NACHOS-4.0

MP2

第 21 組

106062173 吳東弦(一起 TRACE CODE,負責整理 REPORT)

106062271 郭子豪(一起 TRACE CODE,負責實作 CODING)

一、 trace code

由於 Kernel::ExecAll()是被 main()呼叫的，所以直接從 main 開始說明：此外，由與 nachos 中每一個程序都只有一執行緒，故以下的執行緒可視為程序。

1. main(int argc, char ** argv)

main 是 nachos 的進入點，模擬機器開機時的狀況。先啟動作業系統（kernel(argc, argv)），將要執行的檔名（consoleIO_test1 與 consoleIO_test2）與數量（execfileNum）存起來；接著初始化作業系統（kernel->Initialize()），執行「初始」執行緒（Thread("main")），啟動中斷器（Interrupt()）、排程器（scheduler）、計時器（alarm）與檔案系統（fileSystem）等等；最後就開始準備要執行的程式（ExecAll()）。

2. Kernel::ExecAll()

在 ExecAll()中，先跑一個迴圈（Exec(name)）將要執行之程式的分頁表（AddrSpace()）建立好；接著利用「初始」執行緒生出（Fork(&ForkExecute)）子執行緒（Thread(name)），其中分配了所需的記憶體空間（StackAllocate()），順便處理執行緒在實體機器中要被執行（ThreadRoot()）時暫存器的指向（machineState[]）；最後準備將任務結束的「初始」執行緒結束，並切入下個執行緒（Finish()），也就是之前要執行的第一個程式（consoleIO_test1）。

3. Thread::Finish ()

在 Finish()中將結束執行緒的訊號（TRUE）傳給了 Sleep()。

4. Thread::Sleep (bool finishing)

在 Sleep()中，要準備切換執行緒。如果沒有可以切換（FindNextToRun() = = NULL）的執行緒，系統會進入閒置狀態（Idle()），順便監看是否有中斷訊號被發出（CheckIfDue(TRUE)）;如果有執行緒需要被切換，系統就會將下個執行緒加入執行隊列，同時將當前執行緒是否要被結束的訊號傳遞下去（Run(nextThread, finishing)）。

5. Scheduler::Run (Thread *nextThread, bool finishing)

在 Run()中，開始進行執行緒切換（SWITCH()），如果要被切出的執行緒已經執行完畢，就將之結束掉（CheckToBeDestroyed()），如果還沒做完，就把暫存器狀態與分頁表狀態存回「虛擬機」記憶體中。

6. SWITCH(Thread *oldThread, Thread *newThread)

由於執行緒的切換不只要在虛擬機模擬，也得在實體機上處理，所以這個函數被宣告後，並非由 C++代碼定義，而是由組合語言直接對實體機器的暫存器進行操作。在 switch.S 這份組語中，可發現針對不同的指令集有不同的實作，SWITCH()中對稱的組語前半表示當前執行緒被切出，後半表示新的執行緒被切入，當實體機的執行緒切換完成後，暫存器會指向 ThreadRoot()。

7. ThreadRoot()

在 ThreadRoot()中，連續呼叫了三個函數，StartupPCState、InitialPCState、WhenDonePCState，由第二點的 StackAllocate()可知，分別對應 ThreadBegin()、ForkExecute()、ThreadFinish()，表示執行緒真正被啟動、執行、結束的生命週期，其中 ThreadBegin()中也會檢查要結束的執行緒（CheckToBeDestroyed()），如果有就會直接結束掉該執行緒，接著解釋執行緒如何被執行。

8. ForkExecute(Thread *t)

在 ForkExecute()中，consoleIO_test1 的執行緒開始真正被載入（Load()）虛擬機之前所分配好的記憶體中，其中包含了程式代碼（ReadAt(code)）、全域變數（ReadAt(initData)），以及可能有的唯讀資料（ReadAt(readonlyData)），接著，將暫存器（PCReg、NextPCReg、StackReg）內容更新（InitRegisters()）；最後更新虛擬機所指向的分頁表（RestoreState()）。載入完成後，就可以讓虛擬機繼續跑下去了（Run()）。

9. Machine::Run()

在 Run 中()，首先取得指令進行解譯、執行（OneInstruction()），其中，如果有對記憶體進行讀寫（ReadMem/WriteMem），就得將邏輯記憶體轉成實體記憶體（Translate(virtAddr, physAddr)），CPU 會先去 TLB 看，找不到才去找分頁表，取得正確的位址；接著，每執行一個指令，就會遞進一次系統時間（OneTick()）。

10. Interrupt::OneTick()

在 OneTick()，除了偵測是否有中斷發生（CheckIfDue(FALSE)）外，還會去偵測是否需要做執行緒切換（yieldOnReturn）。當要進行切換就會去呼叫 Yield ()。

11. Thread::Yield ()

在 Yield()中，先去取得下一個要被切換的執行緒（FindNextToRun()），接著對系統進行執行前準備（ReadyToRun()），最後就可以將此執行緒加入執行排程了（Run(nextThread, FALSE)）。由此可知，流程會再度回到第五點，只要兩個執行緒沒有結束，就會不斷切換，週而復始。

12. Alarm::CallBack()

另外，過程中我們發現 yieldOnReturn 是由 Interrupt::YieldOnReturn()進行賦值，而 Alarm::CallBack()則呼叫了前述函式，從註解中可知，這個 callback 每隔一個 TimerTicks（100）就會被呼叫，正表現出了處理器做 time sharing 的特性。

二、 Implement page table in NachOS

1. addrspace.cc - AddrSpace::AddrSpace()

在 AddrSpace()中，我們可以發現原本的代碼在為執行緒分配記憶體時，會把所有實體空間都給它（physicalPage = i），並給予存取權（valid = TRUE），同時將整塊記憶體清空（bzero(MemorySize)），這樣導致後面的執行緒會直接覆蓋掉前面所擁有的位址，無法實現 Multi-programming。為了避免這個情況，我們將「取得實體記憶體」、「給予存取權」、「清空記憶體」都註解掉，等後面載入前（Load()）已知所需空間大小時，再取所需記憶體。

```
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 = FALSE;
        ////////////////////////
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
```

2. kernel.h - class recording_physical_table

```
class recording_physical_table {
  private:
    int table[NumPhysPages];
  public:
    recording_physical_table() {
        for (int i = 0; i < NumPhysPages; i++) {
            table[i] = -1;
        }
    }
    ~recording_physical_table() {
        delete table;
    }
```

```
    int get_unuse_physical_page(int thread_ID) {
        int i;
        for (i = 0; i < NumPhysPages; i++) {
            if( table[i] == -1) {
                table[i] = thread_ID;
                return i;
            }
        }
        cerr << "havent enough page" <<endl;
        abort();
    }
    void return_use_physical_page(int thread_ID) {
        for (int i = 0; i < NumPhysPages; i++) {
            if( table[i] == thread_ID) {
                table[i] = -1;
            }
        }
        return;
    }
};
```

在這個 class 中，我們實作了判斷記憶體是否已被使用的功能。建構子會初始化一張與記憶體一一對應的表格，用來記錄頁框使用情形。它在系統初始化時被實體化（Kernel::Initialize()），在系統關掉時釋放（Kernel::~Kernel()）。

```
recording_physical_table() {
    for (int i = 0; i < NumPhysPages; i++) {
        table[i] = -1;
    }
}
```

```
~recording_physical_table() {
    delete table;
}
```

```
void
Kernel::Initialize() {
    // We didn't explicitly allocate the current thread we are running in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.


    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);
    ////////////////////////////////////
    record = new recording_physical_table();
    ////////////////////////////////////
    interrupt->Enable();
}
```

```
Kernel::~Kernel() {
    delete stats;
    delete interrupt;
    delete scheduler;
    delete alarm;
    delete machine;
    delete synchConsoleIn;
    delete synchConsoleOut;
    delete synchDisk;
    delete fileSystem;
    delete postOfficeIn;
    delete postOfficeOut;
    ////////////////
    delete record;
    ////////////////
    Exit(0);
}
```

當執行緒要取得記憶體空間時，會呼叫 get_unuse_physical_page()在紀錄表中找到閒置的頁框，作為之後載入使用；當執行緒要釋放記憶體空間時，則會呼叫 return_use_physical_page()將紀錄表重設為初始值。

```
int get_unuse_physical_page(int thread_ID) {
    int i;
    for (i = 0; i < NumPhysPages; i++) {
        if( table[i] == -1) {
            table[i] = thread_ID;
            return i;
        }
    }
    cerr << "havent enough page" <<endl;
    abort();
}
```

```
void return_use_physical_page(int thread_ID) {
    for (int i = 0; i < NumPhysPages; i++) {
        if( table[i] == thread_ID) {
            table[i] = -1;
        }
    }
    return;
}
```

3. AddrSpace::Load(char *fileName)

在 Load()中，我們已經知道執行緒所需頁框數量（numPages），在這裡可以執行第一點所註解掉的三個部分，分別為：

取得實體記憶體（physicalPage = get_unuse_physical_page(ID)）

給予存取權（valid = TRUE）

清空記憶體（bzero(Memory[pageTable[i].physicalPage * PageSize], PageSize)）

```
int ID = kernel->currentThread->getID();
for (int i = 0; i < numPages; i++) {
    pageTable[i].physicalPage = kernel->record->get_unuse_physical_page(ID);
    pageTable[i].valid = TRUE;
    bzero(&kernel->machine->mainMemory[pageTable[i].physicalPage * PageSize], PageSize);
}
```

接著要將執行緒的代碼、全域資料分別載入頁框，注意 ReadAt()第一個參數的實體位址中，前後兩項分別為：

頁框起始點（Memory[pageTable[virtualAddr/PageSize].physicalPage*PageSize）

頁框位移值（virtualAddr%PageSize）

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

        noffH.initData.size, noffH.initData.inFileAddr);
}
```

三、 Summary & FeedBack

由於這次作業的 spec 只提到兩個函數，為了了解整個流程，從 main 開始全部 trace 了一遍，困難點大概落在 SWITCH 和 ThreadRoot()的執行流程，這裡是看 gdb 猜出來的，不是非常清楚每個暫存器做了什麼。此外，週一上課時，發現 MP2 有許多部分與 MP3 重疊，算是意外的驚喜。