312505017 馮凡哲
312511057 林胤汶

1. Trace Code

- **threads/kernel.cc**
  首先觀察Kernel::Kernel(),它用來解釋command line參數(argc和argv),以確定在初始化過程中應該使用的flags。透過檢查command line參數(argv)中的不同選項(如-rs、-s、-e等)來設置Kernel的相對應選項。
  再來從void Kernel::ExecAll()開始看:

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

  在for迴圈中,它對 execfile陣列的每一項執行了Exec()函式:

```
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;
```

  Exec()創建一個新的 Thread, 並為其分配一個地址空間(AddrSpace), 並調用 Fork 函式, 將 ForkExecute 函式作為參數傳遞給新創建的Thread, 以實現新Thread的執行。

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

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

}
```

  ForkExecute 函式傳入一個指向 Thread的指標 t, 它會呼叫addrspace.cc裡面的Load函式, 將要執行的程式載入Memory中。如果載入失敗, 則直接返回, 否則執行 addrspace.cc裡面的Execute 函式。

- **userprog/addrspace.cc**
  AddrSpace::AddrSpace()用來建立address space以執行user program , 並轉換program memory 到 physical memory。
  再來看 Kernel::ForkExecute ()呼叫的Load和Execute, Load函式主要是用來將user program從file載入到memory中:

```
bool
AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

    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);
```

這部分主要進行了檔案的開啟Open()及相關資訊的讀取ReadAt(),並檢查其格式,若格式不符則使用SwapHeader()進行轉換。

Execute函式:

```
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 progam

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

這邊會將目前thread的定址空間與caller 做link,接著使用InitRegisters()函數初始化user registers,再用RestoreState()
載入這個程式所對應的page table,最後呼叫machine->Run來執行程式。

```
void
AddrSpace::InitRegisters()
{
    Machine *machine = kernel->machine;
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);
    machine->WriteRegister(NextPCReg, 4);
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
    DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}
```

InitRegisters()用來初始化user registers,可以看到函式內部主要是進行暫存器的寫入。

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

RestoreState()中可以看到它指定了 pagetable 和 pagetablesize的值。

- **threads/thread.cc**

在來看Kernel::Exec()中呼叫的 Fork():

```
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);
}
```

主要做三件事

1. 配置Stack
2. 初始化Stack
3. 將thread放入ready queue

其中的StackAllocate():

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

首先配置Stack的空間,之後再針對不同的結構進行初始化。

```
#ifdef PARISC
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif
#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
    *stack = STACK_FENCEPOST;
#endif
#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif
#ifdef DECMIPS
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif
#ifdef ALPHA
    stackTop = stack + StackSize - 8;   // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif
#ifdef x86
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif

#ifdef PARISC
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

回到Kernel::ExecAll(),最後執行了currentThread->Finish():

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

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

由於目前的Thread仍在執行中且位於其Thread Stack上，無法立即釋放。因此，這個函式告知Scheduler在不同Thread的context中運行時呼叫Thread的destructor以完成資源的釋放。

再來看到Sleep():

```
void
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);
}
```

可以看到其中ASSERT中斷已被關閉,所以先前的Finish()才會有關閉中斷的動作(kernel->interrupt->SetLevel(IntOff))。

While判斷nextThread = kernel->scheduler->FindNextToRun(),看是否還有下一條Thread要執行;

若有,則通過kernel->scheduler->Run()繼續執行。

若沒有,則呼叫kernel->interrupt->Idle()使CPU進入閒置狀態。

- **threads/scheduler.cc**

回到Thread::Fork()中呼叫的ReadyToRun:

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

主要是將thread的狀態設為ready,並將thread加入到readyList。

再來看到Sleep()中用來進行判斷的FindNextToRun():

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyList->IsEmpty()) {
            return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

這個函式會檢查readyList是否非空並回傳下一個ready thread。

再看到Run():

```
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();
    }

    oldThread->CheckOverflow();              // check if the old thread
                                             // had an undetected stack overflow
```

首先保存了當前的thread,再來判斷當前thread是否需要刪除及是否是userprogram 要進行相關資訊的儲存,然後檢查了當前thread是否overflow。

```
    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);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    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();
    }
}
```

接著進行thread的切換, 並檢查先前執行的thread是否已經結束, 需要進行清理,最後恢復舊的thread保存的相關資訊。

- **Questions:**
  1. **Explain how NachOS creates a thread (process), load it into memory and place it into the scheduling queue**.
     首先在Kernel::ExecAll()中創建了新的Thread結構,並配置空間給該結構,再來經由ForkExecute中的Load,將要執行的程式載入Memory,最後再由Fork中呼叫的ReadyToRun將thread的狀態設為ready,並將thread加入到Ready Queue。

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

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

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

配置memory空間主要由 AddrSpace::AddrSpace()實現,首先它建立了一個pageTable,用來處理virtual page到 physical page 的 translate

和一些相關配置。
再來使用bzero()將MainMemory內容清空,提供程式乾淨的空間。

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

ForkExecute中會呼叫Load(),該函式主要功能是將是將一個程式(object code)從檔案載入到memory中, 其中的這部分:

```
#ifdef RDATA
// how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
                                                // we need to increase the size
                                                // to leave room for the stack
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
                    + UserStackSize;            // we need to increase the size
                                                // to leave room for the stack
#endif
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);           // check we're not trying
                                                // to run anything too big --
                                                // at least until we have
                                                // virtual memory
```

會根據讀取的headfile, 計算出address space需要的大小。根據不同的情況, 計算出code、initData、uninitData和user stack的總大小。
再依據計算出的大小, 確定address space需要的頁數 numPages, 然後重新計算總大小 size。
最後在判斷code和initData大小是否大於0,如果是,則會讀取其相對應的資料, 並將其載入到虛擬記憶體中對應的位置,以下為initData部分:

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

4. **How does Nachos create and manage the page table?**

在machine/translate.h中定義了class TranslationEntry:

```
class TranslationEntry {
  public:
    int virtualPage;    // The page number in virtual memory.
    int physicalPage;   // The page number in real memory (relative to the
                        //  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.
};
```

它定義了一些項目,這些項目可以用於page table 或 TLB。
在addrspace.h中,定義了pageTable:

```
private:
  TranslationEntry *pageTable;        // Assume linear page table translation
                                      // for now!
```

之後就可以對pageTable進行一些相關的處理。

5. **How does Nachos translate addresses?**
在machine.h和addrspace.cc皆宣告了
ExceptionType Translate(),分別是:

```
ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
                                // Translate an address, and check for
                                // alignment.  Set the use and dirty bits in
                                // the translation entry appropriately,
                                // and return an exception code if the
                                // translation couldn't be completed.
```

```
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)
```

兩者都是用來執行virtual address到physical address的轉換,我認為它們的差別在於使用對象不同。machine.h宣告的Translate()用於硬體層面(page table,TLB)的的位址轉換,而addrspace.cc中的則是主要用於處理process(page table)的位址轉換。

6. **How Nachos initializes the machine status (register, etc) before running a thread (process)**
machine status的初始化主要由addrspace:: InitRegisters()和Thread::StackAllocation()實現。
InitRegisters()主要進行Registers的初始化:

```
void
AddrSpace::InitRegisters()
{
    Machine *machine = kernel->machine;
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);
    machine->WriteRegister(NextPCReg, 4);
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
    DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}
```

StackAllocation()主要進行Stack的初始化:

```
#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;      // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
                                        // least 1 activation record
                                        // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif

#ifdef ALPHA
    stackTop = stack + StackSize - 8;   // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif


#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif

#ifdef PARISC
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
```

7. **Which object in Nachos acts the role of process control block**
   在 Nachos 中，用來代表Process Control Block(PCB)的物件是Thread。

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int *stackTop;                      // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

  public:
    Thread(char* debugName, int threadID);              // initialize a Thread
    ~Thread();                          // deallocate a Thread
                                        // NOTE -- thread being deleted
                                        // must not be running when delete
                                        // is called

    // basic thread operations

    void Fork(VoidFunctionPtr func, void *arg);
                                // Make thread run (*func)(arg)
    void Yield();               // Relinquish the CPU if any
                                // other thread is runnable
    void Sleep(bool finishing); // Put the thread to sleep and
                                // relinquish the processor
    void Begin();               // Startup code for the thread
    void Finish();              // The thread is done executing

    void CheckOverflow();       // Check if thread stack has overflowed
    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
        char* getName() { return (name); }

        int getID() { return (ID); }
    void Print() { cout << name; }
    void SelfTest();            // test whether thread impl is working

  private:
    // some of the private data for this class is listed above

    int *stack;                 // Bottom of the stack
                                // NULL if this is the main thread
                                // (If NULL, don't deallocate stack)
    ThreadStatus status;        // ready, running or blocked
    char* name;
        int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
                                // Allocate a stack for thread.
                                // Used internally by Fork()

// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs];    // user-level CPU register state

  public:
    void SaveUserState();               // save user-level register state
    void RestoreUserState();            // restore user-level register state

    AddrSpace *space;                   // User code this thread is running.
};
```

其中包含了Stack和Machine State、Thread相關的操作函數、Thread的狀態和ID、Stack和Register相關的操作函數等。結構和內容都類似於PCB,也都含有描述和管理thread執行所需的各種資訊。

2. Implementation

- **In addrspace.cc:**

1. 初始化pagetable

```
pageTable = new TranslationEntry[numPages];
for (int i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = kernel->usedPhyPage->checkAndSet();
    pageTable[i].valid = true;
    pageTable[i].use = false;
    pageTable[i].dirty = false;
    pageTable[i].readOnly = false;

    /* zero out this physical page */
    bzero(kernel->machine->mainMemory + pageTable[i].physicalPage *
            PageSize, PageSize);
}
```

使用TranslationEntry建立pagetable並初始化每個page的參數。

2. 新增變數紀錄位置

```
unsigned int physicalAddr;
int unReadSize;
int chunkStart;
int chunkSize;
int inFilePosiotion;
```

3. if (noffH.code.size > 0)

```
unReadSize = noffH.code.size;
chunkStart = noffH.code.virtualAddr;
chunkSize = 0;
inFilePosiotion = 0;
```

```
while(unReadSize > 0) {

    chunkSize =  calChunkSize(chunkStart, unReadSize);

    Translate(chunkStart, &physicalAddr, 1);

    executable->ReadAt(
            &(kernel->machine->mainMemory[physicalAddr]),
            chunkSize, noffH.code.inFileAddr + inFilePosiotion);

    unReadSize = unReadSize - chunkSize;
    chunkStart = chunkStart + chunkSize;
    inFilePosiotion = inFilePosiotion + chunkSize;
}
```

讀取noffH提供的資訊並初始化參數，並將程式讀取進入記憶體。使用自定義的calChunkSize計算預計使用的記憶體區塊大小並使用Translate轉換虛擬地址至物理地址，再用ReadAt讀取一個chunkSize的大小至記憶體，最後更新追蹤參數。

4. if (noffH.initData.size > 0)

```
unReadSize = noffH.initData.size;
chunkStart = noffH.initData.virtualAddr;
chunkSize = 0;
inFilePosiotion = 0;
```

```
while(unReadSize > 0) {
    chunkSize =  calChunkSize(chunkStart, unReadSize);
    Translate(chunkStart, &physicalAddr, 1);
    executable->ReadAt(
            &(kernel->machine->mainMemory[physicalAddr]),
            chunkSize, noffH.initData.inFileAddr + inFilePosiotion);

    unReadSize = unReadSize - chunkSize;
    chunkStart = chunkStart + chunkSize;
    inFilePosiotion = inFilePosiotion + chunkSize;
}
```

與第3步相同差別是讀取初始資料。

5. if (noffH.readonlyData.size > 0)

```
unReadSize = noffH.readonlyData.size;
chunkStart = noffH.readonlyData.virtualAddr;
chunkSize = 0;
inFilePosiotion = 0;
```

```
while(unReadSize > 0) {
    chunkSize =  calChunkSize(chunkStart, unReadSize);
    Translate(chunkStart, &physicalAddr, 1);
    executable->ReadAt(
            &(kernel->machine->mainMemory[physicalAddr]),
            chunkSize, noffH.readonlyData.inFileAddr + inFilePosiotion);

    unReadSize = unReadSize - chunkSize;
    chunkStart = chunkStart + chunkSize;
    inFilePosiotion = inFilePosiotion + chunkSize;
}
```

與第3步相同差別是讀取唯讀資料。

6. calChunkSize function

```
int AddrSpace::calChunkSize(int chunkStart, int unReadSize)
{
    int chunkSize;
    chunkSize = (chunkStart / PageSize + 1) * PageSize - chunkStart;
    if(chunkSize > unReadSize) chunkSize = unReadSize;
    return chunkSize;
}
```

這裡是計算預計使用記憶體區段大小的function。先計算當前需要的區塊大小再確認大小是否超過未讀取的區塊大小。

- **In kernel.cc:**

1. Kernel::Initialize()中加入

```
usedPhyPage = new UsedPhyPage();
```

初始化kernal時建立usedPhyPage。

2. Kernel::~Kernel()中加入

```
delete usedPhyPage;
```

刪除kernal時，釋放usedPhyPage。

3. 定義UsedPhyPage

```
UsedPhyPage::UsedPhyPage()
{
    pages = new int[NumPhysPages];
    memset(pages, 0, sizeof(int) * NumPhysPages);
}

UsedPhyPage::~UsedPhyPage()
{
    delete[] pages;
}
```

4. 計算usedPhyPage未使用的page數量

```
int UsedPhyPage::numUnused()
{
    int count = 0;

    for(int i = 0; i < NumPhysPages; i++) {
        if(pages[i] == 0) count++;
    }
    return count;
}
```

5. 尋找能用的空間並回傳

```
int UsedPhyPage::checkAndSet()
{
    int unUsedPage = -1;

    for(int i = 90; i < NumPhysPages; i--) {
        if(pages[i] == 0) {
            unUsedPage = i;
            break;
        }
    }
    pages[unUsedPage] = 1;
    return unUsedPage;
}
```

尋找能用的page並更新page占用紀錄，最後回傳能使用的page號碼。