进程是具有一定功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源调度和分配的一个独立单位。（关键词：资源调度分配的独立单位）

线程（是进程的实体），是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。（关键词：CPU调度分派的基本单位）

一个进程可以有多个线程（至少一个），多个线程也可以并发执行。

进程作为资源（如内存）分配的基本单位，作为其下属的线程都是可以享用其被分配到的资源的，而且线程可以共享同一块被分配的资源。而进程之间是一般不能分享彼此的资源的，进程想要互相通信，必须通过进程间通信(Inter-process communication，IPC)的机制来完成，主要包括以下几种：

管道（pipe，半双工），流管道（s\_pipe，全双工），有名管道（FIFO，全双工）（pipe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）

信号量（sophomore/mutex）跟Lock机制差不多，读写互斥，写写互斥

消息传递系统：以格式化的消息为单位，将数据封装到消息中利用一组通信原语进行消息传递。

共享内容：基于共享数据结构或者共享存储区

套接字（socket）：这个属于客户机-服务器系统。

进程：

1. 进程是一个**可拥有资源**的独立单位；进程同时又是一个可**独立调度和分派**的基本单位。这两个是使进程成为一个能独立运行的基本单位。一个进程拥有一个PCB，断点信息保存恢复，还可以利用其中的信息对进程进行调度。
2. 在引入线程的os中，线程是可独立调度和分派的基本单位。其共享进程的内存地址和空间。TCB与PCB的作用相同，将所有用于控制和管理线程的信息记录在线程控制块中。

线程的实现：

线程可以再分为两类：

一类是用户级线程(user level thread)。对于这类线程，有关线程管理的所有工作都由应用程序完成，内核意识不到线程的存在。在应用程序启动后，操作系统分配给该程序一个进程号，以及其对应的内存空间等资源。应用程序通常先在一个线程中运行，该线程被成为主线“程。在其运行的某个时刻，可以通过调用线程库中的函数创建一个在相同进程中运行的新线程。 用户级线程的好处是非常高效，不需要进入内核空间，但并发效率不高。因为**其以进程为单位进行调度。**

另一类是内核级线程(kernel level thread)。对于这类线程，有关线程管理的所有工作由内核完成，应用程序没有进行线程管理的代码，只能调用内核线程的接口。内核维护进程及其内部的每个线程，调度也由内核基于线程架构完成。内核级线程的好处是，内核可以将不同线程更好地分配到不同的CPU，以实现真正的并行计算。（以线程为单位进行调度）

事实上，在现代操作系统中，往往使用组合方式实现多线程，即线程创建完全在用户空间中完成，并且一个应用程序中的多个用户级线程被映射到一些内核级线程上，相当于是一种折中方案。

程序是一个静态概念，它是指在计算机的文件系统里以文件形式存储的一段可运行代码。而进程是一个动态概念，它通常是指操作系统里一个程序在一个数据集合上一次运行过程的体现。即进程是程序的运行逻辑实际运作起来的载体。

**处理机的调度：**

**各种处理机调度算法：**

先来先服务，短作业优先，优先级调度，高响应比优先调度

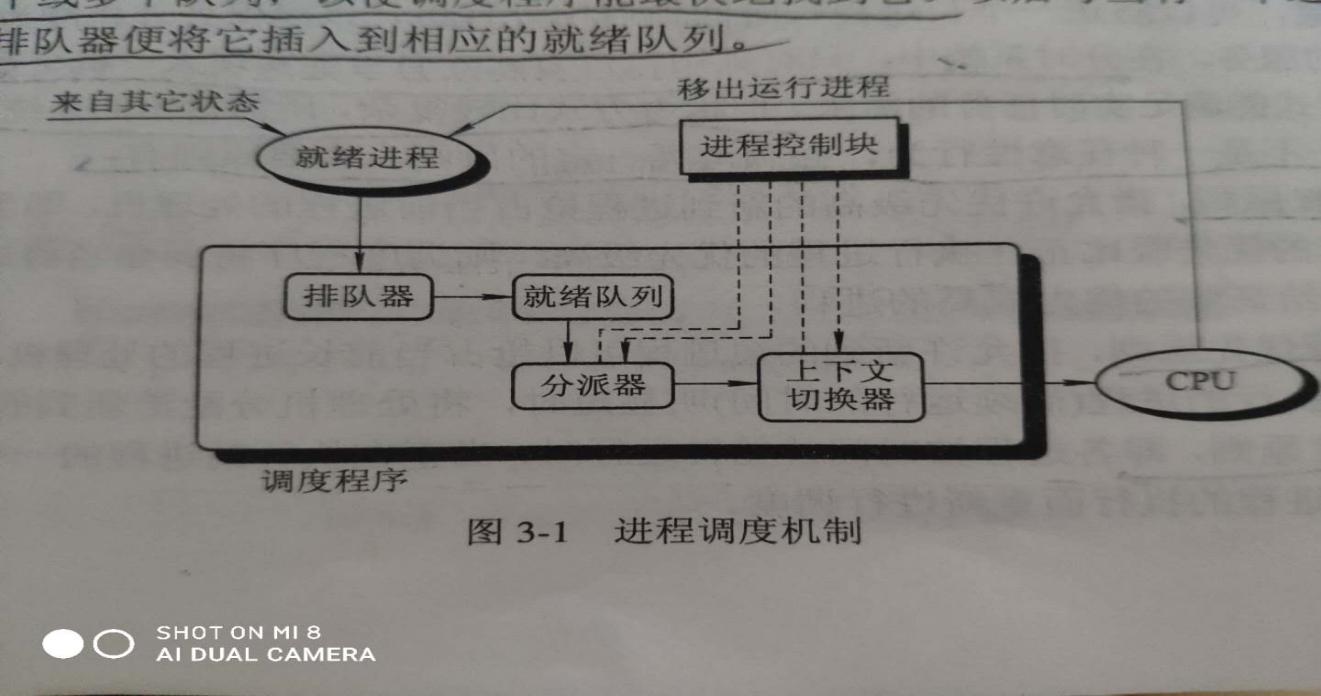
**进程调度的任务**有三个：1）保存处理机的现场信息；2）按某种算法选择进程；3）把处理机分配给进程。

**进程调度机制：**

排队器：将进程排成一个或多个队列

分派器：依据进程调度程序所选定的线程将其从就绪队列中取出，然后进行从分派器到新选出进程间的上下文切换，将处理机分配给新选出的线程。

上下文切换器：保存当前上下文，移出分派程序上下文。



**进程调度方式**：抢占式和非抢占式

进程中的**线程间**同步方式：

1. 互斥量： 采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。

2. 信号量： 它允许同一时刻多个线程来访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。

3. 事件（信号）：通过通知操作的方式来保持多线程同步，还可以方便实现多线程优先级的比较操作。（事件机制，**则允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务**。每个Cevent对象可以有两种状态：有信号状态和无信号状态。）

4.临界区：临界区对象和互斥对象非常相似，只是互斥量允许在进程间使用，而临界区只限制与同一进程的各个线程之间使用，但是更节省资源，更有效率。

临界区： 当多个线程访问一个独占性共享资源时，可以使用临界区对象。拥有临界区的线程可以访问被保护起来的资源或代码段，其他线程若想访问，则被挂起，直到拥有临界区的线程放弃临界区为止。

进程间同步方式：

1、信号量

 2、管程

在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。

**什么是缓冲区溢出？有什么危害？其原因是什么？**

缓冲区为暂时置放输出或输入资料的内存。

缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。

缓冲区溢出是一种非常普遍、非常危险的漏洞，在很多软件与操作系统都存在这种问题。

计算机中，**缓冲区溢出会造成的危害**主要有以下两点：

1.程序崩溃，导致拒绝服务

2.跳转并且执行一段恶意代码

**造成缓冲区溢出的主要原因**是程序中没有仔细检查用户输入是否合理

**死锁产生的四个必要条件（**有一个条件不成立，则不会产生死锁。因此可以打破这四个条件来破坏死锁**）：**

互斥条件：一个资源一次只能被一个进程使用

请求与保持(占有并申请)条件：一个进程因请求资源而阻塞时，对已获得资源保持不放

不剥夺条件：进程获得的资源，在未完全使用完之前，不能强行剥夺。（有不可抢占资源）

循环等待条件：若干进程之间形成一种头尾相接的环形等待资源关系。（你和你弟互相等着对方把作料交出来，不然谁也吃不上同时蘸醋和酱油的饺子）

**预防死锁中，破坏死锁的4个条件会降低系统性能，因此需要一种另一种方法来预防死锁：**

概念：所谓系统安全的，是指系统中的所有进程能够按照某一种次序分配资源，并且依次地运行完毕，这种进程序列{P1，P2，...，Pn}就是安全序列。如果存在这样一个安全序列，则系统是安全的；如果系统不存在这样一个安全序列，则系统是不安全的。 进程的安全序列{P1，P2，...，Pn}是这样组成的：若操作系统按照序列中的顺序为每个进程分配它所需要的资源，可以保证资源够用，进而所有进程的任务顺利执行至完毕，不会发生死锁。

**银行家算法的思路：**

有一系列进程共n个都要请求资源，这些资源共m种，每种有一定数量，每个进程都可能需要多种资源，并且每个进程可能已经占有了一些资源，但仍需要申请新的资源以完成任务。

首先设计一个【安全检测算法】，以用于判断”如果满足了某个进程的资源申请需求，系统是否还处于安全状态“，如果满足这个需求后，系统仍处于安全状态，那当然没问题，系统就分配资源，如果发现会处于不安全状态，那我们就暂时先把这个请求放一边，让它等待。

1)Work:int[m]，它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，int[i]表示第i种资源的系统拥有数量。

2)Finish:int[n]，它表示进程是否完成运行任务。开始时Finish[i]=false，即所有的进程都还没有运行完成（完成了的进程就不会进行资源申请了）;当有足够资源分配给进程时，再令Finish[i]:=ture，表示系统将资源分配给进程使得进程完成了运行任务.

3)Need:int[n][m]：表示每个进程所需的各类资源的数目，Need[i]表示第i个进程所需的资源向量。

4)Allocation:int[n][m]：各个进程当前所已经持有的各类资源数目，即系统之前可能已经分配给各个进程的一部分资源。

**Request[i]<=Need[i]**  
即当前进程的资源请求是否超出它实际所需要以完成任务的资源数目，如果超过了，这种请求显然是不合理的，在浪费资源，那么我们就直接抛出异常。

**Request[i]<=Available[i]**

如果上一步检查通过，我们再检查：Request[i]<=Available[i]，即系统当前拥有的资源是否可以满足这个资源请求，如果不可以，那么说明系统暂时无能为力，这个请求只能先搁置一边，进入等待状态。而如果可以，那么我们就尝试把进程所要求的资源”试探性“地分配给进程。

**解决死锁的常用策略如下：**

**鸵鸟策略、预防策略、避免策略、检测与解除死锁**

**参考网站：<https://blog.csdn.net/bxyill/article/details/8237339>**

**<https://www.cnblogs.com/Jessy/p/3540724.html>**

**进程有哪几种状态？**

就绪状态：进程已获得除处理机以外的所需资源，等待分配处理机资源   
运行状态：占用处理机资源运行，处于此状态的进程数小于等于CPU数   
阻塞状态：进程等待某种条件，在条件满足之前无法执行

**内存：**

段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的 ；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。 段的大小不固定，有它所完成的功能决定；页大大小固定，由系统决定（一般为4k） 段向用户提供二维地址空间；页向用户提供的是一维地址空间 段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

**逻辑地址/物理地址/虚拟内存：**

所谓的逻辑地址，是指计算机用户(例如程序开发者)，看到的地址。例如，当创建一个长度为100的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。**由于整型元素的大小为4个字节，故第二个元素的地址时起始地址加4**，以此类推。事实上，**逻辑地址并不一定是元素存储的真实地址**，即**数组元素的物理地址**(在内存条中所处的位置)，并非是连续的，只是操作系统通过地址映射，将逻辑地址映射成连续的，这样更符合人们的直观思维。

另一个重要概念是**虚拟内存**。**操作系统读写内存的速度可以比读写磁盘的速度快几个量级**。但是，内存价格也相对较高，不能大规模扩展。于是，操作系统可以通过将部分不太常用的数据移出内存，“存放到价格相对较低的**磁盘缓存**，以实现内存扩展。操作系统还可以**通过算法预测哪部分存储到磁盘缓存的数据需要进行读写，提前把这部分数据读回内存**。**虚拟内存空间相对磁盘而言要小很多**，因此，即使搜索虚拟内存空间也比直接搜索磁盘要快。唯一慢于磁盘的可能是，**内存、虚拟内存中都没有所需要的数据，最终还需要从硬盘中直接读取**。**这就是为什么内存和虚拟内存中需要存储会被重复读写的数据**，否则就失去了缓存的意义。

**虚拟存储器：**

是指具有**请求调入功能和置换功能（对装有装有程序的页面置换和调入）**，能从逻辑上对内存容量加以扩充的一种存储器系统。其逻辑容量（**虚拟内存**）由内存容量和外存容量之和决定。

**与内存／虚拟内存相关的还有如下两个概念：**

1) Resident Set

当一个进程在运行的时候，操作系统不会一次性加载进程的所有数据到内存，只会加载一部分正在用，以及预期要用的数据。其他数据可能存储在虚拟内存，交换区和硬盘文件系统上。被加载到内存的部分就是resident set。

2) Thrashing

由于resident set包含预期要用的数据，理想情况下，进程运行过程中用到的数据都会逐步加载进resident set。但事实往往并非如此：每当需要的内存页面(page)不在resident set中时，操作系统必须从虚拟内存或硬盘中读数据，这个过程被称为内存页面错误(page faults)。当操作系统需要花费大量时间去处理页面错误的情况就是thrashing。

**页面置换算法**

**请求调页，也称按需调页，即对不在内存中的“页”，当进程执行时要用时才调入**，否则有可能到程序结束时也不会调入。而内存中给页面留的位置是有限的，**在内存中以帧为单位放置页面**。为了防止请求调页的过程出现过多的内存页面错误（即需要的页面当前不在内存中，需要从硬盘中读数据，也即需要做页面的替换）而使得程序执行效率下降，我们需要设计一些**页面置换算法**，页面按照这些算法进行相互替换时，可以尽量达到较低的错误率。

**FIFO算法**

先入先出，即淘汰最早调入的页面

**LRU(Least-Recently-Used)算法**用过去的历史预测将来，选最近最长时间没有使用的页淘汰(也称最近最少使用)

**中断与系统调用：**

所谓的中断就是在计算机执行程序的过程中，由于出现了某些特殊事情，使得CPU暂停对程序的执行，转而去执行处理这一事件的程序。等这些特殊事情处理完之后再回去执行之前的程序。中断一般分为三类：

1. 由计算机硬件异常或故障引起的中断，称为内部异常中断。
2. 由程序中执行了引起中断的指令而造成的中断，称为软中断（和系统调用相关的中断）
3. 由外部设备请求引起的中断，称为外部中断

**中断级别（即中断的紧急程度）：**

机器错误 > 时钟 > 磁盘 > 网络设备 > 终端 > 软件中断

**系统调用：**

在讲系统调用之前，先说下进程的执行在系统上的两个级别：**用户级和核心级**，也称为**用户态和系统态**(user mode and kernel mode)。

程序的执行一般是在用户态下执行的，但当程序需要使用操作系统提供的服务时，比如说打开某一设备、创建文件、读写文件等，就需要向操作系统发出调用服务的请求，这就是系统调用。

Linux系统有专门的函数库来提供这些请求操作系统服务的入口，这个函数库中包含了操作系统所提供的对外服务的接口。当进程发出系统调用之后，它所处的运行状态就会由用户态变成核心态。但这个时候，**进程本身其实并没有做什么事情，这个时候是由内核在做相应的操作，去完成进程所提出的这些请求**。

**系统调用和中断的关系就在于，当进程发出系统调用申请的时候，会产生一个软件中断。产生这个软件中断以后，系统会去对这个软中断进行处理，这个时候进程就处于核心态了**。

那么用户态和核心态之间的区别是什么呢？（以下区别摘至《UNIX操作系统设计》）

用户态的进程能存取它们自己的指令和数据，但不能存取内核指令和数据（或其他进程的指令和数据）。然而，核心态下的进程能够存取**内核和用户地址**。

**某些机器指令是特权指令，在用户态下执行特权指令会引起错误**   
对此要理解的一个是，在系统中内核并不是作为一个与用户进程平行的进程的集合，**内核是为用户进程运行的。**

**IO多路复用：**

**IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：**

当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。

当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。

如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用

**与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。**

**常见的IO复用实现：（详细解释：**

**https://blog.csdn.net/armlinuxww/article/details/92803381）**

**select(Linux/Windows/BSD Unix), epoll(Linux)，kqueue(BSD/Mac OS X)**

**整体参考网站：**

**<https://blog.csdn.net/qq_32690999/article/details/78069414#%E8%AF%B7%E5%88%86%E5%88%AB%E7%AE%80%E5%8D%95%E8%AF%B4%E4%B8%80%E8%AF%B4%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E4%BB%A5%E5%8F%8A%E5%AE%83%E4%BB%AC%E7%9A%84%E5%8C%BA%E5%88%AB>**

**以这段代码为例：**

//创建socket

int s = socket(AF\_INET, SOCK\_STREAM, 0);

//绑定

bind(s, ...)

//监听

listen(s, ...)

//接受客户端连接

int c = accept(s, ...)

//接收客户端数据

recv(c, ...);

//将数据打印出来

printf(...)

Recv 是个阻塞方法，当程序运行到 Recv 时，它会一直等待，直到接收到数据才往下执行。那么阻塞的原理是什么?

工作队列

操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。

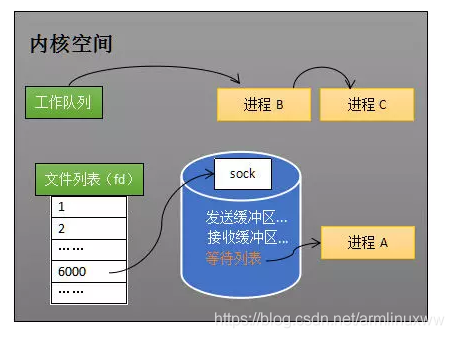
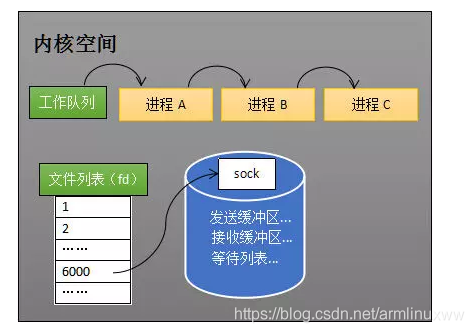
运行状态是进程获得 CPU 使用权，正在执行代码的状态;等待状态是阻塞状态，比如上述程序运行到 Recv 时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。

操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务。

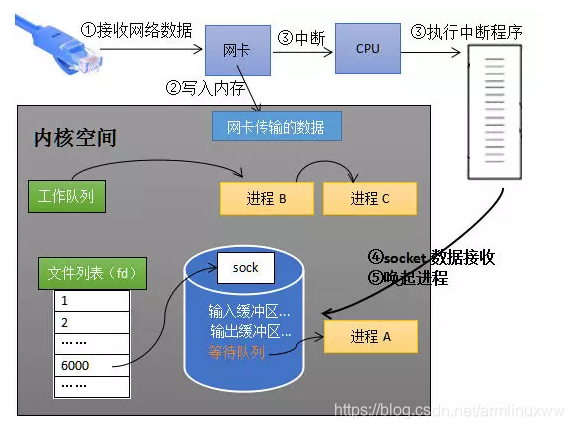
下图的计算机中运行着 A、B 与 C 三个进程，其中进程 A 执行着上述基础网络程序，一开始，这 3 个进程都被操作系统的工作队列所引用，处于运行状态，会分时执行，

创建 Socket这个 Socket 对象包含了发送缓冲区、接收缓冲区与等待队列等成员。等待队列是个非常重要的结构，它指向所有需要等待该 Socket 事件的进程。

当程序执行到 Recv 时，操作系统会将进程 A 从工作队列移动到该 Socket 的等待队列中：



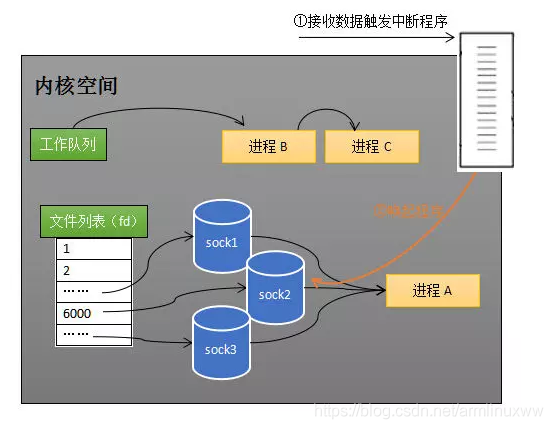
**内核接受网络数据全过程：**



从上面代码可以看到recv只能监控一个socket对象，即只有一个socket对象可以唤醒recv所在的线程。

**Selector:**

因此selector的解决方案是，使用一个selector对象存放所有的socket对象，当socket收到数据时，可以唤醒持有selector对象的线程，从而达到同时监控多个socket对象的目的。



但是其有一个缺点：因为不知道是哪个socket对象接受到了数据，每次都要遍历socket对象数组。

**Epoll解决了selector每次遍历socket数组的消耗：**

在socket接受到数据后，将该socket放到一个就绪列表中（就绪列表由eventepoll对象持有，并且socket中持有eventpoll，目的是为了socket中有数据时调用它），就绪列表中的socket对象就是已经拿到接受缓冲区已经有数据的对象了。