CS342301: Operating System

MP2: Multi-Programming

- Group6 -

I. Team members and contributions

Trace code

Working items	Members
(1.) threads/kernel.cc - **`Kernel::ExecAll()`** ~ (9.) threads/thread.cc - **`Thread::Finish()`**	詹詠絜
(10.) threads/kernel.cc - **`Thread::Sleep()`** ~ (16.) machine/mipssim.cc - **`Machine::Run()`**	許安嫩
Answering questions: Q1, Q3, Q7	詹詠絜
Answering questions: Q2, Q4, Q5, Q6	許安嫩

Implementation

Working items	Members: Coding & Report
Implement page table in NachOS – (1.) ~ (4.)	詹詠絜
Implement page table in NachOS – (5.)	許安嫩
Implement page table in NachOS – (6.) Validation	詹詠絜、許安嫩

II. Trace code

(1.) /threads/main.cc -- main(int argc, char **argv)

啟動 NachOS 後，進入 main()的前半部，會先初始化相關 flags、變數和資料結構，接著根據 user 輸入的 command line arguments 完成初步的設置

```
226     debug = new Debug(debugArg);
227
228     DEBUG(dbgThread, "Entering main");
229
230     kernel = new Kernel(argc, argv);
231
232     kernel->Initialize();
```

L230 創建一個 kernel 物件，傳入 command line arguments，並將該物件的記憶體位置存入全域變數 kernel 中

L232 呼叫 Initialize()，函數定義在**/threads/kernel.cc**，負責設置 NachOS 核心系統的各個重要模組，包括 thread management、interrupt handling、filesystem 以及模擬硬體功能，為接下來 OS 的運行做好準備

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

L100 創建一個 Thread 物件存入變數 currentThread，將其命名為"main"，根據目前的 thread 數量設置它的 ThreadID

L101 將其狀態設置為 RUNNING，當它不再使用 CPU 時，系統能夠記錄其狀態在 currentThread

(2.) /threads/kernel.cc -- Kernel::Kernel(int argc, char **argv)

一開始創建 Kernel 物件時，首先會初始化多個變數為預設值，再解析 command line arguments 執行對應的初始化操作，其中，與 thread management 較相關的部分，說明如下:

```
28  Kernel::Kernel(int argc, char **argv) {
29      randomSlice = FALSE;
30      debugUserProg = FALSE;
31      execExit = FALSE;
32      consoleIn = NULL;   // default is stdin
33      consoleOut = NULL;  // default is stdout
34  #ifndef FILESYS_STUB
35      formatFlag = FALSE;
36  #endif
37      reliability = 1;  // network reliability, default is 1.0
38      hostName = 0;     // machine id, also UNIX socket name
39                        // 0 is the default machine id
40      for (int i = 1; i < argc; i++) {
41          if (strcmp(argv[i], "-rs") == 0) {
42              ASSERT(i + 1 < argc);
43              RandomInit(atoi(argv[i + 1]));  // initialize pseudo-random
44                                              // number generator
45              randomSlice = TRUE;
46              i++;
47          } else if (strcmp(argv[i], "-s") == 0) {
48              debugUserProg = TRUE;
49          } else if (strcmp(argv[i], "-e") == 0) {
50              execfile[++execfileNum] = argv[++i];
51              cout << execfile[execfileNum] << "\n";
```

`L41–L46` 當-rs 參數被指定，系統會初始化一個隨機數生成器 RandomInit()，它會分配隨機時間長度給不同的 threads，讓他們輪流使用 CPU，藉此模擬不同的 workload 和 schedule 策略

`L49-L51` 當-e 參數被指定，系統會將 user 指定的執行檔名稱存入 execfile

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

kernel 物件完成初始化，return 回 main()並接受一些 test routines (thread、console、network)，最後呼叫 ExecAll()

```
255  void Kernel::ExecAll() {
256      for (int i = 1; i <= execfileNum; i++) {
257          int a = Exec(execfile[i]);
258      }
259      currentThread->Finish();
260      // Kernel::Exec();
261  }
```

`L256-L258` 遍歷 execfile 陣列，對每個檔案呼叫 Exec()

L259 當 Finish()被呼叫時，會告知系統 main thread 完成了所有初始化和程序執行的準備工作，但它不會立即刪除或釋放 currentThread 的資源，因為它仍然在運行並且使用它自己的 stack。因此，需要告知 scheduler: 等你開始處理其他 threads 的時候，再讓 destructor 把我清理掉哦!

(4.) /threads/kernel.cc -- Kernel::Exec(char *name)

```
263   int Kernel::Exec(char *name) {
264       t[threadNum] = new Thread(name, threadNum);
265       t[threadNum]->setIsExec();
266       t[threadNum]->space = new AddrSpace();
267       t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
268       threadNum++;
269
270       return threadNum - 1;
```

L264 為傳入的執行檔名稱與 ThreadID 創建一個新的 Thread 物件，並存入 thread array t，以 ThreadID 作為 index

L265 呼叫 setIsExec()時，Thread 物件的 isExec 變數會被標記為 True，表示這是個 user thread

L266 幫這個 Thread 物件 allocate 一塊記憶體空間

L267 使用 Fork()同步執行兩個 threads (procedures)，需要將 ForkExecute 函數作為參數傳入，表示這個 Thread 物件要同步執行此函數

L268 更新 threads 數量，之後創建其他 threads 維持所有 ThreadID 的唯一性

L270 Return ThreadID

(5.) /userprog/addrspace.cc -- AddrSpace::AddrSpace()

```
65  AddrSpace::AddrSpace() {
66      pageTable = new TranslationEntry[NumPhysPages];
67      for (int i = 0; i < NumPhysPages; i++) {
68          pageTable[i].virtualPage = i;  // for now, virt page # = phys page #
69          pageTable[i].physicalPage = i;
70          pageTable[i].valid = TRUE;
71          pageTable[i].use = FALSE;
72          pageTable[i].dirty = FALSE;
73          pageTable[i].readOnly = FALSE;
74      }
75
76      // zero out the entire address space
77      bzero(kernel->machine->mainMemory, MemorySize);
78  }
```

L66 創建共 128(NumPhysPages = 128)個 page table entries，並將這個 TranslationEntry 陣列存入變數 pageTable

L67-L74 初始化每個 entry:

這裡的 translation 方式採用 1:1，也就是一個 virtual page 對應到一個 physical page。

- valid: 該 page 是否已被初始化，讀取內容有效，default 是 True
- readOnly: 是否允許 User 修改該 page 內容，default 是 False
- use: 該 page 每次被 referenced/modified，硬體就會將它設為 True，default 是 False
- dirty: 該 page 每次被 modified，硬體就會將它設為 True，default 是 False

L77 將 physical memory 清空，之後執行 threads 時會陸續存入 user program、code、data

(6.) /threads/threads.cc -- Thread::Fork(VoidFunctionPtr func, void *arg)

```
 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
 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  }
```

L97 呼叫 StackAllocate()分配新的 stack 空間，並將它初始化，之後 switch 到這個 thread 時，就會使用這塊空間

L99 disable interrupt，防止當下 thread 的執行不受其他中斷干擾

L100 呼叫 ReadyToRun()將此 thread 加入 ready queue 中

L102 恢復之前的中斷狀態:enable interrupt

(7.) /threads/threads.cc -- Thread::StackAllocate(VoidFunctionPtr func, void *arg)

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

L302 分配一塊 stack 空間，使用 AllocBoundedArray()得到 StackSize(8192 words)的

陣列，stack 指向此 stack 空間的底部

```
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
```

L333 根據 x86 CPU 架構的 stack 管理機制:

L337 界定 stackTop 的位置

L338 將 ThreadRoot 放在 stack 的第一個 frame 裡，當 SWITCH()結束後會 return 回此函數，接著執行這個 thread (ThreadRoot enables interrupts)

L339 STACK_FENCEPOST 放在 stack 底部，用來偵測是否發生 stack overflow

```
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
```

L349-L353 machineState 相當於 TCB 內的 register set，當 thread 不再使用 CPU 時，便會將 CPU registers 裡的資訊(由上圖不同的函數/參數取得)放在對應的 TCB register 中，用來儲存 thread 的當前狀態

(8.) /threads/scheduler.cc -- Scheduler::ReadyToRun(Thread *thread)

```
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   }
```

L56 檢查是否 disable interrupt，避免不同程序競爭進入 ready queue 的情況

L59 將 thread 的狀態設為 READY，表示它已經準備好在 CPU 上運行，但還未真正獲得 CPU 的控制權

L60 將這個 thread 放入 ready queue 中等待被執行

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

待 StackAllocate() -> ReadyToRun() -> Fork() -> Exec()都完成後，會 return 回 ExecAll()接續呼叫 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   }
```

L168 disable interrupt

L169 檢查當前的 thread 是否為正在執行中的，因為只有它能結束自己(呼叫 Finish())

L172-L176 如果確認了這個 user thread 已完成所有任務被允許退出，就將執行中的 threads 數量減一，如果所有 user threads 都執行完畢，即中止整個系統

L178 將 thread 標記為 finishing(傳入 Sleep()的參數)，並呼叫 Sleep()讓它使用 SWITCH 切換到其他待執行的 thread

(10.)/threads/thread.cc -- Thread::Sleep(bool finishing)

```
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   }
```

L239 檢查當前的 thread 是否為正在執行中的

L240 檢查是否 disable interrupt

L245 將當前 thread 的狀態設為 BLOCKED，相當於進入 waiting queue，等待被 re-scheduled

L247-L249 呼叫 Scheduler::FindNextToRun()，從 ready queue 取得 next thread。若目前 ready queue 是空的，則會呼叫 Interrupt::Idle()等待偵測到 next thread

L251 取得 next thread 後呼叫 Scheduler::Run()，並將 next thread 和標記當前 thread 應該要被刪除的 flag(finishing = TRUE)傳入

(11.)/threads/sheduler.cc -- Scheduler::Run(Thread *nextThread, bool finishing)

```
 99  void Scheduler::Run(Thread *nextThread, bool finishing) {
100      Thread *oldThread = kernel->currentThread;
101
102      ASSERT(kernel->interrupt->getLevel() == IntOff);
103
104      if (finishing) {  // mark that we need to delete current thread
105          ASSERT(toBeDestroyed == NULL);
106          toBeDestroyed = oldThread;
107      }
108
109      if (oldThread->space != NULL) {  // if this thread is a user program,
110          oldThread->SaveUserState();  // save the user's CPU registers
111          oldThread->space->SaveState();
112      }
113
114      oldThread->CheckOverflow();  // check if the old thread
115                                   // had an undetected stack overflow
116
117      kernel->currentThread = nextThread;  // switch to the next thread
118      nextThread->setStatus(RUNNING);      // nextThread is now running
```

L102 檢查是否 disable interrupt

L104-L118 將 toBeDestroyed 指向 oldThread，儲存目前 user program 的狀態，再將 next thread 設為當前的 thread，並將狀態設為 RUNNING

```
127      SWITCH(oldThread, nextThread);
128
129      // we're back, running oldThread
130
131      // interrupts are off when we return from switch!
132      ASSERT(kernel->interrupt->getLevel() == IntOff);
133
134      DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
135
136      CheckToBeDestroyed();  // check if thread we were running
137                             // before this one has finished
138                             // and needs to be cleaned up
139
140      if (oldThread->space != NULL) {     // if there is an address space
141          oldThread->RestoreUserState();  // to restore, do it.
142          oldThread->space->RestoreState();
143      }
144  }
```

L127 呼叫 SWITCH()，其為定義在/threads/switch.S 中的 stub，目的是為了進行 context switch

L132 檢查 SWITCH()回來後 interrupt 是否為 disabled

`L136` 若有需要則將排在 old thread 之前已執行完的 threads 刪除

`L140-L142` 如果 old thread 還要繼續執行(`Finishing = FALSE`)，restore(寫回 machine)原本 user program 的 CPU 狀態

(12.)/threads/switch.S -- `SWITCH`

將 CPU registers 存入 old thread 的 TCB，再從 next thread 的 TCB 拿出其保存的 registers，放到 CPU registers 內，然後再跳到 `ThreadRoot`

(13.)/threads/switch.S – `ThreadRoot`

- 呼叫 `ThreadBegin()->Thread::Begin()`刪除 old thread，並將 interrupt enable
- (目前狀況為)呼叫 `ForkExecute()`，執行當前新的 thread
- 呼叫 `ThreadFinish()->Thread::Finish()`結束當前的 thread

(14.)/threads/kernel.cc – `ForkExecute(Thread *t)`

接續**(4.)**，在 `Exec()`內部，曾將 `ForkExecute` 函數作為參數傳入 `Fork()`，它會在分配 stack space 時存入 `machineState[InitialPCState]`，當呼叫 `SWITCH()`進入存放 `ThreadRoot` 的 stack 後，再透過 `InitialPC` 找到這個 `Thread` 將要執行的程序，即 `ForkExecute` 函數:

```
252  void ForkExecute(Thread *t) {
253      if (!t->space->Load(t->getName())) {
254          return;  // executable not found
255      }
256
257      t->space->Execute(t->getName());
258  }
```

`L253-L255` 將執行檔 load 到 memory 內，若無法成功則直接 return

`L257` 呼叫 `AddrSpace::Execute()`來執行 program

(15.)/userprog/addrspace.cc – `AddrSpace::Load(char *filename)`

```
108  bool AddrSpace::Load(char *fileName) {
109      OpenFile *executable = kernel->fileSystem->Open(fileName);
110      NoffHeader noffH;
111      unsigned int size;
112
113      if (executable == NULL) {
114          cerr << "Unable to open file " << fileName << "\n";
115          return FALSE;
116      }
117
118      executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
119      if ((noffH.noffMagic != NOFFMAGIC) &&
120          (WordToHost(noffH.noffMagic) == NOFFMAGIC))
121          SwapHeader(&noffH);
122      ASSERT(noffH.noffMagic == NOFFMAGIC);
```

L109-L116 嘗試開啟執行檔，若開啟失敗則回傳 FALSE

L118 從執行檔中讀取其 header 的資訊到 NoffHeader 結構中

L119-L122 確保檔案為 NachOS object file 格式

```
124  #ifdef RDATA
125      // how big is address space?
126      size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
127             noffH.uninitData.size + UserStackSize;
128      // we need to increase the size
129      // to leave room for the stack
130  #else
131      // how big is address space?
132      size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize;
133
134  #endif
135      numPages = divRoundUp(size, PageSize);
136      size = numPages * PageSize;
137
138      ASSERT(numPages <= NumPhysPages);  // check we're not trying
139                                         // to run anything too big --
140                                         // at least until we have
141                                         // virtual memory
```

L124-L134 計算 user program 需要的 address space

L135-L138 計算所需的 numPages，並檢查其是否超過 NumPhysPages，確保我們沒有載入過大的程式

```
147     if (noffH.code.size > 0) {
148         DEBUG(dbgAddr, "Initializing code segment.");
149         DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
150         executable->ReadAt(
151             &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
152             noffH.code.size, noffH.code.inFileAddr);
153     }
154     if (noffH.initData.size > 0) {
155         DEBUG(dbgAddr, "Initializing data segment.");
156         DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
157         executable->ReadAt(
158             &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
159             noffH.initData.size, noffH.initData.inFileAddr);
160     }
161
162 #ifdef RDATA
163     if (noffH.readonlyData.size > 0) {
164         DEBUG(dbgAddr, "Initializing read only data segment.");
165         DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
166         executable->ReadAt(
167             &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
168             noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
169     }
170 #endif
171
172     delete executable;  // close file
173     return TRUE;        // success
```

`L147-L170` 從執行檔依序將 code segment、initialized data segment、read-only data 載入 main memory 對應的 physical address，由於這裡使用 uni-programming，所以 physical address 預設和 virtual address 相同

`L172-L173` 關閉執行檔並回傳 TRUE，表示成功載入程式

(16.)/userprog/addrspace.cc – `AddrSpace::Execute(char *filename)`

```
185 void AddrSpace::Execute(char *fileName) {
186     kernel->currentThread->space = this;
187
188     this->InitRegisters();  // set the initial register values
189     this->RestoreState();   // load page table register
190
191     kernel->machine->Run();  // jump to the user progam
192
193     ASSERTNOTREACHED();  // machine->Run never returns;
194                          // the address space exits
195                          // by doing the syscall "exit"
196 }
```

`L186` 將當前的 address space 設定給 `currentThread`

`L188` 初始化 CPU registers、PC、next PC 及 stack register

`L189` 將當前的 `pageTable` 設定給 machine

`L191` 呼叫 `Machine::Run()`開始運行 user program

III. Trace code: Answering questions

1. How does NachOS allocate the memory space for a new thread (process)?
   NachOS 會透過 `Fork()`函數內部的 `StackAllocate()`為新創建的 thread 分配並初始化記憶體空間，包括 stack 空間的分配和切換狀態的設置

2. How does NachOS initialize the memory content of a thread (process), including loading the user binary code in the memory?
   NachOS 創建新的 thread 後會幫他 new 一個 `AddrSpace`，為他創建及初始化 page table，並清空 main memory。之後在 `ForkExecute()->AddrSpace::Load()`時，會讀取其要跑的執行檔的 header，把關於 code、initData、readonlyData 的資訊(size、segment 在 file 中的起始位置、virtual address)載入 main memory 裡

3. How does NachOS create and manage the page table?
   NachOS 在每個 thread 創建完成後，會透過一個 `AddrSpace` 物件來管理該 thread 的 Page Table，它是由 `TranslationEntry` 陣列組成，該陣列中的每一個元素(entry)都對應於一組 virtual page 與 physical page，以及額外的 bits 用來標記每個 page 的狀態: `valid`、`use`、`dirty`、`readOnly`；管理方面，當 CPU 試圖讀取某個 virtual page 時，就會透過查表找到對應的 physical page 和解讀上述 bits 對 physical page 操作

4. How does NachOS translate addresses?
   /machine/translate.cc 中的 `Machine::Translate()`會根據 virtual address 計算出 virtual page number 及 offset，然後用 page table 或 TLB 得到 physical page，將其乘上 page size、加上 offset 後，便得到最終的 physical address。另外，過程中也會處理各種異常狀況，並設置 entry 的 use 和 dirty bit

5. How does NachOS initialize the machine status (registers, etc.) before running a thread (process)?
   NachOS 在 `ForkExecute()->AddrSpace::Execute()`時，會呼叫 `InitRegisters()`來初始化 machine 的 registers。此函數會將所有 registers 設為 0，再分別設置 `PCReg` 為 0、`NextPCReg` 為 4，並讓 `StackReg` 指到當前 address space 的頂部。另外再接續呼叫 `RestoreState()`來將 machine 的 `pageTable` 和 `pageTableSize` 設置成將要執行的 thread 的 `pageTable` 和 `numPages`，接著呼叫 `Run()`就能開始跑這個 address space 的指令了!

6. Which object in NachOS acts the role of process control block?

   Thread 物件(如下圖)。它記錄了 status、threadName、threadID、machineState、界定 stack 範圍的指標等資訊，與 process control block 的用途相似

```
37  Thread::Thread(char *threadName, int threadID) {
38      ID = threadID;
39      name = threadName;
40      isExec = false;
41      stackTop = NULL;
42      stack = NULL;
43      status = JUST_CREATED;
44      for (int i = 0; i < MachineStateSize; i++) {
45          machineState[i] = NULL;  // not strictly necessary, since
46                                   // new thread ignores contents
47                                   // of machine registers
48      }
49      space = NULL;
50  }
```

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

   在 Fork()函數中，當分配好 stack 空間且初始化後，會通知 scheduler 將準備就緒的 thread 放入 ready queue 中，並把該 thread 的狀態設為 READY，等待使用 CPU 的資源

IV. Implement page table in NachOS

(1.) /threads/kernel.h

```
76      int NumAvailablePhysPages;
77      bool isUsedPhysPages[NumPhysPages]; // 0: available, 1: used
```

L76 在 kernel class 中新增變數 NumAvailablePhysPages，用來記錄當前可用的 physical page 數量

L77 由於只需要記錄每個 frame 是否已被使用，且主記憶體的 frames 總數不多(128 frames)，我們選擇較簡易的資料結構——陣列，將其命名為 isUsedPhysPages，以 physical page number 作為 index，False 表示為 empty frame，True 則代表已被使用

(2.) /threads/kernel.cc -- Kernel::Kernel(int argc, char **argv)

```
42      NumAvailablePhysPages = NumPhysPages;
43      for(int i = 0 ; i < NumPhysPages ; i++) isUsedPhysPages[i] = 0;
```

L42-L43 一開始創建 kernel 物件時，physical memory space 還是空的，所以將在**(1.)**定義的 NumAvailablePhysPages 設為主記憶體的 frames 總數，並將所有的 isUsedPhysPages 元素值設為 False，代表每個 frame 都未被使用

(3.) /userprog/addrspace.cc

```
25  #include "kernel.h"
```

L25 因為 AddrSpace 的 destructor 內部需要使用定義在 kernel.h 的變數，所以這裡要 include header file

```
67  Space::AddrSpace() {
68  pageTable = new TranslationEntry[NumPhysPages];
69  for (int i = 0; i < NumPhysPages; i++) {
70      pageTable[i].virtualPage = i;
71      pageTable[i].physicalPage = -1; // Have Not load the program.
72      pageTable[i].valid = FALSE;
73      pageTable[i].use = FALSE;
74      pageTable[i].dirty = FALSE;
75      pageTable[i].readOnly = FALSE;
76  }
```

L71 AddrSpace()函數本身就是用來創建 pageTable 的，不過原本模板的設計是讓 virtual page number 與對應到的 physical page number 相同，所以如果要實作 pure demand paging，需要在一開始創建 pageTable 時，先將每個 entry 對應到的 physical page number 設為-1，表示為空的 entry，且將 valid bit 設為 FALSE

```
87  AddrSpace::~AddrSpace() {
88      // Traverse each entry and clear the loaded physical pages.
89      for(int i = 0 ; i < NumPhysPages ; i++){
90          if(pageTable[i].physicalPage != -1){
91              kernel->isUsedPhysPages[pageTable[i].physicalPage] = 0;
92              kernel->NumAvailablePhysPages++;
93          }
94      }
95      delete pageTable;
96  }
```

L89-L94 當 thread 執行結束後，會使用 AddrSpace 的 destructor 刪除 pageTable，除此之外，還需要把紀錄已使用的 physical page number 的陣列元素值恢復成 False，並還原可使用的 frames 總數

(4.) /machine/machine.h

```
56                  // MP2: Insufficient memory for a thread
57                  MemoryLimitException,
58
59                  NumExceptionTypes
```

L56 新增一個 exception 類型 MemoryLimitException，如果分配給 thread 的記憶體空間不足，就會使用這個 enum 通知 RaiseException()報錯

(5.)/userprog/addrspace.cc-- AddrSpace::Load(char fileName)

```
143     //------------------MP2-------------------//
144     if(numPages > kernel->NumAvailablePhysPages){
145         kernel->machine->RaiseException(MemoryLimitException, 0);
146         return FALSE;
147     }
148     for(int i=0, j=0; i<numPages; i++){
149         while(j<NumPhysPages && kernel->isUsedPhysPages[j]==1) j++;
150         pageTable[i].physicalPage = j;
151         if(pageTable[i].valid == FALSE) pageTable[i].valid = TRUE;
152         kernel->isUsedPhysPages[j] = 1;
153         kernel->NumAvailablePhysPages--;
154     }
```

L144-L147 檢查# of pages 是否會超過可以用的 physical pages，若超過便以 MemoryLimitException 為 Exception type，呼叫 Machine::RaiseException()，然後再回傳 FALSE，表示沒有 load 成功。(RaiseException()是定義在 Machine 的 class 裡的，因此在/machine/machine.h 裡有新增 friend class AddrSpace，如下圖所示)

```
182     //------------------MP2-------------------//
183     friend class AddrSpace; // calls RaiseException()
```

L148-L151 對於每個 entry，找尋一個可以使用的 physical page，並將其 valid bit 設為 TRUE

L148-L151 更新 isUsedPhysPages，並將可用的 physical pages 數減 1

```
160     ExceptionType exception;
161     if (noffH.code.size > 0) {
162         DEBUG(dbgAddr, "Initializing code segment.");
163         DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
164         unsigned int physicalAddr;
165         if((exception = Translate(noffH.code.virtualAddr, &physicalAddr, 1)) != NoException){
166             kernel->machine->RaiseException(exception, 0);
167             return FALSE;
168         }
169         executable->ReadAt(
170             &(kernel->machine->mainMemory[physicalAddr]),
171             noffH.code.size, noffH.code.inFileAddr);
172     }
```

(以 code segment 的部分為例，`initData` 及 `readonlyData` 的部分也是照同樣方式改寫)

`L164-L171` 呼叫 `AddrSpace::Translate()`透過 page table 來將 virtual address 轉成 physical address，並將結果存入 `physicalAddr` 中。`Translate()`內部除了轉換 address 以外，也會同時檢查是否有任何 exception 的狀況發生，若有的話則會在 return 回來時呼叫 `Machine::RaiseException()`，並回傳 `FALSE` 至上一層，表示沒有 load 成功。如果以上都沒發生錯誤，便會接著把 program load 到 `mainMemory` 裡

(6.) Output Verification

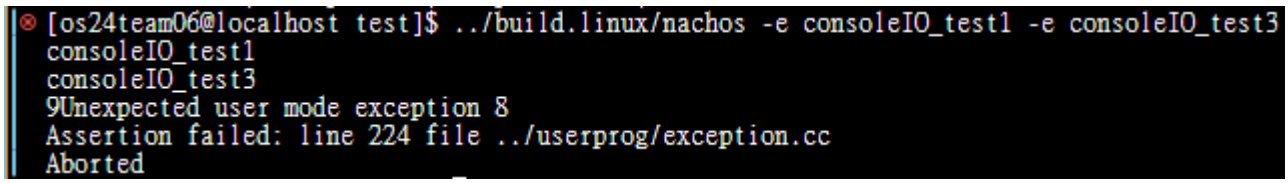
```
[os24team06@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

Correct results with multiprogramming

Self-created test case - /test/consoleIO_test3.c

```
1   #include "syscall.h"
2
3   int main() {
4       char *LittleStar = "Twinkle, twinkle, little star, How I wonder what you are!
    (char [19153])"Twinkle, twinkle, little star, How I wonder what you are! Up above the
    high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you
    Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high,
```

L4 宣告一個長度為 19153 bytes 的 string，讓這個程式的 code size 大於 available page size



Correctly handle the exception about insufficient memory