Operating System
MP2 Report

組別：25

陳凱揚：Trace code、Implement、Report

簡佩如：Trace code、Implement、Report

1. Trace code

(1) /threads/main.cc → main()

在執行 nachos 後，首先會進到 main.cc 的 main()裡，處理 command line argument，處理完後會以這些 argument 建立一個 Kernel 物件且呼叫 Initialize()（圖 1-1），並將物件位址存於 global variable 的 kernel 中。

```
250     kernel = new Kernel(argc, argv);
251
252     kernel->Initialize();
```

▲ 圖 1-1

(2) threads/kernel.cc → Kernel(), Initialize()

在 Kernel 的 constructor 裡（圖 1-2），會將一些變數設置初值，並依據傳進的 argument 有對應的處理，其中特別注意第 55 行處理的"-e"，會將後面接著的執行檔名稱存入 execfile 中。

而 Initialize()主要是建立 Kernel 裡所需的其他物件（圖 1-3），與 constructor 分開的原因是因為他需要參考 kernel 先前設置好的資料。此外，在第 105 行建立了一個新的 thread 叫做 main，也就是第一個執行的 thread。

```
27 Kernel::Kernel(int argc, char **argv)
28 {
29     randomSlice = FALSE;
30     debugUserProg = FALSE;
31     consoleIn = NULL;          // default is stdin
32     consoleOut = NULL;         // default is stdout
33 #ifndef FILESYS_STUB
34     formatFlag = FALSE;
35 #endif
36     reliability = 1;            // network reliability, default is 1.0
37     hostName = 0;               // machine id, also UNIX socket name
38                                 // 0 is the default machine id
39     // ********** MP2 ********** //
40     numAvailablePhysPages = NumPhysPages;
41     for(int i = 0; i < NumPhysPages; i++){
42         usedPhysPages[i] = 0;
43     }
44     // ********** MP2 ********** //
45
46     for (int i = 1; i < argc; i++) {
47         if (strcmp(argv[i], "-rs") == 0) {
48             ASSERT(i + 1 < argc);
49             RandomInit(atoi(argv[i + 1]));// initialize pseudo-random
50             // number generator
51             randomSlice = TRUE;
52             i++;
53         } else if (strcmp(argv[i], "-s") == 0) {
54             debugUserProg = TRUE;
55         } else if (strcmp(argv[i], "-e") == 0) {
56             execfile[++execfileNum]= argv[++i];
57             cout << execfile[execfileNum] << "\n";
```

▲ 圖 1-2

```
 97 void
 98 Kernel::Initialize()
 99 {
100     // We didn't explicitly allocate the current thread we are running in.
101     // But if it ever tries to give up the CPU, we better have a Thread
102     // object to save its state.
103
104
105     currentThread = new Thread("main", threadNum++);
106     currentThread->setStatus(RUNNING);
107
108     stats = new Statistics();           // collect statistics
109     interrupt = new Interrupt;          // start up interrupt handling
110     scheduler = new Scheduler();        // initialize the ready queue
111     alarm = new Alarm(randomSlice);     // start up time slicing
112     machine = new Machine(debugUserProg);
113     synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
114     synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
115     synchDisk = new SynchDisk();    //
116 #ifdef FILESYS_STUB
117     fileSystem = new FileSystem();
118 #else
119     fileSystem = new FileSystem(formatFlag);
120 #endif // FILESYS_STUB
121     postOfficeIn = new PostOfficeInput(10);
122     postOfficeOut = new PostOfficeOutput(reliability);
123
124     interrupt->Enable();
125 }
126
```

▲ 圖 1-3

(3) /threads/kernel.cc → ExecAll(), Exec()

在建好 Kernel 後，會回到 main()繼續向下執行到第 288 行的 ExecAll()（圖 1-4），在 ExecAll()裡會將剛剛儲存在 execfile 裡的每個執行檔名稱作為參數，分別執行 Exec()，execfileNum 指的是有幾個待執行檔（圖 1-5）。而 Exec()裡的 t 是一個 thread 陣列，threadNum 會紀錄目前 thread 數量，接著為每個執行檔建立新的 Thread，這個 thread 會 new 一個新的 Addrspace，並以函數 ForkExecute 及這個 thread 本身作為參數呼叫 Fork()（圖 1-6）。

```
286     // finally, run an initial user program if requested to do so
287
288     kernel->ExecAll();
```

▲ 圖 1-4

```
268 void Kernel::ExecAll()
269 {
270     for (int i=1;i<=execfileNum;i++) {
271         int a = Exec(execfile[i]);
272     }
273     currentThread->Finish();
274     //Kernel::Exec();
275 }
```

▲ 圖 1-5

```
278 int Kernel::Exec(char* name)
279 {
280     t[threadNum] = new Thread(name, threadNum);
281     t[threadNum]->space = new AddrSpace();
282     t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
283     threadNum++;
284
285     return threadNum-1;
```

▲ 圖 1-6

(4) /userprog/addrspace.cc → AddrSpace()

建立一個 thread 的 space 也就是建出對應的 pageTable，功能為將程式的 virtual memory 轉換到 physical memory，所以在 AddrSpace 的 constructor 裡，會建立一個 TranslationEntry 陣列，並將每個 entry 中的每個變數都設為預設值（圖 1-7），在這裡的 physicalPage 預設與 virtualPage 相同，是 uni-programming 的做法。

```
68 AddrSpace::AddrSpace()
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 = i;
74         pageTable[i].valid = TRUE;
75         pageTable[i].use = FALSE;
76         pageTable[i].dirty = FALSE;
77         pageTable[i].readOnly = FALSE;
78     }
79
80     // zero out the entire address space
81     bzero(kernel->machine->mainMemory, MemorySize);
82 }
```

▲ 圖 1-7

(5) /threads/thread.cc → Fork()

在 Fork()裡會先使用 StackAllocate()分配一塊 stack 給現在這個 thread，接著儲存 interrupt 狀態至 oldLevel，再設為 IntOff（不能被打斷），並使用 scheduler 的 ReadyToRun()將這個 thread 加進 ready queue 裡，再將 interrupt 的狀態回復成 oldLevel（圖 1-8）。

```
 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 <<
 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 }
```

▲ 圖 1-8

(6) /threads/thread.cc → StackAllocate()

在 StackAllocate()裡，會先給這個 thread 的 stack 一塊新記憶體（圖 1-9），接著會依據不同的 CPU 架構來執行程式，在這邊是使用 x86 架構（圖 1-10），決定了 stackTop 的位置、並將 ThreadRoot 放在 stack 的第一個 frame 裡（會被 SWITCH()呼叫來開始執行這個 thread）、將 STACK_FENCEPOST 放在 stack 的最後一個 frame 裡（用來判斷是否發生 stack overflow），最後會將被 fork 的 func、被傳遞的 arg 及其他所需 function 放進 machineState 中對應的位置（圖 1-11）。

```
306 void
307 Thread::StackAllocate (VoidFunctionPtr func, void *arg)
308 {
309     stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

▲ 圖 1-9

```
341 #ifdef x86
342     // the x86 passes the return address on the stack.  In order for SWITCH()
343     // to go to ThreadRoot when we switch to this thread, the return addres
344     // used in SWITCH() must be the starting address of ThreadRoot.
345     stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
346     *(--stackTop) = (int) ThreadRoot;
347     *stack = STACK_FENCEPOST;
```

▲ 圖 1-10

```
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
```

▲ 圖 1-11

(7) /threads/scheduler.cc → ReadyToRun()

在 ReadyToRun()裡，首先會確認 interrupt 已經被關掉了，再來會將傳進的 thread 狀態設為 ready 後，放進 readyList 的最後面，也就是將這個 thread 放入 ready queue 裡等待執行（圖 1-12）。

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

▲ 圖 1-12

(8) /threads/thread.cc → Finish()

在 StackAllocate()、ReadyToRun()、Fork()、Exec()執行完後，會回到 ExecAll()裡，執行 Finish()，將一開始建立的第一個 thread(main)結束掉。而在 Finish()裡，首先會將 interrupt 設定為 IntOff（不能被打斷）、檢查是不是 currentThread 呼叫的，因為只有執行中的 thread 才能結束自己，接著會呼叫 Sleep()，並傳入 TRUE 作為參數，表示結束這個 thread（圖 1-13）。

```
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 }
```

▲ 圖 1-13

(9) /threads/thread.cc → Sleep()

這邊會再次檢查呼叫 Sleep 的是 currentThread，且 interrupt 為關閉狀態，接著將這個 thread 狀態設為 BLOCKED，並去 ready queue 找到下一個 thread，如果 ready queue 是空的，就會不斷呼叫 Idle()，表示 CPU 為閒置狀態，並跳到下一個 interrupt 的時間，直到找到 nextThread，就會執行 Run()（圖 1-14）。

```
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->
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 }
```

▲ 圖 1-14

(10) /threads/scheduler.cc → Run()

在 Run()裡，功能主要是將目前執行中的 thread 暫停、並儲存目前狀態，接著使用 SWITCH()將傳進來的 nextThread 轉變成 currentThread（圖 1-15），而之後回到原本的 thread 後會檢查是否需要結束這個 thread，是的話就刪除掉這個 thread，不是的話便回復狀態（圖 1-16）。

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

▲ 圖 1-15

```
135     // we're back, running oldThread
136
137     // interrupts are off when we return from switch!
138     ASSERT(kernel->interrupt->getLevel() == IntOff);
139
140     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
141
142     CheckToBeDestroyed();               // check if thread we were running
143                                         // before this one has finished
144                                         // and needs to be cleaned up
145
146     if (oldThread->space != NULL) {         // if there is an address space
147         oldThread->RestoreUserState();     // to restore, do it.
148         oldThread->space->RestoreState();
149     }
150 }
```

▲ 圖 1-16

(11) /threads/kernel.cc → ForkExecute()

在之前的 Exec()呼叫 Fork()時，會將 ForkExecute 當作參數傳入，並且在 StackAllocate()時放進 machineState[InitialPCState]，而在 SWITCH()作 thread 轉換後，第一個就會執行這個函式，功能是使用 Load()取得執行 thread 所需的記憶體，再呼叫 Execute()執行（圖 1-17）

```
258 void ForkExecute(Thread *t)
259 {
260         if ( !t->space->Load(t->getName()) ) {
261         return;             // executable not found
262     }
263
264     t->space->Execute(t->getName());
265
266 }
```

▲ 圖 1-17

(12) /userprog/addrspace.cc → Load()

在 Load()裡，首先會讀取執行檔最前面的 header，並存進 NoffHeader 資料結構中（圖 1-18），包含了 code、initData、uninitData 的 size 和 address 等資訊，再來就能使用這些資訊算出 thread 所需的 size 和 numPages（圖 1-19）。最後將 file 裡的資料寫進 mainMemory 中，這邊預設 virtualAddr 即為 physicalAddr（圖 1-20），為 uni-programming 的作法，load 成功後會回傳 TRUE 表示成功。

```
 8 #define NOFFMAGIC       0xbadfad        /* magic number denoting Nachos
 9                                         * object code file
10                                         */
11
12 typedef struct segment {
13   int virtualAddr;              /* location of segment in virt addr space */
14   int inFileAddr;               /* location of segment in this file */
15   int size;                     /* size of segment */
16 } Segment;
17
18 typedef struct noffHeader {
19    int noffMagic;               /* should be NOFFMAGIC */
20    Segment code;                /* executable code segment */
21    Segment initData;            /* initialized data segment */
22 #ifdef RDATA
23    Segment readonlyData;        /* read only data */
24 #endif
25    Segment uninitData;          /* uninitialized data segment --
26                                  * should be zero'ed before use
27                                  */
28 } NoffHeader;
```

▲ 圖 1-18

```
105 bool
106 AddrSpace::Load(char *fileName)
107 {
108     OpenFile *executable = kernel->fileSystem->Open(fileName);
109     NoffHeader noffH;
110     unsigned int size;
111
112     if (executable == NULL) {
113         cerr << "Unable to open file " << fileName << "\n";
114         return FALSE;
115     }
116
117     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
118     if ((noffH.noffMagic != NOFFMAGIC) &&
119                 (WordToHost(noffH.noffMagic) == NOFFMAGIC))
120         SwapHeader(&noffH);
121     ASSERT(noffH.noffMagic == NOFFMAGIC);
122
123 #ifdef RDATA
124 // how big is address space?
125     size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
126            noffH.uninitData.size + UserStackSize;
127                                                 // we need to increase the size
128                                                 // to leave room for the stack
129 #else
130 // how big is address space?
131     size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
132                         + UserStackSize;        // we need to increase the size
133                                                 // to leave room for the stack
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 --
```

▲ 圖 1-19

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

▲ 圖 1-20

(13) /userprog/addrspace.cc → Execute()

Execute()會將 space、machine 的 register、pageTable 調整為這個 thread 所需的狀態後，就會呼叫 machine 的 Run()，開始 fetch instruction（圖 1-21）。

```
185 void
186 AddrSpace::Execute(char* fileName)
187 {
188
189     kernel->currentThread->space = this;
190
191     this->InitRegisters();
192     this->RestoreState();
193
194     kernel->machine->Run();
```

▲ 圖 1-21

2. Implement

```
[os21team25@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
```

```
[os21team25@localhost test]$ ../build.linux/nachos -e test
test
Unexpected user mode exception 8
Assertion failed: line 201 file ../userprog/exception.cc
Aborted
```

(1) /threads/kernel.h → class Kernel{}

在 class Kernel 中，新增了兩個成員變數（圖 2-1），numAvailablePhysPages 記錄目前還有多少可以使用的 physical page 數量，而陣列 usedPhysPages 則是記錄每一個 physical page 有沒有被正在使用，1 代表使用中，0 代表未使用。

```
74      // ********** MP2 ********** //
75      int numAvailablePhysPages;
76      int usedPhysPages[NumPhysPages];
77      // ********** MP2 ********** //
```

▲ 圖 2-1

(2) /threads/kernel.cc → Kernel()

在 Kernel 的 constructor 中，numAvailablePhysPages 設為 NumPhysPages，而所有 usedPhysPages 都設為 0，因為所有 physical page 都未被使用（圖 2-2）。

```
39      // ********** MP2 ********** //
40      numAvailablePhysPages = NumPhysPages;
41      for(int i = 0; i < NumPhysPages; i++){
42          usedPhysPages[i] = 0;
43      }
44      // ********** MP2 ********** //
```

▲ 圖 2-2

(3) /machine/machine.h → enum ExcaptionType{}, class Machine{}

在 ExceptionType 裡會新增一個 MemoryLimitException，表示記憶體不足（圖 2-3），而在 load 執行檔進記憶體時，如果發生了記憶體不足或是 translate 失敗的情況，就會需要呼叫 RaiseException，因此會在 class Machine 裡新增 friend class AddrSpace（圖 2-4）。

```
43 enum ExceptionType { NoException,           // Everything ok!
44               SyscallException,      // A program executed a system call.
45               PageFaultException,    // No valid translation found
46               ReadOnlyException,     // Write attempted to page marked
47                         // "read-only"
48               BusErrorException,     // Translation resulted in an
49                         // invalid physical address
50               AddressErrorException, // Unaligned reference or one that
51                         // was beyond the end of the
52                         // address space
53               OverflowException,     // Integer overflow in add or sub.
54               IllegalInstrException, // Unimplemented or reserved instr.
55
56               // ********** MP2 ********** //
57               MemoryLimitException,
58               // ********** MP2 ********** //
59
60               NumExceptionTypes
61 };
```

▲ 圖 2-3

```
186     // ********** MP2 ********** //
187     friend class AddrSpace;   // calls RaiseException()
188     // ********** MP2 ********** //
```

▲ 圖 2-4

(4) /userprog/addrspace.cc → AddrSpace(), ~AddrSpace()

在 AddrSpace 的 constructor 裡，會將 physicalPage 都設為-1，表示目前還沒有 load 進 memory，所以沒有對應的 physicalPage（圖 2-5）。而在 destructor，則會將所有 pageTable 中對應到的 physicalPage 釋出，也就是將 usedPhysPages 設為 0，且增加 numAvailablePhysPages（圖 2-6）。

```
68 AddrSpace::AddrSpace()
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         // ********** MP2 ********** //
74         pageTable[i].physicalPage = -1;
75         // ********** MP2 ********** //
76         pageTable[i].valid = TRUE;
77         pageTable[i].use = FALSE;
78         pageTable[i].dirty = FALSE;
79         pageTable[i].readOnly = FALSE;
80     }
81
82     // zero out the entire address space
83     bzero(kernel->machine->mainMemory, MemorySize);
84 }
```

▲ 圖 2-5

```
 91 AddrSpace::~AddrSpace()
 92 {
 93     // ********** MP2 ********** //
 94     for(int i = 0; i < NumPhysPages; i++){
 95         if(pageTable[i].physicalPage != -1){
 96             kernel->usedPhysPages[pageTable[i].physicalPage] = 0;
 97             kernel->numAvailablePhysPages++;
 98         }
 99     }
100     // ********** MP2 ********** //
101     delete pageTable;
102 }
```

▲ 圖 2-6

(5) /userprog/addrspace.cc → Load()

首先會檢查所需的 numPages 是否超過可使用的 numAvailablePhysPages，如果是的話便會呼叫 RaiseException()，傳進 MemoryLimitException 作為參數，表示記憶體不足，並回傳 FALSE，表示 load 失敗；如果沒有超過，就會開始一個個尋找可用的 physPage，總共需要找到 numPage 個，每找到一個就需要將 usedPhysPages 設為 1，表示使用中，並減少 numAvailablePhysPages（圖 2-7）。

最後分別改寫 code（圖 2-8）、initData（圖 2-9）、readonlyData（圖 2-10）在寫入時使用的 virtualAddr，可以使用 Translate()將 virtualAddr 透過 pageTable 轉換成 physicalAddr，而如果 Translate()過程出現問題會回傳 exception 的 type，此時也需要 RaiseException()，並回傳 FALSE，表示 load 失敗。

```
145     numPages = divRoundUp(size, PageSize);
146     size = numPages * PageSize;
147 
148     // ********** MP2 ********** //
149     if(numPages > kernel->numAvailablePhysPages){
150         kernel->machine->RaiseException(MemoryLimitException, 0);
151         return FALSE;
152     }
153     int idx = 0;
154     for(int i = 0; i < numPages; i++){
155         while(idx < NumPhysPages && kernel->usedPhysPages[idx] == 1) idx++;
156         ASSERT(kernel->usedPhysPages[idx] == 0)
157         pageTable[i].physicalPage = idx;
158         kernel->usedPhysPages[idx] = 1;
159         kernel->numAvailablePhysPages--;
160     }
161     // ********** MP2 ********** //
162 
163     ASSERT(numPages <= NumPhysPages);       // check we're not trying
```

▲ 圖 2-7

```
172     if (noffH.code.size > 0) {
173         DEBUG(dbgAddr, "Initializing code segment.");
174     DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
175         // ********** MP2 ********** //
176         unsigned int physicalAddr;
177         ExceptionType exception = Translate(noffH.code.virtualAddr, &physicalAddr, 1);
178         if(exception != NoException){
179             kernel->machine->RaiseException(exception, 0);
180             return FALSE;
181         }
182         // ********** MP2 ********** //
183         executable->ReadAt(
184         &(kernel->machine->mainMemory[physicalAddr]),
185             noffH.code.size, noffH.code.inFileAddr);
186     }
```

▲ 圖 2-8

```
187     if (noffH.initData.size > 0) {
188         DEBUG(dbgAddr, "Initializing data segment.");
189     DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
190         // ********** MP2 ********** //
191         unsigned int physicalAddr;
192         ExceptionType exception = Translate(noffH.initData.virtualAddr, &physicalAddr, 1);
193         if(exception != NoException){
194             kernel->machine->RaiseException(exception, 0);
195             return FALSE;
196         }
197         // ********** MP2 ********** //
198         executable->ReadAt(
199         &(kernel->machine->mainMemory[physicalAddr]),
200             noffH.initData.size, noffH.initData.inFileAddr);
201     }
```

▲ 圖 2-9

```
203 #ifdef RDATA
204     if (noffH.readonlyData.size > 0) {
205         DEBUG(dbgAddr, "Initializing read only data segment.");
206     DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
207         // ********** MP2 ********** //
208         unsigned int physicalAddr;
209         ExceptionType exception = Translate(noffH.readonlyData.virtualAddr, &physicalAddr, 0);
210         if(exception != NoException){
211             kernel->machine->RaiseException(exception, 0);
212             return FALSE;
213         }
214         // ********** MP2 ********** //
215         executable->ReadAt(
216         &(kernel->machine->mainMemory[physicalAddr]),
217             noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
218     }
219 #endif
```

▲ 圖 2-10

3. Question

(1) How Nachos allocates the memory space for new thread(process)?

thread.cc 裡的 Fork()會使用 StackAllocate()分配並初始化記憶體空間給新的 thread。

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

Nachos 在建立新的 thread 的時候，這個 thread 會 new 一個 AddrSpace 來管理他的 pageTable，並設定初始值。在之後 Load()時，會去打開我們輸入的 binary 檔，且先讀取 binary 檔最前面固定格式的 header 進 NoffHeader 資料結構中，就可以知道這個 binary 檔裡面的 code、initData、readonlyData 的 size、virtualAddr、inFileAddr，最後就可以分別將這些資料從 file 裡寫進 mainMemory 中。

(3) How Nachos creates and manages the page table?

每個 thread 在建立時，都會 new 一個 AddrSpace 來管理 pageTable，pageTable 是一個 TranslationEntry 陣列，儲存了每個 virtualPage 對應到哪一個 physicalPage，其中也會儲存一些狀態，如 valid、dirty、readonly 等，Nachos 可以透過查 pageTable，將 loagical address 轉換成 physical address。

(4) How Nachos translates address?

translate.cc 裡的 Machine::Translate()會使用 pageTable 或 TLB 把 virtual address 轉換成 physical address。過程中如果有錯誤的話會回傳錯誤類型，確認沒有錯誤後，會將 entry 的 use 設為 TRUE，並將算出的 physical address 存進 physAddr 裡。

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

在 Execute()裡，最後執行 Machine::Run()前，會先將 currentThread 的 space 指向現在這個 Address（圖 3-1）；並且呼叫 InitRegisters()，將 program counter 和 stack 的位置寫進 machine 的 register，其他 register 則設為 0（圖 3-2）；再來呼叫 RestoreState()，將 machine 的 pageTable 和 pageTableSize 設為現在這個 AddrSpace 的 pageTable 和 numPages（圖 3-3）。

```
185 void
186 AddrSpace::Execute(char* fileName)
187 {
188
189     kernel->currentThread->space = this;
190
191     this->InitRegisters();
192     this->RestoreState();
193
194     kernel->machine->Run();
```

▲ 圖 3-1

```
261 void
262 AddrSpace::InitRegisters()
263 {
264     Machine *machine = kernel->machine;
265     int i;
266
267     for (i = 0; i < NumTotalRegs; i++)
268     machine->WriteRegister(i, 0);
269
270     // Initial program counter -- must be location of "Start", which
271     //  is assumed to be virtual address zero
272     machine->WriteRegister(PCReg, 0);
273
274     // Need to also tell MIPS where next instruction is, because
275     // of branch delay possibility
276     // Since instructions occupy four bytes each, the next instruction
277     // after start will be at virtual address four.
278     machine->WriteRegister(NextPCReg, 4);
279
280    // Set the stack register to the end of the address space, where we
281    // allocated the stack; but subtract off a bit, to make sure we don't
282    // accidentally reference off the end!
283     machine->WriteRegister(StackReg, numPages * PageSize - 16);
284     DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
285 }
```

▲ 圖 3-2

```
306 void AddrSpace::RestoreState()
307 {
308     kernel->machine->pageTable = pageTable;
309     kernel->machine->pageTableSize = numPages;
310 }
```

▲ 圖 3-3

(6) Which object in Nachos acts the role of process control block?

Thread，裡面包含了 machine state、stack、state、userRegisters 等資料，還有 Fork()、Sleep()等功能。

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

Scheduler 會管理 threads 的 ready list，當某個 thread 可以使用 CPU 的時候就會呼叫 ReadyToRun()，把 thread 的 status 設成 READY，並加進 list 的最後面，像是在 Fork()、Yield()都會呼叫到 ReadyToRun()。

4. Difficulties and Feedback

經過了 MP1 之後對 NachOS 有比較熟悉了，所以這次作業花了比較少的時間在理解架構。但是有時候 trace 到組語就停下來，導致沒有辦法 trace 到函式的盡頭，也因此沒辦法完全掌握它的奧義，比如 trace 到 SWITCH(oldThread, nextThread)時，就像註解說的，我們只能 view of the thread and from the perspective of the "outside world"，有點可惜，也算是在寫作中遇到最大的困難。上課的時候花了很多時間在介紹 memory 的管理，雖然聽課的時候就覺得很神奇了，但是真正實作的感覺就像是看著說明書組樂高一樣，感覺更理解這個章節了。