**CS342301: Operating System**

**MP2: Multi-Programming**

組別: 8

組員:

109062315洪聖祥

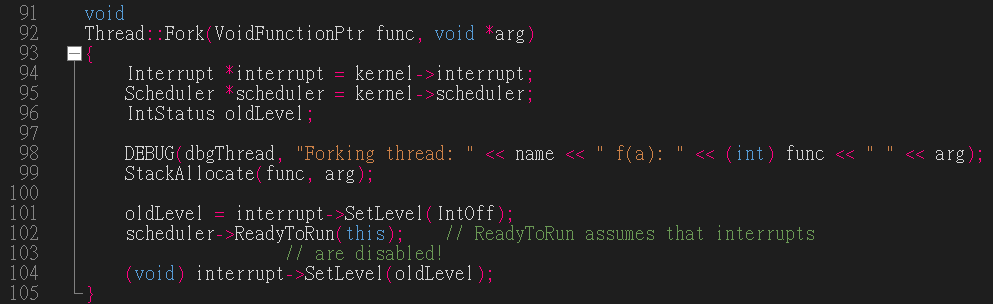
109062314張紘齊

|  |  |
| --- | --- |
| 項目 | 分工 |
| Trace Code | 都有看過全部 |
| Report |  |
| Thread::Sleep() | 張紘齊 |
| Thread::StackAllocate() | 張紘齊 |
| Thread::Finish() | 張紘齊 |
| Thread::Fork() | 張紘齊 |
| AddrSpace::AddrSpace() | 洪聖祥 |
| AddrSpace::Execute() | 洪聖祥 |
| AddrSpace::Load() | 張紘齊 |
| Kernel::Kernel() | 洪聖祥 |
| Kernel::ExecAll() | 洪聖祥 |
| Kernel::Exec() | 洪聖祥 |
| Kernel::ForkExecute() | 洪聖祥 |
| Scheduler::ReadyToRun() | 張紘齊 |
| Scheduler::Run() | 張紘齊 |
| Implementation | 洪聖祥 |
| Debug | 張紘齊 |

**Part I: Trace Code**

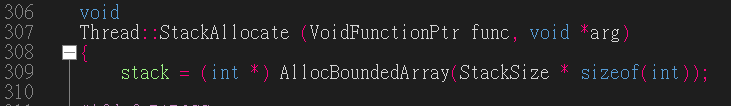
1-1. threads/thread.cc

Thread::Fork()

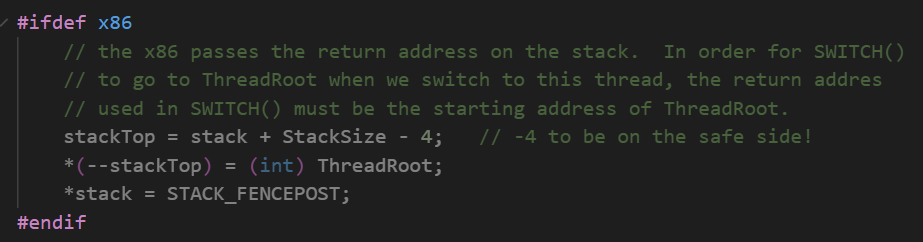


主要是讓caller和callee兩個thread能夠execute concurrently。首先呼叫Thread::StackAllocate替這個thread配置一個stack，並初始化相關變數以及registers的內容，接著讓scheduler把這個thread放到ready queue上，然後要注意這個動作必須是要在interrupt disabled的情況下才能執行

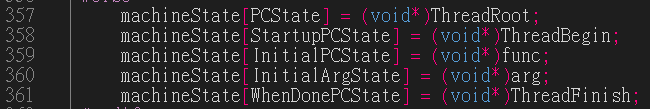
Thread::StackAllocate()



首先配置一塊連續的記憶體給stack，讓thread能存放temporary data

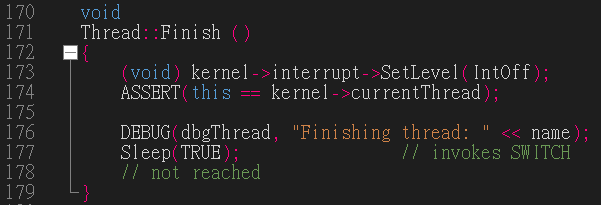


接著設定current stack pointer (stackTop) 的位置、放好ThreadRoot、將Bottom of the stack (\*stack) 設為STACK\_FENCEPOST來判斷stack overflows



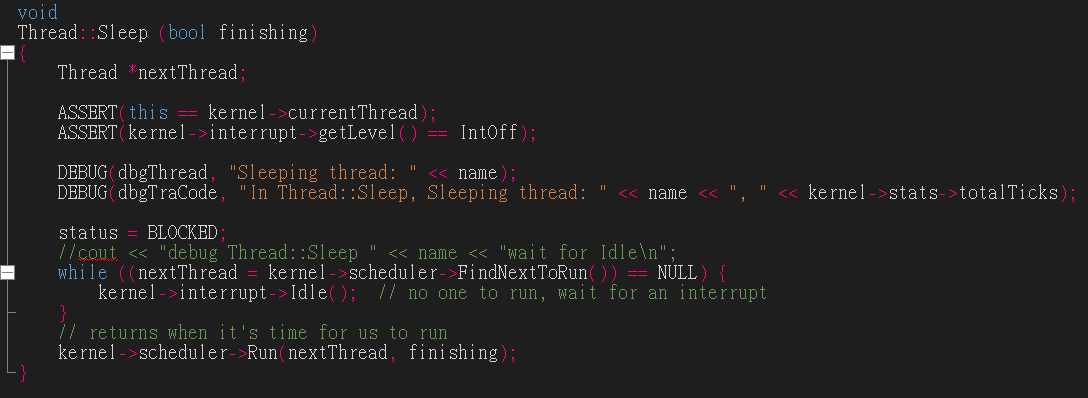
最後設定registers的值，讓Context Switch發生時能夠切換到當前的thread

Thread:: Finish()



主要是處理thread執行完成，這裡的方法是透過呼叫Thread::Sleep，並給予參數TRUE，代表說這個thread之後都用不到，相關資料可以移除

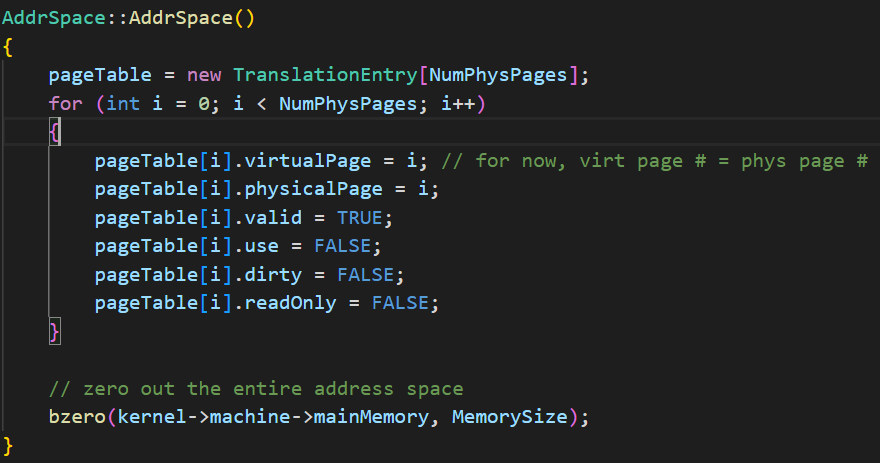
Thread:: Sleep()



主要是處理thread 要被blocked或是finished，因此status會被設為BLOCKED。接下來會透過while迴圈判斷當前ready queue上是否有其他thread，沒有的話就執行interrupt->Idle去看有沒有interrupt要發生，直到ready queue能有可以運行的thread，否則程式就會被停止運行；有的話就請scheduler執行那個thread，並將參數finishing放入，用來判斷當前的thread是否已完成任務

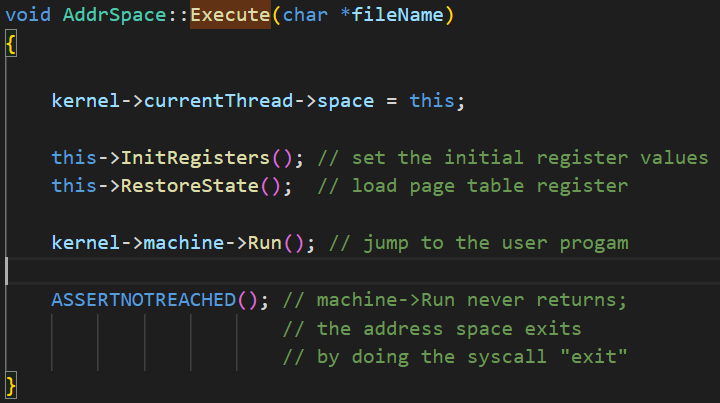
1-2. userprog/addrspace.cc

AddrSpace:: AddrSpace()



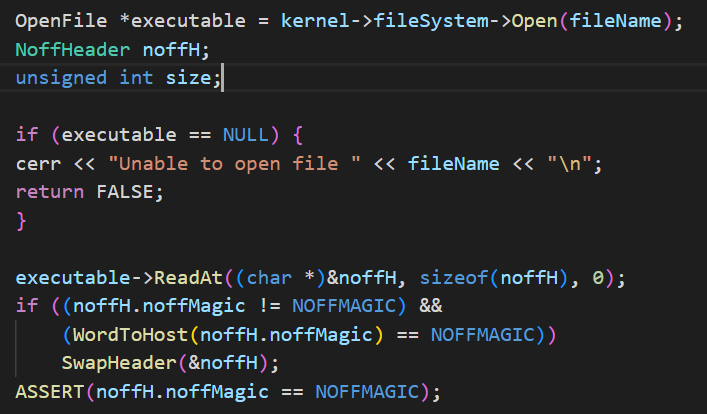
AddrSpace的constructor初始化pageTable，但是現在是沒有multiprogramming 的版本，所以virtualPage直接對應到physicalPage。並且清空所有address space。但是multiprogramming的版本不應該直接清空所有address space，因為有可能在address space有其他process在memory。並且physicalPage也需要做額外的處理。

AddrSpace:: Execute()



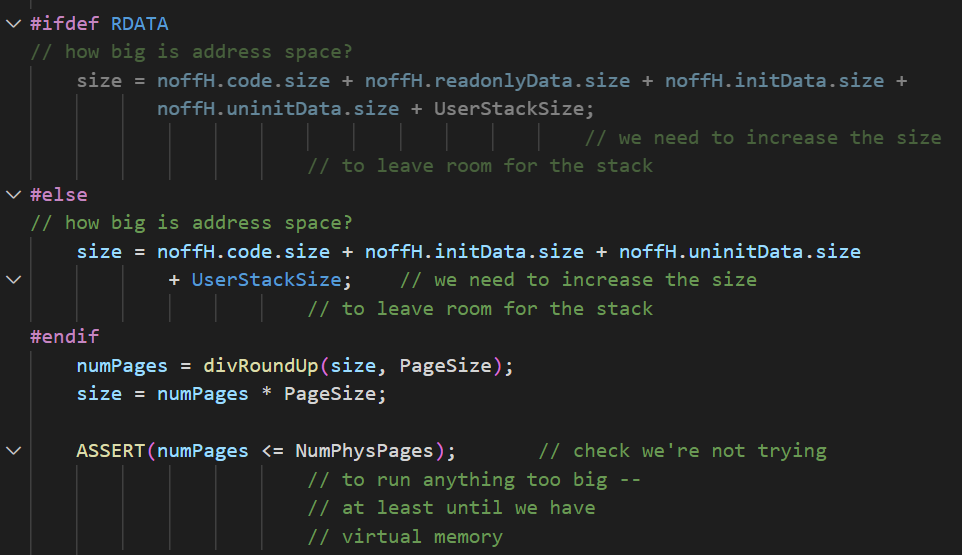
首先設定current thread 的 address space是現在的address space，然後清零所有register，把page table load進來，最後執行user program。

AddrSpace:: Load() 主要就是把user program放入memory之中

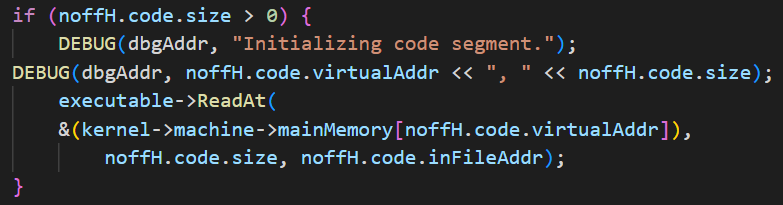


首先根據filename去找對應的檔案，若找不到則直接return並回傳False。

接著判斷是否要呼叫AddrSpace::SwapHeader將little endian轉成big endian。



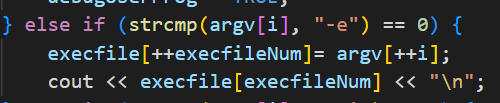
這裡是在計算這個file各個部分大小的總和 (包含了thread stack的大小)，接著計算所需用到的page數量，由於可能不是page size的整數倍，因此計算結果需要無條件進位，最後在判斷所需的page數量是否小於physical memory中能夠容納的page 數量



剩下部分以code segment為例，由於當前的實作virtual address和physical address相同，因此不需要呼叫AddrSpace::Translate，即可直接用virtual address去memory對應的位置將code segment放入memory之中

1-3. threads/kernel.cc

Kernel:: Kernel()



Kernel constructor處理command line指令，這邊以 -e “filename”為例，execfile array存放每個file name。並且execfileNum 紀錄總共有幾個file。

Kernel:: ExecAll()

Kernel 對每一個執行檔呼叫Exec(char\* name)，呼叫完結束current thread。

Kernel:: Exec()

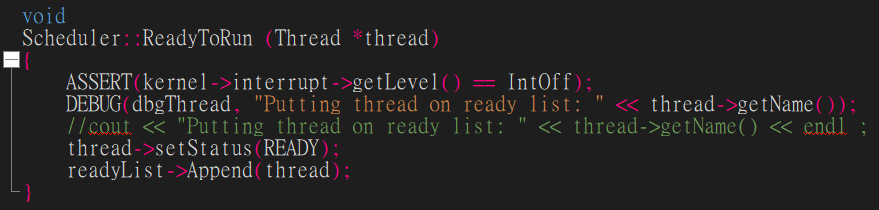
每一個program以一個thread為單位執行(不考慮multithreading)。所以instantiate Thread class。每一個thread 要分配memory space給process，所以instantiate AddrSpace class。最後呼叫thread->Fork() ，初始化 thread state和分配stack給thread。

Kernel:: ForkExecute()

把一個file(program) load 到 memory，並且執行。

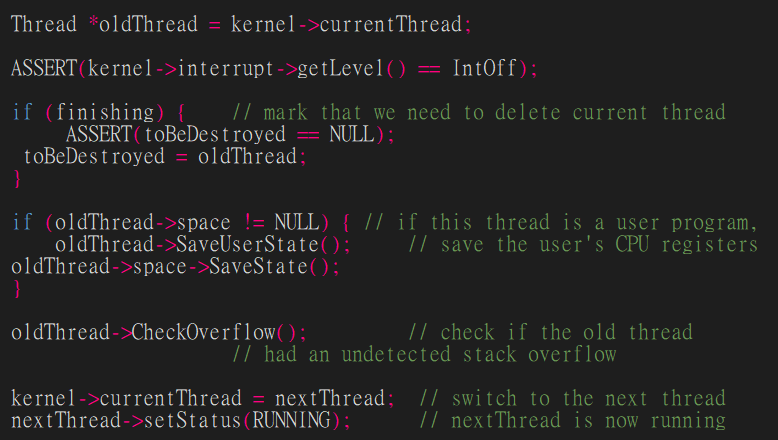
1-4. threads/scheduler.cc

Scheduler:: ReadyToRun()



將thread的status切換成READY，然後把此thread放入ready queue的最後面

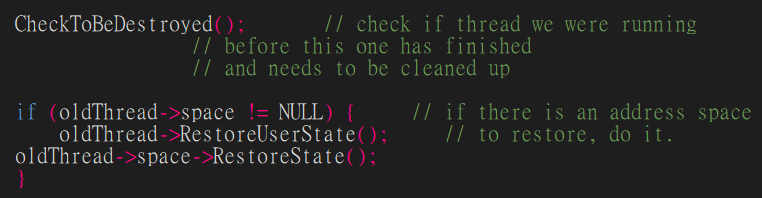
Scheduler:: Run()



這裡主要是在對current thread (oldThread) 做處理，像是這個thread是否將要被刪除、需不需要將它的CPU registers、address space相關資訊存起來、thread中的stack是否發生overflow。接著把current thread切換成nextThread，並將之status設為RUNNING



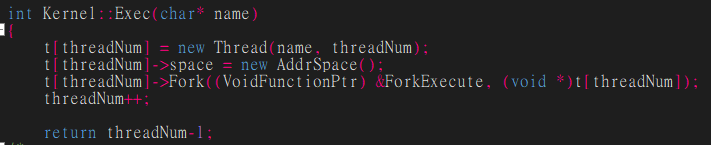
這裡會執行用assembly language寫好的context switch函式



做好context switch之後，由於現在已不會使用到oldThread中的資訊，因此可以呼叫Scheduler::CheckToBeDestroyed去把該thread的資源清除掉。最後假如該thread未來還會被執行(thread的address space不為空)，則須將相關的資料記錄下來

**Part II: How Nachos creates a thread(process), load it into memory and place it into the scheduling queue**

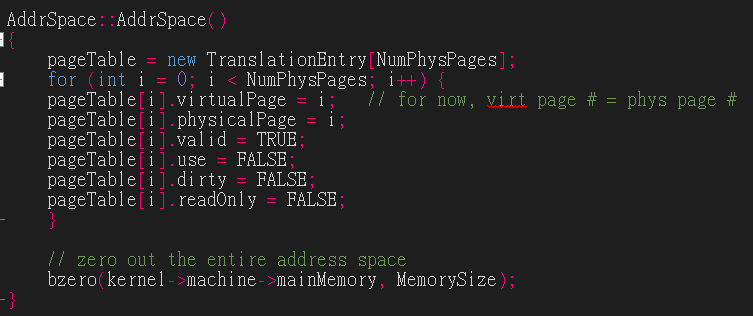
* How does Nachos allocate the memory space for a new thread(process)?



在Kernel::Exec中的第二行會new AddrSpace()，並把它放入thread中的space變數，也就是替thread allocate memory space

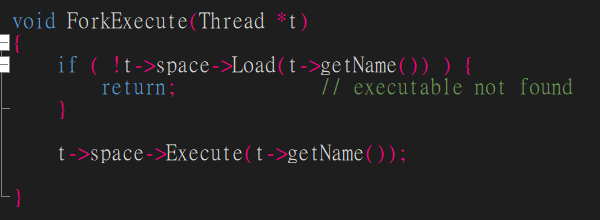
* How does Nachos initialize the memory content of a thread(process), including

loading the user binary code in the memory?



在上一題中，呼叫new AddrSpace() 就會透過bzero先初始化memory content，以及創立page table並初始化各個entry的資訊





接下是當要執行thread的function時，會呼叫machineState[InitialPCState]

中的Kernel::ForkExecute，其所需參數會存在machineState[InitialArgState]。然後在Kernel::ForkExecute中，可看到會根據thread name呼叫AddrSpace::Load，此時就會將該thread所需用到的binary code load進memory裡。

* How does Nachos create and manage the page table?

原本的設計是根據uniprogramming，因此一開始在呼叫new AddrSpace() 時，就會根據NumPhyPages來建立page table，而至於管理，就只有一開始的新增和最後的刪除，且因為設計是uniprogramming，context switch也不該發生，因此也不會呼叫到AddrSpace::RestoreState()去切換page table的相關資訊。

而我們的實作會maintain freeFrame list，並根據user program的大小去建立page table，並對每個virtual address分配freeFrame list裡的physical frame，且在thread的address space要被刪除時，會將它所佔有的frame內容清空，並append回freeFrame list中，最後才刪除page table

* How does Nachos translate addresses?

Original: does not translate address, virtual address = physical address

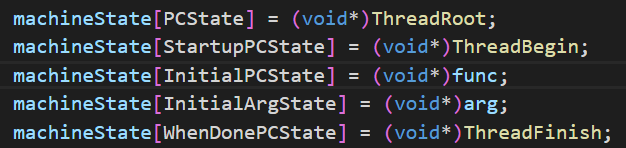
Implementation(multiprogramming): in function Translate(), it first find the virtual page number by dividing virtual address by page size (page number = virtual address / page size) and find offset by taking remainder of virtual address divide by page size (offset = virtual address % page size). Then, access the page table’s virtual page number index to find the corresponding TranslationEntry, in TranslationEntry saves the physical page number. Last, calculate physical address by multiplying page size and physical page number and add offset(physical address = page size\*physical page number + offset).

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

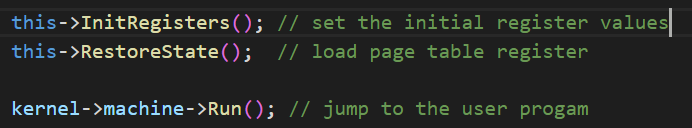
When tracing code, there is a comment saying “A thread running a user program actually has 2 sets of CPU registers – one is for its state while executing user code, one for its state while executing kernel code”.

Before running a thread(process):

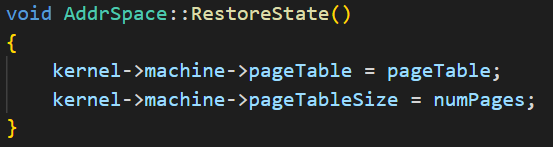
Initialize registers while executing kernel code

In AddrSpace::Execute():

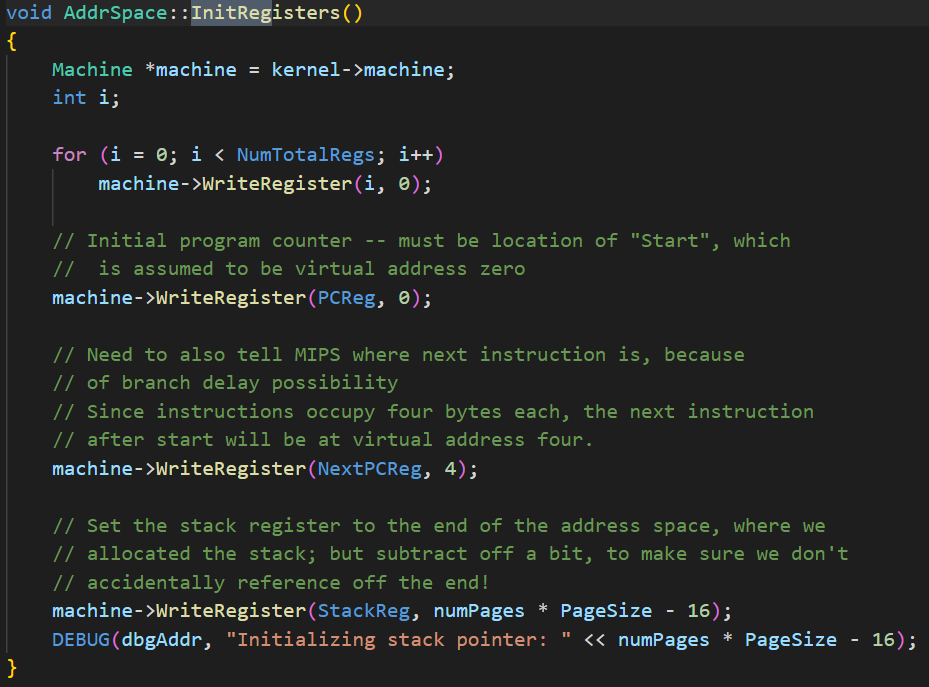
Initialize user-level registers and load page table register



Load page table



Zero out all registers and initialize PC register Next PC register Stack register.



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

Thread object acts the role of PCB because of the following features :

1.ThreadStatus{JUST\_CREATED,RUNNING,READY,BLOCKED,ZOMBIE} status:

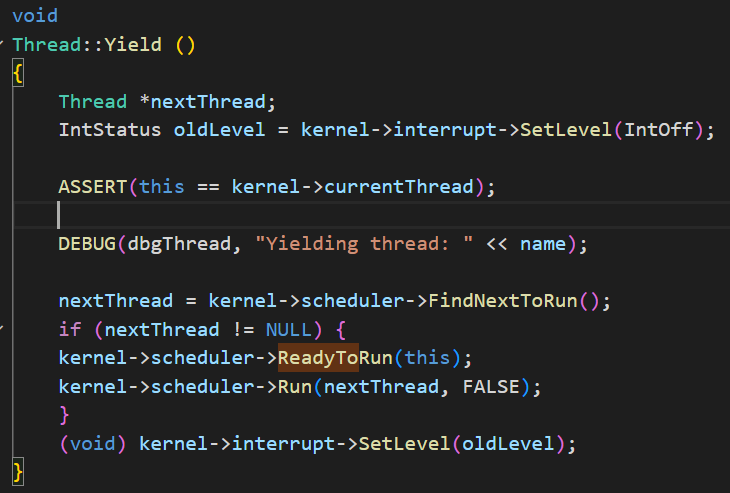
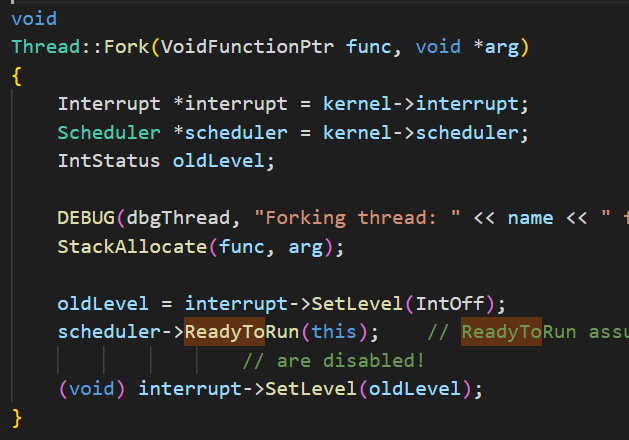
which corresponds to the 5 state New, Run, Read, Wait, Terminate state in process state

2.Stack

3.ID: process’s unique pid

4.userRegisters[NumTotalRegs]: user-level CPU register state

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



Thread::Fork和Thread::Yield之中，scheduler會將當前thread append到ready queue的最後面。而兩著各別會在Kernel::Exec和Interrupt::Onetick被呼叫到

**Part III: Implementation**

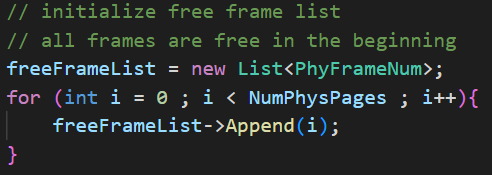
kernel.h





在kernel.h中的class Kernel新增一個資料結構叫做freeFrameList。freeFrameList主要是要記錄 physical memory 中哪些 frame 是free 的，這樣在為process分配 memory 時，一定會分配到memory 是free的 frame，不會有被分配到明明這個frame 已經有其他process 佔據了還被分配到的情況。freeFrameList 以linked list實作，可以用 list.h 中的 class List直接呼叫。

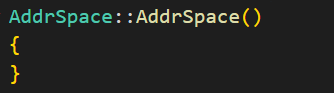
kernel.cc



在class Kernel的constructor中初始化freeFrameList。一開始所有在physical memory的 frame都是free的，所以將每一個frame number都append到freeFrameList中。

addrspace.cc

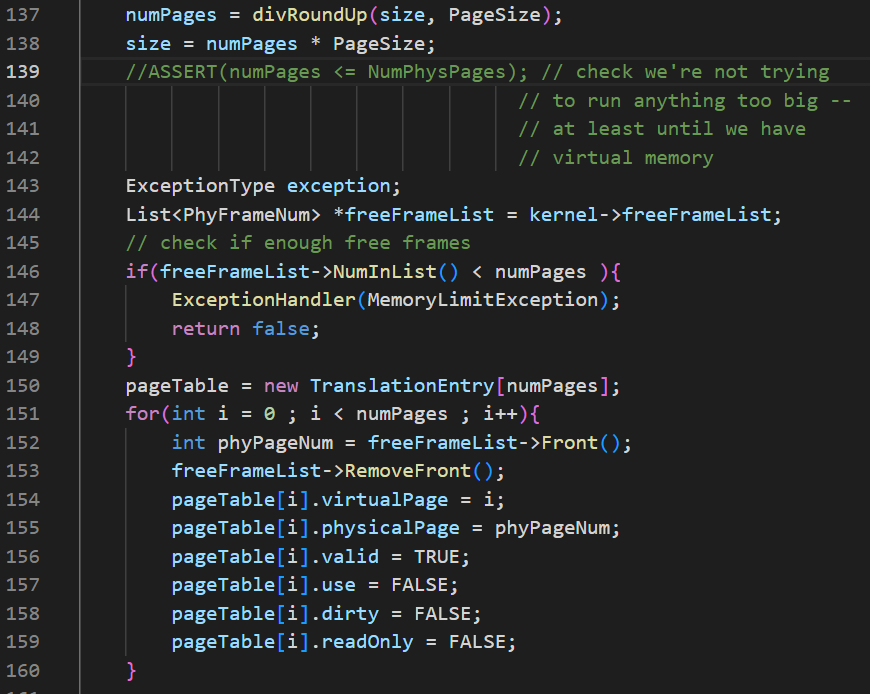
AddrSpace::AddrSpace()



將class AddrSpace的constructor中對pageTable的initialization刪除。因為原來的實作是初始化 NumPhysPages (128) 個 TranslationEntry，但是其實不需要這麼多的TranslationEntry，只需要program size / PageSize 個 TranslationEntry。另一個原因是pageTable其實也會存在physical memory中，多餘的TranslationEntry沒用只會造成memory waste。因為我們在處理 load program 的時候才會知道 process size，所以我把pageTable 的初始或移到 AddrSpace::Load(char\* filename) 中。

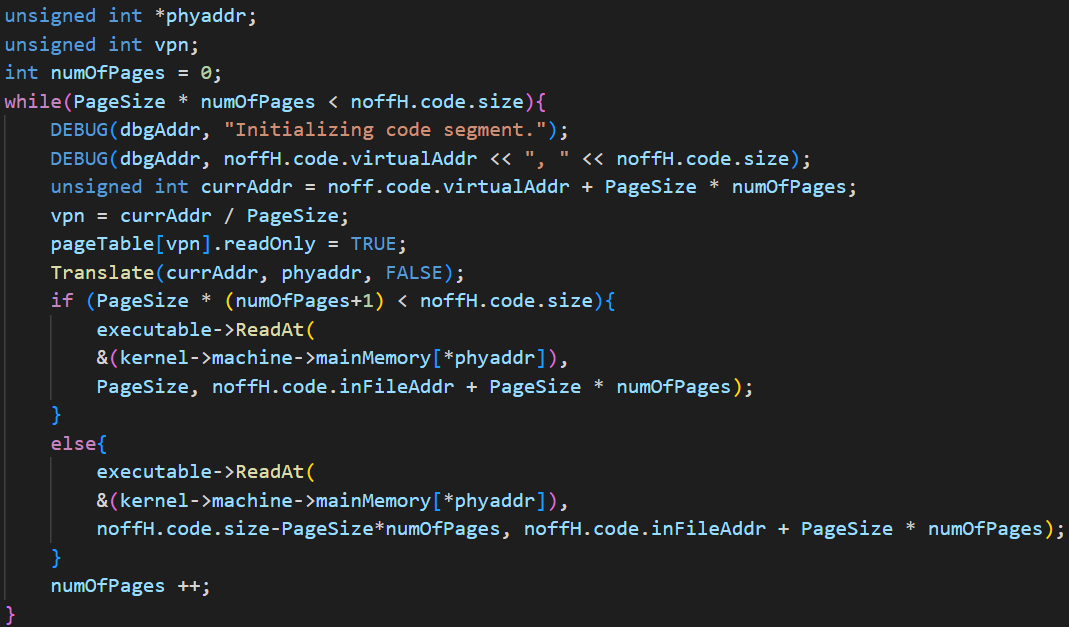
AddrSpace::Load(char\* filename)

Line 143-160 implementation



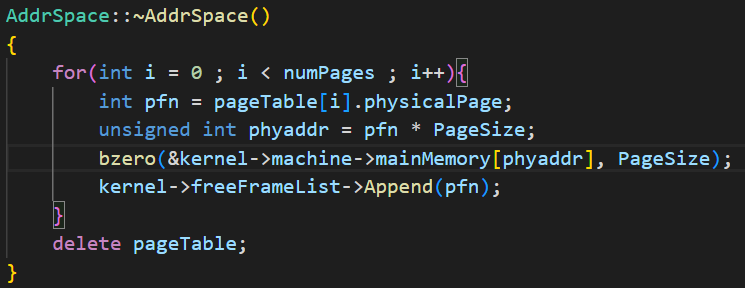
我們在 AddrSpace::Load(char\* filename) 可以知道這個program總共有幾個page (numPages 在137行計算)並且初始化pageTable，總共有 numPages個TranslationEntry。但是我們也要初始化TranslationEntry的值，也就是 physical page number、valid bit、use bit、dirty bit、read only bit要是什麼值。所以我們可以用在kernel.h中定義的freeFrameList來知道哪些frame 是free的，可以分配給program的。如果freeFrameList裡free frame的數量小於numPages，則會呼叫MemoryLimitException。用一個for loop來一個個初始化每一個 TranslationEntry 的physicalPage，把freeFrameList->Front()得知在linked list中第一個free frame number (free page number)是多少，再呼叫freeFrameList->RemoveFront() 把第一個 page number移除(代表這個 page number已經不是free了)。將valid bit設為true，use bit設為false，dirty bit設為false，readOnly bit設為false。readOnly bit會隨著program中不一樣的地方做調整，像是code segment readOnly bit要設為 true (code segment不可以隨意更改)，所以在之後設為false (因為我們目前不知道 code segment 在pageTable的哪一個TranslationEntry，所以只能在load code segment的其間做修改)。

Load code segment into physical memory and maintain pageTable



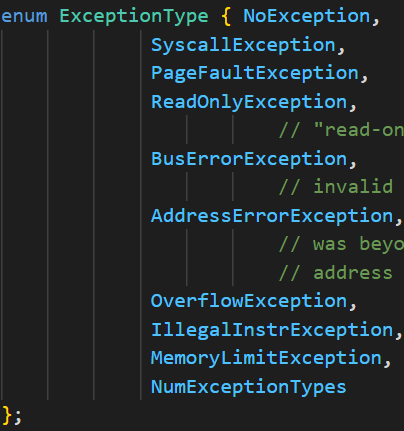
接下來要把code segment load 到physical memory中，首先要做的是將virtual address 轉換成 physical address，並且每一次的load 是以page size為單位。因此我用一個while loop 將code segment以page size為單位load 到physical memory中。Virtual address 轉換到 physical address 是用 Translate() 這個function。轉換好後呼叫executable->ReadAt() 把一個page size的code segment load 到physical memory。需要注意的是，最後一次load到physical memory的code segment可能不足一個page size，也就是所謂的internal fragmentation。所以最後一次load的大小可能小於一個page size。其他像是data segment、read-only data segment也是跟code segment一樣的實作方法。

AddrSpace::~AddrSpace()



在class AddrSpace的destructor中因為這個process已經執行完了，不需要占用physical memory了，所以要把pageTable的所有physical frame number 重新放到freeFrameList，並把process 占用的physical memory設為0。最後把pageTable delete。

machine.h



新增MemoryLimitException來做為當沒有free frame的時候要乎要exception.