Operating System

MP2: Multi-Programming

Team 41 分工

Part I - Trace Code	郭逸洪、黃啟恆
Part II - Implement page table in NachOS	郭逸洪

Contents

Part I - Trace Code ... 4

A. threads/main.cc, int main(int argc, char **argv) ... 4

B. threads/kernel.cc, void Kernel::Kernel(int argc, char **argv) ... 4

C. threads/kernel.cc, void Kernel::Initialize() ... 4

D. threads/kernel.cc, void Kernel::ExecAll() ... 4

E. threads/kernel.cc, void Kernel::Exec() ... 4

F. userprog/addrspace.cc, AddrSpace::AddrSpace() ... 4

G. threads/thread.cc, void Thread::Fork(VoidFunctionPtr func, void *arg) ... 4

H. threads/thread.cc, void Thread::StackAllocate(VoidFunctionPtr func, void *arg) ... 5

I. threads/kernel.cc, void ForkExecute(Thread *t) ... 5

J. userprog/addrspace.cc, bool AddrSpace::Load(char *fileName) ... 5

K. userprog/addrspace.cc, void AddrSpace::Execute(char *fileName) ... 5

L. threads/scheduler.cc, void Scheduler::ReadyToRun (Thread *thread) ... 6

M. threads/thread.cc, void Thread::Finish() ... 6

N. threads/thread.cc, void Thread::Sleep(bool finishing) ... 6

O. machine/interrupt.cc, void Interrupt::Idle() ... 6

P. threads/scheduler.cc, void Scheduler::Run (Thread *nextThread, bool finishing) ... 6

Q. threads/scheduler.cc, void Scheduler::CheckToBeDestroyed() ... 7

R. threads/thread.cc, Thread::~Thread() ... 7

S. threads/switch.S, ThreadRoot (x86) ... 7

T. threads/thread.cc, void Thread::Begin() ... 7

Answers to code tracing questions ... 7

● How does Nachos allocate the memory space for a new thread (process)? ... 7

● How does Nachos initialize the memory content of a thread (process), including loading the user binary code in the memory? ... 7

● How does Nachos create and manage the page table? ... 8

● How does Nachos translate addresses? ... 8

● How does Nachos initialize the machine status (registers, etc) before running a thread (process)? ... 8

● Which object in Nachos acts the role of process control block? ... 8

● When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler? ... 8

● Please look at the following code from urserprog/exception.cc and answer the question:According to the code above, please explain under what circumstances an error will occur if the message size is larger than one page and why? (Hint: Consider the relationship between physical pages and virtual pages.) ... 9

Part II- Implement page table in NachOS ... 10

● code/machine/machine.h ... 10

● code/threads/kernel.h ... 10

● code/threads/kernel.cc ... 10

- code/userprog/addrspace.h... 11
- code/userprog/addrspace.cc... 12

Difficulties encountered.. 17
Feedback...17

Part I - Trace Code

A. threads/main.cc, int main(int argc, char **argv)
   - 程式進入點，讀取命令列傳入的引數
   - 建立Debug物件
   - 建立Kernel物件並初始化
   - 註冊聽到user abort事件時，需呼叫Cleanup清除kernel
   - 如果有任何測試旗標，執行對應的測試
   - 呼叫Kernel::ExecAll執行user programs

B. threads/kernel.cc, void Kernel::Kernel(int argc, char **argv)
   - 根據傳入的引數設定對應的屬性，其中-e引數將會被存在Kernel::execfile陣列中

C. threads/kernel.cc, void Kernel::Initialize()
   - 初始化主執行緒，將其狀態設為RUNNING
   - 建立Statistics、Interrupt、Scheduler、Alarm、Machine、SynchConsoleInput、SynchConsoleOutput、SynchDisk、FileSystem、PostOfficeInput、PostOfficeOutput物件
   - enable interrupt

D. threads/kernel.cc, void Kernel::ExecAll()
   - 將nachos執行指令中-e的引數逐一傳入Kernel::Exec
   - 完成後呼叫Thread::Finish結束main thread

E. threads/kernel.cc, void Kernel::Exec()
   - 創建Thread，傳入名稱與編號，執行緒初始狀態為JUST_CREATED
   - 創建該執行緒的AddrSpace
   - 呼叫該執行緒的Fork方法，傳入一個指到ForkExecute的function pointer與指向該執行緒的指標
   - 增加kernel的執行緒編號
   - 回傳稍早創建的執行緒編號

F. userprog/addrspace.cc, AddrSpace::AddrSpace()
   - 初始化pageTable陣列，指定physical pages與virtual pages的對應關係（原始實作為兩者相同）
   - 將整個16384 bytes（128 bytes/physical page * 128 physical pages）的main memory清零（zero out）

G. threads/thread.cc, void Thread::Fork(VoidFunctionPtr func, void *arg)
   - 將自己的兩個參數再傳入Thread::StackAllocate以配置此執行緒的stack
   - 關閉interrupt

- 呼叫Scheduler::ReadyToRun
- 恢復舊的interrupt狀態

H. threads/thread.cc, void Thread::StackAllocate(VoidFunctionPtr func, void *arg)

- 呼叫sysdep.cc的AllocBoundedArray直接在host的UNIX配置stack所需的空間
- 將stackTop指向stack最後一個整數的位置
- 將組合語言程序ThreadRoot push到stack，此程序定義在switch.S
- 在stack底部設置STACK_FENCEPOST，用於檢查stack是否發生overflow
- 將執行緒初始化方法、執行方法、執行參數、結束方法等指標設定到machineState，這些指標將在ThreadRoot程序中被使用

I. threads/kernel.cc, void ForkExecute(Thread *t)

- 調用AddrSpace::Load將user program載入記憶體
- 若未找到可執行的user program則中斷
- 呼叫AddrSpace::Execute執行user program

J. userprog/addrspace.cc, bool AddrSpace::Load(char *fileName)

- 使用FileSystem::Open開啟可執行檔
- 若未找到可執行檔則返回FALSE
- 讀取可執行檔的NoffHeader，NoffHeader包含code、initialized data與unitialized data segments資訊，每個segment都有virtualAddr、inFileAddr及大小訊息
- 比對NoffHeader的noffMagic來決定是否需要做little endian到big endian的轉換以修正NoffHeader內容
- 確保NoffHeader的noffMagic正確
- 計算要配置的size，其中包含code、initialized data、unitialized data、read only data及預留的UserStackSize
- 計算要配置的page數量，並確保該數量不大於所有的physical page數量
- 從檔案將code、initialized data與read only data讀入記憶體
- 關閉可執行檔
- 回傳TRUE表示成功讀取user program

K. userprog/addrspace.cc, void AddrSpace::Execute(char *fileName)

- 將此AddrSpace指派給kernel currentThread的space
- 調用AddrSpace::InitRegisters將NextPCReg和StackReg之外的所有暫存器清空，NextPCReg被指派為4，StackReg則是這個AddrSpace的大小再減16
- 呼叫AddrSpace::RestoreState復原Machine::pageTable
- 呼叫Machine::Run跳到user program

L. threads/scheduler.cc, void Scheduler::ReadyToRun (Thread *thread)

- 確保interrupt被關閉

    - 將執行緒狀態更新為READY
    - 將執行緒放入readyList

M. threads/thread.cc, void Thread::Finish()
    - 關閉interrupt
    - 檢查被終止的thread是kernel的currentThread
    - 呼叫Thread::Sleep並傳入TRUE

N. threads/thread.cc, void Thread::Sleep(bool finishing)
    - 確保要被sleep的thread為kernel的currentThread
    - 確保interrupt已被關閉
    - 將該thread的狀態改為BLOCKED
    - 如果scheduler的ready queue中並無任何其他thread，呼叫Interrupt::Idle直到有其他thread可執行
    - scheduler安排其他thread執行，呼叫Scheduler::Run傳入接下來要執行的執行緒與結束flag

O. machine/interrupt.cc, void Interrupt::Idle()
    - 將機器的Interrupt::status改為IdleMode
    - 呼叫Interrupt::CheckIfDue檢查是否有任何待執行的interrupt，若有則將機器的Interrupt::status改為SystemMode並返回
    - 若沒有任何待執行的interrupt，表示程式完成，呼叫Interrupt::Halt停止nachos

P. threads/scheduler.cc, void Scheduler::Run (Thread *nextThread, bool finishing)
    - 將kernel的currentThread指派給oldThread變數
    - 如果結束flag為TRUE，將toBeDestroyed設為oldThread
    - 如果oldThread的space不為空，表示其為user program，呼叫Thread::SaveUserState及AddrSpace::SaveState，保存該執行緒的CPU暫存器和address space
    - 檢查oldThread是否發生stack overflow
    - 將kernel的currentThread設為要執行的nextThread，並將其狀態改為RUNNING
    - 呼叫以組合語言定義的SWITCH，在host UNIX機器上切換oldThread和nextThread的context
    - 確保interrupt已被關閉
    - 呼叫Scheduler::CheckToBeDestroyed檢查是否有任何已完成的執行緒需要被清理
    - 如果oldThread的address space需被復原，則呼叫Thread::RestoreUserState及AddrSpace::RestoreState，復原oldThread的CPU暫存器和page table狀態

Q. threads/scheduler.cc, void Scheduler::CheckToBeDestroyed()
    - 檢查是否有執行緒需被摧毀

    ○ 若有則刪除該執行緒，觸發Thread::~Thread()

R. threads/thread.cc, Thread::~Thread()
    ○ 確保被刪除的執行緒不是kernel的currentThread
    ○ 若該執行緒有stack，則呼叫sysdep.cc的DeallocBoundedArray，在host機器上刪除stack

S. threads/switch.S, ThreadRoot (x86)
    ○ 呼叫StartupPC中的執行緒初始化程序Thread::Begin
    ○ 調用於InitialPC中nachos執行緒要執行的方法，並傳入參數InitialArg
    ○ 呼叫WhenDonePC中的執行緒結束程序Thread::Finish

T. threads/thread.cc, void Thread::Begin()
    ○ 確保此執行緒為kernel的currentThread
    ○ 呼叫Scheduler::CheckToBeDestroyed檢查是否有任何已完成的執行緒需要被清理
    ○ enable interrupt

Answers to code tracing questions

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

  Nachos在Kernel::Exec創建新執行緒的AddrSpace，調用該執行緒的Fork方法，傳入一個指到ForkExecute的function pointer與指向該執行緒的指標，隨後在ForkExecute中會將對應的可執行檔載入記憶體。

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

  Nachos在Kernel::Exec創建新執行緒的AddrSpace時，直接指定frame與page為一對一映射、設定每個page為valid、清空整個記憶體，這導致不同執行緒的frame重疊，無法執行多個程式。ForkExecute調用AddrSpace::Load將user program載入記憶體，此時因為page和frame是相同的，不需進行轉換，且Nachos並未針對唯讀segment設置唯讀屬性。我們的實作調整AddrSpace建構式及AddrSpace::Load，在AddrSpace建構時不指定frame與page的映射關係、設定每個page為invalid、不清空整個記憶體，只有在調用Kernel::GetFreeFrame取得可用的frame時，會清空該frame，並在取得frame後，才指定frame與page的映射關係，並設定page為valid。

- How does Nachos create and manage the page table?

  回答同[前一個問題]。

- How does Nachos translate addresses?

  Nachos每個執行緒會有對應的page table，在machine/translate.cc, Machine::Translate中，首先會檢查要讀取的數據大小與虛擬記憶體是否正確對齊，之後使用軟體模擬的TLB或page table其中之一進行記憶體位置轉換，如果有設定USE_TLB旗標，則使用軟體模擬的TLB進行轉換，否則使用page table，轉換時會檢查page number是否合法，此page是否valid，若不合法回傳AddressErrorException，若page invalid則回傳PageFaultException，找到對應的entry後，檢查是否受到唯讀保護，取得pageFramer後，檢查是否在實體記憶體範圍內，若通過檢查，則將該entry的use旗標設為TRUE，若是寫入操作，也將該entry的dirty旗標設為TRUE，回傳NoException表示轉換成功。

- How does Nachos initialize the machine status (registers, etc) before running a thread (process)?

  參見[AddrSpace::Execute]

- Which object in Nachos acts the role of process control block?

  Thread::userRegisters，在[Scheduler::Run]進行context switch前後，會調用Thread::SaveUserState和Thread::RestoreUserState將其保存或復原。

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

  [Thread::Fork]方法會呼叫[Scheduler::ReadyToRun]，將此執行緒放入readyList

- Please look at the following code from urserprog/exception.cc and answer the question:

```
case SC_MSG:
                    DEBUG(dbgSys, "Message received.\n");
                    val = kernel->machine->ReadRegister(4);
                    {
                        char *msg = &(kernel->machine->mainMemory[val]);
                        cout << msg << endl;
                    }
                    SysHalt();
                    ASSERTNOTREACHED();
                    break;
```

According to the code above, please explain under what circumstances an error will occur if the message size is larger than one page and why? (Hint: Consider the relationship between physical pages and virtual pages.)

此段程式碼使用實體記憶體位置(val)存取訊息字元陣列，雖然訊息字元陣列在虛擬記憶體上看起來是連續的，但若實際分配到的frame不連續，讀取超過一個page size，會存取到其他非法的實體記憶體位置。

Part II- Implement page table in NachOS

- code/machine/machine.h

  在ExceptionType的正確位置上加入MemoryLimitException以指示記憶體不足

- code/threads/kernel.h

  在Kernel加入私有屬性frameTable與公開方法GetFreeFrame及ReturnFreeFrame以管理實體記憶體

```
#include "bitmap.h"

class Kernel
{
   // ......
public:
  /**
   * @brief Try to get a free frame from the frame table
   *
   * @return int The number of the free fram
   */
  int GetFreeFrame();
  /**
   * @brief Return a free frame to the kernel
   *
   * @param frameNumber
   */
  void ReturnFreeFrame(int frameNumber);
private:
  Bitmap frameTable;
};
```

- code/threads/kernel.cc

  在Kernel建構式初始化frameTable，實作兩個新的公開方法，GetFreeFrame使用Bitmap::FindAndSet找到第一個可用的位置，將該位置bit設為1後回傳索引值，如果沒有可用的位置則回傳-1，此時直接呼ExceptionHandler(MemoryLimitException)，使程式中斷即可，若有可用的frame，則將該frame zero out後再回傳該frame的索引；

ReturnFreeFrame則相當簡單，直接使用Bitmap::Clear即可，Bitmap::Clear會檢查索引是否在合法範圍，並將對應的bit清空。

```
Kernel::Kernel(int argc, char **argv) : frameTable(Bitmap(NumPhysPages))
{
    // ......
}

int Kernel::GetFreeFrame()
{
    int frameNumber = frameTable.FindAndSet();
    if (frameNumber == -1)
    { // Because scheduler does not yet support swapping, abort the program
directly here
        ExceptionHandler(MemoryLimitException);
    }
    // zero out the frame
    int frameBase = frameNumber * PageSize;
    bzero(machine->mainMemory + frameBase, PageSize);
    return frameNumber;
};

void Kernel::ReturnFreeFrame(int frameNumber)
{
    // Bitmap will check if the number is within the range
    frameTable.Clear(frameNumber);
};
```

- code/userprog/addrspace.h

定義PageFlags與PageFlag，以簡化方法參數，AddrSpace新增私有方法AllocateAndLoad配置實體記憶體並管理page table。

```
#include "noff.h"

typedef int PageFlags;
/**
 * @brief Flags used to set TranslationEntry
 *
```

```
 */
enum PageFlag
{
  VALID = 0b0001,
  READ_ONLY = 0b0010,
  USE = 0b0100,
  DIRTY = 0b1000,
};

class AddrSpace
{
  // ......
private:
  /**
   * @brief Allocate free frames and load the segment into the page table
   *
   * @param segmentName For debug
   * @param segment
   * @param executable The user program to be loaded. NULL if we don't need
to load it from file.
   * @param flags Used to set TranslationEntry
   * @param remaining Remaining space of the last page
   */
  void AllocateAndLoad(char const *segmentName, Segment &segment, OpenFile
*executable, PageFlags flags, int &remaining);
};
```

- code/userprog/addrspace.cc

    修改AddrSpace的建構式與解構式，建構時不再配置實體記憶體，且每個page皆為invalid；解構時，將每個valid page對應的frame歸還kernel。

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; ++i)
    {
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = -1;
```

```
        pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
}


AddrSpace::~AddrSpace()
{
    // release all valid pages
    for (int i = 0; i < NumPhysPages; ++i)
    {
        if (pageTable[i].valid)
        {
            kernel->ReturnFreeFrame(pageTable[i].physicalPage);
        }
    }
    delete pageTable;
}
```

調整AddrSpace::Load，因為記憶體的配置與記憶體是否足夠的檢查分別交由AddrSpace::AllocateAndLoad和Kernel::GetFreeFrame，故在Load當中不再計算size及numPages，又由於虛擬記憶體的連續性，我們使用一個新的變數remaining追蹤最後一個page還有多少剩餘空間，呼叫AllocateAndLoad，傳入對應的segment、page旗標及remaining變數以配置記憶體，值得注意的是uninitData（.BSS）和user stack都是可執行檔中沒有的數據，故executable參數傳入NULL，表示只需要配置記憶體，不需要從檔案讀取，最後同樣關閉executable，回傳TRUE。

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

    int remaining = 0;
    AllocateAndLoad("code", noffH.code, executable, VALID | READ_ONLY,
remaining);
#ifdef RDATA
    AllocateAndLoad("read only data", noffH.readonlyData, executable, VALID
| READ_ONLY, remaining);
#endif
    AllocateAndLoad("data", noffH.initData, executable, VALID, remaining);
    // user stack and uninit data do not exist in the executable
    AllocateAndLoad("uninit data", noffH.uninitData, NULL, VALID,
remaining);
    Segment dummy{-1, -1, UserStackSize}; // just for function
compatibility.
    AllocateAndLoad("user stack", dummy, NULL, VALID, remaining);
    DEBUG(dbgAddr, "AddrSpace " << fileName << " with size " << numPages *
PageSize << " uses " << numPages << " pages");

    delete executable; // close file
    return TRUE;       // success
}
```

在AddrSpace::AllocateAndLoad當中，若該segment為空則直接回傳，使用inFilePosition變數追蹤當前應該從檔案的哪個位置開始讀取，如果最後一個配置的page有剩餘空間，則將該segment的開頭寫入此處並設置旗標，注意segment有可能小於剩餘空間，若寫滿最後一個page的剩餘空間後，還有未配置的部分，則每次和kernel要一個frame，設置page與frame的對應關係與旗標，直到該segment都配置完成，更新remaining變數讓下一個segment知道最後一個配置的page還有多少剩餘空間。

```
void AddrSpace::AllocateAndLoad(char const *segmentName, Segment &segment,
OpenFile *executable, PageFlags flags, int &remaining)
{
    if (segment.size <= 0)
    {
```

```
        return;
    }
    DEBUG(dbgAddr, "Initializing " << segmentName << " segment with size "
<< segment.size);
    int inFilePosition = segment.inFileAddr;
    // use the remaining space of the last page first
    if (remaining > 0)
    {
        if (executable != NULL)
        {
            int frameBase = pageTable[numPages - 1].physicalPage * PageSize;
            // don't read too much into the last page.
            int numBytes = min(remaining, segment.size);
            executable->ReadAt(&(kernel->machine->mainMemory[frameBase]),
numBytes, inFilePosition);
            inFilePosition += numBytes;
        }
        // set up page table entry
        pageTable[numPages - 1].valid = flags & VALID;
        pageTable[numPages - 1].use = flags & USE;
        pageTable[numPages - 1].dirty = flags & DIRTY;
        pageTable[numPages - 1].readOnly = flags & READ_ONLY;
    }
    int unallocatedSize = segment.size - remaining;
    while (unallocatedSize > 0)
    {
        int frameNumber = kernel->GetFreeFrame();
        if (executable != NULL)
        {
            int frameBase = frameNumber * PageSize;
            // don't read too much into the last page.
            int numBytes = min(unallocatedSize, PageSize);
            executable->ReadAt(&(kernel->machine->mainMemory[frameBase]),
numBytes, inFilePosition);
            inFilePosition += numBytes;
        }
        // set up page table entry
        pageTable[numPages].virtualPage = numPages;
        pageTable[numPages].physicalPage = frameNumber;
        pageTable[numPages].valid = flags & VALID;
        pageTable[numPages].use = flags & USE;
```

```
        pageTable[numPages].dirty = flags & DIRTY;
        pageTable[numPages].readOnly = flags & READ_ONLY;
        ++numPages;
        unallocatedSize -= PageSize;
    }
    remaining = -unallocatedSize;
    DEBUG(dbgAddr, segmentName << " segment virtualAddr " <<
segment.virtualAddr << ", segment size " << segment.size);
}
```

Difficulties encountered

- 一開始實作並沒有了解到虛擬記憶體的連續性，每個segment都直接round up給整數倍的page，直到使用sort.c這支程式進行測試，才發現segment在虛擬記憶體應該要是連續的，花了一些時間調整實作，目前的實作下，如果多個segment共用page，則該page的旗標會被後面的segment覆蓋，似乎跟需求有些衝突，但考慮到實務上記憶體保護應該使用page table與segment table，單純只用page table在segment共用page的案例就有可能出現read only segment被寫入的問題。

Feedback

- 建議明確說明多個segment共用page的狀況下，page旗標應該如何設置。
- 建議明確說明是否可直接調用ExceptionHandler結束MemoryLimitException的處理。