

## 一、实验目的与要求:

加深对进程的创建、运行、撤销过程的直观认识;  
掌握通过操作系统的用户接口 (命令行和系统函数) 控制进程状态的方法;  
了解多进程在多核处理机上的并发执行过程;

## 实验内容:

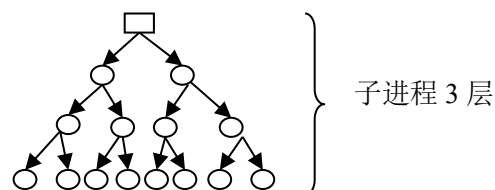
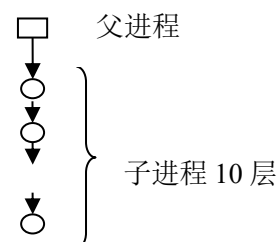
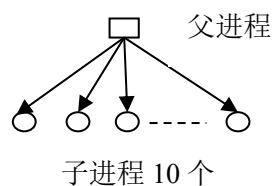
可以使用 Linux 或其它 Unix 类操作系统;  
学习该操作系统提供的命令行启动、撤销进程的方法;  
学习该操作系统提供的系统调用接口 (借助于库函数的形式间接调用) 启动和撤销进程;  
利用该操作系统提供的工具观测这些程序的并发执行过程以及状态转换过程。

## 二、方法、步骤: (说明程序相关的算法原理或知识内容, 程序设计的思路和方法, 可以预备部分:

学习 top、ps、pstree 和 kill 等命令的使用。能通过 top 和 ps j 命令查看进程号、父进程号、可执行文件名 (命令)、运行状态信息, 能通过 pstree 查看系统进程树; 能通过 kill 命令杀死制定 pid 的进程。

### 操作部分:

- 1) 编写 hello-loop.c 程序, 使用 gcc hello-loop.c -o helloworld 生成可执行文件 hello-loop。并在同一个目录下, 通过命令“./hello-loop”执行之。使用 top 和 ps 命令查看该进程, 记录进程号以及进程状态。
- 2) 使用 kill 命令终止 hello-loop 进程。
- 3) 使用 fork() 创建子进程, 形成以下父子关系:



并通过检查进程的 pid 和 ppid 证明你成功创建相应的父子关系并用 pstree 验证其关系。

- 4) 编写一个代码，使得进程循环处于以下状态 5 秒钟运行 5 秒钟阻塞（例如可以使用 sleep()），并使用 top 或 ps 命令检测其运行和阻塞两种状态，并截图记录；

实验报告要求：

- 1) 按学校统一格式
- 2) 需要给出具体命令和自行编写的程序的源代码
- 3) 需要给出实验操作的截图和必要的说明文字

附参考：

**fork()函数，Linux [系统调用](#)**

头文件：

```
#include <unistd.h>
```

函数定义：

```
int fork( void );
```

返回值：

子进程中返回 0，父进程中返回子进程 ID，出错返回 -1

函数说明：

一个现有进程可以调用 fork 函数创建一个新进程。由 fork 创建的新进程被称为子进程 (child process)。fork 函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回 0 值而父进程中返回子进程 ID。

子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本。注意，子进程持有的是上述存储空间的“副本”，这意味着父子进程间不共享这些存储空间，它们之间共享的存储空间只有代码段。

示例代码：

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char ** argv )
```

```
{
```

```
    int pid = fork();
```

```
    if(pid == -1 ) {
```

```
        // print("error!");
```

```
    } else if( pid == 0 ) {
```

```
        // print("This is the child process!");
```

```
    } else {
```

```
        // print("This is the parent process! child process id = %d", pid);
```

```
    }
```

```
    return 0;
```

```
}
```

三. 实验过程及内容: (对程序代码进行说明和分析, 越详细越好, 代码排版要整齐, 可读性要高)

用流程图表述, 程序主要数据结构的设计、主要函数之间的调用关系等)

第一周实验部分: 启动 Ubuntu 安装教学平台

安装依赖: `sudo apt-get install -y build-essential gd`

```
ljt@ljt: ~  
[sudo] ljt 的密码:  
E: 无法获得锁 /var/lib/dpkg/lock - open (11: 资源暂时不可用)  
E: 无法锁定管理目录(/var/lib/dpkg/), 是否有其他进程正占用它?  
ljt@ljt:~$ sudo rm /var/lib/dpkg/lock  
ljt@ljt:~$ sudo apt-get install -y build-essential gdb  
E: 无法获得锁 /var/lib/dpkg/lock - open (11: 资源暂时不可用)  
E: 无法锁定管理目录(/var/lib/dpkg/), 是否有其他进程正占用它?  
ljt@ljt:~$ sudo rm /var/lib/dpkg/lock  
ljt@ljt:~$ sudo apt-get install -y build-essential gdb  
正在读取软件包列表... 完成  
正在分析软件包的依赖关系树  
正在读取状态信息... 完成  
build-essential 已经是最新版 (12.1ubuntu2)。  
gdb 已经是最新版 (7.11.1-0ubuntu1~16.5)。  
下列软件包是自动安装的并且现在不需要了:  
  linux-headers-4.15.0-29 linux-headers-4.15.0-29-generic  
  linux-headers-4.15.0-39 linux-headers-4.15.0-39-generic  
  linux-image-4.15.0-29-generic linux-image-4.15.0-39-generic  
  linux-modules-4.15.0-29-generic linux-modules-4.15.0-39-generic  
  linux-modules-extra-4.15.0-29-generic linux-modules-extra-4.15.0-39-generic  
使用'sudo apt autoremove'来卸载它(它们)。  
升级了 0 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 174 个软件包未被升级。  
ljt@ljt:~$
```

`sudo apt-get install gcc-multilib`

```
ljt@ljt: ~  
ljt@ljt:~$ sudo apt-get install gcc-multilib  
正在读取软件包列表... 完成  
正在分析软件包的依赖关系树  
正在读取状态信息... 完成  
gcc-multilib 已经是最新版 (4:5.3.1-1ubuntu1)。  
下列软件包是自动安装的并且现在不需要了:  
  linux-headers-4.15.0-29 linux-headers-4.15.0-29-generic  
  linux-headers-4.15.0-39 linux-headers-4.15.0-39-generic  
  linux-image-4.15.0-29-generic linux-image-4.15.0-39-generic  
  linux-modules-4.15.0-29-generic linux-modules-4.15.0-39-generic  
  linux-modules-extra-4.15.0-29-generic linux-modules-extra-4.15.0-39-generic  
使用'sudo apt autoremove'来卸载它(它们)。  
升级了 0 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 171 个软件包未被升级。  
ljt@ljt:~$
```

使用 apt-get 安装 qemu

```
ljt@ljt: ~  
ljt@ljt:~$ sudo apt-get install qemu  
正在读取软件包列表... 完成  
正在分析软件包的依赖关系树  
正在读取状态信息... 完成  
qemu 已经是最新版 (1:2.5+dfsg-5ubuntu10.35)。  
下列软件包是自动安装的并且现在不需要了:  
  linux-headers-4.15.0-29 linux-headers-4.15.0-29-generic  
  linux-headers-4.15.0-39 linux-headers-4.15.0-39-generic  
  linux-image-4.15.0-29-generic linux-image-4.15.0-39-generic  
  linux-modules-4.15.0-29-generic linux-modules-4.15.0-39-generic  
  linux-modules-extra-4.15.0-29-generic linux-modules-extra-4.15.0-39-generic  
使用'sudo apt autoremove'来卸载它(它们)。  
升级了 0 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 171 个软件包未被升级。  
ljt@ljt:~$
```

安装 git:

```
ljt@ljt: ~  
ljt@ljt:~$ git clone git://github.com/mit-pdos/xv6-public.git  
程序“git”尚未安装。 您可以使用以下命令安装:  
sudo apt install git  
ljt@ljt:~$
```

```

ljt@ljt: ~
ljt@ljt:~$ sudo apt-get install git
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
git 已经是最新版 (1:2.7.4-0ubuntu1.6)。
下列软件包是自动安装的并且现在不需要了:
  linux-headers-4.15.0-29 linux-headers-4.15.0-29-generic
  linux-headers-4.15.0-39 linux-headers-4.15.0-39-generic
  linux-image-4.15.0-29-generic linux-image-4.15.0-39-generic
  linux-modules-4.15.0-29-generic linux-modules-4.15.0-39-generic
  linux-modules-extra-4.15.0-29-generic linux-modules-extra-4.15.0-39-generic
使用 'sudo apt autoremove' 来卸载它(它们)。
升级了 0 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 171 个软件包未被升级。
ljt@ljt:~$

```

从 github 上克隆下文件:

```

ljt@ljt:~$ git clone git://github.com/mit-pdos/xv6-public.git
正克隆到 'xv6-public'...
remote: Enumerating objects: 13974, done.
remote: Total 13974 (delta 0), reused 0 (delta 0), pack-reused 13974
接收对象中: 100% (13974/13974), 17.15 MiB | 961.00 KiB/s, 完成.
处理 delta 中: 100% (9535/9535), 完成.
检查连接... 完成。
ljt@ljt:~$

```

切换版本:

```

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ git checkout -b xv6-rev7 xv6-rev7
切换到一个新分支 'xv6-rev7'
ljt@ljt:~/xv6-public$

```

打开 Makefile 文件。找到如下这一行, 并添加 qemu 的可执行程序): QEMU = qemu-system-x86\_64

```

# If the makefile can't find QEMU, specify its path here
QEMU = qemu-system-x86_64

```

开始编译:

```

ljt@ljt: ~/xv6-public
ld -m elf_i386 -N -e main -Ttext 0 -o _zombie zombie.o ulib.o usys.o printf.o
umalloc.o
objdump -S _zombie > zombie.asm
objdump -t _zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > zombie.sym
./mkfs fs.img README_cat_echo_forktest_grep_init_kill_ln_ls_mkdir_rm_
sh_stressfs_usertests_wc_zombie
used 29 (bit 1 ninode 26) free 29 log 10 total 1024
ballocc: first 393 blocks have been allocated
ballocc: write bitmap block at sector 28
dd if=/dev/zero of=xv6.img count=10000
记录了10000+0 的读入
记录了10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0505081 s, 101 MB/s
dd if=bootblock of=xv6.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000245436 s, 2.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
记录了248+1 的读入
记录了248+1 的写出
127412 bytes (127 kB, 124 KiB) copied, 0.00109097 s, 117 MB/s
rm wc.o grep.o mkdir.o rm.o ln.o stressfs.o kill.o echo.o init.o usertests.o zom
bie.o cat.o sh.o ls.o
ljt@ljt:~/xv6-public$

```

结果如上，编译成功。接着试着运行 xv6 操作系统吧：

```

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ make qemu
qemu          qemu-gdb      qemu-memfs     qemu-nox      qemu-nox-gdb
ljt@ljt:~/xv6-public$ make qemu-nox
ljt@ljt:~/xv6-public
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
xv6...
cpu1: starting
cpu0: starting
init: starting sh
$ ls
.                1 1 512
..               1 1 512
README          2 2 1929
cat              2 3 9480
echo            2 4 9012
forktest        2 5 5832
grep             2 6 10600
init            2 7 9316
kill            2 8 9028
ln              2 9 8992
ls              2 10 10560
mkdir           2 11 9084
rm              2 12 9068
sh              2 13 16196
stressfs        2 14 9452
usertests       2 15 37524
wc              2 16 9800
zombie          2 17 8800
console         3 18 0
$

```

虽然产生了 warning 但是编译成功了。

预备部分：

学习 top、ps、pstree 和 kill 等命令的使用。能通过 top 和 ps j 命令查看进程号、父进程号、可执行文件名（命令）、运行状态信息，能通过 pstree 查看系统进程树；能通过 kill 命令杀死制定 pid 的进程。

1. 使用 top 查看进程：top

```
ljt@ljt: ~  
top - 20:03:37 up 21 min, 1 user, load average: 0.54, 0.90, 0.67  
Tasks: 328 total, 1 running, 211 sleeping, 0 stopped, 0 zombie  
%Cpu(s): 0.6 us, 0.1 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st  
KiB Mem : 7132420 total, 4399240 free, 1164252 used, 1568928 buff/cache  
KiB Swap: 998396 total, 998396 free, 0 used. 5556784 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1964	ljt	20	0	1734800	271036	95120	S	11.9	3.8	5:17.19	compiz
1051	root	20	0	691380	171548	71232	S	2.6	2.4	0:40.12	Xorg
3581	apt	20	0	52496	5492	5056	S	1.0	0.1	0:01.38	http
3682	ljt	20	0	49000	3920	3144	R	0.7	0.1	0:00.06	top
1679	ljt	20	0	418700	40724	23796	S	0.3	0.6	0:00.76	fcitx
2271	ljt	20	0	616944	45244	35512	S	0.3	0.6	0:02.50	gnome-term+
2832	ljt	20	0	1602596	112636	86656	S	0.3	1.6	0:01.95	WebExtensi+
1	root	20	0	185100	5696	3984	S	0.0	0.1	0:03.51	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:+
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_+
7	root	20	0	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
8	root	20	0	0	0	0	I	0.0	0.0	0:00.26	rcu_sched
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0

2. 使用 ps 查看进程:

使用 ps j 查看当前运行的进程: ps j

```
ljt@ljt: ~  
ljt@ljt:~$ ps j
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
2271	2276	2276	2276	pts/6	3576	Ss	1000	0:00	bash
2271	3656	3656	3656	pts/4	3685	Ss	1000	0:00	bash
3656	3685	3685	3656	pts/4	3685	R+	1000	0:00	ps j

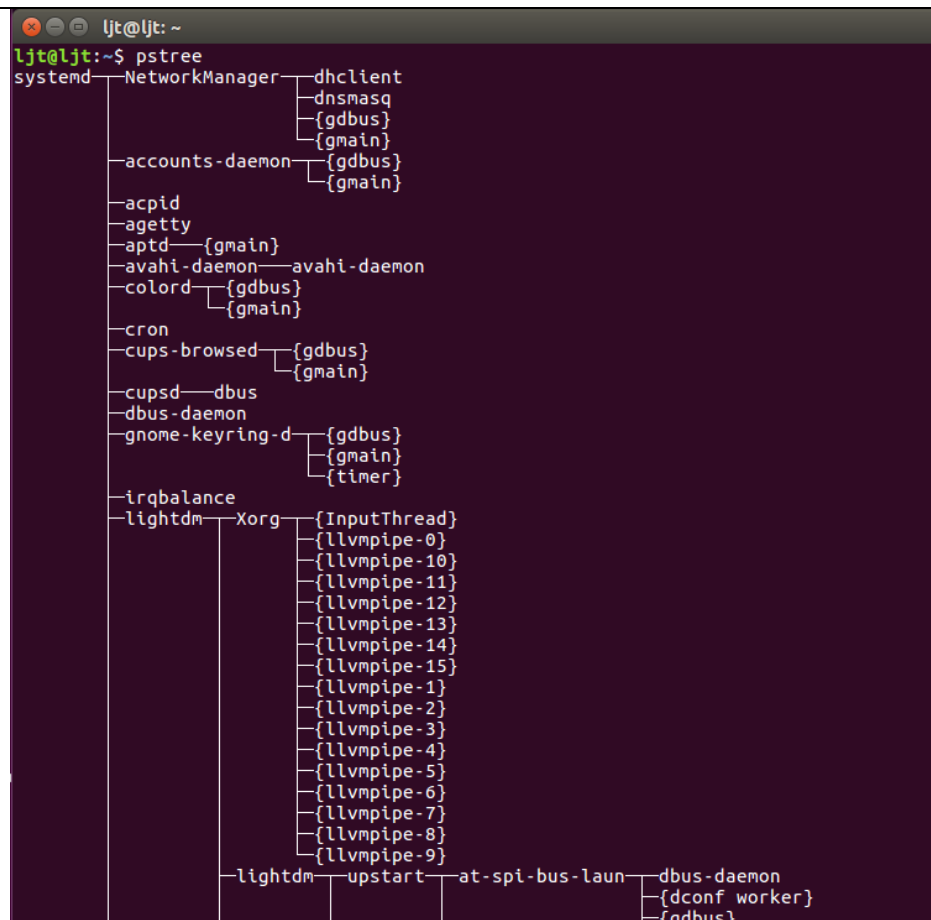
```
ljt@ljt:~$
```

使用 ps -aux 查看所有用户的进程: ps -aux

```
ljt@ljt: ~  
ljt@ljt:~$ ps -aux  
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND  
root         1  0.2  0.0 185100 5696 ?        Ss   19:42   0:03 /sbin/init splash  
root         2  0.0  0.0      0     0 ?        S    19:42   0:00 [kthreadd]  
root         4  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/0:0H]  
root         6  0.0  0.0      0     0 ?        I<   19:42   0:00 [mm_percpu_wq]  
root         7  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/0]  
root         8  0.0  0.0      0     0 ?        I    19:42   0:00 [rcu_sched]  
root         9  0.0  0.0      0     0 ?        I    19:42   0:00 [rcu_bh]  
root        10  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/0]  
root        11  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/0]  
root        12  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/0]  
root        13  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/1]  
root        14  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/1]  
root        15  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/1]  
root        16  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/1]  
root        18  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/1:0H]  
root        19  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/2]  
root        20  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/2]  
root        21  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/2]  
root        22  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/2]  
root        24  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/2:0H]  
root        25  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/3]  
root        26  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/3]  
root        27  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/3]  
root        28  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/3]  
root        30  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/3:0H]  
root        31  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/4]  
root        32  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/4]  
root        33  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/4]  
root        34  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/4]  
root        36  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/4:0H]  
root        37  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/5]  
root        38  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/5]  
root        39  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/5]  
root        40  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/5]  
root        42  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/5:0H]  
root        43  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/6]  
root        44  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/6]  
root        45  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/6]  
root        46  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/6]  
root        48  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/6:0H]  
root        49  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/7]  
root        50  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/7]  
root        51  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/7]  
root        52  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/7]  
root        54  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/7:0H]  
root        55  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/8]  
root        56  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/8]  
root        57  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/8]  
root        58  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/8]  
root        60  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/8:0H]  
root        61  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/9]  
root        62  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/9]  
root        63  0.0  0.0      0     0 ?        S    19:42   0:00 [migration/9]  
root        64  0.0  0.0      0     0 ?        S    19:42   0:00 [ksoftirqd/9]  
root        66  0.0  0.0      0     0 ?        I<   19:42   0:00 [kworker/9:0H]  
root        67  0.0  0.0      0     0 ?        S    19:42   0:00 [cpuhp/10]  
root        68  0.0  0.0      0     0 ?        S    19:42   0:00 [watchdog/10]
```

3. 使用 pstree 查看进程: pstree





操作部分：

- 5) 编写 hello-loop.c 程序, 使用 gcc hello-loop.c -o helloworld 生成可执行文件 hello-loop。并在同一个目录下, 通过命令“./hello-loop”执行之。使用 top 和 ps 命令查看该进程, 记录进程号以及进程状态。

编写 hello-loop.c 程序:

```
hello-loop.c (~/.xv6-public) - gedit
打开(O) [icon]
#include<stdio.h>
int main(){
    printf("Hello world!\n");
    while(1){}
    return 0;
}
```

编译 c 文件:



```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ gedit hello-loop.c
ljt@ljt:~/xv6-public$ gedit hello-loop.c
ljt@ljt:~/xv6-public$ gedit hello-loop.c
ljt@ljt:~/xv6-public$ gcc hello-loop.c -o hello-loop
ljt@ljt:~/xv6-public$ ls | grep hello-loop
hello-loop
hello-loop.c
ljt@ljt:~/xv6-public$
```

出现 hello-loop 文件，说明编译成功；开始运行它吧：

```
ljt@ljt:~/xv6-public$ ./hello-loop
Hello world!
```

显然她是进入了 while 循环中，再开启一个 terminal 查看情况：

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ps j
PPID  PID  PGID  SID  TTY          TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4      11390  Ss     1000    0:00  bash
11270 11357 11357 11357 pts/6      11397  Ss     1000    0:00  bash
11275 11390 11390 11275 pts/4      11390  R+     1000    1:45  ./hello-loop
11357 11397 11397 11357 pts/6      11397  R+     1000    0:00  ps j
```

显然可以看出 hello-loop 正在执行 while 循环，所以处于 R (running) 状态；该进程的进程号此时分配为 11390。

现在我们使用 top 查看命令：

```
ljt@ljt: ~/xv6-public
top - 20:30:00 up 47 min, 1 user, load average: 0.87, 0.72, 0.63
Tasks: 323 total, 2 running, 205 sleeping, 0 stopped, 0 zombie
%Cpu(s): 4.6 us, 0.1 sy, 0.0 ni, 95.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3808320 free, 1108684 used, 2215416 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used, 5612712 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 11390 ljt        20   0   4352    640   572  R   99.7   0.0   0:22.09 hello-loop
  1964 ljt        20   0 1742760 279552 95156 S    4.6   3.9 13:26.68 compiz
   1051 root       20   0 680860 174364 63008 S    0.7   2.4 1:45.95 Xorg
  11391 ljt        20   0   49000   3912   3168 R    0.7   0.1 0:00.12 top
  1784 ljt        20   0 593084 41344 31900 S    0.3   0.6 0:02.40 unity-pane+
  1880 ljt        20   0 1240492 17256 14696 S    0.3   0.2 0:00.17 indicator-+
    1 root       20   0 185180   5724  3984 S    0.0   0.1 0:04.36 systemd
    2 root       20   0      0      0      0 S    0.0   0.0 0:00.01 kthreadd
    4 root       0 -20      0      0      0 I    0.0   0.0 0:00.00 kworker/0:0+
    6 root       0 -20      0      0      0 I    0.0   0.0 0:00.00 mm_percpu_+
    7 root       20   0      0      0      0 S    0.0   0.0 0:00.00 ksoftirqd/0
    8 root       20   0      0      0      0 I    0.0   0.0 0:00.53 rcu_sched
    9 root       20   0      0      0      0 I    0.0   0.0 0:00.00 rcu_bh
   10 root       rt   0      0      0      0 S    0.0   0.0 0:00.00 migration/0
   11 root       rt   0      0      0      0 S    0.0   0.0 0:00.00 watchdog/0
   12 root       20   0      0      0      0 S    0.0   0.0 0:00.00 cpuhp/0
   13 root       20   0      0      0      0 S    0.0   0.0 0:00.00 cpuhp/1
```

可以看出 hello-loop 占用了大部分的 cpu 在执行；此时分配的进程号同样是 11390.进程状态 R (running)

6) 使用 kill 命令终止 hello-loop 进程。

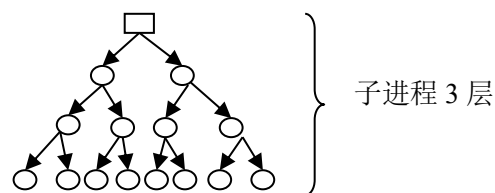
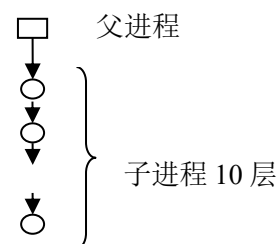
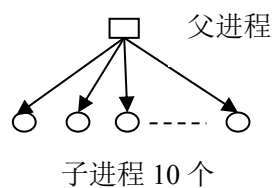
进程在上述中显示是正在运行，现在执行 kill 指令将其删除。

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ps j
  PPID   PID   PGID   SID TTY          TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4        11390  Ss     1000    0:00  bash
11270 11357 11357 11357 pts/6        11408  Ss     1000    0:00  bash
11275 11390 11390 11275 pts/4        11390  R+     1000    4:39  ./hello-loop
11357 11408 11408 11357 pts/6        11408  R+     1000    0:00  ps j
ljt@ljt:~/xv6-public$ kill -9 11390
ljt@ljt:~/xv6-public$ ps j
  PPID   PID   PGID   SID TTY          TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4        11275  Ss+    1000    0:00  bash
11270 11357 11357 11357 pts/6        11409  Ss     1000    0:00  bash
11357 11409 11409 11357 pts/6        11409  R+     1000    0:00  ps j
ljt@ljt:~/xv6-public$
```

在 kill 前后都查看一次进程，进程 11390 被 kill 了。回到前面可以看到：

```
ljt@ljt:~/xv6-public$ ./hello-loop
Hello world!
已杀死
ljt@ljt:~/xv6-public$
```

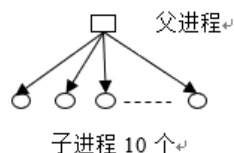
7) 使用 fork()创建子进程，形成以下父子关系：



并通过检查进程的 pid 和 ppid 证明你成功创建相应的父子关系并用 pstree 验证其关系。

\*\*\*\*\*

建立第一个父子关系：



编写代码 a:

```
fath-10son.c (~/.xv6-public) - gedit
打开(O) [icon]

#include <unistd.h>
#include <stdio.h>

int main(int argc, char ** argv )
{
    int i=0,pid=100;
    for(i=0;i<10;i++){
        if(pid!=0)
            pid = fork();
        if(pid == -1 ) {
            printf("error!\n");
        } else if( pid ==0 ) {
            printf("This is the child process!\n"); break;
        } else {
            printf("This is the parent process! child process id = %d\n", pid);
        }
    }
    while(1){}
    return 0;
}
```

```
ljt@ljt: ~/.xv6-public
ljt@ljt: ~/.xv6-public x ljt@ljt: ~/.xv6-public x + v
ljt@ljt:~/.xv6-public$ gcc fath-10son.c -o fath-10son
ljt@ljt:~/.xv6-public$ ./fath-10son
```

编译成功!

```
ljt@ljt: ~/.xv6-public
ljt@ljt: ~/.xv6-public x ljt@ljt: ~/.xv6-public x + v
ljt@ljt:~/.xv6-public$ ./fath-10son
This is the parent process! child process id = 12105
This is the child process!
This is the parent process! child process id = 12106
This is the child process!
This is the parent process! child process id = 12107
This is the child process!
This is the parent process! child process id = 12108
This is the child process!
This is the parent process! child process id = 12109
This is the parent process! child process id = 12110
This is the child process!
This is the child process!
This is the parent process! child process id = 12111
This is the child process!
This is the parent process! child process id = 12112
This is the child process!
This is the parent process! child process id = 12113
This is the child process!
This is the parent process! child process id = 12114
This is the child process!
```

查看输出可知，总共创建 10 个子进程；并最后在 while 循环中卡住，在使用 psj 或者 top 或是 pstree 查看进程情况；

```

ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ps j
  PPID   PID   PGID   SID  TTY      TPGID  STAT   UID    TIME  COMMAND
11270   11275 11275 11275 pts/4    12104  Ss     1000    0:00  bash
11270   11357 11357 11357 pts/6    12119  Ss     1000    0:00  bash
11275   12104 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12105 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12106 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12107 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12108 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12109 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12110 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12111 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12112 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12113 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
12104   12114 12104 11275 pts/4    12104  R+     1000    0:26  ./fath-10son
11357   12119 12119 11357 pts/6    12119  R+     1000    0:00  ps j
ljt@ljt:~/xv6-public$

```

可以看出这里有 11 个进程是 fath-10son 的，其中 PPID 是 12104 的共有 10 个，基本可以确定，是 12104 产生了 10 个子进程。

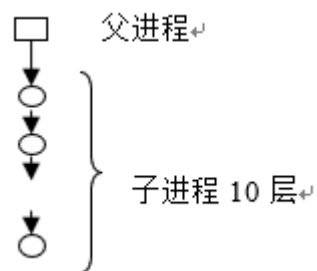
使用 pstree 查看进程:

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ pstree | grep fath-10son
ljt@ljt:~/xv6-public$
```

可以看出 fath10son 的父进程下还有 10 个子进程。

显然创建 10 个子进程成功:

\*\*\*\*\*



### 创建第二个父子关系:

编写代码 b:

```
fath_str_son.c (~/.xv6-public) - gedit
打开(O)  [icon]

#include <unistd.h>
#include <stdio.h>

int main(int argc, char ** argv )
{
    int i=0,pid=0;
    pid = fork();
    for(i=0;i<9;i++){
        if(pid == -1 ) {
            printf("error!\n");
        } else if( pid ==0 ) {
            pid=fork();
            printf("This is the child process!PID:%d\n",pid);
        } else {
            printf("This is the parent process! child process id = %d\n", pid);
            break;
        }
    }
    if(pid!=0&&pid!=-1){
        printf("This is the parent process! child process id = %d\n", pid);
    }
    while(1){}
return 0;
}

#include <unistd.h>
#include <stdio.h>

int main(int argc, char ** argv )
{
    int i=0,pid=100;
    for(i=0;i<10;i++){
        if(pid!=0)
            pid = fork();
        if(pid == -1 ) {
            printf("error!\n");
        } else if( pid ==0 ) {
            printf("This is the child process!\n"); break;
        } else {
            printf("This is the parent process! child process id = %d\n", pid);
        }
    }
    while(1){}
return 0;
}

编译代码:

ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public$ gcc fath_str_son.c -o fath_str_son
ljt@ljt: ~/xv6-public$
```

编译成功:

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./fath_str_son
This is the parent process! child process id = 12451
This is the parent process! child process id = 12451
This is the child process!PID:12452
This is the parent process! child process id = 12452
This is the parent process! child process id = 12452
This is the child process!PID:0
This is the child process!PID:12453
This is the parent process! child process id = 12453
This is the parent process! child process id = 12453
This is the child process!PID:0
This is the child process!PID:12454
This is the parent process! child process id = 12454
This is the parent process! child process id = 12454
This is the child process!PID:0
This is the child process!PID:12455
This is the parent process! child process id = 12455
This is the parent process! child process id = 12455
This is the child process!PID:0
This is the child process!PID:12456
This is the parent process! child process id = 12456
This is the parent process! child process id = 12456
This is the child process!PID:0
This is the child process!PID:12457
This is the parent process! child process id = 12457
This is the parent process! child process id = 12457
This is the child process!PID:0
This is the child process!PID:12458
This is the parent process! child process id = 12458
This is the parent process! child process id = 12458
This is the child process!PID:0
This is the child process!PID:12459
This is the parent process! child process id = 12459
This is the parent process! child process id = 12459
This is the child process!PID:0
This is the child process!PID:12460
This is the parent process! child process id = 12460
This is the child process!PID:0
```

让其在 while 循环中不断运行，我们来查看他的进程状况和进程号的对应关系；如下图所示：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ps j
PPID  PID  PGID  SID  TTY      TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4    12450  Ss     1000   0:00  bash
11270 11357 11357 11357 pts/6    12461  Ss     1000   0:00  bash
11275 12450 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12450 12451 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12451 12452 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12452 12453 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12453 12454 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12454 12455 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12455 12456 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12456 12457 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12457 12458 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12458 12459 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12459 12460 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
11357 12461 12461 11357 pts/6    12461  R+     1000   0:00  ps j
ljt@ljt:~/xv6-public$
```

可以看出每一行的 PID 是下一行的 PPID，即上一个进程是下一个进程的父进程，达到了如上图所示的父子关系；我们再使用 pstree 查看状况：命令：pstree -apnh

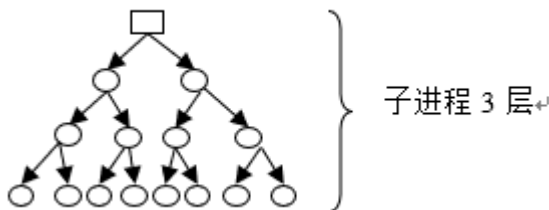
```

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ pstree -apnh | grep fath_str_son
      -fath_str_son,12450
        -fath_str_son,12451
          -fath_str_son,12452
            -fath_str_son,12453
              -fath_str_son,12454
                -fath_str_son,12455
                  -fath_str_son,12456
                    -fath_str_son,12457
                      -fath_str_son,12458
                        -fath_str_son,12459
                          -fath_str_son,12460
ljt@ljt:~/xv6-public$

```

明显可以看出其中的父子关系，进程 12450 到 12460 前者是后者的父进程。  
完成!

\*\*\*\*\*



创建第三个父子关系:

编写代码 c.c:

```

fath-10son.c      x      fath_str_son.c      x      fath_tree_son.c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char ** argv )
{
    int i=0,pid=0;
    for(i=0;i<3;i++){
        if(pid == -1 ) {
            printf("error!\n");
        } else if( pid == 0 ) {
            pid=fork();
            if(pid!=0){
                pid=fork();
            }
            printf("This is the child process!PID:%d\n",pid);
        } else {
            printf("This is the parent process! child process id = %d\n", pid);
            break;
        }
    }
    if(pid!=0&&pid!=-1){
        printf("This is the parent process! child process id = %d\n", pid);
    }
    while(1){}
    return 0;
}

```

#include <unistd.h>

#include <stdio.h>

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

{

int i=0,pid=0;

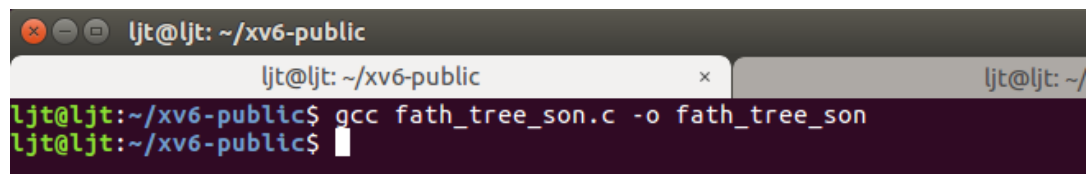
pid = fork();

for(i=0;i<9;i++){



```
    if(pid == -1 ) {  
        printf("error!\n");  
    } else if( pid ==0 ) {  
        pid=fork();  
        printf("This is the child process!PID:%d\n",pid);  
    } else {  
        printf("This is the parent process! child process id = %d\n", pid);  
        break;  
    }  
}  
}  
if(pid!=0&&pid!=-1){  
    printf("This is the parent process! child process id = %d\n", pid);  
}  
}  
while(1){  
return 0;  
}
```

编写完成，编译：

A terminal window with a dark background. The title bar shows 'ljt@ljt: ~/xv6-public'. There are two tabs: 'ljt@ljt: ~/xv6-public' and 'ljt@ljt: ~/'. The terminal shows the command 'gcc fath\_tree\_son.c -o fath\_tree\_son' being executed. The prompt 'ljt@ljt:~/xv6-public\$' is visible twice, once before and once after the command.

```
ljt@ljt: ~/xv6-public  
ljt@ljt:~/xv6-public$ gcc fath_tree_son.c -o fath_tree_son  
ljt@ljt:~/xv6-public$
```

编译成功!

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ gcc fath_tree_son.c -o fath_tree_son
ljt@ljt:~/xv6-public$ ./fath_tree_son
This is the child process!PID:13548
This is the parent process! child process id = 13548
This is the parent process! child process id = 13548
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:13551
This is the child process!PID:0
This is the parent process! child process id = 13551
This is the parent process! child process id = 13551
This is the child process!PID:13552
This is the parent process! child process id = 13552
This is the parent process! child process id = 13552
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:13556
This is the parent process! child process id = 13556
This is the child process!PID:0
This is the child process!PID:13558
This is the child process!PID:13559
This is the child process!PID:0
This is the parent process! child process id = 13558
This is the parent process! child process id = 13559
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:13560
This is the parent process! child process id = 13560
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:0
This is the child process!PID:0
```

同样的是再 while 循环中继续停顿，我们使用 pstree 查看：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ pstree -apnh | grep fath_tree_son
`-fath_tree_son,13546
    |-fath_tree_son,13547
        |-fath_tree_son,13549
            |-fath_tree_son,13553
            `--fath_tree_son,13556
        `--fath_tree_son,13551
            |-fath_tree_son,13555
            `--fath_tree_son,13559
    `--fath_tree_son,13548
        |-fath_tree_son,13550
            |-fath_tree_son,13554
            `--fath_tree_son,13558
        `--fath_tree_son,13552
            |-fath_tree_son,13557
            `--fath_tree_son,13560
    `--grep,13562 --color=auto fath_tree_son
```

显然符合我们之前的图示倒过来的树结构；

再在 ps j 中查看我们的程序

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public x ljt@ljt: ~/x
ljt@ljt:~/xv6-public$ ps j
PPID  PID  PGID  SID  TTY      TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4    13546  Ss     1000   0:00  bash
11270 11357 11357 11357 pts/6    13571  Ss     1000   0:00  bash
11275 13546 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13546 13547 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13546 13548 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13547 13549 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13548 13550 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13547 13551 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13548 13552 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13549 13553 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13550 13554 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13551 13555 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13549 13556 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13552 13557 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13550 13558 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13551 13559 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
13552 13560 13546 11275 pts/4    13546  R+     1000   1:52  ./fath_tree_son
11357 13571 13571 11357 pts/6    13571  R+     1000   0:00  ps j
ljt@ljt:~/xv6-public$
```

此图示没有 pstree 看上去直观，但是同样可以找出其中进程的父亲子关系。成功。

\*\*\*\*\*

- 8) 编写一个代码，使得进程循环处于以下状态 5 秒钟运行 5 秒钟阻塞（例如可以使用 sleep()），并使用 top 或 ps 命令检测其运行和阻塞两种状态，并截图记录；

编写代码 d.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char ** argv )
{
    clock_t begin,end;
    double cost=0;
    begin=clock();
    printf("it need 10 second to running!\n");
    while(cost<10){
        cost=(double)(clock()-begin)/CLOCKS_PER_SEC;
    }
    printf("begin to sleep!\n");
    sleep(10);
    printf("i'm wake up !\n");
    while(1){}
    return 0;
} |
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char ** argv )
```

```
{
```

```
    int i=0,pid=0;
```

```
    for(i=0;i<3;i++){
```

```

        if(pid == -1 ) {
            printf("error!\n");
        } else if( pid ==0 ) {
            pid=fork();
            if(pid!=0){
                pid=fork();
            }
            printf("This is the child process!PID:%d\n",pid);
        } else {
            printf("This is the parent process! child process id = %d\n", pid);
            break;
        }
    }
    if(pid!=0&&pid!=-1){
        printf("This is the parent process! child process id = %d\n", pid);
    }
    while(1){}
return 0;
}

```

代码如上所示，在计算完 10 秒钟之后，退出 while 循环。在这个 while 循环中，程序状态处于 R (running)，退出 while 循环之后程序进入 sleep 状态，进程状态转变为 S (sleep)。

```

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ gcc using_sleep.c -o using_sleep
ljt@ljt:~/xv6-public$

```

代码编译通过  
运行结果如下：

```

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!

ljt@ljt: ~/xv6-public
top - 22:30:58 up 2:48, 1 user, load average: 0.70, 1.34, 5.11
Tasks: 324 total, 3 running, 207 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.6 us, 3.5 sy, 0.0 ni, 94.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3233928 free, 1235772 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 5479436 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
14058 ljt         20   0   4352    644   576  R 100.0   0.0   0:03.04  using_sleep
1964  ljt         20   0 1772492 309108 95860  S  16.2   4.3  36:12.67  compiz
1051  root        20   0 685940 177992 63148  S   2.3   2.5   4:24.62  Xorg
13634 ljt         20   0 49000    3864  3100  R   0.7   0.1   0:04.77  top
2832  ljt         20   0 1614884 126552 86656  S   0.3   1.8   0:13.27  WebExtensions
11270 ljt         20   0 618676 47572  36028  S   0.3   0.7   0:27.33  gnome-terminal-
1    root        20   0 185180    5724  3984  S   0.0   0.1   0:04.38  systemd
2    root        20   0      0      0      0  S   0.0   0.0   0:00.01  kthreadd

```

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!
begin to sleep!
█

ljt@ljt: ~/xv6-public

top - 22:31:07 up 2:48, 1 user, load average: 0.67, 1.31, 5.06
Tasks: 324 total, 1 running, 208 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.1 us, 1.1 sy, 0.0 ni, 97.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3234644 free, 1235056 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 5480156 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 14058 ljt        20   0   4352    644   576  S   30.5   0.0    0:10.00 using_sleep
  1964 ljt        20   0 1772492 309108 95860 S   24.5   4.3   36:14.28 compiz
  1051 root       20   0 685940 177992 63148 S    3.0   2.5    4:24.83 Xorg
  1988 ljt        20   0 1094988 75380 55504 S    0.7   1.1    0:13.10 nautilus
 11270 ljt        20   0 618676 47572 36028 S    0.3   0.7    0:27.36 gnome-terminal-
13634 ljt        20   0  49000   3864  3100 R    0.3   0.1    0:04.81 top
    1 root       20   0 185180   5724  3984 S    0.0   0.1    0:04.38 systemd
    2 root       20   0      0      0      0  S    0.0   0.0    0:00.01 kthreadd

ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!
begin to sleep!
i'm wake up !

ljt@ljt: ~/xv6-public

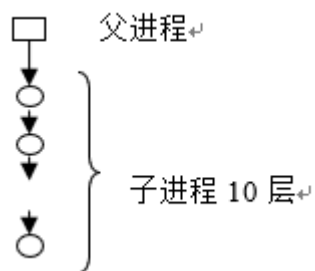
top - 22:31:16 up 2:48, 1 user, load average: 0.62, 1.29, 5.04
Tasks: 324 total, 2 running, 207 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.1 us, 0.1 sy, 0.0 ni, 97.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3234200 free, 1235500 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 5479712 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 14058 ljt        20   0   4352    644   576  R   38.4   0.0    0:11.16 using_sleep
  1964 ljt        20   0 1772492 309108 95860 S    9.6   4.3   36:14.85 compiz
  1051 root       20   0 685940 177992 63148 S    2.0   2.5    4:24.95 Xorg
  1988 ljt        20   0 1094988 75380 55504 S    0.7   1.1    0:13.13 nautilus
   163 root       20   0      0      0      0  I    0.3   0.0    0:03.62 kworker/6:1
 11270 ljt        20   0 618676 47572 36028 S    0.3   0.7    0:27.39 gnome-terminal-
13634 ljt        20   0  49000   3864  3100 R    0.3   0.1    0:04.86 top
    1 root       20   0 185180   5724  3984 S    0.0   0.1    0:04.38 systemd
```

可以从上上张图看出，每过一个状态，top 中的 using\_sleep 进程就会发生变化。主要看其中 S 的部分，当程序运行到 while 循环时候，时间不够 10 秒钟，他便一直在执行 while 循环，top 中显示的状态为 R (running)，当过了十秒钟之后，程序进入 sleep，top 中显示的状态为 S (sleep)。







创建第二个父子关系：

编译代码：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public$ gcc fath_str_son.c -o fath_str_son
ljt@ljt: ~/xv6-public$
```

编译成功：

让其在 while 循环中不断运行，我们来查看他的进程状况和进程号的对应关系；如下图所示：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public$ ps j
PPID  PID  PGID  SID  TTY      TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4    12450  Ss     1000   0:00  bash
11270 11357 11357 11357 pts/6    12461  Ss     1000   0:00  bash
11275 12450 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12450 12451 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12451 12452 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12452 12453 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12453 12454 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12454 12455 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12455 12456 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12456 12457 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12457 12458 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12458 12459 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
12459 12460 12450 11275 pts/4    12450  R+     1000   0:18  ./fath_str_son
11357 12461 12461 11357 pts/6    12461  R+     1000   0:00  ps j
ljt@ljt: ~/xv6-public$
```

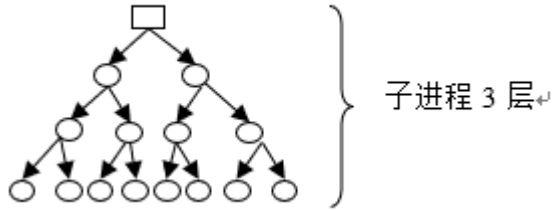
可以看出每一行的 PID 是下一行的 PPID，即上一个进程是下一个进程的父进程，达到了如上图所示的父子关系；我们再使用 pstree 查看状况：命令：pstree -apnh

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public$ pstree -apnh | grep fath_str_son
|--fath_str_son,12450
|   |--fath_str_son,12451
|   |   |--fath_str_son,12452
|   |   |   |--fath_str_son,12453
|   |   |   |   |--fath_str_son,12454
|   |   |   |   |--fath_str_son,12455
|   |   |   |   |--fath_str_son,12456
|   |   |   |   |--fath_str_son,12457
|   |   |   |   |--fath_str_son,12458
|   |   |   |   |--fath_str_son,12459
|   |   |   |   |--fath_str_son,12460
|   |--grep,12491 --color=auto fath_str_son
ljt@ljt: ~/xv6-public$
```

明显可以看出其中的父子关系，进程 12450 到 12460 前者是后者的父进程。完成！



\*\*\*\*\*



创建第三个父子关系：

编写完成，编译：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ gcc fath_tree_son.c -o fath_tree_son
ljt@ljt:~/xv6-public$
```

编译成功！

同样的是再 while 循环中继续停顿，我们使用 pstree 查看：

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ pstree -apnh | grep fath_tree_son
      -fath_tree_son,13546
        -fath_tree_son,13547
          -fath_tree_son,13549
            -fath_tree_son,13553
              -fath_tree_son,13556
                -fath_tree_son,13551
                  -fath_tree_son,13555
                    -fath_tree_son,13559
                      -fath_tree_son,13548
                        -fath_tree_son,13550
                          -fath_tree_son,13554
                            -fath_tree_son,13558
                              -fath_tree_son,13552
                                -fath_tree_son,13557
                                  -fath_tree_son,13560
                                    -grep,13562 --color=auto fath_tree_son
ljt@ljt:~/xv6-public$
```

显然符合我们之前的图示倒过来的树结构；

再在 ps j 中查看我们的程序

```
ljt@ljt: ~/xv6-public
ljt@ljt: ~/xv6-public x ljt@ljt: ~/x
ljt@ljt:~/xv6-public$ ps j
PPID  PID  PGID  SID  TTY      TPGID  STAT   UID    TIME  COMMAND
11270 11275 11275 11275 pts/4    13546  Ss      1000    0:00  bash
11270 11357 11357 11357 pts/6    13571  Ss      1000    0:00  bash
11275 13546 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13546 13547 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13546 13548 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13547 13549 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13548 13550 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13547 13551 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13548 13552 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13549 13553 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13550 13554 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13551 13555 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13549 13556 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13552 13557 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13550 13558 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13551 13559 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
13552 13560 13546 11275 pts/4    13546  R+      1000    1:52  ./fath_tree_son
11357 13571 13571 11357 pts/6    13571  R+      1000    0:00  ps j
ljt@ljt:~/xv6-public$
```

此图示没有 pstree 看上去直观，但是同样可以找出其中进程的父亲关系。  
成功。

\*\*\*\*\*

1) 编写一个代码，使得进程循环处于以下状态 5 秒钟运行 5 秒钟阻塞（例如可以使用 sleep()），并使用 top 或 ps 命令检测其运行和阻塞两种状态，并截图记录；在计算完 10 秒钟之后，退出 while 循环。在这个 while 循环中，程序状态处于 R (running)，退出 while 循环之后程序进入 sleep 状态，进程状态转变为 S (sleep)。

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ gcc using_sleep.c -o using_sleep
ljt@ljt:~/xv6-public$
```

代码编译通过  
运行结果如下：

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!

ljt@ljt: ~/xv6-public
top - 22:30:58 up 2:48, 1 user, load average: 0.70, 1.34, 5.11
Tasks: 324 total, 3 running, 207 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.6 us, 3.5 sy, 0.0 ni, 94.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3233928 free, 1235772 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 5479436 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
14058 ljt         20   0   4352   644   576 R 100.0   0.0   0:03.04 using_sleep
 1964 ljt         20   0 1772492 309108 95860 S 16.2   4.3 36:12.67 compiz
 1051 root        20   0 685940 177992 63148 S  2.3   2.5  4:24.62 Xorg
13634 ljt         20   0 49000   3864   3100 R  0.7   0.1   0:04.77 top
 2832 ljt         20   0 1614884 126552 86656 S  0.3   1.8   0:13.27 WebExtensions
11270 ljt         20   0 618676 47572 36028 S  0.3   0.7   0:27.33 gnome-terminal-
    1 root        20   0 185180   5724   3984 S  0.0   0.1   0:04.38 systemd
    2 root        20   0      0      0      0 S  0.0   0.0   0:00.01 kthreadd
```

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!
begin to sleep!

ljt@ljt: ~/xv6-public
top - 22:31:07 up 2:48, 1 user, load average: 0.67, 1.31, 5.06
Tasks: 324 total, 1 running, 208 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.1 us, 1.1 sy, 0.0 ni, 97.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3234644 free, 1235056 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 5480156 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
14058 ljt         20   0   4352   644   576 S 30.5   0.0   0:10.00 using_sleep
 1964 ljt         20   0 1772492 309108 95860 S 24.5   4.3 36:14.28 compiz
 1051 root        20   0 685940 177992 63148 S  3.0   2.5  4:24.83 Xorg
 1988 ljt         20   0 1094988 75380 55504 S  0.7   1.1   0:13.10 nautilus
11270 ljt         20   0 618676 47572 36028 S  0.3   0.7   0:27.36 gnome-terminal-
13634 ljt         20   0 49000   3864   3100 R  0.3   0.1   0:04.81 top
    1 root        20   0 185180   5724   3984 S  0.0   0.1   0:04.38 systemd
    2 root        20   0      0      0      0 S  0.0   0.0   0:00.01 kthreadd
```

```
ljt@ljt: ~/xv6-public
ljt@ljt:~/xv6-public$ ./using_sleep
it need 10 second to running!
begin to sleep!
i'm wake up !

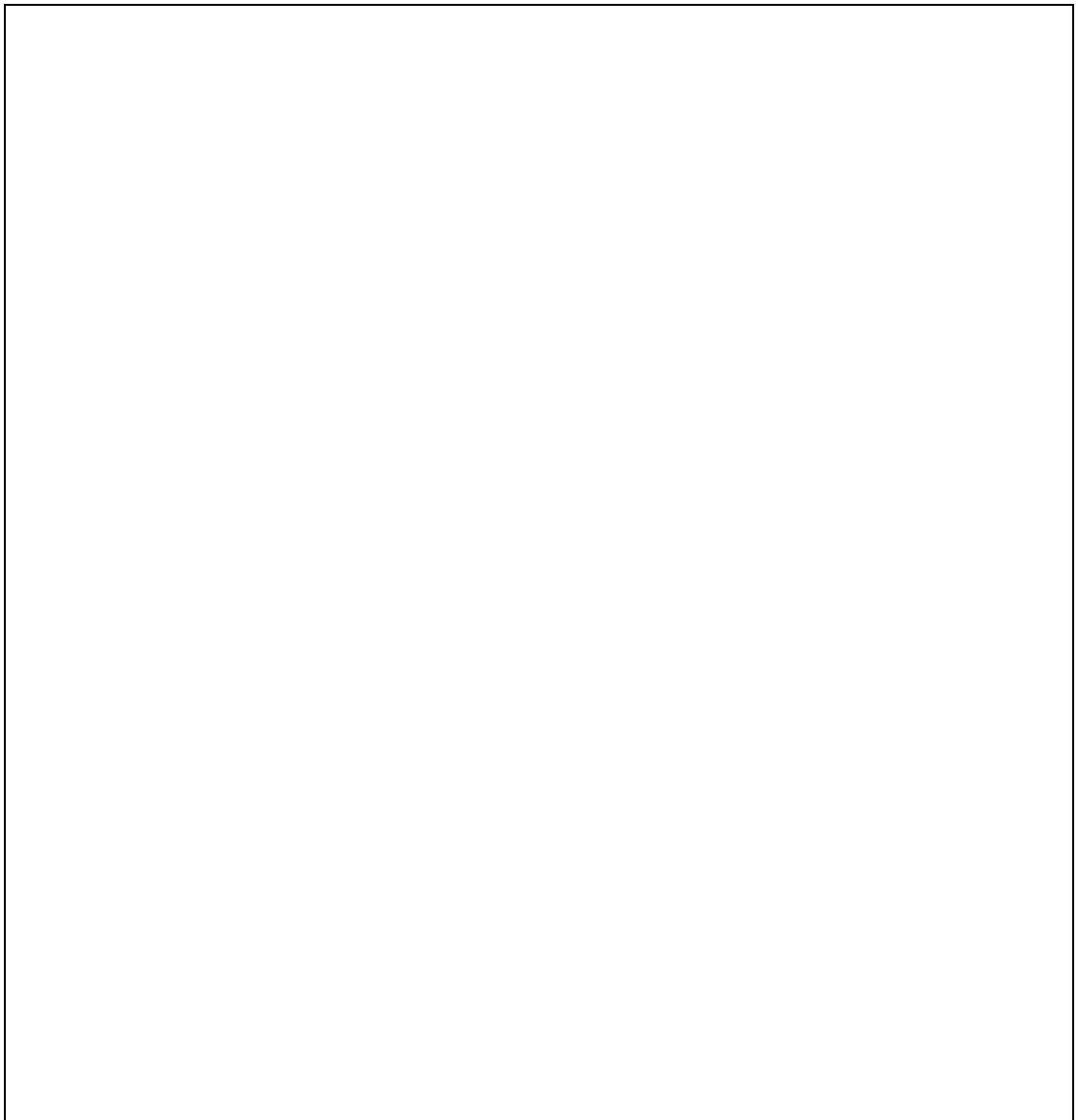
ljt@ljt: ~/xv6-public
top - 22:31:16 up 2:48, 1 user, load average: 0.62, 1.29, 5.04
Tasks: 324 total, 2 running, 207 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.1 us, 0.1 sy, 0.0 ni, 97.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7132420 total, 3234200 free, 1235500 used, 2662720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used, 5479712 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 14058 ljt        20   0    4352     644     576 R   38.4   0.0   0:11.16 using_sleep
  1964 ljt        20   0 1772492 309108 95860 S    9.6   4.3 36:14.85 compiz
  1051 root       20   0 685940 177992 63148 S    2.0   2.5  4:24.95 Xorg
  1988 ljt        20   0 1094988 75380 55504 S    0.7   1.1  0:13.13 nautilus
    163 root       20   0      0      0      0 I    0.3   0.0  0:03.62 kworker/6:1
 11270 ljt        20   0 618676 47572 36028 S    0.3   0.7  0:27.39 gnome-terminal-
 13634 ljt        20   0   49000   3864   3100 R    0.3   0.1  0:04.86 top
      1 root       20   0 185180 5724 3984 S    0.0   0.1  0:04.38 systemd
```

可以从上上张图看出，每过一个状态，top 中的 using\_sleep 进程就会发生变化。主要看其中 S 的部分，当程序运行到 while 循环时候，时间不够 10 秒钟，他便一直在执行 while 循环，top 中显示的状态为 R (running)，当过了十秒钟之后，程序进入 sleep，top 中显示的状态为 S (sleep)。

#### 五、实验体会：（根据自己情况填写）

通过本次实验，对 linux 操作系统有进一步的了解，特别是对操作系统中进程的查看，进程号的查找还有进程的创建和删除有了更深的理解，实际的操作对理论知识的学习有非常好的指导，同时，本次实验也学习到了创建不同父子关系的进程，一个父进程创建多个子进程，一个父进程创建子进程，子进程再创建子子进程，循环创建十个进程。和创建树状结构的父子进程关系，这对进程的创建和管理有更好的理解。将进程挂起和执行，在对其查看 top 中的进程状态，直观的明白了 top 中的 stat。



指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：