操作系统（核心版）

MRL Liu

2022 年 01 月 22 日

第 1 章 内存管理

一、2 种运行模式

你知道操作系统的两种运行模式吗？它们之间如何进行切换？

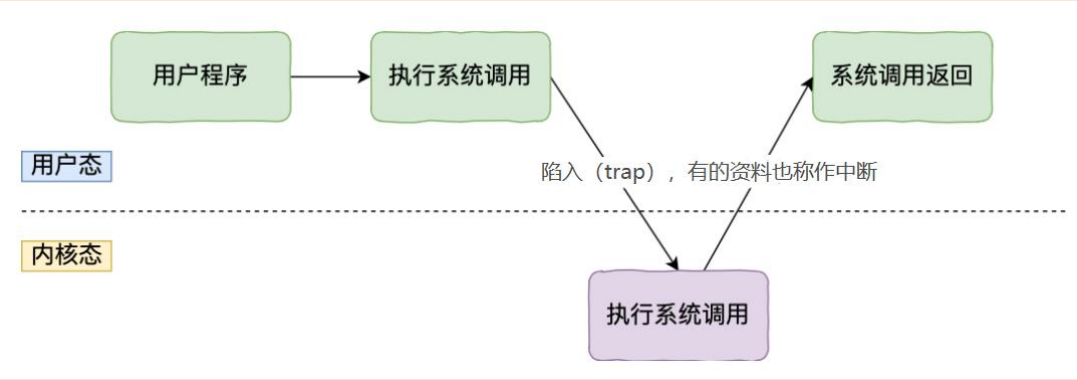
多数计算机的操作系统将内存划分为两个区域：内核空间和用户空间。

不同内存区域上运行的代码分别对应操作系统的两种运行模式：内核态和用户态，其本质也是 CPU 的 2 种运行模式，CPU 的内部寄存器 PSW 中有一个二进制位可以控制这两种运行模式的切换。

在内核态下，CPU 可以执行整个指令集中的每一条指令，使用硬件的所有功能，此时的操作系统便在内核态下运行，可以访问所有硬件资源。内核程序的代码可以访问所有空间的代码。

在用户态下，CPU 只运行执行整个指令集中的一个子集，使用硬件的部分功能，此时的操作系统便在用户态下运行，只能访问部分 CPU 等资源，不能访问 I/O 设备。用户程序只能在用户空间访问一个局部的内存空间。

当用户程序使用系统调用（system call）调用内核代码，本质上也就是用户空间的代码需要访问内核空间，这个时候就是发生了陷入（trap）：TRAP 指令把 CPU 的用户态切换成内核态，然后启用操作系统的内核态程序，系统调用结束后切换回用户态。



二、虚拟地址

请你介绍下操作系统的虚拟内存？

对于计算机来讲，任何类型的存储器的存储空间在计算机看来都是一段连续线性的「物理地址」。

操作系统的程序**要尽可能避免访问绝对的**「物理地址」，所以操作系统为创建的每个进程分配一套独立的「虚拟地址」。基于操作系统的所有进程中的程序可以访问的都是「虚拟地址」，「虚拟地址」在 CPU 执行指令时映射到「物理地址」。（请注意，这里讲的物理地址和虚拟地址都是内存管理的概念，是内存（主存储器，RAM），硬盘上一般不认为有虚拟地址）。由「虚拟地址」构成的地址空间就是「虚拟内存」，相同地，由「物理地址」构成的地址空间就是「物理内存」。「虚拟内存」比「物理内存」要大得多。

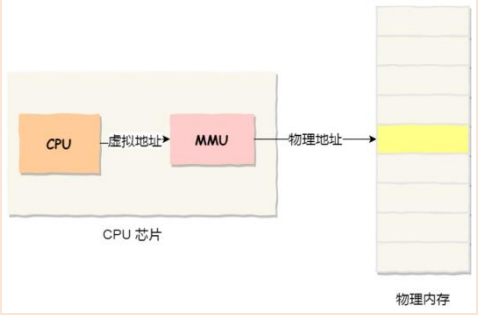
为什么不直接让所有程序使用物理内存而使用虚拟内存呢？

一句话，为了进行内存管理。详细讲就是，（1）方便多道程序设计。操作系统必须同时可以运行多个程序，但是再多的物理地址对于多个应用进程来讲都不够分，所以就需要将不同程序中使用频率低的物理地址回收。程序访问的虚拟地址始终不变，虚拟地址映射的物理地址动态调整，这样子虚拟地址和物理地址也不必是一一对应。（2）安全性。每个程序都可以直接访问所有的物理地址很

不安全，理想的情况是每个程序只能访问特定区域的地址空间。

请你介绍下操作系统的虚拟内存？为什么要使用虚拟内存？

CPU 中的内存管理单元 MMU（Memory Management Unit）会将程序指令访问的「虚拟地址」转换为实际的物理地址」进行执行。

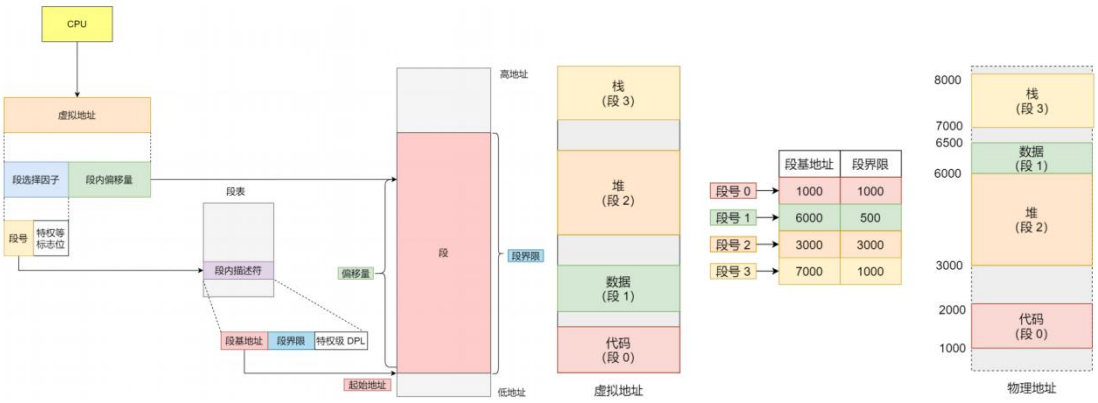


三、内存的分段分页

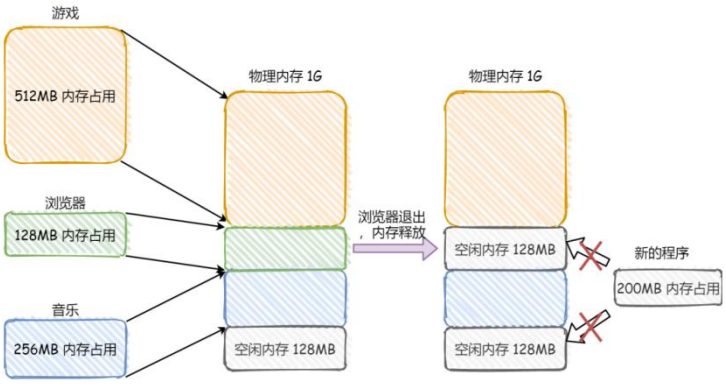
操作系统通过分段分页的方式来管理「虚拟地址」和「物理地址」之间的映射关系。

1、内存分段

分段机制将一个进程中的所有程序指令划分为若干个逻辑分段（Segmentation）：代码段、数据段、堆段和栈段。不同的段有不同的属性。在分段机制中，「虚拟地址」和「物理地址」之间通过「段表」进行映射。操作系统给每一个进程分配的「虚拟地址」由「段选择子」和「段内偏移量」组成。「虚拟地址」通过「段表」与物理地址进行映射的，分段机制会把程序的虚拟地址分成 4 个段，每个段在段表中有一个项，在这一项找到段的基地址，再加上偏移量，于是就能找到物理内存中的地址，如下图：

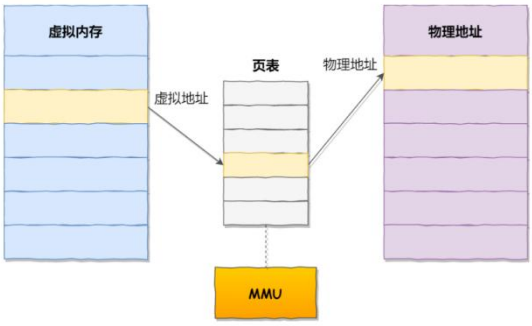


这种分段机制的好处是可以产生连续的内存空间，但是存在的显著问题就是存在内存碎片和内存交换效率低的问题。在多进程的系统中，内存碎片经常产生，内存碎片也分为外部内存碎片和内部内存碎片，解决外部内存碎片的常见方法就是进行内存交换（Swap）：将音乐程序占的 256MB 内存写入硬盘里再分配到已被占用的 512MB 内存下，新的 200MB 程序就有地方了，由于硬盘访问速度慢，如果交换内存的程序过大则效率较低会引起计算机卡顿。



2、内存分页

内存分段可以产生连续的内存空间但是会出现很多内存碎片，为了解决这个问题提出了内存分页技术（Paging）。物理内存一般划分为多个页框（page frame），虚拟内存也要划分为多个页面（page），虚拟内存的页面和物理地址的页框大小一致，Linux 中每一页为 4KB。在分页技术中，「虚拟地址」和「物理地址」之间通过「页表」进行映射。



页表是存储在内存里的，内存管理单元（MMU）就做将虚拟内存地址转换成物理地址的工作。而当进程访问的虚拟地址在页表中查不到时，系统会产生一个缺页异常，进入系统内核空间分配物理内存、更新进程页表，最后再返回用户空间，恢复进程的运行。

分页和分段的不同点？分页技术的优势？

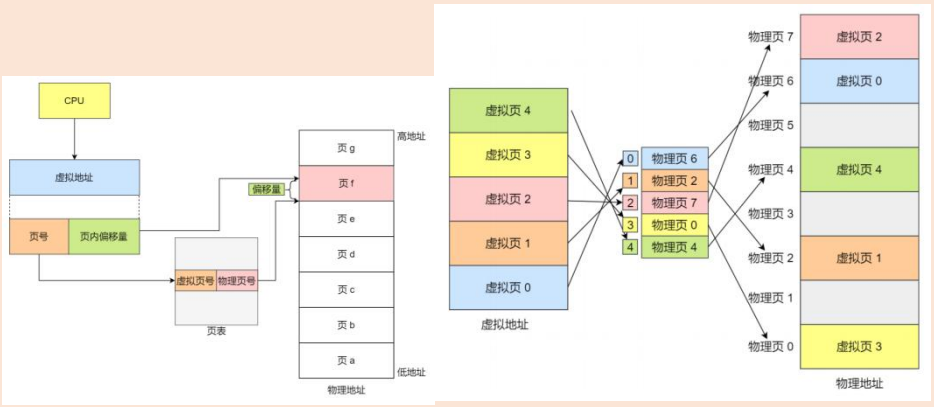
（1）分页在一开始就将整个内存空间按页划分，内存的分配和释放都是按照页为单位；分段则是给每个进程内部按段划分，所以分页技术不会产生间隙非常小的无法利用的内存，只有空闲的页；而分段技术可能由于空闲内存不足而无法分配给新进程。

（2）分页技术允许空闲内存不足时可以将其他运行的进程中的空闲页或者访问频率少的页释放掉写入硬盘中（换出，swap out），需要该页的数据时再重新加载（换入，swap in），这样可以充分利用更多的内存页，效率提升。

（3）分页技术允许在加载进程中的程序指令时，不需要一次性将所有程序加载到物理内存中，完全可以先建立虚拟内存和物理内存的页映射，在程序运行中，需要用到对应虚拟内存页里面的指令和数据时，再加载到物理内存里面去。

分页技术中如何进行虚拟地址和物理地址的映射？

在分页机制下，虚拟地址分为两部分，**页号**和**页内偏移**。**页号**作为**页表**的**索引**，页表包含**物理页**每页所在物理内存的**基地址**，这个**基地址**与**页内偏移**的组合就形成了**物理内存地址**，见下图。

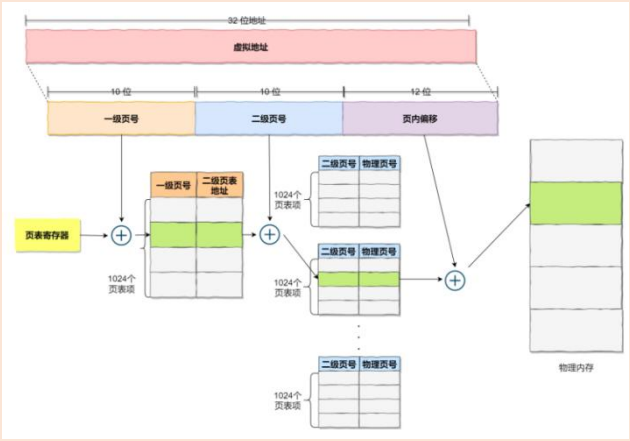


总结一下，对于一个内存地址转换，其实就是这样三个步骤：

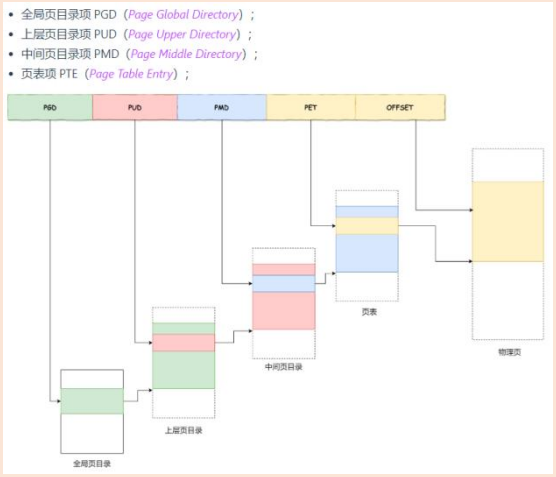
- 把虚拟内存地址，切分成页号和偏移量；
- 根据页号，从页表里面，查询对应的物理页号；
- 直接拿物理页号，加上前面的偏移量，就得到了物理内存地址。

在 Linux 操作系统中，系统为每个进程默认分配 4GB 的虚拟内存，其中 1GB 作为内核空间，3GB 作为用户空间。假设一个页是 4KB 大小，则 4GB 共有大约 100 万个页，假设每个页表项使用 4 个字节来存储，那么 4GB 共需要 4MB 的内存来存储页表。假设有 100 个进程，则需要 400MB 内存来存储页表。

为了减少页表的占用，操作系统进一步采用了多级页表技术。多级页表技术将原有的页表进一步分为固定的 1024 个二级页表，一级页表存储二级页表目录，每个二级页表再包含固定的 1024 个页表项，这样就形成二级分页，如下图：

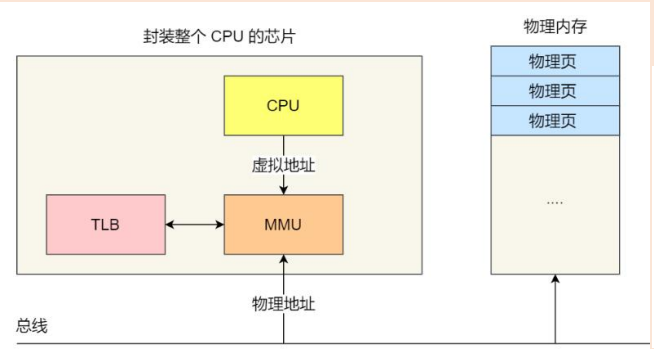


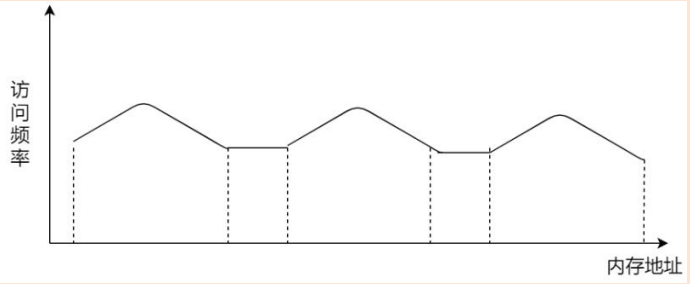
二级分页后其可以映射的页表项数量并没有大幅度降低（1024*1024），但是整个一级页表只有 4KB（4*1024），二级页表可以在需要时被创建，这样就可以减少页表的内存占用。在 64 位系统中，使用的如下四级目录：



多级页表技术的成功在于一是按需创建的局部性原理，二是用时间换空间，减少了空间但是提升了时间。多级页表虚拟地址到物理地址的转换就多了几道转换的工序，这显然就降低了这俩地址转换的速度，也就是带来了时间

上的开销。为了进一步解决这个问题，设计者在 CPU 中加入了一个专门存放程序最常访问的页表项的缓存区，即页表缓存（Translation Lookaside Buffer,TLB），也叫做转址旁路缓存、快表等。TCB 的命中率是非常高的，因为程序在一段时间内访问的内存是固定的。



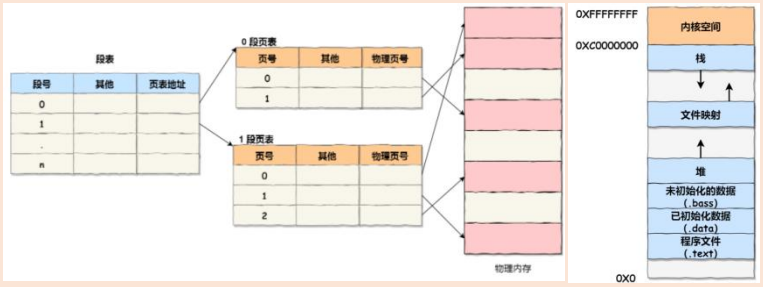


现代操作系统中如何进行内存管理？什么是段页式内存管理？

内存分页技术虽然改进了内存分段技术的不足，但是其并没有取代内存分段，而是和内存分段一起形成了现在操作系统的内存管理机制—段页式内存管理：

整个内存空间按页划分。Linux 操作系统默认为一个进程分配 4GB 的内存空间，然后进程的 4GB 内存空间中按段划分为多个有逻辑意义的段，接着再把每个段包含了多个页，也就是对分段划分出来的连续空间，再划分固定大小的页。这样虚拟地址结构就由段号、段内页号和页内位移三部分组成。

用于段页式地址变换的数据结构是每一个程序一张段表，每个段又建立一张页表，段表中的地址是页表的起始地址，而页表中的地址则为某页的物理页号，如图所示：



第 2 章 进程与线程

一、进程的概念

什么是进程？进程和程序之间的关系是什么？

程序在运行前是存储在硬盘里的静态文件，运行时才被 CPU 加载到内存。**进程本质上就是一个正在运行的程序**，是对运行时程序的封装，其包含了该程序运行的所有资源和操作。简单说，进程就是一个程序的执行流程，内部保存程序运行所需的资源。

CPU、程序、进程和地址空间的关系可以按照以下比喻来理解：

> CPU 的每个核是一个厨师；程序是保存在存储箱里的食谱，其平时一直放在硬盘里，做菜时拿出来放入内存；
>
> 程序的各种输入数据就是做菜的各种原料；程序运行后可以实现的各种功能就是最终做好的菜品；
>
> （内存）地址空间是做菜的厨房；进程就是厨师阅读食谱、获取各种原料并做菜的一系列动作的总和，厨师烹饪每一道菜品都必须在对应的厨房按照对应的食谱进行。

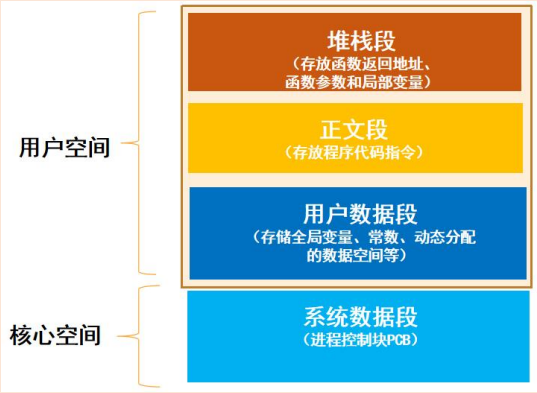
你知道操作系统的进程控制块/PCB 吗？你知道操作系统如何描述一个进程吗？

Linux 系统中一个进程被分配的「虚拟内存」可以分为三段：

（1）数据段：存放全局变量、常数、以及动态分配的数据空间。根据实现的位置不同分为用户数据段和系统数据段。

（2）正文段：存放程序代码。

（3）堆栈段：存放函数返回地址、函数参数以及局部变量。



用户数据段在用户空间实现，系统数据段在内核空间实现。每一个进程的系统数据段存放在操作系统中的进程表，这个系统数据段也叫做进程控制块 PCB（Process Control Block），也是操作系统唯一描述进程的单位。操作系统为每个进程都维护一个 PCB，用来描述当前存在的进程的基本情况和状态变化。

二、进程的状态

1、进程为什么要状态切换

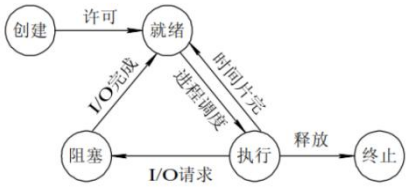
单核的 CPU 在任意时刻只能运行一个程序（一个厨师在任意时刻只能按照一个食谱做一道菜），但是现在的复杂操作系统基本都是多任务的，即 CPU 可以同时运行多个程序。

> 一个厨师在任意时刻只能按照一个食谱进行烹饪，每工作一段时间就切换到另一个食谱。这样子的好处是，对每个菜品来讲，每次它被烹饪时都独占了厨师的精力，对顾客来讲，这些菜品看起来是同时在烹饪的。

同样的原理，单核 CPU 在任意时刻只能运行一个程序，所以单核 CPU 在每个时间片（由程序计时器计数的一小段时间）内运行一个程序，时间片时结束就切换到另一个程序。由于每个时间片足够小，对每个程序来讲，每次运行都似乎独占了 CPU 的资源；对每个用户来讲，每个程序似乎都可以同时运行。我们将这种机制称为 CPU 的并发。

2、一般进程的状态切换

在前文描述中，我们已经知道，同一个进程存在多个状态，具体来讲，进程从创建到终止大致分为如下 5 种状态，即进程的五态模型：



状态类型	状态特点	状态切换条件
创建态	创建进程，调度程序为其分配内存空间等资源	如果成功获取除处理机以外的其他资源则自动进入就绪态。
就绪态	因为其他进程正在运行而暂时停止。	等待分配处理机资源，得到后可立即运行
执行态 （运行态）	该进程正在运行，处于独占用一个 CPU 核心的时间片	时间片用尽后自动进入就绪状态
阻塞态	该进程正在等待某一事件发生而暂时停止运行。	比如等待客户端连接或用户输入。
终止态	进程结束，资源被回收。	可能是任务结束或者遇到已知问题而主动结束也可能是遇到严重未知错误或被其他进程杀死而被动结束

三、Linux 进程的创建

进程的创建主要发生在三类情景：操作系统初始化、用户手动创建新进程、正在运行的进程创建一个进程。

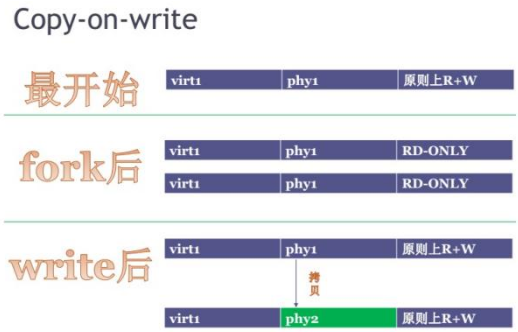
Linux 系统中通过调用库函数接口 **fork()**来复制当前进程创建一个新进程，调用 fork 函数的进程叫做父进程，新创建的进程叫做子进程。Linux 进程创建是**读时共享、写时复制**。

子进程与父进程的区别仅仅在于不同的 PID、PPID 和某些资源及统计量上。注意，Linux 中的 fork()函数使用的是写时复制页的技术，也就是**内核在创建进程时，其资源并没有被复制过来，所有的资源以只读的方式和父进程共享数据，一旦子进程需要写入数据时才进行资源复制**。这种**写时复制技术 COW(Copy-On-Write)**可以使 Linux 拥有快速执行的能力，因此这个优化是非常重要的。详细讲如下：

（1）在 fork 之前，一片内存区对应一份物理地址和一份虚拟地址，内存区的权限为 RW；

（2）在 fork 之后，父子进程看到的内存区虚拟地址相同，物理地址也相同，父子进程使用的其实是同一片物理内存，未发生内存拷贝，操作系统会将此内存区权限改为 RO；

（3）父或子进程对内存区执行写操作将触发 PageFault，操作系统此时会将内存区拷贝一份，父子进程看到的虚拟地址仍然一样，但是物理地址已经不同。



四、Linux 进程的终止

进程的终止有三种情形：程序执行完毕正常结束、程序错误异常结束和外界干预异常结束（被其他进程关闭），有些是自愿退出的，有些是外界关于下非自愿退出的。终止进程的过程如下：

（1）查找需要终止的进程 PCB，如果处于执行状态则立即终止进程；
（2）如果还有子进程则**应将**其所有子进程关闭；
（3）该进程所拥有的全部资源归还给父进程或操作系统；
（4）将其从 PCB 所在队列中删除。

Linux 进程中关闭进程时可能出现孤儿进程或僵尸进程。

（1）孤儿进程

当父进程退出而它的子进程还在运行时，这些子进程就成为**孤儿进程**，孤儿进程将被 Init 进程（PID=1）收养，所以孤儿进程不会浪费系统资源。

（2）僵尸进程

父进程创建子进程后一般需要监控子进程的状态，如果子进程退出但是父进程没有读取其状态，则此时的子进程为僵尸进程。

具体说，在 Linux 系统中，一个进程使用 fork 创建子进程，如果子进程退出但是父进程并没有调用 wait 或 waitpid 来获取子进程的状态信息，那么子进程的进程描述符 task_struct 将会仍然存在，该子进程就是僵尸进程。**僵尸进程会浪费系统资源**。

僵尸进程产生原因：

1、子进程结束后向父进程发出 SIGCHLD 信号，父进程默认忽略了它；
2、父进程没有调用 wait()或 waitpid()函数来等待子进程的结束。

避免僵尸进程方法：

1、父进程调用 wait()或者 waitpid()等待子进程结束，这样处理父进程一般会阻塞在 wait 处而不能处理其他事情。
2、捕捉 SIGCHLD 信号，并在信号处理函数里面调用 wait 函数，这样处理可避免 1 中描述的问题。

（3）守护进程

启动操作系统时，通常会创建若干个进程，其中有些需要同用户交互并替用户完成某些工作，这时这些进程是前台进程，如果进程不需要与用户进行交互但是还有执行一些必要的功能，那么这些进程就是后台进程。

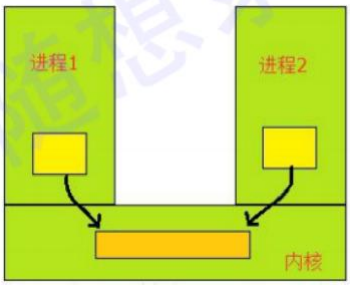
守护进程（daemon，也叫精灵进程）是**一种特殊的后台进程，其通常完全独立于控制终端，并且周期性地执行某种任务或等待处理某些事件，操作系统中有很多守护进程**，例如收发电子邮件的网络进程，在大部分事件下都处于休眠状态，但是当接收到电子邮件时就被唤醒。

五、进程间通信

在很多应用中，进程间的通信可能是不可避免的，因为不同的进程间可能也要进行数据传输、资源共享、通知事件或进程控制。为了实现这些功能，就需要操作系统提供进程间通信的功能。注意这里讨论的进程间通信都是同一主机上的不同进程间的通信，不同主机间的进程通信是网络编程中的套接字（socket）。

1、IPC 的原理

操作系统提供的进程间通信(IPC, InterProcess Communication)就是在内核空间开辟一块缓存区，进程 A 把数据从用户空间拷到内核缓冲区，进程 B 再从内核缓冲区把数据读走。这个缓存区必须在内核中，因为操作系统的每个进程的用户空间都是相互隔离的，彼此数据都不可见，所以进程之间想要进行通信必须通过内核交换数据。



2、IPC 的 5 种技术

请注意，有的文献也将 socket 包含在进程通信中，socket 可以用在不同主机的进程间通信，是网络编程；以下 5 种方法只能用于同一主机间的不同进程间通信。

假设进程 A 要和进程 B 通信，进程间通信的技术有如下几种：

（1）**管道技术（PIPE）**（也叫匿名管道）：是 IPC 中最基本的方式，管道将和另一个进程的通信看做是对一个文件的读写操作，popen()函数类似于 fopen()函数，pipe()函数类似于 open()函数，返回的是对象描述符。**缺陷是：管道必须在亲属进程(同一父进程创建出的相关进程)之间进行数据传输；一个管道只能进行单向通信。**

（2）**命名管道（FIFO）**：命名管道可用于无亲属关系的进程间通信，其原理是通过函数 mkfifo()/mknod()在文件系统中创建一个有路径和名称的 FIFO 文件，不同的进程访问这个文件就可以相互通信。

（3）**消息队列（MSG）**：消息队列是操作系统内核负责维护的一个链表结构，程序员可以创建一个消息队列并获得一个唯一的标识符 key 值，进程双方根据 key 值确定同一个消息队列进程通信，消息必须指定消息类型并封装在专用的 msg 结构中，通信双方可以根据不同的消息类型来获取不同的消息。

	创建	进程 A	进程 B

匿名管道	//创建匿名管道
	int fd[2];//0 读 1 写
	pipe(fd);
		// 子进程 A 负责读 fd[0]
		close(fd[1]);
		read(fd[0],&chr,1);
		printf("%c\n",chr);
		close(fd[0]);
			// 子进程 B 负责写 fd[1]
			close(fd[0]);
			char chr='a';
			write(fd[1],&chr,1);
			close(fd[1]);

命名管道	// 创建命名管道
	mkfifo("fifo",0660);
		char buf[256];
		Int fd=open("fifo",O_RDONLY);
		//子进程读管道中的数据
		read(fd,buf,10);
		buf[10]=0;
		printf("%s",buf);
		close(fd);
			int fd=open("fifo",O_WRONLY);
			//父进程向管道写入数据
			write(fd,"fifo test\n",10);
			close(fd);

消息队列	// 创建消息队列的 key
	#define MSG_KEY 111
		int msqid=msgget(MSG_KEY,0);
		struct msgbuf buf;
		msgrcv(msqid,(void
		*)&buf,sizeof(struct msgbuf),1,0);
		printf("child:rcv a msg
		is %s\n",buf.mtext);
			int mspid=msgget(MSG_KEY,IPC_CREAT|0666);
			struct msgbuf buf;
			buf.mtype=1;
			strcpy(buf.mtext,"Hello World!");
			msgsnd(mspid,(const void *)&buf,sizeof(struct
			msgbuf),0);
			printf("parent:snd a msg is %s\n",buf.mtext);

（4）**信号（Signal）**：一个进程只可以给它的同一进程组的进程发送信号，进程 A 可以向进程 B 发送信号，如果进程 B 此时没有被执行，内核程序会先保存该信号，进程 B 必须提前制定一个信号处理函数来获取所有收到的信号，当 A 的信号达到时，进程 B 立即切换到信号处理函数。我们常用的快捷键很多都使用信号量来实现。该通信方式一般只能用于特定信号的通知，消息的类型一般是固定的，不能传输要交换的任何数据。

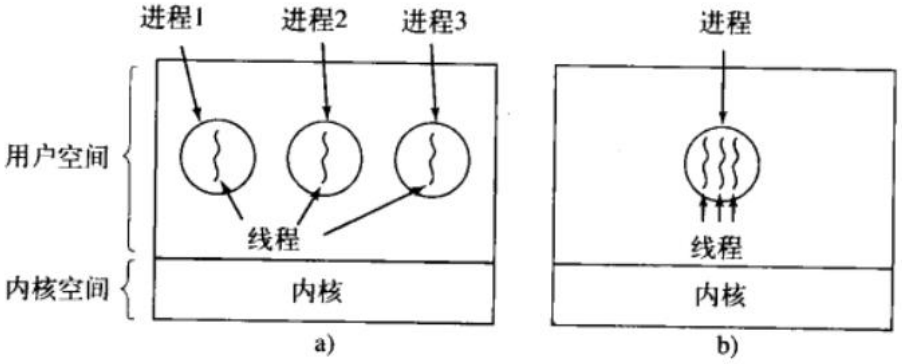
信号和信号量的区别？

信号是进程通信方式；信号量是多线程/进程同步方式，其本质是一个特殊的内核程序中的原子化变量。

（5）**共享内存（SHM）**。是 IPC 中的最快的效率最高的方法之一。因为在内存中操作，所以一个进程向共享内存写入数据，共享的进程立刻就会察觉；允许多个进程同时通信，所以一般需要和信号量等同步手段配合使用。它的实现方式有很多种，主要的有 mmap 系统调用、Posix 共享内存以及 System V 共享内存等。通过这三种“工具”共享地址空间后，通信的目的自然就会达到。

6、线程

线程概念也是随着操作系统不断完善的。传统操作系统中，每个进程都有一个对应的地址空间和一个控制线程。但是现代操作系统中，**同一个进程中可以并行运行多个线程，多个线程共享同一个进程的地址空间和其他资源**等，**在 Linux 系统中线程也叫做轻量级进程**。如下图：



传统操作系统 a） 现代操作系统 b)

（1）线程的优势

使用线程后就可以开发多线程程序，多线程的独特优势？（为什么需要多线程设计？）

（1）提高程序并发性。多线程可以满足需要同时进行多种功能的应用需求。很多应用程序的功能实现需要同时有多种活动同时进行，而且有些活动可能需要阻塞，比如服务器等待客户端连接，服务器在等待客户端连接的同时，可能需要同时处理用户的输入，显然这种需求单进程/单线程是完成不了的。

（2）在并发性程序上比多进程开销更小。多线程是轻量级进程，比单纯的多进程设计更快。多线程是同一个地址空间内的共享同一个地址空间和所有可用数据的并行实体，线程比进程更加轻量级，更容易创建和撤销，在许多系统中创建线程比创建进程要快 10~100 倍。

（3）数据通信、共享数据比多进程方便。多线程可以大大提升需要大量计算或大量 I/O 处理的应用的执行速度。即使一个应用不需要同时执行多种功能，但是只要其计算量大或者读写操作多，都可以考虑多线程。

使用线程的缺点：

1、依赖第三方线程包的库函数，实现上随着库函数迭代可能不稳定；
2、多线程程序的编写和调试都更加困难，维护成本高；
3、进程中的一个线程崩溃时会导致其所属进程中的所有线程崩溃；
4、对信号支持不好。

（2）进程和线程的比较

相同点：

（1）进程和线程的**状态切换基本类似**，都具有就绪、阻塞、运行等状态。

（2）进程和线程在**同步与互斥上机制相似**。

不同点：

（1）**功能不同。**进程是操作系统最小的资源分配的单位，线程是操作系统的最小的执行单位。

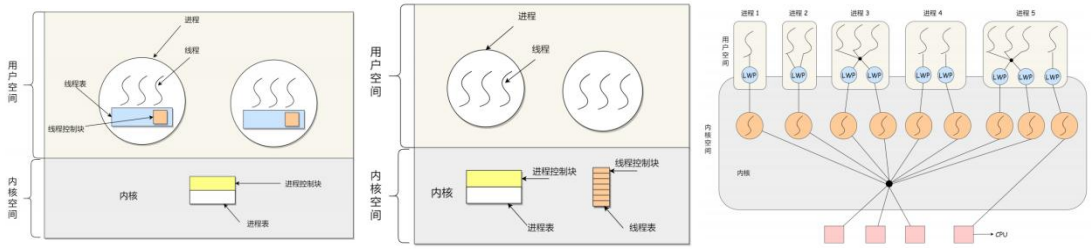
（2）**资源不同。**操作系统只为每一个进程分配资源和回收资源；同一个进程中的所有线程共享该进程的资源，同时每个进程也独享一些资源：寄存器、堆栈、状态和程序计数器。如下：

每个进程中的内容	每个线程中的内容
地址空间 全局变量 打开文件 子进程 即将发生的报警 信号与信号处理程序 账户信息	程序计数器 寄存器 堆栈 状态

（3）**性能不同。**线程切换比进程切换的开销更小，适合程序的并行设计。这是因为进程切换一般需要切换到内核空间，线程一般不需要（Linux 特殊），线程独享的资源少。

（3）线程的 3 种实现方式

有两种主要的方法来实现多线程机制：**用户级线程**和**内核级线程**。前者在操作系统的**用户空间**实现线程包，后者在操作系统的**内核空间**实现线程包。在这两种方式上，Linux 使用了轻量级进程（LightWeight Process），其在内核中支持用户线程。



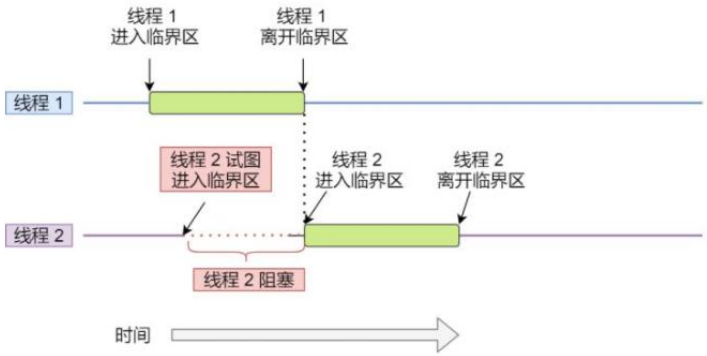
线程类型	特点	优点	缺点
用户级线程	在用户空间实现线程包，在用户空间每个进程都有其独立的线程表	1、**移植性强**，可用于不支持多线程的操作系统 2、**线程切换快**，用户空间的线程切换不需要操作系统在内核态和用户态转换，比内核级线程快一个数量级以上。 3、**自由度高**，允许开发者自定义每个进程中多线程的调度算法。	1、某个用户线程阻塞时该进程下的其他线程都会阻塞 2、某个用户线程运行后且不主动停止，其他线程就不能运行。
内核级线程	在内核空间实现线程包，在内核空间存在记录系统中所有线程的线程表	1、**某个内核线程阻塞时不会影响其他内核线程的执行**。 2、**每个内核线程都可以直接获取 CPU 资源，拥有更多的 CPU 使用时间**	1、需要由内核来维护内核线程的信息，占用内核资源 2、内核线程的创建、终止和切换都是在内核空间中进行的，开销比较大。
轻量级线程（LWP）	轻量级进程(LWP)是建立在内核之上并由内核支持的用户线程，每一个轻量级进程都与一个特定的内核线程关联。	代表系统就是 Linux 系统，综合了内核级线程和用户级线程的优点	

7、多线程同步

CPU 在进程之间的调度可能导致进程在读写一些共享数据时出现问题。（例如进程 A 在读写某个共享变量 a 时恰好被 CPU 打断，然后共享变量 a 被另一个进程 B 进行了修改，这时进程 A 可能会产生错误的结果。）

1、同步和互斥的区别

两个或多个进程读写某些**共享数据**，而最后的结果取决于运行的精确时序，称为**竞争条件**（race condition），包含**竞争条件**的程序大部分运行良好，但是极少数情况会出现错误。想要避免竞争条件，就需要确保一个进程在使用一个共享资源时其他资源不可以做同样的操作，这就是**互斥**（mutual exclusion）。一个进程中会对共享内存进行访问的程序片段叫做**临界区域**（critical region）或**临界区**（critical section）。



同一个进程中的多个线程共享进程内的资源，所以多个线程之间的工作也需要互斥，同时多线程也需要同步。所谓同步就是并发进程/线程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通信息称为进程/线程同步。

同步和互斥的区别：

同步就好比：「操作 A 应在操作 B 之前执行」，「操作 C 必须在操作 A 和操作 B 都完成之后才能执行」等；
互斥就好比：「操作 A 和操作 B 不能在同一时刻执行」；

2、同步和互斥的实现

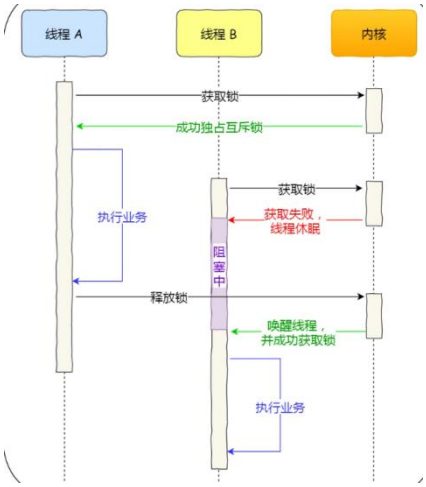
（1）互斥锁

使用加锁(lock)和解锁(unlock)操作来控制对共享资源的访问，就可以解决不同进程或线程之间的互斥问题，这种锁也叫做**互斥锁**或**互斥量**。互斥锁的数据类型是：pthread_mutex_t。

互斥锁的工作原理/流程：

互斥锁是一种互斥机制，本质上是一把锁，在访问共享资源前对加锁，在访问完成后释放锁。
在当前线程加锁后，解锁前，其他任何线程都不能访问该互斥锁锁住的临界区，进入休眠或者忙等待，这体现了互斥性和唯一性。
在当前线程解锁后，内核程序会在等待队列中取出一个等待线程，将其唤醒并让其获得该互斥锁进行执行。

根据回斥锁的实现方式，也可分为**忙等待锁**（**自旋锁**（spin lock））和**无等待锁**。忙等待锁中获取不到锁的线程就会一直进行 while 循环而不做任何事，一直占据 CPU 资源，直到锁可用。无等待锁则是线程没有锁的时候就放入锁的等待队列，然后将 CPU 资源让给其他线程。



互斥锁可以保证任何时刻只能有一个线程访问临界区，请注意把握临界区的选择，如果将线程中的大部分甚至全部代码都设置为临界区，那么线程的执行将趋近于串行执行，失去了并行执行的优势，降低并发效率。

自旋锁加锁和解锁的性能开销比互斥锁要小一点，但是长时间执行会占据 CPU 资源，所以**当临界区的代码执行时间非常短时，应该选用自旋锁，否则使用互斥锁（无等待锁）**。

C++11 通过#include <mutex>提供了互斥锁，该文件提供 4 种锁变量：

```
std::mutex，最基本的 Mutex 类。
std::recursive_mutex，递归 Mutex 类。
std::time_mutex，定时 Mutex 类。
std::recursive_timed_mutex，定时递归 Mutex 类。
```

上述这 4 种锁变量，其中基本都有 lock()、unlock()和 try_lock()等操作，这些操作都需要程序员自行调用。为了更加方便地对互斥量进行上锁和解锁，该文件又提供了 2 种更加方便地上锁的类：

```
std::lock_guard //与 Mutex RAII 相关，上锁后，不能再手动解锁/上锁，只能在声明区结束时自动解锁。
std::unique_lock//与 Mutex RAII 相关，上锁后，可以再次手动解锁或者上锁，声明区结束时自动解锁。
```

unique_lock 效率不如 lock_guard，但是 unique_lock 的颗粒度更细，可以在声明区中手动解锁和上锁。使用 unique_lock 的用法如下：

```
#include <mutex>
std::mutex mtx;// 互斥锁
{
    std::unique_lock<std::mutex> locker(mtx); // 对声明区上锁
    // 执行临界区的代码（保证同一时刻只能有一个线程在执行这段代码）
    locker.unlock();// 临时解锁
    // 执行非临界区的代码（同一时刻可能有多个线程在执行这段代码）
    locker.lock();// 再次加锁
}
```

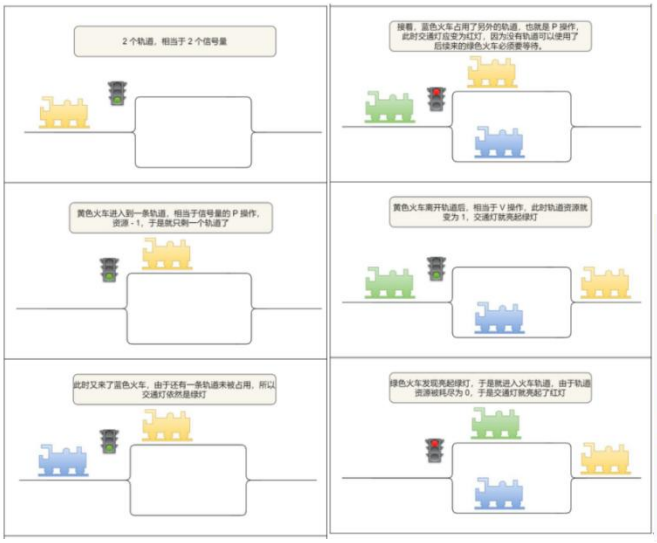
（2）信号量

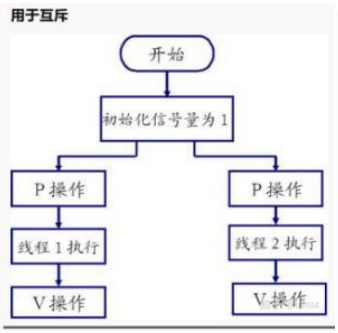
信号量是操作系统提供的一种协调共享资源的方法，它本质是一个特殊变量类型，一个信号量表示系统中某个共享资源的数量。信号量 sem 有两个**原子操作** P（sem）和 V（sem）：（原子操作即单一的不可分割的操作，原子操作要么不执行，要么全部执行，执行中不可中断）。

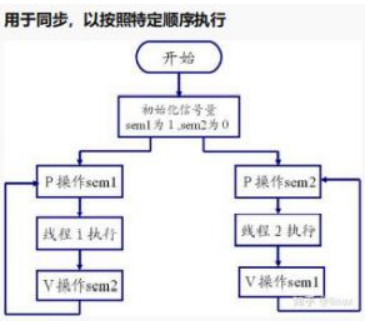
P 操作：将 sem 减 1，相减后，如果 sem<0，则进程/线程进入阻塞等待，否则继续，表明 P 操作可能会阻塞；

V 操作：将 sem 加 1，相加后，如果 sem<=0，唤醒一个等待中的进程/线程，表明 V 操作不会阻塞；

P 操作是用在进入临界区之前，V 操作是用在离开临界区之后，这两个操作是必须成对出现的。举个类比，2 个资源的信号量，相当于 2 条火车轨道，PV 操作如下图过程：







所以信号量也叫做信号灯，最简单的信号量是二进制信号量，即只能取 0 和 1，二进制信号量和互斥锁原理基本就一样。信号量不仅可以实现临界区的互斥访问控制，还可以线程间的事件同步。

（3）条件变量

条件变量是利用线程间共享的全局变量进行同步的线程同步机制；在条件变量上等待的线程以睡眠的方式等待条件变量的满足；一个线程等待"条件变量的条件成立"挂起，另一个线程使"条件成立"；条件变量的使用总是和一个互斥锁结合在一起。

C++11 中条件变量的使用：

```
#include <condition_variable>// 条件变量
std::mutex mtx;// 创建一个互斥锁
std::condition_variable cond;//创建一个条件变量
{
    std::unique_lock<std::mutex> locker(mtx);// 对声明区上锁
    cond.wait(locker);// 阻塞当前线程，释放锁，等待被唤醒
    cnd.wait(locker, []{ return ready;}); //如果条件满足，继续执行;否则阻塞当前线程并释放锁，等待被唤醒
    cnd.wait_for(locker, time_duration, []{ return ready;}); // 如果条件满足或者超时则执行，否则当前线程并释放锁
    cnd.notify_one();//唤醒等待队列中的第一个阻塞线程；不存在锁争用，能够立即获得锁,其余的线程不会被唤醒
    cnd.notify_all();//唤醒等待队列中的所有阻塞线程，存在锁争用，只有一个线程能够获得锁，其余未获取锁的线程继续尝试获得锁
}// 自动解锁
```

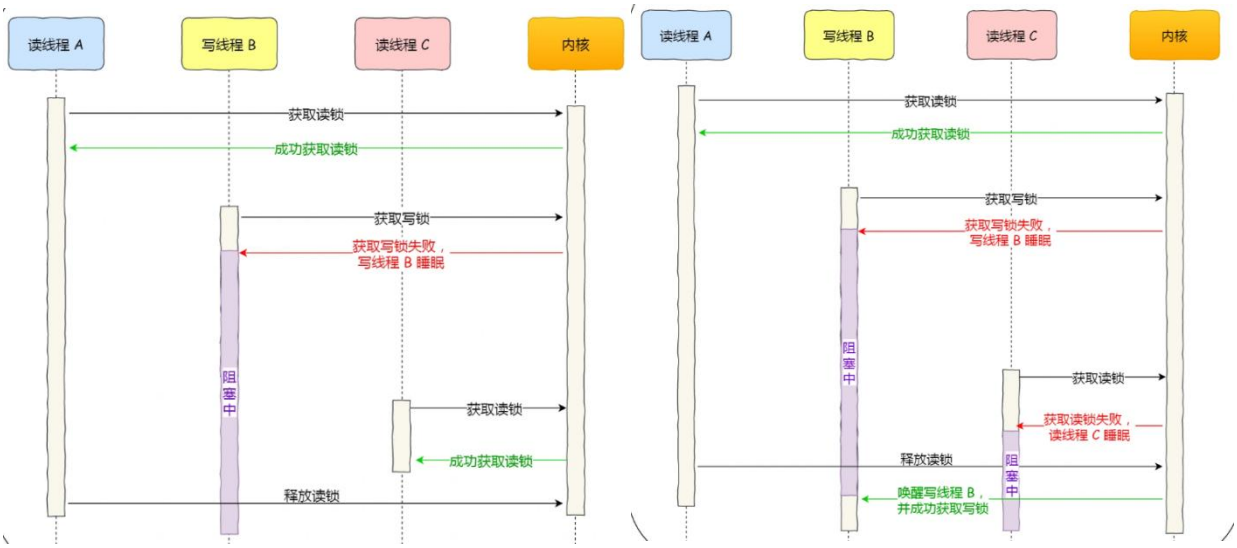
（4）读写锁

在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了**读写锁**来实现。**读写锁适用于能明确区分读操作和写操作的场景。互斥锁和自旋锁都是最基本的锁，**读写锁一般是根据场景来选择这两种锁其中的一个来实现的

读写锁的基本特点是**读读并发**，**读写、写写互斥**。根据实现的不同，读写锁可以分为「**读优先锁**」、「**写优先锁**」和「**公平读写锁**」。

读优先锁是优先服务读线程，其工作方式是：当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 仍然可以成功获取读锁，最后直到读线程 A 和 C 释放读锁后，写线程 B 才可以成功获取写锁。（下图左侧）

写优先锁是优先服务写线程，其工作方式是：当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 获取读锁时会失败，于是读线程 C 将被阻塞在获取读锁的操作，这样只要读线程 A 释放读锁后，写线程 B 就可以成功获取读锁。（下图右侧）



读优先锁对于读线程并发性更好，但如果一直有读线程获取读锁，那么写线程将永远获取不到写锁，造成写线程「饥饿」的现象。写优先锁可以保证写线程不会饿死，但是如果一直有写线程获取写锁，读线程也会被「饿死」。

「**公平读写锁**」不偏袒任何一方，公平读写锁比较简单的一种方式是：**用队列把获取锁的线程排队，不管是写线程还是读线程都按照先进先出的原则加锁即可**，这样读线程仍然可以并发，也不会出现「**饥饿**」的现象。

（5）乐观锁和悲观锁

互斥锁、**自旋锁**、**读写锁**都属于**悲观锁**。

悲观锁做事比较悲观，它认为**多线程同时修改共享资源的概率比较高**，于是很容易出现冲突，它的工作方式是：每次访问共享资源前，先要上锁，操作完成后再解锁。

乐观锁做事比较乐观，它认为**多线程同时修改共享资源的概率比较低**，于是不容易出现冲突，它的工作方式是：先修改完共享资源，再验证这段时间内有没有发生冲突，如果没有其他线程在修改资源，那么操作完成，如果发现有其他线程已经修改过这个资源，就放弃本次操作。你会发现乐观锁全程并没有加锁，所以它也叫**无锁编程**。

放弃后如何重试和具体业务场景息息相关，虽然重试的成本很高，但是冲突的概率足够低的话，还是可以接受的。**乐观锁**虽然去除了加锁解锁的操作，但是一旦发生冲突，重试的成本非常高，所以**只有在冲突概率非常低，且加锁成本非常高的场景时，才考虑使用乐观锁**。

乐观锁的使用场景如：**版本管理工具** SVN、Git 和**在线文档**，它们的主要思想是，同一份共享的资源，先让用户编辑，然后提交的时候，通过资源版本号来判断是否产生了冲突，发生了冲突的地方，需要用户自己修改后再重新提交。

（6）原子操作

在多线程编程中，对某个数据资源的访问必须考虑多线程竞争问题，我们可以为每个数据资源都加上互斥锁，但是类里的每个变量都加上互斥锁实现麻烦还会影响性能。C++11 提供了系统层面的原子操作<atomic>。

atomic 对 int、char、bool 等数据结构进行了原子性封装，在多线程环境中，对 std::atomic 对象的访问不会造成竞争-冒险。利用 std::atomic 可实现数据结构的无锁设计，而且性能比手动加锁要高。

（7）死锁

如果一个进程/线程集合中的每一个进程都在等待只能由该进程/线程集合中的其他进程才能引发的事件，那么该进程集合就是死锁（dead lock）。完全避免死锁是非常困难的。

死锁发生的情景是一个线程/进程需要对多个锁（例如 ABC）进行加锁或者解锁操作；

尽可能避免死锁的方法：

（1）一次性同时加锁，使用 C++提供的 std::lock()

（2）固定顺序来获取锁，线程 1 中以 A、B、C 是顺序获取锁，线程 2 也按照 ABC 顺序。

第 3 章 文件系统

一、文件系统的架构

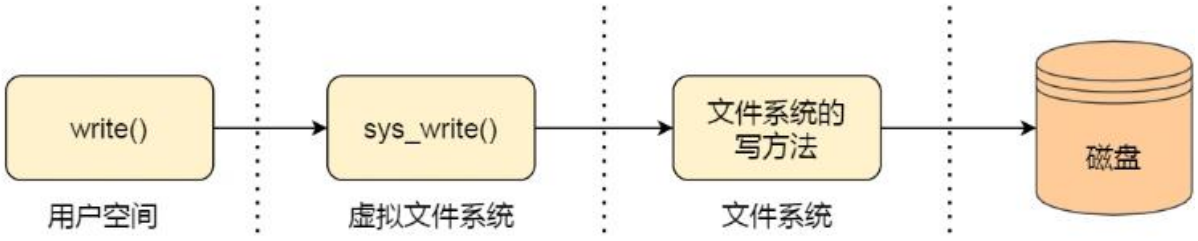
下图则是 Linux 系统文件系统的架构图，其中虚拟文件系统 VFS（Virtual File System）对高层进程和应用程序隐藏了 Linux 支持的所有文件系统之间的区别，为上层服务提供统一的系统调用接口。

2、Linux 的文件系统调用

从程序员角度看文件的话，程序员一般需要通过系统调用函数来打开/操作一个文件，例如：

```
fd = open(name, flag); # 打开文件
...
write(fd,...);         # 写数据
...
close(fd);             # 关闭文件
```

系统调用 write()在操作系统的内核程序中会发生如下调用：



第 4 章 I/O 系统

操作系统的几个概念的区分：，操作系统本质上就是一个周而复始顺序执行的死循环，其不断运行中等待某种事件的发生

事件：操作系统中的消息
信号：操作系统用来管理进程时的异步事件，这个信号一般来自其他进程
中断：计算机中指出现意外情况（多为 I/O 操作），需要 CPU 干预，CPU 会暂停手中工作，停下来专门处理中断，处理完毕后继续执行手中工作。中断分为软件中断和硬件中断。
陷入（trap）：本质是一种软件中断，用户程序执行系统调用时必须让 CPU 从用户态切换到内核态的信号。
异常：CPU 在执行程序指令时发现程序指令有错误而发出的信号

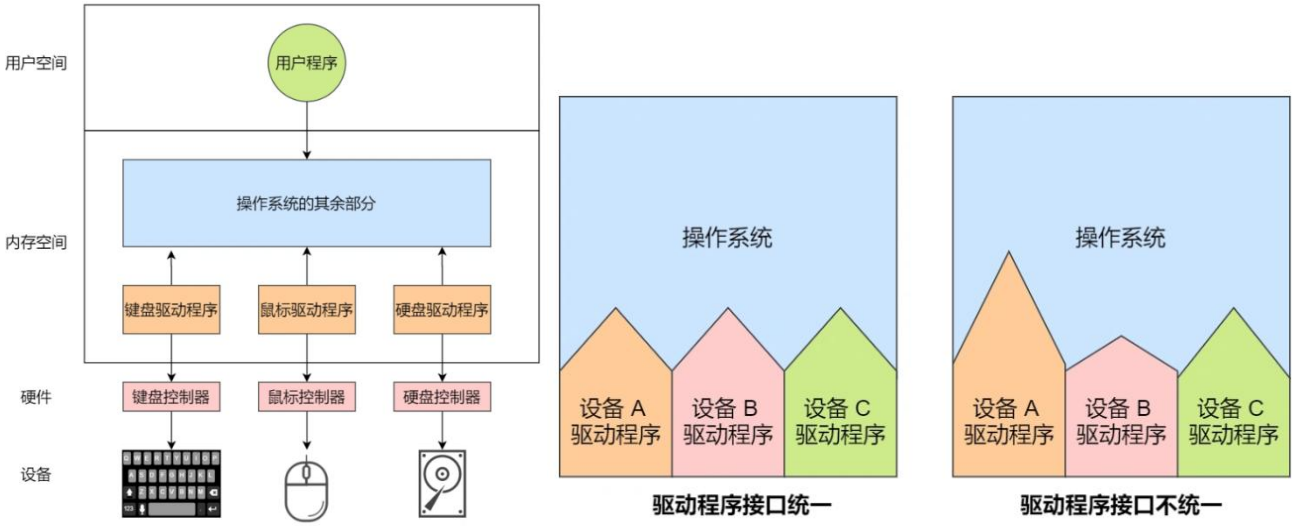
一、I/O 设备管理

I/O 设备也叫做**输入/输出设备**。I/O 设备可以分为两大类：

- **块设备**：以**块**为单位发送或者接收固定大小的数据，**支持寻址**。例如**硬盘**、U 盘、光盘等。
- **字符设备**：以**字符**为单位发送或者接受字符流，**不支持寻址**。例如**鼠标**、**键盘**、**网卡**、打印机等。

后文会讲到，CPU 驱动**块设备**的方式和**字符设备**的方式大不同。

1、设备控制器和设备驱动程序

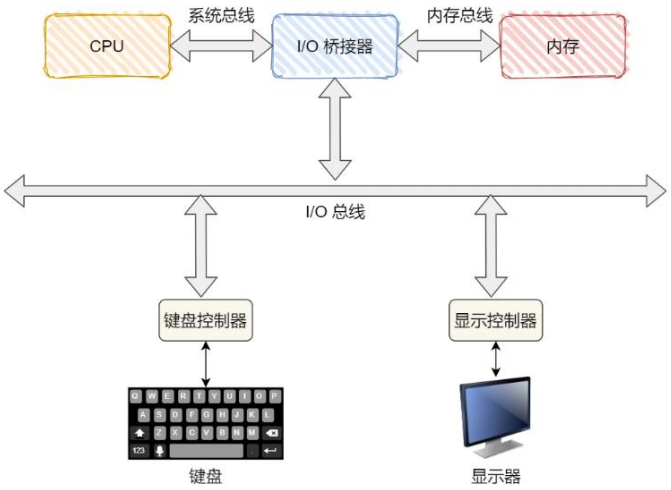


每一个 I/O 设备在**硬件**上都有一个对应的「**设备控制器**」（Device Controller），也叫适配器（adapter），「**设备控制器**」屏蔽了硬件设备的众多细节，减少 CPU 对 I/O 的干预。它本质是 I/O 设备的一种电子组件，由芯片、寄存器和电路组成。块设备传输的数据量很大，通常其设备控制器还集成一个可读写的**数据缓冲区**。

每一个 I/O 设备在软件上都有一个对应的「**设备驱动程序**」，「**设备驱动程序**」属于操作系统层面，它屏蔽了不同**设备控制器**的差异。I/O 设备提供商会针对不同操作系统开发专用的「**设备驱动程序**」，它们会实现该操作系统规定的 API 接口，这样不同的「**设备驱动程序**」就可以以相同或者近似的方式接入操作系统。设备驱动程序在接入 I/O 设备后就会默认安装或者由用户手动安装到操作系统上。

2、I/O 设备控制方式

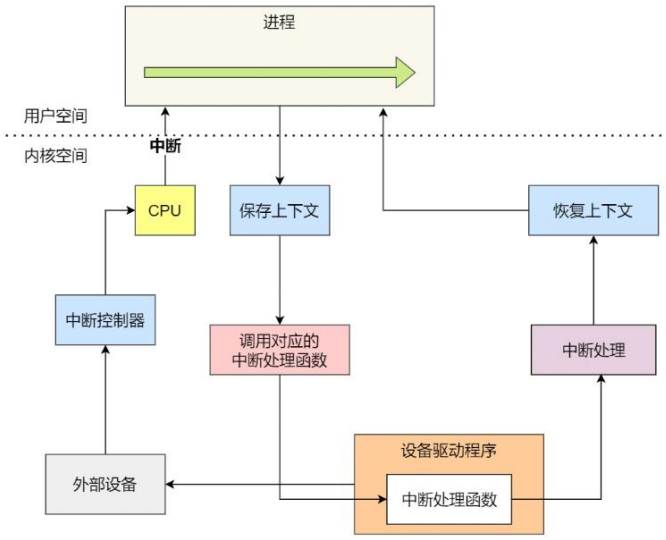
我们先看计算机中 **CPU** 和 **I/O 设备**的架构图：



在上图中，**CPU** 和每个 **I/O 设备**之间存在硬件上的 **I/O 总线**来连接，CPU 如何控制 I/O 设备的读写呢？

方式一：「**程序 I/O 方式**」。在早期计算机系统中，CPU 需要**轮训等待**（也叫**忙等待**）。CPU 定期执行每个 I/O 设备驱动程序，向每个 I/O 设备发出询问，检查其是否需要有请求需要处理，该过程中会占据 CPU 资源，同时由于 CPU 的高速性和 I/O 设备的低速性，会造成 CPU 资源的大量浪费，现代操作系统中已经逐渐淘汰。

方式二：「**中断驱动 I/O 方式**」。当 **I/O 设备**需要处理读写任务时，就会触发**设备控制器**中的**中断控制器**，发送一个**中断请求**给 CPU，CPU 接收到**中断**后就会暂停当前的进程，调用**设备驱动程序**的**中断处理程序**来专门处理**中断请求**。



在 I/O 设备输入每个数据的过程中，由于无需 CPU 干预，因而可使 CPU 与 I/O 设备并行工作，仅当完成一个数据输入时，才需 CPU 花费极短的时间去做一些中断处理。这种驱动方式以**字节**为单位进行 I/O，适合**字符设备**，例如键盘、鼠标。

请注意，中断并不是处理 I/O 设备特有的，**中断**分为**软件中断**和**硬件中断**。软件中断是 CPU 主动执行一条用于中断的 CPU 指令，例如进行系统调用时。

我们以**用户在键盘输入信息**为例，介绍**中断**过程：

当用户在键盘上敲击字符时，**键盘控制器**会产生一种叫做扫描码的数据，将其缓存到**键盘控制器**的寄存器中，然后**键盘控制器**通过 I/O 总线向 CPU 发送**中断请求**。

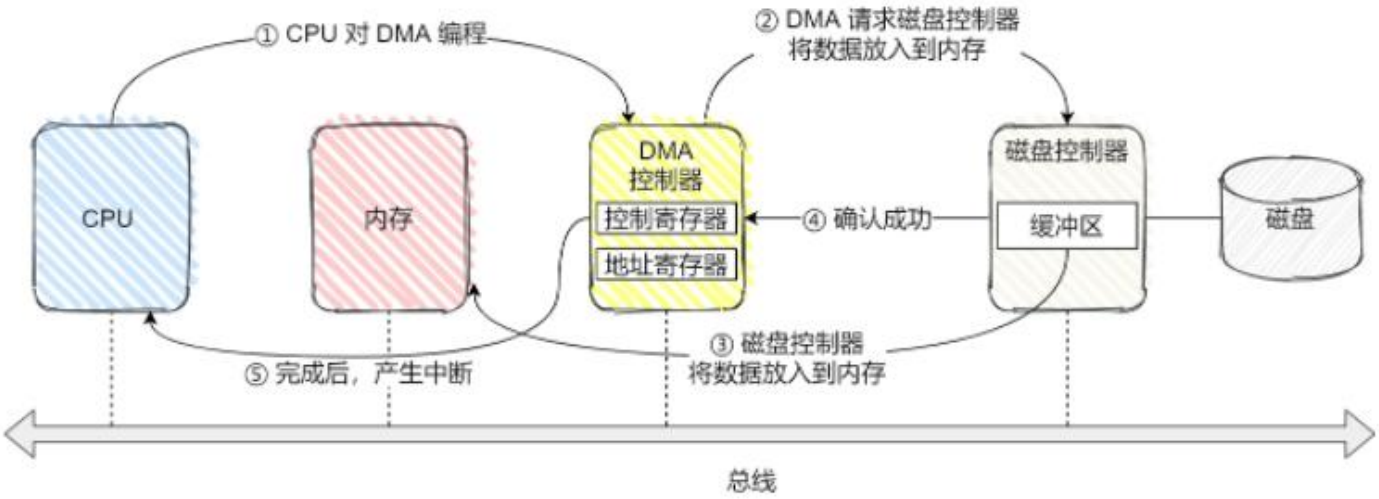
CPU 接收到**中断请求**后，操作系统会保存此时正在执行的进程的 CPU 上下文，然后调用**键盘驱动程序**中的**中断处理程序**：该程序会从**键盘控制器**的**缓冲区**读取扫描码，再根据扫描码找到用户在键盘输入的字符，将其翻译成对应字符对应的 ASCII 码，把 ASCII 码放到**内存**中的「**读缓冲区队列**」。

显示设备的**驱动程序**会定时从内存的「**读缓冲区队列**」读取数据放到「**写缓冲区队列**」，最后把「写缓冲区队列」的数据一个一个写入到显示设备的控制器的寄存器中的**数据缓冲区**，最后将这些数据显示在屏幕里。屏幕显示

出结果后，CPU **恢复被中断进程的上下文**。

方式三：「**直接存储器访问（DMA）I/O 方式**」。**中断 I/O** 适合于键盘、鼠标等**字符**设备，对于**硬盘**等块设备的 I/O 是非常低效的，因为**块设备**传输的一般是**文件**形式的大量数据，每次传输以数据块为单位，而中断 I/O 以字节为单位进行中断。该控制方式依赖于一个叫做 **DMA 控制器**的硬件设备。

DMA（Direct Memory Access，直接存储器访问）控制器可以使**块设备**在 **CPU 不参与**的情况下完成将**数据**从**块设备的缓存区**放入到**内存的缓存区**中。DMA 的工作方式如下：



- CPU 对 DMA 控制器下发指令，告诉它想读取多少数据，读完的数据放在内存的某个地方就可以了；
- 接下来，DMA 控制器会向磁盘控制器发出指令，通知它从磁盘读数据到其内部的缓冲区中，接着磁盘控制器将缓冲区的数据传输到内存；
- 当磁盘控制器把数据传输到内存的操作完成后，磁盘控制器在总线上发出一个确认成功的信号到 DMA 控制器；
- DMA 控制器收到信号后，DMA 控制器发中断通知 CPU 指令完成，CPU 就可以直接用内存里面现成的数据了；

可以看到，CPU 当要读取磁盘数据的时候，只需给 DMA 控制器发送指令，然后返回去做其他事情，当磁盘数据拷贝到内存后，DMA 控制机器通过中断的方式，告诉 CPU 数据已经准备好了，可以从内存读数据了。仅仅在传送开始和结束时需要 CPU 干预。

3、I/O 设备的缓存管理

在进行 I/O 设备管理时，**CPU 处理数据速度非常快**，**I/O 设备处理速度非常慢**，为了缓和 CPU 和 I/O 设备之间速度不配的矛盾，操作系统的内核空间也使用了**缓存区**。

缓存（cache）是一个经常用到的概念，它也在很多地方出现：**CPU** 中存在**缓存**（各种寄存器）、**I/O 设备**的**设备控制器**中存在**缓存**（鼠标、键盘的寄存器、磁盘的缓存区）、**内核空间**中的内存也会开辟专门的**缓存区**，用户空间中的**应用程序**也可能存在**缓存区**。本小节我们只介绍操作系统的**内核空间**如何进行**缓存管理**。

内核空间中使用**数据缓存区**可以具有如下好处：

① **缓和 CPU 和 I/O 设备间速度不匹配的矛盾**。

② 减少对 CPU 的中断频率，放宽对 CPU 中断响应时间的限制。

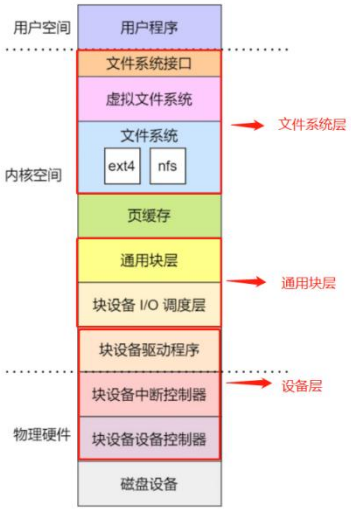
③ **提高 CPU 和 I/O 设备之间的并行性**。

4、Linux 中存储系统分层

操作系统的**文件系统**和**块设备**（磁盘、U 盘等）的 **I/O** 可以暂且统称为 **Linux** 的**存储系统**。可以把 Linux 存储

系统的 I/O 由上到下可以分为三个层次，分别是**文件系统层**、**通用块层**、**设备层**。这三个层次的作用是：

- **文件系统层**：向上为**应用程序**统一提供了标准的**文件访问接口**，向下通过**通用块层**来存储和管理磁盘数据。
- **通用块层**：向上提供访问块设备的标准接口，向下管理不同的设备驱动程序。**Linux 中特有的块设备抽象层**，同时对来自上层的 I/O 请求进行 **I/O 调度**，按照合适的顺序执行 I/O 请求。
- **设备层**：负责执行最终**物理设备**的 I/O 操作。



二、I/O 模型

文件读写方式的各种差异，导致 I/O 的分类多种多样。最常见的有，缓冲与非缓冲 I/O、直接与非直接 I/O、阻塞与非阻塞 I/O、同步与异步 I/O 等。接下来，我们就详细看这四种分类。

1、缓冲 I/O 与非缓冲 I/O

根据**是否利用标准库缓存**，可以把文件 I/O 分为**缓冲 I/O** 与**非缓冲 I/O**。**缓冲 I/O**，是指利用标准库缓存来加速文件的访问，而标准库内部再通过系统调度访问文件。**非缓冲 I/O**，是指直接通过系统调用来访问文件，不再经过标准库缓存。

注意，这里所说的“缓冲”，是指标准库内部实现的缓存。比方说，你可能见到过，很多程序遇到换行时才真正输出，而换行前的内容，其实就是被标准库暂时缓存了起来。无论缓冲 I/O 还是非缓冲 I/O，它们最终还是要经过系统调用来访问文件。我们知道，系统调用后，还会通过**页缓存**，来减少磁盘的 I/O 操作。

2、直接 I/O 与非直接 I/O

根据**是否利用操作系统的页缓存**，可以把文件 I/O 分为**直接 I/O** 与**非直接 I/O**。

- **直接 I/O**：文件读写时，跳过操作系统的页缓存，直接跟文件系统交互来访问文件。
- **非直接 I/O**：文件读写时，先经过操作系统的页缓存，然后再由内核或额外的系统调用，真正写入磁盘。

什么是**页缓存**？

Linux 内核为了减少磁盘 I/O 次数，在系统调用后，会把用户数据拷贝到内核中缓存起来，这个内核缓存空间也就是「**页缓存**」，只有当缓存满足某些条件的时候，才发起磁盘 I/O 的请求。

使用页缓存有两个优势：

（1）内核会对在页缓存操作的 I/O 请求进行 I/O 调度，减少磁盘开销。

（2）内核在页缓存的 I/O 请求存在预读功能。

直接 I/O 和非直接 I/O 本质上还是要通过文件系统访问磁盘的文件。在数据库等场景中，裸 I/O 会跳过文件系

统读写磁盘的情况，也就是我们通常所说的。想要实现直接 I/O，需要你在系统调用中，指定 O_DIRECT 标志。如果没有设置过，默认的是非直接 I/O。

直接 I/O 的常见应用场景如下：

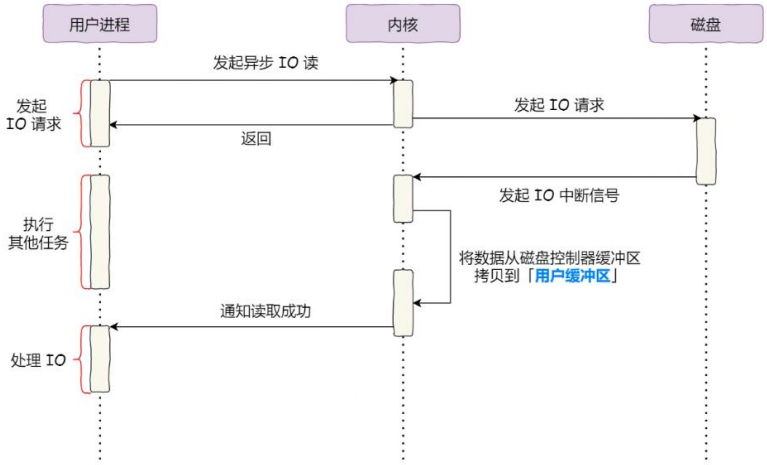
- 应用程序已经实现了磁盘数据的缓存，那么可以不需要 PageCache 再次缓存，减少额外的性能损耗。在 MySQL 数据库中，可以通过参数设置开启直接 I/O，默认是不开启；
  传输大文件。由于大文件难以命中 PageCache 缓存，而且会占满 PageCache 导致「热点」文件无法充分利用缓存，从而增大了性能开销，因此，这时应该使用直接 I/O。

3、同步 I/O 和异步 I/O

根据**是否等待读写的响应结果**，可以把**文件 I/O** 分为**同步 I/O** 和**异步 I/O**。

- **同步 I/O**：是指应用程序执行 I/O 操作后，要一直等到整个 I/O 完成后，才能获得 I/O 响应。
- **异步 I/O**：是指应用程序执行 I/O 操作后，不用等待，可以继续执行。等到这次 I/O 完成后，内核会用事件通知的方式，告诉应用程序。

异步 I/O 不需要等待「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这两个过程。**异步 I/O** 就只支**直接 I/O**，因为它是将数据从**磁盘控制器缓冲区**拷贝到**用户缓存区**，中间不涉及**页缓存**。例如当我们调用 aio_read 之后，内核自动将数据从 I/O 设备的缓存区读取到内存缓存区，再从内存缓存区读取到应用程序的缓存区，这个过程由内核程序自动完成，应用程序并不需要主动拷贝数据。

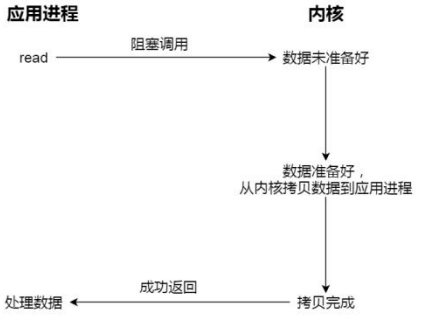


例如，在操作文件时，如果设置了 O_SYNC 或者 O_DSYNC 标志，就代表同步 I/O。如果设置了 O_DSYNC，就要等文件数据写入磁盘后，才能返回；而 O_SYNC，则是在 O_DSYNC 基础上，要求文件元数据也要写入磁盘后，才能返回。再比如，在访问管道或者网络套接字时，设置了 O_ASYNC 选项后，相应的 I/O 就是异步 I/O。这样，内核会再通过 SIGIO 或者 SIGPOLL，来通知进程文件是否可读写。

4、阻塞 I/O、非阻塞 I/O、I/O 复用

根据**应用程序是否阻塞自身运行**，可以把文件 I/O 分为**阻塞 I/O** 和**非阻塞 I/O**

- **阻塞 I/O**：是指执行的系统调用如果没有获得响应就会阻塞当前线程，被操作系统挂起，直到等待的事件发生为止。



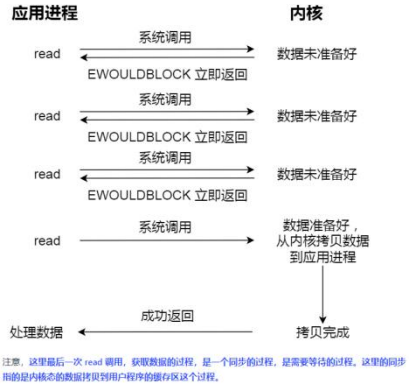

```c
// 先读取鼠标设备的输入
memset(buf,0,sizeof(buf));
fd = open("/dev/input/mouse0",O_RDONLY|O_NONBLOCK);
ret = read(fd,buf,5);//read设置为阻塞，有数据才返回
printf("鼠标读出的内容是：[%s]\n",buf);
// 再读取键盘设备的标准输入
memset(buf,0,sizeof(buf));
ret = read(0,buf,5);//read设置为阻塞，有数据才返回
printf("键盘读出的内容是：[%s]\n",buf);
```

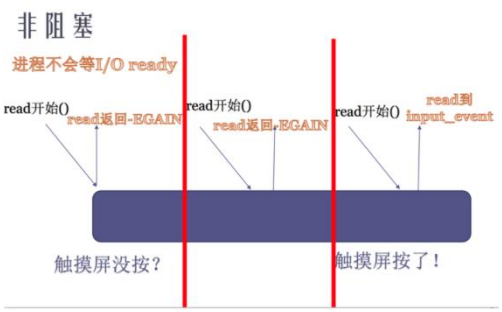
【优点】：在阻塞等待过程中进程被挂起，不消耗 CPU 资源，提升程序性能

【缺点】：阻塞 I/O 只能串行，不适合大量并发场景

【应用场景】适合串行场景，例如各种编程语言提供的 wait()、pause()、sleep()等函数。

- **非阻塞 I/O（也叫并发式 I/O）**：是指应用程序执行的系统调用则总是立即返回，不会阻塞当前的线程，而不管事件是否已经发生。如果事件没有立即发生，这些系统调用就返回-1，和出错的情况一样。此时程序员必须根据 errno 来区分这两种情况。





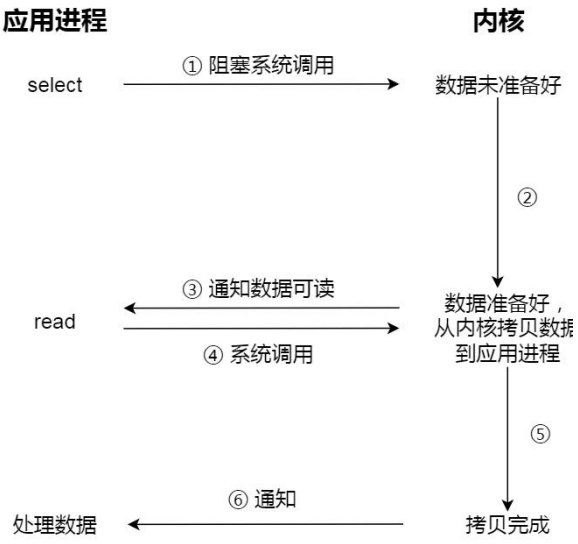

```c
while(1) // 不断检查
{
    // 立即读取鼠标设备的输入
    memset(buf,0,sizeof(buf));
    fd = open("/dev/input/mouse0",O_RDONLY|O_NONBLOCK);
    ret = read(fd,buf,5);//read设置为非阻塞，没有数据也返回
    if(ret > 0){
        printf("鼠标读出的内容是：[%s]\n",buf);
    }
    // 立即读取键盘设备的标准输入
    memset(buf,0,sizeof(buf));
    ret = read(0,buf,5);//read设置为非阻塞，没有数据也返回
    if(ret > 0){
        printf("键盘读出的内容是：[%s]\n",buf);
    }
}
```

【常见优点】适合用于并发场景

【常见缺点】非阻塞 I/O 需要不断检查，可能会消耗大量 CPU 资源

【应用场景】非阻塞 I/O 适用于并发场景，但一般不会单独使用，要和其他 I/O 通知机制一起使用，比如 I/O 复用和 SIGIO 信号。

上述图示是**非阻塞 I/O** 采用轮询方式询问内核程序中的 I/O 数据是否准备好，在并发场景中效率会大大降低，所以基于**非阻塞 I/O** 的 **I/O 复用**技术就出现了，它将 **I/O 事件**分发出去，以**事件**形式通知应用程序，极大提供了对 CPU 的利用率，如下是一个 **I/O 复用**模型 select 的过程：



请注意，**阻塞 I/O**、**非阻塞 I/O**、**I/O 复用**都是一种同步 I/O，因为它们在执行 read 调用时都需要等待「**数据从内核空间拷贝到用户空间**」。

阻塞 I/O 需要等待的是「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这 2 个过程。

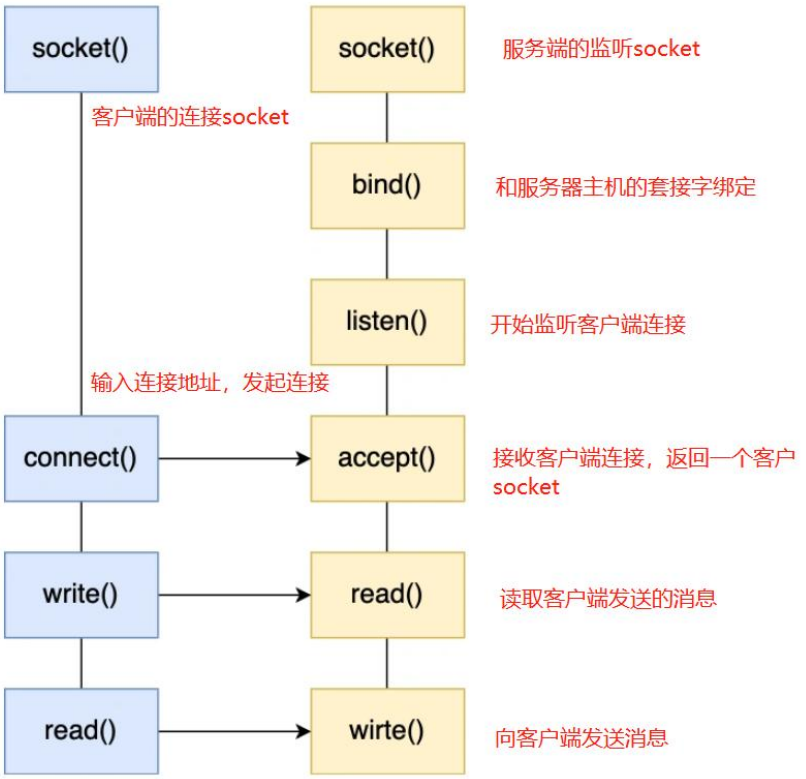
非阻塞 I/O 和 **I/O 复用**需要等待的是「**数据从内核空间拷贝到用户空间**」这 1 个过程。

真正的**异步 I/O** 不需要等待「**内核数据准备好**」和「**数据从内核空间拷贝到用户空间**」这 2 个过程。

比方说，访问管道或者网络套接字时，设置 O_NONBLOCK 标志，就表示用非阻塞方式访问；而如果不做任何设置，默认的就是阻塞访问。

三、I/O 多路复用

想要在不同主机间的进程间通信就必须使用 socket 编程。



socket()创建一个 listened_socket，需要指定网络协议为 IPv4 还是 IPv6

bind()给 listened_socket 绑定一个套接字（IP 地址和端口号）:绑定 IP 地址就是和对应网卡的 IP 地址绑定，一台主机可能有多个网卡；绑定端口号就是选用一个空闲端口号作为当前程序的唯一标识。

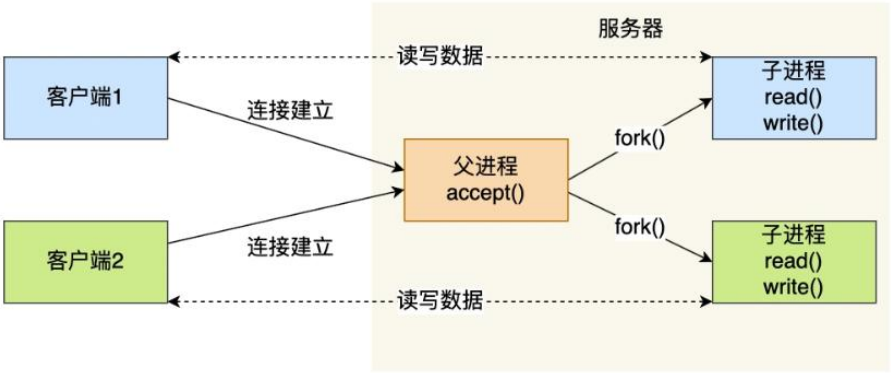
listen()启动当前网络程序来监听绑定的端口号，这个可以作为判定一个网络程序是否启动的条件

accept()从内核程序中来获取客户端的连接，默认采用阻塞 I/O，即没有客户端连接返回就会一直阻塞。

当客户端程序调用 connect()函数来连接指定地址的主机时，客户端和服务端的 TCP 三次握手就开始了，

当内核程序接收到一个客户端连接时，返回一个已经完成连接的 Socket。

上述模型是最基本的网络通信模型，它默认采用了同步阻塞 I/O，基本只能用来一对一通信。

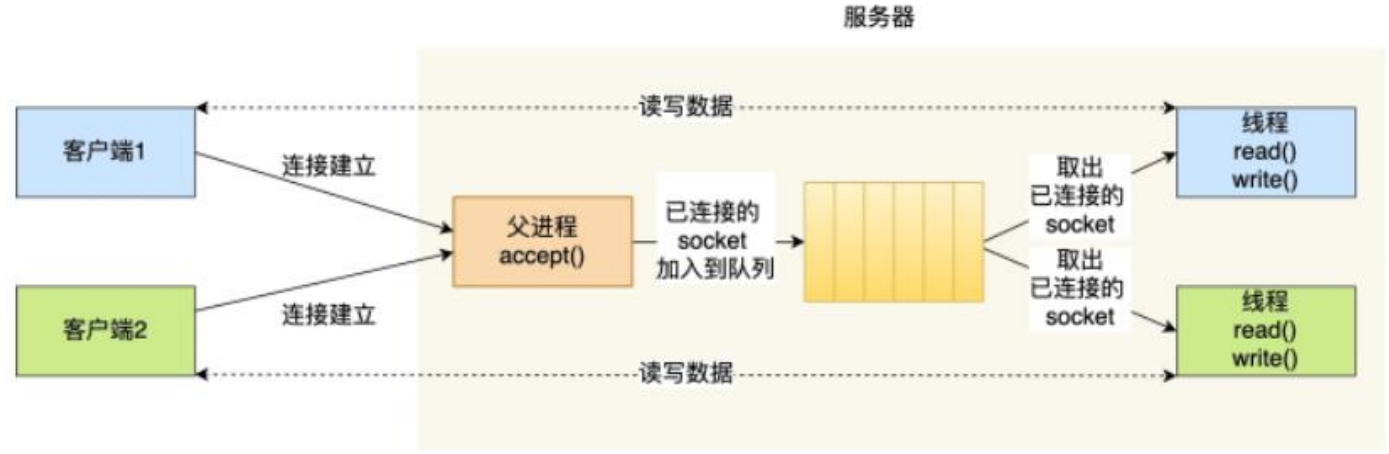


1、多进程模型

多进程模型就是为每个客户端分配一个进程来处理请求。服务器的主进程负责监听客户的连接，一旦与客户端连接完成，accept()函数就会返回一个「已连接 Socket」，这时就通过 fork()函数创建一个子进程，子进程专心处理和「已连接 Socket」的通信。

该模型的不足：创建进程的数量是有上限的（一般不大于 100 个）；频繁创建进程和销毁进程会严重增加系统开销；如果子进程没有做好回收就会成为僵尸进程，最终耗尽系统资源。

2、多线程模型



主进程将接收到的「已连接 Socket」放入一个全局队列，然后使用 pthread_create()来创建线程处理和每个「已连接 Socket」的通信，为了避免频繁地创建和销毁线程，可以使用线程池技术（提前创建若干个线程），同时为了避免多线程竞争，需要互斥锁机制。

该模型的不足：创建线程的数量是有上限的（一般不大于 100 个）。

3、I/O 复用

I/O 复用模型可以在一个进程中维护多个「已连接 Socket」的通信，原理就是多路复用技术：

一个进程虽然任一时刻只能处理一个请求，但是处理每个请求的事件时，耗时控制在 1 毫秒以内，这样 1 秒内就可以处理上千个请求，把时间拉长来看，多个请求复用了一个进程，这就是多路复用，这种思想很类似一个 CPU 并发多个进程，所以也叫做时分多路复用。

阻塞 I/O 需要多进程/线程才能用于并发场景，普通的非阻塞 I/O 需要在应用程序中不断检查才能用于并发场景，两者在并发场景中都不高效。I/O 复用是并发场景中最常使用的 I/O 通知机制。I/O 复用的原理是：

- 应用程序通过 I/O 复用函数向内核程序注册一组事件集合；
- 内核程序不断检查事件集合，通过 I/O 复用函数把其中就绪的事件通知给应用程序。

本质上 IO 复用将普通的非阻塞 I/O 在应用程序中的轮询通过 I/O 复用函数交给了内核程序，所以应用程序可以在一个进程/线程中同时处理多个 I/O 操作。Linux 上常用的 I/O 复用函数是 select、poll 和 epoll_wait。

需要指出的是，I/O 复用函数本身是阻塞的，它们能提高程序效率的原因在于它们可以在同一个线程/进程中具有同时监听多个 I/O 事件的能力，所以 IO 复用可以解决阻塞 I/O 不能用于大量并发的缺陷。

【常见优点】非常适合用于并发场景　【常见缺点】实现复杂。　【应用场景】网络 socket。

（1）select 和 poll

select 和 poll 实现多路复用的方式是：

（一）将已连接 socket 都放到一个文件描述符集合，

（二）然后调用 select/poll 函数，进入阻塞等待。（该函数将文件描述符集合拷贝到内核程序里，让内核通过遍历来检查是否有网络事件产生，当检查到有事件产生后，将此已连接 socket 标记为可读或可写，接着再把整个文件描述符集合拷贝回用户程序里）

（三）用户程序需要遍历 select/poll 函数返回的集合，分类型处理各种情况。

select 和 poll 的区别：

poll 和 select 并没有太大的本质区别，都是使用「线性结构」存储进程关注的 socket 集合，select 使用固定长度的 BitsMap，表示文件描述符集合，该集合默认最大值是 1024；poll 使用动态数组以链表形式来表示文件描述符集合，突破了 select 的文件描述符个数限制，当然还会受到系统文件描述符限制。

select 和 poll 的缺点：

select 和 poll 的程序需要进行 2 次「遍历」文件描述符集合（一次是在内核态里，一次是在用户态里）；还会 2 次「拷贝」文件描述符集合（从用户空间传入内核空间，由内核修改后，再传出到用户空间），其时间复杂度为 O(n)。当客户端数量上升时，性能的损耗也会快速增长。

（2）epoll

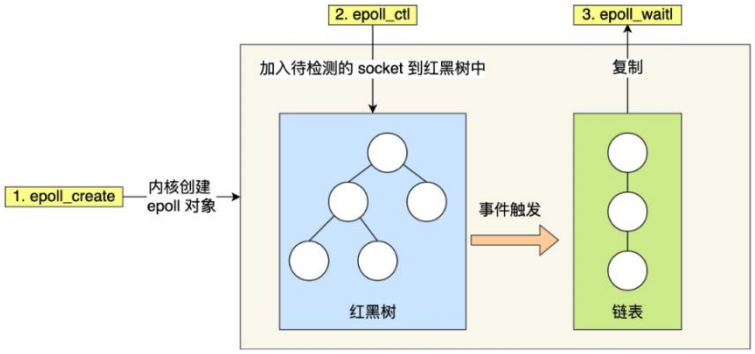
epoll 是 Linux 特有的 IO 复用函数，其实现和使用与 select 或者 poll 有较大差异,其性能也更加高效。epoll 机制需要多个系统调用。

（一）使用 epoll_create()创建 epoll 模型，返回一个 epoll 文件描述符，用来唯一标识要访问的内核事件表
（二）使用 epoll_ctl()向 epoll 模型中注册一个关联 socket 的 epoll 事件（socket 需要设置为非阻塞 I/O）
（三）调用 epoll_wait()来阻塞监听。epoll_wait 函数如果检测到事件，就将所有就绪的事件从内核程序的内核事件表中复制用户程序。
（四）遍历返回的就绪事件，分情况处理不同的 epoll 事件。

epoll 通过如下两个方面很好解决了 select/poll 的问题：

第一点，epoll 在内核程序中使用红黑树来跟踪进程所有待检测的 socket，红黑树是个高效的数据结构，增删查一般时间复杂度是 O(logn)，而且其每次操作都只需要传入一个待检测的 socket，减少了内核和用户空间大量的数据拷贝和内存分配。

第二点，epoll 在内核程序中使用事件驱动机制来返回就绪的 socket，内核维护了一个链表来记录就绪事件，当用户调用 epoll_wait()函数时，只会返回所有的就绪事件。



epoll 支持两种事件触发模式：LT 模式（Level Trigger，水平触发）和 ET 模式（Edge Trigger，边缘触发）。

LT：epoll 的默认工作模式。epoll_wait 每个事件会通知多次，应用程序可以不立即处理，只要满足事件条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户
ET：epoll 的高效工作模式。epoll_wait 每个事件只会通知一次，应用程序必须要立即处理，只有第一次满足条件时才触发，之后就不会再传递同样的事件了。

LT 模式是默认的工作模式，这种模式下 epoll 相当于一个效率较高的 poll。ET 模式在很大程度上降低了同一个 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。select/poll 只有水平触发模式，epoll 默认的触发模式是水平触发，但是可以根据应用场景设置为边缘触发模式。

总结：epoll 是几乎是大规模并行网络程序设计的代名词，一个线程里可以处理大量的 tcp 连接，cpu 消耗也比较低。很多框架模型，nginx, node.js 底层均使用 epoll 实现。

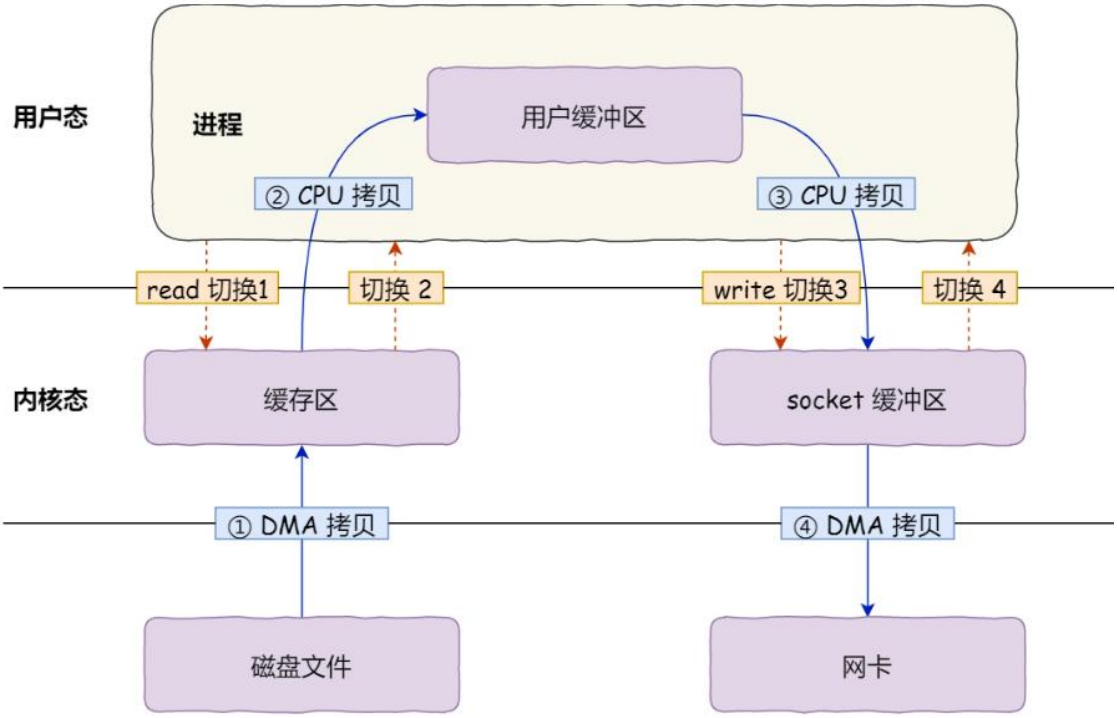
三、文件传输中的零拷贝

在 I/O 设备控制方式中，我们简单介绍了 DMA 技术，它用于磁盘等块设备，自动将数据从磁盘中拷贝到内存的缓存区，或者将数据从内存拷贝到磁盘中，我们暂且将这种拷贝称为 DMA 拷贝。

假设现在在服务端要提供文件传输功能：读取磁盘上的文件，然后通过网络协议发送给客户端。两个文件 I/O 的系统调用如下：

```c
read(file, tmp_buf, len);
write(socket, tmp_buf, len);
```

代码很简单，但是背后发生了很多事情：

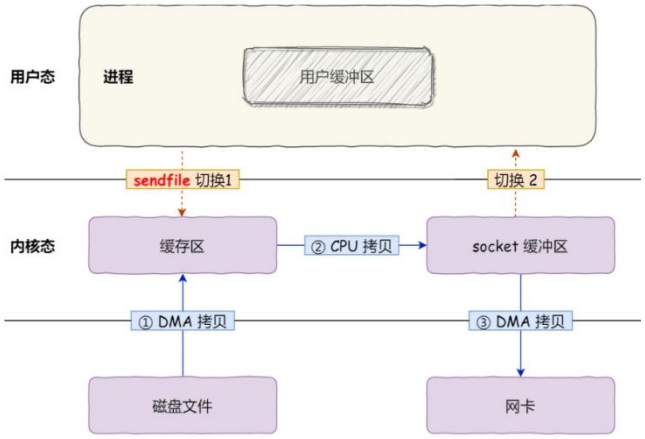


可以看到，两次系统调用共发生了 4 次「**用户态和内核态的上下文切换**」和 4 次「**数据拷贝**」。可以发现，我们发送一份数据，内部却拷贝了 4 次，多余的数据拷贝的时间消耗，在高并发场景下容易被放大和累计，影响性能。

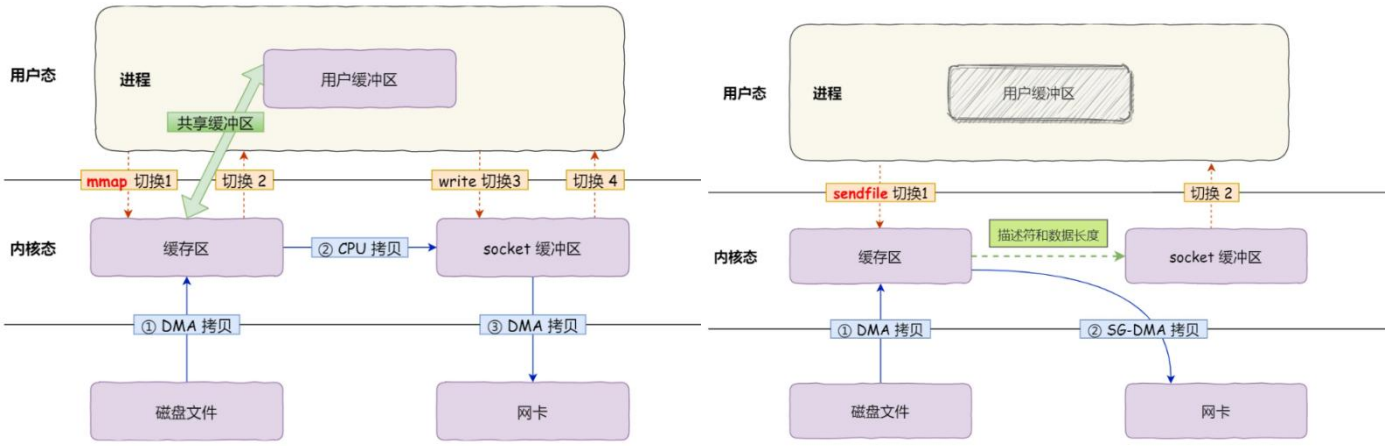
首先在文件传输情景下，用户缓冲区是没必要存在的，因为用户程序并不会修改文件；其次如果可以减少内存中的缓存区，则也会减少「**数据拷贝**」的次数

这种优化思路的实现就是零拷贝技术，零拷贝技术的实现通常存在如下 2 种方式：

（1）使用系统调用函数 mmap+write。用 mmap()替换 read()系统调用函数，mmap()系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，就减少了用户缓存区，减少了一次拷贝。



（2）使用系统调用函数 sendfile。减少一次系统调用，其次如果网卡支持 SG-DMA 技术（The Scatter-Gather Direct Memory Access），可以进一步减少一个缓冲区。



这就是所谓的零拷贝（Zero-copy）技术，因为我们没有在内存层面去拷贝数据，也就是说全程没有通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的。

总结下，零拷贝技术用在文件传输场景下，可以全程不需要 CPU 参与，只通过 DMA 控制，传统文件传输方式，需要 4 次「**用户态和内核态的上下文切换**」和 4 次「**数据拷贝**」，零拷贝技术只需要 2 次「**用户态和内核态的上下文切换**」和 2 次「**数据拷贝**」，文件传输的性能提高至少一倍以上。很多著名的开源项目例如 Kafka 和 Nginx 都使用了零拷贝技术。

接下来介绍磁盘中的**页缓存/磁盘高速缓存**：

CPU 和磁盘之间的读写就是之前说的 CPU 和 I/O 设备的读写，其中间需要添加一个**缓冲区**，对于磁盘来讲，这个缓存区就是**页缓存**（PageCache），也就是**磁盘高速缓存**。请注意，**页缓存**是在逻辑上属于磁盘，在物理上属于内存。

页缓存用来缓存最近被访问的数据，当空间不足时淘汰最久未被访问的缓存。所以，读磁盘数据的时候，优先**在页**缓存找，如果数据存在则可以直接返回；如果没有，则从磁盘中读取，然后缓存 页缓存中。同时，对于机械磁盘来讲，**页缓存也会采取预读功能**，加入 I/O 请求读取 32KB 的数据，但是 CPU 会在页缓存中读取 64KB 的数据，将前 32KB 的数据返回。

当**传输大文件**（GB 级别的文件）**的**时候，**页缓存会不起作用**，这是因为每当用户访问大文件时，如果内核程序就把它们载入页缓存中，页缓存空间将会很快被这些大文件占满，针对大文件的传输，不应该使用 PageCache，也就是说不应该使用零拷贝技术，因为可能由于 PageCache 被大文件占据，而导致「热点」小文件无法利用到 PageCache，这样在高并发的环境下，会带来严重的性能问题。

所以，零拷贝技术使用了页缓存，是一种非直接 I/O，它适合传输热点小文件,不适合传输大文件。传输大文件不使用页缓存，而且采用异步模式，综上，高并发场景下的文件传输：

传输小文件：非直接 I/O+零拷贝技术

传输大文件：直接 I/O+异步 I/O