OS MP2 Report

Team Member:

110062204 呂宜嫻: implement coding部分

110062209 侯茹文: report trace code部分、report question部分

Trace Code Part

threads/thread.cc

- **Thread::Sleep()**

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

    ASSERT(this == kernel->currentThread); //確保kernel 中的thread 是this(現在這個)
    ASSERT(kernel->interrupt->getLevel() == IntOff); //透過getLevel()獲取現在的
interrupt狀態，而IntOff是用來確認是不是"現在所有interrupt"都被禁用
    // IntOff也是最高級別的interrupt，為了確保某些狀態下不會被interrupt影響


    status = BLOCKED; //將status設為block(waiting)

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) { //判斷接下
來有沒有要跑的
        kernel->interrupt->Idle();//如果暫時沒有要跑的，就進入idle狀態
    }
    kernel->scheduler->Run(nextThread, finishing);  //run接下來的thread
}
```

> 當現在的thread結束工作或是在等待時，會進入sleep的狀態。
>
> 並判斷接下來是否有新的thread要執行，若是無就在idle狀態，而若是此時有新的thread，就會離開sleep的狀態，繼續執行下一個thread。

- **Thread::StackAllocate()**

```
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int)); //分配空間給stread

#ifdef PARISC
```

```
    stackTop = stack + 16;  //由stack定義stacktop的位置
    stack[StackSize - 1] = STACK_FENCEPOST; //用來檢測是否溢出
#endif

... //基於不同的系統定義stackTop

#ifdef PARISC  //儲存不同數據的地址，並根據系統不同有不同的操作
    machineState[PCState] = PLabelToAddr(ThreadRoot); //first frame
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin) //current thread的起
始位置
    machineState[InitialPCState] = PLabelToAddr(func); //func 是要被fork的函數
    machineState[InitialArgState] = arg; //函數中會需要的參數
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish); //current thread的
結束位置
#else //幾乎同樣 只是因為不同架構分開定義
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

> 為了現在的new thread分配空間。先利用AllocBoundedArray()獲取空間，並計算stackTop(空間的boundary)。
>
> 而後根據現在的current thread初始化machine state

- **Thread::Finish()**

```
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff); //因為要進入sleep，需要確認現在的
interrupt會被禁止(設成IntOff)
    ASSERT(this == kernel->currentThread); //確認kernel的currentThread是自己(本身的
thread)

    Sleep(TRUE); //呼叫sleep
}
```

> 這一部分是當現在的thread要結束時，會先將現在的interrupt禁止(狀態設為IntOff)。
>
> 而我們無法一開始就直接de-allocoate，因為我們"正在"這個thread上，這樣我們現在的行為也會消失；因此我們需要利用scheduler 呼叫 destructor。也就是透過後續的sleep()到run()來處理這件事。
>
> 接下來會進入sleep()，傳入true代表現在的thread已經執行完畢，之後在Run()不需要再另外存現在的狀態，並直接透過switch切換到下一個thread。

- **Thread::Fork()**

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

    StackAllocate(func, arg); // call StackAllocate() 來初始化一些項目

    oldLevel = interrupt->SetLevel(IntOff); //oldlevel儲存了現在的interrupt
status，並將現在的status設為IntOff
    scheduler->ReadyToRun(this); //call ReadyToRun() 準備可以運作

    (void) interrupt->SetLevel(oldLevel); //跑完可以回到原本的status了
}
```

> fork一個新的thread，並進行一些初始化的操作以及利用StackAllocate()分配空間。
>
> 並且將interrupt狀態儲存後設為IntOff，以防後續操作(如switch)被interrupt影響。而後進入ReadyToRun()，會將此thread放入ready queue，並將其狀態設為ready，代表準備可以跑了。
>
> 結束後回到此function時，就可以將interrupt狀態設回來了。

---

userprog/addrspace.cc

- **AddrSpace::AddrSpace()**

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages]; // 一個對應virtual page 以及
physical page的table
    for (int i = 0; i < NumPhysPages; i++) { //y在page table中設定資料為初始狀態
        pageTable[i].virtualPage = i;   // for now, virt page # = phys page # =>
數字是一樣的
        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); //清空memory
}
```

> 建立並對page table設一些初始狀態，且因為一開始是沒有muti-programming的，pageTable 涵蓋整個 physical memory，因此不需要做任何 virtualPage 和 physicalPage 之間的映射。
>
> bzero將memory清空，用以運行thread。

- **AddrSpace::Execute()**

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

    kernel->currentThread->space = this; //將現在kernal->currentThread->space改為
this(現在addrSpace的)，這樣kernal才會跑現在這個threads

    this->InitRegisters(); //初始化reg的值， 會先把所有reg都存入0
    this->RestoreState(); //讓kernal讀取到現在的page table以及numpages

    kernel->machine->Run(); //讓machine模擬執行用戶程序，轉到user mode

    ASSERTNOTREACHED(); //因為run()不會回傳所以不應該跑到這
}
```

> 準備要運作現在的threads，並初始化reg、讀取需要資料，並轉到user mode執行。

- **AddrSpace::Load()**

```
AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName); //開啟對應檔案
    NoffHeader noffH; //noffH 存有一些user program的資料(可執行的code 、data)
    unsigned int size;

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

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0); //讀取file中的noffH訊息

    if ((noffH.noffMagic != NOFFMAGIC) &&  //檢查magic number(用來確認檔案格式)
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH); //轉換後若還是不同->檔案不符
    ASSERT(noffH.noffMagic == NOFFMAGIC); //檢查檔案格式是否相符

#ifdef RDATA //先計算所需的空間大小(size)，而後依據size計算需要幾個page (pageSize)

    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;   //需另處理read only data

#else

    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
            + UserStackSize;

#endif
```

```
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages); //確認所需的大小<總大小


    if (noffH.code.size > 0) { //將code內容讀到main memory的 virtual address的部分
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) { //將data內容讀取到main memory的virtual address的
部分
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) { //RDATA還需要另外處理read only data的部分，一
樣讀取進main memory virtual address的部分

        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

    delete executable;  //關閉檔案
    return TRUE; //成功運行並回傳
}
```

> 目標是load檔案。首先先計算檔案大小(size、numPages)，而後再將user programu以及data(從"filename"檔案中) load into memory。

---

threads/kernel.cc

- **Kernel::Kernel()**

```
Kernel::Kernel(int argc, char **argv)
{
    randomSlice = FALSE;  //randomSlice: 控制context switch的時間，設為false時 任務
所占用時間一樣 隔相同的一段時間會進行一次context switch；設為true的話 每個任務會被分配到
的時間不一定一樣
    debugUserProg = FALSE; //debugUserProg 是用來控制現在指令運作狀況，開啟之後可以一
步一步地進行 比較好觀測&debug
    consoleIn = NULL; //從哪裡讀取input 設為初始值(=NULL的話 默認從鍵盤讀取input)
    consoleOut = NULL; //檔案要output到哪 設為初始值(=NULL的話 默認顯示在terminal or
控制台)
```

```
#ifndef FILESYS_STUB
    formatFlag = FALSE; //= true, disk就會被格式化
#endif
    reliability = 1; //網路通訊的可靠性，1是可靠的，確保信息不會被丟失
    hostName = 0; //unix sucket 的編號，每個process都有屬於自己的編號

    //根據不同的指令來針對上述內容做不同的設定，而arg**的部分就如上述提到的，將不同的指令壓縮在一個pointer內
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
            ASSERT(i + 1 < argc); //確保(i + 1)存在
            RandomInit(atoi(argv[i + 1])); //隨機數生成

            randomSlice = TRUE;
            i++;
        } else if (strcmp(argv[i], "-s") == 0) {
            debugUserProg = TRUE;
        } else if (strcmp(argv[i], "-e") == 0) {
            execfile[++execfileNum]= argv[++i]; //記錄需要執行的程式到 execfile
            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);

            reliability = atof(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-m") == 0) {
            ASSERT(i + 1 < argc);

            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";
        }
    }
}
```

> 獲取並解析指令，並執行其要求，將相對應的參數設置完成、儲存需要執行的程式。

- **Kernel::ExecAll()**

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

> 逐行執行execfile[i]中的程式，並在結束時call Finish()。

- **Kernel::Exec()**

```
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;
}
```

> 利用name以及threadID為目標程式建立一個新的thread，並存入陣列t。先進入AddrSpace()設定page的初始狀態，並進入fork進行後續的運作(ForkExecute)，最後回傳一個threads的編號。
>
> 註解掉的部分是同時創造多個threads的，作業才會用到，所以我這裡先刪除。

- **Kernel::ForkExecute()**

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) { //load 檔案
        return;
    }

    t->space->Execute(t->getName()); //執行檔案

}
```

> 執行被fork出來的部分，會先load檔案後再執行該檔案。如果找不到檔案就會直接回傳。

threads/scheduler.cc

- **Scheduler::ReadyToRun()**

```
Scheduler::ReadyToRun (Thread *thread)
{
   ASSERT(kernel->interrupt->getLevel() == IntOff); //確保不會被interrupt

   thread->setStatus(READY); //將status設為ready，準備進入ready queue

   readyList->Append(thread); //將現在的threads放入ready queue
}
```

> thread是可以進入ready queue的狀態了，將狀態設為ready並放入ready queue。

- **Scheduler::Run()**

```
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

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

    if (finishing) { //如果上一個threads(currentThreads)已經結束，就可以destroy (這是上面finish提到的要請cpu call destroy) 然後會在CheckToBeDestroyed()中被發現然後刪掉

         ASSERT(toBeDestroyed == NULL);
     toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) { //存現在threads的狀態，執行完新的以便換回來
        oldThread->SaveUserState();
    oldThread->space->SaveState();
    }

    oldThread->CheckOverflow(); //check stack overflow (stack overflow: 是否把threads固定的stack空間用完了)

    kernel->currentThread = nextThread; //換成run下一個threads
    nextThread->setStatus(RUNNING);


    SWITCH(oldThread, nextThread); //switch到下一個thread，跑完後回到原本的

    ASSERT(kernel->interrupt->getLevel() == IntOff); //跑switch的threads是不能interrupt的

    CheckToBeDestroyed(); //看是否有跑完的threads要被刪掉
```

```
    if (oldThread->space != NULL) { //恢復原本狀態 除非threads已經被destroy
        oldThread->RestoreUserState();
    oldThread->space->RestoreState();
    }
}
```

> 暫時停止執行現在的thread(或是現在的thread已經結束了)，並switch到下一個thread運作，然後回來。

Question Part

因為這次問題跟trace code順序有點不一樣，不太確定怎麼解釋比較好所以就把問題拿出來單獨回答了。

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

> 會在Thread::StackAllocate()中，為即將執行的thread分配空間並且計算出可使用的界線(stackTop)。

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

> 在AddrSpace::Load()中在open file後會讀取其noffH訊息，其中就包括user的程式、以及initial data。

- **How does Nachos create and manage the page table?**

> 它在AddrSpace::AddrSpace()中，創造了空的page table，放入對應的virtual page以及physical page (在non-mutiprogramming情況下 會是一樣的number)。並且在其中放入初始的一些數值。

- **How does Nachos translate addresses?**

> AddrSpace::Translate()會使virtual address轉為physical address。

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

> 在AddrSpace::Execute()中，有call到InitRegister()，就可以將reg進行初始化的動作。
>
> 在 Scheduler::Run() 中，從其他 thread 回到 oldThread 時，(如果未結束) 會進行 RestoreState 和 RestoreUserState。

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

> Thread 扮演了PCB的腳色，在thread.h中存有ID、name、thread status以及user registers。

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

> 當我們要運作execfile們時，會先由execAll()，然後進入exec()針對每個execfile產生一個單獨的thread, 並對其page table初始化後會進入fork()。
>
> 在fork call完StackAllocate()後，將其狀態設為IntOff後進入ReadyToRun()。
>
> 在ReadyToRun()中，會將現在的thread狀態設為Ready，並將其放入readylist。

- **According to the code above, please explain under what circumstances an error will occur if the message size is larger than one page and why? (Hint: Consider the relationship between physical**

pages and virtual pages.)

> 當 message size 大於 page size 時，如果 page 不連續或分配到的 page 不足，則可能造成 error。

Implement Part

- **AddrSpace::~AddrSpace()**

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

> 在一個 thread 結束時將原本使用的 page 空間釋出。

- **AddrSpace::Load()**

```
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

pageTable = new TranslationEntry[numPages];
for (int i = 0, j = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;
    while (kernel->usedPhysPages[j] && j < NumPhysPages) j++;
    if (kernel->usedPhysPages[j]) {
        // no page is available
        ExceptionHandler(MemoryLimitException);
    }
    kernel->usedPhysPages[j] = TRUE;
    pageTable[i].physicalPage = j;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    // cout << j << ' ';
}
```

> 將 pageTable 的初始化移到 AddrSpace::Load() 中實作，即可依 code file 大小來決定一個 thread 需要多少 page。若所有 page 皆已被使用，則呼叫 Exception Handler。

```
if (noffH.code.size > 0) {
        int codePhysicPage =
pageTable[noffH.code.virtualAddr/PageSize].physicalPage*PageSize;
        int codePhysicOffset = noffH.code.virtualAddr%PageSize;
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[codePhysicPage + codePhysicOffset]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        int initPhysicPage =
pageTable[noffH.initData.virtualAddr/PageSize].physicalPage*PageSize;
        int initPhysicOffset = noffH.initData.virtualAddr%PageSize;
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[initPhysicPage + initPhysicOffset]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }
```

> 並依照 code 和 initData 的 virtual address 以及被分配到的 page 決定其在 memory 中的 address。

- **kernel.h**

```
bool usedPhysPages[NumPhysPages]; // record used physical page
```

> 新增一個可以記錄 Physical Page 是否正在被使用的 array。

- **machine.h**

```
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.
     MemoryLimitException,
     NumExceptionTypes
};
```

> 新增 MemoryLimitException。