V0.1改动日志：增加了如何运行halt的设置法

V0.15改动日志：增加了Ubuntu如何编译

V0.2改动日志：增加了让echo.coff跑起来的尝试

V0.3改动日志：修改了一些bug，并增加了对文件系统一些坑爹问题的解释，增加了对内存的一些看法

V0.5改动日志：增加了一下内存部分和多进程部分

V0.6改动日志：修改了一些bug，增加了几个数据

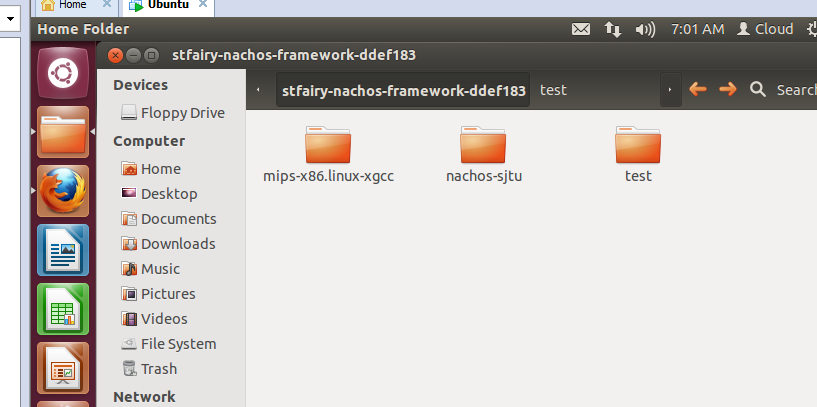
这个phase我们首先面临两个问题：

1. 怎么编译c文件
2. 怎么运行
3. 我随便想了一下，感觉我们只需要.coff文件就行了，所以c文件怎么编译啊现在应该可以不用管，助教说了在Linux下会很方便，所以我觉得不如把这档子事放到Linux下比较好（就是说编译和nachos运行应该是两回事，我们可以把编译好的coff文件拷贝到win下）

从这里可以下到mips编译器

<http://www.cs.berkeley.edu/~kubitron/courses/cs162-F05/Nachos/xgcc.html>

这么放置



然后跑到test文件夹下

直接make就出来一帮子coff啦

如果你想自己写一个.c文件，只需放入test文件夹

再make一次就有coff了

今天四含姐姐和霖哥按照我的notes搞编译但碰到了同样的问题，问了诚牛发现原因是你们系统是64位的

我想吐槽一下：装64位linux不是没事找事么。。。。。。

目前解决方案有这些：

-1.让Mr.Ban教你怎么在win7上make

0.让诚牛给你捣鼓一下就行了。。。

1.最推荐的是重装个32位linux（建议用虚拟机），虚拟机软件和iso什么的葡萄和学校ftp上有，也可以问我要

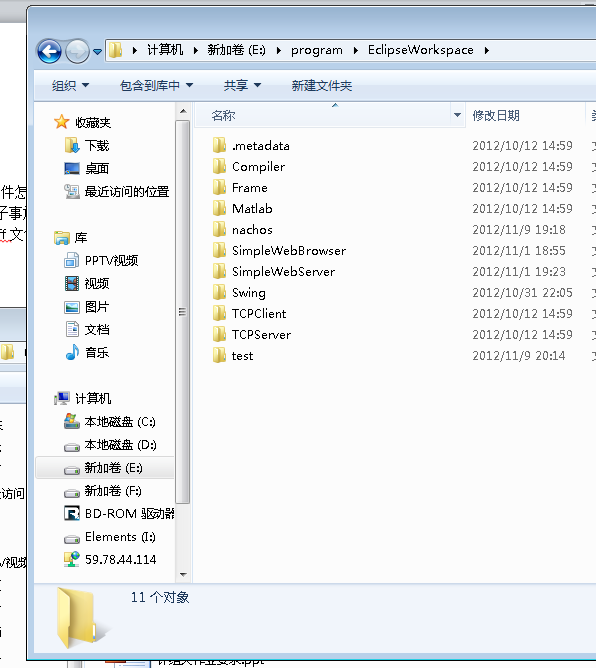
2.按照那个网页上的做法自己建一个cross compiler（没人试过）

3.如果你实在不想弄，可以把你的.c用邮件发给我，我来帮你编译

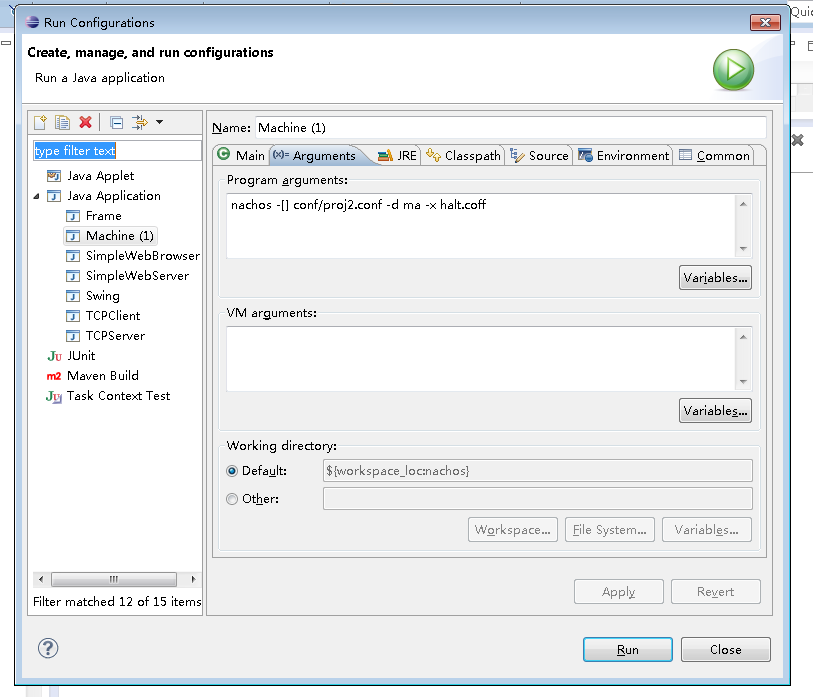
希望没有装64位linux的亲不要用第3个方法，会很伤友情

最重要的是千万不要把自己卡在编译这块上面，没意思，先做后面的也完全没问题

2.为了获得一些自信，我们现在就能跑halt.coff，按照nachos里的一些蹊跷，我们需要把test放到workspace下



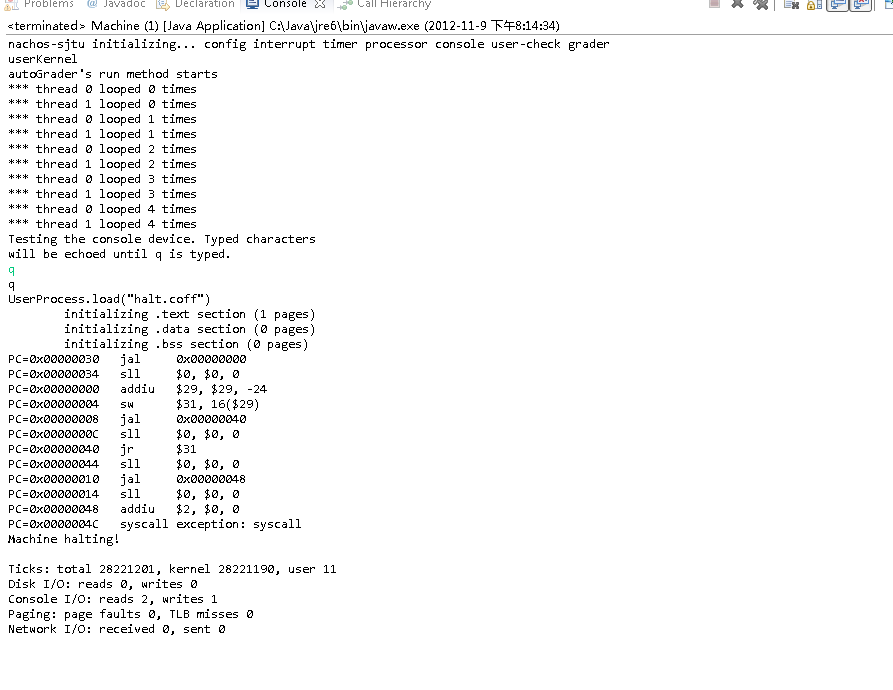
然后参数如下设置



然后不要忘记把machine.java的Grader改回来

**private** **static** String *autoGraderClassName* = "nachos.ag.AutoGrader";

点运行，就能看到执行halt的结果啦



这里程序的运行似乎是在这里进行的

UserKernel.java

/\*\*

\* Start running user programs, by creating a process and running a shell

\* program in it. The name of the shell program it must run is returned by

\* <tt>Machine.getShellProgramName()</tt>.

\*

\* **@see** nachos.machine.Machine#getShellProgramName

\*/

**public** **void** run() {

**super**.run();

UserProcess process = UserProcess.*newUserProcess*();

String shellProgram = Machine.*getShellProgramName*();

Lib.*assertTrue*(process.execute(shellProgram, **new** String[] {}));

KThread.*finish*();

}

这里的String应该是要传点参数进去的，但为什么是空的？

**让echo.coff跑起来**

首先我们要给它一点参数

Lib.*assertTrue*(process.execute(shellProgram, **new** String[] {"wang", "miao"}));

接着我们需要去实现两个sysCall，请注意这两个我都是瞎写的

**private** **int** handleWrite(**int** fd, **int** buffer, **int** size) {

// ThreadedKernel.fileSystem.

**if** (fd == 0) {

**return** -1;

}

**if** (fd == 1) {

OpenFile file = UserKernel.*console*.openForWriting();

**byte**[] readB = **new** **byte**[size];

readVirtualMemory(buffer, readB, 0, size);

**int** succ = file.write(readB, 0, size);

file.close();

**return** succ;

}

**return** -1;

}

**private** **int** handleExit(**int** status) {

Machine.*halt*();

**return** status;

}

**public** **int** handleSyscall(**int** syscall, **int** a0, **int** a1, **int** a2, **int** a3) {

**switch** (syscall) {

**case** *syscallHalt*:

**return** handleHalt();

**case** *syscallWrite*:

**return** handleWrite(a0, a1, a2);

**case** *syscallExit*:

**return** handleExit(a0);

**default**:

Lib.*debug*(*dbgProcess*, "Unknown syscall " + syscall + " a0: " + a0 + " a1: " + a1 + " a2: " + a2);

Lib.*hdebug*("Unknown syscall " + syscall + " a0: " + a0 + " a1: " + a1 + " a2: " + a2);

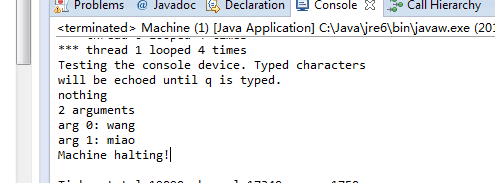
Lib.*assertNotReached*("Unknown system call!");

}

**return** 0;

}

然后一跑就出结果了

**、**

**文件系统坑爹醒目：**

关于上不上锁，open和close的问题，我认为说的最有道理的是王助教的一段话

首先：

陈博：

根据Phase2的原文“Do not implement any kind of file locking; this is the file system's responsibility.” 这一块是不用考虑锁的。

从最后评测的角度来看（你们看到Release的测试数据里的日期。。也应该知道是多少年没变过的了。。）

今年不必实现Phase5，也就是说。。。大家不必考虑文件系统的问题了。。。

第二：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | **Menghui Wang** wmhkebe@gmail.com | | 19:26 (11 分钟前)  https://mail.google.com/mail/images/cleardot.gif |  | **https://mail.google.com/mail/images/cleardot.gif**  **https://mail.google.com/mail/images/cleardot.gif** |
| |  | | --- | | 发送至 nachos-acm-sjtu  https://mail.google.com/mail/images/cleardot.gif | | | |

回忆了一下去年的情况，以下结论仅供你们参考

1. creat返回的fd是可读写的，不必要再open一次；

2. 但是creat之后再open也是允许的。换句话说，一个文件允许被open多次；

3. 既然允许被open多次，就允许被close多次，比如下面的代码合法：

int fd1, fd2;

fd1 = create("test");

fd2 = open("test");

close(fd1);

close(fd2);

  这里的fd1和fd2是否相同，取决于你们实现。我当时是返回不同的fd；  
4. 允许多个进程对同一个文件读写。这在主流操作系统也是允许的(主流操作系统在open的时候大多可以指定是否与其他进程共享这个文件；但是nachos比较简陋，可以认为永远允许)。但是同时读写产生的结果是未定义的；

5. 上面这条的意思是说同时读写同一个文件，read和write可以返回以任意顺序进行读写的返回值，但是不可以造成对文件系统的破坏。这是要在文件系统里做的。

仅供参考，仅供参考

我虔诚地按王助教说的的实现了一遍，感觉可能没问题

大家可以自己试下我给出的writeFile.coff（就是王助教的程序段）

**内存：**

UserProcessor里面本来是使用了全部内存的（假定只有一个process），现在我们要把它们全部改成使用分配的那些页。

所以大家写的时候要注意什么时候自己写的是virtual的，什么时候写的是physical的。

一个小提醒：稍微track一下就能发现，numPages才是所有要申请的内存页数，而不是stackpages。而numPages最后是在load的这个地方决定的

numPages += stackPages;

initialSP = numPages \* *pageSize*;

// and finally reserve 1 page for arguments

numPages++;

**多进程：**

注意UThread是继承kThread的，所以我觉得这里我们用的应该是phase1的东西。比如handleExit我觉得应该是这么写的。。。

**private** **int** handleExit(**int** status) {

UserKernel.*returnPages*(pages);

Iterator<Integer> iter = openedMy.iterator();

**while** (iter.hasNext())

*openFiles*.get(iter.next()).close();

hasExited = **true**;

exitStatus = status;

**if** (pid == *mainPid*)

Machine.*halt*();

KThread.*finish*();

**return** status;

}

另外，因为handleExec有些过于坑爹，我这里直接把代码给出（要理解argv是一个指针数组！）请为了我考虑，大家自行进行一些改动

byte[] buffer = new byte[4];

if (readVirtualMemory(argv + i \* 4, buffer) != buffer.length)

return -1;

int addr = Lib.bytesToInt(buffer, 0);

args[i] = readVirtualMemoryString(addr, 300);

Lib.hdebug(args[i]);

if (args[i] == null)

return -1;

同理，在handleJoin里你可能会用到

**byte**[] buf = Lib.*bytesFromInt*(st);

为了测试exec，我提供了一个cat121314.c和cat121314.coff给大家测试-x cat121314.coff，参数是cat, 12.txt。

我的输出是

222222222222222222222222222

2222222222222222222222222222211111111111131313131313131313131313131313

3

313131313131313131

1

13131313131313131313131313333333333

虽然跑起来了，但感觉不像是对的。。大家如果有输出不一样请告诉我。。。。

还有一个cat121314Join.c是测Join的，结果应该类似：

222222222222222222222222222

222222222222222222222222222223333333333333333333333333

333333333333333333333333333333333311111111111111111111111111

11111111111111111111111

数据readLineAndWrite.c是一个简单的从屏幕读入一行并输出（会输出两次）的测试。