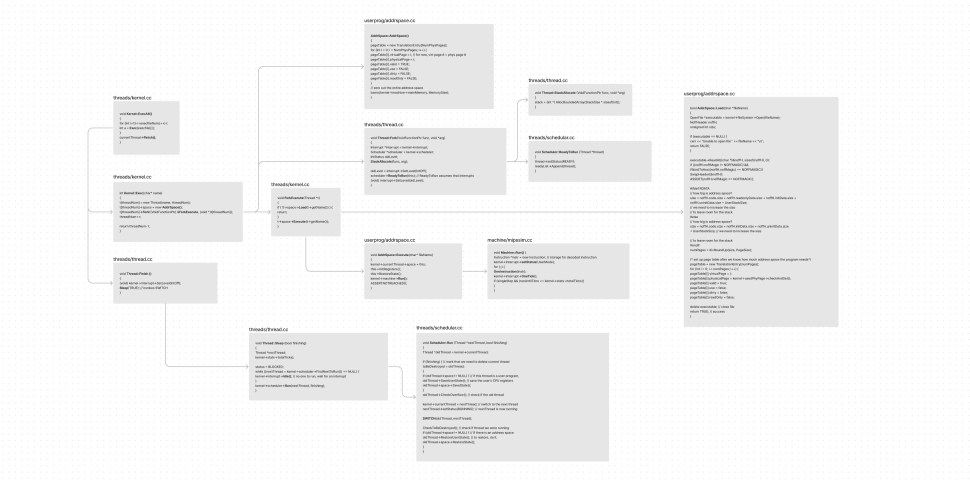
NachOS MP2 109000109 謝秉軒

Trace code



https://www.figma.com/file/oGOfVIMEQIroe7nNslhZZP/Untitled?node-id=0%3A1&t=NINscjHOAgoJtjgl-1

(https://www.figma.com/file/oGOfVIMEQIroe7nNslhZZP/Untitled?node-id=0%3A1&t=NINscjHOAgoJtjgl-1)

threads/thread.h

我們先來看看 class Thread。

```
class Thread {
  private:
    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

    void Fork(VoidFunctionPtr func, void *arg);
    void Yield();             // Relinquish the CPU if any
    void Sleep(bool finishing); // Put the thread to sleep and
    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:
    int *stack;               // Bottom of the stack
    ThreadStatus status;      // ready, running or blocked
    char* name;
        int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
    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.
};
```

我們針對 Sleep()、StackAllocate()、Finish()、Fork()來分析。

1. VOID THREAD::SLEEP (BOOL FINISHING)

threads/thread.cc

只有 currentThread 可以呼叫 Sleep()，currentThread 是目前正在執行的 thread，當currentThread 完成他的工作或是在等待 synchronization variable (Semaphore, Lock, or Condition)，會進入 Sleep()。 當 currentThread 進入 Sleep()，就能從 readyList 中 dequeue 一個 thread 成為 currentThread，使用 CPU。我們將進入 Sleep() 的 thread 的 state 存起來，並且進行 context switch 。當再次輪到在 Sleep() 的 thread 使用 CPU 時，Run() 才會 return。

```
void Thread::Sleep (bool finishing)
{
    Thread *nextThread;
    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    status = BLOCKED;

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        /* no one to run, wait for an interrupt */
        kernel->interrupt->Idle();
    }

    /* returns when it's time for us to run */
    kernel->scheduler->Run(nextThread, finishing);
}
```

2. VOID THREAD::STACKALLOCATE

threads/thread.cc

Allocate array 給 Thread

Q1 : How does Nachos allocate the memory space for a new thread(process)?

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

3. VOID THREAD::FINISH

threads/thread.cc

SetLevel(IntOff) 後 Sleep(TRUE)。

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

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

4. VOID THREAD::FORK

threads/thread.cc

1.Allocate array
2.讓 Scheduler 安排執行
Q7 : When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?

```
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 << " '
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);     // ReadyToRun assumes that interrupts
                                     // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

5. AddrSpace::AddrSpace

userprog/addrspace.cc

在 addrSpace 物件建立時就會建立 pageTable，預設 NachOS 只會運行一個可執行檔，因此這個 pageTable 涵蓋整個 physical memory，因此不需要做任何 virtualPage 和 physicalPage 之間的映射。
Q3 : How does Nachos create and manage the page table?

```
/* userprog/addrspace.cc */

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

6. AddrSpace::Execute

userprog/addrspace.cc

1.初始化 register
2.載入 page table register
3.切換至 user progam
Q5 : How Nachos initializes the machine status (registers, etc) before running a thread(process)?

```
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
    ...
}
```

7. AddrSpace::Load

userprog/addrspace.cc

打開一個 file，如果是 executable 就讀出 header 來獲取檔案的 metadata。

Q2 : How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?

```
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);
    ...
    ...
}
```

8. Kernel::Kernel

threads/kernel.cc

當我們在命令列下指令 -e halt，halt 即是我們想執行的可執行檔。

```
Kernel::Kernel(int argc, char **argv)
{
    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
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
               ASSERT(i + 1 < argc);
               RandomInit(atoi(argv[i + 1]));// initialize pseudo-random
                       // number generator
               randomSlice = TRUE;
               i++;
        } else if (strcmp(argv[i], "-s") == 0) {
            debugUserProg = TRUE;
               } else if (strcmp(argv[i], "-e") == 0) {
               execfile[++execfileNum]= argv[++i];
                       cout << execfile[execfileNum] << "\n";
               } else if (strcmp(argv[i], "-ci") == 0) {
               ASSERT(i + 1 < argc);
               consoleIn = argv[i + 1];
               i++;
               } else if (strcmp(argv[i], "-co") == 0) {
               ASSERT(i + 1 < argc);
               consoleOut = argv[i + 1];
               i++;
#ifndef FILESYS_STUB
               } else if (strcmp(argv[i], "-f") == 0) {
               formatFlag = TRUE;
#endif
        } else if (strcmp(argv[i], "-n") == 0) {
            ASSERT(i + 1 < argc);   // next argument is float
            reliability = atof(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-m") == 0) {
            ASSERT(i + 1 < argc);   // next argument is int
            hostName = atoi(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            cout << "Partial usage: nachos [-rs randomSeed]\n";
                      cout << "Partial usage: nachos [-s]\n";
            cout << "Partial usage: nachos [-ci consoleIn] [-co consoleOut]\n";
#ifndef FILESYS_STUB
               cout << "Partial usage: nachos [-nf]\n";
#endif
            cout << "Partial usage: nachos [-n #] [-m #]\n";
               }
    }
}
```

9. KERNEL::EXECALL

threads/kernel.cc

把所有可執行檔依序丟進 Exec() 執行。

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

10. KERNEL::EXEC

threads/kernel.cc

每一個可執行檔都需要 new 一個新的 thread 來執行。
賦予他一個new AddrSpace()。
透過 Fork & ForkExecute 載入真正要執行的程式碼。
Thread數量+1

```
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;
}
```

11. KERNEL::FORKEXECUTE

threads/kernel.cc

建立 thread 執行所需要的 stack，並把 thread 丟進 readyList 等待執行。

```
void Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

12. SCHEDULER::READYTORUN

threads/scheduler.cc

將 thread 加入 readyList，當 scheduler 需要安排 thread 來執行的時候，就從 readyList dequeue 一個 thread 來執行。

```
void Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
    readyList->Append(thread);
}



class Scheduler {
    Thread* FindNextToRun();
    List<Thread *> *readyList;
};
```

13. VOID SCHEDULER::RUN (THREAD *NEXTTHREAD, BOOL FINISHING)

threads/scheduler.cc

呼叫 SWITCH() 進行 context switch，在 SWITCH() 完成之後，並不會繼續執行 Run() 的下一行，因為控制權已經傳給 nextThread，下一行要被執行的指令是 nextThread 的指令，

當 CheckToBeDestroyed() 被執行，代表已經又輪到這個 thread 使用 CPU。

Q6 : Which object in Nachos acts the role of process control block?

```
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    if (finishing) {
        /* mark that we need to delete current thread */
        ASSERT(toBeDestroyed == NULL);
          toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) {
        /* save the user's state */
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

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

    /* This is a machine-dependent assembly language routine defined
     * in switch.s */
    SWITCH(oldThread, nextThread);

    /* we're back, running oldThread */
    /* check if thread we were running before this one has finished
     * and needs to be cleaned up */
    CheckToBeDestroyed();

    if (oldThread->space != NULL) {
        /* if there is an address space to restore, do it */
        oldThread->RestoreUserState();
          oldThread->space->RestoreState();
    }
}
```

14. MACHINE::TRANSLATE

machine/translate.cc

How does Nachos translate addresses?

```
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    ...
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;
    ...
    pageFrame = entry->physicalPage;
    ...
    *physAddr = pageFrame * PageSize + offset;
    ...
    return NoException;
}
```

Implementation

(a) 更改建立 pageTable building 的方式

在修改之前，在 addrSpace 物件建立時就會建立 pageTable，預設 NachOS 只會運行一個可執行檔，因此這個 pageTable 涵蓋整個 physical memory，因此不需要做任何 virtualPage 和 physicalPage 之間的換算。

```
/* userprog/addrspace.cc */

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

但是當我們想要同時執行一個以上的 threads 時，會發生問題，因為單一個 thread 不能佔有整個 physical memory，每個 thread 都只能擁有部份的 physical memory，因此我修改 pageTable 的建立，每個 thread 的 pageTable 的尺寸只包含他所需要的部份。但是在我們將可執行檔 load 進 memory 之前，我們並不知道這個可執行檔需要多少空間， 因此我將 pageTable 的建立延後，直到可執行檔的 header 讀入，我們知道這個可執行檔的大小之後，才建立 pageTable。

```
/* userprog/addrspace.cc */

bool AddrSpace::Load(char *fileName)
{
    ...
    numPages = divRoundUp(size, PageSize);

    /* setup pageTable after we know how much space the program needs */
    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);
    }
}
```

新增 class UsedPhyPage 來管理 physical memory，使用一個 array 紀錄 physical page 是否已經被使用。

此時 physicalPage 需要經過 translate 才能得到真正的值。這邊透過 class UsedPhyPage 裡的 checkAndSet() 實踐。

```
/* threads/kernel.cc */

class UsedPhyPage {
public:
    UsedPhyPage();
    ~UsedPhyPage();
    int *pages; /* 0 for unused, 1 for used */
    int numUnused();
    int checkAndSet();
};
```

checkAndSet() 回傳一個未使用的 physical page 的 pageNum，如果整個 physical memory 都已經被使用，回傳 -1。使用這個方法建立 pageTable，每個 thread 管理自己的 pageTable，使用屬於自己的 memory space 。

```
pageTable[i].physicalPage = kernel->usedPhyPage->checkAndSet();
```

(b) 更改 executable file loading

我們的目標是把每一段 segment 從他在檔案裡的位置，load 到他的 virtualAddr 所對應的 physicalAddr。未修改前因為整個 memory 只有一個 thread 在使用，所以 physicalAddr 和 virtualAddr 相同，並且因為這個可執行檔是第一個也是唯一一個 load 進 memory 的檔案，因此整個空間是連續且無人使用的，把可執行檔搬移到 memory，每一個 segment 只要搬移一次即可。

```
/* userprog/addrspace.cc */

bool AddrSpace::Load(char *fileName)
{
    /* read header file */
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    ...
#ifdef RDATA
    size = noffH.code.size + noffH.readonlyData.size +
           noffH.initData.size + noffH.uninitData.size + UserStackSize;
    ...
#endif
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    if (noffH.code.size > 0) {
        executable->ReadAt(
                &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
                noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        ...
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        ...
    }
#endif
}
```

在修改之後，memory 由多個 thread 共用，因此在將資料從檔案 load 進 memory 之前，需要先使用 Translate() 將 virtualAddr 轉換成對應的 physicalAddr。因為有多個 thread 共用 memory，memory 中可能已經有些 page 被別的 thread 佔用了，因此資料需要一個 page 一個 page 的 load 進 memory ，而不能整段 segment (包含數個 page) load 進 memory 。 以下圖為例，這個檔案有 4 個 page，每一個 page 有 5 個 bytes，virtal page 和 physical page 的對應如下，可以看到 virtual page 是連續的 (0 , 1 , 2 , 3)，但 physical page 不是連續的 (6 , 9 , 7 , 11)，因此我們在搬移資料時，一次只能搬一個 virtual page 的資料。

假設我們所要搬移的資料從 virtual addr 8 - 17，可以看到他包含了 3 個 page，因此我們需要分成 3 次轉移資料，資料從 inFile addr 搬移到我們指定的 virtual addr 對應到的 physical addr，virtual addr 可以自訂，在 NachOS 裡 virtual addr 和 inFile addr 是相同的。

```
/* userprog/addrspace.cc */

bool AddrSpace::Load(char *fileName)
{
    ...
    unsigned int virtualAddr;
    unsigned int physicalAddr;
    int unReadSize;
    int chunkStart;
    int chunkSize;
    int inFilePosiotion;

    if (noffH.code.size > 0) {
        unReadSize = noffH.code.size;
        chunkStart = noffH.code.virtualAddr;
        chunkSize = 0;
        inFilePosiotion = 0;

        /* while still unread code */
        while(unReadSize > 0) {
            /* first chunk and last chunk might not be full */
            chunkSize =  calChunkSize(chunkStart, unReadSize);
            /* mapping from virtual addr to physical addr */
            Translate(chunkStart, &physicalAddr, 1);

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

            unReadSize = unReadSize - chunkSize;
            chunkStart = chunkStart + chunkSize;
            inFilePosiotion = inFilePosiotion + chunkSize;
        }
    }
    if (noffH.initData.size > 0) {
        ...
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        ...
    }
#endif
    ...
}
```

calChunkSize 回傳區塊的大小給 ReadAt()

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

心得

實作 Multi-Programming 的部分比我想想中難很多，中間還跑回去複習 virtualPage 和 physicalPage 之間的關係。中間也花了很多時間解決 Segmentation fault 的問題，其中很多都是忘記 allocate 導致的。
這次的作業也讓我學到很多作業系統在處理 threads 的細節以及流程。