操作系统实习Lab4--虚拟内存

Exercise2

- 当前Nachos系统的异常处理操作位与`exception.cc`文件中，在`machine.cc`中定义了多种异常类型，但是在最初的`exception.cc`中只对退出操作进行了处理，因此缺页中断是没有处理的。所以在用户程序加载时，nachos将所有的用户程序内存都分配完毕，因此不会发生缺页中断。
- 这里对于TLB miss的情况，设置如下函数处理

```
else if (which == PageFaultException)
    {
        /*
            虚拟内存缺页中断。
            1.对于TLB， 用需要替换的地址替换TLB中现有的某一项，应该查询页表找到对应项，再利用选择算法替换某一项
        */
        int badAddr = machine->ReadRegister(BadVAddrReg);
        DEBUG('a', "Page fault exception of addr %x.\n", badAddr);
        if (machine->tlb != NULL)
        { //使用tlb
            machine->replaceTlb(badAddr);
        }
    }
```

该函数位于`machine`中，作用为当发生TLB miss的情况时，从页表中加载所需的PTE进入TLB，如果需要，则进行选择替换。`replaceTlb()`函数如下：

```
/*
    @author lihaiyang
    1. 查找页表，找到对应的页表项
    2. 选择一个tlb项进行替换
*/
ExceptionType
Machine::replaceTlb(int virtAddr)
{
    unsigned int vpn, offset;
    TranslationEntry *entry;

    vpn = (unsigned)virtAddr / PageSize;
    offset = (unsigned)virtAddr % PageSize;

    if (vpn >= pageTableSize)
    {
        DEBUG('a', "virtual page # %d too large for page table size %d!\n",
              virtAddr, pageTableSize);
        return AddressErrorException;
    }
    else if (!pageTable[vpn].valid)
    {
        DEBUG('a', "virtual page # %d not valid\n", virtAddr);
        return PageFaultException;
    }
    entry = &pageTable[vpn];
```

```
    TranslationEntry *replaceEntry = selectOne(tlb, TLBSize);

    *replaceEntry = *entry;

    return NoException;
}
```

该函数首先查询页表，获得对应虚拟地址的PTE，如果找不到，则说明页表中该页无效，则需要再次抛出异常处理页表缺页。由于Nachos系统不区分TLB miss异常和页表无效异常，都是`PageFault`因此这里抛出异常其实是无法处理的，但是鉴于Nachos不同时存在TLB和页表，因此，在专注于TLB的逻辑时可以简单的将页表都加载进入内存，来避免上述情况出现，但是科学的处理还是应该区分两种异常的类型。

Exercise 3

置换算法选取了两种，分别为LRU和FIFO，LRU的核心思想是对于最近使用的数据给予保留，对于上一次使用距离最远的数据进行替换，而FIFO则是先进先出，为了进行两种算法的计数，在`TranslateEntry`结构体中加入变量`count`。这个变量在不同的情况下进行清零和累计，以用于不同的选择算法。

- 对于LRU来说每次命中给对应PTE的count 清零，并给其余PTE条目增加1，每次替换时替换count最大的PTE即可
- 对于FIFO来说，每次查找TLB都给所有PTE count值增加1， 需要替换时替换count最大的PTE。

Exercise 4

- 使用一个bool数组来表述每个物理页面的使用情况：同时，为了模拟内存交换，在machine中设置简单的链表来模拟一个磁盘，便于后续的 lazy loading和内存页面的导入导出操作。对于bool数组的操作较为简单，每次申请一个物理页面则将其对应的位置设置为`true`，表示物理页面被使用，当没有足够的物理页面时，则需要选择一个物理页面写入磁盘，即新申请一个内存块，加入`disk`列表，并在PTE中记录该内存块的内存地址，同时设置`TranslateEntry`结构体中的变量`onDisk=true`， 表示内存页面在磁盘上，结构体和相关变量如下：

```
  /*  machine.h::Machine  */
  List *disk;                       //虚拟磁盘
  bool pageUsage[NumPhysPages]; //物理页面管理bitmap

/*  translate.h: TranslateEntry*/
bool onDisk; //当前页面是否在磁盘上，和在磁盘上的地址
int diskAddr;
```

Exercise 6

- 基于上述的数据结构，页表的异常处理则相对于TLB来说更复杂一些，可以总结为如下逻辑
    1. 检查地址越界
    2. 检查该虚拟地址对应的页面是否在磁盘上，如果在磁盘上，则将其载入内存
    3. 分配一个物理页面，如果分配成功，则更新页表，反回
    4. 如果分配不成功，调用选择算法，选择页面进行替换
    5. 申请上述的虚拟磁盘空间保存需要换出的物理页内容，更亲需要换出的 PTE
    6. 更新当前PTE为对应物理页面，并拷贝磁盘数据。
       算法如下：

```
/*
```

```
    @author lihaiyang
    1. virtAddr位置未分配物理页面，选择一个空闲的物理页面分配，如果物理页面不足，
        则选择一个物理页面替换掉（当前直接报错）
    2. virtAddr位置的物理页面没有在内存中，需要从磁盘调入物理页面，
        entry中应该存放了物理磁盘的地址（此处用内存模拟磁盘，磁盘地址即为某个内存地址）
    解决：在entry中加入标识位，on disk，表示当前页表缓存在磁盘上，diskAddr表示当前页面在磁盘的位置
    在machine中创建虚拟的内存空间disk来模拟磁盘，为列表形式，忽略磁盘的数据管理等操作。
*/
ExceptionType
Machine::replacePageTable(int virtAddr)
{
    unsigned int vpn, offset;
    TranslationEntry *entry;

    vpn = (unsigned)virtAddr / PageSize;
    offset = (unsigned)virtAddr % PageSize;

    if (vpn >= pageTableSize)
    {
        DEBUG('a', "virtual page # %d too large for page table size %d!\n",
              virtAddr, pageTableSize);
        return AddressErrorException;
    }

    int *pageFromDisk = NULL;
    if (pageTable[vpn].onDisk)
    { //页面在磁盘，从磁盘重新读取页面到pageFromDisk
        for (ListElement *page = disk->getHead(); page != NULL; page = page->next)
        {
            if ((int)page == pageTable[vpn].diskAddr)
            {
                pageFromDisk = (int *)page;
                break;
            }
        }
    }
    //分配一个物理页面，并更新页表
    int pageNO = findNullPyhPage();
    if (pageNO == -1)
    { //物理页面满

        TranslationEntry *replacePage = selectOne(pageTable, pageTableSize); //寻找替换页面

        int *diskPage = new int[PageSize / 4]; //将数据拷贝存储到磁盘
        memcpy(diskPage, mainMemory + replacePage->physicalPage * PageSize, PageSize);
        disk->Append((void *)diskPage);

        replacePage->onDisk = true; //更新替换页表项
        replacePage->diskAddr = (int)diskPage;
        pageNO = replacePage->physicalPage;
    }
    pageTable[vpn].count = 0; //更新当前页表项
    pageTable[vpn].physicalPage = pageNO;
    pageTable[vpn].valid = true;
```

```
    if (pageFromDisk != NULL)
    { //拷贝磁盘数据
        memcpy(mainMemory + pageNO * PageSize, pageFromDisk, PageSize);
    }
}
```

Exercise 5 && Exercise 7

Nachos系统目前只支持单个线程存在于主存之中，其原因是在`AddrSpace::AddrSpace()`函数中，对主存进行清零，同时没有进行真正的虚拟内存的分配，只是简单的进行虚拟内存和物理内存的直接映射，即

```
/* 页表项的初始化和分配 progtest.cc*/
 for (i = 0; i < numPages; i++)
    {
        pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
        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
    }
```

多线程的支持，要求实现真正的虚拟内存机制，即申请物理页面进行映射的机制，同时，将物理页面的分配和加载分开，也就是在上面的逻辑中分配页表，并在如下加载代码的逻辑中进行磁盘地址的映射，载通过上面已经实现的异常处理机制，即可完成多线程并存和延迟加载的操作。

```
/* 代码逻辑的加载 progtest.cc*/
    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);
    }
```

因此，对上面两段代码进行修改，

- 对于页表分配阶段，只初始化页表项，设置其`valid=false`，代表该虚拟内存没有对应的物理页面，访问时缺页异常，申请物理页面。
- 对于用户程序的代码加载阶段，
    1. 假设其需要加载的代码的虚拟地址为`virAddr`， 计算该地址对应的虚拟页面号`vpn`，根据`vpn`在页表中查找页表项`PTE`，

2. 申请一个页面大小的内存（如果`PTE`中标明该条目已经在磁盘上，即多个虚拟地址在同一个内存页上，不需要重复申请），初始地址为`diskAddr`，拷贝该用户数据到该内存的指定偏移处，将`diskAddr`加入`Machine::disk`列表中，模拟为磁盘。
3. 需要注意在拷贝数据时可能超出一个页面的大小，因此需要做边界检查

修改代码如下：

- 初始化页表项：

```
  for (i = 0; i < numPages; i++)
    {
        pageTable[i].virtualPage = i;
        pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
        pageTable[i].onDisk = FALSE;       //不在磁盘上有缓存
    }

    // 不清零主存
    // zero out the entire address space, to zero the unitialized data segment
    // and the stack segment
//    bzero(machine->mainMemory, size);
```

- 读取用户数据和代码部分如下：

```
    if (noffH.code.size > 0)
    {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
              noffH.code.virtualAddr, noffH.code.size);
        /* 将其加载至虚拟磁盘 */
        unsigned codeVirAddr = (unsigned) noffH.code.virtualAddr;
        unsigned int offset = codeVirAddr % PageSize;  //偏移
        unsigned int vpn = vpn = codeVirAddr / PageSize;  //第一个虚拟页号
        int codePages = divRoundUp(noffH.code.size + offset, pageSize);    //总计
需要的页面数量
        for(int i = 0; i < codePages; i ++){

            char* diskPage = new char[pageSize];
            executable->ReadAt(diskPage + offset,
                               noffH.code.size, noffH.code.inFileAddr);
            pageTable[vpn+i].onDisk = true;
            pageTable[vpn+i].diskAddr = diskPage;
            machine->disk-Append((void*)diskPage);  //加入虚拟磁盘
            offset = 0;     //只要第一个页面有偏移
        }
    }
    if (noffH.initData.size > 0)
    {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
              noffH.initData.virtualAddr, noffH.initData.size);
              /* 将其加载至虚拟磁盘 */
        unsigned dataVirAddr = (unsigned) noffH.code.virtualAddr;
        unsigned int offset = dataVirAddr % PageSize;  //偏移
        unsigned int vpn = vpn = dataVirAddr / PageSize;  //第一个虚拟页号
        int dataPages = divRoundUp(noffH.initData.size + offset, pageSize);
//总计需要的页面数量
        for(int i = 0; i < dataPages; i ++){
```

```
            if(pageTable[vpn+i].onDisk == FALSE){
                char* diskPage = new char[pageSize];
                executable->ReadAt(diskPage + offset,
                            noffH.initData.size, noffH.initData.inFileAddr);
                pageTable[vpn+i].onDisk = true;
                pageTable[vpn+i].diskAddr = diskPage;
                machine->disk-Append((void*)diskPage);  //加入虚拟磁盘
            }else{
                executable->ReadAt(pageTable[vpn+i].diskAddr + offset,
                            noffH.initData.size, noffH.initData.inFileAddr);
            }
            offset = 0;     //只有第一个页面有偏移
        }
    }
```

Challenge 2

倒排页表，和物理内存的大小成正比，对于每个进程，其维护的也变以物理内存为索引，也就是说，在进程初始化的最初，其页表为空，每当进程访问虚拟地址`virAddr`时， 遍历其页表，查看是否有某个页表项的虚拟内存地址为`virAddr`并且其有效为`valid=TRUE`，如果有，则其物理地址`phyAddr`就是所需的物理地址。实现所需的工作如下：

1. 初始化页表时页表设置为空，或者只加载用户程序，其余页表项不加载
2. 查询页表时，需要遍历页表查看是否有有效页表项保存了对应的物理地址，没有则触发缺页异常，有则返回对应的页表项
3. 触发缺页异常时，申请新的物理页面，新建一个页表项并加入页表。
4. 修改内存管理结构，对于每个物理页面保存其当前占有的进程号，如果发生内存交换，则需要

- 倒排页表的查找算法如下：

```
/*
倒排页表查找算法
*/
TranslationEntry*
translatePageTableDes(int vpn, int offset, ExceptionType *exception, List*
pageTable){
    unsigned int virtAddr = vpn * PageSize + offset;
    // => page table => vpn is index into table
    if (vpn >= pageTableSize)
    {
        DEBUG('a', "virtual page # %d too large for page table size %d!\n",
              virtAddr, pageTableSize);
        *exception = AddressErrorException;
        return NULL;
    }
    /* 遍历页表 */
    for(TranslationEntry* e=(TranslationEntry*)(pageTable->getHead()); e !=
NULL; e = (TranslationEntry*)(e->next)){
        if(e->valid && e->virtualPage == vpn){
            return e;
        }
    }
    *exception = PageFaultException;
    return NULL;
}
```

- 倒排页表的缺页异常处理算法如下：

```
/*
倒排也表缺页异常处理算法
*/
ExceptionType
Machine::replacePageTableDes(int virtAddr, List* pageTable)
{
    unsigned int vpn, offset;
    TranslationEntry *entry;

    vpn = (unsigned)virtAddr / PageSize;
    offset = (unsigned)virtAddr % PageSize;

    if (vpn >= pageTableSize)
    {
        DEBUG('a', "virtual page # %d too large for page table size %d!\n",
              virtAddr, pageTableSize);
        return AddressErrorException;
    }
    /* 遍历页表 查找是否有磁盘缓存*/
    for(TranslationEntry* e=(TranslationEntry*)(pageTable->getHead()); e !=
NULL; e = (TranslationEntry*)(e->next)){
        if(e->virtualPage == vpn && e->onDisk){
            entry = e;
            break;
        }
    }

    int *pageFromDisk = NULL;
    if (entry != NULL)
    { //页面在磁盘，从磁盘重新读取页面到pageFromDisk
        for (ListElement *page = disk->getHead(); page != NULL; page = page-
>next)
        {
            if ((int)page == pageTable[vpn].diskAddr)
            {
                pageFromDisk = (int *)page;
                break;
            }
        }
    }
    //分配一个物理页面，并更新页表
    int pageNO = findNullPyhPage();
    if (pageNO == -1)
    { //物理页面满
        TranslationEntry *replacePage = selectOne(pageTable, pageTableSize);

        int *diskPage = new int[PageSize / 4]; //将数据拷贝存储到磁盘
        memcpy(diskPage, mainMemory + replacePage->physicalPage * PageSize,
PageSize);
        disk->Append((void *)diskPage);

        replacePage->onDisk = true; //更新替换页表项
        replacePage->valid = false;
        replacePage->diskAddr = (int)diskPage;
        pageNO = replacePage->physicalPage;
```

```
    }
    pageTable[vpn].count = 0; //更新当前页表项
    pageTable[vpn].physicalPage = pageNO;
    pageTable[vpn].valid = true;

    if (pageFromDisk != NULL)
    { //拷贝磁盘数据
        memcpy(mainMemory + pageNO * PageSize, pageFromDisk, PageSize);
    }
}
```