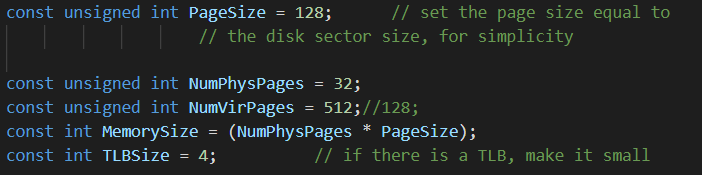
Operating System HW－Multiprogramming and Page Replacement

Group 18

106061132 黃友廷 106061146 陳兆廷

1. Code Tracing and Implementation
   1. Declaration and Definition
      1. machine/machine.h



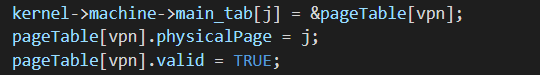
一張含有 文字 的圖片

自動產生的描述

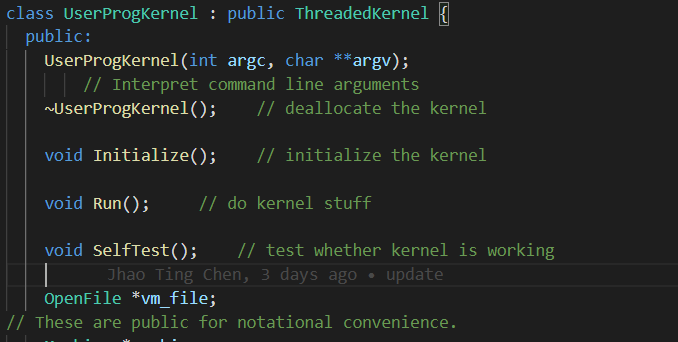
NumPhysPages代表main memory被分成幾頁page，因為這個作業是希望page fault越低越好，所以盡量讓page size越大會越好，但page size越大也意謂著I/O的時間會越久。而我們定為32頁，因為此作業有限制memory不能大於4096 byte，所以我們的PageSize就是128 byte (4096/32)。NumVirPages則是代表virtual memory最大可以有幾頁page，我們定為512頁，所以我們的virtual memory最大可以用65536 byte。

usedPhyPage[]、usedVirPage[]這兩個array是用來記錄main\_tab和pageTable裡有哪些entry被使用了，換句話說 usedPhyPage[]就是代表哪些entry是有紀錄被load進main memory的page，而 usedVirPage[]則是記錄user program共用到了哪些頁的virtual memory page。

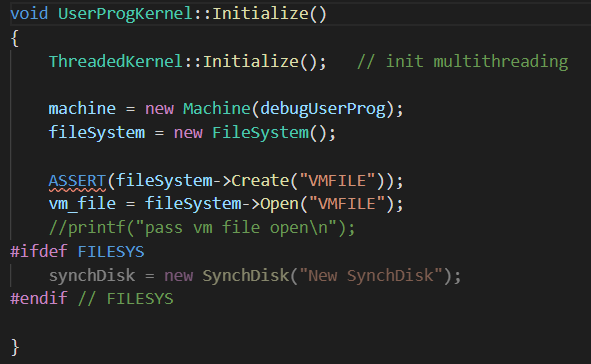
pageTable是用來記錄整個vurtual memory有用到的部分所分成pages的資訊(ex: valid bit, dirty bit, process ID, page number...)，我們會根據要執行多大的userprogram去new多少個entry給pageTable。main\_tab則是去紀錄有load到main memory內的pages，但我們並不會另外new entry給它，而是當有page被load到main memory時，就直接將main\_tab[]這個pointer直接指向那個page的pageTable entry，並且將valid bit改成true，如下圖。



* + 1. userprog/userkernel.h

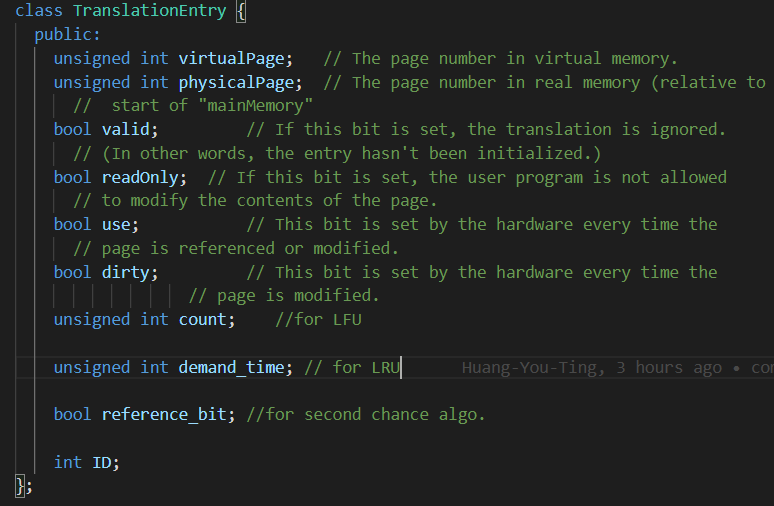


我們的virtual memory作法是去開一個file當作virtual memory，所以需要在UserProgKernel的class裡面加一個OpenFile的pointer (vm\_file)，這樣我們就能使用WriteAt()、ReadAt()對file進行讀寫。



所以我們也需要在UserProgKernel::Initialize()裡面create一個file並且將它open，如此一來之後操作的時候我們才能進行讀寫。

* + 1. machine/translate.h



TranslationEntry這個class是pageTable entry的data type，virtualPage是記錄這個entry是屬於哪一個virtual memory page的(也就是記錄它的virtual memory page number)，physicalPage則是記錄這個page會被存到哪一個main memory中的page。

valid是記錄這個page是否還在main memory裡面，dirty則是記錄這個page在memory中時是否有被寫過，ID則是記錄這個page是屬於哪一個process。

Count、demand\_time、reference\_bit這三個variable是用在page replacement algorithm中。count是去記錄這個page總共被access幾次，用在LFU。demand\_time記錄這個page上次被access的時間，用在LRU。reference\_bit則是用在second chance algorithm。

* 1. Load program
     1. userprog/userkernel.cc - UserProgKernel::Run()

當nachos將arguments中的-e、檔案名稱讀入時，會進入Run()執行欲執行的檔案。

一張含有 文字 的圖片

自動產生的描述

Run()會將每個欲執行的檔案分配為一個Thread，並將這些Thread的空間經過AddrSpace()初始化。最後再由ThreadedKernel::Run()執行。

* + 1. userprog/addrspace.cc - AddrSpace::AddrSpace()

在初始化AddrSpace這個物件時，尚未新增MultiProgramming時AddrSpace會依照這個OS(nachos)的Memory大小來初始化PageTable。這個PageTable就是系統在machine/machine.h中定義的NumPhysPage及PageSize組成，因為還沒有MultiProgramming的功能。如下圖。

一張含有 文字 的圖片

自動產生的描述

而當我們新增MultiProgramming功能後，就無法只使用固定大小的PageTable，因此要在後續Load的地方再來定義PageTable，如下圖。唯一要宣告的只有用來計算Page Replacement Algorithm的ID。

一張含有 文字 的圖片

自動產生的描述

* + 1. threads/kernel.cc - ThreadedKernel::Run()

直接進入Thread::Finish()。

* + 1. threads/thread.cc - Thread::Finish()

一張含有 文字 的圖片

自動產生的描述

模擬正在執行Thread的過程，進入Sleep。

* + 1. threads/thread.cc - Thread::Sleep (bool finishing)

一張含有 文字 的圖片

自動產生的描述

Sleep模擬執行完Thread的過程。如果執行完了，就要利用FindNextToRun()去CPU Scheduler的ready queue找下一個需要執行的Thread。最後同樣再用Run()執行。

* + 1. threads/scheduler.cc - Scheduler::Run (Thread \*nextThread, bool finishing)

因為需要將原先在執行的Thread與將要執行的Thread進行替換，必須在系統中更新一些state及將兩個Thread位址交換。

一張含有 文字 的圖片

自動產生的描述

除了將nextThread的狀態設為Running，進入SWITCH。

* + 1. threads/switch.s - SWITCH ( thread \*t1, thread \*t2 )

如同前次作業的交換：

一張含有 文字 的圖片

自動產生的描述

在 SWITCH 一開始，t2 是存在 8(esp)，t1 存在 4(esp)，而(esp)為 return 的位址。ESP 為 Stack Pointer。

一張含有 文字 的圖片

自動產生的描述

340、341 行在把原本放在reg eax的值取出存在\_eax\_save後，把t1放入eax。

342～348、350 是將所有reg依照eax的位置存好，也就是 t1 的位置。

349 將原本 eax 的值，也就是eax\_save存入ebx。

351、352 把return address存入ebx中，並存在PC storage裡。

354 將 t2 放入 eax。

356、357把新eax的值放入ebx，並存在\_eax\_save。

358～364把t2的register value存回原本的register。

365把原本存在PC的return address放回eax。

366 把eax指向4(%esp)，也就是把原本return address存在stack的位置(%esp)往上移了4 bytes。

367把原本eax的值放回eax。

* + 1. thread/thread.cc - ThreadBegin()

回到一開始UserProgKernel::Run()的地方，在所有Thread開始前會經過ThreadBegin()將現在的Thread進入Begin()的狀態，初始化scheduler及interrupt等。

* + 1. userprog/userkernel.cc - ForkExecute (Thread \*t)

一張含有 文字 的圖片

自動產生的描述

直接進入AddrSpace::Execute()。

* + 1. userprog/addrspace.cc - AddrSpace::Execute (char \*fileName)

一張含有 文字 的圖片

自動產生的描述

Execute前要先把檔案Load進Memory及定義PageTable。在這之前我們使用了一個AddrSpace的變數(定義於addrspace.h)：pt\_is\_load，來限制當兩個Thread同時在存取Physical Memory時，不會因為同時執行而造成Memory狀態讀取錯誤。

* + 1. userprog/addrspace.cc - AddrSpace::Load (char \*fileName)

一張含有 文字 的圖片

自動產生的描述

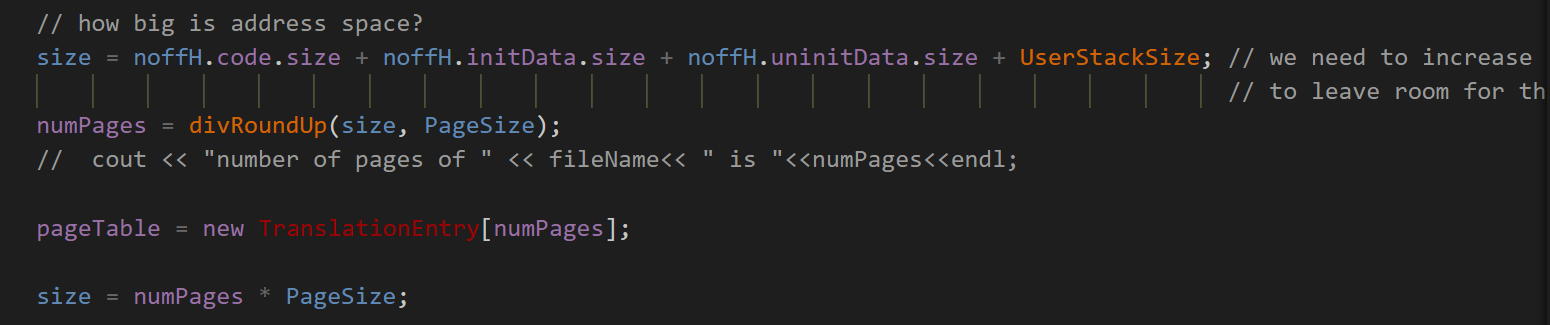
首先先用FileSystem Open將欲執行檔案打開以供讀取，並將檔案轉乘noffH格式。

noffH格式為將檔案的程式及Data部分用Segmentation分開，以利讀取。回傳code的大小、位址；Data的大小、位址以及其餘資訊。

一張含有 文字 的圖片

自動產生的描述

由於作業說明中沒有說明，也無法得知如何將Code及Data讀入PageTable，只能盲目的測試：code segment要直接接續Data segment放入PageTable及Memory。



獲得檔案大小後，我們的Memory還要放入UserStack，保留空間執行後續的程式。在addrspace.h中定義。因此總大小為：code segment + data segment + UserStack。接著要計算所需的頁數，並初始化PageTable。

一張含有 文字 的圖片

自動產生的描述

接著將code segment、Data segment依序放入PageTable。在中間交接的頁數（code未滿一頁、接續Data的前面資料），要用變數及計算offset的方式放好中間這一頁，後續Data segment也繼續放入PageTable。­­

一張含有 文字 的圖片

自動產生的描述

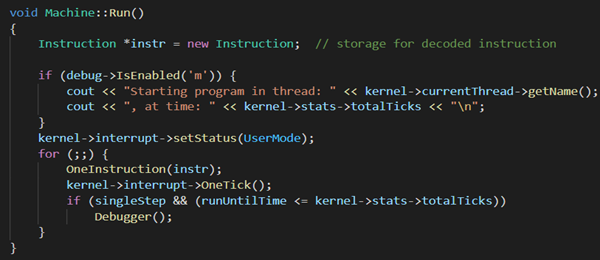
原先的程式是放入Memory中，但由於要使用Virtual Memory，我們放入我們的Virtual Memory裡，並將PageTable定義好。首先要先尋找沒有使用過的Virtual Page，接著將後續需要計算的變數都初始化，把Page的內容讀入buf並寫入Virtual Memory中頁數的位置。

一張含有 文字 的圖片

自動產生的描述

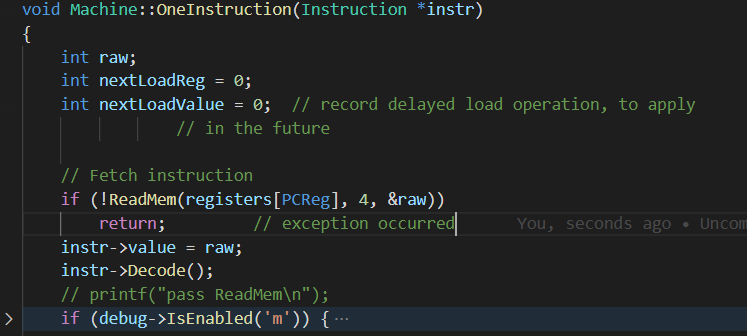
而code與Data中間的交接處，就使用變數處理好。

* 1. Page Faults
     1. machine/mipssim.cc - Machine::Run()

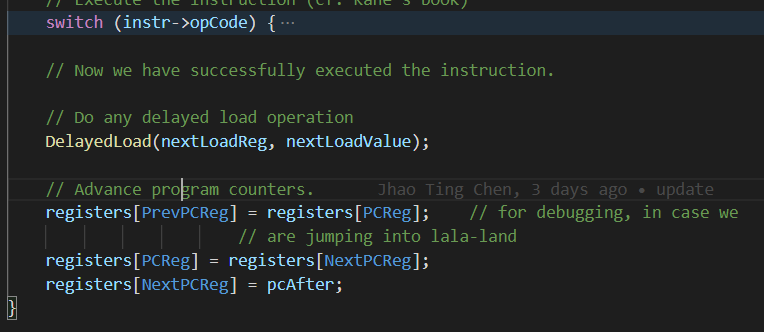


Machine::Run()的功能是在NachOS上模擬執行user的program，setStatus(UserMode)會先將OS的設定成user mode，接著進入for loop去呼叫OneInstruction()去從已經被compile成binary file的User program中讀取instruction並執行。Interrupt->OneTick()則是會去檢查是否有interrupt被呼叫。

* + 1. machine/ mipssim.cc - Machine::OneInstruction(Instruction \*instr)

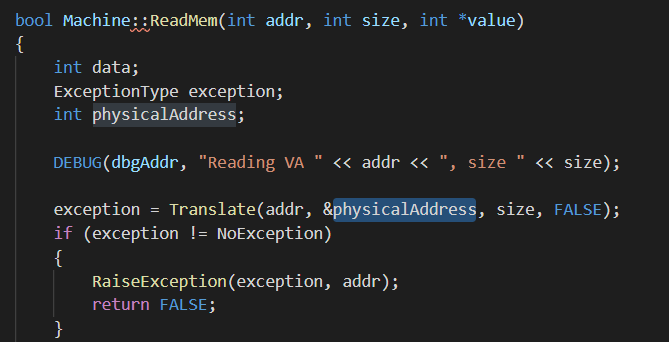


Machine::OneInstruction()功能是從memory中把instruction讀出來，並且去解析這個instruction需要執行甚麼事。ReadMem()會根據Program Counter的位置從記憶體內將資料搬到raw內，接著使用instr->Decoder()去對raw進行解碼，並找出對應的opcode。

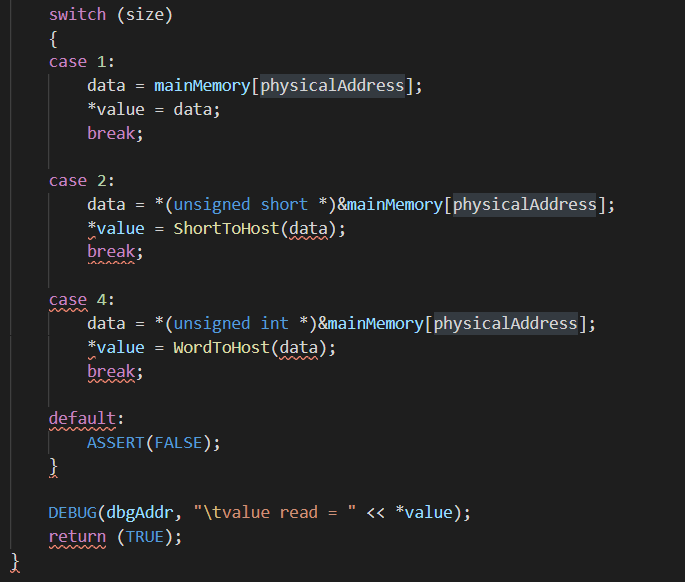


解析出來的opcode會藉由這個switch去對應到各自所需要被執行的工作，有部分工作會使用 RaiseException() 去處理，然後最後會把Program Counter指向User program下一個要被執行位置。

* + 1. machine/translate.cc - Machine::ReadMem(int addr, int size, int \*value)



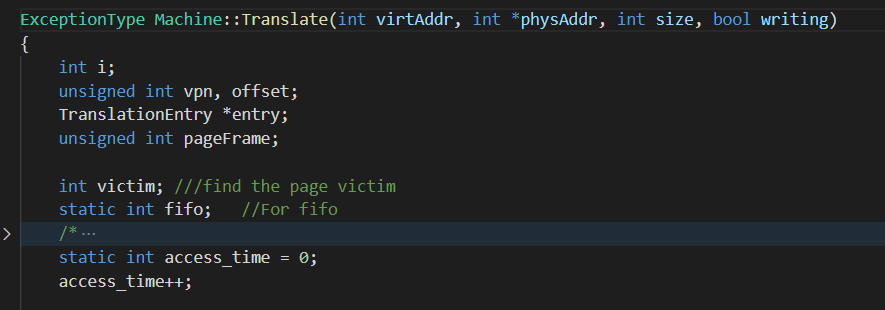
Machine::Read()會根據addr去從main memory中讀取大小為size的data並且放到value裡面。Translate()是負責將logical address (addr)轉換成physical address，RaiseException()則是去處理Translate()的執行過程中是否有exception的發生，像是address error、alignment problem、page fault...等。



得到physical address後就會從main memory中把data讀出來並放到value這個pointer裡。

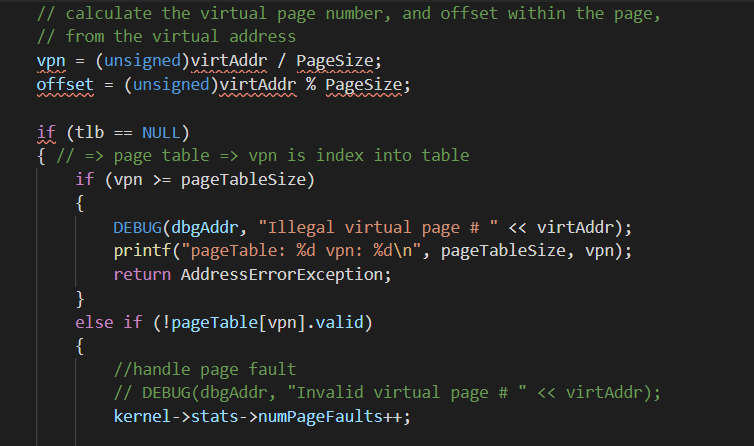
* + 1. machine/translate.cc - Machine::Translate(int virtAddr, int\* physAddr, int size, bool writing)

在原本的nachos裡面並沒有使用virtual memory，virtual address會和physical address相同，但是這次作業所要執行的檔案太大沒辦法放進main memory裡面，所以必須使用virtual memory，因此需要做virtual address和physical address的轉換，並且需要去處理page fault的問題，而這部分也是這次作業需要實作的其中一個部分。

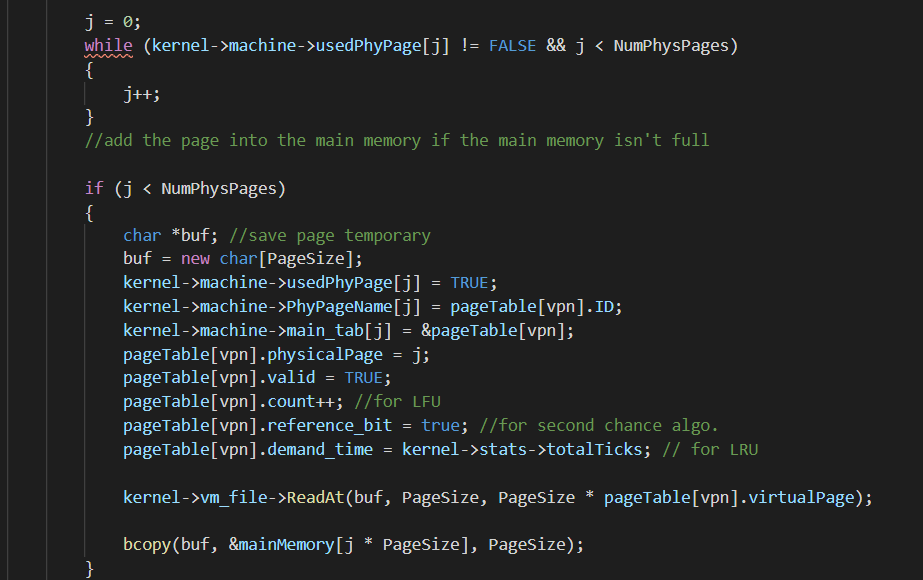


首先宣告一些會用到variable，vpn是virtAddr在virtual memory中的page number，offset是virtAddr在這個page中的位置，entry則是記錄這個page在pageTable中的entry，pageFrame是紀錄virtAddr會被存在main memory中的哪一個page的page number。

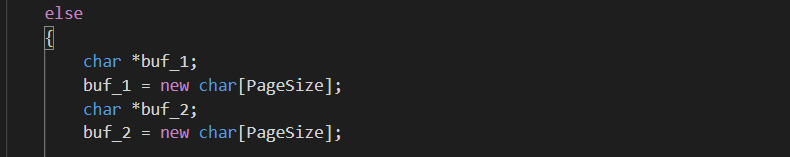
victim是用page replacement時去選擇哪一個page要被進行交換，fifo是用在first in first out的page replacement中，access\_time是去紀錄總共對main memory讀取了幾次，主要是用在 LFU的page replacement中。



再來便是去算出virtAddr是位於哪個page裡面 (pageTable的index)和在page中的哪一個位置，並且把它們存到vpn、offset裡面。接著去判斷這個vpn是否是合理，如果不合理就會return AddressErrorException，如果合理才會去判斷是否在main memory中(根據valid bit)，如果不在就會發生page fault，而原本的nachos只有return PageFaultException並沒有進行處理，但我們需要去判斷main memory中的page是否滿了以及去完成page replacement algorithm來把page從virtual memory存進main memory。



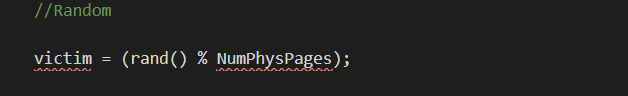
上面的while是用來判斷目前main memory中已經放了多少個page。如果還有空位(j < NumPhysPages)就使用ReadAt()把page從virtual memory讀出來放到buf內，接著使用bcopy()把buf存回main memory的對應位置中。再放入之前需要將對應usedPhyPage、PhyPagName、physicalPage進行更新，並且把main\_tab對應的entry指向這個page的pageTable entry。此外還需要把這個page的valid bit改成true，並且把一些用於page replacement algorithm的variable(count、reference\_bit、demand\_time)進行更新。



如果main memory滿了我就需要使用 page replacement algorithm找出需要被替換的page。首先，宣告兩個buffer用來存需要被移出的page和需要被移入的page。

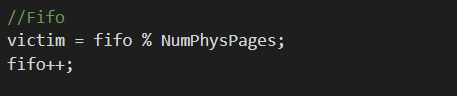
Replacement algorithm :

* + - 1. Random



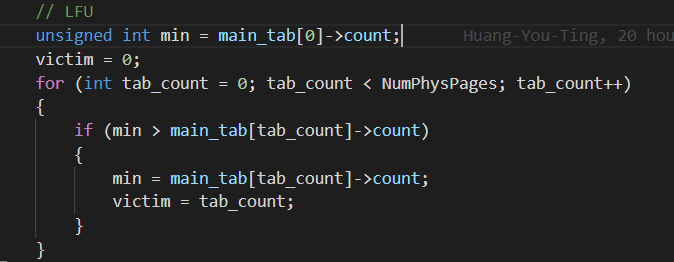
Random policy很簡單就是使用rand()去隨機選出要被換掉的page。

* + - 1. First in first out

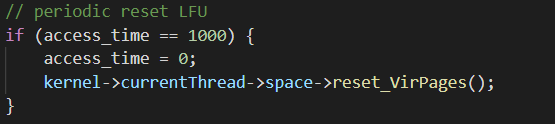


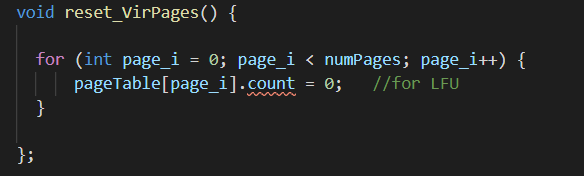
因為我們在把page放入main memory時是從頭按照順序放的，所以我們只要使用static來宣告fifo，並且每次都加1就可以完成first in first out policy。

* + - 1. Least frequency used (LFU)



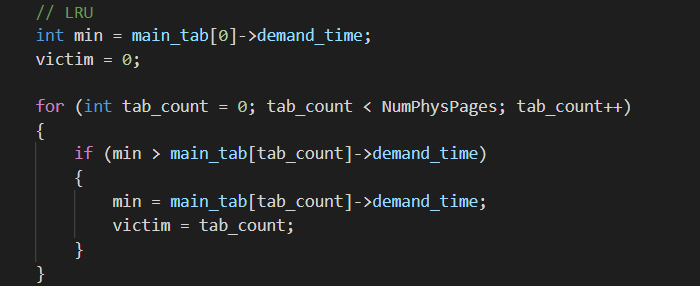
因為每次access main memory的時候都會去更新每個page的count，紀錄這個page總共被access了幾次。所以我們只要去比較每一個page的count就可以找出用最少次的page，並且把他定為被交換的Page (victim)就好。





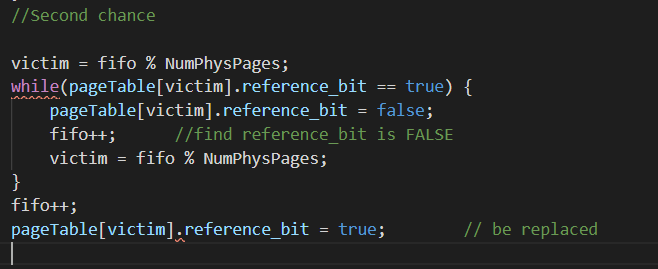
但是LFU有一個問題就是容易受到以前用過很多次的page影響，但這個page很可能所屬的process已經完成了所以不會再被用到，但因為他的count很大所以很可能會一直佔著main memory，因此我們需要定期去把count歸零，如此一來除了可以避免這個問題，還可以避免count overflow的問題。

* + - 1. Least recent used (LRU)



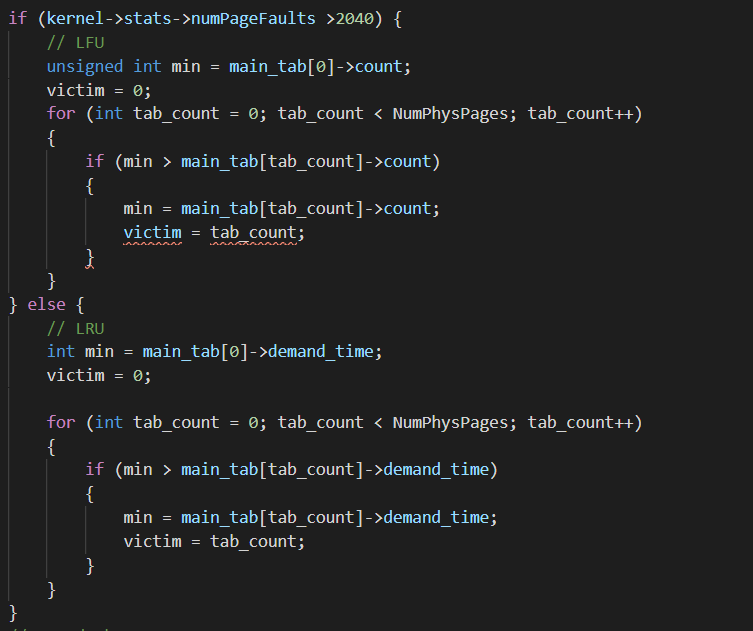
我們會在access main memory page時把當下的totalTicks存到demand\_time裡面，所以我們就可以根據 demand\_time去找出最久以前被access的page進行交換，而demand\_time會是越新的時間越大，越久以前的越小，因此只要找出demand\_time最小的page進行交換即可。

* + - 1. Second chance

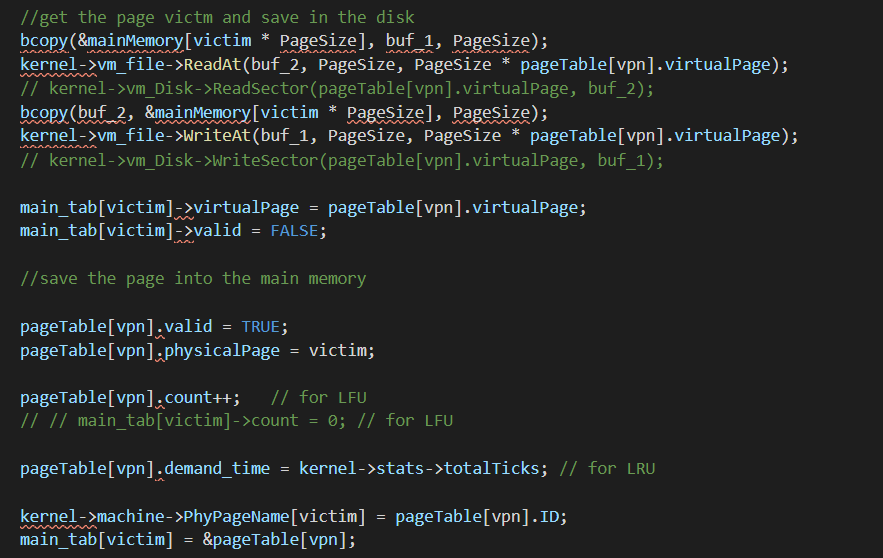


跟fifo類似，只是如果使用fifo找到的victim的reference\_bit是true，代表是first chance，所以我們只會將他的 reference\_bit改成false並繼續找下一個victim直到找到victim的reference\_bit是false為止。

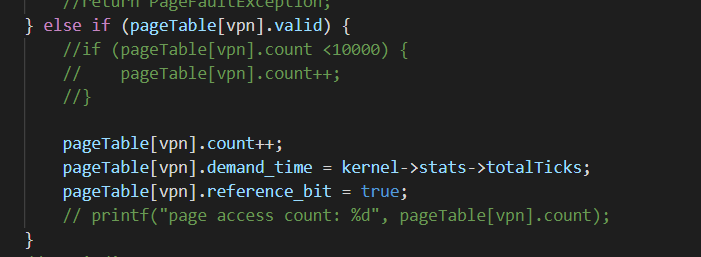
* + - 1. Mix policy (LRU + LFU)



簡單來說就是在較少page fault時採用LRU，較多page fault時採用LFU，至於如此實作的原因會在Analysis中說明



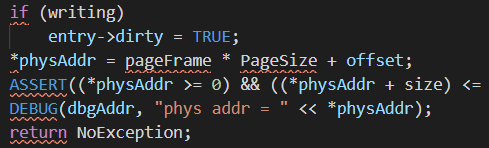
選好victim後，我們一樣使用bcopy()、ReadAt()、WriteAt()去把要換出去的page讀出放回virtual memory以及將要放入的page放入main memory中，把main\_tab對應的entry指向放入page的pageTable entry。一樣我們也需要去更新page的valid bit和一些replacement policy會用到的variable。



如果這個page原本就在main memory中，我們就只需要更新 replacement policy會用到的variable。







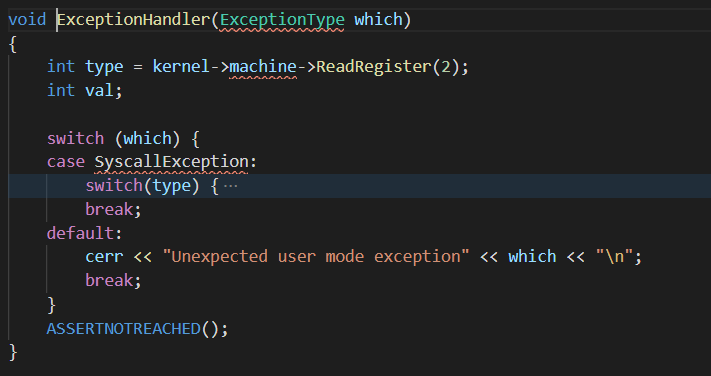
最後就只要使用找到的physicalPage number跟offset就可以算出physical addresss了，並且 return NoException。

* + 1. machine/machine.cc - Machine::RaiseException(ExceptionType which, int badVAddr)



Machine::RaiseException()因為要進行Exception的執行，所以需要先將OS設定成SystemMode，然後呼叫ExceptionHandler()去分辨是哪種exception需要處理，執行完後再將OS改回UserMode。

* + 1. userprog/exception.cc - ExceptionHandler(ExceptionType which)



ExceptionHandler()會去區分是哪種exception要執行並且去呼叫對應function來完成，而如果有發生 AddressErrorException、PageFaultException等exception時因為不屬於 SyscallException，所以只會使用cerr將訊息印出來(page fault我們在 Translate()中已經處理)。

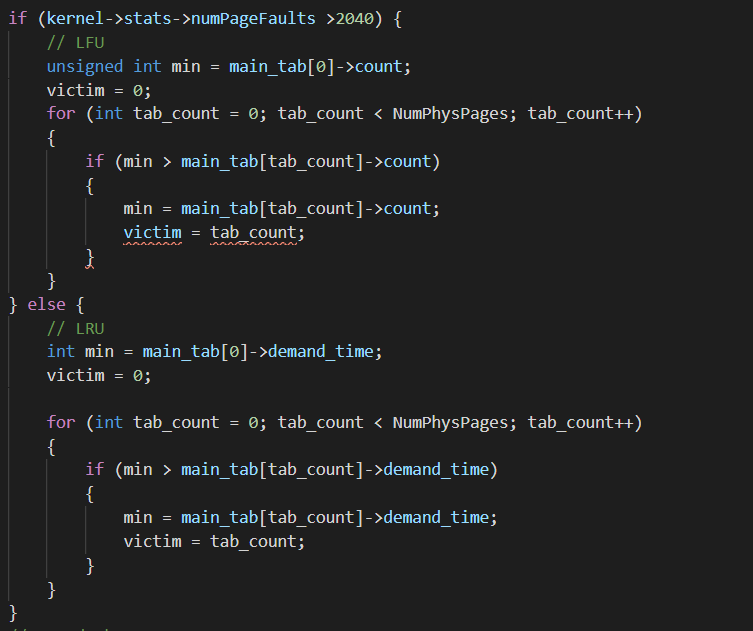
1. Analysis
   1. 不同page replacement algorithm page fault數量比較

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | random | fifo | LFU | LRU | Second chance |
| merge+quick+bubble | 20790 | 43130 | 33784 | 34847 | 28949 |
| bubble | 15970 | 39939 | 9046 | 33594 | 21067 |
| merge | 410 | 370 | 606 | 343 | 373 |
| quick | 128 | 103 | 199 | 82 | 98 |
| Multi-process overhead | 4282 | 2718 | 23933 | 828 | 7411 |

這次作業目標是讓multi-process page fault越少越好，從上表可以發現merge、quick、bubble**三個process一起跑的時候表現最佳的algorithm是random**，但是各別執行的話**bubble用LFU最佳**，**merge、quick用LRU最佳**。可以得出一個結論就是各個algorithm在跑單一的process時都有各自擅長的部份，但是在multi-process下，因為不同process會不斷進行switch，所以不僅導致優勢不見，還會產生額外的overhead，使得最後random的表現最好。

因此我們就想說把各自process表現最佳的algorithm結合起來另一方面盡量讓overhead可以降低，根據表格內可以發現overhead最低的algorithm是LRU，所以我們就想把LRU和LFU結合。

從時間複雜度上來說可以知道merge ≤ quick < bubble，而且bubble與另外兩者的運行時間會叫起來長許多，所以可以把multi-process三個一起跑的時間想成兩階段，第一階段是三者會不斷進行switch交錯運行直到merge、quick結束，第二階段是只有bubble自己運行，所以我們就想讓**第一階段的時候使用LRU**，因為它除了在merge、quick的表現最佳之外，在overhead的表現也是最佳的，而**第二階段就是使用LFU**。而兩階段的分別方法是去判斷page fault的數量，當page fault的數量大於一定數值後就是進入第二階段了，所以我們的policy就會如下圖。



* 1. 比較mix policy和random policy的page fault數量

|  |  |  |
| --- | --- | --- |
|  | random | Mix policy (LRU+LFU) |
| merge+quick+bubble | 20790 | 10842 |
| bubble | 15970 | 10502 |
| merge | 410 | 343 |
| quick | 128 | 82 |

由上表數據顯示，我們的猜想是沒錯的，LRU+LFU的policy表現最佳。

* 1. page size不同大小比較 ，main memory總量都是4096 byte(都跑multi-process)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Page Size | 512byte | 256byte | 128 bytes | 64 bytes | 32 bytes |
| Page fault | 371856 | 39054 | 10842 | 56833 | 124774 |

由上表可以發現大致上是page size越大，page fault越少。可是在超過一定大小後page fault就會又開始增加，儘管大的page size在單一process時，page fault較少，但在 multi-process的時候這個優勢就不見了。

* 1. 結論:

我們設定page size為128 byte，使用 LRU+LFU的policy表現最佳。

1. 小組分工

黃友廷 : code、report (page fault)

陳兆廷 : code、report (load program)