OS HW02 GROUP 18

Part 1:Trace Code

1. **Explain function**
    1. **threads/thread.cc**
        a. **Thread::Sleep()**

```
void
Thread::Sleep (bool finishing){
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Sleeping thread: " << name);

    status = BLOCKED;

    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);
}
```

只有 currentThread 可以呼叫此 *Thread::Sleep()*，然後將 currentThread 的 status 設為 BLOCKED，準備做 content switch，接著跑 *FindNextToRun()*，從 Ready_List 中找到下一個要進入 CPU 的 nextThread，並執行 *Run()*，若 Ready_list 裡面沒有任何 element 就不會有任何的執行，等待 interrupt，其中的 argument finishing 若為 true 表示此 thread 執行完成了，等下在後續的 function 中會刪除此 thread。

b. Thread::StackAllocate()

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

#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
    *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
}
```

Thread::StackAllocate() *會經由* *Fork()呼叫*，它有各種的 def 要去對不同的 device 做設置(PARISC，SPARC，PowerPC，decmips，alpha，x86，PARISC)，他會分配以及初始化 stack。

c. **Thread::Finish()**

```
Void Thread::Finish ()
{
	(void) kernel->interrupt->SetLevel(IntOff);

	ASSERT(this == kernel->currentThread);

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

而在 *Thread::Finish()* 會先設置 interrupt 為 IntOff (因 *Sleep()* 的需求)，之後會再呼叫 *Sleep()*，其中這邊輸入的 argument 為 True，表示目前 currentThread 是真的執行完了，在後面的 Sleep(), Run() 中會刪除這個做完的 currentThread。

d. Thread::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);
}
```

Fork() 會使 Thread 和 kernel 裡的物件都指到同一個 interrupt 和 scheduler，然後再利用 *StackAllocate()* 分配和初始化 stack，而 function pointer func 這邊會呼叫 Kernel::ForkExecute，arg 是要傳給 procedure 的參數 (這邊為某一個 Thread)，然後因 *ReadyToRun()* 的要求，將 interrupt 設為 IntOff，之後再執行 *ReadyToRun()*，將目前 this 這個 Thread 物件加入 Ready_List 中，之後再還原 interrupt 的 Level。

2. **userprog/addrspace.cc**
   a. **AddrSpace::AddrSpace()**

```
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);
}
```

AddrSpace() 為 class AddSpace 的 constructor 首先會先創造 pageTable (class TranslationEntry)，然後在 pageTable 裡有 virtualPage (在 virtual memory 所代表的 page number)，physicalPage (在 physical memory 所代表的 page number)，裡面還會存其他資訊 valid bit , use bit , dirty bit , readOnly，其中可以注意的是這邊的 virtual page number = physical page number。

b. AdrSpace::Execute()

```
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;
}
```

而 *Execute()* 它是用 current thread 跑我們要的 user program，首先初始化 register 的值，之後再 load page table 的 register，之後呼叫 *Machine::Run()* 去執行這個 user program。

c. AddrSpace::Load()

```
bool    AddrSpace::Load(char *fileName)
{
        OpenFile *executable = kernel->fileSystem->Open(fileName);
        NoffHeader noffH;
        unsigned int size;
        executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
        if ((noffH.noffMagic != NOFFMAGIC) &&
                (WordToHost(noffH.noffMagic) == NOFFMAGIC))
                SwapHeader(&noffH);
#ifdef RDATA
                // how big is address space?
        size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
                        noffH.uninitData.size + UserStackSize;
#else
                // how big is address space?
        size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
                                        + UserStackSize;   // we need to increase the size
#endif
        numPages = divRoundUp(size, PageSize);
        size = numPages * PageSize;
        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);
        }
```

而一開始會先做開檔（*Open()*）的動作，之後做讀檔（*ReadAt()*）的動作，並將檔案 load 進 nofHH (其為 noffHeader struct 的資料型態，裡面會區分成 4 個 segment，為 code , initData , readonlyData , uninitData)，由此我們就可以知道這些 segment 在檔案中的位置以及大小，並會依照是否為 readonly 來計算 address 的 size 大小，並利用 *divRoundUp()* 來計算要分成幾個 page number，並在之後將 segment load 進 memory 之中

3. threads/kernel.cc

a. Kernel::Kernel()

```
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";
                }
    }
}
```

執行 kernel 的 constructor，會讀取 command line 的內容，並且根據此初始化此 kernel object 的參數，如“-e“ 的那個 if 判斷式會計算要執行幾個程式，並將要執行的程式放進 execfile 裡

b. Kernel::ExecAll()

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

ExecAll() 會對所有的 execfile (要執行的指令或檔案) 做 Exec()，然後呼叫 *Finish()* 來終止執行 *ExecAll()* 的 Thread。

c. Kernel::Exec()

```
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() 會先創造一條新的 Thread 給要執行的程式，之後呼叫 *AddrSpace()* 給 Thread 一個對應的 address space，最後在透過 Fork() 讀取要執行的程式 (Fork() 透過 VoidFunctionPtr 指向要使用的 function ForkExecute())，並累加 total thread 的數量。

d. Kernel::ForkExecute()

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

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

ForkExecute() 會利用 *Load()* function 將要執行的程式 load 進 memory，若 load 失敗或非執行檔，則不會執行，直接 return，若 *Load()* 成功則會用 *Execute()* 去執行 thread

4. **threads/scheduler.cc**

a. Scheduler::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);
}
```

而在 *ReadyToRun()*，它會將 thread 的 status 設為 ready，然後再將 ready 的 thread 放入 ready_list，等待後續執行.

b. Scheduler::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
```

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

        DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " <<
nextThread->getName());
        SWITCH(oldThread, nextThread);
        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();
        }
```

若 finishing 為 true 代表此 currentThread 之前有呼叫過 *Finish()*，這邊用 toBeDestroyed 記錄此要刪除的 thread，接著準備做 content switch，會先儲存 oldThread 的資訊，並將 nexThread 的 status 設為 RUNNING 並且 assign 給 currentThread (即將此 thread 放到 CPU 上跑)，接這便呼叫 *SWITCH()* 進行 context switch，而 content switch 完成後並不會執行下一行程式，因為 CPU 現在的控制權在 nextThread 手上，而當 CPU 的控制權又 content switch 回原本的 thread 時，才會做 *SWITCH()* 後續的程式碼，其會讀取原本的 Thread 的資訊並繼續後續指令的執行，而 *CheckToBeDestroyed()* 會刪除 toBeDestroyed 中記錄要刪除的 thread。

c. Scheduler:: FindNextToRun ()

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

        if (readyList->IsEmpty()) {
            return NULL;
        }
        else {
            Thread *a = readyList->RemoveFront();
            return a;
        }
}
```

FindNextToRun() 會從 Ready_List 中找到下一個要執行的 Thread

2. **Answer question**

1. **Explain how NachOS creates a thread (process), load it into memory and place it into the scheduling queue.**

   在 Kernel::Exec() 裡會使用 t[threadNum] = new Thread(name, threadNum) 去建立新的 thread，然後透過 t[threadNum]->space = new AddrSpace() 去給 thread 創造新的 address 空間去跑 user program，之後在透過在 Fork() 裡的 ReadtoRun()，會把 thread 推到 Ready queue.

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

   在 fork()裡的 StackAllocate()，它會初始化 thread 的 stack，並分配記憶體空間。

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

   會在 Kernel::Exec() 裡創一個 new Thread，並做初始化，之後在給一個 new AddrSpace() 給 thread，之後在 AddrSpace() 裡會執行 bzero() 來清除 Memory 確保裡面是空的，接下來 Thread 會呼叫 Fork()，而 Fork() 裡會需要用到 FunctionPtr ForkExecute ,然後裡面會在需要用到 AddrSpace::Load()，在這邊它會計算可執行檔的大小，去配置相對應可執行 page 的數量，並將 data , code , 等資訊 load 進去 memory 裡。

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

   # pageTable = new TranslationEntry[NumPhysPages];

   定義 TranslationEntry *pageTable (假設是 linear page table translation) ，然後再由 AddrSpace::AddrSpace() 去執行上述放的程式碼 ， NumPhysPages 表示在 page table 裡有幾個 page，其裡面的 entry 是用來做 virtual memory 跟 physical memory 的轉換，而在 load() 時會需要它操作 page table。

5. **How does Nachos translate addresses?**

   在 machine.h 和 addrspace.h 都分別定義了 Translate(ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing); 和 ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);)，而 Translate () 是在檢查 program 使用 Page 的合法性，size 大小是否符合，是否被使用修改過，然後還有透過去計算 page number 和 page offset 去找尋 physical address value，假如最後都沒有回報偵錯，回傳 NoException 表示沒問題。

6. **How Nachos initializes the machine status (register, etc) before running a thread (process)?**

   machineStates 大部分是在 thread.cc 文件中的 Thread::Thread 完成初始化的，並且在調用 Fork() 的時候也會呼叫 StackAllocate 來進一步設定 machineStates，確保執行緒能夠順利運行。

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

   在 class thread 裡面有做一些關於 process control block(PCB) 的資訊以及操作，其中包含紀錄包含儲存 thread 的執行狀態，register 的內容，current stack pointer 等訊息，而操作方面也包含關於 thread 的基本的 operation 其中包含，允許 Scheduler 在 thread 之間進行 context switch 時能夠準確地恢復和管理每個 thread 的 status。

Part 2:Implementation

1. **Detail of your implementation**
    1. **Userprog/addrspace.cc**
        a. **更改 pagetable**

```
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);
}
```

上圖為原本的 pagetable 創造方式，在建立 thread 時就會同時 allocate 給他 pagetable，但是這樣會無法執行 multi-programming，因 physical page 會被覆蓋掉，故我們要在程式 load 進來後再 allocate pagetable，以避免 physical page 重疊。

```
bool AddrSpace::Load(char *fileName)
{
    …
    pageTable = new TranslationEntry[numPages];
    for(int i=0;i<numPages;i++){
        pageTable[i].virtualPage = i;
        // 調整 physical number 的設定，使 page number != frame number
        pageTable[i].physicalPage = kernel->usedPhyPage->setPhyAddr();
        pageTable[i].valid = true;
        pageTable[i].use = false;
        pageTable[i].dirty = false;
        pageTable[i].readOnly = false;

        ASSERT(pageTable[i].physicalPage != -1);
        bzero(kernel->machine->mainMemory + pageTable[i].physicalPage *
                PageSize, PageSize);
    }
    …
}
```

上圖為我們修正的 pagetable allocate 方式，我們延後了 pagetable 的 allocate 時間，到了實際知道執行檔的各個 segment 大小後才建立 pagetable，其中最關鍵的就是我們使用了 *setPhyAddr()* 來設定 thread 的 pagetable 中的 physical page number，以避免 physical page 重複使用。

b. 修正讀 load segment 的方式

```
bool    AddrSpace::Load(char *fileName)
{
        ……
        unsigned int physicalAddr;

        int unReadSize;
        int chunkStart;
        int chunkSize;
        int inFilePosiotion;

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

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

                while(unReadSize > 0) {
                        /* first chunk and last chunk might not be full */
                        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;
                }
        }
```

這邊因為檔案過長分為三頁，我們目標是將各段 segment 給 load 到 virtualAddr 對應的 PhysicalAddr (PhysicalAddr 由上頁設定)，這邊是使用 *AddrSpace::Translate()* 進行 address 的轉換，然而因為 segment 無法一次 load 進 memory，要一個一個 page load 進 memory 中，故需要 chunkSize , inFilePosition 等來記錄已經 load 進 memory 的部分，可注意的是因為是一個一個 page load 進來，故 segment 在 physical 中可能並不連續。此頁為 load segment 的 code 部分。

```
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);

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

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

修改方式相同，此頁為 load segment 的 initData 部分。

```
#ifdef RDATA
	if (noffH.readonlyData.size > 0) {
		DEBUG(dbgAddr, "Initializing read only data segment.");
		DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);

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

			unReadSize = unReadSize - chunkSize;
			chunkStart = chunkStart + chunkSize;
			inFilePosiotion = inFilePosiotion + chunkSize;
		}
	}
#endif

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

若有 readonly 則會執行此頁，修改方式相同，此頁為 load segment 的 readonlyData 部分。

c. 修正 ~AddrSpace()

```
AddrSpace::~AddrSpace()
{
    for(int i = 0; i < numPages; i++)
        kernel->usedPhyPage->pages[pageTable[i].physicalPage] = 0;

    delete pageTable;
}
```

我們會使用 pages 來記錄 physical page 使否有被使用，故在 deconstructor 時，會將 pages 轉回 0 表示釋放掉這個 physical page，且同時刪除這個 pagetable。

2. **Threads/kernel.h**

   a. **增加一個 class**

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

        ~UsedPhyPage(){
                delete[] pages;
        };
        int numUnused(){
                int count = 0;
                for(int i = 0; i < NumPhysPages; i++) {
                        if(pages[i] == 0)
                                count++;
                }
                return count;
        };

        int setPhyAddr(){
              int pickPhyPage = -1;

              while(1){
                  pickPhyPage = rand()%NumPhysPages;
                    if(pages[pickPhyPage] == 0) {
                                break;
                    }
              }
                pages[pickPhyPage] = 1;
                return pickPhyPage;
        };

        int *pages;
};
```

在 kernel.h 中新增一個 UsedPhyPage 的 class 用來做 virtualAddr 和 PhysicalAddr 的 mapping，裡面會利用一個 int array called pages 來記錄那些 Physical page 被使用過了 (沒使用過為 0，有使用過為 1)，而 *setPhyAddr()* 會隨機挑選的 physical page，若此 page 沒人用過則回傳 physical page number 給 virtualAddr 做 mapping，若全部都使用過則回傳 -1，表失敗，而 *numUnused()* 則用來回傳沒被使用的 page 數量。

2. Implementation results

TA 模擬結果

```
===================================
Running the test: 1
===================================
9
15
16
17
18
19
8
7
6
===================================
Running the test: 2
===================================
20
22
24
26
28
30
32
34
36
38
40
15
16
17
18
19
9
8
7
6
done
OK (2.018 sec real, 2.020 sec wall)
Finished: SUCCESS
```

我們有額外的調整 stats.h 裡的參數 tick，調整 instruction 和 interrupt 的相對時間關係，可實現 interleave 的效果

```
===================================
Running the test: 1
===================================
9
15
8
16
7
17
6
18
19
===================================
Running the test: 2
===================================
20
15
9
22
16
8
24
17
7
26
18
6
28
19
30
32
34
36
38
40
done
OK (2.020 sec real, 2.020 sec wall)
Finished: SUCCESS
```

Part 3:Contribution

1. **Describe details and percentage of each member's contribution.**

	張世傑 A101156	吳孟儒 A111121
Part I - Trace files	50 %	50 %
Part I - Question answering	60 %	40 %
Part I - report	40 %	60 %
Part II - implementation	50 %	50 %
Part II - report	50 %	50 %