OS2022 HW1 - Thread Management

Student: 游家權**(r10942152)**

Issue Description

Command Execute: ./nachos -e ../test/test1 -e ../test/test2
This command executes two different processes in Nachos at the same time. Test1 prints 9,8,7,6 in sequence, while test2 prints 20,21,22,23,24,25. These processes behave normally when I execute one at a time, however, if I execute test1 and test2 simultaneously, their behavior becomes unpredictable as shown below.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/userprog$
./nachos -e ../test/test2 -e ../test/test1
Total threads number is 2
Thread ../test/test2 is executing.
Thread ../test/test1 is executing.
Print integer:20
Print integer:21
Print integer:22
Print integer:9
Print integer:8
Print integer:7
Print integer:6
Print integer:23
Print integer:22
Print integer:21
Print integer:20
Print integer:19
return value:0
Print integer:18
Print integer:17
Print integer:16
Print integer:15
Print integer:14
Print integer:13
Print integer:12
Print integer:11
Print integer:10
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 600, idle 23, system 100, user 477
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
```

Why is the result not congruent with expected?

Because Nachos currently doesn't support multi-programming, it can only support one program running on the OS at a time, i.e., uni-programming. Therefore, when I ask Nachos to execute two processes at the same time, all processes will share the same memory space and page table. This will lead to the disaster that process A can modify variables that are declared in process B, when the context switch happens. This false memory sharing phenomenon is the reason why the program's result becomes so unpredictable. To be specific, variable 'i' declared in test1 and variable 'n' declared in test2 are actually sharing the same memory space.

Code Tracing and Observation

To resolve this issue, I have to assign physical memory to every process and make sure it doesn't interfere with other processes' storage. Therefore, I need to create a page table for every process and make sure the virtual addresses are all pointing at free, unused physical pages. In addition, I should distribute physical pages evenly among all programs instead of letting one process occupy the whole physical memory.

However, to achieve my goal, I need to trace nachos' code and find code segments that are related to the page table. I list my discovers below:

code/machine/translate.h and code/machine/translate.cc

These two files define page tables and TLB that can be used to translate virtual addresses to physical addresses. Therefore, I can directly utilize the class "TranslationEntry" and its member variables to create my own page table.

```
class TranslationEntry {
  public:
    unsigned int virtualPage;   // The page number in virtual memory.
    unsigned int physicalPage;  // The page number in real memory (relative to the
                        //  start of "mainMemory"
    bool valid;         // If this bit is set, the translation is ignored.
                        // (In other words, the entry hasn't been initialized.)
    bool readOnly;      // If this bit is set, the user program is not allowed
                        // to modify the contents of the page.
    bool use;           // This bit is set by the hardware every time the
                        // page is referenced or modified.
    bool dirty;         // This bit is set by the hardware every time the
                        // page is modified.
};
```

In this file. It also specifies page size to be 128 and number of total physical pages is 32, which means the total physical memory available is 32*128 byte = 4KB.

code/userprog/userkernal.cc

This file defines a multiprogramming entry point. New threads and address space will be created here to each thread. The AddrSpace() is the part I want to modify.

```
void
UserProgKernel::Run()
{

        cout << "Total threads number is " << execfileNum << endl;
        for (int n=1;n<=execfileNum;n++)
                {
                t[n] = new Thread(execfile[n]);
                t[n]->space = new AddrSpace();
                t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
                cout << "Thread " << execfile[n] << " is executing." << endl;
                }
```

code/userprog/addrspace.h and code/userprog/addrspace.cc

These two files define memory space that is assigned to the process. It also defines a page table and calculates the number of needed pages in this program.

With this code tracing work and observation, I figured out addrspace.h and addrspace.cc are two files that I want to modify to achieve my goal. I will elaborate how I plan to do this in detail in the next chapter.

```
class AddrSpace {
  public:
    AddrSpace();                        // Create an address space.
    ~AddrSpace();                       // De-allocate an address space

    void Execute(char *fileName);       // Run the the program
                                        // stored in the file "executable"

    void SaveState();                   // Save/restore address space-specific
    void RestoreState();                // info on a context switch

  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

};
```

How to solve the issue

In code/userprog/addrspace.h, I need to create new data structures to keep track of physical pages' status to record whether they are used by processes or not; therefore, I create a static bool array "isPhyPageUsed" to record status and also create an integer "numFreePhyPage" to record how many physical page are free. When the entry of "isPhyPageUsed" is false, it means the i-th physical pages is free, and the entry is true if the pages is occupied.
Finally, I declare a new function "VirtoPhys" to translate virtual address to physical address by using page table I created.

```
class AddrSpace {
  public:
    AddrSpace();                        // Create an address space.
    ~AddrSpace();                       // De-allocate an address space

    void Execute(char *fileName);       // Run the the program
                                        // stored in the file "executable"

    void SaveState();                   // Save/restore address space-specific
    void RestoreState();                // info on a context switch

  private:
    TranslationEntry *pageTable;        // Assume linear page table translation
                                        // for now!
    unsigned int numPages;              // Number of pages in the virtual
                                        // address space
    // TODO, add numFreePhyPage to keep track of number of free pages
    static unsigned int numFreePhyPage;
    // TODO, add isPhyPageFree to record the status of every physical pages.
    static bool isPhyPageUsed[NumPhysPages];

    int VirtoPhys(int virtualAddr);     // Translate virtual address to phyiscal


    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

};
```

In code/userprog/addrspace.cc, I initialize the newly defined data structure and switch all valid bits in the page table to false to disable all entries to hint that they haven't been initialized yet. The size of virtual address space can be arbitrary; therefore, I decide to not change the number of virtual page and create 32 virtual pages on the page table.

```
// TODO, Initialize Data structure that help record status of physical memory
bool AddrSpace::isPhyPageUsed[NumPhysPages] = {false};
unsigned int AddrSpace::numFreePhyPage = NumPhysPages;

AddrSpace::AddrSpace()
{
    // Allcoate so many pages is a bit overkill, but if we implement visual memo
ry correctly, this won't be a problem
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = i;
        // TODO, switch all valid  bit to False because I haven't init them
        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);
}
```

In code/userprog/addrspace.cc AddrSpace::Load(), the function calculates the number of pages required by this process. Firstly, I change the assertion condition to make sure there are enough free pages for this program. After that, I initialize the page table of this process and allocate free physical pages to every required virtual page by assigning a physical page number to it. I linearly search through the static bool array “isPhyPageUsed” to find the free pages and update its status after the assignment. Lastly, valid bits of assigned pages need to switch to true to enable address translation.

```
// how big is address space?
size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
                    + UserStackSize;        // we need to increase the size
                                            // to leave room for the stack
numPages = divRoundUp(size, PageSize);
//  cout << "number of pages of " << fileName<< " is "<<numPages<<endl;
size = numPages * PageSize;

// TODO, check free physical page is enough for this process
ASSERT(numPages <= numFreePhyPage);
// cout << "numFreePhyPage = " << numFreePhyPage << endl;

// TODO, Update pagetable of this process, and only allocate free physical page to it.
// cout << "numPages used = " << numPages << endl;
for (unsigned int i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;
    // Find a free up physical page by linear search
    for (unsigned int j = 0; j < NumPhysPages; j++){
        // cout << isPhyPageFree[j] << endl;
        if (not isPhyPageUsed[j]){
            // Allocate physical page to this visual page
            pageTable[i].physicalPage = j;
            isPhyPageUsed[j] = true;
            numFreePhyPage--;
            break;
        }
    }
    pageTable[i].valid = TRUE;
}
// cout << "numFreePhyPage = " << numFreePhyPage << endl;
```

In code/userprog/addrspace.cc AddrSpace::Load(), I need to translate virtual addresses to physical one when accessing memory. To achieve that, I write a function “VirtoPhys” to help me translate addresses. The virtual page number(i.e., virtual_addrs/PageSize) will be converted to physical address by accessing page table entries. The address offset is unaffected by the translation, so I use virtual address offset(i.e., virtual_addrs%PageSize) to get the physical offset. Finally, I add the physical page number and offset together and get the physical address.

```
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);
    // TODO need to translate virtualAddr to PhyiscalAddr
    executable->ReadAt(
            &(kernel->machine->mainMemory[VirtoPhys(noffH.code.virtualAddr)]),
                    noffH.code.size, noffH.code.inFileAddr);
}
    if (noffH.initData.size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
    // TODO need to translate virtualAddr to PhyiscalAddr
    executable->ReadAt(
            &(kernel->machine->mainMemory[VirtoPhys(noffH.initData.virtualAddr)]),
                    noffH.initData.size, noffH.initData.inFileAddr);
}
```

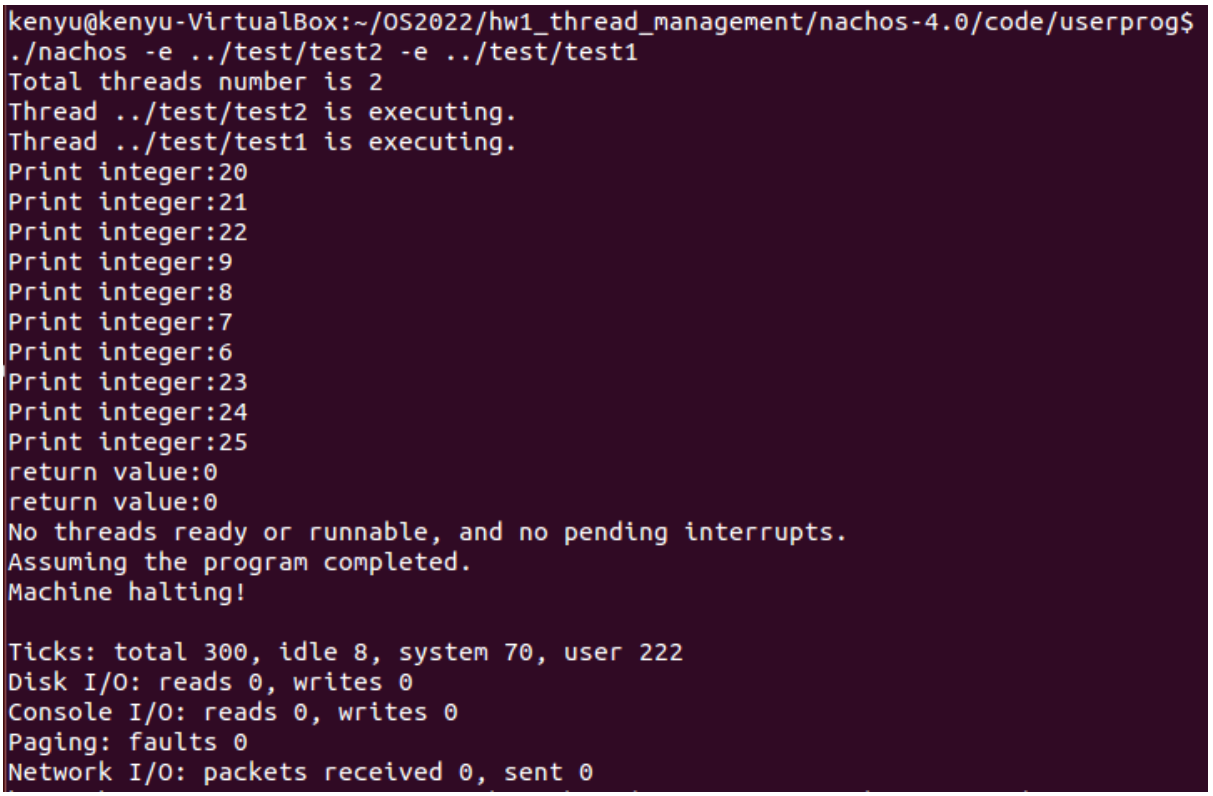
```
// TODO Implement translattion
int AddrSpace::VirtoPhys(int v_addr){
    return pageTable[v_addr/PageSize].physicalPage*PageSize + v_addr%PageSize;
}
```

Lastly, when the process is terminated, the OS will call this deconstructor to free up memory space that was occupied by the process. Hence, I need to add a code segment here to release physical pages by traversing the page table entries and updating page status.

```
AddrSpace::~AddrSpace()
{
    // TODO, need to release physical pages that occupied by process
    for (unsigned int i = 0; i < numPages; i++) {
        if (isPhyPageUsed[pageTable[i].physicalPage] == true){
            isPhyPageUsed[pageTable[i].physicalPage] = false;
            numFreePhyPage++;
        }
    }
    delete pageTable;
}
```

Experiment result

Through these adjustments, these two processes are able to work properly when executed at the same time and the result is the same as expected.



Extra Observation

Three processes multiprogramming

When I print the number of free pages after allocation, it shows that every process used 11 physical pages in total; however, the system only has 32 physical pages by default. Therefore, if I run three processes at the same time, there should be one process that can't correctly load.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/userprog$
./nachos -e ../test/test1 -e ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
number of pages of ../test/test1 is 11
numFreePhyPage after allocate = 21
Print integer:9
Print integer:8
Print integer:7
number of pages of ../test/test2 is 11
numFreePhyPage after allocate = 10
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!
```

I conduct this experiment by duplicating test1.c and name it test3.c. When I run test1, test2, and test3 together, the assertion check will throw an exception and stop new processes from loading because there isn't enough free physical page for the new process.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/userprog$
./nachos -e ../test/test1 -e ../test/test2 -e ../test/test3
Total threads number is 3
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Thread ../test/test3 is executing.
number of pages of ../test/test1 is 11
numFreePhyPage after allocate = 21
Print integer:9
Print integer:8
number of pages of ../test/test2 is 11
numFreePhyPage after allocate = 10
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
number of pages of ../test/test3 is 11
Assertion failed: line 134 file ../userprog/addrspace.cc
Aborted (core dumped)
```

If we want to solve this issue, we need to go to code/machine/machine.h and increase the “NumPhysPage” from 32 to a bigger number e.g. 64. After this minor adjustment, these three processes can execute successfully as shown below.

```
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Thread ../test/test3 is executing.
number of pages of ../test/test1 is 12
noffH.code.size = 496
noffH.initData.size = 0
noffH.uninitData.size = 0
UserStackSize = 1024
numFreePhyPage after allocate = 52
number of pages of ../test/test2 is 11
noffH.code.size = 336
noffH.initData.size = 0
noffH.uninitData.size = 0
UserStackSize = 1024
numFreePhyPage after allocate = 41
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
number of pages of ../test/test3 is 11
noffH.code.size = 336
noffH.initData.size = 0
noffH.uninitData.size = 0
UserStackSize = 1024
numFreePhyPage after allocate = 30
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
Print integer:105
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
```

Memory Space investigation

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/userprog$
./nachos -e ../test/test1
Total threads number is 1
Thread ../test/test1 is executing.
number of pages of ../test/test1 is 11
noffH.code.size = 336
noffH.initData.size = 0
noffH.uninitData.size = 0
UserStackSize = 1024
numFreePhyPage after allocate = 21
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

I print out the constitution of used pages in a process and find out the code segment occupies three pages (ceiling(336/128) = 3) and the stack occupies 8 pages (1024/128 = 8), making the total used pages be 11.

If I add 10 lines of addition operation into test1.c as shown below, the code size will become 496 byte, which implies that every “i++” takes 16 bytes of memory to save. It is bigger than I thought, but if we consider “i++” means loading memory content, addition operation, and saving the result back to memory, it’s somewhat reasonable.

```
#include "syscall.h"
main()
        {
                int i;
                i++;
                i++;
                i++;
                i++;
                i++;
                i++;
                i++;
                i++;
                i++;
                i++;
                for (i=9;i>5;i--){
                    PrintInt(i);
                }
        }
```

Reference:

[1] https://www.geeksforgeeks.org/memory-layout-of-c-program/
[2] http://tanviramin.com/documents/nachos2.pdf