

# OS Project 1: Thread Management

Student ID: R06921095

Name: Jui-Che Wu

## ➤ Motivation

Thread management is an important issue in operation system, especially in the multicore system. With the ability of multithreading, the operating system can have better use of CPU resource. Simultaneous and parallelized occurrence of tasks. In project 1, we practice the management of thread in NachOS.

First, let's see the execution result using single thread. In test1.c, expected result is to print the data in descent order from 9 to 6, the execution result is shown as follow:

```
chris@chris: ~/Desktop/OS/nachos-4.0/code/userprog
chris@chris:~/Desktop/OS/nachos-4.0/code/userprog$ ./nachos -e ../test/test1
Total threads number is 1
Thread ../test/test1 is executing.
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 200, idle 66, system 40, user 94
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
chris@chris:~/Desktop/OS/nachos-4.0/code/userprog$
```

In test2.c, the expected result is to print the data in ascent order from 20 to 25, the execution result is given as follow:

```
chris@chris: ~/Desktop/OS/nachos-4.0/code/userprog
chris@chris:~/Desktop/OS/nachos-4.0/code/userprog$ ./nachos -e ../test/test2
Total threads number is 1
Thread ../test/test2 is executing.
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 200, idle 32, system 40, user 128
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
chris@chris:~/Desktop/OS/nachos-4.0/code/userprog$
```

Now, let's execute the program using multiple thread. we execute test1.c and test2.c simultaneously with 2 threads. The expected result is to print a descent order from 9 to 6(test1.c) as well as ascent order from 20 to 25(test2.c). However, both program print the result in ascent order as show as follow:

```
chris@chris: ~/Desktop/OS/nachos-4.0/code/userprog
chris@chris:~/Desktop/OS/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.
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
Print integer:7
Print integer:8
Print integer:9
Print integer:10
Print integer:12
Print integer:13
Print integer:14
Print integer:15
Print integer:16
Print integer:16
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:21
Print integer:21
Print integer:23
Print integer:24
Print integer:25
return value:0
Print integer:26
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Thus, the problem is caused by the problem that the original NachOS doesn't deal with memory management in multiple program. If multiple programs are imported simultaneously, code segment in the same area is operated. Since each process has an address space. Thus, the plan to deal with the problem is trying to label the physical address in the main memory, and record the related physical address in the logical address. When executing a process, it will find the corresponding physical address. Then, it will not execute the same code.

## ➤ Implementation

First, in `addresspace.cc`, utilize a global function “PhyPageHandler” to record the status of main memory page, and initialize it as 0, which indicates unused. The source code is shown as follow:

```
#include "copyright.h"
#include "main.h"
#include "addresspace.h"
#include "machine.h"
#include "noff.h"
#include <typeinfo>
//-----
// SwapHeader
// Do little endian to big endian conversion on the bytes in the
// object file header, in case the file was generated on a little
// endian machine, and we're now running on a big endian machine.
//-----
bool PhyPageHandler[NumPhysPages] = {0};
```

Second, we are going to modify the code in `AddrSpace::Load`, the following code is to allocate the physical page, and clear the page to be allocated later:

```
static void setPageTablestate(int i, int j, TranslationEntry* pageTable)
{
    // cout << "hi\n";
    PhyPageHandler[j] = true;
    pageTable[i].physicalPage = j;
    pageTable[i].valid = true;
    pageTable[i].readOnly = false;
    pageTable[i].use = false;
    pageTable[i].dirty = false;
}
```

```
//code modified
pageTable = new TranslationEntry[numPages];
for(int i = 0; i < (int)numPages; i++)
{
    pageTable[i].virtualPage = i;
    int j = 0;
    for(int jj = 0; (jj < (int)NumPhysPages && PhyPageHandler[jj] == true); jj++)
        j++;
    setPageTablestate(i, j, pageTable);
}
```

Third, we are going to modify the access position. Before modification, the data is put in the virtual Address, our task is to find the physical position after mapping. We index the `pageTable` and find the corresponding physical page. The implementation source code is shown as follow:

```

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.code.virtualAddr%PageSize)]),
        noffH.initData.size, noffH.initData.inFileAddr);
}

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

```

Last but not least, after the process, we will release the resources after process in destructor. Reset the labeled page to unused.

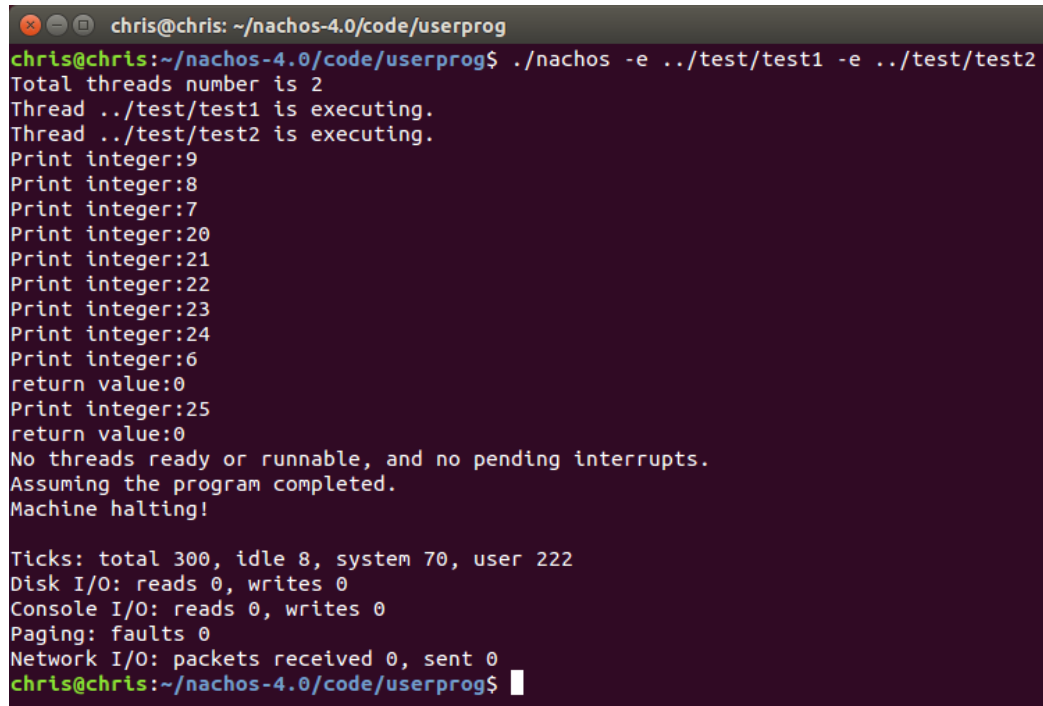
```

AddrSpace::~AddrSpace()
{
    for(size_t i = 0; i < numPages; i++)
    {
        size_t phyPage = pageTable[i].physicalPage;
        PhyPageHandler[phyPage] = false;
    }
    delete pageTable;
}

```

## ➤ Result

After the modification of the above design, the experimental result is given as follow:

A terminal window with a dark purple background and light green text. The title bar shows 'chris@chris: ~/nachos-4.0/code/userprog'. The prompt is 'chris@chris:~/nachos-4.0/code/userprog\$'. The command entered is './nachos -e ../test/test1 -e ../test/test2'. The output shows two threads executing simultaneously, printing integers from 6 to 25 in a specific interleaved order. It also shows system statistics like ticks, disk I/O, console I/O, paging, and network I/O, ending with 'Machine halting!' and a new prompt.

```
chris@chris: ~/nachos-4.0/code/userprog
chris@chris:~/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.
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
chris@chris:~/nachos-4.0/code/userprog$
```

We can see that in the output of the program that 2 thread is executing simultaneously, and the individual program: test1.c and test2.c output in a correct order. That is because the above design solves the problem of two programs operate in the same area code segment.