CS342301 2024 MP2

Multi-Programming

I. Team member & contribution:

I. 江佩霖 111062118

i. Trace cod
ii. Implement function
iii. 測試, Debug

II. 陳庭竣 111020025

i. Trace code
ii. Implement function
iii. 測試, Debug

II. Trace Code

I. Threads/thread.cc

i. Thread::Sleep()

```
236  void Thread::Sleep(bool finishing) {
237      Thread *nextThread;
238
239      ASSERT(this == kernel->currentThread);
240      ASSERT(kernel->interrupt->getLevel() == IntOff);
241
242      DEBUG(dbgThread, "Sleeping thread: " << name);
243      DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
244
245      status = BLOCKED;
246      // cout << "debug Thread::Sleep " << name << "wait for Idle\n";
247      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
248          kernel->interrupt->Idle();  // no one to run, wait for an interrupt
249      }
250      // returns when it's time for us to run
251      kernel->scheduler->Run(nextThread, finishing);
252  }
```

將當前 thread 的 state 設定為 bolck，並從等待序列中 schedule 一個新的 thread 執行(若沒有則將 CPU mode 設置為 idle)。

ii. Thread::StackAllocate()

```
301  void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
302      stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
303
304  #ifdef PARISC
305      // HP stack works from low addresses to high addresses
306      // everyone else works the other way: from high addresses to low addresses
307      stackTop = stack + 16;  // HP requires 64-byte frame marker
308      stack[StackSize - 1] = STACK_FENCEPOST;
309  #endif
310
311  #ifdef SPARC
312      stackTop = stack + StackSize - 96;  // SPARC stack must contains at
313                                          // least 1 activation record
314                                          // to start with.
315      *stack = STACK_FENCEPOST;
316  #endif
317
318  #ifdef PowerPC                          // RS6000
319      stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
320      *stack = STACK_FENCEPOST;
321  #endif
322
323  #ifdef DECMIPS
324      stackTop = stack + StackSize - 4;  // -4 to be on the safe side!
325      *stack = STACK_FENCEPOST;
326  #endif
```

```
301  void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
328  #ifdef ALPHA
329      stackTop = stack + StackSize - 8;  // -8 to be on the safe side!
330      *stack = STACK_FENCEPOST;
331  #endif
332
333  #ifdef x86
334      // the x86 passes the return address on the stack.  In order for SWITCH()
335      // to go to ThreadRoot when we switch to this thread, the return addres
336      // used in SWITCH() must be the starting address of ThreadRoot.
337      stackTop = stack + StackSize - 4;  // -4 to be on the safe side!
338      *(--stackTop) = (int)ThreadRoot;
339      *stack = STACK_FENCEPOST;
340  #endif
341
342  #ifdef PARISC
343      machineState[PCState] = PLabelToAddr(ThreadRoot);
344      machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
345      machineState[InitialPCState] = PLabelToAddr(func);
346      machineState[InitialArgState] = arg;
347      machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
348  #else
349      machineState[PCState] = (void *)ThreadRoot;
350      machineState[StartupPCState] = (void *)ThreadBegin;
351      machineState[InitialPCState] = (void *)func;
352      machineState[InitialArgState] = (void *)arg;
353      machineState[WhenDonePCState] = (void *)ThreadFinish;
354  #endif
355  }
```

Allocate 一個 8192*sizeof(int)大小的空間位址作為 thread 的 stack 空間。
並根據不同系統架構設計了不同的堆疊方式。
最後會使用 machineState 這個數據結構保存最終的 CPU reg 等等 thread 參數。

iii. Thread::Finish()

```
167  void Thread::Finish() {
168      (void)kernel->interrupt->SetLevel(IntOff);
169      ASSERT(this == kernel->currentThread);
170
171      DEBUG(dbgThread, "Finishing thread: " << name);
172      if (kernel->execExit && this->getIsExec()) {
173          kernel->execRunningNum--;
174          if (kernel->execRunningNum == 0) {
175              kernel->interrupt->Halt();
176          }
177      }
178      Sleep(TRUE);  // invokes SWITCH
179      // not reached
180  }
```

對應 Thread::Begin()，將 interrupt 設定程 IntOff 狀態後將判斷工作完成的 thread 刪除，若全部的 thread 都完成後則呼叫 Halt()結束 Process。
後面的 Sleep()則會將 thread 工作完成的 bool 值傳入 Run()，並完成 context switch。

iv. Thread::Fork()

```
91  void Thread::Fork(VoidFunctionPtr func, void *arg) {
92      Interrupt *interrupt = kernel->interrupt;
93      Scheduler *scheduler = kernel->scheduler;
94      IntStatus oldLevel;
95
96      DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " " << arg);
97      StackAllocate(func, arg);
98
99      oldLevel = interrupt->SetLevel(IntOff);
100     scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts
101                                   // are disabled!
102     (void)interrupt->SetLevel(oldLevel);
103 }
```

呼叫 StackAllocate()並傳入 PCState()、ArgState()參數(分別是 func, arg，用來設置新 allocate 的 thread 參數)，接著保存 interrupt 狀態並將其設置為 IntOff 以將新 fork 的 thread 加入 ReadyToRun()中，最後恢復 interrupt 狀態。

I. userprog/addrspace.cc

i. AddrSpace::AddrSpace()

```
//----------------------------------------------------------------------
// 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];
    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, ensuring that
    // all physical memory is cleared before being allocated to user programs
    bzero(kernel->machine->mainMemory, MemorySize);
}
```

為 user program 創造一個 address space，來執行它，並建立 pagetable，在 virtual pages 及 physical pages 間 mapping。

建立一個大小為 NumPhysPages(Physical page 數量)的 TranslationEntry array，形成 pagetable。

更新每個 physical page，將 virtual page 的編號及 physical page 的編號設為相同 (Pagetable 的第幾個 entry)、valid bit 設為 1、dirty bit 設為 0、readonly 設為 false。

在 user program 使用前，將 physical page 中所有 byte 設為 0，讓 user program 在乾淨的環境執行。

ii. AddrSpace::Execute()

```
//----------------------------------------------------------------------
// 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;

    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"
}
```

執行 load 到 address space 中的 user program。它設置了 program 運行所需的狀態，接著將控制權轉移到該程式。

為現在 thread 分配 address space，接著初始化 register 的值，並 load page table register，執行 Run()，將控制權交給 user，處理 memory 中 user program 指令。

iii. AddrSpace::Load()

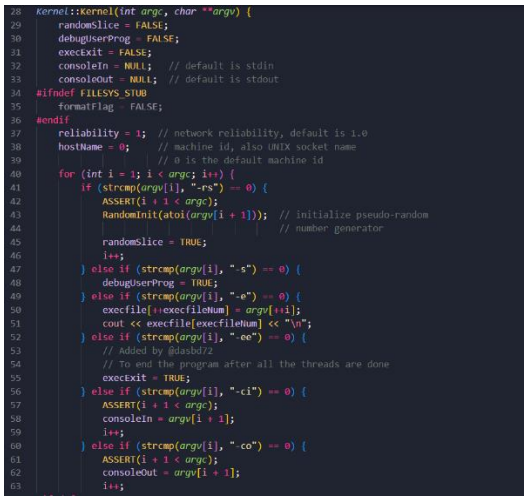
```
bool
AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

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

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    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
    numPages = divRoundUp(size, PageSize);
    // calaulaten the executable need how many pages
    size = numPages * PageSize;
    // size is adjusted to the total memory allocation in bytes

    // TODO

    if(numPages > NumPhysPages){
        kernel->interrupt->setStatus(SystemMode);
        ExceptionHandler(MemoryLimitException);
        kernel->interrupt->setStatus(UserMode);
    }


    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);
```

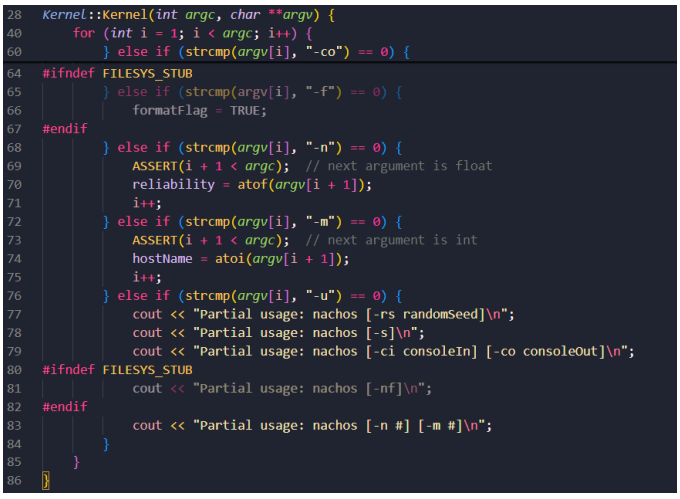
此函示主要目的是將 user program 從 file 載入 memory 中，先讀取檔案，再計算所需 page 數量，調整分配的 memory 大小，若所需 page 大於 physical page 能提供的數量，則產生 MemoryLimitException，處理 exception，最終，把資料存入 memory 中。

II. threads/kernel.cc

i. Kernel::Kernel()



```
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
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
            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) {
            execfile[++execfileNum] = argv[++i];
            cout << execfile[execfileNum] << "\n";
        } else if (strcmp(argv[i], "-ee") == 0) {
            // Added by @dasbd72
            // To end the program after all the threads are done
            execExit = TRUE;
        } else if (strcmp(argv[i], "-ci") == 0) {
            ASSERT(i + 1 < argc);
            consoleIn = argv[i + 1];
            i++;
        } else if (strcmp(argv[i], "-co") == 0) {
            ASSERT(i + 1 < argc);
            consoleOut = argv[i + 1];
            i++;
```



```
Kernel::Kernel(int argc, char **argv) {
    for (int i = 1; i < argc; i++) {
        } else if (strcmp(argv[i], "-co") == 0) {
#ifndef FILESYS_STUB
        } else if (strcmp(argv[i], "-f") == 0) {
            formatFlag = TRUE;
#endif
        } else if (strcmp(argv[i], "-n") == 0) {
            ASSERT(i + 1 < argc);  // next argument is float
            reliability = atof(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-m") == 0) {
            ASSERT(i + 1 < argc);  // next argument is int
            hostName = atoi(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            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";
        }
    }
}
```

函式主要功能是解讀 command line argument，並設定相對應的 flag。(ASSERT(i + 1 < argc)是為了確保 argv[i+1]存在，防止讀取到 command line argument 之外的位置。)

-rs：以 argv[i+1]的值為 seed，初始化隨機數生成器，並啟用 randomslice。

-s：啟用 debugUserProg，進入 debug 模式。

-e：將 argv 內下個參數指定為欲執行的檔案名稱，並存入 execfile 陣列中。

-ee：啟用 execExit，當所有 thread 執行完畢後，program 應該退出。

-ci：將 consoleIn 設為 argv[i+1]的內容。例如 program -ci input.txt，會將 consoleIn 設為 "input.txt"，就可以通過 consoleIn 到指定檔案讀取資料。

-co：將 consoleOut 設為 argv[i+1]的內容。

-f：啟用 formatFlag，格式化文件系統。

-n：將 argv[i+1]的值作為可靠度。(argv[i+1]轉換成 float 的值應屆於 0~1)

-m：將 argv[i+1]的值作為 hostName(機器 ID)。

-u：顯示基本 command line argument 的說明。

ii. Kernel::ExecAll()

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

執行所有 execute file，完成後，呼叫 Finish()。

iii. Kernel::Exec()

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

    return threadNum - 1;
```

建一個新的 Thread 物件存入 t[threadNum]，呼叫 setIsExec()，使這個 thread 處在執行狀態，並分配給它新的地址物件，再呼叫 fork()，讓 threadNum 加一，並回傳原本的 threadNum。

iv. Kernel::ForkExecute()

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

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

查看 Thread 的 space 中，執行檔是否存在。如果 load 執行檔成功，呼叫 Execute()。

III. threads/scheduler.cc

i. Scheduler::ReadyToRun()

```
54
55  void Scheduler::ReadyToRun(Thread *thread) {
56      ASSERT(kernel->interrupt->getLevel() == IntOff);
57      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
58      // cout << "Putting thread on ready list: " << thread->getName() << endl ;
59      thread->setStatus(READY);
60      readyList->Append(thread);
61  }
```

先確認 interrupt 的狀態是否是 IntOff(檢查 Fork()是否正確將 interrupt 關閉)，將 thread 狀態設置為 Ready 後放入 list 中等待排程。

ii. Scheduler::Run()

```
99  void Scheduler::Run(Thread *nextThread, bool finishing) {
100     Thread *oldThread = kernel->currentThread;
101     ASSERT(kernel->interrupt->getLevel() == IntOff);
102
103     if (finishing) {  // mark that we need to delete current thread
104         ASSERT(toBeDestroyed == NULL);
105         toBeDestroyed = oldThread;
106     }
107     if (oldThread->space != NULL) {  // if this thread is a user program,
108         oldThread->SaveUserState();  // save the user's CPU registers
109         oldThread->space->SaveState();
110     }
111
112     oldThread->CheckOverflow();  // check if the old thread
113                                  // had an undetected stack overflow
114
115     kernel->currentThread = nextThread;  // switch to the next thread
116     nextThread->setStatus(RUNNING);      // nextThread is now running
117     DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
118     // This is a machine-dependent assembly language routine defined
119     // in switch.s.  You may have to think
120     // a bit to figure out what happens after this, both from the point
121     // of view of the thread and from the perspective of the "outside world".
122     SWITCH(oldThread, nextThread);
123     // we're back, running oldThread
124     // interrupts are off when we return from switch!
125     ASSERT(kernel->interrupt->getLevel() == IntOff);
126     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
127     CheckToBeDestroyed();  // check if thread we were running
128                            // before this one has finished
129                            // and needs to be cleaned up
130     if (oldThread->space != NULL) {     // if there is an address space
131         oldThread->RestoreUserState();  // to restore, do it.
132         oldThread->space->RestoreState();
133     }
134 }
```

檢查 interrupt 狀態(IntOff)；調用 SWITCH 完成 context switch(硬體)。

若 oldthread 已完成(finishing == True)，SWITCH 返回後 CheckToBeDestroyed()會檢查並刪除 oldthread；否則將呼叫 RestoreUserState()、RestoreState()，回復之前儲存的狀態(SaveUserState()、SaveState())

A. The Question must Cover

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

```
void Thread::StackAllocate(VoidFunctionPtr func, void *arg)
```

在 kernel:ExecAll()中，會呼叫 kernel:Exec()，Exec()中會設置新的 thread 及 address space，再呼叫 Thread:Fork()及 Thread::StackAllocate()，建立新的 thread，最後呼叫 ForkExecute()執行 thread。

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

```
AddrSpace::Load // function
bzero(kernel->machine->mainMemory + frame_number * PageSize,
PageSize); // initialize the memory content
executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),
LoadSize, noffH.code.inFileAddr + cur); // loading user binary code
```

如上圖所示。

iii. How does Nachos create and manage the page table?

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

原：將 memory 空間全部直接分配 (static allocate)，physical page 和 virtual page 的數量及對應的 index 相同，有直接對應的關係。

修改後：在 load 函式中依據需要的 page 數量分配相對應的 page (dynamic allocate)，找到空的 frame 就可以放 page，兩者 index 沒有直接對應關係。

iv. How does Nachos translate addresses?

```
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int
isReadWrite)
```

進行 address translation 時，需確認 page number 小於 page 數量、frame number 小於 physical pages 數量及檢測 readOnly bit 等，若出現錯誤，回傳相對應的 exception，接著利用 frame number、page size 及 page offset 求得 physical address，若沒有錯誤，則回傳。

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

```
void AddrSpace::InitRegisters()
```

將所有 register 內容清空，PC 設為 0，假設從 virtual address = 0 開始跑，next PC 設為 4，再將 stack 起始位置設為 address space 最後位置減 16，減 16 目的為預留緩衝空間。

vi. Which object in Nachos acts the role of process control block

```
class Thread {
    private:
        int *stack;
        ThreadStatus status;
        char *name;
        int ID;
        bool isExec;
        void StackAllocate(VoidFunctionPtr func, void *arg);
        int userRegisters[NumTotalRegs];
};
```

Thread 這個 object 用 class 儲存了 process 的資訊(ID、name、state 等等)。

```
Thread::Thread(char *threadName, int threadID)
```

在 thread()中設定。

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

```
void Thread::Yield() //function 1
void Thread::Fork(VoidFunctionPtr func, void *arg) //function 2
void Semaphore::V() //function 3
```

1. Yield() : switch thread，呼叫 ReadyToRun。
2. Fork() : create child thread，呼叫 ReadyToRun。
3. V() : awake new thread，呼叫 ReadyToRun。

III. Implement page table in NachOS

I. Working item: Modify its memory management code to make NachOS support

i. kernel.cc

```
Frame_Table_status = new int[NumPhysPages];         // int array which size is the amount of physycal pages
for(int i = 0; i < NumPhysPages; i++) Frame_Table_status[i] = 0;     // initialize frame table
```

在 Initialize 中，初始化 page table。

```
int Kernel::Find_Frame(){
    for(int i = 0; i < NumPhysPages; i++){
        if(Frame_Table_status[i] == 0){       // frame[i] is empty
            Frame_Table_status[i] = 1;
            return i;
        }
    }
    return -1;
}
```

建立 Find_Frame 函數，尋找還有空位的 frame，回傳 frame number，並將該 frame 設為 1 代表已有 page 佔用，若沒有所剩的 page，則回傳-1。

ii. kernel.h

```
int Find_Frame();    // To find available frame
```

```
int *Frame_Table_status;  // a pointer which point to an array store the frame is empty or not
```

宣告 Find_Frame 及 Frame_Table_status。

iii. machine.h

```
enum ExceptionType { NoException,
                     SyscallException,
                     PageFaultException,
                     ReadOnlyException,
                     BusErrorException,
                     AddressErrorException,
                     OverflowException,
                     IllegalInstrException,
                     MemoryLimitException,
                     NumExceptionTypes
};
```

在 ExceptionType 中加入 MemoryLimitException。

iv. addrspace.cc

```
pageTable = new TranslationEntry[numPages];  // create page table which size is the amount of numPages for this thread

for (int i = 0; i < numPages; i++) {
    int frame_number = kernel->Find_Frame();

    if(frame_number == -1) {
        ExceptionHandler(MemoryLimitException);
        return false;
    }

    pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = frame_number;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    bzero(kernel->machine->mainMemory + frame_number * PageSize, PageSize);
}
```

在 Load 函式中，為 thread 建立一個符合它所需 page 數量的 page table，為每個 page 尋找對應的空的 frame，並設定 valid bit、dirty bit、readonly 等，若已經沒有空的 frame，即為 MemoryLimitException，呼叫 ExceptionHandler 處理。

```
unsigned int physAddr;
ExceptionType status;

if (noffH.code.size > 0) {

    DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);

    unsigned int page_amount = divRoundUp(noffH.code.size, PageSize);   // the amount of pages in code

    int cur = 0;
    for(int i = 0; i < page_amount; i++){

        status = Translate(noffH.code.virtualAddr + cur, &physAddr, 0);

        if (status != NoException)
        {
            kernel->interrupt->setStatus(SystemMode);
            ExceptionHandler(status);
            kernel->interrupt->setStatus(UserMode);
        }

        unsigned int LoadSize;
        if(i == page_amount - 1) LoadSize = noffH.code.size - (page_amount - 1) * PageSize;
        else LoadSize = PageSize;

        executable->ReadAt(&(kernel->machine->mainMemory[physAddr]), LoadSize, noffH.code.inFileAddr + cur);
        cur += PageSize;
    }
}
```

計算 code segment 使用到的 page 數量，將每個 page 的 virtual address 送入 Translation 函式，檢驗是否有 Exception，若發生，進入 kernel mode 處理 Exception，設定 LoadSize 為每個 page 的 size，最後一個 page 可能發生 internal fragmentation，需另外計算，運用 LoadSize、noffH.code.inFileAddr 及 cur 將 code segment 分段載入到 main memory。

```
if (noffH.initData.size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);

    unsigned int page_amount = divRoundUp(noffH.initData.size, PageSize);   // the amount of pages in initData

    int cur = 0;
    for(int i = 0; i < page_amount; i++){

        status = Translate(noffH.initData.virtualAddr + cur, &physAddr, 0);

        if (status != NoException)
        {
            kernel->interrupt->setStatus(SystemMode);
            ExceptionHandler(status);
            kernel->interrupt->setStatus(UserMode);
        }

        unsigned int LoadSize;
        if(i == page_amount - 1) LoadSize = noffH.initData.size - (page_amount - 1) * PageSize;
        else LoadSize = PageSize;

        executable->ReadAt(&(kernel->machine->mainMemory[physAddr]), LoadSize, noffH.initData.inFileAddr + cur);
        cur += PageSize;
    }
}
```

將 init data 分段載入到 main memory，實作方式與上述相同。

```
#ifdef RDATA

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

        unsigned int page_amount = divRoundUp(noffH.readonlyData.size, PageSize);   // the amount of pages in readonlyData
        unsigned int page_num;                                                      // the virtual page number
        for(int i = 0; i < page_amount; i++){
            page_num = (noffH.readonlyData.virtualAddr / PageSize) + i;             // calculate virtual page number
            pageTable[page_num].readOnly = TRUE;
        }

        int cur = 0;
        for(int i = 0; i < page_amount; i++){

            status = Translate(noffH.readonlyData.virtualAddr + cur, &physAddr, 0);
            //if(status != NoException) RaiseException(status);
            if (status != NoException)
            {
                kernel->interrupt->setStatus(SystemMode);
                ExceptionHandler(status);
                kernel->interrupt->setStatus(UserMode);
            }


            unsigned int LoadSize;
            if(i == page_amount - 1) LoadSize = noffH.readonlyData.size - (page_amount - 1) * PageSize;
            else LoadSize = PageSize;

            executable->ReadAt(&(kernel->machine->mainMemory[physAddr]), LoadSize, noffH.readonlyData.inFileAddr + cur);
            cur += PageSize;
        }
    }

#endif
```

將 read only data 分段載入到 main memory，實作方式與上述相同，但在這個部分需將每個 page 的 read only bit 設為 1。

```
if(numPages > NumPhysPages){
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(MemoryLimitException);
    kernel->interrupt->setStatus(UserMode);
}
```

若所需的 page 數量大於實際有的 page 數量，產生 MemoryLimitException。

```
AddrSpace::~AddrSpace() {
    for (int i = 0; i < numPages; i++) {
        if (pageTable[i].valid == 1) {
            int phys_Page_num = pageTable[i].physicalPage;
            kernel->Frame_Table_status[phys_Page_num] = false;
            pageTable[i].valid = false;
        }
    }
    delete pageTable;
}
```

在~AddrSpace 函式中，將 frame 空位清出、page table 的 valid bit 設為 0、page table 刪除，釋放 page table。

II. Verification

i. Correct results with multiprogramming

```
[os24team26@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
```

ii. Correctly handle the exception about insufficient memory

```
[os24team26@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test3
consoleIO_test1
consoleIO_test3
9Unexpected user mode exception 8
Assertion failed: line 297 file ../userprog/exception.cc
Aborted
```