

OS MP2 Report

Team 27 : 112062232賴允中、111070070林標松

分工表

組員	Trace Code	Report編寫	Implementation
賴允中	Y	修訂	Y
林標松	Y	初稿	Y

1. Trace Code

我們從NachOS的入口開始依序解釋。

- threads/main.cc => main()

此函數為NachOS的入口，主要在解析NachOS收到的執行參數，並根據參數做出相應的初始化，並執行user program。

```
debug = new Debug(debugArg);  
  
DEBUG(dbgThread, "Entering main");  
  
kernel = new Kernel(argc, argv);  
  
kernel->Initialize();
```

在 threads/main.h 裡定義了 kernel 和 debug 這兩個 global variable。用 main 接收到的 command line arguments (argc: 參數數量，argv: 參數字串的陣列) 傳給 [Kernel::Kernel\(\)](#)，建構kernel物件。

```
// finally, run an initial user program if requested to do so  
  
kernel->ExecAll();
```

當初始化完所有要執行的user program後，呼叫 [Kernel::ExecAll\(\)](#)，執行剛剛 execfile[] 裡的 program。

- threads/kernel.cc => Kernel::Kernel()

此函數是 Kernel 物件的 constructor。

```
Kernel::Kernel(int argc, char **argv) {
    randomSlice = FALSE;
    debugUserProg = FALSE;
    execExit = FALSE;
    consoleIn = NULL; // default is stdin
    consoleOut = NULL; // default is stdout
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
    reliability = 1; // network reliability, default is 1.0
    hostName = 0; // machine id, also UNIX socket name
    // 0 is the default machine id
```

初始化 Kernel 物件裡的 flag，例如：

execExit: 當所有thread執行完畢是否呼叫Exit()

consoleIn: 從何處讀取 console input，可以是檔案，預設為 stdin

consoleOut: 將 console output 寫入何處，可以是檔案，預設為 stdout

```
else if (strcmp(argv[i], "-e") == 0) {
    execfile[++execfileNum] = argv[++i];
    cout << execfile[execfileNum] << "\n";
```

接下來的迴圈會遍歷 argv 字串陣列裡的元素，並檢查每個 command 字串，執行該 command 對應的動作。

以MP2舉例，在實作時，會呼叫以下指令：

```
"./build.linux/nachos -e consoleIO_test1 -e consoleIO_test2"
```

當讀到 "-e"，會把下一個讀到的字串放進 Kernel 的 execfile[] 裡。上述指令會將 consoleIO_test1 和 consoleIO_test2 這兩個檔案作為要執行的兩隻 user program，放入execfile[]中。

- threads/kernel.cc => Kernel::Initialize()

創建完kernel物件後，接著會執行 `Kernel::Initialize()`，此函數主要功能為初始化kernel物件中的變數和資料結構。有些變數在建構 kernel 物件時已經被初始化了，這裡將kernel的初始化工作分成兩個函數，是因為某些資料結構的初始化需要用到更早被初始化的變數。

例如 `synchConsoleIn/Out` 就會用到前面的 `consoleIn/Out`，這兩個成員的初始化應該要有先後之分，而不是同時執行。

```
void Kernel::Initialize() {
    // We didn't explicitly allocate the current thread we are running in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.

    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);

    stats = new Statistics();           // collect statistics
    interrupt = new Interrupt();       // start up interrupt handling
    scheduler = new Scheduler();        // initialize the ready queue
    alarm = new Alarm(randomSlice);    // start up time slicing
    machine = new Machine(debugUserProg);
    synchConsoleIn = new SynchConsoleInput(consoleIn);      // input from stdin
    synchConsoleOut = new SynchConsoleOutput(consoleOut);   // output to stdout
    synchDisk = new SynchDisk();         //

#ifdef FILESYS_STUB
    fileSystem = new FileSystem();
#else
    fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
    postOfficeIn = new PostOfficeInput(10);
    postOfficeOut = new PostOfficeOutput(reliability);

    interrupt->Enable();
}
```

首先，建構一個名字叫”main”且 `threadNum = 0` 的 `Thread` 物件指派給 `kernel` 的 `currentThread`，並把此 main thread 設為RUNNING(執行中)，代表現在佔用CPU裡的是這個 thread。

接著初始化其他 kernel 的變數。

- threads/kernel.cc => Kernel::ExecAll()

處理完 kernel 的初始化，最後才呼叫 `Kernel::ExecAll()`。此函數會執行剛剛初始化時，放在 `execfile[]` 裡所有要執行的 user program。

```
void Kernel::ExecAll() {
    for (int i = 1; i <= execfileNum; i++) {
        int a = Exec(execfile[i]);
    }
    currentThread->Finish();
    // Kernel::Exec();
}
```

Main thread 會循環呼叫 `Kernel::Exec()`，執行 `execfile[]` 裡的 program。當所有 program 都執行完後，呼叫 `Thread::Finish()` 以結束 current thread，釋放 thread 的資源。

- threads/kernel.cc => Kernel::Exec()

此函數會為 user program 初始化它的 thread control block (TCB)，並分配一個 address space 紿此 thread，用來管理 thread 在記憶體中的狀態。最後將此 thread 放到 ready queue 中等待執行。

```
int Kernel::Exec(char *name) {
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->setIsExec();
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum - 1;
}
```

首先，為此 user program 建構一個 thread 物件 (TCB)，並把這個 TCB 儲存在 kernel 存放 thread 的 `t[]` 中 (`threadNum` 為其 index)。然後，將其 `isExec` 設成為 true，代表此為用來執行 user program 的 thread。

接著，呼叫 `AddrSpace::AddrSpace()` 來分配這個 thread 可用的 address space。最後，呼叫 `Thread::Fork()` 把要執行的程式碼載入，這邊可發現傳入了 `ForkExecute()` 函數，並將現有 thread 數量+1。

- **threads/kernel.cc => Kernel::ForkExecute()**

此函數將要執行的 user program 從檔案中載入 thread 的 address space，如果成功就執行這個 program。

```
void ForkExecute(Thread *t) {
    if (!t->space->Load(t->getName())) {
        return; // executable not found
    }

    t->space->Execute(t->getName());
}
```

首先，呼叫 `AddrSpace::Load()` 來把執行檔載入到記憶體，如果找不到可執行檔，就直接 return。成功把執行檔載入到記憶體後，呼叫 `AddrSpace::Execute()` 去執行 program。

- **userprog/addrspace.cc => AddrSpace::Load()**

此函數將執行檔 load 到 memory。

```
bool AddrSpace::Load(char *fileName) {
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

    if (executable == NULL) {
        cerr << "Unable to open file " << fileName << "\n";
        return FALSE;
    }
```

首先，利用 `fileSystem` 裡的 `Open()` 開啟執行檔，如果開啟失敗就回傳 False。

```

executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
if [(noffH.noffMagic != NOFFMAGIC) &&
    (WordToHost(noffH.noffMagic) == NOFFMAGIC)]
    SwapHeader(&noffH);
ASSERT(noffH.noffMagic == NOFFMAGIC);

```

接著讀取執行檔，將其讀進 NoffHeader 中(包含code、data segments等資訊)。這裡還要處理 big-endian 和 little-endian 的問題，成功時 noffH.noffMagic要等於NOFFMAGIC。

```

#ifndef RDATA
    // how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
          noffH.uninitData.size + UserStackSize;
    // we need to increase the size
    // to leave room for the stack
#else
    // how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize; // we need to
    increase the size

```

這邊在計算 address space 的大小，兩種計算方式取決於資料是否有唯讀的部分。

```

numPages = divRoundUp(size, PageSize);
size = numPages * PageSize;

ASSERT(numPages <= NumPhysPages); // check we're not trying
// to run anything too big --
// at least until we have
// virtual memory

```

這邊在計算這個程式需要的 page 數量，並保證 page 數量不會超過實際記憶體的 page 數量。

```

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

```

```

#define RDATA
    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[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

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

```

將 code 和 data segment 複製到記憶體中，其中 initData 為一些已被賦值的全域變數或常數，readonlyData 為剛剛提到的唯讀部分。

複製完後關閉執行檔，回傳 TRUE (load成功) 。

- **userprog/addrspace.cc => AddrSpace::Execute()**

此函數透過當前持有CPU資源的 current thread 來執行 user program 。

```

void AddrSpace::Execute(char *fileName) {
    kernel->currentThread->space = this;

    this->InitRegisters(); // set the initial register values
    this->RestoreState(); // load page table register

    kernel->machine->Run(); // jump to the user program

    ASSERTNOTREACHED(); // machine->Run never returns;
                        // the address space exits
                        // by doing the syscall "exit"
}

```

首先，將 kernel 當前 thread 的 address space 設為此函數 caller 的。接著初始化 user-level register，並把 page table register 載入進來，讓 MMU可以正確使用這個 process 的 page table。最後，呼叫 Machine::Run() 來執行user program 。

- userprog/addrspace.cc => AddrSpace::AddrSpace()

講完 `ForkExecute()` 函數以及它會呼叫的兩個 `AddrSpace` 物件的 member function 後，回到 `Kernel::Exec()`。它會呼叫 `AddrSpace` 物件的 constructor 創建一個專屬於這個 thread 的 address space。這個建構子會分配一塊記憶體空間讓 user program 可以跑在上面，並初始化這程式的 page table。

```
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);
}
```

首先，創建一個 page table (`TranslationEntry` 這個資料結構被定義在 `translate.h` 裡面)，表示 page table 或 TLB 的一個 entry，每一個 entry 裡除了有 virtual page number 和 physical page number 的對照，還有關於這個 page 的相關資訊，如 valid、readOnly、use、dirty bit)。而一開始因為沒有 multi-programming 的需求，所以將 virtual address 直接 1-1 對應到 physical address，可以省去複雜的轉換過程。

`valid` bit 設為 True 代表這個 page 可以被 access；`use` bit 設為 False 代表這個 page 還未被 reference 或 modify 過；`dirty` bit 設為 False 代表這個 page 沒被更改過；`readOnly` bit 設為 False 代表這個 page 並非唯讀，可以被修改。最後，利用 `bzero()` 將 main memory 清出一塊記憶體給這個 thread，相當於 `memset()` 的功能。

- threads/thread.cc => Thread::Fork()

在 Kernel::Exec() 裡，初始化完 thread 的 address space 後，會呼叫 Fork() 函數。此函數會分配一塊 stack 記憶體空間給 thread 使用，並把 thread 放到 ready queue 中等待被執行。

```
void Thread::Fork(VoidFunctionPtr func, void *arg) {
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " " << arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
    // are disabled!
    (void)interrupt->SetLevel(oldLevel);
}
```

首先，透過變數 interrupt 和 scheduler 獲取 kernel 的 interrupt 狀態以及 CPU scheduler。然後，呼叫 StackAllocate() 來初始化此 thread 的 stack 空間。這裡把 ForkExecute() 傳進去作為 thread 的執行起點，開始執行 user program。

分配完 stack 後，接著呼叫 Scheduler::ReadyToRun() 來將 thread 放到 ready queue 裡，等待被 scheduler 排程使用 CPU。在此要注意先把 kernel interrupt 關掉，原因是防止其他地方同時更動 ready queue，導致 thread 排入過程中的 race condition，避免 data inconsistent 的問題。當 ReadyToRun() 返回後，再把 interrupt 狀態恢復。

- threads/thread.cc => Thread::StackAllocate()

在 Thread::Fork() 裡，呼叫此函數來初始化 stack memory。此函數會配置一塊記憶體空間給 thread 的 stack (可以存 thread 的 local variable、return address 等等)，並初始化 stack frame。

```
void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
    stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
```

首先，呼叫 `AllocBoundedArray()` 來配置一塊固定大小的stack記憶體空間，`StackSize` (定義在`thread.h`裡)是 8×1024 words，而這裡是配置 $\text{StackSize} * \text{sizeof(int)} = 8192$ int 空間的 stack memory。

配置完stack空間後，會根據不同的處理器架構，執行不同的區塊。而 NachOS 是使用 x86 架構，故執行以下程式：

```
#ifdef x86
    // the x86 passes the return address on the stack. In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return address
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4; // -4 to be on the safe side!
    *[--stackTop] = (int)ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif
```

由於 `stack` 是從高地址向低地址增長的，所以首先將 `stackTop` 設置為 stack memory 的最高位置，然後將 `ThreadRoot` 放置在 `stackTop` 上。這樣當 `thread` 開始運行時，它將從 `ThreadRoot` 開始。

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

最後，把 `ThreadRoot` 和其他的 routine 存入 kernel 的 register，確保 `thread` 在正確的 routine 和參數下執行。

- threads/scheduler.cc => Scheduler::ReadyToRun()

在 `Thread::Fork()` 裡，配置完 `thread` 的 stack memory 後，就會呼叫此函數把 `thread` 放入 ready queue 中。而此函數目的很簡單，就是把傳入的 `thread` 狀態設成 Ready，並放到 scheduler 的 ready list 裡，等待排程。

```
void Scheduler::ReadyToRun(Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    // cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

一開始，先確保 interrupt 已經被 disable 掉，而原因已在 [Fork\(\)](#) 解釋過。接著，將此 thread 的狀態設為 READY，放進 ready list。

```
// Thread state
enum ThreadStatus { JUST_CREATED,
                     RUNNING,
                     READY,
                     BLOCKED,
                     ZOMBIE };
```

▲Thread的五種狀態

- **threads/thread.cc => Thread::Finish()**

當在 [Kernel::ExecAll\(\)](#) 中，把所有 user program 都執行完成後，會呼叫 [Thread::Finish\(\)](#) 來結束current thread。

```
void Thread::Finish() {
    (void)kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
```

首先，先 disable interrupt，因為最後呼叫的 [Sleep\(\)](#)，一樣要避免發生 race condition。隨後確認 caller 是現在握有 CPU 的 current thread，畢竟只有目前運行的 thread 本身可以呼叫 Finish 終止自己。

```
if (kernel->execExit && this->getIsExec()) {  
    kernel->execRunningNum--;  
    if (kernel->execRunningNum == 0) {  
        kernel->interrupt->Halt();  
    }  
}
```

如果 execExit 為真（ thread 執行完是否要 halt ），且此 thread 的 isExec 為真（是否為 user program thread ），就把 kernel 的 execRunningNum (running user program thread的數量) 減一，而如果發現 execRunningNum 已經減到0了，代表沒有 running thread了，就直接呼叫 Halt() ，關掉 NachOS 。

```
Sleep(TRUE); // invokes SWITCH
```

最後呼叫 Thread::Sleep() 函數，傳入 TRUE 代表此 thread 已經完成工作，等待被釋放，將會引發 context switch ，把CPU讓給其他 thread 。

- threads/thread.cc => Thread::Sleep()

在 Thread::Finish() 中，如果 execRunningNum 還不為0，代表還有其他 thread 在等待排程，就會呼叫 Sleep() ，進行 context switch 讓出CPU資源。

此函數會釋放手中的CPU資源，因為 thread 已經完成，或是被 blocked 在等待 synchronization （如果是這種情形， Sleep() 會在其他函數被呼叫，而非 Finish() ，因為工作尚未完成） 。

```
void Thread::Sleep(bool finishing) {
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
```

首先，檢查 caller 是否為 current thread，且確保 interrupt 已經被 disable 了，這是為了保障從 ready queue 拿出 front thread，到 switch 完成前不會被其他 incoming interrupt 打斷。

```
status = BLOCKED;
// cout << "debug Thread::Sleep " << name << "wait for Idle\n";
while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle(); // no one to run, wait for an interrupt
}
```

把當前的 thread (currentThread) 狀態設為BLOCKED，並進入 while 迴圈，呼叫 [Scheduler::FindNextToRun\(\)](#) 來看在 ready queue 裡是否有 thread 等待使用CPU，如果沒有 thread 在 ready queue 裡，就呼叫 [Interrupt::Idle\(\)](#)，去檢查有沒有 pending interrupt，來產生 thread 放到 ready queue 裡（I/O interrupt 會造成在 waiting queue 的 thread 回到 ready queue 裡）。如果也沒有 pending interrupt 要處理，就呼叫 [Halt\(\)](#)，直接關機。

```
// returns when it's time for us to run
kernel->scheduler->Run(nextThread, finishing);
```

如果 ready queue 裡有下一個要等待使用CPU的 ready thread，就呼叫 [Scheduler::Run\(\)](#) 來執行下一個 thread，此時進行 context switch。

- threads/scheduler.cc => Scheduler::Run()

在 Thread::Sleep() 中，因為有下一個 ready thread 要使用CPU，因此呼叫此函數。此函數主要目的就是做 context switch 的動作。

```
void Scheduler::Run(Thread *nextThread, bool finishing) {  
    Thread *oldThread = kernel->currentThread;  
  
    ASSERT(kernel->interrupt->getLevel() == IntOff);
```

首先，把現在的 thread(currentThread) 記錄到 oldThread 裡，並確保 interrupt 已經被 disable 掉，以保障 context switch 過程中不會被打斷。

```
if (finishing) { // mark that we need to delete current thread  
    ASSERT(toBeDestroyed == NULL);  
    toBeDestroyed = oldThread;  
}
```

如果傳入的參數 finishing 為 True，代表現在的 thread 已經執行完畢，把 oldThread(即currentThread) 設給 toBeDestroyed，表示可以釋放此 thread 的空間。

```
if (oldThread->space != NULL) { // if this thread is a user program,  
    oldThread->SaveUserState(); // save the user's CPU registers  
    oldThread->space->SaveState();  
  
    oldThread->CheckOverflow(); // check if the old thread  
    // had an undetected stack overflow
```

如果 current thread 是在執行 user program thread (會有addrspace)，要在做 context switch 前先把 user-level register 和 address space 的資訊都儲存下來，並檢查是否有 undetected stack overflow 發生。

```
kernel->currentThread = nextThread; // switch to the next thread  
nextThread->setStatus(RUNNING); // nextThread is now running
```

```
SWITCH(oldThread, nextThread);
```

將 currentThread 設為下一個新的 nextThread，並把要切換的 thread 狀態設為 RUNNING，接著呼叫 `SWITCH()` 執行 context switch。

```
// interrupts are off when we return from switch!  
ASSERT(kernel->interrupt->getLevel() == IntOff);  
  
DEBUG(dbgThread, "Now in thread: " << oldThread->getName());  
  
CheckToBeDestroyed(); // check if thread we were running  
// before this one has finished  
// and needs to be cleaned up  
  
if (oldThread->space != NULL) { // if there is an address space  
    oldThread->RestoreUserState(); // to restore, do it.  
    oldThread->space->RestoreState();  
}
```

context switch 完成後回來 oldThread，一樣先檢查 interrupt 是否有 disable 掉，確保等等執行過程不會被打斷。呼叫 `CheckToBeDestroyed()` 來檢查是否有執行完成的 thread 要被釋放，如果剛才傳入的 finishing 參數為 True，則 toBeDestroyed 不是 NULL，那麼該 thread 就會被釋放掉。(Thread的destructor只會把thread的stack memory釋放掉)

-
- **threads/thread.cc => Thread::~Thread()**

```
Thread::~Thread() {
    DEBUG(dbgThread, "Deleting thread: " << name);
    ASSERT(this != kernel->currentThread);
    if (stack != NULL)
        DeallocBoundedArray((char *)stack, StackSize * sizeof(int));
}
```

在 thread 的解構子中，可以發現其只是呼叫 `DeallocBoundedArray()` 函數將剛剛 thread 在 `Thread::StackAllocate()` 中被分配到 stack 記憶體空間刪除，而不會把整個 thread 的記憶體空間都釋放掉。

如果剛才傳入的 `finishing` 參數為 `False`，代表這個 `oldThread` 還需要繼續執行。而如果這個之前的 thread (`oldThread`) 是 user program thread，就需要把 user-level register 和其 address space 相關資訊重新載回。

2. Q&A

(1) How does Nachos allocate the memory space for a new thread(process)?

- Nachos 會在創造好 Thread 物件後，呼叫 `AddrSpace::AddrSpace()`，分配一個可以運行這個 user program 的 memory space 出來。

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

- 接下來，在 `Thread::Fork()` 裡，會呼叫 `Thread::StackAllocate()`，來分配這個 thread 可以使用的 stack memory。

```
StackAllocate(func, arg);
```

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

- 首先，會先透過 `AddrSpace::AddrSpace()` 函數來初始化 thread 的 memory space，在此函數的最後會呼叫 `bzero()` 把 main memory 清空，全部內容設成 0 (NachOS 一開始設定是 uniprogramming，不影響其他程式的記憶體)

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

- `AddrSpace::Load()` 中，會透過 Noffheader 物件讀取執行檔的相關資訊，其中包含了 noffMagic (判斷是否為 noff file)，4 個 Segment 物件 (code, initData, readonlyData, uninitData)，算出此 thread 需要多少個 page 級它的 address space。接著把執行檔中的 code、data segments、readOnlyData (如果有) 全部載入到 memory 裡。

```
// then, copy in the code and data segments into memory  
// Note: this code assumes that virtual address = physical address  
if (noffH.code.size > 0) {  
    DEBUG(dbgAddr, "Initializing code segment.");  
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);  
    executable->ReadAt(  
        &(kernel->machine->mainMemory[noffH.code.virtualAddr]),  
        noffH.code.size, noffH.code.inFileAddr);  
}
```

(3) How does Nachos create and manage the page table?

- 利用 `AddrSpace::AddrSpace()` 來創建 page table，而因為 Nachos 預設只有 uniprogramming，所以創建一個 entry 和 physical page number 一樣多的 page table，並且用一個 for loop 初始化所有 page table entry 裡的資訊(包括 virtual page number、physical page number、extra bits)

```
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;
}
```

- 在管理方面，透過valid bit(是否能使用)、use bit(是否被reference過或修改過)、dirty bit(是否修改過)、readOnly(是否只能讀)，來管理page的狀態。

(4) How does Nachos translate addresses?

- 在 `Machine::OneInstruction()` 函數裡，會去執行 user program 裡的指令，裡面會呼叫 `Machine::ReadMem()` 抓取指令，並獲取 register 中的資料。

```
// Fetch instruction
if (!ReadMem(registers[PCReg], 4, &raw))
    return; // exception occurred
```

- 而 `ReadMem()` 函數裡會呼叫 `Machine::Translate()` 來做地址的轉換，同時檢查address的轉換有沒有發生exception。`Machine::Translate()` 主要功能為將 virtual address 轉換成對應的 physical address。
- 在 `Machine::Translate()` 裡，會先計算 virtual page number (vpn)，即為 virtual address 除以 page size 取商數(前半部)；page offset 為 virtual address 除以 page size 取餘數(後半部)。

virtual address	page number	page offset
-----------------	-------------	-------------

- NachOS中，並不是採用課程中所學之以 TLB 協助 page table 的做法，而是只允許選用其中一種方式翻譯地址。所以這裡分成兩種 case 進行轉譯。

```

if (tlb == NULL) { // => page table => vpn is index into table
    if (vpn >= pageTableSize) {
        DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
        return AddressErrorException;
    } else if (!pageTable[vpn].valid) {
        DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
        return PageFaultException;
    }
    entry = &pageTable[vpn];
} else {
    for (entry = NULL, i = 0; i < TLBSize; i++)
        if (tlb[i].valid && (tlb[i].virtualPage == ((int)vpn))) {
            entry = &tlb[i]; // FOUND!
            break;
        }
    if (entry == NULL) { // not found
        DEBUG(dbgAddr, "Invalid TLB entry for this virtual page!");
        return PageFaultException; // really, this is a TLB fault,
                                    // the page may be in memory,
                                    // but not in the TLB
    }
}

```

如果 TLB 不存在 (`tlb == NULL`)，代表採用 page table，那我們就要從 `pageTable[]` 嘗試讀取physical address。首先要判斷 `vpn` 是否合法（不能超過 `pageTableSize`，否則報錯 `AddressError`），再來如果 `pageTable[vpn]` 還沒被設為可存取，代表發生 page fault，需要做進一步處理。

反之，則直接遍歷 TLB（時間很短因為TLB很小），如果沒找到一樣代表發生 page fault。若有找到，將 `entry` 設為 hit 的欄位。`entry` 為 `TranslationEntry` 物件，其中包含的資訊已在問題(3)中詳述。

- 得到 entry 後，先檢查是否正在對唯讀的 page 做寫入，如果是的話也會報錯。若沒報錯，隨後得到 pageFrame (frame number)。

```
if (entry->readOnly && writing) { // trying to write to a read-only page
    DEBUG(dbgAddr, "Write to read-only page at " << virtAddr);
    return ReadOnlyException;
}
pageFrame = entry->physicalPage;
```

- 得到 pageFrame 後還沒結束，要再確認這個實體地址不會超出實體記憶體的大小。

```
// if the pageFrame is too big, there is something really wrong!
// An invalid translation was loaded into the page table or TLB.
if (pageFrame >= NumPhysPages) {
    DEBUG(dbgAddr, "Illegal pageframe " << pageFrame);
    return BusErrorException;
}
```

- 因為 reference 了這個page，所以把 use bit 設為 True，如果 writing 為 True (代表進行寫入)，也要把 dirty bit 設為 True。

```
entry->use = TRUE; // set the use, dirty bits
if (writing)
    entry->dirty = TRUE;
```

- 最後，將 physical address 根據剛剛得到的資訊計算出來，我們知道 $\text{physical address} = \text{pageTable}[\text{page number}] * \text{page size} + \text{page offset}$ ，而 pageTable[page number] 即對應到 frame number(pfn)，因此可套進公式，即得到physical address，完成轉譯。

```
*physAddr = pageFrame * PageSize + offset;
```

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

- `AddrSpace::Execute()` 負責做 user program 執行前的初始化，它會呼叫 `InitRegister()` 以更新 CPU registers。

```
void AddrSpace::InitRegisters() {
    Machine *machine = kernel->machine;
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister(i, 0);

    // Initial program counter -- must be location of "Start", which
    // is assumed to be virtual address zero
    machine->WriteRegister(PCReg, 0);

    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    // Since instructions occupy four bytes each, the next instruction
    // after start will be at virtual address four.
    machine->WriteRegister(NextPCReg, 4);

    // Set the stack register to the end of the address space, where we
    // allocated the stack; but subtract off a bit, to make sure we don't
    // accidentally reference off the end!
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
}
```

- 首先用 for 迴圈清空所有 register，接著把 PC 和 NextPC 各別設為 0 和 4，最後把 stack pointer 設為該 thread 所分配到的空間 -16，因為要保障記憶體存取的安全性（不會存取到超出 stack end 的地方）。

(6) Which object in Nachos acts the role of process control block

- `Thread` 這個物件就等於 thread control block，而因為 Nachos 沒有 process 的概念，只有 thread 的執行單位，因此 thread 就相當於 process control block。
- 在 `Thread` 這個物件裡，記錄著關於 thread 的各種重要資訊，包括 name、ID、status、machineState、userRegisters、address space 等等資訊，還有對於 `Thread` 物件的各項操作函數，包括 `Fork()`、`Sleep()`、`Yield()`、`Finish()` 等等 operation function。

(7) When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?

- 在 `Kernel::Exec()` 裡，當創建好 Thread 物件，並也創造好這個 thread 的 address space 後，就會透過 `Thread::Fork()` 函數來分配此 thread 的 stack memory，裡面呼叫了 `Scheduler::ReadyToRun()` 把此 thread 放入 ready queue 裡。另一種可能是呼叫了 `Thread::Yield()`（自願讓出CPU給其他 thread 使用），`Yield()` 裡也會呼叫 `ReadyToRun()`。

```
void Thread::Fork(VoidFunctionPtr func, void *arg) {
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " " << arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
                                // are disabled!
    (void)interrupt->SetLevel(oldLevel);
}
```

```
void Thread::Yield() {
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

3. Implementation

一開始，NachOS 的設定是只支持 uniprogramming，所以在初始化 user program 的 address space 時，直接將全部 physical memory 空間給單一程式，在 [AddrSpace::AddrSpace\(\)](#) 函數裡可以看到初始化 page table 的過程。但為了支援 multiprogramming，將 physical memory 分配給多個 user program，應該在計算完 user program 所需要的 address space 後，再為此程式創建 page table。

- 原本的AddrSpace::AddrSpace()函數

```
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);
}
```

可以發現，此函數在創建 pageTable 時，直接把 NumPhysPages (physical page 數量) 當作 page table entry 的數量，並且把 virtual page number 直接等於 physical page number，將所有的記憶體空間都給了單一 user program。

因此，我們改成在得知此 user program 所需的 memory 空間後，在 [AddrSpace::Load\(\)](#) 函數裡會計算出 address space 大小，再去為它開其所需要的空間，達到 physical memory 能夠分配給多個 user program 使用，實現 multiprogramming。

- 先在 AddrSpace class 裡(定義在 addrspace.h 中)宣告兩個 static 變數(static 變數宣告在 class 裡，代表此變數所有 AddrSpace 物件都可以使用)：
 1. **usedPhyPage[NumPhysPages]**: 紀錄physical frame的使用狀況，false代表無人使用(free frame)；True代表有人在使用。
 2. **freeFrameNum**: 紀錄現在有多少 free frame 能夠使用，當一個 user program 所需要的 page number 超過 free frame 數量時，就代表沒有足夠的記憶體空間能分配給它，此時丟出 Exception 細 ExceptionHandler。
- 在addrspace.h裡的AddrSpace class新增兩個static變數

```
class AddrSpace {
public:
    AddrSpace(); // Create an address space.
    ~AddrSpace(); // De-allocate an address space

    static bool usedPhyPage [NumPhysPages]; //紀錄physical frame的使用狀況
    static int freeFrameNum; //紀錄目前有多少free frame可以用
```

- 在addrspace.cc裡初始化兩個static變數

```
bool AddrSpace::usedPhyPage [NumPhysPages] = {0}; //初始化所有physical frame都沒有被使用
int AddrSpace::freeFrameNum = NumPhysPages; //初始化所有physical frame都是free frame
```

而因為我們要在知道 user program 確切需要多少的 memory 空間，即需要多少個page，在 [AddrSpace::Load\(\)](#) 函數裡有計算 address space size 的實作。因此，我們將初始化 page table 的實作移到 [AddrSpace::Load\(\)](#) 裡，根據 user program 所需要的 page 數量去動態建立一個 page table。

- 把原本在AddrSpace::AddrSpace()裡的初始化page table動作刪除

```
AddrSpace::AddrSpace() {
    pageTable = NULL; //先初始化pageTable為NULL
}
```

將原本在這裡面的 page table 初始化的動作刪除，只先將page table初始化為 NULL。

- AddrSpace::Load()裡計算user program需要的page數量

```
#ifdef RDATA
    // how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
    // we need to increase the size
    // to leave room for the stack
#else
    // how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size +
           UserStackSize; // we need to increase the size
    // to leave room for the stack
#endif
    numPages = divRoundUp(size, PageSize); //計算thread需要多少page
```

- 得知numPages後，先檢查是否有足夠的記憶體能夠分配

```
ExceptionType ex = NoException; //用來處理exception
if(numPages > AddrSpace::freeFrameNum) //檢查是否有足夠的physical frame可以使用，如果不夠就丟出exception
{
    ex = MemoryLimitException;
    ExceptionHandler(ex); //丟出exception，呼叫ExceptionHandler()來處理
}
```

如果 user program 所需要的 page 數量超過此時可用 free frame 的數量 (freeFrameNum)，則代表此時沒有足夠的記憶體能夠分配給此 user program，丟出 MemoryLimitException 給 ExceptionHandler 處理。

- 在 machine.h 裡的 ExceptionType 中新增 MemoryLimitException

```
enum ExceptionType { NoException,           // Everything ok!
                     SyscallException,    // A program executed a system call.
                     PageFaultException, // No valid translation found
                     ReadOnlyException,  // Write attempted to page marked
                                         // "read-only"
                     BusErrorException,  // Translation resulted in an
                                         // invalid physical address
                     AddressErrorException, // Unaligned reference or one that
                                         // was beyond the end of the
                                         // address space
                     OverflowException,   // Integer overflow in add or sub.
                     IllegalInstrException, // Unimplemented or reserved instr.
                     MemoryLimitException, // insufficient physical memory for a
                                         // thread
                     NumExceptionTypes
};
```

- 將 Exception 丟給 ExceptionHandler() 處理

```
default:
    cerr << "Unexpected user mode exception " << (int)which << "\n";
    break;
}
ASSERTNOTREACHED();
```

跳到 default 的程式碼，印出 ”Unexpected user mode exception” ，接著跳到最後面 **ASSERTNOTREACHED()** ，中止程式運行。

回到 **AddrSpace::Load()** 裡，當確認 user program 需要的 page 數量不會超過可以用的 free frame 數量後，就可以創建和初始化專屬於它的 page table，我們拿 numPages 來建立適配它大小的page table entry，以實現 multiprogramming 。

- 得知numPages後，分配記憶體空間給此user program

```
pageTable = new TranslationEntry[numPages]; //根據numPages來創建pageTable
//初始化pageTable entry
for(unsigned int i = 0, j = 0; i < numPages; i++) //j是用來找尋可以用的physical
frame，每次都要從頭找，因為可能會有其他thread釋放掉physical frame，散落在其他前面
{
    pageTable[i].virtualPage = i; //設定virtual page number，即page table的index
    while(j < NumPhysPages && AddrSpace::usedPhyPage[j] == true) //找尋可以用的
    physical frame
    {
        j++;
    }
    AddrSpace::usedPhyPage[j] = true; //將這個physical frame設為true，表示已經被使用
    AddrSpace::freeFrameNum--; //減掉這個free frame數量
    //初始化在main memory裡被分配到的frame，利用bzero()將這個frame的內容設為0
    bzero(&(kernel->machine->mainMemory[j * PageSize]), PageSize); //(j * PageSize)
    是這個frame在main memory的base address
    pageTable[i].physicalPage = j; //將這個frame number分配給virtual page
    pageTable[i].valid = true; //將valid設為true
    pageTable[i].use = false; //其餘設為false
    pageTable[i].dirty = false;
    pageTable[i].readOnly = false;
}
```

1. 先使用計算好的 numPages 建立 page table。
2. 使用 for loop 來設定 page table，這裡會先去用一個 while 迴圈來找此時的 free frame，即 usedPhyPage[j] == false，找到 free frame 後才將其分配給這個 virtual page。
3. 找到可以用的frame後，記得更新這個frame已經被使用了(usedPhyPage[j] = true)，防止後面的page用到同一塊frame，也記得更新free frame的數量，將其減一 (freeFrameNum--)。
4. 呼叫 `bzero()`，來將分配的 frame 的記憶體空間清空，設為0。
5. 將分配到的 free frame 紿此 virtual page (pageTable[i].physicalPage = j)。
6. 設定此 page 的 extra bit，將 valid bit 設為 true(代表此 page 在記憶體裡，可以 access)，use/dirty /readOnly bit 設為 false (spec裡的要求)。

接著，去更改copy code和data segment的位置，因做法雷同，這裡只針對copy code的部分做說明。

- 改變code放入main memory中的位置的讀法

```
// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
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);
}
```

我們將放入記憶體中的位置改成做完 mapping 後的 physical address，計算步驟如下：

1. 先將 virtual address 除以 page size 取商數，即 page number，當作 page table 的 index 去搜尋 (pageTable[noffH.code.virtualAddr / PageSize])。
2. 去 page table 找到此 page 對應到的 physical page number 後，乘以 page size，即得到此 physical page 在記憶體的 base address。
3. 把 virtual address 除以 page size 取餘數，即 page offset，將 frame number 加上 page offset 後，即可以得到 physical address 來存放 code 在記憶體中的位置。

- 改變data segment放入main memory中的位置的讀法(做法相同)

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

#ifndef RDATA
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[pageTable[noffH.readonlyData.virtualAddr/PageSize].physicalPage * PageSize +
        (noffH.readonlyData.virtualAddr%PageSize)]),
        noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
}
#endif
```

最後，當thread執行結束後，要釋放其page table，並把frame讓出來給後面的程式使用。

- 在**AddrSpace::~AddrSpace()**裡增加些程式碼，來更新frame的狀態

```
AddrSpace::~AddrSpace() {
    for(int i = 0; i < numPages; i++)
    {
        AddrSpace::usedPhyPage[pageTable[i].physicalPage] = false;
        //將這個thread使用到的physical frame設為false
        AddrSpace::freeFrameNum++; //增加free frame數量
    }
    delete pageTable;
}
```

在釋放記憶體空間時，記得將釋放的frame (physical page) 設為未被使用，以及將free frame數量增加。

4. Implementation Result

- Result with multiprogramming

```
[os24team27@localhost test]$ ../../build/linux/nachos -e consoleIO_test1 -e console  
IO_test2  
consoleIO_test1  
consoleIO_test2  
9  
8  
7  
6  
1return value:0  
5  
16  
17  
18  
19  
return value:0  
^C  
Cleaning up after signal 2
```

- Result with handling exception of insufficient memory

```
[os24team27@localhost test]$ ../../build/linux/nachos -e consoleIO_test1 -e console  
IO_test2  
consoleIO_test1  
consoleIO_test2  
numPages: 12  
9numPages: 12  
  
8  
7  
6  
1return value:0  
5  
16  
17  
18  
19  
return value:0  
^C  
Cleaning up after signal 2  
[os24team27@localhost test]$ █
```

- 將 user program 所需要的 page 數量印出來，可得知一個程式要12個 page

```
[os24team27@localhost test]$ ../../build/linux/nachos -e consoleIO_test1 -e console  
IO_test2  
consoleIO_test1  
consoleIO_test2  
9Unexpected user mode exception 8  
Assertion failed: line 197 file ../../userprog/exception.cc  
Aborted
```

- 把定義在 machine.h 裡 NumPhysPages 的數量改為16，以模擬記憶體只夠執行一個程式，兩個以上則不足的情形。結果如上，可順利丟出exception。

5. Difficulties we encounter when implementing this assignment

林櫻松：這次的作業明顯比第一次困難許多，畢竟memory management這章節本身就較困難，除了要看的程式碼會一直跳去別的地方的函數不斷呼叫下去再return回來，使其更加複雜追蹤code path之外，加上我們是等考完期中考才開始做，因此時間上也有壓力。我認為最有挑戰的地方是實作的部分，不像MP1可以有模板參考，這次更加靈活，需要先將Nachos如何創建及管理page table理解清楚，加上對virtual address轉換成physical address的過程要瞭若指掌，才能將此次作業完成。也謝謝組員幫忙更正我許多理解錯誤的地方，並一起討論，讓我可以對Nachos有更佳的理解，雖然過程波折，但收穫許多，希望接下來能夠順利！

賴允中：這次trace code的部分真的很麻煩，除了spec沒有給一條明確的順序提示以外，很多地方的運作是不斷呼叫不同的function到很深層的地方才return回來，導致我好幾次好不容易理清深處的細節時，回頭卻又有點困惑了。然後這次的程式部分使用了非常多的指標，我也看見了非常巧妙的函數設計方式，例如 [Machine::Translate\(\)](#)，雖然回傳的是ExceptionType物件，卻因為傳入的 physAddr 是指標，所以轉譯完其實相當於可以同時回傳狀態跟物理地址！非常巧妙的做法。過程中我還有個比較大的困擾，就是我一度無法理解Nachos的兩組CPU registers到底是怎麼運作的，在這裡卡住很久。

時間管理方面，這次的作業我們分配的時間太少，因為剛好前面卡到期中周，所以開始做以後很趕，幾乎沒時間休息。我覺得下次這種大型作業我應該要分多一點時間做，因為讀這種大規模的程式我很容易腦袋打結，需要足夠的時間休息跟思考。好險我的隊友很給力，很快的就把大架構都打好了，我則是花比較多時間在修正細節。

經過很多的討論，我們修正了很多理解錯誤的地方，終於也對記憶體管理的部分有更透徹的了解。之前計算機結構雖然已經學過了，但當時的感覺就像在紙上談兵，整個記憶體的章節學完，我甚至無法好好解釋virtual memory到底在做些什麼...愛OS