OS Homework2 Report

Team21

葉宥忻 109062301

陳禹勳 109062134

contribution: 一起

Trace code

int main() - threads/main.cc

```
    kernel = new Kernel(argc, argv);

    kernel->Initialize();
```

```
            kernel->ExecAll();
```

在main.cc的int main中，會interpret一些command line上輸入的argument(argv)，然後create一個kernel，呼叫kernel->Initialize()去初始化kernel，再透過kernel->ExecAll()來run user program

Kernel::Kernel() - threads/kernel.cc

```
{
    randomSlice = FALSE;
    debugUserProg = FALSE;
    consoleIn = NULL;          // default is stdin
    consoleOut = NULL;         // default is stdout
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
    reliability = 1;            // network reliability, default is 1.0
    hostName = 0;               // machine id, also UNIX socket name
                                // 0 is the default machine id
```

```
        } else if (strcmp(argv[i], "-e") == 0) {
            execfile[++execfileNum]= argv[++i];
            cout << execfile[execfileNum] << "\n";
```

Construct一個kernel，初始化一些variable，並且interpret一些command line上輸入的argument(argv)來定義flag，例如-e會把我們要執行的檔案紀錄到execfile中

Kernel::Initialize() - threads/kernel.cc

```
    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);

    stats = new Statistics();        // collect statistics
    interrupt = new Interrupt;       // start up interrupt handling
    scheduler = new Scheduler();     // initialize the ready queue
    alarm = new Alarm(randomSlice); // start up time slicing
    machine = new Machine(debugUserProg);
    synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
    synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
    synchDisk = new SynchDisk();     //
#ifdef FILESYS_STUB
    fileSystem = new FileSystem();
#else
    fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
    postOfficeIn = new PostOfficeInput(10);
    postOfficeOut = new PostOfficeOutput(reliability);

    interrupt->Enable();
}
```

Create main這個thread作為currentThread，狀態設為running，透過Kernel::Kernel設定的參數來初始化某些data structure，例如Scheduler、Interrupt、FileSystem等，並且在最後enable interrupt

Kernel::ExecAll() - threads/kernel.cc

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

呼叫Kernel::Exec()去個別生成thread來執行kernel中所有的execfile，**執行完後呼叫 currentThread的Finish()，也就是結束掉**main thread

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

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

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

當一個thread做完就會呼叫，先Disable interrupt，實作方式是呼叫Sleep裡面傳TRUE，讓currentThread Sleep，因為現在還沒完成switch，所以還不能刪除這個thread，Sleep的參數TRUE代表這個Thread已經結束了，之後才會在Scheduler::CheckToBeDestroyed將這個Thread刪除

Thread::Sleep() - threads/thread.cc

```cpp
    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);
}
```

把thread status設為BLOCKED，如果ready queue是空的，呼叫Interrupt::Idle去檢查有沒有interrupt要處理，如果ready queue有thread在等待，**把CPU從A thread交給ready queue的第一個 thread，A thread可能是因為已經做完或在等synchronization variable，若是前者，A thread之後會被刪掉；若是後者之後會把A thread重新放回ready queue，未來會叫醒A thread**

Interrupt::Idle() – machine/interrupt.cc

```
{
    DEBUG(dbgInt, "Machine idling; checking for interrupts.");
    status = IdleMode;
    DEBUG(dbgTraCode, "In Interrupt::Idle, into CheckIfDue, " << kernel->stats->totalTicks);
    if (CheckIfDue(TRUE)) { // check for any pending interrupts
    DEBUG(dbgTraCode, "In Interrupt::Idle, return true from CheckIfDue, " << kernel->stats->totalTicks);
    status = SystemMode;
    return;         // return in case there's now
                // a runnable thread
    }
```

因為ready queue中是空的，所以先將interrupt status設為IdleMode，確認有沒有interrupt要處理，如果有interrupt，處理完後把status設為SystemMode

```
    // if there are no pending interrupts, and nothing is on the ready
    // queue, it is time to stop.   If the console or the network is
    // operating, there are *always* pending interrupts, so this code
    // is not reached.  Instead, the halt must be invoked by the user program.

    DEBUG(dbgInt, "Machine idle.  No interrupts to do.");
    cout << "No threads ready or runnable, and no pending interrupts.\n";
    cout << "Assuming the program completed.\n";
    Halt();
```

如果沒有interrupt，代表已經沒有事情可以做了，所以呼叫Halt()來停止

Scheduler::Run() - threads/scheduler.cc

```
    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

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

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將currentThread設為nextThread並把nextThread的status設為RUNNING

```
    SWITCH(oldThread, nextThread);
```

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
    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();
    }
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，**有的話在把之前**oldThread**存下來的**state**還原回去**

Kernel::Exec() - threads/kernel.cc

```
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
```

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

新增一個thread**給要執行的**user program，**再**make**一個新的**pageTable**給這個**thread，**接著呼叫**Thread::Fork**來做**allocate stack，**把**thread**放進**ready queue**等操作，最後回傳這個**thread**的**thread number

AddrSpace::AddrSpace() - userprog/addrspace.cc

```
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);
}
```

How does Nachos create and manage the page table?

Create page table來translate virtual memory到physical memory，然後初始化pageTable中的valid bit、dirty bit等，在範例中假設只有一個user program，所以讓virtual page number = physical page number，也就是把所有的physical memory都給一個user program

Kernel::ForkExecute() - threads/kernel.cc

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

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

}
```

把user program load到memory，成功就可以呼叫AddrSpace::Execute來執行

AddrSpace::Load() - userprog/addrspace.cc

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

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

```
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
```

先把要執行的file打開放進executable，把object code放到noffHeader這個data structure中

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

判斷code size及data size是否大於零，是的話分別將code和initData透過inFileAddr和size從file存到mainMemory中，在範例中因為假設virtual address = physical address，所以直接用virtual address去指定main memory的位置

AddrSpace::Execute() - userprog/addrspace.cc

```
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"
}
```

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

將currentThread space的指標指向這個address space，先初始化registers，再load page table，再開始run program的instruction

Thread::Fork() - threads/thread.cc

```cpp
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);
}
```

先做StackAllocate()，allocate空間給要執行的thread，接下來disable interrupt，把thread放進ready queue，回復interrupt原本的狀態

Thread::StackAllocate() - threads/thread.cc

```cpp
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

```cpp
    int pgSize = getpagesize();
    char *ptr = new char[pgSize * 2 + size];

    mprotect(ptr, pgSize, 0);
    mprotect(ptr + pgSize + size, pgSize, 0);
    return ptr + pgSize;
```

create一塊pagesize*2+size的空間，保護前後兩個page，來偵測有沒有overflow，回傳第二個page，也就是第一個可以用的

```
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
```

將ThreadRoot、ThreadBegin、func procedure、arg等pointer，存進machineState這個register，之後要呼叫這些procedure或使用argument就從這些register找

Scheduler::ReadyToRun() - threads/scheduler.cc

```
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);
}
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

How does Nachos translate addresses?

使用AddrSpace::Translate()可以幫助我們建立page table時轉換address

translate.cc中Machine::Translate會在run program時轉換virtual address成physical address，才能讀到正確的memory content

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

```
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
```

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

Thread.h的class Thread能夠紀錄userRegisters、status等狀態，有saveUserState和RestoreUserState function讓我們在context switch可以儲存和回復register,

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

```
    int userRegisters[NumTotalRegs];
```

```
    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 *stackTop;        // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop
```

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

Thread::Fork(): 要產生新的thread要去run user program，所以create好後要將thread放進ready queue，因此會呼叫ReadyToRun()

Thread::Yield(): 當currentThread要把CPU的使用權交給其他thread，並且currentThread還沒finish，之後還要執行，所以把currentThread再放回ready queue，因此會呼叫ReadyToRun()

```
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);
}
```

```
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

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

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
    kernel->scheduler->ReadyToRun(this);
    kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

Implement page table in NachOS

ExceptionHandler:

在machine.h新增一種exception: MemoryLimitException

```
enum ExceptionType { NoException,           // Everything ok!
                     SyscallException,      // A program executed a system call.
                     PageFaultException,    // No valid translation found
                     ReadOnlyException,     // Write attempted to page marked
                                            // "read-only"
                     BusErrorException,     // Translation resulted in an
                                            // invalid physical address
                     AddressErrorException, // Unaligned reference or one that
                                            // was beyond the end of the
                                            // address space
                     OverflowException,     // Integer overflow in add or sub.
                     IllegalInstrException, // Unimplemented or reserved instr.

                     MemoryLimitException,

                     NumExceptionTypes
};
```

若要用到的Page超過剩餘可以用的PhysicalPage，就呼叫ExceptionHandler()，參數傳MemoryLimitException

```
    if(numPages > kernel->numFreePhysicalPage){
        ExceptionHandler(MemoryLimitException);
    }
```

進到Exceptionhandler後，找到對應的case，然印出是哪種exception

```
        case MemoryLimitException:
            cerr << "Unexpected user mode exception " << (int)which << "\n";
            ASSERTNOTREACHED();
            break;
```

AddrSpace::AddrSpace() - addrspace.cc

```cpp
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);
}
```

原本AddrSpace的constructor假設virtual page等於physical page，所以我們將他註解

AddrSpace::~AddrSpace() - addrspace.cc

```cpp
AddrSpace::~AddrSpace()
{
    for (int i = 0; i < numPages; i++){
        kernel->usedPhysicalPage[pageTable[i].physicalPage] = FALSE;
        kernel->numFreePhysicalPage ++;
    }
    delete pageTable;
}
```

用for迴圈把所有usedPhysicalPage從true設成false，意思是把page table裡存的PhysicalPage從標記為「用過」改成標記為「沒用過」，再把可用numFreePhysicalPage+1

AddrSpace::Load() - addrspace.cc

```cpp
// 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;
```

計算program所需的size和page數

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

    if(numPages > kernel->numFreePhysicalPage){
        ExceptionHandler(MemoryLimitException);
    }
```

如果需要的page數超過可用的page數，則呼叫exception

```
    pageTable = new TranslationEntry[numPages];
    int freePhysicalIndex = 0;
    for (int i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;
        while (freePhysicalIndex < NumPhysPages && kernel->usedPhysicalPage[freePhysicalIndex] == TRUE){
            freePhysicalIndex = freePhysicalIndex + 1;
        }
        pageTable[i].physicalPage = freePhysicalIndex;
        kernel->usedPhysicalPage[freePhysicalIndex] = TRUE;
        kernel->numFreePhysicalPage --;
        bzero(&kernel->machine->mainMemory[freePhysicalIndex*PageSize], PageSize);

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

        DEBUG(dbgAddr, "VirtualPageNumber = " << pageTable[i].virtualPage);
        DEBUG(dbgAddr, "PhysicalPageNumber = " << pageTable[i].physicalPage);
    }
```

先初始化一個pageTable，接下來利用kernel的usedPhysicalPage從physical page中尋找可用的page，再讓page table紀錄所有virtual page對應的physical page，分配好後將那個physical page清零，並且設定pageTable中valid、use、dirty等field

```
    int current_code_size = noffH.code.size;
    int current_code_inFileAddr = noffH.code.inFileAddr;
    int current_code_virtualAddr = noffH.code.virtualAddr;

    while (current_code_size > 0) {
        int physical_addr = pageTable[current_code_virtualAddr/PageSize].physicalPage * PageSize + (current_code_virtualAddr%PageSize);
        int used_size = PageSize;
        if (current_code_size < PageSize){
            used_size = current_code_size;
        }
        executable->ReadAt(
            &(kernel->machine->mainMemory[physical_addr]),
            used_size,
            current_code_inFileAddr
        );
        DEBUG(dbgAddr, "========================================");
        DEBUG(dbgAddr, physical_addr << "    " << used_size);
        DEBUG(dbgAddr, "========================================");
        current_code_size = current_code_size - used_size;
        current_code_inFileAddr = current_code_inFileAddr + used_size;
        current_code_virtualAddr = current_code_virtualAddr + used_size;
    }
```

這邊我們要將code和initData等資料從executable放進mainMemory，我們用virtual address和pagesize來計算實際的physical address，將各個virtual memory的資料放到對應的physical memory

class Kernel - kernel.h

```
    bool usedPhysicalPage[NumPhysPages];
    int numFreePhysicalPage;
```

宣告usedPhysicalPage來記錄那些physical page被使用了

宣告numFreePhysicalPage代表剩幾個physical page沒被使用

Kernel::Initialize() - kernel.cc

```
    usedPhysicalPage = {FALSE};
    numFreePhysicalPage = NumPhysPages;
```

初始化usedPhysicalPage和numFreePhysicalPage，代表還沒有physical page被使用

Difficulties

1、在trace Scheduler::Run的142行的時候，一開始覺得很奇怪，因為112行有可能會將toBeDestroyed設為oldThread，那在142行不就有可能把自己給delete掉，但是thread又沒辦法在running的時候被刪除，所以想了很久，後來才想到toBeDestroyed會在context switch到別的thread的時候，被設定為其他值，因此在142行時才不會將自己刪除

```
110     if (finishing) {    // mark that we need to delete current thread
111         ASSERT(toBeDestroyed == NULL);
112      toBeDestroyed = oldThread;
113     }
```

```
142         CheckToBeDestroyed();
```

2、一開始我們不知道為甚麼machineState[]要存的是pointer，而不是在要用function的時候直接呼叫function，trace code後發現這樣實做的話，可以減少執行thread時所需load進memory的code，避免浪費空間。

```
356  #else
357      machineState[PCState] = (void*)ThreadRoot;
358      machineState[StartupPCState] = (void*)ThreadBegin;
359      machineState[InitialPCState] = (void*)func;
360      machineState[InitialArgState] = (void*)arg;
361      machineState[WhenDonePCState] = (void*)ThreadFinish;
```

3、在實作page table時嘗試了很多次，因為有很多的細節要考慮，例如virtual和physical address的轉換，還有在將資料放進mainMemory時，我們原本的寫法沒有考慮到code size可能大於page size，所以直接將整個code放入連續的physical page中，如果遇到physical page不是連續的，就會發生錯誤，後來才改為一次對應一個page