Operating System MP2

Report

Team 47

2020/11/15

Team Members:

A discussion between two members was held on the implementation and the code tracing.

廖允寬 (Liao) 106034071 - Worked on Part II and Part III of the report.

洪紫珈 (Evelyn) 107062174 - Worked on Part I and Part III of the report.

PART I

Explain “threads/kernel.cc Kernel::ExecAll()”, “threads/thread.cc thread::Sleep”, and “machine/mipssim.cc Machine::Run()”.

In ExecAll(), we use a for loop to execute all the files that await execution.

```
for (int i=1;i<=execfileNum;i++)
       { int a = Exec(execfile[i]); }
```

During each file execution, we use a prepared threadNum to create a new thread and place it in a thread list. Then it asks for a new address space by ‘new AddrSpace()’which sets up the virtual address for this thread and the translation from program memory to physical memory.

```
t[threadNum]->space = new AddrSpace();
```

This thread is then executed with ForkExecute which distributes an execution stack with stackAllocate(func, arg).

```
t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
                                       and
StackAllocate(func, arg);
```

Disable the interrupts and set this thread(process) to ready so it will be scheduled to execute, then return to the original level which enables the interrupts.

```
oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
```

Then ‘threadNum++’ prepares the next available thread number for the next thread.

In the ForkExecute function, there’s a statement ‘t->space->Execute(t->getName())’, and Execute() calls ‘kernel->machine->Run()’.

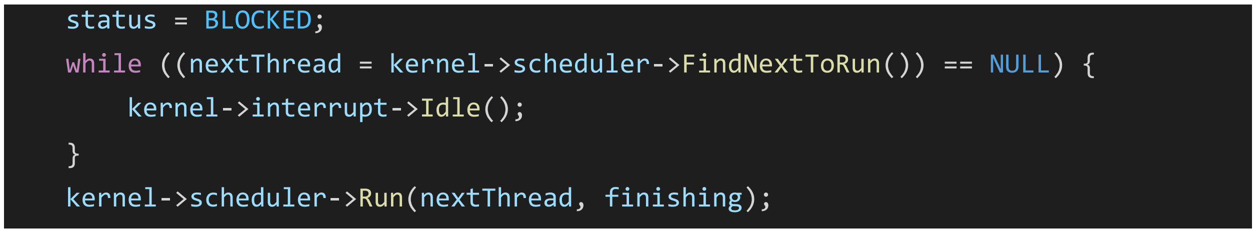
```
t->space->Execute(t->getName());
                                       and
kernel->machine->Run();
```

This ‘run()’ function is called by the kernel when a thread starts execution. After setting the status to ‘usermode’, it uses a for(;;) loop to run through all the instructions one by one with a ‘OneInstruction()’ function and updates the time by ‘OneTick()’. It is important to note that it is a reentrant function, implying that it can be shared by many processes at the same time and can be interrupted by another process to restart the function.

After all files are executed, ExecAll() calls 'currentThread->Finish()’ which calls Sleep(true) after disable the interrupt.

```
currentThread->Finish();
                                       and
Sleep(TRUE);
```

In Sleep(), we first set this thread to the mode ‘BLOCKED’ which indicates it’s completed and will not run again. Then, the function uses a while loop to check if there is another thread to be executed, else it will stay in Idle state. Otherwise, it breaks the loop and runs the next thread.



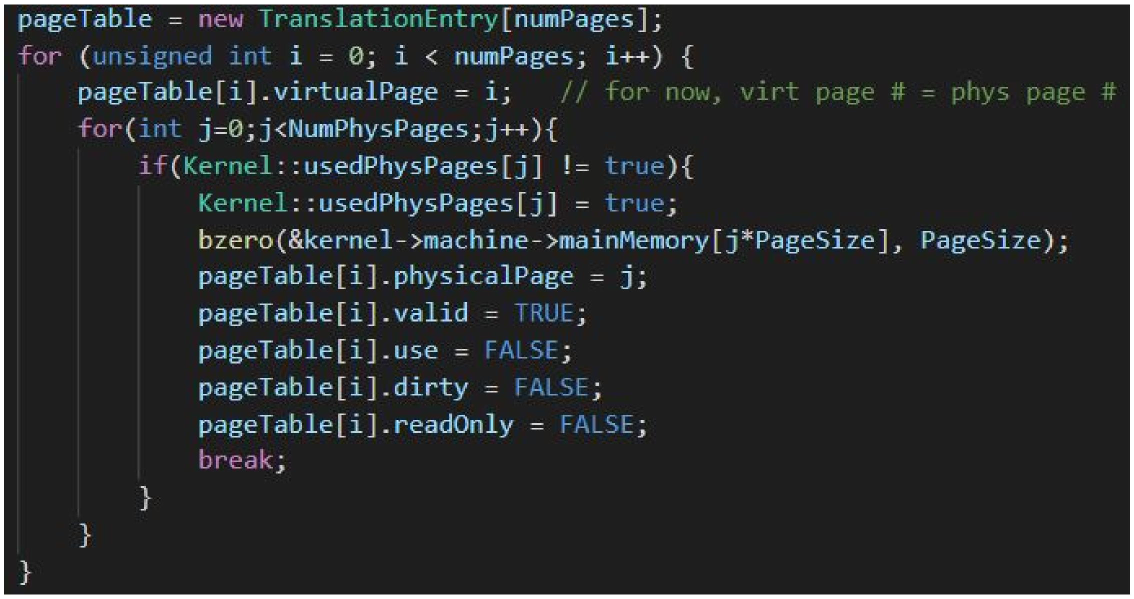
```
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
```

PART II

Explain your implementation as requested in Part II-2.

這次作業我們修改了addrspace.cc和kernel.h兩個檔案

1. Addrspace.cc
   在建構子的地方，我們把原本建立pagetable的程式碼註解掉，因為在load的時候才能知道該程式的大小，按照個這個大小去分配pagetable才能夠節省空間，而不是全部都用NumPhysPages，且這樣context switch發生時，才不會有pagetable混用的情況。
   所以我們把pagetable設定的程式碼搬到Load()裡面，其大小是根據前面算出的numPages做分配空間。由for迴圈去找現在還有哪個physical page可以用，就占用並且存入對應的資訊，usedPhysPage是一個在kernel.h新宣告的array。



```
pageTable = new TranslationEntry[numPages];
for (unsigned int i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    for(int j=0;j<NumPhysPages;j++){
        if(Kernel::usedPhysPages[j] != true){
            Kernel::usedPhysPages[j] = true;
            bzero(&kernel->machine->mainMemory[j*PageSize], PageSize);
            pageTable[i].physicalPage = j;
            pageTable[i].valid = TRUE;
            pageTable[i].use = FALSE;
            pageTable[i].dirty = FALSE;
            pageTable[i].readOnly = FALSE;
            break;
        }
    }
}
```

   接著我們根據上課教過的virtual address to physical address轉換的方法，算出phtsical address，放到下方Readat main memory的位置，這樣才會讀取到正確的資訊。

```
unsigned int PhysAdr =
pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize +
(noffH.code.virtualAddr%PageSize);
```

```
executable->ReadAt(
        &(kernel->machine->mainMemory[PhysAdr]),
        noffH.code.size, noffH.code.inFileAddr);
```

2. Kernel.h:
在class kernel public裡加上usedPhysPages，以方便我們紀錄哪寫physpage用過哪些還沒。

```
static bool usedPhysPages[NumPhysPages];
```

PART III

Explain how NachOS creates a thread(process), load it into memory and place it into the scheduling queue as requested in Part II-1.

1. How NachOS allocates the memory space for new thread(process)?

   NachOS allocates the memory space for a new thread with the 't[threadNum]->space = new AddrSpace()' which is in Kernel::Exec(char* name). This function initializes the address space by giving the virtual address. And with the address, we can translate it to a physical address that allows memory access.

2. How Nachos initializes the memory content of a thread(process), including loading the user binary code in the memory?

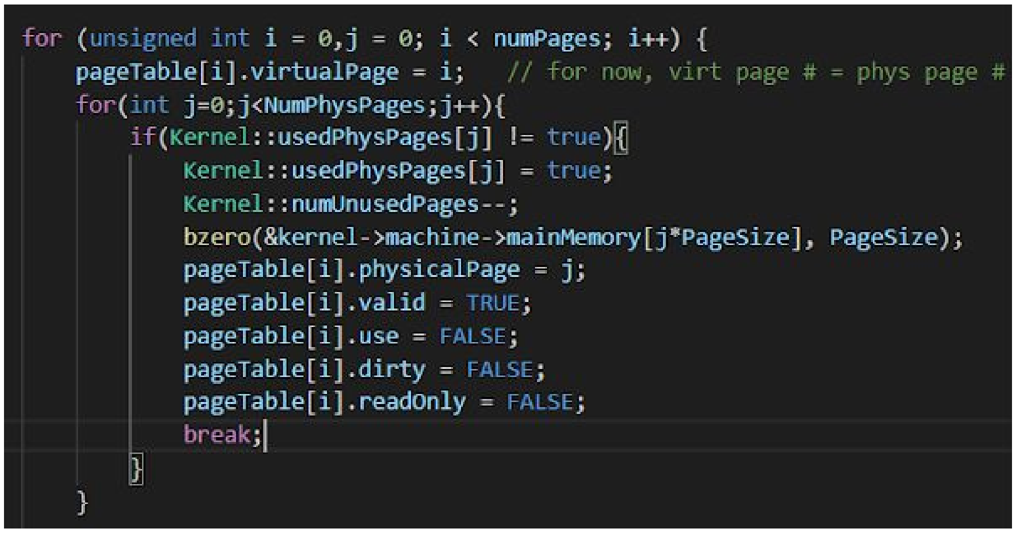
   在AddrSpace::Load()裡面，在建立pagetable時，我們用下面這行在每次分配一個page的時候做初始化。

```
bzero(&kernel->machine->mainMemory[j*PageSize], PageSize);
```

根據Load()傳入的filename去開啟user program，由noffH.code.virtualAddr與pagesize等資訊計算出physical address，有了physical address，就能用executable->ReadAt()把code segment與data segment load進memory裡面了。

3. <u>How NachOS creates and manages the page table?</u>

Originally, a page table was created by 'new TranslationEntry[NumPhysPages]' which existed in the constructor 'AddrSpace::AddrSpace()'. Now we remove this creation and place it in the load(). Here, we then start assigning the elements in this newly created page table as follows. We will loop through the physical pages and find an unused one to do a physical-page assignment.

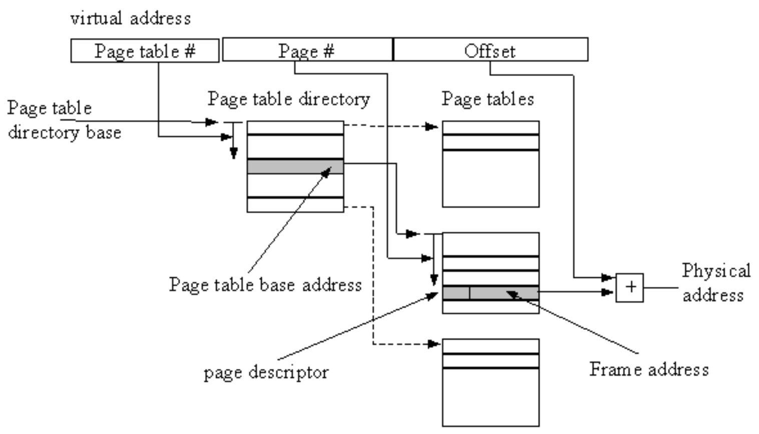


```
for (unsigned int i = 0,j = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    for(int j=0;j<NumPhysPages;j++){
        if(Kernel::usedPhysPages[j] != true){
            Kernel::usedPhysPages[j] = true;
            Kernel::numUnusedPages--;
            bzero(&kernel->machine->mainMemory[j*PageSize], PageSize);
            pageTable[i].physicalPage = j;
            pageTable[i].valid = TRUE;
            pageTable[i].use = FALSE;
            pageTable[i].dirty = FALSE;
            pageTable[i].readOnly = FALSE;
            break;
        }
    }
```

For the explanation of releasing a pageTable, please refer to the implementation (Part II).

4. <u>How NachOS translates address?</u>

Originally, the NachOS uses virtualAddr to access the memory, however, physicalAddr should be used instead. So even if NachOS did implement virtual memory, it would not perform multi-programming. Therefore, we have managed to use translated physicalAddr to access the memory, we use the following method:



```
unsigned int PhysAdr =
pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize +
(noffH.code.virtualAddr%PageSize);
```

5. <u>How NachOS initializes the machine status (registers, etc) before running a thread(process)?</u>

   When a thread is created, the constructor is called. And the constructor initializes values for this thread, for example, name, ID, status, space, and etc.

   In a file execution, AddrSpace::Execute() is called to run current thread. Here, Execute() sets the initial register values by InitRegisters() and loads page table registers by RestoreState(). The InitRegisters() function initializes all registers to 0, also PCReg, NextPCReg, and the location for StackReg.

```
AddrSpace::InitRegisters() {
machine->WriteRegister(i, 0);
machine->WriteRegister(PCReg, 0);
machine->WriteRegister(NextPCReg, 4);
machine->WriteRegister(StackReg, numPages * PageSize - 16);
}
```

   The RestoreState() function stores the current page table and the current number of pages to the kernel on a context switch. Now, all initialization before Run() is completed.

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

6. <u>Which object in NachOS acts the rule of process control block?</u>

   In NachOS, the 'Thread' class acts as the PCB of a process. It stores the status, id, and some other identities of the thread. Also, it provides functions to several thread operations.

7. <u>When and how does a thread get added into the ReadyToRun queue of NachOS CPU scheduler?</u>

   In the Fork() function, there is a ReadyToRun() function that passes the thread to it.

```
scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
```

Then, there are

```
thread->setStatus(READY);
readyList->Append(thread);
```

which sets the thread status to ready and adds it into the ready queue of CPU scheduler.