**Linux高阶**

# 系统进程

## 单任务操作系统

一台计算机一个时刻内只让一个用户访问，该用户一次只能执行一个任务，提交一个作业，一个用户 独占系统全部硬件资源与软件资源, 比较经典的单任务系统有:MS-DOS PC-DOS CP/M等等...

## 多任务操作系统

多个程序共 享计算机硬件

### 进程基础

* PCB

PCB为进程控制块，每个进程都有属于自己的PCB，系统唯一 ，用来记录进程详细信息，例如进程pid等等都保存在PCB中， 如果一个进程结束，PCB也要被释放回收。 进程控制块(记录了详细的进程信息)结构体形式,成员数量340+

• Kernel\_stack/PCB内核栈

• kernel stack (PCB内核栈)进程保存恢复处理器现场

* 调度单位（线程，进程）

调度单位就是寄存器和栈寄存器表明可以访问占用cpu（完成处理或运算）栈：是内核栈保存恢复处理器现场

• 合格的调度单元

• 既可以遵循分时复用原则使用cpu完成任务，又可以通过保存恢复处理器现场，让多进程任务完美衔接，那么这就是一个合格的调度单位(进程，线程)

* 进程生存环境

• 内核空间/层

3G~4G 里面有PCB

• 3G~4G 里面有PCB

• 0~3g/保存者进程的核心工作

• 用户空间

• 命令行参数与环境变量

• 环境变量

由系统拷贝继承过来的一些配置参数。拷贝系统环境变量，进程修改环境变量对系统环境变量不影响。

• 命令行参数

• 命令行参数:存储接收用户的外部传参

• 进程栈

• 进程栈:存储一些程序的临时数据

• 库空间

• 库空间:保存程序的库资源(运行时联入外部加载)静态库 or动态库

• 堆空间

• 堆空间:主动申请的资源空间, new . malloc colloc relloc.

• BSS

• 未初始化的全局变量与静态变量

• DATA

• 存储已初始化的全局变量或静态变量

• TEXT代码段

• 进程主要的运行逻辑

• null

• 哨兵节点

* 内存

• 物理内存

• 内存条

• 虚拟内存

• 储存在硬盘中的

• 虚拟内存地址只是物理内存的指针

• cpu 与Disk硬盘关系

cpu：运算处理能力极强。运算器硬盘：磁盘的访问效率低下。关系：读写（缓存与读取运算数据），硬盘会拖垮cpu效率

• cpu：运算处理能力极强。运算器

• 硬盘：磁盘的访问效率低下。

• 关系：读写（缓存与读取运算数据），硬盘会拖垮cpu效率

• cpu/内存/硬盘

• 为什么要虚拟内存？

1.上层不允许直接访问下层设备(物理设备)加入中间层)虚拟内存与disk硬盘关联1.物理内存咎量受限，其次制作成本高昂，软件开发者程序使用内存也要考虑占用优化等等要素(较为珍贵的资源)2.FILESIZE 20GB 物理内存:8GB#物理内存不足以缓存特大型文件，可以使用虚拟内存(swp 关联disk)做缓存，减轻物理内存压力##进程指向虚拟地址（硬盘），虚拟地址指向物理内存，当进程在物理内存的资源长时间不用那么系统会利用swp交换技术，将资源转移到硬盘中，什么时候用在拷回来（提高内存复用性）

* 多任务操作系统设计核心

可以将cpu设备及资源更好的分配给多个进程使用，提高系统整体处理性能，发挥cpu能力[如何有效的让多进程很好的利用cpu资源]

• cpu的分时复用

• 内核在切换调度进程时要给进程分配时间片（进程佳用特定时长cpu)，使用完时间片会被强制中断该进程的占用，交替给其他进程

• cpu保存恢复处理器现场

cpu用来保存临时数据的寄存器设备有限，为了让多进程很好的使用寄存器设备不引发冲突，在进程切换时通过保存/恢复处理 器现场（保存临时数据及运算过程）的方式多进程共享使用寄存器，每一个进程被中断继续后都能正常向下执行， 一个成熟 的多任务系统，分时复用与保存恢复现场，这两个都是不可或缺的。

• 当进程中断，进程会创建一个保存恢复处理器，当进程再次分配时间片，进程会调用保存恢复处理器，来恢复之前的工作

• 串行执行:单进程执行功能和代码逻辑是串行

• 并发执行:

• 交替运行，占用时间短，交替频率快(每个进程都能执行任务，外部观察感觉同时运行

• 并行执行:

• 为每一个执行单元分配一个处理核心，实现硬件并行

• 效率问题

并发进程效率更高，原因一个进程在一定时间内可能堵塞

* 进程状态转换

进程单位，在执行过程中可能转变为不同的N种状态， 状态之间可以转换，状态的含义也不同

• 就绪态

• 进程准备就绪，等待系统资源，当获取到系统资源(时间片)后就可以切换到运行态。

• 运行态

• 获取了系统资源，进程正在运行，执行特定任务。

• 睡眠态/堵塞态

• 通过sleep()函数或组阻塞函数都可能将进程切换为睡眠态，释放所有系统资源等待（可强制中断）

• 挂起态

• 通过pause函数和挂起信号等可将进程挂起，释放所有系统资源等待(不可强制中断)

• 终止态

• 进程结束，不可逆

• 僵尸进程

• 子先于父亲结束会产生，僵尸进程

• 孤儿进程

• 父亲先于子结束，会产生孤儿进程

* 用户层内核层驱动层

列如printf函数printf是用户级函数，调用printf会发出一个信号转到内核层，在由内核层函数转到驱动层函数，在转到硬件设备

• 用户层到内核层为什么会有系统调用

• 用户层权限最低

操作系统中最低权限,只能访问很小—部分开放内容，不允许访问物理设备

• cpu权限转换

cpu分多级权限，0级权限最强

• 处于0级状态的cpu :操作系统中拥有最高权限，访问任何软件或硬件设备都没问题,没有任何限制

• 零级执行：内核级执行

• 驱动层

• 不安全

• 系统调用

• cpu的权限级别的转换

• 一次系统调用:两次切换用户内核,内核>用户

• 可以完成用户内核切换(权级转换)的事件

• 系统调用

• 一般调用用户函数

• 软件中断

• 时间片用完，系统把软件停止，切换到内核层，让进程就绪

• 异常

• 从用户层切换到内核层检测异常

• 用户/内核

• 用户：系统低权访问

• 内核：系统高权访问

# 进程源语

## linux进程系统

linux下进程都是fork出来的

### Linux系统中进程间具有强亲缘关系

* 1.负责创建子进程

• 2.继承资源

• 3.负责回收

* 子进程从创建执行到进程终止，父进程都要参与

### 系统的进程关系

* 父进程
* 子进程

### init进程

系统开机会检查硬件，开机后第一个进程就是init进程

* init，系统核心继承，系统所有进程的父级，其他进程向上追溯都可以追溯到init

### fork（叉子）

* fork=调用一次创建一个进程，调用者为父进程，被创建者为子进程
* 参数空，返回值pid\_t

• 进程pid，进程身份标识，系统唯一

• 返回子进程pid

* FORK创建子进程的继承问题

3~4G PCB pid 0~3G堆栈/环境变量/代码段等 fork（）

• 用户层代码段里面fork

• KERNEL内核

• CRECT()创建进程（空壳）

• 子进程（未就绪）

拷贝父亲进程PCB部分东西CRECT9()创建的是一个空壳拷贝继承父亲进程主要任务（完全拷贝）

• \_\_CLONG()

• 第一版问题

• EXEC函数，允许开发者为一个进程自定义重载用户空间数据，如果子进程可以自行准备用户空间，那么父进程克隆拷贝用户空间没有任何意义

• 父子调用fork子进程也调用嘛？会不会有问题？

子进程会调用，父进程调用fork和子进程调用fork返回值不一样，子进程调用fork不会创建进程和拷贝进程

* 区分父子工作区问题

代码段(父子进程代码段完全一致，业务逻辑,变量,代码等等)如果我们知道那个是父子进程就可以通过ifelse

• 首先区分父子进程(判断出父子后，通过if,else区分任务）

• 父进程fork返回值cpid，子进程返回值0，如果有错误返回-1

• 177acbda10d3166f07d33dd03f16f66.png

* 子进程执行完任务区代码不要踏出任务区

• 执行完子进程结束，不允许踏出任务区执行其他代码

• 子进程结束后面加exit（0）；结束进程，或者while（1）让进程卡那

* 创建多进程模型（一父多子模型）

• 注意for循环里面判断pid号防止子进程创建进程

• 可以根据flag区分进程

### 两个返回pid函数

* pid\_t pid=getpid(void)

• 进程调用里面调用此函数返回进程自身id

* pid\_t pid=getppid(void)

• #getppid函数可以返回调用进程父进程pid

### VFORK()

* 第二版VFORK()，使用vfork创建的子进程是没有用户层数据的，使用者需要自行准备或exec加载用户层数据给子进程，才能使用

### 第三版FORK()

* 第三版FORK(通过读共享，写复制机制)解决用户数据争议

• 将父进程用户数据映射给子进程，子进程可以读访问父进程所有资源

• 当子进程要对映射数据进行写访问时，父进程检测到子进程[需要一份用户层数据，执行写复制机制,拷贝一份给子进程I

• 写复制,父进程写也会触发

• 父进程在写之前将父进程用户数据拷贝到子进程里

## EXEC进程功能重载

path=加载命令的位置name=程序名占位argv=—堆的执行参数所有的exec函数最后的参数都必须传null，否则异常

# 僵尸进程

## 僵尸进程是内存泄漏的一种

## 僵尸进程产生原因

### 父子进程模型中，子进程先于父进程结束，未对子进程回收处理，父进程必须回收子进程资源（PCB），否则会产生内存泄漏（僵尸进程）

## 僵尸进程的危害

### 僵尸进程有比较大的内存泄漏危害（PCB是庞大的结构体，内部成员众多，许多成员还额外占用内存空间）

### 僵尸进程导致PCB残留，影响新进程的创建

## 父进程可以通过wait或者waitpid进行回收操作(处理僵尸）

### pid\_t zpid=wait(int\* status)

* status :可以从pcb中提取子进程退出特征，便于父进程了解情况,如果不关心则传NULL
* 进程在执行时，被信号杀死(Linux消息机制，杀死进程)，异常退出
* WIFEXITED(status)正常退出返回1，异常退出返回0

• WEXITSTATUS(status)获取退出码

* WIFSIGNALED(status)异常退出返回1

• WTERMSIG(status)返回杀死进程的信号

* wait函数使用时的问题:阻塞!，父进程如果采用阻塞回收方案，导致自身任务被搁置，一直到回收彻底完毕都不能执行

### pid\_t zpid = waitpid(pid\_t pid , int \* status , int opt)

pid\_t zpid = waitpid(pid\_t pid , int \* status , int opt)argv1 = pid:回收方式pid >0:回收指定的一个子进程pid = -1:回收任意子进程pid = 0 :回收与调用进程同组的所有子进程(如果子进程在其他进程组，那么这种方式无法回收) (同组回收)pid < -1:回收指定进程组中的所有子进程(跨组回收)例: pid=-4300(PGID)argv2 =传出子进程的退出信息，用于父进程校验退出原因argv3 opt= WNOHANG(非阻塞关键)RETURN VALIEzpid > 0;回收僵尸成功，并成功僵尸进程pidzpid ==0:子进程还在执行不可回收,waitpid非阻塞返回0zpid = -1,回收失败

* 可以通过waitpid函数进行非阻塞轮询回收
* waitpid支持非堵塞回收，可以在回收过程中穿插父进程任务，拥有更好的灵活性

## 僵尸进程回收流程

## 为什么父进程回收僵尸进程

### 内核KERNEL没有权限回收PCB

### 父进程回收时可进行“验尸”?可以获知子进程是否正常退出,如果被信号杀死可以获取杀死子进程的信号编号

### wait对应回收PCB

# 并发模型（多进程并发）

## 1、相比传统单进程模型，多进程模型可以获取更多时间片

### 多进程完成共有任务

* 多进程模型可以获取更多系统资源，加快任务执行速度，缩短任务完成周期·

### 多进程每个进程业务不同

* 灵活度较高

## 2、多进程模型稳定性强

### 完成共有任务时一个进程崩了，其他进程也能工作

## 3.多进程模型中某些进程因执行逻辑挂起或阻塞会放弃cpu资源，但是其他进程可以获取运行(提高系统资源利用率)

## 并不是所有场合与业务都需要多进程，乱使用反而会增大系统开销

### 最大下行速度20m/s如果单进程就可以满足最大下行速率完成数据接收，不需要多进程

## DEMO多进程拷贝实例

### access（文件，关键子）可以校验文件是否存在，以及文件权限

# IPC

## 进程间通信在内核层完成

### 为什么？

* 进程用户层资源，进程独占，其他进程无法访问修改

• 用户层要频繁的访问与统计修改,所以设计为独占模式,避免多个用户访问冲突

不同进程变量可能相同，会冲突

* 进程用户空间是独占资源，内核空间共享资源

内核空间也是虚拟地址，但是所有内核虚拟地址都指向一个块物理地址，A进程的数据在传输到共享内存中B进程是可以看到的

• 用户对内核层是没有访问权,很的无需担心访问冲突尚题

### 绝大多数进程间通信技术都是利用内核层完成进程通信消息传递的

## 数据传输方向

### 单工传输：确定了传输方向后，任何时刻非读即写，不可变更

### 半双工：不需要确定通信方向， 可以在读写间任意切换，同一时刻非读即写 (可调节单工)

### 双工：同一时刻完成读写，同时进行 (Socket套接字)

## 管道[匿名管道PIPE , 命名管道FIFO]

1、管道是一个在内核内存中维护的缓冲器；2、管道拥有文件的特质：读和写操作。匿名管道没有文件实体，而有名管道有文件实体；3、通过管道传递的数据是有顺序的，读的顺序和写的顺序是一致的；4、管道中的数据传递方向是单向的，一端用于写入，另一端用于读取，半双工（单工：只能单向传输，如遥控器向电视发送信号，但电视不能给遥控器发送信号；双工：双向都能同时传输信号，如两个在打电话，一个说话的时候也不影响对方说话，双方都能同时收到对方的信息；半双工：一段时间内只能一个方向传输，如果这个方向传输完毕另一端也会反向传输过来，如对讲机）；5、从管道读数据是一次性的，数据一旦被读走，数据将会被抛弃，释放空间以便储存新的数据；6、匿名管道只能在具有公共祖先的进程（父与子进程、两个兄弟进程、亲戚之间）使用。

### 匿名管道

首先制作一个简易的单向传输发送信号在双向通信时一定要注意：子进程和父进程的读写顺序是不一样的，因为子进程要先向父进程发送数据后才能收到由父进程发来的数据，所以对子进程来说是先write(),再read()。而父进程是先接收数据，再给子进程发送数据，所以对父进程来说是先read(),再write(),否则会发生阻塞。

* 调用:1.在内核层创建管道2.成功传出管道的访问描述符

• 子主题 1

* 创建

• int pipe[2]; pipe(pipe);

• 函数pipe返回0代表成功，返回-1代表失败

* 匿名管道只能完成亲缘间进程通信(缺陷，不相干的进程无法使用)
* 使用时先创建管道pipe(),再去创建子进程fork)
* 子进程可以调用pipe？不能
* 子进程没有访问管道的描述符，子进程可以利用继承获取访问描述符
* 管道使用完毕一定要销毁管道，将所有指向管道的描述符close，指向管道的fd为0,内核释放管道空间
* 匿名管道的缺点：

匿名管道的缺点： 使用受限，只能完成亲缘间进程通信[父子进程] 匿名管道大多数情况下单工使用（传输受限） 管道默认传输无格式字节流（不便于读取解析与处理）

* 管道缓冲区（4096bytes）采用环形队列,管道是用环形队列实现的,数据从写端流入从读端流出,这样就实现了进程间通信。

### 特性

* 管道中的数据传递方向是单向的，一端用于写入，另一端用于读取，半双工
* 管道拥有文件的特质：读和写操作。匿名管道没有文件实体，而有名管道有文件实体

### 有名管道

* 创建

• 1、函数创建 mkfifo (const char\* name ,int mod)

• 创建管道成功,会生成管道文件(有名管道文件)，在生成缓冲区

• 2、命令创建 mkfifo name

* Fifo\_File

大小size：0类型type：p没有数据存储能力，不可编辑

• 管道文件与管道缓冲区关联，文件有缓冲区有，文件销毁缓冲区也销毁

• 管道缓冲区（4096字节）

• Fifo\_File

* 流程

• 写端 int wfd=open(Fifo\_File,O\_WRONLY)

• 通过wfd找到管道文件，在写到管道缓冲区

• 读端 int rfd=open(Fifo\_File,O\_RDONLY)

• 通过rfd找到管道文件，读管道缓冲区

• 数据并没有再管道文件中存储并中转,而是通过关联直接使·管道缓冲区

• 管道文件磁盘不分扇区

• 为什么？

• 为了去掉匿名管道的缺点（只能在亲缘进程使用），管道文件可以两个不相干的进程访问，

• 删除文件unlink（name文件名）

* 概要

• 有名管道独有的特殊情况

• 如果进程种有多个读序列，那么阻塞只对第一个读序列生效，其他都被设置为非阻塞//为什么有这样特性

• 读序列：多个线程，每个线程里面都要read/

• 多个读序列每个都可以读数据，但只要，一个排队就可以，一个等着，和一帮等着没区别

• 访问管道时，可以根据需求切换原子访问或非原子访问

只有重新开始写，系统才会判断是原子访问还是非原子访问

• 写入量小于管道大小(原子访问)

• 优势：好处就是你的数据，可以保证能写进去，不会被丢掉，原子写，数据不会被分割中断,保证数据的完整性

• 原子访问，写操作，写入的数据量必须小于等于可用量，否则会被内核挂起

• 缺点：(原子使用管道，传输效率低下，某些线程经常因为不满足条件被挂起)

• 非原子访问

• 无法保证数据包完整性，但是效率极高

• 管道缓冲区有空间就写入

• 有名管道:读端未开启,写端阻塞的原因?为什么，没有杀死写端

• 因为读端没开启，写端堵塞在open

• 想要访问操作管道文件， 必须凑齐两种操作权限(读写)，如果只有一种，那么open阻塞等待另外一种

### 数据结构：环形队列

### 概要

* 匿名管道使用时特殊情况：

写端存在未向管道写数据，读端读取管道内数据，管道为空读端读阻塞 写端结束，读端读取完管道剩余的内容，再次读返回0，读取结束读端存在但未读数据，写端持续向管道写数据写满管道后，写端阻塞读端结束，写端尝试向管道写数据，内核向写端发送信号[SIGPIPE]，杀死写端进程

• 写端存在未向管道写数据，读端读取管道内数据，管道为读端读阻塞

• 写端结束，读端读取完管道剩余的内容，再次读返回0，读取结束

• 读端存在但未读数据，写端持续向管道写数据写满管道后，写端阻塞

• 读端结束，写端尝试向管道写数据，内核向写端发送信号[SIGPIPE]，杀死写端进程

## MMAP内存共享映射

### Map\_Files（映射文件）size文件大小：不能为0,修改访问文件数据，没有直接对文件修改，而是采用sync技术同步变更

* 也可以不需要文件，拿一块内存替换也可以，windows可以，linux不知道
* 映射方式

• MAP\_ PRIVATE(私有映射)

• 私有映射又叫拷贝映射，将映射文件的内容全部拷贝到映射内存中，如果映射内存改动数据不会影响到映射文件

• MAP SHARED(共享映射)

• 共享映射，真正的数据共享，将映射文件的内容分享到映射内存，如果某块数据变更，通过sync同步机制，对多个关联实时同步变更，其实也是拷贝

• 1.无论是私有映射还是共享映射都可以将映射文件的内容映射到内存中

### 调用mmap（）函数会帮你创建一个映射内存

### void \* ptr = mmap(NULL , size , PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd , 0)函数，跟malloc差不多，返回的是映射内存地址

* 参数

• 1.映射内存的地址，可以自行分配(malloc)，也可以系统分配，需要系统分配传NULL即可

• 2映射大小(一般是映射文件的大小)，映射大小绝对不能为0，映射会失败

• 3.指定映射内存的权限，如果要同时指定多种权限使用维位或 PROT\_READ)PROT\_WRITE|PROT EXEC|PROT\_NONE

• 4.映射方式，共享映射或者私有映射(MAP SHARED，MAP\_PRIVATE)

• 5.映射文件的描述符

• 6.offset映射偏移量，如果不偏移传O即可

* RETURN VALUE =成功返回映射内存的地址，失败返回MAP FAILED
* 对应的回收内存函数munmap（void\*ptr，int size）释放回收映射内存
* MMAP使用时可能会出现权限问题

• 映射:MAP\_SHARED ,PROT\_READ|PROT\_WRITE如果可以映射成功,那么表示可以修改编辑文件

• open(File , O RDONLY)，如何映射，拥有哪种权限，完全取决于打开文件的权限

### 怎么用mmap实现进程通信

* 可以两个进程采用共享映射同一个文件

### 应用场合：ipc，数据处理，网络应用开发，mmap速度比较快

* 文件拓展,截断的方式ftruncate(fd,size)

## 消息队列

### Posix消息队列数据存储形式为链表，可以通过该链表多进程交换信息，与管道不同，消息队列使用没有太多限制，任何时刻写端都可以打开获取或写入消息

### Posix消息队列其中一个关键的要素是消息优先级，获取消息时，读端优先逐条读取

### POSIX消息队列的创建与回收：（随进程持续，随系统持续）

* 消息队列是具有引用计数的，当所有访问占用消息队列的端全部结束，消息队列引用计数为0，才会被内核销毁释放
* Posix消息队提供了异步同通知能力，当消息队列中有可用消息时第一时间通知读端进程

### POSIX消息队列常用API函数接口：

* mq\_open()，创建一个新的或者打开一个现有的消息队列，并且可以再创建时指定消息队列权限，成功返回消息队列描述符
* mq\_send(，写段通过该函数向消息队列发送一条消息
* mq\_recevie)，该函数从消息队列读取一条消息
* mq\_close(，关闭消息队列描述符，引用计数-1
* mq\_unlink(，根据消息队列名删除消息队列
* 消息队列创建

• 函数使用

• mqd\_t mq\_open(const char \* name , int flag , mode\_t mode , struct mq\_attr \* attr);

• name：消息队列的名称标记，用户打开或创建

• flag：选项，O\_CREAT|O\_EXCL(可以严重消息队列是否存在，结合CREAT使用，如果已存在，函数错误返回）|O\_RDONLY|O\_WRONLY|O\_RDWR|O\_NONBLOCK

• mode：权限掩码，0664，0775

• mq\_attr:可以在创建消息队列时，自定义消息队列属性

• 成功返回指向消息队列的描述符 mqd\_t

• int mq send(mqd t , char \* msg , size t msg len , unsigned int msg prio)，写段通过该函数向消息队列发送一条消息

• mqd\_t:消息队列描述符

• msg：消息内容

• msg\_len：消息长度，必须小于等于属性中的msgsize，否则报错EMSGSIZE

• msg \_prio:消息优先级，可以为不同的消息设置优先级，读取时按优先级从高到低渎取，如果不需要优先级，那么该参数可以传0

• nt mq\_recevie(mqd\_t , char \* buffer, size\_t msg\_len , unsigned int\* msg\_priop），注意后面是个指针

• mqd\_t:消息队列描述符

• msg：消息内容

• msg\_len：消息长度，必须大于等于属性中的msgsize，否则报错EMSGSIZE

• msg\_prio:消息优先级，读取时按优先级从高到低读取，!如果不需要优先级，那么该参数可以传0

• mq\_close(mqd\_t)，关闭消息队列描述符，引用计数-1，关闭消息队列并不会删除消息队列实体

• mq\_unlink(const char \* name)，根据消息队列名删除消息队列实体

### 消息队列属性获取与设置

* mq getattr(mqd t , struct mq attr \* attr)，可以获取一个现有消息队列属性
* mq\_setattr(mqd\_t , const struct mq\_attr \* newattr)，可以设置一个现有消息队列属性

• 消息队列(链表）创建成功后，链表是不可更改的，也就是说对maxmsg和msgsize重新设置大小，不会有任何效果

* struct mq\_attr//消息队列属性类型

• long mq\_flags //消息标记，0默认阻塞消息队列，O\_NONBLOCK非阻塞消息队列

• long mq\_maxmsg //消息队列最大消息数

• long mq\_msgsize //消息大小

• long mq\_curmsg //当前消息数量

# Linux进程间关系

## 父子进程模型

### 列如创建一个程序while（1）gcc去编译成app，./app，原理是父进程终端Fork一个子进程，子进程需要execl重载子进程，./app就是重载进程需要的路径

* 在终端创建的进程都叫终端进程

## 进程组关系

### 进程组构成：由一个组长进程与若干的组员进程构成（一般进程组是组长进程创建的） ps ajx 该命令用于查看进程间关系

pid\_t getpgrp(void) // 调用返回当前进程的组 PGID pid\_t setpgid(pid\_t pid , pid\_t pgid) //设置进程组，将一个进程转移到其他组中也可以创建新组 //使用时两个参数都传递相同的进程pid,为创建组，创建进程组只允许组员进程进行(非组长进程)

* 进程组组长标识：PID == PGID
* 终端进程被创建默认就是组长进程
* 终端进程创建多个子进程（就近原则）将所有子进程归纳到父进程同组(成为组员
* 进程组关系与亲缘关系没有必然联系（进程组构成复杂）
* 进程组的创建与销毁

• 一般由组长进程创建

• 进程组的生命周期较长（直到组中最后一个进程(退出，结束或转移），进程组为空，内核回收该组

* 无论是结束进程组(kill -9 -pgid) 还是转移一个进程都需要足够的用户权限，否则失效 kill -9 -加进程组号，杀死进程组里面所有进程
* 进程组相互之间的转移，需要每个进程组权限

### 一个终端进程被执行之后，会分配一个进程组，这个组长就是终端进程

* pid\_t getpgrp(void)，函数调用返回当前进程的组 PGID
* pid\_t setpgid(pid\_t pid , pid\_t pgid) //设置进程组，将一个进程转移到其他组中也可以创建新组 //使用时两个参数都传递相同的进程pid,为创建组，创建进程组只允许组员进程进行(非组长进程)

### bash和终端进程组关系问题？

* bash终端，创建的终端进程和bash是没有组关系的，有亲缘关系

## 进程会话关系

### 进程组关系与进程会话关系

* bash会话发起人
* pgid 3000 ./父进程终端进程组长进程会话参与
* pgid 4000 子进程组员进程会话参与
* bash关闭会将参与会话的组杀死

### 终端进程的缺陷，因为隶属同一会话，导致终端进程被终端限制，终端结束，终端进程以组为单位被杀死

### 如何让进程脱离控制终端？

* 进程可以创建新组，也可以采用会话的方式
* 直属的终端进程与bash深度绑定，无法脱离，但是其子进程可以通过若干方式脱离终端

### 系统会话关系构成：由一个会话发起者与若干个会话参与者构成

* 会话发起者如果结束，会以组为单位结束会话参与者(Bash)
* 会话发起者唯一标识：PID == PGID == SID
* 所有终端下执行的程序或进程都属于终端进程，受限于会话

### 函数

* getsid(pid\_t pid); //成功，返回当前进程的会话id , 参数为进程pid
* void setsid(void); //调用该函数帮助调用进程创建一个会话//setsid 可以帮助一个组员进程创建组并且创建新会话 (组长进程不可用)

# 孤儿进程

## 概念

### 父进程先于子进程嗝屁，子进程失去约束（失控）

## 孤儿进程危害

### 1、废弃的孤儿进程影响新进程的创建

### 孤儿进程的危害是有弹性的，可能忽略不计，也可能很严重

## 父进程终止，会有托管进程对父进程的子进程托管

### 托管进程只负责回收

* 托管进程：UI进程可视化进程

## 孤儿进程预防，检测与处理

### 预防：让父进程足够健壮，代码逻辑检查，不要出现逻辑运行异常，有很好的容错处理能力，尽量避免系统杀死父进程。

### 检测与处理

* 子进程定期检测，检测父进程是否存活

• kill(3000,0)第一个参数是pid，第二个参数传0，表示试探进程是否存活

* 2.创建第三方的检测处理进程，完成周期检测与孤儿处理

• 第三方进程保存，父子进程pid，由第三方进程去kill试探父进程的存活

## 精灵进程/守护进程（孤儿进程的一种，开发者自行创建）

### 普通进程（随着使用者或者功能持续，使用完毕立即终止），进程生命周期较短

### 精灵进程（生命周期较长，随着操作系统持续，开机自启，关机结束）

### 精灵进程守护（守护操作系统，或者某个软件的稳定运行)，精灵进程执行任务并不是持续执行的，而是条件运行或定时运行

### 所有精灵进程都是后台服务进程，不允许干涉前台

### demo:定时每间隔3s向指定的日志log文件中写入当前的系统时间，守护进程后台执行,可以使用shell对森实现开机启动

### 创建七个步骤

* 1.创建父子进程，父进程结束
* 2.子进程成立新的会话，脱离现有控制终端
* 3.关闭无用的文件描述符，指的是标准输入输出，close（STDIN\_FILENO），close（STDOUT\_FILENO）
* 4.修改进程的工作目录，修改为当前主机的根目录chdir（“/”）

• 程序跑起来默认的工作目录，是程序的所在文件

* 5.修改进程umask文件权限掩码，不能修改系统的umask
* 6.守护进程的核心任务（定时执行，条件触发）
* 7.守护进程退出处理（关机时触发，清理释放资源）

# SIGNAL信号机制

信号是Linux经典的消息机制， 主要的作用是对进程进行干预（杀死进程、挂起进程、继续进程） , 操作系统可以利用信号结 束违规进程， 普通进程相互之间可以利用信号完成终止与挂起 , 除了信号基本功能外， 1.信号绑定任务（信号触发） ， 2.利 用信号机制完成进程间通信。

## Linux下利用Signal信号机制也可以实现类似qt的绑定触发效果

### #Signal信号机制与QT下的信号槽是两种技术

## Kill命令

### 只能向你有权限干预的进程发送信号才有用，否则发送完事，那边不执行

### 1-31 , Unix经典信号 (软件开发工程师使用)

* 34-64 , 自定义信号/实时信号 (驱动开发工程常用)

• 32,33 , 隐藏预留给NPTL线程库使用

## 无论是任何的操作系统

### Linux windows macOs Unix前台进程永远只有一个，后台进程有N个

### 要看焦点在哪一个软件上，其他的都被置于后台，后台进程用户不可操作，但是一样可以执行(获取时间片)

* 被置于后台不一定就是挂起，可能也在运行也分配时间片，比如录屏软件

### 终端组合按键产生信号（只对前台进程有效）

* 1).ctl+c (SIGINT|2) 结束进程
* 2) ctl+\ (SIGQUIT|3) 退出进程(转储核心，dump Core)
* 3) ctl+z (SIGTSTP|20) 挂起进程

• 通过jobs命令查看挂起的作业编号

• fg 作业编号（唤醒到前台继续执行）

• bg 作业编号 (唤醒到后台继续执行)

## 函数

#include <signal.h>kill(pid\_t pid , int signo); //向任意进程发送任意信号raise(int signo); //向自身调用进程发送任意信号abort(void); //向自身进程发送固定的SIGABRT信号sigqueue(pid\_t pid , int signo , union sigval); //向任意进程发送任意信号并且携带自定义数据包

### kill(pid\_t pid , int signo); //向任意进程发送任意信号

### sigqueue(pid\_t pid , int signo , union sigval); //向任意进程发送任意信号并且携带自定义数据包

* 对硬件异常访问，会产生段错误，11号信号就是段错误，给程序发送11信号，程序会被段错误杀死

## 硬件触发异常

### 对内存进行非法使用（段错误：非法操作内存） (SIGSEGV|11)

### CPU运算异常 (浮点数例外) (SIGFPE|8)

* 出现这种错误，表示代码逻辑出现错误

### 堆栈异常，调度异常 (总线错误) (SIGBUS|7)

## 软条件触发信号

### 匿名管道，读端终止，写端向管道写数据(软条件已触发)，内核向写端发送信号 (SIGPIPE|13)

### 定时器开始定时， 定时器到时(软条件触发)，内核向定时进程发送信号 (SIGALRM|14)

### unsigned int alarm(unsigned int seconds);//参数为定时的秒数，返回未定时够的描述//定时器到时，内核向定时进程发送SIGALRM信号，终止该进程

## 信号的三大行为与五种默认处理动作

选择默认行为，会处置进程，不是杀死就是挂起选择忽略行为，会让信号失效

### 默认行为SIG\_DFL

* TERM:终止进程
* CORE:终止进程并生成CORE文件，运行核心，运行过程，把程序的运行过程保存在CORE文件里
* IGN:不处置进程，什么也不捕捉，信号默认动作是IGN,那么不处置进程，64个信号就一个默认状态是IGN
* STOP:挂起进程
* CONT:继续被挂起进程
* 这些都是默认行为的处理动作

### SIG\_IGN忽略行为

* 没有处理动作不会影响进程

### SIG\_ACTION捕捉行为

* (捕捉行为可以实现，功能邦定触发)
* 处理动作用户自定义实现

• 捕捉函数（用户自定义实现）

## 信号传递过程

### 列如ctl+c(2)走到线路规程

* 向内核发送通知，在由内核向前台进程的pcb发送2号信号，

• pcb里面有未决信号集，和屏蔽字，未决信号集对应的信号编号位为0才可以让信号通过

• 信号通过未决的对应位置，对应位置置为1，类似通过屏蔽字，注意如果信号通过屏蔽字那么对应的未决集重新置为0

• 未决集是内核翻转01

• 如果信号在未决和屏蔽字之间卡着，意味着信号被屏蔽

• 屏蔽字用户翻转01

### 如果要查看进程当前信号的处理情况，查看未决信号集是最准确

* sigpending(sigset\_t \* pset)#调用后将未决信号集传出到pset

## 信号屏蔽设置

#include sigset\_t newset , oldset//信号集类型 sigemptyset(sigset\_t \* set); //初始化信号集，将所有信号位初始化为0 sigfillset(sigset\_t \* set); //初始化信号集，将所有信号位初始化为1 sigaddset(sigset\_t \* , int signo); //对信号集中某一个信号位，置1 sigdelset(sigset\_t \* , int signo); //对信号集中某一个信号位，置0 int reval = sigismember(sigset\_t \* , int signo); //查看某信号集，特定信号位是0还是1，并且返回 sigprocmask(SIG\_SETMASK , &newset ,&oldset ) //SIG\_SETMASK:直接覆盖 //SIG\_BLOCK:位或 //SIG\_UNBLOCK:取反求与

### sigset\_t类型 newset #信号集类

### sigemptyset(&newset)函数初始化信号集

* sigaddset(&newset , SIGINT)#将对应信号位，置为1

• sigprocmask(how ,&newset , &oldset )函数，将自定义的信号集集去覆盖屏蔽集，参数1表示怎么覆盖，参数2自定义的参数集，参数3表示将原有屏蔽字保留

SiG\_SETMASK(要盖)SIG\_BLOCK(位或,置1)SIG\_UNBLOCK(取反求与)

* sigdelset(&newset , SIGINT)#将对应位置0

### int value = sigismember(sigset t \* set , int signo)可以判断—个信号集位设章情况，返回一个信号的位码，需要for循环

## 改信号行为

### struct sigaction newact结构体

* act.handler=SIG\_DFL|SIG\_IGN|SIGACTION#存放信号行为的成员

• act.sa\_flags=0;与handler绑定，设置为0即可

• act.sa\_mask;临时屏蔽字，初始化使用sigemptyset

### sigaction(SIGINT，&newact , &oldact)#传入自定义的新信号行为，传出进程原有信号行为，不传出传NULL

## 信号捕捉设定，一样是通过修改信号行为的方式

### 捕捉函数是系统内核帮你调用

### struct sigaction newact结构体

* act.sa\_handler=sig\_eat#存捕捉函数地址，act.sa\_handler是void(\*sa\_handler)(int)函数指针

• act.sa\_flags=0;与handler绑定，设置为0即可

• act.sa\_mask;临时屏蔽字，初始化使用sigemptyset

• 不同的信号绑定了相同的捕捉函数(捕捉函数中访问共享资源)会导致冲突问题可以设置l临时屏蔽字，再处理某个信号时临时屏蔽其他信号，处理完解除

### sigaction(SIGINT，&newact , &oldact)#传入自定义的新信号行为，传出进程原有信号行为，不传出传NULL

## 经典信号不支持排队，但传递时有其特殊性

### 一个信号未处理完，内核会临时设置对应的信号屏蔽，直到信号处理完毕，解除屏蔽

* 自定义信号/实时信号34~64支持排队

• 排队序列有次数限制

## 信号的忽略、屏蔽与捕捉都可以让信号失效，但各有不同

### 1.信号屏蔽:信号并没有被处理，延缓处理

### ⒉.信号忽略:没有信号处理行为,速度更快

### 3.信号捕捉:有自定义信号处理行为，信号递达必须调用捕捉函数

### 如果所有的信号都可以被这三种方式处理掉，操作系统还安全么?

9、19直接为内核服务，拥有最高的权限和职能，只要发出必然递达，无法被捕捉，忽略和屏蔽

### 自定义实现sleep（）

需要定时处理、挂起、唤醒

* alarm()#定时到时后，内核向定时进程发送SIGALRM信号，杀死定时进程，必须通过忽略或捕捉的方式处理信号
* pause0#调用即可立即挂起当前进程，察觉一个信号即可唤醒（SIGALRM），需要信号可以被处理的，所以alarm需要用捕捉的方式处理信号
* pause只能察觉递达并且有处理行为的信号，那么只有信号捕捉符合要求
* pause察觉信号不是特定的,任意信号只要符合要求都可以唤醒进程
* 定时器属于内核对象,不受进程限制. cpu计时
* 信号引起的时序竞态问题:

• 可以通过原子方法避免这种问题

• int sigsuspend(sigset\_t \* mask); //挂起+临时解除屏蔽方法//sigsuspend调用时会将当前进程挂起，并使用mask信号集临时替换进程屏蔽字， 函数执行完毕后， 对进程屏蔽字复位还原

## 捕捉函数的调用过程：

### 过程

* 1、进程执行逻辑语句串行执行
* 2、信号触发
* 3、等待用户层与内核层切换事件：系统调用、软件中断、错误异常
* 4、完成调用、恢复中断、处理异常
* 5、完成任务尝试切回用户层（检测是否有未递送的信号）
* 6、检测到有待处置信号(进行信号处理),如果信号行为是捕捉，携权限进入用户层完成捕捉函数的调用
* 7、捕捉执行完毕SIG RETURN返回内核空间
* 8、内核层切换回用户层
* 9、从中断的位置继续运行main

### 使用信号摘捉技术时，不允许使用不可重入函数

### 可重入函数;函数内部使用独立资源，没有使用任何全局或静态数据，绝对不会产生冲突．信号开发可以放心使用

开发时可以通过man 7 signal通过手册查看信号技术可以用的可重入函数

### 不可重入函数:函数接口内部使用了全局或静态资源，如果捕捉函数与主函数都使用该函数，引发冲突异常，信号捕捉技术禁用不可重入函数

### 限制

(1)信号处理不是实时处理(延时处理)(2)内核调用捕捉函数。使用进程资源，不会额外申请(cpu , mem)(3)捕捉函数与主函数不能同时执行(4)主的数先执行，过程中触发了信号,内核调用捕护函数,捕捉函数先处理完后，返回主函数继续执行(5)如果捕捉函数与主函数共享使用全剧资源很容易引发冲突和异常(信号捕捉无法调试)

* 捕捉函数与主函数不能同时执行
* 不会因为捕捉函数而分配更多的时间片，占用的还是进程本身资源

### 

# 基于信号的进程间通信

## sigqueue(pid\_t pid , int signo , union sigval); //向任意进程发送任意信号并且携带自定义数据包

### union sigval#数据联合体，包含int siva\_int#整型数据，void\*sival\_ptr 指针数据

## 信号选择？SIGUSR1、SIGUSR2#给开发者提供的信号

### 两信号的默认动作都是trem，#要设计好信号处理

## #旧的sa\_handler无法接收额外数据，信号通信无法使用该捕捉函数

### 新的接口捕捉函数void sig\_ child(int , siginfo\_t \* info , void \* arg)

* 查看或使用通信数据：info->si\_int(接收整型数据),info->si\_ptr（接收指针数据）

## DEMO:父子进程利用信号技术完成交叉报数(进程通信)

# Pthread线程基础

## 线程是操作系统的调度单位，与进程一样可以获取系统资源完成特定事件与任务，比进程更轻量，占用资源更少 ， 大多数情况下，线程的速度更快。

### 线程独占内存资源，独占cpu，独占时间片

## 进程是操作系统的调度单位，操作系统为进程分配系统资源,让进程完成特定任务，进程是最小的分配资源单位（内存资源）

### 进程创建时，系统对进程进行内存划分，虚拟内存，物理内存

## 线程

### 线程是进程一部分，它是工作于进程内部的，线程寄生在进程，占用的内存是进程的，没有分配内存，所以线程是最小的调度单位

* 线程无法独立存活

### 线程置于进程中，多线程开发模型必须保证进程健康，否则进程结束，所有线程终止

### 线程调度

cpu核心处理器里面有个调度器，它可以识别出系统的调度单元

### 用户级线程:内核无法识别线程,以进程为单位分配系统资源的

* 好处不用过多的和内核去交互

用户级线程是在用户空间定义与使用的无需内核干预的，便于使用，速度较快

• NPTL为用户级线程

### 内核级线程:内核支持线程技术，可识别钱程，会将所有进程与线程都作为调度单位分配k,每个进全都能得到时间片（可以得到更多cpu，加快任务完成速度）

* 内核级线程调度与使用需要内核干预,例如配置调度或者上下文切换,由较为庞大的调度开销，速度较慢

### 混合型线程:线程是在用户空间定义的，调度开销较小，但是会为每一个线程创建一个内核对象,可以被操作系统识别，并分配系统资源(时间片)

## 进程的退化：进程可以退化成为线程

### 进程里面创建线程：资源被共享，无法独占

### 如果进程中出现了线程，那么变为多线程模型，由1个主控线程与n个普通线程构成

### 初始时,所有的普通线程都是由主控线程创建的

## #操作系统会为每一个识别出的调度单位分配一个key,到时向其分配系统资源(时间片)

## 线程间共享资源与非共享资源

### 共享资源

* 全局资源共享
* 共享堆
* 文件描述符
* 进程工作目录共享
* PID GID 共享， 用户id ，用户组id共享
* 线程间信号行为共享

• 一个线程设置对信号的捕捉，其他线程也会对这个信号捕捉，不同线程屏蔽字独立，可以设置其线程屏蔽字防止其他线程对信号捕捉

### 线程非共享资源

* 用户线程栈非共享(8M)虚拟地址，线程内核栈非共享
* 线程屏蔽字非共享
* 线程信息(TCB)tid非共享
* 线程调度优先级
* ERRNO(全局变量)非共享

• 线程有自己的错误处理方式，无需使用全局ERRNO

## 线程并发与进程并发的利弊

### 进程并发模型：

* 优点：可以获取更多系统资源(更多使用cpu) , 多进程稳定性更好
* 缺点：系统开销较大， 切换上下文，调度缓慢

### 线程并发模型:

* 优点： 开销小， 更轻量
* 缺点： 稳定性比较差(线程崩溃导致进程终止)， 相较于多进程模型需要注意的开发细节更多（死锁，互斥等等）

# 线程源语

## 使用NPTL线程库中的方法，编译时必须链接线程库gcc \*.c -lpthread -o app

## 线程的函数

### 线程创建 pthread\_create

#include <pthread.h>pthread\_t tid //线程id类型int err = pthread\_create(pthread\_t \* tid , pthread\_attr\_t \* attr , void \* (\*t\_job)(void \*) , void \* arg );//参数介绍://tid = 线程创建成功，内核将线程id传出到tid变量中//attr = 线程属性， 传NULL表示使用默认线程属性(可以满足绝大多数开发需求)//t\_job = 线程工作地址， 开发者定义线程任务并实现， 线程创建后开始执行//arg = 线程工作传参//RETURN VALUE//调用成功返回0 ， 失败返回错误号(errno)char \* 错误原因 = strerror(错误号); //线程开发，通过该函数获取错误原因

* 成功返回0，错误返回错误码

### int err = pthread\_exit(void \* exitcode);//线程退出，参数为线程退出码（整型），其他线程可以对退出码回收校验

### 线程tid获取 pthread\_self

* pthread\_t tid = pthread\_self(void);

//无论哪个线程调用该函数，返回调用线程的tid(线程id) 在普通线程内调用self()与主线程创建传出的tid值相等，但是否等价？ 普通线程内使用self()显示自身tid,可以保证当前线程存活并有效 主线程内传出tid， 无法确认线程是否存活（可能已结束）

• 通过pthread\_self在线程内部获取tid,有效性更强

### 线程回收 pthread\_join

* int err = pthread\_join(pthread\_t tid , void \*\* reval);

//如果一个线程是回收态线程，那么线程退出后需要join回收，否则会引起内存泄漏(僵线程) //arg1 = 指定需要回收线程的tid //arg2 = 回收成功传出线程的退出码或返回值 , 如果不需要接收，传NULL即可 //除主控线程外，其他线程间也可以通过tid,进行回收

• //阻塞回收函数:

### 线程取消,发出取消事件：pthread\_cancel(pthread\_t tid)

* int err = pthread\_cancel(pthread\_t tid);

//参数为要取消的线程tid,可以通过该函数取消结束其他线程, 线程取消没有限制，只要可以得到对方tid就可以进行取消操作

• //线程取消要求被取消的线程需要系统调用，否则无法结束void pthread\_testcancel(void); //该函数只产生一次系统调用(空调用)

### 变更线程退出状态

* pthread\_detach(pthread\_t tid)

• 如果需要获取分析线程退出码，则需要回收处理使用join，否则将线程设置为分离态即可

### 线程间信号传递 pthread\_kill

* int err = pthread\_kill(pthread\_t , int signo);

• //向指定线程发送任意信号，必须对信号进行捕捉和屏蔽，否则信号默认会杀死整个进程

### 线程的错误处理

* 判断线程函数的返回值，如果返回值err大于0 , 表示该函数可能调用失败 ， 返回值中存储了对应的错误号， 通过strerror函数，获取错误信息。

• printf("call failed error : %s\n",strerror(err));

## ps -eLf查看线程命令，ps -Lf pid查看指定进程的线程

### 线程tid与lwp不是一个概念（便于管理），lwp为线程编号，tid为线程id

## 线程退出状态

### 回收态

DFL\_EXITSTAT:回收 (PTHREAD\_JOINABLE)

* 回收态:线程被创建后，线程退出时，创建者需要对其进行回收操作，否则可能产生僵线程，造成内存泄漏

• 回收态属于线程的默认退出状态，如果没有特殊设置，线程创建时统一都是回收太线程

### 变更线程退出状态

* pthread\_detach(pthread\_t tid)

• 如果需要获取分析线程退出码，则需要回收处理使用join，否则将线程设置为分离态即可

### 分离态

NEW\_EXITSTAT:分离态(PTHREAD\_DETACHED)

* 分离态;线程退出时内核自动回收线程资源，包括获取退出吗(但是内核会将退出码丢弃)

• 多线程模型如果不想进行繁琐的回收处理，可以将所有线程设置为分离太，结束后内核自行回收资源

### 退出状态互斥

* 一个线程必须只有一种退出状态（回收 or 分离）

• 1.不允许对已经设置分离的线程进行回收操作(pthread join) , pthread join回收必然失败

• 2.不允许对一个已经处于回收阶段的线程进行分离设置，分离设置会失败

pthread\_join和pthread detach谁更先执行，慢的必然失败

## 线程四种退出方式：

### RETURN

* 主控线程使用RETURN,会导致整个进程退出(影响所有线程)
* 普通线程使用RETURN,线程退出(与进程无关)

### EXIT

* 主控线程使用EXIT,整个进程退出
* 普通线程使用EXIT,整个进程退出

### PTHREAD\_EXIT

* 主控线程使用，线程退出(与进程无关)
* 普通线程使用，线程退出（与进程无关）

### PTHREAD\_CANCEL

### 通过tid,取消结束某一个线程

# 线程属性

在使用多线程开发时默认属性可以满足绝大多数需求 ， 除 非模型中对线程有特殊需要（修改属性） ， 修改线程属性是有风险的 ， 可能使多线程模型更不稳定，谨慎使用。

## 属性

### 线程优先级

* 线程权重指针

### 线程栈警界缓冲区

* 防溢出防越界

• 线程会在线程栈上方设置一个线程缓冲区，如果线程栈溢出往缓冲区里面写数据那么会产生段错误

### 线程状态

* 线程创建前，决定线程退出状态(分离或回收)

### 线程栈大小

* 线程栈地址

• 突破默认线程数量限制，提高创建线程数量

## 使用流程

所有PTHREAD开头的函数编译时一定要加入库链接 -lpthread 某些线程函数编译时即使不链接库也不会失败，但是无法正常使用。

### 线程属性类型

* pthread\_attr\_t attr //线程属性类型
* pthread\_attr\_init(pthread\_attr\_t \* attr); //初始化线程属性[内核申请空间]
* pthread\_attr\_destroy(pthread\_attr\_t \* attr) //销毁释放线程属性

### 可以修改属性中的退出状态，决定要创建的线程是分离线程还是回收线程

* pthread\_attr\_setdetachstate(pthread\_attr\_t \* setattr , int detachstate);

//SET对线程属性进行设置，将其中的状态选项设置为分离或回收分离态关键字:PTHREAD\_CREATE\_DETACHED 回收态关键字:PTHREAD\_CREATE\_JOINABLE pthread\_attr\_getdetachstate(pthread\_attr\_t \* getattr , int \* detachstate); //获取属性中的退出状态，传出到detachstate变量中

### 何时使用Detach函数设置分离 ？ 何时使用属性设置分离？

* 多线程模型中，对分离线程批创建，修改属性的做法是很好的选择。
* 多线程模型中，退出状态是混合型， 采用detach函数延迟设置

### 可以修改属性中的线程栈信息，提高线程创建数量

修改线程栈信息虽然可以提高线程量，但是可能会出现溢出问题，降低线程的稳定性，

* pthread\_attr\_setstack(pthread\_attr\_t \* attr , void \* stackaddr , size\_t stacksize);//利用该函数对属性中的栈进行设
* pthread\_attr\_getstack(pthread\_attr\_t \* attr , void \*\* stackaddr , size\_t \* stacksize) //获取线程属性中的栈信息

# 线程安全(PTHREAD SAFE)

## 多线程共享使用全局资源冲突问题（锁技术）

### 互斥锁

互斥锁解决方案(某个线程使用资源时，其他线程被迫挂起)

* 1.上锁操作:锁未被占用，那么请求成功，可以访问使用全局资源(独占使用)
* 2.上锁操作:锁已被占用，当前请求的线程会被内核挂起等待，直到占用着释放，可能才会被唤醒
* 占用线程释放后,资源如何分发,线程如何唤醒获取资源:

• 1.占用者释放，所有任都唤醒，争抢资源(惊群问题)有问题的方法

大量的线程都会被调度，唤醒或挂起，但是资源有限，很多线程的调度没有任何意义(争夺不到资源)，反而增大了系统开销。

• 指定唤醒

• kernel会有唤醒标记给等待队列

• 就近原则:某个线程方释放资源马上再次申请,大多数情况下，该线程可以继续占用资源

• #唤醒顺序,以队列排列为准

• 唤醒标记在占用线程解除占用时发放

* 互斥锁类型以及相关的API:

• pthread\_mutex\_t //互斥锁类型

• pthread\_mutex\_init(pthread\_mutex\_t \* lock , pthread\_mutexattr\_t \* attr);初始化

• //arg1=互斥锁地址 arg2=互斥锁属性地址 ， arg2=NULL表示使用默认互斥锁属性

• pthread\_mutex\_destroy(pthread\_mutex\_t \* lock);

• //使用destroy对互斥锁进行回收释放

• pthread\_mutex\_lock(pthread\_mutex\_t \* lock);

• //上锁，创建临界区(临界区起点)

• pthread\_mutex\_unlock(pthread\_mutex\_t \* lock);

• //解锁，（临界区终点）

* 请求锁方式：阻塞请求锁（访问锁被占用线程挂起），非阻塞请求锁，定时阻塞请求锁（线程在一定时间内等待锁，到时间返回干别的事）
* demo：两个线程对—个全局资源各加5000次，(结果异常)，通过互斥锁避免冲突

### 读写锁

* 读写锁允许多个线程读访问全局资源， 读共享，写独占 , 读写互斥
* 读写锁类型以及相关的API:

• pthread\_rwlock\_t //读写锁类型

• pthread\_rwlock\_init(pthread\_rwlock\_t \* lock , pthread\_rwlockattr\_t \* attr);

• //arg1=读写锁地址 arg2=读写锁属性 , arg2 = NULL 使用默认读写锁属性

• pthread\_rwlock\_destroy(pthread\_rwlock\_t \* lock);

• //使用destroy对读写锁进行回收释放

• pthread\_rwlock\_rdlock(pthread\_rwlock\_t \* lock);//申请读锁

• pthread\_rwlock\_wrlock(pthread\_rwlock\_t \* lock); //申请写锁

• pthread\_rwlock\_unlock(pthread\_rwlock\_t \* lock); //解除读写锁

• 读写锁可以提高全局资源的利用率某些需要读共享数据的设计场景，是绝佳的选择

* 使用注意事项：

• 某线程占用写锁,其他线程访问写锁会被挂起等待

• 某线程占用读锁，其他线程请求读锁可以成功，共享访问，但是读锁有数量限制，如果耗尽，再请求读锁会被挂起

* demo :创建8个线程，3个写线程(用写锁)，5个读线程，使用读写锁交替访问使用全局资源

## 进程互斥锁

默认以情况下使用PTHREAD\_MUTEX为线程互斥锁，解决多线程全局资源访问互斥，可以通过修改互斥锁属性的方式，将其变更为进程互斥锁，供多进程实现多进程全局资源访问互斥

### 多进程共享资源?(与线程不同,并不是全局资源)

* 例如系统文件亦或者多进程共享交互的数据，在使用这些数据资源时一样要保证访问互斥，避免冲突问题

### 如何使用进程锁？

* 定义互斥锁属性结构体

pthread\_mutexattr\_t互斥锁属性类型

* 初始化互斥锁属性

pthread\_mutexattr\_init()初始化互斥锁属性,出事后后为默认属性(线程互斥)

* 根据使用需求设置互斥锁属性中的项

可以将属性中的线程互斥改为进程互斥

* 初始化互斥锁

pthread\_mutex\_init()用户自定义互斥锁属性初始化互斥锁(进程互斥锁)

* PTHREAD\_MUTEX\_T互斥锁

### 实例:父子进程对共享资源code进行加操作，通过进程互斥锁进行互斥访问(避免访问冲突)

## 文件读写锁

### 文件也属于操作系统共享资源，如果多进程或多线程一起访问操/作文件，可能产生冲突，那么可以利用文件读写锁避兔冲突

* 文件读写锁与读写锁基本相同，只不过用于文件

### 文件属性

* POS读写指针位置、文件权限信息、文件访问方式、文件锁(flock)

### 文件锁属性

* 文件锁设置选项

• l-\_type:F\_RDLCK、F\_WRLOCK、F\_UNLCK

* 上锁位置

• 绝对位置

• l\_whence : SEEK\_SET ,SEEK\_CUR ，SEEK\_END

• 相对位置

• 一个锁在绝对位置60，相对位置锁30，那么上锁位置是90

• l\_start :相对偏移量(如果不使用传0)

* 锁长度

• l\_len:传入的锁多长数据（传0锁整个文件）

* 占用进程pid

• l\_pid:保存占用进程的pid

### fcntl（fd，cmd，&lock）（修改文件属性）

* 设置关键字CMD：F\_SETLK、F\_GETLK、F\_SETLKW

• F\_SETLK:设置锁(非阻塞设置，,如果失败立即返回)

• F\_GETLK:设置锁(阻塞版，如果失败阻塞等待)

• F\_SETLKW:传出文件当前锁属性,可以用于判断文件锁使用情况

### 根据自己的上锁需求设置定义设置文件锁结构体

l type = F\_WRLCKl\_whence = SEEK\_SETstart = O;Tlen = 0;

## 旋转锁

### 一个线程占锁，另一个线程持续申请锁（不会挂起）

* 旋转锁开销较大，以全速运行的方式持续申请，直到获取为止
* 阻塞互斥锁，占用者解锁后需要一段时间(唤醒，标记分发等等)才能被再次占用，延迟占用，但是旋转锁并没有这个问题，占用者解除占用后立即使用，用锁频率更高

## 排他锁上锁方式：

### 阻塞申请锁资源,如果获取未成功一直等待

### 非阻塞申请锁资源,不会阻塞等待锁资源，获取失败立即返回

### 定时阻塞申请锁资源，设置指定固定时长阻塞等待

## 死锁问题(BUG)，多线程用锁不当导致的程序异常(永久挂起)

### 死锁就是—种相互等待的情况(—直持续下去)

### 多线程资源有限，而且多线程还要共享使用多个资源,那么特别容易产生死锁

### 1.大多数时间，每个进程都用通过自己的锁占用自己的资源，每个进程某个时刻需要申请对方的资源,完成任务

### 产生死锁的四个必要条件：

* 1、互斥条件（占用资源后，其他线程无法使用需要等待）
* 4、请求与保持，保持自身资源，请求他人资源
* 2、不可剥夺条件，资源被占用后，无法强制剥夺其资源
* 3、循环等待条件，多进程或者多线程产生资源等待环路，期待其他人手中的资源

# 哲学家就餐问题

# 线程控制

## 条件变量技术，可以通过条件变量实现线程的控制调度（让多线程按照开发者想要的模式有条理的执行)(线程的挂起与唤醒)

## 基于条件变量实现生产者消费者模式(线程间利用该模型完成任务分发与传递),

## #控制线程是否执行，如果不执行则将该线程挂起到特定变量中，如果需要其执行则唤醒该线程，通过这种方式实现线程控制的目的

## 条件变量的数据以什么为依据确定？

### 线程是否能执行，是取决于执行条件的，条件变量的数量取决于条件数量

## 两类线程的工作条件是具有关联性的，A类执行时，B类不满足条件，通常A类执行结束，B类满足工作条件

## 条件变量相关的类型及函数接口:

### pthread\_cond\_t cd //条件变量类型

### pthread\_cond\_init(pthread\_cond\_t \* cd , pthread\_condattr\_t \* attr);//条件变量初始化，第二个参数条件遍历属性设置可以传空

### pthread\_cond\_destroy(pthread\_cond\_t \* cd);//条件变量销毁

### pthread\_cond\_wait(pthread\_cond\_t \* cd , pthread\_mutex\_t \* lock);

pthread\_cond\_wait()两次调用： 第一次调用: 挂起当前线程并解锁互斥锁 第二次调用：被唤醒后上锁互斥锁

### pthread\_cond\_signal(pthread\_cond\_t \* cd);//唤醒一个被挂起在某条件变量上的线程

### pthread\_cond\_broadcast(pthread\_cont\_t \* cd);//唤醒所有被挂起在某条件变量上的线程

## 消费者模型

pthread\_cond\_t not\_full; //生产者条件变量pthread\_cond\_t not\_empty; //消费者条件变量pthread\_mutex\_t lock; //互斥锁任务包：typedef struct{void \* (\*tjob)(void \*);void \* arg;}taks\_t; //任务类型任务容器（环形队列）typedef struct{int front ;//环形队列头索引int rear; //环形队列尾索引int max; //任务容器最大数int cur;//记录当前任务数task\_t \* taskqueue; //任务队列}queue\_t;模块分析：int pthread\_add\_task(queue\_t \* ,taks\_t); //生产任务的模块，哪个线程调用该模块，此刻即是生产者线程void \* pthread\_customer\_job(void \* arg); //消费者线程默认工作，执行后等待接收任务并处理 , 参数为任务队列queue\_t \* pthread\_container\_create(int max);//任务队列初始化，参数为队列大小int pthread\_container\_destroy(queue\_t \* );//任务容器销毁释放

# Socket套接字基础

## 1.Socket文件描述符与传统描述的区别

### 相同点：在Linux，系统将大多数设备的访问与使用都统一化，可以以访问文件的形式访问多种系统设备(管道，消息队列，EPOLL, OCKET)

### 不同点：

## 2.Socket的组成部分

### IP:主机A应用程序想要和主机B应用程序通信，应用程序的服务器将主机B的ip交给主机A，主机A接着利用路由器最短路径寻址，找到主机B

### PORT端口号：区分主机上的应用型程序，数据包转发与接收是通过端口号

## 3.Socket网络信息结构体;struct sockaddr\_in addr

### 可以为socket赋予关键的网络信息，列如ip和PORT端口

### add.sin\_family指定IP协议类型

* AF\_INET(ipv4(32))
* AF\_INET6(ipv6（128）)

### addr.sin\_addr.s\_addr存储ip地址（大端序的存储）

### addr.sin\_port存储端口信息（大端序的存储）

## 大小端转换函数#include <arpa/inet.h>

//h=小端 n=大端 l=ip s=port

### 大端序ip = htonl(小端序ip)

### 大端序端口 = htons(小端序port)

### 小端序ip = ntohl(大端序ip)

### 小端序port = ntohs(大端序端口)

### inet\_ntop(AF\_INET,void \* ptr, char \* iparray , size\_t ipsize ) //大端序ip转字符串ip

### inet\_pton(AF\_INET,"192.168.11.145",void \* ptr); //字符串ip转大端ip

## SOCKET() 创建socketfd

#include <sys/socket.h>int sockfd = socket(AF\_INET , SOCK\_STREAM , 0);argv1 = 指定IP协议版本 , ipv4 or ipv6 AF\_INET(ipv4) AF\_INET6(ipv6)argv2 = 指定传输层协议， SOCK\_STREAM(流式协议) SOCK\_DGRAM(报文)argv3 = protocal = 0 ,表示默认写STREAM默认协议为TCP , DGRAM默认协议为UDPRETURN VALUE:成功返回Sockfd , 失败返回-1并且ERRNO被设置

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

## BIND 设置socket网络信息

### 一个sockfd被创建后，会自带默认网络信息 ip=本机任意ip port = 随机端口,如果用户需要定制sockfd中的网络信息，就需要使用bind进行设置

### 经典的C/S架构中， 通常Server需要绑定设置 ， Client是否需要绑定看需求

## LISTEN 监听Server端网络事件

### 1.使用Listen函数，可以帮助应用监听网络连接状态转换以及一些网络IO事件（读|写|异常事件)，只在tcp模型中使用

## ACCEPT 服务端等待建立连接(阻塞函数)

int clientfd = accept(int serverfd , struct sockaddr \* clientaddr , socklen\_t size);argv1 = 阻塞在特定sockfd等待连接argv2 = 连接成功传出对方的网络信息结构体argv3 = 传入预存储网络信息结构体大小，传出实际网络信息结构体大小RETURN VALUE:成功返回客户端的sockfd , 失败返回-1,并且设置ERRNO

### 1.调用accept函数后，进程则阻塞等待连接)，如果有网络揣请求连接， accept进行响应，完成TCP三次握手过程，socket状态变更为数据传输态，而后两响通信

### 2.accept可以完成tcp连接的建立，可以传出请求链接端的网络信息，还可以传出请求连接端的sockfd

## .CONNET,主动请求TCP链接(主动端执行)

### 1.通过connect函数可以向服务端请求tcp连接，首先获悉服务饬ip和端口，而后客户端通过该网络信息向服务端发送网络连接请求即tcp连接湾求

## CLOSE(SOCKET\_FD)，触发断开连接(四次挥手)

### close(serverfdi) , close(clientfd)#Close将文件描述符引用计数-1，但想要释放描述符引用计数必须清0

## .网络应用开发，常用的读写方法

### TCP:read(),write(),send(),recv()

### UDP:recvfrom(),sendto()

## 同步异步区别

同步和异步最大的区别就在于：一个需要等待，一个不需要等待。比如广播，就是一个异步例子。发起者不关心接收者的状态。不需要等待接收者的返回信息电话，就是一个同步例子。发起者需要等待接收者，接通电话后，通信才开始。需要等待接收者的返回信息

### 同步:发送一个请求,等待返回,然后再发送下一个请求

### 异步:发送一个请求,不等待返回,随时可以再发送下一个请求

# 服务器

## 两个pc间通信流程

### pc发送数据到路由器

* 路由器根据应用的ip和端口号寻找服务器

• 第一次寻找按照最短路由寻址，找到服务器将服务器缓存到路由表中

• TTL(跳转):数据包从源头发出，经过若干次路由跳转，255次后还未到目的地，会被最后一个接收路由直接丢弃

• 寻找本地路由表

* 数据到达服务器，服务器根据缓存好的用户的ip地址转发到对应的路由器

• 对应的路由器，会根据缓存好的路由表寻找对应主机

## 软件服务器

### 软件服务器根据软件功能设计研发，做数据支持与技术支持

### 根据软件需求,指定传输协议(tcp+udp)

### 没计较好的协议6消息类型)

## WEB服务器B/S模型

### 通用http https协议（超文本传输协议）

## 服务器的特性与任务

### 服务器特性：

* 服务器需要较强的处理能力(并发性)
* 服务器需要较强的稳定性
* 可靠性强/安全性
* 可管理性强， 可扩展性

### 服务器任务：

* 服务器有较强的存储能力（便于共享软硬件资源），持久化用户数据信息
* 服务器的数据中转(网络穿透)

## 通过网络地址连接服务器，地址是否是固定的？

### 如果软件中固定填写了服务端网络信息，服务器信息如果变更，导致软件要进行更新，否则与服务器失去联系

### 可以采用dns域名解析服务

域名与ip具有绑定关系，通过域名可以随时找到对应的公网ip，无论ip如何变更，域名都可以找到其最新的ip地址进行连接访问

# socket函数二次包装

## 如果recv读到0表示，客户端退出

# 服务器模型

## 单进程服务器

### ACCEPT,RECV冲突问题

#阻塞等待建立TCP连接，完成与客户端的三次握手\*当前阻塞等待连接，这时有任发送业务请求,不能及时读取#阻塞读取擦户端请求;而后处理并将结果反馈给客户端\*当前正在阻塞读取客户端数据，此时来了新的连接请求，无法及时建立连接

* 将recv读取数据变更为非阻塞读取，是否可以避免冲突(不能根本问题，稍微缓解阻塞冲突)

1.accept阻塞,recv一样无法及时读取客户端请求数据2.recv非阻塞，客户端没有请求数据，recv立即返回，但是如果客户端有请求,recv还是要读取处理的，这是有新连接，一样无法及时连接

## 一对多进程处理模型

### 采用多进程模型，为每一个客户端创建一个子进程，子进程负责处理客户端业务请求，父进程负责与客户端建立连接，子进程的数量随客户端数量变化，客户端结束对应的子进程也销毁释放

### 多进程模型必须回收僵尸，但是如何设计回收?

* 解决方案：父进程有两个线程（主控线程（accept），普通线程（wait））：普通线程：捕捉SIGCHLD信号，捕捉成功完成回收操作（普通线程捕捉，主线程不处理）

首先wait回收僵尸进程原理：子进程结束会发送一个SIGCHLD信号给父进程，父进程wait才会使用

• 1、主线程设置屏蔽SIGCHLD

• 2、创建普通线程（继承屏蔽字）

• 3、普通线程完成捕捉设定（被动回收方案）

• 4、普通线程解除屏蔽

• 信号只是触发回收条件，一次信号尽可能回收多个进程（避免漏回收）

### 多进程模型，稳定性较强，某个服务器进程崩溃不会影响主体，调度开销与系统开销较大，父进程将任务转交给子进程，子进程负责处理实际请求业务

## 多线程处理模型

工作模式:主线程或进程负责建立连接与任务分发，线程或子进程完成业务处理

### 多线程模型，内部逻辑结构与多进程模型相似，调度开销与系统开销较小，稳定性较差，可能还要考虑线程安全问题（死锁等)

# 多路IO转接模型/多路IO复用

为网络应用提供支持的技术，监听socket，服务器客户端都可以用完成多路网络io事件监听监听，就绪，处理就绪

## 多路io复用技术，同时监听多路网络io事件的技术，网络io就是socket 文件描述符，可以单进程就实现服务端一对多效果

总结:该技术可以同时监听多个socket在网络交互中，因读写产生的网络事件

### 网络io sock fd

* 读事件

#其他网络io向你发消息，会触发读事件

• 普通读数据

• 优先级带外数据

* 写事件

#向其他网络io发消息

• 普通读数据

• 优先级带外数据

* 异常事件

#网络io读写异常，触发异常事件。比如一端关闭，另一端发数据，就会触发异常事件

* 挂起事件

## 传统单进程模型冲突：

### accept（阻塞）

监听server fd ->读事件监听，客户端tcp连接请求数据（syn）读事件监听听到了触发“读事件就绪”给server fd，server fd给accept阻塞唤醒，完成三次握手(tcp连接成功)(处理就读事件/处理就绪事件)

### recv

### #每一个监听的网络事件都要正确处理

### 阻塞监听的冲突

* #一个模型如果可以同时监听多个socket的网络事件，那么就可以解决冲突，让连接与读取都可以很好的进行，最终实现单进程一对多效果

## select模型

accept and recv无需阻塞直接处理因为select模块帮助监听

### fd\_set set类型#监听集合类型，最大可监1024socket

* 对应监听的socket描述符:监听位，监听位为1则监听，否则不监听

### 需要建立fd\_set类型集合，将服务器socket放入里面，socket监听位置为1，将建立的集合传入select模块，由select模块轮询监听这些socket

* 客户端发送连接请求，select模块监听到，返回事件就绪的数量，然后”查找辨别就绪“看是服务端socket还是客户端socket

• 服务端socket，执行accept（连接），将连接到客户端socket加入监听集合和客户端socket数组

• 客户端socket，执行recv（读请求）处理请求，send（反馈结果），客户端异常：删除客户端socket，监听集合socket也删掉，socket数组中也要删除，close（socket）

### selcet函数传入监听集合，一旦监听到事件传出就绪集合

* 就绪集合中，就绪的保留为1，未就绪的置为0
* 比较就绪

• 如果是客户端就绪那么就去socket数组中查找

### 缺点

1、内核中对select模型可监视的fd数量限制为1024，如果要修改这个数字就必须对内核重新编译；2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大。

* 1.select的监听数量限制为1024，(无法满足高监听需求)
* 2.没有将传入传出分离，使用者要自行设置(set,oset)
* 3.select并不会返回就绪的socket，只返回就绪的数量，使用者需要自行遍历查找谁就绪
* 4.select采用轮询模式监听事件，轮询问题(随着轮询数量的增加，IO处理性能呈线性下降)，这也是为什么select监听的特别少的原因
* 5. select对监听集合设置监听是批处理的，无法对不同的socket设置不同的监听事件，灵活性不足
* 6.select监听是要进行多轮的，在多轮监听过程中可能会出现一些重复性开销，这些开销没有任何意义，是一种浪费

• select底层原理

• 1、监听集合（用户层），select模块将集合copy到内核层。

• 2、内核层监听集合的每个sockfd都挂载（mount）到io设备等待队列（轮询监听io事件）

• 3、如果监听到了事件，就根据就绪情况，修改监听集合，变为就绪集合

• 4、在将修改的监听集合，由内核层copy到用户层

### 优点

* 1.实现较为简单，单进程服务端使用select就能实现基本的一对多处理效果
* 2.多个操作系统和平台都有对select的支持和实现，跨平台能力较强
* 3.如果业务场景中对监听的事件精度要求高，select可以支持微秒级别的定时阻塞
* 4.如果需求环境为局域网，select应该可以满足需求

### select相关函数

* fd\_set set; //监听集合类型 ，1024的监听限制源于fd\_set
* FD\_ZERO(fd\_set \* set); //初始化监听集合， 将所有的监听项初始化为0
* FD\_SET(int sockfd , fd\_set \* set); //将某个监听集合中，特定的socket对应位置1 ， 该函数用于设置socket监听
* FD\_CLR(int sockfd , fd\_set \* set); //将某个监听集合中 ， 特定的socket对应位置0 ， 该函数用于取消socket监听
* FD\_ISSET(int sockfd , fd\_set \* set); //判断某个监听集合中，某个socket位是0还是1并直接返回， 如果该函数结合就绪集合使用
* select开启轮询监听socket事件

### int select(int maxfd + 1 , fd set \* read set , fd set \* write set , fd set \*err set , struct timeval timeout)

* maxfd =表示select监听的数量，值得注意的是，select监听是以文件描述符表为依据的，所以监听数量选择maxfd + 1
* read set =将监听集合传参到该位置，监听所有socket的读事件
* write set =将监听集合传参到该位置，监听所有socket的写事件\*如果无需监听某种事件，参数位置传NULL即可
* err set = 将监听集合传参到该位置，监听所有socket的异常事件
* timeout = 等待超时选项，设置select模型的工作模式，阻塞模式(NULL)，非阻塞模式，定时阻塞模式
* select调用成功，监听到就绪,返回就绪的socket数量(readycode)select，调用失败，返回-1，并且errno被设置，可以通过perror进行错误处理，获取错误信息

### selecti调用传入的监听集合为传入传出参数,就绪后内核会修改该集合，所以此集合不能重复使用，使用者需要自行分离传入和传出参数

## poll模型

与select很相像

### \* Poll模型采用结构体数组作为监听集合，结构体类型为pollfd，数组长度可以任意设置(监听socket数量不受限)

### struct pollfd listen array[1024]#监听数组定义

* (poll的监听集合可以存储sock fd，无需定义整型client array保存sock了)
* struct pollfd node;poll模型结构体成员

• node.fd =用于保存要监听的socket fd,如果保存了sock表示监听该sock,如果存储-1表示不监听

• node.events =用于设置监听的I0事件，较为常用POLLIN(读事件)POLLOUT(写事件)POLL ERR(异常事件)，还有其他的事件可选….

• node.revents =当监听的sock\_fd就绪后，内核将就绪事件传出到revents，用户可以通过revents判断sock就绪

### poll模型·api接口

* int poll(struct pollfd \* listen array , int nfds , int timeout)

• pram

• listen array =监听数组的首地址

• nfds = 监听最大数，一般与监听数组长度相同

• timeout =等待超时选项，设置poll模型的工作模式，阻塞模式(-1)，非阻塞模式(0)，定时阻塞模式(>0)

• Return

• poll调用成功，监听到就绪，返回就绪的socket数量(readycode)

• poll调用失败，返回-1，并且errno被设置，可以通过perror进行错误处理，获取错误信息

### PoLL模型的优缺点

* 优点

• 1.poll模型用自定义长度数组作为监听集合，没有了例如select的1024限制

• 2.poll监听事件设置较为出色，可以对不同的sock设置不同的监听，可监听的事件丰富

• 3.如果需求环境为局域网，poll应该可以满足需求

• 4.传入传出分离，events设置监听，revents判断就绪

* 缺点

• 1.定时阻塞只支持毫秒级或秒级

• ⒉.某些特定的平台支持poll，兼容比较差，移植困难

• 3.poll并不会返回就绪的socket，只返回就绪的数量，使用者需要自行遍历查找谁就绪

• 4.poll采用轮询模式监听事件，轮询问题(随着轮询数量的增加，IO处理性能呈线性下降)

• 5.poll监听是要进行多轮的，在多轮监听过程中可能会出现一些重复性开销，这些开销没有任何意义，是一种浪费

## Epoll模型

### \*Epoll监听能力非常强，并没有监听数量限制，理论上使用epoll监听主机上可创建的最大socket数量，也不会有额外的开销和负担

### Epoll采用红黑树作为监听集合

### struct epoll event env;#监听节点类型

* env.data.fa=存储要监听的sockfd描述符
* env.events=存储要监听的IO事件，EPOLLIN(读事件)， EPOLLOUT(写事件)

### epoll函数api

* int epfd = epoll create(int max);#创建监听集合(红黑树)，参数为监听最大数，返回值为指向红黑树的描述符epfd
* epoll ctl(int epfd , int option , int index , struct epoll event\* node) #它通过该函数访问红黑树，进行对监听集合的增删改，改只能改事件

mod修改，只允许修改节点监听事件，无法修改监听sock fd

• epfd =监听树的描述符

• option =/操作树的方式，EPOLL CTL ADD(添加)EPOLL CTL DEL(删除) EPOLL CTL MOD(修改)

• node = 如果是删除操作，该参数传NULL，否则传递节点地址

• index =需要sock fd描述符，添加树节点时通过socket作为索引，便于后续的查找与使用

* int epoll\_wait(int epfd , struct epoll\_event \* ready\_array , int maxnfds，int timeout) #epoll阻塞监听函数

• epfd =监听树的描述符

• #就绪队列数组struct epoll\_event ready\_array[ ]

当epol监听到就绪，不仅会返回就绪数量readycode ，还会将就绪的socket节点传出到就绪数组中，用户遍历处理即可，无需像select或poll一样，自行查找就绪了

• epoll\_create( int maxsize)struct epoll\_event ready\_array[size]epoll\_wait(int maxnfds),这个参数值使用时需要相等

### EpolI模型优缺点:

* 优点

• 1.epolI监听socket采用的并不是轮询模式，更像是异步回调机制，所以epoll并不会受轮询问题影响，理论epoll模型可以监听系统最大socket数量○

• 2.epoll设置事件监听很灵活，可以对不同的socket设置不同的事件监听，而且可监听的事件丰富

• 3.epol监听到就绪后，不仅会返回就绪数量，还会传出就绪队列，队列中包含了就绪的若干个socket,用户依次处理即可，使用较为方便（无需用户查找和判断就绪)

• 4.epoll的监听树是在内核层创建的(使用时无需拷贝整个监听集合)，用户在设置监听时需要将监听的节点拷贝到内核空间，挂载到监听树上，每一个监听节点只会拷贝一次并且挂载一次(可以有效避免无意义的重复拷贝问题)

• 5.epoll并没有使用IO设备等待队列(轮询)完成socket事件监听，内部自定义实现了epoll监听队列，体积更小，监听效率更高

• 相比SELECT与POLL EPOLL业务处理更及时，1.不需要轮询监听(延迟)2不需要查找就绪(延迟) ,大大提高业务处理的实时性

• EPOLL兼容性比较好，LINUX与UNIX平台兼容出色

• epoll自带线程安全，多线程使用epoll不用担心监听树冲突问题

* 缺点

• 1.epoll监听能力相比select和poll出色，但是处理能力三种模型是相同的，如果是单进程模型，不允许一个客户端任务长时间占用服务器

• ⒉.如果对epoll监听树操作过于频繁(查询)，到时系统开销增大，内存消耗增加

### epoll实现

* epoll自己在定义了一个监听队列（网络设备，反馈网络事件）
* 内核层创建监听集合监听树，树中每个节点都有一个回调函数，当节点有socket，就与网络设备绑定（将节点中的回调注册到网络设备）
* 某一刻产生就绪网络设备调用ep\_callback（）将就绪事件传出到就绪链表epitem事件类型
* rdllist (就绪链表(双向链表))，epoll模型检测该就绪链表，如果为NULL则睡眠等待，否则返回就绪，内核层链表节点copy到用户层就绪队列（用户自己设定）
* mmap进行拷贝可以节省拷贝开销(减少拷贝次数)

### epoll两种触发模式

* epoll水平触发模式（EPOLLLT）

可以对不同的socket设置监听模式，默认为水平触发，当第一轮监听,监听到sockfd就绪后，使用者必须将就绪事件处理完毕(建立连接或读取全部数据)，否则无法使用epoll\_wait进行第二轮监听，未处理完就绪事件前调用epoll\_wait,epoll\_wait函数立即返回，返回值为未处理的数量,水平触发模式下，内核会持续检测事件处理进度，并且持续向用户发送处理通知水平触发模式，内核高度参与，轮询检查事件处理情况，系统开销较大，但是内核监督事件处理，可以保证不会漏处理事件，保证事件处理的实时性

• 水平触发模式的优势：可以协助督促使用者及时处理完所有sock事件， 保证就绪事件处理的及时性有效性

• 水平触发模式弊端：系统开销较大， epoll不停轮询检测事件处理情况，并反复向上层发送处理通知

* epoll边缘触发模式（EPOLLET）

可将sockfd监听模式设置为边缘触发模式， 当第一轮监听，监听到sockfd就绪后，eoll只向使用者发送一次处理通知， 也仅有 一次， 后续使用者是否处理就绪与epoll无关 ， 随时都可以调用epoll\_wait进行新一轮监听

• 边缘触发模式的优势：epoll开销较小 ， 因为监听到就绪只返回一次就绪，发送一次通知，不会持续检测就绪处理进度

• 边缘触发模式的弊端：因为边缘模式下的sockfd就绪，epoll只反馈一次通知， 需要使用者及时将事件或事件数据处理掉 ， 否则事件处理不及时，可能会出现事件数据丢失问题

• 数据到来，触发一次，读走需要的数据后缓冲区还有剩下的数据，只有下次数据到来才会触发，如果用户自己不处理完，数据会一直放在那里,直到下一次有数据到来时才有触发

### Epoll边缘触发模式+非阻塞读取网络IO

* 边缘触发模式监听socket ,就绪后只会发送一次处理通知，使用者要利用这次通知将所有数据读取完毕,通常采用循环读取，但是如果采用的是阻塞读取，数据读取完毕后进程陷入阻案态，任务无法推进，所以要利用非阻骞读取数据，数据读取完毕立即返回

# EPOLL+线程池模型（高并发低活跃）

## 保有大量的在线有效用户，要求较高的并发量，用户与服务器交互特别频繁密集，要求服务器拥有极强的处理能力和响应速度

## 高并发低活跃：大量的在线有效用户，要求高并发，用户活跃度较低，用户使用频率与访问频率较低，服务器不会有特别大的处理压力和负载大量的在线有效用户，要求高并发，用户活跃度较低，用户使用频率与访问频率较低，服务器不会有特别大的处理压力和负载

## 为什么采用epoll

### 服务端会产生大量的网络链接，采用EPOLL模型完成监听工作,epoll监听能力较强

## 线程池概述：

线程池是经典的多线程容器，若干的处理线程被预创建到线程池中待用，相比于传统的多线程使用模式，线程池加入了线程状态检测，线程管理机制，可以更好的控制与使用线程，线程池中包含任务传递模式（生产者消费者模式)，可以更好的在多钱程间传递任务，提高任务传递及任务处理的可控性和逻辑性

### 线程池为多线程容器，线程便于管理和调度线程，提高了线程的可用性和重用性，合理使用线程池会减小一些不必要的系统开销

### 线程池减小开销?

* 传统的多线程使用，需要则创建线程，使用完毕则销毁，如奥创建销毁过于频繁，导致系统开销增大
* 线程池模式，在线程池中预创建一批线程，这些线程可以重复完成任务，提高了线程的重用性，避免频繁创建与销毁的开销

### ⒉.相比于基本的多线程使用，线程池可以更好更及时的处理任务?

* 线程池会预创建一部分线程待用，如果需要随时都可以调用，快速响应执行任务
* 普通多线程是来一个事件创建一个线程，用完线程就销毁·，相比于线程池来说，多了创建线程的工作

### 3.线程池便于线程调度与管理

* 线程池包含阈值概念，可以通过阈值记录线程池中线程的状态，数量，使用情况等等详细信息，便于使用者观察管理线程池
* 根据阈值动态调整线程,完成线每的扩容和缩减
* 线程池阈值

Thread max线程最大值Thread\_Shutdown线程池开关True or FalseThread min 线程最小值Thread exit缩减量Thread alive线程存活量Thread busy繁忙线程量Thread ldel 闲置线程量

## 线程池重要特征和组件:

### 线程预创建：根据任务处理情况，灵活的扩容缩减线程池中的线程

### 线程预创建：预先在线程池中准备一部分线程，任务递达时立即处理该任务（提高任务处理的及时性)

### 线程管理(阈值)：\*记录线程状态，线程数量等等关健数据，便于管理多线程

### 线程池开关：Shutdown True(开启)Shutdown False(关闭)

### 线程任务传递模式：任务是如何产生的?如何如何传递?任务如何执行?

### 线程池重用性：\*线程池不能与单一任务高度绑定，要灵活设计，便于重用

### 线程可用性：处理线程是可以执行多次任务的