CSE-314

Nachos 3 Assignment Guidelines:

Demand Paging & Page Replacement

Submission deadline:
Mark distribution: at the bottom

Additional Details:
https://docs.google.com/document/d/1cNDO2DKhjH7ZFqVqudcaQXkLwITgBG1PNXWyFN3BKNs/edit#

This assignment is built on top of Nachos Project 2. Therefore if you have completed Project 2 perfectly, then you should use your code otherwise complete assignment 2 first or use someone else's code.

Task 1: Start up

1. If you have your own working Nachos2 project you can skip this phase.
2. ~~First download this project. Extract it. Then test it in your environment.~~
3. To Run,
    a. Suppose if you are in code directory, then whenever you make any changes to your test c programs or your nachos headerfile/.cc/.h file then you should apply make command
    make
    ->After that you give following commands,
    cd test
    make
    cd ../userprog/
    make
    ./nachos -x ../test/halt

b. As you can see you have to test and compile your program many times. it's better to create a bash file for this.

~~c. I have included run.sh in your code directory for this case. But before this you should give make command to compile your programs.~~

D. You should write some printf("testing testing\n") here and there to test whether you changes are working.

Task 2: Demand paging

Now that you have working programs. Your first task is to implement demand paging. To do this,

//addrspace.cc

1. First you need to,
    a) comment: ASSERT(numPages <= NumPhysPages).
    b) In pageTable entry simply set each physicalPage to -1, valid to false
    c) Comment: zeroing entire machine->mainMemory
    d) Comment: loading code segment, loading datasegment

2. Test your program with halt.c. Your output should throw pageFaultException. A gentle reminder, while doing following tasks test your program regularly.

3. In your, exception.cc file, look for `pagefaultexception` handling function. If you check machine.h then you will see there are lots of defined register number. Among them 39 is defined as badAddrsRegister. This means the virtual address for which the page fault was generated, that address is stored in the 39th register.

In this case, in the following code segment you should start your coding

```
else if (which == PageFaultException)
{
        //You are going to write your codes here
        //ExitProcess();
}

*************************
* Find
        faulting address: addr=machine->ReadRegister(39);

* Find virtualPageNumber
        vpn= addr/PageSize
            - PageSize may not be available, include correct header file for this

* Get a physical Page No:
```

```
if( any free physical memory){
        physicalPageNo=memoryManager->Alloc()
}
else{
         a//will force to free page
        //will do this later
}
```

* call: currentThread->space->loadIntoFreePage(addr, physicalPage);

4. The loadIntoFreePage function belongs to addrspace.cc, so you should declare it properly into, addrspace.h and addrspace.cc

```
int loadIntoFreePage(int addr, int physicalPageNo){

      return 0;
}
```

5. Now our target is to load the faulting page from executable file to physicalPage position.

But before doing that, keep in mind that now we need to hold onto executable file and noffHeader, for our entire process life time.

a) So in progtest.cc at startprocess function, ‘delete exectuable’ should be omitted and moved to addrspace.cc ‘s destructor method. keep a local variable in addrspace for this.

b) Also you will use loadIntoFreePage function for loading pages into memory, so noffH should aslo be avaible here. Make *noffH a member variable to AddrSpace class.

6. Now back to our loadIntoFreePage function, here we will do following tasks,

loadIntoFreePage(int addr, int physicalPageNo)

a) Find Virtual Page No: vpn
b) Update pageTableEntry of the corresponding vpn
    i) Set physical page to argument physicalPageNo
    ii) Mark entry valid

Loading corresponding segment to main memory:

Nachos Object File Format (Noff)

The nachos object file format (noff) is described using the nachos (C style) kernel stru

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

typedef struct segment {
  int virtualAddr;              /* location of segment in virt addr space */
  int inFileAddr;               /* location of segment in this file */
  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;
```

As you can see from the noff file structure,

The executable file divided into three primary segment, code, initialized data and uninitialized data.

Each segment has virtualAddress and inFileAddress and size.

virtualAddress means the virtualBaseAddress of this segment,

inFileAddress means the actual base location of that segment in the file,

Now our generated badAddress is a virtual address, therefore our task is to first determine which segments it belongs and then load it accordingly using inFileAddr.

Details here: http://condor.depaul.edu/glancast/546class/docs/lec5.html (not needed actually)

d) Now think about the cases,

The code segment, data segment are not exact multiple of page sizes so it is entirely possible that page starting in one segment and closing in another, so you have to carefully handle it.

e) loading in memory cases:

If addr is between code.virtualBaseAddress to code.virtualBaseAddress+size

//Find Codeoffset, means how many pages addr from the virtual base

codeoffset=(addr-code.virtualBaseAddrss)/PageSize

//Find how many bytes can actually we read from the code segment

codeSize=minimum(size-codeoffset*pageSize, PageSize)

//load the codesize amount data into memory . take a closer look how entire code and data was loaded into memory before. Use that insight.
executable->ReadAt();

If codeSize<PageSize then //overlapping case
Determine dataSize
Loadfrom initData Segment to memory

end
end

-> if addr falls into initDataSegment do same as above considering overlap with uninitialized
-for uninitialized segment zero'd the memory

-> if addr falls into uninitDataSegment zero'd the memory

-> if addr falls in any other cases except above, zero'd the memory

I am not sure about the cases that base address is in code segment and base+PageSize belong to uninitData segment. So you should check into that

f) Now compile and test your programs, it should work correctly But do this first,
A correction in the exception.cc file is in the ExitProcess Function
Here instead of freeing all pages,
First check for valid page table entries, if valid then free from the physical pages,
Because due to demand paging, all physical pages might not be needed to load for completing program. So removing an unloaded page will raise exception.

Task 3: Page Replacement without SwapFile

Part 1: Random Replacement

To do this, in the MemoryManager class we are going to include following two arrays,

```
Int *processMap;
TranslationEntry *entries;

//initalize these into constructor otherwise you will get errors
```

That means, now during allocation of a physical page number, we are also going to keep which process we are giving it to and will maintain a reference of that process's pagetable entry.

Now in addition to the previous memorymanager-> Alloc() function , add following functions,

1. int Alloc(int processNo, TranlationEntry &entry);
2. int AllocByForce();

1. This one is same as before just if ret is valid then keep which processNo, and entry

2. In AllocByForce() just return a random page, do not update processNo, entry now, because we have to relocate that page into swap memory later on.

a) In exception.cc pagefault cases,

```
If there are free memory then
        physicalPageNo=alloc(currentThread->id,machine->pagetable[vpn
])
else
        physical pageNo=allocByForce()

        //do swapfile case later
end
```

machine->space->loadIntoFreePage(int addr, int physicalPageNo)

b) Test this part by first making the number of physical pages small like 3 or 4. Then, run programs. Check which addresses are bad, which pages are getting evicted.

Part 2: LRU Replacement

// You can do this part later as this is independent.

1. In the function AllocByForce(), Now you can apply different algorithm to select which page you are going to evict. If some page is not used then you can evict that or you can keep timestamp of last time that page got access.

2. To do that in the Translate.h you need to change TranslationEntry to now additionally keep access time.
3. Then search for the pages with lower access time and evict it. Also you can chose to not evict dirty pages due to performance issues.

Task 4: Page Replacement with SwapFile

In the previous page replacement methodologies, we were always reading directly from files, which is expensive. Also we didn't actually deal with dirty pages.

Now we are going create a separate memory from HardDisk where we will keep the physical pages that we are replacing.

In pageFault, first we will look into this swapmemory. if not available we will load from executable.

While replacing with a page,

if the page not in swap then we will store it there
If it dirty then we will store it in swap
Otherwise nothing to do

Use SwapFile as FileSys Or Simply Byte Array

Some of you have already used STUB FILESYSTEM for this, if you could use it then it's perfectly fine

If you check userprog/MakeFile in the definition you will see nachos uses stub file system

In filesys.h and filesys.cc you will see based on the definition of the make file the system uses either FILESYS_STUB or FILESYS

Now to create a swap memory in disk, we have to use FILESYS,

1. In UserProg/MakeFile, comment this line
   DEFINES = -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
   And use this,
   DEFINES = -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS

2. Now in the filesys.cc
   Create a new OpenFile *SwapFile and its initialization.
   For your convenience, you can look into this
   https://drive.google.com/open?id=0B-SybtAwa8vcS2hjdHlhSDM2UTA

3. If you do this, then you have a chunk of swapmemory as OpenFile *SwapFile, which you can read and write.

**** if the above procedure doesn't work Create a SwapPage class and Maintain an SwapPage* array in memory manager as swap memory. Malloc 256 of them
In SwapPage you should keep a byte array of size 1 physical page

**** Or Substitute with an byte array of large size 128*256 in addrspace.h if above procedure doesn't work**********

**** Or your any suitable way

4. Now create another object MemoryManager *swapMemoryManager; int ProgTest.cc and intialize it to swapPagesSizes.

5. SwapMemoryManager will keep track of your free swapspace. As swapmemory is large you will always get a free swap page.

6. In addrspace.cc add functions
   saveIntoSwapSpace(int vpn)
   loadFromSwapSpace(int vpn).
   isSwapPageExists(int vpn)

a) saveIntoSwapSpace(int vpn) will store the
   Take a free SwapPage
   Keep map between vpn to swapPageNo
   pagetable[vpn].physicalPage page from machine->mainmemory allocated swapPage

b) loadFromSwapSpace(int vpn) will load the swap page into machine->mainmemory, pagetable[vpn].physicalPage

c) isSwapPageExists(int vpn) this function will check for a machine->space->isSwapPageExists(vpn)

7. So in the addrspace.cc inside this loadIntoFreePage(int addr, int physicalPageNo)

Function we will check if the vpn=addr/PageSize is available in swap memory if exists load from there otherwise load as usual.

8. And while evicting a page: in exception.cc, the evicted page should be saved into corresponding process' swapspace.

Mark Distribution:

1. Demand Paging : 20%
2. Demand Paging test working: 10%
3. Random Page Replacement: 10%
4. Random Page Replacement testing: 10%
5. LRU Page Replacement: 10%
6. LRU Page Replacement testing: 10%
7. SwapFile Page Replacement: 20%
8. Submission: 10%

Overall test Programs are: Halt.c, matmult.c, sort.c, array.c etc,
You must write custom c program files for different cases.

Do's and Don'ts:

1. Always run your programs, compile it, test it.
2. Use a good IDE
3. Don't copy codes from another.
4. Check for header files, use extern for accessing one variable from another file
5. Start couple of days early :)