2017OSteam19 MP2 Trace

# NachOS如何創造一個Thread

- 我們把NachOS創造Thread的步驟可以分為三個流程說明
  1. **Kernel Part**
  2. **Thread Part**
  3. **Addrspace Part**

---

## Kernel Part

- Kernel::Kernel()
  - main.cc第250行new一個kernel時執行
  - kernel的建構子
  - 當收到-e參數時把要執行的檔案加入execfile裡面
  - 執行完畢後回到main.cc繼續執行

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

---

- Kernel::ExecAll()
  - main.cc第288行call入
  - 用一個for迴圈把exefile裡面每一個元素都call一次Exec()去執行

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

  - 完成後用currentThread->Finish()把目前占用CPU的Thread free掉並call Sleep來觸發Switch做context switch

---

- Kernel::Exec(char* name)

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


return threadNum-1;
```

- 這個function先new一個Thread object(定義在Thread.cc)
- 接著透過AddrSpace.cc幫這個thread allocate一些memory以及創建pageTable
- 接著call thread.cc 的fork，把kernel::ForkExecute傳進Thread::Fork內
- Thread::Fork幫傳進來的function allocate一塊execution stack，並把Thread放進Ready-queue裡面等待執行
- Kernel::ForkExecute內又call了Addrspace::Execute
- Addrspace::Execute先把page table load進kernel，再call kernel->machine->Run()
- kernel->machine->Run()把kernel->currentThread設成傳進來的Thread

---

## Thread Part

- 在NachOS中Process等於Thread
- Thread.cc定義了在NachOS中一個Thread是如何被創造與控制的
- 在這邊我們只說明從創建Thread到把它放進Ready-queue會使用到的function，其他像是Yield或是Sleep這些用來change Thread state以handle context switch或是interrupt的function就沒有說明
- Thread::Thread(char* threadName, int threadID)

```
Thread::Thread(char* threadName, int threadID)
{
    ID = threadID;
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
    machineState[i] = NULL;
    }
    space = NULL;
}
```

  - Thread的建構子只先初始化Thread Control Block的一些設定值，像是ID以及thread就回到kernel::Exec了

---

- Thread::Fork(VoidFunctionPtr func, void *arg)

```
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);
    (void) interrupt->SetLevel(oldLevel);
}
```

- Thread::Fork讓被傳進來的function(kernel::ForkExecute)可以和call Thread::Fork的function可以work concurrently
- Thread在這裡被放進Ready-queue內排程準備執行
- Interrupt會先被關掉，等Thread被放進scheduler的Ready-queue後才設回原本的狀態
- 被傳進來的ForkExecute會去call Addrspace::Load，準備創建pageTable並把thread的page搬到MainMemory裡面來

---

## Addrspace Part

- Addrspace.cc定義了我們如何為每個Thread allocate pageTable以及如何將page load到physical memory裡面
- kernel::ForkExecute(Thread *t)

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

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

- 這個function先把Thread t load到MainMemeory內
- 然後call AddrSpace::Execute執行它

---

- AddrSpace::Load(char* fileName)

```
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);
```

- 這個function先用file system把在disk上的thread資料打開，並把它Read進新創立的NoffHeader物件
- 接著做一些endian的轉換
- NoffHeader object使用segmentation，分成三個segment
    1. Code
    2. initData
    3. readonlyData或uninitData
- NoffHeader定義NachOS的object code format

---

```
# 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

DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);
```

- 然後依據是不是ReadOnly data來計算整個thread的大小，再除以PageSize就可以知道需要多少page

```
//handle code segment
if (code_size > 0) {
    DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);

    while(code_size > 0) {
        // cout << "\n[!] code_size remain: " << code_size << endl;
        exception = Translate(virtual_addr, &phyaddr, 0);
        // cout << "[!] code_segment exception: " << exception << endl << "[!]phyaddr: "
<< phyaddr <<endl;
        if (code_size <= PageSize) {
            executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size,
infile_addr);
            code_size = 0;
        } else {
            executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize,
infile_addr);
            code_size -= PageSize;
            virtual_addr += PageSize;
            infile_addr += PageSize;
        }
    }
}

// handle data segment
code_size = noffH.initData.size;
virtual_addr = noffH.initData.virtualAddr;
infile_addr = noffH.initData.inFileAddr;
if (code_size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
    while(code_size > 0) {
        //cout << "[!] data_size remain: " << code_size << endl;
        exception = Translate(virtual_addr, &phyaddr, 1);
        // cout << "[!!]data_segment exception: " << exception << endl;
        if (code_size < PageSize) {
            executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size,
infile_addr);
            break;
        } else {
            executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize,
infile_addr);
            code_size -= PageSize;
            virtual_addr += PageSize;
            infile_addr += PageSize;
        }
    }
}

#ifdef RDATA
code_size = noffH.readonlyData.size;
virtual_addr = noffH.readonlyData.virtualAddr;
infile_addr = noffH.readonlyData.inFileAddr;
```

```
49
50  if (code_size > 0) {
51      DEBUG(dbgAddr, "Initializing read only data segment.");
52      DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
53      while (code_size > 0) {
54          //cout << "[!] RDATA_size remain: " << code_size << endl;
55          exception = Translate(virtual_addr , &phyaddr ,0);
56          // cout << "[!!]RDATA_segment exception: " << exception << endl;
57          if (code_size < PageSize) {
58              executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size,
    infile_addr);
59              break;
60          } else {
61              executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize,
    infile_addr);
62              code_size -= PageSize;
63              virtual_addr += PageSize;
64              infile_addr += PageSize;
65          }
66      }
67  }
68  #endif
```

- 接著把已經讀進thread data的NoffHeader依照它內部的結構一個部分一個部分的load到MainMemory裡面
- 關於Translate以及pageTable如何實作我們放在Report部分
- 最後再把開起來的檔案關掉

---

- AddrSpace::Execute(char* fileName)

```
1   AddrSpace::Execute(char* fileName)
2   {
3
4       kernel->currentThread->space = this;
5
6       this->InitRegisters();      // set the initial register values
7       this->RestoreState();       // load page table register
8
9       kernel->machine->Run();     // jump to the user progam
10
11      ASSERTNOTREACHED();         // machine->Run never returns;
12                  // the address space exits
13                  // by doing the syscall "exit"
14  }
```

- 這個function先把currentThread的AddrSpace設成自己
- 然後修改machine裡面的Program Counter與pageTable
- 接著開始執行Thread
- 因為kernel->machine->Run()不會return回來，所以在call了Run之後要放一個ASSERTNOTREACHED()來偵測是不是執行出了問題讓Run return回來了

---

# 2017OSteam19 MP2 Report

## kernel.h

我們在public的地方，新增一個可以紀錄frame有無被使用過的array。

```
    bool  usedPhysicalPage[NumPhysPages];

```

## kernel.cc

接著將原本的AddrSpace()改為AddrSpace(usedPhysicalPage) 傳入frame的資訊來實作multiprogramming。

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace(usedPhysicalPage);
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;

```

## addrspace.h

將constructor改成可以接usedPhysicalPage指標的參數，然後新增一個used指標。

```
    AddrSpace(bool *usedPhysicalPage);
    bool *used;

```

## addrspace.cc

用used指標指到剛剛在kernel.h所maintain的array，如此一來thread才知道有哪些frame可以用。 原本是在這邊做address mapping，但是我們將它改到Load(char* filename)再做。因為原本是一個thread就佔據整個physical memory，如此一來就無法實作multiprogramming。所以要等到算出thread所佔的page數後再來mapping。

```
AddrSpace::AddrSpace(bool *usedPhysicalPage)
{
    used = usedPhysicalPage;
}

```

---

假如thread要被free掉的時候，將不用的frame改為false，釋放空間出來，並且將Pagetable delete掉。

```
AddrSpace::~AddrSpace()
{
    for (int i = 0; i < NumPhysPages; i++) {
        used[pageTable[i].physicalPage] = false;
    }
    delete pageTable;
}

```

這邊一開始先create一個pagetable。 接著我們改變了6~9行的code。先用numPages的for loop判斷需要給到幾個pages，再用while loop去尋找到可以使用的frame將它改為TRUE，然後將這個資訊給pagetable，把它紀錄下來。最後再紀錄下這個page的 valid、use、dirty、readonly的資訊。

**page table**

| virtualPage | physicalPage | valid | use | dirty | readOnly |
|---|---|---|---|---|---|
| i | find_empty_page | TRUE | False | False | False |

```
pageTable = new TranslationEntry[numPages];
int find_empty_page = 0;
    // cout << "Pages needed: " << numPages << endl;
    for (int i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
        while(used[find_empty_page] == TRUE) {
            find_empty_page ++;
        }
        used[find_empty_page] = TRUE;
        pageTable[i].physicalPage = find_empty_page;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

```

我們將noffH的資料結構傳到我們設的變數裡面，方便programming。

```
    ExceptionType exception;
    unsigned int code_size = noffH.code.size;
    unsigned int virtual_addr = noffH.code.virtualAddr;
    unsigned int infile_addr = noffH.code.inFileAddr;
    unsigned int phyaddr = 0;

```

最後是將page做translate寫到memory的部分。將thread的3個 segment用page為單位一個一個寫回memory。而physical address則是Translate這個function來決定，它會去查pagetable來把virtual address對應的physical address算出來。

再來是將寫到memory的size分成兩種情況:

1. size比pagesize還小的
2. size可以用一個以上pagesize來表示

如果比pagesize小就傳剩下的size進ReadAt就好，如果比較大就以pagesize為單位來切，切到剩下比pagesize小就用 **1.** 的情況解決。

然後每次做完一次translate都要更新 code_size、virtual_addr、infile_addr。最後用code_size來判斷是否做完所有的translate。

```
    //handle code segment
    if (code_size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);

        while(code_size > 0) {
            // cout << "\n[!] code_size remain: " << code_size << endl;
            exception = Translate(virtual_addr, &phyaddr, 0);
            // cout << "[!] code_segment exception: " << exception << endl << "[!]phyaddr: " << phyaddr <<endl;
            if (code_size <= PageSize) {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size, infile_addr);
                code_size = 0;
            } else {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize, infile_addr);
                code_size -= PageSize;
                virtual_addr += PageSize;
                infile_addr += PageSize;
            }
        }
    }

```

# Translate

**vpn:** 算出第幾個virtual pagetable **offset:** 算出offset

把pte拿去指向pagetable[vpn]，再拿pfn紀錄放在pagetable的physicalPage。 然後再利用paddr = pfn*Pagesize + offset算出實體位址，這樣就translate完成了。

```
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)
{
    TranslationEntry *pte;
    int               pfn;
    unsigned int      vpn    = vaddr / PageSize;
    unsigned int      offset = vaddr % PageSize;
    //cout << "[!] Vpn: " << vpn << " numpages: " << numPages << endl;
    if(vpn >= numPages) {
        return AddressErrorException;
    }

    pte = &pageTable[vpn];
    if(isReadWrite && pte->readOnly) {
        return ReadOnlyException;
    }

    pfn = pte->physicalPage;
    // if the pageFrame is too big, there is something really wrong!
    // An invalid translation was loaded into the page table or TLB.
    if (pfn >= NumPhysPages) {
        DEBUG(dbgAddr, "Illegal physical page " << pfn);
        return BusErrorException;
    }

    pte->use = TRUE;          // set the use, dirty bits

    if(isReadWrite)
        pte->dirty = TRUE;

    *paddr = pfn*PageSize + offset;
    cout << "physical page: " << pfn << ", " << "Mainmemory: " << *paddr << endl;
    ASSERT((*paddr < MemorySize));

    //cerr << " -- AddrSpace::Translate(): vaddr: " << vaddr <<
    //  ", paddr: " << *paddr << "\n";

    return NoException;
}

```

---

**initialData** 與上面相同

```
    // handle data segment
    code_size = noffH.initData.size;
    virtual_addr = noffH.initData.virtualAddr;
    infile_addr = noffH.initData.inFileAddr;
    if (code_size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        while(code_size > 0) {
            //cout << "[!] data_size remain: " << code_size << endl;
            exception = Translate(virtual_addr, &phyaddr, 1);
            // cout << "[!!]data_segment exception: " << exception << endl;
            if (code_size < PageSize) {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size,
infile_addr);
                break;
            } else {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize,
infile_addr);
                code_size -= PageSize;
                virtual_addr += PageSize;
                infile_addr += PageSize;
            }
        }
    }

```

**readonly** 與上面相同

```
#ifdef RDATA
    code_size = noffH.readonlyData.size;
    virtual_addr = noffH.readonlyData.virtualAddr;
    infile_addr = noffH.readonlyData.inFileAddr;

    if (code_size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
        DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
        while (code_size > 0) {
            //cout << "[!] RDATA_size remain: " << code_size << endl;
            exception = Translate(virtual_addr , &phyaddr ,0);
            // cout << "[!!]RDATA_segment exception: " << exception << endl;
            if (code_size < PageSize) {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), code_size,
infile_addr);
                break;
            } else {
                executable->ReadAt(&(kernel->machine->mainMemory[phyaddr]), PageSize,
infile_addr);
                code_size -= PageSize;
                virtual_addr += PageSize;
                infile_addr += PageSize;
            }
        }
    }

```

# Group Contribution

陳麒懋**:** trace code , code report , debugging 鍾昀諠**:** trace code , coding , trace report , debugging