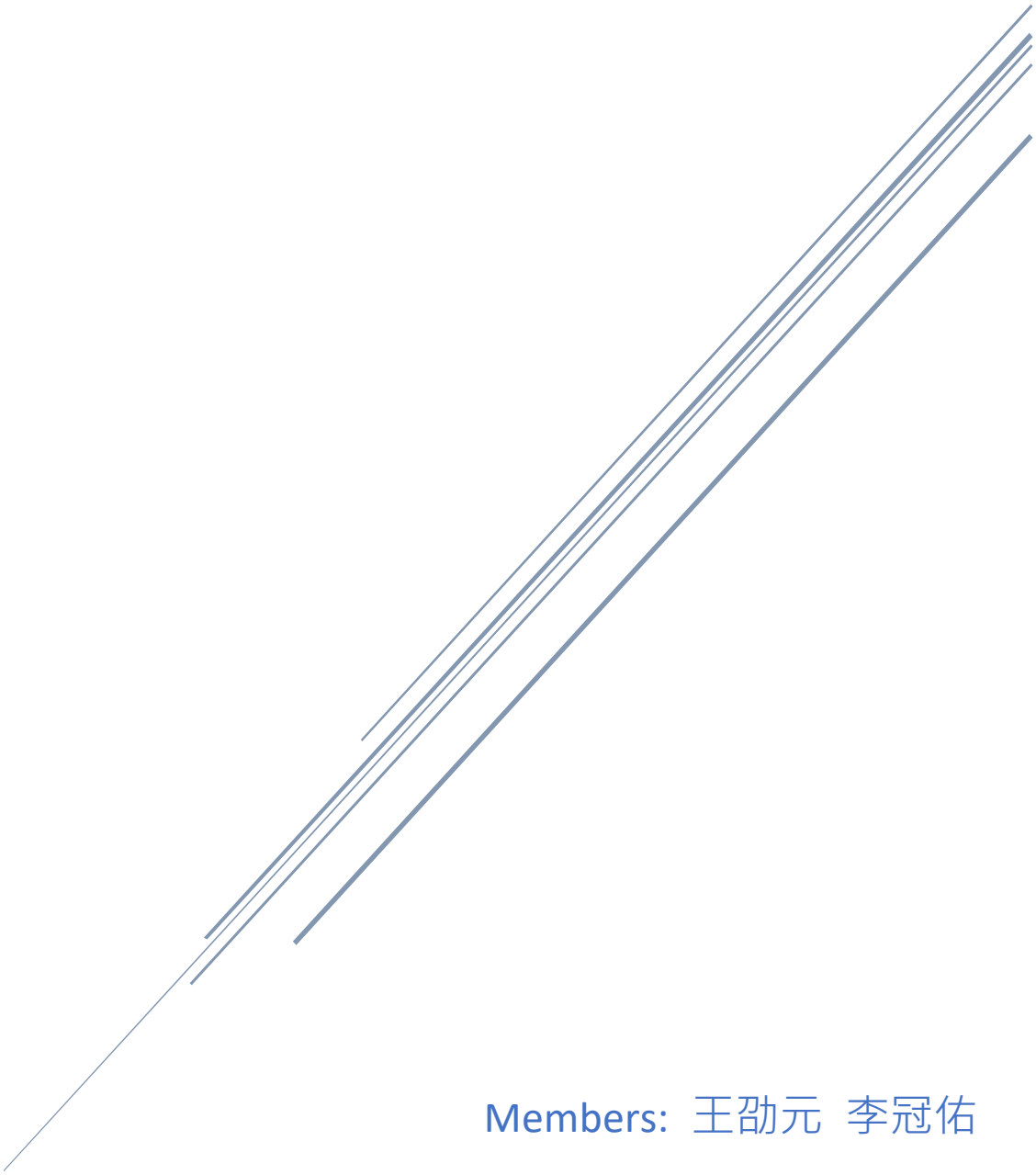
MP2:MUTI-PROGRAMMING

Report



Members: 王劭元 李冠佑

Contribution: 50 % 50 %

Part 1 Implementation

1. kernel.h

   在這個 file 裡，由於 spec 要我們把紀錄哪個 memory frame 被用了的資料結構記在這邊，所以我們先宣告了一個 private 的 PageUsed[NumPhysPages]的陣列，當裡面的值被設為 1 時，代表此 frame 的 memory 已經被用了。

   此外我們再宣告了另一個 private 的 freepages，代表現在還有多少 frame 是還沒被使用的。

   ```
   int freepages;
   int PageUsed[NumPhysPages];
   ```

2. kernel.cc

   這邊的 implement，我們是修改 ExecAll()，在要 new 一個新的 AddrSpace 的時候，我們會多把我們的 PageUsed 和指到 freepages 的指標往下傳遞給 AddrSpace。

   ```
   t[threadNum]->space = new AddrSpace(PageUsed, &freepages);
   ```

3. AddrSpace.h

   在 AddrSpace 這邊，我們先修改了他的 constructor，使他可以接收兩個參數，一個為 PageUsed 一個為 freepages 的指標。

   ```
   AddrSpace(int *UsedPage, int* freepages);
   ```

   接著我們也在這邊多宣告了兩個 public 的指標，PhysPagesUsed 是 for 接 kernel 傳下來的 UsedPage 另一個為接 kernel 傳下來的 freepages。

```
int *PhysPagesUsed;
int *FreePhysPages;
```

4. AddrSpace.cc

```
AddrSpace::AddrSpace(int *UsedPage, int* freepages)
{
    PhysPagesUsed = UsedPage;
    FreePhysPages = freepages;
    // zero out the entire address space
    //bzero(kernel->machine->mainMemory, MemorySize);
}
```

首先在 constructor 的地方，可以看到我們把原本在這邊把 virtual page 和 frame 連起來的地方被拿掉了，並且也不在這裡 initialize 整個 memory 了。在這邊就只有將 PhysPagesUsed 和 kernel 傳下來的

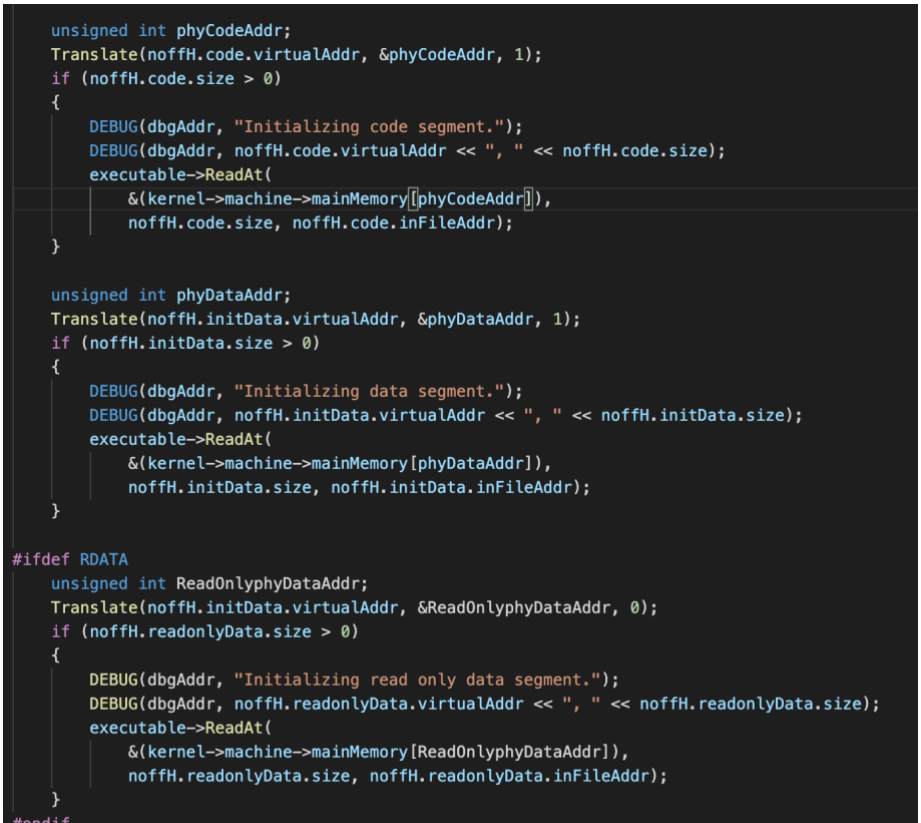
```
// Team42 Add
pageTable = new TranslationEntry[numPages];
for (int i = 0, j = 0; i < numPages; i++)
{
    pageTable[i].virtualPage = i;
    while (j < NumPhysPages && PhysPagesUsed[j])
        j++;
    (*FreePhysPages)--;
    bzero(&kernel->machine->mainMemory[j * PageSize], PageSize);
    PhysPagesUsed[j] = TRUE;
    pageTable[i].physicalPage = j;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}
// Team42 Add
```

UsedPage 以及將 FreePhysPages 和 kernel 傳下來的 freepages 接起來。

這邊是我們在 Load 裡面多寫的部分，當今天要從 file 把 program load 進 memory 時，就會做到這一步。整體上大致的架構就是把原本在 constructor 做的事搬下來做，並且不與整個 memory 連結。只把需要的 page 和 frame 去連接。

首先我們會跑一個我們總共需要的數量的迴圈，接著為了要在 memory 中找到 free frame，我們就跑一個 while 迴圈去掃過 PhysPagesUsed，直到找到空的 frame。找到後就把現有的 freeframes 數量減一，並將 memory 的這個 frame 用 bzero initialize。然後把這個 entry 的 physical page bind 上這個 index，並且把 valid bit 設成 TRUE，use, dirty, readOnly 的 bits 設為 FALSE 後，即可繼續把下一個 page 去 bind frame。

再來，Load 裡另一個要改的地方是當要把 binary code load 進 memory 時，我們必須對原本的 virtual address 做些改變，不能單單用 virtual address



```
    unsigned int phyCodeAddr;
    Translate(noffH.code.virtualAddr, &phyCodeAddr, 1);
    if (noffH.code.size > 0)
    {
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[phyCodeAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }

    unsigned int phyDataAddr;
    Translate(noffH.initData.virtualAddr, &phyDataAddr, 1);
    if (noffH.initData.size > 0)
    {
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[phyDataAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

#ifdef RDATA
    unsigned int ReadOnlyphyDataAddr;
    Translate(noffH.initData.virtualAddr, &ReadOnlyphyDataAddr, 0);
    if (noffH.readonlyData.size > 0)
    {
        DEBUG(dbgAddr, "Initializing read only data segment.");
        DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[ReadOnlyphyDataAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif
```

我們分別創了三個 physical address，phyCodeAddr, phyDataAddr 和 ReadOnlyphyDataAddr，然後去 call AddrSpace 的 translate，以 pass by reference 的方式獲得 virtualAddr 所轉成的 physicalAddr，如果是 readOnly 的話最後一個引數就放 0，其他放 1，這樣就可以將 code segment, data segment read 進 mainMemory 裡對應的位置。

Part 2: Trace code

我們這部分是以 implement 前的 code 來解釋整個流程

1. *Which object in Nachos act the role of process control block?*

最一開始會藉由 execfile name 和一個 threadNum 去 new 一個 thread control block(process control block)，這個 control block 就是 Thread.h 的 class thread，藉由 constructor 去做初始化，在 private 部份會存這個 thread 的 current stack pointer 以及 machine state register，由於 SWITCH 的關係，所以這兩者的順序不能亂動，private 還存有 thread status 和一個指到 bottom of the stack 的 pointer，在 public 裡，CheckOverFlow()會去檢查 stack 是否 overflow，Fork()是拿來讓 thread

在要執行時 run procedure 去把 binary code load 到 memory 裡並將 pageTable 設定好後才去 execute，Yield()是在判斷是否有 thread 可以執行 context switch，Sleep()會將 thread block 住，等待有可執行的 thread 再去執行，Begin()和 Finish()會在 thread 的開始和結束去 call 到，StackAllocate()會去分配一個 stack 給 thread，並 initialize machineState register，space 代表這個 thread 會 run 的 user code，SaveUserState 和 RestoreUserState 是在在 context switch 去儲存舊的 thread state 以及 swich 回來後將舊的 thread state restore 回來

2. *How Nachos initializes the machine status(register, etc) before running a thread(process)?*

首先，在 Thread::StackAllocate()的最下面，可以看到有幾行在對 Machine status 做一些初始的設定，告訴 Thread，若要開始執行，要先從哪邊開始執行（設定 Threadroot）。並在要 SWITCH 換別的 Thread 執行時，在 switch.s 中，可以看到有幾行是在做先把要被換掉的 thread 的 SP, callee-saved register, frame pointer, PC 都先存下來的地方，並且接下來也把要被換進去的 thread 相對應的 register 的值 load 進真的 machine register。然後最後就會 jump 到 ra 這個 register 內的位置，開始執行，也就是 Threadroot。

故每次在要 running 一個 Thread 前，都會透過上面的流程先把一些

register initialize，並且也會在 scheduler::Run()的最後去 RestoreUserState 和 RestoreState，分別去將這個 thread 的 register 和 pageTable restore 回來。而被 SWITCH 掉的 thread 在被 switch 前也有透過 SaveUserState 和 SaveState 把資訊記錄下來。

3. *How Nachos allocate the memory space for new thread(process)?*

這裡會由這個 new 出來的 thread 去 call AddrSpace 去 allocate memory space，讓 thread 可以 run user code，在 AddrSpace 裡會去 new 一個 NumPhysPages 大小的 TranslationEntry 來 create pagetable，裡面包括有 virtualPage, physicalPage, valid bit, readOnly bit, use bit 和 dirty bit 以供之後來管理 page table，接著跑回圈讓 physical 和 virtual page 一對一對應(uni-programming)，將 valid, readOnly, use, dirty 都初始設為 false，valid bit 代表這個 page entry 有沒有被 initialize 過，readOnly bit 表示是否允許這個 page 的內容被修改，use bit 用來判斷這個 page 有沒有被 reference 或 modify 過，dirty bit 則是拿來看 page 是否被 modify 如果有的話就必須將 disk 裡的內容更改

4. *How Nachos initialize the memory content of a thread(process) including loading the user binary code in the memory?*

StackAllocate 完後的 Thread 會存在 readyList 中，當 context switch 輪到這個 thread 執行時，Nachos 會去執行它 fork 的 procedure 去 load

這個 thread，在 load 的時候它會由 ReadAt 將 binary code load 到 mainMemory 裡面，原本使用的方法是直接取 virtual address(因為 virtual page 和 physical page 是一對一)，但之後在 implement multi-programming 時候，必須對 virtual address 作些轉換，詳細部分會在 implement 提到，如此一來就可以將 code segment 和 data segment load 進 mainMemory 裡，load 執行完後就會去執行 execute()，把 register 初始化和 restore 這個 thread 的 page table，最後就可以跳到 user program 去執行指令。

5. *When and how does a thread get added into the ReadyToRun queue of Nachos CPU Scheduler?*

在 Thread 去 call Thread::Fork()後，會讓 caller 和 callee 可以同時運行，接著有兩件事情要做，第一是先去 Allocate 一個 stack 出來，第二是要 initialize 這個 stack，讓此 thread 在 context switch 時，可以開始 run procedure。而這兩件事情的完成，是在 Thread::StackAllocate()中，以

```
machineState[PCState] = (void *)ThreadRoot;
machineState[StartupPCState] = (void *)ThreadBegin;
machineState[InitialPCState] = (void *)func;
machineState[InitialArgState] = (void *)arg;
machineState[WhenDonePCState] = (void *)ThreadFinish;
stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
```

下可以看到在這兩處分別去 allocate 出一個 stack，以及去將一些 state information 放進 state register 中。

ThreadRoot 是在 execution stack 上的第一個 frame，會去 call ThreadBegin，代表即將開始運作。若此 thread 執行的 process 有 return 的時候，就會 call ThreadFinish 來讓這個 Thread 終止。

以上做完後，就代表一個 Thread 已經準備好可以被 run 了，那這時候 return 回 Fork()後，就會由 scheduler 去 call ReadytoRun()，在這裡面，就會把 Thread 的 Status 設定好，設成 Ready 後，就會將此準備好的 Thread 放進去一個 readyList 裡。這樣 scheduler 在挑要 run 哪個 Thread 時，就會看到他了。

6. How Nachos create and manage page table?

在 AddrSpace 的建構子中，由於尚未被我們 implement 的 Nachos 只能 uniprogramming，所以就直接創建出一個 memory size 的 page table。Initialize page table 的部分，就很直接地將每一個 page 對到實體上每一個 memory 的 frame，並且將 valid bit 設成 TRUE，use, dirty, readOnly bit 都設為 FALSE。

而在 manage 上，當今天有 illegal page access 時，valid bit 會阻止 user 去任意 access，而 use bit 和 dirty bit 可以提供這個 page 有沒有被讀寫過。若 readOnly bit 有被 set 的話，那這個 page 也不會被

modified。

7. How Nachos translate address?

在 translate.cc 中的 Machine::Translate()可以看到 Nachos 是如何 translate address 的。

首先會先去檢查傳進來的 virtual address 有沒有什麼 alignment 上的問題，若沒有也會檢查是否有 TLB 或 Pagetable 其中一項。

接著用 vpn = (unsigned)virtAddr / PageSize 來換算得到 page number。並用(unsigned)virtAddr % PageSize 來得到 offset。

接下來若沒有 TLB 那就會直接檢查 pagetable，在開始前會檢查一下 vpn 有沒有超過 pagetable 的 size，以及是不是 valid。若有錯誤即可直接 return exception 回去。若沒錯誤就可以 access 到 pagetable 拿到 entry，最後可以再從 entry 裡拿出 frame number。

若有 TLB 的話，那就會掃過整個 TLB 看有沒有在 TLB 內的 page 的 page number 跟現在要找的一樣，若有一樣就可以找到 entry。若沒有，就會 return Pagefault 回去(命名問題，其實這裡會去處理 TLB fault)。

拿到 entry 後，就可以判斷若此 page 為 readonly 但又要 write 的話，就可以 return ReadOnlyException 回去。

若都沒碰到問題，就可以去取出 frame number。但若 frame number 大於 physical memory size 的話，一樣會 return exception 回去，度過這些重重關卡後，再把 use, dirty bits 設定一下後，終於可以正式拿到 physical address 了，最後就會 return no exception 回去。

統整：

這邊做個總整理，整個 thread 產生和執行的流程是先 create 出一個 thread，並且 create 一個 address space 來讓這個 thread 可以跑 user code，接下來他就會去 fork procedure，產生一個 stack 並 initialize machineState register，完成後就會被放進 ready queue 裡等待被執行，等到我們需要的 thread 都被 create 完後，會先將 kernel thread finish，所以此 kernel thread 會去執行到 Sleep()的部分。

接著去找尋其他可以執行的 thread，當 scheduler 找到合適的 thread 要去 context switch 時，輪到的這個 thread，它就會在 switch.s 中找到 threadroot，並接著跑 threadBegin 等等，然後去 run ForkExecute()，在這裡面會去呼叫自己 AddrSpace 的 load()把 binary code load 進 memory 裡，接著就可以去 execute()裡 init register 和 restore state，最後就會跳到 machine->run()的地方，開始要去做 oneinstruction()及執行 oneTick()。這樣就可以抓到此 Thread 要執行的程式。