OS Project 1

工科海洋四 劉名凱 B08505048

1. Why does the result behave differently from our expectation?

Because from addrspace.cc file we can see that the author assumes the nachos OS is only uni-programming. Therefore, the author does not actually implement the concept of virtual memory. Instead, the author assigns each process a page table with its size equal to physical memory's size.

addrspace.cc

```
// AddrSpace::AddrSpace
//  Create an address space to run a user program.
//  Set up the translation from program memory to physical
//  memory.  For now, this is really simple (1:1), since we are
//  only uniprogramming, and we have a single unsegmented page table
//----------------------------------------------------------------------

AddrSpace::AddrSpace()
                                        NumPhysPages = Total number of physical pages
  {
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    pageTable[i].physicalPage = i;
//  pageTable[i].physicalPage = 0;
    pageTable[i].valid = TRUE;
//  pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
//  bzero(kernel->machine->mainMemory, MemorySize);
  }
```

When we run two processes at the same time, these two processes both possess page table with the size of physical memory, i.e., they share same region of physical memory (or whole physical memory), which cause the code/data segment of these processes mixed together. As a result, running multiple processes at the same time won't generate desired output.

2. How to solve the issue?

We need to implement the concept of virtual memory, allocating memory space to each process according to their demand of memory. Besides, different processes

must possess different region of the virtual memory; otherwise, the code/data segment from different processes will mixed together.

First adding usedPhysPages and FreePages in addrspace.h to record used physical pages and number of free pages in physical memory respectively. usedPhysPages and FreePages are both shared by all processes (static)

addrspace.h

```
private:
  TranslationEntry *pageTable;    // Assume linear page table translation
                  // for now!
  unsigned int numPages;      // Number of pages in the virtual
                  // address space

  bool Load(char *fileName);      // Load the program into memory
                  // return false if not found

  void InitRegisters();       // Initialize user-level CPU registers,
                  // before jumping to user code

  static bool usedPhysPages[NumPhysPages]; // Used physical pages
  static unsigned int FreePages;
```

Initialize usedPhysPages with all entries equal to false; Freepages is equal to total number of physical pages at the beginning.

addrspace.cc

```
#include "copyright.h"
#include "main.h"                          False = unoccupied
#include "addrspace.h"                     True = occupied
#include "machine.h"
#include "noff.h"
bool AddrSpace::usedPhysPages[NumPhysPages] = {false}; // initialize all physical pages to unused state
unsigned int AddrSpace::FreePages = NumPhysPages; // all pages are available
```

Delete unnecessary code in constructor

addrspace.cc (AddrSpace::AddrSpace())

```
AddrSpace::AddrSpace()
{
}
```

Declare page table in AddrSpace::Load()
Here I used first fit to find free physical pages, i.e., find the first empty physical page from top in each iteration. Each time I discover an empty physical page, I'll modified usedPhysPages and FreePages to indicate current physical page is occupied; I'll also clear the content of current physical page before assigning data to it. (bzero())
addrspace.cc (AddrSpace::Load())

```
ASSERT(numPages <= FreePages);      // check we're not trying
                    // to run anything too big --
                    // at least until we have
                    // virtual memory

/* load page table */
pageTable = new TranslationEntry[numPages];   numPages = number of pages required by the processes
for (unsigned int i = 0, idx = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;
    while(idx < NumPhysPages && AddrSpace::usedPhysPages[idx] == true) idx++;
    AddrSpace::usedPhysPages[idx] = true;
    AddrSpace::FreePages--;
    bzero(&kernel->machine->mainMemory[idx * PageSize], PageSize);
    pageTable[i].physicalPage = idx;              Free current page
    pageTable[i].valid = true;
    pageTable[i].use = false;
    pageTable[i].dirty = false;
    pageTable[i].readOnly = false;
}
```

After implementing concept of virtual memory, we need to convert virtual memory address to physical memory address when loading process' code and data in to main memory. The relation between physical memory address and virtual memory address is equal to:

Physical memory address = pageTable[virtual memory address / PageSize].physicalPage*PageSize + (virtual memory address % PageSize)

virtual memory address / PageSize = corresponding index of page Table
virtual memory address % PageSize = offset

Combining page Table values and offset, we can acquire the actual physical memory address in main memory.

addrspace.cc (AddrSpace::Load())

```
    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        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);
    }
    if (noffH.initData.size > 0) {
        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.initData.virtualAddr%PageSize)]), noffH.initData.size, noffH.initData.inFileAddr);
    }

    delete executable;          // close file
    return TRUE;            // success
}
```

Load code

Load data

After process finish running, we need to deallocate used memory using deconstructor.

addrspace.cc (AddrSpace::~AddrSpace())

```
AddrSpace::~AddrSpace()
{
    // deallocate physical pages
    for (unsigned int i = 0; i<numPages; i++) {
        AddrSpace::usedPhysPages[pageTable[i].physicalPage] = FALSE;
        AddrSpace::FreePages++;
    }
    delete pageTable;
}
```

3. Result

```
daniel@daniel-VirtualBox: ~/downloads/nachos-4.0/code/userprog
daniel@daniel-VirtualBox:~$ cd downloads/nachos-4.0/code/userprog
daniel@daniel-VirtualBox:~/downloads/nachos-4.0/code/userprog$ ./nachos -e ../te
st/test1 -e ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300, idle 8, system 70, user 222
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
daniel@daniel-VirtualBox:~/downloads/nachos-4.0/code/userprog$
```

4. Discussion

The difficulty I encountered is figuring out how system execute a process, how system allocate memory to process. Finding the definition of system variable is also quite annoying. But thanks to google and the NTHU online courses, I am able to find solution by my own. Google really helps a lot.

The main problem to this code is how finding free physical pages is implemented. In each iteration, system has to find first free physical pages from the top of the physical memory, which greatly decrease the efficiency of allocating memory. A better way is to implement a linked-list recording free physical pages. In this case, system can allocate free physical page to process in O(1).

5. Feedback

Modifying nachos is a helpful and valuable experiences. Instead of reading textbook regarding Operating system, I can write code in operating system and execute them. It helps me better understand the structure of operating system.