操作系统课程实验报告

**2020年春季学期**

|  |  |
| --- | --- |
| 姓 名： | 王少博 |
| 学 号： | **181110315** |
| 院 系： | **计算机科学与技术学院 软件工程系** |
| 电 话： | **+86 18156824881** |
| 得 分： |  |

**哈尔滨工业大学（威海） 计算机科学与技术学院**

**2020年6月**

**目 录**

[**1.** fork（）系统调用实验（20分） 1](#_Toc42119126)

[**2.** 共享内存进程间通信实验（10分） 2](#_Toc42119127)

[**3.** 线程实验（10分） 3](#_Toc42119128)

[**4.** 线程竞争范围、调度算法确认实验（10分） 4](#_Toc42119129)

[**5.** 调度程序例子代码（10分） 5](#_Toc42119130)

[**6.** 消息队列进程间通信实验（10分） 6](#_Toc42119131)

[**7.** Make Utility实验（10分） 7](#_Toc42119132)

[**8.** 静态链接、动态链接、动态加载实验（20分） 8](#_Toc42119133)

# fork（）系统调用实验（20分）

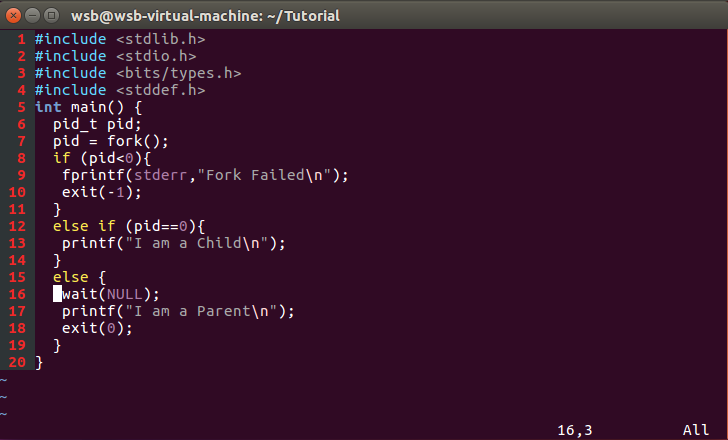
|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

* 1. 学习目标（需要掌握的知识点）

掌握fork()函数的使用；父子进程之间资源分配关系；掌握wait()函数的使用

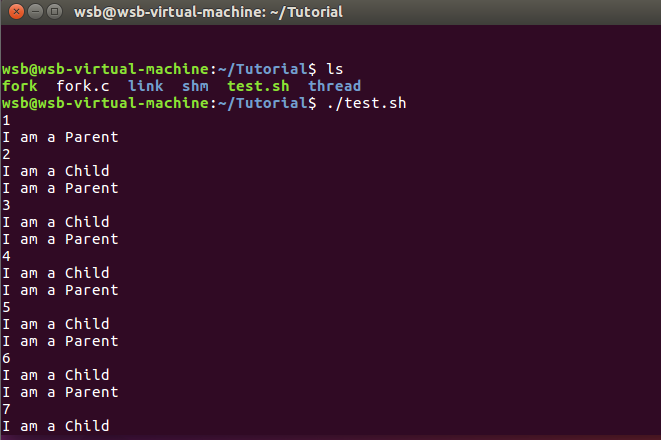
* 1. 实验内容（描述实验内容及代码，用截图图片展示并能证明代码及阶段性结果）
* 有wait(NULL)时，运行100次的结果

代码：



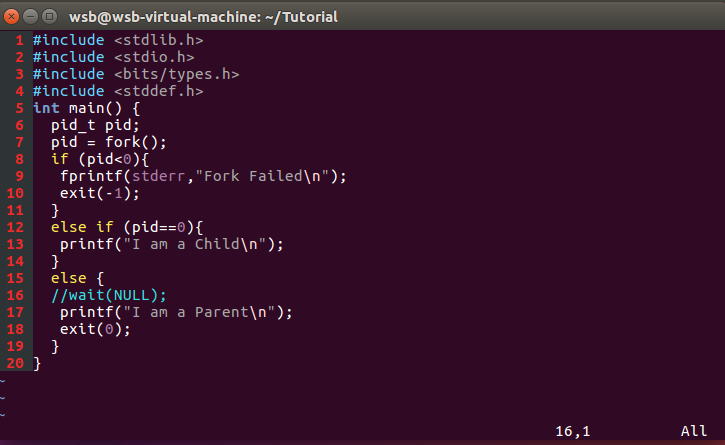
fork()根据进程号判断是父进程还是子进程。父进程对应的pid是0，子进程对应的pid>0。如果异常情况出现pid<0。对应打印一些提示语句。

结果：



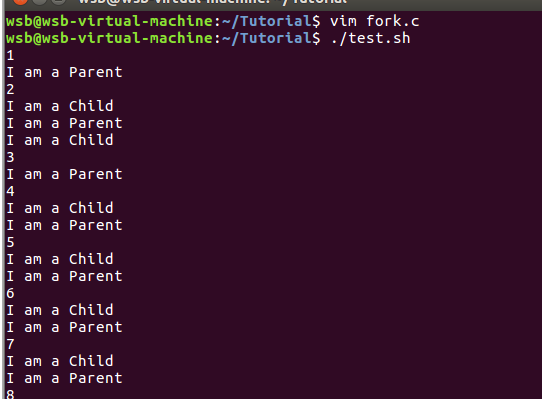
* 无wait(NULL)时，运行100次的结果

代码：



fork()根据进程号判断是父进程还是子进程。父进程对应的pid是0，子进程对应的pid>0。如果异常情况出现pid<0。对应打印一些提示语句。

结果：



* 1. 实验结论

子进程通过wait()函数的返回值告诉父进程自己结束了，然后父进程可以释放资源继续运行。因此，去掉wait()之前，子进程结束之后，父进程进行，但是去掉wait()之后，两者执行的先后顺序是不确定的，所以输出的结果顺序也是不确定的。

# 共享内存进程间通信实验（10分）

|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

* 1. 学习目标（需要掌握的知识点）

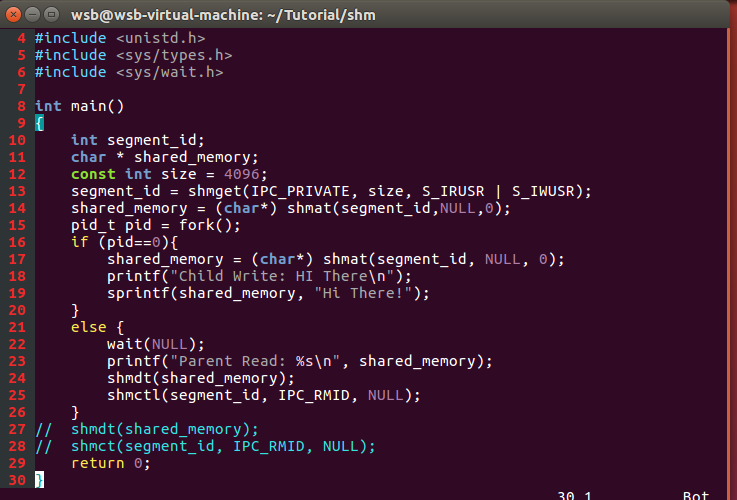
掌握进程间的通信的原理，顺序以及共享内存的通信过程。

掌握共享内存函数的使用。

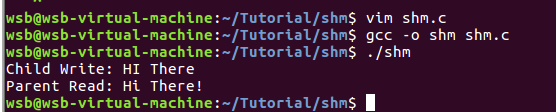
* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

含有wait()

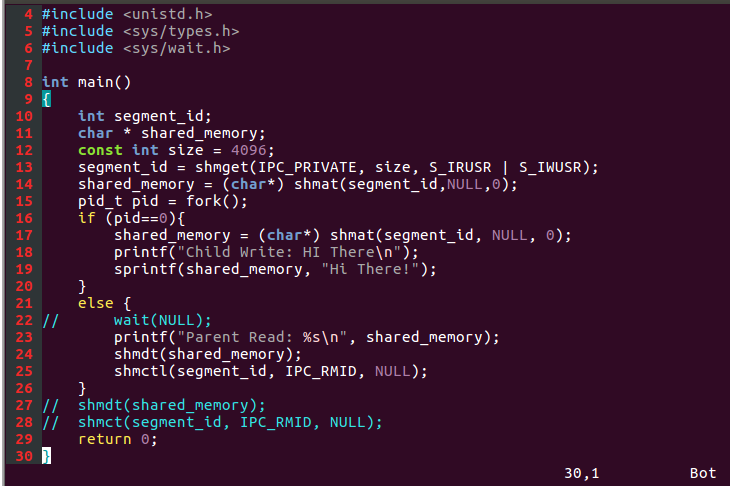
代码：



结果：



代码：



结果：

多了新情况，



* 1. 实验结论

zi进程和父进程共享同一块内存，如果父进程先枪战，但是子进程还没有向内存写入内容，父进程就得不到数据。反之子进程先运行并写入内容，父进程就可以得到数据了。所以，去掉wait之后，如果子进程先得到CPU资源，就和之前得情况一样，反之父进程先得到数据，输出结果就是空。因此会有两种不同结果。

# 线程实验（10分）

|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

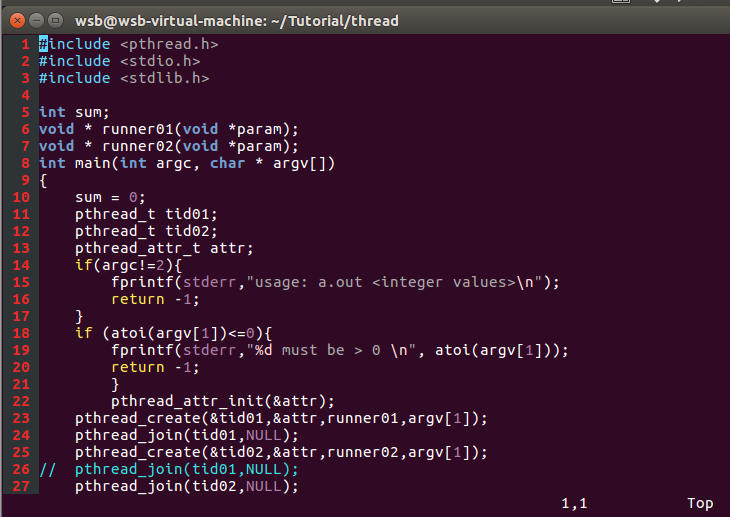
* 1. 学习目标（需要掌握的知识点）

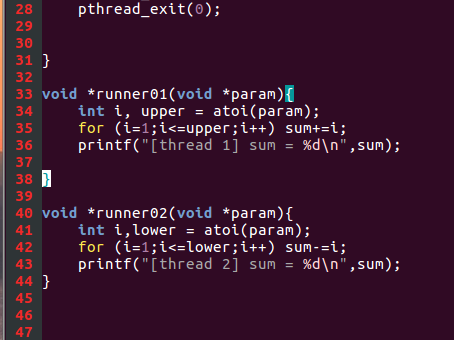
掌握线程的相关知识点，掌握与线程创建、线程执行等操作的相关函数，如pthread\_create()、pthread\_join()等函数

* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）
* 运行一次的运行结果

代码：只放上了修改后的代码，修改前仅仅是顺序不同。两个线程在主进程之间顺序是不确定，join函数的位置决定了他们的运行顺序。

修改后的代码：

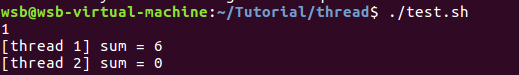
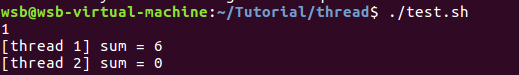




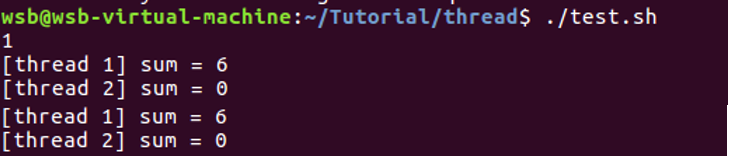
分别创建两个线程运行runner01,runner02。

结果：

* + 修改前：会出现-6，0和6，0两种情况，因为两个线程的顺序是不确定的。

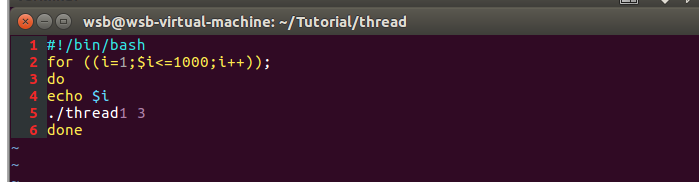


* + 修改后



* 用Shell 运行1000次的运行结果以及比较

写一个脚本来循环运行1000次

两次代码的差距就在join的顺序上做了区别。线程1的执行放在了线程2的创建之前，让线程一执行结束之后再运行线程二

* 1. 实验结论

了解了几个函数的作用。join等待一个线程的结束，但是我们在主进程中调用线程的join，是相对于主进程而言的，并非是线程线程之间的等待关系。所以修改了代码之后，让他们独立出来，就可以体现出等待的关系了，否则运行的顺序是不确定的。

另外就是编译的时候，需要链接一下线程库，-pthread来链接静态库

# 线程竞争范围、调度算法确认实验（10分）

|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

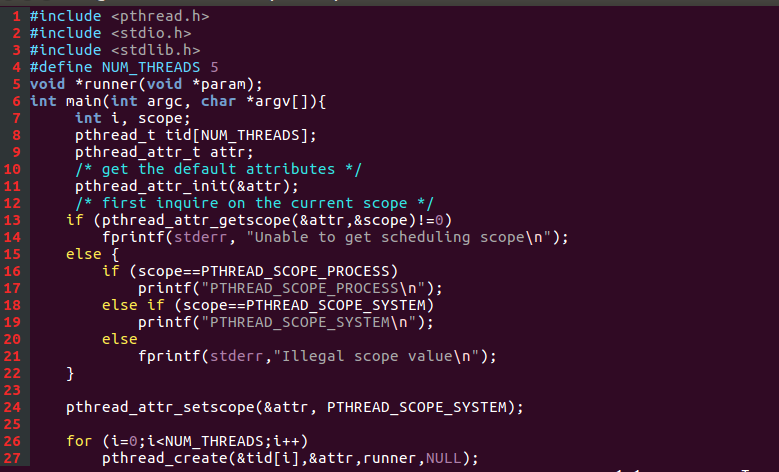
* 1. 学习目标（需要掌握的知识点）

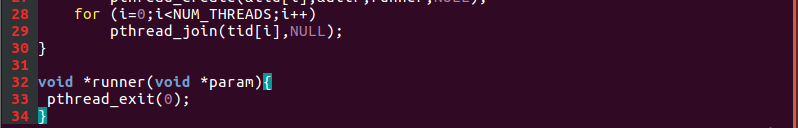
掌握线程竞争以及调度算法的相关知识点；

了解线程竞争的种类

* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

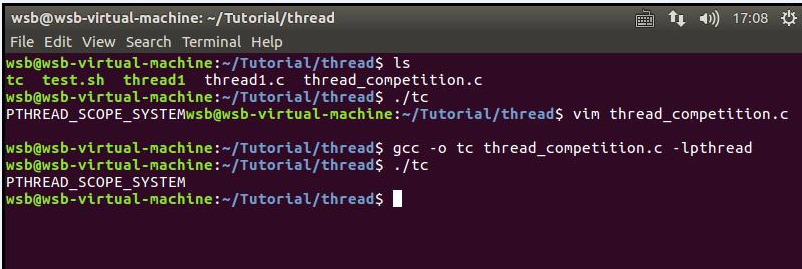
代码：





创建线程初始化。通过pthread\_attr\_getscope()得到竞争种类的参数结果。 pthread\_attr\_setscope()可以设置竞争的种类。

结果：



* 1. 实验结论

系统默认的线程竞争范围是系统竞争范围“PTHREAD\_SCOPE\_SYSTEM”，线程与系统内所有线程进行竞争，竞争内存等资源。

# 调度程序例子代码（10分）

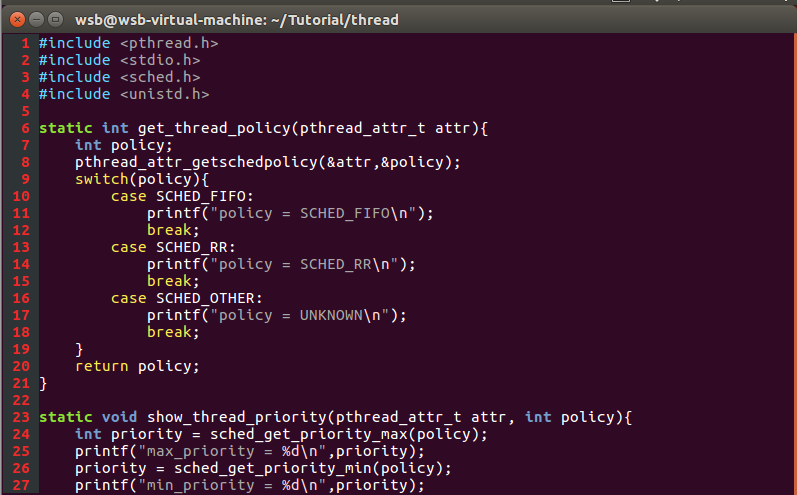
|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

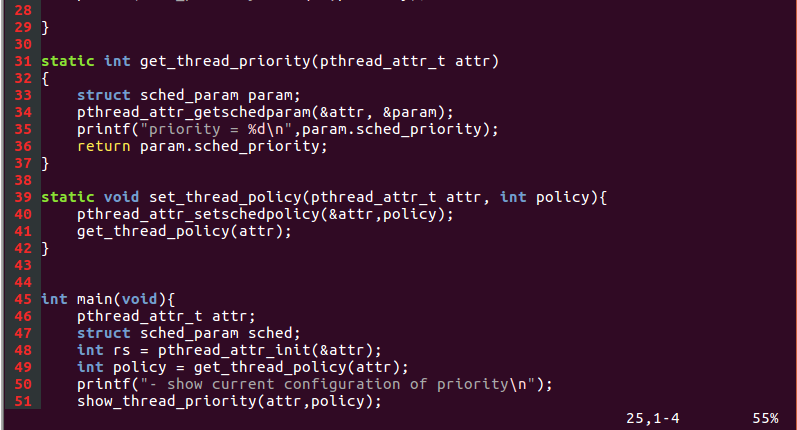
* 1. 学习目标（需要掌握的知识点）

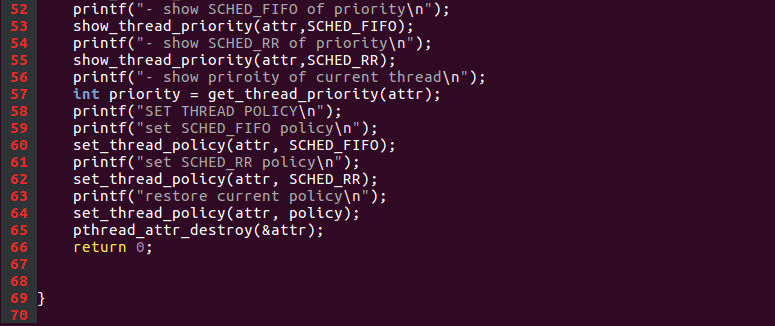
掌握CPU调度程序的相关算法和原理

* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

代码：

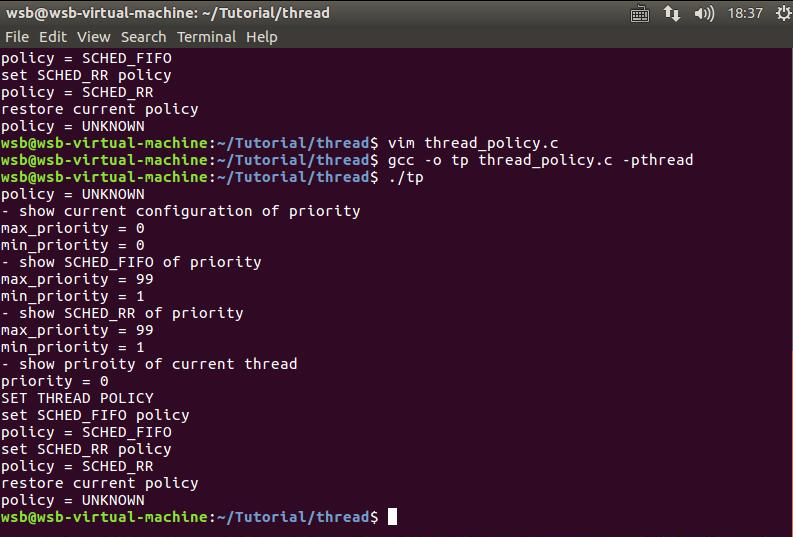






通过set\_thread\_policy()函数，适当参数实现不同的算法。

结果：



* 1. 实验结论

当前的调度算法未知，最大优先级为0最小优先级也为0；

FIFO最大优先级为99，最小优先级为1；

RR算法最大优先级为99，最小优先级为0；

# 消息队列进程间通信实验（10分）

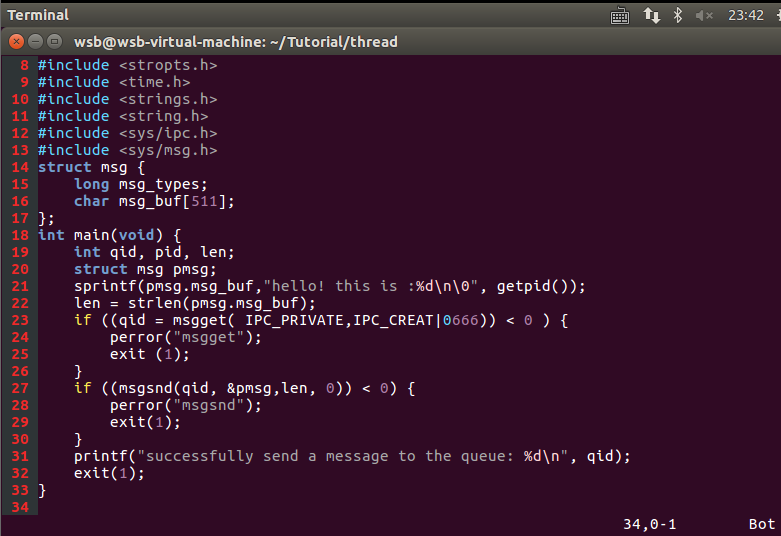
|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

* 1. 学习目标（需要掌握的知识点）

父进程创建两个子进程P1和P2，并使子进程P1和P2通 过消息队列相互通信。通过该实验了解两个进程之间的通信机制，以及收发消息

* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

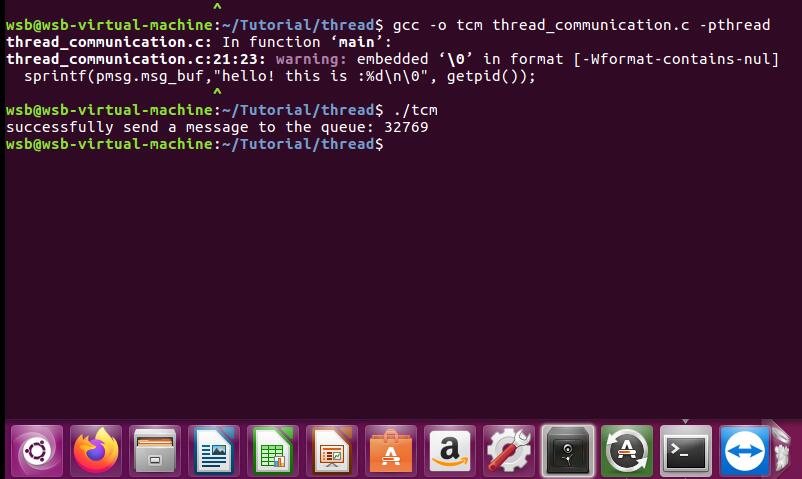
代码：



msg结构体存放具体的信息内容

创建两个进程，检测是否创建成功。

如果创建成功，判断是子进程就完成对消息的收发

结果

* 1. 实验结论

在一个进程还未写入消息时，另一个进程就进行读取，结果肯定是什么也没有获取到。这个如果能加入一些消息顺序控制的机制，就能避免这样的结果发生了。还可以通过互斥锁的方式来。同时，消息队列一定是可以服用的，因为有一个id作为标识来指明具体是哪一个队列。

# Make Utility实验（10分）

|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

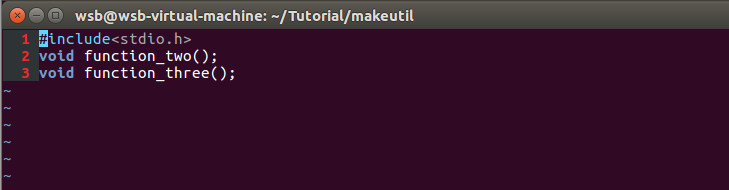
* 1. 学习目标（需要掌握的知识点）

make utility可以省去重新编译所耗费的大量时间，通过命令解释配置指令和规则。并且它针对修改的源文件重新编译。

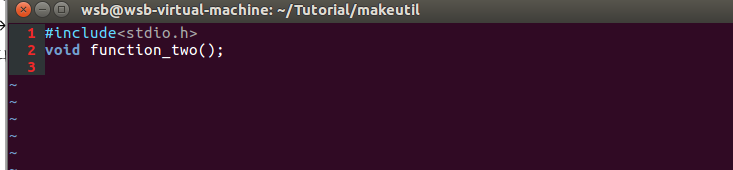
* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

代码：

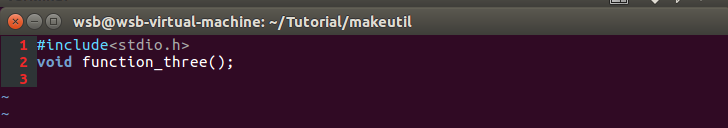
a.h文件：

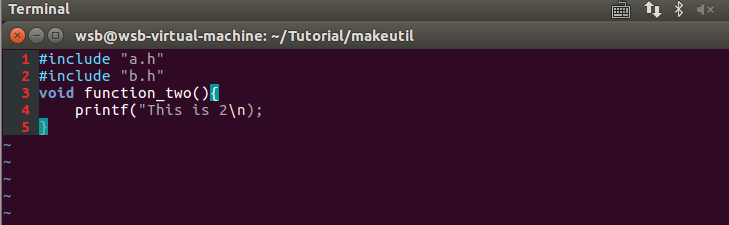


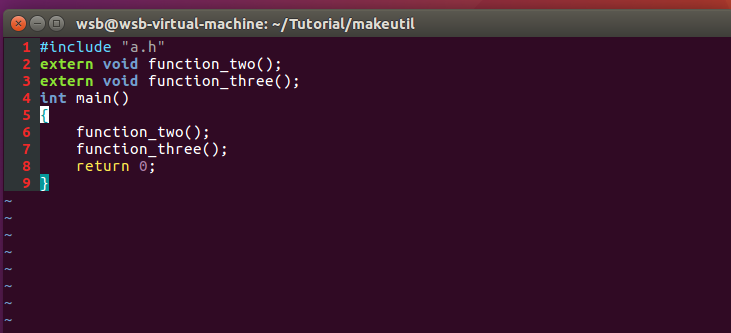
b.h文件：

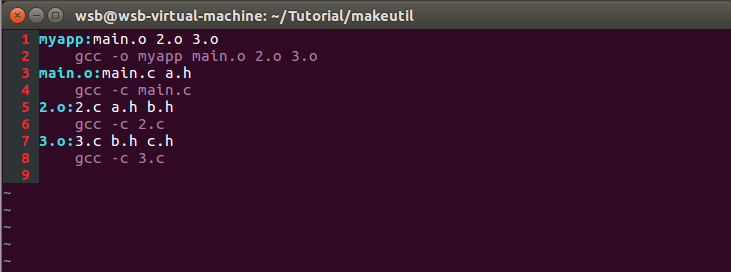


c.h文件：



2.c文件：

 main.c文件

 makefile文件

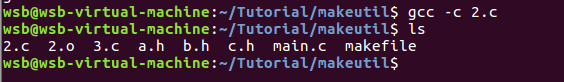
我们的可执行文件myapp与 main.o, 2.o, 3.o 文件有依赖性

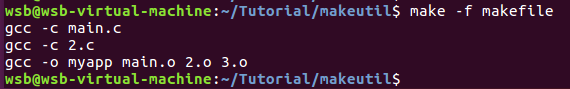
Main.o目标文件与main.c, a.h文件有依赖关系

2.o目标文件与2.c, a.h, b.h 文件有依赖关系

3.o 目标文件与3.c, b.h, c.h 文件有依赖关系

结果：





* 1. 实验结论

Makefile文件指定了生成的目标文件与源文件的依赖关系，指定从源文件生成目标文件的生成规则。通过编译生成的目标文件、库文件、可执行文件等需要方便去管理，从而提高重新编译时的速度，提高效率。

# 静态链接、动态链接、动态加载实验（20分）

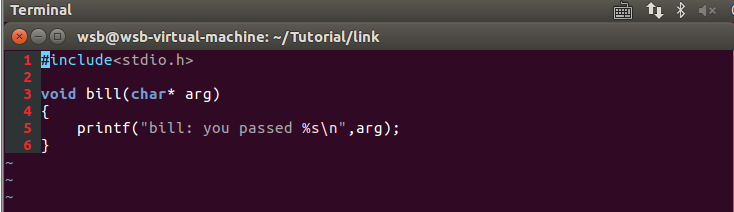
|  |  |  |  |
| --- | --- | --- | --- |
| 实验时间 |  | 实验地点 |  |
| 得分 |  | | |

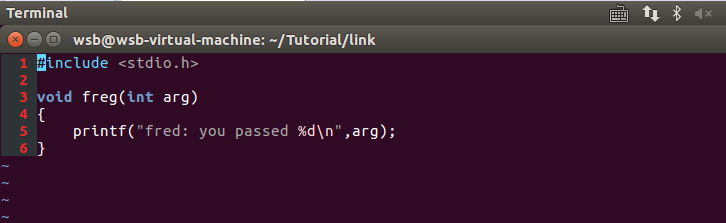
* 1. 学习目标（需要掌握的知识点）

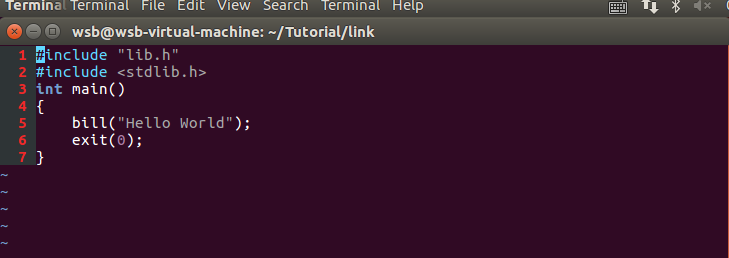
掌握动态链接、静态链接、动态加载各自的过程，以及他们之间的关系和原理，以及所占用的资源大小

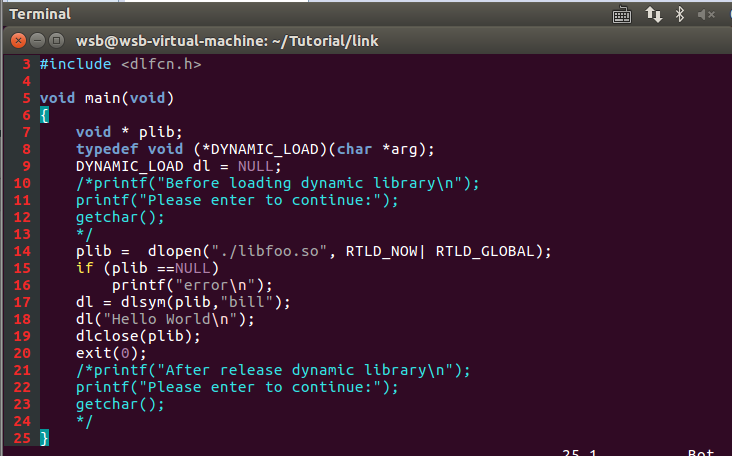
* 1. 实验内容（描述实验内容及代码，用截图展示并能证明代码及阶段性结果）

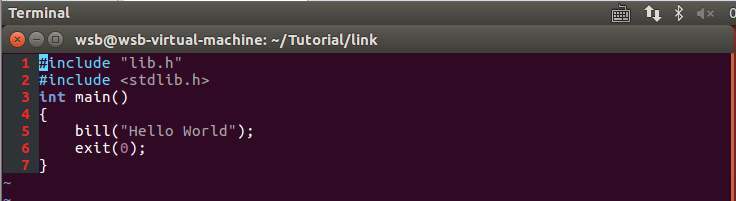
代码：

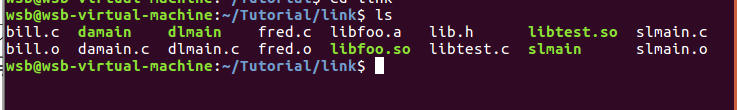
bill.c:

fred.c:

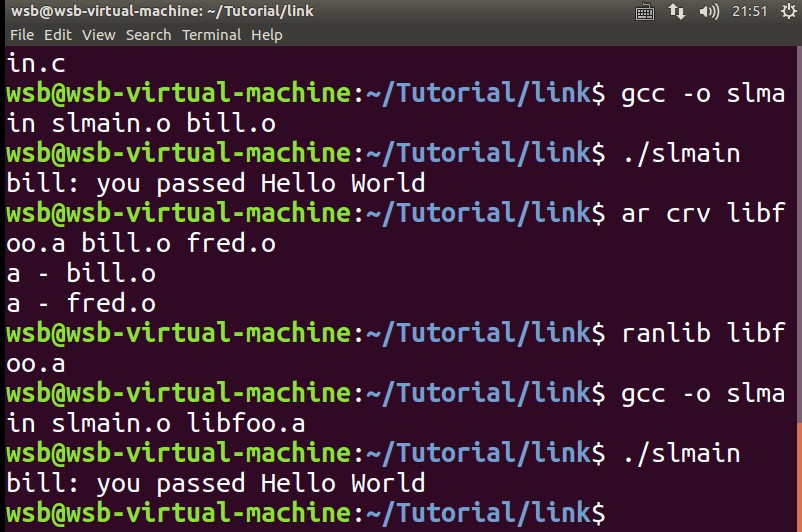
slmain.c

damain.c

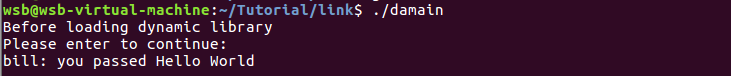
dlmain.c

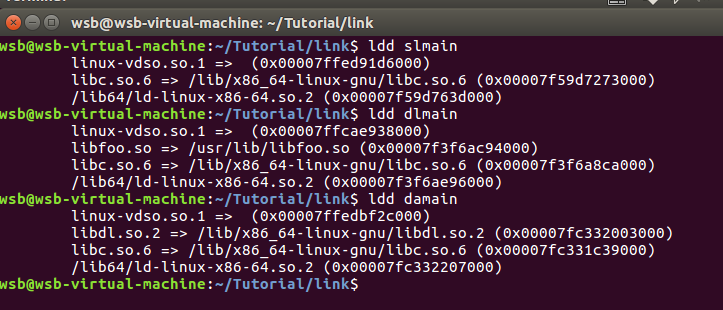
结果：

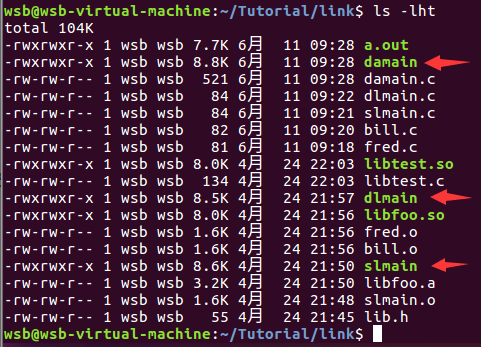
静态库



动态链接

动态加载

大小比较



damain 8.8k

dlmain 8.5k

slmain 8.6k

* 1. 实验结论

同等代码量下动态链接要比静态链接节省内存空间

动态加载的代码量比其他两个要多所以内存可能会大了一些。

静态库一开始链接就加载完成。动态库是需要用的时候才会加载，动态链接、动态加载、静态链接的依赖关系可以从图中看初。

关于dlclose还有一个补充，就是动态库计数器=0的时候，才会被系统卸载，其余情况不会。