MP2_Report_36

109062233蘇裕恆：

50%trace code 50% implementation

109062320朱季葳：

50%trace code 50% implementation

Trace code

Overall execution

KERNEL::KERNEL()

```
Kernel::Kernel(int argc, char **argv)
{
    randomSlice = FALSE;
    debugUserProg = 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], "-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++;
#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";
        }
    }
}
```

首先initialize一些下面將會用到的參數：

- randomSlice:
  設成false而在下面call RandomInit()之後設成true，而RandomInit()的功用在於產生一個隨機的亂數，而在network.h的註解中寫道random noumber是用來選擇要丟掉的packet的。
- debugUserProg:
  用來表示是否要在每個user instruction執行過後跳到debugger
- consoleIn, consoleOut則是console input/output的file
  而在下面實作的部分，程式依舊分成FILESYS_STUB以及FILESYS來實作。
  這裡有個比較特別的點可以對照到MP1的部分，
  formatFlag在後來被傳入FILESYS的FileSystem()
  function，來表示disk裡面有沒有資料，而FILESYS_STUB的部分則沒有傳入這個參數。
  而在這裡他定義了一個kernel並且設定了假設今天在kernel後面加了一些flag會如何處理
  e.g.
  nachos -rs 5 就是將peusdo-random的seed設為 5 並將randomslice設為true
  像我們在mp1時候的
  nashos -e halt 就是將execfile[]加上 halt
  整體來說就是這樣 因為有太多flag就不一一贅述了。

KERNAL::EXECALL()

在這裡有一個for loop iterate through execfile然後call Exec去執行execfile[i]，在結束之後call finish()讓現在在CPU的thread結束執行。

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

KERNAL::EXEC()

在這邊，程式create一個thread並allocate address space給他(create new AddrSpace)，再將原本的execute的東西pass到Fork裡面讓Fork call StackAllocate去分空間給thread的動作，最後將threadnum++ 並將create的thread number 回傳。

- **New a thread**
- **ForkExecute**

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

    return threadNum-1;
```

KERNEL::FORKEXECUTE()

在Exec的地方有提到會call StackAllocate去分空間給thread的部分，而在stack allocate裏面，程式會把ForkExecute()放到machine[InitialPCState]裏面，如此一來在initial的時候就會call到ForkExecute()並執行它。
在這個部分首先，他會先Load(要execute的file的data)到physical memory，load成功才會call AddrSpace::execute去執行這個file

```
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  }
```

請見下面兩個(AddrSpace::Load以及AddrSpace::Execute)

ADDRSPACE::LOAD

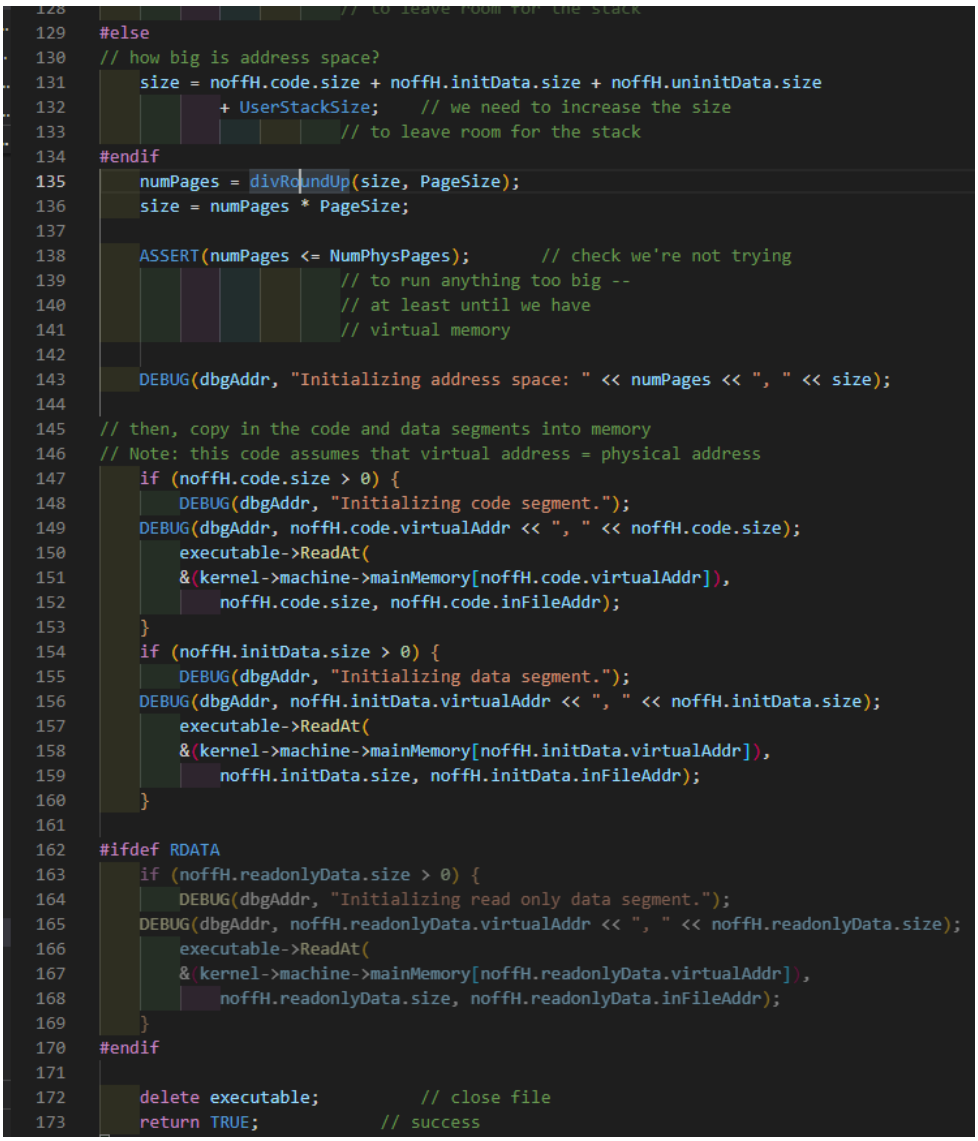
```
105  bool
106  AddrSpace::Load(char *fileName)
107  {
108      OpenFile *executable = kernel->fileSystem->Open(fileName);
109      NoffHeader noffH;
110      unsigned int size;
111
112      if (executable == NULL) {
113      cerr << "Unable to open file " << fileName << "\n";
114      return FALSE;
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
```

首先，他會先將file依據他的filename打開，並確認他是否打開成功以及是否為NOFF(Nachos Object File Format)格式。
注 底下為noffH的structure

```
typedef struct noffHeader {
   int noffMagic;            /* should be NOFFMAGIC */
   Segment code;             /* executable code segment */
   Segment initData;         /* initialized data segment */
#ifdef RDATA
   Segment readonlyData;     /* read only data */
#endif
   Segment uninitData;              /* uninitialized data segment --
                             * should be zero'ed before use
                             */
} NoffHeader;

```

而在這邊的if判斷式call到的WordToHost則是將big endian 轉成 little endian 的format，假如noffMagic!= NOFFMAGIC且經過WordToHost的轉換之後noffMagic = NOFFMAGIC的話就call SwapHeader()來把noffH裡面的資料都轉換成little endian來統一資料的格式。如果轉換之後仍然不符合所求的話就assert fail



```
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
           + UserStackSize;    // we need to increase the size
                               // to leave room for the stack
#endif
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);       // check we're not trying
                               // to run anything too big --
                               // at least until we have
                               // virtual memory

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
    DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

    delete executable;          // close file
    return TRUE;            // success
```

接著，他會先整體需要的size設定完成，以我們這次的作業來說就是size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize;

再者，我們將numPage = size / PageSize 並將 size resize 為 PageSize* numPage，因為我們要以page作為存取的單位。

假設都順利進行並且code.size等都為合法的話，他們就會將executable的值儲存到mainMemory[virtual_mem]中。
在這裡有個特別的點是因為尚未支援multiprogramming的關係，所以virtual address會直接對應到physial address，如同在AddrSpace的contructor所示（詳情請見下面的連結），而這個部分我們將在實作時進行改動。

- **AddrSpace()**

最後再將file close並return true，整個load 就結束了。

AddrSpace::Execute()

```
185  void
186  AddrSpace::Execute(char* fileName)
187  {
188
189      kernel->currentThread->space = this;
190
191      this->InitRegisters();       // set the initial register values
192      this->RestoreState();        // load page table register
193
194      kernel->machine->Run();      // jump to the user progam
195
196      ASSERTNOTREACHED();          // machine->Run never returns;
197                                   // the address space exits
198                                   // by doing the syscall "exit"
199  }
200
201
```

因為原始的code尚未支援multiprogramming，所以我們在這裡先就原始的code來做解釋並提出之後implementation將會如何改動來支援multiprogramming。
這邊主要執行的是user program (因為 currentThread->space 都是user program)
這裡會先將整個current thread的space設為addrspace (definition of addrspace : Data structures to keep track of executing user programs)
並且call addrspace的 InitRegisters 與 RestoreState，

1. init:
   會先將全部reg的value初始化成0，接著再改動一些具有特定意義的reg的值，如下方所列：

- PCReg:因為要從最一開始的instruction開始執行，所以將PCReg設成0
- NextPCReg:因為每個instruction佔4byte的記憶體，所以NextPCReg為PCReg + 4 = 0 + 4 = 4，而NextPCReg的功用主要是為了支援branch instruction。
- StackReg:存放address space的界線，所以將StackReg設為numPages * PageSize - 16，最後會-16是要避免越界到

其他thread的部分。

2. restore :
   在這裡直接將machine->pageTable與machine->pageTableSize設定為pageTable以及pageTableSize
   最後，就進入了Run去執行thread裡的instruction(在第一次作業有寫過，這邊就不贅述了)

而在trace這邊的code的時候我們也有看到有另外一個function叫做SaveState()且這個函式目前是沒有實作到的，而我們將在implementation的部分改動Execute以及SaveState(), RestoreState()的部分來實現支援multiprogramming的狀態下，當要進行context switch的時候必須利用SaveState()去記錄在context switch之前，CPU所執行的那個thread在哪個state，然後在換回執行該thread的時候call RestoreState()去接續之前的state並繼續執行程式。

New_A_Thread

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

    return threadNum-1;
```

如同我們在前面Kernel::Exec()所提到的，程式會create一個thread並allocate地址給他，而這個時候就會呼叫到AddrSpace的constructor。

AddrSpace::AddrSpace()

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
    bzero(kernel->machine->mainMemory, MemorySize);
}
```

在nachos裡面 他的做法其實其實很naive，因為是uniprogramming。因此在創立的時候將virtualPage跟physicalPage 設為一樣。
之後再將memory zero out。

THREAD::FORK()

```
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 }
```

在這裡程式invoke傳進來的(*func)(arg)，在這裡的意思不是call那個函式的意思，而是可以讓caller(Fork)以及callee(ForkExecute)同時運行，下列補充有更多關於(*func)(arg)的解釋。

- **Supplement**
  而在這個部分他call了StackAllocate()去allocate空間給thread，如同我們在Kernal::Exec()的部分所解釋的一樣，然後去disable interrupt，並call ReadyToRun()把這個thread加到ready list裡面，最後再把interrupt enable回來。

SCHEDULER::READYTORUN()

```
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  }
```

這邊將這個thread的status設為ready (但還沒有跑)
並將這個thread加到Scheduler::readyList裡面。

THREAD::STACKALLOCATE()

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

基本上他會先將一個stack 用AllocateBoundedArray()創造出來

並會依據不同的ISA來做判斷 以本次平台為例 使用UNAME -A(會印出系統的所有的資訊) 可以發現使用的架構為X86

```
x86_64
[os22team36@localhost ~]$ uname -a
Linux localhost.localdomain 3.10.0-1160.45.1.el7.x86_64 #1 SMP Wed Oct 13 17:20:51 UTC 2021 x86_64 x86_64 x86_64
GNU/Linux
```

因此stacktop等地配置為下。

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

最後將machineState的配置設定完成並即結束整個StackAllocate的流程，而machineState會在之後進行context switch的時候派上用場。

```
350   #ifdef PARISC
351       machineState[PCState] = PLabelToAddr(ThreadRoot);
352       machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
353       machineState[InitialPCState] = PLabelToAddr(func);
354       machineState[InitialArgState] = arg;
355       machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
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   }
```

THREAD::FINISH()

我們在 StackAllocate()的最後會將machineState[WhenDonePCState] 設為(void*)ThreadFinish;

其實只是因為c不支援直接pointers to member functions

因此需要連到(void*)ThreadFinish，而ThreadFinish()會call到Thread::Finish()，如下圖所示。

```
static void ThreadFinish()    { kernel->currentThread->Finish(); }
```

而在Thread::Finish()中，他將interrupt disable掉並且call

Sleep() (因為Sleep 需要再interrupt 的status為off)，而在這裡的註解中還有提到為什麼不直接de-allocate掉thread，而是call sleep之後再間接de-allocate掉thread。這是因為在這個階段程式可能還正在跑這個thread或者還在stack上，所以在之後檢查到程式正在跑其他thread的或者這個thread finish的時候才會de-allocate掉這個thread(詳情請見Scheduler::Run()的解說） 。

```
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   }
180
```

THREAD::SLEEP()

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

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

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

    status = BLOCKED;
    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();  // no one to run, wait for an interrupt
    }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

在這裡他會先把currenThread block掉，然後call FindNextToRun( )，並於return值為NULL(代表readyList裡面沒有thread)的時候把CPU idle掉，直到readyList裡面有thread為止，而這個thread只能是做完I/O之後從waiting state回到ready state的thread，因為在idle()裡面程式會call checkIfDue()來確保pending interrupts有被處理到，而如果沒有pending interrupt的話，他則會return false，然後idle()會Halt掉整個kernel。(詳情請見下列補充)

- **Supplement**
  如果後續kernel沒有被halt掉的話(代表有pending interrupt->有runnable thread)，程式會call Scheduler::Run()去run 下一個thread。

SCHEDULER::RUN()

```
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: " << nextThread->getName());
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);
134
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  }
```

這個function的主要目的是為了Run傳進來的NextThread，而這裡主要分為兩種情況

- 在原本的thread還沒有跑完的情況下切到下一個thread
- 原本的thread已經finish了(finishing = true)，而這個正是create a new thread的路徑所會走到的部分。
    - 在這裡首先，程式會先將原本的thread設為oldThread，並且將toBeDestroy指到oldThread。
      而假設oldThread是User program，我們就需要call Thread::SaveUserState()將他的registers記錄起來，因為user program會有兩組CPU register，分別是給user mode跟kernel mode，而透過SaveUserState()可以只用一組CPU register來實作，因為每當有context switch發生的時候都會把該user program的CPU register的值存下來。
      接下來確認oldThread確認有沒有overflow，然後將currentThread設為next thread 並將他的Status設為running。
      下一步，call switch，詳情請見Switch.s (基本上就是將整個stack pointer與其他pointer的位置調換掉)

最後call CheckToBeDestroyed()來將已經finish(toBeDestroyed!=NULL)的thread delete掉。
假如oldThread沒有被delete掉的話，將CPU register的資料以及PageTable, PageTableSize restore回去。

Answering question

1. How does Nachos allocate the memory space for a new thread(process)?
   Ans :
   Create 一個AddrSpace給t[threadNum]
   ```
   t[threadNum]->space = new AddrSpace();
   ```

2. How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?
   Ans :
   他會call AddrSpace() 來initialize the memory content of a thread，並且同時這個function會call bzero將整個memory清空
   Loading 的話請詳見AddrSpace::Load() 基本上就是將noff檔read出來並會依據他的snoffH.code.size 還有UserStackSize 等來決定最後的Size
   並用readAt來獲取他的content

3. How does Nachos create and manage the page table?
   Ans :
   Create -> AddrSpace::AddrSpace()
   manage -> AddrSpace::RestoreState()

4. How does Nachos translate addresses?
   Ans :
   從Machine::ReadMem()以及Machine::WriteMem可以看到他們call了Machine::Translate()去將virtual address轉換成physical address。
   而Translate()裡面基本上就是計算出virtual page number以及offset並找到相對應的page table，然後依據該page table所對應到的physical page number(page frame)來得到前段的地址，然後在乘上pageSize之後再加上offset即為physical address，如下圖所示。
   ```
   *physAddr = pageFrame * PageSize + offset;
   ```

- physical page number的計算:virtAddr / PageSize
- offset的計算:virtAddr % PageSize

5. How Nachos initializes the machine status (registers, etc) before running a thread(process)
   Ans :
   他在forkexecute裡面，會先call scheduler->ReadyToRun(this) 將 thread->setStatus(READY) 並在之後在AddrSpace::Execute() 裡面，call InitRegistor() 來將全部的registor設定完成(詳情請見前面AddrSpace::Execute()的解釋) 之後就可以跑了。

6. Which object in Nachos acts the role of process control block
   Ans :
   Thread

7. When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?
   Ans :
   當Thread::Fork()被call到的時候，在這個function裏面會call ReadyToRun()把thread加到ready list裡面。

Supplement

CODE/LIB/UTILITY.H

```
27  // This declares the type "VoidFunctionPtr" to be a "pointer to a
28  // function taking an arbitrary pointer argument and returning nothing".  With
29  // such a function pointer (say it is "func"), we can call it like this:
30  //
31  //  (*func) ("help!");
32  //
33  // This is used by Thread::Fork as well as a couple of other places.
34
35  typedef void (*VoidFunctionPtr)(void *arg);
36  typedef void (*VoidNoArgFunctionPtr)();
```

由於在上面是call ForkExexcute，因此本身他的做法是讓試騎在allocate的時候也會叫到StackAllocate並將ForkExecute也傳進去。

INTERRUPT::IDLE()

```
Interrupt::Idle()
{
    DEBUG(dbgInt, "Machine idling; checking for interrupts.");
    status = IdleMode;
    DEBUG(dbgTraCode, "In Interrupt::Idle, into CheckIfDue, " << kernel->stats->totalTicks);
    if (CheckIfDue(TRUE)) { // check for any pending interrupts
    DEBUG(dbgTraCode, "In Interrupt::Idle, return true from CheckIfDue, " << kernel->stats->totalTicks);
    status = SystemMode;
    return;         // return in case there's now
                    // a runnable thread
    }
    DEBUG(dbgTraCode, "In Interrupt::Idle, return false from CheckIfDue, " << kernel->stats->totalTicks);

    // if there are no pending interrupts, and nothing is on the ready
    // queue, it is time to stop.   If the console or the network is
    // operating, there are *always* pending interrupts, so this code
    // is not reached.  Instead, the halt must be invoked by the user program.

    DEBUG(dbgInt, "Machine idle.  No interrupts to do.");
    cout << "No threads ready or runnable, and no pending interrupts.\n";
    cout << "Assuming the program completed.\n";
    Halt();
}
```

他會先CheckIfDue(True)並且假設裡面有任何的pending interrupt救回return true否則的話就會將整個kernel Halt。
而CheckIFDue()裡面有兩種情況會return false:

- no pending interrupt
- if the scheduled time for the interrupt is not yet and there exist thread in ready queue, it's not the time for the interrupt handler to handle next interrupt

而在這裡的情況下我們已經確定ready queue沒東西所以第二種情況不會發生

Implementation

machine.h

在這裡我們只是單純把MemoryLimitException加入ExceptionType來符合spec的要求，如下圖。

```
enum ExceptionType { NoException,           // Everything ok!
                     SyscallException,      // A program executed a system call.
                     PageFaultException,    // No valid translation found
                     ReadOnlyException,     // Write attempted to page marked
                                            // "read-only"
                     BusErrorException,     // Translation resulted in an
                                            // invalid physical address
                     AddressErrorException, // Unaligned reference or one that
                                            // was beyond the end of the
                                            // address space
                     OverflowException,     // Integer overflow in add or sub.
                     IllegalInstrException, // Unimplemented or reserved instr.
                     // 109062233
                         MemoryLimitException,
                     NumExceptionTypes
};
```

kernel.h && kernel.cc (http://kernel.cc)

我們在kernel.h定義kernel class的地方加入UsedPhysAddr[NumPhysPages]來記錄每個physical page是否有被用到了。
而在kernel.cc裡，我們在kernel::Initialize()初始化UsedPhysAddr裡的每個element為false，代表他們都還沒有被用過。

addrSpace.cc (http://addrSpace.cc)

AddrSpace::AddrSpace()

我們在這裡把原本的東西都刪掉，改到AddrSpace::Load()才initialize page table，因為要支援multiprogramming的關係。

AddrSpace::~AddrSpace()

我們在這裡把這個addrSpace有用到的physical page table都設為沒有用過(改動UsedPhysAddr)，然後delete掉這個addrSpace自己的pageTable。

AddrSpace::Load(char *fileName)

在這裡主要分為三個部分

1. 處理MLE
- thread本身所需page多於physical page:
  在這個部分其實在原本的template code就有assert fail了，但是由於他也符合spec中所提到的MLE範疇，所以我們在檢查到這個情形後，在assert fail之前就raise exception。
- run out physical page
  這個部分則是在跑for loop尋找找適合的physical page來放資料時，發現資料還沒放完結果physical page table就全部被放滿了，所以我們在檢查到這樣的情況就raise exception。

2. 找到適當的page table去load file的資料到main memory
   在這裡我們利用原本template就有的numPage(代表這個thread需要多少page)，還有NumPhysPages來建一個雙層迴圈，幫這個thread所需要的page都找到相對應且沒有被放資料的physical page去放他，然後把被取用的physical page註記(利用UsedPhysAddr)，使得其他thread不會把它拿來用，然後再做一些初始化pageTable跟要用到的main memory的區塊。

3. 把該file的code還有initData透過readAt放到main memory
   在這裡code,initData,readonly data的放法其實一模一樣所以統一做說明。
   我們的放法主要就是先去檢查他們的size，如果都等於0就直接return true，然後接下來再個別檢查是否大於0(合法範圍)然後把資料放入main memory。而我們要先檢查他們的size是否大於一個page table的size，如果大於的話就先

以page table為單位來放，然後每次都把他們的size減掉page table的size來記錄還有多少沒放到，直到放完最後的部分。
要注意的點是最後的部分因為小於page table的size所以在填readAt的第二個argument的時候直接放入剩餘要填的size就好。
詳細情形如下圖所示:

```
for (int i = 0; i < numPages; i++) {
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    if(code_size > 0){
        if(code_size > PageSize){
            executable->ReadAt(&(kernel->machine->mainMemory[pageTable[code_starting_page].physicalPage*PageSize
            + code_remainder]), PageSize, noffH.code.inFileAddr + i*PageSize);
            code_starting_page++;
            code_size -= PageSize;
        }

        else{
            executable->ReadAt(&(kernel->machine->mainMemory[pageTable[code_starting_page].physicalPage*PageSize
            + code_remainder]), code_size, noffH.code.inFileAddr + i*PageSize);
            code_starting_page++;
            code_size -= code_size_rem;
        }
    }
    else break;
}
```

最後還有一點比較特別是在處理readonly data的時候我們才把pageTable的readOnly設為true，因為前面是統一設為false的。

AddrSpace::SaveState()

在這裡是為了支援可能有context switch的部分，我們把kernel->machine->pageTable存到pageTable，然後把kernel->machine->pageTableSize存到numPages就結束了，這裡可以跟RestoreState()相對應。

Testcase

- base case:execute consoleIO_test1 and consoleIO_test2:

```
[os22team36@localhost build.linux]$ cd ..
[os22team36@localhost code]$ cd test/
[os22team36@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
```

- test executing 4 files and no MLE happen

- test result

```
[os22team36@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2 -e consoleIO_test3 -e consoleIO_test4
consoleIO_test1
consoleIO_test2
consoleIO_test3
consoleIO_test4
9return value:0

8
10
9
8
15
16
17
7
6
1return value:0
8
19
return value:0
7
6
5
4
3
2
1
return value:0
^C
```

- debug message

```
[os22team36@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2 -e consoleIO_test3 -e consoleIO_test4 -d u
consoleIO_test1
consoleIO_test2
consoleIO_test3
consoleIO_test4
Initialize the addrspace

And the numPage is 12
the noff file have code size of  480 and initData size = 0with the physical page 0

Received Exception 1 type: 16

Print Int

Initialize the addrspace

And the numPage is 12
the noff file have code size of  480 and initData size = 0with the physical page 12

Received Exception 1 type: 16

Print Int

Initialize the addrspace

And the numPage is 14
the noff file have code size of  592 and initData size = 0with the physical page 24

Received Exception 1 type: 16

Print Int

Initialize the addrspace

And the numPage is 74
the noff file have code size of  384 and initData size = 0with the physical page 38
```

從上述結果我們可以看到physical page沒有被用完，然後符合我們預期的依序填資料下去。

- test executing 4 files and MLE happen in consoleIO_test4(file本身所需page多於physical page)
  - test result

```
[os22team36@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2 -e consoleIO_test3 -e consoleIO_test4
consoleIO_test1
consoleIO_test2
consoleIO_test3
consoleIO_test4
Unexpected user mode exception 8
Assertion failed: line 228 file ../userprog/exception.cc
Aborted
```

  - debug message

```
physical page used now 37

the noff file have code size of  592 and initData size = 0with the physical page 24

Received Exception 1 type: 16

Print Int

we need numPage 1262here, and the physical page number is128

Received Exception 8 type: 16

Unexpected user mode exception 8
Assertion failed: line 228 file ../userprog/exception.cc
Aborted
```

在這裡我們把第四個file的陣列開到4萬，符合達到MLE的範疇，而這裡的MLE是因為file自己本身所需的page數量就已經比physical page的數量多了，而在原本的template會Assert fail，但是我們在他Assert fail之前就先raise exception了。

- test executing 4 files and MLE happen in consoleIO_test4(run out physical page)

- test result

```
[os22team36@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2 -e consoleIO_test3 -e consoleIO_test4
consoleIO_test1
consoleIO_test2
consoleIO_test3
consoleIO_test4
Unexpected user mode exception 8
Assertion failed: line 228 file ../userprog/exception.cc
Aborted
```

- debug message

```
Print Int

we need numPage 105here, and the physical page number is128

Initialize the addrspace

And the numPage is 105
physical page used now 38
physical page used now 124

physical page used now 125

physical page used now 126

physical page used now 127

Received Exception 8 type: 16

Unexpected user mode exception 8
Assertion failed: line 228 file ../userprog/exception.cc
Aborted
```

在這裡所有的file所需的page都小於physical page的數量，但由於跑到最後一個file的時候physical page不夠用了，所以raise exception

註：上圖中debug message所呈現的physical page used now是從0開始計算到127，所以總共有128個physical pages