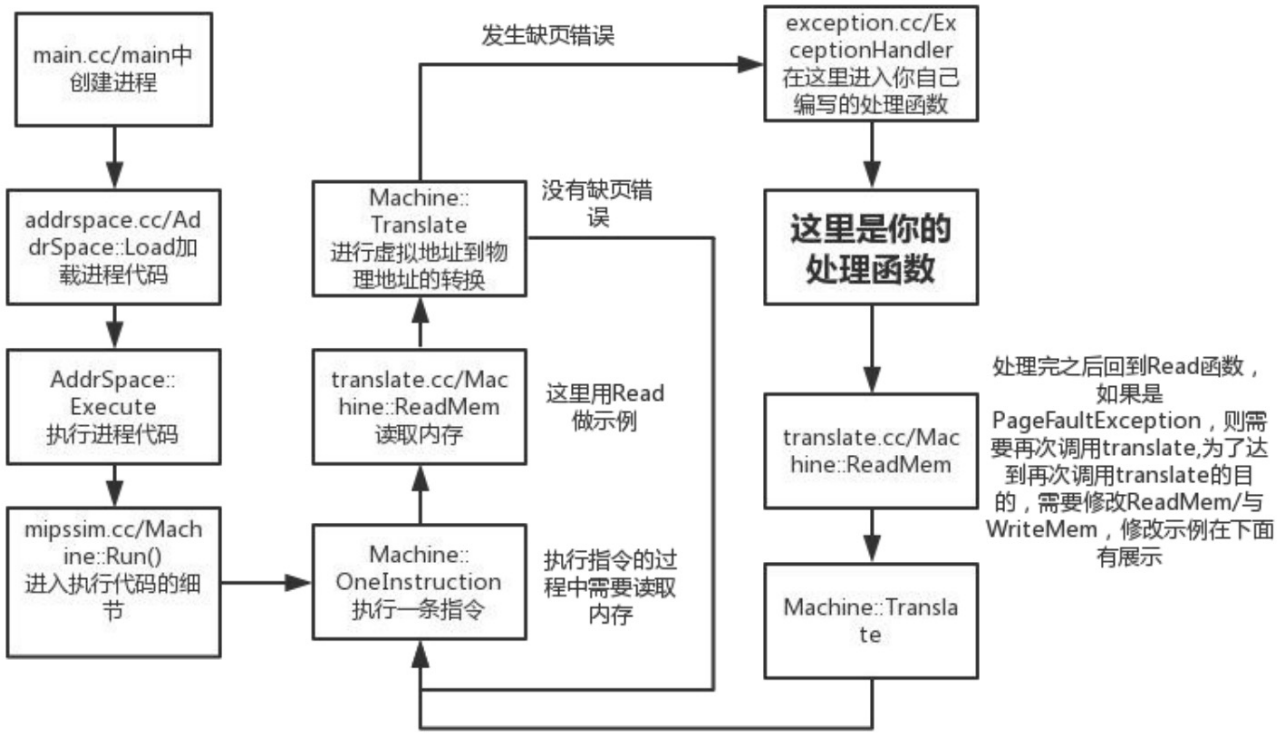
# NachOS内存管理实验讲解

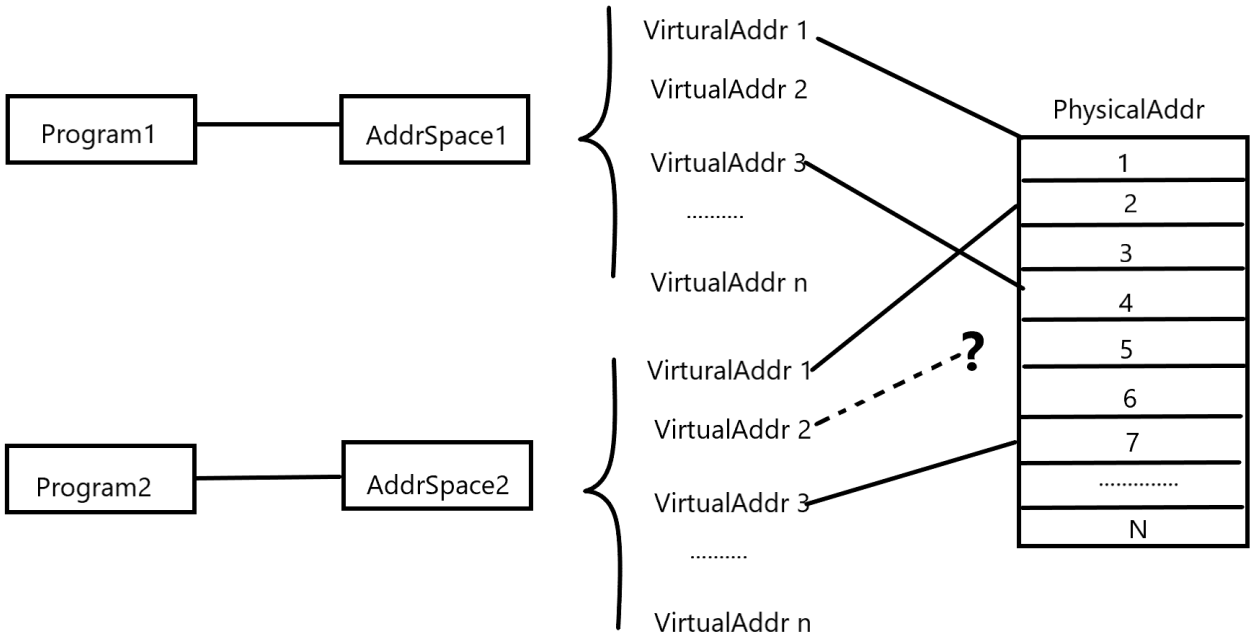
# NachOS程序的执行

- NachOS在执行用户程序的时候使用main.cc来读取命令行指令，创建程序的地址空间（AddrSpace）并且在地址空间中逐条执行指令
- 执行的指令是将程序代码从磁盘中读取后放到内存中，并且使用Program Counter来指向对应的位置以便操作系统进行执行



# 地址空间和虚拟内存

- 实现多道程序的关键在于：多个程序都使用自己地址空间中的虚拟地址而不使用物理地址，虚拟地址需要由操作系统映射为物理地址后才能执行对应的操作。在A程序运行时，B程序可能部分还存在于内存中
- 由于物理内存的大小限制，因此不是每一个地址空间中的每一个虚拟地址都有一个对应的物理地址，但是在没有使用这个虚拟地址的时候，操作系统并不关心这个虚拟地址是否有对应的物理地址
- 但是如果需要的虚拟地址在内存中没有对应的物理地址，此时就会发生缺页(Page Fault)。因此我们可以得知，缺页是在读取内存(readMem)和写入内存(writeMem)的时候，转换地址(Translate)的失败时发生的



# 三个页表：程序页表，虚拟页表和物理页表

- 程序页表指的是地址空间内，虚拟页到物理页的转换规则表。这个表在本地址空间对应的程序被加载时，读入到Machine中作为虚拟页表使用

- 虚拟页表指的是当前执行的程序地址空间中，虚拟地址到物理地址的转换规则表。执行不同的程序，使用不同的地址空间，当前的虚拟页表就不同。

- 物理页表指的是真实的全局物理页分配表，用于表示当前物理页是否已分配。我的实现方式是：由于物理页可能被多个程序地址空间的虚拟页占有，因此既需要记录当前物理页是否被分配，还需要记录占有这个页是哪一个地址空间的虚拟页。因此我的全局页表实现中，以物理页为下标，记录占有物理页的虚拟页号和地址空间中对应的程序页表指针。（定义在Machine.h中）

# 全局物理页表的定义和声明

```
91 //全局页表数据项
92 class GlobalEntry{
93         public:
94                 //引用本物理页的虚拟页号
95                 int VirNum=-1;
96                 //使用的时间戳
97                 long int useStamp = 0;
98                 //目前为止，引用本物理页的虚拟页号对应的地址空间的页表，用于在置换时将原虚拟页置为无效和读取对应信息
99                 TranslationEntry* RefPageTable=NULL;
00                 void print();
01 };
```

Machine.h中

Machine.h中

```
147 //全局页表
148     GlobalEntry* GlobalPageTable;
149     TranslationEntry *tlb;              // this pointer should be considered
150                                         // "read-only" to Nachos kernel code
```

# TranslationEntry的修改

- 为了能够在缺页中断处理中，读取其对应的程序的名称以打开磁盘文件进行IO操作，需要记录每一个地址空间的文件名称。

- 此名称定义在页表项，translate.h的TranslationEntry中，并且在AddrSpace的Load中初始化这个文件名称。

Translate.h中

```
30 class TranslationEntry {
31   public:
32     int virtualPage;     // The page numbe
33     int physicalPage;    // The page numbe
34                          //  start of "mai
35     bool valid;          // If this bit is
36                          // (In other word
37     bool readOnly;       // If this bit is
38                          // to modify the
39     bool use;            // This bit is se
40                          // page is refere
41     bool dirty;          // This bit is se
42                          // page is modifi
43
44 //添加一个用于存储加载程序的程序名称
45     char* DiskFile;
46 };
```

# AddrSpace程序页表的修改

- NachOS中的地址空间的实现是通过类AddrSpace。初始时，系统默认虚拟页与物理页一一对应，同时在申请内存时会清空整个内存。要支持多道程序，必须在加载程序的时候才动态的分配物理页。因此，将构造器中的语句注释掉，同时将程序页表的初始化放在Load方法中，同时，使用Machine中的定义方法findFreeFrame来找到物理页。

AddrSpace.cc中

```
143     cout << "loading program " << fileName << endl;
144     pageTable = new TranslationEntry[NumPhysPages];
145     //本来应该是i < numPages，但是由于程序所用的page过少，如果按需分配将会导致无法测试缺页
146     //因此这里改为i < NumPhysPages， 即有多少个分配多少个，可以导致足够的缺页数量进行测试
147     for (int i = 0; i < NumPhysPages; i++) {
148         pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
149         pageTable[i].physicalPage = kernel->machine->findFreeFrame(i, pageTable);     //修改为寻找空闲的Frame
150         pageTable[i].valid = TRUE;
151         pageTable[i].use = FALSE;
152         pageTable[i].dirty = FALSE;
153         pageTable[i].readOnly = FALSE;
154         //存储本程序加载的程序名称，用于在缺页中写回页面
155         pageTable[i].DiskFile = fileName;
156     }
```

# FindFreeFrame方法的声明和实现

- 为此，需要在Machine.h中定义在全局物理页表中寻找空闲页的方法findFreeFrame。声明和实现如下：

Machine.h中

```
146         //寻找空闲的物理页
147     int findFreeFrame(int, TranslationEntry*);
```

Machine.cc中

```
226 //寻找物理内存内的空闲Frame，若没有则返回-1
227 int Machine::findFreeFrame(int virAddr, TranslationEntry* ref){
228         cout << "trying to find a free frame..." << endl;
229         for (int i=0; i < NumPhysPages; i++){
230                 //如果某物理页引用指向了了某一个地址空间，则代表该页上有数据，非空闲
231                 if (GlobalPageTable[i].RefPageTable==NULL){
232                         cout << "finding memory frame " << i << " !" << endl;
233                         //找到可用的空闲frame后，更新全局的页表
234                         GlobalPageTable[i].VirNum = virAddr;
235                         GlobalPageTable[i].RefPageTable = ref;
236                         GlobalPageTable[i].useStamp = 0;
237                         return i;
238                 }
239         }
240         cout << "no free frame found and return -1" << endl;
241         return -1;
242 }
```

# AddrSpace的地址转换修改

- 观察Load的代码，发现在后面将会从磁盘中读取可执行文件的代码(code)，数据(initData)和只读数据(readonlyData)等信息，但是其使用的虚拟地址按照了原本的定义与物理地址一一对应了。因此需要将虚拟地址按照程序页表转换为物理地址才行（前提是该虚拟地址对应的虚拟页分配到了物理页）
- 转换公式见右图的注释

原、

```
then, copy in the code and data segments into memory
Note: this code assumes that virtual address = physical address
 if (noffH.code.size > 0) {
     DEBUG(dbgAddr, "Initializing code segment.");
     DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
     executable->ReadAt(
             &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
                     noffH.code.size, noffH.code.inFileAddr);
 }
 if (noffH.initData.size > 0) {
```

AddrSpace.cc的Load中

```
 if (noffH.code.size > 0) {
     DEBUG(dbgAddr, "Initializing code segment.");
     DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
     //逐字节读入代码，但是需要把虚拟地址通过页表转化为物理地址，以下同
     //转换公式：virtualAddr/PageSize得到虚拟页号，利用该虚拟页号查页表得到物理页号
     //再利用物理页号*PageSize得到对应物理页起始地址，再加上virtualAddr%PageSize得到页内偏移得到物理地址
     for (int i=0; i < noffH.code.size; i++){
             vaddr = noffH.code.virtualAddr+i;
             add = pageTable[vaddr/PageSize].physicalPage*PageSize + vaddr%PageSize;
             //只有分配到物理页的数据才会读入
             if (pageTable[vaddr/PageSize].physicalPage >= 0){
                     executable->ReadAt(&(kernel->machine->mainMemory[add]), 1, noffH.code.inFileAddr+i);
             }
     }
 */

if (noffH.initData.size > 0) {
     DEBUG(dbgAddr, "Initializing data segment.");
     DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
     for (int i=0; i < noffH.initData.size; i++){
             vaddr = noffH.initData.virtualAddr+i;
             add = pageTable[vaddr/PageSize].physicalPage*PageSize + vaddr%PageSize;
             if (pageTable[vaddr/PageSize].physicalPage >= 0){
                     executable->ReadAt(&(kernel->machine->mainMemory[add]), 1, noffH.initData.inFileAddr+i);
             }
     }/*


if (noffH.readonlyData.size > 0) {
     DEBUG(dbgAddr, "Initializing read only data segment.");
     DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
     for (int i=0; i < noffH.readonlyData.size; i++){
             vaddr = noffH.readonlyData.virtualAddr+i;
             add = pageTable[vaddr/PageSize].physicalPage*PageSize + vaddr%PageSize;
             if (pageTable[vaddr/PageSize].physicalPage >= 0){
                     executable->ReadAt(&(kernel->machine->mainMemory[add]), 1, noffH.readonlyData.inFileAddr+i);
             }
     }
```

# AddrSpace的程序元信息的保存

- 从上面的LOAD函数的代码我们知道了加载一个程序需要知道其代码的起始虚拟地址，大小和在可执行文件中的位置等信息，称为元信息metadata。因此，为了方便在处理缺页中断时读取缺页的对应数据，因此需要保存当前加载程序的元信息。

- 在AddrSpace.h中定义FileAddr，在load方法中对其进行初始化

AddrSpace.h中

```
173     //程序元信息
174     int* FileAddr;
```

AddrSpace.cc的Load中

```
162     //存储本程序的元信息
163     FileAddr = new int[9];
164     FileAddr[0] = noffH.code.virtualAddr;
165     FileAddr[1] = noffH.code.size;
166     FileAddr[2] = noffH.code.inFileAddr;
167     FileAddr[3] = noffH.initData.virtualAddr;
168     FileAddr[4] = noffH.initData.size;
169     FileAddr[5] = noffH.initData.inFileAddr;
170     FileAddr[6] = noffH.readonlyData.virtualAddr;
171     FileAddr[7] = noffH.readonlyData.size;
172     FileAddr[8] = noffH.readonlyData.inFileAddr;
173
174     cout << "*************** Page Table ***************" << endl;
175     //kernel->machine->printPt();
176     for (int i=0; i < PageSize; i++){
177         cout << "virtual page "<< pageTable[i].virtualPage << " physical page: " << pageTable[i].physicalPage << endl;
178     }
```

# AddrSpace析构函数的修改

- 为了在程序退出时能够正确释放其占有的所有物理页，因此在析构函数中释放所有当前程序页表中虚拟页对应的存在的物理页

```
82 AddrSpace::~AddrSpace()
83 {
84     for (int i=0; i < NumPhysPages; i++){
85         if (pageTable[i].physicalPage != -1){
86                 //只清除自己的地址空间中占用的物理内存页
87                 bzero(&(kernel->machine->mainMemory[pageTable[i].physicalPage*PageSize]), PageSize);
88                 //同时将全局页表的引用改为null，使得其可以作为freeframe被找到
89                 kernel->machine->GlobalPageTable[pageTable[i].physicalPage].RefPageTable = NULL;
90                 kernel->machine->GlobalPageTable[pageTable[i].physicalPage].VirNum = -1;
91         }
92     }
93     delete pageTable;
94 }
```

# AddrSpace中保存和恢复现场

- 由于程序的执行切换会导致当前程序停止运行，换成其他程序运行，为了能够恢复程序，需要保存寄存器中的值并在恢复现场时恢复寄存器值

- 因此在AddrSpace.h中声明s_reg用于暂存寄存器值，在AddrSpace的restoreState和saveState方法中恢复和暂存寄存器的值

AddrSpace.h中

```
// address space

    //寄存器的暂存
    int* s_reg;
```

AddrSpace.cc中

```
void AddrSpace::SaveState()
{
        cout << "saving state!" << endl;
        //暂存寄存器内容
        for (int i=0; i < NumTotalRegs; i++){
                s_reg[i] = kernel->machine->ReadRegister(i);
        }
        cout << "Successfully saving state!" << endl;
}
```

AddrSpace.cc中

```
void AddrSpace::RestoreState()
{
    cout << "restoring state!" << endl;
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
    //也要写入加载程序的元信息
    kernel->machine->FileAddr = FileAddr;
    //将暂存内容写回寄存器
    for (int i=0; i < NumTotalRegs; i++){
        kernel->machine->WriteRegister(i, s_reg[i]);
    }
    cout << "successfully restoring state!" << endl;
}
```

```
void AddrSpace::SaveState(
{}
```

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

# 陷阱！

AddrSpace.cc中

- 观察AddrSpace的代码，发现在Execute方法中，会先执行InitRegister初始化寄存器的方法，再执行restoreState方法来恢复现场使得当前程序开始运行。但是由于我们已经在restoreState方法中添加了恢复寄存器的指令，使得之前的初始化直接无效，让PC，Stack等寄存器陷入混乱使得程序崩溃

- 因此需要将此处的restoreState注释掉，改为只赋值程序页表，页数量和程序元信息的指令。

```
267 void
268 AddrSpace::Execute()
269 {
270     cout << "**************Execute!***************" << endl;
271     kernel->currentThread->space = this;
272
273     this->InitRegisters();              // set the initial register values
274
275     //不能直接调用restore方法，因为会覆盖掉寄存器的初始化
276     //应该将pageTable等信息直接赋值给machine中的pageTable等变量
277     //this->RestoreState();             // load page table register
278     kernel->machine->pageTable = pageTable;
279     kernel->machine->pageTableSize = numPages;
280     kernel->machine->FileAddr = FileAddr;
281
282     kernel->machine->Run();             // jump to the user progam
283
284     ASSERTNOTREACHED();                 // machine->Run never returns;
285                                         // the address space exits
286                                         // by doing the syscall "exit"
287 }
```

# LU算法的实现

- LRU是最近最少使用原则，本应该在每个Page使用的时候都做上记录以便记录下页的调用序列。但是为了实现方便，我实现的是一个较为简单的LRU的变种：LU，即最少使用原则。

- 在machine.h中定义findFrameByLU()方法，同时在machine.cc中实现。该算法会在一个页被访问时（Translate后的物理地址对应的页）将useStamp加1，代表一次访问。选择全局页表中使用最少，即useStamp最小的有效物理页返回下标。

```
int Machine::findFreeByLU(){
        int mi = 0;
        for (int i=0; i < NumPhysPages; i++){
                ASSERT(GlobalPageTable[i].RefPageTable!=NULL);
                //比较时间戳
                if (GlobalPageTable[i].useStamp < GlobalPageTable[mi].useStamp)
                        mi = i;
        }
        return mi;
}
```

Translate.cc中的Translate方法的最后

```
//每次该地址被访问，都将时间戳加1
GlobalPageTable[*physAddr/PageSize].useStamp += 1;
ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
DEBUG(dbgAddr, "phys addr = " << *physAddr);
cout << "translating from virtual address " << virtAddr << " to " << *physAddr << endl;
return NoException;
```

# 缺页的检测

- 缺页发生在虚拟地址转换为物理地址的时候。当程序发现虚拟地址在程序页表中没有对应的物理页(physicalPage=-1)，或者对应的物理页已经失效(valid = FALSE)，返回一个缺页异常给Machine进行raise操作，将控制流引向exception.cc中的exceptionHandler

- 需要将缺页异常的检测进行一些修改

translate.cc的Translate中

```
225         //如果地址空间页表中，虚拟页对应项被置为了valid或者没有分配空的物理页，则说明缺页
226         else if (!pageTable[vpn].valid || pageTable[vpn].physicalPage==-1) {
227             DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
228             //cout << "valid bit is FALSE or -1 memory address and cause a page fault at vpn: "
    << vpn << endl;
229             return PageFaultException;
230         }
```

# writeMem与readMem的修改

- 原本在writeMem写内存和readMem读内存时使用的转换操作Translate发生缺页时，原程序默认将会退出方法的执行返回FALSE。由于我们的程序能够处理缺页中断，使得虚拟地址能够正确的对应到物理地址，因此需要修改程序，使得不直接返回FALSE，而是再次Tanslate使得程序继续执行。

translate.cc中

```
86 Machine::ReadMem(int addr, int size, int *value)
87 {
88     cout << "Reading virtual address " << addr << endl;
89     int data;
90     ExceptionType exception;
91     int physicalAddress;
92
93     DEBUG(dbgAddr, "Reading VA " << addr << ", size " << size);
94
95     exception = Translate(addr, &physicalAddress, size, FALSE);
96     if (exception != NoException) {
97         RaiseException(exception, addr);
98         //如果是缺页，则抛出缺页异常以后再度translate
99         if (exception == PageFaultException)
100                Translate(addr, &physicalAddress, size, FALSE);
101        else return FALSE;
102    }
```

translate.cc中

```
139 bool
140 Machine::WriteMem(int addr, int size, int value)
141 {
142     cout << "Writing to virtual address " << addr << endl;
143     ExceptionType exception;
144     int physicalAddress;
145
146     DEBUG(dbgAddr, "Writing VA " << addr << ", size " << size << ", value " << value);
147
148     exception = Translate(addr, &physicalAddress, size, TRUE);
149     if (exception != NoException) {
150         RaiseException(exception, addr);
151         //如果是缺页，则抛出缺页异常以后再度translate
152         if (exception == PageFaultException)
153                 Translate(addr, &physicalAddress, size, TRUE);
154         else return FALSE;
```

# 缺页异常的处理——读取缺页地址

- 在exception.cc的ExceptionHandler中添加case PageFaultException。

- 首先，从寄存器BadVAddr中读取发生缺页的虚拟地址virAddr，计算其对应的虚拟页vpn号和偏移量offset

```
    break;

case PageFaultException:
    int virAddr, vpn, offset, phy, vir;
    //磁盘文件
    OpenFile* out_file, *in_file;
    //待读取的虚拟entry和待替换的entry
    virAddr = kernel->machine->ReadRegister(BadVAddrReg);
    //虚拟页号
    vpn = virAddr/PageSize;
    //偏移量
    offset = virAddr % PageSize;
    cout << "page " << vpn << " offset " << offset << " tr
```

# 缺页异常的处理——找到物理页

- 优先查看当前是否有空闲物理页可以分配给虚拟页，调用findFreeFrame方法，并判断返回值是否有效

- 如果找到了空闲页，则直接使用。否则，需要使用置换算法LU找到最少使用的页作为牺牲页。调用findFreeByLU方法找到牺牲页的下标，根据其脏位dirty判断是否有必要将其写回原磁盘文件

```
        //实际的替换物理页号
        phy = kernel->machine->findFreeFrame(vpn, kernel->machine->pageTable);

        //没有空闲的页了
        if (phy == -1){
                //待替换的全局页表中的物理页号
                phy = kernel->machine->findFreeByLU();
                //待替换的全局页表中的原引用的虚拟页号
                vir = kernel->machine->GlobalPageTable[phy].VirNum;

                cout << "no free frame and choose virtual page " << vir << " memory frame " << phy << " as replacement by LU!" <<
endl;


                //读取程序的名称便于将程序从磁盘读入到内存中
                char* fileName = kernel->machine->GlobalPageTable[phy].RefPageTable[vir].DiskFile;

                out_file = kernel->fileSystem->Open(fileName);
                cout << "previous frame will be write back to " << fileName << endl;

                ASSERT(out_file!=NULL)
                //如果是该Frame被写过，才会写回disk
                if (kernel->machine->GlobalPageTable[phy].RefPageTable[vir].dirty){
                        out_file->WriteAt(&(kernel->machine->mainMemory[phy*PageSize]), PageSize, kernel->machine-
>GlobalPageTable[phy].RefPageTable[vir].virtualPage*PageSize);
                }
                cout << "successfully write frame " << phy << " back to disk!" << endl;
                delete out_file;
        }
```

# 缺页异常的处理——数据写入内存

- 接着读取待读入页的程序元信息，根据元信息，逐字节判断数据位于代码区还是数据区并根据区域将数据从磁盘读取到内存中

```
//else cout << "free memory frame " << phy << " is used to tackle page fault!" << endl;
in_file = kernel->fileSystem->Open(kernel->machine->pageTable[vpn].DiskFile);
cout << "Opening " << kernel->machine->pageTable[vpn].DiskFile << " !" << endl;
ASSERT(in_file!=NULL)

//读取程序的元信息
int cVir, cSize, cIn, dVir, dSize, dIn, roVir, roSize, roIn;
cVir = kernel->machine->FileAddr[0];
cSize = kernel->machine->FileAddr[1];
cIn = kernel->machine->FileAddr[2];
dVir = kernel->machine->FileAddr[3];
dSize = kernel->machine->FileAddr[4];
dIn = kernel->machine->FileAddr[5];
roVir = kernel->machine->FileAddr[6];
roSize = kernel->machine->FileAddr[7];
roIn = kernel->machine->FileAddr[8];

//逐字节将虚拟地址对应的页的内容从磁盘写入内存中找到的物理页中
for (int i=0; i < PageSize; i++){
        int vAddr = vpn*PageSize + i;
        int pAddr = phy*PageSize + i;
        //如果该数据位于代码段
        if(vAddr >= cVir && vAddr < (cVir+cSize))
        {
                //cout << "********code fault!********" << endl;
                in_file->ReadAt(&(kernel->machine->mainMemory[pAddr]),1,cIn+vAddr-cVir); // cIn+vAddr-cVir
        }//如果在数据段
        else if(vAddr>=dVir && vAddr<(dVir+dSize))
         {
                //cout << "********data fault!********" << endl;
                in_file->ReadAt(&(kernel->machine->mainMemory[pAddr]),1,dIn+vAddr-dVir); //-dVir
        }
        //如果在只读数据段
        else if (vAddr >= roVir && vAddr < (roVir+roSize)){
                in_file->ReadAt(&(kernel->machine->mainMemory[pAddr]), 1, roIn+vAddr-roVir);
        }
        else {
                //执行到这里，说明目前发生的缺页的原本目的并非读入程序的代码和数据，而是开辟新的空间用于存储
                /*
```

# 缺页异常的处理——更新页表

如果是找的牺牲页，则需要将原引用这个物理页的程序页表中对应虚拟页置为无效

更新全局页表，使得时间戳为0，虚拟页号为vpn，引用的程序页表为当前运行程序的页表

更新程序页表，将物理页置为找到的空闲页或者牺牲页，然后初始化valid，dirty等参数

```
//更新全局页表:将旧的的物理页对应的全局页表项删除，同时更新缺页的虚拟地址信息
cout << "global update ... virtual page " << vir << ", physical page " << ph
//因为该原页的物理地址被占，因此原引用地址空间的页表对应的虚拟页失效
if (kernel->machine->GlobalPageTable[phy].RefPageTable != NULL)
        kernel->machine->GlobalPageTable[phy].RefPageTable[vir].valid = FALSE;
//更新的引用该物理页的虚拟页号
kernel->machine->GlobalPageTable[phy].VirNum = kernel->machine->pageTable[vpn].virtualPage;
//更新的引用该物理页的地址空间的页表
kernel->machine->GlobalPageTable[phy].RefPageTable = kernel->machine->pageTable;
kernel->machine->GlobalPageTable[phy].useStamp = 0;
cout << "new global entry : " << phy << " , " << kernel->machine->pageTable[vpn].virtualPage << endl;

//更新地址空间的程序页表
kernel->machine->pageTable[vpn].physicalPage = phy;
kernel->machine->pageTable[vpn].valid = TRUE;
kernel->machine->pageTable[vpn].use = FALSE;
kernel->machine->pageTable[vpn].dirty = FALSE;
kernel->machine->pageTable[vpn].readOnly = FALSE;
delete in_file;
cout << "page fault exception ends!new GlobalPageTable:" << endl;
```

# 测试程序的编写

- 使用add.c文件进行测试。测试前，先修改main.cc，使得其声明两个程序空间，只执行第二个程序。
- 修改add.c，使其更复杂一些
- 修改ksyscall.h中的SysAdd方法，使其能够打印出调试信息

main.cc中

```
//按ppt上的，申请两个地址空间，但只运行第二个
if (userProgName != NULL) {
  AddrSpace *space1 = new AddrSpace;
  AddrSpace* space2 = new AddrSpace;
  ASSERT(space1 != (AddrSpace *)NULL);
  ASSERT(space2 != (AddrSpace *)NULL);
  if (space1->Load(userProgName)) {  // load the program into the space
    if (space2->Load(userProgName)){
          space2->Execute();              // run the program
          ASSERTNOTREACHED();             // Execute never returns
    }
  }
}
```

test/add.c中

```
#include "syscall.h"

//修改add测试程序
int
main()
{
  int result, another;

  result = Add(1, 2);
  another = 3;
  result = Add(result, another);

  Halt();
  /* not reached */
}
```

usrprog/ksyscall.h中

```
//打印add系统调用的结果以便显示

int SysAdd(int op1, int op2)

{
  cout << "**********系统调用结果**********" << endl;
  cout << op1 << " + " << op2 << " = " << op1+op2 << endl;
  return op1 + op2;

}
```

# 编译代码并执行测试

- 先进入NachOS-4.1/code/build.linux中依次执行make clean和make

- 成功后，进入到code/test中，依次执行make clean和make

- 返回build.linux，执行./nachos –x ../test/add.noff，查看结果

# 测试结果

- 由于程序的实现中，一个地址空间就将所有的物理页全部分配完了（按理应该按需分配，但是为了测试缺页），因此第二个程序必定会出现缺页中断

- 在虚拟地址0发现没有对应的物理页时无法转换时，引发缺页中吨（第一个红线）。由于LU算法找到了牺牲页（第二个红线），因此置换该页。最后，更新页表（第三个红线）
- 。更新后的全局页表如右图，可以发现第一个物理页的程序页表引用已经改为了第二个程序

```
Reading virtual address 0
virtual address 0 with page 0 offset 0 triggering a page fault!

trying to find a free frame...
no free frame found and return -1
no free frame and choose virtual page 0 memory frame 0 as replacement by LRU!
previous frame will be write back to ../test/add.noff
successfully write frame 0 back to disk!
Opening ../test/add.noff !
successfully writing frame 0 to memory from disk!
global update: previous virtual page 0, physical page 0 referencing is invalid!
new global entry : 0 , 0
```

```
**************全局页表如下**************
phy: 0 vir: 0 ref: 0x8798300
phy: 1 vir: 1 ref: 0x8797ac0
phy: 2 vir: 2 ref: 0x8797ac0
phy: 3 vir: 3 ref: 0x8797ac0
phy: 4 vir: 4 ref: 0x8797ac0
phy: 5 vir: 5 ref: 0x8797ac0
phy: 6 vir: 6 ref: 0x8797ac0
phy: 7 vir: 7 ref: 0x8797ac0
phy: 8 vir: 8 ref: 0x8797ac0
phy: 9 vir: 9 ref: 0x8797ac0
phy: 10 vir: 10 ref: 0x8797ac0
phy: 11 vir: 11 ref: 0x8797ac0
phy: 12 vir: 12 ref: 0x8797ac0
phy: 13 vir: 13 ref: 0x8797ac0
phy: 14 vir: 14 ref: 0x8797ac0
phy: 15 vir: 15 ref: 0x8797ac0
phy: 16 vir: 16 ref: 0x8797ac0
phy: 17 vir: 17 ref: 0x8797ac0
phy: 18 vir: 18 ref: 0x8797ac0
phy: 19 vir: 19 ref: 0x8797ac0
phy: 20 vir: 20 ref: 0x8797ac0
phy: 21 vir: 21 ref: 0x8797ac0
phy: 22 vir: 22 ref: 0x8797ac0
phy: 23 vir: 23 ref: 0x8797ac0
phy: 24 vir: 24 ref: 0x8797ac0
```

# 测试结果

- 而之前缺页的虚拟地址0在处理完缺页中断后也正确的被转换为了物理地址0

- 其他缺页的处理结果类同

```
translating from virtual address 0 to 0
```

```
Reading virtual address 384
virtual address 384 with page 3 offset 0 triggering a page fault!

trying to find a free frame...
no free frame found and return -1
no free frame and choose virtual page 1 memory frame 1 as replacement by LRU!
previous frame will be write back to ../test/add.noff
successfully write frame 1 back to disk!
Opening ../test/add.noff !
successfully writing frame 3 to memory from disk!
global update: previous virtual page 1, physical page 1 referencing is invalid!
new global entry : 1 , 3
page fault exception ends!new GlobalPageTable:
**************全局页表如下**************
phy: 0 vir: 0 ref: 0x8798300
phy: 1 vir: 3 ref: 0x8798300
phy: 2 vir: 2 ref: 0x8797ac0
phy: 3 vir: 3 ref: 0x8797ac0
phy: 4 vir: 4 ref: 0x8797ac0
phy: 5 vir: 5 ref: 0x8797ac0
phy: 6 vir: 6 ref: 0x8797ac0
phy: 7 vir: 7 ref: 0x8797ac0
phy: 8 vir: 8 ref: 0x8797ac0
phy: 9 vir: 9 ref: 0x8797ac0
phy: 10 vir: 10 ref: 0x8797ac0
phy: 11 vir: 11 ref: 0x8797ac0
```

```
translating from virtual address 384 to 128
```

# 测试结果

- 最终的测试结果正确，程序页正常的终止，测试通过

```
Reading virtual address 36
translating from virtual address 36 to 36
************************系统调用结果****************
1 + 2 = 3
Reading virtual address 40
translating from virtual address 40 to 40
Reading virtual address 44
translating from virtual address 44 to 44
Reading virtual address 412
translating from virtual address 412 to 156
Reading virtual address 416
translating from virtual address 416 to 160
Reading virtual address 420
translating from virtual address 420 to 164
Reading virtual address 32
translating from virtual address 32 to 32
Reading virtual address 36
translating from virtual address 36 to 36
************************系统调用结果****************
3 + 3 = 6
Reading virtual address 40
translating from virtual address 40 to 40
Reading virtual address 44
```

```
Reading virtual address 424
translating from virtual address 424 to 168
Reading virtual address 428
translating from virtual address 428 to 172
Reading virtual address 16
translating from virtual address 16 to 16
Reading virtual address 20
translating from virtual address 20 to 20
Machine halting!

Ticks: total 35, idle 0, system 10, user 25
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
root@asichurter-virtual-machine:/usr/local/NachOS-4.1/code/build.linux#
```