Table of Contents

Lab1 Nachos实验环境准备、安装与源码分析

1.1 Nachos环境安装

实验以`Ubuntu 20.04.2.0 LTS Desktop amd64` (Focal Fossa) (64位，iso文件：`ubuntu-20.04.2.0-desktop-amd64.iso` 2,877,227,008 字节)为例，介绍在虚拟机中安装64位Ubuntu及Nachos的过程。

1.1.1 更新Ubuntu的源

`sudo apt update`

1.2.2 安装gcc，g++，make，及一些gcc库

1. `sudo apt install gcc`
2. `sudo apt install g++`
3. `sudo apt install make`
4. `sudo apt install gcc-multilib g++-multilib`

1.1.3 安装用于MIPS的交叉编译器

1. `将压缩包 gcc-2.8.1-mips.tar.gz 复制到 ~ (Home，用户主目录)`
2. `cd /usr/local`
3. `sudo tar -xzvf ~/gcc-2.8.1-mips.tar.gz`

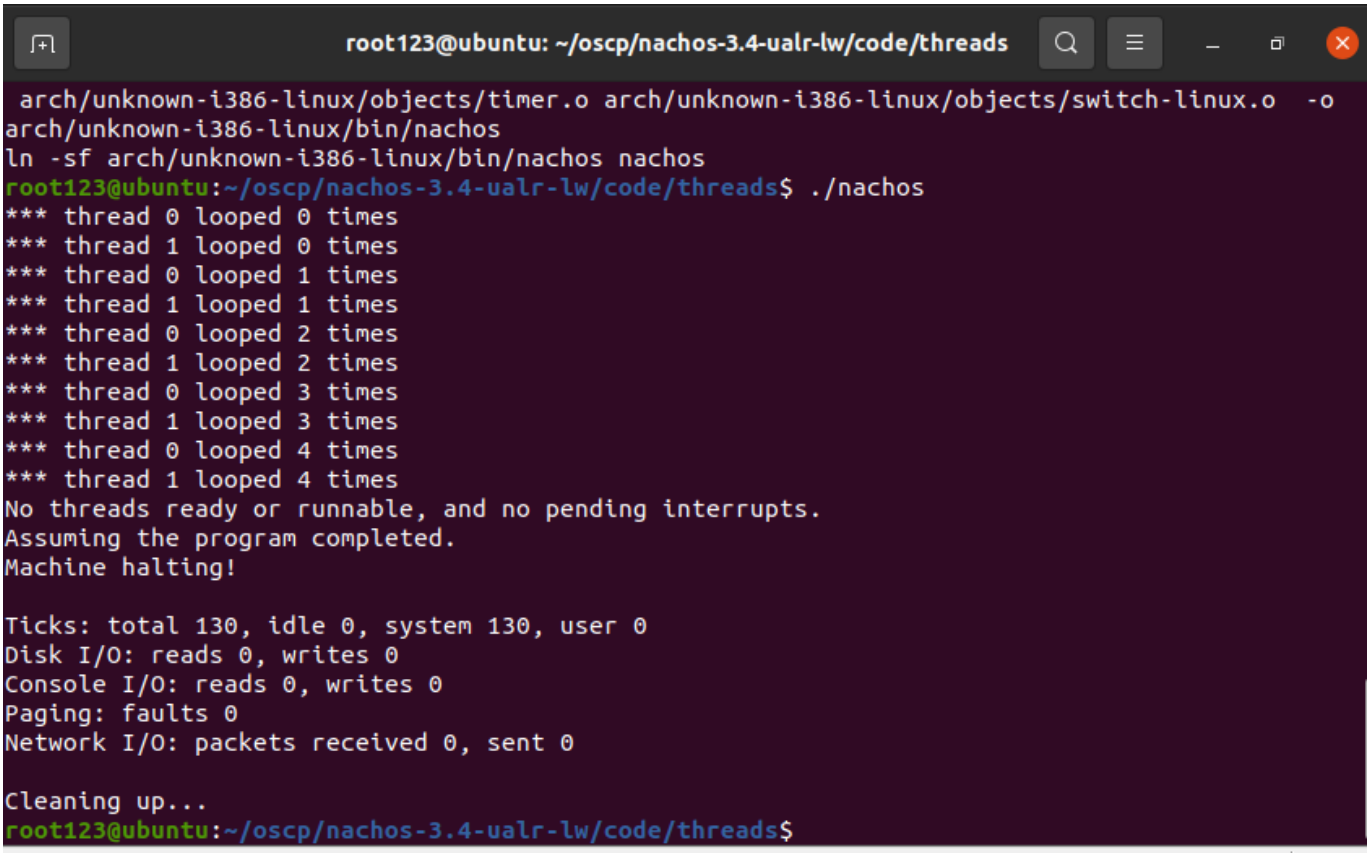
1.1.4 安装Nachos 3.4

1. `cd ~`
2. `mkdir oscp`
3. `cd oscp`
4. `将压缩包 nachos-3.4-ualr-lw.tar.gz 复制到 ~/oscp`
5. `tar -xzvf nachos-3.4-ualr-lw.tar.gz`

1.1.5 测试Nachos threads

1. `cd ~/oscp/nachos-3.4-ualr-lw/code/threads`
2. `make clean``
3. `make`
4. `./nachos`

测试程序截图



1.2 Make分析

1.2.1 make基本原理

`make` 是一种控制编译或重复编译软件的工具软件，make 可以自动管理软件的编译内容、 编译方式和编译时机。使用 make 需要你为你所编写的软件的开发过程和组织结构编写一个 Makefile 文件。make 将根据 Makefile 中的说明去自动管理你的软件的开 发过程。Makefile 是一个文本形式的数据库文件。可应包含以下目标软件的生成规则：

```
target: dependency […]
    command1
    command2
    […]
```

- target：目标体（target），即 make 要建立的目标文件。
- dependency：目标的依赖体（dependency）列表，通常为要编译的源文件或要连接的浮动目标代码文件。
- command： 从目标依赖体创建目标体的命令（command）列表，通常为编译或连接命令。

例如我们编写了一个 C 程序存放在`hello.c`和一个`hello.h`文件中，为了使用 make 自动管理这个 C 程序的开发，可以编写以下 Makefile 文件：

```
hell.o:    hello.c hello.h
    gcc -c hello.c hello.h
hello: hello.o
    gcc hello.o -o hello
clean:
    rm -f *.o
```

这样我们就可以在**命令行**中使用 `make` 命令 按我们说明在 Makefile 中的编译规则编译我们的程序了：

```
make                              #生成可执行文件 hello
make hello.o                     #生成浮动模块文件 hello.o
make clean                        #清除所有.o 文件
```

make 怎样知道什么时候需要重新编译或无需重新编译或编译部分文件呢?

- 如果指定的目标体 make 找不到，make 就根据该目标体在 Makefile 中说明的生成规则**建立**它。
- 如果目标体存在，make 就对目标体和依赖体的时间戳进行比较，若有一个或 多个依赖体比目标体新，make 就根据生成命令重新生成目标体。这意味着每个 依赖体的改动都将使目标体重新生成。

1.2.2 make中的变量与规则

make 中的宏变量：在 Makefile 中可以定义宏变量。**变量的定义格式为：变量名=字符串1 字符串2 ....，变量的引用格式为：$(变量名)**

如之前hello例可改写为：

```
obj=hello.o
hello: $(obj)
    gcc $(obj) -o hello
```

make 中的自动变量：make 中提供了一组元字符用来表示自动变量，自动变量用来匹配某种规则，它们有：

```
$@ 规则的目标体所对应的文件名
$< 规则中第一个相关文件名
$^ 规则中所有相关文件名的列表
$? 规则中所有日期新于目标文件名的列表
$(@D) 目标文件的目录部分
$(@F) 目标文件的文件名部分
```

make 中的预定义变量：

```
AR 归档维护程序，默认值=ar
AS 汇编程序，默认值=as
CC C 编译程序，默认值=gcc
CPP C++编译程序，默认值=cpp
RM 删除程序，默认值=rm -f
ARFIAGS 归档选项开关，默认值=rv
ASFLAGE 汇编选项开关
CFLAGS C 编译选项开关
CPPFLAGS C++编译选项开关
LDFILAGS 链接选项开关
```

make中隐式规则（静态规则）： 编译过程中一些固定的规则可以省略说明，称为隐式规则。如上例中目标体 hello.o 的规则隐含在目标体 hello 的规则中，就属于隐式规则，可以省略为：

```
obj=hello.c
hello: $(obj)
    gcc $(obj) -o hello
```

make 中的模式规则： % 用于匹配目标体和依赖体中任意非空字符串，例如：

```
%.o: %.c
    $(cc) -c $^ -o $@
```

以上的模式规则表示，用 g++编译器编译依赖体中所有的.c 文件，生成.o 浮动目标模块，目标文件名采用目标体文件名。

1.2.3 Nachos的Makefile结构

在Nachos的 code 目录中有个子目录公用的 Makefile 文件：`Makefile.common`，在 code/下的每个子目录中各自都有两个 Makefile 文件：`Makefile,Makefile.local`，即 Nachos 系统的 Makefile 结构为：

```
../code/Makefile.common,Makefile.dep
|____threads /Makefile,Makefile.local
|____userprog/ Makefile,Makefile.local
.
.
.
|____filesys /Makefile,Makefile.local
```

在code/下的每个子目录中的Makefile都有以下两行代码，分别导入同级目录的`Makefile.local`与上一级目录公用的`Makefile.common`：

```
include Makefile.local
include ../Makefile.common
```

- `Makefile.local`： Makefile.local 每个子目录中都不同，主要用于说明本目录中文件特有的依赖关系。 其中预定义变量的值为：
  - CCFILE 构造本目录中 Nachos 系统所用到的 C++源文件的文件名串
  - INCPATH 指示 g++编译器查找 C++源程序中括入的.h 文件的路径名串
  - DEFINES 传递给 g++编译器的标号串

    例如在threads/目录下的Makefile.local的定义为：

    ```
    CCFILES = main.cc\
      list.cc\
      scheduler.cc\
      synch.cc\
      synchlist.cc\
      system.cc\
      thread.cc\
      utility.cc\
      threadtest.cc\
      synchtest.cc\
      interrupt.cc\
      sysdep.cc\
      stats.cc\
      timer.cc
    INCPATH += -I../threads -I../machine
    DEFINES += -DTHREADS
    ```

    Nachos允许你在code/下任建的一个新目录中利用原有的内核源代码扩充和修改后重新构造。在这个**新目录中可以仅有你想改变的源代码文件或增加一些你为内核源代码新增**

的文件。

例如，我们要在空目录../lab2/目录中重新构造一个仅改变了调度算法的新版Nachos内核。 假设这需要改变`Scheduler`类，使用新的`scheduler.h`和`scheduler.cc`文件。而其他所有的文件仍然使用在../threads/，../machine/等目录中原有的文件。

为了这样做，首先你需要在../lab2/目录中重建或从../threads/目录中拷贝`scheduler.h`和`scheduler.cc`文件，从../threads/目录中递归的拷贝../arch/目录和Makefile, Makefile.local文件。接下来的工作是修改../lab2/中的Makefile.local文件，以便能在../lab2/中正确的构建新的Nachos

在Makefile. local文件中定义了基本的`CCFILE`宏和重定义的`INCPATH`宏。**如果新增了.cc文件你需要在CCFILE中声明。**本例中CCFILE宏无需改变，因为你没有增加新的.cc文件，make会沿着vpaths定义的路径顺序(vpath定义在Makefile.common中)查找所有不在当前目录中.cc文件。重定义的INCPATH需要修改。首先要把当前新建的目录**添加到INCPATH中：

```
INCPATH += -I- -I../labe –I../threads –I../machine
```

`-I-`作用：编译开关。-I-开关禁止处理与.cc文件在同一个目 录中的.h文件，即关闭由g++ MM产生的依赖关系，让每个.cc文件按INCPATH定义 的路径查找.h文件。

> 不添加-I-的话会产生如下影响：INCPATH += -I../labe –I../threads –I../machine
>
> 这样做仅是声明了.cc文件中直接扩入的.h文件的查找路径，但是一些.cc文件中间接括入的.h文件的查找路径并不是按照INCPATH定义的路径查找的,它们是按照由g++ MM产生的依赖关系来查找的。因此一些不在当前目录中而又间接括入了当前目录中.h文件的.cc文件不会随着当前目录中.h文件的修改而重新编译。
>
> 例如main.cc文件括入了system.h文件， 而system.h文件又括入了scheduler.h文件，现在的main.cc不会随着scheduler.h的 改变而重新编译。
>
> 解决这个问题的第一种方法是：查出不在本目录中所有与要修改的.h文件有间接关系的文件，将它们拷贝到当前目录中。但这种方法比较麻烦。**解决这个问题的第二种方法是：利用-I-编译开关。**
>
> 现在不需要查找和拷贝不在本目录中所有与要修改的.h文件有间接关系的文件了，make会根据我们在当前目录中所作的修改正确的重构新的系统。

- `Makefile.dep`：在 code/目录中的 Makefile.dep 文件用于定义由 g++使用的系统依赖关系的宏。**它被括入在 code/Makefile.common 文件中。**当前发行的 Nachos 可以在 4 种不同的 unix/linux 系统中编译并生成可执行的二进制文件 nachos。可执行文件统一放在 arch 目

录的特定目录下。例如在 i386 的 linux 系统中可执行的 nachos 程序应放在 arch/unknown-i386-linux/bin/目录中。

这些在 Makefile.dep 定义的依赖系统的宏有：

- HOST 主机系统架构
- arch 文档存放路径
- CPP C++编译器的名字
- CPPFLAGS C++编译开关
- GCCDIR g++安装路径
- LDFLAGS 程序链接开关
- ASFLAGS 汇编开关

  例如：当前系统为 i386 架构，linux 操作系统，则以上的宏定义为：

  ```
  HOST_LINUX=-linux
  HOST = -DHOST_i386 -DHOST_LINUX
  CPP=/lib/cpp
  CPPFLAGS = $(INCDIR) -D HOST_i386 -D HOST_LINUX
  arch = unknown-i386-linux
  ```

  在这个文件中还定义了其他一些依赖系统的宏，它们是：

  ```
  arch_dir = arch/$(arch) #归档文件目录
  obj_dir = $(arch_dir)/objects #存放目标文件的目录
  bin_dir = $(arch_dir)/bin #存放可执行文件的目录
  depends_dir = $(arch_dir)/depends #存放依赖关系文件的目录
  ```

  例如在i386/linux系统中最后3个目录为： 、

- arch/unknown-i386-linux/objects
- arch/unknown-i386-linux/bin
- arch/unknown-i386-linux/depends

- `Makefile.common`：

code/目录中的 Makefile.common 首先括入 Makefile.dep，然后用 vpath 定义各类 文件搜索路径。

```
include ../Makefile.dep
vpath %.cc ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.h ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.s ../network:../filesys:../vm:../userprog:../threads:../machine
```

vpath 定义告诉 make 到哪儿去查找在当前目录中找不到的文件。这就是为什么我们在一个新的目录中构造一个新的 Nachos 系统时不必复制那些我们不作修改的文件的原因。

然后定义了根据.cc和.h、.c和.h、.s文件编译链接生成.o文件（Linux系统存储在unknown-i386-linux/bin中）的指令。以及根据所有的.o 文件构造二进制可执行文件 nachos的命令。具体内容可见`Makefile.common`

1.3 Nachos操作系统概述

Nachos 是美国加州大学伯克莱分校在操作系统课程中已多次使用的操作系统课程设计平台，在美国很多大学中得到了应用，它具有一下几个突出的特点：

- 采用通用虚拟机：

  Nachos 是建立在一个**软件模拟的虚拟机**之上的，模拟了 `MIPS R2/3000` 的指令集、主存、中断系统、网络以及磁盘系统等操作系统所必须的硬件系统。许多现代操作系统大多是 先在用软件模拟的硬件上建立并调试，最后才在真正的硬件上运行。用软件模拟硬件的可靠性比真实硬件高得多，不会因为硬件故障而导致系统出错，便于调试。虚拟机可以 在运行时报告详尽的出错信息，更重要的是采用虚拟机使Nachos的移植变得非常容易， 在不同机器上移植 Nachos，只需对虚拟机部分作移植即可。

  采用`R2/3000`指令集的原因是该指令集为`RISC`指令集，其指令数目比较少。Nachos 虚拟机模拟了其中的 63 条指令。由于`R2/3000`指令集是一个比较常用的指令集，许多 现有的编译器如 gc++能够直接将 C 或 C++源程序编译成该指令集的目标代码，于是就**不必编写编译器，读者就可以直接用 C/C++语言编写应用程序**，使得在 Nachos 上开发 大型的应用程序也成为可能。

- 面向对象性：

  Nachos 的主体是用 C++的一个子集来实现的。它能够清楚地描述操作系统各个部分的接口。Nachos 没有用到面向对象语言的所有特征，如继承性、多态性等，代码就更容易阅读和理解。

安装Nachos在Linux操作系统之上后生成一个`nachos-3.4-ualr-lw`的文件夹，该目录中主要包含一下几个部分：

文件夹名	子文件夹名	作用
c++ example		介绍C++的实例
code	----	
	bin	包含有用户程序目标码变换的程序
	filesys	Nachos文件系统管理部分源代码
	machine	Nachos 虚拟机模拟部分源代码
	threads	Nachos 线程管理部分源代码
	userprog	Nachos 用户程序部分源代码
	network	Nachos 网络管理部分源代码
	lab2-7	具体实验的文件夹，里面包含有Makefile和一些基础文件
	Makefile.common	code/目录中的 Makefile.common 首先括入 Makefile.dep，然后用 vpath 定义各类 文件搜索路径。
	Makefile.dep	Makefile.dep 文件用于定义由 g++使用的系统依赖关系的宏。它 被括入在 code/Makefile.common 文件中。
doc		Nachos各个部分的介绍

Nachos的各个部分都可以独立编译运行，也可以同时编译各个部分。

由于`Linux`指令集和`R2/3000`指令集不同，用户编写的应用程序用 Linux 系统中标准 gcc 编译后，不能直接在 Nachos 虚拟机环境下运行。所以需要采用**交叉编译技术**。所谓交叉编译技术是在一个操作系统下将源码编译成另一个操作系统的目标码，这里就是在 Linux 下通过 gcc 交叉编译版本将用户程序的源码编译成 R2/3000 指令集的目标码。

将交叉编译工具解开至/usr/local/目录下：

```
# gzip -dc cross-compiler.tgz | tar xf -
```

在编译用户程序时，用交叉编译器将源码编译成 R2/3000 指令集的目标代码，再经过一个简单的转换就可以在 Nachos 虚拟机上运行。

1.4 Nachos虚拟机

Nachos 是建立在**一个软件模拟的虚拟机**上的。该虚拟机包括计算机的基本部分：如 CPU、 主存、寄存器、中断系统，还包括一些外部设备，如终端设备、网络以及磁盘系统。

用软件来模拟硬件另一个优点是充分利用了宿主机操作系统的软件资源，避免了编写复杂的硬件控制程序。更重要的是提高了程序的可移植性，**只要在不同硬件上实现 Nachos 虚拟机就完成了 Nachos 的大部分移植工作。**我们将 Nachos 移植到 Linux 上的工作就受益于这种设计。下面先对Nachos的机器模拟部分做简单介绍：

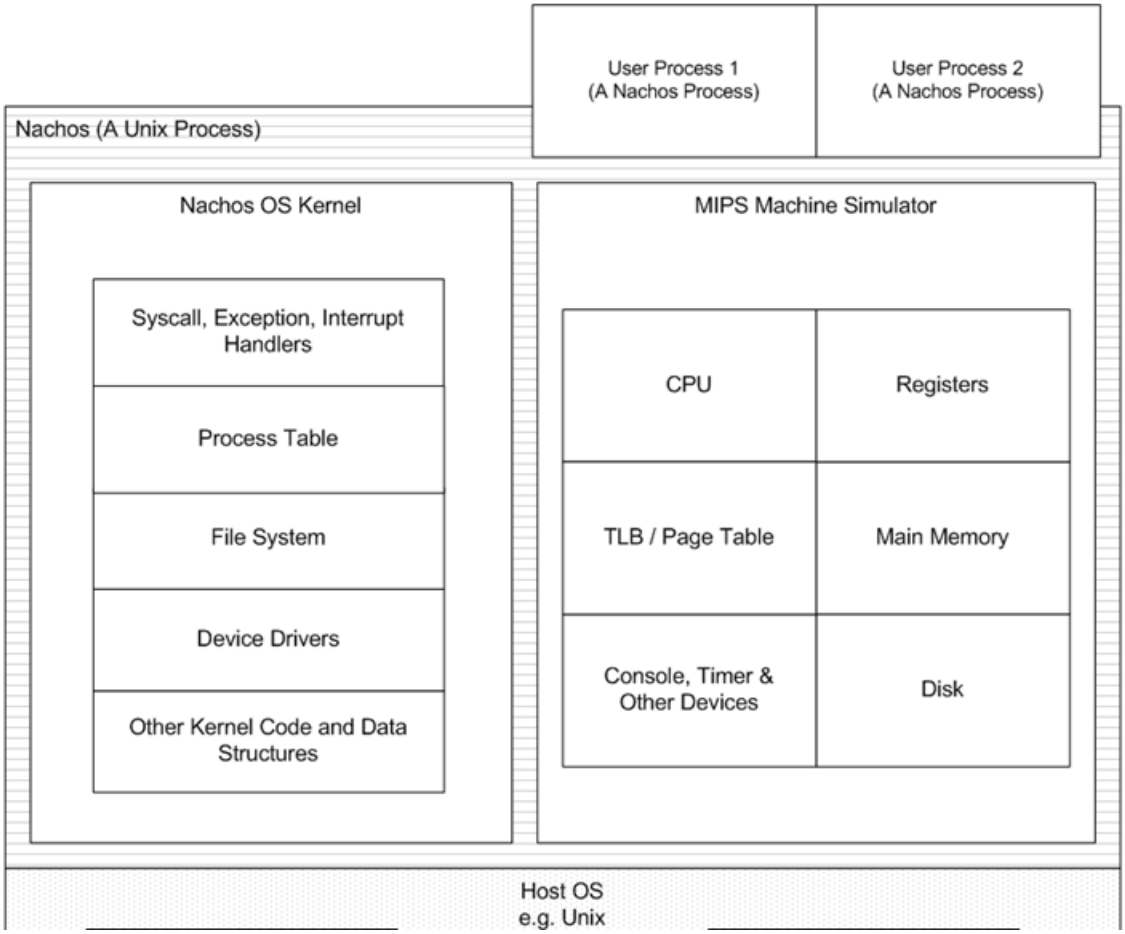
- `Machine`类：模拟计算机主机。
- `Interrupt`类：用来模拟硬件中断系统。在这个中断系统中，定义了中断状态、中断类型、机器状态。中断系统提供的功能有开/关中断，读/写机器状态， 将一个即将发生中断放入中断队列，以及使机器时钟前进一步。

  在`Interrupt`类中有一个记录即将发生中断的队列，称为**中断等待队列**。中断等待队列中每个等待处理的中断包含中断类型、中断处理程序的地址及参数、中断应当发生的时间等信息。 一般是由**硬件设备模拟程序**把将要发生的中断放入中断队列。
- 在这个中断系统基础上，Nachos 模拟了各种硬件设备，这些设备都是异步设备，依靠中断来与主机通信。
  - `Timer` 类模拟定时器。定时器每隔 X 个时钟周期就向`CPU`发一个时钟中断。它是时间片管理必不可少的硬件基础。
  - `Console` 类模拟的是控制台设备。
  - `Disk` 类模拟了物理磁盘，它一次只能接受一个读写请求，当读写操作完成后向 CPU 发一个磁盘中断。

中断系统成为整个 Nachos 虚拟机的基础，其它的模拟硬件设备都是建立在中断系统之上的。在此之上，加上 `Machine` 类模拟的指令解释器，可以实现 Nachos 的线程管理、文件系统管理、虚拟内存、用户程序和网络管理等所有操作系统功能。

图 2.2 Nachos 系统的整体结构

Nachos与主机操作系统的关系如图，可以看到Nachos建立在用户主机操作系统(Linux之上)：



1.4.1 Nachos Machine分析

`Machine`类用来模拟计算机主机。它提供的功能有：读写寄存器。读写主存、运行一条用户程序的汇编指令、运行用户程序、单步调试用户程序、显示主存和寄存器状态、将虚拟内存地址转换为物理内存地址、陷入 Nachos 内核等等。

`Machine` 类实现方法是在宿主机上分配两块内存分别作为虚拟机的寄存器和物理内存。运行用户程序时，先将用户程序从 Nachos 文件系统中读出，写入模拟的物理内存中，然后调用 指令模拟模块对每一条用户指令解释执行。将用户程序的读写内存要求，转变为对物理内存地址的读写。

`Machine` 类提供了单步调试用户程序的功能，执行一条指令后会自动停下来， 让用户查看系统状态，不过这里的单步调试是汇编指令级的，需要读者对 R2/3000 指令比较 熟悉。如果用户程序想使用操作系统提供的功能或者发出异常信号时，Machine 调用系统异常陷入功能，进入 Nachos 的核心部分。

1.4.2 Nachos Interrupt分析

中断模块的主要作用是**模拟计算机底层的中断机制**。可以通过该模拟机制来启动和禁止中断(SetLevel)；该中断机制模拟了 Nachos 系统需要处理的所有的中断，包括时钟中断、磁盘中断、终端读/终端写中断以及网络接收/网络发送中断。中断模块定义在`machine/interrupt.cc`与`machine/interrupt.h`中

中断的发生总是有一定的时间。比如当向硬盘发出读请求，硬盘处理请求完毕后会发生中断； 在**请求和处理完毕之间需要经过一定的时间。**所以在该模块中，模拟了时钟的前进。为了实现简单和便于统计各种活动所占用的时间起见，Nachos 规定**系统时间**在以下三种情况下前进：

- **执行用户态指令**：执行用户态指令，时钟前进是显而易见的。我们认为，Nachos 执行每条指令所需时间是固定的，为一个时钟单位`Tick`。
- **重新打开中断**：一般系统态在进行中断处理程序时，**需要关中断。**但是中断处理程序本身也需要消耗时 间，而在关闭中断到重新打开中断之间无法非常准确地计算时间，所以当中断重新打开的时候，加上一个中断处理所需时间的平均值。
- **就绪队列中没有进程**：当系统中没有就绪进程时（进程全部处于等待状态），系统处于`Idle`状态。这种状态可能是系统中所有的进程都在**等待**各自的某种操作完成。也就是说，系统将在未来某个时间发生中断，到中断发生的时候中断处理程序将进行中断处理。在系统模拟中，有一个**中断等待队列**，专门存放将来发生的中断。

  在这种情况下，可以将系统时间直接跳到中断等待队列第一项所对应的时间，（将来一定会发生第一项所对应的中断）以免不必要的等待。

当**前面两种情况**需要时钟前进时，调用`OneTick`方法。`OneTick`方法将系统态和用户态的时间分开进行处理：

```
void Interrupt::OneTick()
{
    MachineStatus old = status;

    // advance simulated time
    if (status == SystemMode) {// 系统态
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {// 用户态
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
       ......
    ......
}
```

中断等待队列是 Nachos 虚拟机最重要的数据结构之一，它记录了当前虚拟机可以预测的将在未来发生的所有中断。当系统进行了某种操作可能引起未来发生的中断时，如磁盘的写入、 向网络写入数据等都会将中断插入到中断等待队列中.

对于一些定期需要发生的中断，如时钟中断、终端读取中断等，系统会在中断处理后将下一次要发生的中断插入到中断等待队列中。中断的插入过程是一个优先队列的插入过程，其**优先级是中**

断发生的时间，也就是说， 先发生的中断将优先得到处理。

中断处理程序是在某种特定的中断发生时被调用。

`Interrupt.h`中首先声明了一些预定义枚举变量：

- 包括是否开中断：

```
// Interrupts can be disabled (IntOff) or enabled (IntOn)
enum IntStatus { IntOff, IntOn };
```

- Nachos 操作系统运行的三种状态：

```
// Nachos can be running kernel code (SystemMode), user code (UserMode),
// or there can be no runnable thread, because the ready list is empty (IdleMode).
enum MachineStatus {IdleMode, SystemMode, UserMode};
```

  - `IdleMode`： **系统 CPU 处于空闲状态，没有就绪线程可以运行。**如果中断等待队列中有需要处理的除了时钟中断以外的中断，说明系统还没有结束，将时钟调整到发生中断的时间，进行中断处理；否则认为系统结束所有的工作，**退出关机。**

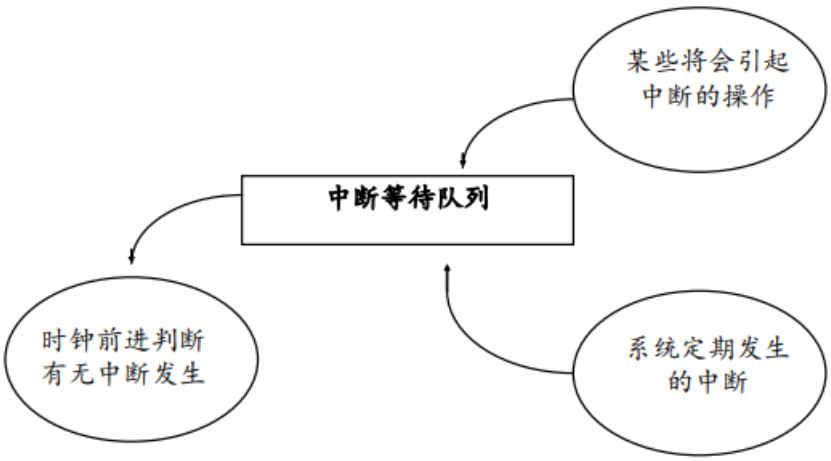


图 2.3　对中断等待队列的操作

  - `SystemMode`：Nachos 执行系统程序。Nachos 虽然模拟了虚拟机的内存，**但是 Nachos 系统程序本身的运行不是在该模拟内存中，而是利用宿主机的存储资源。**这是 Nachos 操作系统同真正操作系统的重要区别。

  - `UserMode`：系统执行用户程序。当执行用户程序时，每条指令占用空间是 Nachos 的模拟内存。

- Nachos需要处理的中断种类：

```
// IntType records which hardware device generated an interrupt.
// 时钟中断、 磁盘（读/写）中断、终端写中断、终端读终端、网络发送中断、网络接收中断
enum IntType { TimerInt, DiskInt, ConsoleWriteInt, ConsoleReadInt,
                NetworkSendInt, NetworkRecvInt};
```

1.4.2.1 PendingInterrupt类

```
class PendingInterrupt {
  public:
    PendingInterrupt(VoidFunctionPtr func, _int param, int time, IntType kind);
                // initialize an interrupt that will occur in the future
    VoidFunctionPtr handler; // 中断发生时对应的中断处理程序函数
    _int arg;            // 中断处理函数的参数
    int when;             // 中断发生的时机
    IntType type;         // for debugging 中断类型
};
```

这个类定义了一个**中断等待队列中**需要处理的中断的数据结构。为了方便起见，所有类的数据和成员函数都设置为 public 的，不需要其它的 Get 和 Set 等存取内部数据的函数。初始化函数就是为 对应的参数赋值。

1.4.2.2 Interrupt类

`Interrupt`类定义了模拟的硬件中断，在 Nachpos 中只存在一个`Interrupt`类对象。 它记录中断是启用还是禁用`IntStatus level`，中断等待队列`List *pending`，系统状态`MachineStatus status`…

```
class Interrupt {
  private:
    IntStatus level; // 中断的开关状态
    List *pending; // 当前系统中等待中断队列
    bool inHandler; // 是否正在进行中断处理标志
    bool yieldOnReturn; // 中断处理后是否需要正文切换标志
    MachineStatus status; // 当前虚拟机运行状态
    bool CheckIfDue(bool advanceClock);// 检查当前时刻是否有要处理的中断
    void ChangeLevel(IntStatus old, IntStatus now);// 改变当前中断的开关状态，但是不前进
模拟时钟

  public:
    Interrupt();
    ~Interrupt();
    IntStatus SetLevel(IntStatus level);// 开关中断，并且返回之前的状态
    void Enable(); // 开中断
    IntStatus getLevel() {return level;}// 取回当前中断的开关状态
    void Idle(); // 当进程就绪队列为空时，执行该函数
    void Halt(); // 退出系统，并打印状态
    void YieldOnReturn(); // 设置中断结束后要进行进程切换的标志
    MachineStatus getStatus() { return status; }// 返回系统当前的状态
    void setStatus(MachineStatus st) { status = st; }// 设置系统当前的状态
    void DumpState(); // 调试当前中断队列状态用
    void Schedule(VoidFunctionPtr handler, int arg, int when, IntType type);// 在中断等
待队列中，增加一个等待中断
    void OneTick(); // 模拟时钟前进
};
```

其中，`Schedule` 和 `OneTick` 两个方法虽然标明是 public 的，但是除了虚拟机模拟部分以外的其它类方法是不能调用这两个方法的。将它们设置成 public 的原因是因为虚拟机模拟的其它类方法需要直接调用这两个方法。下面是这两种方法的解释：

方法	参数	功能	实现
void Schedule		将一个中断插入等待处理中断队列。	调用 List.SoftInsert()方法。
	VoidFunctionPtr handler	中断处理函数	
	int arg	中断处理函数的参数	
	int fromNow	中断发生的时刻和现在时刻的差值	
	IntType type	中断的类型	
void OneTick	无	时钟前进一个单位（系统态时间单位是用户态时间单位的 10 倍。用户态执行一条用户指令花费一个用户态时间单位；当开中断时前进一个系统态时间单位，系统态时间单位是一个平均值）	·1.根据当前状态为用户态或是系统态时钟分别前进一个用户态时间单位或系统态时间单位，并且对 Nachos 运行的各项时间（用户态时间、系统态时间）进行统 计。 2.检查当前时刻是否有中断发生；如果有，进行中断处理 3.如果 yieldOnReturn 标志设置，作进程切换

```
void Interrupt::Schedule(VoidFunctionPtr handler, _int arg, int fromNow, IntType type)
{
    int when = stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(handler, arg, when, type);

    DEBUG('i', "Scheduling interrupt handler the %s at time = %d\n",
                    intTypeNames[type], when);
    ASSERT(fromNow > 0);
    pending->SortedInsert(toOccur, when);
}
```

重点分析`CheckIfDue`与`Idle`两个后面经常用到的方法

`CheckIfDue(bool advanceClock)`测试当前等待中断队列中是否要有中断发生，并根据不同情况作出不同处理（执行中断），具体过程：

1. 如果有中断，在等待处理的中断队列中取出第一项（最早会发生的中断）

```
PendingInterrupt *toOccur =
        (PendingInterrupt *)pending->SortedRemove(&when);
```

2. 如果不存在任何中断，返回 **FALSE。**

```
if (toOccur == NULL)        // no pending interrupts
        return FALSE;
```

3. 如果该中断的发生时机没有到：

   - 如果 advanceClock=TRUE，**系统时间 totalTicks 跳到中断将要发生的时间。** 说明中断马上就要发生。
   - 如果 advanceClock=FALSE，将取出的中断放回原处，等待将来处理，返回**FALSE**

   ```
   if (advanceClock && when > stats->totalTicks) {    // advance the clock
         stats->idleTicks += (when - stats->totalTicks);
         stats->totalTicks = when;
     } else if (when > stats->totalTicks) {    // not time yet, put it back
         pending->SortedInsert(toOccur, when);
         return FALSE;
   }
   ```

4. 如果当前的状态是 `Idle` 态（就绪队列里没有线程），**而且取出的中断是时钟中断，同时等待中断队列中没有其它的中断，意味着系统将退出。**但是系统的退出不在这里处理，而是将该中断放回原处，等待以后处理；并返回 **FALSE。**

```
if ((status == IdleMode) && (toOccur->type == TimerInt)
                && pending->IsEmpty()) {
     pending->SortedInsert(toOccur, when);
     return FALSE;
}
```

5. **中断发生！**

```
inHandler = TRUE;
status = SystemMode;            // whatever we were doing,
                                // we are now going to be running in the kernel
(*(toOccur->handler))(toOccur->arg);    // 执行中断处理程序
status = old;                // restore the machine status
inHandler = FALSE;
delete toOccur;
return TRUE;
```

`Idle()`：当就绪队列中没有任何东西时调用的函数。由于为了将一个线程放到就绪队列上，必须运行一些东西，所以唯一要做的就是**将模拟时间提前到下一个预定的硬件中断**（CheckIfDue函数实现），处理在新的时刻其它需要发生的中断。如果没有中断，停机退出Nachos。

```
void Interrupt::Idle()
{
    DEBUG('i', "Machine idling; checking for interrupts.\n");
    status = IdleMode;    // 将系统状态调为Idle态
    if (CheckIfDue(TRUE)) {         // 中断队列上有中断，模拟时间提前到下一个预定的硬件中
断,中断处理完后会自动进行上下文切换，将阻塞的线程放到就绪对列中
        while (CheckIfDue(FALSE));    // check for any other pending interrupts
        yieldOnReturn = FALSE;        // since there's nothing in the
                                      // ready queue, the yield is automatic
        status = SystemMode;
        return;                // 返回到调用Sleep()的进程
    }


    // 没有中断程序，也没有就绪线程，停机
    DEBUG('i', "Machine idle.  No interrupts to do.\n");
    printf("No threads ready or runnable, and no pending interrupts.\n");
    printf("Assuming the program completed.\n");
    Halt();// 停机函数
}
```

1.4.3 Nachos Timer分析

该模块的作用是模拟时钟中断。Nachos 虚拟机可以如同实际的硬件一样，每隔一定的时间会发生一次时钟中断。时钟中断间隔由 `TimerTicks` 宏决定（100 倍 Tick 的时间）。

> 这是一个可选项，原始的 Nachos 还没有充分发挥时钟中断的作用，只有在Nachos 指定线程随机切换时（Nachos -rs 参数，见线程管理部分Nachos 主控模块分析） 启动时钟中断，在每次的时钟中断处理的最后，加入了线程的切换。实际上，时钟中断的作用远不止如此，但Nachos还未实现以下方法：
>
> - 线程管理中的时间片轮转法的时钟控制，（详见线程管理系统中的实现实例中，对线程调度的改进部分）不一定每次时钟中断都会引起线程的切换，而是由该线程是否的时间片是否已经用完来决定。
> - 分时系统线程优先级的计算（详见线程管理系统中的实现实例中，对线程调度的改进部分）
> - 线程进入睡眠状态时的时间计算 可以通过时钟中断机制来实现 sleep 系统调用，在时钟中断处理程序中，每隔一定的时 间对定时睡眠线程的时间进行一次评估，判断是否需要唤醒它们。

`timer.h` 类定义如下所示：

```
class Timer {
  private:
    bool randomize;         // 是否需要随机时钟中断标志
    VoidFunctionPtr handler;    //时钟中断处理函数
    _int arg;             // 处理函数参数
  public:
    Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom);// 初始化时钟，每个时间片调用timerHandler时钟中断处理函数
    ~Timer() {}

    // 内部调用函数，除Nachos模拟程序其他不调用
    void TimerExpired();    //当时钟中断发生时调用
    int TimeOfNextInterrupt();  // 计算下一次时钟中断发生的时机
};
```

`timer.cc`：

```
static void TimerHandler(_int arg)
{ Timer *p = (Timer *)arg; p->TimerExpired(); }

Timer::Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom){
        randomize = doRandom;
        handler = timerHandler;
        arg = callArg;
           // 添加第一个时钟中断到等待队列中
        interrupt->Schedule(TimerHandler, (_int) this, TimeOfNextInterrupt(),
TimerInt);
}

void Timer::TimerExpired() {
        // 添加新的时钟中断到中断等待队列中
        interrupt->Schedule(TimerHandler, (_int) this, TimeOfNextInterrupt(),
TimerInt);
        // 调用timerHandker中断处理程序
        (*handler)(arg);
}

int Timer::TimeOfNextInterrupt() {
        if (randomize)
            return 1 + (Random() % (TimerTicks * 2));
        else
            return TimerTicks;
}
```

`Timer`类的实现很简单，当生成出一个`Timer`类的实例时，就设计了一个模拟的时钟中断。 这里考虑的问题是：怎样实现定期发生时钟中断?

在 Timer 的初始化函数中，该时钟中断函数是`TimerHandler`内部函数（见第 1 行）。为什么不直接用初始化函数中的 `timerHandler` **中断处理函数指针**参数作为中断处理函数呢?

因为我们**不仅要执行该时钟中断的中断函数，还要将新的时钟中断插入到中断等待队列中**，这样 Nachos 就可以定时的收到时钟中断。因此真正的时钟中断处理函数不只是 `timerHandler`函数，我们编写`TimerExpired()`函数表示这个过程，**但C++不允许指针指向类成员函数**，因此借用`TimerHandler`内部函数调用`TimerExpired()`方法。

`TimeOfextInterrupt()`方法的作用是**计算下一次时钟中断发生的时机**，如果需要时钟中断发生的时机是随机的，可以在 Nachos 命令行中设置 –rs 选项。这样，Nachos 的线程切换的时机将会是随机的。但是此时时钟中断则不能作为系统计时的标准了。

1.4.4 Nachos控制台与统计信息

`Console` 类模拟的是控制台设备。该模块的作用是模拟实现终端的输入和输出。包括两个部分，即键盘的输入和显示输出。终端输入输出的模拟是异步的，也就是说当发出终端的输入输出请求

后系统即返回，需要等待中断发生后才是真正完成了整个过程。

```
class Console {
 public:
    Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
    VoidFunctionPtr writeDone, int callArg);// 初始化方法
                                            // readAvail: 键盘读入中断处理函数
                                            // WriteDonel: 显示输出中断处理函数
     ~Console(); // 析构方法
    void PutChar(char ch); // 将字符 ch 向终端上输出
    char GetChar(); // 从终端上读取一个字符
    void WriteDone(); // 写终端中断时调用
    void CheckCharAvail(); // 读终端中断时调用
 private:
    int readFileNo; // 模拟键盘输入的文件标识符
    int writeFileNo; // 模拟显示器的文件标识符
    VoidFunctionPtr writeHandler; // 写中断处理函数
    VoidFunctionPtr readHandler; // 读中断处理函数
    int handlerArg; // 中断处理函数参数
    bool putBusy; // 正在写终端标志
    char incoming; // 读取终端字符的暂存空间
};
```

Nachos 的终端模拟借助了两个文件，即在生成函数 `Console()` 中的 `readFile` 和 `writeFile` 。这两个文件分 别模拟键盘输入和屏幕显示。

对 Nachos 运行情况进行统计的类 `stats` 。这并不属于机器模拟的一部分，但是为了了解自己设计的操作系统的各种运行情况。 `stats` 类中包含的各种统计项是非常有价值的。Statistics 类的定义和实现如下：

```
class Statistics {
 public:
    int totalTicks; // Nachos 运行的时间
    int idleTicks; // Nachos 在 Idle 态的时间
    int systemTicks; // Nachos 在系统态运行的时间
    int userTicks; // Nachos 在用户态运行的时间
    int numDiskReads; // Nachos 发出的读磁盘请求次数
    int numDiskWrites; // Nachos 发出的写磁盘请求次数
    int numConsoleCharsRead; // Nachos 读取的终端字符数
    int numConsoleCharsWritten; // Nachos 输出的字符数
    int numPageFaults; // 页转换出错陷入次数
    int numPacketsSent; // 向网络发送的数据包数
    int numPacketsRecvd; // 从网络接收的数据包数
    Statistics(); // 初始化方法，将所有的统计信息值都置 0
    void Print(); // 系统结束时，打印统计信息
};


Statistics::Statistics()
{
    totalTicks = idleTicks = systemTicks = userTicks = 0;
    numDiskReads = numDiskWrites = 0;
    numConsoleCharsRead = numConsoleCharsWritten = 0;
    numPageFaults = numPacketsSent = numPacketsRecvd = 0;
}

//----------------------------------------------------------------------
// Statistics::Print
//     Print performance metrics, when we've finished everything
//    at system shutdown.
//----------------------------------------------------------------------

void
Statistics::Print()
{
    printf("Ticks: total %d, idle %d, system %d, user %d\n", totalTicks,
    idleTicks, systemTicks, userTicks);
    printf("Disk I/O: reads %d, writes %d\n", numDiskReads, numDiskWrites);
    printf("Console I/O: reads %d, writes %d\n", numConsoleCharsRead,
    numConsoleCharsWritten);
    printf("Paging: faults %d\n", numPageFaults);
    printf("Network I/O: packets received %d, sent %d\n", numPacketsRecvd,
    numPacketsSent);
}
```

1.4.5 Nachos Disk分析

将会放在文件系统的分析中

1.5 Nachos启动分析

Nachos的主控模块是整个Nachos系统的入口，包括`main.cc,system.cc,system.h`等，如其他操作系统一样，Nachos 内核也是操作系统的一部分。最小的 Nachos 内核仅包含 Nachos 线程管理，可以在 threads 目录中编译生成。

Nachos 内核组成包括：

- 一个CPU调度器
- 一个中断模拟器
- 一个时钟模拟器
- 统计信息模块
- 至少一个内核线程(main线程)

`system.h`中定义了Nachos这些内核组件的**全局变量**，并且导出到整个项目中：

```
extern void Initialize(int argc, char **argv);     // Initialization,
                        // called before anything else
extern void Cleanup();                // Cleanup, called when
                        // Nachos is done.

extern Thread *currentThread;            // 当前CPU中运行的线程
extern Thread *threadToBeDestroyed;          // 刚调用finish()的线程
extern Scheduler *scheduler;            // 线程调度器
extern Interrupt *interrupt;            // 中断模拟
extern Statistics *stats;            // 统计性能信息
extern Timer *timer;                // 时钟中断硬件模拟

//根据定义不同的宏，声明一些特殊的全局变量：
#ifdef USER_PROGRAM
#include "machine.h"
extern Machine* machine;    // user program memory and registers
#endif

#ifdef FILESYS_NEEDED         // FILESYS or FILESYS_STUB
#include "filesys.h"
extern FileSystem  *fileSystem;
#endif

#ifdef FILESYS
#include "synchdisk.h"
extern SynchDisk   *synchDisk;
#endif

#ifdef NETWORK
#include "post.h"
extern PostOffice* postOffice;
#endif
```

`system.cc`中实现了`Initialize(argc,argv)`函数，该函数在`main.cc`中被调用。

主要作用：**初始化Nachos全局数据结构。解释处理Nachos启动命令行参数，以确定初始化的标志**：

- argc："argc"是命令行参数的数量(包括命令的名称)——例如："nachos -d +" -> argc = 3
- argv："arqv"是一个字符串数组，每个字符串对应一个命令行参数，例如："nachos -d +" -> argv = {"nachos"， "-d"， "4"}

```
./nachos xx(参数)
一般选项：
-d: 显示特定的调试信息
-rs: 使得线程可以随机切换
-z: 打印版权信息
和用户进程有关的选项：
-s: 使用户进程进入单步调试模式
-x: 执行一个用户程序
-c: 测试终端输入输出
和文件系统有关的选项：
-f: 格式化模拟磁盘
-cp: 将一个文件从宿主机拷贝到 Nachos 模拟磁盘上
-p: 将 Nachos 磁盘上的文件显示出来
-r: 将一个文件从 Nachos 模拟磁盘上删除
-l: 列出 Nachos 模拟磁盘上的文件
-D: 打印出 Nachos 文件系统的内容
-t: 测试 Nachos 文件系统的效率
和网络有关的选项：
-n: 设置网络的可靠度（在 0-1 之间的一个小数）
-m: 设置自己的 HostID
-o: 执行网络测试程序
```

`Initialize(argc,argv)`代码如下：

```
void Initialize(int argc, char **argv)
{
    int argCount;
    char* debugArgs = (char*)"";
    bool randomYield = FALSE;

    //定义不同的宏执行不同的局部变量初始化操作
    // 例如
#ifdef USER_PROGRAM
    bool debugUserProg = FALSE;    // single step user program
#endif
    // ...

    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
        argCount = 1;
        if (!strcmp(*argv, "-d")) {
            if (argc == 1)
            debugArgs = (char*)"+";    // turn on all debug flags
            else {
                debugArgs = *(argv + 1);
                argCount = 2;
            }
        } else if (!strcmp(*argv, "-rs")) {
            ASSERT(argc > 1);
            RandomInit(atoi(*(argv + 1)));    // initialize pseudo-random
                            // number generator
            randomYield = TRUE;
            argCount = 2;
        }
    //定义不同的宏执行不同的命令行解释操作
    //例如：
#ifdef USER_PROGRAM
    if (!strcmp(*argv, "-s"))
        debugUserProg = TRUE;
#endif
    // ...
    }

    DebugInit(debugArgs);            // initialize DEBUG messages
    stats = new Statistics();            // collect statistics
    interrupt = new Interrupt;            // start up interrupt handling
    scheduler = new Scheduler();        // initialize the ready queue
    if (randomYield)                // start the timer (if needed)
        timer = new Timer(TimerInterruptHandler, 0, randomYield);

    threadToBeDestroyed = NULL;

    //在内核创建新线程之前，当前运行的线程
    currentThread = new Thread("main");
    currentThread->setStatus(RUNNING);
```

```
    interrupt->Enable();
    CallOnUserAbort(Cleanup);            // if user hits ctl-C

    //定义不同的宏执行不同的全局变量初始化操作
    //例如：
#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);    // this must come first
#endif
    //...
}
```

在内核创建新线程之前，当前运行的线程就是`Initialize()`中创建的main线程，他有以下特点：

- 它是由内核模块中的`main.cc`的`main()`函数启动的线程。
- 它是作为第一个"运行"线程诞生的。
- 它不是以`Fork(func, arg)`开始的。因此不遵循`ThreadRoot`定义的三个阶段。
- 它确实需要一个线程控制块来进行上下文切换。
- 它通过直接调用`Finish()`来终止自身。

转到`main.cc`，Nachos 内核`main()`函数是内核程序的启动入口。可以在`threads/main.cc`中看到：

```
int main (int argc, char **argv)
{
    (void) Initialize(argc, argv);// 初始化内核组件与第一个线程,定义在system.cc中
    //一系列预编译指令 main.cc编译在不同的文件夹下执行不同的功能
    currentThread -> Finish ();
    return (0); // 此行执行不到。
}
```

在`main`函数的最后，是`currentThread->Finish()`语句。为什么不直接退出呢?

这是因为 Nachos 是在宿主机上运行的一个普通的进程，当 main 函数退出时，整个占用的空间要释放，进程也相应的结束。但是实际上在 Nachos 中，**main 函数的结束并不能代表系统的结束**，因为可能还有其它的就绪线程。所以在这里我们只是将 main 函数作为 Nachos 中一个**特殊线程**进行处理，该线程结束只是作为一个线程的结束，系统并不会退出。这个特殊的线程将在上下文切换之后被下一线程删除。当所有线程都终止之后，Nachos 内核将从 Unix/Linux 系统中退出。

1.6 Nachos Thread分析

什么是线程，什么是线程和进程之间的不同?

线程包含于进程中，线程实际上是一个抽象的并发程序执行顺序.属于同一进程的多个线程共享着进程的正文和数据部分、标识以及进程资源。但是每个线程具有各自的寄存器和栈空间。

为什么我们需要把线程的栈和寄存器分开呢?

因为栈和寄存器集决定了程序执行中动态上下文的内容。栈保存了函数调用的返回点和传递的参数，而寄存器组保存了当前指令执行后的结果、状态和下条要执行指令的地址。 现在我们就有了一个分级的程序执行的结构：一个系统中可以具有多个进程而且每个进程可以具有多个线程，它们共享着进程的代码、数据、堆、标识和资源。 但进程和线程共享许多相同的概念：

- 状态转换
- 控制块
- 上下文切换

以下我们主要通过 NACHOS 来讨论线程的实现和控制。NACHOS 中的线程是由类`Thread`定义的。线程控制块是作为线程类中的一部分数据成员来说明的。`Thread.h`文件定义了`Thread`结构

```
#ifndef THREAD_H
#define THREAD_H

#include "copyright.h"
#include "utility.h"

#ifdef USER_PROGRAM
#include "machine.h"
#include "addrspace.h"
#endif


#define MachineStateSize 18  //存放寄存器指针的最大长度
#define StackSize    (sizeof(_int) * 1024)    // in words
// 线程状态
enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
//外部函数, dummy routine whose sole job is to call Thread::Print
extern void ThreadPrint(_int arg);

class Thread {
  private:
    int* stackTop;              // 指向整数的指针变量 stackTop 是当前栈顶指针 SP。
    _int machineState[MachineStateSize];  //其他的寄存器包括PC都被存储在数组元素类型为宿主机机器字长的数组中。
machineState[MachineStateSize]数组中
    int* stack;              // 指向整数的指针变量 stack 用于存储栈底(对栈溢出做检查)
                            // NULL if this is the main thread(If NULL, don't deallocate stack)
    ThreadStatus status;    // 保存了线程的状态：ready, running or blocked
    char* name;
    void StackAllocate(VoidFunctionPtr func, _int arg);
                        // Allocate a stack for thread Used internally by Fork()

  public:
    Thread(const char* debugName);    //线程对象的构造函数。仅仅是建立对象的数据结构和将对象状态设置为 JUST_CREATED。
    ~Thread();                // deallocate a Thread
                            // NOTE -- thread being deleted
                            // must not be running when delete is called

    // basic thread operations 线程状态转换控制原语
    void Fork(VoidFunctionPtr func, _int arg);     // Make thread run (*func)(arg)
    void Yield();                   // Relinquish the CPU if any other thread is runnable
    void Sleep();                   // Put the thread to sleep and relinquish the processor
    void Finish();              // The thread is done executing

    void CheckOverflow();       // 检查该进程的栈是否溢出
    void setStatus(ThreadStatus st) { status = st; }
```

```
    char* getName() { return (name); }
    void Print() { printf("%s, ", name); }
};
```

下面是定义在`Theard.h`中的线程状态转换控制原语，这些函数的具体实现在`Thread.cc`中。

需要说明的是，很多函数执行部分都用`IntStatus oldLevel = interrupt->SetLevel(IntOff)`与`(void) interrupt->SetLevel(oldLevel)`包裹起来，这两条语句的作用分别是关闭中断，保存原中断状态与恢复中断状态。`interrupt`是保存在`system.h`中的一个全局指针变量，这样做的原因是Nachos是**单线程操作系统**，保证函数执行的部分能是**原子操作**（所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch（切换到另一个线程））

- `Thread()`是线程对象的构造函数。它仅仅是建立对象的数据结构和将对象状态设置为`JUST_CREATED`。

  ```
  Thread::Thread(const char* threadName)
  {
      name = (char*)threadName;
      stackTop = NULL;
      stack = NULL;
      status = JUST_CREATED;
  }
  ```

- `Fork()`用于产生线程状态从`JUST_CREATE`到`READY`的状态转换，并生成线程实例可运行的环境。

  ```
  void Thread::Fork(VoidFunctionPtr func, _int arg)
  {
      StackAllocate(func, arg);

      IntStatus oldLevel = interrupt->SetLevel(IntOff);//关闭中断
      scheduler->ReadyToRun(this);    //调用线程调度器的ReadyToRun,assumes that
  interrupts are disabled!
      (void) interrupt->SetLevel(oldLevel);//恢复中断状态
  }
  ```

  `Fork`中调用了`StackAllcate(func, arg)`方法，用于分配栈空间同时初始化machineState[]数组

```
void Thread::StackAllocate (VoidFunctionPtr func, _int arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    stackTop = stack + StackSize - 4;     // -4 to be on the safe side!

    machineState[PCState] = (_int) ThreadRoot;//设置PC指针，使得每个线程从ThreadRoot
开始运行
    machineState[StartupPCState] = (_int) InterruptEnable;
    machineState[InitialPCState] = (_int) func;
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = (_int) ThreadFinish;
}
```

`ThreadRoot` 是一个定义在 `switch.s` 中的汇编语言函数，它是**每个线程首次执行时调用的过程。**

> 当新线程被调上 CPU 时，要用 SWITCH 函数切换线程，SWITCH 函数返回时，会从栈顶取出返回地址，**于是将 ThreadRoot 放在栈顶，在 SWITCH 结束后就会立即执行 ThreadRoot 函数。**ThreadRoot 是所有线程的入口，它会调用 Fork 的两个参数，运行用户指定的函数；

`InterruptEnable` 和 `ThreadFinish` 是定义在 `thread.cc` 中的两个静态函数，`InterruptEnable` 用于打开中断，`ThreadFinish` 用于终止线程的执行。 `func` 是**传入的线程执行函数入口地址（类比simpleThread函数）**， arg 是 func 所携带的参数，它俩都是由 Fork 函数的参数传递过来的。

- `Yield()`：**用于本线程放弃CPU转到就绪队列。**当就绪队列非空时将**当前调用的线程**状态从 `RUNNING` 转换为 `READY` 。它将当前进程（即调用 Yield 的线程）放入就绪队列尾部并且通过上下文切换将就绪队列中的一个线程变为运行状态。如果就绪队列为空，它没有任何作用并且继续运行当前线程。

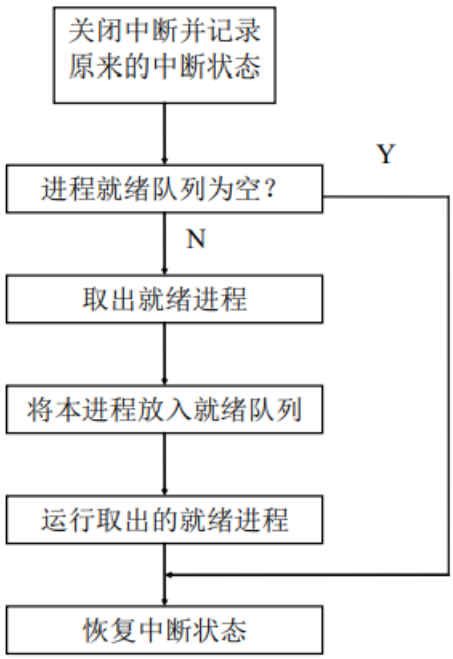


图 3.7　Yield 函数

```
void Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);//关中断

    ASSERT(this == currentThread);//其他线程不能调用另一个线程的yield()，只能自己放弃执行

    DEBUG('t', "Yielding thread \"%s\"\n", getName());
    //先寻找下一个线程再将自己推到ready队列中，保证不会切换到自己
    nextThread = scheduler->FindNextToRun();//利用线程调度器寻找下一个要执行的线程
    if (nextThread != NULL) {
        scheduler->ReadyToRun(this);// 将原线程放在等待队列中
        scheduler->Run(nextThread);// 调用run方法切换到nextThread线程
    }
    (void) interrupt->SetLevel(oldLevel);//恢复中断优先级
}
```

- `Sleep()`：。Sleep 方法可以使当前线程转入阻塞态，并放弃 CPU，直到被另一个线程唤醒，把它放回就绪线程队列。在没有就绪线程时，就把时钟前进到一个中断发生的时刻，让中断发生并处理此中断，这是因为在没有线程占用 CPU 时，只有中断处理程序可能唤醒一个线程，并把它放入就绪线程队列。线程要等到本线程被唤醒后，并且又被线程调度模块调上 CPU 时，才会从 Sleep 函数返回。

  具体来说就是将调用者线程从 `RUNNING` 转变为 `BLOCKED`，并从就绪队列中切换一个线程为运行。如果就绪队列为空，CPU 状态将变为空闲，直到有一个就绪线程要运行。

`Sleep()` 通常用于当线程开始 I/O 请求或要等待某个事件，它不能继续向前推进需要等待 I/O 完成或事件发生。**在调用这个函数之前，线程通常将自己放入对应的I/O等待或事件有关的队列中。**

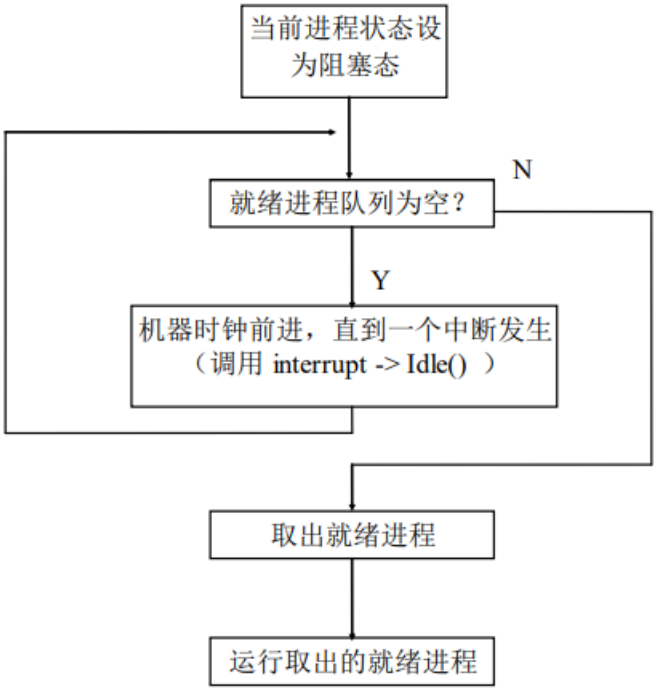


图 3.8　Sleep 函数

```
void Thread::Sleep ()
{
    Thread *nextThread;

    ASSERT(this == currentThread);//保证自己调用Sleep
    ASSERT(interrupt->getLevel() == IntOff);

    DEBUG('t', "Sleeping thread \"%s\"\n", getName());

    status = BLOCKED;
    while ((nextThread = scheduler->FindNextToRun()) == NULL)
        interrupt->Idle();        // no one to run, wait for an interrupt

    scheduler->Run(nextThread); // returns when we've been signalled
}
```

- `Finish()` 用于终止一个线程，实际上线程调用后会**不会删除自身**(C++**无法**delete this)，而是进入睡眠状态，直到下一个线程在**上下文切换**后负责销毁该线程。

```
void Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);

    DEBUG('t', "Finishing thread \"%s\"\n", getName());

    threadToBeDestroyed = currentThread;//将当前运行的线程的指针保存在一个全局变量
中，下一个线程负责Destory该线程
    Sleep();                    // invokes SWITCH
    // not reached
}
```

全局变量`threadToDestoryed`保存在`system.h`中

```
extern Thread *threadToBeDestroyed;          // the thread that just finished
```

要删除的线程声明它应该被删除，通过设置全局变量`threadToBeDestroyed`来指向自己。接下来，这个线程将控制权转移给另一个线程。新线程在Run方法中删除要删除的控件。

1.7 Nachos Schedule分析

一个线程或进程在他们的生命期间将通过许多次状态切换。在所有这些状态中**就绪队列**用于放置所有就绪状态的线程或进程。 其他队列对应的放置在因为申请不同 I/O 设备而处于阻塞状态的进程或线程，它们等待响应 I/O 请求的完成。线程或进程由**作业调度者**在队列中按调度策略移动。

在Nachos中，线程调度是由定义在`scheduler.h`和`scheduler.cc`的`Scheduler`类的一个全局对象来完成的。这个类的方法提供了线程和进程的所有调度功能。当 Nachos 首次启动时，首先在`system.h`建立一个`Scheduler`类的全局实例对象的引用`*scheduler`,由它负责完成线程或进程的调度任务。这个类的定义见`scheduler.h`文件。

```
extern Scheduler *scheduler;    // the ready list
```

下面是对`Scheduler`类的分析：

```
class Scheduler {
  public:
    Scheduler();            // Initialize list of ready threads
    ~Scheduler();            // De-allocate ready list

    void ReadyToRun(Thread* thread);    // Thread can be dispatched.
    Thread* FindNextToRun();        // Dequeue first thread on the ready
                                    // list, if any, and return thread.
    void Run(Thread* nextThread);    // Cause nextThread to start running
    void Print();            // Print contents of ready list

  private:
    List *readyList;          // queue of threads that are ready to run,
                              // but not running
};
```

`Scheduler`类仅有一个私有对象它就是指向`list`对象的一个指针（见 list.h 和 list.cc）。`readyList`存放着所有`status = READY`的线程，可以将其理解为一个就绪队列。

- `ReadyToRun(Thread* thread)`：将一个线程推入该队列尾

  ```
  void Scheduler::ReadyToRun (Thread *thread)
  {
      DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

      thread->setStatus(READY);
      readyList->Append((void *)thread);// nachos默认实现先来先服务的调度
  }
  ```

- `FindNextToRun()`：从队列返回出队线程的**指针**（或 NULL,如果队列为空）。

  ```
  Thread* Scheduler::FindNextToRun ()
  {
      return (Thread *)readyList->Remove();
  }
  ```

- `run(Thread* thread)`：这个函数调用汇编语言函数`SWITCH(Thread*,Thread*)`将当前线程切换到由第二参数指向的另一线程。

  函数`Scheduler::Run(Thread *nextThread)`首先将`currentThread`保存到变量`oldThread`中并将`currentThread`指向`nextThread`所指向的线程对象。然后**调用汇编函数`SWITCH(oldThread, nextThread)`真正实现当前运行线程的切换。**

```
void Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;
    oldThread->CheckOverflow();        // check if the old thread
                                       // had an undetected stack overflow
    currentThread = nextThread;            // currentThread切换到下一个线程
    currentThread->setStatus(RUNNING);  // 设置新线程的状态为RUNNING

    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",
      oldThread->getName(), nextThread->getName());

    SWITCH(oldThread, nextThread); // oldThread会进入等待状态，已经切换到nextThread
                                   // 线程中执行，oldThread需要等待其他线程的SWITCH
    DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
    if (threadToBeDestroyed != NULL) {//由新的线程销毁记录在threadToBeDestroyed的线
程
        delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }
}
```

整个`Run`函数运行于内核，因为它属于`Nachos`内核进程。

注意调用`Run`函数的`oldThread`线程它不会立即返回，实际上它将**不会自动返回**，而是进入等到状态，**此时系统已经开始执行`nextThread`新线程，直到有系统调度事件发生其他线程调用`run(oldThread)`后才可能被切换回来再次成为当前线程继续运行。**

1.8 Nachos Semaphore分析

1.8.1 Nachos Semaphore

一个 Nachos 中的信号量是作为 `Semaphore` 类的对象实现的。 `Semaphore` 类的定义可 以在 threads/synch.h 中找到。其算法可以描述为：

```
P(){
   While(信号量的值 V=0)
     将调用者线程推入阻塞队列 B，调用者线程阻塞；
   V = V-1
}
V(){
   If (阻塞队列B非空){
     从阻塞队列B中取出一个线程；
     把它推入系统就绪队列R；
   }
   V = V+1;
}
```

`Semaphore` 类的定义如下：

```
class Semaphore {
  public:
    Semaphore(const char* debugName, int initialValue);    // set initial value
    ~Semaphore();                                          // de-allocate semaphore
    char* getName() { return name;}                        // debugging assist

    void P();     // these are the only operations on a semaphore
    void V();     // they are both *atomic*

  private:
    char* name;        // useful for debugging
    int value;         // 资源可利用量, always >= 0
    List *queue;       // threads waiting in P() for the value to be > 0 queue
};

Semaphore::Semaphore(const char* debugName, int initialValue)
{
    name = (char*)debugName;
    value = initialValue;
    queue = new List;
}
Semaphore::~Semaphore()
{
    delete queue;
}

void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);    // disable interrupts

    while (value == 0) {             // semaphore not available
        queue->Append((void *)currentThread);    // 将当前线程加入阻塞队列
        currentThread->Sleep();
    }
    value--;                     // semaphore available, consume its value

    (void) interrupt->SetLevel(oldLevel);    // re-enable interrupts
}

void Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)       // make thread ready, consuming the V immediately
        scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

每个信号量维护一个队列`queue`用于指向所有在该信号量上阻塞的线程。`name`表示当前信号量的名称，`value`表示当前信号量的信号值。

PV操作都要保证为原子操作。因此都需要开关中断。Nachos 中的 P()、V()操作关键的概念是保持信号量的信号值始终大于等于 0。即信号量的值代表了资源可利用量，当资源量等于 0 时说明线程无资源可用必需等待可用资源的释放。

如果在使用 while 语句的地方使用了 if 语句，一些条件可能引起信号量的值小于 0，从而发生错误。

> 1. 线程 A 因请求该资源，引用 P()操作而阻塞；
> 2. 线程 B 释放该资源，引用 V()操作唤醒了线程 A，将 A 推入了就绪队列，使 V=1；
> 3. 线程 C 首先从就绪队列中被选中执行，C 也请求该资源，引用 P()操作，使 V=0；开始访问该资源；
> 4. 线程 A 从就绪队列中被选中执行，如果这里使用 if 语句，线程 A 不会再去判断 V 是否等于 0,而是使 V=-1，也开始访问该资源，从而发生了与线程 C 非互斥 的使用同一资源的错误。而如果这里使用 while 语句，线程 A 会发现 V 再次等于 0,而再次进入阻塞队列，保证了 V 的值始终大于等于 0，从而避免了与线程 C 同时使用同一资源的错误。
> 5. **由此可见，那些由 V()操作唤醒刚进入就绪队列的线程仍然被当作阻塞态线程，他们还需要进行一次判断信号量的操作，因为它们还未完成它们调用 P()操作中递减 V 值的工作。**

1.8.2 ring分析

环形缓冲类定义在`ring.h`中，包括环形缓冲区的内元素`slot`——插槽的定义。

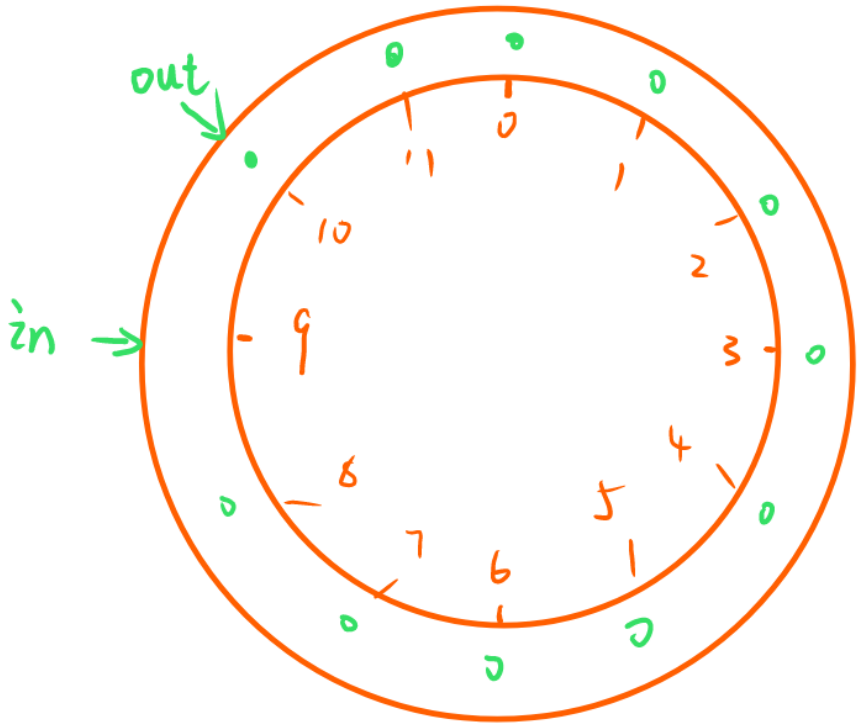
```
class slot {
    public:
        slot(int id, int number);
        slot() { thread_id = 0; value = 0;};

    int thread_id;
    int value;
};

slot::slot(int id, int number)
{
    thread_id = id;
    value = number;
}
```

一个先进先出的环形的缓冲区分别有两个指针`in,out`：

- `in`：新的`slot`放入缓冲区的位置
- `out`：当前最旧的`slot`的下标

当缓冲区的已被填满时，第一个插入缓冲区的`slot`将被下一个新的`slot`覆盖，达到一个环形的效果，因此缓冲区实时大小为`(in-out)%size-1`(数组下标从0开始)：

- 判断缓冲区为空：`in-out=0`
- 判断缓冲区已满：`(in+1)%size=out`，即out指针在in指针的后一位。



假设siez=12，当in=9，out=10时表示缓冲区已被填满了。

```
class Ring {
  public:
    Ring(int sz);    // Constructor:  sz表示插槽的数量
    ~Ring();         // Destructor:
    void Put(slot *message); // Put a message the next empty slot.

    void Get(slot *message); // Get a message from the next full slot.

    int Full();       // Returns non-0 if the ring is full, 0 otherwise.
    int Empty();      // Returns non-0 if the ring is empty, 0 otherwise.

  private:
    int size;         // 缓冲区内插槽的数量.
    int in, out;      // Index of
    slot *buffer;     // 缓冲区数组，保存插槽

};

Ring::Ring(int sz)
{
    if (sz < 1) {
        fprintf(stderr, "Error: Ring: size %d too small\n", sz);
        exit(1);
    }

    // Initialize the data members of the ring object.
    size = sz;
    in = 0;
    out = 0;
    buffer = new slot[size]; //allocate an array of slots.
}

Ring::~Ring()
{
    // Some compilers and books tell you to write this as:
    // delete [size] stack;
    // but apparently G++ doesn't like that.
    delete [] buffer;
}


// 将一个新的插槽放入缓冲区的in处
void Ring::Put(slot *message)
{
    buffer[in].thread_id = message->thread_id;
    buffer[in].value = message->value;
    in = (in + 1) % size;
}

void Ring::Get(slot *message)
```

```
{
    message->thread_id = buffer[out].thread_id;
    message->value = buffer[out].value;
    out = (out + 1) % size;
}
int Ring::Empty()
{
    return in == out;
}
int Ring::Full()
{
    return ((in + 1) % size) == out;
}
```

1.9 Nachos文件系统

1.9.1 Nachos文件系统的组织结构

Nachos文件系统使用7个模块实现了文件系统的5个层次的基本功能，它们是：

- Filesys
- Directory
- OpenFile
- FileHeader
- BitMap
- SynchDisk
- Disk

关系如下图所示：

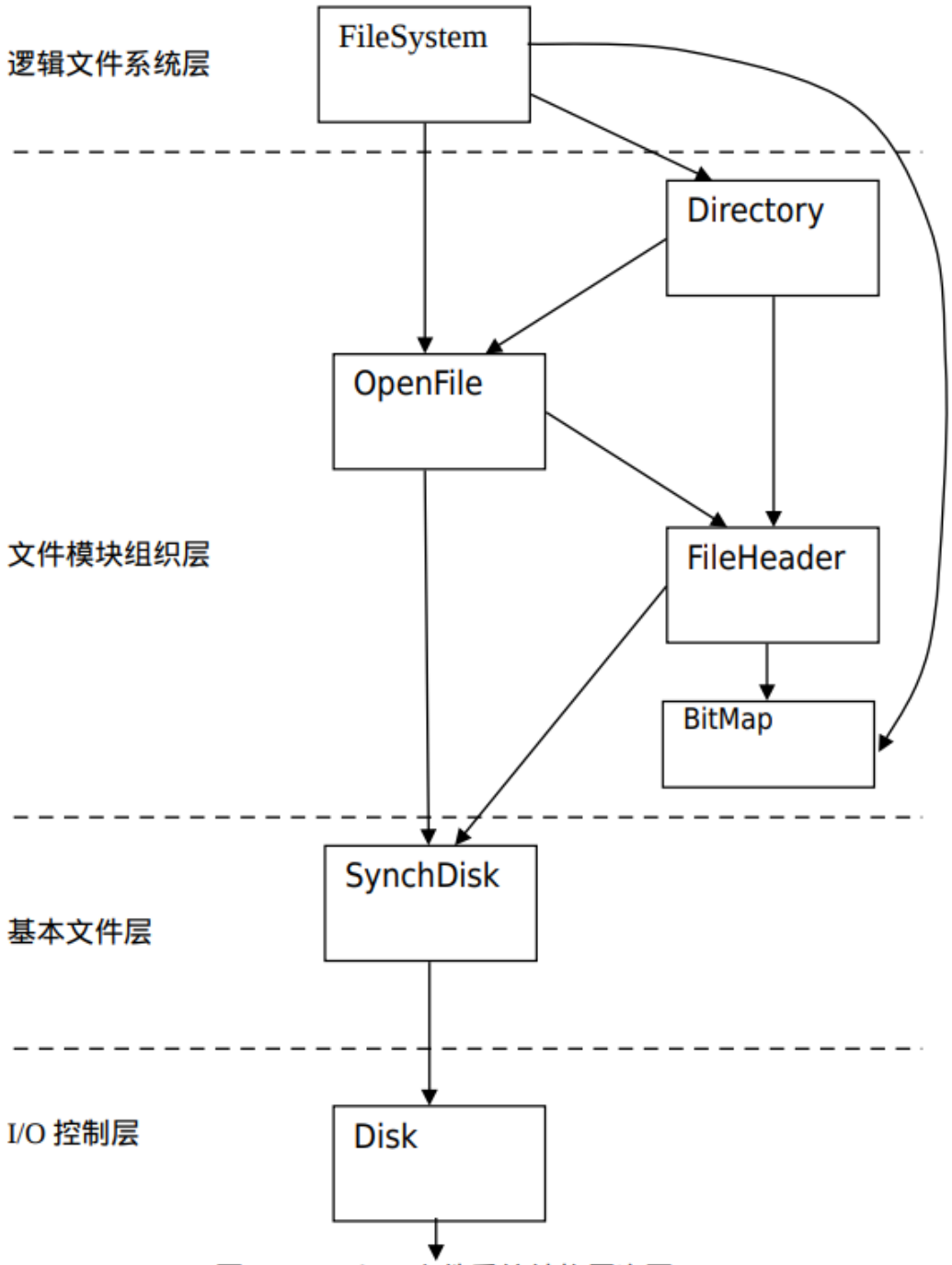


图 4.1 Nachos 文件系统结构层次图

在当前的Nacho文件系统中磁盘的I/O中断驱动是由两个模块模拟的：Disk、SynchDisk。

Disk模拟了设备控制和磁盘的自身驱动，SynchDisk模拟了系统内核I/O系统的一部分。

1.9.2 Nachos Disk分析

模拟磁盘的物理参数：

```
#define SectorSize          128    // 每个磁盘扇区的字节数
#define SectorsPerTrack      32    // 每个磁盘磁道扇区数
#define NumTracks           32    // 每个磁盘的磁道数
#define NumSectors          (SectorsPerTrack * NumTracks)    // 每个磁盘扇区的总数
```

从中我们可以看到磁盘预定义的容量 = SectorSize * SectorsPerTrack * NumTracks = 128KB

Disk类中的方法以及一些参数

```
class Disk {
  public:
    Disk(const char* name, VoidFunctionPtr callWhenDone, _int callArg);
                        // Create a simulated disk.
                    // Invoke (*callWhenDone)(callArg)
                    // every time a request completes.
    ~Disk();                // Deallocate the disk.

    void ReadRequest(int sectorNumber, char* data);
                        // Read/write an single disk sector.
                    // These routines send a request to
                        // the disk and return immediately.
                        // Only one request allowed at a time!
    void WriteRequest(int sectorNumber, char* data);

    void HandleInterrupt();        // Interrupt handler, invoked when
                    // disk request finishes.

    int ComputeLatency(int newSector, bool writing);
                        // Return how long a request to
                    // newSector will take:
                    // (seek + rotational delay + transfer)

  private:
    int fileno;                // UNIX file number for simulated disk
    VoidFunctionPtr handler;        // 要调用的中断处理程序，当任何磁盘请求完成时
    _int handlerArg;            // 中断处理程序的参数
    bool active;                // Is a disk operation in progress?
    int lastSector;            // The previous disk request
    int bufferInit;            // 当轨道缓冲区开始加载时

    int TimeToSeek(int newSector, int *rotate); // time to get to the new track
    int ModuloDiff(int to, int from);        // # sectors between to and from
    void UpdateLast(int newSector);
};
```

在硬盘由Disk (char* name, VoidFunctionPtr callWhenDone, _int callArg) 构造时，私有的类成员函数 **handler** 被设置为指向基本文件层磁盘管理程序**callWhenDone**，其参数**callArg**也被保存在

私有类成员变量handlerArg中。

这样当一次硬盘操作完成时可以回调该函数完成硬盘中断处理。

Disk的构造函数

```
Disk::Disk(const char* name, VoidFunctionPtr callWhenDone, _int callArg)
{
    int magicNum;
    int tmp = 0;

    DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone, callArg);
    handler = callWhenDone;
    handlerArg = callArg;
    lastSector = 0;
    bufferInit = 0;

    fileno = OpenForReadWrite((char*)name, FALSE);
    if (fileno >= 0) {             // file exists, check magic number
    Read(fileno, (char *) &magicNum, MagicSize);
    ASSERT(magicNum == MagicNumber);
    } else {                // file doesn't exist, create it
        fileno = OpenForWrite((char*)name);
    magicNum = MagicNumber;
    WriteFile(fileno, (char *) &magicNum, MagicSize); // write magic number

    // need to write at end of file, so that reads will not return EOF
        Lseek(fileno, DiskSize - sizeof(int), 0);
    WriteFile(fileno, (char *)&tmp, sizeof(int));
    }
    active = FALSE;
}
```

ReadRequest函数 和 WriteRequest函数

```
void
Disk::ReadRequest(int sectorNumber, char* data)
{
    int ticks = ComputeLatency(sectorNumber, FALSE);

    ASSERT(!active);                // only one request at a time
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Reading from sector %d\n", sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    Read(fileno, data, SectorSize);
    if (DebugIsEnabled('d'))
    PrintSector(FALSE, sectorNumber, data);

    active = TRUE;
    UpdateLast(sectorNumber);
    stats->numDiskReads++;
    interrupt->Schedule(DiskDone, (_int) this, ticks, DiskInt);
}

void
Disk::WriteRequest(int sectorNumber, char* data)
{
    int ticks = ComputeLatency(sectorNumber, TRUE);

    ASSERT(!active);
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Writing to sector %d\n", sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    WriteFile(fileno, data, SectorSize);
    if (DebugIsEnabled('d'))
    PrintSector(TRUE, sectorNumber, data);

    active = TRUE;
    UpdateLast(sectorNumber);
    stats->numDiskWrites++;
    interrupt->Schedule(DiskDone, (_int) this, ticks, DiskInt);
}
```

在说明的扇被从UNIX文件中读出或者写入后，active被设置为TRUE，并且模拟系统中断，注册一个在将来tick时间后将要发生的事件。这个事件到达的时间代表了磁盘操作完成的时间。当事件到达时，硬中断模拟器将通过中断处理程序DiskDone，用事先传递给它的参数完成对硬盘操作的中断处理

DiskDone函数

```
static void DiskDone(_int arg) { ((Disk *)arg)->HandleInterrupt(); }
```

它简单的使用了一个参数arg指向一个Disk对象，并且调用该对象的>HandleInterrupt()函数。

HandleInterrupt函数

```
void
Disk::HandleInterrupt ()
{
    active = FALSE;
    (*handler)(handlerArg);
}
```

可以看到该函数重新将变量active设置为FALSE，并调用基本文件层在构造硬盘对象时安排的中断管理函数完成最终的硬盘中断处理。

1.9.3 Nachos SynchDisk分析

这一层也叫"基本文件系统"。关于硬盘I/O中断驱动，我们应当提供一种机制去阻止进程在没有完成I/O处理就向硬盘连续发I/O访问请求。硬盘管理程序也应提供在磁盘I/O完成后的中断处理。磁盘管理程序的另一项任务是按多进程或线程去同步当前磁盘的访问。所有这些对磁盘同步的管理任务在Nachos文件系统中都是由文件synchdisk.h中定义的SynchDisk类完成的。

SynchDisk类

```
class SynchDisk {
  public:
    SynchDisk(const char* name);          // Initialize a synchronous disk,
                // by initializing the raw Disk.
    ~SynchDisk();        // De-allocate the synch disk data

    void ReadSector(int sectorNumber, char* data);
                    // Read/write a disk sector, returning
                    // only once the data is actually read
                // or written.  These call
                    // Disk::ReadRequest/WriteRequest and
                // then wait until the request is done.
    void WriteSector(int sectorNumber, char* data);

    void RequestDone();          // Called by the disk device interrupt
                // handler, to signal that the
                // current disk operation is complete.

  private:
    Disk *disk;              // Raw disk device
    Semaphore *semaphore;     // To synchronize requesting thread
                // with the interrupt handler
    Lock *lock;              // Only one read/write request
                // can be sent to the disk at a time
};
```

这里的信号量显然是用来控制被阻塞进程的,同时锁被用来提供互斥的访问磁盘。当然这个类必须要访问的是硬盘DISK。

函数RequestDone()用于当I/O请求完成后释放磁盘中断处理程序。

同步化的I/O操作函数：ReadSector(int sectorNumber, char* data)和WriteSector(int sectorNumber, char* data)是在附加了同步操作再调用对应的裸盘上的I/O操作实现的。

ReadSector函数和WriteSector函数

```
void
SynchDisk::ReadSector(int sectorNumber, char* data)
{
    lock->Acquire();         // only one disk I/O at a time
    disk->ReadRequest(sectorNumber, data);
    semaphore->P();          // wait for interrupt
    lock->Release();
}

//----------------------------------------------------------------------
// SynchDisk::WriteSector
//     Write the contents of a buffer into a disk sector.  Return only
// after the data has been written.
//
// "sectorNumber" -- the disk sector to be written
// "data" -- the new contents of the disk sector
//----------------------------------------------------------------------

void
SynchDisk::WriteSector(int sectorNumber, char* data)
{
    lock->Acquire();         // only one disk I/O at a time
    disk->WriteRequest(sectorNumber, data);
    semaphore->P();          // wait for interrupt
    lock->Release();
}

//----------------------------------------------------------------------
// SynchDisk::RequestDone
//     Disk interrupt handler.  Wake up any thread waiting for the disk
// request to finish.
//----------------------------------------------------------------------
```

lock用于互斥多线程对于磁盘的访问，等待访问磁盘的线程被放置在lock等待队列中。信号量semaphore用于同步正在访问磁盘的线程对于磁盘的操作，一个线程在发出了磁盘操作命令后将被放置在中断事件队列中直到裸盘访问操作完成后再被唤醒。当磁盘I/O完成后，裸盘的中断处理程序将重新调用带有一个整形参数的函数DiskRequestDone，而这个函数实际上又调用了磁盘同步函数RequestDone。函数RequestDone的工作是调用semaphore信号量上的V()操作，因此唤醒了阻塞在中断事件队列中的磁盘访问线程。

1.9.4 Nachos BitMap分析

扇是磁盘中最小的存储单位，也是磁盘I/O操作最基本的单位。在Nachos中以扇区为基本单位的自由存储空间是由位示图类BitMap管理的。

BitMap类

```
class BitMap {
  public:
    BitMap(int nitems);       // Initialize a bitmap, with "nitems" bits
            // initially, all bits are cleared.
    ~BitMap();       // De-allocate bitmap

    void Mark(int which);      // Set the "nth" bit
    void Clear(int which);     // Clear the "nth" bit
    bool Test(int which);      // Is the "nth" bit set?
    int Find();                // Return the # of a clear bit, and as a side
            // effect, set the bit.
            // If no bits are clear, return -1.
    int NumClear();       // Return the number of clear bits

    void Print();     // Print contents of bitmap

    // These aren't needed until FILESYS, when we will need to read and
    // write the bitmap to a file
    void FetchFrom(OpenFile *file);    // fetch contents from disk
    void WriteBack(OpenFile *file);    // write contents to disk

  private:
    int numBits;         // number of bits in the bitmap
    int numWords;        // number of words of bitmap storage
                // (rounded up if numBits is not a
                //  multiple of the number of bits in
                //  a word)
    unsigned int *map;       // bit storage
};
```

私有成员变量map指向一个保存位示图的内存,变量numBits保存了能表示扇号的位数，numWords保存了组成位示图的字数。如果要占用一个扇则与其扇号对应的位就置”1”,如果要释放一个扇则与其扇号对应的位就置”0”。注意函数find()的作用，它返回找到的第一个空闲位的索引同时将该位置”1”。因为内存是易失性的，所以对应硬盘的位示图需要作为一个文件保存到磁盘上。它作为内核一个特殊文件被管理。函数FetchFrom(OpenFile *file)和WriteBack(OpenFile *file)用于完成这一目的。

1.9.5 Nachos FileHeader分析

FileHeader类

在基本Nachos系统中使用File Header类仅实现了简单的一级索引分配方法及其结构。

```
class FileHeader {
  public:
    bool Allocate(BitMap *bitMap, int fileSize);// Initialize a file header,
                  //  including allocating space
                  //  on disk for the file data
    void Deallocate(BitMap *bitMap);      // De-allocate this file's
                  //  data blocks

    void FetchFrom(int sectorNumber);  // Initialize file header from disk
    void WriteBack(int sectorNumber);  // Write modifications to file header
              //  back to disk

    int ByteToSector(int offset);  // Convert a byte offset into the file
              // to the disk sector containing
              // the byte

    int FileLength();        // Return the length of the file
              // in bytes

    void Print();        // Print the contents of the file.

  private:
    int numBytes;        // Number of bytes in the file
    int numSectors;          // Number of data sectors in the file
    int dataSectors[NumDirect];       // Disk sector numbers for each data
              // block in the file
};
```

这里的dataSectors[NumDirect]是一个扇区号的索引表，它记录了一个文件所占用的扇区号。numBytes记录了文件的字节长度，numSectors记录了文件占用的扇区数。这三个私有数据成员组成了Nachos文件的一个I-node。

其中有两个重要的常量定义：

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
```

NumDirect是索引表dataSectors的长度，它是一个Nachos文件所能占用的数据块的最大数字。它也决定了一个Nachos文件的最大容量MaxFileSize的值。

Allocate函数

在Nachos文件系统中当要建立一个新文件时首先调用FileHeader类的成员函数Allocate(BitMap *bitMap, int fileSize)。

```
bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors  = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numSectors)
   return FALSE;     // not enough space

    for (int i = 0; i < numSectors; i++)
   dataSectors[i] = freeMap->Find();
    return TRUE;
}
```

这个函数根据指定的文件长度fileSize换算出文件数据要占用的磁盘扇区数，并将申请到的自由扇区号登记到扇区索引表dataSectors中。即它完成了一个Nachos文件I-node的初始化工作。

Deallocate函数

```
void
FileHeader::Deallocate(BitMap *freeMap)
{
    for (int i = 0; i < numSectors; i++) {
   ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
   freeMap->Clear((int) dataSectors[i]);
    }
}
```

释放一个扇区。

1.9.6 Nachos OpenFile分析

一个打开的文件是文件系统中的一个层次。正像Unix系统中一样，Nachos系统中的每个进程都具有一个打开文件表。打开文件的数据结构应提供：

- 打开文件当前的位移位置
- 打开文件I-node的一个引用
- 打开文件的访问函数如read或write

OpenFile类

```
class OpenFile {
  public:
    OpenFile(int sector);     // Open a file whose header is located
                // at "sector" on the disk
    ~OpenFile();         // Close the file

    void Seek(int position);      // Set the position from which to
                // start reading/writing -- UNIX lseek

    int Read(char *into, int numBytes); // Read/write bytes from the file,
                // starting at the implicit position.
                // Return the # actually read/written,
                // and increment position in file.
    int Write(char *from, int numBytes);

    int ReadAt(char *into, int numBytes, int position);
                    // Read/write bytes from the file,
                // bypassing the implicit position.
    int WriteAt(char *from, int numBytes, int position);

    int Length();        // Return the number of bytes in the
                // file (this interface is simpler
                // than the UNIX idiom -- lseek to
                // end of file, tell, lseek back

  private:
    FileHeader *hdr;         // Header for this file
    int seekPosition;        // Current position within the file
};
```

可以看到在类OpenFile中除了说明了一系列的读写函数外，seekPosition说明了打开文件的位移位置，hdr说明了打开文件的I-node。

这个类的构造函数怎样从磁盘上装入一个I-node和设置一个位移量为零的OpenFile的：

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
}
```

Openfile类中的成员函数Seek(..)和Length()是很直观的。函数Read(…)和Write(…)分别是由调用下层的ReadAt(…)和WriteAt(…)实现的，在此就不一一分析了。

1.9.7 Nachos directory分析

Directory类

```
class Directory {
  public:
    Directory(int size);      // Initialize an empty directory
              // with space for "size" files
    ~Directory();        // De-allocate the directory

    void FetchFrom(OpenFile *file);    // Init directory contents from disk
    void WriteBack(OpenFile *file);    // Write modifications to
              // directory contents back to disk

    int Find(char *name);     // Find the sector number of the
              // FileHeader for file: "name"

    bool Add(char *name, int newSector);  // Add a file name into the directory

    bool Remove(char *name);      // Remove a file from the directory

    void List();         // Print the names of all the files
              //  in the directory
    void Print();        // Verbose print of the contents
              //  of the directory -- all the file
              //  names and their contents.

  private:
    int tableSize;       // Number of directory entries
    DirectoryEntry *table;    // Table of pairs:
              // <file name, file header location>

    int FindIndex(char *name);    // Find the index into the directory
              //  table corresponding to "name"
};
```

其中的类DirectoryEntry定义了Nachos文件系统的目录结构

```
class DirectoryEntry {
  public:
    bool inUse;             // Is this directory entry in use?
    int sector;             // Location on disk to find the
              //   FileHeader for this file
    char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
              // the trailing '\0'
};
```

DirectoryEntry类中的布尔变量inUse用于指示这个目录项是否已占用。字符数组name是一个文件名串。变量sector指示该文件的I-node所在的扇号。

Nachos中的每个文件都有一个DirectoryEntry类，多个文件就构成了Nachos的文件目录表，Directory类中的私有数据成员table指向了这个表的首项。

目录本身也是一个文件，它需要经常从磁盘中取出和写回。Director类中成员函数FetchFrom(OpenFile *file)和WriteBack(OpenFile *file)用于此目的。函数Add(char *name, int newSector)和Remove(char *name)则用于从目录表中加入或删除一条目录项。

Add函数

```
bool
Directory::Add(char *name, int newSector)
{
    if (FindIndex(name) != -1)
   return FALSE;

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse) {
            table[i].inUse = TRUE;
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
        return TRUE;
   }
    return FALSE;  // no space.  Fix when we have extensible files.
}
```

基本Nachos文件中的目录表的长度是固定的，因此目录文件的长度也被固定了，所以这个函数仅简单的在表中查找第一个未用的目录入口并向里放一个文件名和I-node扇号。

函数List()类似Unix系统的"ls"命令，列出文件目录中所有的文件名。

1.9.8 Nachos FileSystem分析

FileSystem类

```
class FileSystem {
  public:
    FileSystem(bool format);      // Initialize the file system.
               // Must be called *after* "synchDisk"
               // has been initialized.
                   // If "format", there is nothing on
               // the disk, so initialize the directory
                   // and the bitmap of free blocks.

    bool Create(char *name, int initialSize);
               // Create a file (UNIX creat)

    OpenFile* Open(char *name);    // Open a file (UNIX open)

    bool Remove(char *name);      // Delete a file (UNIX unlink)

    void List();         // List all the files in the file system

    void Print();        // List all the files and their contents

  private:
   OpenFile* freeMapFile;     // Bit map of free disk blocks,
               // represented as a file
   OpenFile* directoryFile;       // "Root" directory -- list of
               // file names, represented as a file
};
```

FileSystem::FileSystem函数

```
FileSystem::FileSystem(bool format)
{
    DEBUG('f', "Initializing the file system.\n");
    if (format) {
        BitMap *freeMap = new BitMap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
   FileHeader *mapHdr = new FileHeader;
   FileHeader *dirHdr = new FileHeader;

        DEBUG('f', "Formatting the file system.\n");

    // First, allocate space for FileHeaders for the directory and bitmap
    // (make sure no one else grabs these!)
   freeMap->Mark(FreeMapSector);
   freeMap->Mark(DirectorySector);

    // Second, allocate space for the data blocks containing the contents
    // of the directory and bitmap files.  There better be enough space!

   ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
   ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

    // Flush the bitmap and directory FileHeaders back to disk
    // We need to do this before we can "Open" the file, since open
    // reads the file header off of disk (and currently the disk has garbage
    // on it!).

        DEBUG('f', "Writing headers back to disk.\n");
   mapHdr->WriteBack(FreeMapSector);
   dirHdr->WriteBack(DirectorySector);

    // OK to open the bitmap and directory files now
    // The file system operations assume these two files are left open
    // while Nachos is running.

        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);

    // Once we have the files "open", we can write the initial version
    // of each file back to disk.  The directory at this point is completely
    // empty; but the bitmap has been changed to reflect the fact that
    // sectors on the disk have been allocated for the file headers and
    // to hold the file data for the directory and bitmap.

        DEBUG('f', "Writing bitmap and directory back to disk.\n");
   freeMap->WriteBack(freeMapFile);    // flush changes to disk
   directory->WriteBack(directoryFile);

   if (DebugIsEnabled('f')) {
       freeMap->Print();
```

```
        directory->Print();

         delete freeMap;
    delete directory;
    delete mapHdr;
    delete dirHdr;
    }
     } else {
     // if we are not formatting the disk, just open the files representing
     // the bitmap and directory; these are left open while Nachos is running
         freeMapFile = new OpenFile(FreeMapSector);
         directoryFile = new OpenFile(DirectorySector);
     }
}
```

在这个文件系统构造函数中生成了目录和位示图文件的空间，并打开了这两个文件准备好对于用户文件的操作。

对于目录和位示图文件的长度进行了定义：

```
#define FreeMapFileSize     (NumSectors / BitsInByte)   //扇区的大小
#define NumDirEntries     10      // 加载到磁盘上的最大文件数量
#define DirectoryFileSize  (sizeof(DirectoryEntry) * NumDirEntries)    //目录的大小
```

NumDirEntries定义了目录表项数，它是可建立的最大文件数。FreeMapFileSize定义了BitMap文件的长度，DirectoryFileSize定义了目录文件的长度。 分析一下文件系统构造函数首次工作的过程。首先它建立一个位示图和一个目录对象(84-85)，接着为这两个对象建立对应的文件头对象(86-87)，并在位示图中置位它们所占用的扇号（位示图文件为0，目录文件为1）。接下来要把这两对象的I-node保存到磁盘上(108-109)。现在可以打开这两个文件了(115-116)，这两个文件在内存中已经有了新的的内容，将这些新内容写入其对应的文件中(125-126)。现在内存中的对象freeMap 、directory、mapHdr、dirHdr已经不再有用，可以将其从内存中删除了(132-135)。

1.9.9 Nachos 创建文件

- 文件系统最高层——Filesys调用它的成员函数Create(char *name, int initialSize)
- 然后访问Directory,判断是否还有空目录项
- 访问Bitmap，查看是否有空扇区，来放置header
- 如果有空间，增加新目录项，创建新的FileHeader，在hdr中给文件分配空间，再次访问Bitmap，检查是否还有空扇区
- 如果空间足够，文件allocate成功，将header写回磁盘（位置是第三步分配的扇区），将目录写回磁盘

1.9.10 Nachos 读写文件

- 文件系统最高层——Filesys调用它的成员函数Open(char *name)
- 然后访问Directory，根据文件名Find到该文件i-node扇区位置
- OpenFile利用i-node位置，完成文件打开操作
- 在OpenFile里调用 Write(char *from, int numBytes)，开始写文件
- 写的时候，会访问header，查看FileSize等属性
- 写文件调用SynchDisk，SynchDisk再调用Disk

1.9.11 文件为什么不可扩展

由前两步，对文件创建，读写的操作流程，可以看出。

- 一个文件的大小是在创建时定的，在header，allocate时候，传入参数fileSize，按fileSize分配恰好合适大小的扇区数量
- 在openfile的write时，当写的位置超过fileSize时，会直接切掉超出部分，只是write不超出部分

1.10 Nachos内存管理

为了在计算机上运行一个应用程序，我们应当进行一些什么工作呢？主要涉及到三 个步骤：

1. 编译和转换源代码到它的目标代码
2. 连接程序将所有的标代码模块链接成一个单一的可执行模块，即众所周知的可装入模块。
3. 装入程序将可装入模块装入计算机内存。

装入模块 `addrSpace` 提供程序的逻辑地址空间。这个地址空间的开始地址是 0，并且模块所有的地址都参考这个开始的 0 地址。 在程序运行之前，所有的逻辑地址空间都需要变换为物理地址空间。这一变换通常都由**硬件的内存管理部件** MMU 完成。操作系统可以由改变重定位寄存器的值来重定位程序的装入模块到内存不同的段上。

基本的 Nachos 系统是采用**页式内存分配方式**管理用户内存空间的：

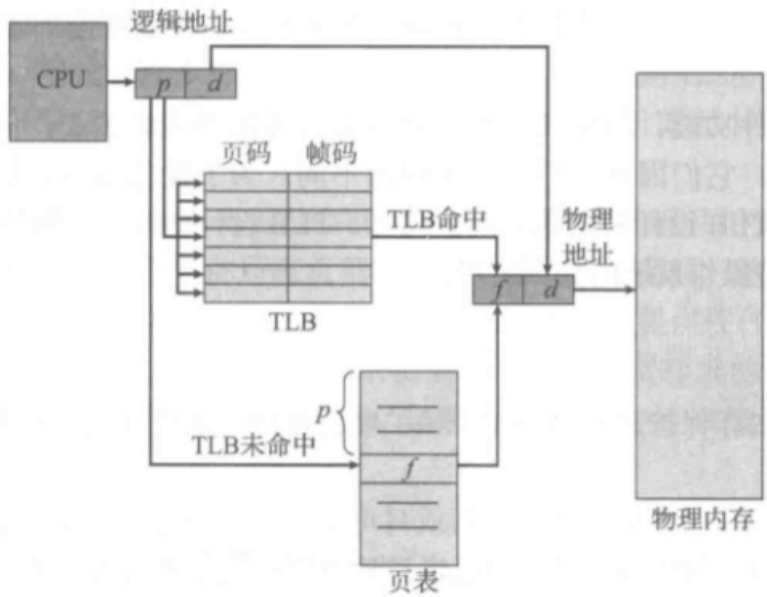


图 8-14　带 TLB 的分页硬件

Nachos内存管理包括以下几个模块：

- Nachos中可执行文件格式：`noff.h`
- 装入noff用户程序到逻辑地址空间：`userprog/AddrSpace`
- Nachos内存页表结构：`machine/translate`
- 将逻辑地址变换为物理地址的MMU硬件：Nachos 模拟带有 TLB 的页式内存管理。MMU 由函数 `Machine` 类的 `Translate()` 模拟。定义在 `translate.cc` 中
- 执行可执行文件的MIPS模拟器：`machine/Machine`

1.10.1 可执行文件格式

在 Nachos 系统中用户的 MIPS 的可执行二进制程序是由以下两个步骤产生的：

- 用 gcc MIPS 交叉编译程序编译用户的.c 源代码文件，产生通常在 Uinx 系统中可执行的 `Coff` 格式文件。
- 使用 Coff2noff 程序转换以上 `Coff` 文件，在../bin 目录中产生在 Nachos 系统中可执行的 `Noff` 格式文件。

`Noff` 格式是 Nachos 系统规定的可执行文件格式。它类似于 `Coff` 格式。 `Noff` 格式在 `bin/noff.h` 文件中定义如下：

```
#define NOFFMAGIC    0xbadfad     /* magic number denoting Nachos object code file*/

typedef struct segment {
  int virtualAddr;        /* 段在虚拟地址空间中的位置 */
  int inFileAddr;        /* 该段在Noff文件中实际开始的偏移量 */
  int size;                /* size of segment */
} Segment;

typedef struct noffHeader {
   int noffMagic;        /* should be NOFFMAGIC */
   Segment code;        /* executable code segment */
   Segment initData;    /* initialized data segment */
   Segment uninitData;    /* uninitialized data segment -- should be zero'ed before
use */
} NoffHeader;
```

`Noff`格式头主要定义了 Nachos 系统中用户可执行文件的段表格式以及段的类型。Nachos 系统将根据这一定义将`Noff`格式的文件加载到 Nachos 的用户内存中解释执行。

> 用于测试的用户 C 程序源代码文件都在目录 test/中。为了产生其中 C 程序的`Noff`格式文件，如果已安装好 gcc MIPS 交叉编译程序，就 可以在 test/目录中执行 make。test/目录中的 Makefile 文件声明将用 coff2noff 和 coff2float 程序转换`Coff`格式的 MIPS 可执行文件到`Noff`和`flat`格式的可执行文件
>
> make 完成后 test/目录中将产生与.c文件相对应的一系列`.noff`和`.flat`文件：
>
> halt.noff halt.flat manult.noff manult.flat ...

1.10.2 MIPS模拟机

由 Nachos 支持的用户程序是一个 MIPS 指令的二进制可执行文件（`.noff`）。为了能使用户程序在非 MIPS 机上的 Nachos 内核上工作，Nachos 发行版都带有一个 MIPS 模拟器。一个 MIPS 机的模拟器用于在 Nachos 中运行用户程序。它是通过定义在`machine/machine.h`第 107-190 行的一个`Machine`类实现的。

MIPS 机的主要部件定义在`Machine.h`的public属性中：

- 用户寄存器组：`Int registers[NumTotlRegs]`
- 主存：`char *mainMemory`
- 当前用户进程的**虚拟**页表：`TranslationEntry *pageTable`

  值得注意的是，**Nachos任意时刻只有一个页表**，当一个用户进程获得CPU时会调用`AddrSpace->RestoreState()`函数恢复现场：

```
void AddrSpace::RestoreState()
{
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
}
```

- 快表： `TranslationEntry *tlb`

机器的操作由以下各种函数模拟， `Run` 与 `OneInstruction` 函数的实现在 `machine/mipssim.cc` 中， `ReadMem` 与 `WriteMem` 函数的实现在 `machine/translate.cc` ：

```
#define NumPhysPages    32
#define MemorySize     (NumPhysPages * PageSize)
#define TLBSize        4        // if there is a TLB, make it small
class Machine {
  public:
    Machine(bool debug);    // Initialize the simulation of the hardware
                            // for running user programs
    ~Machine();                // De-allocate the data structures

    //指令的解释执行：
    void Run();// 运行用户程序
    void OneInstruction(Instruction *instr);// 执行一条 MIPS 指令.

    // registers寄存器读写
    int ReadRegister(int num);    // 从编号为num的CPU寄存器中读出数据
    void WriteRegister(int num, int value);// 将value单元中的数据写入编号为num的CPU寄存器中

    // mainMemory内存读写：
    bool ReadMem(int addr, int size, int* value);//从addr单元中读长度为size字节的数据到value指向的单元中
    bool WriteMem(int addr, int size, int value);//向addr单元中写入value单元中长度为size字节的数据

    //地址变换:将单元长度为size的逻辑地址virAddr变换为物理地址physAddr《该单元是否可写由writing决定。
    ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);

    //异常处理：处理一个类型为which的异常
    void RaiseException(ExceptionType which, int badVAddr);
    void DelayedLoad(int nextReg, int nextVal); // Do a pending delayed load (modifying a reg)

    void Debugger();        // invoke the user program debugger
    void DumpState();        // print the user CPU and memory state

    char *mainMemory;             // 主存：存储执行的用户程序的代码段数据段
    int registers[NumTotalRegs]; // 存储执行用户程序寄存器
    TranslationEntry *tlb;    // 快表
    TranslationEntry *pageTable;// 当前执行用户进程的虚拟页表
    unsigned int pageTableSize;// 虚拟页表长度

  private:
    bool singleStep;        // drop back into the debugger after each simulated instruction
    int runUntilTime;        // drop back into the debugger when simulated time reaches this value
};
```

- `Run()` 函数负责设置机器进入用户态，并且在一个无限循环中模拟取CPU指令和执行CPU指令的过程。指令 `Instruction` 类定义在machine/Machine中

```
void Machine::Run()
{
    Instruction *instr = new Instruction;  // 初始化一个指令

    if(DebugIsEnabled('m'))
        printf("Starting thread \"%s\" at time %d\n",
           currentThread->getName(), stats->totalTicks);
    interrupt->setStatus(UserMode);// 设置系统为用户态，执行用户程序
    for (;;) {
        OneInstruction(instr);// 取下一条指令和执行一条指令
        interrupt->OneTick();// 时钟前进
        if (singleStep && (runUntilTime <= stats->totalTicks))
          Debugger();
    }
}
```

- `OneInstuction()` 模拟取下一条指令和执行一条指令的周期

```
void Machine::OneInstruction(Instruction *instr)
{
       int raw;
    int nextLoadReg = 0;
    int nextLoadValue = 0;     // record delayed load operation, to apply in the
future

    // 取出一条指令
    if (!machine->ReadMem(registers[PCReg], 4, &raw))
         return; // exception occurred
    instr->value = raw;
    instr->Decode();//指令解码

    //....

    // 计算下一条指令的PC指针，但是先不切换
    int pcAfter = registers[NextPCReg] + 4;
    int sum, diff, tmp, value;
    unsigned int rs, rt, imm;

    // 具体的指令执行过程，省略
    switch (instr->opCode) {
        case OP_ADD:
            sum = registers[instr->rs] + registers[instr->rt];
            if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
            ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
                RaiseException(OverflowException, 0);
                return;
            }
            registers[instr->rd] = sum;
            break;
        //...
        case OP_RES:
        case OP_UNIMP:
            RaiseException(IllegalInstrException, 0);
            return;
        default:
        ASSERT(FALSE);
    }

    // 到达这里指令执行成功

    // Do any delayed load operation
    DelayedLoad(nextLoadReg, nextLoadValue);
    // Advance program counters.
    registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we
    // are jumping into lala-land
    registers[PCReg] = registers[NextPCReg];
    registers[NextPCReg] = pcAfter;//指令计数器被向前推进，准备执行下一条指令。
}
```

1.10.3 内存页表结构

Nachos 系统中的**页表**结构是由machine/translate.h中的`TranslationEntry`类定义的。页表是由若干个表项（TranslationEntry）组成的。每个表项记录**程序逻辑页到内存实页的映射关系**，和实页的使用状态、访问权限信息。类`TranslationEntry`描述了表项的结构：

```
class TranslationEntry {
  public:
    int virtualPage;      // 逻辑页号
    int physicalPage;      // 内存物理页号
    bool valid;          // True表示该页有效
    bool readOnly;        // 对应页的访问属性，TRUE 示只读，否则为读写
    bool use;            // 该Entry是否被使用过，每次访问后置为TRUE
    bool dirty;          // 对应的物理页使用情况，TRUE表示被写过
};
```

use 是引用位，dirty 是改写位，利用这两位信息可以构造虚拟内存。

上述的这个数据结构用于管理 “虚页->实页” 的转换，用于管理存储用户程序的物理内存。而且该数据结构在 Nachos 有两个用途，一个是作为页表项使用，另一个是用作 TLB 项。

1.10.4 用户程序的地址空间

创建一个逻辑地址空间装入正在执行的noff用户程序，其中地址空间由类`AddrSpace`定义的。这个类的定义在`userprog/addrspace.h`文件中。

```
#define UserStackSize		1024 	// increase this as necessary!

class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);    // Create an address space,
                                        // initializing it with the program
                                        // stored in the file "executable"
    ~AddrSpace();			// De-allocate an address space

    void InitRegisters();		// Initialize user-level CPU registers,
                                        // before jumping to user code

    void SaveState();			// Save/restore address space-specific
    void RestoreState();		// info on a context switch

  private:
    TranslationEntry *pageTable;	// Assume linear page table translation for now!
    unsigned int numPages;		// Number of pages in the virtual address space
};
```

可以看到 AddrSpace 中包括了初始化和管理用户进程空间的一些方法：

- `AddrSpace()` 根据打开的用户可执行文件构造用户内存空间。
- `InitRegisters()` 初始化用户 CPU 寄存器。
- `SaveState()` 保存用户空间现场。
- `RestoreState()` 恢复用户空间现场。

一个指向 `TranslationEntry` 类的指针 `pageTable` 给出了页表数组的起始地址，**这个页表数组存储了整个用户进程的所有代码的页表（可以理解为整个用户程序的虚拟页表）**。 而整数 `numPages` 给出了页表数。

一个用户程序想要运行，最重要的部分就是将用户程序的代码段与初始化数据段加载到内存中

用户进程的地址空间在 AddrSpace 构造函数中初始化，并且装入到实际内存中(char *mainMemory)：

```
AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);// 可执行文件读出noff Header信
息
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
            + UserStackSize;    // 计算地址空间长度

    numPages = divRoundUp(size, PageSize);//计算出地址空间需要多少页
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);        // check we're not trying
                        // to run anything too big --
                        // at least until we have
                        // virtual memory

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
                    numPages, size);
// first, set up the translation
    pageTable = new TranslationEntry[numPages];// 初始化页表数组
    for (i = 0; i < numPages; i++) {// 对每个页表初始化
        pageTable[i].virtualPage = i;    //目前为止，物理地址等于虚拟逻辑地址
        pageTable[i].physicalPage = i;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
                                        // a separate page, we could set its
                                        // pages to be read-only
    }

// zero out the entire address space, to zero the unitialized data segment
// and the stack segment
    bzero(machine->mainMemory, size);//清理 machine->mainMemory 模拟的物理内存。

//代码段和初始化的数据段从可执行文件中被装载到分配给它的物理地址空间中。非初始化的数据段不必
装入，因为它的对应段已包含了初始化的0值。

    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
            noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
```

```
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
            noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

}
```

1.10.5 从逻辑地址到物理地址

在程序执行时**需要查询页表将逻辑地址需要需要变换为物理空间**。在真实的计算机中，这一工作是由MMU硬件完成的。当变换发生错误时MMU会自动发出各种异常中断。

Nachos 模拟页式内存管理。MMU由`machine`类的函数`Translate()`模拟。该函数定义在`translate.cc`中，两个函数`ReadMem()`和`WriteMem()`在访问物理内存之前都要调用函数`Translate()`将要访问的逻辑地址变换为物理地址。并检查地址是否对齐以及其它错误。 如果没有错误，则在页表项中设置use、dirty位初值，并将转换后的物理地址保存在`*physAddr`变量中。如果变换发生错误函数`Translate()`会返回一个异常。

> 当前的 MIPS 模拟器允许页表和快表两种地址变换机制。这个函数假设我们可用**页式(machine->pageTable)和快表(machine->TLB)两种变换方式**，但当前的实现假设我们要么使用页表要么使用快表，但不 能两者同时都用。为了模拟带有快表的真正的页式变换，这个函数需要改进.。可以在`Machine`类中看到，`Machine`类已经提供了一个指向当前用户进程的指针和一个指向系统`TLB`的指针。

参数列表：

- `virtAddr`：虚拟地址
- `physAddr`：存储转换结果
- `size`：写或读的字节数
- `writing`：可写标记

```
ExceptionType Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    int i;
    unsigned int vpn, offset;
    TranslationEntry *entry;// 页表项
    unsigned int pageFrame;

    DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" : "read");

// 检查对齐错误，即如果size = 4，则地址为4的倍数，即低两位为0
// 若size = 2，则地址为2的倍数，即最低位为0
    if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){
        DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
        return AddressErrorException;
    }
// 要么使用页表要么使用快表
    ASSERT(tlb == NULL || pageTable == NULL);
    ASSERT(tlb != NULL || pageTable != NULL);
// 通过虚拟地址计算虚拟页编号以及页内偏移量
    vpn = (unsigned) virtAddr / PageSize;// 虚拟页编号
    offset = (unsigned) virtAddr % PageSize;// 页内偏移量

    if (tlb == NULL) {// TLB为空，则使用页表
        if (vpn >= pageTableSize) { // vpn大于页表大小，即返回地址错误
            DEBUG('a', "virtual page # %d too large for page table size %d!\n",
                virtAddr, pageTableSize);
            return AddressErrorException;
        } else if (!pageTable[vpn].valid) {// vpn所在页不可用，即返回缺页错误
            DEBUG('a', "virtual page # %d too large for page table size %d!\n",
                virtAddr, pageTableSize);
            return PageFaultException;
        }
        entry = &pageTable[vpn];// 获得页表中该虚拟地址对应页表项
    } else {// TLB不为空，则使用TLB
        for (entry = NULL, i = 0; i < TLBSize; i++)// 遍历TLB搜索
            if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {
                entry = &tlb[i];            // 找到虚拟地址所在页表项!
                break;
        }
        if (entry == NULL) {                // 在TLB中没有找到，可能在主存中，返回缺页错
误
                DEBUG('a', "*** no valid TLB entry found for this virtual page!\n");
                return PageFaultException;
        }
    }

    if (entry->readOnly && writing) {    // 想要向只读页写数据，返回只读错误
        DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);
        return ReadOnlyException;
    }
```

```
    pageFrame = entry->physicalPage;// 由页表项可得到物理页框号

    if (pageFrame >= NumPhysPages) { // 物理页框号过大，返回越界错误
        DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
        return BusErrorException;
    }
    entry->use = TRUE;         // 设置该页表项正在使用
    if (writing)// 设置该页表项被修改了，即dirty位为true
        entry->dirty = TRUE;
    *physAddr = pageFrame * PageSize + offset;// 得到物理地址
    ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));// 物理地址不可越界
    DEBUG('a', "phys addr = 0x%x\n", *physAddr);
    return NoException;
}
```

1.10.6 内存管理总结

Nachos MIPS模拟机是由`machine`类实现的，类成员变量`char *mainMemory`定义了操作系统的内存。

Nachos的内存系统采用的是**分页**的分配方式，在硬件模拟machine类中有一个`TranslationEntry *pageTable`数组表示页表，还有一个`TranslationEntry *tlb`数组表示快表(高速缓存)。

`TranslationEntry`定义了页表项的内部结构，主要包括虚拟逻辑地址到实际`mainMemory`数组的下标的映射。

在程序执行时需要**查询页表**将虚拟逻辑地址需要需要变换为物理地址。由`machine`类的函数`Translate()`模拟。

用户可执行文件格式是`.noff`格式，Noff文件头中定义了代码段、初始化数据段、未初始化数据段的虚拟内存地址与实际文件中的偏移量。

如何装入用户可执行程序到内存中？需要使用`AddSpace`类保存所有可执行文件的页表项以及页表数目，并且在初始化`AddSpace()`类时将可执行文件的内容写入到`mainMemory`中。

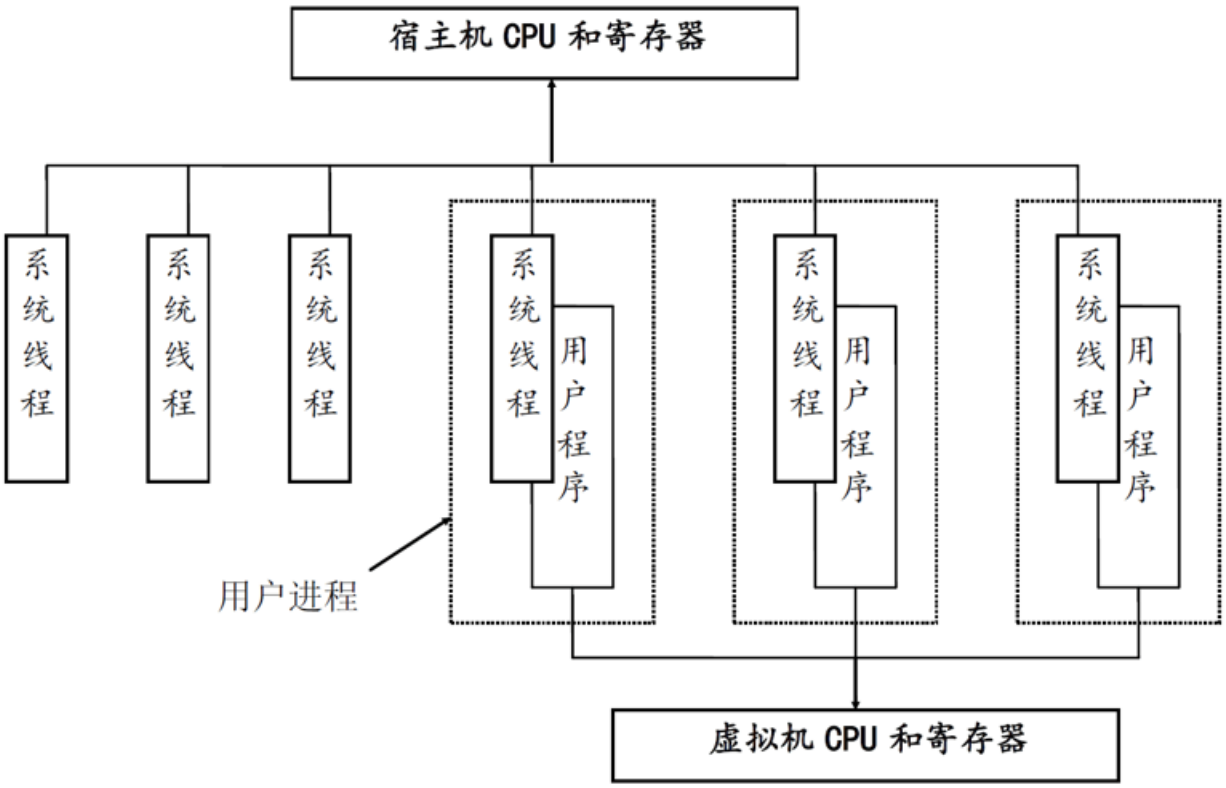
1.11 Nachos用户程序

1.10节讨论了用户程序的可执行化，以及用户程序如何装入内存中的，重点关注如何实现**从内核线程到用户进程的转化。**

Nachos 用户进程是构建在 Nachos 的线程之上的。回顾一下 Thread 类中有关用户进程的定义：

```
#ifdef USER_PROGRAM
    int userRegisters[NumTotalRegs];    // user-level CPU register state

  public:
    void SaveUserState();        // save user-level register state
    void RestoreUserState();    // restore user-level register state

    AddrSpace *space;            // User code this thread is running.
#endif
```

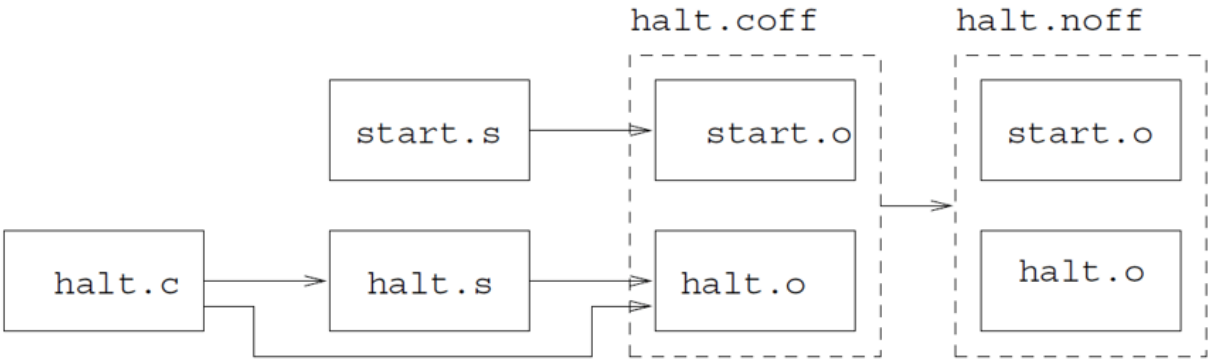
显示了内核线程有**用于保存用户寄存器的数组和一个内核线程的用户地址空间指针。**重新理解系统线程与用户线程的结构如图：



当你在 userprog 目录中编译时，其中 Makefile 文件定义了 USER_PROGRAM 标 志。此时以上 124-132 行的代码将会被编译进内核中去。 定义在../userprog/progtest.cc 文件中的函数 `StartProcess()` 说明了**如何由一个内核线程构造并启动一个用户进程**的过程：

```
//该参数是在 Nachos 系统启动时由命令行参数传入的 Noff 格式的用户可执行文件名字符串。
void StartProcess(char *filename)
{
    //文件系统调用函数Open根据可执行文件名打开要装入的文件
    OpenFile *executable = fileSystem->Open(filename);

    AddrSpace *space;

    if (executable == NULL) {
        printf("Unable to open file %s\n", filename);
        return;
    }
    //使用这个打开的文件建立和初始化好一个用户进程空间并且返回一个指向该进程空间的指针。
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;            // close file

    space->InitRegisters();        // set the initial register values
    space->RestoreState();        // 保存页表到machine中

    // 跳转到用户进程
    machine->Run();            // jump to the user progam
    ASSERT(FALSE);            // machine->Run never returns;
                            // the address space exits
                            // by doing the syscall "exit"
}
```

当调用`machine->Run()`后**当前线程就变成了运行在 MIPS 模拟机上的用户进程**，当然之后的机器工作状态就由系统的核心态转变为用户态。而当用户进程结束时不会返回到Run()函数处，而是依赖于**系统调用exit()**切换回系统态。

> **Nachos用户程序的编译：**
>
> 实际这些 C 语言编写的用户程序在由`gcc MIPS`交叉编译后都在前面连接上一个由 MIPS 汇编程序 start.s 生成的叫 start. o 的目标模块。实际上 start 是用户程序真正的启动入口，由它来调用 C 程序的 main 函数。所以不要求用户编程时一定要把 main 函数作为第一个函数。
>
> 例如 C 程序 halt.c 被编译为 halt.o，同时 start.s 也被汇编为 start.o。之后两个目标模块被连接成可执行的 Coff 格式的可执行文件，最后这个 Coff 文件又被转换为 Nof f 格式的 Nachos 可执行文件。
>
> 
>
>
> Construction of Nachos user program

1.12 Nachos系统调用

系统调用是用户程序和操作系统内核的接口。用户程序**从系统调用函数取得系统服务。**

- 当 CPU 控制从用户程序切换到系统态时，CPU 的工作方式由用户态改变为系统态。
- 而当内核完成系统调用功能时，CPU 工作状态又从系统态改变回用户态并且将控制再次返回给用户程序。

两种不同的 CPU 工作状态提供了操作系统基本的保护方式。

所有 Nachos 系统调用的接口原型都定义在文件`userprog/syscall.h`中。当编译用户程序时编译器会括入这个文件并取得这些系统调用接口原型的信息

```
/* syscalls.h
 *     Nachos system call interface.  These are Nachos kernel operations
 *     that can be invoked from user programs, by trapping to the kernel
 *    via the "syscall" instruction.
 *
 *    This file is included by user programs and by the Nachos kernel.
 *
 * Copyright (c) 1992-1993 The Regents of the University of California.
 * All rights reserved.  See copyright.h for copyright notice and limitation
 * of liability and disclaimer of warranty provisions.
 */

#ifndef SYSCALLS_H
#define SYSCALLS_H

#include "copyright.h"

/* system call codes -- used by the stubs to tell the kernel which system call
 * is being asked for
 */
#define SC_Halt        0
#define SC_Exit        1
#define SC_Exec        2
#define SC_Join        3
#define SC_Create    4
#define SC_Open        5
#define SC_Read        6
#define SC_Write    7
#define SC_Close    8
#define SC_Fork        9
#define SC_Yield    10

#ifndef IN_ASM

/* The system call interface.  These are the operations the Nachos
 * kernel needs to support, to be able to run user programs.
 *
 * Each of these is invoked by a user program by simply calling the
 * procedure; an assembly language stub stuffs the system call code
 * into a register, and traps to the kernel.  The kernel procedures
 * are then invoked in the Nachos kernel, after appropriate error checking,
 * from the system call entry point in exception.cc.
 */

/* Stop Nachos, and print out performance stats */
void Halt();

/* Address space control operations: Exit, Exec, and Join */

/* This user program is done (status = 0 means exited normally). */
```

```
void Exit(int status);

/* A unique identifier for an executing user program (address space) */
typedef int SpaceId;

/* Run the executable, stored in the Nachos file "name", and return the
 * address space identifier
 */
SpaceId Exec(char *name);

/* Only return once the the user program "id" has finished.
 * Return the exit status.
 */
int Join(SpaceId id);

/* File system operations: Create, Open, Read, Write, Close
 * These functions are patterned after UNIX -- files represent
 * both files *and* hardware I/O devices.
 *
 * If this assignment is done before doing the file system assignment,
 * note that the Nachos file system has a stub implementation, which
 * will work for the purposes of testing out these routines.
 */

/* A unique identifier for an open Nachos file. */
typedef int OpenFileId;

/* when an address space starts up, it has two open files, representing
 * keyboard input and display output (in UNIX terms, stdin and stdout).
 * Read and Write can be used directly on these, without first opening
 * the console device.
 */

#define ConsoleInput    0
#define ConsoleOutput    1

/* Create a Nachos file, with "name" */
void Create(char *name);

/* Open the Nachos file "name", and return an "OpenFileId" that can
 * be used to read and write to the file.
 */
OpenFileId Open(char *name);

/* Write "size" bytes from "buffer" to the open file. */
void Write(char *buffer, int size, OpenFileId id);

/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
```

```
 * you should always wait until you can return at least one character).
 */
int Read(char *buffer, int size, OpenFileId id);

/* Close the file, we're done reading and writing to it. */
void Close(OpenFileId id);

/* User-level thread operations: Fork and Yield.  To allow multiple
 * threads to run within a user program.
 */

/* Fork a thread to run a procedure ("func") in the *same* address space
 * as the current thread.
 */
void Fork(void (*func)());

/* Yield the CPU to another runnable thread, whether in this address space
 * or not.
 */
void Yield();

#endif /* IN_ASM */

#endif /* SYSCALL_H */
```

对应的系统调用的汇编语言存根在 test/start.s 文件中的 45-131 行。**如果你要添加你自己的系统调用，就应当首先在 `syscall.h` 和 `start.s` 中声明你的系统调用原型和存根**

当一个系统调用由一个用户进程发出时，由汇编语言编写的对应于存根的程序就被执行。然后，这个存根程序会由执行一个系统调用指令而引发一个**异常或自陷**处理该系统调用。 在 `start.s` 中的这些系统调用的接口程序代码都是一样的。即：

- 将对应的系统调用的编码送$2寄存器
- 执行系统调用指令 SYSCALL
- 返回到用户程序

```
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j    $31
    .end Halt

    .globl Exit
    .ent    Exit
```

Nachos中的异常与自陷

模拟 MIPS 计算机的**异常和自陷管理**的是 Machine 类中的函数 `RaiseException(ExceptionType which, int badVAddr)`。其中的第一个参数 which 是 一个 `ExceptionType` 枚举类型的变量。 `ExceptionType` 类型的定义也在 machine/machine.h 文件中：

系统调用是`SyscallException`类型，MIPS 计算机的"SYSCALL"指令在 Nachos 中是由 machine/mipssim.cc 中 534-536 行上的通过触发系统调用异常模拟的：

```
case OP_SYSCALL:
    RaiseException(SyscallException, 0);
    return;
```

函 数 `RaiseException(ExceptionType which, int badVAddr)`的代码在 machine/machine.cc 文件中：

```
void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);

//  ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);            // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);        // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

这个函数模拟硬件的动作，切换到系统态并且在异常处理完成后返回到用户态。 **9行上的`ExceptionHandler(which)`函数调用模拟硬件的动作发一个异常中断到对应的异常处理程序。**这个函数定在 userprog/execution.cc 中：

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);// 寄存器$2存储着系统调用类型

    if ((which == SyscallException) && (type == SC_Halt)) {// 实现系统调用Halt
        DEBUG('a', "Shutdown, initiated by user program.\n");
           interrupt->Halt();
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

对于系统调用 `Halt` 的异常处理只是简单的模拟了 `Interrupt` 类指向的中断函数 `Halt()`。

1.13 Nachos 虚拟内存

Nachos默认的用户进程分配策略是：用户进程需要整个地址空间都进入物理内存后才能开始运行。这种进程全部进入内存后再运行的策略是很受限制和浪费内存空间的。实际上多数程序的工作方式是符合"局部性原理"的，即当前要执行的程序的和当前要访问的数据 往往是集中在某些内存页面上，而当前执行不到程序和访问不到的数据完全可以先放在第二存储器中，等到用到时再从第二存储器中装入物理内存。

虚拟内存技术就是解决程序部分装入物理内存也能执行的技术。由于允许进程部分驻留物理内存，所以一个进程的逻辑空间可以远远大于分配给进程工作的物理空间。由于Nachos采用分页的内存管理机制，所以我们可以采用请求分页技术实现虚拟内存：其关键的技术是，当请求的页不在物理内存时使用缺页异常或自陷进入内核，由内核缺页异常处理程序从外存将该页装入一空闲内存帧中，如果无空闲帧，将选择已在物理内存中的一页将其置换。

Nachos引发缺页异常的地方在translate.cc的 `Translate()` 产生的，这个函数是由函数 `ReadMem` 和 `WriteMem` 调用，**注意函数 `ReadMem` 在完成缺页异常处理之后将返回一个 FALSE(13 行)，这一点很重要，它将使调用 `ReadMem` 的指令重新执行。**：

```
bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);

    exception = Translate(addr, &physicalAddress, size, FALSE);// 可能产生缺页异常
    if (exception != NoException) {
        machine->RaiseException(exception, addr);
        return FALSE;
    }
    ...
}
```

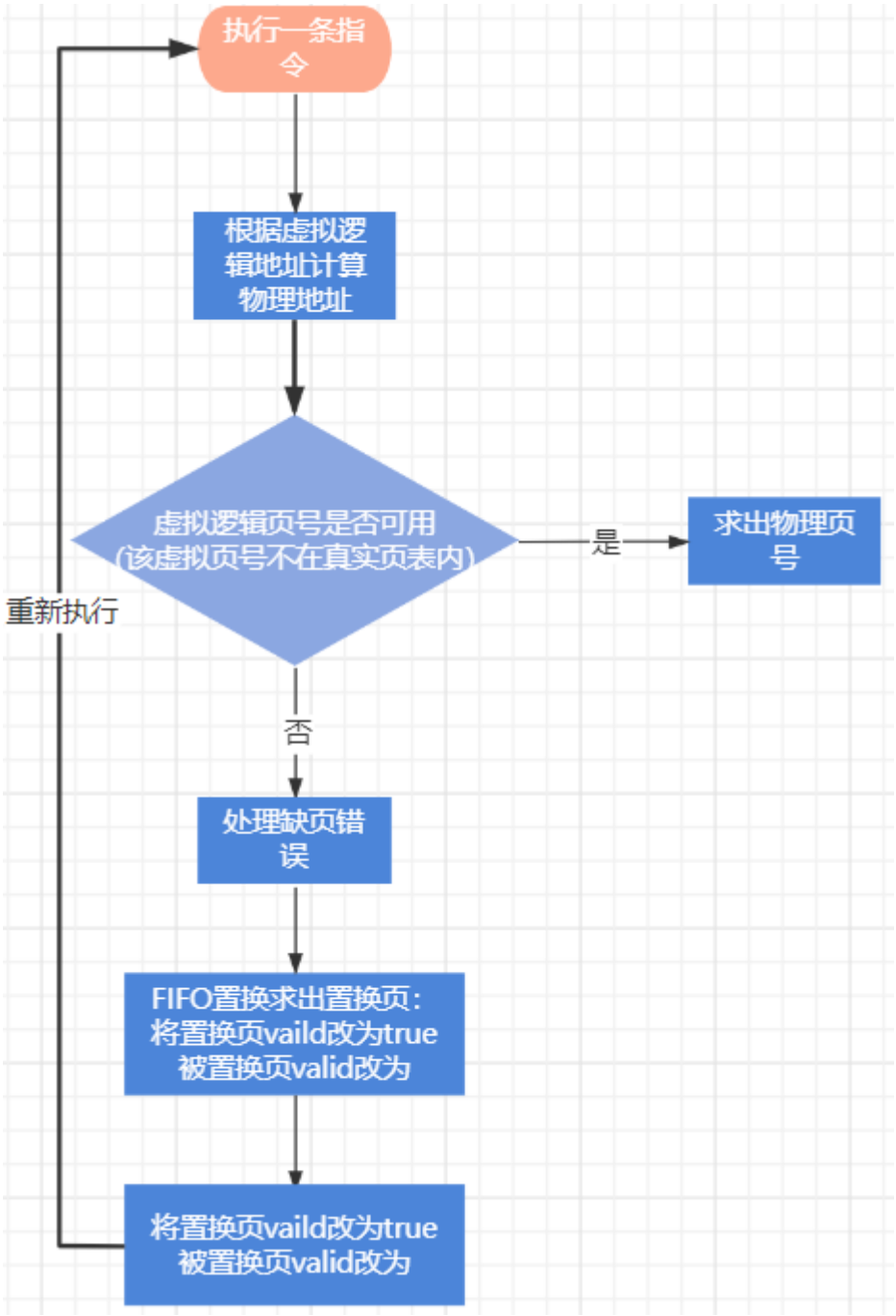
参数 addr 是一条指令发出的要寻找的**逻辑地址**。这个地址通过调用函数 `translate()` 被转换为物理地址。如果发生缺页异常，`Machine::RaiseException()` 会将发生异常的虚拟地址存入 Register[BadVAddrReg]，因此我们可以从 Register[BadVAddrReg] 中获取发生异常的虚拟地址。最后调用 exception.cc 里的 `ExceptionHandler()` 函数对异常进行处理，、

```
void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);

//  ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;// 将出错的虚拟地址存入寄存器中
    DelayedLoad(0, 0);       // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);      // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

因此整个缺页错误的编程入口应该在 exception.cc 里的 `ExceptionHandler()` 函数对异常进行处理

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if (which == SyscallException) {
        switch (type) {
            case SC_Halt:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            }
            case SC_Exec:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                printf("Execute System Call of Exec()\n");
                interrupt->Exec();
                AdvancePC();// 程序计数器向前推进
                break;
            }
            default:{
                printf("Unexpected user mode exception %d %d\n", which, type);
                ASSERT(FALSE);
            }
        }
    }else if(which == PageFaultException){
        // 这里执行页错误。

    }
    else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

指的注意的是，Nachos没有严格限制用户进程的虚拟页表的大小，**只要用户进程的初始化代码页数+自定义可自由调度页数<内存总页数(32)**，理论上就可以执行。实际过程中因为系统内只有一个虚拟页表，**因此真实可用的页表用页表项的valid=TRUE位表示。**整个过程如下：



Lab2 具有优先级的线程调度

2.1 实验内容

1. 熟悉Nachos原有的线程调度策略；
2. 设计并实现具有**静态优先级**的非抢占式线程调度策略。

提示：List类中已有的SortedInsert方法可加以利用。

2.2 实验思路

我们需要给所有线程维护一个优先级属性，每次调用`scheduler->FindNextToRun()`选出队列中优先级最大的线程出队列。因此由两个思路：

- 维护一个优先队列，每次出队列时选择队首元素。
- 按照先来先进队列顺序，但每次出队列时遍历整个队列。

为了利用队列中原有的`SortedInsert()`方法，同时又因为维护优先队列的时间复杂度由于遍历队列。我们采用方案一。

2.3 实验代码

2.3.1 Thread

给每个线程添加一个`int priority`属性，编写对应的`set`与`get`方法，同时重载一个构造函数`Thread(char* name,int threadPriority)`：

- `Thread.h`：

```
private:
    int priority; // 进程的优先级
public:
    void SetPriority(int priority){
        //优先级-20~20
        ASSERT(priority>=-20 && priority<=20);
        priority=prior;
    }
    int getPriority(){return priority;}
```

- `Thread.cc`：重载的构造函数

```
Thread::Thread(const char* threadName,int threadPriority)
{
    name = (char*)threadName;
    priority = threadPriority;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
```

2.3.2 List.cc

修改`List.cc`中`SortedInsert()`方法中判断权重大小部分的符号，从`sortKey < first->key、sortKey < ptr->next->key`改为：`sortKey > first->key、sortKey > ptr->next->key`

```
// 改为按照从大到小的优先级插入
void List::SortedInsert(void *item, int sortKey)
{
    ListElement *element = new ListElement(item, sortKey);// 创建一个队列元素
    ListElement *ptr;        // keep track

    if (IsEmpty()) {    // if list is empty, put
        first = element;
        last = element;
    } else if (sortKey > first->key) {// 将插入线程放在队列首
        // item goes on front of list
        element->next = first;
        first = element;
    } else {        // look for first elt in list bigger than item
        for (ptr = first; ptr->next != NULL; ptr = ptr->next) {// 迭代寻找合适的位置
            if (sortKey > ptr->next->key) {
        element->next = ptr->next;
            ptr->next = element;
        return;
        }
    }
        last->next = element;        // item goes at end of list
        last = element;
    }
}
```

2.3.3 scheduler.cc

修改`scheduler.cc`中的`ReadyToRun()`与`FindNextToRun ()`方法，使得线程进入与弹出等待队列按照优先级。

```
void Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);
    readyList->SortedInsert((void *)thread,thread->getPriority());// 按照优先级插入，维
护一个优先队列
}

Thread* Scheduler::FindNextToRun ()
{
    return (Thread *)readyList->Remove();// 取出优先队列队首的元素
}
```

2.3.4 threadtest.cc

修改测试类测试优先级调度

```
//传入参数为线程编号
void SimpleThread(_int which){
    int num;

    for (num = 0; num < 5; num++) {
    printf("*** thread %d looped %d times\n", (int) which, num);
        currentThread->Yield();
    }
}

void ThreadTest(){
    DEBUG('t', "Entering SimpleTest");

    currentThread->SetPriority(1);// 给当前运行的线程赋优先级
    printf("name=%s,priority=%d\n",currentThread->getName(),currentThread-
>getPriority());

    Thread* t1 = new Thread("Thread1", 2);
    printf("name=%s,priority=%d\n",t1->getName(),t1->getPriority());
    t1->Fork(SimpleThread, 1);

    Thread* t2 = new Thread("Thread2", 3);
    printf("name=%s,priority=%d\n",t2->getName(),t2->getPriority());
    t2->Fork(SimpleThread, 2);

    Thread* t3 = new Thread("Thread3", 4);
    printf("name=%s,priority=%d\n",t3->getName(),t3->getPriority());
    t3->Fork(SimpleThread, 3);

    SimpleThread(0);
}
```

2.4 实验结果

赋值修改的文件到`lab2`文件夹中，修改`makefile.local`的`INCPATH`为`INCPATH += -I- -I../lab2 -I../threads -I../machine`。切换到`lab2`文件夹下，依次执行命令编译：

```
make clean
make
./nachos
```

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab2$ ./nachos
name=main,threadId=0,priority=1
name=Thread1,priority=2
name=Thread2,priority=3
name=Thread3,priority=4
*** thread 0 looped 0 times
*** thread 3 looped 0 times
*** thread 2 looped 0 times
*** thread 3 looped 1 times
*** thread 2 looped 1 times
*** thread 3 looped 2 times
*** thread 2 looped 2 times
*** thread 3 looped 3 times
*** thread 2 looped 3 times
*** thread 3 looped 4 times
*** thread 2 looped 4 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

可以看到我们额外创建了三个线程，分别赋予其优先级，nachos操作系统默认执行main线程，`SimpleThread(0)`方法被执行，main线程调用`Yield()`方法，main线程重新回到等待队列中，`scheduler`对象执行`FindNextToRun()`方法寻找优先级最高的下一个线程（Thread3），`scheduler`对象调用`run(Thread3)`方法切换上下文，执行Thread3中的`SimpleThread(1)`，重复此过程....

lab3 使用信号量解决生产者/消费者同步问题

3.1 实验内容

使用操作系统信号量机制，编写程序解决生产者/消费者同步问题。多个生产者和消费者线程访问在共享内存中的环形缓冲。生产者生产产品并将它放入环形缓冲，同时消费者从缓冲中取出产品并消费。当缓冲区满时生产者阻塞并且当缓冲区有空时生产者又重新 工作。类似的，消费者当缓冲区空时阻塞并且当缓冲区有产品时又重新工作。显然， 生产者和消费者需要一种同步机制以协调它们的工作。

包括:

1. 理解Nachos的信号量是如何实现的;
2. 生产者/消费者问题是如何用信号量实现的;
3. 在Nachos中是如何创建并发线程的;

4. 在Nachos下是如何测试和debug的。

3.2 实验思路

lab3中的文件包括`mian.cc，prodcons++.cc，ring.cc 和 ring.h`。 文件`ring.cc`和`ring.h`定义和实现了一个为生产者和消费者使用的环形缓冲类`ring`。 `ring` 类的分析见1.8。这两个文件已经完成了，不需要我们作任何改动。在这个目录中的 main.cc 是../threads 目录中 main.cc 的改进版，我们也不必改动了。 在新的 main.cc 中调用了函数 `ProdCons()`而不是函数 `ThreadTest()`。

函数`ProdCons()`定义在文件 prodcons++.cc 中。假设这个文件包含着建立两个生产者和两个消费者线程以及实现两个生产者消费者问题算法的代码。不过，这个文件并不完整，我们的实验任务就是完成`prodcons.cc`，构造一个能工作的生产者/消费者问题算法。

重点分析`prodcons.cc`，源代码如下：

```
// prodcons++.cc
//    C++ version of producer and consumer problem using a ring buffer.
//
//    Create N_PROD producer threads and N_CONS consumer thread.
//    Producer and consumer threads are communicating via a shared
//      ring buffer object. The operations on the shared ring buffer
//      are synchronized with semaphores.
//
//
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "copyright.h"
#include "system.h"

#include "synch.h"
#include "ring.h"

#define BUFF_SIZE 3  // the size of the round buffer
#define N_PROD    2  // the number of producers
#define N_CONS    2  // the number of consumers
#define N_MESSG   5  // the number of messages produced by each producer
#define MAX_NAME  16 // the maximum lengh of a name

#define MAXLEN    48
#define LINELEN    24


Thread *producers[N_PROD]; //array of pointers to the producer
Thread *consumers[N_CONS];  // and consumer threads;

char prod_names[N_PROD][MAX_NAME];  //array of charater string for prod names
char cons_names[N_CONS][MAX_NAME];  //array of charater string for cons names

Semaphore *nempty, *nfull; //two semaphores for empty and full slots
Semaphore *mutex;          //semaphore for the mutual exclusion

Ring *ring;

void Producer(_int which){}
void Consumer(_int which){}
void ProdCons(){}
```

该源文件主要定义了以下宏与全局变量：

宏或全局变量	含义
BUFF_SIZE=3	环形缓冲区大小为3
N_PROD=2	生产者线程数量
N_CONS=2	消费者线程数量
N_MESSG 5	每个生产者线程产生的消息数
MAX_NAME=16	线程名称最大长度
MAXLEN、LINELEN	与最终输出的文件有关
Thread *producers[N_PROD]	生产者线程数组
Thread *consumers[N_CONS]	消费者线程数组
prod_names	生产者名称数组
cons_names	消费者名称数组
Semaphore *nempty	缓冲区已生产空间信号量
Semaphore *nfull	缓冲区未生产空间信号量
Semaphore *mutex	缓冲区互斥锁
Ring *ring	环形缓冲区

该源文件还定义了三个方法：

- `Producer(_int which)`：生产者线程调用的函数

```
void
Producer(_int which)
{
    int num;
    slot *message = new slot(0,0);

//  This loop is to generate N_MESSG messages to put into to ring buffer
//   by calling  ring->Put(message). Each message carries a message id
//   which is represened by integer "num". This message id should be put
//   into "value" field of the slot. It should also carry the id
//   of the producer thread to be stored in "thread_id" field so that
//   consumer threads can know which producer generates the message later
//   on. You need to put synchronization code
//   before and after the call ring->Put(message). See the algorithms in
//   page 182 of the textbook.

    for (num = 0; num < N_MESSG ; num++) {
      // Put the code to prepare the message here.
      // ...

      // Put the code for synchronization before  ring->Put(message) here.
      // ...

      ring->Put(message);

      // Put the code for synchronization after  ring->Put(message) here.
      // ...

    }
}
```

开头先初始化了一个消息插槽作为生产者生产的消息（消息包括消息的值与创建该消息的线程），对于每个生产者进程来说循环`N_MESSG`次，我们需要在`ring->Put(message)`前加入对缓冲区未生产信号量判断，后加上对缓冲区已生产信号量的判断，同时为了防止写冲突应该加上一个互斥信号量，在一个线程写入时其他线程无法写入。

- `Consumer(_int which)`：消费者线程调用的函数

```
void
Consumer(_int which)
{
    char str[MAXLEN];
    char fname[LINELEN];
    int fd;

    slot *message = new slot(0,0);

    // to form a output file name for this consumer thread.
    // all the messages received by this consumer will be recorded in
    // this file.
    sprintf(fname, "tmp_%d", which);

    // create a file. Note that this is a UNIX system call.
    if ( (fd = creat(fname, 0600) ) == -1)
    {
    perror("creat: file create failed");
    exit(1);
    }

    for (; ; ) {
      // Put the code for synchronization before ring->Get(message) here.
      // ...

      ring->Get(message);

      // Put the code for synchronization after ring->Get(message) here.
      // ...
```

```
    // form a string to record the message
    sprintf(str,"producer id --> %d; Message number --> %d;\n",
      message->thread_id,
      message->value);
    // write this string into the output file of this consumer.
    // note that this is another UNIX system call.
    if ( write(fd, str, strlen(str)) == -1 ) {
      perror("write: write failed");
      exit(1);
    }
  }
```

```
}
```

前面是关于消费者读取到消息并且写入文件的配置，for循环内是我们要修改的地方，同样需要在`ring->Get(message);`前后添加对缓冲区已生产空间信号量与缓冲区未生产空间信号量的PV操作

- `ProdCons()`：main.cc调用的方法，实现生产者消费者问题初始化

```
void ProdCons()
{
    int i;
    DEBUG('t', "Entering ProdCons");

    // Put the code to construct all the semaphores here.
    // ....

    // Put the code to construct a ring buffer object with size
    //BUFF_SIZE here.
    // ...
```

```
  // create and fork N_PROD of producer threads
  for (i=0; i < N_PROD; i++)
  {
    // this statemet is to form a string to be used as the name for
    // produder i.
    sprintf(prod_names[i], "producer_%d", i);

    // Put the code to create and fork a new producer thread using
    //     the name in prod_names[i] and
    //     integer i as the argument of function "Producer"
    //  ...

  };

  // create and fork N_CONS of consumer threads
  for (i=0; i < N_CONS; i++)
  {
    // this statemet is to form a string to be used as the name for
    // consumer i.
    sprintf(cons_names[i], "consumer_%d", i);
    // Put the code to create and fork a new consumer thread using
    //     the name in cons_names[i] and
    //     integer i as the argument of function "Consumer"
    //  ...

  };
```

}

我们需要添加对信号量的初始化、对缓冲区的初始化、以及对生产者线程与消费者线程的创建到转变为`running`态（调用fork）

3.3 实验代码

修改后的`prodcons++.cc`如下：

```
// prodcons++.cc
//    C++ version of producer and consumer problem using a ring buffer.
//
//    Create N_PROD producer threads and N_CONS consumer thread.
//    Producer and consumer threads are communicating via a shared
//      ring buffer object. The operations on the shared ring buffer
//      are synchronized with semaphores.
//
//
// Copyright (c) 1995 The Regents of the University of Southern Queensland.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "copyright.h"
#include "system.h"

#include "synch.h"
#include "ring.h"

#define BUFF_SIZE 3  // the size of the round buffer
#define N_PROD    2  // the number of producers
#define N_CONS    2  // the number of consumers
#define N_MESSG   5  // the number of messages produced by each producer
#define MAX_NAME  16 // the maximum lengh of a name

#define MAXLEN    48
#define LINELEN    24


Thread *producers[N_PROD]; //array of pointers to the producer
Thread *consumers[N_CONS]; // and consumer threads;

char prod_names[N_PROD][MAX_NAME];  //array of charater string for prod names
char cons_names[N_CONS][MAX_NAME];  //array of charater string for cons names

Semaphore *nempty, *nfull; //two semaphores for empty and full slots 未生产空间与已生产
空间
Semaphore *mutex;          //semaphore for the mutual exclusion 互斥信号量

Ring *ring;// 环缓冲区

void Producer(_int which)
{
    int num;
```

```
    slot *message = new slot(0,0);// 初始化一个消息slot，但未设置消息具体内容以及生产者

    for (num = 0; num < N_MESSG ; num++) {
      // Put the code to prepare the message here.
      // ...
      message->value = num;// 生产消息的值
      message->thread_id = which;// 生产者线程id
      // Put the code for synchronization before  ring->Put(message) here.
      // ...

      nempty->P();// 执行P操作 未生产空间-1
      mutex->P();// 开启互斥锁

      ring->Put(message);

      // Put the code for synchronization after  ring->Put(message) here.
      // ...
      nfull->V();// 执行V操作 已生产空间+1
      mutex->V();// 关闭互斥锁

    }

}

void Consumer(_int which)
{
    char str[MAXLEN];
    char fname[LINELEN];
    int fd;

    slot *message = new slot(0,0);

    // to form a output file name for this consumer thread.
    // all the messages received by this consumer will be recorded in
    // this file.
    sprintf(fname, "tmp_%d", which);

    // create a file. Note that this is a UNIX system call.
    if ( (fd = creat(fname, 0600) ) == -1)
    {
        perror("creat: file create failed");
        exit(1);
    }

    for (; ;) {

      // Put the code for synchronization before ring->Get(message) here.
      // ...
      nfull->P();// 执行P操作，已生产空间-1
      mutex->P();// 开启互斥锁
```

```
      ring->Get(message);

      // Put the code for synchronization after ring->Get(message) here.
      // ...

      nempty->V();// 执行V操作，未生产空间+1
      mutex->V();// 关闭互斥锁

      // form a string to record the message
      sprintf(str,"producer id --> %d; Message number --> %d;\n",
        message->thread_id,
        message->value);
      // write this string into the output file of this consumer.
      // note that this is another UNIX system call.
      if ( write(fd, str, strlen(str)) == -1 ) {
        perror("write: write failed");
        exit(1);
      }
    }

}

void ProdCons()
{
    int i;
    DEBUG('t', "Entering ProdCons");

    // Put the code to construct all the semaphores here.
    // ....
    nempty = new Semaphore("nempty",BUFF_SIZE);// 初始化缓冲区未生产空间=BUFF_SIZE
    nfull = new Semaphore("nfull",0);// 初始化缓冲区已生产空间=0
    mutex = new Semaphore("mutex",1);// 初始化互斥锁

    // Put the code to construct a ring buffer object with size
    //BUFF_SIZE here.
    // ...
    ring = new Ring(BUFF_SIZE);

    // create and fork N_PROD of producer threads
    for (i=0; i < N_PROD; i++)
    {
      // this statemet is to form a string to be used as the name for
      // produder i.
      sprintf(prod_names[i], "producer_%d", i);

      // Put the code to create and fork a new producer thread using
      //     the name in prod_names[i] and
      //     integer i as the argument of function "Producer"
      //  ...
      producers[i] = new Thread(prod_names[i]);// 初始化生产者线程
      producers[i]->Fork(Producer, i);// 线程fork 执行Producer(i)
```

```
    };

    // create and fork N_CONS of consumer threads
    for (i=0; i < N_CONS; i++)
    {
      // this statemet is to form a string to be used as the name for
      // consumer i.
      sprintf(cons_names[i], "consumer_%d", i);
      // Put the code to create and fork a new consumer thread using
      //     the name in cons_names[i] and
      //     integer i as the argument of function "Consumer"
      //  ...
      consumers[i]=new Thread(cons_names[i]);// 初始化消费者线程
      consumers[i]->Fork(Consumer,i);// 线程fork 执行Consume(i)
    };
}
```

3.4 实验结果

注意，本次运行操作系统应添加命令行参数`./nachos -rs 123`，表示随机切换线程。运行后生成两个文件表示0号消费者与1号消费者读取的消息，分别读取可以看到，两个消费者将生产者生产的所有消息随机有顺序的读出：

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab3$ ./nachos -rs 123
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1052, idle 42, system 1010, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab3$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 2;
producer id --> 1; Message number --> 3;
producer id --> 1; Message number --> 4;
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab3$ cat tmp_1
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 0;
producer id --> 1; Message number --> 1;
producer id --> 0; Message number --> 4;
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab3$
```

输入`./nachos -d t`转换为DEBUG模式，`t`**表示只读取有关线程的DEBUG信息**：

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/lab3$ ./nachos -d t
Entering ProdConsForking thread "producer_0" with func = 0x56641ea9, arg = 0
Putting thread producer_0 on ready list.
Forking thread "producer_1" with func = 0x56641ea9, arg = 1
Putting thread producer_1 on ready list.
Forking thread "consumer_0" with func = 0x56641fa7, arg = 0
Putting thread consumer_0 on ready list.
Forking thread "consumer_1" with func = 0x56641fa7, arg = 1
Putting thread consumer_1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "producer_0"
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Putting thread producer_0 on ready list.
Putting thread producer_1 on ready list.
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_0"
Now in thread "producer_0"
Deleting thread "main"
Putting thread consumer_0 on ready list.
Putting thread consumer_1 on ready list.
Finishing thread "producer_0"
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Now in thread "producer_1"
Deleting thread "producer_0"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Now in thread "consumer_0"
Putting thread producer_1 on ready list.
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Now in thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_1"
Now in thread "producer_1"
```

lab4 扩展文件系统

4.1 实验内容

扩展Nachos的基本文件系统。Nachos的文件系统是一个简单并且能力有限的系统，限制之一就是文件的大小是不可扩展的。通过扩展，使得文件的大小是可变的。在扩展写入文件内容时，一边写入，一边动态调整文件的长度及所占用的数据扇区。

4.2 实验思路

首先我们必须理解Nachos文件为什么是不可扩展的?

- 一个文件的大小是在创建时定的，在header，allocate时候，传入参数fileSize，按fileSize分配恰好合适大小的扇区数量
- 在openfile的write时，当写的位置超过fileSize时，会直接切掉超出部分，只是write不超出部分
- 所以一个文件的大小从创建开始是不会改变的

接下来就是文件拓展的思路：

就是在接收文件长度的时候，判断一下文件长度是否大于磁盘规定的位图中的位数，如果大于的话，就进行拓展操作，重新对位图中的内容进行编写，如果小于或等于的话，就保持之前的操作。这样的话，就可以完美实现文件的拓展操作。

4.3 实验代码

4.3.1 给FileHeader增加Extend（int newSize)

判断输入的所有长度是否超过扇区规定的长度，如果超过长度就进行扩展操作

```
bool
FileHeader::Extend(int newSize)
{

    if(newSize<numBytes)return FALSE;   //if not a extend operation

    if(newSize==numBytes)return TRUE;   //if size not change

    int newNumSectors  = divRoundUp(newSize, SectorSize);   //the number of sectors
the new size need to be allocated.
    if(newNumSectors == numSectors){
        numBytes = newSize;
        return TRUE;    //if number of sectors new size need equals the

    }

    int diffSector = newNumSectors - numSectors;

    OpenFile *bitmapfile = new OpenFile(0);
    BitMap *freeMap;
    freeMap = new BitMap(NumSectors);
    freeMap->FetchFrom(bitmapfile);
//    printf("debug in fhdr extend where new Sector=%d \n",freeMap->NumClear());
    if(newNumSectors>NumDirect||freeMap->NumClear()< diffSector)return FALSE;   //if
disk is full or file size is too big.

    //allocate the new sectors and store them into file header
    int i;
    for(i = numSectors; i<newNumSectors; i++)
    {
        dataSectors[i] = freeMap->Find();
    }
    freeMap->WriteBack(bitmapfile);
    numBytes = newSize;
    numSectors = newNumSectors;

    return TRUE;
}
```

4.3.2 更改OpenFile中的writeAt函数

writeAt函数的空间不够时，将触发文件扩展

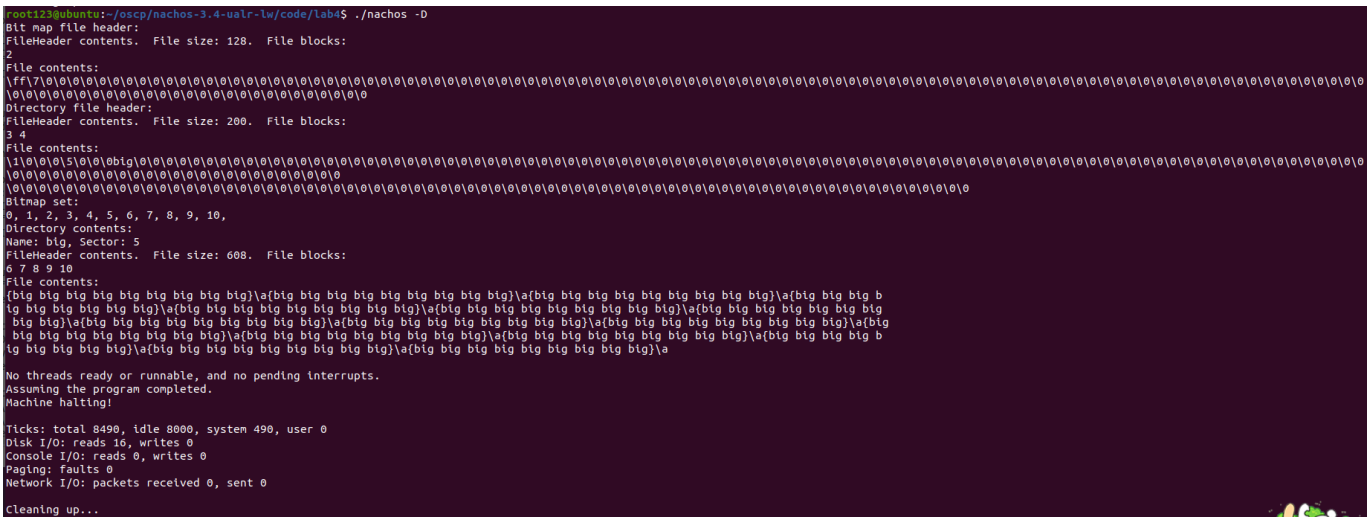
```
 if (numBytes <= 0) {// For original Nachos file system
//    if ((numBytes <= 0) || (position > fileLength))  // For lab4 ...
        return 0;
    }               // check request
    if ((position + numBytes) > fileLength) {
        hdr->Extend(position + numBytes);
        //printf("extend secess");
    }
```

4.3.3 在OpenFile中增加writeback函数，更新磁盘内容

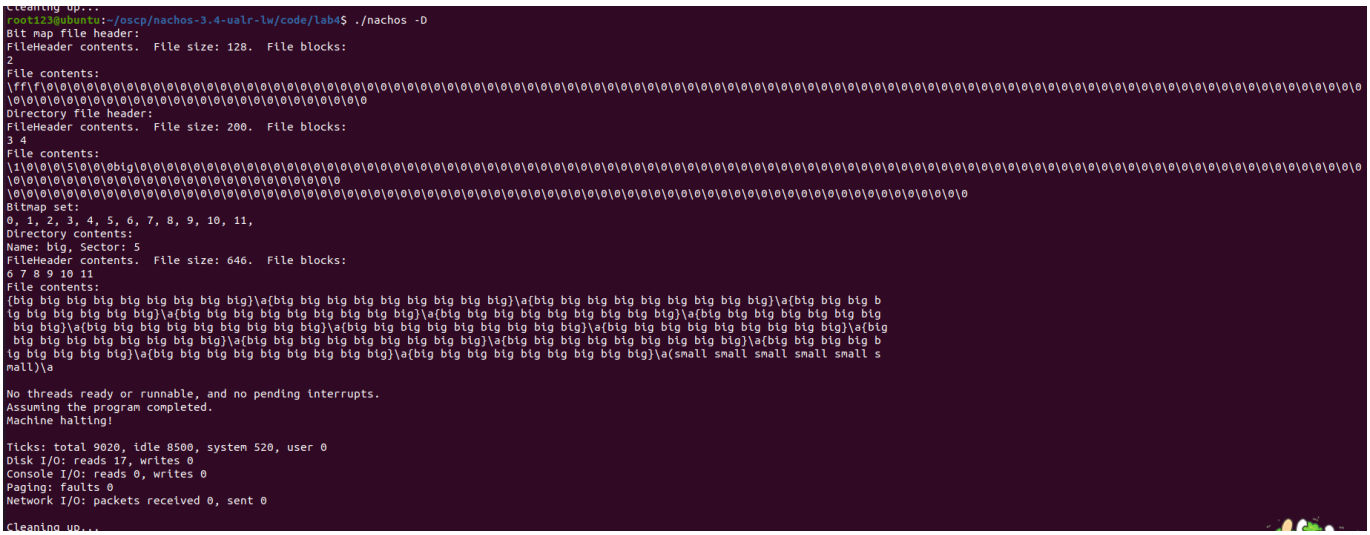
```
void OpenFile::WriteBack() {
hdr->WriteBack(hdrSector);
}
```

4.4 实验结果

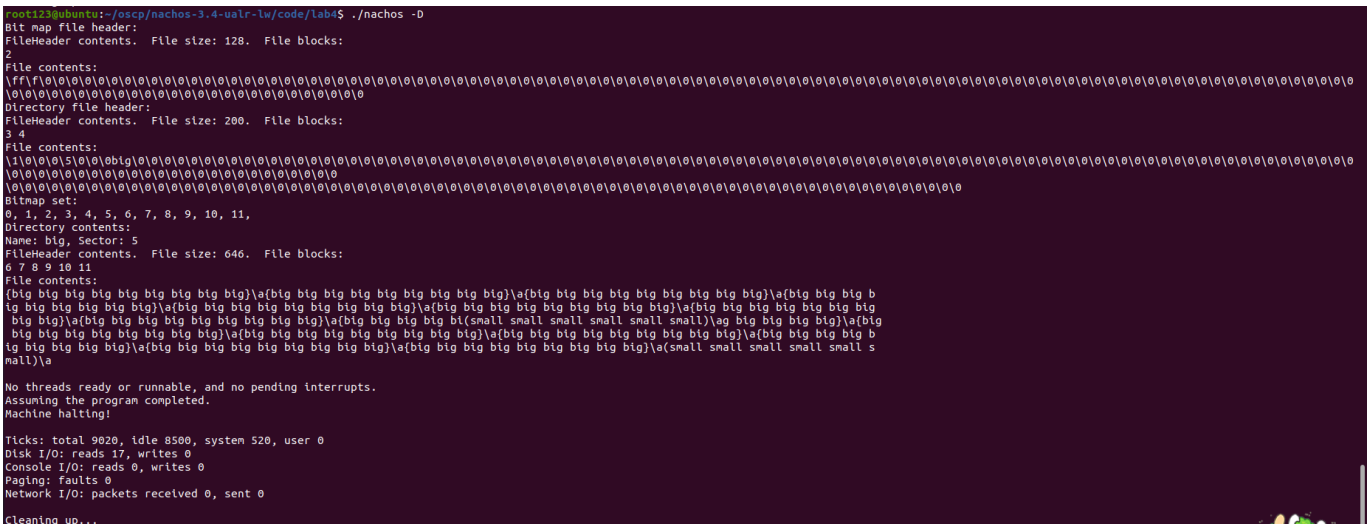
```
./nachos -f
./nachos -cp test/big big
./nachos -D
```




```
./nachos -ap test/small big
./nachos -D
```



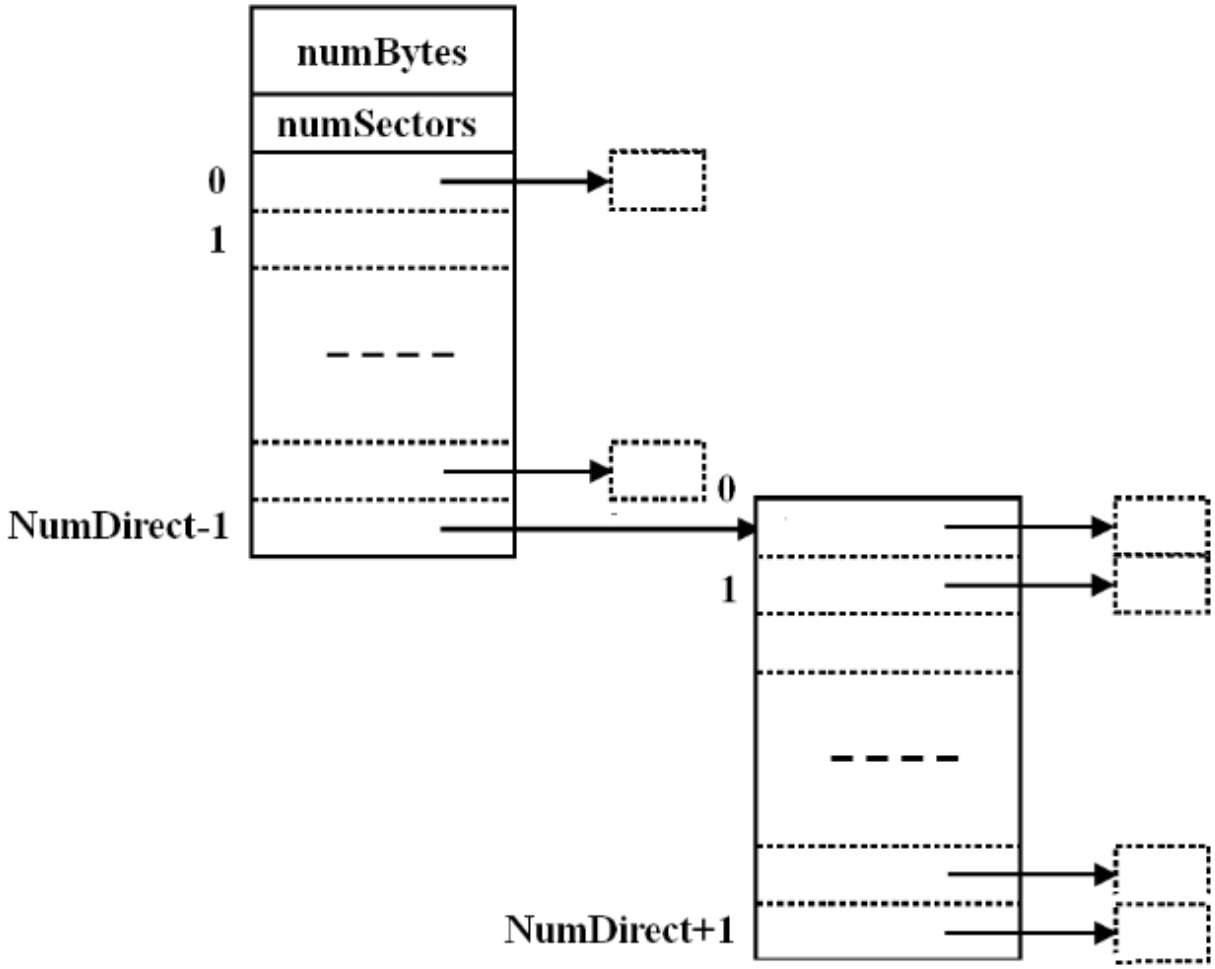

```
./nachos -hap test/small big
./nachos -D`
```



lab5 具有二级索引的文件系统

5.1 实验内容

Nachos系统原有的文件系统只支持单级索引，最大能存取NumDirect * SectorSize = 30 * 128 = 3840字节大小的文件。本实验将在理解原文件系统的组织结构基础上扩展原有的文件系统，设计并实现具有二级索引的文件系统。



二级索引文件头i-node设计：

如上图所示，构建具有二级索引的i-node，原先的前NumDirect-1项还是直接索引，最后一项(数组下标为NumDirect-1)指向一个二级索引块，这个块存放新的索引条目，共NumDirect+2项(数组下标为0~NumDirect+1)。扩大后的文件最大长度为（29 + 32）* 128 = 7808字节。

二级索引块是动态产生的，当文件大小不需要它时，一级索引块的最后一项设置为-1，此时不存在二级索引块。当文件大小增长到一级索引无法支持时，再分配一个新的块存二级索引，并将其扇区号存入一级索引块的最后一项，形成上图所示的结构。

5.2 实现思路

一个sector里128个字节，sectorSize = 128, sizeof(int) = 4，则NumDirect = 30，即有存储文件消耗30个sector，30*128 = 3840，即文件可占字节为3840。想要扩展文件系统，还是从存储文件字节的数组下手。建立一个dataSectors2数组，之前单级索引是用数组的最后1位作为索引号，dataSectors[0]-dataSectors[28]作为存取数据的块号；如果dataSectors[29]=-1，则表明没有没有

二级索引块，如果dataSectors[29]为任意正值，则与二级索引块相应块号对应，即dataSectors2[]值。文件超过dataSectors[]可存放大小，则放到dataSector2[]中，dataSectors2[]返回的Sector号对应着dataSectors[-1]的值。

5.3 实验代码

5.3.1 FileHeader中的Allocate（）

```
bool FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
   numBytes = fileSize;
   int lastindex = NumDirect - 1;
   numSectors = divRoundUp(fileSize, SectorSize);
   if (freeMap->NumClear() < numSectors)
      return FALSE; // not enough space
   else if (NumDirect + NumDirect2 <= numSectors)
      return false;
   if (numSectors < lastindex)
   {
      for (int i = 0; i < numSectors; i++)
         dataSectors[i] = freeMap->Find();
      dataSectors[lastindex] = -1;
   }
   else
   {
      for (int i = 0; i < lastindex; i++)
         dataSectors[i] = freeMap->Find();
      dataSectors[lastindex] = freeMap->Find();
      int dataSector2[NumDirect2];
      for (int i = 0; i < numSectors - NumDirect; i++)
         dataSector2[i] = freeMap->Find();
      synchDisk->WriteSector(dataSectors[lastindex], (char *)dataSector2);
   }
   return TRUE;
}
```

5.3.2 FileHeader中的deallocate（）

```
void FileHeader::Deallocate(BitMap *freeMap)
{
   int lastIndex = NumDirect - 1;
   if (dataSectors[lastIndex] == -1)
   {
      for (int i = 0; i < numSectors; i++)
      {
         ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
         freeMap->Clear((int)dataSectors[i]);
      }
   }
   else
   {
      int i = 0;
      for (; i < lastIndex; i++)
      {
         ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
         freeMap->Clear((int)dataSectors[i]);
      }
      int dataSector2[NumDirect2];
      synchDisk->ReadSector(dataSectors[lastIndex], (char *)dataSector2);
      freeMap->Clear((int)dataSectors[lastIndex]);
      for (; i < numSectors; i++)
         freeMap->Clear((int)dataSectors[i - lastIndex]);
   }
}
```

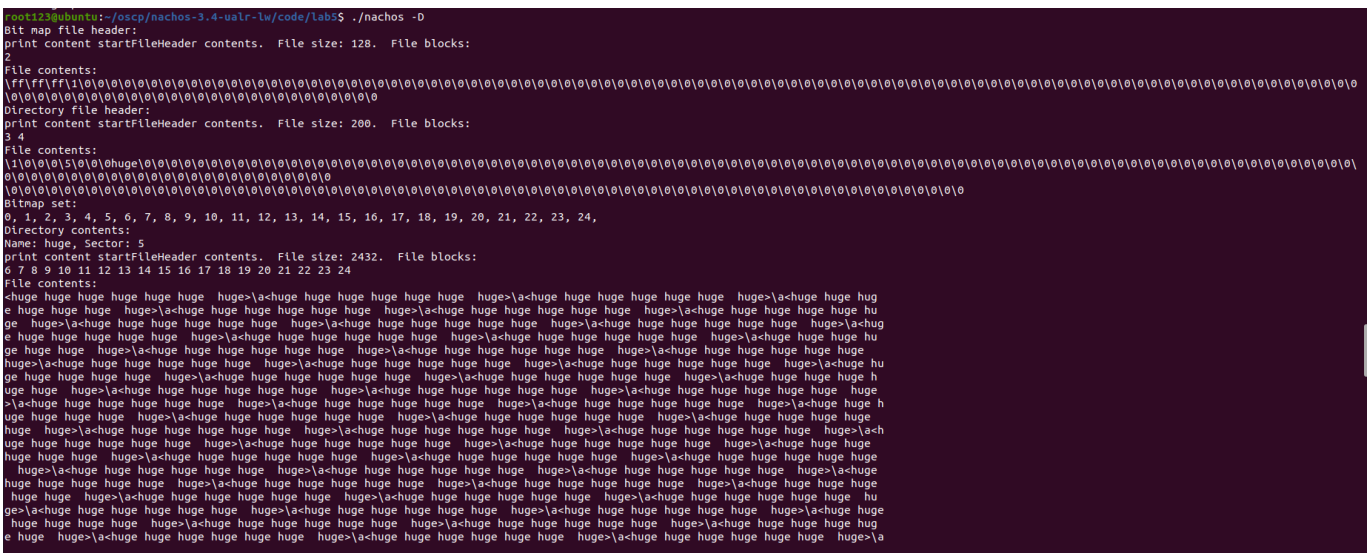
5.3.3 OpenFile中的Extend（）

```
bool FileHeader::ExtendFileSize(int filesize)
{
// printf("start extend \n");
   int newNumSectors = divRoundUp(filesize, SectorSize); //上取整

   if (newNumSectors == numSectors)
   {
      numBytes = filesize;
      return true; //扇区数量不变
   }

   int diffSector = newNumSectors - numSectors;

   OpenFile *bitmapfile = new OpenFile(0);
   BitMap *freeMap = new BitMap(NumSectors);
   freeMap->FetchFrom(bitmapfile);

   //printf("debug in fhdr extend where new Sector=%d \n",newNumSectors);
   if (newNumSectors > (NumDirect + NumDirect2) || freeMap->NumClear() < diffSector)
   {
      return false; //磁盘空间不足
   }
   //allocate the new sectors and store them into file header
   int i;

   if (newNumSectors < NumDirect)
   {

      for (i = numSectors; i < newNumSectors; i++)
      {
         dataSectors[i] = freeMap->Find();
      }
   }
   else
   {
//    printf("start extend, append \n");
      if (numSectors < NumDirect)
      {
         //原来无二级索引
         for (i = numSectors; i < NumDirect; i++)
         {
            dataSectors[i] = freeMap->Find();
         }
         int dataSectors2[NumDirect2];
         for (i = 0; i < newNumSectors - NumDirect + 1; i++)
            dataSectors2[i] = freeMap->Find();
         //将二级索引保存
         synchDisk->WriteSector(dataSectors[NumDirect - 1], (char *)dataSectors2);
      }
      else
```

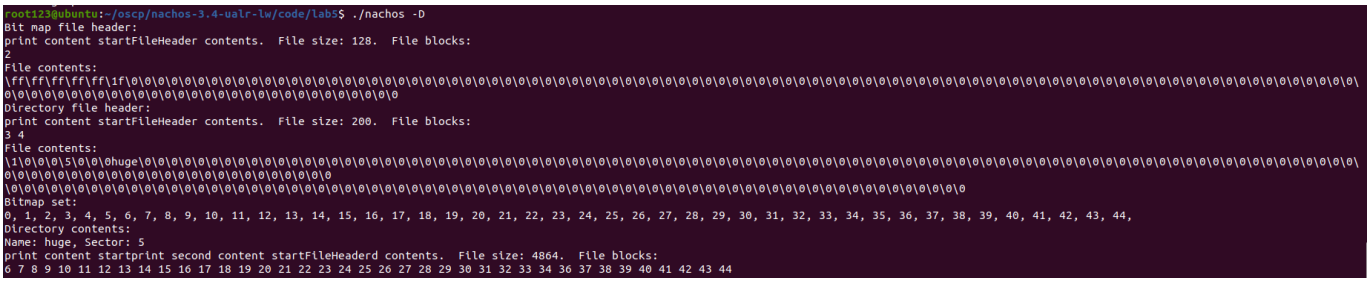
```
        { //原来有二级索引
            int dataSectors2[NumDirect2];
            synchDisk->ReadSector(dataSectors[NumDirect - 1], (char *)dataSectors2);
            for (i = numSectors - NumDirect + 1; i < newNumSectors - NumDirect + 1; i++)
               dataSectors2[i] = freeMap->Find();
            //将二级索引保存
            synchDisk->WriteSector(dataSectors[NumDirect - 1], (char *)dataSectors2);
        }
    }
    numBytes = filesize;
    numSectors = newNumSectors;

    freeMap->WriteBack(bitmapfile);

// printf("finish extend, append \n");
//    printf("finish extend \n");

    return true;
}
```

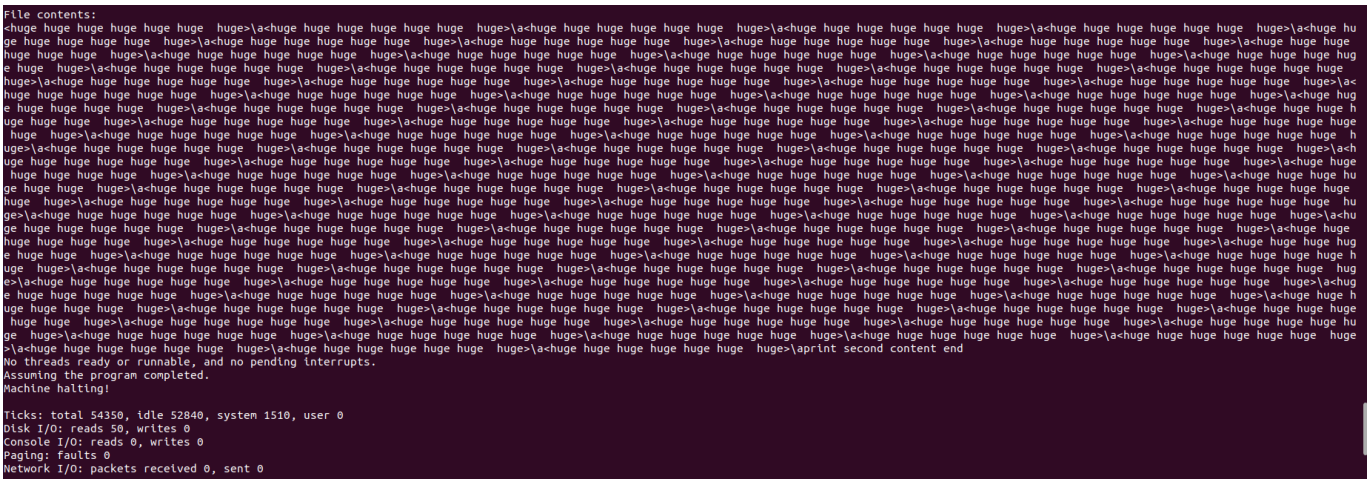
5.4 实验结果

```
./nachos -f
./nachos -cp test/huge huge
./nachos -D
```

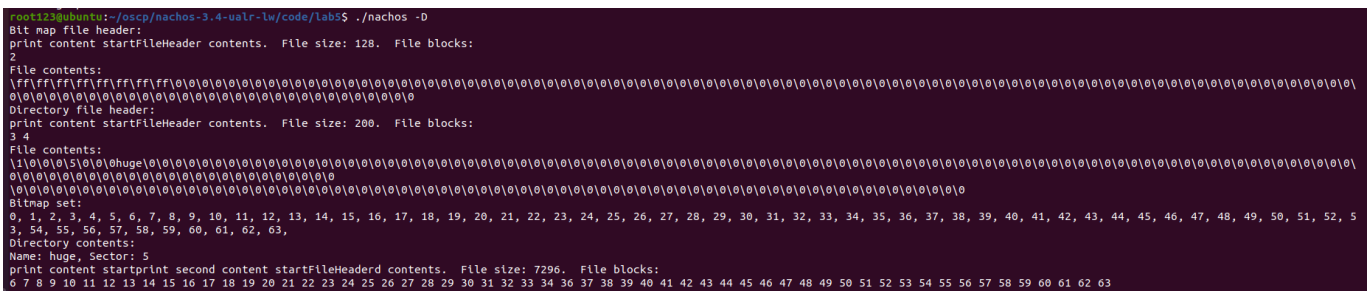


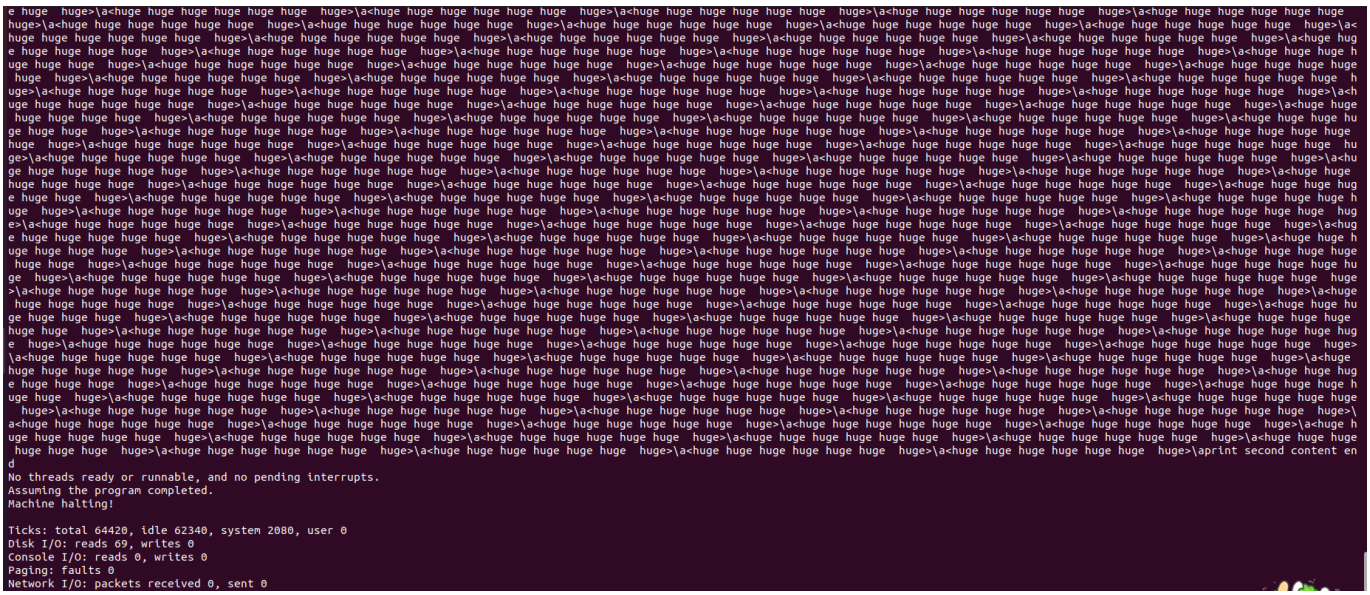
```
./nachos -ap test/huge huge
./nachos -D
```






```
./nachos -ap test/huge huge
./nachos -D
```



Lab6 系统调用与多道用户程序

6.1 实验内容

1. 扩展现有的class AddrSpace的实现，使得Nachos可以实现多道用户程序。
2. 按照实验指导书中的方法，完成class AddrSpace中的Print函数。
3. 实现Nachos 系统调用： `Exec()` 。

6.2 实验思路

- **如何理解用户进程如何映射到一个核心线程**

  在 progtest.cc中的StartProcess 函数，在为用户程序初始化了地址空间之后，执行一条命令 `currentThread->space = space` ，此命令即将用户进程映射到了核心线程之上。

- **如何理解当前进程的页表是如何与CPU使用的页表进行关联的**

  在创建用户进程的地址空间时，创建用户进程的页表项

```
pageTable = new TranslationEntry[numPages];// 初始化页表数组
for (i = 0; i < numPages; i++) {// 对每个页表初始化
    pageTable[i].virtualPage = i;    //目前为止，物理地址等于虚拟逻辑地址
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}
```

创建完页表之后，回到 `StartProcess` 函数中，调用了 `space->RestoreState()` 命令，该命令将用户进程的页表赋值给了 Machine 页表，具体代码如下所示：

```
void AddrSpace::RestoreState()
{
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
}
```

之后程序运行的过程就是通过 PC 寄存器中的虚拟地址通过 Machine 中的页表转化为物理地址，然后在将根据指令类型执行该指令。

- **如何实现内存页面分配**

实现多进程机制比较关键的问题就是**页**的分配问题，**Nachos默认机制下的内存分配是每个进程的虚拟逻辑地址与实际物理地址相同**，且每个进程都是从0开始分配，这样如果有多道用户程序，内存中就会来回被覆盖：

```
pageTable[i].virtualPage = i;    //目前为止，物理地址等于虚拟逻辑地址
pageTable[i].physicalPage = i;
```

我们采用`bitmap`数据结构保存空余页：MIPS模拟机(machine类)内存中有32个物理页面，也就是需要一个int(32位)，可以保存32个内存页面的空闲情况。bitmap的大小为32bits。主要使用`bitmap`的`find()`方法

```
#define BitsInByte     8
#define BitsInWord     32

class BitMap {
  public:
    BitMap(int nitems);        // Initialize a bitmap, with "nitems" bits
                              // initially, all bits are cleared.
    ~BitMap();                  // De-allocate bitmap

    void Mark(int which);       // Set the "nth" bit
    void Clear(int which);      // Clear the "nth" bit
    bool Test(int which);       // Is the "nth" bit set?
    int Find();                    // Return the # of a clear bit, and as a side
从前到后找到第一个0的位置，顺便置为1
                              // effect, set the bit.
                              // If no bits are clear, return -1.
    int NumClear();            // Return the number of clear bits

    void Print();             // Print contents of bitmap

    // These aren't needed until FILESYS, when we will need to read and
    // write the bitmap to a file
    void FetchFrom(OpenFile *file);     // fetch contents from disk
    void WriteBack(OpenFile *file);     // write contents to disk

  private:
    int numBits;            // number of bits in the bitmap
    int numWords;            // number of words of bitmap storage
                              // (rounded up if numBits is not a
                              //  multiple of the number of bits in
                              //  a word)
    unsigned int *map;        // bit storage
};
```

- **如何实现多道用户程序**

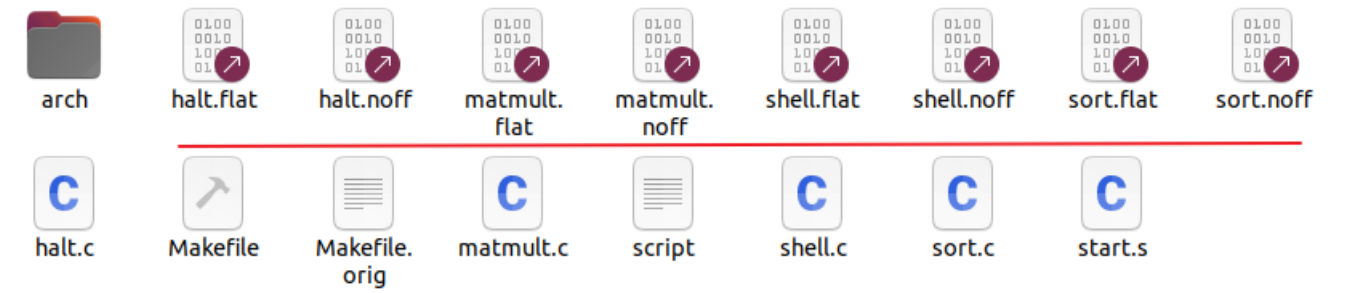
  在我们完成了多个程序同时驻留内存的内存分配算法后，我们就应当考虑用户父子进程并发执行的问题了。为了完成这一功能需要实现`Exec`系统调用的异常处理函数。

6.3 实验代码

6.3.1 添加Print函数

研究一下../test 目录中 Makefile 文件的内容，../test 中现有 5 个 C 语言用户源程序，可以通过 make 命令一次性编译连接生成它们的可执行文件和其在该目录中的符号链接。

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/test$ make
>>> Building dependency file for start.s <<<
>>> Building dependency file for sort.c <<<
>>> Building dependency file for matmult.c <<<
>>> Building dependency file for shell.c <<<
>>> Building dependency file for halt.c <<<
>>> Assembling start.s <<<
/lib/cpp -I../userprog -I../threads -D HOST_i386 -D HOST_LINUX start.s > arch/unknown-i386-linux/objects/tmp.s
/usr/local/mips/bin/decstation-ultrix-as   -o arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/tmp.s
rm arch/unknown-i386-linux/objects/tmp.s
>>> Compiling halt.c <<<
/usr/local/mips/bin/decstation-ultrix-gcc  -G 0 -c -I../userprog -I../threads -c -o arch/unknown-i386-linux/objects/halt.o halt.c
>>> Linking arch/unknown-i386-linux/objects/halt.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/halt.o -o
>>> Converting to noff file: arch/unknown-i386-linux/bin/halt.noff <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/halt.coff arch/unknown-i386-linux/bin/halt.noff
numsections 3
Loading 3 sections:
        ".text", filepos 0xd0, mempos 0x0, size 0x100
        ".data", filepos 0x1d0, mempos 0x100, size 0x0
        ".bss", filepos 0x0, mempos 0x100, size 0x0
ln -sf arch/unknown-i386-linux/bin/halt.noff halt.noff
>>> Compiling shell.c <<<
/usr/local/mips/bin/decstation-ultrix-gcc  -G 0 -c -I../userprog -I../threads -c -o arch/unknown-i386-linux/objects/shell.o shell.c
>>> Linking arch/unknown-i386-linux/objects/shell.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/shell.o -o
>>> Converting to noff file: arch/unknown-i386-linux/bin/shell.noff <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/shell.coff arch/unknown-i386-linux/bin/shell.noff
numsections 3
Loading 3 sections:
        ".text", filepos 0xd0, mempos 0x0, size 0x200
        ".data", filepos 0x2d0, mempos 0x200, size 0x0
        ".bss", filepos 0x0, mempos 0x200, size 0x0
ln -sf arch/unknown-i386-linux/bin/shell.noff shell.noff
>>> Compiling matmult.c <<<
/usr/local/mips/bin/decstation-ultrix-gcc  -G 0 -c -I../userprog -I../threads -c -o arch/unknown-i386-linux/objects/matmult.o matmult.c
>>> Linking arch/unknown-i386-linux/objects/matmult.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/matmult.o
ff
>>> Converting to noff file: arch/unknown-i386-linux/bin/matmult.noff <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/matmult.coff arch/unknown-i386-linux/bin/matmult.noff
numsections 3
Loading 3 sections:
        ".text", filepos 0xd0, mempos 0x0, size 0x3c0
        ".data", filepos 0x490, mempos 0x3c0, size 0x0
```



切换到./userprog/文件夹下，为了能够了解 Nachos 中多用户程序驻留内存的情况，可以在 `AddSpace` 类中增加以下打印成员函数 `Print()`：

```
void AddrSpace::Print() {
    printf("process spaceId: %d",spaceId);
    printf("page table dump: %d pages in total\n", numPages);
    printf("============================================\n");
    printf("\tVirtPage, \tPhysPage\n");
    for (int i=0; i < numPages; i++) {
        printf("\t%d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);
    }
    printf("============================================\n\n");
}
```

在 `progtest.cc` 的 `StartProcess(char *filename)` 方法中添加使得当为一个应用程序新建一个空间后，调用 Print() 函数来输出页表信息，具体修改结果如下所示：

```
void StartProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
    printf("Unable to open file %s\n", filename);
    return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;
    space->Print();// 打印内存使用情况

    delete executable;            // close file

    space->InitRegisters();        // set the initial register values
    space->RestoreState();        // load page table register

    machine->Run();            // jump to the user progam
    ASSERT(FALSE);            // machine->Run never returns;
                              // the address space exits
                              // by doing the syscall "exit"
}
```

编译Nachos内核，执行一个用户程序：

```
./nachos -x ../test/halt.noff
```

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-ualr-lw/code/userprog$ ./nachos -x ../test/halt.noff
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
page table dump: 10 pages in total
==========================================
        VirtPage,       PhysPage
        0,              0
        1,              1
        2,              2
        3,              3
        4,              4
        5,              5
        6,              6
        7,              7
        8,              8
        9,              9
==========================================

Machine halting!

Ticks: total 132, idle 0, system 120, user 12
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

查看 `halt.noff` 虚拟逻辑地址页号与物理地址页号的对应关系，以及 Nachos 为该程序分配的实页数。可以看到 Nachos 为该程序分配了 10 个实页。

6.3.2 扩展Nachos AddSpace类

对于实现内存页面的调用，添加两个静态全局变量 `Bitmap` 标识内存中的空页与进程空间池，添加一个成员变量 `spaceID` 表示当前进程的地址空间标识符，以及一个get函数：

```
    int getSpaceId();
  private:
    static BitMap *freeMap,*spaceIdMap;
    int spaceId; // 地址空间标识符
```

在 `AddSpace.cc` 的文件内进行初始化（全局变量不能在构造函数中初始化，类成员共有）

```
BitMap *AddrSpace:: freeMap = new BitMap(NumPhysPages);// 初始化bitMap
BitMap *AddrSpace:: spaceIdMap = new BitMap(256);// 定义spaceID池，
                                        //一个用户进程对应一个唯一的地址空间id，大小为
256个
```

初始化`AddSpace`时给`sapceID`赋值，在位图`SpaceIdMap`中查找未被分配的id号：

```
// 分配进程空间标识符
ASSERT(spaceIdMap->NumClear() > 0);// 确认页面足够分配
spaceId = spaceIdMap->Find();
```

接下来我们修改虚实页面分配的代码，对于每一个虚页，**我们在位图`freeMap`中找一个未被分配的页面作为虚页映射。**除此之外，我们需要保证在分配之前物理内存中空闲页面数量大于等于我们需要的页面数量，具体代码如下所示。

```
pageTable = new TranslationEntry[numPages];
ASSERT(freeMap->NumClear()>= numPages);// 确认页面足够分配
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;// 虚拟页面视图，从0开始
    pageTable[i].physicalPage = freeMap->Find();// 在位图中找到空闲页分配
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}
```

进行虚实页映射之后，我们需要将 noff 文件中的数据拷贝到`machine`的物理内存`mainMemory`中，因此我们需要将虚拟地址所对应的物理地址求出：**求出代码段或数据段的页表项，根据该页表项的物理页号 × 页大小 = 物理内存地址，再求出代码段或数据段的偏移量（无法被页大小整除的部分），相加得到起始物理内存地址。**

```
if (noffH.code.size > 0) {
        // pageTable[noffH.code.virtualAddr/PageSize]表示起始的页表项
        int pagePosition = pageTable[noffH.code.virtualAddr/PageSize].physicalPage *
PageSize;//计算出代码段在内存数组的起始下标
        int offset = noffH.code.virtualAddr % PageSize;// 代码段的偏移量
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
              (pagePosition+offset), noffH.code.size);// 修改DEBUG信息
        executable->ReadAt(&(machine->mainMemory[pagePosition+offset]),
            noffH.code.size, noffH.code.inFileAddr);//从pagePosition+offset处分配size大
小的内存空间
}
if (noffH.initData.size > 0) {
        int pagePosition = pageTable[noffH.initData.virtualAddr/PageSize].physicalPage
* PageSize;//计算出代码段在内存数组的起始下标
        int offset = noffH.initData.virtualAddr % PageSize;// 代码段的偏移量
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
              (pagePosition+offset), noffH.initData.size);// 修改DEBUG信息
        executable->ReadAt(&(machine->mainMemory[pagePosition+offset]),
            noffH.initData.size, noffH.initData.inFileAddr);//从pagePosition+offset处分
配size大小的内存空间
}
```

最后我们需要修改 AddrSpace 类的析构函数，我们需要在 AddrSpace 析构的时候将对应的位图的标识释放，具体代码如下所示：

```
AddrSpace::~AddrSpace()
{
    for(int i = 0;i < numPages;i++){
        freeMap->Clear(pageTable[i].physicalPage);// 释放位图中的内存页标识
    }
    spaceIdMap->Clear(spaceId);// 释放位图中的进程地址空间标识符
    delete [] pageTable;
}
```

6.3.3 实现系统调用Exec()

为了完成这一功能首先可以准备一个作为父进程的用户程序 `exec.c`：

```
#include "syscall.h"
int main()
{
    SpaceId pid;
    pid = Exec("../test/halt.noff");
    Halt();
}
```

为了能和已有的用户 C 程序一起生成可执行文件，可以修改../test/Makefile 文件将 exec 加入到 targets 定义中，在../test 中重新 make 生成 exec.noff 可执行文件。

```
targets = halt shell matmult sort exec
```

获取参数：在 `exec.c` 中为了生成子进程 `halt.noff`，使用了 Nachos 的系统调用 Exec。它带有一个字符串参数，是一个可执行文件名。在发生系统调用时系统内核需要得到这个**参数**并根据它建立子进程。

> 这里我们先看对应于 exec.c 的汇编代码，了解一下 MIPS 机指令系统对于参数传递是如何安排的：

```
    .file    1 "exec.c"
gcc2_compiled.:
__gnu_compiled_c:
    .rdata
    .align    2
$LC0:
    .ascii    "../test/exec.noff\000" # 用户地址空间
    .text
    .align    2       # 2 字节对齐，即 2*2
    .globl    main    # 全局变量
    .ent    main      # main函数入口
main:
# 汇编伪指令 frame 用来声明堆栈布局
# 该指令有三个参数：
# （1）第一个参数 framereg：声明用于访问局部堆栈的寄存器，一般为 $sp
# （2）第二个参数 framesize：声明该函数已分配堆栈的大小，符合 $sp+framesize = $sp
# （3）第三个参数 returnreg：这个寄存器用来保存返回地址
# $fp 为栈指针，该函数层栈大小为 32 字节，函数返回地址存放在 $31
    .frame    $fp,24,$31        # vars= 0, regs= 2/0, args= 16, extra= 0
    .mask    0xc0000000,-4
    .fmask    0x00000000,0
    # 栈采用向下生长的方式，即由大地址向小地址生长，栈指针指向栈的最小地址
    # $sp - 32 -> $sp，构造 main() 的栈 frame
    # $sp 的原值应该是执行 main() 之前的栈
    # 上一函数对应栈 frame 的顶（最小地址处）
    subu    $sp,$sp,24
    sw    $31,20($sp)     # $31 -> memory[$sp+20]
    sw    $fp,16($sp)     # $fp -> memory[$sp+16]
    move    $fp,$sp   # $sp -> $fp，执行 Exec() 会修改 $sp
    jal    __main     # PC+4 -> $31，goto_main
    # $LCO -> $4，将 Exec("../test/halt.noff\000")的参数的地址传给$4
    # $4 -> $7，传递函数的前四个参数给子程序，不够的用堆栈
    la    $4,$LC0
    # 转到 start.s 中的 Exec 处执行
    # PC+4 -> $31，goto Exec
    # PC 是调用函数时的指令地址
    # PC+4 是函数的下条指令地址，以便从函数返回时再调用
    # 函数的下条指令开始继续执行原程序
    jal    Exec
    jal    Halt
$L1:
   # $fp -> $sp
    move    $sp,$fp
    # memory[$sp+20] -> $31，取 main() 的返回值
    lw    $31,20($sp)
    # memory[$sp+16] -> $fp，恢复 $fp
    lw    $fp,16($sp)
    # $sp+24 -> $sp，释放 main() 对应的在栈中的 frame
    addu    $sp,$sp,24
    # goto $31，main() 函数返回
```

```
        j    $31
        .end    main
```

因此我们可以从 4 **号寄存器**中获取参数在内存的地址，然后根据该地址读出该参数并执行，初代的系统中断处理函数如下（**不完整**）：

```
case SC_Exec:{
                int fileAddr = machine->ReadRegister(4);
                char filename[50];
                for (int i=0;;i++){
                    machine->ReadMem(fileAddr + i, 1, (int *)&filename[i]);
                    if (filename[i] == '\0')
                        break;
                }
                printf("%s\n", filename);//输出文件名
                interrupt->Halt();//停机
                break;
}
```

```
huhao@huhao-virtual-machine:~/oscp/nachos-3.4-uair-lw/code/lab6$ ./nachos -x ../test/exec.noff
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
process spaceId: 0
page table dump: 11 pages in total
==========================================
        VirtPage,       PhysPage
        0,              0
        1,              1
        2,              2
        3,              3
        4,              4
        5,              5
        6,              6
        7,              7
        8,              8
        9,              9
        10,             10
==========================================

../test/halt.noff
Machine halting!

Ticks: total 133, idle 0, system 120, user 13
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看到我们成功输出了系统调用的参数。还要注意在文件../machine/mipssim.cc 中处理系统调用模拟指令的操作是以 return 返回的：

```
case OP_SYSCALL:
    RaiseException(SyscallException, 0);
    return;

...

// Advance program counters.
    registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
                        // are jumping into lala-land
    registers[PCReg] = registers[NextPCReg];
    registers[NextPCReg] = pcAfter;
```

这意味着在执行完系统调用后**是否令程序计数器向前推进（PC 值会指向下一条指令所在地址）**的工作交给了对应的**系统调用处理函数`ExceptionHandler(ExceptionType which)`**去决定。所以你还应当在`exception.cc`准备一个函数`AdvancePC()`以便当系统调用成功后向前推进程序计数器：

```
void AdvancePC() {
    machine->WriteRegister(PrevPCReg,machine->ReadRegister(PCReg));//前一个PC
    machine->WriteRegister(PCReg, machine->ReadRegister(PCReg) + 4);//当前PC
    machine->WriteRegister(NextPCReg, machine->ReadRegister(NextPCReg) + 4);//下一条
PC，对应下一个执行的指令
}
```

下面我们就可以考虑怎样实现 Exec 系统调用的异常处理函数 Exec 了。由于**系统调用引发异常**属于一种中断处理，因此可以把这类处理函数都封装到`Interrupt`类中，作为`Interrupt`类的成员函数：

```
// 新线程执行用户进程初始化操作
void InitProcess(int spaceId) {
    ASSERT(currentThread->space->getSpaceId() == spaceId);
    currentThread->space->InitRegisters();     // 设置寄存器初值
    currentThread->space->RestoreState();      // 加载页表寄存器
    machine->Run();             // 运行
    ASSERT(FALSE);
}

// 硬件中断处理
void Interrupt::Exec() {
    int fileAddr = machine->ReadRegister(4);
    char filename[50];
    for (int i=0;;i++){
        machine->ReadMem(fileAddr + i, 1, (int *)&filename[i]);
        if (filename[i] == '\0')
            break;
    }
    OpenFile *executable = fileSystem->Open(filename);
    if(executable == NULL) {
        printf("Unable to open file %s\n",filename);
        return;
    }
    printf("Exec(%s)\n",filename);
    AddrSpace *space = new AddrSpace(executable);// 建立新地址空间
    space->Print();   // 输出新分配的地址空间
    delete executable;    // 关闭文件

    Thread *thread = new Thread(filename); // 建立新核心线程
    thread->space = space; // 将用户进程映射到核心线程上

    thread->Fork(InitProcess,(int)space->getSpaceId());
    machine->WriteRegister(2,space->getSpaceId());// 返回地址空间标识符

    currentThread->Yield();// 当前线程放弃CPU 切换到新的线程
}
```

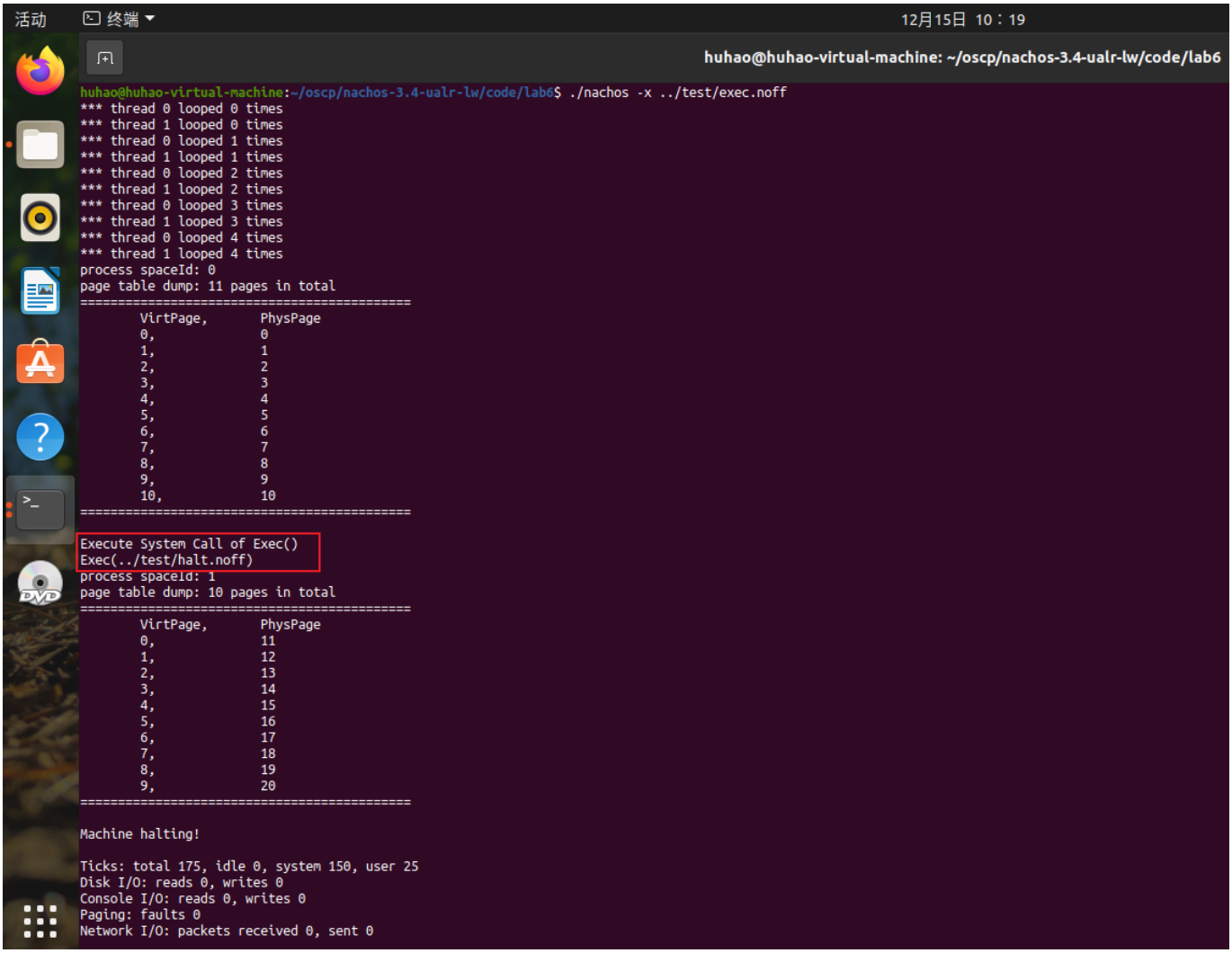
值得注意的是，当创建新的线程并且将其加入到等待队列后要将当前线程下CPU，否则执行无法调度新线程到RUNNING态，整个系统会卡死。

当异常的类型是系统调用异常且系统调用类型是`Exec`时，调用`interrupt->Exec()`进行中断处理。

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if (which == SyscallException) {
        switch (type) {
            case SC_Halt:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            }
            case SC_Exec:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Exec();// 触发硬件中断
                AdvancePC();// 程序计数器向前推进
                break;
            }
            default:{
                printf("Unexpected user mode exception %d %d\n", which, type);
                ASSERT(FALSE);
            }

        }
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

6.4 实验结果

运行exec.noff后实验结果如图：



Lab7 虚拟内存

7.1 实验内容

在未实现虚拟内存管理之前，Nachos在运行一个用户进程的时候，需要将程序运行所需所有内存空间一次性分配。虚拟内存实现将突破物理内存限制。本实验核心任务为根据理论学习中涉及的对换（Swapping）技术，设计并实现用户空间的虚拟内存管理。

页置换算法可以采用FIFO、二次机会、增强型二次机会、LRU等算法之一，或自己认为合适的其他算法。

注：可以用耗费用户内存比较多的code/test/sort.c作为虚拟内存的用户测试程序。但需要改两个地方。一是ARRAYSIZE原来定义的值1024太大，以至于程序输出刷屏太厉害，需要改小。二是最后的Exit系统调用还没有实现，可暂时简单地改为Halt系统调用

7.2 实验思路

虚拟内存的页面置换算法采用FIFO的页面置换算法，Nachos默认不使用TLB，需要在`execption.cc`中添加一个`PageFaultException`缺页错误的处理。

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if (which == SyscallException) {
        ...
    }else if(which == PageFaultException){
        printf("page fault exception %d %d\n", which, type);
        // 这里处理缺页错误
    }
    else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

对于FIFO算法，我们使用自带的List类，将可交换的页号添加到队列中。同时在为了获取页命中率，需要定义两个全局变量记录页命中次数与总的页请求次数。

7.3 实验代码

7.3.1 添加页统计信息

我在machine.h中定义了`PageHitCount = 0`;(记录页 Hit次数)； `TranslateCount = 0`(记录进程页面访问次数), 并在machine.cc中对它们初始化为0。每当调用translate.cc中的`translate()`函数时`TranslateCount++`，每当tlb命中时，`PageHitCount++`

```
int TLBHitCount;
int TranslateCount;

Machine::Machine(bool debug)
{
    int i;

    PageHitCount = 0;
    TranslateCount = 0;

    ...
}
```

在停机中断函数中添加对页命中信息的打印：

```
void
Interrupt::Halt()
{
    //    在程序执行结束后打印TLB信息
    printf("Page Miss: %d, Page Hit: %d, Total Translate: %d, Page Hit Rate:
%.2lf%%\n",
           machine->TranslateCount-machine->PageHitCount,
           machine->PageHitCount,
           Tmachine->TranslateCount,
           (double)(machine->PageHitCount*100)/(machine->TranslateCount));

    printf("Machine halting!\n\n");
    stats->Print();
    Cleanup();     // Never returns.
}
```

7.3.2 添加缺页错误处理

1.13节分析过：`Machine::RaiseException()`会将发生异常的虚拟地址存入Register[BadVAddrReg]，因此我们可以从 Register[BadVAddrReg] 中获取发生异常的虚拟地址。因为缺页异常也会引发中断，因此需要交给中断处理：

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if (which == SyscallException) {
        switch (type) {
            case SC_Halt:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            }
            case SC_Exec:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                printf("Execute System Call of Exec()\n");
                interrupt->Exec();
                AdvancePC();// 程序计数器向前推进
                break;
            }
            default:{
                printf("Unexpected user mode exception %d %d\n", which, type);
                ASSERT(FALSE);
            }

        }
    }else if(which == PageFaultException){
        int BadVAddr = machine->ReadRegister(BadVAddrReg);
        printf("page fault exception badAddress:%d\n", BadVAddr);
        interrupt->PageFault(BadVAddr);

    }
    else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

`PageFault(BadVAddr)`的实现如下：

```
/**
 * 系统发生缺页错误触发的中断函数
 * @param badVAddr 发生缺页错误的虚拟逻辑地址
 */
void Interrupt::PageFault(int badVAddr) {
    // 获取当前执行用户进程的虚拟页表
    TranslationEntry * virtualPageTable =  currentThread->space->getPageTable();

    // 虚拟页表中该页的valid=FALSE
    int newPage = badVAddr / PageSize;

    //调用FIFO算法确定置换页
    unsigned int oldPage = currentThread->space->FIFO(newPage);

    printf("发生页置换，虚拟页号: %d换出，虚拟页号: %d换入\n",oldPage,newPage);
    currentThread->space->writeBack(oldPage);
    virtualPageTable[oldPage].valid=false;
    virtualPageTable[newPage].physicalPage = virtualPageTable[oldPage].physicalPage;
    virtualPageTable[newPage].valid=true;
    virtualPageTable[newPage].dirty = false;
    virtualPageTable[newPage].readOnly = false;

    //读取需要置换的到内存中
    currentThread->space->readIn(newPage);
    // 打印相关信息
    currentThread->space->Print();
}
```

7.3.3 修改AddrSpace类

先添加**用户进程可执行文件名与页号队列**

```
  private:
    char* fileName;
    List* pageList;// 当前内存页号队列
```

需要着重修改该类的构造函数，将构造函数传入的参数改为**可执行的文件名**，char* FileName，已修改的部分如下，未修改的部分...省略。

每个用户进程最多可被分配 `max(4,代码段+初始化数据段的页长度=initPages)` 个可用的真实页表项，定义为 `necessaryPages` ，真实可用的页表项用 `valid=TRUE` 表示：

注意**代码段+初始化数据段的页表永远无法被换出**，自定义可换页长度为3，在自定义可换页初始化的时候将页号添加到FIFO队列中。

```
AddrSpace::AddrSpace(char* FileName)
{
    fileName = FileName;// 赋值
    pageList = new List;// 初始化一个FIFO队列
    OpenFile *executable = fileSystem->Open(FileName);

    ...

    // 整个用户进程的虚拟页长度
    numPages = divRoundUp(size, PageSize);

    // 用户进程的初始化部分（代码段+初始化数据段的页长度）,永远不能被换出
    int initPages = divRoundUp(noffH.code.size + noffH.initData.size,PageSize);

    // 用户进程无法初始化（代码段+数据段大于目前可用页长度）
    ASSERT(initPages <= NumPhysPages && initPages <= freeMap->NumClear());

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
          numPages, size);

    // 每个用户进程必须的页数（包含了之后可自由置换的页：3个）
    int necessaryPages = max(4,initPages+3);

    size = numPages * PageSize;

//  声明虚拟页表
    pageTable = new TranslationEntry[numPages];
//    ASSERT(freeMap->NumClear() >= numPages);// 确认页面足够分配
    for (i = 0; i < necessaryPages; i++) {

        if(i >= initPages)// 可置换的页放入优先队列
            pageList->Append((void*)i);

        // 用户进程的初始化部分的虚拟页表初始化
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = freeMap->Find();
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
                                        // a separate page, we could set its
                                        // pages to be read-only
    }
    for(i;i<numPages;i++){
        // 用户进程的其他部分的虚拟页表的初始化
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = -1;
        pageTable[i].valid = false;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
```

```
        pageTable[i].readOnly = FALSE;
    }

    ...
}
```

`AddrSpace`需要添加的函数如下：

```
public:
     // 获取虚拟页表
   TranslationEntry *getPageTable();
   // 打印
   void Print();
   // 被修改的页写回磁盘
   void writeBack(int oldPage);
   // 读取需要置换的页到内存中
   void readIn(int newPage);
   // 先进先出的置换
   int FIFO(int newPage);
```

- `getPageTable()`：返回该用户进程的虚拟页表

  ```
  TranslationEntry * AddrSpace::getPageTable(){
      return pageTable;
  }
  ```

- `Print()`：打印该进程的虚拟页表使用情况，**其中valid=True的页表项有效，视为在真实页表中**

  ```
  void AddrSpace::Print()
  {
      printf("spaceID: %d\n",spaceId);
      printf("page table dump: %d pages in total\n", numPages);
      printf("============================================\n");
      printf("  VirtPage, PhysPage, valid\n");
      for (int i=0; i < numPages; i++) {
          printf("\t%d,\t%d,\t%d\n",
  pageTable[i].virtualPage,pageTable[i].physicalPage,pageTable[i].valid);
      }
      printf("============================================\n\n");
  }
  ```

- `void writeBack(int oldPage)`：如果修改了，将虚拟页表中`oldPage`号页重新写入文件

```
void AddrSpace::writeBack(int oldPage)
{
    if(pageTable[oldPage].dirty){
        printf("被修改 dirty!写回磁盘，spaceId:%d,oldPage:%d\n",spaceId,oldPage);
        OpenFile *executable = fileSystem->Open(fileName);

        if (executable == NULL) {
            printf("Unable to open file %s\n", fileName);
            return;
        }
        executable->WriteAt(&(machine-
>mainMemory[pageTable[oldPage].physicalPage]),PageSize,oldPage*PageSize);
        delete executable;
    }else{
        printf("无需修改\n");
    }
}
```

- `void readIn(int newPage)`：读取需要置换的第`newPage`号虚拟页表到内存中：

```
void AddrSpace::readIn(int newPage) {
    OpenFile *executable = fileSystem->Open(fileName);
    if (executable == NULL) {
        printf("Unable to open file %s\n", fileName);
        return;
    }
    executable->ReadAt(&(machine-
>mainMemory[pageTable[newPage].physicalPage]),PageSize,newPage*PageSize);
    delete executable;
    printf("置换页已写入内存!");
}
```

- `int FIFO(int newPage)`：返回队列头部的页表项

```
int AddrSpace::FIFO(int newPage) {
    pageList->Append((void*)newPage);
    return (int)pageList->Remove();
}
```

7.3.4 修改progtest.cc

因为修改了`AddrSpace`的构造函数，所以要对progtest.cc进行修改，我们在命令行输入

```
Nachos -x 可执行文件路径(../test/exec.noff)
```

会自动调用这个函数

```
void StartProcess(char *filename)
{
    ...

    space = new AddrSpace(filename);

    ...
}
```

7.4 实验结果

测试时需要注意的是，因为没有实现Exit()系统调用，sort.c源文件需要将Exit()改为Halt()停机。同时缩小数组规模方便观察：执行sort.noff结果如下：

初始虚拟页表情况，此时队列为6-7-8：

```
spaceID: 0
page table dump: 16 pages in total
==========================================
  VirtPage, PhysPage, valid
         0,        0,     1
         1,        1,     1
         2,        2,     1
         3,        3,     1
         4,        4,     1
         5,        5,     1
         6,        6,     1
         7,        7,     1
         8,        8,     1
         9,       -1,     0
         10,      -1,     0
         11,      -1,     0
         12,      -1,     0
         13,      -1,     0
         14,      -1,     0
         15,      -1,     0
==========================================
```

第一次缺页，6换出15换入，此时FIFO队列为7-8-15：

```
page fault exception badAddress:2028
发生页置换，虚拟页号：6换出，虚拟页号：15换入
无需修改
置换页已写入内存!spaceID: 0
page table dump: 16 pages in total
==========================================
  VirtPage, PhysPage, valid
        0,      0,      1
        1,      1,      1
        2,      2,      1
        3,      3,      1
        4,      4,      1
        5,      5,      1
        6,      6,      0
        7,      7,      1
        8,      8,      1
        9,      -1,     0
        10,     -1,     0
        11,     -1,     0
        12,     -1,     0
        13,     -1,     0
        14,     -1,     0
        15,     6,      1
==========================================
```

第二次缺页，7换出6换入，此时FIFO队列为8-15-6：

```
page fault exception badAddress:768
发生页置换，虚拟页号：7换出，虚拟页号：6换入
无需修改
置换页已写入内存!spaceID: 0
page table dump: 16 pages in total
==========================================
  VirtPage, PhysPage, valid
        0,      0,      1
        1,      1,      1
        2,      2,      1
        3,      3,      1
        4,      4,      1
        5,      5,      1
        6,      7,      1
        7,      7,      0
        8,      8,      1
        9,      -1,     0
        10,     -1,     0
        11,     -1,     0
        12,     -1,     0
        13,     -1,     0
        14,     -1,     0
        15,     6,      1
==========================================
```

第三次缺页，8换出7换入，系统停机，可以看到一共发送3次缺页，命中率约等于100%：

```
page fault exception badAddress:896
发生页置换，虚拟页号：8换出，虚拟页号：7换入
无需修改
置换页已写入内存!spaceID: 0
page table dump: 16 pages in total
==========================================
  VirtPage, PhysPage, valid
        0,      0,      1
        1,      1,      1
        2,      2,      1
        3,      3,      1
        4,      4,      1
        5,      5,      1
        6,      7,      1
        7,      8,      1
        8,      8,      0
        9,      -1,     0
        10,     -1,     0
        11,     -1,     0
        12,     -1,     0
        13,     -1,     0
        14,     -1,     0
        15,     6,      1
==========================================

Page Miss: 3, Page Hit: 269930, Total Translate: 269933, Page Hit Rate: 100.00%
Machine halting!

Ticks: total 212209, idle 0, system 120, user 212089
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```