

**数学与信息学院学生实验报告**

**实验课程名称：** 操作系统 **教师：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **操作系统用户接口实验** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** | **078** | **年级专业班级** |  |
| **小组成员** |  | | | **实验日期** | **2017 年 9 月 18日** |

**一、实验目的**

熟悉操作系统的命令行、图形、程序接口

**二、实验软硬件环境**

软件：VMware 操作系统：ubuntu 硬件：PC

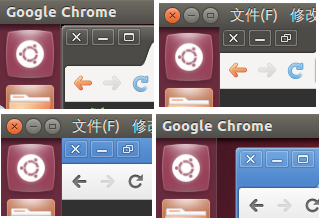
**三、实验内容及要求**

1. 安装操作系统（选做）
2. 熟悉操作系统的命令行、图形、程序接口

**四、实验记录（按教师需求写出相应的实验步骤、流程图、程序代码、运行结果及分析等项目)**

一、操作系统的图形用户接口

1.ubuntu-unity用户界面



2.ubuntu-unity用户界面发展史

### 出生于特殊时期

Unity 用户界面的最早出现，是在 2010 年的 5 月份，当时 Ubuntu 的母公司 Canonical 创始人 Mark Shuttleworth 发表博文《[Unity, and Ubuntu Light](http://www.markshuttleworth.com/archives/383)》首次介绍了两个新概念：Unity 界面，和 Ubuntu Light 操作系统。

2010 年中期正值上网本大行其道的年代，Linux 借助上网本进军桌面市场的思潮也正在盛行。在当时的双启动上网本概念中，用户可以在数秒内快速进入一个简化的 Linux 系统处理一些浏览网络、收取邮件的简单任务，而当需要完成大型任务时则可以启动 Windows 系统。为了迎合这一趋势，Canonical 准备推出名为 Ubuntu Light 的概念系统，而 Unity 就是应用于 Ubuntu Light 系统的用户界面。最初的 Unity 界面概念图是这样的：



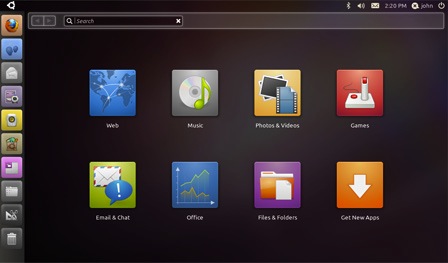
在这张概念图中已经可以见到今天 Unity 界面的基本结构和很多界面元素。所不同的是，在当时的概念中，左侧快速启动栏上的项目主要是一些浏览器的快捷方式（图中有 Gmail、Facebook、Youtube、Flickr 等），而界面的主体部分则基本完全是一个最大化的浏览器（图中正在运行一个地图网页）。总之，在最初的构想中，Unity 是一个彻底为快速完成网络浏览任务而打造的界面。

### 成长于特殊时期

此时，Canonical 方面也计划在 Ubuntu 10.10 的上网本专用版（Netbook Remix）中应用这一界面。当时距离 Ubuntu 10.10 的发布还有将近六个月的时间，Mark Shuttleworth [认为](http://www.markshuttleworth.com/archives/383)，在这六个月的时间里实现以下三者，即可成功地用 Unity 界面替换掉当时被认为体验不佳的上网本界面。

* 应用管理：在 Ubuntu Light 中，用户基本只需要一个浏览器，Unity 的界面也确实基本都留给了浏览器。而上网本专用版作为一个完整的桌面版本，则需要满足用户安装、卸载和管理软件的需求。
* 文件管理：Ubuntu Light 作为一个轻量级的系统，只需为用户提供浏览器界面既可。而上网本专用版则需具有文件管理的能力。
* 全局菜单：在[稍早的博文](http://www.markshuttleworth.com/archives/359)中，Mark Shuttleworth 宣布，为了尽量节省上网本小屏幕上的宝贵空间，Ubuntu 10.10 的上网本专用版系统将使用全局菜单替换掉传统菜单。

为了实现上面三项功能，Ubuntu 的开发者们为 Unity 界面添加了一个名为 Dash 的特性。用户可以在 Dash 中完成应用管理、文件管理等任务。而全局菜单也被逐步地实现出来。就这样，Ubuntu 10.10 的上网本专用版成为了第一个使用 Unity 界面的正式版本。



而也正由于这样的成长历程，导致了 Unity 始终无法摆脱“上网本界面”的影子。很多为上网本的小屏幕进行优化而来的特点，到后来都成为了累赘和缺陷。

### 转正于特殊时期

按照惯例，每个新版 Ubuntu 系统发布之后，Canonical 都要举行一次开发者大会。2010 年 10 月底，在 Ubuntu 10.10 版本发布之后的那次开发者大会上，Mark Shuttleworth [宣布](http://www.omgubuntu.co.uk/2010/10/ubuntu-11-04-unity-default-desktop/)下一个版本，即 Ubuntu 11.04，将把上网本专用版和普通桌面版合二为一，统一后的 Ubuntu 11.04 版本将全部使用 Unity 界面，取代掉传统上沿用多年的 Gnome 2 桌面环境的界面。

2010 年末 2011 年初对于 Linux 的主流桌面环境来说也是一个比较特殊的时期。Ubuntu 传统上使用的 Gnome 2 桌面环境已经持续小修小补了 9 年，而新版 Gnome 3 在经历了多年的难产和跳票之后处于即将发布的阶段。Gnome 3 的默认用户界面是 Gnome-shell，从当时已有的情况来看，这一变革过大的界面并不一定符合用户的口味，并且其硬件支持状况十分堪忧。而 Canonical 此时也正意欲借 Unity 界面来打造自己独特的用户体验，在这种特殊的情况下，Unity 用户界面就从一个上网本界面转正为 Ubuntu 的标准界面。

于是，6 个月后的 2011 年 4 月初，以 Gnome-shell 为用户界面的 Gnome 3 桌面环境发布，十几天后，Ubuntu 11.04 发布，以 Unity 为默认用户界面。并且不兼容 Gnome 3。

1. ubuntu界面的一些优缺点  
   优点：

Dash 和 Lens

Dash 是 Unity 的应用管理和文件管理界面。点击左上角的 Ubuntu 标志或者在桌面按下 Super 键（Windows 键/Command 键）即可进入，Lens 是 Dash 界面下方的一行图标，点击每个图标都可以切换到对应标签页，每个标签页致力于满足用户的一类特定需求。Dash 界面的基本结构如下：



Dash 在首页上显示最近使用的应用、打开的文件和下载的内容，而其后的各个 Lens 则分别满足各项特定的需求，默认的 Lens 有软件（应用程序管理）、文件（文件管理）、音乐（音乐管理）和视频（视频管理）。每个 Lens 都可以对相关的内容进行搜索、展示和分类过滤。

此外，用户还可以自行添加 Lens 来满足特定的需求。例如社交网络 Lens 可以快速地搜索、显示和过滤社交网络信息：



Launcher 与 Quicklist

Unity 界面的最左侧部分是一条纵向的快速启动条，正式名称是 Launcher。快速启动条上的图标有三类：系统强制放置的功能图标（Dash 主页、工作区切换器和回收站），用户自定义放置的常用程序图标，以及正在运行中的应用程序图标。

Launcher 的用户体验中规中矩，细节处理得相对比较理想。程序图标的左右两侧可以附加小三角形指示标志。正在运行的程序图标会在左侧有小三角形指示，如果正在运行的程序包括多个窗口，则小三角形的数量也会随之变化。而当前的活动窗口所属的程序，则同时还会在图标右侧显示一个小三角形进行指示。



二、操作系统的命令行接口

1.shell发展史

Bourne Shell     是1977年发布的Unix Version 7 中的默认Shell。目前大多数Unix系统仍保留着原始的 /bin/sh，或者保留一个软连接到其他Shell上 。该程序由 Stephen Bourne 在贝尔实验室工作期间开发，它的开发是做为 Thompson Shell 的一个替代。随着《The UNIX Programming Environment》（作者Brain W. Kernighan、Rob Pike）一书的出版，Bourne Shell开始变得流行起来。随着时间的发展，伴随着不断更新的系统（Version 7、SystemIII、SVR2、SVR3、SVR4），AT&T不断地增强shell的功能，但是因为shell一直没有版本号，所以我们只能依靠其支持的特性来区分SHELL的版本。对于SHELL的一些批评：交互式的使用不够友好（当时C Shell提供了history、aliases、job control等命令）；Unix系统使用C写成，但是shell的语法缺不像C，倒是有点像ALGOL；缺少正则表达式支持。

C Shell （csh）在1970年代，由当时还是加州大学伯克利分校的学生Bill Joy编写。在1978年，随着BSD Unix系统的第二个版本广泛发布。早期贡献代码的成员还包括：Michael Ubell, Eric Allman, Mike O'Brien and Jim Kulp。在一些系统如Mac OS X和Red Hat Linux上，csh实际上是tchsh，一种增强型的shell。csh包含了当今shell的绝大多数特性，包括filename wildcarding, piping, here documents, command substitution, variables and control structures for condition-testing and iteration。

虽然csh有着众多的交互特性以及快速的执行速度，但是最终未能流行起来。整个80年代，csh不能够确保出现在所有的Unix系统上，但是sh可以，这也是造成csh不够流行的主要因素。90年代中期，csh的分发开始广泛起来，这时又遭受了POSIX委员会的质疑，指出语法不够完善、特性缺失、扩展性差等问题[8]。

Korn Shell （ksh）由 David Korn 编写，1983年在USENIX上宣布。它向后兼容Bourne Shell，同时吸取了C Shell中的一些优点（job control）。代码贡献者还包括当时在Bell Labs的Mike Veach和Pat Sullivan。目前为开源软件：http://www.kornshell.org/。ksh的早期版本为 ksh88，是AIX4上的默认shell。后来又发布了ksh93版本。

Bash （the Bourne-Agian shell）由 Brian Fox 为GNU项目编写，目的是代替sh。首次发布于1989年，作为GNU项目的一个程序广泛分布在Linux、Mac OS X、Cygwin上。Brain Fox作为FSF的员工，首次开始工作是在1988年的1月10号，一年之后，发布了.99的测试版本。

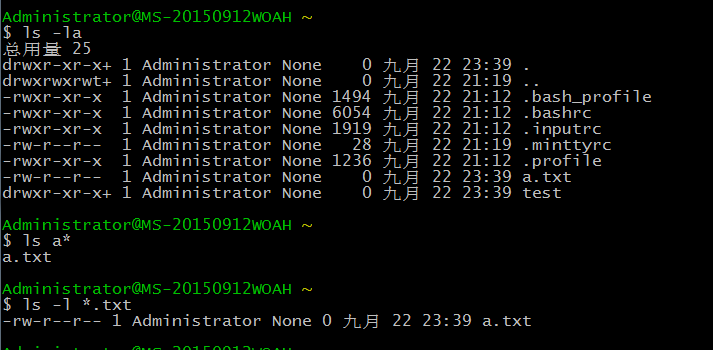
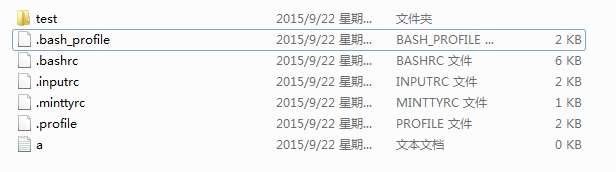
1. shell常用命令总结

**1  ls命令：列出文件**

ls -la 列出当前目录下的所有文件和文件夹

ls a\* 列出当前目录下所有以a字母开头的文件

ls -l \*.txt 列出当前目录下所有后缀名为txt的文件

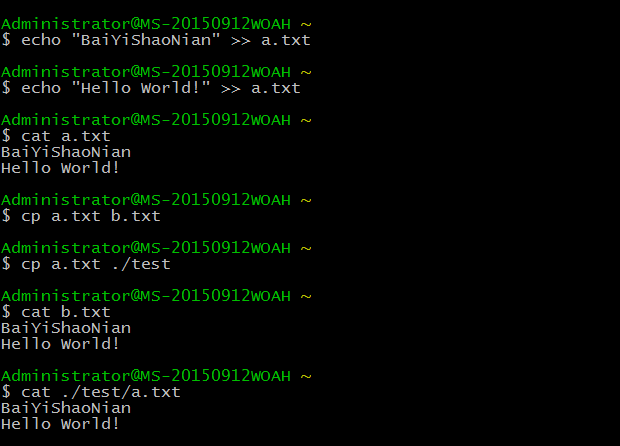


**2 cp命令：复制**

cp a.txt b.txt : 把文件a的内容复制到b文件

cp a.txt ./test : 把文件a复制到text目录下

cp -a test test2：递归的把目录test下所有文件（包括隐藏的文件）复制到新的目录 test2

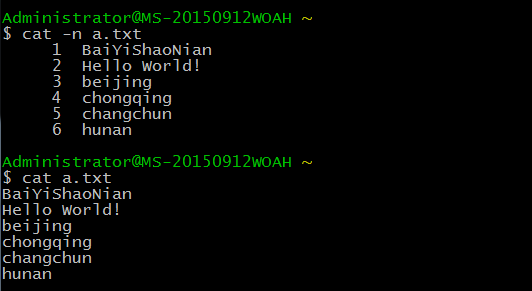
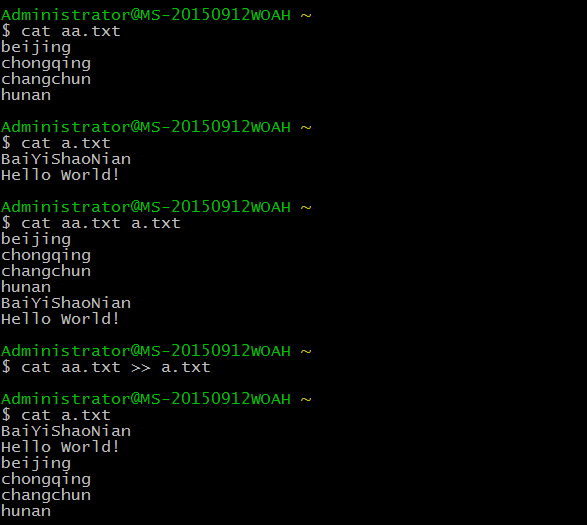


**3  cat命令：查看 组合文件**

cat a.txt：查看文件的内容

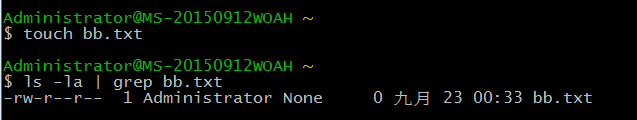
cat a.txt >> b.txt：把a文件的内容组合到b文件内容的末尾

cat -n a.txt：查看文件并给文件标上行号



**4  touch命令：建立文件**

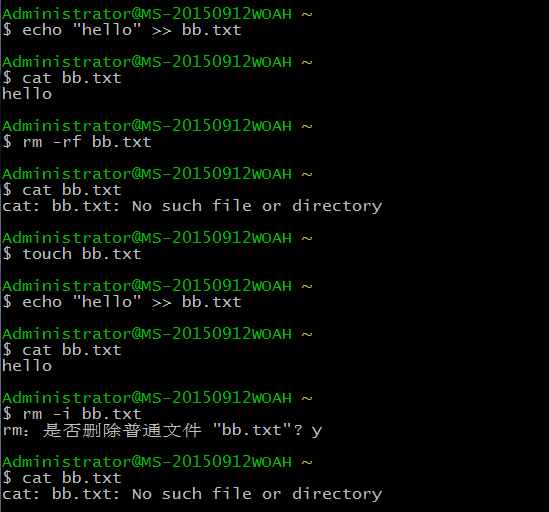
touch a.txt：建立一个名为a的txt类型文件



**5  rm命令：删除文件**

rm -rf a.txt：强制删除文件a.txt

tm -i a.txt：删除文件前会有提示是否确定删除该文件

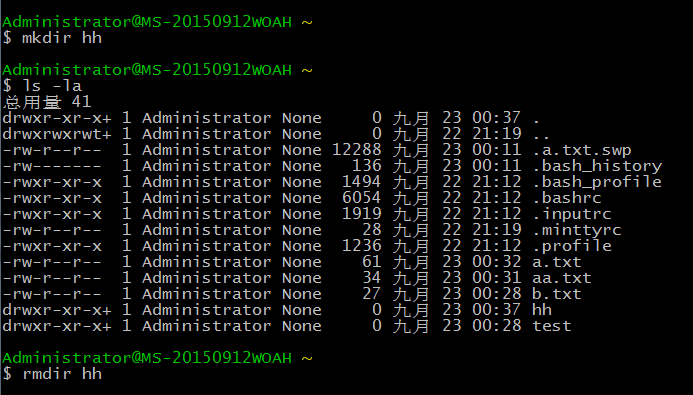


**6  mkdir命令：创建目录**

mkdir test：创建一个名为test的目录

**7  rmdir命令：删除目录**

tmdir test：删除一个目录

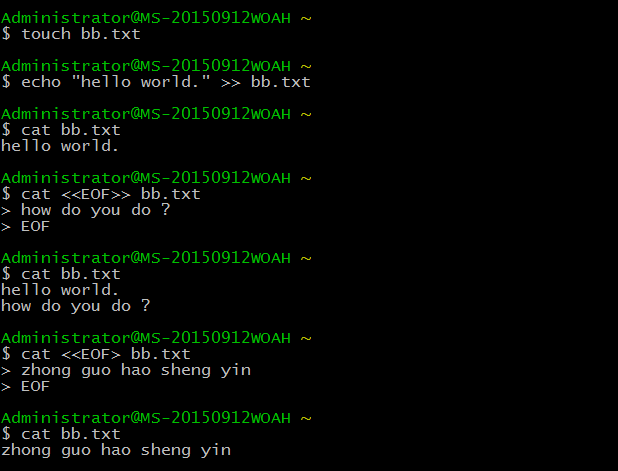


**8  echo、cat命令：添加内容**

echo “hello world!” >> a.txt：添加内容到文件a里面

cat <<EOF>> a.txt : 可以添加多行语句到文件本身内容的末尾

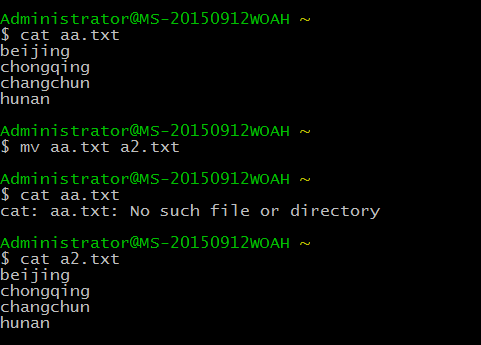
cat <<EOF> a.txt：添加内容到文件并覆盖到原始的内容



**9  mv命令：移动 重命名文件**

mv a.txt b.txt：文件a重新命名为b

mv a.txt ./test：把文件移动到一个目录下



**10  cd命令：更换目录**

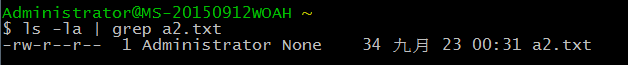
cd ~ : 切换到用户目录

cd .. ：返回到上一层目录

cd ../.. ：返回到上二层目录

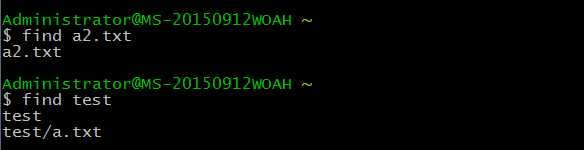
**11  grep命令：搜索文件**

ls -la | grep a.txt ：搜索a.txt文件



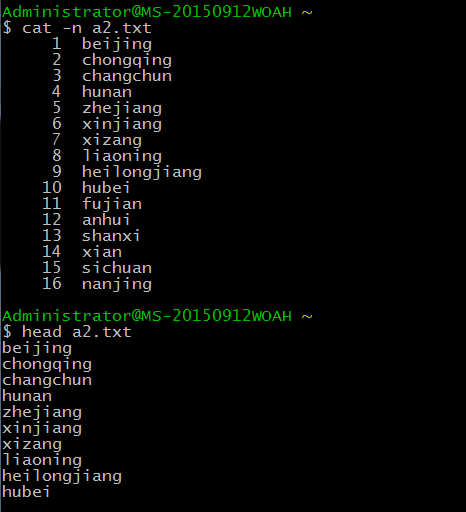
**12  find命令：查找文件和目录**

find filename：查找当前目录下是否有该文件/目录

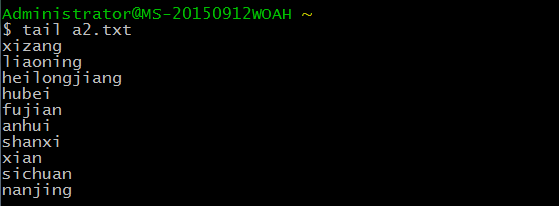


**13  rz sz命令：上传和下载文件**

**14  head命令：显示文件的前10行内容**



**15  tail命令：显示文件最后10行内容**



1. 操作系统的应用编程接口
2. folk（）系统调用

代码：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

{ int pid; /\* fork another process \*/ pid = fork();

if (pid < 0)

{ /\* error occurred \*/ fprintf(stderr,"Fork Failed!");

exit(-1); }

else if (pid == 0)

{ /\* child process \*/ printf("This is Child Process!\n"); }

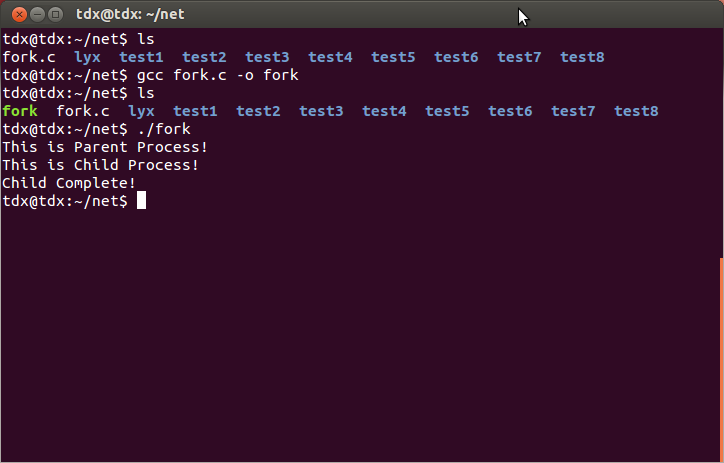
else

{ /\* parent process \*/ printf("This is Parent Process!\n");

/\* parent will wait for the child to complete\*/ wait(NULL);

printf("Child Complete!\n"); } }

运行结果：



fork()函数又叫计算机程序设计中的分叉函数，fork是一个很有意思的函数，它可以建立一个新进程，把当前的进程分为父进程和子进程，新进程称为子进程，而原进程称为父进程。fork调用一次，返回两次，这两个返回分别带回它们各自的返回值，其中在父进程中的返回值是子进程的PID，而子进程中的返回值则返回 0。因此，可以通过返回值来判定该进程是父进程还是子进程。还有一个很奇妙的是：fork函数将运行着的程序分成2个（几乎）完全一样的进程，每个进程都启动一个从代码的同一位置开始执行的线程。这两个进程中的线程继续执行，就像是两个用户同时启动了该应用程序的两个副本。

新创建的子进程几乎但是不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同（但是独立）的一份拷贝，包括文本，数据和bss段、堆以及用户栈。子进程还获得与父进程任何打开文件描述符相同的拷贝。这就是意味着当父进程调用fork时候，子进程还可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大区别在于他们有着不同的PID。

UNIX将复制父进程的地址空间内容给子进程，因此，子进程有了独立的地址空间。在不同的UNIX (Like)系统下，我们无法确定fork之后是子进程先运行还是父进程先运行，这依赖于系统的实现。

下面我们再以一个最简单的代码来更简单说明fork()函数：

#include <stdio.h>

#include <unistd.h>

int mian(void){

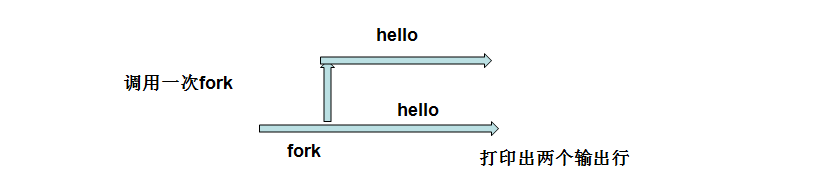
fork();

printf("hello");

return 0;

}

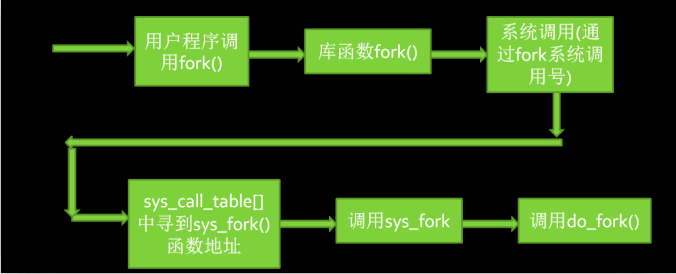
他的运行结果我们以一个图的形式展现出来，如下：



这个例子简单明了的把fork的作用表现出来了。在这里父进程和子进程都执行printf()，所以会输出两个hello。如果fork出现错误，则fork返回一个负值。

### fork()内核处理过程

Linux下用于创建进程的API有三个fork,vfork和clone，这三个函数分别是通过系统调用sys\_fork,sys\_vfork以及sys\_clone实现的 （这里目前讨论的都是基于x86架构的）。而且这三个系统调用，都是通过do\_fork来实现的，只是传入了不同的参数。所以我们可以得出结论：所有的子进程是在do\_fork实现创建和调用的。下面我们就来整理一下整个进程的在用户态到内核态的过程是怎么样的。fork系统调用如下：



下面我们来重点看看do\_fork的代码

long do\_fork(unsigned long clone\_flags,

unsigned long stack\_start,

unsigned long stack\_size,

int \_\_user \*parent\_tidptr,

int \_\_user \*child\_tidptr)

{

struct task\_struct \*p;

int trace = 0;

long nr;

/\*

\* Determine whether and which event to report to ptracer. When

\* called from kernel\_thread or CLONE\_UNTRACED is explicitly

\* requested, no event is reported; otherwise, report if the event

\* for the type of forking is enabled.

\*/

if (!(clone\_flags & CLONE\_UNTRACED)) {

if (clone\_flags & CLONE\_VFORK)

trace = PTRACE\_EVENT\_VFORK;

else if ((clone\_flags & CSIGNAL) != SIGCHLD)

trace = PTRACE\_EVENT\_CLONE;

else

trace = PTRACE\_EVENT\_FORK;

if (likely(!ptrace\_event\_enabled(current, trace)))

trace = 0;

}

p = copy\_process(clone\_flags, stack\_start, stack\_size,

child\_tidptr, NULL, trace);

/\*

\* Do this prior waking up the new thread - the thread pointer

\* might get invalid after that point, if the thread exits quickly.

\*/

if (!IS\_ERR(p)) {

struct completion vfork;

struct pid \*pid;

trace\_sched\_process\_fork(current, p);

pid = get\_task\_pid(p, PIDTYPE\_PID);

nr = pid\_vnr(pid);

if (clone\_flags & CLONE\_PARENT\_SETTID)

put\_user(nr, parent\_tidptr);

if (clone\_flags & CLONE\_VFORK) {

p->vfork\_done = &vfork;

init\_completion(&vfork);

get\_task\_struct(p);

}

wake\_up\_new\_task(p);

/\* forking complete and child started to run, tell ptracer \*/

if (unlikely(trace))

ptrace\_event\_pid(trace, pid);

if (clone\_flags & CLONE\_VFORK) {

if (!wait\_for\_vfork\_done(p, &vfork))

ptrace\_event\_pid(PTRACE\_EVENT\_VFORK\_DONE, pid);

}

put\_pid(pid);

} else {

nr = PTR\_ERR(p);

}

return nr;

}

整段代码挺长，涉及到很多工作的处理，但是整个创建新进程是在上述代码的第29行copy\_process()z这个函数实现的。我们前面已经说过，子进程是通过复制实现的。为了探个究竟，我们进入到copy\_process()这个函数体里可以看到几个很重要的函数，列举如下：

复制一个PCB——task\_struct

p = dup\_task\_struct(current);

复制当前进程的PCB描述符task\_struct。我们在进入到该函数dup\_task\_struct体内就可以看到这个pcb是如何复制的。主要的赋值函数是

err = arch\_dup\_task\_struct(tsk, orig);//这一句是赋值操作

当然在dup\_task\_struct函数体内还有其他的一次辅助操作例如：

tsk = alloc\_task\_struct\_node(node);

ti = alloc\_thread\_info\_node(tsk, node);

tsk->stack = ti;

setup\_thread\_stack(tsk, orig);//这里只是复制thread\_info，而非复制内核堆栈

然而我们再 往dup\_task\_struct(current)函数往下看，后面是大量的修改进程的内容，也就是对复制过来的东西修改为子进程所拥有的数据。也就是初始化一个子进程。我们再往下看，在copy\_process()函数的第1396行有一个非常重要的函数copy\_thread。进去之后我们可以看到，一部分重要代码如下：

struct pt\_regs \*childregs = task\_pt\_regs(p);

struct task\_struct \*tsk;

int err;

p->thread.sp = (unsigned long) childregs;

p->thread.sp0 = (unsigned long) (childregs+1);

memset(p->thread.ptrace\_bps, 0, sizeof(p->thread.ptrace\_bps));

if (unlikely(p->flags & PF\_KTHREAD)) {

/\* kernel thread \*/

memset(childregs, 0, sizeof(struct pt\_regs));

p->thread.ip = (unsigned long) ret\_from\_kernel\_thread;

task\_user\_gs(p) = \_\_KERNEL\_STACK\_CANARY;

childregs->ds = \_\_USER\_DS;

childregs->es = \_\_USER\_DS;

childregs->fs = \_\_KERNEL\_PERCPU;

childregs->bx = sp; /\* function \*/

childregs->bp = arg;

childregs->orig\_ax = -1;

childregs->cs = \_\_KERNEL\_CS | get\_kernel\_rpl();

childregs->flags = X86\_EFLAGS\_IF | X86\_EFLAGS\_FIXED;

p->thread.io\_bitmap\_ptr = NULL;

return 0;

}

\*childregs = \*current\_pt\_regs();//拷贝父进程的内核堆栈栈底，也就是已有的内核堆栈数据的拷贝

childregs->ax = 0;//给eax赋值为0，因为子进程返回的是0，系统调用是通过eax返回的，

if (sp)

childregs->sp = sp;//修改栈顶

p->thread.ip = (unsigned long) ret\_from\_fork;//给ip赋值，这就是子进程执行的起点

从用户态的代码看fork();函数返回了两次，即在父子进程中各返回一次，父进程从系统调用中返回比较容易理解，子进程从系统调用中返回，那它在系统调用处理过程中的哪里开始执行的呢？这就涉及子进程的内核堆栈数据状态和task\_struct中thread记录的sp和ip的一致性问题，这是在哪里设定的？

copy\_thread in copy\_process

struct pt\_regs \*childregs = task\_pt\_regs(p);

\*childregs = \*current\_pt\_regs(); //复制内核堆栈栈底

childregs->ax = 0; //为什么子进程的fork返回0，这里就是原因！

p->thread.sp = (unsigned long) childregs; //调度到子进程时的内核栈顶

p->thread.ip = (unsigned long) ret\_from\_fork; //调度到子进程时的第一条指令地址

上面赋值复制的内核堆栈并不是父进程的所有内核堆栈的内容，那复制的是哪些部分呢？我们可以看上面代码的第一句，其中复制的内容就是pt\_regs里面的内容。里面的代码如下：

struct pt\_regs {

long ebx;

long ecx;

long edx;

long esi;

long edi;

long ebp;

long eax;

int xds;

int xes;

int xfs;

int xgs;

long orig\_eax;

long eip;

int xcs;

long eflags;

long esp;

int xss;

};

父进程堆栈复制给子进程的就是上面那些参数。从copy\_thread中我们就已经得出堆栈复制和子进程开始执行的起始地方。综上所述，我们对整个do\_fork的分析到此就可以告一段落了。我们在回过头来总结一下，do\_fork()的实现，主要是靠copy\_process()完成的，这就是一环套一环。整个过程实现如下：

p = dup\_task\_struct(current);　为新进程创建一个内核栈、thread\_iofo和task\_struct,这里完全copy父进程的内容，所以到目前为止，父进程和子进程是没有任何区别的。

为新进程在其内存上建立内核堆栈

对子进程task\_struct任务结构体中部分变量进行初始化设置，检查所有的进程数目是否已经超出了系统规定的最大进程数，如果没有的话，那么就开始设置进程描诉符中的初始值，从这开始，父进程和子进程就开始区别开了。

把父进程的有关信息复制给子进程，建立共享关系

设置子进程的状态为不可被TASK\_UNINTERRUPTIBLE，从而保证这个进程现在不能被投入运行，因为还有很多的标志位、数据等没有被设置

复制标志位（falgs成员）以及权限位(PE\_SUPERPRIV)和其他的一些标志

调用get\_pid()给子进程获取一个有效的并且是唯一的进程标识符PID

return ret\_from\_fork;返回一个指向子进程的指针，开始执行

**五、实验小结（心得体会、遇到的问题及解决方式、未解决/需进一步研讨的问题或建议新实验方法等）**

在实验过程中，体会到了一个操作系统带给新用户的新鲜感，了解ubuntu的unity用户界面的优缺点，以及shell的一些常用命令，最后也粗略地了解了fork（）系统调用的过程内容，在linux中创建一个新的进程是从复制开始的，在系统内核里首先是将父进程的进程控制块PCB进行拷贝，然后再根据自己的情况修改相应的参数，获取自己的进程号，再开始执行。我觉得整个过程重点就是理解子进程如何创建，在内核调用的几个重要的内核函数，以及子进程怎么返回开始执行的。