作業系統 - MP2 – Multi-Programming

tags NachOS

studentID:1102003S

workshop account:os23s77

teamID:陳觀宇 (kcem104@gmail.com)

name:陳觀宇

HackMD link: LINK

Part 1 Trace Code

Code執行流程:

- Kernel::Kernel() →Kernel::ExecAll() →Kernel::Exec(),Kernel::Finish()
- Kernel::Exec() →AddrSpace::AddrSpace(),Kernel::ForkExecute(),Thread::Fork()
- AddrSpace::AddrSpace()
- Kernel::ForkExecute() →AddrSpace::Load() →AddrSpace::Execute() →Machine::Run()
- Thread::Fork() →Thread::StackAllocate() →Scheduler::ReadyToRun()
- Thread::Finish() →Thread::Sleep() →Scheduler::Run()

1-3. threads/kernel.cc Kernel::Kernel()

- 根據threads/kernel.cc文件說明，Kernel::Kernel()是解讀不同command line參數，根據參數在kernel初始化時設定各項機器參數，以利後thread各自紀錄MachineState。

```
//----------------------------------------------------------------------
// Kernel::Kernel
//  Interpret command line arguments in order to determine flags
//  for the initialization (see also comments in main.cc)
//----------------------------------------------------------------------

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

```
{

    randomSlice = FALSE;
    // randomSlice: 表示是否啟用隨機時間片，預設為FALSE。
    debugUserProg = FALSE;
    // debugUserProg：表示是否啟用用戶程序調試模式，預設為FALSE。
    consoleIn = NULL;
    // default is stdin
    // consoleIn：表示控制台輸入文件名稱，預設為NULL，即stdin。
    consoleOut = NULL;
    // default is stdout
    // consoleOut：表示控制台輸出文件名稱，預設為NULL，即stdout。

#ifndef FILESYS_STUB
    formatFlag = FALSE;
    // formatFlag：表示是否在虛擬磁盤上格式化文件系統，預設為FALSE。
    // 如果不是FILESYS_STUB，則可使用此選項。
#endif
    reliability = 1;
    // network reliability, default is 1.0
    // reliability：表示網絡可靠性，默認值為1.0。
    hostName = 0;
    // machine id, also UNIX socket name
    // 0 is the default machine id
    // hostName：表示主機ID，也是UNIX套接字名稱，默認值為0。
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
            // 引數是"-rs"，那麼從下一個引數中讀取一個整數，
            // 用來初始化亂數種子，使得後續的時間片大小可以隨機生成。
            ASSERT(i + 1 < argc);
            RandomInit(atoi(argv[i + 1]));
            // initialize pseudo-random
            // number generator
            randomSlice = TRUE;
            i++;
        } else if (strcmp(argv[i], "-s") == 0) {
            debugUserProg = TRUE;
        } else if (strcmp(argv[i], "-e") == 0) {
                // 引數是"-e"，那麼將下一個引數存儲到一個全局數組execfile中，
                // 表示要加載的用戶程序。
            execfile[++execfileNum]= argv[++i];
                // execfile：表示需要執行的程序名稱，
                // 這是一個陣列，用於存儲傳遞給-e選項的程序名稱。
                // execfileNum：表示需要執行的程序數量，從1開始。
        cout << execfile[execfileNum] << "\n";
        } else if (strcmp(argv[i], "-ci") == 0) {
                // 引數是"-ci"，那麼從下一個引數中讀取一個字符串，
                // 表示要用作標準輸入的文件名。
            ASSERT(i + 1 < argc);
            consoleIn = argv[i + 1];
            i++;
        } else if (strcmp(argv[i], "-co") == 0) {
                // 引數是"-co"，那麼從下一個引數中讀取一個字符串，
                // 表示要用作標準輸出的文件名。
```

```
            ASSERT(i + 1 < argc);
            consoleOut = argv[i + 1];
            i++;
#ifndef FILESYS_STUB
        } else if (strcmp(argv[i], "-f") == 0) {
                // 引數是"-f"，那麼將formatFlag標誌設置為TRUE，
                // 表示需要格式化磁盤。
            formatFlag = TRUE;
#endif
        } else if (strcmp(argv[i], "-n") == 0) {
            // 引數是"-n"，那麼從下一個引數中讀取一個浮點數，
            // 用作網絡可靠性的參數。
            ASSERT(i + 1 < argc);   // next argument is float
            reliability = atof(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-m") == 0) {
            // 引數是"-m"，那麼從下一個引數中讀取一個整數，
            // 用作機器的id。
            ASSERT(i + 1 < argc);   // next argument is int
            hostName = atoi(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            // 引數是"-u"，那麼輸出Nachos的部分用法信息，包括各種命令列選項。
            cout << "Partial usage: nachos [-rs randomSeed]\n";
            cout << "Partial usage: nachos [-s]\n";
            cout << "Partial usage: nachos [-ci consoleIn] [-co consoleOut]\n";
#ifndef FILESYS_STUB
            cout << "Partial usage: nachos [-nf]\n";
#endif
            cout << "Partial usage: nachos [-n #] [-m #]\n";
        }
    }
}
```

1-3. threads/kernel.cc Kernel::ExecAll()

- 根據threads/kernel.cc文件說明，Kernel::ExecAll()是讓每個文件調用Exec()函數執行所有已載入的可執行文件(execfile)，並在所有文件被執行後終止當前線程。
- 簡述實作步驟:
    - 1.執行所有已載入的可執行文件(execfile)
    - 2.所有文件執行後終止當前thread

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        // 利用for迴圈遍歷系統中每個可執行的文件，
        // 從1到execfileNum（已加載的可執行文件的總數）。
        // 對於每個文件，該方法使用相應的文件名作為參數調用Exec()函數。
        int a = Exec(execfile[i]);
                // Exec()函數創建一個新的線程來執行指定的可執行文件，並返回一個整數值，
```

```
                    // 指示操作的成功或失敗。每次迭代中，該整數值存儲在變量"a"中。
    }
    currentThread->Finish();
        // 當所有可執行文件都被執行後，通過調用Finish()函數終止當前線程。
        // 該函數將當前線程的狀態設置為THREAD_ZOMBIE，並安排下一個線程運行。
}
```

1-3. threads/kernel.cc Kernel::Exec()

- 根據threads/kernel.cc文件說明，Kernel::Exec()函式創建一個新的執行緒來執行指定的可執行檔，在分叉該執行緒，在新執行緒中執行該文件之前設置其地址空間。
- 簡述實作步驟:
    - 1.建立新的thread
    - 2.設定新thread的地址空間AddrSpace
    - 3.呼叫Fork傳遞該thread參數
    - 4.回傳新建立的thread對應的threadNum

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
        // 使用 "new" 為新執行緒創建內存，並將執行緒名稱和編號作為參數傳遞。
    t[threadNum]->space = new AddrSpace();
        // 使用 "new" 為新執行緒的地址空間分配內存。
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
        // 函式在新執行緒上調用 Fork() 方法，
        // 將 ForkExecute() 函式的地址作為第一個參數，
        // 將新執行緒的指針作為第二個參數傳遞。
        // ForkExecute() 是負責在新執行緒中執行指定可執行檔的函式。
    threadNum++;
        // 增加 threadNum 變數的值

    return threadNum-1;
        // 返回新創建的執行緒的編號。
}
```

1-2. userprog/addrspace.cc AddrSpace::AddrSpace()

- 根據userprog/addrspace.cc文件說明，AddrSpace::AddrSpace()是AddrSpace的constructor，此constructor是在需要創建新的地址空間以執行user program時調用。
- 處理program memory(virtual memory)與physical memory的轉換關係，由於NachOS預設為uniprogramming，因此兩者為簡單的一對一關係，只有單一一個unsegmented page table。
- 簡述實作步驟:
    - 1.建立thread的pageTable
    - 2.將pageTable每一頁的virtualPage與physicalPage做1對1的對應
    - 3.將pageTable每一頁的valid設定為TRUE
    - 4.將pageTable每一頁的use、dirty、readOnly設定為False

```
//----------------------------------------------------------------------
// AddrSpace::AddrSpace
//  Create an address space to run a user program.
//  Set up the translation from program memory to physical
//  memory.  For now, this is really simple (1:1), since we are
//  only uniprogramming, and we have a single unsegmented page table
//----------------------------------------------------------------------

AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    // 構造函數創建了一個頁表，
    // 將用戶程序使用的virtual memroy mapping到系統可用的physical memory。
    // 由於系統是uniprogramming且僅有一個unsegmented page table，
    // 因此page table是簡單的一對一映射 one to one。
    for (int i = 0; i < NumPhysPages; i++) {
        // 初始化了page table中的每個項目
    // for now, virt page # = phys page #
        pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = i;
       // 將virtual page number設定為physical page number
    pageTable[i].valid = TRUE;
       // 表示該項目當前正在使用並映射了一個有效的內存頁
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
       // use和dirty標誌被設置為FALSE，表示該頁尚未被訪問或修改。
    pageTable[i].readOnly = FALSE;
       // readOnly標誌被設置為FALSE，表示該頁既可以讀取也可以寫入。
    }

    // zero out the entire address space
    bzero(kernel->machine->mainMemory, MemorySize);
    // 使用bzero()函數清空整個地址空間，將所有字節設置為零。
    // 這確保user program使用這塊空間之前，地址空間完全為空。
}
```

1-3. threads/kernel.cc Kernel::ForkExecute()

- 根據threads/kernel.cc文件說明，Kernel::ForkExecute()是當負責在新線程的地址空間中進行載入load並執行user program。
- 簡述實作步驟:
    - 1.利用thread名稱進行載入到address space判斷,判斷失敗則終止ForkExecute()
    - 2.載入成功則呼叫Execute()執行thread

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) {
        return;
```

```
        // executable not found
        // 先用AddrSpace類的Load()做檢查是否可以使用線程名稱指定的可執行文件，
        // 並且將線程名稱指定的可執行文件加載到線程的地址空間中。
        // 如果文件無法加載，則return，表示找不到可執行文件。
    }

    t->space->Execute(t->getName());
    // 如果可執行文件可以加載到地址空間中，
    // 該函數調用AddrSpace類的Execute()方法，
    // 並傳入可執行文件的名稱。

}
```

1-2. userprog/addrspace.cc AddrSpace::Load()

- 根據userprog/addrspace.cc文件說明，AddrSpace::Load()負責將user program對應的object code file 載入load進memory，如果能載入檔案成功回傳True，無法載入則回傳False。
- 簡述步驟如下:
    - 1.打開指定的文件，讀取文件的 NOFF 標頭，並檢查 NOFF magic number是否匹配。如果文件打開失敗或 NOFF magic number不匹配，則返回失敗或是終止。
    - 2.計算需要多少以page size為單位向上取整的size大小，用來存儲代碼段code segment、數據段data segment、堆棧user stack segment等內容，(用divRoundUp計算numPages，再向上取整後再更新所需的size大小)
    - 3.利用virtual memory與physical memory對應，個別將code segment、data segment copy到memory。
    - 4.載入成功則關閉executable file並回傳True。

```
//----------------------------------------------------------------------
// AddrSpace::Load
//  Load a user program into memory from a file.
//
//  Assumes that the page table has been initialized, and that
//  the object code file is in NOFF format.
//
//  "fileName" is the file containing the object code to load into 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;
    }
    // 文件打開失，敗返回False。
```

```
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    // 檢查 NOFF magic number是否匹配，NOFF magic number不匹配，則終止。
    ASSERT(noffH.noffMagic == NOFFMAGIC);


#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
    // 計算出需要多少內存空間，用於存儲代碼段、數據段、堆棧等內容。
    // 為了在內存中分配足夠的空間，需要將需要的空間大小轉換為內存頁的數量，這樣可以確保分配的內存空間是按頁對齊的。
    // 該函數使用 divRoundUp 函數來執行這個轉換。

    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;
    // 該函數使用 divRoundUp 函數來執行這個轉換。

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

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

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

    // 由於NachOS預設為uniprogramming,
    // 因此virtual memory與physical memory 1對1對應,
    // 直接從mainMory[noffH.code.virtualAddr]作為起始點,
    // 連續載入noffH.code.size大小,
    // 對應object code file位置inFileAddr

    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);
    }
```

```
    // 這段處理code segments

    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);
    }
    // 這段處理initData segments

#ifdef 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);
    }
    // 這段處理readonlyData segments
#endif

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

1-2. userprog/addrspace.cc AddrSpace::Execute()

- 根據userprog/addrspace.cc文件說明，AddrSpace::Execute()函數負責初始化使用者程式user program的暫存器register、還原頁表page table的狀態以及跳轉到user program的進入點，讓程式開始執行。
- 簡述步驟如下:
    - 1.將目前的執行緒的記憶體空間設為當前的記憶體空間。
    - 2.初始化使用者程式的暫存器。
    - 3.載入page table register，還原page table的狀態，讓使用者程式能夠存取虛擬記憶體。
    - 4.跳轉到使用者程式，從指定的進入點開始執行程式。

```
//----------------------------------------------------------------------
// AddrSpace::Execute
//  Run a user program using the current thread
//
//      The program is assumed to have already been loaded into
//      the address space
//
//----------------------------------------------------------------------

void
```

```
AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;
    // 假設程式已經載入到記憶體空間中，
    // 將目前的執行緒的記憶體空間設為當前的記憶體空間，
    // 方法內會將目前的記憶體空間指定給目前的執行緒的 space 成員變數。
    this->InitRegisters();
    // set the initial register values
    // AddrSpace::InitRegisters()初始化使用者程式的暫存器，
    // 該函數設定程式計數器program counter(PC)、
    // 堆疊指標 (SP) 和其他暫存器的初始值。
    this->RestoreState();
    // load page table register
    // AddrSpace::RestoreState()載入page table register，
    // 還原page table的狀態，
    // 讓使用者程式能夠存取虛擬記憶體。
    kernel->machine->Run();
    // jump to the user progam
    // Machine::Run()跳轉到使用者程式，從指定的進入點開始執行程式。
    ASSERTNOTREACHED();
    // machine->Run never returns;
    // the address space exits
    // by doing the syscall "exit"
    // 由於 machine->Run() 方法永遠不會返回，
    // 結尾使用一個斷言assert，以確保程式不會繼續執行到這行。
}
```

1-1. threads/thread.cc Thread::Fork()

- 根據threads/thread.cc文件說明，Thread::Fork()負責創建一個新線程並將其添加到ready queue就緒隊列中，從而允許caller與callee線程並行執行。
- Fork()有兩個參數，一個是VoidFunctionPtr類型的funtion pointer func，表示new thread需要執行的function；另一個是void類型的pointer arg，表示func function的參數。
- 實作步驟依序為:
    - 1. 分配出一塊stack空間 (Allocate a stack)。
    - 2. 初始化stack以便call去switch讓它執行procedure (Initialize the stack so that a call to SWITCH will cause it to run the procedure)。
    - 3. 設定interrupt Level為IntOff(禁用)，並且用oldLevel記錄原本中斷狀態。
    - 4. 將thread放入ready queue (Put the thread on the ready queue)。
    - 5. 恢復interrupt Level為oldLevel(原本中斷狀態)。

```
//----------------------------------------------------------------------
// Thread::Fork
//  Invoke (*func)(arg), allowing caller and callee to execute
//  concurrently.
//
//  NOTE: although our definition allows only a single argument
//  to be passed to the procedure, it is possible to pass multiple
```

```
//  arguments by making them fields of a structure, and passing a pointer
//  to the structure as "arg".
//
//  Implemented as the following steps:
//      1. Allocate a stack
//      2. Initialize the stack so that a call to SWITCH will
//      cause it to run the procedure
//      3. Put the thread on the ready queue
//
//  "func" is the procedure to run concurrently.
//  "arg" is a single argument to be passed to the procedure.
//----------------------------------------------------------------------


void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    // 建立interrupt與scheduler的pointer變數
    // 個別紀錄指向當前kernel的中斷interrupt和排程scehduler的pointer。
    IntStatus oldLevel;
    // 建立oldLevel變數，用來記錄中斷級別原本的狀態
    StackAllocate(func, arg);
    // StackAllocate() 函數來為線程分配一個新堆棧stack，
    // 並初始化它以運行指定的函數和給定的參數。

    oldLevel = interrupt->SetLevel(IntOff);
    // 透過SetLevel(IntOff)將中斷級別設置為禁用，
    // 並且回傳原本中斷級別狀態用oldLevel變數做紀錄
    scheduler->ReadyToRun(this);
    // ReadyToRun assumes that interrupts
    // are disabled!
    // Scheduler::ReadyToRun() 將新線程添加到就緒隊列中
    (void) interrupt->SetLevel(oldLevel);
    // 透過SetLevel(oldLevel)恢復中斷級別到其先前的狀態。
}
```

1-1. threads/thread.cc Thread::StackAllocate()

- 根據threads/thread.cc文件說明，Thread::StackAllocate()負責為新thread分配和初始化執行棧，函數最後返回一個指向新執行棧的pointer，並且該pointer將用於在後續的thread創建中指定執行棧。
- 實作步驟依序為:
    - 1. 使用AllocBoundedArray分配一塊大小為 StackSize * sizeof(int) 的內存給新線程的執行棧。
    - 2. 然後依據不同的平台設置執行棧的大小和結構。 其中將stackTop設定為stack頂端位置。 在某些平台上有利用FENCEPOST值做stack溢出檢查做安全措施。
    - 3. 函式將初始化好的機器狀態（machine state）寫入新線程對應的 machineState 數組。其中包括：
        - (1) PCState：程序計數器（Program Counter），指向執行棧中下一條要執行的指令。
        - (2) StartupPCState：啟動程序計數器，指向 ThreadBegin 函式。
        - (3) InitialPCState：初始程序計數器，指向要 fork 的函式 func。

- (4) InitialArgState：初始引數狀態，傳給 func 的參數 arg。
- (5) WhenDonePCState：執行完畢程式計數器狀態，當線程完成時跳轉的指令，指向 ThreadFinish 函式。

```
//----------------------------------------------------------------------
// Thread::StackAllocate
//  Allocate and initialize an execution stack.  The stack is
//  initialized with an initial stack frame for ThreadRoot, which:
//      enables interrupts
//      calls (*func)(arg)
//      calls Thread::Finish
//
//  "func" is the procedure to be forked
//  "arg" is the parameter to be passed to the procedure
//----------------------------------------------------------------------

void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    // 分配了一個大小為StackSize的整數數組


    // 堆棧初始化的具體細節因機器的架構而異，
    // 並由條件編譯器指令條件地編譯每個架構的適當代碼進行處理。
    // 據運行Nachos操作系統的機器的架構，
    // 它將堆棧指針（stackTop）設置為堆棧上的適當位置。
#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;
    // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
    stackTop = stack + StackSize - 96;
    // SPARC stack must contains at
    // least 1 activation record
    // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;
    // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4;
    // -4 to be on the safe side!
```

```
    *stack = STACK_FENCEPOST;
#endif

#ifdef ALPHA
    stackTop = stack + StackSize - 8;
    // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif


#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 addres
    // 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

    // ThreadRoot啟用中斷，調用指定的函數並使用給定的參數，
    // ThreadRoot執行指定的函數並在線程執行完成後，
    // 呼叫ThreadFinish進行清理。
#ifdef PARISC
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

1-4. threads/scheduler.cc Scheduler::ReadyToRun()

- 根據threads/scheduler.cc文件說明，Scheduler::ReadyToRun()是將指定的 thread 標記為 ready (準備執行) ，但還不會被執行，並將它放到 readyList 上等待後續被schedular安排到 CPU 上執行。
- ReadyToRun()傳入參數是要標記為 ready 的 thread。
- 實作步驟依序為:
    - 1. 檢查中斷是否已禁用，因為如果啟用了中斷，修改就準備好的列表可能會導致競爭條件。
    - 2. 將線程的狀態設置為 READY，表示此thread已經準備好執行。
    - 3. 將該線程加到準備列表readyList的末尾。

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
```

```
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 檢查當前中斷的狀態是否為關閉狀態。
    // 在加入就緒隊列之前必須禁用中斷，以避免競爭條件和不一致的狀態。
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    // 將 thread 的狀態設為 READY
    readyList->Append(thread);
    // 將 thread 添加到 readyList 之中，
    // 等待schedular決定何時執行該 thread
}
```

1-1. threads/thread.cc Thread::Finish()

- 根據threads/thread.cc文件說明，Thread::Finish()調用 Sleep()，從就緒列表中移除當前線程，以使其他線程運行，並透過禁用中斷以確保 Sleep()能安全執行。
- 實作步驟依序為:
    - 1. SetLevel(IntOff)設定interrupt level為禁用，使得Sleep()能安全執行。
    - 2. Sleep(TRUE)讓當前線程睡眠並從就緒列表中移除，使得其他線程可以運行。

```
//----------------------------------------------------------------------
// Thread::Finish
//  Called by ThreadRoot when a thread is done executing the
//  forked procedure.
//
//  NOTE: we can't immediately de-allocate the thread data structure
//  or the execution stack, because we're still running in the thread
//  and we're still on the stack!  Instead, we tell the scheduler
//  to call the destructor, once it is running in the context of a different
thread.
//
//  NOTE: we disable interrupts, because Sleep() assumes interrupts
//  are disabled.
//----------------------------------------------------------------------

//
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    // 禁用中斷，以確保可以安全地執行 Sleep()
    ASSERT(this == kernel->currentThread);
```

```
    // 斷言檢查當前線程與正在完成的線程是同一個
    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);
    // invokes SWITCH
    // 當前線程睡眠並從就緒列表中移除，使得其他線程可以運行。
    // 調用調度程序的 SWITCH 方法來選擇下一個要運行的線程。
    // not reached
}
```

1-1. threads/thread.cc Thread::Sleep()

- 根據threads/thread.cc文件說明，Thread::Sleep()是放棄 CPU，允許其他執行緒運行，同時等待同步變數訊號。如果沒有執行緒可以運行，該方法將使 CPU 進入閒置狀態，直到發生中斷。
- Thread::Sleep()參數為finishing，表示呼叫 Sleep() 的執行緒是否已經完成執行。如果 finishing 為 true，schedular最終將在不同執行緒的上下文中呼叫該執行緒的deconstrutor。
- 實作步驟依序為:
    1. 斷言判斷當前執行緒是調用 Sleep() 的執行緒，並且中斷已經被禁用。
    2. 將該執行緒的狀態被設置為 BLOCKED，表示它未準備好運行，正在等待事件發生。
    3. 使用while迴圈呼叫schedular的 FindNextToRun()在就緒列表readylist以尋找下一個要運行的執行緒。
    4. 如果沒有執行緒在就緒列表readylist上，該方法將調用 Interrupt::Idle() 方法使 CPU 進入閒置狀態，直到下一個 I/O 中斷發生。
    5. 一旦找到要運行的執行緒，該方法調用schedular的Run()以切換到該執行緒並開始執行它。

```
//----------------------------------------------------------------------
// Thread::Sleep
//  Relinquish the CPU, because the current thread has either
//  finished or is blocked waiting on a synchronization
//  variable (Semaphore, Lock, or Condition).  In the latter case,
//  eventually some thread will wake this thread up, and put it
//  back on the ready queue, so that it can be re-scheduled.
//
//  NOTE: if there are no threads on the ready queue, that means
//  we have no thread to run.  "Interrupt::Idle" is called
//  to signify that we should idle the CPU until the next I/O interrupt
//  occurs (the only thing that could cause a thread to become
//  ready to run).
//
//  NOTE: we assume interrupts are already disabled, because it
//  is called from the synchronization routines which must
//  disable interrupts for atomicity.   We need interrupts off
//  so that there can't be a time slice between pulling the first thread
//  off the ready list, and switching to it.
//----------------------------------------------------------------------
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
```

```
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言判斷當前執行緒是調用 Sleep() 的執行緒，並且中斷已經被禁用

    DEBUG(dbgThread, "Sleeping thread: " << name);
    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " <<
kernel->stats->totalTicks);

    status = BLOCKED;
    // 將該執行緒的狀態被設置為 BLOCKED
    // 表示它未準備好運行，正在等待事件發生
    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    // 使用while迴圈呼叫schedular的 FindNextToRun()
    // 在就緒列表readylist以尋找下一個要運行的執行緒
        kernel->interrupt->Idle();
        // no one to run, wait for an interrupt
        // 如果沒有執行緒在就緒列表readylist上
        // 該方法將調用 Interrupt::Idle() 方法使 CPU 進入閒置狀態
        // 直到下一個 I/O 中斷發生
    }

    kernel->scheduler->Run(nextThread, finishing);
    // returns when it's time for us to run
    // 一旦找到要運行的執行緒
    // 該方法調用schedular的Run()以切換到該執行緒並開始執行它
}
```

1-4. threads/scheduler.cc Scheduler::Run()

- 根據threads/scheduler.cc文件說明，Scheduler::Run()是用於切換到下一個執行緒的運行。
- 實作步驟依序為:
    1. 將當前的執行緒存儲在 oldThread 中，並且使用 ASSERT 驗證中斷狀態是否為關閉狀態（IntOff）。
    2. 假如finishing 為 true，表示 oldThread 已經完成任務，需要標記為 toBeDestroyed，在後續的清理過程中刪除該執行緒。
    3. 如果 oldThread 是一個用戶程序執行緒，則保存用戶程序的 CPU 狀態和地址空間狀態。
    4. 調用 oldThread->CheckOverflow() 函數檢查舊的執行緒是否存在堆疊溢出的情況。
    5. kernel->currentThread 指向下一個執行緒 nextThread，並將 nextThread 的狀態設置為 RUNNING，表示下一個執行緒正在運行。
    6. 調用 SWITCH(oldThread, nextThread) 函數切換到下一個執行緒的運行。SWITCH 函數是一個與機器相關的組合語言例程，用於在不同的執行緒之間進行切換。
    7. 當 SWITCH 函數返回時，調用 CheckToBeDestroyed() 函數檢查以前的執行緒是否已經完成任務並且需要清理。
    8. 如果 oldThread 是一個用戶程序執行緒，則恢復用戶程序的 CPU 狀態和地址空間狀態。

```
//----------------------------------------------------------------------
// Scheduler::Run
//  Dispatch the CPU to nextThread.  Save the state of the old thread,
//  and load the state of the new thread, by calling the machine
```

```
//  dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//  already been changed from running to blocked or ready (depending).
// Side effect:
//  The global variable kernel->currentThread becomes nextThread.
//
//  "nextThread" is the thread to be put into the CPU.
//  "finishing" is set if the current thread is to be deleted
//      once we're no longer running on its stack
//      (when the next thread starts running)
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    // 將當前的執行緒存儲在 oldThread

    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 使用 ASSERT 驗證中斷狀態是否為關閉狀態（IntOff）
    if (finishing) {
    // mark that we need to delete current thread
    // finishing 為 true，表示 oldThread 已經完成任務，

         ASSERT(toBeDestroyed == NULL);
     toBeDestroyed = oldThread;
        // 需要標記為 toBeDestroyed，在後續的清理過程中刪除該執行緒。
    }

    if (oldThread->space != NULL) {
    // if this thread is a user program,
        oldThread->SaveUserState();
        // save the user's CPU registers
    oldThread->space->SaveState();
    // 如果 oldThread 是一個用戶程序執行緒，
    // 則保存用戶程序的 CPU 狀態和地址空間狀態。
    }


    oldThread->CheckOverflow();
    // check if the old thread
    // had an undetected stack overflow
    // 調用 oldThread->CheckOverflow() 函數
    // 檢查舊的執行緒是否存在堆疊溢出的情況。
    kernel->currentThread = nextThread;
    // switch to the next thread
    // kernel->currentThread 指向下一個執行緒 nextThread
    nextThread->setStatus(RUNNING);
    // nextThread is now running
    // 並將 nextThread 的狀態設置為 RUNNING，表示下一個執行緒正在運行

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " <<
nextThread->getName());
```

```
    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);
    // 調用 SWITCH(oldThread, nextThread)函數切換到下一個執行緒的運行
    // SWITCH 函數是一個與機器相關的組合語言例程，
    // 用於在不同的執行緒之間進行切換。
    // we're back, running oldThread

    // 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
    // 當 SWITCH 函數返回時，
    // 調用 CheckToBeDestroyed() 函數
    // 檢查以前的執行緒是否已經完成任務並且需要清理。
    if (oldThread->space != NULL) {
    // if there is an address space
        oldThread->RestoreUserState();
        // to restore, do it.
    oldThread->space->RestoreState();
    // 如果 oldThread 是一個用戶程序執行緒，
    // 則恢復用戶程序的 CPU 狀態和地址空間狀態。
    }
}
```

Problems: Explain how NachOS creates a thread (process), load it into memory and place it into the scheduling queue as requested in the Trace code part

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

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

- 1. AddrSpace():負責為新的thread初始化新的地址空間，宣告一個PageTable將virtual Page對應到physical Page。
- 2. StackAllocate() 負責為新thread分配和初始化執行棧，函數最後返回一個指向新執行棧的pointer，並且該pointer將用於在後續的thread創建中指定執行棧。
- 3. Load()負責將user program對應的object code file 載入load進memory。利用page table(也就是virtual memory與physical memory對應關係)，個別將code segment、data segment copy到memory。

How does Nachos create and manage the page table?

- AddrSpace():負責為新的thread初始化新的地址空間，宣告一個PageTable將virtual Page對應到physical Page。由於NachOS預設為uniprogramming，因此在AddrSpace()中建立的PageTable是以virtul memory與physical memory以1對1映射的方式進行。

How does Nachos translate addresses?

- 在Nachos中，地址轉換是由頁表完成的，頁表是一種將虛擬地址映射到物理地址的數據結構。每個線程都有自己的頁表，由AddrSpace類創建和管理。由於NachOS預設為uniprogramming，因此在AddrSpace()中建立的PageTable是以virtul memory與physical memory以1對1映射的方式進行轉址。
- 若要讓NachOS支援multiprogramming，則需要將virtual address/page size得到virtual page number以及virtual address%page size得到page offset。而virtual page number是拿來當作page table的index找到physical page number,而page offset則是在physical page的具體位置。physical page number與page offset兩者結合可轉換為physical address。

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

- 在 Nachos 中，在運行一個線程（process）之前，需要初始化機器狀態，包括初始化 CPU 註冊和設置堆棧指針。
- Nachos核心會在啟動時呼叫Kernel::Kernel()初始化機器狀態 利用command line arguments 初始 NachOS kernel各項機器參數設定。
- 當創建一個新線程時，Nachos核心會在 Thread::StackAllocate()負責初始化好的機器狀態（machine state）寫入新線程對應的 machineState數組
- 其中包括： - (1) PCState：程序計數器（Program Counter），指向執行棧中下一條要執行的指令。 - (2) StartupPCState：啟動程序計數器，指向 ThreadBegin 函式。 - (3) InitialPCState：初始程序計數器，指向要 fork 的函式 func。 - (4) InitialArgState：初始引數狀態，傳給 func 的參數 arg。 - (5) WhenDonePCState：執行完畢程式計數器狀態，當線程完成時跳轉的指令，指向 ThreadFinish 函式。

Which object in Nachos acts the role of process control block

```
Thread::Thread(char* threadName, int threadID)
{
    ID = threadID;
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
    machineState[i] = NULL;     // not strictly necessary, since
                    // new thread ignores contents
                    // of machine registers
    }
    space = NULL;
}
```

- 在Nachos中，充當進程控制塊（process control block）角色的物件稱為Thread。每個Thread物件代表Nachos內核中的一個執行緒（thread）。

- Thread類別包括各種成員變數，用於存儲關於該執行緒狀態的信息，例如其程式計數器（program counter）、堆疊指標（stack pointer）和暫存器值。它還包括管理執行緒執行的方法，例如Fork()，用於創建新的執行緒，以及Yield()，用於將執行緒讓出CPU給其他執行緒。
- 在Nachos中，Thread類別作為進程控制塊，通過代表和管理系統中單個執行緒的狀態。

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

```
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    // 建立interrupt與scheduler的pointer變數
    // 個別紀錄指向當前kernel的中斷interrupt和排程scehduler的pointer。
    IntStatus oldLevel;
    // 建立oldLevel變數，用來記錄中斷級別原本的狀態
    StackAllocate(func, arg);
    // StackAllocate() 函數來為線程分配一個新堆棧stack，
    // 並初始化它以運行指定的函數和給定的參數。

    oldLevel = interrupt->SetLevel(IntOff);
    // 透過SetLevel(IntOff)將中斷級別設置為禁用，
    // 並且回傳原本中斷級別狀態用oldLevel變數做紀錄
    scheduler->ReadyToRun(this);
    // ReadyToRun assumes that interrupts
    // are disabled!
    // Scheduler::ReadyToRun() 將新線程添加到就緒隊列中
    (void) interrupt->SetLevel(oldLevel);
    // 透過SetLevel(oldLevel)恢復中斷級別到其先前的狀態。
}
```

- threads/thread.cc文件說明，Thread::Fork()負責創建一個新線程並將其添加到ready queue就緒隊列中，從而允許caller與callee線程並行執行。
- 實作步驟依序為:
    1. 分配出一塊stack空間。 (Allocate a stack)。
    2. 初始化stack以便call去switch讓它執行procedure。 (Initialize the stack so that a call to SWITCH will cause it to run the procedure)。
    3. 設定interrupt Level為IntOff(禁用)，並且用oldLevel記錄原本中斷狀態。
    4. 將thread放入ready queue。 (Put the thread on the ready queue)。
    5. 恢復interrupt Level為oldLevel(原本中斷狀態)。

Part 2 Implement page table in NachOS

- Hint: The following files “may” be modified…
    - userprog/addrspace.*
    - threads/kernel.*
    - machine/machine.h

Modified Codes

threads/kerenl.h && threads/kerenl.cc

```
class Kernel {
  public:
    static int usedPhysicalPages[NumPhysPages];
    // build a array to record used Physical Pages in the kernel.h
    // make sure threads won't use the same physical page
    // and prevent them covering each other
};
```

- 依據requirement要求，在kerenl.h的Kernel constructor放入一個static array紀錄已被使用過的PhysicalPages，設定為static是為了要讓所有的thread都能共享這個usedPhysicalPages Array資訊，避免將program load進來的時候，會互相覆蓋同一塊PhysicalPages而造成錯誤。

threads/kerenl.cc

```
void
Kernel::Initialize()
{

    usedPhysicalPages[NumPhysPages];
    // declare the static array again in the kernel.cc
}
```

- 同樣在kerenl.cc再次宣告static array usedPhysicalPages

userprog/addrspace.cc

```
#include "copyright.h"
#include "main.h"
#include "addrspace.h"
#include "machine.h"
#include "noff.h"

int Kernel::usedPhysicalPages[NumPhysPages]={0};
// initialized the static array usedPhysicalPages in the addrspace.cc
```

- 在addrspace.cc的開頭將usedPhysicalPages進行初始化，讓addrspace的相關member function能夠透過kernel使用這個static array。

userprog/addrspace.cc AddrSpace::AddrSpace()

```
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);
    */
}
```

- 由於原本NachOS預設為uniprograming,因此virtual memory與physical memory為一對一對應，但為了要調整為multiprogramming，因此把預設的pagetable對應表註解，因為需要先計算所需的空間大小才知道要怎麼分配空間，因此延後pagetable設定到AddrSpace::Load()實作。

userprog/addrspace.cc AddrSpace::~AddrSpace()

```
AddrSpace::~AddrSpace()
{
   // When the thread is finished,
   // make sure to release the address space and restore physical page status.
   for(int i=0;i< numPages;i++)
   {
        kernel->usedPhysicalPages[pageTable[i].physicalPage]=0;
        // every thread own their pagetable,
        // so use the numPages(virtual) to use pageTable[i].physicalPage
        // to find corresponding usedPhysicalPages
        // usedPhysicalPages[pageTable[i].physicalPage]=0
        // is to release the PhysicalPages which thread uses
   }

   delete []pageTable;
}
```

- 在~AddrSpace()中，因為每個thread都有自己的pagetable，利用thread的virtual number of Pages透過pageTable[i].physicalPage找出對應thread所使用的PhysicalPages，由於usedPhysicalPages為static，因此所有的thread都能看到這個static array。將對應使用的PhysicalPages歸零則代表thread執行完畢後釋放了PhysicalPages給其他thread使用。
- 最後將thread所屬的pagetable刪除。

userprog/addrspace.cc AddrSpace::Load() <僅擷取一部分>

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

#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);
    size = numPages * PageSize;

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

    pageTable = new TranslationEntry[numPages];
    for(unsigned int i = 0; i < numPages; i++){
        pageTable[i].virtualPage = i;
        int index = 0;
        while(index < NumPhysPages && kernel->usedPhysicalPages[index] ==
1)index++;
        // find the usedPhysicalPages
        if(index == NumPhysPages) ExceptionHandler(MemoryLimitException);
        // if there is no usedPhysicalPage,then it means the insufficent
memory,call exception
        pageTable[i].physicalPage = index;
        kernel->usedPhysicalPages[index] = 1;
        pageTable[i].valid = true;
        pageTable[i].use = false;
        pageTable[i].dirty = false;
        pageTable[i].readOnly = false;
    }

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

// 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);

        int page_number = noffH.code.virtualAddr / PageSize;
        // calculate the page_number
```

```
        int page_offset = noffH.code.virtualAddr % PageSize;
        // calculate the page_offset
        int remaining_Data_Size = noffH.code.size;
        // record the remaing_Data_Size
        int inFileAddr = noffH.code.inFileAddr;
        // record the inFileAddr

        while (remaining_Data_Size > 0) {
            int Data_to_Load_Size = std::min(PageSize-page_offset,
remaining_Data_Size);
            // calculate the the data size can be loaded this time
            // first time would be start from page_offset to the end of the page
            // last time would be the remaining data size which is less than a
page size

            executable->ReadAt(
                &(kernel->machine->mainMemory[pageTable[page_number].physicalPage
* PageSize + page_offset]),
                Data_to_Load_Size, inFileAddr);
                // load data from the inFileAddrr(which is from the disk) to the
mainMemory
                // and copy the Data_to_Load_Size from the inFileAddr to
mainMemory
                // the physical address in the mainMemory would be
                // the the the index of physical Page * PageSize + page_offset

            inFileAddr += Data_to_Load_Size;
            // change the start point from the inFileAddr
            remaining_Data_Size -= Data_to_Load_Size;
            // update the remaining_Data_Size
            page_offset = 0;
            // from the start of the page
            page_number++;
            // page_number+1, move to next virtual page
        }
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);

        int page_number = noffH.initData.virtualAddr / PageSize;
        // calculate the page_number
        int page_offset = noffH.initData.virtualAddr % PageSize;
        // calculate the page_offset
        int remaining_Data_Size = noffH.initData.size;
        // record the remaing_Data_Size
        int inFileAddr = noffH.initData.inFileAddr;
        // record the inFileAddr
        while (remaining_Data_Size > 0) {
            int Data_to_Load_Size = std::min(PageSize-page_offset,
remaining_Data_Size);
            // calculate the the data size can be loaded this time
            // first time would be start from page_offset to the end of the page
            // last time would be the remaining data size which is less than a
```

```
page size
            executable->ReadAt(
                &(kernel->machine->mainMemory[pageTable[page_number].physicalPage
* PageSize + page_offset]),
                Data_to_Load_Size, inFileAddr);
                // load data from the inFileAddrr(which is from the disk) to the
mainMemory
                // and copy the Data_to_Load_Size from the inFileAddr to
mainMemory
                // the physical address in the mainMemory would be
                // the the the index of physical Page * PageSize + page_offset
            inFileAddr += Data_to_Load_Size;
            // change the start point from the inFileAddr
            remaining_Data_Size -= Data_to_Load_Size;
            // update the remaining_Data_Size
            page_offset = 0;
            // from the start of the page
            page_number++;
            // page_number+1, move to next virtual page
        }
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
        DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " <<
noffH.readonlyData.size);

        int page_number = noffH.readonlyData.virtualAddr / PageSize;
        // calculate the page_number
        int page_offset = noffH.readonlyData.virtualAddr % PageSize;
        // calculate the page_offset
        int remaining_Data_Size = noffH.readonlyData.size;
        // record the remaing_Data_Size
        int inFileAddr = noffH.readonlyData.inFileAddr;
        // record the inFileAddr

        while (remaining_Data_Size > 0) {
            int Data_to_Load_Size = std::min(PageSize-page_offset,
remaining_Data_Size);
            // calculate the the data size can be loaded this time
            // first time would be start from page_offset to the end of the page
            // last time would be the remaining data size which is less than a
page size
            executable->ReadAt(
                &(kernel->machine->mainMemory[pageTable[page_number].physicalPage
* PageSize + page_offset]),
                Data_to_Load_Size, inFileAddr);
                // load data from the inFileAddrr(which is from the disk) to the
mainMemory
                // and copy the Data_to_Load_Size from the inFileAddr to
mainMemory
                // the physical address in the mainMemory would be
                // the the the index of physical Page * PageSize + page_offset
```

```
            inFileAddr += Data_to_Load_Size;
            // change the start point from the inFileAddr
            remaining_Data_Size -= Data_to_Load_Size;
            // update the remaining_Data_Size
            page_offset = 0;
            // from the start of the page
            page_number++;
            // page_number+1, move to next virtual page
        }
    }
#endif

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

- 調整pagetable對應關係，利用usedPhysicalPages找到尚未使用的PhysicalPages空間，將code segment與data segment透過page table,virtual address與physical address轉換計算，從inFileAddr(virtual)將資料copy到mainMemory(physical)。
- 在當出現所有PhysicalPages皆被使用的情況下，代表沒有足夠的空間可以給當前的thread把資料load進Physical Memory(insuffcient memory)，此時會呼叫exceptionhandler進行例外處理。
- 計算方法補充:因為原本uniProgramming，所以將virtualAddr直接放入檔案結構對應出physicalAddr ，若改成multiprogramming則需要修正，找出page table映射後的實體位置，所以利用virtualAddr先除 PageSize 求得是第幾個 page，然後索引 pageTable 找到對應的實體頁是第幾頁，接著乘上每個 page 的大小得到該實體頁的實體記憶體，此時就知道在第幾個實體頁了，但不知道在這一頁的哪裡，所以我們拿本來的 virtualAddr mod PageSize 求得在 page 內的偏移offset，加上剛剛拿到的實體頁，就是對應的實體位址。

```
#### machine/machine.h

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,
             // when there is no more physical page that the currnet thread can
use,
             // which means it occurs Insufficient memory,
             // so this situation belongs to MemoryLimitException
             NumExceptionTypes
};
```

- 在machine.h新增MemoryLimitException(對應錯誤碼為8)，以利在ExceptionHandler()做對應Exception處理。

userprpg/exception.cc ExceptionHandler()

```
void ExceptionHandler(ExceptionType which) {
  char ch;
  int val;
  int type = kernel->machine->ReadRegister(2);
  int status, exit, threadID, programID, fileID, numChar, size;
  DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
  DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << "
type: " << type << ", " << kernel->stats->totalTicks);
  switch (which) {

    case MemoryLimitException:
      cerr << "Unexpected user mode exception " << (int)which << "\n";
      ASSERT(false);
      // in this moment, it must occur MemoryLimitException,
      // so we just use  ASSERT(false) to trigger assertion messages and abort
      break;
    default:
      cerr << "Unexpected user mode exception " << (int)which << "\n";
      break;
  }
  ASSERTNOTREACHED();
}
```

- 在ExceptionHandler()新增MemoryLimitException，使得程式能夠處理insuffient memory的情況。在case為MemoryLimitException，則印出"Unexpected user mode exception 8"的錯誤訊息，接著因為進入到此階段代表一定有發生MemoryLimitException透過asset(false)先印出Assert error message與呼叫Abort終止程式。

Verification

Correct results with multiprogramming 執行結果

```
// Definitions related to the size, and format of user memory

const int PageSize = 128;       // set the page size equal to
                    // the disk sector size, for simplicity

//
// You are allowed to change this value.
// Doing so will change the number of pages of physical memory
// available on the simulated machine.
```

```
//
const int NumPhysPages = 128;
```

- 在machine.h中可看出PageSize與NumPhysPages數值預設值皆為128，經過multiprogramming調整後，load進兩個userprogram，確實有按照program 1與program2個別分開印出遞減與遞增數值，代表兩個program產生的thread會確實load到不同的physical page，而沒有互相覆蓋資料的情況。

```
[os23s77@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

Correctly handle the exception about insufficient memory 執行結果

```
// Definitions related to the size, and format of user memory

const int PageSize = 4;         // set the page size equal to
                    // the disk sector size, for simplicity

//
// You are allowed to change this value.
// Doing so will change the number of pages of physical memory
// available on the simulated machine.
//
const int NumPhysPages = 4;
```

- 在machine.h中降低PageSize與NumPhysPages數值，讓load進第一個userprogram時就會發生insufficient memory的情況，由下圖可看出確實有針對insufficient memory做出正確的exception處理。

```
[os23s77@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
Unexpected user mode exception 8
Assertion failed: line 201 file ../userprog/exception.cc
Aborted
[os23s77@localhost test]$
```