

2021 OS MP2

TEAM15

108062217 傅詠軒

108062203江浩辰

Contribution:

江浩辰50%

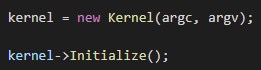
傅詠軒50%

**II.1 trace code**

照著流程走：

**threads/main.cc main**

首先會先傳入command line argument來建立並初始化kernel，接著會先run一些test routines（如果有要求的話）。

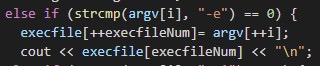


之後透過kernel::ExecAll()開始執行待執行的user program。



**threads/kernel.cc Kernel::Kernel**

main.cc的main首先會new一個kernel，Kernel的constructor會根據傳入的command line arg來決定一些相對應的initialization，由於我們要做的是執行test file，所以讀到的command會是’’-e’’，對應到要做的事就是把要執行的execfile（傳入於argv），放到execfile array內。



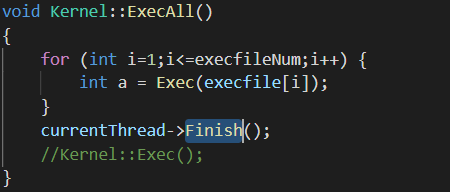
**threads/kernel.cc Kernel::Initialize**

接著main.cc會呼叫kernel->Initialize，會建立名叫 ’’main’’ 的kernel thread，並把狀態設為running，之後把一些data structure初始化，例如stats、ready queue initialize、建立file system等等，最後把interrupt enable。



**threads/kernel.cc Kernel::ExecAll**

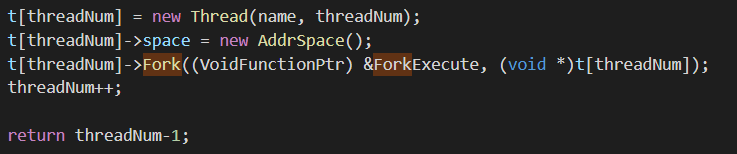
main.cc最後會呼叫kernel->ExecAll()來執行execfile array中要執行的user program，進到ExecAll()中，用for loop呼叫Exec()來一一執行所有user programs，待全部執行完畢後，會呼叫Thread::finish()表示main thread執行完所有execfiles了。



**threads/kernel.cc Kernel::Exec**

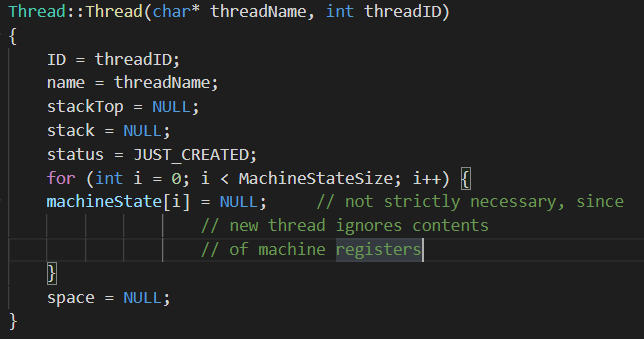
ExecAll呼叫Exec後，由於在NachOS內，thread就等同於process，這邊對每一個user program，都會create一個thread來執行user program，以我們的test檔案為例，我們的command line如下：-e consoleIO\_test1 -e consoleIO\_test2，

我們是要執行兩個user programs，所以建立出來的threads總數會有三個：分別為kernel main thread、thread for consoleIO\_test1和thread for consoleIO\_test2。在這邊透過以下function來allocate一個new thread object、allocate user address space給這個thread，以及call Thread::Fork()來對這個thread做一些設定。



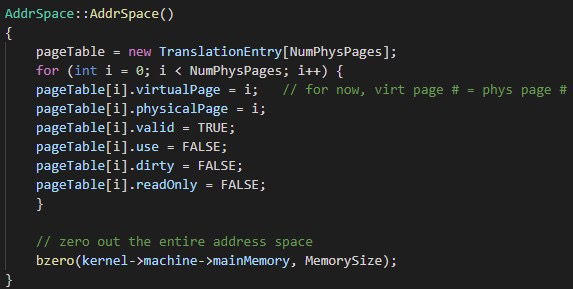
**threads/thread.cc Thread::Thread**

這邊會根據thread name跟ID來allocate一個thread object（也就是一份thread control block）並且initialize thread control block的content。



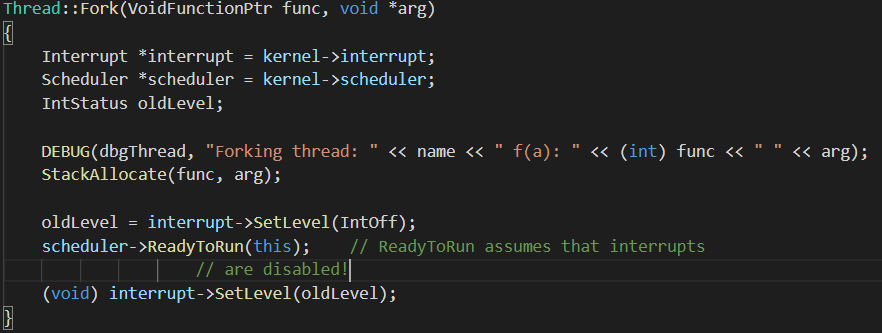
**userprog/addrspace.cc AddrSpace::AddrSpace**

allocate一塊user address space給剛剛建立出來的thread，建立page table，並設定好logical memory to physical memory的轉譯方式。在此由於非multi-programming，轉譯為簡單的 1：1的對應。

****

**threads/thread.cc Thread::Fork**

呼叫StackAllocate()來allocate一個stack、初始化stack，設定好machinestate，將interrupt設為disabled，呼叫Scheduler->ReadyToRun(this) 將此thread放入ready queue中，最後將interrupt設成原本的狀態。

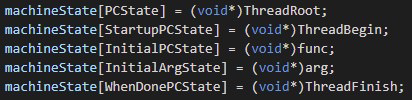


**threads/thread.cc Thread::StackAllocate**

allocate一個stack並初始化stack。



並將machinestate設定好，包含root，開始結束要call的function。



**threads/scheduler.cc Scheduler::ReadyToRun**

將thread的狀態設定為Ready，但還沒Running。

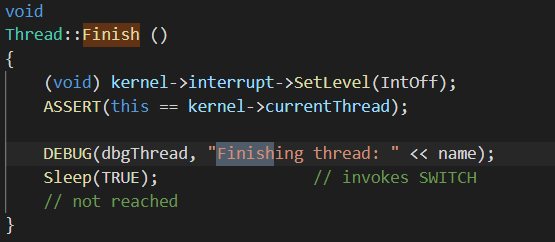
並將此thread排入ready queue之中，等待被執行。



**Threads/thread.cc Thread::Finish**

在Kernel::ExecAll中所有待執行的user programs都建立好threads了，main thread便會呼叫Finish()代表這個thread執行完成。

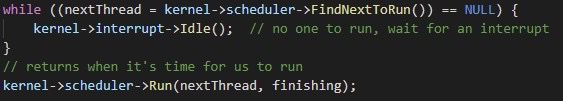
當某個thread結束執行完畢時就會呼叫，會先把interrupt disable，之後呼叫Sleep(true)來invoke context switch。



**threads/thread.cc Thread::Sleep**

Sleep被呼叫有兩種情況：（1）目前的thread執行完畢，傳入的finishing為true（2）目前的thread被block住（semaphore, lock等等），以上兩種情況，我們都會讓目前的thread釋放CPU資源，並且透過scheduler來找到新的thread來執行，達成context switch。

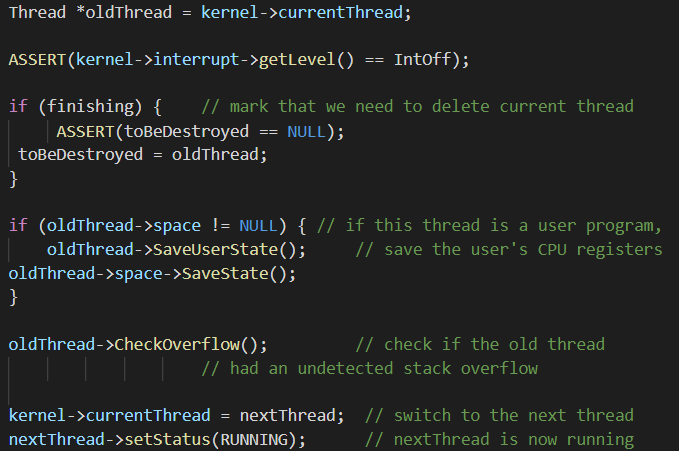
如果在ready queue中找不到可以執行的thread時，CPU處於Idle狀態，等待新的I/O interrupt（這樣才會有新的thread可以run），如果有可以執行的thread則透過scheduler繼續run下一個thread。



**threads/scheduler.cc Scheduler::Run**

這邊Scheduler會讓CPU執行下一個thread，並且前一個thread如果已經finish需要delete的話會在這邊執行。

首先記錄下原本的thread並且如果需要delete，就把它設成需delete，接著如果原本thread的space不是NULL（是user program），記錄下舊thread的CPU regs，並把kernel紀錄的currentThread改成下個thread。

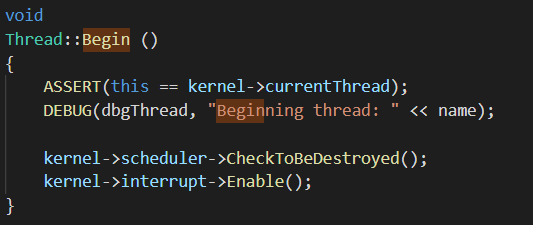


之後透過呼叫machine dependent context switch routine：「SWITCH」來實施context switch。



**threads/thread.cc Thread:Begin**

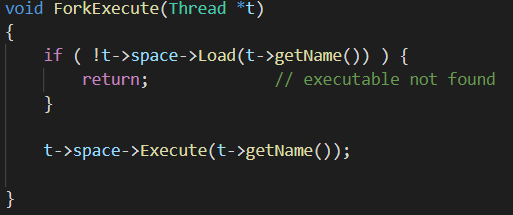
SWITCH會透過machingState[StartupPCState] 去call switch後的newThread開始時所要call的Thread::Begin，首先會呼叫scheduler->CheckToBeDestroyed()確認OldThread是不是finish的，是的話就把OldThread delete掉，再來把interrupt enable。



**threads/kernel.cc ForkExecute**

接著SWITCH會透過machingState[InitialPCState]（&ForkExecute在stackAllocate時放入），以及machineState[InitialArgState]當argument（對應到的thread在stackAllocate時放入）來call ForkExecute來執行這個thread。

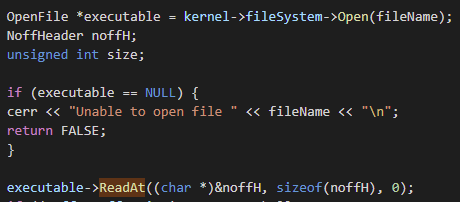
首先會把這個thread的memory space load好，接著透過Execute開始執行。



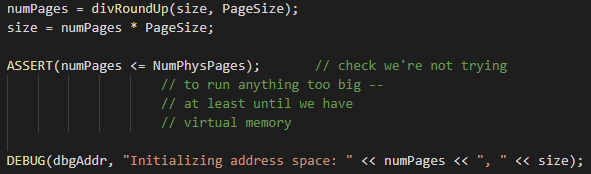
**threads/addrspace.cc::Addrspace::Load()**

由於目前fork出來的thread的memory儲存的並不是我們所要執行的program，所以必須利用Addrspace::Load()從file中load我們所要執行的user program到memory中。

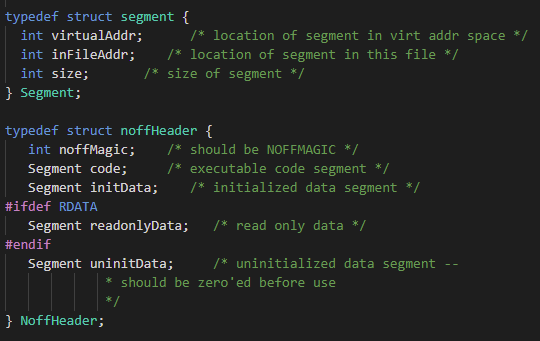
先利用filesystem->open()來開啟檔案，檢查是否開啟成功，再將檔案讀入名為noffH的noffheader object。



****根據實際的program address space size計算page number數以及含括這些pages所需的size，並檢查memory是否足夠（numPages <= NumPhysPages）。



最後將user program（noffH）的code以及data segments各別讀入main memory中。



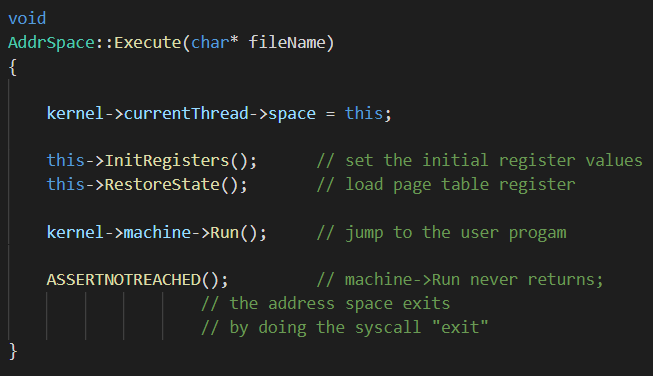
**threads/addrsapce.cc::Addrspace::Execute()**

把currentThread的space pointer指到這個Addrspace object。

initRegisters()設定好user-level的register set。設定PCReg為virtual memory的頭(zero)，並設定NextPCReg為PCReg+4，因為一個instruction為4 bytes，以及設定StackReg為virtual memory的尾。

RestoreState()把目前thread的page table跟size load到machine內。

最後call kernel->machine->Run()來執行指令



**II.2 implementation**

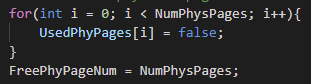
在kernel.h的kernel class中新增兩個變數

Bool UsedPhyPages[NumPhysPages] 用來記錄哪些physical pages正在被使用

Int FreePhyPageNum 紀錄還剩下多少physical pages是available的



在kernel.cc的kernel constructor中初始化這兩個變數



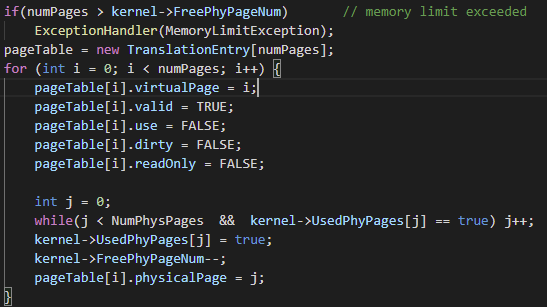
在machine.h的ExceptionType中新增MemoryLimitException。

將AddrSpace::AddrSpace()的內容全部註解掉。

因為必須知道目前這個thread需要多少pages，再開相對大小的page table，而numpages是在Load()時計算出來的，所以要在AddrSpace::Load()裡建立page table。

計算出numpages後，必須先判斷是否超過剩餘physical memory大小，如果有則呼叫ExceptionHandler()，沒有才繼續執行。

接著會建立pageTable，初始化valid、use、dirty等bits，對每個virtualPage，在UsedPhyPages中找到尚未被occupied的frame，並設為此virtualPage的physicalPage。

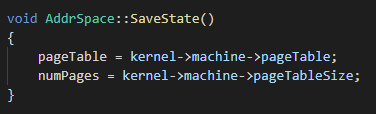


將noffH.code.virtualAddr跟noffH.initData.virtualAddr轉換成Physical pages的address。

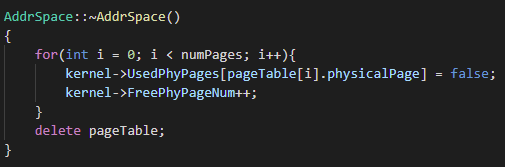
**** 

AddrSpace::SaveState()

在context switch時儲存目前的machine state。

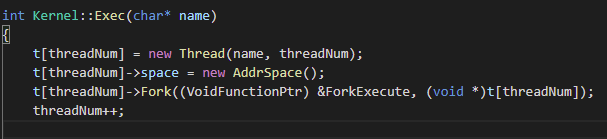


修改AddrSpace的destructor，release address space並restore physical page status。



**II.3 Report Questions：**

* **How Nachos allocates the memory space for new thread(process)?**



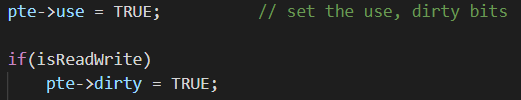
在kernel->Exec()中，在建立完thread object之後會呼叫AddrSpace()分配memory space給thread。

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

AddrSpace::Load()負責將user program的code及data segments load至thread的memory space。

* **How Nachos creates and manages the page table?**

在Exec()中每new完一個thread時，都會接著透過new一個AddrSpace來allocate address space給thread。在AddrSpace的constructor中會建立好一個page table，也就是TranslationEntry array，並對一些性質進行設定，管理的部分則是會在AddrSpace::Translate()內針對特定的page table entry進行update。



* **How Nachos translates address?**

In AddrSpace::Translate()

vpn(virtual pages number) = vaddr / Pagesize;

offset = vaddr % Pagesize;

pte (page table entry) = &pageTable[vpn];

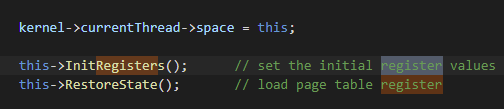
pfn (page frame number) = pte->physicalPage;

\*paddr = pfn\*PageSize + offset;

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

AddrSpace::InitRegister()會初始化machine registers

AddrSpace::RestoreState()負責load page table



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

Thread這個class代表process(thread) control block。

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

Thread在Kernel::Exec()被建立跟allocate address space後，在Thread::Fork()中會呼叫scheduler->ReadyToRun()並把這個thread加入ready queue中。