CS342301: Operating System

MP2: Multi-Programming

組別: 8

組員:

109062315 洪聖祥

109062314 張紘齊

項目	分工
Trace Code	都有看過全部
Report	
Thread::Sleep()	張紘齊
Thread::StackAllocate()	張紘齊
Thread::Finish()	張紘齊
Thread::Fork()	張紘齊
AddrSpace::AddrSpace()	洪聖祥
AddrSpace::Execute()	洪聖祥
AddrSpace::Load()	張紘齊
Kernel::Kernel()	洪聖祥
Kernel::ExecAll()	洪聖祥
Kernel::Exec()	洪聖祥
Kernel::ForkExecute()	洪聖祥
Scheduler::ReadyToRun()	張紘齊
Scheduler::Run()	張紘齊
Implementation	洪聖祥
Debug	張紘齊

Part I: Trace Code

1-1. threads/thread.cc

Thread::Fork()

```
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);
}
```

主要是讓 caller 和 callee 兩個 thread 能夠 execute concurrently。首先呼叫 Thread::StackAllocate 替這個 thread 配置一個 stack，並初始化相關變數以及 registers 的內容，接著讓 scheduler 把這個 thread 放到 ready queue 上，然後要注意這個動作必須是要在 interrupt disabled 的情況下才能執行

Thread::StackAllocate()

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

首先配置一塊連續的記憶體給 stack，讓 thread 能存放 temporary data

```
#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
```

接著設定 current stack pointer (stackTop) 的位置、放好 ThreadRoot、將 Bottom of the stack (*stack) 設為 STACK_FENCEPOST 來判斷 stack overflows

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

最後設定 registers 的值，讓 Context Switch 發生時能夠切換到當前的 thread

Thread:: Finish()

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);                // invokes SWITCH
    // not reached
}
```

主要是處理 thread 執行完成，這裡的方法是透過呼叫 Thread::Sleep，並給予參數 TRUE，代表說這個 thread 之後都用不到，相關資料可以移除

Thread:: Sleep()

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

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

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

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

主要是處理 thread 要被 blocked 或是 finished，因此 status 會被設為 BLOCKED。接下來會透過 while 迴圈判斷當前 ready queue 上是否有其他 thread，沒有的話就執行 interrupt->Idle 去看有沒有 interrupt 要發生，直到 ready queue 能有可以運行的 thread，否則程式就會被停止運行；有的話就請 scheduler 執行那個 thread，並將參數 finishing 放入，用來判斷當前的 thread 是否已完成任務

1-2. 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);
}
```

AddrSpace 的 constructor 初始化 pageTable，但是現在是沒有 multiprogramming 的版本，所以 virtualPage 直接對應到 physicalPage。並且清空所有 address space。但是 multiprogramming 的版本不應該直接清空所有 address space，因為有可能在 address space 有其他 process 在 memory。並且 physicalPage 也需要做額外的處理。

AddrSpace:: Execute()

```
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 progam

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

首先設定 current thread 的 address space 是現在的 address space，然後清零所有 register，把 page table load 進來，最後執行 user program。

AddrSpace:: Load() 主要就是把 user program 放入 memory 之中

```
OpenFile *executable = kernel->fileSystem->Open(fileName);
NoffHeader noffH;
unsigned int size;

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

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

首先根據 filename 去找對應的檔案，若找不到則直接 return 並回傳 False。接著判斷是否要呼叫 AddrSpace::SwapHeader 將 little endian 轉成 big endian。

```
#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
```

這裡是在計算這個 file 各個部分大小的總和 (包含了 thread stack 的大小)，接著計算所需用到的 page 數量，由於可能不是 page size 的整數倍，因此計算結果需要無條件進位，最後在判斷所需的 page 數量是否小於 physical memory 中能夠容納的 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);
}
```

剩下部分以 code segment 為例，由於當前的實作 virtual address 和 physical address 相同，因此不需要呼叫 AddrSpace::Translate，即可直接用 virtual address 去 memory 對應的位置將 code segment 放入 memory 之中

1-3. threads/kernel.cc

Kernel:: Kernel()

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

Kernel constructor 處理 command line 指令，這邊以 -e “filename”為例，execfile array 存放每個 file name。並且 execfileNum 紀錄總共有幾個 file。

Kernel:: ExecAll()

Kernel 對每一個執行檔呼叫 Exec(char* name)，呼叫完結束 current thread。

Kernel:: Exec()

每一個 program 以一個 thread 為單位執行(不考慮 multithreading)。所以 instantiate Thread class。每一個 thread 要分配 memory space 給 process，所以 instantiate AddrSpace class。最後呼叫 thread->Fork() ，初始化 thread state 和分配 stack 給 thread。

Kernel:: ForkExecute()

把一個 file(program) load 到 memory，並且執行。

1-4. threads/scheduler.cc

Scheduler:: ReadyToRun()

```
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);
}
```

將 thread 的 status 切換成 READY，然後把此 thread 放入 ready queue 的最後面

Scheduler:: Run()

```
Thread *oldThread = kernel->currentThread;

ASSERT(kernel->interrupt->getLevel() == IntOff);

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

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

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

這裡主要是在對 current thread (oldThread) 做處理，像是這個 thread 是否將要被刪除、需不需要將它的 CPU registers、address space 相關資訊存起來、thread 中的 stack 是否發生 overflow。接著把 current thread 切換成 nextThread，並將之 status 設為 RUNNING

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

這裡會執行用 assembly language 寫好的 context switch 函式

```
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();
}
```

做好 context switch 之後，由於現在已不會使用到 oldThread 中的資訊，因此可以呼叫 Scheduler::CheckToBeDestroyed 去把該 thread 的資源清除掉。最後假如該 thread 未來還會被執行(thread 的 address space 不為空)，則須將相關的資料記錄下來

Part II: How Nachos creates a thread(process), load it into memory and place it into the scheduling queue

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

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

    return threadNum-1;
```

在 Kernel::Exec 中的第二行會 new AddrSpace()，並把它放入 thread 中的 space 變數，也就是替 thread allocate memory space

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

```
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);
}
```

在上一題中，呼叫 new AddrSpace() 就會透過 bzero 先初始化 memory content，以及創立 page table 並初始化各個 entry 的資訊

```
machineState[InitialPCState] = (void*)func;
machineState[InitialArgState] = (void*)arg;
```

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

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

}
```

接下是當要執行 thread 的 function 時，會呼叫 machineState[InitialPCState] 中的 Kernel::ForkExecute，其所需參數會存在 machineState[InitialArgState]。然後在 Kernel::ForkExecute 中，可看到會根據 thread name 呼叫 AddrSpace::Load，此時就會將該 thread 所需用到的 binary code load 進 memory 裡。

- How does Nachos create and manage the page table?

  原本的設計是根據 uniprogramming，因此一開始在呼叫 new AddrSpace() 時，就會根據 NumPhyPages 來建立 page table，而至於管理，就只有一開始的新增和最後的刪除，且因為設計是 uniprogramming，context switch 也不該發生，因此也不會呼叫到 AddrSpace::RestoreState()去切換 page table 的相關資訊。

  而我們的實作會 maintain freeFrame list，並根據 user program 的大小去建立 page table，並對每個 virtual address 分配 freeFrame list 裡的 physical frame，且在 thread 的 address space 要被刪除時，會將它所佔有的 frame 內容清空，並 append 回 freeFrame list 中，最後才刪除 page table

- How does Nachos translate addresses?

  Original: does not translate address, virtual address = physical address
  Implementation(multiprogramming): in function Translate(), it first find the virtual page number by dividing virtual address by page size (page number = virtual address / page size) and find offset by taking remainder of virtual address divide by page size (offset = virtual address % page size). Then, access the page table's virtual page number index to find the corresponding TranslationEntry, in TranslationEntry saves the physical page number. Last, calculate physical address by multiplying page size and physical page number and add offset(physical address = page size*physical page number + offset).

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

  When tracing code, there is a comment saying "A thread running a user program actually has 2 sets of CPU registers – one is for its state while executing user code, one for its state while executing kernel code".
  Before running a thread(process):
  Initialize registers while executing kernel code

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

  In AddrSpace::Execute():
  Initialize user-level registers and load page table register

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

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

  Load page table

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

Zero out all registers and initialize PC register Next PC register Stack register.

```
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);
    DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}
```

- Which object in Nachos acts the role of process control block

Thread object acts the role of PCB because of the following features :

1.ThreadStatus{JUST_CREATED,RUNNING,READY,BLOCKED,ZOMBIE} status: which corresponds to the 5 state New, Run, Read, Wait, Terminate state in process state

2.Stack

3.ID: process's unique pid

4.userRegisters[NumTotalRegs]: user-level CPU register state

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

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

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

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);    // ReadyToRun assu
                                    // 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);
}
```

Thread::Fork 和 Thread::Yield 之中，scheduler 會將當前 thread append 到 ready queue 的最後面。而兩著各別會在 Kernel::Exec 和 Interrupt::Onetick 被呼叫到

Part III: Implementation

kernel.h

```
typedef int PhyFrameNum;
```

```
List<PhyFrameNum> *freeFrameList;
```

在 kernel.h 中的 class Kernel 新增一個資料結構叫做 freeFrameList。freeFrameList 主要是要記錄 physical memory 中哪些 frame 是 free 的，這樣在為 process 分配 memory 時，一定會分配到 memory 是 free 的 frame，不會有被分配到明明這個 frame 已經有其他 process 佔據了還被分配到的情況。freeFrameList 以 linked list 實作，可以用 list.h 中的 class List 直接呼叫。

kernel.cc

```
// initialize free frame list
// all frames are free in the beginning
freeFrameList = new List<PhyFrameNum>;
for (int i = 0 ; i < NumPhysPages ; i++){
    freeFrameList->Append(i);
}
```

在 class Kernel 的 constructor 中初始化 freeFrameList。一開始所有在 physical memory 的 frame 都是 free 的，所以將每一個 frame number 都 append 到 freeFrameList 中。

addrspace.cc

AddrSpace::AddrSpace()

```
AddrSpace::AddrSpace()
{
}
```

將 class AddrSpace 的 constructor 中對 pageTable 的 initialization 刪除。因為原來的實作是初始化 NumPhysPages (128) 個 TranslationEntry，但是其實不需要這麼多的 TranslationEntry，只需要 program size / PageSize 個 TranslationEntry。另

一個原因是 pageTable 其實也會存在 physical memory 中，多餘的 TranslationEntry 沒用只會造成 memory waste。因為我們在處理 load program 的時候才會知道 process size，所以我把 pageTable 的初始或移到 AddrSpace::Load(char* filename) 中。

AddrSpace::Load(char* filename)

Line 143-160 implementation

```
137     numPages = divRoundUp(size, PageSize);
138     size = numPages * PageSize;
139     //ASSERT(numPages <= NumPhysPages); // check we're not trying
140                                         // to run anything too big --
141                                         // at least until we have
142                                         // virtual memory
143     ExceptionType exception;
144     List<PhyFrameNum> *freeFrameList = kernel->freeFrameList;
145     // check if enough free frames
146     if(freeFrameList->NumInList() < numPages ){
147         ExceptionHandler(MemoryLimitException);
148         return false;
149     }
150     pageTable = new TranslationEntry[numPages];
151     for(int i = 0 ; i < numPages ; i++){
152         int phyPageNum = freeFrameList->Front();
153         freeFrameList->RemoveFront();
154         pageTable[i].virtualPage = i;
155         pageTable[i].physicalPage = phyPageNum;
156         pageTable[i].valid = TRUE;
157         pageTable[i].use = FALSE;
158         pageTable[i].dirty = FALSE;
159         pageTable[i].readOnly = FALSE;
160     }
```

我們在 AddrSpace::Load(char* filename) 可以知道這個 program 總共有幾個 page (numPages 在 137 行計算)並且初始化 pageTable，總共有 numPages 個 TranslationEntry。但是我們也要初始化 TranslationEntry 的值，也就是 physical page number、valid bit、use bit、dirty bit、read only bit 要是什麼值。所以我們可以用在 kernel.h 中定義的 freeFrameList 來知道哪些 frame 是 free 的，可以分配給 program 的。如果 freeFrameList 裡 free frame 的數量小於 numPages，則會呼叫 MemoryLimitException。用一個 for loop 來一個個初始化每一個 TranslationEntry 的 physicalPage，把 freeFrameList->Front()得知在 linked list 中第一個 free frame number (free page number)是多少，再呼叫 freeFrameList->RemoveFront() 把第一個 page number 移除(代表這個 page number 已經不是 free 了)。將 valid bit 設為 true，use bit 設為 false，dirty bit 設為 false，readOnly bit 設為 false。readOnly bit 會隨著 program 中不一樣的地方做調整，像是 code segment readOnly bit 要設為 true (code segment 不可以隨意更改)，所以在之後

設為 false (因為我們目前不知道 code segment 在 pageTable 的哪一個 TranslationEntry，所以只能在 load code segment 的其間做修改)。

Load code segment into physical memory and maintain pageTable

```
unsigned int *phyaddr;
unsigned int vpn;
int numOfPages = 0;
while(PageSize * numOfPages < noffH.code.size){
    DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    unsigned int currAddr = noff.code.virtualAddr + PageSize * numOfPages;
    vpn = currAddr / PageSize;
    pageTable[vpn].readOnly = TRUE;
    Translate(currAddr, phyaddr, FALSE);
    if (PageSize * (numOfPages+1) < noffH.code.size){
        executable->ReadAt(
        &(kernel->machine->mainMemory[*phyaddr]),
        PageSize, noffH.code.inFileAddr + PageSize * numOfPages);
    }
    else{
        executable->ReadAt(
        &(kernel->machine->mainMemory[*phyaddr]),
        noffH.code.size-PageSize*numOfPages, noffH.code.inFileAddr + PageSize * numOfPages);
    }
    numOfPages ++;
}
```

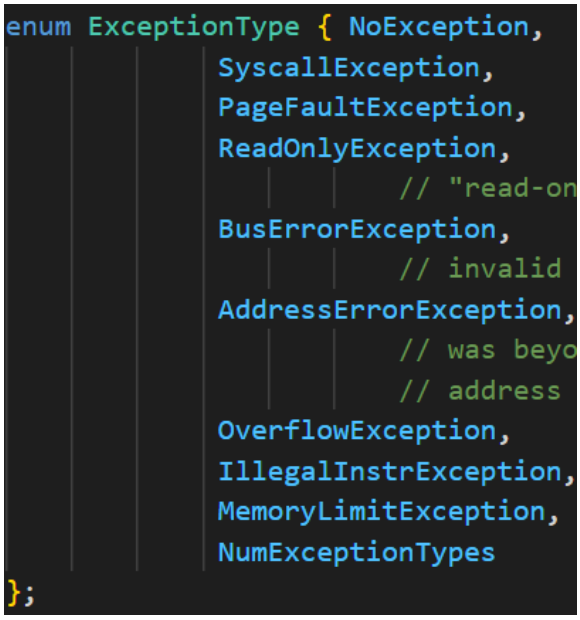
接下來要把 code segment load 到 physical memory 中，首先要做的是將 virtual address 轉換成 physical address，並且每一次的 load 是以 page size 為單位。因此我用一個 while loop 將 code segment 以 page size 為單位 load 到 physical memory 中。Virtual address 轉換到 physical address 是用 Translate() 這個 function。轉換好後呼叫 executable->ReadAt() 把一個 page size 的 code segment load 到 physical memory。需要注意的是，最後一次 load 到 physical memory 的 code segment 可能不足一個 page size，也就是所謂的 internal fragmentation。所以最後一次 load 的大小可能小於一個 page size。其他像是 data segment、read-only data segment 也是跟 code segment 一樣的實作方法。

AddrSpace::~AddrSpace()

```
AddrSpace::~AddrSpace()
{
    for(int i = 0 ; i < numPages ; i++){
        int pfn = pageTable[i].physicalPage;
        unsigned int phyaddr = pfn * PageSize;
        bzero(&kernel->machine->mainMemory[phyaddr], PageSize);
        kernel->freeFrameList->Append(pfn);
    }
    delete pageTable;
}
```

在 class AddrSpace 的 destructor 中因為這個 process 已經執行完了，不需要占用 physical memory 了，所以要把 pageTable 的所有 physical frame number 重新放到 freeFrameList，並把 process 占用的 physical memory 設為 0。最後把 pageTable delete。

machine.h



新增 MemoryLimitException 來做為當沒有 free frame 的時候要乎要 exception.