目录

内容一：总体概述

本次 lab 主要是理解 nachos 内存机制如何工作的。从一开始的单道程序设计到支持多道程序的虚拟内存机制。理解内存在操作系统中工作的地位，体会程序如何寻址读写数据运行下去。理解页表机制，虚存机制等。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise7	Challenge
	Y	Y	Y	Y	Y	Y	Y	Y

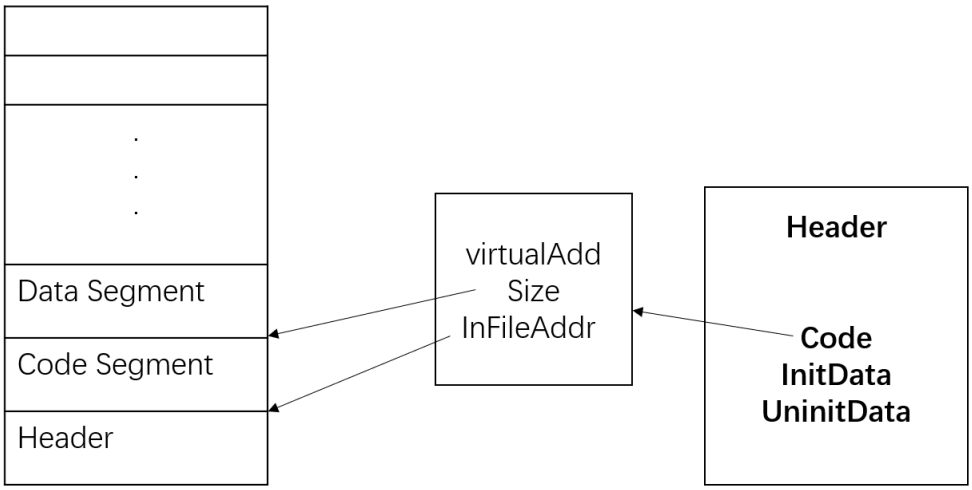
具体 Exercise 的完成情况

第一部分

Exercise1 源代码阅读

Nachos 设置了特定的机制来执行用户程序。

Nachos 的可执行文件组织形式以文件头开始，文件头记录了该文件的代码段位置，大小，记录了该文件的数据段位置，大小。通过这种记录方式，在文件读取入内存的时候找到相应的位置。

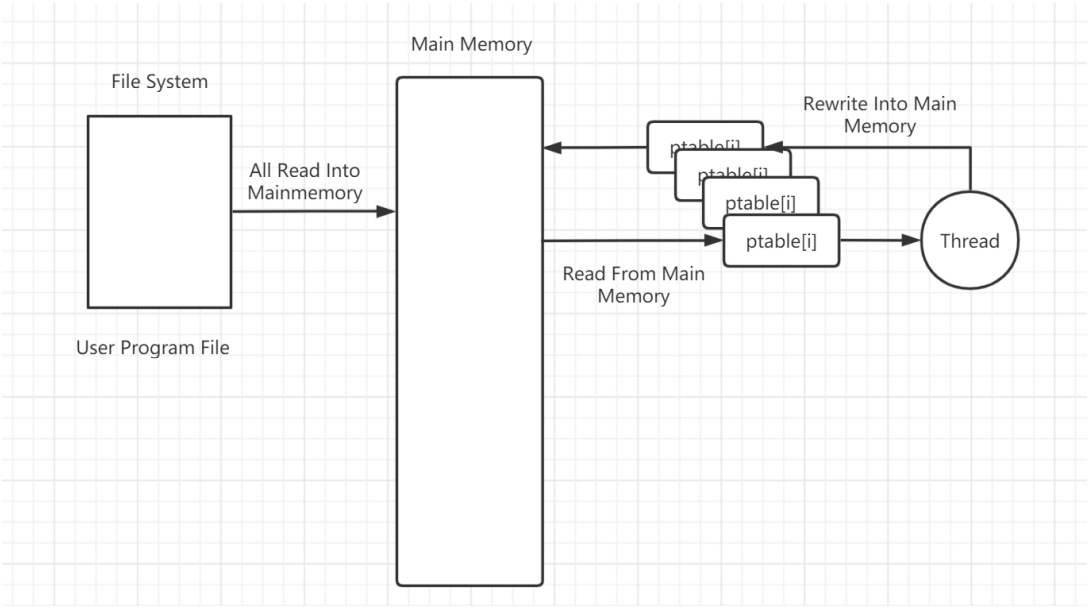


Nachos 定义了 machine 类来模拟用户程序执行过程中硬件设施。其中定义了 Mainmemory，pagetable 数据成员，Mainmemory 主要模拟了系统的主存，pagetable 主要模拟了系统的页表机制。通过这些机制，模拟用户程序如何从虚拟地址转换成物理地址，并从

物理地址读取或写入数据的机制。Nachos 代码中是先将用户的程序文件直接一次性读入主存，读入主存以后，在用户程序线程所属的页表中记录虚拟页号和物理页号之间的对应关系，通过这个对应关系，每次使用虚拟地址可以得到物理地址。现阶段 nachos 的地址转化没有做什么工作，他只是简单地将虚拟页号等于物理页号，通过这样找到数据和指令所在的物理地址。

除了页表机制，nachos 还模拟了 TLB 快表机制。开启快表机制之后，每次虚拟地址和物理地址的转换都会把相应的表项存进 TLB。每次读取内存的时候都会从 TLB 找到相应的表项进行地址转换。

Nachos 也考虑到了缺页异常的问题。事实上，在 nachos 自己提供的 test/sort.c 函数中，这个文件太大无法一次性直接装入内存，所有在实验过程中必然要一部分一部分装入内存，这就会触发缺页异常。Nachos 的缺页异常标志是个枚举类型 PageFault，通过 RaiseExpression，调用 Expressionhandler 函数。Nachos 的缺页异常包括两种类型，第一种类型是 TLB 的缺页，第二种是 pagetable 的缺页，这两种缺页异常都需要我们自己去实现。



Exercise2 TLB MISS 异常处理

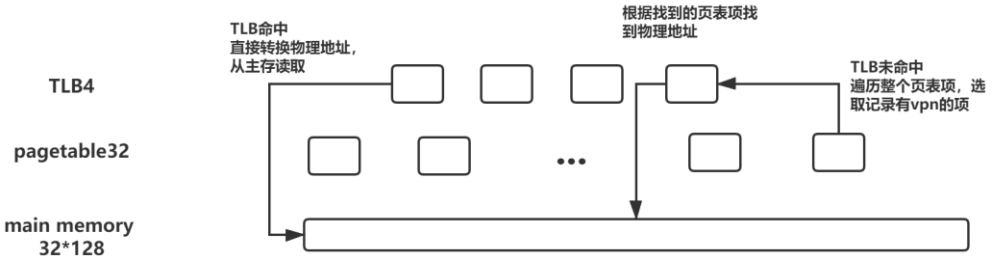
Nachos 默认没有开启模拟 TLB。TLB 部分代码属于条件编译语句，需要在 makefile 文件中配置编译#ifdef USE_TLB 中的代码以开启 TLB。开启 TLB 之后，在需要读写内存调用 translate 函数进行虚地址和实地址转换的时候，nachos 会直接遍历 TLB，如果找不到对应的记录，它将会先把当前需要翻译的地址，放在 BadVAddrReg 寄存器中。而后通过 RaiseException 函数，抛出一个 PageFaultException 类型的异常。在该异常的的处理函数中，我们可以添加异常处理的程序，将需要的页表项复制到 TLB 中。

为了让用户程序回到正轨，我们需要修改 machine 中对内存的读写操作函数。缺页异常发生在地址翻译的过程中，所以，在进行地址翻译的时候，第一次抛出异常，等待异常处理函数结束后，再次尝试翻译地址，这样就可以让用户程序回到正轨。

程序修改的地方：

Makefile:　Define -DUSE_TLB

Expectation.cc:　ExceptionHandler()

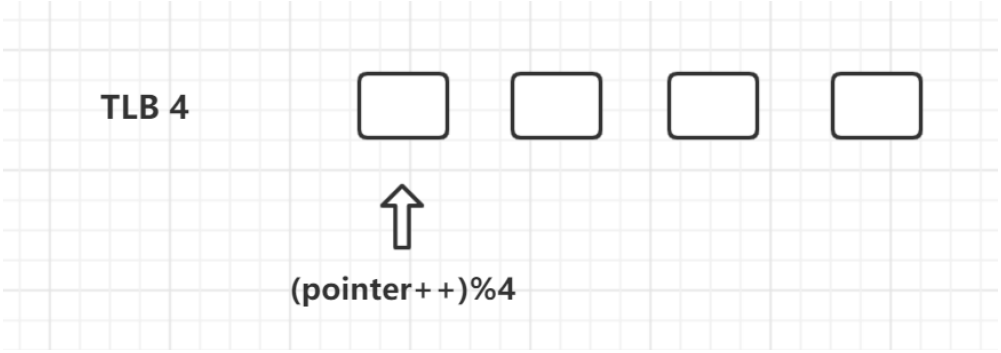


Exercise3 置换算法

接上一个实验。tlb 大小只有 4 项，需要不断换入换出 tlb 页表项。本实验实现了 FIFO 算法和 LRU 算法。为了衡量替换算法的性能，在 machine 类中，加入需要计量的变量。维护两个变量 findtimes 和 misstimes 分别代表查 TLB 表的次数，和缺页异常的次数。每次遍历 TLB 的时候，findtimes 自增，每次缺页异常的时候 misstimes 自增，统计这两个数据以衡量置换算法效率。

为了实现 FIFO 算法，在 machine 类中，维护一个 queuePointer 变量，该变量总是指向一个循环队列队首的下一个元素。通过这种方式，总是可以将最先进入队列的页表项替换出来。从而实现先进先出的策略。

为了实现 LRU 算法，在 machine 类中，维护一个 flag 数组，flag 下标对应 TLB 的相应的下标，flag 的内容是访问标志位，每一次访问，都会让被访问的 TLB 表项 flag 位置 0，其他表项的 flag 位加 1。每次发生缺页异常只需要找到 flag 位最大的那个表项替换出来就可以。这样就实现了替换最久未使用的表项。



修改的部分：

Machine.h:

```
int queuePointer; // FIFO 算法队列指针.
int tlbflag[TLBSize];//LRU 算法队列指针
int findtimes;     //统计信息 tlb 访问总次数
int misstimes;   //统计信息 tlb 失效次数
int Swaptlb();//LRU 算法淘汰函数
void Hittlb(int hiti);//LRU 算法计数函数
```

exception.cc

```
#define SWAPALG 2//两种置换算法  1 位 FIFO 2 位 LRU
ExceptionHandler()//增加处理函数淘汰替换 TLB
```

测试结果：

对于 FIFO 算法和 LRU 算法，分别使用 test/sort 中的排序程序查看 TLBmissRate。

由于给定的程序待排序数组太大，故而修改数组大小位 10，50，100，500。分别测量置换

算法的效率。

FIFO：

Size: 10

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:2904    Miss:140    MissRate:5.07%
Machine halting!

Ticks: total 2334, idle 0, system 10, user 2324
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...
```

Size:50

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:60504    Miss:3755    MissRate:6.62%
Machine halting!

Ticks: total 48614, idle 0, system 10, user 48604
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...
```

Size:100

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:236004    Miss:15125    MissRate:6.85%
Machine halting!

Ticks: total 189714, idle 0, system 10, user 189704
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...
```

Size:500

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:5780004    Miss:378908    MissRate:7.02%
Machine halting!

Ticks: total 4648514, idle 0, system 10, user 4648504
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
```

LRU：

Size:10

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:2904   Miss:92   MissRate:3.27%
Machine halting!

Ticks: total 2334, idle 0, system 10, user 2324
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
```

Size:50

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:60504   Miss:3127   MissRate:5.45%
Machine halting!

Ticks: total 48614, idle 0, system 10, user 48604
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
```

Size:100

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:236004   Miss:12735   MissRate:5.70%
Machine halting!

Ticks: total 189714, idle 0, system 10, user 189704
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
```

Size:500

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
StartProcess ENter
ALL:5780004   Miss:320252   MissRate:5.87%
Machine halting!

Ticks: total 4648514, idle 0, system 10, user 4648504
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
```

由上图的比较可以看出，LRU 算法的效率一般高于 FIFO 的效率。

第二部分

Exercise4 内存全局管理数据结构

在 nachos 中已经实现了一个比特图类型的数据结构 BitMap。在 BitMap 类中定义了一个 int 数组来记录所有的空闲内存块。定义一系列方法来维护这个数据结构。在内存分配过程中，最重要的方法主要是 Find(),Clear(),Print()。Find 方法主要目的是找到一个未分配的物理内存，并且将该内存进行分配，将该内存对应的位置为不可用。Clear 方法主要是将一个物理页框对应的位置置为空闲，这样，下次需要分配物理页号的时候，这个位置就可以继续使用了。Print()方法打印所有已经使用的页框号，可以很直观看出当前系统的内存分配情况。

了解了 BitMap 数据结构，可以在 machine 类中添加一个 bitMap 类型的成员变量，用来记录全局的内存分配信息。在每次 machine 类初始化的时候，也会将这个成员变量进行初始化。在每次分配内存的时候，调用其基本的方法 Find 找到页框号，进行记录。分配页框的地方主要在 AddrSapce 类的构造函数当中。在 AddrSpace 类的析构函数中，把分配出去的页框号回收。在 machine 类的析构函数中销毁对应的 BiteMap。使用这种方法，得到的实验结果如下：

开启 TLB：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
USED TLB
StartProcess Enter
After Allocating Addr Space:
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
After Recollect the Allocated Addr Space:
Bitmap set:

ALL:236004   Miss:12735   MissRate:5.70%
Machine halting!

Ticks: total 189714, idle 0, system 10, user 189704
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...
```

未开启 TLB

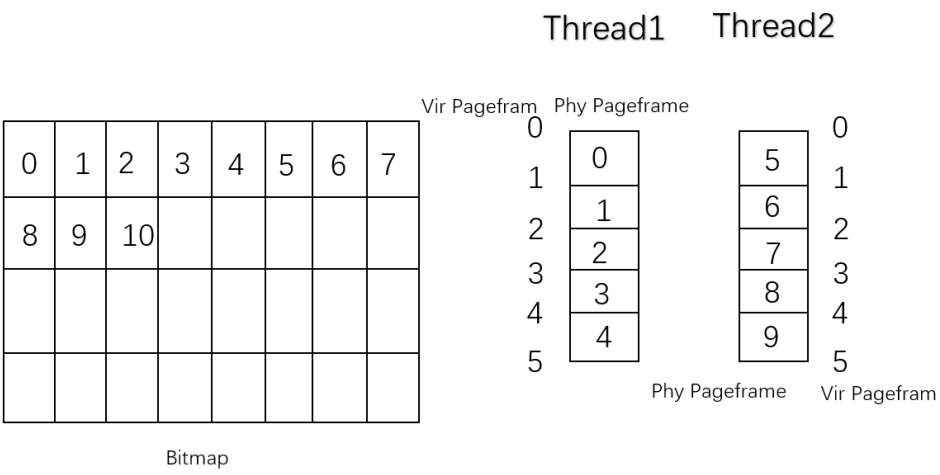
```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
USED Linear page table
StartProcess Enter
After Allocating Addr Space:
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
After Recollect the Allocated Addr Space:
Bitmap set:

Machine halting!

Ticks: total 189714, idle 0, system 10, user 189704
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...
```

Exercise5 多线程支持

为了支持多线程运行在 nachos 上，需要把主存 mainmemory 分按照字节分成多页。在上一个练习中实现的 bitmap 功能只能支持单道程序，本质上，由于程序代码和数据段在装入内存的时候，是直接连续从 0 读入主存中的，物理页号和逻辑页号还是需要保持一致，才能让程序正常运行。在本实验中，修改了将程序段读入主存的方法。具体来说，会先为虚拟地址分配对应的物理页框号，存入用户程序的页表中。这样，在把程序段装入主存中的时候，需要按照对应的转换规则，把虚拟地址转换成物理地址，再把对应的代码和数据段填入主存相应的位置。为了让虚拟地址和物理地址不相同，在分配页框的时候直接从后往前分配，先分配最后一个页框，再往小递减分配。同时，由于开启了 TLB，所以每次切换的时候，TLB 会失效，这个时候就需要把 TLB 所有的页的有效位设置为 FALSE。

测试是否支持多线程，主要步骤是在 StartProcess 中再开一个线程，这个第二个线程 fork

后直接抢占 CPU，先执行。执行完毕后再切下 CPU。观察内存占用情况，内存一步步增加，当线程消亡后，其分配过的内存又会被回收。



代码修改：

exception.cc

exceptionHandler()函数

// 修改系统调用 EXIT 的处理函数，主要是将线程 finish 掉

addrspace.cc

AddrSpace(OpenFile *executable)//构造函数，添加装调用户数据和代码段的处理流程

Thread.cc

Fork()//添加了抢占机制

progtest.cc

StartProcess()//在程序中新开了一个线程

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Successfully Add Thread main to GlobalList!
USED TLB
Main Thread strat and First User thread Start!
After Allocating Addr Space:
Bitmap set:
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
After Allocating Addr Space:
Bitmap set:
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
Successfully Add Thread SecondThread to GlobalList!
SecondThread Userpro Start!
Thread SecondThread EXIT
Successfully Remove Thread SecondThread from GlobalList!
After Recollect Thread SecondThread the Allocated Addr Space:
Bitmap set:
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
Thread main EXIT
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Assuming the program memory has been recllected.
ALL:1718   Miss:60   MissRate:3.62%
Machine halting!

Ticks: total 1400, idle 0, system 30, user 1370
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
```

Exercise6　缺页中断处理

与 Exercise7 一起实践。

第三部分

Exercise7 Lazy-loading

关于 TLB 的缺页中断在上个实验已经实现了。在本次实验中，主要实现基于页表的 lazy-loading。为了展示 lazy-loading 和缺页中断机制，在实验中限制了页表的页表项数量为 1，这样，在程序运行过程中就需要不断进行从磁盘到主存的装载。由于整个系统只存在一个页面，即使 nachos 为主存分配了 32 个页面，但是实际上使用的页面数只有一页。

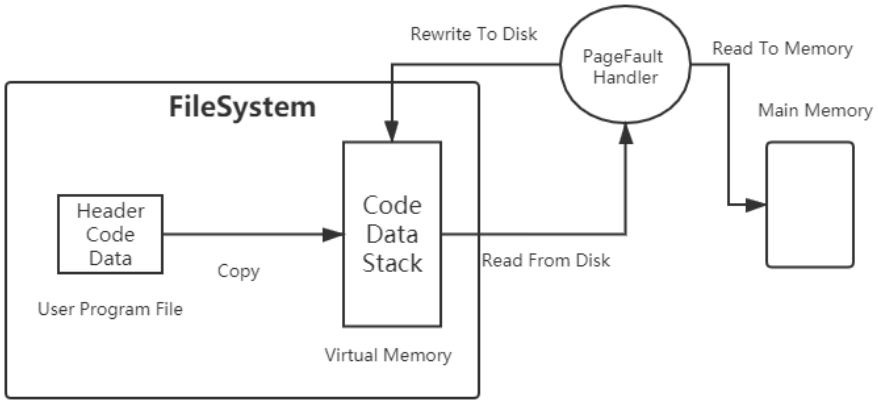
实验从现在开始使用虚拟内存的概念。在磁盘空间开辟一个文件，用这个文件来模拟虚拟内存。修改 addrspace 函数，该函数之前是将用户程序文件直接全部装载到主存中，现在将不会直接读入主存，因为对于用户的 test/sort.c 程序来说，排序数量为 1000 的时候，主存就装不下了。主存空间有限，但是磁盘的空间是可观的。用磁盘模拟虚拟内存，需要先将用户程序的各个段，都读入虚拟内存，同时要考虑到用户堆栈。这样，在用户程序眼中，所有内容都已经装进了虚拟内存。然而实际上，主存中并没有任何关于用户程序的内容。在每次运行需要读取内存的时候，系统才会从虚拟内存中读取一页到主存中。在用户程序涉及的指令和数据不在主存中的时候，就需要将新的页面调入主存。

具体的流程为，开辟一个虚存文件，文件大小为用户文件长度加上用户程序堆栈长度。模拟将用户程序读入虚存。用户程序开始运行后，需要取指令和数据。这时主存是空的，发生了缺页中断。进而转到缺页中断代码。处理缺页中断的逻辑，是从虚存文件中读取一页大小的数据到主存中。并将虚拟的页框号和物理页框号记录到页表项中。这样程序就可以取指令和数据运行下去了。

我们知道，在程序运行过程中，不只有读取的操作的，还有写入的操作。具体写入的操作，我们会把页表项的 dirty 位置为 True，标志着该页框的值被修改了。这时，将该页框换出的时候需要将这个页框的内容写回虚拟内存。

下图是一个页框时程序运行截图。

每次缺页异常和脏位回写，都会打印出来。



代码修改：

exception.cc

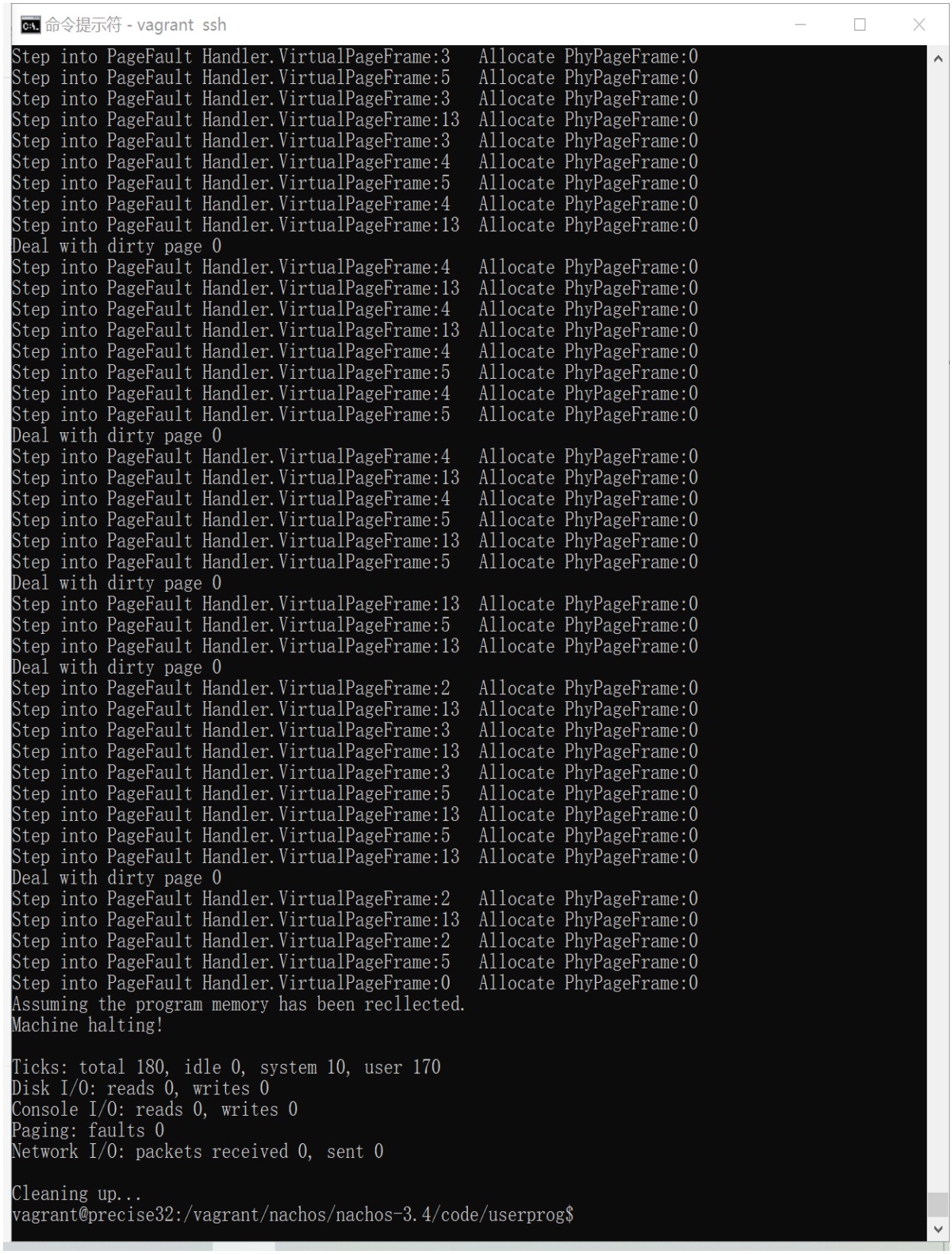
exceptionHandler()//增加页表缺页处理机制

addrspace.cc

AddrSpace(OpenFile *executable)//将用户文件直接装入虚拟内存，不再装入主存

Bitmap.cc

Find()//增加脏位回写机制

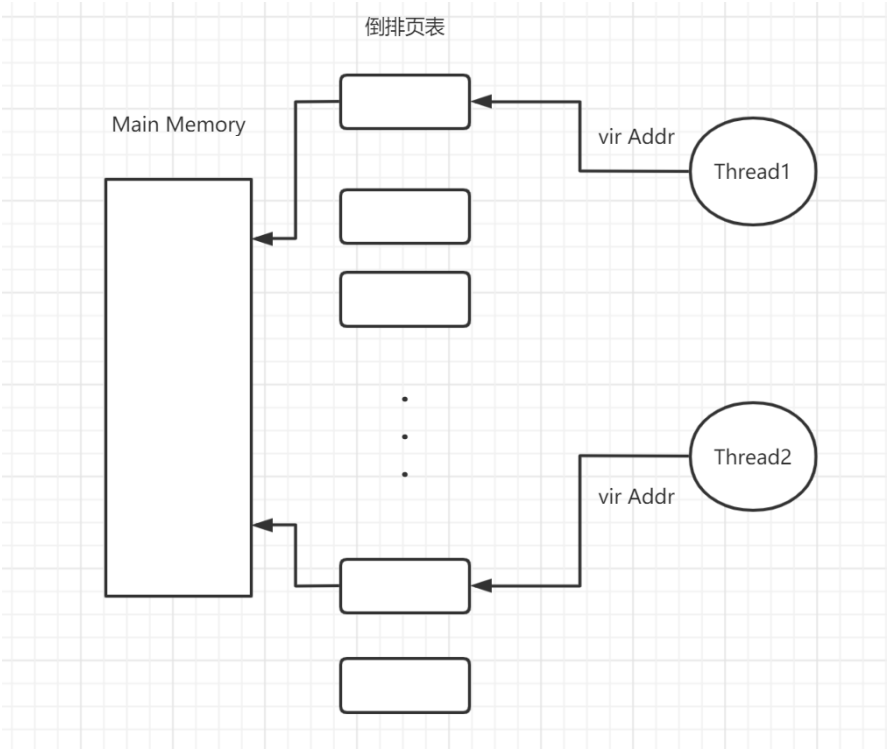


Challenges

Challenge1 倒排页表

倒排页表取消了每个进程的页目录，使得整个系统只存在一个页目录，这个页目录记录了每个物理页框对应的虚拟地址。在用户程序运行的过程中，会检查页目录中的每一项，看看需要的虚拟地址有无对应的物理地址转换项。为了支持多道程序，位页表项增加了 thread 位，这个位标志着该页框记录的是哪个线程的内存转换关系。在遍历页目录的时候，也需要比较该页是否属于该线程。

根据物理空间大小，分配了 32 个物理页框。置换算法使用最简单的 FIFO 算法。在 machine 类中维护一个页指针，指向每次缺页需要换下内存的页。



代码修改：

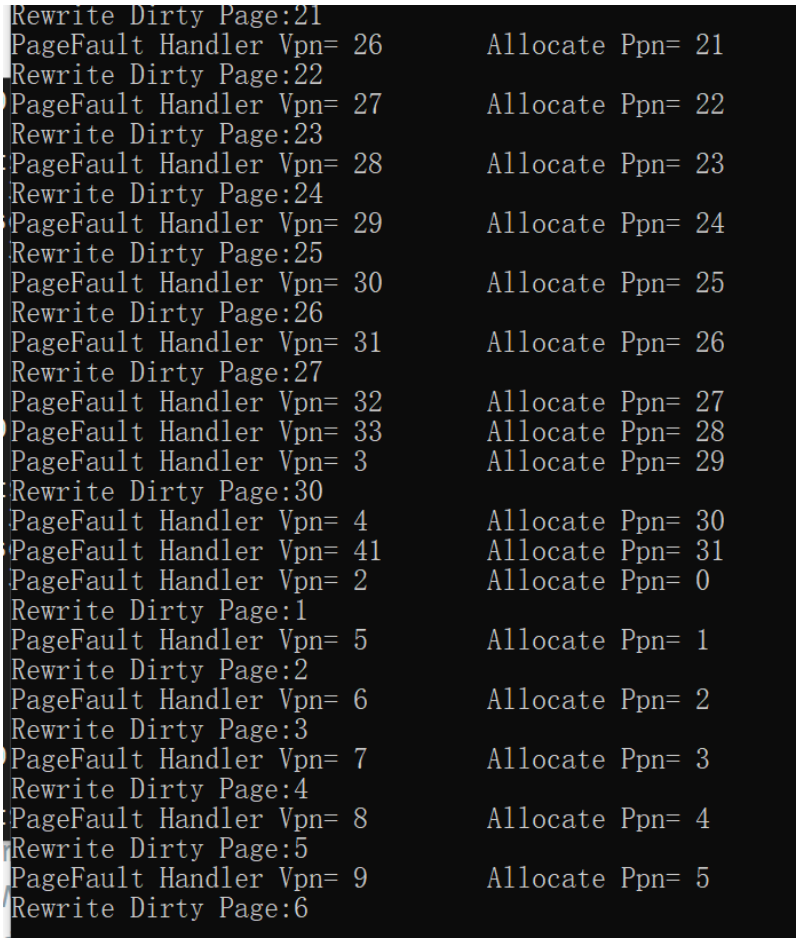
Machie.h//增加 FIFO 指针

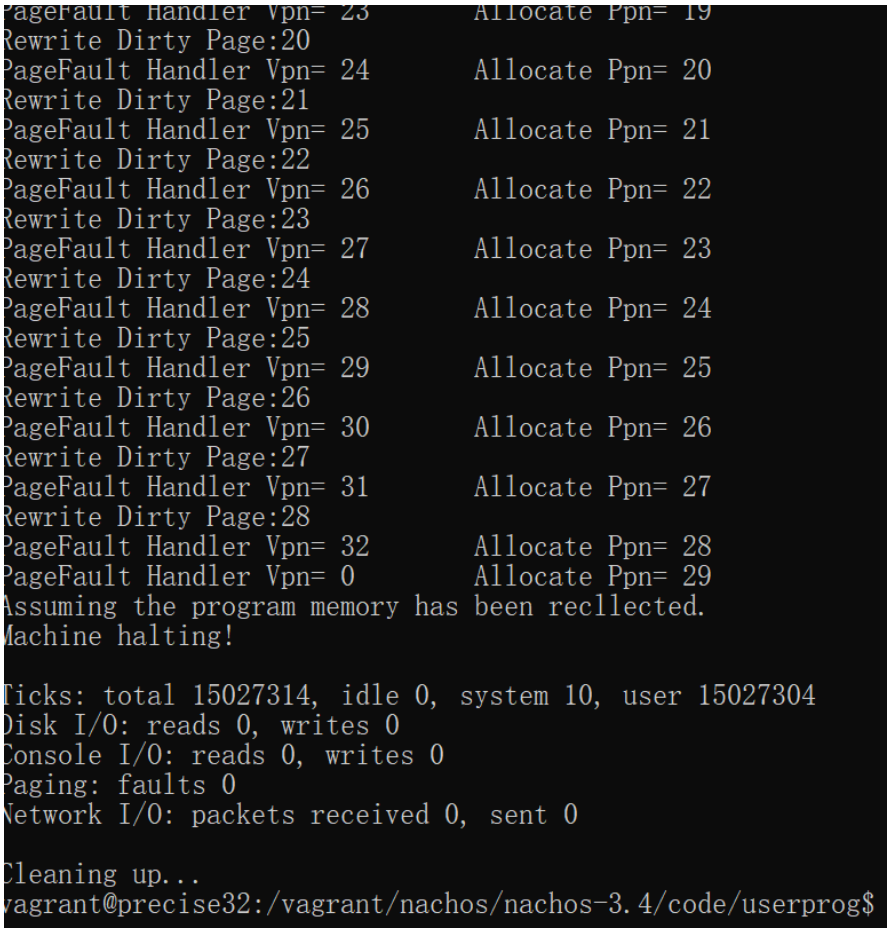
exception.cc

exceptionHandler()//修改页表缺页处理机制支持多张页表

Bitmap.cc

Find()//增加 FIFO 机制


Rewrite Dirty Page:21
PageFault Handler Vpn= 26 Allocate Ppn= 21
Rewrite Dirty Page:22
PageFault Handler Vpn= 27 Allocate Ppn= 22
Rewrite Dirty Page:23
PageFault Handler Vpn= 28 Allocate Ppn= 23
Rewrite Dirty Page:24
PageFault Handler Vpn= 29 Allocate Ppn= 24
Rewrite Dirty Page:25
PageFault Handler Vpn= 30 Allocate Ppn= 25
Rewrite Dirty Page:26
PageFault Handler Vpn= 31 Allocate Ppn= 26
Rewrite Dirty Page:27
PageFault Handler Vpn= 32 Allocate Ppn= 27
PageFault Handler Vpn= 33 Allocate Ppn= 28
PageFault Handler Vpn= 3 Allocate Ppn= 29
Rewrite Dirty Page:30
PageFault Handler Vpn= 4 Allocate Ppn= 30
PageFault Handler Vpn= 41 Allocate Ppn= 31
PageFault Handler Vpn= 2 Allocate Ppn= 0
Rewrite Dirty Page:1
PageFault Handler Vpn= 5 Allocate Ppn= 1
Rewrite Dirty Page:2
PageFault Handler Vpn= 6 Allocate Ppn= 2
Rewrite Dirty Page:3
PageFault Handler Vpn= 7 Allocate Ppn= 3
Rewrite Dirty Page:4
PageFault Handler Vpn= 8 Allocate Ppn= 4
Rewrite Dirty Page:5
PageFault Handler Vpn= 9 Allocate Ppn= 5
Rewrite Dirty Page:6



内容三：遇到的困难以及解决方法

困难 1 Makefile 编译过程

由于需要频繁开关 TLB，所有需要有时候开启 USE_TLB 机制，这些代码包括自己添加的 TLB 程序都需要条件编译。实际上有时候有的文件没有被修改，而仅仅修改了 makefile，文件并不会重新编译。这就导致了有时候会开启 TLB，有时候没开启，有的地方开启了，有的地方没开启。后来多次实验终于发现了这个特点，每次编译的时候，需要编译的文件无论有没有修改，都会加一个无关紧要的换行。

困难 2 Debug 过程

内存部分的 Debug 看不见摸不着的时候最痛苦，因为不知道文件装进内存哪里出错了，物理地址都是自己定义的，故而相同的程序运行可能会取不同的物理地址。出错的时候 Debug 不知道如何下手。最后，我开了两个相同的 nachos，运行完全相同的程序，比较每次读写的虚拟地址和每次读写的结果，value。通过这种方式逐步比较，看到底是哪里出现问题。

内容四：收获及感想

之前一直觉得对虚拟内存的掌握还可以，但是一直到直接上手改 nachos 的时候，才发现自己很多地方其实是没有考虑周全的。这次实验不仅让我对内存机制有了具体的深入的理解，同时通过软件模拟硬件的方式，让我对计算机程序的运行过程有了更加深刻的认识。

内容五：对课程的意见和建议

希望可以有讲解部分。

内容六：参考文献

[1]《nachos 中文教程》
https://wenku.baidu.com/view/905197a9e209581b6bd97f19227916888486b90b.html
[2]《nachos 学习笔记（三）》https://blog.csdn.net/darord/article/details/83303765
[3] 《Nachos 实习——Lab2 虚拟内存实习报告》
https://blog.csdn.net/sinat_40875078/article/details/109472895?utm_source=app