很想写点关于多进程和多线程的东西，我确实很爱他们。但是每每想动手写点关于他们的东西，却总是求全心理作祟，始终动不了手。

今天终于下了决心，写点东西，以后可以再修修补补也无妨。

一.为何需要多进程（或者多线程），为何需要并发？

这个问题或许本身都不是个问题。但是对于没有接触过多进程编程的朋友来说，他们确实无法感受到并发的魅力以及必要性。

我想，只要你不是整天都写那种int main()到底的代码的人，那么或多或少你会遇到代码响应不够用的情况，也应该有尝过并发编程的甜头。就像一个快餐点的服务员，既要在前台接待客户点餐，又要接电话送外卖，没有分身术肯定会忙得你焦头烂额的。幸运的是确实有这么一种技术，让你可以像孙悟空一样分身，灵魂出窍，乐哉乐哉地轻松应付一切状况,这就是多进程/线程技术。

并发技术，就是可以让你在同一时间同时执行多条任务的技术。你的代码将不仅仅是从上到下，从左到右这样规规矩矩的一条线执行。你可以一条线在main函数里跟你的客户交流，另一条线，你早就把你外卖送到了其他客户的手里。

所以，为何需要并发？因为我们需要更强大的功能，提供更多的服务，所以并发，必不可少。

二.多进程

什么是进程。最直观的就是一个个pid,官方的说法就：进程是程序在计算机上的一次执行活动。

说得简单点，下面这段代码执行的时候

**[cpp]** view plain copy

1. **int** main()
3. {
5. printf(”pid is %d/n”,getpid() );
7. **return** 0;
9. }

进入main函数，这就是一个进程，进程pid会打印出来，然后运行到return，该函数就退出，然后由于该函数是该进程的唯一的一次执行，所以return后，该进程也会退出。

看看多进程。**Linux**下创建子进程的调用是fork();

**[cpp]** view plain copy

1. #include <unistd.h>
2. #include <sys/types.h>
3. #include <stdio.h>


7. **void** print\_exit()
8. {
9. printf("the exit pid:%d/n",getpid() );
10. }
12. main ()
13. {
14. pid\_t pid;
15. atexit( print\_exit );      //注册该进程退出时的回调函数
16. pid=fork();
17. **if** (pid < 0)
18. printf("error in fork!");
19. **else** **if** (pid == 0)
20. printf("i am the child process, my process id is %d/n",getpid());
21. **else**
22. {
23. printf("i am the parent process, my process id is %d/n",getpid());
24. sleep(2);
25. wait();
26. }
28. }

i am the child process, my process id is 15806  
the exit pid:15806  
i am the parent process, my process id is 15805  
the exit pid:15805

这是gcc**测试**下的运行结果。

关于fork函数，功能就是产生子进程，由于前面说过，进程就是执行的流程活动。

那么fork产生子进程的表现就是它会返回2次，一次返回0，顺序执行下面的代码。这是子进程。

一次返回子进程的pid，也顺序执行下面的代码，这是父进程。

（为何父进程需要获取子进程的pid呢？这个有很多原因，其中一个原因：看最后的wait，就知道父进程等待子进程的终结后，处理其task\_struct结构，否则会产生僵尸进程,扯远了，有兴趣可以自己google）。

如果fork失败，会返回-1.

额外说下atexit( print\_exit ); 需要的参数肯定是函数的调用地址。

这里的print\_exit 是函数名还是函数指针呢？答案是函数指针，函数名永远都只是一串无用的字符串。

某本书上的规则：函数名在用于非函数调用的时候，都等效于函数指针。

说到子进程只是一个额外的流程，那他跟父进程的联系和区别是什么呢？

我很想建议你看看**linux**内核的注解（有兴趣可以看看，那里才有本质上的了解），总之,fork后，子进程会复制父进程的task\_struct结构，并为子进程的堆栈分配物理页。理论上来说，子进程应该完整地复制父进程的堆，栈以及数据空间，但是2者共享正文段。

关于写时复制：由于一般 fork后面都接着exec，所以，现在的 fork都在用写时复制的技术，顾名思意，就是，数据段，堆，栈，一开始并不复制，由父，子进程共享，并将这些内存设置为只读。直到父，子进程一方尝试写这些区域，则内核才为需要修改的那片内存拷贝副本。这样做可以提高 fork的效率。

三.多线程

线程是可执行代码的可分派单元。这个名称来源于“执行的线索”的概念。在基于线程的多任务的环境中，所有进程有至少一个线程，但是它们可以具有多个任务。这意味着单个程序可以并发执行两个或者多个任务。

简而言之，线程就是把一个进程分为很多片，每一片都可以是一个独立的流程。这已经明显不同于多进程了，进程是一个拷贝的流程，而线程只是把一条河流截成很多条小溪。它没有拷贝这些额外的开销，但是仅仅是现存的一条河流，就被多线程技术几乎无开销地转成很多条小流程，它的伟大就在于它少之又少的系统开销。（当然伟大的后面又引发了重入性等种种问题，这个后面慢慢比较）。

还是先看linux提供的多线程的系统调用：

|  |
| --- |
| int pthread\_create(pthread\_t \*restrict tidp,                    const pthread\_attr\_t \*restrict attr,                    void \*(\*start\_rtn)(void),                     void \*restrict arg); |
| Returns: 0 if OK, error number on failure |

第一个参数为指向线程标识符的指针。  
第二个参数用来设置线程属性。  
第三个参数是线程运行函数的起始地址。  
最后一个参数是运行函数的参数。

**[cpp]** view plain copy

1. #include<stdio.h>
2. #include<string.h>
3. #include<stdlib.h>
4. #include<unistd.h>
5. #include<pthread.h>

8. **void**\* task1(**void**\*);
9. **void**\* task2(**void**\*);

12. **void** usr();
13. **int** p1,p2;
15. **int** main()
16. {
17. usr();
18. getchar();
19. **return** 1;
20. }


24. **void** usr()
25. {
26. pthread\_t pid1, pid2;
27. pthread\_attr\_t attr;
28. **void** \*p;
29. **int** ret=0;
30. pthread\_attr\_init(&attr);         //初始化线程属性结构
31. pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);   //设置attr结构为分离
32. pthread\_create(&pid1, &attr, task1, NULL);         //创建线程，返回线程号给pid1,线程属性设置为attr的属性，线程函数入口为task1，参数为NULL
33. pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);
34. pthread\_create(&pid2, &attr, task2, NULL);
35. //前台工作
37. ret=pthread\_join(pid2, &p);         //等待pid2返回，返回值赋给p
38. printf("after pthread2:ret=%d,p=%d/n", ret,(**int**)p);
40. }
42. **void**\* task1(**void** \*arg1)
43. {
44. printf("task1/n");
45. //艰苦而无法预料的工作，设置为分离线程，任其自生自灭
46. pthread\_exit( (**void** \*)1);
48. }
50. **void**\* task2(**void** \*arg2)
51. {
52. **int** i=0;
53. printf("thread2 begin./n");
54. //继续送外卖的工作
55. pthread\_exit((**void** \*)2);
56. }

这个多线程的例子应该很明了了，主线程做自己的事情，生成2个子线程，task1为分离，任其自生自灭，而task2还是继续送外卖，需要等待返回。（因该还记得前面说过僵尸进程吧，线程也是需要等待的。如果不想等待，就设置线程为分离线程）

 额外的说下，linux下要编译使用线程的代码，一定要记得调用pthread库。如下编译：

 gcc -o pthrea -pthread  pthrea.c

四.比较以及注意事项

1.看完前面，应该对多进程和多线程有个直观的认识。如果总结多进程和多线程的区别，你肯定能说，前者开销大，后者开销较小。确实，这就是最基本的区别。

2.线程函数的可重入性：

说到函数的可重入，和线程安全，我偷懒了，引用网上的一些总结。

线程安全：概念比较直观。一般说来，一个函数被称为线程安全的，当且仅当被多个并发线程反复调用时，它会一直产生正确的结果。

可重入：概念基本没有比较正式的完整解释，但是它比线程安全要求更严格。根据经验，所谓“重入”，常见的情况是，程序执行到某个函数foo()时，收到信号，于是暂停目前正在执行的函数，转到信号处理函数，而这个信号处理函数的执行过程中，又恰恰也会进入到刚刚执行的函数foo()，这样便发生了所谓的重入。此时如果foo()能够正确的运行，而且处理完成后，之前暂停的foo()也能够正确运行，则说明它是可重入的。

线程安全的条件：

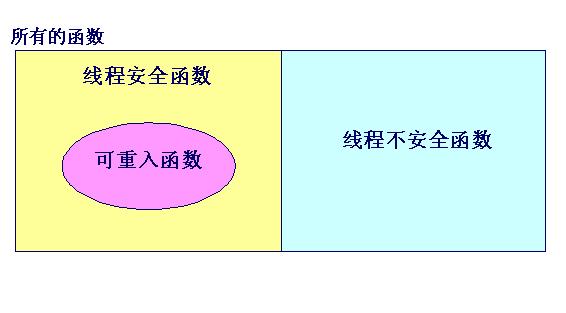
要确保函数线程安全，主要需要考虑的是线程之间的共享变量。属于同一进程的不同线程会共享进程内存空间中的全局区和堆，而私有的线程空间则主要包括栈和寄存器。因此，对于同一进程的不同线程来说，每个线程的局部变量都是私有的，而全局变量、局部静态变量、分配于堆的变量都是共享的。在对这些共享变量进行访问时，如果要保证线程安全，则必须通过加锁的方式。

可重入的判断条件：

要确保函数可重入，需满足一下几个条件：

1、不在函数内部使用静态或全局数据   
2、不返回静态或全局数据，所有数据都由函数的调用者提供。   
3、使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据。  
4、不调用不可重入函数。

可重入与线程安全并不等同，一般说来，可重入的函数一定是线程安全的，但反过来不一定成立。它们的关系可用下图来表示：



比如：strtok函数是既不可重入的，也不是线程安全的；加锁的strtok不是可重入的，但线程安全；而strtok\_r既是可重入的，也是线程安全的。

如果我们的线程函数不是线程安全的，那在多线程调用的情况下，可能导致的后果是显而易见的——共享变量的值由于不同线程的访问，可能发生不可预料的变化，进而导致程序的错误，甚至崩溃。

3.关于IPC(进程间通信)

由于多进程要并发协调工作，进程间的同步，通信是在所难免的。

稍微列举一下linux常见的IPC.

linux下进程间通信的几种主要手段简介：

1. 管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；
2. 信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）；
3. 报文（Message）队列（消息队列）：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
4. 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
5. 信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。
6. 套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

或许你会有疑问，那多线程间要通信，应该怎么做？前面已经说了，多数的多线程都是在同一个进程下的，它们共享该进程的全局变量，我们可以通过全局变量来实现线程间通信。如果是不同的进程下的2个线程间通信，直接参考进程间通信。

4.关于线程的堆栈

说一下线程自己的堆栈问题。

是的，生成子线程后，它会获取一部分该进程的堆栈空间，作为其名义上的独立的私有空间。（为何是名义上的呢？)由于，这些线程属于同一个进程，其他线程只要获取了你私有堆栈上某些数据的指针，其他线程便可以自由访问你的名义上的私有空间上的数据变量。（注：而多进程是不可以的，因为不同的进程，相同的虚拟地址，基本不可能映射到相同的物理地址）

5.在子线程里fork

看过好几次有人问，在子线程函数里调用system或者 fork为何出错，或者fork产生的子进程是完全复制父进程的吗？

我测试过，只要你的线程函数满足前面的要求，都是正常的。

**[cpp]** view plain copy

1. #include<stdio.h>
2. #include<string.h>
3. #include<stdlib.h>
4. #include<unistd.h>
5. #include<pthread.h>
7. **void**\* task1(**void** \*arg1)
8. {
9. printf("task1/n");
10. system("ls");
11. pthread\_exit( (**void** \*)1);
12. }
14. **int** main()
15. {
16. **int** ret=0;
17. **void** \*p;
18. **int** p1=0;
19. pthread\_t pid1;
20. pthread\_create(&pid1, NULL, task1, NULL);
21. ret=pthread\_join(pid1, &p);
22. printf("end main/n");
23. **return** 1;
24. }

上面这段代码就可以正常得调用ls指令。

不过，在同时调用多进程（子进程里也调用线程函数）和多线程的情况下，函数体内很有可能死锁。

具体的例子可以看看这篇文章。

http://www.cppblog.com/lymons/archive/2008/06/01/51836.aspx