fork（） 是多进程 不同于多线程<http://blog.csdn.net/cywosp/article/details/27316803>

一、fork()函数

在操作系统的基本概念中进程是程序的一次执行，且是拥有资源的最小单位和调度单位（在引入线程的操作系统中，线程是最小的调度单位）。在Linux系统中创建进程有两种方式：一是由操作系统创建，二是由父进程创建进程（通常为子进程）。系统调用函数fork()是创建一个新进程的唯一方式，当然vfork()也可以创建进程，但是实际上其还是调用了fork()函数。fork()函数是Linux系统中一个比较特殊的函数，其一次调用会有两个返回值，下面是fork()函数的声明：

#include <unistd.h>

// On success, The PID of the process is returned in the parent, and 0 is returned in the child. On failure,

// -1 is returned in the parent, no child process is created, and errno is set appropriately.

pid\_t fork (void);

当程序调用fork()函数并返回成功之后，程序就将变成两个进程，调用fork()者为父进程，后来生成者为子进程。这两个进程将执行相同的程序文本，但却各自拥有不同的栈段、数据段以及堆栈拷贝。子进程的栈、数据以及栈段开始时是父进程内存相应各部分的完全拷贝，因此它们互不影响。从性能方面考虑，父进程到子进程的数据拷贝并不是创建时就拷贝了的，而是采用了写时拷贝（copy-on -write）技术来处理。调用fork()之后，父进程与子进程的执行顺序是我们无法确定的（即调度进程使用CPU），意识到这一点极为重要，因为在一些设计不好的程序中会导致资源竞争，从而出现不可预知的问题。

二 线程

与进程类似，线程（thread）是允许应用程序并发执行多个任务的一种机制。一个进程中可以包含多个线程，同一个程序中的所有线程均会独立执行，且共享同一份全局内存区域，其中包括初始化数据段（initialized data），未初始化数据段（uninitialized data），以及堆内存段（heap segment）。在多处理器环境下，多个线程可以同时执行，如果线程数超过了CPU的个数，那么每个线程的执行顺序将是无法确定的，因此对于一些全局共享数据据需要使用同步机制来确保其的正确性。

在系统中，线程也是稀缺资源，一个进程能同时创建多少个线程这取决于地址空间的大小和内核参数，一台机器可以同时并发运行多少个线程也受限于CPU的数目。在进行程序设计时，我们应该精心规划线程的个数，特别是根据机器CPU的数目来设置工作线程的数目，并为关键任务保留足够的计算资源。如果你设计的程序在背地里启动了额外的线程来执行任务，那这也属于资源规划漏算的情况，从而影响关键任务的执行，最终导致无法达到预期的性能。很多程序中都存在全局对象，这些全局对象的初始化工作都是在进入main()函数之前进行的，为了能保证全局对象的安全初始化（按顺序的），因此在程序进入main()函数之前应该避免线程的创建，从而杜绝未知错误的发生。

三、fork()与多线程

推荐在多线程程序中调用fork()的唯一情况是：其后立即调用exec()函数执行另一个程序，彻底隔断子进程与父进程的关系。由新的进程覆盖掉原有的内存，使得子进程中的所有pthreads对象消失。

对于那些必须执行fork()，而其后又无exec()紧随其后的程序来说，pthreads API提供了一种机制：fork()处理函数。利用函数pthread\_atfork()来创建fork()处理函数。pthread\_atfork()声明如下：

#include <pthread.h>

// Upon successful completion, pthread\_atfork() shall return a value of zero; otherwise, an error number shall be returned to indicate the error.

// @prepare 新进程产生之前被调用

// @parent 新进程产生之后在父进程被调用

// @child 新进程产生之后，在子进程被调用

int pthread\_atfork (void (\*prepare) (void), void (\*parent) (void), void (\*child) (void));

该函数的作用就是往进程中注册三个函数，以便在不同的阶段调用，有了这三个参数，我们就可以在对应的函数中加入对应的处理功能。同时需要注意的是，每次调用pthread\_atfork()函数会将prepare添加到一个函数列表中，创建子进程之前会（按与注册次序相反的顺序）自动执行该函数列表中函数。parent与child也会被添加到一个函数列表中，在fork()返回前，分别在父子进程中自动执行（按注册的顺序）。具体事例可参考：<http://blog.chinaunix.net/uid-26885237-id-3210394.html>

四、总结

fork()函数的调用会导致在子进程中除调用线程外的其它线程全都终止执行并消失，因此在多线程的情况下会导致死锁和内存泄露的情况。在进行多线程编程的时候尽量避免fork()的调用，同时在程序在进入main函数之前应避免创建线程，因为这会影响到全局对象的安全初始化。线程不应该被强行终止，因为这样它就没有机会调用清理函数来做相应的操作，同时也就没有机会来释放已被锁住的锁，如果另一线程对未被解锁的锁进行加锁，那么将会立即发生死锁，从而导致程序无法正常运行。

//================================================

孤儿进程、僵尸进程

fork系统调用之后，父子进程将交替执行，执行顺序不定。

如果父进程先退出，子进程还没退出那么子进程的父进程将变成init进程（托孤给了init进程）。（注：任何一个进程都必须有父进程）

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵进程（僵尸进程：只保留一些退出信息供父进程查询）

进程终止

\_exit()函数的作用最为简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；exit() 函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序。

exit()函数与\_exit()函数最大的区别就在于exit()函数在调用exit系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是"清理I/O缓冲"。

exit()在结束调用它的进程之前，要进行如下步骤：

1.调用atexit()注册的函数（出口函数）；按ATEXIT注册时相反的顺序调用所有由它注册的函数,这使得我们可以指定在程序终止时执行自己的清理动作.例如,保存程序状态信息于某个文件,解开对共享数据库上的锁等.

2.cleanup()；关闭所有打开的流，这将导致写所有被缓冲的输出，删除用TMPFILE函数建立的所有临时文件.

3.最后调用\_exit()函数终止进程。

\_exit做3件事（man）：

1，Any open file descriptors belonging to the process are closed

2，any children of the process are inherited by process 1, init

3，the process's parent is sent a SIGCHLD signal

exit执行完清理工作后就调用\_exit来终止进程。

如何避免僵尸进程

调用wait或者waitpid函数查询子进程退出状态，此方法父进程会被挂起。

如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG\_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的

更精确的延迟

**nanosleep()**

#include <time.h>

int nanosleep(const struct timespec \*req, struct timespec \*rem);

CPU时间与处理器耗时

CPU时间(CPU TIME)是指当应用进程启动后，占用CPU进行计算所进行的时间绝对值，或叫时间点。如果进程进入中断、挂起、休眠等行为时，是不占用CPU的，所以CPU时间并不会跟着增加，且进程恢复运行后所获得的CPU时间值与运行中断之前的时间值是连续的，并不会因为运行的暂停而导致CPU时间的跳跃。这是和普通的日常时间值不一样的点。另一个不同的点是CPU时间并不是以秒为单位的，而是处理器的时钟周期为单位。

获取CPU时间的函数

#include <time.h>

clock\_t clock(void)

该函数返回CPU时间，CPU时间并不一定会从0开始，所以要计算进程的一段代码所消耗的CPU时间记录需要在代码前后获取各自的起始和终止CPU时间，两者之差即为所消耗的CPU时间，即为处理器耗时(Processor Time)。

要将clock\_t转换为秒数，需要除以宏CLOCKS\_PER\_SEC生成的值，即为秒数。但要注意的是由于在不同的系统中clock\_t或CLOCKS\_PER\_SEC可能为int也可能为浮点型数值，所以在计算时需要在计算过程中并其强制转换为double以保证计算结果。

另外需要注意的是clock()函数所返回的值在32位系统上每72分钟就会恢复原始值一次。

clock()如果执行错误，则会返回-1。

获取CPU时间与更详细的进程时间还有另外一个函数：

#include <sys/times.h>

clock\_t times(struct tms \*buf);

struct tms {

clock\_t tms\_utime; /\* user time \*/

clock\_t tms\_stime; /\* system time \*/

clock\_t tms\_cutime; /\* user time of children \*/

clock\_t tms\_cstime; /\* system time of children \*/

};

函数times()会将进程的处理器时间分类。在参数tms中各字段含义为：

tms\_utime:用户层处理器耗时。即执行该进程的实现代码所消耗的CPU时间和。

tms\_stime:系统层处理器耗时。即该进程实现的代码调用了系统内核代码，由这部分内核代码执行所消耗的CPU时间和。

tms\_cutime:子进程用户层处理器耗时。即如果当前进程通过调用system()或fork()函数导致产生新的子进程(数量前后可能为多个)，则记录所有子进程的实现代码所消耗的CPU时间和。

tms\_cstime:子进程用户层处理器耗时。即如果当前进程通过调用system()或fork()函数导致产生新的子进程(数量前后可能为多个)，则记录所有子进程实现的代码调用了系统内核代码，由这部分内核代码执行所消耗的CPU时间和。

函数times()所返回的值单位都是时钟滴哒(clock tick)数。要获取每秒钟的时钟滴哒可以调用函数sysconf(\_SC\_CLK\_TCK);来获取。为了保证精度，在计算耗时时，也要在计算过程中将参与的计算值转换为double以保证计算结果。

函数times()所返回的值表示过去某个时间点到函数调用此刻的时间消耗长度。所以进程的中断、挂起、休眠等行为仍会导致返回值的持续增加。这一点是与clock()的不同所在。通常也利用这个特点来计算进程的前后自然时间耗时时长。

在计算时长方面，times()通过精确到秒以下(可能是毫秒，因为是double类型的计算)，但缺点是参数buf必须不为空；difftime()由于其参数设定的原因只能精确到秒，能力是最弱的；gettimeofday()所获取的timeval则可精确到微秒，只是计算时间差值时麻烦一点。