NachOS – MP2

Multi-Programming

宋體淮, R09921135, Electrical Engineering

1. Trace code

1.1. Explain how NachOS creates a thread(process), load it into memory and place it into scheduling queue

main()：

- 首先在 `main.cc` 中可以看到一開始會創建一個 kernel，並做初始化。
- 做完初始化後，執行 `kernel->ExecAll()` 來跑使用者程式。

```
kernel = new Kernel(argc, argv);

kernel->Initialize();

CallOnUserAbort(Cleanup);

if (threadTestFlag) {
  kernel->ThreadSelfTest();
}
if (consoleTestFlag) {
  kernel->ConsoleTest();
}
if (networkTestFlag) {
  kernel->NetworkTest();
}

kernel->ExecAll();
```

Kernel::Initialize()：

- 將 `currentThread` 設為當前執行的 main thread(主程式也是一條 thread)，並將狀態設為 `RUNNING`。

```
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");
    currentThread->setStatus(RUNNING);
```

Kernel::Kernel()：

- 在 kernel 的建立過程中，會檢查 command line 參數，如果是-e 代表我們要執行某個程式，並把 execfileNum 數量加一，且將其後要執行的程式的 fileName 存入到 execfile 中。

```
else if (strcmp(argv[i], "-e") == 0) {
execfile[++execfileNum]= argv[i];
        cout << execfile[execfileNum] << "\n";
        i++;
}
```

Kernel::ExecAll()：

- 回到 Kernel::ExecAll()，他會從 execfileNum 判斷有多少要執行的程式(或是稱作 thread)，並以 Exec()來一一執行。
- 跑完之後因為 main thread 的功能完成了，因此讓 main thread -> Finish()。

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

Kernel::Exec ()：

- 首先為這個程式創造一條 thread，並給予其定址空間(AddrSpace)。
- 透過 Fork()載入真正要執行的程式，傳入一個 function pointer，&ForkExecute，以及這個 thread 自己，t[threadNum]。

```
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;
}
```

ForkExecute()：

- 這個函式會呼叫 AddrSpace:: Load()來將 thread 載入到記憶體中(待會

會更詳細解釋這部分)，並呼叫 AddrSpace::Execute() 來執行它。

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

AddrSpace::Execute()：

- 在此會將 currentThread 與目前 thread(this)的定址空間拉在一起。
- 初始化 user register，像是設定 program counter 的值為 0，讓機器可以從頭開始執行程式，之後 fetch instruction 就是從 program counter 儲存的地址拿 instruction。
- 把該 thread 的 page table 載入進來，讓 kernel->machine->pageTable 等於現在的 page table。
- 呼叫 kernel->machine->Run() 來開始執行程式，fetch instruction。

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

    kernel->currentThread->space = this;

    this->InitRegisters();
    this->RestoreState();

    kernel->machine->Run();

    ASSERTNOTREACHED();
}
```

Thread::Fork()：

- 回到 Fork()，它會呼叫 StackAllocate() 來為此 thread 安排其 stack 空間。
- 並將此 thread 加進 ReadyToRun 裡面(就是在等待 CPU 的 ready queue)。

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

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

Thread:: finish()：

- 回到 Thread:: finish()，當所有程式都執行完之後，就要把 kernel thread 結束掉，所以他會先確認現在這個 thread 是不是 kernel thread，是的話才會呼叫 `Sleep()`。

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

Thread:: Sleep()：

- 首先把自己的狀態設為 BLOCKED，接著去 `kernel->scheduler` 看是否有下個要執行的 thread。
- 若有的話，就繼續往下跑；若沒有的話，進入 idle Mode，此時會判斷是否沒有任何 interrupt 跟 thread 要執行了，若無，整個 NachOS 運作結束 (Halt)。

```
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);
}
```

- 所以統整一下 NachOS 執行一個程式的流程：
  - 為該程式建立一條 thread，並給於其定址空間 AddrSpace。
  - 藉由 `Fork()` 來將程式載入到該 thread 當中(也是記憶體中)。
  - `Fork()` 會先接收到一個 function pointer(到時候要執行的程式)，先初始化一些 thread 的 stack，透過 `machineState[InitialPCState] = (void*)func;`，讓原先的 function pointer 成為未來 program counter 要執行的程式。

— 此時 thread 大致初始完畢，接著就將剛剛的 thread 放入 ready queue，將來準備讓 CPU 執行。

— CPU scheduler 未來會從 ready queue 中選取某條 thread，並讀取其 Program counter 的值

---

2. Implement page table to make NachOS support multi-programming.

- 首先觀察原本錯誤的輸出，兩個程式有互相干擾的狀況，所以很明顯是兩個程式匯入時，操作到相同區域的記憶體區段。

```
henry@henry-VirtualBox:~/nachos/nachos-4.0/code/userprog$ ./nachos -e ../test/test1 -e ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
Print integer:7
Print integer:8
Print integer:9
Print integer:10
Print integer:12
Print integer:13
Print integer:14
Print integer:15
Print integer:16
kPrint integer:16
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:21
Print integer:21
Print integer:23
Print integer:24
Print integer:25
return value:0
Print integer:26
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

AddrSpace::AddrSpace()：

- 接著來看到這個函式，可以知道原本的 page table 設定成跟整個記憶體空間(`NumPhysPages`)一樣大，且 virtual address 跟 physical address 是一對一的關係，因此所有程式的 address space 都是共用的，也才會有互相干擾的狀

況。

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
        pageTable[i].virtualPage = i;
        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);
}
```

AddrSpace.h：

- 所以我們在 Addrsapce 這個 class 裡面多宣告一個共享變數(使用 static)，紀錄被使用過的 physical pages，usedPhyPage[NumPhysPages]。

```
class AddrSpace {
  public:
    AddrSpace();                        // Create an address space.
    ~AddrSpace();                       // De-allocate an address space

    void Execute(char *fileName);       // Run the the program
                                        // stored in the file "executable"

    void SaveState();                   // Save/restore address space-specific
    void RestoreState();                // info on a context switch

    static bool usedPhyPage[NumPhysPages];
```

AddrSpace.cc：

- 接著進到 AddrSpace.cc，首先將剛才的 usedPhyPage 在一開始都設定為還沒有被使用中。

```
bool AddrSpace::usedPhyPage[NumPhysPages] = {0};
```

AddrSpace::Load()：

- 首先可以看到會使用 kernel->fileSystem->Open(fileName)來開啟要執行的程式，並用一個指標 executable 指向它。

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

noff.h()：

- 在 AddrSpace::Load()創建了一個 noffH 的資料結構，該結構的定義在 noff.h 中被定義。
- 一個 address space 由 code、initData、unintData 所組成，所以這個資料結構是在記錄這些 segment 的 virtual address 或是 size 等等。

```
typedef struct segment {
  int virtualAddr;              /* location of segment in virt addr space */
  int inFileAddr;               /* location of segment in this file */
  int size;                     /* size of segment */
} Segment;

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

AddrSpace::Load()：

- 回到 AddrSpace::Load()，接著會利用 ReadAt()函式來將程式資訊先讀進 noffH 中(但不知道他到底怎麼讀到 virtual address 或 size 這些資訊)，ReadAt()其實是呼叫 C library 裡面的函式(可參考 filesys/openfile.h)
- noffH 在此做為一個中繼的 buffer，希望利用它去計算這個程式的所需的大小，就是 code ＋initData ＋unintData 的總和(還有我們自己定義的 stack 大小)，因此最後可以知道要用多少的 pages。

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

// 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
    numPages = divRoundUp(size, PageSize);
//      cout << "number of pages of " << fileName<< " is "<<numPages<<endl;
```

- 接著開始來建立 page table，首先創建一個 numPages 大小的 page table。
- 以 j 當作 index 來依序尋找現在有哪些 frame 是未使用狀態，如果沒被使

用才可以指派給 page table。

```
pageTable = new TranslationEntry[numPages];
for (unsigned int i = 0, j = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;
    while(j < NumPhysPages && usedPhyPage[j] == TRUE)  // find first not-used phys. page
        j++;
    usedPhyPage[j] = TRUE;
    pageTable[i].physicalPage = j;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}

size = numPages * PageSize;
```

- 指派好 virtual page 與 physical page 的關係之後，就要真的來把程式載入到 nachos 系統裡的 main memory 了。
- 這裡同樣利用 `ReadAt()` 函式來將程式的 `code.virtualAddr`（應該是 code segment 的起始點）以及 `initData.virtualAddr` 讀到 `kernel->machine->mainMemory`。就是這一步讓我們把 compile 後的 user program 放進 nachos 的 memory。
- 我們知道 base + offset = main memory address，而 base 的算法就是先看是第幾個 page 然後再乘以 `PageSize`，而 offset 則是 `virtualAddr` mod `PageSize`。
- 注意到這邊是 `ReadAt` 整塊 code section 的大小或是整塊 initData section 的大小，所以其實並沒有真的切成 page (MP4 才有)，因此可以想像第一個被 load 進來的程式擁有一整塊連續的 memory，而第二個 load 進來的程式一樣也是擁有另一塊連續的 memory，我們現在有讓他們錯開而不再是共用同一塊 memory。

```
// then, copy in the code and data segments into memory
       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);
   }
```

Result：

- 重新編譯之後並執行，可以看到輸出正確的結果，兩隻程式並不互相干擾，達成 multi-programming。

```
henry@henry-VirtualBox:~/nachos/nachos-4.0/code/userprog$ ./nachos -e ../test/test1 -e ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```