五、Linux

面向进程和线程学习操作系统。

目录

内容

进程线程模型

线程和进程的概念已经在操作系统书中被翻来覆去讲了很多遍。很多概念虽然都是套话，但没能理解透其中深意会导致很多内容理解不清晰。对于进程和线程的理解和把握可以说基本奠定了对系统的认知和把控能力。其核心意义绝不仅仅是“线程是调度的基本单位，进程是资源分配的基本单位”这么简单。

多线程

我们这里讨论的是用户态的多线程模型，同一个进程内部有多个线程，所有的线程共享同一个进程的内存空间，进程中定义的全局变量会被虽有的线程共享，比如有全局变量int i = 10，这一进程中所有并发运行的线程都可以读取和修改这个i的值，而多个线程被CPU调度的顺序又是不可控的，所以对临界资源的访问尤其需要注意安全。我们必须知道，做一次简单的i = i + 1在计算机中并不是原子操作，涉及内存取数，计算和写入内存几个环节，而线程的切换有可能发生在上述任何一个环节中间，所以不同的操作顺序很有可能带来意想不到的结果。

但是，虽然线程在安全性方面会引入许多新挑战，但是线程带来的好处也是有目共睹的。首先，原先顺序执行的程序（暂时不考虑多进程）可以被拆分

成几个独立的逻辑流，这些逻辑流可以独立完成一些任务（最好这些任务是不相关的）。比如QQ可以一个线程处理聊天一个线程处理上传文件，两个线程互不干涉，在用户看来是同步在执行两个任务，试想如果线性完成这个任务的话，在数据传输完成之前用户聊天被一直阻塞会是多么尴尬的情况。

对于线程，我认为弄清以下两点非常重要：

- 线程之间有无先后访问顺序（线程依赖关系）
- 多个线程共享访问同一变量（同步互斥问题）

另外，我们通常只会去说同一进程的多个线程共享进程的资源，但是每个线程特有的部分却很少提及，除了标识线程的tid，每个线程还有自己独立的栈空间，线程彼此之间是无法访问其他线程栈上内容的。而作为处理机调度的最小单位，线程调度只需要保存线程栈、寄存器数据和PC即可，相比进程切换开销要小很多。

线程相关接口不少，主要需要了解各个参数意义和返回值意义。

1. 线程创建和结束
    - 背景知识：

      在一个文件内的多个函数通常都是按照main函数中出现的顺序来执行，但是在分时系统下，我们可以让每个函数都作为一个逻辑流并发执行，最简单的方式就是采用多线程策略。在main函数中调用多线程接口创建线程，每个线程对应特定的函数（操作），这样就可以不按照main函数中各个函数出现的顺序来执行，避免了忙等的情况。线程基本操作的接口如下。

    - 相关接口：
        - 创建线程：int pthread_create(pthread_t *pthread, const pthread_attr_t* attr, void *(*start_routine)(void *), void* agr);

          创建一个新线程，pthread和start_routine不可或缺，分别用于标识线程和执行体入口，其他可以填NULL。

            - pthread：用来返回线程的tid，*pthread值即为tid，类型pthread_t == unsigned long int。
            - attr：指向线程属性结构体的指针，用于改变所创线程的属性，填NULL使用默认值。

        - start_routine：线程执行函数的首地址，传入函数指针。
        - arg：通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变量的地址，在被传入函数中先强制类型转换成所需类型即可。
    - 获得线程ID：pthread_t pthread_self();

      调用时，会打印线程ID。
    - 等待线程结束：int pthread_join(pthread_t tid, void** retval);

      主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread_join的线程会被阻塞。
        - tid：创建线程时通过指针得到tid值。
        - retval：指向返回值的指针。
    - 结束线程：pthread_exit(void *retval);

      子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread_join获得。
        - retval：同上。
    - 分离线程：int pthread_detach(pthread_t tid);

      主线程、子线程均可调用。主线程中pthread_detach(tid)，子线程中pthread_detach(pthread_self())，调用后和主线程分离，子线程结束时自己立即回收资源。
        - tid：同上。
2. 线程属性值修改
    - 背景知识：

      线程属性对象类型为pthread_attr_t，结构体定义如下：
      `C++ typedef struct{ int etachstate; // 线程分离的状态 int schedpolicy; // 线程调度策略 struct sched_param schedparam; // 线程的调度参数 int inheritsched; // 线程的继承性 int scope; // 线程的作用域 // 以下为线程栈的设置 size_t guardsize; // 线程栈`

`末尾警戒缓冲大小 int stackaddr_set; // 线程的栈设置 void * stackaddr; // 线程栈的位置 size_t stacksize; // 线程栈大小 } pthread_arrt_t;`

- 相关接口：

  对上述结构体中各参数大多有：pthread_attr_get***()和 pthread_attr_set***()系统调用函数来设置和获取。这里不一一罗列。

3. 线程同步

   - 详见同步互斥专题

多进程

每一个进程是资源分配的基本单位。进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以共享。实际上在父进程创建子进程之后，父、子进程除了pid外，几乎所有的部分几乎一样，子进程创建时拷贝父进程PCB中大部分内容，而PCB的内容实际上是各种数据、代码的地址或索引表地址，所以复制了PCB中这些指针实际就等于获取了全部父进程可访问数据。所以简单来说，创建新进程需要复制整个PCB，之后操作系统将PCB添加到进程核心堆栈底部，这样就可以被操作系统感知和调度了。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。如果子进程想要运行自己的代码段，还可以通过调用execv()函数重新加载新的代码段，之后就和父进程独立开了。我们在shell中执行程序就是通过shell进程先fork()一个子进程再通过execv()重新加载新的代码段的过程。

1. 进程创建与结束

   - 背景知识：

     进程有两种创建方式，一种是操作系统创建的一种是父进程创建的。从计算机启动到终端执行程序的过程为：0号进程 -> 1号内核进程 -> 1号用户进程(init进程) -> getty进程 -> shell进程 -> 命令行执行进程。所以我们在命令行中通过 ./program执行可执行文件时，所有创建的进程都是shell进程的子进程，这也就是为什么shell一关闭，在shell中执行的进程都自动被关闭的原因。从shell进程到创建其他子进程需要通过以下接口。

- 相关接口：
  - 创建进程：pid_t fork(void);

    返回值：出错返回-1；父进程中返回pid > 0；子进程中pid == 0
  - 结束进程：void exit(int status);
    - status是退出状态，保存在全局变量中S?，通常0表示正常退出。
  - 获得PID：pid_t getpid(void);

    返回调用者pid。
  - 获得父进程PID：pid_t getppid(void);

    返回父进程pid。
- 其他补充：
  - 正常退出方式：exit()、_exit()、return（在main中）。

    exit()和_exit()区别：exit()是对_exit()的封装，都会终止进程并做相关收尾工作，最主要的区别是_exit()函数关闭全部描述符和清理函数后不会刷新流，但是exit()会在调用_exit()函数前刷新数据流。

    return和exit()区别：exit()是函数，但有参数，执行完之后控制权交给系统。return若是在调用函数中，执行完之后控制权交给调用进程，若是在main函数中，控制权交给系统。
  - 异常退出方式：abort()、终止信号。

2. 僵尸进程、孤儿进程
   - 背景知识：

     父进程在调用fork接口之后和子进程已经可以独立开，之后父进程和子进程就以未知的顺序向下执行（异步过程）。所以父进程和子进程都有可能先执行完。当父进程先结束，子进程此时就会变成孤儿进程，不过这种情况问题不大，孤儿进程会自动向上被init进程收养，init进程完成对状态收集工作。而且这种过继的方式也是守护进程能够实现的因素。如果子进程先结束，父进程并未调用wait或者

waitpid获取进程状态信息，那么子进程描述符就会一直保存在系统中，这种进程称为僵尸进程。

- 相关接口：
- 回收进程（1）：pid_t wait(int *status);

```
一旦调用wait()，就会立即阻塞自己，wait()自动分析某个子进程是否已经退

- status：指向子进程结束状态值。
```

  - 回收进程（2）：pid_t waitpid(pid_t pid, int *status, int options);

    返回值：返回pid：返回收集的子进程id。返回-1：出错。返回0：没有被手机的子进程。

    - pid：子进程识别码，控制等待哪些子进程。
      a. pid < -1，等待进程组识别码为pid绝对值的任何进程。
      b. pid = -1，等待任何子进程。
      c. pid = 0，等待进程组识别码与目前进程相同的任何子进程。
      d. pid > 0，等待任何子进程识别码为pid的子进程。
    - status：指向返回码的指针。
    - options：选项决定父进程调用waitpid后的状态。
      a. options = WNOHANG，即使没有子进程退出也会立即返回。
      b. options = WUNYRACED，子进程进入暂停马上返回，但结束状态不予理会。

3. 守护进程
4. 背景知识：

守护进程是脱离终端并在后台运行的进程，执行过程中信息不会显示在终端上并且也不会被终端发出的信号打断。

5. 操作步骤：
    - 创建子进程，父进程退出：fork() + if(pid > 0){exit(0);}，使子进程称为孤儿进程被init进程收养。
    - 在子进程中创建新会话：setsid()。
    - 改变当前目录结构为根：chdir("/")。
    - 重设文件掩码：umask(0)。
    - 关闭文件描述符：for(int i = 0; i < 65535; ++i){close(i);}。
6. Linux进程控制
7. 进程地址空间（地址空间）

    虚拟存储器为每个进程提供了独占系统地址空间的假象。尽管每个进程地址空间内容不尽相同，但是他们的都有相似的结构。X86 Linux进程的地址空间底部是保留给用户程序的，包括文本、数据、堆、栈等，其中文本区和数据区是通过存储器映射方式将磁盘中可执行文件的相应段映射至虚拟存储器地址空间中。有一些"敏感"的地址需要注意下，对于32位进程来说，代码段从0x08048000开始。从0xC0000000开始到0xFFFFFFFF是内核地址空间，通常情况下代码运行在用户态（使用0x00000000 ~ 0xC00000000的用户地址空间），当发生系统调用、进程切换等操作时CPU控制寄存器设置模式位，进入内和模式，在该状态（超级用户模式）下进程可以访问全部存储器位置和执行全部指令。也就说32位进程的地址空间都是4G，但用户态下只能访问低3G的地址空间，若要访问3G ~ 4G的地址空间则只有进入内核态才行。
8. 进程控制块（处理机）

    进程的调度实际就是内核选择相应的进程控制块，被选择的进程控制块中包含了一个进程基本的信息。
9. 上下文切换

    内核管理所有进程控制块，而进程控制块记录了进程全部状态信息。每一次进程调度就是一次上下文切换，所谓的上下文本质上就是当前运行状态，主要包括通用寄存器、浮点寄存器、状态寄存器、程序计数器、

用户栈和内核数据结构（页表、进程表、文件表）等。进程执行时刻，内核可以决定抢占当前进程并开始新的进程，这个过程由内核调度器完成，当调度器选择了某个进程时称为该进程被调度，该过程通过上下文切换来改变当前状态。一次完整的上下文切换通常是进程原先运行于用户态，之后因系统调用或时间片到切换到内核态执行内核指令，完成上下文切换后回到用户态，此时已经切换到进程B。

线程、进程比较

关于进程和线程的区别这里就不一一罗列了，主要对比下线程和进程操作中主要的接口。

- fork()和pthread_create()

  负责创建。调用fork()后返回两次，一次标识主进程一次标识子进程；调用pthread_create()后得到一个可以独立执行的线程。

- wait()和pthread_join()

  负责回收。调用wait()后父进程阻塞；调用pthread_join()后主线程阻塞。

- exit()和pthread_exit()

  负责退出。调用exit()后调用进程退出，控制权交给系统；调用pthread_exit()后线程退出，控制权交给主线程。

进程间通信

Linux几乎支持全部UNIX进程间通信方法，包括管道（有名管道和无名管道）、消息队列、共享内存、信号量和套接字。其中前四个属于同一台机器下进程间的通信，套接字则是用于网络通信。

管道

- 无名管道
  - 无名管道特点：
    - 无名管道是一种特殊的文件，这种文件只存在于内存中。

        - 无名管道只能用于父子进程或兄弟进程之间，必须用于具有亲缘关系的进程间的通信。
        - 无名管道只能由一端向另一端发送数据，是半双工方式，如果双方需要同时收发数据需要两个管道。
    - 相关接口：
        - int pipe(int fd[2]);
            - fd[2]：管道两端用fd[0]和fd[1]来描述，读的一端用fd[0]表示，写的一端用fd[1]表示。通信双方的进程中写数据的一方需要把fd[0]先close掉，读的一方需要先把fd[1]给close掉。
- 有名管道：
    - 有名管道特点：
        - 有名管道是FIFO文件，存在于文件系统中，可以通过文件路径名来指出。
        - 无名管道可以在不具有亲缘关系的进程间进行通信。
    - 相关接口：
        - int mkfifo(const char *pathname, mode_t mode);
            - pathname：即将创建的FIFO文件路径，如果文件存在需要先删除。
            - mode：和open()中的参数相同。

消息队列

共享内存

进程可以将同一段共享内存连接到它们自己的地址空间，所有进程都可以访问共享内存中的地址，如果某个进程向共享内存内写入数据，所做的改动将立即影响到可以访问该共享内存的其他所有进程。

- 相关接口
    - 创建共享内存：int shmget(key_t key, int size, int flag);

成功时返回一个和key相关的共享内存标识符，失败范湖范围-1。

- key：为共享内存段命名，多个共享同一片内存的进程使用同一个key。
- size：共享内存容量。
- flag：权限标志位，和open的mode参数一样。

◦ 连接到共享内存地址空间：void *shmat(int shmid, void* addr, int flag);

返回值即共享内存实际地址。

- shmid：shmget()返回的标识。
- addr：决定以什么方式连接地址。
- flag：访问模式。

◦ 从共享内存分离：int shmdt(const void *shmaddr);

调用成功返回0，失败返回-1。

- shmaddr：是shmat()返回的地址指针。

- 其他补充

共享内存的方式像极了多线程中线程对全局变量的访问，大家都对等地有权去修改这块内存的值，这就导致在多进程并发下，最终结果是不可预期的。所以对这块临界区的访问需要通过信号量来进行进程同步。

但共享内存的优势也很明显，首先可以通过共享内存进行通信的进程不需要像无名管道一样需要通信的进程间有亲缘关系。其次内存共享的速度也比较快，不存在读取文件、消息传递等过程，只需要到相应映射到的内存地址直接读写数据即可。

信号量

在提到共享内存方式时也提到，进程共享内存和多线程共享全局变量非常相似。所以在使用内存共享的方式是也需要通过信号量来完成进程间同步。多

线程同步的信号量是POSIX信号量，
而在进程里使用SYSTEM V信号量。

- 相关接口
  - 创建信号量：int semget(key_t key, int nsems, int semflag);

    创建成功返回信号量标识符，失败返回-1。

    - key：进程pid。
    - nsems：创建信号量的个数。
    - semflag：指定信号量读写权限。
  - 改变信号量值：int semop(int semid, struct sembuf *sops, unsigned nsops);

    我们所需要做的主要工作就是串讲sembuf变量并设置其值，然后调用semop，把设置好的sembuf变量传递进去。

    struct sembuf结构体定义如下：
    `C++ struct sembuf{ short sem_num; short sem_op; short sem_flg; };`
    成功返回信号量标识符，失败返回-1。

    - semid：信号量集标识符，由semget()函数返回。
    - sops：指向struct sembuf结构的指针，先设置好sembuf值再通过指针传递。
    - nsops：进行操作信号量的个数，即sops结构变量的个数，需大于或等于1。最常见设置此值等于1，只完成对一个信号量的操作。
  - 直接控制信号量信息：int semctl(int semid, int semnum, int cmd, union semun arg);

    - semid：信号量集标识符。
    - semnum：信号量集数组上的下标，表示某一个信号量。
    - arg：union semun类型。

辅助命令

ipcs命令用于报告共享内存、信号量和消息队列信息。

- ipcs -a：列出共享内存、信号量和消息队列信息。
- ipcs -l：列出系统限额。
- ipcs -u：列出当前使用情况。

套接字

- 详见socket交互流程
- 详见网络I/O模型

同步互斥机制

> 待补充

网络I/O模型

在描述这块内容的诸多书籍中，很多都只说笼统的概念，我们将问题具体化，暂时只考虑服务器端的网络I/O情形。我们假定目前的情形是服务器已经在监听用户请求，建立连接后服务器调用read()函数等待读取用户发送过来的数据流，之后将接收到的数据打印出来。

所以服务器端简单是这样的流程：建立连接 -> 监听请求 -> 等待用户数据 -> 打印数据。我们总结网络通信中的等待：

- 建立连接时等待对方的ACK包（TCP）。
- 等待客户端请求（HTTP）。
- 输入等待：服务器用户数据到达内核缓冲区（read函数等待）。
- 输出等待：用户端等待缓冲区有足够空间可以输入（write函数等待）。

另外为了能够解释清楚网络I/O模型，还需要了解一些基础。对服务器而言，打印出用户输入的字符串（printf函数）和从网络中获取数据（read函数）需要单独来看。服务器首先accept用户连接请求后首先调用read函数等待数据，这里的read函数是系统调用，运行于内核态，使用的也是内核地址空间，并且从网络中取得的数据需要先写入到内核缓冲区。当read系统调用获

取到数据后将这些数据再复制到用户地址空间的用户缓冲区中，之后返回到用户态执行printf函数打印字符串。我们需要明确两点：

- read执行在内核态且数据流先读入内核缓冲区；printf运行于用户态，打印的数据会先从内核缓冲区复制到进程的用户缓冲区，之后打印出来。
- printf函数一定是在read函数已经准备好数据之后才能执行，但read函数作为I/O操作通常需要等待而触发阻塞。调用read函数的是服务器进程，一旦被read调用阻塞，整个服务器在获取到用户数据前都不能接受任何其他用户的请求（单进程/线程）。

有了上面的基础，我们就可以介绍下面四种网路I/O模型。

阻塞式

- 阻塞表示一旦调用I/O函数必须等整个I/O完成才返回。正如上面提到的那种情形，当服务器调用了read函数之后，如果不是立即接收到数据，服务器进程会被阻塞，之后一直在等待用户数据到达，用户数据到达后首先会写进内核缓冲区，之后内核缓冲区数据复制到用户进程（服务器进程）缓冲区。完成了上述所有的工作后，才会把执行权限返回给用户（从内核态 -> 用户态）。
- 很显然，阻塞式I/O的效率实在太低，如果用户输入数据迟迟不到的话，整个服务器就会一直被阻塞（单进程/线程）。为了不影响服务器接收其他进程的连接，我们可以考虑多进程模型，这样当服务器建立连接后为连接的用户创建新线程，新线程即使是使用阻塞式I/O也仅仅是这一个线程被阻塞，不会影响服务器等待接收新的连接。
- 多线程模型下，主线程等待用户请求，用户有请求到达时创建新线程。新线程负责具体的工作，即使是因为调用了read函数被阻塞也不会影响服务器。我们还可以进一步优化创建连接池和线程池以减小频繁调用I/O接口的开销。但新问题随之产生，每个新线程或者进程（加入使用对进程模型）都会占用大量系统资源，除此之外过多的线程和进程在调度方面开销也会大很对，所以这种模型并不适合大并发量。

非阻塞I/O

- 阻塞和非阻塞最大的区别在于调用I/O系统调用后，是等整个I/O过程完成再把操作权限返回给用户还是会立即返回。
- 可以使用以下语句将句柄fd设置为非阻塞I/O：fcntl(fd, F_SETFL, O_NONBLOCK);

- 非阻塞I/O在调用后会立即返回，用户进程对返回的返回值判断以区分是否完成了I/O。如果返回大于0表示完成了数据读取，返回值即读取的字节数；返回0表示连接已经正常断开；返回-1表示错误，接下来用户进程会不停地询问kernel是否准备完毕。
- 非阻塞I/O虽然不再会完全阻塞用户进程，但实际上由于用户进程需要不停地询问kernel是否准备完数据，所以整体效率依旧非常低，不适合做并发。

I/O多路复用（事件驱动模型）

前面已经论述了多进程、多进程模型会因为开销巨大和调度困难而导致并不能承受高并发量。但不适用这种模型的话，无论是阻塞还是非阻塞方式都会导致整个服务器停滞。

所以对于大并发量，我们需要一种代理模型可以帮助我们集中去管理所有的socket连接，一旦某个socket数据到达了就执行其对应的用户进程，I/O多路复用就是这么一种模型。Linux下I/O多路复用的系统调用有select，poll和epoll，但从本质上来讲他们都是同步I/O范畴。

1. select
    - 相关接口：

      int select (int maxfd, fd_set *readfds, fd_set* writefds, fd_set *errorfds, struct timeval* timeout);

      FD_ZERO(int fd, fd_set* fds) //清空集合

      FD_SET(int fd, fd_set* fds) //将给定的描述符加入集合

      FD_ISSET(int fd, fd_set* fds) //将给定的描述符从文件中删除

      FD_CLR(int fd, fd_set* fds) //判断指定描述符是否在集合中

    - 参数：
      maxfd：当前最大文件描述符的值+1（≠ MAX_CONN）。

      readfds：指向读文件队列集合（fd_set）的指针。

      writefds：同上，指向读集合的指针。

      writefds：同上，指向错误集合的指针。

timeout：指向timeval结构指针，用于设置超时。

- 其他：

  判断和操作对象为set_fd集合，集合大小为单个进程可打开的最大文件数1024或2048（可重新编译内核修改但不建议）。

2. poll

   - 相关接口：
     int poll(struct pollfd *fds, unsigned int nfds, int timeout);

   - 结构体定义：
     struct pollfd{
     int fd; // 文件描述符
     short events; // 等到的事件
     short revents; // 实际发生的事件
     }

   - 参数：
     fds：指向pollfd结构体数组的指针。

     nfds：pollfd数组当前已被使用的最大下标。

     timeout：等待毫秒数。

   - 其他：

     判断和操作对象是元素为pollfd类型的数组，数组大小自己设定，即为最大连接数。

3. epoll

   - 相关接口：
     int epoll_create(int size); // 创建epoll句柄
     int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event); // 事件注册函数
     int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

   - 结构体定义：
     struct epoll_event{
     __uint32_t events;

```
epoll_data_t data;
};
typedef union epoll_data{
void *ptr;
int fd;
__uint32_t u32;
__uint64_t u64;
}epoll_data_t;
```

- 参数：

  size：用来告诉内核要监听的数目。

  epfd：epoll函数的返回值。

  op：表示动作（EPOLL_CTL_ADD/EPOLL_CTL_FD/EPOLL_CTL_DEL）。

  fd：需要监听的fd。

  events：指向epoll_event的指针，该结构记录监听的事件。

  maxevents：告诉内核events的大小。

  timeout：超时时间（ms为单位，0表示立即返回，-1将不确定）。

4. select、poll和epoll区别

   - 操作方式及效率：

     select是遍历，需要遍历fd_set每一个比特位（= MAX_CONN），O(n)；poll是遍历，但只遍历到pollfd数组当前已使用的最大下标（≠ MAX_CONN），O(n)；epoll是回调，O(1)。

   - 最大连接数：

     select为1024/2048（一个进程打开的文件数是有限制的）；poll无上限；epoll无上限。

   - fd拷贝：

     select每次都需要把fd集合从用户态拷贝到内核态；poll每次都需要把fd集合从用户态拷贝到内核态；epoll调用epoll_ctl时拷贝进内核

并放到事件表中，但用户进程和内核通过mmap映射共享同一块存储，避免了fd从内核赋值到用户空间。

- 其他：

  select每次内核仅仅是通知有消息到了需要处理，具体是哪一个需要遍历所有的描述符才能找到。epoll不仅通知有I/O到来还可通过callback函数具体定位到活跃的socket，实现伪AIO。

异步I/O模型

- 上面三种I/O方式均属于同步I/O。
- 从阻塞式I/O到非阻塞I/O，我们已经做到了调用I/O请求后立即返回，但不停轮询的操作效率又很低，如果能够既像非阻塞I/O能够立即返回又能不一直轮询的话会更符合我们的预期。
- 之所以用户进程会不停轮询就是因为在数据准备完毕后内核不会回调用户进程，只能通过用户进程一次又一次轮询来查询I/O结果。如果内核能够在完成I/O后通过消息告知用户进程来处理已经得到的数据自然是最好的，异步I/O就是这么回事。
- 异步I/O就是当用户进程发起I/O请求后立即返回，直到内核发送一个信号，告知进程I/O已完成，在整个过程中，都没有进程被阻塞。看上去异步I/O和非阻塞I/O的区别在于：判断数据是否准备完毕的任务从用户进程本身被委托给内核来完成。这里所谓的异步只是操作系统提供的一直机制罢了。

1.常用的linux下的命令

ls ps kill start ...

2.大的log文件中，统计异常出现的次数、排序，或者指定输出多少行多少列的内容。(主要考察awk)

3.linux下的调查问题思路：内存、CPU、句柄数、过滤、查找、模拟POST和GET请求等等场景

4.linux系统里，你知道buffer和cache如何区分吗？

1).buffer(缓冲)是为了提高内存和硬盘之间的数据交换的速度而设计的。
2).cache(缓存)
（1）从CPU角度考虑，是为了提高CPU和内存之间的数据交换速度而设计的。

（２）从内存读取和磁盘读取角度考虑，cache可以理解为系统为了更高的读取效率，更多的使用内存来缓存可能被再次访问的数据。

4.描述Linux系统从开机到登陆界面的启动过程

⑴开机BIOS自检，加载硬盘。

⑵读取MBR,MBR引导。

⑶grub引导菜单(Boot Loader)。

⑷加载内核kernel。

⑸启动init进程，依据inittab文件设定运行级别

⑹init进程，执行rc.sysinit文件。

⑺启动内核模块，执行不同级别的脚本程序。

⑻执行/etc/rc.d/rc.local

⑼启动mingetty，进入系统登陆界面。

5.