MP2: Multi-Programming

110000132 詹振暘, 110021121陶威綸

Team member contribution

陶威綸：code tracing, coding, report

詹振暘：code tracing, report

Function Explanation

1-1 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);
}
```

這段程式碼主要的目的是要讓NachOs的Thread轉變為Sleep(休眠)狀態, 在一開始的時候, 會先透過Assert()來確認當前this是否就是當前thread, 並且確認當前interrupt是屬於InOff狀態, 若其中一項不滿足則中斷(Abort())並顯示錯誤, 接著將狀態設為Block並呼叫cheduler->FindNextToRun()來確認是否有其他Thread需執行, 若無則進Idle, 避免CPU資源浪費, 當找到新的可執行Thread後, kernal會重新啟動nextThread以繼續執行。

1-2 Thread::StackAllocate()

```
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 SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
                                        // least 1 activation record
```

```
                                        // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC                          // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif

...
...
```

StackAllocate()會在一開始建立一個StrackSize大小的stack，用來儲存thread在運行過程中的狀態，包括函數呼叫紀錄和local variable，利用 #ifdef 指令根據特定不同的電腦架構（ex: x86、SPARC、PowerPC）設置stack的起始位置和大小。

1-2 Thread::Finish()

```
void Thread::Finish() {
    (void)kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name);
    if (kernel->execExit && this->getIsExec()) {
        kernel->execRunningNum--;
        if (kernel->execRunningNum == 0) {
            kernel->interrupt->Halt();
        }
    }
    Sleep(TRUE);  // invokes SWITCH
    // not reached
}
```

Thread::Finish() 的功能是透過Sleep()在來處理一個thread的結束，並且透過for loop確認是否所有的exec都完成運行，若成立則透過呼叫Halt()關閉整個NachOS系統。

1-2 Thread::Fork()

```
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);
}
```

Fork()的目的是啟動一個新的thread，首先他會宣告interrupt 和 scheduler以及oldLevel來儲存中斷時的狀態，接著透過StackAllocate來分配stack給thread，最後將新增的thread透過schedular加入排程中，並在最後恢復之前的中斷設定，讓執行回到interrupt前的狀態。

2-1 AddrSpace::AddrSpace()

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

程式已經被加載到記憶體後，用現在的執行緒跑一個使用者程式

1. 設定kernel中正在運行的執行緒位址
2. 初始化registers（使用者層級的registers）
3. 讀取PageTable的位置跟page的數量

2-2 AddrSpace::Execute()

```
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"
}
```

PageTable已被初始化後，將程序從一個檔案儲存到記憶體中。

假設檔案為NOFF檔

1. 打開檔案
2. 將資料分為code, readonly, init, uninit四個部分，並且以總和(size)分配Page空間。分配方式為：
    a. numSize = Round(size / PageSize)
    b. size = Round(size / PageSize) * PageSize
3. 分別將四種資料儲存進mainMemory中的四個位址
4. 關閉檔案

2-3 AddrSpace::Load()

```
bool AddrSpace::Load(char *fileName) {
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);

#ifdef RDATA
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
#else
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size +
UserStackSize;
#endif
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;
    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) {
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }
#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif
    delete executable;  // close file
    return TRUE;        // success
}
```

AddrSpace::Load() 主要功能是將user program載入memory，主要負責讀取executible files、解析 NOFF 格式檔案頭、計算記憶體需求(#ifdef RDATA)，並將程式的代碼段和資料段載入到主記憶體中。

3-1 Kernel::Kernel()

```
Kernel::Kernel(int argc, char **argv) {
    randomSlice = FALSE;
    debugUserProg = FALSE;
    execExit = 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], "-ee") == 0) {
            // Added by @dasbd72
            // To end the program after all the threads are done
            execExit = TRUE;
        } 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";
        }
    }
}
```

Kernel() 的主要功能是在初始化要使用的變數跟指標，接著解析指令的Command-line Options（如-e, -rs, -ci等等）。在 -e 指令中，execfile[++execfileNum] = argv[++i]; 會逐一將argv所分割出的檔案名稱存入execfile，並會透過cout將現在在處理的檔案名印出來。

3-2 Kernel::ExecAll()

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

ExecAll() 會執行全部的threads，並在執行結束後將currentThread設為Finish()來決定要Halt()或是Sleep()。

3-3 Kernel::Exec()

```
int Kernel::Exec(char *name) {
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->setIsExec();
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
    threadNum++;
    return threadNum - 1;
}
```

Exec()會創建一個新的thread並為其分配新的addres space，接著，接著透過Fork將其加入執行序列，準備執行，最後增加threadNum的數量。

3-4 Kernel::ForkExecute()

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

    t->space->Execute(t->getName());
}
```

透過Load(t->getName)，系統會嘗試載入thread t 所對應的程式碼，若失敗則直接返回，若成功載入，則呼叫 Execute 函數來執行該程式。

4-1 Scheduler::ReadyToRun()

```
void Scheduler::ReadyToRun(Thread *thread) {
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

輸入:執行緒的位址。

1. 將將thread的狀態設為ready
2. 將thread放盡 ready queue

當一個執行緒被創建並呼叫 Fork 時，會通過 ReadyToRun 加入Ready Queue。

4-1 Scheduler::Run()

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
   Thread *oldThread = kernel->currentThread;
   if (finishing) {  // mark that we need to delete current thread
       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
   kernel->currentThread = nextThread;  // switch to the next thread
   nextThread->setStatus(RUNNING);      // nextThread is now running
   SWITCH(oldThread, nextThread);
   CheckToBeDestroyed();  // check if thread we were running
   if (oldThread->space != NULL) {     // if there is an address space
       oldThread->RestoreUserState();  // to restore, do it.
       oldThread->space->RestoreState();
   }
}
```

如果舊的thread將結束，將舊的thread刪除。

1. 將舊的CPU state暫存。
2. 若舊的thread state不是 FINISHED，則將其狀態設為 BLOCKED 或 READY，並保存 CPU 狀態
3. 執行context switch
4. 回覆old thread（如果舊的已執行完被刪除，則不需要回復 nextThread->RestoreCPUState();）

Implementation Explanation

1. threads/kernel.h

```
int execRunningNum;  // number of running threads
bool usedFrames[NumPhysPages];
int hostName;  // machine identifier
```

在Kernel新增usedFrames陣列，存取各個Frame的狀態（被使用或是沒被使用。）

2. userprog/addrspace.cc

```
int frameIndex{0}, freeFrameNumber{0};
    for (int i = 0;i < NumPhysPages && freeFrameNumber < numPages;i++){
        freeFrameNumber += !kernel->usedFrames[i];
    }
```

紀錄沒被使用的Frame總數。

```
    if (freeFrameNumber < numPages){
        ExceptionHandler(MemoryLimitException);
    }
```

如果沒被使用的比需要用到的少，那就MemoryLimitException。

```
    for (int i = 0;i < numPages;i++){
        while (kernel->usedFrames[frameIndex] && frameIndex < NumPhysPages){
            frameIndex++;
        }
        kernel->usedFrames[frameIndex] = TRUE;
        pageTable[i].physicalPage = frameIndex;
        pageTable[i].virtualPage = i;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
```

確認有足夠的Frame後迭代每個Frame，利用Kernel->usedFrames找到可以用的Frame後，將其初始化並寫入pageTable。

```
if (noffH.initData.size <= 0){noffH.code.virtualAddr = 0;}
    int pageNum = noffH.code.virtualAddr / PageSize;
    int Offset = noffH.code.virtualAddr % PageSize;
    int physicalPage = pageTable[pageNum].physicalPage * PageSize + Offset;
```

寫入Memory時，透過pageTable將pageNum轉換，得到physicalAddress

```
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[physicalPage]),
            noffH.code.size, noffH.code.inFileAddr);
    }
```

ReadAt將檔案讀取進mainMemory。

這部分的code只包含code segment，除了code部分之外，對initData，readonlyData都做一樣的處理。

```
AddrSpace::~AddrSpace() {
   for (int i = 0;i < numPages;i++){
       kernel->usedFrames[pageTable[i].physicalPage] = FALSE;
   }
   delete pageTabl
e;
}
```

當刪除AddrSpace時，將使用到的Frame全部紀錄為未使用。

3. **userprog/addrspace.cc**

```
enum ExceptionType { NoException,            // Everything ok!
                     SyscallException,       // A program executed a system
                     PageFaultException,     // No valid translation found
                     ReadOnlyException,      // Write attempted to page
                     BusErrorException,
                     AddressErrorException,  // Unaligned reference or one
                     OverflowException,      // Integer overflow in add or
                     IllegalInstrException,  // Unimplemented or reserved
                     MemoryLimitException,
                     NumExceptionTypes
};
```

新增MemoryLimitException。

Report Question Explanations

4. **How does Nachos allocate the memory space for a new thread(process)?**

   NachOS透過Kernel::Execx()來創建一個新的thread，並在其中使用t->space = new AddrSpace(); 來分配memory space。

5. **How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?**

   NachOS先生成一個Thread，分配空間給AddrSpace連到thread，呼叫Fork時分配Stack的空間，並且在func ptr傳入ForkExecute，接著 ForkExecute會呼叫AddrSpace Load跟Execute，Load會將file輸入MainMemory，而Execute會設置當前thread的memory space，初始化reg，並通過 kernel->machine->Run() 開始執行user program。

6. **How does Nachos create and manage the page table?**

NachOS呼叫AddrSpace::AddrSpace()會分配一個pageTable空間，並且初始化裡面的值。AddrSpace::Load會計算所需的pageNumber看usedFrame裏面是否還有frame未被使用，然後分配這些空間給程式並且紀錄在PageTable上的entry.physicalAddress。AddrSpace::~AddrSpace() 釋放空間時，先依照PageTable有記錄的entry將usedFrame[i]更新成false。

7. How does Nachos translate addresses?

NachOS使用machine/translate.cc Translate函數確認是否合法並且轉換。

```
Machine::Translate(int virtAddr, int *physAddr, int size, bool writing)
```

流程如下：

1. 將virtAddr分解為虛擬頁面號(VPN)和頁內偏移量(offset)。
2. 通過 VPN 查詢 pageTable，獲取對應的物理頁面號(PFN)。
3. 使用公式 physicalAddress = PFN * PageSize + offset 計算physAddr。
4. 驗證該page 是否valid。若為invalid，則返回對應的exeption(如 PageFaultException 或 ReadOnlyException等等)。

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

NachOS在Threadg生成時完成初始化，machineState就是thread的registers。

```
Thread::Thread(char *threadName, int threadID) {
    ID = threadID;
    name = threadName;
    isExec = false;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
        machineState[i] = NULL;  // not strictly necessary, since
    }
    space = NULL;
}
```

9. Which object in Nachos acts the role of process control block

Thread 這個class本身會紀錄PCB的所有資訊，所以Thread這個class是PCB。

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

Thread會呼叫Fork, 這時Fork會先把資料讀取到stack裡(`StackAllocate`), 其中的ReadyToRun函數會將thread丟到scheduler的ReadyToRun Queue, 接著排程器會逐一從 Ready Queue中取出thread, 呼叫 Scheduler::Run 切換到當前thread並執行。