程的协调运行的，在采用信号量机制时，它要求在每个进程中都设置有相应的用于同步的信号量。在 PCB 中还具有用于实现进程通信的区域或通信队列指针等。

现代操作系统一般将 OS 划分为若干层次，再将 OS 的不同功能分别设置在不同的层次中。通常将一些与硬件紧密相关的模块（如中断处理程序等）、各种

常用设备的驱动程序以及运行频率较高的模块（如时钟管理、进程调度和许多模块所公用的一些基本操作），都安排在紧靠硬件的软件层次中，将它们常驻内存，即通常被称为的 OS 内核。这种安排方式的目的在于两方面：一是便于对这些软件进行保护，防止遭受其他应用程序的破坏；二是可以提高 OS 的运行效率。

相对应的是，为了防止 OS 本身及关键数据（如 PCB 等）遭受到应用程序有意或无意的破坏，通常也**将处理机的执行状态分成系统态和用户态两种**：①**系统态**：又称为管态，也称为内核态。它具有较高的特权，能执行一切指令，访问所有寄存器和存储区，传统的 OS 都在系统态运行。②**用户态**：又称为目态。它是具有较低特权的执行状态，仅能执行规定的指令，访问指定的寄存器和存储区。一般情况下，应用程序只能在用户态运行，不能去执行 OS 指令及访问 OS 区域，这样可以防止应用程序对 OS 的破坏。

总体而言，不同类型和规模的 OS，它们的内核所包含的功能间存在着一定的差异，但大多数 OS 内核都包含以下两大方面的功能：

（1）支撑功能

该功能是提供给 OS 其它众多模块所需要的一些基本功能，以便支撑这些模块工作。其中三种最基本的支撑功能是：**中断处理**、**时钟管理**和**原语操作**。

①**中断处理**。中断处理是内核最基本的功能，是整个操作系统赖以活动的基础，OS 中许多重要的活动，如各种类型的系统调用、键盘命令的输入、进程调度、设备驱动等，无不依赖于中断。通常，为减少处理机中断的时间，提高程序执行的并发性，内核在对中断进行“有限处理”后，便转入相关的进程，由这些进程继续完成后续的处理工作。

②**时钟管理**。时钟管理是内核的一项基本功能，在 OS 中的许多活动都需要得到它的支撑，如在时间片轮转调度中，每当时间片用完时，便由时钟管理产生一个中断信号，促使调度程序重新进行调度。同样，在实时系统中的截止时间控制、批处理系统中的最长运行时间控制等，也无不依赖于时钟管理功能。

③**原语操作**。所谓原语，就是由若干条指令组成的，用于完成一定功能的一个过程。它与一般过程的区别在于：它们是“原子操作”。所谓原子操作是指，一个操作中的所有动作要么全做，要么全不做。换言之，它是一个不可分割的基本单位。因此，原语在执行过程中不允许被中断。原子操作在系统态下执行，常驻内存。在内核中可能有许多原语，如用于链表进行操作的原语、用于实现进程同步的原语等。

（2）资源管理功能

①**进程管理**。在进程管理中，或者由于各个功能模块的运行频率较高，如进

程的调度与分派、进程的创建与撤销等；或者由于它们为多种功能模块所需要，如用于实现进程同步的原语、常用的进程通信原语等。通常都将它们放在内核中，以提高 OS 的性能。

②**存储器管理**。存储器管理软件的运行频率也比较高，如用于实现将用户空间的逻辑地址变换为内存空间的物理地址的地址转换机构、内存分配与回收的功能模块以及实现内存保护和对换功能的模块等。通常也将它们放在内核中，以保存存储器管理具有较高的运行速度。

③**设备管理**。由于设备管理与硬件（设备）紧密相关，因此其中很大部分也都设置在内核中。如各类设备的驱动程序、用于缓和 CPU 与 I/O 速度不匹配矛盾的缓冲管理、用于实现设备分配和设备独立性功能的模块。

进程的创建过程：

（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。这些资源或从操作系统或仅从其父进程获得。新进程对这些资源的需求详情一般也要提前告知操作系统或其父进程。例如，为新进程的程序和数据以及用户栈分配必要的内存空间时，操作系统必须知道新进程所需内存的大小：①对于批处理作业，其大小可在用户提出创建进程要求时提供；②若是为应用进程创建子进程，也应是在该进程提出创建进程的请求中给出所需内存的大小；③对于交互型作业，用户可以不给出内存要求而由系统分配一定的空间；如果新进程要共享某个已在内存的地址空间（即已装入内存的共享段），则必须建立相应的链接。

（3）**初始化进程控制块（PCB）**。PCB 的初始化包括：①**初始化标识信息**，将系统分配的标识符和父进程标识符填入新 PCB 中；②**初始化处理机状态信息**，使程序计数器指向程序的入口地址，使栈指针指向栈顶；③**初始化处理机控制信息**，将进程的状态设置为就绪状态或静止就绪状态，对于优先级，通常是将它设置为最低优先级，除非用户以显式方式提出高优先级要求。

（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。

进程的阻塞过程：

正在执行的进程，如果发生了引起进程阻塞的事件，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。

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。

block 原语和 wakeup 原语是一对作用刚好相反的原语。在使用它们时，必须成对使用，即如果在某进程中调用了阻塞原语，则必须在与之相合作的、或其它相关的进程中安排一条相应的唤醒原语，以便能唤醒被阻塞进程；否则，阻塞进程将会因不能被唤醒而永久地处于阻塞状态，再无机会继续运行。

进程的挂起：

当系统中出现了引起进程挂起的事件时，OS 将利用挂起原语 suspend 将指定进程或处于阻塞状态的进程挂起。suspend 的执行过程是：首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞；为了方便用于或父进程考查该进程的运行情况，而把该进程的 PCB 复制到某指定的内存区域；最后，若被挂起的进程正在执行，则转向调度程序重新调度。

进程的激活：

当系统中发生激活进程的事件时，OS 将利用激活原语 active，将指定进程激活。激活原语先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，便将之改为活动就绪；若为静止阻塞，便将之改为活动阻塞。假如采用的是抢占调度策略，则每当有静止就绪进程被激活而插入就绪队列时，便应检查是否要进行重新调度，即由调度程序将被激活的进程与当前进程两者的优先级进行比较，如果被激活进程的优先级低，就不必重新调度；否则，**立即剥夺当前进程的运行，把处理机分配给刚刚被激活的进程**。

在临界区前面增加一段用于进行检查临界资源是否正被其他进程访问的代码，这段代码称为**进入区**（entry section）。相应的，在临界区后面也要加上一段称为**退出区**（exit section）的代码。

虽然信号量机制是一种既方便、又有效的进程同步机制，但每个要访问临界资源的进程都必须自备同步操作 wait(S)和 signal(S)。这就使大量的同步操作分散在各个进程中。这不仅给系统的管理带来了麻烦，而且还会因同步操作的使用不当而导致系统死锁。这样，在解决上述问题的过程中，便产生了一种新的进程同步工具——**管程**（Monitors）。

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。因此，可以利用共享数据结构抽象地表示系统中的共享资源，并且将对该共享数据结构实施的特定操作定义为一组过程。**进程对共享资源的申请、释放和其它操作必须通过这组过程，间接地对共享数据结构实现操作。对于请求访问共享资源的诸多并发进程，可以根据资源的情况接受或阻塞，确保每次仅有一个进程进入管程，执行这组管程，执行这组过程，使用共享资源，达到对共享资源所有访问的统一管理，有效地实现进程互斥。**

代表共享资源的数据结构以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序共同构成了一个操作系统的资源管理模块，我们称之为**管程**。Hansan 为管程所下的定义是："**一个管程定义了一个数据结构和能为并发进程所指向（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据。**

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**。

实际上，管程中包含了面向对象的思想，它将表征共享资源的数据结构及其对数据结构操作的一组过程，包括同步机制，都集中并封装在一个对象内部，隐藏了实现细节。**封装于管程内部的数据结构仅能被封装于管程内部的过程所访问，任何管程外的过程都不能访问它**；反之，**封装于管程内部的过程也仅能访问管程内的数据结构**。所有进程要访问临界资源时，都只能通过管程间接访问，而管程每次只准许一个进程进入管程，执行管程内的过程，从而实现了进程互斥。

管程是一种程序设计语言的结构成分，它和信号量有同等的表达能力，从语言的角度看，管程主要有以下特性：①模块化，即管程是一个基本程序单位，可以单独编译；②抽象数据类型，指管程中不仅有数据，而且有对数据的操作；③

信息掩蔽，指管程中的数据结构只能被管程中的过程访问，这些过程也是在管程内部定义的，供管程外的进程调用，而管程中的数据结构以及过程（函数）的具体实现外部不可见。

管程和进程不同：①虽然二者都定义了数据结构，但**进程定义的是私有数据结构 PCB，管程定义的是公共数据结构**，如消息队列等；②二者都存在对各自数据结构上的操作，但**进程是由顺序程序执行有关操作**，而**管程主要是进行同步操作和初始化操作**；③**设置进程的目的在于实现系统的并发性，而管程的设置则是解决共享资源的互斥使用问题**；④进程通过调用管程中的过程对共享数据结构实行操作，该过程就如通常的子程序一样被调用，因而**管程为被动工作方式，进程则为主动工作方式**；⑤**进程之间能并发执行，而管程则不能与其调用者并发**；⑥**进程具有动态性，由“创建”而诞生，由“撤销”而消亡，而管程则是操作系统中的一个资源管理模块，供程序调用**。

随着 OS 的发展，用于进程之间实现通信的机制也在发展，并已由早期的低级进程通信机制发展为能传送大量数据的高级通信工具机制。目前，高级通信机制可归结为四大类：**共享存储器系统**、**管道通信系统**、**消息传递系统**以及**客户机-服务器系统**。

共享存储器系统

在共享存储器系统中，相互通信的进程共享某些数据结构或共享存储区，进程之间能够通过这些空间进行通信。据此，又可把它们分成以下两种类型：

（1）基于**共享数据结构**的通信方式。在这种通信方式中，要求诸进程公用某些数据结构，借以实现诸进程间的信息交换，如在生产者-消费者问题中的有界缓冲区。操作系统仅提供共享存储器，由程序员负责对公用数据结构的设置及对进程间同步的处理。这种通信方式仅适于传递相对少量的数据，通信效率低下，属于低级通信。

（2）基于**共享存储区**的通信方式。为了传输大量数据，在内存中划出了一块共享存储区域，诸进程可通过对该共享区的读或写交换信息，实现通信，数据的形式和位置甚至访问控制都是由进程负责，而不是 OS。这种通信方式属于高级通信。需要通信的进程在通信前，先向系统申请获得共享存储区的一个分区，并将其附加到自己的地址空间中，便可对其中的数据进行正常读、写，读写完成或不再需要时，将其归还给共享存储区。

管道通信系统

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 4KB）的数据写入 pipe，便去睡眠等待，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。

消息传递系统

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为单位，将通信数据封装在消息中，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。

该方式隐藏了通信实现细节，使通信过程对用户透明化，降低了通信程序设计的复杂性和错误率，成为当前应用最为广泛的一类进程间通信的机制。例如：在计算机网络中，消息又称为报文；在微内核操作系统中，微内核与服务器之间的通信无一例外都是采用了消息传递机制；由于该机制能很好地支持多处理机系统、分布式系统和计算机网络，因此也成为这些领域最主要的通信工具。

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**，是指发送进程利用 OS 所提供的发送原语，直接把消息发送给目标进程；

（2）**间接通信方式**，是指发送和接收进程，都通过共享中间实体（称为邮箱）的方式进行消息的发送和接收，完成进程间的通信。

客户机-服务器系统

前面所述的共享内存、消息传递等技术，虽然也可以用于实现不同计算机间进程的双向通信，但客户机-服务器系统的通信机制，在网络环境的各种应用领域已成为当前主流的通信实现机制，其主要的实现方法分为三类：**套接字**、**远程过程调用**和**远程方法调用**。

（1）套接字

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，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。

一个套接字就是一个通信标识类型的数据结构，包含了通信目的地址、通信使用的端口号、通信网络的传输层协议、进程所在的网络地址，以及针对客户或服务器程序提供的不同系统调用（或 API 函数）等，是进程通信和网络通信的基本构件。套接字是为客户/服务器模型而设计的，通常，套接字包括两类：

①基于文件型：通信进程都运行在同一台机器的环境中，套接字是基于本地文件系统支持的，**一个套接字关联到一个特殊的文件**，通信双方通过对这个特殊文件的读写实现通信，其原理类似于前面所讲的管道。

②基于网络型：该类型通常采用的是非对称方式通信，即发送者需要提供接收者命名。通信双方的进程运行在不同主机的网络环境下，被分配了一对套接字，一个属于接收进程（或服务器端），一个属于发送进程（或客户端）。一般地，发送进程（或客户端）发出连接请求时，随机申请一个套接字，主机为之分配一个端口，与该套接字绑定，不再分配给其它进程。接收进程（或服务器端）拥有全局公认的套接字和指定的端口（如 ftp 服务器监听端口为 21，Web 或 http 服务器监听端口为 80），并通过监听端口等待客户请求。因此，任何进程都可以向它发出连接请求和信息请求，以方便进程之间通信连接的建立。接收进程（或服务器端）一旦收到请求，就接受来自发送进程（或客户端）的连接，完成连接，即在主机间传输的数据可以准确地发送到通信进程，实现进程间的通信；当通信结束时，系统通过关闭接收进程（或服务器端）的套接字撤销连接。

套接字的优势在于，**它不仅适用于同一台计算机内部的进程通信，也适用于网络环境中不同计算机间的进程通信**。由于每个套接字拥有唯一的套接字号（也称为套接字标识符），这样系统中所有的连接都持有唯一的一对套接字及端口连接，对于来自不同应用程序进程或网络连接的通信，能够方便地加以区分，确保了通信双方之间逻辑链路的唯一性，便于实现数据传输的并发服务，而且隐藏了通信设施及实现细节，采用统一的接口进行处理。

（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作为远程方法调用。

负责处理远程过程调用的进程有两个，一个是本地客户进程，另一个是远程服务器进程，这两个进程通常也被称为网络守护进程，主要负责在网络间的消息传递，一般情况下，这两个进程都是处于阻塞状态，等待消息。

为了使远程过程调用看上去与本地过程调用一样，即希望实现 RPC 的透明性，使得调用者感觉不到此次调用的过程是在其他主机（远程）上执行的，RPC 引入一个**存根**（stub）的概念：在本地客户端，每个能够独立运行的远程过程都拥有一个**客户存根**（client stubborn），本地进程调用远程过程实际是调用该过程关联的存根；与此类似，在每个远程进程所在的服务器端，其所对应的实际可执行进程也存在一个**服务器存根**（stub）与其关联。本地客户存根与对应的远程服务器存根一般也是处于阻塞状态，等待消息。

实际上，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端的消息传递，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。

这样，本地调用者再次获得控制权，并且得到了所需的数据，得以继续运行。显然，上述步骤的主要作用在于：**将客户过程的本地调用转化为客户存根，再转化为服务器过程的本地调用**，对客户与服务器来说，它们的中间步骤是不可见的，因此，调用者在整个过程中并不知道该过程的执行是在远程，而不是在本地。

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 PCB 中的信息来恢复进程运行的现场。正是由于进程这两个基本属性，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。

由于线程具有许多传统进程所具有的特征，所以又称之为**轻型进程**（Light-Weight Process）或**进程元**，相应地，把传统进程称为**重型进程**（Heavy-Weight Process）。它相当于只有一个线程的任务。下面我们从调度性、并发性、系统开销和拥有资源等方面对线程和进程进行比较。

（1）调度的基本单位

在传统的 OS 中，进程是作为独立调度和分派的基本单位，因而进程是能独立运行的基本单位。在每次被调度时，都需要进行上下文切换，开销较大。而在引入线程的 OS 中，已把线程作为调度和分派的基本单位，因而**线程是能独立运行的基本单位。当线程切换时，仅需保存和设置少量寄存器内容，切换代价远低于进程**。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，必然会引起进程的切换。

（2）并发性

在引入线程的 OS 中，不仅**进程之间可以并发执行**，而且在**一个进程的多个线程之间亦可并发执行**，甚至还允许在一个进程中的所有线程都能并发执行。同样，不同进程中的线程也能并发执行。这使得 OS 具有更好的并发性，从而能更加有效地提高系统资源的利用率和系统的吞吐量。

此外，有的应用程序需要执行多个相似的任务。例如，一个网络服务器经常会接到许多客户的请求，如果仍采用传统的单线程的进程来执行该任务，则每次只能为一个客户服务。但如果在一个进程中可以设置多个线程，将其中的一个专用于监听客户的请求，则每当有一个客户请求时，便立即创建一个线程来处理该客户的请求。

（3）拥有资源

进程可以拥有资源，并作为系统中拥有资源的一个基本单位。然而，**线程本**

身并不拥有系统资源，而是仅有一点必不可少的、能保证独立运行的资源。比如，在每个线程中都应具有一个用于控制线程运行的线程控制块 TCB、用于指示被执行指令序列的程序计数器、保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。

线程除了拥有自己的少量资源外，还允许多个线程共享该进程所拥有的资源，这首先表现在：属于同一进程的所有线程都具有相同的地址空间，这意味着，线程可以访问该地址空间中的每一个虚地址；此外，还可以访问进程所拥有的资源，如已打开的文件、定时器、信号量机构等的内存空间和它所申请到的 I/O 设备等。

（4）独立性

在同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。这是因为，为防止进程之间彼此破坏和干扰，每个进程都拥有一个独立的地址空间和其它资源，除了共享全局变量外，不允许其它进程的访问。但是同一进程中的不同线程往往为了提高并发性以及进行相互之间的合作而创建的，它们共享进程的内存地址空间和资源，如每个线程都可以访问它们所属进程地址空间中的所有地址，如一个线程的堆栈可以被其它线程读、写，甚至完全消除。由一个线程打开的文件可以供其它线程读、写。

（5）系统开销

在创建或撤销进程时，系统都要为之分配和回收进程控制块、分配或回收其它资源，如内存空间和 I/O 设备等。OS 为此所付出的开销，明显大于线程创建和撤销时所付出的开销。此外，由于一个进程中的多个线程具有相同的地址空间，线程之间的同步和通信也比进程的简单。因此，在一些 OS 中，线程的切换、同步和通信都无需操作系统内核的干预。

（6）支持多处理机系统

在多处理机系统中，对于传统的进程，即单线程进程，不管有多少处理机，该进程只能运行在一个处理机上。但**对于多线程进程，就可以将一个进程中的多个线程分配到多个处理机上，使它们并行执行，这无疑将加速进程的完成**。因此，现代多处理机 OS 都无一例外地引入了多线程。

如同每个进程有一个进程控制块一样，系统也为每个线程配置了一个线程控制块 TCB，将所有用于控制和管理线程的信息记录在线程控制块中。线程控制块通常有这样几项：①线程标识符，为每个线程赋予一个唯一的线程标识符；②一组寄存器，包括程序计数器 PC、状态寄存器和通用寄存器的内容；③线程运行状态，用于描述线程正处于何种运行状态；④优先级，描述线程执行的优先程度；

⑤线程专有存储区，用于线程切换时存放现场保护信息，和与该线程相关的统计信息等；⑥信号屏蔽，即对某些信号加以屏蔽；⑦堆栈指针，在线程运行时，经常会进行过程调用，而过程的调用通常会出现多重嵌套的情况，这样，就必须将每次过程调用中所使用的局部变量以及返回地址保存起来。为此，应为每个线程设置一个堆栈，用它来保存局部变量和返回地址。相应地，在 TCB 中，也须设置两个指向堆栈的指针：**指向自己堆栈的指针**和**指向核心栈的指针**。**前者是指当线程运行在用户态时，使用用户自己的用户栈来保存局部变量和返回地址，后者是指当线程运行在核心态时使用系统的核心栈**。

通常在多线程 OS 中的进程都包含了多个线程，并为它们提供资源。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：

（1）**进程是一个可拥有资源的基本单位**。在多线程 OS 中，进程仍是作为小系统资源分配的基本单位，任一进程所拥有的资源都包括：用户的地址空间、实现进程（线程）间同步和通信的机制、已打开的文件和已申请到的 I/O 设备，以及一张由核心进程维护的地址映射表，该表用于实现用户程序的逻辑地址到其内存物理地址的映射。

（2）多个线程可并发执行。通常一个进程都含有若干个相对独立的线程，其数目可多可少，但**至少要有一个线程**。由进程为这些（个）线程提供资源及运行环境，使它们能并发执行。在 OS 中的所有线程都只能属于某一个特定进程。实际上，现在把传统进程的执行方法称为单线程方法。如传统的 UNIX 系统能支持多用户进程，但只支持单线程方法。反之，将每个进程支持多个线程执行的方法称为多线程方法。如 Java 的运行环境是单进程多线程的，Windows 2000、Solaris、Match 等采用的则是多进程多线程的方法。

（3）**进程已不是可执行的实体**。在多线程 OS 中，是把线程作为独立运行（或称调度）的基本单位。此时的进程已不再是一个基本的可执行实体。虽然如此，进程仍具有与执行相关的状态。例如，**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**。此外，对进程所施加的与进程状态有关的操作也对其线程起作用。例如，**在把某个进程挂起时，该进程中的所有线程也都将被挂起**；又如，**在把某进程激活时，属于该进程的所有线程也都将被激活**。

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、撤销和切换等，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。

这种线程实现方式主要有四个主要优点：

（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**。

内核支持线程的主要缺点是：**对于用户的线程切换而言，其模式切换的开销较大，在同一个进程中，从一个线程切换到另一个线程时，需要从用户态转到核心态进行，这是因为用户进程的线程在用户态运行，而线程调度和管理是在内核实现的，系统开销较大**。

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、撤销、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。

值得说明的是，对于设置了用户级线程的系统，其调度仍是以进程为单位进行的。在采用轮转调度算法时，各个进程轮流执行一个时间片，这对诸进程而言貌似是公平的。但假如在进程 A 中包含了一个用户级线程，而在另一个进程 B 中含有 100 个用户级线程，这样，进程 A 中线程的运行时间将是进程 B 中各线程运行时间的 100 倍；相应地，其速度要快上 100 倍，因此说实质上并不公平。

假如系统中设置的内核支持线程，则调度便是以线程为单位进行的。在采用轮转法调度时，是各个线程轮流执行一个时间片。同样假定进程 A 中只有一个内核支持线程，而在进程 B 中有 100 个内核支持线程。此时进程 B 可以获得的 CPU 时间是进程 A 的 100 倍，且进程 B 可使 100 个系统调用并发工作。

使用用户级线程方式有许多优点：

（1）线程切换不需要转换到内核空间。对一个进程而言，其所有线程的管理数据结构均在该进程的用户空间中，管理线程切换的线程库也在用户地址空间运行，因此**进程不必切换到内核方式来做线程管理，从而节省了模式切换的开销**。

（2）调度算法可以是进程专用的。在不干扰 OS 调度的情况下，不同的进程可以根据自身需要选择不同的调度算法。对自己的线程进行管理和调度，而与 OS 的低级调度算法是无关的。

（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与之进行共享。因此，用户级线程甚至可以在不支持线程机制的操作系统平台上实现。

而用户级线程方式的主要缺点则在于：

（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

有些 OS 把用户级线程和内核支持线程两种方式进行组合，提供了组合方式 ULT/KST 线程。在组合方式线程系统中，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。一些内核支持线程对部分或全部内核支持线程进行多路复用，程序员可按应用需要和机器配置，对内核支持线程数目进行调整，以达到较好效果。组合方式线程中，**同一个进程内的多个线程可以同时在多处理器上并行执行，而且在阻塞一个线程时不需要将整个进程阻塞**。所以，组合方式多线程机制能够结合 KST 和 ULT 两者的优点，并克服了其各自的不足。由于用户级线程和内核支持线程连接方式的不同，从而形成了三种不同的模型：多对一模型、一对一模型和多对多模型：

（1）**多对一模型**，即**将用户线程映射到一个内核控制线程**。这些用户线程一般属于一个进程，运行在该进程的用户空间，对这些线程的调度和管理也是在该进程的用户空间中完成。仅当用户线程需要访问内核时，才将其映射到一个内核控制线程上，但每次只允许一个线程进行映射。该模型的主要优点是线程管理的

开销小，效率高；其主要缺点在于，如果一个线程在访问内核时发生阻塞，则整个进程都会被阻塞；此外，在任一时刻，只有一个线程能够访问内核，多个线程不能同时在多个处理机上运行。

（2）**一对一模型**，即**将每一个用户级线程映射到一个内核支持线程**。为每一个用户线程都设置一个内核控制线程与之连接。该模型的主要优点是：**当一个线程阻塞时，允许调度另一个线程运行，所以它提供了比多对一模型更好的并发功能**。此外，**在多处理机系统中，它允许多个线程并行地在多处理机系统上**。该模型的唯一缺点是：**每创建一个用户线程，相应地就需要创建一个内核线程，开销较大，因此需要限制整个系统的线程数**。**Java 线程模型**。

（3）**多对多模型**，即**将许多用户线程映射到同样数量或更少数量的内核线程上**。内核控制线程的数目可以根据应用进程和系统的不同而变化，可以比用户线程少，也可以与之相同。该模型结合上述两种模型的优点，它可以像一对一模型那样，使一个进程的多个线程并行地运行在多处理机系统上，也可像多对一模型那样，减少线程管理的开销和提高效率。

作业调度算法

先来先服务（first-come first-served，FCFS）调度算法

FCFS 是最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，系统将按照作业到达的先后次序来进行调度，或者说它是优先考虑在系统中等待时间最长的作业，而不管该作业所需执行时间的长短，从后备作业队列中选择几个最先进入该队列的作业，将它们调入内存，为它们分配资源和创建进程。然后把它放入就绪队列。

当在进程调度中采用 FCFS 算法时，每次调度是从就绪的进程队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后，进程调度程序才将处理机分配给其它进程。

顺便说明。FCFS 算法在单处理机系统中已很少作为主调度算法，但经常把它与其它调度算法相结合使用，形成一种更为有效的调度算法。例如，可以在系统中按进程的优先级设置多个队列，每个优先级一个队列，其中每一个队列的调度都基于 FCFS 算法。

短作业优先（short job first，SJF）调度算法

SJF 是以作业的长短来计算优先级，作业越短，其优先级越高。作业的长短

是以作业所要求的运行时间来衡量的。SJF 算法可以分别用于作业调度和进程调度。在把短作业优先调度算法用于作业调度时，它将从外存的作业后备队列中选择若干个估计运行时间最短的作业，优先将它们调入内存运行。

SJF 调度算法较之 FCFS 算法有了明显的改进，但仍然存在不容忽视的缺点：

（1）必须预知作业的运行时间。在采用这种算法时，要先知道每个作业的运行时间，即时是程序员也很难准确估计作业的运行时间，如果估计过低，系统可能按估计的时间终止作业的运行，但此时作业并未完成，故一般都会偏长估计。

（2）对长作业非常不利，长作业的周转时间会明显地增长。更严重的是，该算法完全忽视作业的等待时间，可能使作业等待时间过长，出现饥饿现象。

（3）该调度算法完全未考虑作业的紧迫程度，故不能保证紧迫性作业能得到及时处理。

优先级调度算法（priority-scheduling algorithm，PSA）

我们可以这样来看作业的优先级，对于先来先服务调度算法，作业的等待时间就是作业的优先级，等待时间越长，其优先级越高。对于短作业优先调度算法，作业的长短就是作业的优先级，作业所需运行的时间越短，其优先级越高。但上述两种优先级都不能反映作业的紧迫程度。而在优先级调度算法中，则是基于作业的紧迫程度，由外部赋予作业相应的优先级，调度算法是根据该优先级进行调度的。这样就可以保证紧迫性作业优先运行。优先级调度算法可作为作业调度算法，也可作为进程调度算法。当把该算法用于作业调度时，系统是从后备队列中选择若干个优先级最高的作业转入内存。

高响应比优先调度算法（Highest Response Ratio Next，HRRN）

在批处理系统中，FCFS 算法所考虑的只是作业的等待时间，而忽视了作业的运行时间。而 SJF 算法正好与之相反，只考虑作业的运行时间，而忽视了作业的等待时间。高响应比优先调度算法则是既考虑了作业的等待时间，又考虑作业运行时间的调度算法，因此既照顾了短作业，又不致使长作业的等待时间过长，从而改善了处理机调度的性能。

优先权 = （等待时间 + 要求服务时间） / （要求服务时间）

进程调度算法

轮转调度算法

在分时系统中，最简单的也是较常用的是基于时间片的轮转（round robin，RR）调度算法。该算法采取了非常公平的处理机分配方式，即让就绪队列上的每个进程每次仅允许一个时间片。如果就绪队列上有 n 个进程，则每个进程每次大约都可获得 1 / n 的处理机时间。

多队列调度算法

该算法将系统中的进程就绪队列从一个拆分为若干个，将不同类型或性质的进程固定分配在不同的就绪队列，不同的就绪队列采用不同的调度算法，一个就绪队列中的进程可以设置不同的优先级，不同的就绪队列本身也可以设置不同的优先级。

多队列调度算法由于设置多个就绪队列，因此对每个就绪队列就可以实施不同的调度算法，因此，系统针对不同用户进程的需求，很容易提供多种调度策略。

在多处理机系统中，该算法由于安排了多个就绪队列，因此，很方便**为每个处理机设置一个单独的就绪队列**。这样，不仅对每个处理机的调度可以实施各自不同的调度策略，而且对于一个含有多个线程的进程而言，可以根据其要求将其所有线程分配在一个就绪队列，全部在一个处理机上运行。再者，对于一组需要相互合作的进程或线程而言，也可以将它们分配到一组处理机所对应的多个就绪队列，使得它们能同时获得处理机并行执行。

多级反馈队列（multileveled feedback queue）调度算法

（1）**设置多个就绪队列。在系统中设置多个就绪队列，并为每个队列赋予不同的优先级**。第一个队列的优先级最高，第二个次之，其余队列的优先级逐个降低。**该算法为不同队列中的进程所赋予的执行时间片的大小也各不相同，在优先级愈高的队列中，其时间片就愈小**。例如第二个队列的时间片要比第一个的时间片长一倍，……，第 i + 1 个队列的时间片要比第 i 个的时间片长一倍。

（2）**每个队列都采用 FCFS 算法**。当新进程进入内存后，首先将它放入第一队列的末尾，否则按 FCFS 原则等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。否则，即**它在一个时间片结束时尚未完成，调度程序将其转入第二队列的末尾等待调度；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入将它放入第三队列，……**，依此类推。当进程最后被降到第 n 队列后，**在第 n 队列中便采取按轮转调度方式运行**。

（3）按队列优先级调度。调度程序首先调度最高优先级队列中的诸进程运行，仅当第一队列空闲时才调度第二队列中的进程运行；换言之，**仅当第 1~(i –**

1)所有队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时又有新进程进入任一优先级较高的队列，此时须立即把正在运行的进程放回到第 i 队列的末尾，而把处理机分配给新到的高优先级进程。

在多级反馈队列调度算法中，**如果规定第一个队列的时间片略大于多数人机交互所需处理时间时，便能较好地满足各种类型用户的需要**。

（1）终端型用户。由于终端型用户提交的作业多属于交互型作业，通常较小，系统只要能使这些作业在第一队列规定的时间片内完成，便可使终端型用户感到满意。

（2）短批处理作业用户。对于这类作业，如果可在第一队列中执行完成，便获得与终端型作业一样的响应时间。对于稍长的短作业，也只需在第二和第三队列各执行一时间片完成，其周转时间仍然较短。

（3）长批处理作业用户。对于长作业，它将依次在第 1，2，……，n 个队列中运行，然后再按轮转方式运行，用户不必担心起作业长期得不到处理。

优先级倒置的解决措施：

（1）进程 P3 在进入临界区后 P3 所占用的处理机就不允许被抢占。

（2）当高优先级进程 P1 要进入临界区，去使用临界资源 R，如果已有一个低优先级进程 P3 正在使用该资源，此时一方面 **P1 被阻塞**，另一方面**由 P3 继承 P1 的优先级，并一直保持到 P3 退出临界区**。

在系统中有许多不同类型的资源，其中可以引起死锁的主要是，需要采用**互斥访问**方法的、**不可以被抢占**的资源。如果一组进程中的每一个进程都在等待仅由该组进程中的其它进程才能引发的事件，那么该组进程是**死锁**的。

虽然进程在运行过程中可能会发生死锁，但产生进程死锁是必须具备一定条件的。产生死锁必须同时具备下面四个必要条件，只要其中任一个条件不成立，死锁就不会发生：

（1）**互斥条件**。进程对所分配到的资源进行排它性使用，即在一段时间内，其资源只能被一个进程占用。如果此时还有其它进程请求该资源，则请求进程只能等待，直至占有该资源的进程用毕释放。

（2）**请求和保持条件**。进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

（3）**不可抢占条件**。进程已获得的资源在未使用完之前不能被抢占，只能在

进程使用完时由自己释放。

（4）**循环等待条件**。在发生死锁时，必然存在一个进程-资源的循环链，即进程集合{P0，P1，P2，……，Pn}中的 P0 正在等待一个 P1 占用的资源，P1 正在等待 P2 占用的资源，……，Pn 正在等待已被 P0 占用的资源。

预防死锁的方法是通过**破坏产生死锁的四个必要条件中的一个或几个**，以避免发生死锁。由于互斥条件是非共享设备所必须的，不仅不能改变，还应加以保证，因此主要是破坏死锁的后三个条件。

（1）破坏“请求和保持”条件

①所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源。

②允许一个进程只获得运行初期所需的资源后，便开始运行。进程运行过程中再逐步释放已分配给自己的、且已用毕的全部资源，然后再请求新的所需资源。

（2）破坏“不可抢占”条件

为了能破坏“不可抢占”条件，协议中规定，当一个已经保持了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。这意味着进程已占有的资源会被暂时地释放，或者说是被抢占了，从而破坏了“不可抢占”条件。

该方法实现起来比较复杂，且需付出很大的代价。因为一个不可抢占的资源如打印机、CD 刻录机等在使用一段时间后被抢占，可能会造成进程前一阶段工作的失效，即使是采取了某些防范措施，也还会使进程前后两次运行的信息不连续，这种策略还可能因为反复地申请和释放资源致使进程的执行被无限地推迟，这不仅**延长了进程的周转时间**，而且也**增加了系统开销，降低了系统吞吐量**。

（3）破坏“循环等待”条件

对系统所有资源类型进行线性排序，并赋予不同的序号，并规定每个进程必须按序号递增的顺序请求资源。假如某进程已请求到一些序号较高的资源，后来它又想请求一个序号低的资源时，它**必须先释放所有具有相同和更高序号的资源后，才能申请序号低的资源**。

避免死锁同样是属于事先预防的策略，但并不是事先采取某种限制措施，破坏产生死锁的必要条件，而是**在资源动态分配过程中，防止系统进入不安全状态**，以避免发生死锁。这种方法所施加的限制条件较弱，可能获得较好的系统性能，目前常用此方法来避免发生死锁。

安全状态。在该方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，才可将资源分配给进程，否则，令进程等待。所谓**安全状态**，是指系统按某种进程推进顺序（P1，P2，……，Pn）为每个进程 Pi 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。此时称（P1，P2，……，Pn）为安全序列。如果系统无法找到这样的一个安全序列，则称系统处于不安全状态。虽然并非所有不安全状态都必然会转为死锁状态，但当系统进入不安全状态后，就有可能进入死锁状态。反之，只要系统处于安全状态，系统便不会进入死锁状态。因此，避免死锁的实质在于，**系统在进行资源分配时，应使系统不进入不安全序列**。

银行家算法：每一个新进程在进入系统时，它必须申明在运行过程中，可能需要每种资源类型的最大单元数目，其数目不应超过系统所拥有的资源总量。**当进程请求一组资源时，系统必须首先确定是否有足够的资源分配给该进程**。若有，再**进一步计算在将这些资源分配给该进程后，是否会使系统处于不安全状态**。如果不会，才将资源分配给它，否则让进程等待。

对于资源分配图，所有的简化顺序都将得到相同的不可简化图。同样可以证明：**S 为死锁状态的充分条件是：当且仅当 S 状态的资源分配图是不可完全简化的**。该充分条件被称为**死锁定理**。

对于通用计算机而言，存储层次至少应具有三级：最高层为 **CPU 寄存器**，中间为**主存**，最底层是**辅存**。在较高档的计算机中，还可以根据具体的功能细分为**寄存器**、**高速缓存**、**主存储器**、**磁盘缓存**、**固定磁盘**、**可移动存储介质**等 6 层。**在存储层次中，层次越高（越靠近 CPU），存储介质的访问速度越快，价格也越高，相对所配置的存储容量也越小**。其中，寄存器、高速缓存、主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴，掉电后它们中存储的信息不再存在。而低层的固定磁盘和可移动存储介质则属于设备管理的管辖范畴，它们存储的信息将被长期保存。

在计算机系统的存储层次中，**寄存器**和**主存储器**又被称为**可执行存储器**。对于存放于其中的信息，与存放于辅存中的信息相比较而言，计算机所采用的访问机制是不同的，所需耗费的时间也是不同的。**进程可以在很少的时钟周期内使用一条 load 和 store 指令对可执行存储器进行访问。但对辅存的访问则需要通过 I/O 设备实现**。

主存储器简称**内存**或**主存**，是计算机系统中的主要部件，用于保存进程运行时的程序和数据，也称可执行存储器。通常，**处理机都是从主存储器中取得指令和数据的，并将其所取得的指令放入指令寄存器中，而将其所读取的数据装入数据寄存器中**；或者反之，**将寄存器中的数据存入到主存储器**。

寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，完全能与 CPU 协调工作，但价格却十分昂贵，因此容量不可能做得很大。

高速缓存是现代计算机结构中的一个重要部件，它是**介于寄存器和存储器之间的存储器，主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数，这样可大幅度提高程序执行速度**。高速缓存容量远大于寄存器，而比内存约小两到三个数量级左右，从几十 KB 到几 MB，访问速度快于主存储器。在计算机系统中，**为了缓和内存与处理机速度之间的矛盾，许多地方都设置了高速缓存**。

由于目前磁盘的 I/O 速度远低于对主存的访问速度，为了缓和两者之间在速度上的不匹配，而设置了**磁盘缓存**，主要是用于暂时存放频繁使用的一部分磁盘数据和信息，以减少访问磁盘的次数。但磁盘缓存与高速缓存不同，它本身并不是一种实际存在的存储器，而是**利用主存中的部分存储空间暂时存放从磁盘中读出（或写入）的信息**。主存也可以看作是辅存的高速缓存，因为辅存中的数据必须复制到主存方能使用，反之，数据也必须先存在主存中，才能输出到辅存。

一个文件的数据可能先后出现在不同层次的存储器中，例如，一个文件的数据通常被存储在辅存中（如硬盘），当其需要运行或被访问时，就必须调入主存，也可以暂时存放在主存的磁盘高速缓存中。大容量的辅存常常使用磁盘，磁盘数据经常备份到磁带或可移动磁盘组上，以防止硬盘故障时丢失数据。有些系统自动地把老文件数据从辅存转储到海量存储器中，如磁带上，这样做还能降低存储价格。

用户程序要在系统中运行，必须先将它转入内存，然后将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

（1）**编译**，**由编译程序（Compiler）对用户源程序进行编译，形成若干个目标模块（Object Module）**；

（2）**链接**，**由链接程序（Linker）将编译后形成的一组目标模块以及它们所需要的库函数链接在一起，形成一个完整的装入模块（Load Module）**；

（3）**装入**，**由装入程序（Loader）将装入模块装入内存**。

程序的装入：

（1）**绝对装入方式**（Absolute Loading Mode）

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。用户程序经编译后，将**产生绝对地址（即物理地址）的目标代码**。

（2）**可重定位装入方式**（Relocation Loading Mode）

在多道程序环境下，编译程序不可能预知经编译后所得到的目标模块应放在内存的何处。因此，对于用户程序编译所形成的若干个目标模块，它们的起始地址通常都是从 0 开始的，程序中的其它地址也都是相对于起始地址计算的。此时不可能再用绝对装入方式，而应采用可重定位装入方式，它可以根据内存的具体情况将装入模块装入到内存的适当位置。

（3）**动态运行时的装入方式**（Dynamic Run-time Loading）

动态运行时的装入程序**在把装入模块装入内存后，并不立即把装入模块中的逻辑地址转换为物理地址，而是把这种地址转换推迟到程序真正要执行时才进行**。因此，装入内存后的所有地址都仍是逻辑地址。为使地址转换不影响指令的执行速度，这种方式需要一个**重定位寄存器**的支持。

程序的链接：

源程序经过编译后，可得到一组目标模块。链接程序的功能是将这组目标模块以及它们所需要的库函数装配成一个完整的装入模块。在对目标模块进行链接时，根据进行链接的时间不同，可把链接分成如下三种：

（1）**静态链接**（Static Linking）方式

在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。

（2）**装入时动态链接**（Load-time Dynamic Linking）

将用户源程序编译后所得到的一组目标模块，在装入内存时，采用边装入边链接的链接方式。即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入模块去找出相应的外部目标模块，并将它装入内存。

（3）**运行时动态链接**（Run-time Dynamic Linking）

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往有部分

目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

动态链接方式是对上述装入时链接方式的一种改进。这种链接方式是，将对某些模块的链接推迟到程序执行时才进行。亦即，**在执行过程中，当发现一个被调用模块尚未装入内存时，立即由 OS 去找到该模块，并将之装入内存，将其链接到调用者模块上**。凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅能加快程序的装入过程，而且可节省大量的内存空间。

为了能将用户程序装入内存，必须为它分配一定大小的内存空间。**连续分配方式**是最早出现的一种存储器分配方式，该分配方式为一个用户程序分配一个连续的内存空间，即程序中代码或数据的逻辑地址相邻，体现在内存空间分配时物理地址的相邻。连续分配方式可分为四类：**单一连续分配**、**固定分区分配**、**动态分区分配**以及**动态可重定位分区分配**算法四种方式。

在单道程序环境下，当时的存储器管理方式是把内存分为**系统区**和**用户区**两部分，**系统区仅提供给 OS 使用，它通常是放在内存的低址部分**。而**在用户区内存中，仅装有一道用户程序，即整个内存的用户空间由该程序独占**。这样的存储器分配方式被称为**单一连续分配**方式。

固定分区分配将整个用户空间划分为若干个固定大小的区域，在每个分区中只装入一道作业，这样就形成了最早的、也是最简单的一种可运行多道程序的分区式存储管理方式。如果在内存中有四个用户分区，便允许四个程序并发运行。

动态分区分配又称为可变分区分配，它是根据进程的实际需要，动态地为之分配内存空间。

为了实现动态分区分配，通常是将系统中的空闲分区链接成一个链。所谓**顺序搜索**，是指依次搜索空闲分区链上的空闲分区，去寻找一个其大小能满足要求的分区。基于顺序搜索的动态分区分配算法有如下四种：**首次适应算法**、**循环首次适应算法**、**最佳适应算法**和**最坏适应算法**。

首次适应（first fit，FF）算法倾向于优先利用内存中低址部分的空闲分区，从而保留了高址部分的大空闲区。这为以后到达的大作业分配大的内存空间创造了条件。其缺点是低址部分不断被划分，会**留下许多难以利用的、很小的空闲分区，称为碎片**。而每次查找又都是从低址部分开始的，这无疑又会**增加查找可用空闲分区时的开销**。

循环首次适应（next fit，NF）算法不再是每次都从链首开始查找，而是从上

次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。该算法能**使内存中的空闲分区分布得更均匀**，从而减少了查找空闲分区时的开销，但这样会**缺乏大的空闲分区**。

最佳适应（best fit，BF）算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。这样，第一次找到的能满足要求的空闲区必然是最佳的。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的碎片。

最坏适应（worst fit，WF）算法在扫描整个空闲分区表或链表时，总是挑选一个最大的空闲区，从中分割一部分存储空间给作业使用，以至于存储器中缺乏大的空闲分区。它的优点是**可使剩下的空闲区不至于太小，产生碎片的可能性最小，对中、小作业有利**。同时，最坏适应分配算法查找效率很高，该算法要求，**将所有的空闲分区，按其容量从大到小的顺序形成一空闲分区链，查找时，只要看第一个分区是否满足作业要求即可**。

基于顺序搜索的动态分区分配算法，比较适用于不太大的系统。当系统很大时，系统中的内存分区可能会很多，相应的空闲分区链就可能很长，这时采用顺序搜索分区方法可能会很慢。为了提高搜索空闲分区的速度，在大、中型系统中往往会采用基于**索引搜索**的动态分区分配算法，目前常用的有**快速适应算法**、**伙伴系统**和**哈希算法**。

快速适应算法又称为**分类搜索法**，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，这样系统中存在多个空闲分区链表。同时，在内存中设立一张管理索引表，其中的每一个索引表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。

伙伴系统规定，无论已分配分区或空闲分区，**其大小均为 2 的 k 次幂**（k 为整数，1<=k<=m）。通常 2^m 是可分配内存的大小。

哈希算法就是利用哈希快速查找的优点，以及空闲分区在可利用空闲区表中的分布规律，建立哈希函数，构造一张以空闲分区大小为关键字的哈希表，该表的每一表项记录了一个对应的空闲分区链表表头指针。

连续分配方式的一个重要特点是，一个系统或用户必须被装入一片连续的内存空间中。当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。即时这些分散的许多小分区的容量总和大于要装入的程序，但**由于这些分区不相邻接，也无法把该程序装入内存**。这种不能被利用的小分区即是前已提及的“**碎片**”，或称为“**零头**”。

在每次“紧凑”后，都必须**对移动了的程序或数据进行重定位**。为了提高内存的利用率，系统在运行过程中是经常需要进行“紧凑”的，每“紧凑”一次，就要对移动了的程序或数据的地址进行修改，这不仅是一件相当麻烦的事，而且还大大影响到系统的效率。**动态重定位**方法在系统中增设一个**重定位寄存器**，用它来存放程序（数据）在内存中的起始地址。程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。

在多道程序环境下，一方面，在内存中的某些进程由于某事件尚未发生而被阻塞运行，但它却占用了大量的内存空间。甚至有时可能出现在内存中所有进程都被阻塞，而无可运行之进程，迫使 CPU 停止下来等待的情况；另一方面，却又有着许多作业，因内存空间不足，一直驻留在外存上，而不能进入内存运行。显然这对系统资源是一种严重的浪费，且使系统吞吐量下降。为了解决这一问题，在系统中又增设了**对换**设施。所谓“对换”，是指**把内存中暂时不能运行的进程或者暂时不用的程序和数据换出到外存上以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据换入内存**。对换是改善内存利用率的有效措施，它可以直接提高处理机的利用率和系统的吞吐量。

在每次对换时，都是将一定数量的程序或数据换入或换出内存。根据每次对换时所对换的数量，可将对换分为如下两类：

（1）**整体对换**。处理机中级调度实际上就是存储器的对换功能，其目的是用来解决内存紧张问题，并可进一步提高内存的利用率和系统的吞吐量。由于**在中级调度中对换是以整个进程为单位的**，故称之为“**进程对换**”或“**整体对换**”。这种对换被广泛地应用于多道程序系统中，并作为**处理机中级调度**。

（2）**页面（分段）对换**。如果**对换是以进程的一个“页面”或“分段”为单位进行的**，则分别称之为“**页面对换**”和“**分段对换**”，又统称为“**部分对换**”。这种对换方法是实现**请求分页**和**请求分段式存储管理**的基础，其目的是为了支持**虚拟存储系统**。

在具有对换功能的 OS 中，通常把磁盘空间分为**文件区**和**对换区**两部分。对文件区管理的**主要目标是提高文件存储空间的利用率，然后才是提高对文件的访问速度**。因此，对文件区空间的管理采取**离散**分配方式。对对换空间管理的**主要目标，是提高进程换入和换出的速度，然后才是提高文件存储空间的利用率**。为此，对对换区空间的管理采取**连续**分配方式，较少考虑外存中的碎片问题。

连续分配方式会形成许多“碎片”，虽然可通过“紧凑”方法将许多碎片拼接成可用的大块空间，但须为之付出很大开销。如果允许将一个进程直接分散地装入到许多不相邻接的分区中，便可充分地利用内存空间，而无须再进行“紧凑”。基于这一思想而产生了离散分配方式。根据在离散分配方式时所分配地址空间的基本单位的不同，又可将离散分配分为以下三种：

（1）**分页存储管理方式**。在该方式中，**将用户程序的地址空间分为若干个固定大小的区域，称为“页”或“页面”**。典型的页面大小为 1KB。相应地，也**将内存空间分为若干个物理块或页框（frame），页和块的大小相同**。这样可将用户程序的任一页放入任一物理块中，实现了离散分配。

（2）**分段存储管理方式**。这是为了满足用户要求而形成的一种存储管理方式。它**把用户程序的地址空间分为若干个大小不同的段，每段可定义一组相对完整的信息**。在存储器分配时，以段为单位，这些段在内存中可以不相邻接，所以也同样实现了离散分配。

（3）**段页式存储管理方式**。这是分页和分段两种存储管理方式相结合的产物。它同时具有两者的优点，是目前应用较广泛的一种存储管理方式。

分页存储管理将进程的逻辑地址空间分成若干个页，并为各页加以编号，从 0 开始，如第 0 页、第 1 页等。相应地，也把内存的物理地址空间分成若干个块，同样也为它们加以编号，如 0#块、1#块等等。在为进程分配内存时，以块为单位，将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于**进程的最后一页经常装不满一块，而形成了不可利用的碎片，称之为“页内碎片”**。

在分页系统中，若选择过小的页面大小，虽然一方面可以减小内存碎片，起到减少内存碎片总空间的作用，有利于内存利用率的提高，但另一方面却会**造成每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存**。此外，还会降低页面换进换出的效率。然而，如果选择的页面过大，虽然可以减少页表的长度，提高页面换进换出的速度，但却又会**使页内碎片增大**。因此，页面的大小应选择适中，且页面大小应是 2 的幂，通常为 1KB~8KB。

在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称**页表**。在进程地址空间内的所有页（0~n），依次在页表中有一页表项，其中记录了相应页在内存中对应的物理块号。在配置了页表后，进程执行时，通过查找该表，即可找到每页在内存中的物理块号。可见，**页表的作用是实现从页号到物理块号的地址映射**。

即使在简单的分页小系统中，也常在页表的表项中设置一存取控制字段，用于对该存储块中的内容加以保护。当存取控制字段仅有一位时，可用来规定该存储块中的内容是允许读/写还是只读；若存取控制字段为二位，则可规定为读/写、只读和只执行等存取方式。如果有一进程试图去写一个只允许读的存储块时，将引起操作系统的一次中断。

为了能将用户地址空间中的逻辑地址转换为内存空间中的物理地址，在系统中必须设置**地址变换机构**。该机构的基本任务是实现从逻辑地址到物理地址的转换。由于页内地址和物理地址是一一对应的（例如，对于页面大小是 1KB 的页内地址是 0~1023，其相应的物理块内的地址也是 0~1023，无需再进行转换），因此，地址变换机构的任务实际上只是将逻辑地址中的页号转换为内存中的物理块号。又因为页面映射表的作用就是用于实现从页号到物理块号的变换，因此，**地址变换任务是借助于页表来完成的**。

进程在运行期间，需要对程序和数据的地址进行变换，即将用户地址空间中的逻辑地址变换为内存空间中的物理地址，由于它执行的频率非常高，每条指令的地址都需要进行变换，因此需要采用**硬件**来实现。**页表大多驻留在内存中**。在系统中只设置一个**页表寄存器 PTR（Page-Table Register）**，在其中存放页表在内存的始址和页表的长度。**平时，进程未执行时，页表的始址和页表长度存放在本进程的 PCB 中。当调度程序调度到某进程时，才将这两个数据装入页表寄存器中**。因此，在单处理机环境下，虽然系统中可以运行多个进程，但只需一个页表寄存器。

当进程要访问某个逻辑地址中的数据时，分页地址变换机构会自动地将有效地址（相对地址）分为页号和页内地址两部分，再以页号为索引去检索页表。

由于页表是存放在内存中的，这使 CPU 在每存取一个数据时，都要两次访问内存。第一次是**访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量 W 拼接，以形成物理地址**。第二次访问内存时，才是**从第一次所得地址中获得所需数据（或向此地址中写入数据）**。因此，采用这种方式将使计算机的处理速度降低近 1/2。

为了提高地址变换速度，可在地址变换机构中增设一个具有并行查寻能力的特殊高速缓冲寄存器，又称为“**联想寄存器**”，或称为“**快表**”，用以存放当前访问的那些页表项。此时的地址变换过程是：**在 CPU 给出有效地址后，由地址变换机构自动地将页号 P 送入高速缓冲寄存器，并将此页号与高速缓存中的所有页号进行比较，若其中有与此相匹配的页号，便表示所要访问的页表项在快表中。于是，可直接从快表中读出该页所对应的物理块号，并送到物理地址寄存器中。**

如在快表中未找到对应的页表项，则还须再访问内存中的页表，找到后，把从页表项中读出的物理块号送往地址寄存器；同时，再将此页表项存入快表的一个寄存器单元中，亦即，重新修改快表。但如果联想寄存器已满，则 OS 必须找到一个老的且已被认为是不再需要的页表项，将它换出。

现代的大多数计算机系统都支持非常大的逻辑地址空间（2^{32}B~2^{64}B）。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。例如，对于一个具有 32 位逻辑地址空间的分页系统，规定页面大小为 4KB 即 2^{12}B，则在每个进程页表中的页表项数可达 1MB 之多，又因为每个页表项占用一个字节，故每个进程仅仅其页表就要占用 1MB 的内存空间，而且还要求是连续的。显然这是不现实的，我们可以采用这样两个方法来解决这一问题：①**对于页表所需的内存空间，可采用离散分配方式，以解决难以找到一块连续的大内存空间的问题**；②**只将当前需要的部分页表项调入内存，其余的页表项仍驻留在磁盘上，需要时再调入。**

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从 0 开始编址，并采用一段连续的地址空间。**段的长度由相应的逻辑信息组的长度决定，因此各段的长度并不相等。整个作业的地址空间由于被分成多个段，所以呈现出二维特性，亦即，每个段既包含了一部分地址空间，又标识了逻辑关系。**其逻辑地址由**段号（段名）**和**段内地址**所组成。

进程中的各个段，可以离散地装入内存中不同的分区中。为保证程序能正常运行，就必须能从物理内存中找出每个逻辑段所对应的位置。为此，在系统中，类似于分页系统，需为每个进程建立一张段映射表，简称“**段表**”。每个段在表中占有一个表项，其中记录了该段在内存中的起始地址（又称为“基址”）和段的长度。段表可以存放在一组寄存器中，以利于提高地址转换速度。但更常见的方法是将段表放在内存中。可见，**段表是用于实现从逻辑段到物理内存区的映射的。**

为了实现进程从逻辑地址到物理地址的变换功能，在系统中设置了**段表寄存器**，用于存放段表起始地址和段表长度 TL。像分页系统一样，当段表放在内存中时，每要访问一个数据，都须访问两次内存，从而成倍地降低了计算机的速率。解决的方法和分页系统类似，也增设一个**联想寄存器**，用于保存最近常用的段表项。一般情况下，由于是段比页大，因而段表项的数目比页表项的数目少，其所

需的连续寄存器也相对较小。

分页和分段系统有许多相似之处。比如，两者都采用离散分配方式，且都是通过地址映射机构实现地址变换。但在概念上两者完全不同，主要表现在下述三个方面：

(1) **页是信息的物理单位**。采用分页存储管理方式是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说，分页仅仅只是系统管理上的需要，完全是系统的行为，对用户是不可见的。分段存储管理方式中的**段则是信息的逻辑单位**，它通常包含的是一组意义相对完整的信息。分段的目的主要在于能更好地满足用户的需要。

(2) **页的大小固定且由系统决定**。在采用分页存储管理方式的系统中，在硬件结构上，就把用户程序的逻辑地址划分为页号和页内地址两部分，也就是说是直接由硬件实现的，因而在每个系统中只能有一种大小的页面。而**段的长度却不固定**，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

(3) **分页的用户程序地址空间是一维的**。分页完全是系统行为，故在分页系统中，用户程序的地址是属于单一的线性地址空间，程序员只需利用一个记忆符即可表示一个地址。而分段是用户的行为，故**在分段系统中，用户程序的地址空间是二维的**，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

段页式系统的基本原理是分段和分页原理的结合，即**先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名**。在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置**段表**和**页表**。**段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度**。

在段页式系统中，为了获得一条指令或数据，须**三次**访问内存。**第一次访问是访问内存中的段表，从中取得页表始址**；**第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与页内地址一起形成指令或数据的物理地址**；**第三次访问才是真正从第二次访问所得的地址中取出指令或数据**。显然，这使访问内存的次数增加了近两倍。为了提高执行速度，在地址变换机构中增设一个**高速缓冲寄存器**。每次访问它时，都须**同时利用段号和页号去检索高速缓存**，若找到匹配的表项，便可从中得到相应页的物理块号，用来与页内地址一起形成物理地址；若未找到匹配表项，则仍需第三次访问内存。

传统存储器管理方式，它们全都具有如下两个共同的特征：

（1）**一次性**，是指作业必须一次性地全部装入内存后方能开始运行。

（2）**驻留性**，是指作业被装入内存后，整个作业都一直驻留在内存中，其中任何部分都不会被换出，直至作业运行结束。

局部性原理：

（1）程序执行时，除了少部分的转移和过程调用指令外，在大多数情况下是顺序执行的。

（2）过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域。但经研究看出，过程调用的深度在大多数情况下都不超过 5。这就是说，程序将会在一段时间内，都局限在这些过程的范围内运行。

（3）程序中存在许多循环结构，这些结构虽然只由少数指令构成，但是它们将被多次执行。

（4）程序中还包括许多对数据结构的处理，如对数组进行操作，这些处理往往都局限于很小的范围内。

局限性又表现在下述两个方面：

（1）**时间局限性**。**如果程序中的某条指令被执行，则不久以后该指令可能被再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问**。产生时间局限性的典型原因是在程序中存在着大量的**循环操作**。

（2）**空间局限性**。**一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问**，即**程序在一段时间内所访问的地址可能集中在一定的范围之内**，其典型情况便是程序的**顺序执行**。

基于局部性原理可知，应用程序在运行之前没有必要将之全部装入内存，而仅须将那些当前要运行的少数页面或段先装入内存便可运行，其余部分暂留在盘上。程序在运行时，如果它所要访问的页（段）已调入内存，便可继续执行下去；但如果程序所要访问的页（段）尚未调入内存（称为缺页或缺段），便发出缺页（段）中断请求，此时 OS 将利用请求调页（段）功能将它们调入内存，以使进程能继续执行下去。如果此时内存已满，无法再装入新的页（段），OS 还须再利用页（段）的置换功能，将内存中暂时不用的页（段）调至盘上，腾出足够的内存空间后，再将要访问的页（段）调入内存，使程序继续执行下去。这样，便可使一个大的用户程序在较小的内存空间中运行，也可在内存中同时装入更多的进

程，使它们并发执行。

与传统的存储器管理方式比较，虚拟存储器具有以下三个重要特征：

（1）**多次性**。多次性是相对于传统存储器管理方式的一次性而言的，是指一个作业中的程序和数据无需在作业运行时一次性地全部装入内存，而是允许被分成多次调入内存运行，即只需将当前要运行的那部分程序和数据装入内存即可开始运行。

（2）**对换性**。对换性是相对于传统存储器管理方式的常驻性而言的，是指一个作业中的程序和数据，无须在作业运行时一直常驻内存，而是允许在作业的运行过程中进行换进、换出，亦即，在进程运行期间，允许将那些暂不使用的代码和数据从内存调至外存的对换区（换出），待以后需要时再将它们从外存调至内存（换进）。

（3）**虚拟性**。虚拟性是指能够从逻辑上扩充内存容量，使用户所看到的内存容量远大于实际内存容量。**虚拟性是以多次性和对换性为基础的**。

虚拟存储器的实现是建立在离散分配存储管理方式的基础上的。目前，所有的虚拟存储器都是采用下述方式之一实现的。

分页请求系统

分页请求系统是在分页系统的基础上增加了请求调页功能和页面置换功能所形成的页式虚拟存储系统。它允许用户程序只装入少数页面的程序（及数据）即可启动运行。以后，再通过调页功能及页面置换功能陆续性把即将运行的页面调入内存，同时把暂不运行的页面换出到外存上。**置换时以页面为单位**。

请求分段系统

请求分段系统是在分段系统的基础上，增加了请求调段及分段置换功能后所形成的段氏虚拟存储系统。它允许用户程序只要装入少数段（而非所有的段）的程序和数据即可启动运行。以后通过调段功能和段的置换功能将暂不运行的段调出，再调入即将运行的段。**置换时以段为单位**。

为了实现请求分页，系统必须提供一定的硬件支持。计算机系统除了要求一定容量的内存和外存外，还需要有**请求页表机制**、**缺页中断机构**以及**地址变换机构**。

在请求分页系统中需要的主要数据结构是**请求页表**，其基本作用仍然是**将用户地址空间中的逻辑地址映射为内存空间中的物理地址**。为了满足页面换进换出

的需要，在请求页表中又增加了四个字段。这样，在请求分页系统中的每个页表应含以下诸项：

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址

（1）状态位 P：指示该页是否已调入内存。

（2）访问字段 A：记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，提供给置换算法（程序）在选择换出页面时参考。

（3）修改位 M：标识该页在调入内存后是否被修改过。由于**内存中的每一页都在外存上保留一份副本**，因此，在置换该页时，**若未被修改，就不需再将该页写回到外存上**，以减少系统的开销和启动磁盘的次数；**若已被修改，则必须将该页重写到外存上**，以保证外存中所保留的副本始终是最新的。

（4）外存地址：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

缺页中断是一种特殊的中断，它与一般的中断相比有着明显的区别，主要表现在下面两个方面：

（1）**在指令执行期间产生和处理中断信号**。通常，CPU 都是在一条指令执行完后，才检查是否有中断请求到达。若有，便去响应，否则，继续执行下一条指令。然而，缺页中断是在指令执行期间，若发现所要访问的指令或数据不在内存时，便立即产生和处理缺页中断信号，以便能及时将所缺之页面调入内存。

（2）**一条指令在执行期间可能产生多次缺页中断**。

请求分页系统中的**地址变换机构**是在分页系统地址变换机构的基础上，为实现虚拟存储器，再增加了某些功能所形成的，如**产生和处理缺页中断**，以及**从内存中换出一页的功能**等等。

在进行地址变换时，首先检索快表，试图从中找出所要访问的页。若找到，便修改页表项中的访问位，供置换算法选换出页面时的参考。对于写指令，还须将修改位置成“1”，表示该页在调入内存后已被修改。然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。

如果在快表中未找到该页的页表项，则**应到内存中去查找页表，再从找到的页表项中的状态位 P 来了解该页是否已调入内存。若该页已调入内存，这时应将该页的页表项写入快表。当快表已满时，则应先调出按某种算法所确定的页的页表项，然后再写入该页的页表项；若该页尚未调入内存，这时应产生缺页中断，请求 OS 从外存把该页调入内存**。

最小物理块数是指能保证进程正常运行所需的最小物理块数，当系统为进程分配的物理块数少于此值时，进程将无法运行。

在请求分页系统中，可采取两种内存分配策略，即固定和可变分配策略。在进行置换时，也可采取两种策略，即全局置换和局部置换。于是可组合出以下三种适用的策略。

（1）**固定分配局部置换**

所谓**固定分配**，是指**为每个进程分配一组固定数目的物理块，在进程运行期间不再改变**。所谓**局部置换**，是指**如果进程在运行中发现缺页，则只能从分配给该进程的 n 个页面中选出一页换出，然后再调入一页，以保证分配给该进程的内存空间不变**。

（2）**可变分配全局置换**

所谓**可变分配**，是指**先为每个进程分配一定数目的物理块，在进程运行期间，可根据情况做适当的增加或减少**。所谓**全局置换**，是指**如果进程在运行中发现缺页，则将 OS 所保留的空闲物理块（一般组织为一个空闲物理块队列）取出一块分配给该进程，或者以所有进程的全部物理块为标的，选择一块换出，然后将所缺之页调入**。这样，分配给该进程的内存空间就随之增加。在采用这种策略时，**凡产生缺页（中断）的进程，都将获得新的物理块，仅当空闲物理块队列中的物理块用完时，OS 才能从内存中选择一页调出。被选择调出的页可能是系统中任何一个进程的页，因此这个被选中的进程拥有的物理块会减少，这将导致其缺页率增加**。

（3）**可变分配局部置换**

为每个进程分配一定数目的物理块，但当某进程发现缺页时，只允许从该进程在内存的页面中选择一页换出，这样就不会影响其它进程的运行。如果进程在运行中频繁地发生缺页中断，则系统须再为该进程分配若干附加的物理块，直至该进程的缺页率减少到适当程度为止。反之，**若一个进程在运行过程中的缺页率特别低，则此时可适当减少分配给该进程的物理块数**，但不应引起其缺页率的明显增加。

将请求分页系统中的外存分为两部分：用于存放文件的文件区和用于存放对换页面的对换区。通常，由于对换区是采用连续分配方式，而文件区是采用离散分配方式，所以对换区的数据存取（磁盘 I/O）速度比文件区的高。这样，每当

发生缺页请求时，系统应从何处将缺页调入内存，可分成如下三种情况进行：

（1）**系统拥有足够的对换区空间，这时可以全部从对换区调入所需页面，以提高调页速度**。为此，在程序运行前，便须将与该进程有关的文件从文件区拷贝到对换区。

（2）**系统缺少足够的对换区空间，这时凡是不会被修改的文件，都直接从文件区调入**；而当换出这些页面时，由于它们未被修改，则不必再将它们重写到磁盘（换出），以后再调用时，仍从文件区直接调入。但**对于那些可能被修改的部分，在将它们换出时便须调到对换区，以后需要时再从对换区调入**。

（3）UNIX 方式。由于与进程有关的文件都放在文件区，故**凡是未运行过的页面，都应从文件区调入**。而**对于曾经运行过但又被换出的页面，由于是被放在对换区，因此在下次调入时应从对换区调入。由于 UNIX 系统允许页面共享，因此，某进程所请求的页面有可能已被其它进程调入内存，此时也就无需再从对换区调入**。

每当程序所要访问的页面未在内存时（存在位为 “0”），便向 CPU 发出一缺页中断，中断处理程序首先保留 CPU 环境，分析中断原因后转入缺页中断处理程序。该程序通过查找页表得到该页在外存的物理块后，如果此时内存能容纳新页，则启动磁盘 I/O，将所缺之页调入内存，然后修改页表。如果内存已满，则须先按照某种置换算法，从内存中选出一页准备换出；如果该页未被修改过（修改位为 “0”），可不必将该页写回磁盘；但如果此页已被修改（修改位为 “1”），则必须将它写回磁盘，然后再把所缺的页调入内存，并修改页表中的相应表项，置其存在位为 “1”，并将此页表项写入快表中。在缺页调入内存后，利用修改后的页表形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

不适当的页面置换算法可能会导致进程发生 “**抖动**”（Thrashing），即刚被换出的页很快又要被访问，需要将它重新调入，此时又需要再选一页调出；而此刚被调出的页很快又被访问，又需将它调入，如此频繁地更换页面，以致一个进程在运行中把大部分时间都花费在页面置换工作上，我们称该进程发生了 “抖动”。

最佳（Optimal）置换算法：其所选择的被淘汰页面将是以后永不使用的，或许是在最长（未来）时间内不再被访问的页面。无法实现。

先进先出（FIFO）页面置换算法：总是淘汰最先进入内存的页面，即选择在

内存中驻留时间最久的页面予以淘汰。

LRU（Least Recently Used）置换算法：根据页面调入内存后的使用情况作出决策。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t。当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。

最少使用（Least Frequently Used，LFU）置换算法：在采用 LFU 算法时，应为内存中的每个页面设置一个移位寄存器，用来记录该页面被访问的频率。该置换算法选择在最近时期使用最少的页面作为淘汰页。

简单的 Clock 置换算法：当某页被访问时，其访问位被置 1。置换算法在选择一页淘汰时，只需检查页的访问位。如果是 0，就选择该页换出；若为 1，则重新将它置为 0，暂不换出，给予该页第二次驻留内存的机会，再按照 FIFO 算法检查下一个页面。当检查到队列中的最后一个页面时，若其访问位仍为 1，则再返回到队首去检查第一个页面。

改进型 Clock 置换算法：在将一个页面换出时，如果该页已被修改过，便须将该页重新写回到磁盘上；但如果该页未被修改过，则不必将它拷回磁盘。换而言之，对于修改过的页面，在换出时所付出的开销比未修改过的页面大，或者说，置换代价大。在改进型 Clock 算法中，除须考虑页面的使用情况外，还须再增加一个因素——置换代价。这样，选择页面换出时，既要是未使用过的页面，又要是未被修改过的页面。把同时满足这两个条件的页面作为首选淘汰的页面。由访问位 A 和修改位 M 可以组合成下面四种类型的页面：

1 类（A=0，M=0）：表示该页最近既未被访问，又未被修改，是最佳淘汰页。

2 类（A=0，M=1）：表示该页最近未被访问，但已被修改，并不是很好的淘汰页。

3 类（A=1，M=0）：表示最近已被访问，但未被修改，该页有可能再被访问。

4 类（A=1，M=1）：表示最近已被访问且被修改，该页可能再被访问。

页面缓冲算法（Page Buffering Algorithm，PBA）的主要特点是：**①显著地降低了页面换进、换出的频率，使磁盘 I/O 的操作次数大为减少，因而减少了页面换进、换出的开销；②正是由于换入换出的开销大幅度减小，才能使其采用一种较简单的置换策略，如先进先出（FIFO）算法，它不需要特殊硬件的支持，实现起来非常简单**。页面缓冲算法已在不少系统中使用，下面我们介绍 VAX/VMS 操作系统中所使用的页面缓冲算法。在该系统中，内存分配策略上采用了可变分配和局部置换方式，系统为每个进程分配一定数目的物理块，系统自己保留一部分

空闲物理块。为了能显著地降低页面换进、换出的频率，在内存中设置了如下两个链表：

（1）**空闲页面链表**

实际上该链表是一个空闲物理块链表，是系统掌握的空闲物理块，用于分配给频繁发生缺页的进程，以降低该进程的缺页率。当这样的进程需要读入一个页面时，便可利用空闲物理块链表中的第一个物理块来装入该页。当**有一个未被修改的页要换出时，实际上并不将它换出到外存，而是把它们所在的物理块挂在空闲链表的末尾**。应当注意，**这些挂在空闲链表上的未被修改的页面是有数据的**，如果以后某进程需要这些页面中的数据时，便可从空闲链表上将它们取下，免除了从磁盘读入数据的操作，减少了页面换进的开销。

（2）**修改页面链表**

它是由已修改的页面所形成的链表。设置该链表的目的是为了减少已修改的页面换出的次数。当系统需要将一个已修改的页面换出时，系统并不立即把它换出到外存上，而是将它所在的物理块挂在修改页面链表的末尾。这样做的目的是：降低将已修改页面写回磁盘的频率，降低将磁盘内容读入内存的频率。

与基本分页存储管理方式不同，在请求分页管理方式中，内存有效访问时间不仅要考虑访问页表和访问实际物理地址数据的时间，还必须要考虑到缺页中断的处理时间。这样，在具有快表机制的请求分页管理方式中，存在下面三种方式的内存访问操作，其有效访问时间的计算公式也有所不同：

（1）**被访问页在内存中，且其对应的页表项在快表中**

显然，此时不存在缺页中断情况，内存的有效访问时间（EAT）分为**查找快表的时间**和**访问实际物理地址所需的时间**。

（2）**被访问页在内存中，且其对应的页表项不在快表中**

显然，此时也不存在缺页中断情况，但需要两次访问内存，一次读取页表，一次读取数据，另外还需要更新快表。所以，这种情况内存的有效访问时间可分为**查找快表的时间**、**查找页表的时间**、**修改快表的时间**和**访问实际物理地址的时间**。

（3）**被访问页不在内存中**

因为被访问页不在内存中，需要进行缺页中断处理，所以这种情况的内存的有效访问时间可分为**查找快表的时间**、**查找页表的时间**、**处理缺页中断的时间**、**更新快表的时间**和**访问实际物理地址的时间**。

所谓**工作集**，是指在某段时间间隔△里，进程实际所要访问页面的集合。

为了保证系统具有较大的吞吐量，必须防止“抖动”的发生。目前已有许多防止“抖动”发生的方法。这些方法几乎都是采用调节多道程序度来控制“抖动”发生的。下面介绍几个较常用的预防“抖动”发生的方法。

（1）**采取局部置换策略**

在页面分配和置换策略中，如果采取的是可变分配方式，则为了预防发生“抖动”，可采取局部置换策略。根据这种策略，当某进程发生缺页时，只能在分配给自己的内存空间内进行置换，不允许从其它进程去获得新的物理块。这样，**即使该进程发生了“抖动”，也不会对其它进程产生影响**，于是可把该进程“抖动”所造成的影响限制在较小的范围内。该方法虽然简单易行，但效果不是很好，因为**在某进程发生“抖动”后，它还会长期处在磁盘 I/O 的等待队列中，使队列的长度增加，这会延长其它进程缺页中断的处理时间，也就是延长了其它进程对磁盘的访问时间**。

（2）**把工作集算法融入到处理机调度中**

当调度程序发现处理机利用率低下时，它将试图从外存调入一个新作业进入内存，来改善处理机的利用率。如果在调度中融入了工作集算法，则**在调度程序从外存调入作业之前，必须先检查每个进程在内存的驻留页面是否足够多**。如果都已足够多，此时便可以从外存调入新的作业，不会因新作业的调入而导致缺页率的增加；反之，如果有些进程的内存页面不足，则应首先为那些缺页率居高的作业增加新的物理块，此时将不再调入新的作业。

（3）**利用“L=S”准则调节缺页率**

Denning 于 1980 年提出了“L=S”的准则来调节多道程序度，其中 **L 是缺页之间的平均时间，S 是平均缺页服务时间，即用于置换一个页面所需的时间。如果 L 远比 S 大，说明很少发生缺页，磁盘的能力尚未得到充分的利用**；反之，**如果是 L 比 S 小，则说明频繁发生缺页，缺页的速度已超过磁盘的处理能力**。只有当 L 与 S 接近时，磁盘和处理机都可达到它们的最大利用率。理论和实践都已证明，利用“L=S”准则，对于调节缺页率是十分有效的。

（4）**选择暂停的进程**

当多道程序度偏高时，已影响到处理机的利用率，为了防止发生“抖动”，系统必须减少多道程序的数目。此时应基于某种原则选择**暂停某些当前活动的进程，将它们调出到磁盘上，以便把腾出的内存空间分配给缺页率发生偏高的进程**。系统通常都是采取与调度程序一致的策略，即首先选择暂停优先级最低的进程，若

需要，再选择优先级较低的进程。当内存还显拥挤时，还可进一步选择暂停一个并不十分重要、但却较大的进程，以便能释放出较多的物理块，或者暂停剩余执行时间最多的进程等。

在分段基础上建立的请求分段式虚拟存储器系统是以**分段**为单位进行换入、换出的。与请求分页系统相似，在请求分段系统中所需的硬件支持有**段表机制**、**缺段中断机构**，以及**地址变换机构**。

在请求分段式管理中所需的主要数据结构使请求段表。在该表中除了具有请求分页机制中有的访问字段 A、修改位 M、存在位 P 和外存起始地址外，还增加了存储方式字段和增补位。这些字段供程序在调进、调出时参考。下面给出请求分段的**段表**项。

段名	段长	段基址	存储方式	访问字段 A	修改位 M	存在位 P	增补位	外存始址

在段表项中，除了段名（号）、段长、段在内存中的起始地址（段基址）外，还增加了以下字段：

（1）存取方式。**由于应用程序中的段是信息的逻辑单位，可根据该信息的属性对它实施保护，故在段表中增加存取方式字段**，如果该字段为两位，则存取属性是只执行、只读和允许读/写。

（2）访问字段 A。其含义与请求分页的相应字段相同，用于记录该段被访问的频繁程度。提供给置换算法选择换出页面时参考。

（3）修改位 M。该字段用于表示该页在进入内存后是否已被修改过，供置换页面时参考。

（4）存在位 P。该字段用于指示本段是否已调入内存，供程序访问时参考。

（5）增补位。这时请求分段式管理中所特有的字段，用于表示本段在运行过程中是否做过动态增长。

（6）外存始址。指示本段在外存中的起始地址，即起始盘块号。

与缺页中断机构类似，**缺段中断机构**同样需要**在一条指令的执行期间产生和处理中断**，以及**在一条指令执行期间，可能产生多次缺段中断**。由于段是不定长的，这使对缺段中断的处理要比对缺页中断的处理负责。

请求分段系统中的**地址变换机构**是在分段系统地址变换机构的基础上形成的。因为被访问的段并非全在内存中，所以在地址变换时，若发现所要访问的段不在内存，必须先**将所缺的段调入内存，并修改段表，然后才能再利用段表进行**

地址变换。

共享段表

段名	段长	内存始址	状态	外存始址
共享进程计数 count				
状态	进程名	进程号	段号	存取控制
……	……	……	……	……

存取控制字段：对于一个共享段，应**为不同的进程赋予不同的存取权限**。

段号：对于**一个共享段，在不同的进程中可以具有不同的段号**，每个进程可用自己进程的段号去访问该共享段。

在为共享段分配内存时，对第一个请求使用该共享段的进程，由系统为该共享段分配一物理区，再把共享段调入该区，同时将该区的始址填入请求进程的段表的相应项中，还须在共享段表中增加一表项，填写请求使用该共享段的进程名、段号和存取控制等有关数据，把 count 置 1。当又有其它进程需要调用该共享段时，由于该共享段已被调入内存，故此时无须再为该段分配内存，而只需在调用进程的段表中增加一表项，填写该共享段的物理地址。在共享段的段表中增加一个表项，填上调用进程的进程名、该共享段在本进程中的段号、存取控制等，再执行 count=count+1 操作，以表明有两个进程共享该段。

当共享此段的某进程不再需要该段时，应将该段释放，包括撤销在该进程段表中共享段所对应的表项，以及执行 count=count-1 操作。若结果为 0，则须由系统回收该共享段的物理内存，以及取消在共享段表中该段所对应的表项，表明此时已没有进程使用该段；否则（减 1 结果不为 0），只是取消调用者进程在共享段表中的有关记录。

在分段系统中，由于每个分段在逻辑上是相对独立的，因而比较容易实现信息保护。目前，常采用以下几种措施来确保信息的安全。

（1）**越界检查**

越界检查是利用地址变换机构来完成的。为此，在地址变换机构中设置了段表寄存器，用于存放段表始址和段表长度信息。在进行地址变换时，首先将逻辑地址空间的段号与段表长度进行比较，如果段号等于或大于段表长度，将发出地址越界中断信号。此外，还在段表中为每个段设置有段长字段，在进行地址变换时，还要检查段内地址是否等于或大于段长，若大于段长，将产生地址越界中断

信号，从而保证了每个进程只能在自己的地址空间内运行。

（2）**存取控制检查**

存取控制检查是以段为基本单位进行的。对于共享段而言，存取控制显得尤为重要，因而**对不同的进程应赋予不同的读写权限**。

（3）**环保护机构**

这时一种功能完善的保护机制。在该机制中规定 :低编号的环具有高优先权。OS 核心处于 0 号环内；某些重要的实用程序和操作系统服务占据中间环；而一般的应用程序，则被安排在外环上。在环系统中，程序的访问和调用应遵循以下规则：

①**一个程序可以访问驻留在相同环或较低特权环（外环）中的数据**。

②**一个程序可以调用驻留在相同环或较高特权环（内环）中的服务**。