Cover Page

Team61

Member List

> 108070039 羅稑涵
> 108070038 簡志宇

Member Contribution

> Trace code 羅稑涵, 簡志宇
> Part I report 羅稑涵
> Implementation 簡志宇
> Part II report 簡志宇

Part II-1

threads/kernel.cc

- Kernel::Kernel()

  Kernel 的作用是模擬 OS，因此當我們從 terminal 輸入指令時，就會先建構一個 Kernel object(詳見 main.cc)，初始化各種與 kernel 相關的 Flag (例如：I/O來源、FileSystem 的 formatFlag)，並且解析指令，執行對應的動作。

```
Kernel::Kernel(int argc, char **argv)
{
// 初始化
    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
// 以下開始解析指令
    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;
        .
        .
        .
        .
        .
        .
        .
    }
}
```

- Kernel::Initialize()

  這個 function 的作用也是初始化，不過對象主要是 OS 管理的各種功能(object, 像是 Machine, Scheduler, Interrupt等)，因為部分初始化會用到 constructor 裡初始化的 flag，所以才分開寫。

```
void
Kernel::Initialize()
{
    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);

    stats = new Statistics();   // collect statistics
    interrupt = new Interrupt;    // start up interrupt handling
    scheduler = new Scheduler();  // initialize the ready queue
    alarm = new Alarm(randomSlice); // start up time slicing
    machine = new Machine(debugUserProg);
    synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
    synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
```

```
    synchDisk = new SynchDisk();    //
#ifdef FILESYS_STUB
    fileSystem = new FileSystem();
#else
    fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
    postOfficeIn = new PostOfficeInput(10);
    postOfficeOut = new PostOfficeOutput(reliability);

    for(int i = 0; i < NumPhysPages; i++)
        usedFrameTable[i] = 0;

    interrupt->Enable();
}
```

- Kernel::ExecAll()

  這個 function 的目的是執行指令中輸入的各個執行檔(也就是之後會被 load 進來的 program)，所以會在 for loop 中呼叫 Exec。當全部都執行結束後，currntThread(Kernel) 會呼叫 Finish 執行後續動作(請見thread.cc)。

```
void Kernel::ExecAll()
{
    // 用一個loop依序執行所有的execfile
  for (int i=1;i<=execfileNum;i++) {
    int a = Exec(execfile[i]);
  }
    // 執行完就進入terminate state(參見thread.cc)
  currentThread->Finish();
    //Kernel::Exec();
    //cout << execfileNum << '\n';
}
```

- Kernel::Exec()

  這個 function 執行傳進來的 execfile，流程如下：

  1. 先建立一個新的 Thread object 作為執行此程式的 control block
  2. 賦予此 thread 一個 Addrspace，之後會把 program load 進這個 object 管理的空間
  3. 呼叫 Fork 來 fork 這個 child thread。

```
int Kernel::Exec(char* name)
{
    // 要執行execfile就要建一個新的child thread(參見thread.cc)
  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()

  fork 傳進來的 execfile，先把 program load 進 addrspace，再呼叫 Execute 執行程式。

```
void ForkExecute(Thread *t)
{
    // 確認program是否已經有load到addrspace(參見addrspace.cc)
```

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

    // 執行thread(參見addrspace.cc)
    t->space->Execute(t->getName());

}
```

threads/thread.cc

- Thread::Sleep()

  Current thread現在的狀態是blocked的，next thread若不存在則會讓CPU處於idle的狀態，直到有新的thread被傳入ready queue才會跳出while-loop。

```
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;

  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);
}
```

- Thread::StackAllocate()

  這個function前半段是在初始化stack與stackTop(current thread pionter)，後半段machineState由ThreadRoot()開頭，接著將procedure需要的物件輸入到machineState裡，最後的ThreadFinish是用來終止thread。

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
  stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
  #ifdef PARISC
  // HP stack works from low addresses to high addresses
  // everyone else works the other way: from high addresses to low addresses
  stackTop = stack + 16;  // HP requires 64-byte frame marker
  stack[StackSize - 1] = STACK_FENCEPOST;
  #endif
  .
  .
  .
  .
  #ifdef x86
  stackTop = stack + StackSize - 4; // -4 to be on the safe side!
  *(--stackTop) = (int) ThreadRoot;
  *stack = STACK_FENCEPOST;
  #endif
  #ifdef PARISC
  machineState[PCState] = PLabelToAddr(ThreadRoot);
```

```
  machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
  machineState[InitialPCState] = PLabelToAddr(func);
  machineState[InitialArgState] = arg;
  machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
  #else
  machineState[PCState] = (void*)ThreadRoot;
  machineState[StartupPCState] = (void*)ThreadBegin;
  machineState[InitialPCState] = (void*)func;
  machineState[InitialArgState] = (void*)arg;
  machineState[WhenDonePCState] = (void*)ThreadFinish;
  #endif
}
```

- Thread::Finish()

  當fork程序結束，會呼叫這個function。初始會先將interrupt設為disabled因為Thread::Sleep()是預設interrupt是disabled的，最後就會呼叫Thread::Sleep()。

```
void
Thread::Finish ()
{
  (void) kernel->interrupt->SetLevel(IntOff);

  ASSERT(this == kernel->currentThread);

  DEBUG(dbgThread, "Finishing thread: " << name);
  Sleep(TRUE);        // invokes SWITCH
  // not reached
}
```

- Thread::Fork()

  此function裡的Thread::StackAllocate()是來設置一個stack，再來先將interrupt設為disabled，因為Scheduler::ReadyToRun()是假設interrupt是disabled的，藉由Scheduler::ReadyToRun()讓thread變成ready的狀態。

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts
          // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

- Thread::Yield()

  這個function是當有其他的thread在ready狀態時會被呼叫。首先將interrupt設為disabled，呼叫FindNextToRun()來看是否有thread是ready的，如果有的話則會呼叫Scheduler::ReadyToRun()與Scheduler::Run()來存取current thread。

```
void
Thread::Yield ()
```

```
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
      kernel->scheduler->ReadyToRun(this);
      kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

- ThreadRoot()

  如果有一個新開始的thread的話，這個function會被SWITCH()呼叫。下面稍微解釋一下各個參數代表的意思， StartupPC是指當thread被開始時所需經的routine; InitialArg是指thread所需要的物件; InitialPC是給program counter跑thread所使用的; WhenDonePC是當thread return時所需經的routine。

```
.globl ThreadRoot
.ent  ThreadRoot,0
ThreadRoot:
  or  fp,z,z    # Clearing the frame pointer here
        # makes gdb backtraces of thread stacks
        # end here (I hope!)

  jal StartupPC # call startup procedure
  move  a0, InitialArg
  jal InitialPC # call main procedure
  jal   WhenDonePC  # when done, call clean up procedure

  # NEVER REACHED
  .end ThreadRoot
```

- SWITCH()

  停止舊的thread運作，讓新的thread能load進來。這裡的a0是代表pointer對上舊的thread，a1代表的是pointer對上新的thread。

```
.globl SWITCH
.ent  SWITCH,0
SWITCH:
  sw  sp, SP(a0)    # save new stack pointer
  sw  s0, S0(a0)    # save all the callee-save registers
  sw  s1, S1(a0)
  .
  .
  .
  sw  fp, FP(a0)    # save frame pointer
  sw  ra, PC(a0)    # save return address

  lw  sp, SP(a1)    # load the new stack pointer
  lw  s0, S0(a1)    # load the callee-save registers
  .
  .
  .
  lw  s7, S7(a1)
  lw  fp, FP(a1)
  lw  ra, PC(a1)    # load the return address
```

```
  j ra
  .end SWITCH
```

threads/scheduler.cc

- Scheduler::ReadyToRun()

  當一個thread是ready的狀態時，會將這個thread加入ready queue裡面。

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY);
    readyList->Append(thread);
}
```

- Scheduler::FindNextToRun()

  這個function可以拿出ready queue中的下一個thread來準備執行，若ready queue中沒有等待被執行的thread，則回傳NULL。

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

- Scheduler::Run()

  這個function主要目地是要讓next thread開始執行。首先，當finishing為True時，會將current thread標示為要被刪除，接著檢查current thread是否有overflow。再來新的thread變為current thread，將新的thread狀態設為Running，並藉由呼叫SWITCH來交換舊的thread與新的thread，然後將舊的thread給刪除掉。

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    if (finishing) {  // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }
    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();   // save the user's CPU registers
        oldThread->space->SaveState();
    }
    oldThread->CheckOverflow();       // check if the old thread
              // had an undetected stack overflow
    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running
    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

```
    SWITCH(oldThread, nextThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();     // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

userprog/addrspace.cc

- AddrSpace::Addrspace()

  原本在這個 constructor 中是會建立一個 pageTable，其大小等於 physical page 的數量，而內容則是記錄 virtual page 以及 physical page 相對應的 page number，和一些記錄 protection bit 的變數。這個部分在實作中被我註解掉，並且移到 Load 函式中，其原因請見 Part II-2 implementation 的說明。

```
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);
}
```

- AddrSpace::Load()

  從 kernel 的 ForkExecute 裡呼叫這個 function，在執行前先將 program load 到 memory 裡。這個步驟分成幾個部分：

  1. 轉檔：把執行檔轉成 NachOS 看得懂的格式(noff)

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

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

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
    (WordToHost(noffH.noffMagic) == NOFFMAGIC))
      SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);
```

2. 計算所需 pages：要知道需要多少 page，就需要先知道 program size 總共多大。因此會先判斷是否為 readonly，再計算所需大小，然後再透過以下計算算出需要的 pages。

> numPages = divRoundUp(size, PageSize); // 等同 size/PageSize 取 ceiling

```
#ifdef RDATA
// how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
                                                // we need to increase the size
            // to leave room for the stack
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
      + UserStackSize;  // we need to increase the size
            // to leave room for the stack
#endif
    // 這邊計算要被執行的 executable 總共會需要多少 pages
    numPages = divRoundUp(size, PageSize); // 這一行的size是實際上需要多少bytes
    size = numPages * PageSize; // 這一行的的size則是總共被分到多少bytes

    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);
```

3. 建立 pageTable 並把 program load 進 memory：
   - 這邊可看到原本直接在 constructor 建立 pageTable 即可，是因為所需page數直接等於所有的 physical page 數。但實作了 multiprogramming 以後，因為需要知道所需 virtual memory page 數，所以在這邊算完後才能建立 pageTable。
   - 建完 pageTable 後就能把 program load 進去了。首先算出前面被分配到的 page 的 physical address(by AddrSpace::Translate())，再把資料放入 main memory 裡的這些位置就完成了。這邊要注意的是，必須一個 page 一個 page 慢慢 load，因為這樣才能在Translate 裡面設定每個 page 的 protection bit。

```
// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
//
// MP2: 要實作 page table, 所以 virtual address != physical address
// virtual address % page size => page offset
// page offset + frame number * page size = physical address
// (等同於把 page offset append 到 frame number 後面)

    // 108070038
    pageTable = new TranslationEntry[numPages];
    for (int i = 0; i < numPages; i++) {

        int frame = kernel->FindAvailableFrame();

        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = frame;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
```

```
        bzero(kernel->machine->mainMemory + frame * PageSize, PageSize);
    }
    unsigned int physAddr;
    ExceptionType err;

    if (noffH.code.size > 0) {

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

        int total, curOffset = 0;
        for(total = noffH.code.size, curOffset = 0; total > 0; total -= PageSize, curOffset += Pa
geSize){

            unsigned int loadSize;
            if(total < PageSize)
                loadSize = total;
            else loadSize = PageSize;

            err = Translate(noffH.code.virtualAddr + curOffset, &physAddr, 1);
            if(err != NoException)
                RaiseException(err);

            executable->ReadAt(
            &(kernel->machine->mainMemory[
            physAddr
            ]), loadSize, noffH.code.inFileAddr + curOffset);
        }

    }

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

        int total, curOffset = 0;
        for(total = noffH.initData.size, curOffset = 0; total > 0; total -= PageSize, curOffset +
= PageSize){

            unsigned int loadSize;
            if(total < PageSize)
                loadSize = total;
            else loadSize = PageSize;

            err = Translate(noffH.initData.virtualAddr, &physAddr, 1);
            if(err != NoException)
                RaiseException(err);

            executable->ReadAt(
            &(kernel->machine->mainMemory[
            physAddr
            ]), loadSize, noffH.initData.inFileAddr + curOffset);
        }
    }

#ifdef RDATA

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

        int total, curOffset = 0;
        for(total = noffH.readonlyData.size, curOffset = 0; total > 0; total -= PageSize, curOffs
et += PageSize){

            unsigned int p = divRoundUp(noffH.readonlyData.size, PageSize);
            for(int i = 0; i < p; i++){
                pageTable[noffH.readonlyData.virtualAddr/PageSize + i].readOnly = TRUE;
            }
```

```
            unsigned int loadSize;
            if(total < PageSize)
                loadSize = total;
            else loadSize = PageSize;

            err = Translate(noffH.readonlyData.virtualAddr, &physAddr, 0);
            if(err != NoException)
                RaiseException(err);

            executable->ReadAt(
            &(kernel->machine->mainMemory[
            physAddr
            ]), loadSize, noffH.readonlyData.inFileAddr + curOffset);
        }
    }
```

- AddrSpace::Execute()

  從 kernel 的 ForkExecute 裡呼叫這個 function，主要工作就是執行 user program 的事前準備，流程如下：

  1. 將現在正要執行的 thread 與 space 連結起來
  2. InitRegisters() 設定 machine register 的初始值
  3. RestoreState() 找到對應的 page table
  4. 跳到 machine->Run() 開始執行

```
void
AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

    this->InitRegisters();    // set the initial register values
    this->RestoreState();   // load page table register

    kernel->machine->Run();   // jump to the user progam

    ASSERTNOTREACHED();     // machine->Run never returns;
          // the address space exits
          // by doing the syscall "exit"
}
```

流程

1. Kernel::ExecAll()

   依序執行所有 program

2. Kernel::Exec()

   a. 前一個 thread 執行完 ExecAll() 裡的迴圈後

      i. Thread::Finish()

         當所有 program 都 fork 完成後原本的 thread 就會進入 Finish()

      ii. Thread::Sleep()

          1. Scheduler::FindNextToRun()

這時候新的 thread 已經被放入 ready queue

2. Scheduler::Run()

iii. SWITCH()

context switch —> 接下來新的 thread 會執行ThreadRoot()

(接到 b. 的 ii.)

b. 新的 thread

i. Thread::Fork()

1. Thread::StackAllocate()

2. Scheduler::ReadyToRun()

(請見 a. 的 ii. 的 1.)

ii. ThreadRoot：Thread::Begin()

這邊會 delete 掉前一個 thread，因此回到只有一個 thread 的狀態

3. ThreadRoot：Kernel::ForkExecute()

a. AddrSpace::Load()

b. AddrSpace::Execute()

i. Machine::Run()

執行 user program

4. ThreadRoot：Thread::Finish()

Q&A

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

  Ans：

  首先，會呼叫Kernel::ExecAll()，ExecAll()裡的for-loop會呼叫Kernel::Exec()，而Exec()則會設置新的thread與address space，呼叫Thread::Fork()來建立新的thread。跳出for-loop後，會呼叫Thread::Finish()導入下一個thread。

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

  Ans：

  AddrSpace::Load()的中半段 if 判斷裡面的executable→ReadAt()的功用分別是將code與data段寫入記憶體中。Load()後半段呼叫AddrSpace::Translate()將noffH.code中的virtual address轉成physical address。

- How Nachos creates and manages the page table?

  Ans：

  在原本的AddrSpace::Addrspace()是讓virtual page對應相同數量的 physical page。然而，為了解決page分配的問題，我們把:AddrSpace()裡的code註解掉，改到AddrSpace::Load()時再做分配。

因此 Load 裡面會先計算需要用到的 page 數量，再來建立並初始化合適大小的 pageTable，然後當要寫入 data 到 memory 時再設定 protection bit。

- How Nachos translates address?

  Ans：

  當在進行address translation時，會先判斷virtual page number是否大於等於page numbers，沒有問題的話則會讓virtual page number對到相對應的page table，再來檢查 pte是否是readOnly 的狀態，最後則是看physical page frame是否大於number of physical pages。如果上面都沒有問題且設定好use與dirty bit後，經由計算page number及page offset求得physical address。

```
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)
{
    TranslationEntry *pte;
    int               pfn;
    unsigned int      vpn    = vaddr / PageSize;
    unsigned int      offset = vaddr % PageSize;

    if(vpn >= numPages) {
        //cout << "in Translate: " << "vpn = " << vpn << '\n';
        //cout << "in Translate: " << "numPages = " << numPages << '\n';
        return AddressErrorException;
    }

    pte = &pageTable[vpn];

    if(isReadWrite && pte->readOnly) {
        return ReadOnlyException;
    }

    pfn = pte->physicalPage;

    // if the pageFrame is too big, there is something really wrong!
    // An invalid translation was loaded into the page table or TLB.
    if (pfn >= NumPhysPages) {
        DEBUG(dbgAddr, "Illegal physical page " << pfn);
        return BusErrorException;
    }

    pte->use = TRUE;          // set the use, dirty bits

    if(isReadWrite)
        pte->dirty = TRUE;

    *paddr = pfn*PageSize + offset;

    if(*paddr >= MemorySize) {
        return MemoryLimitException;
    }

    //ASSERT((*paddr < MemorySize));
    //cerr << " -- AddrSpace::Translate(): vaddr: " << vaddr <<
    //  ", paddr: " << *paddr << "\n";

    return NoException;
}
```

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

  Ans：

- AddrSpace::InitRegisters()負責將 registers 設定好，流程如下：
  1. 先清空所以 register 的內容
  2. 將 PC 設成 0，假定從virtual address = 0 開始跑
  3. 然後把 next PC 設定成 4(通常一個指令為 4 bytes)
  4. 再來將 stack 起始位置設成 address space 最後面，減16是為了預留些空間以防萬一

```
void
AddrSpace::InitRegisters()
{
    Machine *machine = kernel->machine;
    int i;

    for (i = 0; i < NumTotalRegs; i++)
    machine->WriteRegister(i, 0);

    // Initial program counter -- must be location of "Start", which
    //  is assumed to be virtual address zero
    machine->WriteRegister(PCReg, 0);

    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    // Since instructions occupy four bytes each, the next instruction
    // after start will be at virtual address four.
    machine->WriteRegister(NextPCReg, 4);

   // Set the stack register to the end of the address space, where we
   // allocated the stack; but subtract off a bit, to make sure we don't
   // accidentally reference off the end!
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
    DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}
```

- 另外，AddrSpace::RestoreState()則處理 context switch 之後執行新thread時的設定。他會讓替換 machine 相對應此 thread 的 pageTable 以及 page 數量。

  至於 register 則會透過 Scheduler::run() 裡的 SWITCH() 替換成新 thread 的。

- Which object in Nachos acts the role of process control block?

  Ans：

  Process control block包含process的ID、process的名字、process的狀態等，而有符合上述條件是Thread這個object。

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

  Ans：

  當呼叫Thread::Fork()時，會呼叫Scheduler::ReadyToRun()將thread加入ready queue裡。

Part II-2

說明

從 Part II-1 可以發現，每當執行一個新的 thread 時，若他是 user program 則我們會給他一個包含有 pageTable 的 address space，用來讓他 load data。而為了實現 multiprogramming，我們需要每個thread 的 virtual page 對應到不同的 physical page，這次作業主要就是要實作這個部分。

Implementation

addrspace.cc

- AddrSpace::AddrSpace

  這個 constructor 原本有建立 pageTable，virtual page 數量與 physical page 一樣。但因為要實作 multiprogramming，所以需要先算好真正需要的 page 數，因此把建立 pageTable 搬到 Load 去。

  ```
  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);
  }
  ```

- AddrSpace::Load
  - 建立 pageTable：

    建立這個 thread 專屬的 pageTable，其大小為 numPages(也就是 Load 前一部分所計算的 program 所需 page 數)。

呼叫 kernel->FindAvailableFrame(後面會解說) 去找出可使用的 physical page, 並將其對應的欄位的 protection bit 都初始化，然後把 mainMemory 裡的內容清理掉。

```
bool
AddrSpace::Load(char *fileName)
{
// 計算numPages

// MP2: 要實作 page table, 所以 virtual address != physical address
// virtual address % page size => page offset
// page offset + frame number * page size = physical address
// (等同於把 page offset append 到 frame number 後面)

    // 108070038
    pageTable = new TranslationEntry[numPages];
    for (int i = 0; i < numPages; i++) {

        int frame = kernel->FindAvailableFrame();

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

- 把 program load 進 mainMemory(以readOnly data 為例子，其他雷同)：
    1. 因為這部分的 data 是 read-only，所以我們需要將其對應的 readOnly bit 設成 TRUE，因此先計算要用到多少 page(變數p)，再用一個 for loop 將所有 page 的 readOnly 都設成TRUE。這個地方 code segment 跟 initdata 都不需要做。
    2. 再來要 load data，首先先用 Translate 這個函式找到 virtual address 對應的 physical address(isReadWrite 設成0因為這個部分是 read-only，前兩種 data 就設成1)。在轉換過程中若有發生 exception 則呼叫 RaiseException 來處理(下面說明)。

       有了 physical address 後就把 data load 進 kernel->machine->mainMemory[physAddr] 的位址就行了。這個 load 要用迴圈一頁一頁做，如此一來才能讓每一頁的 protection bit 被設定到(在 Translate 裡)。

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

```
// 計算numPages
// 建立pageTable

    unsigned int physAddr;
    ExceptionType err;
#ifdef RDATA

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

        int total, curOffset = 0;
        for(total = noffH.readonlyData.size, curOffset = 0; total > 0; total
 -= PageSize, curOffset += PageSize){

            unsigned int p = divRoundUp(noffH.readonlyData.size, PageSize);
            for(int i = 0; i < p; i++){
                pageTable[noffH.readonlyData.virtualAddr/PageSize + i].readOn
ly = TRUE;
            }

            unsigned int loadSize;
            if(total < PageSize)
                loadSize = total;
            else loadSize = PageSize;

            err = Translate(noffH.readonlyData.virtualAddr, &physAddr, 0);
            if(err != NoException)
                RaiseException(err);

            executable->ReadAt(
            &(kernel->machine->mainMemory[
            physAddr
            ]), loadSize, noffH.readonlyData.inFileAddr + curOffset);
        }
    }
#endif

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

- AddrSpace::RaiseException

  我在 AddrSpace 新加這個 function，目的是當 Translate address 的時候發生 exception，就用這個函式呼叫 ExceptionHandler 來處理。

```
void
AddrSpace::RaiseException(ExceptionType which)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
```

```
    kernel->interrupt->setStatus(UserMode);
}
```

kernel.*

- 宣告一個記錄 physical page 使用狀況的 table。
  - kernel.h

    ```
    // this table is to record the pages in physical memory
    // are used or not, initialized as 0 and will be set 1
    // if used -- for MP2
    int *usedFrameTable;
    ```

  - kernel.cc/kernel::Initialize()

    ```
    usedFrameTable = new int[NumPhysPages];
    for(int i = 0; i < NumPhysPages; i++)
        usedFrameTable[i] = 0;
    ```

- kernel::FindAvailableFrame

  查找 usedFrameTable，將還未被使用的 physical page return 回去，同時把這個 page 的欄位設為1表示已使用。若沒有找到可使用的 physical page 則 return -1。

  ```
  int Kernel::FindAvailableFrame(){

      int frame = 0;

      for(frame = 0; frame < NumPhysPages; frame++){
          if(!usedFrameTable[frame]){
              usedFrameTable[frame] = 1;
              return frame;
          }
      }
      return -1;
  }
  ```

Difficulties & Feedback

簡志宇

這次的作業我認為較上次難上許多，尤其是 trace code 的部分。在 fork 一個新的thread 的時候要做 context switch，而 scheduler::Run 裡的 SWITCH(oldThread, nextThread) 就是在做這件事，但是實行的確切流程很難理解，一直搞不懂 oldThread 在 SWITCH 前後的角色，這個部分花了我將近一天的時間來了解。

至於 implementation 的部分，一開始比較沒頭緒該如何下手，可是了解整體運作再跟 spec 連結之後就簡單許多了。

羅稑涵

這次作業trace code不比上次的少，但是難度提升了不少，尤其是在理解整個thread的流程，後來是再重看一次老師的講義才比較了解，光這部分就花了不少時間。再來最近遇到段考這次開始還比較晚，所以繳交時已經接近期限差點無法完成本次作業。