#### 操作系统课程设计报告-实验二

# MIPS环境的配置

## 1.在虚拟机中配置环境

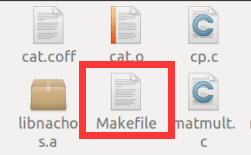
在Ubuntu虚拟机中配置MIPS交叉环境。由于实验需要用到MIPS交叉编译环境，而Windows中对于如何配置MIPS交叉环境的资料很少，但是有关Linux系统上的配置有一大堆，并且上学期使用过Ubuntu虚拟机，于是采用在Linux上配置环境。

首先把mips-x86.linux-xgcc.tgz解压到nachos目录下，注意要使用命令行的方式进行解压而不能直接解压缩，否则有可能会出现install 的error。进入nachos/test目录下，配置环境变量archdir

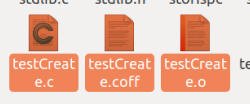
# export ARCHDIR=../mips-x86.linux-xgcc  
# export PATH=/root/usr/wp/nachos/mips-x86.linux-xgcc:$PATH

这样mips的环境就配置好了，如果需要使用mips只需要将编写的.c文件复制到test目录下。同时修改Makefile文件在该代码行处，添加C代码文件的名称。





进入终端，先执行make clean命令，在执行make命令即可得到对应的o文件和coff文件。



这时将编译出.coff文件，将该文件复制到eclipse中，相应的方法便可以使用了。

## 2. Nachos MIPS模拟器

（1）MIPS CPU模拟器

模拟的MIPS机器实际上只是Nachos发行版中的一个大程序。此过程了解MIPS指令的格式以及MIPS体系结构定义的那些指令的预期行为。当MIPS模拟器执行``用户程序''时，它会通过执行紧密循环，从模拟的机器存储器中获取MIPS指令并通过转换模拟的状态来``执行''来模拟真实的MIPS CPU的行为根据MIPS体系结构规范中指令的定义含义，存储和模拟机器寄存器。模拟计算机的物理内存和寄存器是 nachos 程序中的数据结构。

（2）内核与机器之间的交互

您的Nachos内核可以控制仿真机器的方式与真实内核控制真实机器的方式相同。就像真实计算机上的真实内核一样，您的内核可以指示模拟计算机在用户模式下的特定内存地址处开始执行代码。 如果用户程序执行系统调用陷阱指令，或者发生中断或其他机器异常，则机器将控制权返回给内核（通过调用Nachos内核过程 ExceptionHandler）。

像真实的内核一样，您的Nachos内核必须检查并修改机器寄存器和其他机器状态，以便为异常提供服务并运行用户程序。例如，系统调用参数和结果通过计算机的寄存器在用户程序和内核之间传递。您的内核还将检查和修改 页表 数据结构，该表表数据结构将由模拟机用来转换当前用户程序中的虚拟地址。从Nachos内核的角度来看，所有机器状态-寄存器，内存和页表-都是内核地址空间中的简单数组，可通过 Machine 对象访问。

（3）I / O设备，中断和控制台

Nachos发行版扩展了MIPS CPU仿真器，以仿真某些设备，例如磁盘，计时器和控制台。Nachos维护排定要发生的中断队列（例如，完成未完成的磁盘操作），并通过在适当的时间调用内核中断处理程序来模拟这些中断的传递。

为什么启用控制台时Nachos挂起？ 当前版本的Nachos具有令人讨厌的``功能''，这给某些团队带来了麻烦。即使没有线程或进程可以运行，Nachos内核也不会关闭，除非有待处理的I / O操作。这是您所期望的行为，但是Nachos通过使用中断队列反复轮询控制台上键入的字符来模拟控制台，因此控制台上始终存在未决的I / O操作。这意味着，如果您 为以后的任务分配所需的 控制台对象，则Nachos在运行完测试程序后将永远不会关闭。相反，它将像真实内核一样空闲，等待控制台输入。随意使用 ctrl-C杀死它 。保持闲置的Nachos进程运行很不好，因为它们会占用大量CPU时间。

# 设计实现

## Task2.1文件系统调用

### 实验要求

* (30%, 125 lines)

Implement the file system calls (creat, open, read, write, close, and unlink, documented in syscall.h). You will see the code for halt in UserProcess.java; it is best for you to place your new system calls here too. Note that you are *not* implementing a file system; rather, you are simply giving user processes the ability to access a file system that we have implemented for you.

* We have provided you the assembly code necessary to invoke system calls from user programs (see start.s; the SYSCALLSTUB macro generates assembly code for each syscall).
* You will need to *bullet-proof* the Nachos kernel from user program errors; there should be nothing a user program can do to crash the operating system (with the exception of explicitly invoking the halt()syscall). In other words, you must be sure that user programs do not pass bogus arguments to the kernel which causes the kernel to corrupt its internal state or that of other processes. Also, you must take steps to ensure that if a user process does anything illegal -- causing the Processor to throw an exception -- that the process will be killed cleanly and its resources freed.
* You should make it so that the halt() system call can only be invoked by the "root" process -- that is, the first process in the system. If another process attempts to invoke halt(), the system call should be ignored and return immediately.
* Since the memory addresses passed as arguments to the system calls are virtual addresses, you need to use UserProcess.readVirtualMemory and UserProcess.writeVirtualMemory to transfer memory between the user process and the kernel.
* User processes store filenames and other string arguments as null-terminated strings in their virtual address space. The maximum length of for strings passed as arguments to system calls is 256 bytes.
* When a system call wishes to indicate an error condition to the user, it should return -1 (**not** throw an exception within the kernel!). Otherwise, the system call should return the appropriate value as documented in test/syscall.h.
* When any process is started, its file descriptors 0 and 1 must refer to standard input and standard output. Use UserKernel.console.openForReading() and UserKernel.console.openForWriting() to make this easier. A user process*is* allowed to close these descriptors, just like descriptors returned by open(). Do not worry about reopening them if they are closed.
* A stub file system interface to the UNIX file system is already provided for you; the interface is given by the class machine/FileSystem.java. You can access the stub filesystem through the static field ThreadedKernel.fileSystem. (Note that since UserKernel extends ThreadedKernel, you can still access this field.) This filesystem is capable of accessing the test directory in your Nachos distribution, which is going to be useful when you want to support the exec system call (see below). You do not need to implement any file system functionality. You should examine carefully the specifications for FileSystem and StubFileSystem in order to determine what functionality you should provide in your syscalls, and what is handled by the file system.
* Do not implement any kind of file locking; this is the file system's responsibility. If ThreadedKernel.fileSystem.open() returns a non-null OpenFile, then the user process is allowed to access the given file; otherwise, you should signal an error. Likewise, you do not need to worry about the details of what happens if multiple processes attempt to access the same file at once; the stub filesystem handles these details for you.
* Your implementation should support up to 16 concurrently open files per process, including stdin and stdout. Each file that a process has opened should have a unique file descriptor associated with it (see syscall.h for details). The file descriptor should be a non-negative integer that is simply used to index into a table of currently-open files by that process. Note that a given file descriptor can be reused if the file associated with it is closed, and that different processes can use the same file descriptor (i.e. integer) to refer to different files.

实现文件系统调用（creat, open, read, write, close, and unlink，记录在syscall.h中）。您将在UserProcess.java中看到the code for halt;最好也将您的新系统调用放在这里。请注意，您没有实现文件系统;相反，您只是在为用户进程提供访问我们为您实现的文件系统的能力。

* 我们已为您提供了从用户程序调用系统调用所必需的汇编代码（请参阅start.s; SYSCALLSTUB宏为每个syscall生成汇编代码）。
* 您需要从用户程序错误中防止Nachos内核成为子弹;用户程序不能执行任何操作来使操作系统崩溃（除了显式调用halt（）syscall之外）。换句话说，必须确保用户程序不会将虚假参数传递给内核，这会导致内核破坏其内部状态或其他进程的内部状态。另外，您必须采取措施确保用户进程执行任何非法操作（导致处理器抛出异常），以确保该进程将被彻底杀死并释放其资源。
* 您应该这样做，使得halt（）系统调用只能由“ root”进程（即系统中的第一个进程）调用。如果另一个进程尝试调用halt（），则应忽略系统调用并立即返回。
* 由于作为参数传递给系统调用**的内存地址是虚拟地址**，因此需要使用UserProcess.readVirtualMemory和UserProcess.writeVirtualMemory在用户进程和内核之间传输内存。
* 用户进程在其虚拟地址空间中将文件名和其他字符串参数存储为以空字符结尾的字符串。作为参数传递给系统调用的字符串的最大长度为256个字节。
* 当系统调用希望向用户指示错误情况时，它应返回-1（不要在内核中引发异常！）。否则，系统调用应返回test / syscall.h中记录的适当值。
* 启动任何进程时，其文件描述符0和1必须引用标准输入和标准输出。使用UserKernel.console.openForReading（）和UserKernel.console.openForWriting（）可简化此过程。用户进程可以关闭这些描述符，就像open（）返回的描述符一样。如果它们已关闭，请不要担心会重新打开它们。
* to已经为您提供了UNIX文件系统的存根文件系统接口；接口由machine / FileSystem.java类提供。您可以通过静态字段ThreadedKernel.fileSystem访问存根文件系统。 （请注意，由于UserKernel扩展了ThreadedKernel，因此您仍然可以访问此字段。）此文件系统能够访问Nachos发行版中的test目录，这在您要支持exec系统调用时将很有用（请参见下文）。您不需要实现任何文件系统功能。您应仔细检查FileSystem和StubFileSystem的规范，以确定应在系统调用中提供哪些功能以及文件系统要处理的功能。
* 请勿执行任何类型的文件锁定；这是文件系统的责任。如果ThreadedKernel.fileSystem.open（）返回一个非null的OpenFile，则允许用户进程访问给定的文件;否则，您应该发出错误信号。同样，您无需担心如果多个进程试图一次访问同一文件会发生什么情况的细节；存根文件系统会为您处理这些详细信息。
* implementation您的实现应支持每个进程最多16个同时打开的文件，包括stdin和stdout。进程已打开的每个文件都应具有唯一的文件描述符（有关详细信息，请参见syscall.h）。文件描述符应该是一个非负整数，该整数仅用于通过该进程索引到当前打开的文件表中。请注意，如果给定的文件描述符关闭了，则可以重用该文件描述符，并且不同的进程可以使用相同的文件描述符（即整数）来引用不同的文件。

### 设计方案

由题目要求可知，文件系统已经实现，我们要做的只是理解并学会利用这个文件系统。要确保如下几点：① 稳定性：不能因为一个进程的非法系统调用就使操作系统崩溃，而应该返回错误代码； ② halt 系统调用只能由第一个进程(根进程) 执行； ③ 系统调用需要读写内存时,通过UserProcess.readVirtualMemory 和 UserProcess.writeVirtualMemory进行；④ 文件名以 nu1l 结尾，不超过 256 字符； ⑤如果系统调用出错，应返回-1； ⑥ 为每个打开的IO文件分配一个“文件描述符”，用整数表示，同时题目要求每个进程最多16个同时打开的文件，其中0和1应分配给标准输入和标准输出，由SynchConsele类管理.不同进程可以用相同的文件描述符处理不同的文件； ⑦ nachos已经提供了一个简单的文件系统Machine.FileSystem，可以通过ThreadedKernel.fileSystem访问； ⑧ 系统不需要考虑文件访问的互斥等问题 。

2.1 halt()

为每个进程设置一个进程号ID，只有进程号为0的进程（也就是第一个进程root进程）才能调用halt,然后会停机。

2.2 create（）

先从内存中将文件名读出，然后利用文件系统的打开文件。利用不存在就创建的方法，在物理磁盘中创建文件。

1.根据传入的参数fileAddress，从内存中读出文件名；

2.判断文件名是否存在，即fileAddress是否为空；

3.如果不存在则创建一个文件；

先去找合适的位置，如果已经有大于16个文件，则退出；

小于16个文件，则创建；

创建的时候调用FileSystem中的open函数（传入的布尔值create为true，就会创建文件）

4.将fileAddress对应的openfile数组的位置放入刚刚创建的文件描述符

2.3 open（）

先从内存中将文件名读出，然后利用文件系统的打开文件。不存在直接返回null， 仅打开。

1.根据传入的参数fileAddress，从内存中读出文件名；

2.参数fileAddress不能为空，需要检查；

3.在openfile中找到合适的位置，如果大于16个表明打开文件达到上限，退出；否则打开对应的文件，调用FileSystem中的open（传入布尔值create为false，即只是打开文件）；

4.将fileAddress对应的openfile数组的位置放入刚刚打开的文件描述符。

2.4 read（）

使用打开文件描述符，利用文件系统的读方法将数据从文件中读数据到数组中，然后使用内存写操作，写入内存，然后返回写入的数量。

  read系统调用应该给出要read的文件、准备写入的虚拟内存地址、读取的字节数（写入的字节数）。

1.如果打开文件数目大于16或小于0、对应fileDescriptor的openfile为空都为异常情况直接退出；

2.将文件内容读出到暂存数组temp[]，并返回读出字节的个数；

3.如果返回的读出字节的个数小于等于0,表明没有读出数据，出错，返回；

4.若大于0，则将读出的内容写到虚拟内存地址，返回写成功的字节数。

2.5 write（）

使用打开文件描述符，利用内存读操作将数据从内存中读到数组中，然后利用文件系统的写方法，写入文件，然后返回写入文件的数量。

  write系统调用应该给出要write的文件、准备读出的虚拟内存地址、写入的字节数。

1.如果打开文件数目大于16或小于0、对应fileDescriptor的openfile为空都为异常情况直接退出；

2.将虚拟内存地址中的数据读出到暂存数组temp[]，并返回读出字节的个数；

3.如果返回的字节个数小于等于0，表明没有读出数据，出错，返回；

4.若大于0，将读出的内容写入文件，返回写成功的字节数；

5.如果写成功的字节数小于要写的字节数，仍是错误情况，返回-1；

2.6 close（）

使用打开文件描述符，将文件描述符指向的文件利用文件系统的关闭方法关闭。

close系统调用应该给出需要关闭的文件的文件描述符。

1.如果打开文件数目大于16或小于0、对应fileDescriptor的openfile为空都为异常情况直接退出；

2.关闭openfile数组中的文件，并把此位置置为null。

2.7 unlink（）

用于删除文件，先从内存中将文件名读出，然后利用文件系统的删除操作将文件从物理磁盘中删除。

1.根据传入的文件地址参数fileAddress，从虚拟内存中读出文件名

2.如果不存在不必删除

3.如果存在，就删除文件

### 3.实现代码

3.1声明变量：

  // 2.1 start  
protected int pid = 0;// 这是UserKernel中的相当于进程计数  
protected OpenFile openfile[] = new OpenFile[16];// 每个进程最多16个同时打开的文件  
// 2.1 end

3.2方法实现：

/\*\*  
​  
 \* Handle the halt() system call. 处理halt（）系统调用。  
   \*/  
 private int handleHalt() {   
// 2.1 start  
if (pid == 0)// 只有root进程才能停机  
// 2.1 end  
Machine.halt();  
​  
Lib.assertNotReached("Machine.halt() did not halt machine!");  
return 0;  
}  
​  
// 2.1 start  
/\*  
\* create(): 先从内存中将文件名读出，然后利用文件系统的打开文件。   
\* 利用不存在就创建的方法，在物理磁盘中创建文件。  
\*   
\* 文件名存在--打开文件数目没有到达16--可以create  
\*/  
private int handleCreate(int fileAddress) {  
// 处理create()系统调用，创建一个文件，返回文件描述符  
// 读出文件名  
String filename = readVirtualMemoryString(fileAddress, 256);// 文件名长度不得超过256字符  
if (filename == null)  
return -1; // 文件名不存在，创建失败  
// 在openfile中找空的位置  
int fileDescriptor = findEmpty();  
if (fileDescriptor == -1)  
return -1; // fileDescriptor=-1进程打开文件数已经达到上限，无法创建并打开  
  
/\* 执行到此处fileDescriptor=openfile为空位的下标，可以创建文件 \*/  
// 创建  
else {  
// 创建一个文件，并且将此时的openfile[fileDescriptor]中放入相应的描述符  
openfile[fileDescriptor] = ThreadedKernel.fileSystem.open(filename, true);// 文件不存在直接创建  
// 返回文件描述符  
return fileDescriptor;  
}  
}  
​  
/\*  
\* open（）： 先从内存中将文件名读出，然后利用文件系统的打开文件。 利用不存在直接返回null，仅打开  
\*   
\* 文件名存在--打开文件数目没有到达16--可以open  
\*/  
private int handleOpen(int fileAddress) {  
// 处理open()的系统调用，打开一个文件  
String filename = readVirtualMemoryString(fileAddress, 256);  
if (filename == null)  
return -1; // 文件名不存在  
// 在openfile中找空的位置  
int fileDescriptor = findEmpty();  
if (fileDescriptor == -1)  
return -1; // 打开文件数已经达到上限，无法打开  
/\* 执行到此处说明可以打开文件 \*/  
else {  
openfile[fileDescriptor] = ThreadedKernel.fileSystem.open(filename, false);  
return fileDescriptor;// 打开成功返回文件描述符  
}  
}  
​  
/\*  
\* read(): 使用打开文件描述符，利用文件系统的读方法将数据 从文件中 读到数组中， 然后使用内存写操作，写入内存。返回写入的数量 int  
\* fileDescriptor:利用文件描述符-打开文件-读出length字节 int bufferAddress:使用内存写操作，写入内存 int  
\* length:要读取的字节数，要写入的字节数 return : 写入的字节数目。  
\*/  
private int handleRead(int fileDescriptor, int bufferAddress, int length) {  
// 处理read()的系统调用，从文件中读出数据写入 指定虚拟地址  
// 检查给定的文件描述符  
if (fileDescriptor > 15 || fileDescriptor < 0 || openfile[fileDescriptor] == null)  
return -1; // 文件未打开，出错  
byte temp[] = new byte[length];  
// 读文件  
int readNumber = openfile[fileDescriptor].read(temp, 0, length);  
if (readNumber <= 0)  
return 0; // 没有读出数据  
int writeNumber = writeVirtualMemory(bufferAddress, temp);  
return writeNumber;  
}  
​  
/\*  
\* write(): 使用打开文件描述符，利用内存读操作将数据从内存中读到数组中， 然后使用文件写操作，写入文件。返回写入的数量  
\*/  
private int handleWrite(int fileDescriptor, int bufferAddress, int length) {  
// 处理write()的系统调用，将指定虚拟内存地址的数据写入文件  
if (fileDescriptor > 15 || fileDescriptor < 0 || openfile[fileDescriptor] == null)  
return -1; // 文件未打开，出错  
byte temp[] = new byte[length];  
// 读出虚拟内存地址中的数据到temp中  
int readNumber = readVirtualMemory(bufferAddress, temp);  
// 数据读出后保存在temp中  
if (readNumber <= 0)  
return 0; // 未读出数据  
// 将Temp中的数据 写入文件  
int writeNumber = openfile[fileDescriptor].write(temp, 0, length);  
if (writeNumber < length)  
return -1;// 未完全写入，出错  
return writeNumber;  
}  
​  
/\*  
\* close(): 使用文件描述符，将文件描述符指向的文件 利用文件系统的关闭方法关闭  
\*   
\*/  
private int handleClose(int fileDescriptor) {  
// 处理close()的系统调用，用于关闭打开的文件  
if (fileDescriptor > 15 || fileDescriptor < 0 || openfile[fileDescriptor] == null)  
return -1; // 文件不存在，关闭出错  
openfile[fileDescriptor].close();  
openfile[fileDescriptor] = null;  
return 0;  
}  
​  
/\*  
\* unlink(): 先从内存中将文件名读出，利用文件系统的删除操作将文件从物理磁盘中删除  
\*/  
private int handleUnlink(int fileAddress) {  
// 处理unlink的系统调用，用于删除文件  
// 获得文件名  
String filename = readVirtualMemoryString(fileAddress, 256);  
if (filename == null)  
return 0; // 文件不存在，不必删除  
if (ThreadedKernel.fileSystem.remove(filename))// 删除磁盘中实际的文件  
return 0;// 成功删除  
else  
return -1;  
}  
​  
/\*\*  
\* function: 从openfile中找到一个空的文件描述符位。  
\*   
\* @return 数组下标,如果没有空的，则返回-1  
\*/  
private int findEmpty() {  
for (int i = 0; i < 16; i++) {  
if (openfile[i] == null)  
return i;  
}  
return -1;  
}  
// 2.1 end  
​  
​  
​

### 4.测试

Task2.1与Task2.2与Task2.3的测试合并，详见Task2.3的测试

## Task2.2 多进程内存分配及访问

### 实验要求

* (25%, 100 lines) Implement support for multiprogramming. The code we have given you is restricted to running one user process at a time; your job is to make it work for multiple user processes.
* Come up with a way of allocating the machine's physical memory so that different processes do not overlap in their memory usage. Note that the user programs do not make use of malloc() or free(), meaning that user programs effectively have no dynamic memory allocation needs (and therefore, no heap). What this means is that you know the complete memory needs of a process when it is created. You can allocate a fixed number of pages for the processe's stack; 8 pages should be sufficient.
* We suggest maintaining a global linked list of free physical pages (perhaps as part of the UserKernel class). Be sure to use synchronization where necessary when accessing this list. Your solution must make efficient use of memory by allocating pages for the new process wherever possible. This means that it is not acceptable to only allocate pages in a contiguous block; your solution must be able to make use of "gaps" in the free memory pool.
* Also be sure that all of a process's memory is freed on exit (whether it exits normally, via the syscall exit(), or abnormally, due to an illegal operation).
* Modify UserProcess.readVirtualMemory and UserProcess.writeVirtualMemory, which copy data between the kernel and the user's virtual address space, so that they work with multiple user processes.
* The physical memory of the MIPS machine is accessed through the method Machine.processor().getMemory(); the total number of physical pages is Machine.processor().getNumPhysPages(). You should maintain the pageTable for each user process, which maps the user's virtual addresses to physical addresses. The TranslationEntry class represents a single virtual-to-physical page translation.
* The field TranslationEntry.readOnly should be set to true if the page is coming from a COFF section which is marked as read-only. You can determine this using the method CoffSection.isReadOnly().
* Note that these methods should not throw exceptions when they fail; instead, they must always return the number of bytes transferred (even if that number is zero).
* Modify UserProcess.loadSections() so that it allocates the number of pages that it needs (that is, based on the size of the user program), using the allocation policy that you decided upon above. This method should also set up the pageTable structure for the process so that the process is loaded into the correct physical memory pages. If the new user process cannot fit into physical memory, exec() should return an error.

（25％，100行）实现对多程序的支持。我们提供给您的代码仅限于一次运行一个用户进程。您的工作是使其可用于多个用户进程。

* 提出一种**分配机器物理内存的方法**，以使不同的进程在内存使用上不会重叠。请注意，用户程序不使用malloc（）或free（），这意味着用户程序实际上没有动态内存分配需求（因此也没有堆）。这意味着您知道一个进程在创建时的完整内存需求。您可以为**进程的堆栈分配固定数量的页面。 8页就足够了。**
* 我们建议维护一个免费物理页面的**全局链接列表（可能是UserKernel类的一部分）。访问此列表时，请确保在必要时使用同步。**您的解决方案必须通过尽可能为新进程分配页面来有效利用内存。这意味着仅在连续的块中分配页面是不可接受的。您的解决方案必须能够利用空闲内存池中的“间隙”。
* 另外，请确保退出时释放了所有进程的内存（无论是通过syscall exit（）正常退出，还是由于非法操作而异常退出）。
* 修改UserProcess.readVirtualMemory和UserProcess.writeVirtualMemory，它们在内核和用户的虚拟地址空间之间复制数据，以便它们可以与多个用户进程一起使用。
* 通过方法Machine.processor（）.getMemory（）;访问MIPS机器的物理内存。物理页面的总数为Machine.processor（）.getNumPhysPages（）。您应该为每个用户进程维护pageTable，该表将用户的虚拟地址映射到物理地址。 TranslationEntry类表示单个虚拟到物理页面的翻译。
* 如果页面来自标记为只读的COFF部分，则应将TranslationEntry.readOnly字段设置为true。您可以使用方法CoffSection.isReadOnly（）来确定。
* 请注意，这些方法在失败时不应引发异常。相反，它们必须始终返回传输的字节数（即使该数字为零）。
* 修改UserProcess.loadSections（），以使用上面确定的分配策略分配所需的页数（即，基于用户程序的大小）。此方法还应该为进程设置pageTable结构，以便将进程加载到正确的物理内存页中。如果新用户进程无法放入物理内存，则exec（）应该返回错误。

### 设计方案

本题目要求我们实现多进程内存分配及访问，因为 Nachos 支持多道程序设计，由题目要求，我们应该做的是保证不同的进程在内存使用上不会重叠。

由题目可知，Nachos 不支持动态内存分配，故在装入程序时就要为每个进程一次性分配固定的物握内存，在进程结束时收回它们。

我们需要实现简单的虚拟内存方案：每个进程的地址空间是连续的虚拟内存(以页面为单位,1 页=1024 字节)，但这些连续的虚拟页在物理内存中却不一定是连续的；这个方案很简单，根据题目要求，我们要为每个用户进程维护pageTable，该页表将用户的虚拟地址映射到物理地址，TranslationEntry类表示单个虚拟页到物理页面的翻译，映射机制只是从虚拟内存到物理内存的一一映射。同时，我们还要维护一个全局空闲物理页面的链表。

2.1 准备

1.系统提供页表pageTable，它以虚拟页号为下标，pageTable里的每一个元素都是一个machine.TranslationEntry类型的对象。

2.声明一个全局列表，存放当前空闲的物理页号。

3.需要加载的用户进程是一个machine.Coff类型的对象,它包括许多**段(section),每一段是一个machine.CoffSection类型的对象,每一段又包含了很多页。**

4.每启动一个新的用户进程，UserProcess.load()都负责将具有指定名称的可执行文件加载到此进程中，并准备将指定的参数传递给它。打开可执行文件，读取它的头信息，并将部分和参数复制到这个进程的虚拟内存中。load()函数nachos以及给出，我们需要完善的是loadSections()函数，它是load()函数一部分。

5.load()首先确保文件存在，然后转为coff类型对象，然后make sure the sections are contiguous （连续）and start at page 0，然后make sure the argv array will fit in one page，然后调用loadSections()，成功后再装入参数。

2.2 **boolean** loadSections()

@function:为进程分配内存

numPages中保存了进程执行所需要的页的数量（在load()中已经计算出来）

1. 判断需要的页的数量与物理内存页的数量的大小关系，只有物理页数量大于需要的页

数量才能进行内存分配，否则内存分配失败，返回false。（nachos已经给出）

1. 实例化页表pageTable数组（TranslationEntry对象类型）。

3. 从空闲物理页表号链表（memoryLinkedList）中取出一个空闲物理页号。

4. 在页表数组中实例化一个TranslationEntry对象，保存关联的虚拟页号与物理页号及相关信息。

5. 加载coffSection，对于每个section对象有很多的页，内层for循环对于每一页的页号，然后将它标记为只读，然后装入页（根据物理地址装入页）。

2.3 **int** readVirtualMemory(**int** vaddr, **byte**[] data, **int** offset,**int** length)

@function：从进程的虚拟内存中将数据复制到指定的数组，这个方法处理地址转换的细节。这个方法在发生错误时，不能损坏当前进程，但是应该返回成功复制的字节的数量，如果没有数据复制返回0.

1.计算从vaddr该虚拟内存首地址开始的页表剩余字节的个数，若总共剩下的字节< length,那么length最大也就是剩下的字节数，若总共剩下的字节 > or = length,那么 length 就是 length。

2.计算data数组中能否存下 length长度的数据，如果存不下则减小length，即将数据传输大小设置为可以存的大小。

3.检查完合法性，开始进行数据传输：因为可能涉及到翻页，所以需要用do-while循环。

3.1 通过给定首地址和已传输的数据的字节长度的加和计算对应的页号和页偏移量，注意要判断页号的合法性（页号大于页表的长度或者为负是异常情况）；

3.2 根据页偏移量计算当前页号的页剩余的页的容量

3.3 将要传输的数据字节的长度：页剩余字节和剩余传输数据字节中的较小值

3.4 计算物理内存的地址，虚拟页号对应的物理页号\*页大小+页偏移量

3.5 将物理内存的东西传输到data数组

3.6 增加变量transferredbyte传输成功的字节数

3.7 transferredbyte<length则继续循环操作

2.4 **int** writeVirtualMemory(**int** vaddr, **byte**[] data, **int** offset,**int** length)

@function：将指定的数组的数据复制到进程的虚拟内存中，这个方法处理地址转换的细节。这个方法在发生错误时，不能损坏当前进程，但是应该返回成功复制的字节的数量，如果没有数据复制返回0.

与3.3 readVirtualMemory()几乎一模一样，不同点仅在于 3.5 将物理内存的东西传输到data数组，本函数应该是：从data写入到物理内存

### 3.代码实现：

3.1声明变量：

// 2.2start

public static Lock allocateMemoryLock = null;

public static LinkedList memoryLinkedList = null;

// 2.2end

/\*\*

\* Initialize this kernel. Creates a synchronized console and sets the

\* processor's exception handler.

\*/

public void initialize(String[] args) {

// 2.2 start

allocateMemoryLock = new Lock();

// 初始化的时候，使memoryLinkedList包含所有的页号

memoryLinkedList = new LinkedList<Integer>();

for (int i = 0; i < 1024; i++) {

memoryLinkedList.add(i);

}

// 2.2 end

super.initialize(args);

console = new SynchConsole(Machine.console());

Machine.processor().setExceptionHandler(new Runnable() {

public void run() {

exceptionHandler();

}

});

}

3.2方法实现：

/\*\*

\* Allocate a new process.

\*/

public UserProcess() {

//2.3start

pid=this.numberOfProcess++;

this.numberOfProcess++;

openfile[0]=UserKernel.console.openForReading();

openfile[1]=UserKernel.console.openForWriting();

//2.3end

int numPhysPages = Machine.processor().getNumPhysPages();

pageTable = new TranslationEntry[numPhysPages];

for (int i = 0; i < numPhysPages; i++)

pageTable[i] = new TranslationEntry(i, i, true, false, false, false);

}

protected boolean loadSections() {

// 2.2 start

UserKernel.allocateMemoryLock.acquire();// 获取分配的内存的锁

// 2.2 end allocateMemoryLock

if (numPages > Machine.processor().getNumPhysPages()) {// 页数量大于实际物理内存的页数量

coff.close();

Lib.debug(dbgProcess, "\tinsufficient physical memory");

return false;

}

// 2.2 start

pageTable = new TranslationEntry[numPages];// 实例化页表

// 一共numpages个页，要把页 装入到 物理地址

for (int i = 0; i < numPages; i++) {

// 从空闲物理页号链表中拿出一个

int nextPage = (int) UserKernel.memoryLinkedList.remove();

// virtual page number,physical page number,valid,readOnly,used,dirty

pageTable[i] = new TranslationEntry(i, nextPage, true, false, false, false);

}

UserKernel.allocateMemoryLock.release();// 释放分配的内存的锁

// 2.2 end

// load sections

for (int s = 0; s < coff.getNumSections(); s++) {

CoffSection section = coff.getSection(s);

Lib.debug(dbgProcess,

"\tinitializing " + section.getName() + " section (" + section.getLength() + " pages)");

for (int i = 0; i < section.getLength(); i++) {

int vpn = section.getFirstVPN() + i;

// 2.2

pageTable[vpn].readOnly = section.isReadOnly();// 标记为只读

// for now, just assume virtual addresses=physical addresses

// 装入物理页

section.loadPage(i, pageTable[vpn].ppn);

// 2.2

// for now, just assume virtual addresses=physical addresses

// section.loadPage(i, vpn);

}

}

return true;

}

public int readVirtualMemory(int vaddr, byte[] data, int offset, int length) {

Lib.assertTrue(offset >= 0 && length >= 0 && offset + length <= data.length);

byte[] memory = Machine.processor().getMemory();// 获取物理内存的引用

// 2.2 start

// 计算剩下的页表字节个数 (再次强调，vaddr:要读的虚拟内存的首字节)

// length是要从虚拟内存读出到数组的字节数，

// 总共剩下的字节 小于< length,那么length最大也就是剩下的字节数

// 总共剩下的字节 > or = length,那么 length 就是 length

if (length > (pageSize \* numPages - vaddr))

length = pageSize \* numPages - vaddr;

// 计算能够传输的数据的大小，如果data数组中存不下length，则减小length（传输字节数）

// 疑问？前面不是已经保证 偏移量offset+长度length<=总的数据 数组 长度 了吗

// if (data.length - offset < length)

// length = data.length - offset;

// 转换成功的字节数

int transferredbyte = 0;

do {

// 计算页号

int pageNum = Processor.pageFromAddress(vaddr + transferredbyte);

// 页号大于 页表的长度 或者 为负 是异常情况

if (pageNum < 0 || pageNum >= pageTable.length)

return 0;

// 计算页偏移量

int pageOffset = Processor.offsetFromAddress(vaddr + transferredbyte);

// 计算剩余页的容量

int leftByte = pageSize - pageOffset;

// 计算下一次传送的数量:剩余页容量和需要转移的字节数中较小者

int amount = Math.min(leftByte, length - transferredbyte);

// 计算物理内存的地址

int realAddress = pageTable[pageNum].ppn \* pageSize + pageOffset;

// 将物理内存的东西传输到虚拟内存

System.arraycopy(memory, realAddress, data, offset + transferredbyte, amount);

// 修改传输成功的字节数

transferredbyte = transferredbyte + amount;

} while (transferredbyte < length);

return transferredbyte;

// 2.2 end

// // for now, just assume that virtual addresses equal physical addresses

// if (vaddr < 0 || vaddr >= memory.length)

// return 0;

//

// int amount = Math.min(length, memory.length - vaddr);

// System.arraycopy(memory, vaddr, data, offset, amount);

//

// return amount;

}

public int writeVirtualMemory(int vaddr, byte[] data, int offset, int length) {

Lib.assertTrue(offset >= 0 && length >= 0 && offset + length <= data.length);

byte[] memory = Machine.processor().getMemory();

// 2.2 start

// 写内存的长度如果超过页剩余量,就要以小的页剩余量为准，不然要写入的length太长了，页剩余量不够，也写不进去啊

if (length > (pageSize \* numPages - vaddr))

length = pageSize \* numPages - vaddr;

// 如果数组中要写的长度比给定的小，则给length减为数组剩余的长度

// 同样的疑问？前面不是已经保证 偏移量offset+长度length<=总的数据 数组 长度 了吗

if (data.length - offset < length)

length = data.length - offset;

// 转换成功的字节数

int transferredbyte = 0;

do {

// 此函数返回给定地址的页号

int pageNum = Processor.pageFromAddress(vaddr + transferredbyte);

if (pageNum < 0 || pageNum >= pageTable.length)

return 0;

// 此函数返回给定地址的页偏移量

int pageOffset = Processor.offsetFromAddress(vaddr + transferredbyte);

// 页剩余的字节数

int leftByte = pageSize - pageOffset;

// 设置本次转移的数量

int amount = Math.min(leftByte, length - transferredbyte);

int realAddress = pageTable[pageNum].ppn \* pageSize + pageOffset;

// 从虚拟内存写入到物理内存

System.arraycopy(data, offset + transferredbyte, memory, realAddress, amount);

// 改变写成功的字节数

transferredbyte = transferredbyte + amount;

} while (transferredbyte < length);

return transferredbyte;

// 2.2 end

// // for now, just assume that virtual addresses equal physical addresses

// if (vaddr < 0 || vaddr >= memory.length)

// return 0;

//

// int amount = Math.min(length, memory.length - vaddr);

// System.arraycopy(data, offset, memory, vaddr, amount);

//

// return amount;

}

### 4.测试

Task2.1与Task2.2与Task2.3的测试合并，详见Task2.3的测试

## Task2.3 系统调用

### 实验要求

* (30%, 125 lines)

Implement the system calls (exec, join, and exit, also documented in syscall.h).

* Again, all the addresses passed in registers to exec and join are virtual addresses. You should use readVirtualMemory and readVirtualMemoryString to transfer memory between the kernel and the user process.
* Again, you must bullet-proof these syscalls.
* Note that the state of the child process is entirely private to this process. This means that the parent and child do not directly share memory or file descriptors. Note that two processes can of course open the same file; for example, all processes should have file descriptors 0 and 1 mapped to the system console, as described above.
* Unlike KThread.join(), only a process's parent can join to it. For instance, if A executes B and B executes C, A is not allowed to join to C, but B is allowed to join to C.
* join takes a process ID as an argument, used to uniquely identify the child process which the parent wishes to join with. The process ID should be a globally unique positive integer, assigned to each process when it is created. (Although for this project the only use of the process ID is in join, for later project phases it is important that the process ID is unique across all running processes in the system.) The easiest way of accomplishing this is to maintain a static counter which indicates the next process ID to assign. Since the process ID is an int, then it may be possible for this value to overflow if there are many processes in the system. For this project you are not expected to deal with this case; that is, assume that the process ID counter will not overflow.
* When a process calls exit(), its thread should be terminated immediately, and the process should clean up any state associated with it (i.e. free up memory, close open files, etc). Perform the same cleanup if a process exits abnormally.
* The exit status of the exiting process should be transferred to the parent, in case the parent calls the join system call. The exit status of a process that exits abnormally is up to you. For the purposes of join, a child process exits normally if it calls the exit syscall with any status, and abnormally if the kernel kills it (e.g. due to an unhandled exception).
* The last process to call exit() should cause the machine to halt by calling Kernel.kernel.terminate(). (Note that only the root process should be allowed to invoke the halt() system call, but the last exiting process should call Kernel.kernel.terminate() directly.)
* （30％，125行）实现系统调用（exec，join和exit，也记录在syscall.h中）。
* 同样，在寄存器中传递给exec和join的所有地址都是虚拟地址。您应该使用readVirtualMemory和readVirtualMemoryString在内核和用户进程之间传输内存。

同样，您必须对这些系统调用进行防弹。

* 请注意，子进程的状态完全是该进程专用的。这意味着父级和子级不直接共享内存或文件描述符。注意，两个进程当然可以打开同一个文件。例如，如上所述，所有进程都应将文件描述符0和1映射到系统控制台。
* 与KThread.join（）不同，只有进程的父级可以加入该进程。例如，如果A执行B，而B执行C，则不允许A加入C，但是允许B加入C。
* join将进程ID作为参数，用于唯一地标识父级希望加入的子进程。进程ID应该是全局唯一的正整数，在创建时将分配给每个进程。 （尽管对于此项目，仅在联接中使用了进程ID，但对于以后的项目阶段，在系统中所有正在运行的进程中，进程ID都是唯一的很重要。）完成此操作的最简单方法是维护静态计数器指示要分配的下一个进程ID。由于进程ID是一个整数，因此，如果系统中有许多进程，则此值可能会溢出。对于这个项目，您不应该处理这种情况。也就是说，假定进程ID计数器不会溢出。
* 当进程调用exit（）时，其线程应立即终止，并且该进程应清除与其相关的任何状态（即释放内存，关闭打开的文件等）。如果进程异常退出，请执行相同的清理。
* 如果父级调用联接系统调用，则退出过程的退出状态应转移给父级。异常退出的进程的退出状态取决于您。为了进行连接，如果子进程以任何状态调用exit syscall，子进程都会正常退出，如果内核将其杀死（例如由于未处理的异常），则子进程会异常退出。
* 调用exit（）的最后一个过程应通过调用Kernel.kernel.terminate（）来使计算机停止运行。 （请注意，只应允许根进程调用halt（）系统调用，但最后一个退出的进程应直接调用Kernel.kernel.terminate（）。）

### 设计方案

我们要实现三个系统调用：exec、join、exit,同样在syscall.h，这三个系统调用与进程调度有关。

1. exce 启动一个新的进程；
2. join与线程中的join 操作类似，等待某进程结束后当前线程继续；
3. exit退出当前进程。

需要注意的是：

1. 父进程和子进程不共享任何的内存、文件或其它状态
2. **只有父进程能对子进程进行join 操作**，

例如 A执行B，B执行C，则A不允许Join C， 而B允许 join C

1. 需要为每个进程分配一个唯一的进程编号
2. exit 操作将使当前进程立即结束，如果父进程对其进行join操作，

返回代码应返回，Root Process 调用 exit 的进程将使系统停机。

2.1 exec系统调用

**int** handleExec(**int** fileAddress,**int** argc,**int** argvAddress)

**@param** fileAddress 文件名地址

**@param** argc 参数个数

**@param** argvAddress 参数地址

1.读虚拟内存获得文件名，判断参数合法性：得到的文件名不能为空，参数个数不能小于0，地址不能小于0，地址不能超过总的容量（numPages\****pageSize***）

2.将文件内容读入虚拟内存，先从argvAddress及其后续偏移地址分别取出参数的地址放入argsAddress（byte类型数组中），再从该地址数组中拿地址取得相应参数得string值

3.创建子进程，将文件和参数标加载到子进程，然后执行子进程

4.设置子线程的参数信息：父线程设置为当前线程，将子线程加入到当前线程的子线程链表中

2.2 join系统调用

**int** handleJoin(**int** pid,**int** statusAddress)

**@param** pid join线程的pid

**@param** statusAddress 一个地址用来保存子进程的返回值

1.利用进程号pid确定join的是哪一个进程，检查当前线程的childProcess，遍历子进程链表，确定join的进程是子进程，如果子进程编号不在其中则返回，不能执行join

2.获得join的锁，拿到锁后使当前进程在其子进程的joinCondition条件变量中睡眠直到被子进程唤醒

3.唤醒后释放锁，将子进程的运行状态存入父进程的内存中

4.判断当前进程是否正常退出

2.3 exit系统调用

**int** handleExit(**int** status)

先关闭coff文件，因为即将退出该进程。

1.判断是否在该子进程openfile中有打开的文件，如果有则关闭，并且把位置内容置为null，保存status状态（打开或关闭）

2.如果该进程的父进程不为空，就唤醒父进程，并从父进程的childProcess中删除这个子进程，再释放内存

3.判断当前正在运行的进程数量是否为1，如果只有一个线程运行表示只有main线程了，执行停机指令

### 3.代码实现

3.1声明变量

*// 2.3 start*   
 */\*\** UserProcess*的父进程* *\*/*   
 **private** UserProcess parentProcess;   
   
 LinkedList<UserProcess> childProcess = **new** LinkedList<UserProcess>();   
 **private** KThread thread;   
 **private** Lock joinLock = **new** Lock();   
 Condition joinCondition = **new** Condition(joinLock);   
 **private** static int numOfRunningProcess = 0;   
 **public** boolean normalExit;*// 退出状态，是否正常退出*   
 **public** int status = 0;*// 进程运行的状态 (打开/关闭)*   
 **private** static int numberOfProcess = 0;   
*// 2.3 end*

3.2实现方法

//2.3start   
private int handleExec(int fileAddress, int argc, int argvAddress) {   
 // 读虚拟内存获得文件名   
 String filename = readVirtualMemoryString(fileAddress, 256);   
 // 判断合法性:得到的文件名不能为空，参数个数不能小于0，地址不能小于0，地址不能超过总的容量   
 if (filename == null || argc < 0 || argvAddress < 0 || argvAddress > numPages \* pageSize)   
 return -1;   
 // args 参数数组:将由参数地址-物理地址-得到的内容-存到args里面   
 String[] args = new String[argc];   
 // 将文件内容读入虚拟内存   
 for (int i = 0; i < argc; i++)// 对于 参数的数量 argc i=0~argc-1   
 {   
 // 先从argvAddress及其后续偏移地址分别取出 参数的地址   
 // 放入argsAddress（byte类型数组中），再从该地址数组中拿地址取得相应参数得string值   
 byte[] argsAddress = new byte[4];// ？   
   
 // \*4的原因:前四个字节存的是指针   
 // argvAddress作为参数表数组的首址,读取虚拟内存地址   
 if (readVirtualMemory(argvAddress + i \* 4, argsAddress) > 0)// 从地址读出，存入argsAddress   
 // argsAddress中保存了读出的参数   
 // 依次读出每个参数   
 args[i] = readVirtualMemoryString(Lib.bytesToInt(argsAddress, 0), 256);   
 }   
 // 创建子进程，将文件和参数标加载到子进程   
 UserProcess process = UserProcess.newUserProcess();   
 // 创建并执行子进程   
 if (!process.execute(filename, args))// 文件打开失败，退出   
 return -1;   
 // 将这个子进程的父进程置为该进程   
 process.parentProcess = this;// 将当前父进程信息赋予子进程   
 // 将子进程加入到子进程表中（是此父进程的子进程表）   
 childProcess.add(process);   
 return process.pid;   
 }   
   
 private int handleJoin(int pid, int statusAddress) {   
 UserProcess process = null;   
 // 遍历子进程链表，确定join的进程是子进程   
 for (int i = 0; i < childProcess.size(); i++) {   
 if (pid == childProcess.get(i).pid)// 如果子进程是当前运行程序则返回   
 {   
 process = childProcess.get(i);   
 break;   
 }   
 }   
 if (process == null)   
 return -1;   
 // 获得join锁，让该进程休眠，直到子进程唤醒   
 // 得到锁，保持互斥   
 process.joinLock.acquire();// 得到锁   
 // 在该线程   
 process.joinCondition.sleep();// 进程睡眠，加入等待队列等待被唤醒   
 process.joinLock.release();   
 byte[] childstat = new byte[4];   
 // 取出子进程的运行状态   
 childstat = Lib.bytesFromInt(process.status);   
 // 将子进程的状态存入内存中，判断当前进程是否正常结束   
 int numWriteByte = writeVirtualMemory(statusAddress, childstat);   
 if (process.normalExit && numWriteByte == 4)   
 return 1;   
 return 0;   
 }   
   
 private int handleExit(int status) {   
 coff.close();// 关闭coff区，因为即将退出该进程   
 // 判断是否在该子进程openfile中有打开的文件   
 for (int i = 0; i < 16; i++) {   
 if (openfile[i] != null) {   
 // 无内容可以写，关闭   
 openfile[i].close();// 如果有则关闭   
 openfile[i] = null;// 并且把位置内容置为null   
 }   
 }   
   
 this.status = status;// 把状态置入(打开或关闭)   
 normalExit = true;// 正常退出   
 // 如果有父进程，就从父进程的子进程链表删除，而且如果父进程中join子进程，则唤醒父进程   
 if (parentProcess != null) {   
 joinLock.acquire();// 实现互斥   
 joinCondition.wake();   
 joinLock.release();   
 parentProcess.childProcess.remove(this);   
 }   
 // 释放内存   
 unloadSections();   
 if (numOfRunningProcess == 1)   
 Machine.halt();   
 numOfRunningProcess--;   
 KThread.currentThread().finish();   
 return 0;   
 }   
   
 // 2.3 end

### 4.测试

4.1编写用户程序

（1）编写程序test\_123.c

用来测试create、open、write、read、close系统调用：

#include "syscall.h"   
#include "stdio.h"   
#include "stdlib.h"   
int mystr(char \*buffer)   
{   
 int i;   
 **for**(i=0;i<500;i++)   
 {   
 **if**(buffer[i]==0)   
 **return** i;   
 }   
 **return** -1;   
}   
void main()   
{   
 int fd=0;   
 char \*filename="a.txt";   
 int ByteNum;   
 char \* buffer="this is test for 2.1 2.2 2.3 \n";   
 char buffersize=mystr(buffer);   
 char buf[40];   
 creat(filename);   
 printf("calling 'create(filename)'...");   
 printf("done ! \n");   
 fd=open(filename);   
 printf("calling 'fd=open(filename)'...");   
 printf("return value of fd=");   
 printf("\n");   
 write(fd,buffer,buffersize);   
 close(fd);   
 printf("calling 'write(fd,buffer,buffersize)'...\n");   
 fd=open(filename);   
 int i;   
 ByteNum=read(fd,buf,40);   
 printf("calling 'read(fd,buf,40)'...\n");   
 printf(buf);   
 close(fd);   
}

（2）使用系统自带的halt.c和remove.c测试halt和unlink系统调用

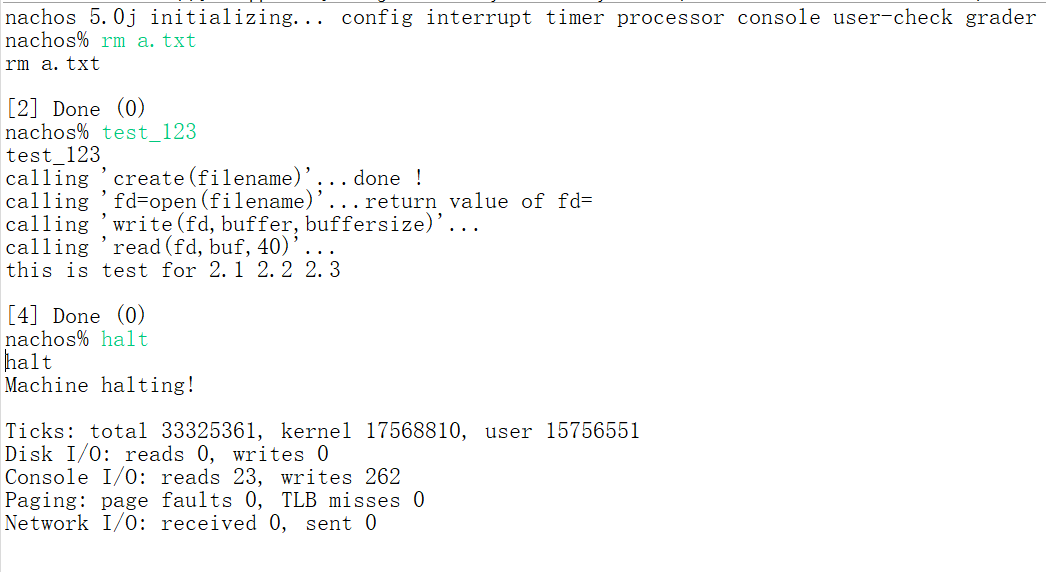
（3）编写testjoin测试exec系统调用和join系统调用，在sh.c运行过程中，输入exit,即测试了exit系统调用

#include "syscall.h"   
#include "stdio.h"   
#include "stdlib.h"   
   
int main(int argc, char\*\* argv)   
{   
 int i = 0;   
 **for**(; i < 15; ++i)   
 {   
 **if**(i == 3)   
 {   
 char\* a[] = {"chang"};   
 printf("child execute\n");   
 int p = exec("echo.coff", 1, a);   
 int b = -1;   
 join(p, &b);   
 }   
 printf("father process %d\n", i);   
 }   
 **return** 0;   
}

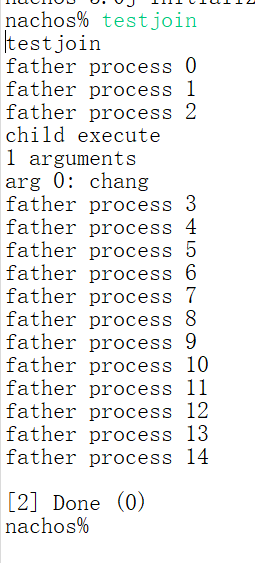
在ubuntu中编译生成coff文件。

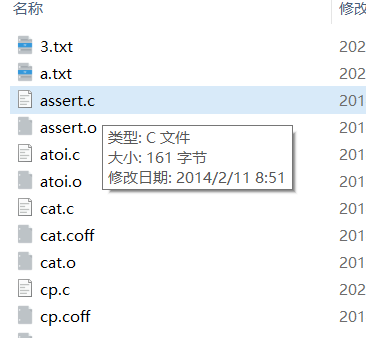
4.2测试结果：

Test\_123



Testjoin





结果正常。

## Task2.4 彩票调度

### 实验要求

* (15%, 50 lines + existing priority scheduler)

Implement a lottery scheduler (place it in threads/LotteryScheduler.java). Note that this class extends PriorityScheduler, you should be able to reuse most of the functionality of that class; the lottery scheduler should not be a large amount of additional code. The only major difference is the mechanism used to pick a thread from a queue: a lottery is held, instead of just picking the thread with the most priority. Your lottery scheduler should implement priority donation. (Note that since this is a lottery scheduler, priority inversion can't actually lead to starvation! However, your scheduler must do priority donation anyway.)

* In a lottery scheduler, instead of donating priority, waiting threads transfer tickets to threads they wait for. Unlike a standard priority scheduler, a waiting thread always adds its ticket count to the ticket count of the queue owner; that is, the owner's ticket count is the sum of its own tickets and the tickets of all waiters, not the max. Be sure to implement this correctly.
* Your solution should work even if there are billions of tickets in the system (i.e. do not keep an array containing an entry for every ticket).
* When LotteryScheduler.increasePriority() is called, the number of tickets held by a process should be incremented by one. Similarly, for decreasePriority(), the number should be decremented by one.
* The total number of (real) tickets in the system is guaranteed not to exceed Integer.MAX\_VALUE. The maximum individual priority is now also Integer.MAX\_VALUE, rather than 7 (PriorityScheduler.priorityMaximum). If you wish, you may also assume that the minimum priority is increased to 1 (from 0).
* （15％，50行+现有优先级调度程序）

实现彩票调度程序（将其放置在threads / LotteryScheduler.java中）。注意，该类扩展了PriorityScheduler，您应该能够重用该类的大多数功能；彩票调度程序不应包含大量附加代码。唯一的主要区别是用于从队列中选择线程的机制：持有彩票，而不只是选择具有最高优先级的线程。您的彩票调度员应实施优先捐赠。 （请注意，由于这是彩票调度程序，因此优先级倒置实际上不会导致饥饿！但是，无论如何，您的调度程序必须进行优先级捐赠。）

* 在彩票调度程序中，等待线程将票证转移到等待的线程上，而不是捐赠优先级。与标准优先级调度程序不同，等待线程始终将其票证计数添加到队列所有者的票证计数中。也就是说，所有者的票数是其自己的票数和所有服务员的票数之和，而不是最大值。确保正确实施此操作。
* solution即使系统中有数十亿张票证，您的解决方案也应能正常工作（即不要保留每个票证都包含一个条目的数组）。
* Lot调用LotteryScheduler.increasePriority（）时，进程持有的票证数量应增加一。同样，对于reducePriority（），该数字应减一。
* 保证系统中（实际）票证的总数不超过Integer.MAX\_VALUE。现在，最大的个人优先级也是Integer.MAX\_VALUE，而不是7（PriorityScheduler.priorityMaximum）。如果愿意，您还可以假定最小优先级从1增加到1（从0开始）。

### 设计方案

与Task1.5的优先级调度十分相似。有两点改变：①“优先级”的概念换成了“彩票”，即表示该线程下次被运行的概率 ②在调度选择下一个运行的线程过程中，并不是选择最大优先级（即不是最大彩票数），而是随机抽取一张彩票，让彩票的主人运行，这样，彩票越多，下次得到的运行机会就越大。

注意修改优先级最大值为Integer.MAX\_VALUE；最小值可修改为1.有效优先级设置为1.

主要修改PriorityScheduler.java中的两处来实现彩票调度LotteryScheduler.java:

1. 首先在getEffectivePriority()中{实际上是changeEffectivePriority()函数}将选取最大优先级的过程改成将所有等待线程的优先级（彩票调度中叫:彩票数）加到owner上（即等待者把自己拥有的彩票给获得锁的线程）
2. 其次将pickNextThread()修改如下:首先遍历一遍队列，计算出当前队列中的所有线程的彩票总数（考虑过捐赠之后的）S，然后生成一个0到S-1之间的随机数R表示抽取的彩票，最后再次遍历一遍队列，用累加器T计算彩票总数，当T的当前值大于R时说明当前循环到的线程正好落在目标区间内，即选出来该线程  
    由于彩票调度和优先线调度差异很小，nachos已经把彩票调度的实现继承PriorityScheduler，然后重写PriorityQueue.changeEffectivePriority(), PriorityQueue.pickNextThread()方法即可。

### 代码实现

3.1声明变量

*/\*\**   
 *\** The default priority for a new thread*.* Do not change this value*.*   
 *\*/*   
 **public** static final int priorityDefault = 1;   
 */\*\**   
 *\** The minimum priority that a thread can have*.* Do not change this value*.*   
 *\*/*   
 **public** static final int priorityMinimum = 1;   
 */\*\**   
 *\** The maximum priority that a thread can have*.* Do not change this value*.*   
 *\*/*   
 **public** static final int priorityMaximum = Integer.MAX\_VALUE;   
 **public** static Random random = **new** Random();

3.2实现方法

public class PriorityQueue extends PriorityScheduler.PriorityQueue   
 {   
   
 PriorityQueue(boolean transferPriority)   
 {   
 super(transferPriority);   
 this.transferPriority = transferPriority;   
 waitQueue = new LinkedList<>();   
 holder = null;   
 }   
   
 @Override   
 public ThreadState pickNextThread()   
 {   
 ThreadState r = null;   
   
 //temp : waitQueue的每一个线程的彩票总数   
 int temp = 0;   
 for(int i = 0; i < waitQueue.size(); ++i)   
 {   
 temp += waitQueue.get(i).effectivePriority;   
 }   
   
 //彩票总数不为0   
 if(temp != 0)   
 {   
 int rand = random.nextInt(temp) + 1; //随机抽取的彩票数是rand   
 temp = 0;   
   
 /\*再次对 waitQueue的每一个线程的彩票数，开始相加   
 \* 直到 rand < 不断相加的彩票总数,那么当前线程就是获得彩票的那个(即下一个要执行的)线程。   
 \*/   
 for(int i = 0; i < waitQueue.size(); ++i)   
 {   
 temp += waitQueue.get(i).effectivePriority;   
 if(rand <= temp)   
 {   
 r = waitQueue.remove(i);//r 就是获得彩票的那个ThreadState,即下一个要执行的。   
 break;   
 }   
 }   
 }   
 //每去掉（执行）一个线程，都要改变holder（如果有holder的话）的EffectivePriority   
 if(holder != null)   
 {   
 holder.holdingQueues.remove(this);   
 holder.changeEffectivePriority();   
 holder = r;   
 }   
 //r 就是获得彩票的那个ThreadState,即下一个要执行的。   
 if(r != null)   
 r.waitQueue = null;   
   
 return r;   
 }   
   
 protected LinkedList<ThreadState> waitQueue;   
 protected LotteryScheduler.ThreadState holder;   
}   
   
   
protected class ThreadState extends PriorityScheduler.ThreadState   
 {   
 public ThreadState(KThread thread) {   
 super(thread);   
 holdingQueues = new LinkedList<>();   
 this.thread = thread;   
 setPriority(priorityDefault);   
 super.effectivePriority = priorityDefault;   
 }   
   
 @Override   
 public void waitForAccess(PriorityScheduler.PriorityQueue wait)   
 {   
 PriorityQueue p = (PriorityQueue)wait;   
 boolean oldStatus = Machine.interrupt().disable();   
   
 this.waitQueue = p;   
 p.waitQueue.add(this);   
   
 //每增加waitForAccess一个线程，都要改变holder（如果有holder的话）的EffectivePriority   
 if(p.holder != null)   
 p.holder.changeEffectivePriority();   
   
 Machine.interrupt().restore(oldStatus);   
 }   
   
 @Override   
 public void acquire(PriorityScheduler.PriorityQueue wait)   
 {   
 boolean oldStatus = Machine.interrupt().disable();   
 PriorityQueue p = (PriorityQueue)wait;   
 if(p.transferPriority)   
 {   
 holdingQueues.add(p);   
 p.holder = this;   
 }   
 Machine.interrupt().restore(oldStatus);   
 }   
   
 protected void changeEffectivePriority()   
 {   
 effectivePriority = priority;   
 if(holdingQueues == null)   
 {   
 System.out.println();   
 }   
   
 /\* this线程 的 有效彩票数 = 自身的彩票数priority + 等待队列里所有线程的彩票数目   
 \* 一开始是effectivePriority = priority   
 \* 然后遍历holdingQueues的 几个PriorityQueue，对于每一个PriorityQueue，遍历其中的线程，   
 \* 不断加遍历的线程的彩票数，   
 \* 最终遍历完holdingQueues，所加得的effectivePriority就是 this线程 的 有效彩票数   
 \* \*/   
 for(PriorityQueue p : holdingQueues)   
 {   
 for(int i = 0; i < p.waitQueue.size(); ++i)   
 {   
 Lib.assertTrue(effectivePriority + p.waitQueue.get(i).effectivePriority < priorityMaximum);   
 effectivePriority += p.waitQueue.get(i).effectivePriority;   
 }   
 }   
 //this线程 的 有效彩票数effectivePriority修改了，   
 //如若waitQueue != null && waitQueue.holder != null，   
 //那么waitQueue.holder也要跟着修改EffectivePriority   
 if(waitQueue != null && waitQueue.holder != null)   
 waitQueue.holder.changeEffectivePriority();   
 }   
   
 protected LinkedList<PriorityQueue> holdingQueues;   
 protected PriorityQueue waitQueue;   
 }

### 测试

4.1将配置文件nachos.conf的调度方法改为彩票调度：



4.2测试代码

*// 2.4test start*   
 **public** static void testLottery() {   
 System.out.println("-----Now we begin to testLottery()-----");   
 boolean oldStatus = Machine.interrupt().disable();   
   
 ThreadQueue q11 = ThreadedKernel.scheduler.newThreadQueue(**true**);   
 ThreadQueue q12 = ThreadedKernel.scheduler.newThreadQueue(**true**);   
 LotteryScheduler.PriorityQueue q = (LotteryScheduler.PriorityQueue) q11;   
 LotteryScheduler.PriorityQueue q1 = (LotteryScheduler.PriorityQueue) q12;   
   
 KThread k1 = **new** KThread().setName("k1");   
 KThread k2 = **new** KThread().setName("k2");   
 KThread k3 = **new** KThread().setName("k3");   
   
 q.acquire(k1);   
 System.out.println("k1得到q队列,q队列:[k1]");   
 System.out.println("k1 effectivePriority = " + ThreadedKernel.scheduler.getEffectivePriority(k1));   
 System.out.println();   
   
 q.waitForAccess(k2);   
 System.out.println("k2加入q队列,q队列:[k1],k2");   
 System.out.println("k1 effectivePriority = " + ThreadedKernel.scheduler.getEffectivePriority(k1));   
 System.out.println("k2 effectivePriority = " + ThreadedKernel.scheduler.getEffectivePriority(k2));   
 System.out.println();   
   
 q1.acquire(k2);   
 System.out.println("k2得到q1队列,q1队列:[k2]");   
 q1.waitForAccess(k3);   
 System.out.println("k3加入q1队列,q1队列:[k2],k3");   
 System.out.println("k1 effectivePriority = " + ThreadedKernel.scheduler.getEffectivePriority(k1));   
 System.out.println("k2 effectivePriority = " + ThreadedKernel.scheduler.getEffectivePriority(k2));   
   
 Machine.interrupt().restore(oldStatus);   
 }   
*// 2.4test end*

4.3测试结果

