

# Operating System HW – Multiprogramming and Page Replacement

Group 18

106061132 黃友廷 106061146 陳兆廷

## 1. Code Tracing and Implementation

### a. Declaration and Definition

#### i. machine/machine.h

```
const unsigned int PageSize = 128;    // set the page size equal to
// the disk sector size, for simplicity
```

```
const unsigned int NumPhysPages = 32;
const unsigned int NumVirPages = 512; // 128;
const int MemorySize = (NumPhysPages * PageSize);
const int TLBSize = 4;                // if there is a TLB, make it small
```

```
TranslationEntry *pageTable;
unsigned int pageTableSize;
bool ReadMem(int addr, int size, int* value);
bool usedPhyPage[NumPhysPages]; // record which the page in the main me
bool usedVirPage[NumVirPages];
int ID_num;
int PhyPageName[NumPhysPages];
/* ...
TranslationEntry *main_tab[NumPhysPages];
```

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 共用了哪些頁的 virtual memory page。

pageTable 是用來記錄整個 virtual 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。

```
kernel->machine->main_tab[j] = &pageTable[vpn];
pageTable[vpn].physicalPage = j;
pageTable[vpn].valid = TRUE;
```

#### ii. userprog/userkernel.h

```

class UserProgKernel : public ThreadedKernel {
public:
    UserProgKernel(int argc, char **argv);
    // Interpret command line arguments
    ~UserProgKernel(); // deallocate the kernel

    void Initialize(); // initialize the kernel

    void Run(); // do kernel stuff

    void SelfTest(); // test whether kernel is working
    // Zhao Ting Chen, 3 days ago • update
    OpenFile *vm_file;
    // These are public for notational convenience.
};

```

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。

```

void UserProgKernel::Initialize()
{
    ThreadedKernel::Initialize(); // init multithreading

    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();

    ASSERT(fileSystem->Create("VMFILE"));
    vm_file = fileSystem->Open("VMFILE");
    //printf("pass vm file open\n");
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS
}

```

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。

### iii. machine/translate.h

```

class TranslationEntry {
public:
    unsigned int virtualPage; // The page number in virtual memory.
    unsigned int physicalPage; // The page number in real memory (relative to
    // start of "mainMemory"
    bool valid; // If this bit is set, the translation is ignored.
    // (In other words, the entry hasn't been initialized.)
    bool readOnly; // If this bit is set, the user program is not allowed
    // to modify the contents of the page.
    bool use; // This bit is set by the hardware every time the
    // page is referenced or modified.
    bool dirty; // This bit is set by the hardware every time the
    // page is modified.
    unsigned int count; //for LFU

    unsigned int demand_time; // for LRU
    bool reference_bit; //for second chance algo.

    int ID;
};

```

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。

b. Load program

i. 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。

```
void UserProgKernel::Run()
{
    cout << "Total threads number is " << execfileNum << endl;
    for (int n = 1; n <= execfileNum; n++)
    {
        t[n] = new Thread(execfile[n]);
        t[n]->space = new AddrSpace();
        t[n]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[n]);
        cout << "Thread " << execfile[n] << " is executing." << endl;
    }
}
```

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。

ii. 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。

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++)
    {
        pageTable[i].virtualPage = i; // for now, virt page # = phys page #
        pageTable[i].physicalPage = i;
        // pageTable[i].physicalPage = 0;
        pageTable[i].valid = TRUE;
        // pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
    // bzero(kernel->machine->mainMemory, MemorySize);
}
```

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。

```
AddrSpace::AddrSpace()
{
    ID = (kernel->machine->ID_num) + 1;
    kernel->machine->ID_num = kernel->machine->ID_num + 1;
    // zero out the entire address space
    // bzero(kernel->machine->mainMemory, MemorySize);
}
```

iii. threads/kernel.cc - ThreadedKernel::Run()

直接進入 Thread::Finish()。

iv. threads/thread.cc - Thread::Finish()

```

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

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

    Sleep(TRUE); // invokes SWITCH
    // not reached
}

```

模擬正在執行 Thread 的過程，進入 Sleep。

- v. threads/thread.cc - Thread::Sleep (bool finishing)

```

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

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

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

    status = BLOCKED;
    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);
}

```

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。

- vi. 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。

```

kernel->currentThread = nextThread; // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running

DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

// This is a machine-dependent assembly language routine defined
// in switch.s. You may have to think
// a bit to figure out what happens after this, both from the point
// of view of the thread and from the perspective of the "outside world".

SWITCH(oldThread, nextThread);

```

除了將 nextThread 的狀態設為 Running，進入 SWITCH。

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

如同前次作業的交換：

```

/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp) ->      thread *t2
**      4(esp) ->      thread *t1
**      (esp) ->      return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/

```

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。

```
336      .comm  _eax_save,4
337
338      .globl  SWITCH
339  SWITCH:
340      movl    %eax,_eax_save      # save the value of eax
341      movl    4(%esp),%eax        # move pointer to t1 into eax
342      movl    %ebx,_EBX(%eax)    # save registers
343      movl    %ecx,_ECX(%eax)
344      movl    %edx,_EDX(%eax)
345      movl    %esi,_ESI(%eax)
346      movl    %edi,_EDI(%eax)
347      movl    %ebp,_EBP(%eax)
348      movl    %esp,_ESP(%eax)    # save stack pointer
349      movl    _eax_save,%ebx      # get the saved value of eax
350      movl    %ebx,_EAX(%eax)    # store it
351      movl    0(%esp),%ebx       # get return address from stack into ebx
352      movl    %ebx,_PC(%eax)     # save it into the pc storage
353
354      movl    8(%esp),%eax        # move pointer to t2 into eax
355
356      movl    _EAX(%eax),%ebx     # get new value for eax into ebx
357      movl    %ebx,_eax_save     # save it
358      movl    _EBX(%eax),%ebx    # restore old registers
359      movl    _ECX(%eax),%ecx
360      movl    _EDX(%eax),%edx
361      movl    _ESI(%eax),%esi
362      movl    _EDI(%eax),%edi
363      movl    _EBP(%eax),%ebp
364      movl    _ESP(%eax),%esp     # restore stack pointer
365      movl    _PC(%eax),%eax     # restore return address into eax
366      movl    %eax,4(%esp)       # copy over the ret address on the stack
367      movl    _eax_save,%eax
368
369      ret
```

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。

viii. 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。

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

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

直接進入 AddrSpace::Execute()。

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

```

void AddrSpace::Execute(char *fileName)
{
    pt_is_load = FALSE;
    if (!Load(fileName))
    {
        cout << "inside !Load(fileName)" << endl;
        return; // executable not found
    }

    //kernel->currentThread->space = this;
    this->InitRegisters(); // set the initial register values
    this->RestoreState(); // load page table register
    pt_is_load = TRUE;
    kernel->machine->Run(); // jump to the user program

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

```

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。

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

```

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

unsigned int size, k;

if (executable == NULL)
{
    cerr << "Unable to open file " << fileName << "\n";
    return FALSE;
}
executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
if ((noffH.noffMagic != NOFFMAGIC) &&
    (WordToHost(noffH.noffMagic) == NOFFMAGIC))
    SwapHeader(&noffH);
ASSERT(noffH.noffMagic == NOFFMAGIC);

```

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。

```

static void
SwapHeader(NoffHeader *noffH)
{
    noffH->noffMagic = WordToHost(noffH->noffMagic);
    noffH->code.size = WordToHost(noffH->code.size);
    noffH->code.virtualAddr = WordToHost(noffH->code.virtualAddr);
    noffH->code.inFileAddr = WordToHost(noffH->code.inFileAddr);
    noffH->initData.size = WordToHost(noffH->initData.size);
    noffH->initData.virtualAddr = WordToHost(noffH->initData.virtualAddr);
    noffH->initData.inFileAddr = WordToHost(noffH->initData.inFileAddr);
    noffH->uninitData.size = WordToHost(noffH->uninitData.size);
    noffH->uninitData.virtualAddr = WordToHost(noffH->uninitData.virtualAddr);
    noffH->uninitData.inFileAddr = WordToHost(noffH->uninitData.inFileAddr);
}

```

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。

```
// how big is address space?
size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize; // we need to increase
// to leave room for th
numPages = divRoundUp(size, PageSize);
// cout << "number of pages of " << fileName<< " is "<<numPages<<endl;

pageTable = new TranslationEntry[numPages];

size = numPages * PageSize;
```

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。

```
int pageIndex = 0;
while ((pageIndex + 1) * PageSize < noffH.code.size)
{
    PutInPageTable(pageIndex, executable, pageTable, noffH.code.inFileAddr, 0, pageIndex);
    pageIndex++;
}

int offset = noffH.code.size - pageIndex * PageSize;
// cout << "offset = " << offset << endl;

if (noffH.initData.size == 0)
{
    while (pageIndex < numPages)
    {
        PutInPageTable(pageIndex, executable, pageTable, noffH.code.inFileAddr, 0, pageIndex);
        pageIndex++;
    }
}
else
{
    PutInPageTableWithOffset(pageIndex, executable, pageTable, noffH.code.inFileAddr, noffH.initData.inFileAddr, offset);
    pageIndex++;

    int InitDataPageIndex = pageIndex;
    // while ((pageIndex + 1) * PageSize < noffH.code.size + noffH.initData.size)
    while (pageIndex < numPages)
    {
        PutInPageTable(pageIndex, executable, pageTable, noffH.initData.inFileAddr, PageSize - offset, pageIndex - InitDataPageIndex);
        // printf("Put page %d of InitData + %d at Page %d.\n", pageIndex - InitDataPageIndex, PageSize - offset, pageIndex);
        pageIndex++;
    }
}
```

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。

```
void AddrSpace::PutInPageTable(int i, OpenFile *executable, TranslationEntry *pageTable, int Addr, int Start, int i_2)
{
    char *buf;
    buf = new char[PageSize];
    int k = 0;
    while (kernel->machine->usedvirPage[k] != FALSE)
    {
        k++;
    }

    kernel->machine->usedvirPage[k] = true;
    pageTable[i].virtualPage = k; //record which virtualpage you save
    pageTable[i].valid = FALSE; //not load in main_memory
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    pageTable[i].ID = ID;
    pageTable[i].count = 0;
    pageTable[i].reference_bit = true;
    executable->ReadAt(buf, PageSize, Start + Addr + (i_2 * PageSize));
    // printf("enter writeat\n");
    kernel->vm_file->WriteAt(buf, PageSize, k * PageSize);
    // printf("pass putintable\n");
    // kernel->vm_Disk->WriteSector(k, buf); //call virtual_disk write in virtual memory
}
```

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。



```

executable->ReadAt(buf1, offset, Addr + (i * PageSize));
executable->ReadAt(buf2, PageSize - offset, Addr2);
buf = concat(buf1, buf2, offset);
kernel->vm_file->WriteAt(buf, PageSize, k * PageSize);

```

而 code 與 Data 中間的交接處，就使用變數處理好。

### c. Page Faults

#### i. machine/mipssim.cc - Machine::Run()

```

void Machine::Run()
{
    Instruction *instr = new Instruction; // storage for decoded instruction

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}

```

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。

#### ii. machine/ mipssim.cc - Machine::OneInstruction(Instruction \*instr)

```

void Machine::OneInstruction(Instruction *instr)
{
    int raw;
    int nextLoadReg = 0;
    int nextLoadValue = 0; // record delayed load operation, to apply
                          // in the future

    // Fetch instruction
    if (!ReadMem(registers[PCReg], 4, &raw))
        return; // exception occurred
    instr->value = raw;
    instr->Decode();
    // printf("pass ReadMem\n");
    if (debug->IsEnabled('m')) { ...
}

```

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。

```

// Execute the instruction (cf. Kane's book)
switch (instr->opCode) { ...

    // Now we have successfully executed the instruction.

    // Do any delayed load operation
    DelayedLoad(nextLoadReg, nextLoadValue);

    // Advance program counters.
    registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we
    registers[PCReg] = registers[NextPCReg]; // are jumping into lala-land
    registers[NextPCReg] = pcAfter;
}

```

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。



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

```
bool Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG(dbgAddr, "Reading VA " << addr << ", size " << size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException)
    {
        RaiseException(exception, addr);
        return FALSE;
    }
}
```

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。

```
switch (size)
{
case 1:
    data = mainMemory[physicalAddress];
    *value = data;
    break;

case 2:
    data = *(unsigned short *)&mainMemory[physicalAddress];
    *value = ShortToHost(data);
    break;

case 4:
    data = *(unsigned int *)&mainMemory[physicalAddress];
    *value = WordToHost(data);
    break;

default:
    ASSERT(FALSE);
}

DEBUG(dbgAddr, "\tvalue read = " << *value);
return (TRUE);
}
```

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

iv. 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。

```
ExceptionType Machine::Translate(int virtAddr, int *physAddr, int size, bool writing)
{
    int i;
    unsigned int vpn, offset;
    TranslationEntry *entry;
    unsigned int pageFrame;

    int victim; ///find the page victim
    static int fifo; ///For fifo
    > /*...
    static int access_time = 0;
    access_time++;
}
```

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。

```
// calculate the virtual page number, and offset within the page,
// from the virtual address
vpn = (unsigned)virtAddr / PageSize;
offset = (unsigned)virtAddr % PageSize;

if (tlb == NULL)
{ // => page table => vpn is index into table
    if (vpn >= pageTableSize)
    {
        DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
        printf("pageTable: %d vpn: %d\n", pageTableSize, vpn);
        return AddressErrorException;
    }
    else if (!pageTable[vpn].valid)
    {
        //handle page fault
        // DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
        kernel->stats->numPageFaults++;
    }
}
```

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。

```
j = 0;
while (kernel->machine->usedPhyPage[j] != FALSE && j < NumPhysPages)
{
    j++;
}
//add the page into the main memory if the main memory isn't full

if (j < NumPhysPages)
{
    char *buf; //save page temporary
    buf = new char[PageSize];
    kernel->machine->usedPhyPage[j] = TRUE;
    kernel->machine->PhyPageName[j] = pageTable[vpn].ID;
    kernel->machine->main_tab[j] = &pageTable[vpn];
    pageTable[vpn].physicalPage = j;
    pageTable[vpn].valid = TRUE;
    pageTable[vpn].count++; //for LFU
    pageTable[vpn].reference_bit = true; //for second chance algo.
    pageTable[vpn].demand_time = kernel->stats->totalTicks; // for LRU

    kernel->vm_file->ReadAt(buf, PageSize, PageSize * pageTable[vpn].virtualPage);

    bcopy(buf, &mainMemory[j * PageSize], PageSize);
}
```

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、PhyPageName、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。

```

else
{
    char *buf_1;
    buf_1 = new char[PageSize];
    char *buf_2;
    buf_2 = new char[PageSize];
}

```

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
victim = (rand() % NumPhysPages);

```

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

#### (2.) First in first out

```

//Fifo
victim = fifo % NumPhysPages;
fifo++;

```

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。

#### (3.) Least frequency used (LFU)

```

// LFU
unsigned int min = main_tab[0]->count;
victim = 0;
for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
{
    if (min > main_tab[tab_count]->count)
    {
        min = main_tab[tab_count]->count;
        victim = tab_count;
    }
}

```

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。

```

// periodic reset LFU
if (access_time == 1000) {
    access_time = 0;
    kernel->currentThread->space->reset_VirPages();
}

```

```

void reset_VirPages() {
    for (int page_i = 0; page_i < numPages; page_i++) {
        pageTable[page_i].count = 0; //for LFU
    }
};

```

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。

#### (4.) Least recent used (LRU)

```
// LRU
int min = main_tab[0]->demand_time;
victim = 0;

for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
{
    if (min > main_tab[tab_count]->demand_time)
    {
        min = main_tab[tab_count]->demand_time;
        victim = tab_count;
    }
}
```

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。

#### (5.) Second chance

```
//Second chance
victim = fifo % NumPhysPages;
while(pageTable[victim].reference_bit == true) {
    pageTable[victim].reference_bit = false;
    fifo++; //find reference_bit is FALSE
    victim = fifo % NumPhysPages;
}
fifo++;
pageTable[victim].reference_bit = true; // be replaced
```

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。

#### (6.) Mix policy (LRU + LFU)

```
if (kernel->stats->numPageFaults > 2040) {
    // LFU
    unsigned int min = main_tab[0]->count;
    victim = 0;
    for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
    {
        if (min > main_tab[tab_count]->count)
        {
            min = main_tab[tab_count]->count;
            victim = tab_count;
        }
    }
} else {
    // LRU
    int min = main_tab[0]->demand_time;
    victim = 0;

    for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
    {
        if (min > main_tab[tab_count]->demand_time)
        {
            min = main_tab[tab_count]->demand_time;
            victim = tab_count;
        }
    }
}
```

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

```

//get the page victim and save in the disk
bcopy(&mainMemory[victim * PageSize], buf_1, PageSize);
kernel->vm_file->ReadAt(buf_2, PageSize, PageSize * pageTable[vpn].virtualPage);
// kernel->vm_Disk->ReadSector(pageTable[vpn].virtualPage, buf_2);
bcopy(buf_2, &mainMemory[victim * PageSize], PageSize);
kernel->vm_file->WriteAt(buf_1, PageSize, PageSize * pageTable[vpn].virtualPage);
// kernel->vm_Disk->WriteSector(pageTable[vpn].virtualPage, buf_1);

main_tab[victim]->virtualPage = pageTable[vpn].virtualPage;
main_tab[victim]->valid = FALSE;

//save the page into the main memory

pageTable[vpn].valid = TRUE;
pageTable[vpn].physicalPage = victim;

pageTable[vpn].count++; // for LFU
// // main_tab[victim]->count = 0; // for LFU

pageTable[vpn].demand_time = kernel->stats->totalTicks; // for LRU

kernel->machine->PhyPageName[victim] = pageTable[vpn].ID;
main_tab[victim] = &pageTable[vpn];

```

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。

```

//return PageFaultException;
} else if (pageTable[vpn].valid) {
    //if (pageTable[vpn].count < 10000) {
    //    pageTable[vpn].count++;
    //}

    pageTable[vpn].count++;
    pageTable[vpn].demand_time = kernel->stats->totalTicks;
    pageTable[vpn].reference_bit = true;
    // printf("page access count: %d", pageTable[vpn].count);
}

```

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。

```
entry = &pageTable[vpn];
```

```
pageFrame = entry->physicalPage;
```

```

if (writing)
    entry->dirty = TRUE;
*physAddr = pageFrame * PageSize + offset;
ASSERT((*physAddr >= 0) && ((*physAddr + size) <=
DEBUG(dbgAddr, "phys addr = " << *physAddr);
return NoException;

```

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

- v. machine/machine.cc - Machine::RaiseException(ExceptionType which, int badVAddr)

```

void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);

    registers[badVAddrReg] = badVAddr;
    DelayedLoad(0, 0); // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
    // cout << "entering system mode...\n";
    ExceptionHandler(which); // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
    // cout << "entering user mode...\n";
}

```

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。

vi. userprog/exception.cc - ExceptionHandler(ExceptionType which)

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    int val;

    switch (which) {
    case SyscallException:
        switch(type) { ...
        break;
    default:
        cerr << "Unexpected user mode exception" << which << "\n";
        break;
    }
    ASSERTNOTREACHED();
}
```

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)。

## 2. Analysis

### a. 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  $\text{merge} \leq \text{quick} < \text{bubble}$ ，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。

```

if (kernel->stats->numPageFaults > 2040) {
    // LFU
    unsigned int min = main_tab[0]->count;
    victim = 0;
    for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
    {
        if (min > main_tab[tab_count]->count)
        {
            min = main_tab[tab_count]->count;
            victim = tab_count;
        }
    }
} else {
    // LRU
    int min = main_tab[0]->demand_time;
    victim = 0;

    for (int tab_count = 0; tab_count < NumPhysPages; tab_count++)
    {
        if (min > main_tab[tab_count]->demand_time)
        {
            min = main_tab[tab_count]->demand_time;
            victim = tab_count;
        }
    }
}
}

```

b. 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。

c. 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。

d. 結論:

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

3. 小組分工

黃友廷：code、report (page fault)

陳兆廷：code、report (load program)