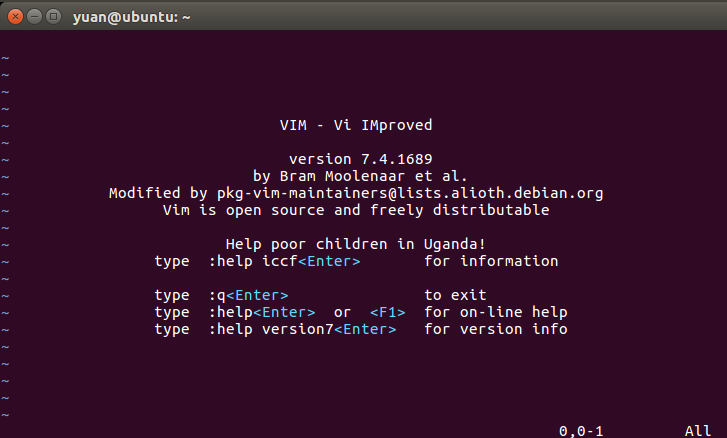
**实验二: 进程控制**

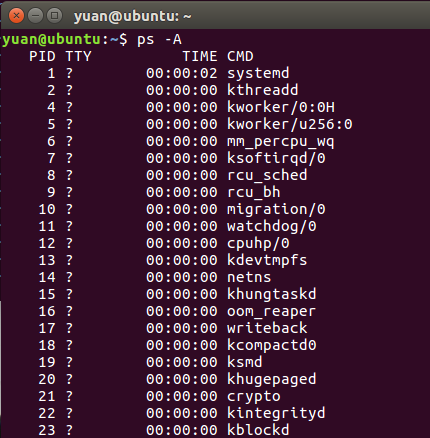
袁少随 16281054 安全1601

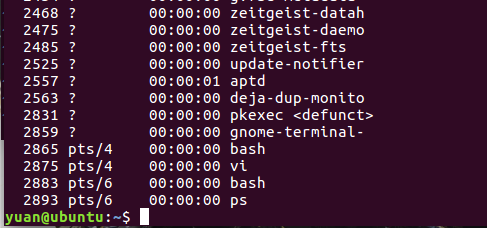
* **Task 1:打开一个vi进程。通过ps命令以及选择合适的参数，只显示名字为vi的进程。寻找vi进程的父进程，直到init进程为止。记录过程中所有进程的ID和父进程ID。将得到的进程树和由pstree命令的得到的进程树进行比较。**

(1)打开一个终端，输入vi,回车打开一个vi进程；



（2）打开另一个终端，通过 ps -A 显示所有进程；



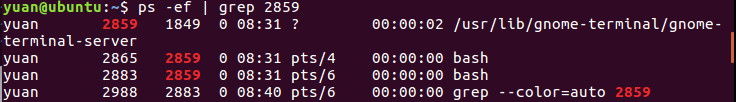
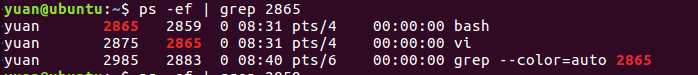


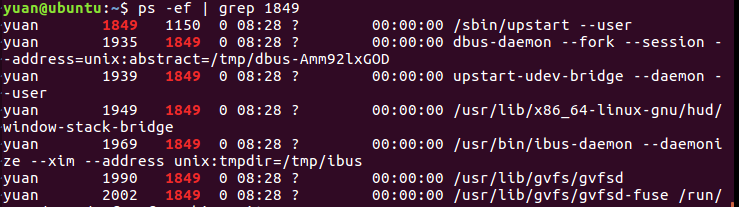
（3）通过 ps aux |grep vi 命令显示名字为vi的进程；

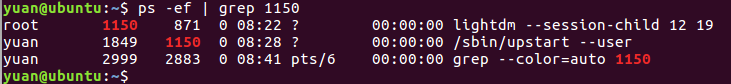


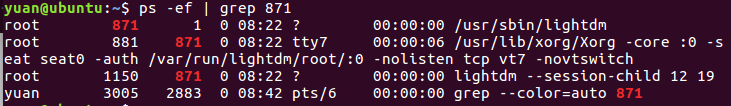
由上图可知vi进程的PID为2875；

（4）通过命令 ps -ef | grep PID 寻找vi 的父进程，直到init进程；









可知：

2875->2865->2859->1849->1150->871->1

**ps -ef|grep详解**

ps命令将某个进程显示出来;

grep命令是查找;

中间的|是管道命令 是指ps命令与grep同时执行;

PS是LINUX下最常用的也是非常强大的进程查看命令;

grep命令是查找，是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。grep全称是Global Regular Expression Print，表示全局正则表达式版本，它的使用权限是所有用户。

以下这条命令是检查java 进程是否存在：ps -ef |grep java

字段含义如下：

UID PID PPID C STIME TTY TIME CMD

zzw 14124 13991 0 00:38 pts/0 00:00:00 grep --color=auto dae

UID：程序被该 UID 所拥有

PID：就是这个程序的 ID

PPID：则是其上级父程序的ID

C：CPU使用的资源百分比

STIME：系统启动时间

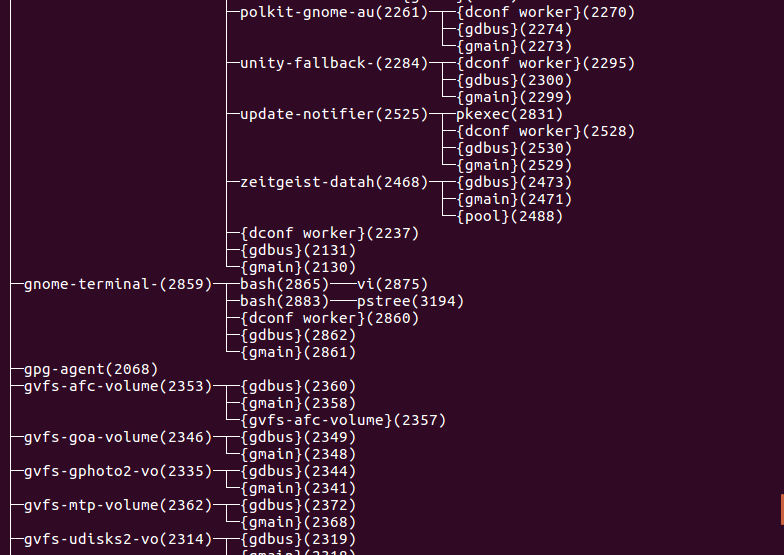
TTY：登入者的终端机位置

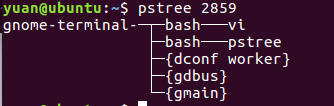
TIME：使用掉的CPU时间。

CMD：所下达的是什么指令

（5）通过命令 pstree PID得到进程树；(pstree -p为查询整个进程树）

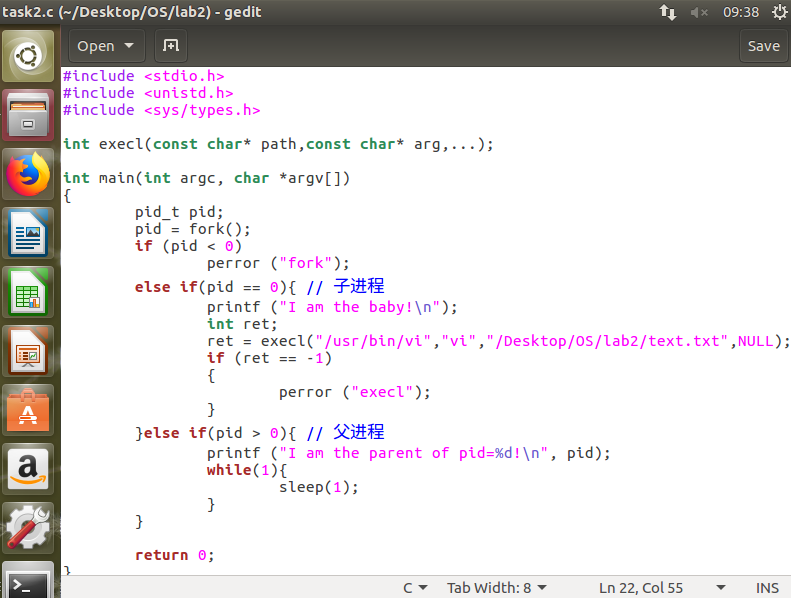






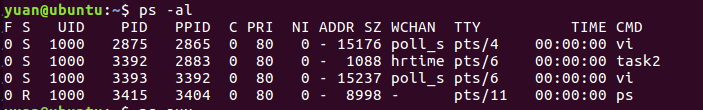
对比发现两种方式所得结果一样；

* **2、编写程序，首先使用fork系统调用，创建子进程。在父进程中继续执行空循环操作；在子进程中调用exec打开vi编辑器。然后在另外一个终端中，通过ps –Al命令、ps aux或者top等命令，查看vi进程及其父进程的运行状态，理解每个参数所表达的意义。选择合适的命令参数，对所有进程按照cpu占用率排序。**

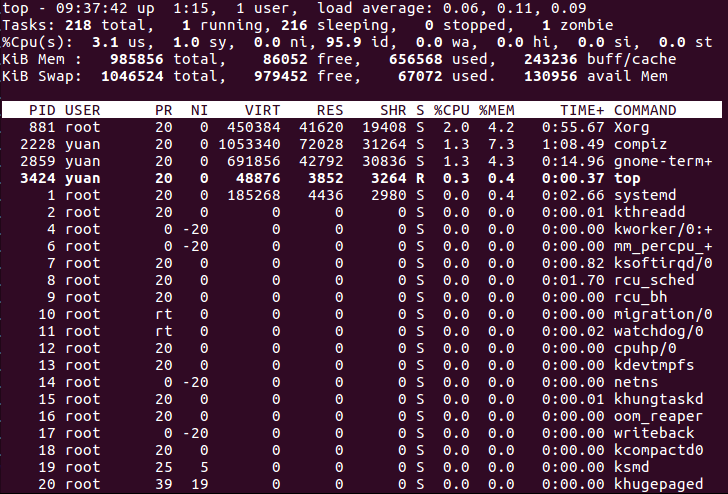


程序代码截图

ps –Al命令查看vi进程及其父进程的运行状态：



所有进程按照cpu占用率排序：



参数命令解析：

PID 进程id

PPID 父进程id

RUSER Real user name

UID 进程所有者的用户id

USER 进程所有者的用户名

GROUP 进程所有者的组名

TTY 启动进程的终端名。不是从终端启动的进程则显示为 ?

PR 优先级

NI nice值。负值表示高优先级，正值表示低优先级

P 最后使用的CPU，仅在多CPU环境下有意义

%CPU 上次更新到现在的CPU时间占用百分比

TIME 进程使用的CPU时间总计，单位秒

TIME+ 进程使用的CPU时间总计，单位1/100秒

%MEM 进程使用的物理内存百分比

VIRT 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES

SWAP 进程使用的虚拟内存中，被换出的大小，单位kb。

RES 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA

CODE 可执行代码占用的物理内存大小，单位kb

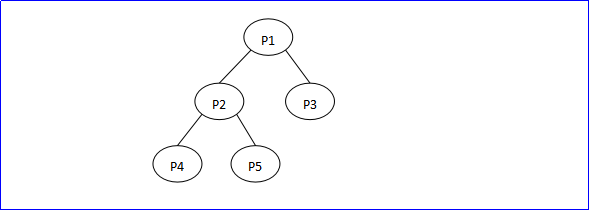
DATA 可执行代码以外的部分(数据段+栈)占用的物理内存大小，单位kb

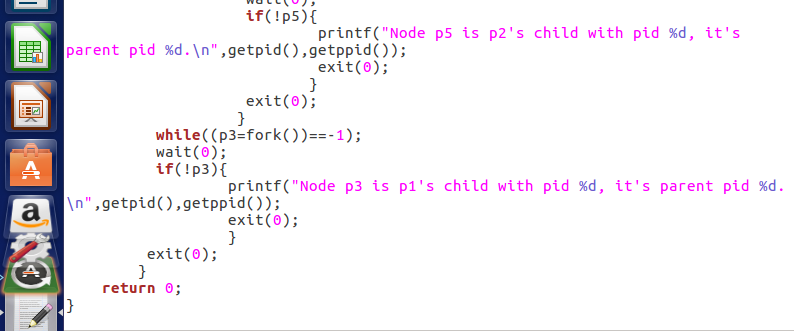
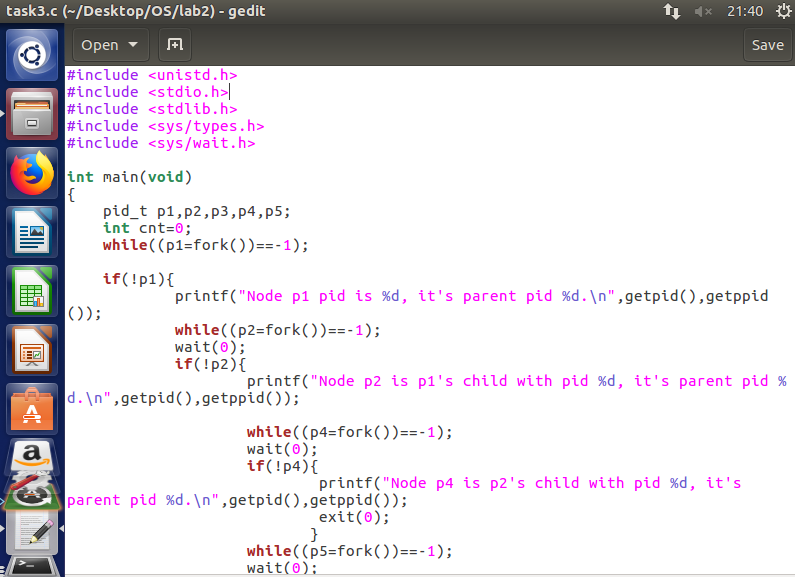
SHR 共享内存大小，单位kb

nFLT 页面错误次数

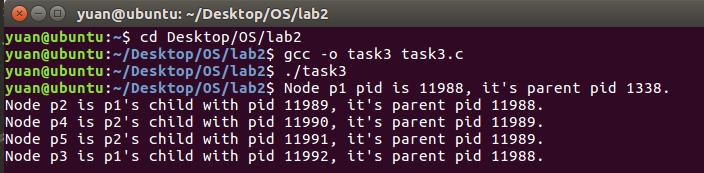
nDRT 最后一次写入到现在，被修改过的页面数。

* **3、使用fork系统调用，创建如下进程树，并使每个进程输出自己的ID和父进程的ID。观察进程的执行顺序和运行状态的变化。**

****



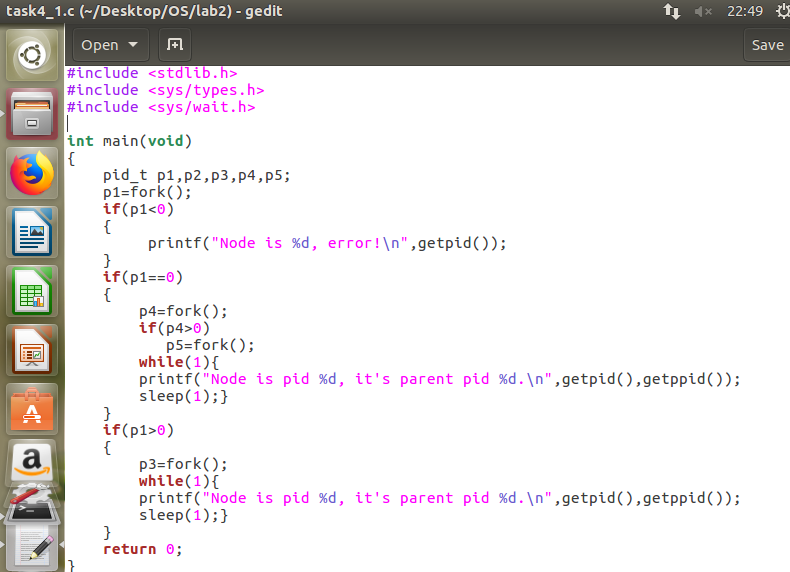
程序代码截图



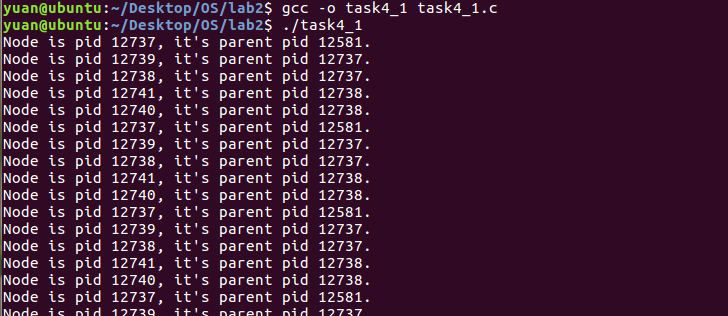
程序输出结果

执行顺序为：P1->P2->P4->P5->P3

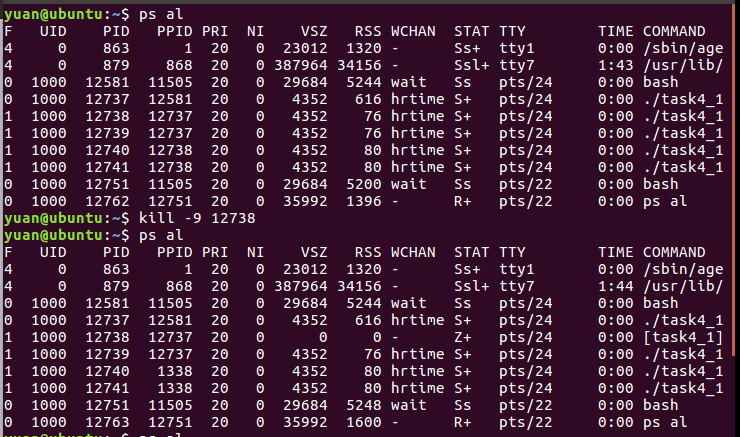
* **4、修改上述进程树中的进程，使得所有进程都循环输出自己的ID和父进程的ID。然后终止p2进程(分别采用kill -9 、自己正常退出exit()、段错误退出)，观察p1、p3、p4、p5进程的运行状态和其他相关参数有何改变。**



程序代码截图

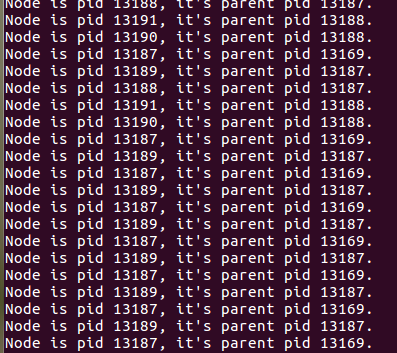
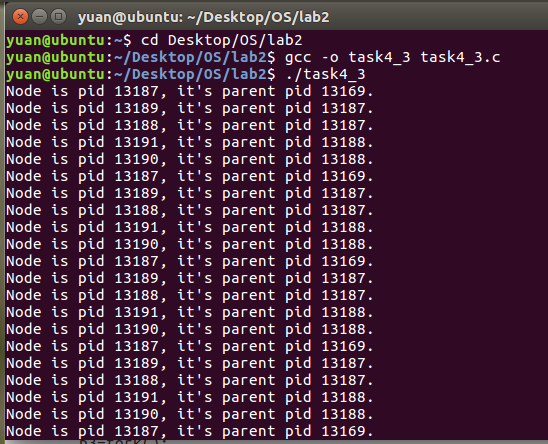


程序输出结果

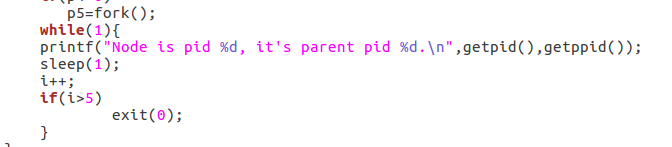


kill -9 终止P2 进程结果

P2 进程被杀死后，P4、P5 进程还在，不过父进程改变了。P2 进程的STAT状态由S+变为Z+;

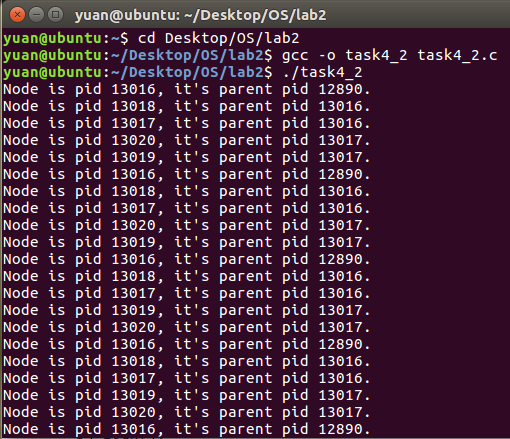


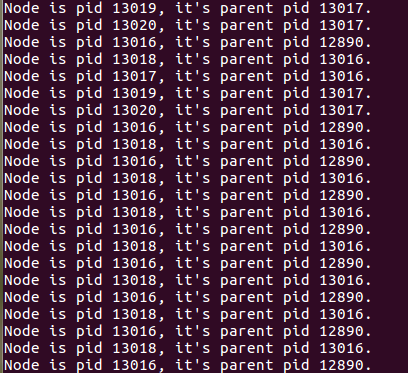
自己正常退出exit()程序结果

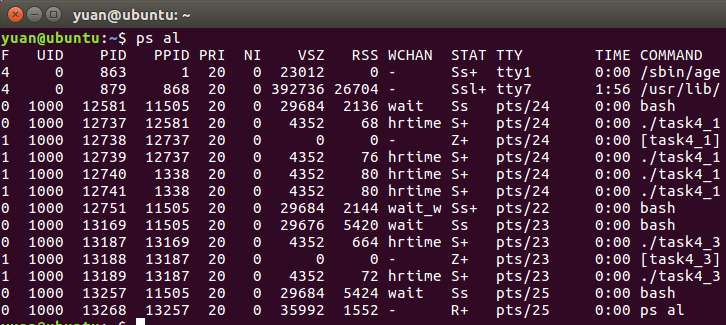


自己正常退出exit()部分代码更改

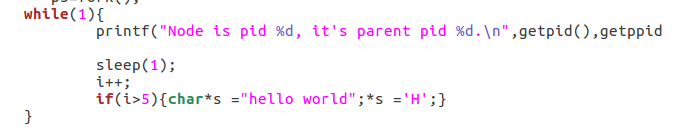
自己正常退出exit():利用i 变量设置循环5次后执行exit(0),退出。







段错误退出程序结果



段错误退出部分代码更改

**段错误注解：**

段错误就是指访问的内存超出了系统所给这个程序的内存空间，通常这个值是由gd tr来保存的，他是一个48位的寄存器，其中的32位是保存由它指向的 gdt表，后13位保存 相应于gdt的下标，最后3位包括了程序是否在内存中以及程序的在cpu中的运行级别，指向 的gdt是由以64位为一个单位的表，在这张表中就保存着程序运行的代码段以及数据段的起 始地址以及与此相应的段限和页面交换还有程序运行级别还有内存粒度等等的信息。

**常用段错误：**

1. int main(void){

char\*s ="hello world";

\*s ='H';

}

被装载时，系统把“hello world” 连同其它的字符串和const型数据放入到内存的只读区。执行时，一个变量s被设为指向该字符串的位置，当再试图向该位置写时，就会产生段错误。

2，

int\*ptr = NULL;

\*ptr =1;

因为该代码只创建了一个空指针，并没有指向一个具体空间，当赋值时，产生段错误。

3，

int main(void){

main();

return0;

}

无限递归，这会导致栈溢出，也会产生段错误。