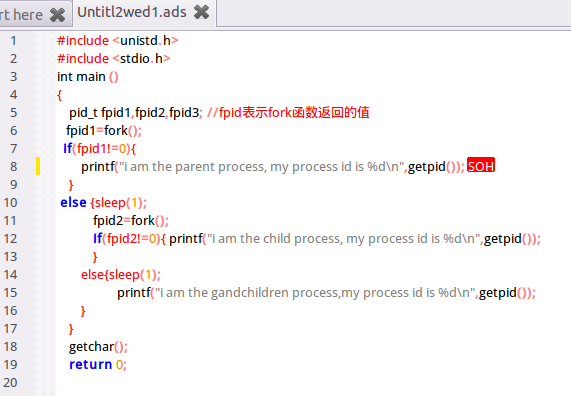
fork()函数及pthread\_create()函数学习报告

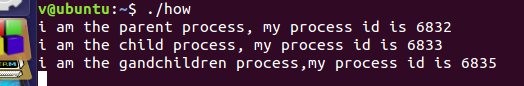
2014301500374 张巍

fork()函数的作用是copy代码段中剩余的函数，并新建与原函数一致的变量分配，建立一个与原函数几乎一致的子进程。（但是子进程并不会再次执行fork之前的函数，即每个fork只会被一个进程执行一次，否则会陷入无限循环）

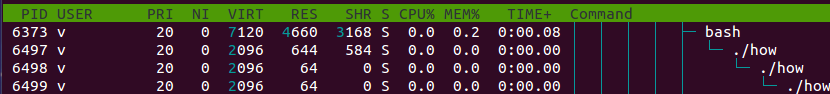
为了建立一个三层次的父子进程，写出一下代码



为了判断fork出的进程是父进程还是子进程，引入了fork的返回值fork1，fork2.因为fork函数会返回创建出的子进程的pid，而在子进程中fork函数不会被再次执行，故在子进程再次执行本身的第二次fork前，fork的值是0（只声明，未赋值）。则可以引入对fork的返回值的判断，来确定当前进程是父进程、子进程、还是孙子进程。首先用一个fork造出子进程，再判断出子进程后fork出孙子进程，而父进程和孙子进程不再进行fork。此时便成功构造出一个三级父子进程。

运行结果如下

进程监视结果如下

而pthread()函数的作用，则是再进程下创建线程，以执行额外的函数命令。

int  pthread\_create(pthread\_t  \*  thread,

pthread\_attr\_t \* attr,

void \* (\*start\_routine)(void \*),

void \* arg)

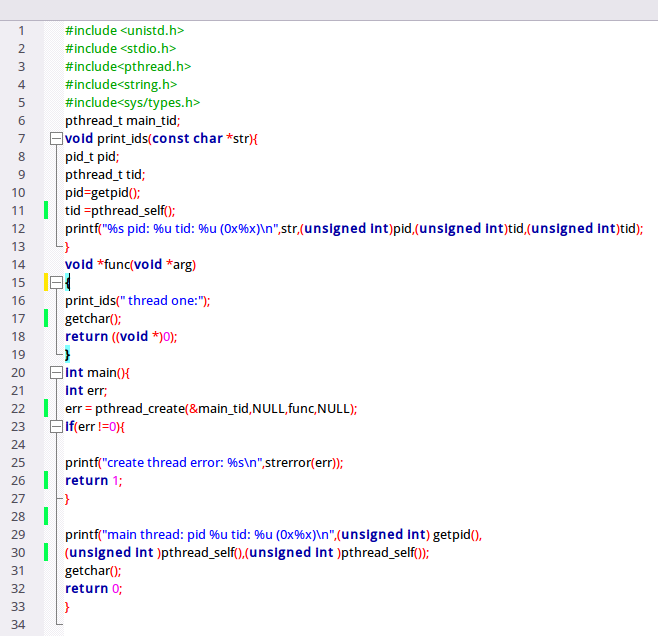
thread：返回创建的线程的ID

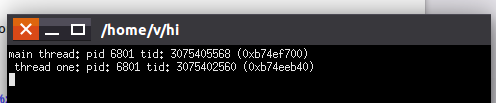
attr：线程属性，调度策略、优先级等都在这里设置，如果为NULL则表示用默认属性

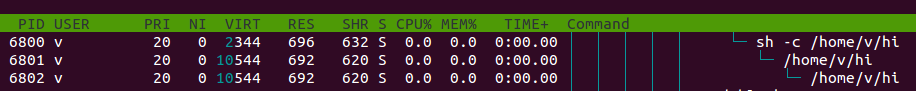
start\_routine：线程入口函数，可以返回一个void\*类型的返回值，该返回值可由pthread\_join()捕获，即线程需要执行的函数。

arg：传给start\_routine的参数， 可以为NULL

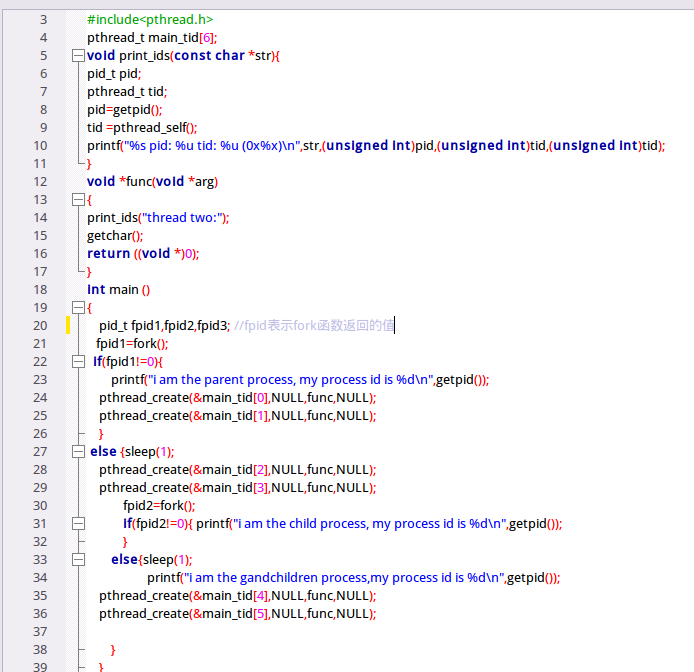
使用以下代码实现对进程赋予多线程

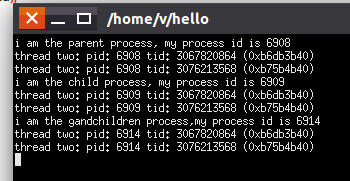
运行结果如下



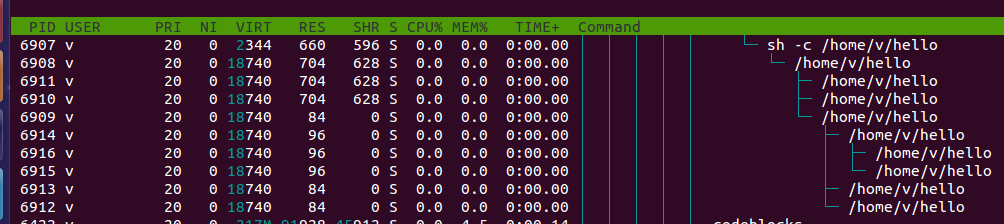
内存使用情况如下

将两者代码综合得出以下结果

若要讲两者综合，只需在判定目标是父/子/孙进程后，执行相应的pthreadcreate()，变可得到需求的结果。代码如下在判定该进程是父/子/孙进程后，再进行线程的创建操作，避免pthread\_create()被循环调用。运算结果如下



进程监视结果如下

每个进程下有一个子进程和两个线程。

查阅相关资料后，可以得知linux系统的线程操作实际上是使用了进程来进行模仿的，所以获得的线程实际上也被分配了一个pid。

实际讲fork与pthread结合起来之后，发现fork出的子进程中的线程的tid与父进程的tid是一样的。理论上线程之间的tid是唯一而且互异的。

出现如上情况的原因是，fork产生的子进程照搬了父进程中的参量值，导致pthread调用时会使用相同的指针构建线程。

并且业内也尽量避免fork与pthread的混用，因为这种调用会复制同一个mutex进入子进程，很容易造出子进程死锁。引入了pthread\_atfork()函数来控制fork（）的具体执行过程。由于时间原因，未进行更深入的探讨。

附言：

在查阅csdn等相关网站的和博客时，发现关于fork和pthread的混用存在一些问题，是linux系统的历史遗留问题，因为在最早时程序都是单线程的，fork()函数可以正常运行，而90年代出，多线程的兴起，导致了fork（）函数出现了诸多问题。

关于编译环境的搭建：由于<pthread.h>库的静态引用在linux下会出现问题，需要在compiler中加入额外的linker来实现对库的调用，否则会出现若干pthread函数未定义的错误。并且需要加入-lpthread的options。

关于对ubuntu的体验：在vmware下搭建的ubuntu15.10环境，有着诸多的问题，最常见的就是codeblock的崩溃与假死，导致编译过程极为艰辛。中文环境的不完善，gcc/g++的安装等诸多问题。vmtools对15.10的兼容性问题，再查阅官网更新文档后才得意解决。整体体验较差，作为PC的OS十分难受。。。

以上