
MULTI-PROGRAMMING

1st Project of Operating System

Author

Name: Man-Chen Hsu

Student ID: R11942135

Due: October 19th, 2023



Contents

1	Multi-programming	3
1.1	Page Memory Management	3
1.2	Trace Code and Find the Problem	4
1.3	Solution	7
2	System Call Tracing	11

1 Multi-programming

1.1 Page Memory Management

- Logical address/virtual address: generated by CPU
- Physical address: seen by the memory module
- **Method:**
 - Divide physical memory into fixed-sized blocks called frames
 - Divide logical address space into blocks of the same size called pages
 - To run a program of n pages, need to find n free frames and load the program
 - Keep track of free frames
 - Set up a page table to translate logical to physical addresses

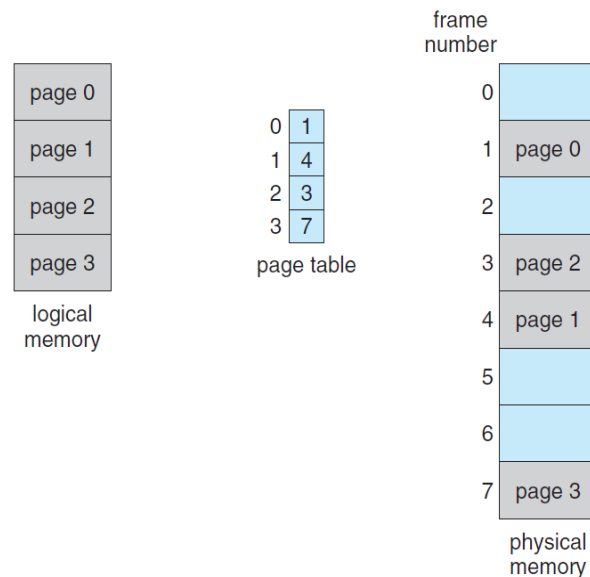


Figure 1: Paging model of logical and physical memory

- **Address Translation Scheme:**
 - Logical address is divided into parts:
 - * Page number (p): an index into a page table which contains base address of each page in physical memory, which equals to logical address/page size

- * Page offset (d): combined with base address to define the physical memory address that is sent to the memory unit, which equals to logical address % page size
- Physical address = page base address + page offset

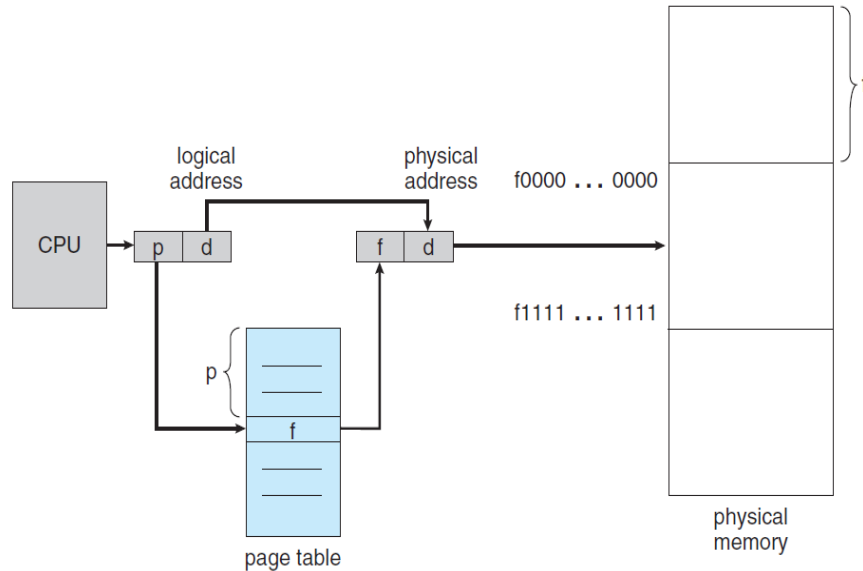


Figure 2: Paging hardware

1.2 Trace Code and Find the Problem

First, execute `../test/test1`, and we can find that the output increases.

```
gina@gina-VirtualBox:~/repo/Nachos/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
```

Then execute `../test/test2`, and we can find that the output decreases.

```
gina@gina-VirtualBox:~/repo/Nachos/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
```

But when we execute the two programs concurrently, we can find that the output of test1 becomes increasing. It seems that the state of output in test1 has been overloaded with that in test2. Therefore, I guess if the memory of the two programs has been overloaded.

```
gina@gina-VirtualBox:~/repo/Nachos/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!

Ticks: total 800, idle 67, system 120, user 613
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
```

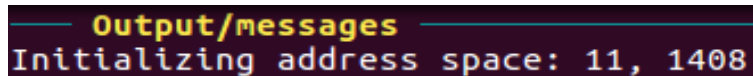
So, I trace the `../userprog/addrspace.cc`. There is a function called `AddrSpace::Load`, which will load a user program into memory from a file. Accordingly, I guess the problem may occur here. Now I am going to trace the code.

Since the data structures defining the Nachos object code is noff format, so look up `../bin/noff.h`. There are some information about address and code segment defined in this file, such as location of segment in virtual address space, executable code segment and etc. Then we can read a portion of a file through `OpenFile::ReadAt` in `../filesystem/openfile.cc`. Now we have to calculate the address space. We need to increase the size to leave room for the stack. First, calculate the size from the information read previously. And calculate the number of pages, which equals dividing size by page size and rounding up. Lastly, upgrade size as multiplying number of pages by page size.

From line 125 of file `addrspace.cc`, the code and data segments are copied into memory. So I set a break point at this line and debug with gdb. In the following pictures, I will show the result.

- thread 1:

- Initializing address space:



```
— Output/messages —
Initializing address space: 11, 1408
```

- Initializing code segment:



```
— Output/messages —
0, 336
```

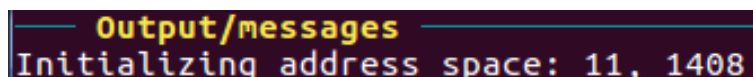
- Initializing stack pointer:



```
— Output/messages —
Initializing stack pointer: 1392
```

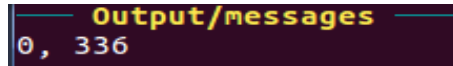
- thread 2:

- Initializing address space:



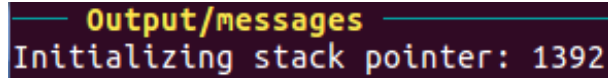
```
— Output/messages —
Initializing address space: 11, 1408
```

- Initializing code segment:



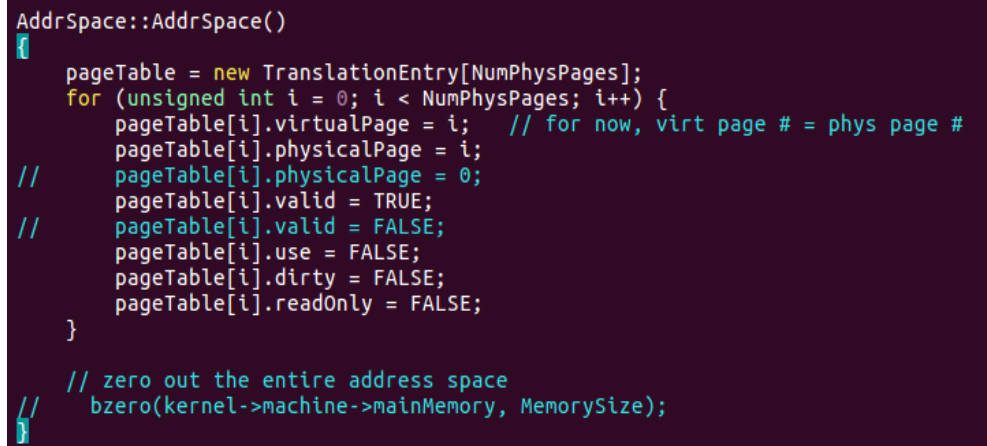
```
Output/messages
0, 336
```

- Initializing stack pointer:



```
Output/messages
Initializing stack pointer: 1392
```

From the above pictures, we can find that the virtual address and code segment of the two files are the same. Then, we should examine if the physical address of the two programs are the same. From function `AddrSpace::AddrSpace` (in `../userprog/addrspace.cc`), the physical address will be the same as the virtual address. Therefore, this would cause that the state of output of `test1` will be the same as that of `test2`.



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

1.3 Solution

First, I create a static bool array `UsedPhysicalPageState` to record the physical page that has been used. The maximum number of the used physical page must equal to the number of physical page.

```

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

    static bool UsedPhysicalPageState[NumPhysPages];

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
};

```

In the beginning of `addrspace.cc`, we should initialize the `UsedPhysicalPageState` as zero.

```

18 #include "copyright.h"
19 #include "main.h"
20 #include "addrspace.h"
21 #include "machine.h"
22 #include "noff.h"
23
24 bool AddrSpace::UsedPhysicalPageState[NumPhysPages]={0};
25

```

Then we are going to allocate the physical page. We would like to throw the address into the empty page. When the i^{th} physical page is used, `UsedPhysicalPageState[i]` will be 1. And our goal is to find one physical page that is empty and put the data into that page.


```

99 bool
100 AddrSpace::Load(char *fileName)
101 {
102     OpenFile *executable = kernel->fileSystem->Open(fileName);
103     NoffHeader noffH;
104     unsigned int size;
105
106     if (executable == NULL) {
107         cerr << "Unable to open file " << fileName << "\n";
108         return FALSE;
109     }
110     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
111     if ((noffH.noffMagic != NOFFMAGIC) &&
112         (WordToHost(noffH.noffMagic) == NOFFMAGIC))
113         SwapHeader(&noffH);
114     ASSERT(noffH.noffMagic == NOFFMAGIC);
115
116     // how big is address space?
117     size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
118           + UserStackSize; // we need to increase the size
119                           // to leave room for the stack
120     numPages = divRoundUp(size, PageSize);
121
122     pageTable=new TranslationEntry[numPages];
123     for(unsigned int i=0,j=0;i<numPages;i++){
124         pageTable[i].virtualPage=i;
125         while(j<NumPhysPages && AddrSpace::UsedPhysicalPageState[j]==true)
126             j++;
127         AddrSpace::UsedPhysicalPageState[j]=true;
128         pageTable[i].physicalPage=j;
129         pageTable[i].valid=true;
130         pageTable[i].use=false;
131         pageTable[i].dirty=false;
132         pageTable[i].readOnly=false;
133     }
134
135     // cout << "number of pages of " << fileName<< " is "<<numPages<<endl;
136     size = numPages * PageSize;
137
138     ASSERT(numPages <= NumPhysPages); // check we're not trying
139                                       // to run anything too big --
140                                       // at least until we have
141                                       // virtual memory
142
143     DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);
144

```

And don't forget to delete the address space created in `AddrSpace::AddrSpace`.

```

56 AddrSpace::AddrSpace()
57 {
58     // pageTable = new TranslationEntry[NumPhysPages];
59     // for (unsigned int i = 0; i < NumPhysPages; i++) {
60     //     pageTable[i].virtualPage = i; // for now, virt page # = phys page #
61     //     pageTable[i].physicalPage = i;
62     //     pageTable[i].physicalPage = 0;
63     //     pageTable[i].valid = TRUE;
64     //     pageTable[i].valid = FALSE;
65     //     pageTable[i].use = FALSE;
66     //     pageTable[i].dirty = FALSE;
67     //     pageTable[i].readOnly = FALSE;
68     // }
69
70     // zero out the entire address space
71     // bzero(kernel->machine->mainMemory, MemorySize);
72 }
73

```

When we want to obtain the physical address, we should calculate the page base address and page offset first based on the formulation of physical address of paging mentioned above. Divide virtual address by page size, we can get which page the physical address is in. According to the page table, find the physical page. Then multiply it by page size, we will obtain the page base address. For page offset, divide virtual address by page size, then the remainder is page offset.

```

145 // then, copy in the code and data segments into memory
146 if (noffH.code.size > 0) {
147     DEBUG(dbgAddr, "Initializing code segment.");
148     DEBUG(dbgAddr, noffH.code.virtualAddr << " ", " << noffH.code.size);
149     executable->ReadAt(
150         &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage*PageSize+(noffH.code.virtualAddr%PageSize)]),
151         noffH.code.size, noffH.code.inFileAddr);
152 }
153 if (noffH.initData.size > 0) {
154     DEBUG(dbgAddr, "Initializing data segment.");
155     DEBUG(dbgAddr, noffH.initData.virtualAddr << " ", " << noffH.initData.size);
156     executable->ReadAt(
157         &(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage*PageSize+(noffH.code.virtualAddr%PageSize)]),
158         noffH.initData.size, noffH.initData.inFileAddr);
159 }
160
161 delete executable;          // close file
162 return TRUE;               // success
163 }

```

When the execution finishes, the physical page should be released, then other programs could use this page table.

```

AddrSpace::~AddrSpace()
{
    for (unsigned int i=0;i<numPages;i++){
        AddrSpace::UsedPhysicalPageState[pageTable[i].physicalPage]=false;
    }
    delete pageTable;
}

```

Then the result would be correct.

```

gina@gina-VirtualBox:~/repo/Nachos/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

```

2 System Call Tracing

Based on the slide of Homework 1, I first set a breakpoint at `Machine::Run()` (in `../machine/mipssim.cc:52`). It simulate the execution of a user-level program on Nachos. It is called by the kernel when the program starts up and never returns. This routine is re-entrant, in that it can be called multiple times concurrently (one for each thread executing user code). In the line 60 of `Machine::Run()`, `kernel->interrupt` will be set as `UserMode` (`void setStatus(MachineStatus st) status = st;` in `interrupt.h`). Then we get into an infinite loop and first execute `Machine::OneInstruction()` (in `../machine/mipssim.cc:115`).

The `Machine::OneInstruction()` is to execute one instruction from a user-level program. If there is any kind of exception or interrupt, we involve the exception handler, and when it returns, we return to `Run()`, which we will re-involve us in a loop. On a syscall, the OS software must increment the PC so execution begins at the instruction immediately after the syscall. The routine is re-entrant, in that it can be called concurrently. We always re-start the simulation from scratch each time we are called (or after trapping back to the Nachos kernel on an exception or interrupt), and we always store all data back to the machine registers and memory before leaving. Nachos kernel could control our behavior by controlling the contents of memory, the translation table, and the register set. In line 557 of `Machine::OneInstruction(Instruction instr*)`,

there exists one case related to system call. In `case OP_SYSCALL`, we execute `RaiseException(SyscallException, 0)`. That is, we will execute `Machine::RaiseException` (in `../machine/machine.cc:100`), when an interrupt (or exception) happen.

In `Machine::RaiseException`, the mode will be transferred into Nachos kernel mode from user mode, since the user program either invoked a system call, or some exception occurred (such as the address translation failed). Prior to the transformation, the virtual address, causing the trap, will be recorded (seen in line 105). Then anything in progress will be finished. In line 107, `kernel->interrupt->setStatus(SystemMode)` means that we are trapped into SystemMode (Kernel mode). From `which` variable, we can find which type of the kernel trap happens. Here, the type is `SyscallException` (which=`SyscallException`). Now we are going to execute `ExceptionHandler(ExceptionType which)` (in `../userprog/exception.cc:51`).

`ExceptionHandler` is the entry point into the Nachos kernel. It is called when a user program is executing, and either does a syscall, or generates an addressing or arithmetic exception. Since the type of the system call code (e.g. `SC_Halt`, `SC_PrintInt` and etc) are recorded in register `r2`, so first we have to read it through `Machine::Register` (in `../machine/machine.cc:205`). And we get `type=0`, so we will execute `SC_Halt`. Then `Interrupt::Halt()` (in `../machine/interrupt.cc:233`) will be executed.

`Interrupt::Halt` is to shut down Nachos cleanly, printing out performance statistics. In `Statistics::Print()` (in `../machine/stats.cc:33`), `totalTicks`, `idleTicks`, `userTicks`, `numDiskReads` and etc. (performance statistics) will be printed. Then we will get back to `Interrupt::Halt` and delete kernel, which will never return. Then we will get into `UserProgKernel::~~UserProgKernel` (in `../userprog/userkernel.cc:72`) to de-allocate global data structures, such as `fileSystem`, `machine` and `synchDisk` (if defined). And we also get into `Machine::~~Machine()` (in `../machine/machine.cc:83`) to de-allocate the data structures used to simulate user program execution. Since Nachos is halting, we get into `ThreadedKernel::~~ThreadedKernel()` (in `../threads/kernel.cc:70`) to de-allocate some global data structures, such as `alarm`, `interrupt`, `scheduler`, and `stats`. And the code will exit. Then the process of system call is done.