NTU Operating System Project 3

tags: `NTU_OS` `Operating System` `NachOS` `Memory Management`

Description First

I used the code from `Project2` directly and obtained the correct value, 1 and 7220, by the command `./nachos -e ../test/sort -e ../test/matmult` respectively. After debugging a while, I knew what's the matter. In `project2`, I set `const unsigned int NumPhysPages = 256;` so that it can handle huge computing resource. In order to address this project, we must change it back to 32.

Motivation

- Normally speaking, if you follow the command `./nachos -e ../test/matmult` and `./nachos -e ../test/sort` in `Project2`, you'll get

  ```
  Total threads number is 1
  Thread ../test/sort is executing.
  Assertion failed: line 118 file ../userprog/addrspace.cc
  Aborted (core dumped)
  ```

  ```
  Total threads number is 1
  Thread ../test/matmult is executing.
  Assertion failed: line 118 file ../userprog/addrspace.cc
  Aborted (core dumped)
  ```

- Our goal is to obtain the correct outcome `1` and `7220` from `/test/sort` and `/test/matmult` respectively that shown as below

  ```
  $ ./nachos -e ../test/sort -e ../test/matmult
  Total threads number is 2
  Thread ../test/sort is executing.
  Thread ../test/matmult is executing.
  return value:7220
  return value:1
  No threads ready or runnable, and no pending interrupts.
  Assuming the program completed.
  Machine halting!
  ```

```
Ticks: total 43818400, idle 40, system 4381880, user 39436480
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
```

- Now, let's try to analyze the problem. NachOS has physical memory limitation, so that it cannot handle the program that needs huge memory resource such as `sort.c` and `matmult.c`. Thus, we can utilize `Demand Paging` to address it → **realize a virtual memory if the main memory has no space**

  > When the space in main memory is enough, pages will be stored in main memory. If the space is insufficient, the pages and data will swap out. When there is valid space, i.e. the space is released, and the data is needed, the pages and data will then swap in main memory. The scheduling method of the page replacement algorithm is implemented by LRU(Least Recently Used). - by wangssuming

Implementation

- `/code/userprog/userkernel.h`
  create a new `SynchDisk` called `SwapDisk` and replace `debugUserProg` from private to public

```
class UserProgKernel : public ThreadedKernel
{
    public:
        ...
        /*-----------------------Homework for Memory Management-------------
-----------*/
        // Create a new SynchDisk called SwapDisk to simulate the secondary
storage
        SynchDisk *SwapDisk;     // SwapDisk saves pages if main memory is
not enough
        bool debugUserProg;     // single step user program
        /*-----------------------Homework for Memory Management-------------
-----------*/
        Machine *machine;
        FileSystem *fileSystem;
        ...
    private:
        ...
        /*-----------------------Homework for Memory Management-------------
-----------*/
        // No longer needed for HW3
        //bool debugUserProg;       // single step user program
        /*-----------------------Homework for Memory Management-------------
-----------*/
        Thread* t[10];
        ...
};
```

- `/code/userprog/userkernel.cc`
  Initialize `SwapDisk`

```
void UserProgKernel::Initialize()
{
    /*-----------------------Homework for Memory Management-----------------
-------*/
    ThreadedKernel::Initialize(RR);
    // Initialized SwapDisk
    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
    SwapDisk = new SynchDisk("New SwapDisk");// Swap disk for virtual memory
    #ifdef FILESYS
        synchDisk = new SynchDisk("New SynchDisk");
    #endif // FILESYS
    /*-----------------------Homework for Memory Management-----------------
-------*/
}
void UserProgKernel::Initialize(SchedulerType type)//
{
    ...
    /*-----------------------Homework for Memory Management-----------------
-------*/
    SwapDisk = new SynchDisk("New SwapDisk");// Swap disk for virtual memory
    /*-----------------------Homework for Memory Management-----------------
-------*/
    #ifdef FILESYS
    ...
};
```

- `/code/machine/machine.h`
  Defined lots of variable in `Machine` class contained `UsedPhyPage` to record whether the physical memory is used or not, `UsedVirtualPage` to record whether the virtual memory is used or not, etc.

```
  class Instruction;
  class Interrupt;

  class Machine
  {
      public:
          ...
          TranslationEntry *pageTable;
          unsigned int pageTableSize;
          bool ReadMem(int addr, int size, int* value);
          /*-----------------------Homework for Memory Management-----------
-------------*/
          bool UsedPhyPage[NumPhysPages]; //record the pages in the main
memory
          bool UsedVirtualPage[NumPhysPages]; //record the pages in the
virtual memory
          int ID_number; // machine ID
          int PhyPageInfo[NumPhysPages]; //record physical page info (ID)
          TranslationEntry *main_tab[NumPhysPages]; // pagetable
          /*-----------------------Homework for Memory Management-----------
-------------*/
      private:
        ...
};
```

- `/code/userprog/addrspace.cc`
    - In line `45`

        > Since we are handling virtual memory, the ASSERT to guarantee the number of pages does not exceed the number of physical pages in main memory in `/code/userprog/addrspace.cc` is no longer needed. - by pai445

    - The `for` loop in line `79` is used to find an available page for current process. The `while` loop in line `82` is as similar as `project2` that used variable `j` to find free memory page.

        > There are two different cases in the following. When the **main memory still have empty frame**, then we can put the page into the main memory and update the information to the page table. This step is achieved by the function `ReadAt`. The other case will be the **main memory is fulled**. Then we have check the available virtual memory space by the similar while loop and write the page in to `SwapDisk` by the `WriteSector` function. - by pai445

    - `Execute` function in line `135` and `SaveState` in line `147`
      We use a flag, `pageTable_is_load`, defined in `/code/userprog/addrspace.h` to check whether the page table is successfully loaded to make the context-switch work.

```
#define PAGE_OCCU true
#define PAGE_FREE false
/*-----------------------Homework for Memory Management---------------------
---*/
// There's no longer needed in HW3
// bool AddrSpace::PhyPageStatus[NumPhysPages] = {PAGE_FREE};
// int AddrSpace::NumFreePages = NumPhysPages;
/*-----------------------Homework for Memory Management---------------------
---*/
AddrSpace::AddrSpace()
{
    /*-----------------------Homework for Memory Management-----------------
-------*/
    ID=(kernel->machine->ID_number)++;
    kernel->machine->ID_number=(kernel->machine->ID_number)++;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    ...
}

AddrSpace::~AddrSpace()
{
    // Free the physical page that this program used
    /*-----------------------Homework for Memory Management-----------------
-------*/
    // No longer needed for HW3
    // for(int i = 0; i < numPages; i++)
    // {
```

```
    //     AddrSpace::PhyPageStatus[pageTable[i].physicalPage] = PAGE_FREE;
    //     AddrSpace::NumFreePages++;
    // }
    /*-----------------------Homework for Memory Management-----------------
-------*/
    delete pageTable;
}

bool AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

    /*-----------------------Homework for Memory Management-----------------
-------*/
    unsigned int tmp;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    ...
    size = numPages * PageSize;

    /*-----------------------Homework for Memory Management-----------------
-------*/
    /* For HW3 is no longer needed  */
    // ASSERT(numPages <= NumPhysPages);    // check we're not trying
                                            // to run anything too big -
-
                                            // at least until we have
                                            // virtual memory

    // Allocate
    pageTable = new TranslationEntry[numPages];
    // No longer needed for HW3
    /*for(unsigned int i = 0, idx = 0; i < numPages; i++)
    {
        pageTable[i].virtualPage = i;
        while(idx < NumPhysPages && AddrSpace::PhyPageStatus[idx] ==
PAGE_OCCU) idx++;
        AddrSpace::PhyPageStatus[idx] = PAGE_OCCU;
        AddrSpace::NumFreePages--;
        // Clean the page that'll be used soon
        bzero(&kernel->machine->mainMemory[idx * PageSize], PageSize);
        pageTable[i].physicalPage = idx;
        pageTable[i].valid = true;
        pageTable[i].use = false;
        pageTable[i].dirty = false;
        pageTable[i].readOnly = false;
    }*/
    /*-----------------------Homework for Memory Management-----------------
-------*/

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " <<
size);
    if (noffH.code.size > 0)
    {
        /*-----------------------Homework for Multithread-------------------
-----*/;
        // For HW3 is no longer needed
        // DEBUG(dbgAddr, "Initializing code segment.");
        // DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size)
        // executable->ReadAt(&(kernel->machine-
>mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage *
PageSize + (noffH.code.virtualAddr%PageSize)]), noffH.code.size,
noffH.code.inFileAddr);
        // executable->ReadAt(&(kernel->machine-
>mainMemory[noffH.code.virtualAddr]), noffH.code.size,
noffH.code.inFileAddr);

        for(unsigned int j=0,i=0;i < numPages ;i++)
        {
            j=0;
            while(kernel->machine->UsedPhyPage[j]!=FALSE&&j<NumPhysPages)
{j++;}

            // main memory is enough, put the page to main memory
            if(j<NumPhysPages)
            {
                kernel->machine->UsedPhyPage[j]=TRUE;
                kernel->machine->PhyPageInfo[j]=ID;
                kernel->machine->main_tab[j]=&pageTable[i];
                pageTable[i].physicalPage = j;
                pageTable[i].valid = TRUE;
                pageTable[i].use = FALSE;
                pageTable[i].dirty = FALSE;
                pageTable[i].readOnly = FALSE;
                pageTable[i].ID =ID;
                pageTable[i].LRU_counter++; // LRU counter when save in
memory
                executable->ReadAt(&(kernel->machine-
>mainMemory[j*PageSize]),PageSize, noffH.code.inFileAddr+(i*PageSize));
            }
            // main memory is not enough, use virtual memory
            else
            {
                char *buffer;
                buffer = new char[PageSize];
                tmp=0;
                while(kernel->machine->UsedVirtualPage[tmp]!=FALSE){tmp++;}
                kernel->machine->UsedVirtualPage[tmp]=true;
                pageTable[i].virtualPage=tmp; //record the virtual page we
save
                pageTable[i].valid = FALSE; //not load in main memory
                pageTable[i].use = FALSE;
                pageTable[i].dirty = FALSE;
                pageTable[i].readOnly = FALSE;
                pageTable[i].ID =ID;
                executable->ReadAt(buffer,PageSize, noffH.code.inFileAddr+
(i*PageSize));
```

```
                kernel->SwapDisk->WriteSector(tmp,buffer); // write in
virtual memory (SwapDisk)
            }
        }
        /*-----------------------Homework for Memory Management-------------
-----------*/
    }
    /*-----------------------Homework for Memory Management-----------------
-------*/
    if (noffH.initData.size > 0)
    {
        // For HW1, it's needed, but not in HW3
        // DEBUG(dbgAddr, "Initializing data segment.");
        // DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " <<
noffH.initData.size);
        // executable->ReadAt(&(kernel->machine-
>mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage *
PageSize + (noffH.code.virtualAddr%PageSize)]), noffH.initData.size,
noffH.initData.inFileAddr);

        // For HW3, it's needed, but not in HW1
        executable->ReadAt(&(kernel->machine-
>mainMemory[noffH.initData.virtualAddr]),noffH.initData.size,
noffH.initData.inFileAddr);
    }
    /*-----------------------Homework for Memory Management-----------------
-------*/

    delete executable;      // close file
    return TRUE;            // success
}
void AddrSpace::Execute(char *fileName)
{
    /*-----------------------Homework for Memory Management-----------------
-------*/
    pageTable_is_load=FALSE;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    if (!Load(fileName))
    ...
    /*-----------------------Homework for Memory Management-----------------
-------*/
    pageTable_is_load=TRUE;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    ...
}
void AddrSpace::SaveState()
{
    /*-----------------------Homework for Memory Management-----------------
-------*/
    if(pageTable_is_load)
    {
        pageTable=kernel->machine->pageTable;
        numPages=kernel->machine->pageTableSize;
    }
    /*-----------------------Homework for Memory Management-----------------
-------*/
}
```

- `/code/userprog/addrspace.h`
  Defined variable `ID` and `pageTable_is_load`

  ```
  class AddrSpace
  {
      public:
          ...
          /*-----------------------Homework for Memory Management-------------
  -----------*/
          int ID;
          /*-----------------------Homework for Memory Management-------------
  -----------*/
      private:
          ...
          static int NumFreePages;
          /*-----------------------Homework for Memory Management-------------
  -----------*/
          bool pageTable_is_load;
          /*-----------------------Homework for Memory Management-------------
  -----------*/
  };
  ```

Now, the OS can implement that swap the data from main memory to virtual memory when main memory is full. For the following step, we have to implement page replacement algorithms, Least Recently Used (`LRU`).

- `/code/machine/translate.h`

  > We implement by a hardware counter `LRU_counter` in `/code/machine/traslate.h`. - by pai445

  ```
  class TranslationEntry
  {
      public:
          ...
          /*-----------------------Homework for Memory Management-------------
  -----------*/
          int LRU_counter;    // counter for LRU
          int ID;             // page table ID
          /*-----------------------Homework for Memory Management-------------
  -----------*/
  };
  ```

- `/code/machine/translate.cc`

> First, we check the valid-invalid bit of the page table. If is not valid, it means the page is not in the main memory. Hence, we need to load from the secondary storage. There are two cases that may happen. First, if there are some empty frame in the main memory, then we can just load the page into it. The other case is that the main memory is fulled. In this cases, we create two buffers and runs least recently used (LRU) algorithm to find the victim. The ReadSector and WriteSector are used to read/write the temporarily saved pages found by the LRU algorithm. The victim is pull out from the main memory, and then swapped by our page into the corresponding frame. The page table will be updated correspondingly in both cases. In our implementation, LRU will perform linear search on LRU_counter to find the least recently used page. - by pai445

- In previous TA's Project documentation, we used `kernel->swap->WriteSector` and `kernel->swap->ReadSector` to store the virtual memory from hard disk(`swap` is a object of `SynchDisk`)
  Therefore, in line `34-35`, we used `ReadSector` to read `pageTable[vpn].virtualPage` and store into the buffer. Then use `bcopy` function to move byte sequence from `src`→`buffer` to `dest`→`&mainMemory[j*PageSize]` with `size=PageSize` only if the main memory is sufficient. **Reference[1]**
- In line `63-66`. If the main memory is full, then we used `LRU` algorithm to find the least recently used memory and switch to disk and replace an desired data to main memory. That is, `memory section was found by LRU algorithm`→`buffer1`→`virtual memory` and `desired memory section stored in virtual memory`→`buffer2`→`main memory`

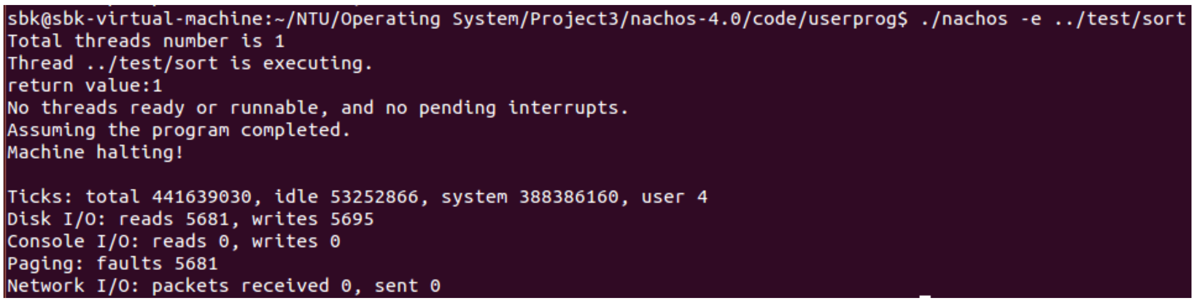
```
ExceptionType Machine::Translate(int virtAddr, int* physAddr, int size, bool
writing)
{
    unsigned int pageFrame;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    int victim;///find the page victim
    unsigned int j;
    /*-----------------------Homework for Memory Management-----------------
-------*/
    DEBUG(dbgAddr, "\tTranslate " << virtAddr << (writing ? " , write" : " ,
read"));
    ...
    if (tlb == NULL)
    {       // => page table => vpn is index into table
        if (vpn >= pageTableSize){...}
        else if (!pageTable[vpn].valid)
        {
            /*-----------------------Homework for Memory Management---------
---------------*/
            // For HW3, these two lines are no longer needed
            // DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
            // return PageFaultException;
            kernel->stats->numPageFaults++; // page fault
            j=0;
            while(kernel->machine->UsedPhyPage[j]!=FALSE&&j<NumPhysPages)
{j++;}
            // load the page into the main memory if the main memory is not
full
```

```
            if(j<NumPhysPages)
            {
                char *buffer; //temporary save page
                buffer = new char[PageSize];
                kernel->machine->UsedPhyPage[j]=TRUE;
                kernel->machine->PhyPageInfo[j]=pageTable[vpn].ID;
                kernel->machine->main_tab[j]=&pageTable[vpn];
                pageTable[vpn].physicalPage = j;
                pageTable[vpn].valid = TRUE;
                pageTable[vpn].LRU_counter++; // counter for LRU

                kernel->SwapDisk->ReadSector(pageTable[vpn].virtualPage,
buffer);
                bcopy(buffer,&mainMemory[j*PageSize],PageSize);
            }
            // main memory is full, page replacement
            else
            {
                char *buffer1;
                char *buffer2;
                buffer1 = new char[PageSize];
                buffer2 = new char[PageSize];
                //Random
                victim = (rand()%32);

                //LRU
                int min = pageTable[0].LRU_counter;
                victim=0;
                for(int index=0;index<32;index++)
                {
                    if(min > pageTable[index].LRU_counter)
                    {
                        min = pageTable[index].LRU_counter;
                        victim = index;
                    }
                }
                pageTable[victim].LRU_counter++;

                //printf("Number %d page is swapped out\n",victim);

                // perform page replacement, write victim frame to disk,
read desired frame to memory
                bcopy(&mainMemory[victim*PageSize],buffer1,PageSize);
                kernel->SwapDisk->ReadSector(pageTable[vpn].virtualPage,
buffer2);
                bcopy(buffer2,&mainMemory[victim*PageSize],PageSize);
                kernel->SwapDisk-
>WriteSector(pageTable[vpn].virtualPage,buffer1);

                main_tab[victim]->virtualPage=pageTable[vpn].virtualPage;
                main_tab[victim]->valid=FALSE;

                //save the page into the main memory

                pageTable[vpn].valid = TRUE;
```

```
                    pageTable[vpn].physicalPage=victim;
                    kernel->machine->PhyPageInfo[victim]=pageTable[vpn].ID;
                    main_tab[victim]=&pageTable[vpn];
                    //printf("Page replacement finish\n");
                }
                /*-----------------------Homework for Memory Management---------
---------------*/
            }
            entry = &pageTable[vpn];
        }
        ...
}
```

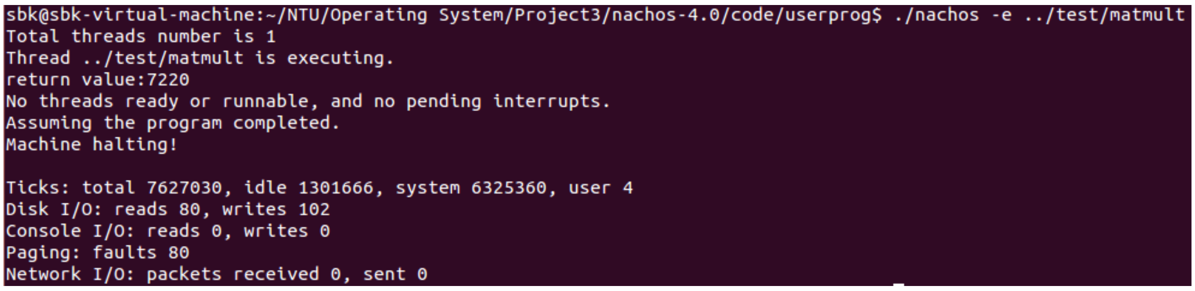
Result

`./nachos -e ../test/sort`



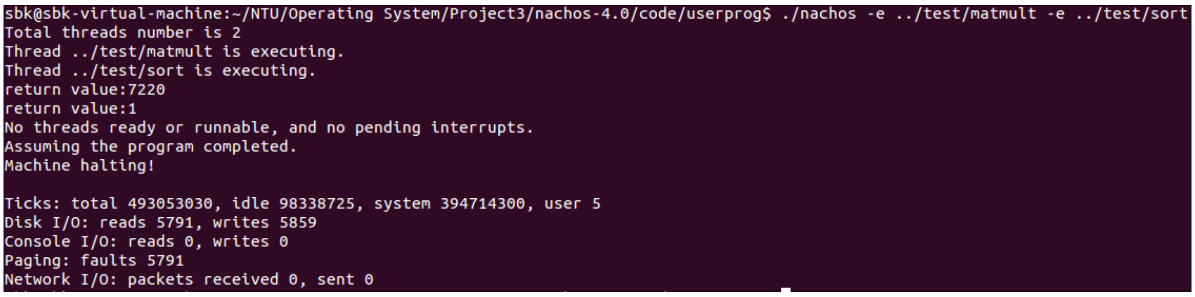
```
sbk@sbk-virtual-machine:~/NTU/Operating System/Project3/nachos-4.0/code/userprog$ ./nachos -e ../test/sort
Total threads number is 1
Thread ../test/sort is executing.
return value:1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 441639030, idle 53252866, system 388386160, user 4
Disk I/O: reads 5681, writes 5695
Console I/O: reads 0, writes 0
Paging: faults 5681
Network I/O: packets received 0, sent 0
```

`./nachos -e ../test/matmult`



```
sbk@sbk-virtual-machine:~/NTU/Operating System/Project3/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult
Total threads number is 1
Thread ../test/matmult is executing.
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7627030, idle 1301666, system 6325360, user 4
Disk I/O: reads 80, writes 102
Console I/O: reads 0, writes 0
Paging: faults 80
Network I/O: packets received 0, sent 0
```

`./nachos -e ../test/matmult -e ../test/sort`



```
sbk@sbk-virtual-machine:~/NTU/Operating System/Project3/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult -e ../test/sort
Total threads number is 2
Thread ../test/matmult is executing.
Thread ../test/sort is executing.
return value:7220
return value:1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 493053030, idle 98338725, system 394714300, user 5
Disk I/O: reads 5791, writes 5859
Console I/O: reads 0, writes 0
Paging: faults 5791
Network I/O: packets received 0, sent 0
```

Reference

- github/pai4451
- github/wangssuming
- [1] Supplementary Note for `bcopy`