MP2_report_56

組員：

- 資工大三 **108020022** 周昱宏
- 資工大三 **108071003** 李彥璋

分工表：

工作項目	負責人
Trace code與報告撰寫	周昱宏、李彥璋
報告整合與回答問題	周昱宏
implementation	李彥璋
測試、**Debug**	周昱宏、李彥璋

II. 1.

threads/kernel.cc – Kernel::Kernel()

kernel.cc 的主要工作為：初始化 Nachos 作業系統的 kernel。

Kernel::Kernel()的註解
「Interpret command line arguments in order to determine flags for the initialization」，
說明了它的功能（Kernel::Kernel()是被 threads/main.cc – int main()呼叫的，特別是針對有 set kernel parameters 的 command line），
而回溯到 threads/main.cc – int main()，
我們可以知道 Kernel::Kernel(int argc, char **argv)傳入參數的意思：

> argc is the number of command line arguments (including the name of the command) ex: "nachos -d +" -> argc = 3.
> argv is an array of strings, one for each command line argument ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}.

以下是 threads/main.cc – int main()大致的過程：

1. Create kernel object.
   kernel = new Kernel(argc, argv);
   在這個步驟很重要的是，**argv 讀取到「-e」**：

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

   execfile[] 儲存每個 execfile 的檔名；
   execfile 同時也會儲存現在共有幾個 execfile。

2. Initialize the system.
   kernel->Initialize();

3. The kernel is ready to do something.
   Run some tests, if requested.

4. Run an initial user program if requested to do so.
   kernel->ExecAll();

因此，我們接著 trace 「Kernel:: ExecAll()」。

threads/kernel.cc – Kernel:: ExecAll()

Kernel::ExecAll()主要的工作為：
[263-265]呼叫 Kernel::Exec()，執行 execfile[]中所有檔案（user program）。

```
261  void Kernel::ExecAll()
262  {
263      for (int i=1;i<=execfileNum;i++) {
264          int a = Exec(execfile[i]);
265      }
266      currentThread->Finish();
267      //Kernel::Exec();
268  }
```

[266] The thread is done executing.

threads/kernel.cc – Kernel::Exec()

被 Kernel::ExecAll()呼叫，實際執行 execfile[]中所有的檔案。

```
271  int Kernel::Exec(char* name)
272  {
273      t[threadNum] = new Thread(name, threadNum);
274      t[threadNum]->space = new AddrSpace();
275      t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
276      threadNum++;
277
278      return threadNum-1;
```

[273] Initialize a thread control block.

[274] Create an address space.

[275] turning a thread into one that the CPU can schedule and execute.

[276] increase the number of thread.

問題回答：Which object in Nachos acts the role of process control block?

創造的Thread object其實就扮演了PCB的角色，因其attribute包含該thread狀態、stack指標、address space指標…等PCB會記載的資訊。

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

底下將介紹Exec()所調用的函數 - AddrSpace::AddrSpace()與Thread::Fork()。

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

賦予創造好的 Thread 定址空間，以下的寫法只針對對 uniprogramming 沒辦法支援 multiprogramming，因為現在的 program memory 與 physical memory 是一對一映射，因此若同時有兩個 process 在 program memory 處理，可能會對應到相同的 physical memory。

bzero()函式會清空整個主記憶體(目前沒辦法支援 multiprogramming)。

問題回答：How Nachos creates and manages the page table?

問題回答：How Nachos allocates the memory space for new thread(process)?

參照上述。

```
65  AddrSpace::AddrSpace()
66  {
67      pageTable = new TranslationEntry[NumPhysPages];
68      for (int i = 0; i < NumPhysPages; i++)
69      {
70          pageTable[i].virtualPage = i; // for now, virt page # = phys page #
71          pageTable[i].physicalPage = i;
72          pageTable[i].valid = TRUE;
73          pageTable[i].use = FALSE;
74          pageTable[i].dirty = FALSE;
75          pageTable[i].readOnly = FALSE;
76      }
77
78      // zero out the entire address space
79      bzero(kernel->machine->mainMemory, MemorySize);
80  }
```

threads/thread.cc – Thread::Fork()

分配此 Thread 一個 Stack 並初始化(調用 StackAllocate())，並先將 interrupt disable 掉，將此 Thread 放入 ReadyQueue(ReadyToRun(this))。

```
91  void
92  Thread::Fork(VoidFunctionPtr func, void *arg)
93  {
94      Interrupt *interrupt = kernel->interrupt;
95      Scheduler *scheduler = kernel->scheduler;
96      IntStatus oldLevel;
97
98      DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
99      StackAllocate(func, arg);
100
101     oldLevel = interrupt->SetLevel(IntOff);
102     scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
103                                     // are disabled!
104     (void) interrupt->SetLevel(oldLevel);
105 }
```

threads/thread.cc – Thread::StackAllocate()

根據不同型號的機器初始化不同的 Stack 執行空間，machineState(除了 stack top 的資訊)會將此函式傳遞進來的 func(caller)、arg(callee)與 ThreadBegin、ThreadEnd 分別放置入相對應的 machineState 記錄 stack 狀態。

問題回答：How Nachos initializes the machine status (registers, etc) before running a thread(process)?

參照上述。

```
306  void
307  Thread::StackAllocate (VoidFunctionPtr func, void *arg)
308  {
309      stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
356  #else
357      machineState[PCState] = (void*)ThreadRoot;
358      machineState[StartupPCState] = (void*)ThreadBegin;
359      machineState[InitialPCState] = (void*)func;
360      machineState[InitialArgState] = (void*)arg;
361      machineState[WhenDonePCState] = (void*)ThreadFinish;
362  #endif
363  }
```

threads/kernel.cc – Kernel::ForkExecute()

Kernel::ForkExecute()是新的 thread 運行的函數。

```
251  void ForkExecute(Thread *t)
252  {
253      if ( !t->space->Load(t->getName()) ) {
254          return;             // executable not found
255      }
256
257      t->space->Execute(t->getName());
258
259  }
```

[253] Load a user program into memory from a file.

[257] After user program loaded into the address space, running it by using the current thread.

底下將介紹 ForkExecute()所調用的函數 - AddrSpace::Load()與 AddrSpace::Execute ()。

userprog/addrspace.cc – AddrSpace::Load()

AddrSpace::Load()：目的將 user program 檔案裡的 object code 與 data 轉入至 memory 裡，並根據內容大小初始化 Page Table 的 size。

問題回答：How Nachos initializes the memory content of a thread(process), including loading the user binary code in the memory?

透過ReadAt()。

問題回答：How Nachos translates address?

透過Translate()函式。

```
102 bool AddrSpace::Load(char *fileName)
103 {
104     OpenFile *executable = kernel->fileSystem->Open(fileName);
105     NoffHeader noffH;
106     unsigned int size;
107 
108     if (executable == NULL)
109     {
110         cerr << "Unable to open file " << fileName << "\n";
111         return FALSE;
112     }
113 
114     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
115     if ((noffH.noffMagic != NOFFMAGIC) &&
116         (WordToHost(noffH.noffMagic) == NOFFMAGIC))
117         SwapHeader(&noffH);
118     ASSERT(noffH.noffMagic == NOFFMAGIC);
```

```
130 #endif
131     numPages = divRoundUp(size, PageSize);
132     size = numPages * PageSize;
133
134     ASSERT(numPages <= NumPhysPages); // check we're not trying
135                                       // to run anything too big --
136                                       // at least until we have
137                                       // virtual memory
138
139     DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " <<
140
141     // then, copy in the code and data segments into memory
142     // Note: this code assumes that virtual address = physical address
143     if (noffH.code.size > 0)
144     {
145         DEBUG(dbgAddr, "Initializing code segment.");
146         DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size
147         executable->ReadAt(
148             &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
149             noffH.code.size, noffH.code.inFileAddr);
150     }
151     if (noffH.initData.size > 0)
152     {
153         DEBUG(dbgAddr, "Initializing data segment.");
154         DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initD
155         executable->ReadAt(
156             &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
157             noffH.initData.size, noffH.initData.inFileAddr);

171     delete executable; // close file
172     return TRUE;       // success
173 }
```

userprog/addrspace.cc – AddrSpace::Execute()

將 currentThread 變成此 Thread，並利用 InitRegisters()初始化設定 Program Counter，利用 RestoreState()更換至此 Thread 的 Page Table，最後呼叫 Run()逐行 Instruction 開始執行。

```
184 void AddrSpace::Execute(char *fileName)
185   {
186
187       kernel->currentThread->space = this;
188
189       this->InitRegisters(); // set the initial register values
190       this->RestoreState();  // load page table register
191
192       kernel->machine->Run(); // jump to the user progam
193
194       ASSERTNOTREACHED(); // machine->Run never returns;
195                           // the address space exits
196                           // by doing the syscall "exit"
197   }
```

threads/scheduler.cc – Scheduler::ReadyToRun()

Scheduler::ReadyToRun()是被 Thread::Fork()所調用。

```
56  void
57  Scheduler::ReadyToRun (Thread *thread)
58  {
59      ASSERT(kernel->interrupt->getLevel() == IntOff);
60      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
61      //cout << "Putting thread on ready list: " << thread->getName() << endl ;
62      thread->setStatus(READY);
63      readyList->Append(thread);
64  }
```

[59] ReadyToRun assumes that interrupts are disabled!

[62] Mark a thread as ready.

[63] Put thread on the ready list, for later scheduling onto the CPU.

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

ReadyToRun()。

threads/scheduler.cc – Scheduler::Run()

將當前運行強制切換至 nextThread，若 Old Thread 執行完畢則將他 Destroy 掉(finishing == true)，如果是 user program 將他的 CPU register 暫存起來，並利用 CheckOverflow()檢查此 Thread 的 Stack 是否溢位。最後利用 SWITCH()強制換成 nextThread 即完成轉換。

當 Kernel::ExecAll() 迴圈執行完所有的 execfile 時，接著會執行 Thread::Finish()。

threads/thread.cc – Thread::Finish()

釋放此已執行完畢的 Thread 空間，中斷 Interrupt，並呼叫 Thread::Sleep()。

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

threads/thread.cc – Thread::Sleep()

Thread::Sleep()：將當前 Thread Block 住，並利用迴圈持續呼叫 FindNextToRun() 檢查 ReadyQueue 是否還有 Thread，若為空則讓 Kernel 進入 Idle 模式，若否則利用 Run 進入下一個 Thread。

```
238   void
239   Thread::Sleep (bool finishing)
240   {
241       Thread *nextThread;
242
243       ASSERT(this == kernel->currentThread);
244       ASSERT(kernel->interrupt->getLevel() == IntOff);
245
246       DEBUG(dbgThread, "Sleeping thread: " << name);
247       DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
248
249       status = BLOCKED;
250       //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
251       while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
252           kernel->interrupt->Idle();  // no one to run, wait for an interrupt
253       }
254       // returns when it's time for us to run
255       kernel->scheduler->Run(nextThread, finishing);
256   }
```

II. 2.

根據
Hint: The following files “may” be modified…
■ userprog/addrspace.*
■ threads/kernel.*
我們修改了 addrspace.h、addrspace.cc、kernel.h、kernel.cc 四個檔案。
*另外還有 machine.h 新增了一個 MemoryLimitException。

kernel.h

依照 Requirement—You must put the data structure recording used physical memory in kernel.h / kernel.cc，
我們在 kernel.h 加了 usedPhyMem 陣列。

```
int usedPhyMem[NumPhysPages]; //有改 有改 有改
```

kernel.cc

在 kernel.cc—Kernel::Kernel()裡，我們初始化 usedPhyMem 陣列。

```
for(int i=0 ; i<NumPhysPages ; i++) usedPhyMem[i] = 0; //有改 有改 有改
```

= 0 代表該 physical memory 還沒被用到。

在 kernel.cc—Kernel::Exec()裡，Create an address space 時，將 usedPhyMem 陣列傳入 AddrSpace()。

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

Addrspace.h

在 addrspace.h 更改 AddrSpace()的定義，傳入 int 陣列。

```
AddrSpace(int *);     // Create an address space. //有改 有改 有改
```

在 addrspace.h 新定義 2 個東西，

usedPhysMemCopy 陣列：

複製 usedPhyMem 陣列，因為 usedPhyMem 陣列是定義在 kernel.h 當中，但 addrspace.cc 的其他函式也需要用到它。

Idx：紀錄從哪裡開始 physical memory 還沒被用到。

```
int *usedPhysMemCopy; //有改 有改 有改
int Idx; //有改 有改 有改
```

Addrspace.cc

```
68  AddrSpace::AddrSpace(int *usedPhysMem) //有改 有改 有改
69  {
70      pageTable = new TranslationEntry[NumPhysPages];
71      for (int i = 0; i < NumPhysPages; i++) {
72      pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
73      pageTable[i].physicalPage = -1; //有改 有改 有改
74      pageTable[i].valid = TRUE;
75      pageTable[i].use = FALSE;
76      pageTable[i].dirty = FALSE;
77      pageTable[i].readOnly = FALSE;
78      }
79      usedPhysMemCopy = usedPhysMem; //有改 有改 有改
80
81      // zero out the entire address space
82      bzero(kernel->machine->mainMemory, MemorySize);
83  }
```

[74-77] 依照 Requirement—You must set up “valid, readOnly, use, and dirty” field for your page table。

[68] 增加傳遞 int 陣列的部分。
[73] 原本 = i，是因為 virt page # = phys page #。這裡先改成-1，待會 Load() 的時候會給予正確的值。
[79] 因為 addrspace.cc 的其他函式也要用到被傳入 Addrspace()的 int[]，所以這裡複製一份下來。

```
90  AddrSpace::~AddrSpace()
91  {
92      for(int i=Idx ; i<Idx+numPages ; i++) usedPhysMemCopy[i] = 0; //有改 有改 有改
93     delete pageTable;
94  }
```

[92-93] 依照 Requirement—When the thread is finished, make sure to release the address space and restore physical page status，將完成的 thread 所占用的 usedPhysMemCopy[]部分改為 0（= 0 代表該 physical memory 還沒被用到）。

以下在 Addrspace.cc—AddrSpace::Load()裡：

```
140     for(int i=0 ; i<NumPhysPages ; i++) { //有改 有改 有改
141         if(usedPhysMemCopy[i] == 0) {
142             Idx = i;
143             break;
144         }
145     }
146     for(int i=0 ; i<numPages ; i++) pageTable[i].physicalPage = Idx + i; //有改 有改 有改
147
148     for(int i=Idx ; i<Idx+numPages ; i++) usedPhysMemCopy[i] = 1; //有改 有改 有改
```

[140-145] 以 Idx 紀錄從哪裡開始 physical memory 還沒被用到。

[146] 將剛剛還沒設定好的 pageTable[i].physicalPage 完成，指到正確的 physical page 位置。

[148] 別忘了要把用掉的 physical memory 記錄下來（= 1 表示 physical memory 已經被用到掉了）。

接著，copy in the code and data segments into memory.

```
159     unsigned int physAddr; //有改 有改 有改
160     if( Translate(noffH.code.virtualAddr , &physAddr , 1) == MemoryLimitException){ //有改 有改 有改
161         kernel->interrupt->setStatus(SystemMode);
162         ExceptionHandler(MemoryLimitException);
163         kernel->interrupt->setStatus(UserMode);
164     }
165     if (noffH.code.size > 0) {
166         DEBUG(dbgAddr, "Initializing code segment.");
167     DEBUG(dbgAddr, physAddr << ", " << noffH.code.size); //有改 有改 有改
168 
169         executable->ReadAt(
170         &(kernel->machine->mainMemory[physAddr]), //有改 有改 有改
171             noffH.code.size, noffH.code.inFileAddr);
172     }
173     if (noffH.initData.size > 0) {
174         DEBUG(dbgAddr, "Initializing data segment.");
175     DEBUG(dbgAddr, physAddr << ", " << noffH.initData.size); //有改 有改 有改
176         executable->ReadAt(
177         &(kernel->machine->mainMemory[physAddr]), //有改 有改 有改
178             noffH.initData.size, noffH.initData.inFileAddr);
179     }
```

```
181 #ifdef RDATA
182     unsigned readPhysAddr; //有改 有改 有改
183     if( Translate(noffH.readonlyData.virtualAddr , &readPhysAddr , 0) == MemoryLimitException){ //有改 有改 有改
184         kernel->interrupt->setStatus(SystemMode);
185         ExceptionHandler(MemoryLimitException);
186         kernel->interrupt->setStatus(UserMode);
187     }
188 
189     if (noffH.readonlyData.size > 0) {
190         DEBUG(dbgAddr, "Initializing read only data segment.");
191     DEBUG(dbgAddr, readPhysAddr << ", " << noffH.readonlyData.size); //有改 有改 有改
192         executable->ReadAt(
193         &(kernel->machine->mainMemory[readPhysAddr]), //有改 有改 有改
194             noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
195     }
196 #endif
```

[160、183] Translate()的工作：Translate the virtual address in to a physical address.這裡也依照 Requirement—You must call ExceptionHandler to handle the exception when there is insufficient memory for a thread，如果 insufficient memory 的情況發生，Translate() 會 return MemoryLimitException，我們再呼叫 ExceptionHandler()處理。

在 Addrspace.cc—AddrSpace::Translate()當中，

```
332         if(*paddr >= MemorySize) { //測試中 測試中 測試中
333             return MemoryLimitException;
334         }
```

[332-334] 判斷是否 insufficient memory 發生。

執行結果：

```
[os21team56@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
```