实验2 2.1进程管理与进程创建

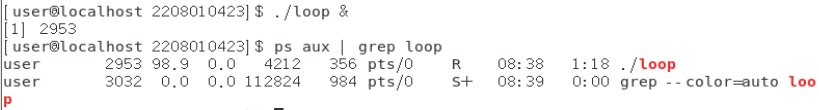
**实验目的**

1. 掌握Linux中进程的创建方法及执行情况。
2. 加深对进程、进程树等概念的理解。

**实验内容**

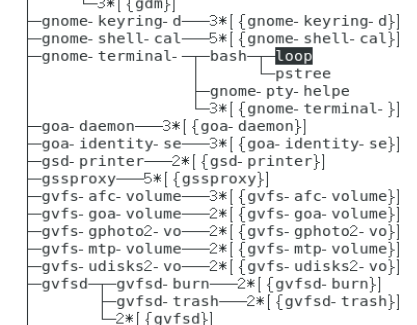
**注意：所有操作和实验程序都要保存到自己学号命名的目录中！！**

1. **进程操作命令（上传命令及运行截图）**
2. 运行loop程序（实验1），使用ps命令查看该进程的详细信息并分析该进程的执行情况



1. 使用命令pstree查看loop进程所在的进程树，分析loop进程的父进程信息

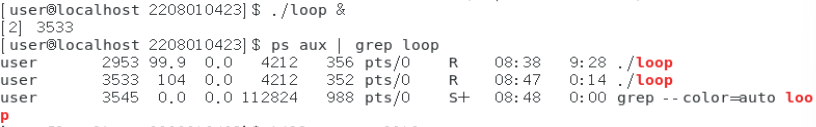
pstree

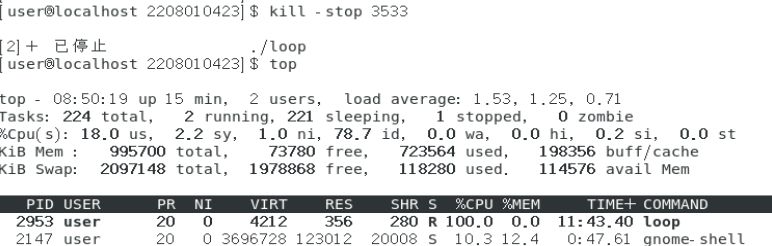


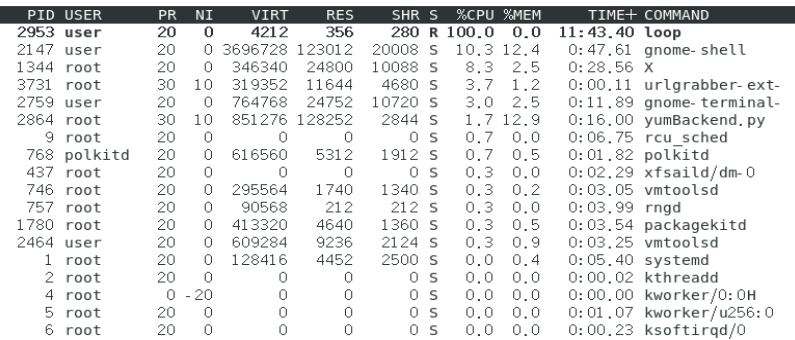
父进程信息



1. 再运行一个loop程序，并使用命令kill使该进程暂停，然后使用top命令查看自己Linux系统的进程情况，简要分析自己的系统的CPU和内存使用情况以及系统中的进程情况







**1, CPU使用情况分析 :**

18.0 us : 用户空间占用 CPU 的百分比（运行用户进程）。

2.2sy : 内核空间占用 CPU 的百分比（运行内核进程）。

1.0 ni : 以优先级为非负的用户进程占用的 CPU 百分比。

78.7id : 空闲 CPU 百分比（未被占用的时间）。

0.0 wa: 等待 I/O 的时间百分比（进程处于等待状态，等待某个 I/O 操作完成）。

0.0 hi: 硬中断的时间百分比。

0.2 si: 软件中断的时间百分比。

0.0 st: 虚拟机占用的时间百分比。

从 CPU 使用情况来看，系统的 CPU 利用率相对较低，绝大多数时间处于空闲状态（78.7%），这表明 CPU 负载并不高，虽然存在 2 个正在运行的进程，但大部分时间并没有在繁忙运行。

1. **内存使用情况**

**KiB Mem:**

总内存 (total): 995700 KiB

空闲内存 (free): 73780 KiB

已使用内存 (used): 723564 KiB

缓冲区/缓存 (buff/cache): 198356 KiB

内存使用情况显示出大部分内存已经被使用（723564 KiB），而可用内存相对较少（73780 KiB），但是也有缓存和缓冲区占用（198356 KiB）。

**KiB Swap:**

总交换内存 (total): 2097148 KiB

空闲交换内存 (free): 1978868 KiB

已使用交换内存 (used): 118280 KiB

可用内存 (avail Mem): 114576 KiB

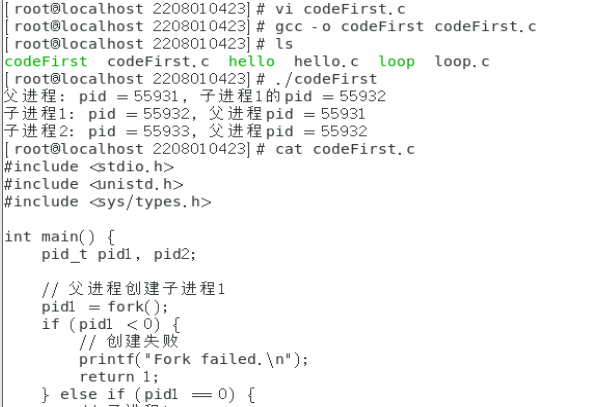
交换内存使用率相对较低（118280 KiB），这表明系统可以使用交换空间，但并不高，能够有效地处理内存压力。

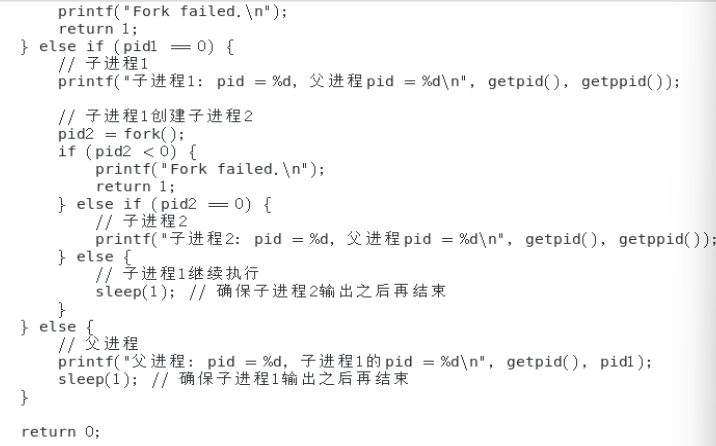
**3, 进程情况**

1. 在进程列表中，PID 2953（loop）的 CPU 使用率达到了 100%，这意味着这个进程正在进行大量计算或进入了死循环。其他进程的 CPU 使用率都相对较低。
2. gnome-shell（PID 2147）占用了约 12.4% 的内存，X（PID 1344）占用了约 2.5% 的内存，表明它们在正常使用内存，主要用于图形界面的渲染和处理。
3. **进程控制**

1.编写一段程序，使用系统调用fork( )创建两个子进程。

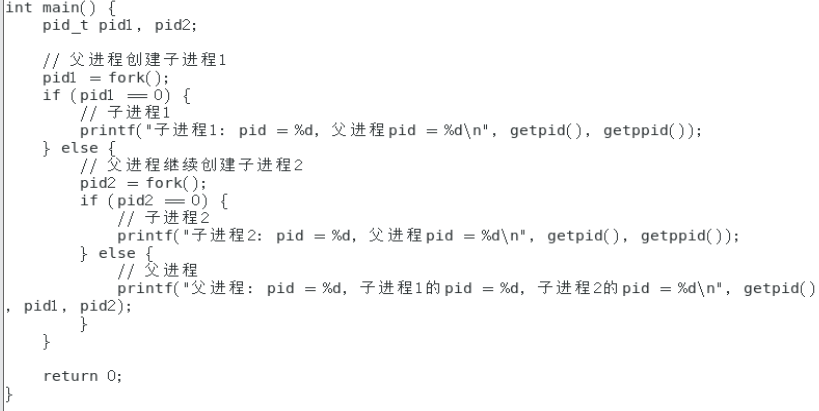
（1）父进程创建子进程1，子进程1创建子进程2，三个进程分别输出一行字符串，其中包含进程各自的pid和父进程id。





1. 父进程创建子进程1和子进程2，三个进程分别输出一行字符串，其中包含进程各自的pid和父进程id。





**求：两种情况的源代码截图和运行结果截图。**

**注意分析运行结果是否符合程序设计。**

2.运行以下程序，分析程序执行过程中产生的进程情况。

/\*forktree.c\*/

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

int main()

{

int p;

p=fork();

printf("fork 1\n");

if (p>0) /\*如果是父进程\*/

{

fork();

printf("fork 2\n"); /\*子进程和父进程重复执行\*/

}

else{ /\*如果是子进程\*/

fork();

printf("fork 3\n"); /\*子进程和孙进程重复执行\*/

fork(); /\*子进程和孙进程重复执行\*/

printf("fork 4\n"); /\*子进程、两个孙进程和曾孙进程重复执行\*/

}

sleep(20);

return 0；

}

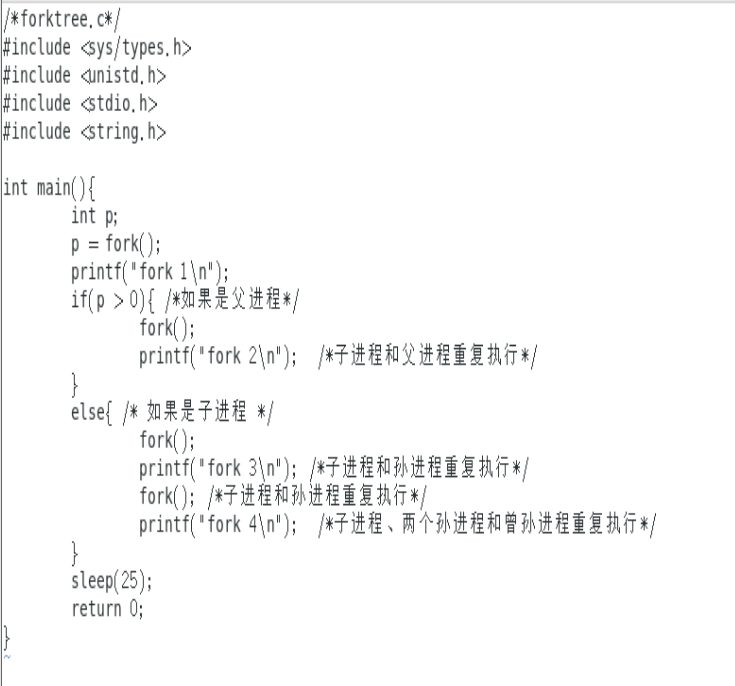
实验步骤：

* 编写上面的源代码生成源文件forktree.c
* 编译源文件生成可执行程序forktree：$gcc–o forktree forktree.c
* 后台运行：$./forktree &
* 查看进程树：$pstree –p pid （替换为forktree的进程id）或者用命令 ps axf查看几个进程的关联关系
* 分析：最后一个系统调用fork()会为什么被执行2次？最后一条printf输出语句为什么会被执行4次？

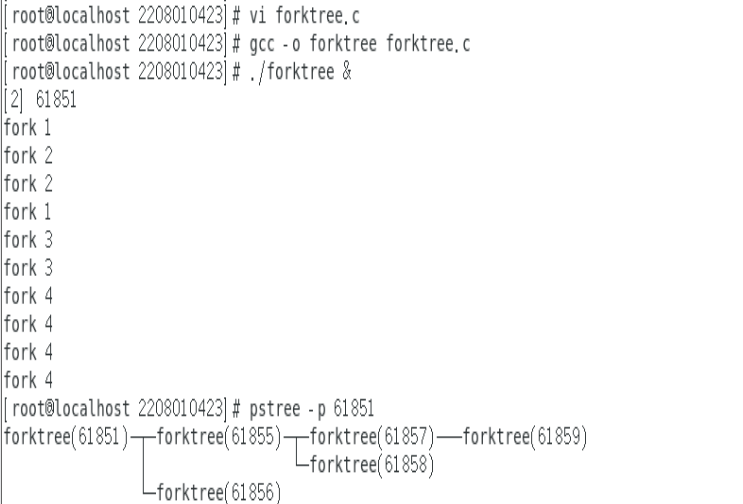
**要求：1.程序源代码和运行结果截图（注意截图要能看到forktree相关的进程树）**

1. **结果分析，并回答上述问题**

代码



运行结果



**分析**

**（一）**为什么最后一个 fork() 会执行两次？

第一个进程 fork 出来的子进程一会进入 else 语句。

进来之后会 fork 一次，假设产生了子进程二。

这样遇到最后一个 fork 的时候，子进程一和子进程二都会 fork 一次，即最后一个 fork 执行 2 次。此，最后一个 fork() 被执行两次：一次在孙进程中，一次在父进程中。

（二）最后一条printf输出语句为什么会被执行4次？

**父进程第一次 fork ，假设产生了 P1 进程，P1 进程会进入 else {} 语句。**

**P1 进入之后，fork 出来 P2，之后遇到下一条语句时，P1 fork 出 P3，P2 fork 出 P4，总共就是 4 个进程，所以最后一条 printf 语句执行了 4 次。**

3.（选做）运行程序，分析以下程序运行产生的进程树情况。

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main()

{

int p,i;

for( i=0;i<2;i++)

{ p=fork();

if(p==0) exit(0);

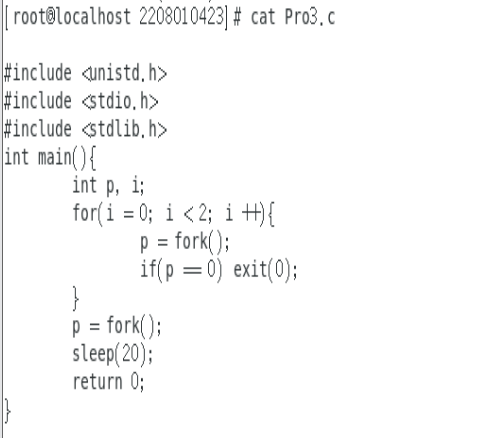
}

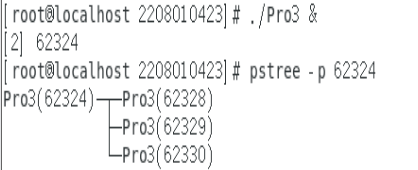
p=fork();

sleep(20);

return 0;

}





**结果分析 :**

**假设最开始的进程是 P0, 第一次循环 fork 出来 P1, 但是 P1 fork 的返回值是 0，被 exit 了。**

**第二次循环同理，fork 出来了 P2，但是 P2 exit 了。**

**退出循环的时候，P0 再 fork 出来 P3。**

**所以进程树就是第一个进程 P0 有三个子进程 P1, P2, P3。**

4.（选做）分析验证课堂中的实例创建的进程情况

int main(int argc, char\* argv[])

{

fork();

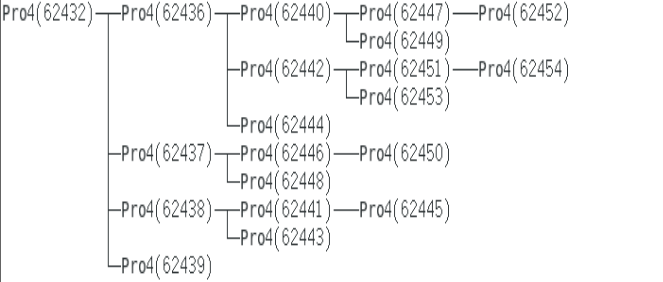
fork()&& fork()||fork();

fork();

sleep(20);

return 0;

}



**逻辑分析 :**

**对于最开始的 P0, 第一次 fork 出来 P1, 进入 fork()&&fork()||fork() 的时候，**

**P0 fork 一次得到 P2, P1 fork() 一次得到 P3。**

**执行下一个 fork（） 的时候，P0 继续 fork 出 P4, P1, 继续 fork 出 P5, P2和P3由于 && 的逻辑中断，不再执行 fork()&&fork()||fork() 中的第二条 fork 语句。**

**再分析 P0 和 P1，由于 || 的逻辑中断，不执行 fork()&&fork()||fork() 中的第三条语句。但是P2, P3, P4, P5 会继续 fork, 假设他们 fork 出来了P6, P7, P8, P9 这些进程。此时一共 10 个进程，来到最后一条语句时，这些进程还会全部 fork 一次。**