**进程和线程**

进程是资源分配的单位；线程是处理器调度的基本单位。

进程是程序的一次执行；线程可以理解为进程中的一个程序段。

进程是独立的，有自己独立的地址空间；线程是属于进程的，同一进程中的线程共享进程的地址空间。所以如果一个线程奔溃，会影响同进程中的其他线程。

线程因为共享同一进程的资源，执行时一般都要进行同步。

两者均可并发执行

进程接口

fork

exec

wait

waitpid

abort

exit

孤儿进程

**一个进程退出，但是它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。**

**孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。**

僵尸进程

**如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。**

**危害：其子进程号就会一直被占用，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。**

**解决：1.通过信号的方法，子进程结束时，父进程会收到SIGCHILD信号（默认忽略）。如果父进程通过 signal 接收并处理这个信号，就可以处理子进程的退出。**

**2. fork 两次。父进程通过fork 产生一个子进程，在这个子进程中再调用 fork 产生一个次子进程后退出，所有这个次子进程就成了孤儿进程，被 init 进程收养。**

线程接口

pthread\_create

pthread\_join

pthread\_cancel

pthread\_exit

进程同步方式

**管道**

int pipe(int fds[2]);

pipe 返回两个文件描述符， fd[1] 写，fd[0] 读。一般用在通过 fork 创建的两个进程之间通信，因为子进程会复制父进程的文件描述符。

特点：单向流动，按先入先出原则；使用read 和 write 函数实现数据传输；一般用在父子进程间。

**命名管道**

一种特殊类型的文件。

命令行创建：mkfifo filename

函数创建：int mkfifo(const char \*filename, mode\_t mode);

用open 打开FIFO文件，可以设置读、写和阻塞、非阻塞组合方式。

特定：可以在不相关的进程间通信；不依赖与进程而存在；使用read和write函数实现数据传输。

信号

一种比较复杂的通信方式，一般用于通知接收进程某个时间已经发生。

特点：每次只能传递一个信号值。

IPC**（信号量、消息队列、共享内存）**

信号量主要用来保护共享资源，使得资源在一个时刻只能被一个进程或现场拥有。

信号量：利用一个计数器，如果信号量为正，则进程可以使用资源，信号量的值减1。如果信号量的值为0，进程等待，直到信号量的值大于0。

当进程不再使用共享资源，信号量的值加 1 。

System V：（使用复杂，一般用于进程同步）

semget

semctl

Posix：（使用简单）

有名信号量：（信号量的值保存在文件中，一般用于进程间）

sem\_open 创建或打开一个有名信号量

sem\_close 关闭有名信号量

sem\_unlink 删除信号量（只有信号量为0时才能删除）

无名信号量：（保存在内存，一般用于线程间）

sem\_init 初始化一个信号量

sem\_destory() 销毁一个信号量

sem\_wait 申请资源，等待直到信号量的值大于0。阻塞。

sem\_post 释放资源，信号量值加1

sem\_trywait 非阻塞

特点：一般用于进程或线程同步。

消息队列

int msgget(key\_t key, int msgflg)；

msgctl 控制消息队列

msgsnd 将数据放到消息队列

msgrcv 从消息队列取数据

特点：可以在不相关的进程间通信；提供读写函数，不需要提供同步方法。

共享内存

允许多个进程共享一个给定的存储区。可以用信号量同步对共享内存的访问。

shmget 创建一个共享内存段，并获得标识

shmctl

shmat 将创建的共享内存段连接到进程的地址空间中

shmdt 从进程的地址空间分离共享内存段

特点：可以在不相关的进程间通信；速度最快；需要其他方式来同步。

套接字

可以用于不同进程和不同主机之间通信。

线程同步方式

**互斥锁**：在访问共享内存之前，对互斥量进行加锁，访问完成后释放。互斥量加锁之后，其他线程如果再要对互斥量加锁，线程就会阻塞直到当前线程释放互斥量。

pthread\_mutex\_init

pthread\_mutex\_lock

pthread\_mutex\_unlock

pthread\_mutex\_destroy

如果一个线程对一个互斥量加锁两次，就会进入死锁。

**条件变量**

**信号量**

线程与信号

每个线程都有信号屏蔽字，但是信号处理是进程中所有线程共享的。因此每个线程可能要重新设置信号处理方式。

pthread\_sigmask

sigwait 等待信号

pthread\_kill 发生信号

常用并发网络模型

① PPC/TPC 模型

PPC 是为它开了一个进程，而 TPC 开了一个线程。但是这两种方式是有代价的，连接多了之后，多进程会占用系统的资源，而多线程会占用CPU的资源。同时多进程和多线程的创建和销毁、切换都需要很多开销，而且进程和线程的控制比较复杂。

② select 模型

先构造一张文件描述符的列表，然后调用函数，等待直到这些描述符中的一个准备好时才返回。

select 函数：

设置读、写或者异常条件的文件描述符集合；设置select 函数的等待时间（永久等待，不等待，等待指定秒或微秒）

返回已经准备好的文件描述符总的数量。返回-1，出错；返回0，没有文件描述符准备好（超过等待时间）

其他函数：FD\_ZERO 所以位置零，FD\_SET 将指定文件描述符位打开，FD\_CLR 清除指定位，FD\_ISSET 测试指定位文件描述符是否打开。

1. 最大并发数限制。因为一个进程所打开的 FD（文件描述符）是有限制的，由 FD\_SETSIZE 设置，默认值是 1024/2048 ，因此 Select 模型的最大并发数就被相应限制了。

2. 效率问题。 select 采用轮询处理，每次调用都会扫描全部的 FD 集合。

3. 内存拷贝问题。每次 select 调用，都会把fd集合从用户态拷贝到内核态。所以效率也会比较低。

③ epoll 模型

epoll 基于事件的回调机制。首先对文件描述符的事件进行注册，注册的时候就会把文件描述符拷贝到内核中。一旦某个事件准备好了，就会执行回调函数，把准备好的事件加到一个双向链表中。当epoll\_wait调用时，只需要观察这个双向链表里有没有数据即可。

工作模式：水平触发LT（缺省）、边缘触发ET（高速）

水平触发(LT)：如果不做任何操作，内核依旧会不断的通知进程文件描述符准备就绪。

边缘触发(ET): 只有当状态发生变化的时候，内核才会通知进程文件描述符准备就绪。之后如果不在发生文件描述符状态变化，内核就不会再通知进程文件描述符已准备就绪。

1. Epoll 没有最大并发连接的限制。上限是最大可以打开文件的数目，这个数字一般远大于 2048。
2. 效率提升。 Epoll 最大的优点就在于它只需要遍历就绪链表 。（如果活跃的连接一直很多，效率可能会降低）
3. 内存拷贝。Epoll 的FD集合采用了共享内存的方式 。注册的时候就把所有的fd拷贝进内核，保证了每个fd在整个过程中只会拷贝一次。

int epoll\_create (int size);

返回一个epoll 文件描述符。申请一个内核空间，用来存放 fd 上是否发生以及发生了什么事件。size 指定能关注的最大 fd 数（只有内存有关）。

int epoll\_ctl (int epfd, int op, int fd, struct epoll\_event \*event );

控制某个 Epoll 文件描述符上的事件：注册、修改、删除。

int epoll\_wait (int epfd,struct epoll\_event \* events,int maxevents,int timeout);

等待 I/O 事件的发生：返回发生事件的数量。

**gdb调试**

gcc -g -o test main.c -g 调试模式

gdb test 进入调试

list 显示源码

break 设置断点

run 执行程序

step 单步（进入子函数）

next 单步（不进入子函数）

cont 跳到下一个断点或程序结尾

print 显示变量或表达式值

finish 运行到函数结束

quit 退出gdb

**调试正在运行的程序**

gdb命令进入调试模式，用 **attach pid** 附加到指定pid的进程，进程会中断。

多进程调试

默认情况下，在调试多进程程序时GDB只会调试主进程。只需要设置follow-fork-mode(默认值：parent)和detach-on-fork（默认值：on）即可。

|  |  |  |
| --- | --- | --- |
| follow-fork-mode | detach-on-fork | 说明 |
| parent | on | 只调试主进程（GDB默认） |
| child | on | 只调试子进程 |
| parent | off | 同时调试两个进程，gdb跟主进程，子进程block在fork位置 |
| child | off | 同时调试两个进程，gdb跟子进程，主进程block在fork位置 |
| 设置方法 | set follow-fork-mode [parent|child] | |
| set detach-on-fork [on|off] | |
| 查看进程 | info inferiors （带 \* 为当前调试进程） | |
| 切换进程 | inferior id | |

设置断点之后，用 cont 命令跳到第一个断点处。id 为inferior 的编号 Num。

多线程调试

|  |  |
| --- | --- |
| 查看线程 | info threads （带 \* 为当前调试进程） |
| 切换线程 | thread id |

id 为线程thread 的编号 Num。

**makefile**

objects=main.o list.o    #指定变量 objects

show: $(objects)

gcc -o list $(objects)

main.o: main.c list.h

gcc -c main.c

list.o: list.c list.h

gcc -c list.c

-I：指定头文件的目录

-L：指定库文件的目录

-l：指定的动态库文件

**指令**

查看文件、目录

ls

cat

head

tail

操作文件、目录

touch

mkdir

cp

mv

rm

文件、目录权限

ls -l

chmod

chown

chogrp

目录操作

/var/log/ 系统日志

cat /var/log/messages

程序资源

ps aux

top 可以持续检测程序的工作状态（可以设置更新周期，默认5秒）

df 磁盘空间

free 内存使用

sar CPU使用

w 查看负载（CPU、内存、IO三部分使用情况）

网络

netstat 查看网络状态：套接字、tcp封装包、udp封装包、网络程序PID

ifconfig 网络端口：网卡、IP地址

route 查询路由表

ip 包含上述两个命令

traceroute 数据包经过的路由

ping 测试网络连接是否连通

优先级

nice 设置程序的优先级

renice 重新设置

**grep**

搜寻特定字符串: grep -n ‘word’ text.txt

行首字符 ^ : grep -n ‘**^**word’ text.txt

行尾字符 $ : grep -n ‘word$’ text.txt

[ ] 来搜寻集合字符 : grep -n ‘**^[a-z1-9]**’ text.txt

任意一个字符 . 与重复字符 \* : grep -n ‘w..d’ text.txt

**sed 工具**

sed 可以分析标准输入的数据， 将数据经过处理后，再将其输出到标准输出上。注意：只对标准输入进行处理，并不改变源文件内容。可用-f写如文件。

处理功能有：取代、删除、新增、撷取特定行等等。

新增：cat text.txt | sed ‘1a new line’ 第1行的下一行增加一行

取代：cat text.txt | sed ‘1, 2c new 1-2line’ 1-2行取代

删除：cat text.txt | sed ‘1, 2d’ 删除1-2行

插入：cat text.txt | sed ‘1i new line’ 第1行上一行插入一行

动态链接和静态链接的区别  
指在生成可执行文件时不将所有程序用到的函数链接到一个文件，因为有许多函数在操作系统带的dll文件中，当程序运行时直接从操作系统中找。 而静态链接就是把所有用到的函数全部链接到exe文件中。  
动态链接是只建立一个引用的接口，而真正的代码和数据存放在另外的可执行模块中，在运行时再装入；而静态链接是把所有的代码和数据都复制到本模块中，运行时就不再需要库了。

死锁

产生条件：1、资源不能被共享，只能由一个进程使用。

1. 请求与保持条件（Hold and wait）：已经得到资源的进程可以再次申请新的资源。
2. 非剥夺条件（No pre-emption）：已经分配的资源不能从相应的进程中被强制地剥夺。
3. 循环等待条件（Circular wait）：多个进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。

处理死锁的策略：

1. 仔细地对资源进行动态分配，以避免死锁。

2. 通过破除死锁四个必要条件之一，来防止死锁产生。

**内存泄漏定位**

valgrind 工具

g++ -g -o leak leak.c # -g 调试模式

memcheck是valgrind中最常用的工具，用来检测程序中出现的内存问题，所有对内存的读写都会被检测到，一切对malloc/free/new/delete的调用都会被捕获。

使用：valgrind --tool=memcheck ./leak

其它参数：

--leak-check=full 完全检查内存泄漏；

--show-reachable=yes 显示内存泄漏的地点；

--trace-children=yes 跟入子进程。

**编译的四个阶段**

**预处理阶段**：预处理器根据以字符#开头的的命令，读取指定系统头文件的内容插入到程序中，得到一个扩展文件hello.i。

**编译阶段**：编译器将扩展文件hello.i翻译成文本文件hello.s，包含一个汇编语言程序。汇编程序可以为不同高级语言提供通用的的输出语言。

**汇编阶段**：汇编器将hello.s翻译成机器语言指令，并打包到目标文件中hello.o。

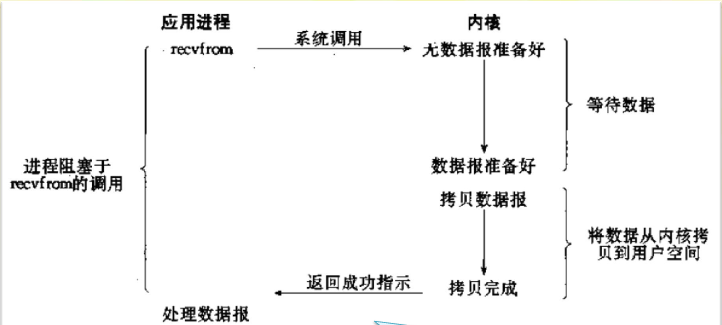
**链接阶段**：在程序中调用了标准C库中的函数，链接器负责把这些函数的目标文件合并到我们的程序中。最终得到可执行文件。

同步、异步、阻塞、非阻塞

在[linux](http://lib.csdn.net/base/linux)中，默认情况下所有的socket都是blocking。

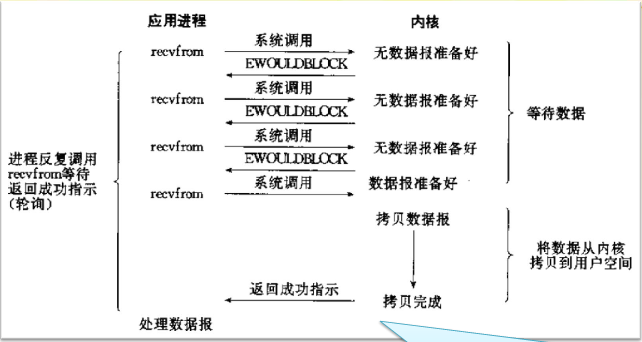
网络IO模型

同步阻塞



当调用recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。

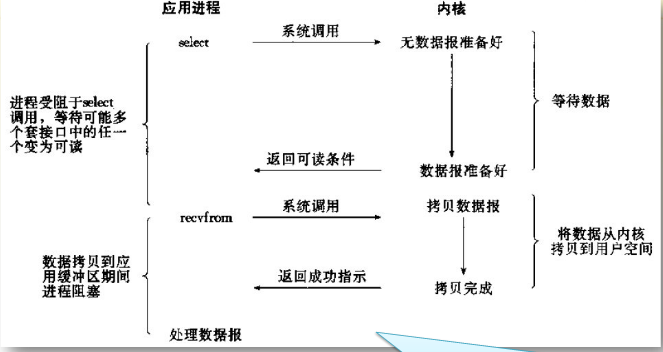
同步非阻塞



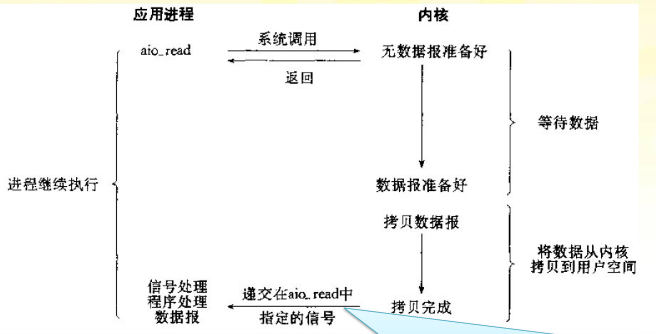
非阻塞IO通过进程反复调用IO函数（IO函数不阻塞，多次系统调用，并马上返回）；在数据准备好时，将数据从系统缓冲区复制到用户空间。

当所请求的I/O操作无法完成时，**不要将进程睡眠**，而是返回一个错误。这样我们的I/O操作函数将不断的[测试](http://lib.csdn.net/base/softwaretest)数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。

IO复用（同步非阻塞）



异步非阻塞



调用的读方法不阻塞，立刻返回。而且不需要关心数据是否准备好，只需要提供回调操作函数，内核会完成拷贝。

以read为例，read的具体操作分为以下两个部分：

　　（1）内核等待数据可读（发起IO请求）

　　（2）将内核读到的数据拷贝到进程（实际IO操作）

阻塞与非阻塞是对函数调用而言的：

阻塞：数据没有准备好时一直等待。

非阻塞：数据没有准备好时会立刻返回，然后不断调用。

同步与异步是针对这个操作和其他操作而言的：

同步：进程阻塞，直到IO操作（发起IO请求、拷贝过程）完成。

异步：IO操作由内核完成，进程发起IO请求之后可以做其他的事。

老板（进程）想喝咖啡，派我（函数）去看看咖啡好了没有（第一部分），好了就拿过来（第二部分）。

同步阻塞：我一直等在咖啡机旁边直到咖啡煮好，然后端过去。这个期间，老板也不能做其他的事。

同步非阻塞：没煮好我就回去报告，来回看有没有煮好，煮好之后端过去。这个期间老板也不能做其他的事。

异步非阻塞：我带一个机器人（回调函数）过去然后回来，煮好之后机器人会把咖啡端过来。这个期间老板可以做其他的事。