

## NachOS執行program的流程：

NachOS執行的起點在main.cc的main()，NachOS從terminal讀入的參數，初始化一些數據，建立一個kernel，並對kernel做一些測試，之後就執行kernel該執行的動作，以下為main中最重要的三個地方，接下來會一一trace這三個function(都在kernel.cc中)。

```
kernel = new Kernel(argc, argv);  
  
kernel->Initialize();
```

```
kernel->ExecAll();
```

**1.kernel的建構式：**在建構式中，會依據main()傳入的argv參數來設定NachOS，像是這兩次的project都是輸入 -e 選項，kernel會做以下動作，將所有要執行的檔名都會儲存在execfile[]陣列裡。

```
debuggerProg = TRUE;  
} else if (strcmp(argv[i], "-e") == 0) {  
    execfile[++execfileNum] = argv[++i];
```

**2.Initialize()：**這邊會建構並初始化kernel的各個member，如scheduler,machine...等等都會被new出來，但最重要的是會建立一個main thread，一個沒有AddrSpace的thread(之後會提到一個thread擁有哪些資訊)，並且指派他給kernel的currentThread並且設為RUNNING，意思就是現在kernel在執行main thread，但此thread沒有AddrSpace。

```
currentThread = new Thread("main", threadNum++);  
currentThread->setStatus(RUNNING);
```

**3.ExecAll()**：有兩件重要的事，一個是Exec()另一個是currentThread->Finish()。我們先trace Exec()，之後再來說currentThread。

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

**3.1Exec()**：會拿到要執行的檔名，並為他建立一個Thread，把這個Thread的name設成要執行的檔名，並且給一個數字ID，之後為每個Thread建立一個AddrSpace，最後最重要的就是Fork()。

```
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;
}
```

**Fork()**：利用StackAllocate()設定這個thread的 stack 和 machine State[]，接下來將這個thread放入ready queue裡面(scheduler->ReadyToRun(this))。

```

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);
}

```

**StackAllocate()**：比較底層一點，基本上就是把這個thread的stack allocate好(依據底層機器的不同)，並且設定這個thread的machineState[] (如下圖)，這是為了底層系統的context switch，這邊就大概講述一下：

- 1.使底層系統context switch完會直接執行ThreadRoot(一段跟context switch有關的組語，下面第二張圖)，ThreadRoot基本上就是讓底層系統執行ThreadBegin(), func, ThreadFinish())
- 2.將ThreadBegin()的位址放在machineState[StartupPCState]，ThreadBegin會透過 jal StartupPC被叫到，他會使kernel的currentThread去執行Begin()，這個Begin()會檢查context switch前的那個Thread需不需要被摧毀，需要的話就摧毀，並且enable interrupt。
- 3.func位址放machineState[InitialPCState]上，讓thread會執行到func(透過jal InitialPC)，而func會讀到的參數則是arg(透過move a0, InitialArg)。
- 4.將ThreadFinish()的位址放在machineState[WhenDonePCState]上，透過 jal WhenDonePC，讓Thread執行完func時會執行ThreadFinish()，這樣會讓kernel的currentThread執行Finish()，而Finish()會叫Sleep(True)，等一下會說明Sleep()的動作。

```

machineState[PCState] = (void*)ThreadRoot;
machineState[StartupPCState] = (void*)ThreadBegin;
machineState[InitialPCState] = (void*)func;
machineState[InitialArgState] = (void*)arg;
machineState[WhenDonePCState] = (void*)ThreadFinish;

```

```

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

```

注意這邊的func是從kernel傳來的ForkExecute()的位址

**ForkExecute()**：讓這個Thread的space Load 要執行的code進來，每個thread的name就是要執行的檔名，最後去執行space的Execute()。

```

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

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

```

**Load()**：首先去開啟讀到的檔名，拿到一個OpenFile object，之後將資料讀到noffH裡面(換成NachOS格式)，之後去算這個程式總共需要多少pages，並算出需要多空間放到size，他還會順便檢查這個程式是不是大到NachOS無法執行，最後將整個程式load到NachOS的main memory上(下圖非原版NachOS,為MP2完成版的NachOS)。

```

OpenFile *executable = kernel->fileSystem->Open(fileName);
NoffHeader noffH;
unsigned int size;

```

```

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

```

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

```
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);  
  
    unsigned int paddr;  
    AddrSpace::Translate( noffH.code.virtualAddr, &paddr, 0);  
  
    executable->ReadAt(  
        &(kernel->machine->mainMemory[paddr]),  
        noffH.code.size, noffH.code.inFileAddr);  
}  
if (noffH.initData.size > 0) {  
    DEBUG(dbgAddr, "Initializing data segment.");  
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);  
  
    unsigned int paddr;  
    AddrSpace::Translate( noffH.code.virtualAddr, &paddr, 0);  
  
    executable->ReadAt(  
        &(kernel->machine->mainMemory[paddr]),  
        noffH.initData.size, noffH.initData.inFileAddr);  
}
```

**Execute()**：讓kernel currentThread的space換到這個space，然後InitResgister()將kernel machine的register初始化好，以執行這個thread，RestoreState()則是將kernel machine的page table換成這個thread的page table，最後執行machine Run()來執行程式(無限迴圈不斷抓取instruction)。

```

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 program

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

```

總結一下，Exec()會建立一個thread，把他放入ready queue，並初始化好這個thread的stack，machineState[]，讓底層系統context switch到這個thread就會執行ThreadRoot，之後load程式到memory，並執行machine的Run()抓取instruction。

**3.2currentThread->Finish()**：在建好各個要執行的thread以後，接下來就是要讓現在的currentThread(這邊的currentThread是kernel在初始化時的main thread)結束，把CPU讓給其他thread。

**Finish()**：這邊會直接 call Sleep(True)

```

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

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

```

**Sleep()**：首先去ready queue找下一個要執行的thread，找得到話就把下一個 thread 丟給scheduler 的 Run() 並且標示現在要Sleep的這個Thread是不是要finish，從上面可以知道這邊是要finish main thread所以會是true，而從ready queue裡面拿出來的thread都是Exec()所塞進去的thread。

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

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

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

    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);
}
```

**Run()**：這邊是context switch的地方，首先將kernel正在執行的thread放在oldThread，然後如果要結束old thread就會把它記在toBeDestroyed上等待摧毀，然後將old thread現在執行的user register，page table 資料都存好(SaveUserState(), SaveState())之後讓kernel 的 currentThread 換成新的thread，把thread的狀態設為RUNNING。

每個thread都有2種register set，一個是userRegisters[]用來存放user level register的資料(kernel machine用的register)，另一個是machineState[]存放kernel level的資料(底層系統用的register)。每個thread也各自有一個stack，和space，而space中則是存放page table。

接下來呼叫SWITCH組語來把底層系統的register資料都load成新thread的machineState資料，一換好以後底層系統會直接從ThreadRoot這段組語開始執行(上面Fork()時設定好的)，如同上面StackAllocate()所說的流程一樣，執行



ThreadBegin(), func( 即ForkExecute() → 會load 程式到NachOS memory並透過machine Run()來執行) , 最後則是ThreadFinish()。

ThreadRoot執行結束以後才會return回這裡 , 如果old thread沒有被刪除就換回old thread執行(透過RestoreUserState(), RestoreState())把old thread的register, page table 資料 load 回來 , 並繼續執行。

注意上面提到的main thread會在底層context switch完執行ThreadBegin()時被刪掉。

```
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();
}
```

```
// had an undetected stack overflow
```

```
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
                                // before this one has finished
                                // and needs to be cleaned up

```

```

if (oldThread->space != NULL) {    // if there is an address space
    oldThread->RestoreUserState();  // to restore, do it.
oldThread->space->RestoreState();

```

總結一下，Exec()為要執行的程式建立一個thread，並透過Fork()設定好這個thread的machineState[]，把thread都放到ready queue以後，透過finish main thread來讓scheduler去ready queue找要執行的thread，並進行context switch以後，底層系統因為register都被換成上面Fork()設定好的資料，所以會直接執行ThreadRoot這段組語，一連串的執行到ForkExecute()，把該程式load到NachOS的memory以後，利用machine的Run()來不斷的抓取instruction，這邊NachOS才開始執行user要執行的程式。

## 運行中的Context Switch：

上面提到當一個程式在NachOS執行時，就是用machine的Run()不斷的抓取instruction，這邊每執行完一個instruction就會觸發interrupt的OneTick()

```

kernel->interrupt->SetStatus(0);
for (;;) {
    OneInstruction(instr);
    kernel->interrupt->OneTick();
    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
        Debugger();
}

```

**OneTick()**：這邊我主要還是說明跟context switch有關的部分，其他跟interrupt有關(檢查有沒有interrupt發生)或是增加NachOS紀錄的ticks我就略過。下圖為跟context switch有關的部分，首先是yieldOnReturn若是true就代表timer 要求context switch，timer 倒數為0就會將這個值設為True，接下來就執行Yield進行context switch。

```
if (yieldOnReturn) {    // if the timer device handler asked
                        // for a context switch, ok to do it now
yieldOnReturn = FALSE;
status = SystemMode;    // yield is a kernel routine
kernel->currentThread->Yield();
status = oldStatus;
}
```

**Yield()**：首先去ready queue裡面找到下一個要執行的thread，把現在執行的thread放到ready queue，然後執行scheduler 的 Run()進行context switch，由於old thread並不是finish所以Run()的第二個參數設成False，否則old thread會被刪掉。

```
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);
}
```

這樣就完成了context switch，而下一個thread執行到Run()時，每抓完一次 instruction 都會檢查是不是被timer要求context switch，之後就是不斷的循環。