Operating systems
MP2

Team 67

108062128 林育丞

108062135 呂佳恩

分工表

林育丞	Trace Code, Question 回答整理, Report 整理
呂佳恩	Trace Code, Implementation, Report 整理

I. Trace Code

Threads/thread.cc

(1) Thread::Sleep(bool finishing)

在兩個狀況下，Sleep() 會被呼叫：

1. Current thread is finished, the argument passed in will be "TRUE"
2. Current thread blocked waiting on a synchronization variable, the argument passed in will be "False"

Sleep 會進入一個判斷式，若有新的 Thread 在 Scheduler 當中，會 Run, 若沒有，FindNextToRun()會回傳 NULL，此時進入 While 迴圈，讓當前的 Thread 進入 Idle 模式。

(2) Thread::StackAllocate(VoidFuntionPtr func, void *arg)

1. Func 是指 procedure to be forked (&ForkExecute)
   arg 是將要被 passed through 的 arguments (指向 thread 指標)
2. Call AllocBoundedArray( ) allocate 所需要的空間
3. **#ifdef** directive allows for conditional compilation
4. 利用下面的 code 設定 correct Machine state
   (實際執行情形不一定如下，要考慮編譯的設定)

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

5. 將 PC 設定成指向 Threadroot routine，Thread 的執行並非在 user 提供的 routine 內，而是藉由 Assembly language 來讓它執行

(3) Thread::Finish ( )

1. 當一個 Thread 完成所有該做的事情時，由 ThreadRoot 呼叫結束
2. 將 Interrupt 設定為 Off，因為 Sleep 當中，在 OS 將 thread 從 Ready List 到 switch 至他開始執行當中，不能有 interrupt 使兩者當中出現 time slice
3. 呼叫 Sleep 並傳入 finishing 為 TRUE

(4) Thread::Fork (VoidFuntionPtr func, void *arg)

1. Func 是指 procedure 在 thread 要開始執行後的 procedure address (在這情況下，是 ForkExecute 的函式位址)
2. 呼叫 StackAllocate( ) 讓其將空間以及初始化做完
3. Set interrupt to off
4. Put current thread into scheduler for run **(ReadyToRun)**
5. Set interrupt back to the original value

(5) ThreadRoot

1. 以 Assembly Language 寫成
2. 會呼叫 startup function (即 Thread::Begin)和讀取初始參數(在 StackAllocate 時候被記錄，即 ForkExecute 函式位址，和指向該 thread 的指標)，以及讀取最後該呼叫的 Thread::Finish。
   (示意圖，實際執行情形不一定會是這個 #ifdef 的 #else)

```
#else
    machineState[PCState] = (void *)ThreadRoot;
    machineState[StartupPCState] = (void *)ThreadBegin;
    machineState[InitialPCState] = (void *)func;
    machineState[InitialArgState] = (void *)arg;
    machineState[WhenDonePCState] = (void *)ThreadFinish;
#endif
}
```

3. 在 main thread SWITCH 之後，會呼叫 thread::Finish()，將 thread terminate，再以此函式為開端，讓 OS 繼續執行其他 thread。
   (若沒有這個動作，則可能會讓 main thread 一路 return，從 Sleep -> Finish -> ExceAll -> main -> ASSERTNOTREACHED，進而中止。)

(6) Switch

1. 以 Assembly Language 寫成
2. 移動 Stack pointer，讓他從舊的 thread 指向新的

(7) Thread::CheckOverflow( )

1. 如果 Stack overflowed，則程式中斷(assertion failed)

Userprog/addrspace.cc

(1)AddrSpace::AddrSpace( )

原版未修改的 code 當中，virtual memory 被 1:1 對應到 physical memory 當中

1. Create Entry for a new thread in PageTable
2. Set the initial values for the Thread
3. Zero out the entire address space (**bzero** : set N bytes to zero)

(2)AddrSpace::Execute(Char *filename)

1. Assumes the program is already loaded into the address space
2. Set register by calling Init Registers and RestoreState
3. Call Machine::Run to jump to the user program to run the program.

(3)AddrSpace::Load(Char *filename)

1. Use the FileSystem to Open the File
2. Use NoffHeader 去判斷大小，以及其他資訊，再讀進 memory

Threads/kernel.cc

(1) Kernel::Kernel(int argc, char **argv)

1. 由 main.cc 去呼叫這個建構式
2. Use strcmp to compare different input values and do the according actions
3. Execfile[++execfilenum]會記錄 -e 參數的後面一個參數的名稱

(2) Kernel::ExecAll( )

1. Call Exec() for each file that needs execution
2. Call finish when the loop ends
   (讓當前的 Thread (即 main thread) 進入睡眠，以觸發 **SWITCH**)

(3) Kernel::Exec(char *name)

1. Create new thread by the name and thread number
2. Allocate the space for physical to virtual memory translation
3. Fork the thread

(4) Kernel::ForkExecute(Thread *t)

1. Load information for space
2. Call Execute if the executable is found

Threads/scheduler.cc

(1) Scheduler::ReadyToRun(Thread *thread)

1. Set this thread to Ready state
2. Append it on the ready list

(2) Scheduler::Run(Thread *nextThread, bool finishing)

1. Check if finishing, if true, it destroys the current thread
2. Check if thread is user program, if so, save CPU regs
3. CheckOverflow()
4. Change to nextThread and set state to Running
5. Call SWITCH to stop current thread and start new thread
   SWITCH 在 switch.S 中被 Implement
   Main thread 進入 SWITCH 後，返回值指向 ThreadRoot 函式；其他的 thread 進入 SWITCH 後，返回值指向 Scheduler::Run 函式中的 SWITCH 函式語句 (也就是原來的地方)。
6. Call CheckToBeDestroyed if there is any toBeDestroyed thread, if so, delete IT
   並沒有在第一時間就刪除，是為了完成 SWITCH 的程序，以及維護 Thread Stack 的結構。
7. Try to restore available address spaces.

(II). Questions

Q1 : How Nachos allocates the memory space for new thread(process)?

A : 在 Fork 的時候，會 new 一個 AddrSpace 給 thread->space。

Q2 : How Nachos initializes the memory content of a thread(process), including loading the user binary code in the memory?

A : 使用 AddrSpace::Load()，將讀取的 file 切進記憶體。

Q3 : How Nachos creates and manages the page table?

A :

1. 實作之前是直接在建構式宣告一個 NumPhysPages 大小的 page table。
2. 實作之後則是把建立的這件事，等到 Load 的時候，再依據讀取進來的程式大小，宣告適當尺寸的 page table。
3. 管理的部分，則是 Load 在宣告完 page table 之後，會接續進行。

Q4 : How Nachos translates address?

A :

1. translate.cc 中的 machine::Translate

   *physAddr = pageFrame * PageSize + offset;
2. AddrSpace.cc 中 (在實作後)

   Virtual page number = virtualAddr / PageSize

   Offset = virtualAddr % PageSize

Q5 : How Nachos initializes the machine status (registers, etc) before running a thread(process)

A : 在 Thread::StackAllocate() 的時候，會把自己的 ThreadRoot() 與相關資訊存進 Stack 裡面。相關資訊包含：ThreadBegin、ThreadFinish、該 thread

的 ForkExecute 函式指標，與指向自己的指標 (Thread *)。

Q6 : Which object in Nachos acts the role of process control block

A : Thread (這一個 class)。因為它包含了 page table、register 等資訊。

Q7 : When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?

A : 在 Thread::Fork() 裡面，會呼叫 Scheduler::ReadyToRun( )，並在此時把這個 thread 放進 readyList 裡面。

Q8 : 為何這次下實作的指令之後，必須手動 Ctrl + C 才會停下?

A : Thread 執行完畢，因為沒有 next to run，所以當前的 Thread 會進入 Idle 狀態，然後會不斷檢查是否有 pending interrupt。
Nachos 會排定 **timer**、**console read**、**network recv** 這三個常駐的 interrupt 進而導致 OS 永遠不會結束。

(III) Implementation

Kernel.h

```
bool PhyPageStatus[NumPhysPages];
int NumFreePages;
```

Add two members in the Kernel class,

PhyPageStatus records whether if this page is in use or not.

NumFreePages records the number of pages that are free.

Kernel.cc

```
Kernel::Kernel(int argc, char **argv)
{
    PhyPageStatus[NumPhysPages] = {false};
    NumFreePages = NumPhysPages;
```

Add the initial values of the two members in the constructor of Kernel.

Set all off PhyPageStatus to false since none are in use when initialization, and the value of NumFreePages to the value of NumPhysPages since all are free among initialization.

Machine.h

```
                    // address spa
        OverflowException,      //
        IllegalInstrException, //
        MemoryLimitException,
        NumExceptionTypes
};
```

In order to handle MemoryLimitException, we add it into the ExceptionType

Addrspace.cc

```
AddrSpace::AddrSpace()
{
    // pageTable = new TranslationEntry[NumPhysPages];
    // for (int i = 0; i < NumPhysPages; i++) {
    // pageTable[i].virtualPage = i;    // for now, virt page # = phys page #
    // pageTable[i].physicalPage = i;
    // pageTable[i].valid = TRUE;
    // pageTable[i].use = FALSE;
    // pageTable[i].dirty = FALSE;
    // pageTable[i].readOnly = FALSE;
    // }

    // // zero out the entire address space
    // bzero(kernel->machine->mainMemory, MemorySize);
}
```

We comment all the contents of AddrSpace, we will set the values when we in AddrSpace::Load, the reason is that we need to know how many Pages does this thread need in order to set the pageTable, we could also handle the part where a program size exceed a pagesize.

```
bool AddrSpace::Load(char *fileName)
```

```
ASSERT(numPages <= NumPhysPages); // check we're no
                                  // to run anythin
                                  // at least until
                                  // virtual memory
if(numPages > NumPhysPages){
    ExceptionHandler(MemoryLimitException);
}
```

We first handle the exception that the Memory Limit is exceeded.

```
for (int i = 0, idx = 0; i < numPages; i++)
{
    pageTable[i].virtualPage = i;
    while (idx < NumPhysPages && kernel->PhyPageStatus[idx])
        idx++;
    ASSERT(kernel->PhyPageStatus[idx] == false) // last check
    kernel->PhyPageStatus[idx] = true;
    kernel->NumFreePages--;
    bzero(&kernel->machine->mainMemory[idx * PageSize], PageSize); // clear the page
    pageTable[i].physicalPage = idx;                               //set virtual and phy page id
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}
```

Then we set the required value, it is similar to the constructor of AddrSpace but we use a for loop to get the number of how many pages the thread needs.

```
executable->ReadAt( // set the correct address for the translation of the physical Page address
    &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage* PageSize +
    (noffH.code.virtualAddr%PageSize)]),
    noffH.code.size, noffH.code.inFileAddr);
```

We then set noffH.code.virtualAddr to the PhysicalPage address.

```
AddrSpace::~AddrSpace()
{
    if(pageTable){
        for (int i = 0; i < numPages ; i++){
            kernel->PhyPageStatus[pageTable[i].physicalPage] = false;
            kernel->NumFreePages++;
        }
        delete pageTable;
    }
}
```

Modify the destructor with the addition of the two members that we added.

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

Modify the SaveState in order to record the two members that we added.

(IV). 心得

呂佳恩：透過這次的實作與 trace code 我更了解了 OS 的運作，這次比較困難的部分是 trace code 這時做則稍稍容易一些

林育丞：這次深入去 trace code，發現裡面包含一些 Assembly language 的運作部分，雖然不易理解，但讓我對於 Nachos 整體運作更有概念了。