MP1: SYSTEM CALL

Team 21

Operating System

(a) Team List & Contibutions:

Report question explanation: 107061334 樊明膀

Code Implement: 107062333 湯睿哲

(B)Trace code

void Kernel::ExecAll()

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

這個 function 裡面 call 了一個 for 迴圈，做 Exec，execfileNum 是要做的數量，做完迴圈之後讓現在的 thread 進到 finish

currentThread 是目前，掌握著 CPU 的 thread

```
Kernel::Kernel(int argc, char **argv)
{
```

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

execfileNum 的數字的處理在 kernel 的創建的時候，在處理 command 的時候，如果 command 有-e 的話數量就會增加，並且使 execfile 對應到輸入的字上面，對應到的是-e 後面的字串。

```
char*    execfile[10];
```

Execfile 是 char*的型別，用來存儲字串，在 kernel.h 裡面

int Kernel::Exec(char* name)

```
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;
```

在 Exec 裡面第一步先創一個新的 Thread
這裡的 name 是當初 execfile 裡面的字串，也就是-e 後面那一個

```
Thread* t[10];
```

T 是 Thread 的 pointer

```
currentThread = new Thread("main", threadNum++);
```

Threadnum 在 Kernel::Initialize()會增加

第二行則是創建了 AddrSpace

```
AddrSpace *space;      // User code this thread is running.
```

space 是 AddrSpace 的 pointer，AddrSpace 是一個資料的結構，用來記錄使用者 memory 程式的狀況，後面會解釋
第三是 thread 去 call 了 fork，可以讓 thread run，後面會在解釋
最後是 threadnum 的數量增加

AddrSpace::AddrSpace()

```
// AddrSpace::AddrSpace
//  Create an address space to run a user program.
//  Set up the translation from program memory to physical
//  memory.  For now, this is really simple (1:1), since we are
//  only uniprogramming, and we have a single unsegmented page table
//----------------------------------------------------------------------

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);

}
```

首先是創造一個 pagetable 用 TranslationEntry 這個 class，裡面有分別幾個數字對應到位置跟資訊等等，virtualpage、physicalpage、vaild、readonly、use、dirty

```
class TranslationEntry {
  public:
    int virtualPage;     // The page number in virtual memory.
    int physicalPage;    // The page number in real memory (relative to the
      //  start of "mainMemory"
    bool valid;          // If this bit is set, the translation is ignored.
      // (In other words, the entry hasn't been initialized.)
    bool readOnly;   // If this bit is set, the user program is not allowed
      // to modify the contents of the page.
    bool use;            // This bit is set by the hardware every time the
      // page is referenced or modified.
    bool dirty;          // This bit is set by the hardware every time the
      // page is modified.
};
```

Machine/Translate.h

```
// Definitions related to the size, and format of user memory
const int PageSize = 128;          // set the page size equal to
                                   // the disk sector size, for simplicity

// You are allowed to change this value.
// Doing so will change the number of pages of physical memory
// available on the simulated machine.
const int NumPhysPages = 128;

const int MemorySize = (NumPhysPages * PageSize);
const int TLBSize = 4;             // if there is a TLB, make it small
```

Machine/machine.h

NumPhysPages 的數量或是其他的像是 pagesize、memorysize、TLBsize 在這上面

然後上面因為是 uniprogram 所以在做迴圈的時候，把所有的 NumPhysPages，都對應上去了，這樣的就完全用完 PhysPages 給的 memory 了

```
bzero(kernel->machine->mainMemory, MemorySize);
```

這個 function 把 machine 的 memory 資料全部給清空

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

exec 第三行 Thread 呼叫了 Fork，使 thread 可以運作，允許 caller and callee 的運作同時進行

```
void Fork(VoidFunctionPtr func, void *arg);
                // Make thread run (*func)(arg)
```

```
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);
}
```

傳了一個 function(ForkExecute)跟一個 pointer(thread*)進去，然後取了 kernel 的 interrupt,scheduler，以及創一個 IntStatus

```
IntStatus SetLevel(IntStatus level);
                // Disable or enable interrupts
        // and return previous setting.
```

可以停止或起動 interrupt 但這邊是停止，並將回傳前一個的設定存在 oldLevel 裡面

```
IntStatus
Interrupt::SetLevel(IntStatus now)
{
    IntStatus old = level;

    // interrupt handlers are prohibited from enabling interrupts
    ASSERT((now == IntOff) || (inHandler == FALSE));

    ChangeLevel(old, now);          // change to new state
    if ((now == IntOn) && (old == IntOff)) {
    OneTick();                      // advance simulated time
    }
    return old;
}
```

到最後面的時候在加到 readyqueue 的後面，有在將他設回 oldlevel

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

啟動且設置好 stack，func 是要 fork 的 procedure，*arg 是要給 procedure 的 parameter

```
// Size of the thread's private execution stack.
// WATCH OUT IF THIS ISN'T BIG ENOUGH!!!!!
const int StackSize = (8 * 1024); // in words
```
StackSize 的大小

```
char *
AllocBoundedArray(int size)
{
#ifdef NO_MPROT
    return new char[size];
#else
    int pgSize = getpagesize();
    char *ptr = new char[pgSize * 2 + size];

    mprotect(ptr, pgSize, 0);
    mprotect(ptr + pgSize + size, pgSize, 0);
    return ptr + pgSize;
#endif
}
```
sysdep.cc

設置 stack 大小的 function
pgSize 就是 page 的 size，stack 的大小是 page*2+StackSize(上面有查到)
mprotect 就是將這一塊 memory 保護起來不要讓別人用

fork 最後 call scheduler->ReadyToRun(this);
this 就是這個 thread 使他加入到 ready queue，ReadyToRun 把這個 thread 標記成 ready，並加入到 readylist

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

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);
}
```

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

在 StackAllocate 的最後設置了 machineState 等等的東西

```
extern "C" {
// First frame on thread execution stack;
//     call ThreadBegin
//   call "func"
//   (when func returns, if ever) call ThreadFinish()
void ThreadRoot();
```

Threadroot 是 thread 在執行時候的框架 從 begin 到 func 直到結束 call finish 裡面有初始化需要用到的 register，以便於傳給 threadRoot 值，第一組的 register 是給 threadroot 用的，第二組是 thread 這個 object 裡面用的在 StackAllocte()也就是上面那邊

```
   The following are the initial registers we need to set up to
 * pass values into ThreadRoot (for instance, containing the procedure
 * for the thread to run).  The first set is the registers as used
 * by ThreadRoot; the second set is the locations for these initial
 * values in the Thread object -- used in Thread::StackAllocate().
 */
#define InitialPC    s0
#define InitialArg   s1
#define WhenDonePC   s2
#define StartupPC    s3

#define PCState      (PC/8-1)
#define FPState      (S6/8-1)
#define InitialPCState  (S0/8-1)
#define InitialArgState (S1/8-1)
#define WhenDonePCState (S2/8-1)
#define StartupPCState  (S3/8-1)
```

Func 的 pointer 是 forkExecute 這一個，arg 就是那個 thread 的 pointer

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

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

}
```

thread/kernel.cc

bool AddrSpace::Load(char *fileName)

將 user program 從 file 變成 memory

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

userprogram/addrspace

首先先打開 file，"fileName" is the file containing the object code to load into memory，而我們的 object code 也是一種 noff 型式

```
    NoffHeader noffH;
```

```
Assumes that the page table has been initialized, and that
the object code file is in NOFF format.
```

```
typedef struct noffHeader {
    int noffMagic;      /* should be NOFFMAGIC */
    Segment code;       /* executable code segment */
    Segment initData;     /* initialized data segment */
#ifdef RDATA
    Segment readonlyData;    /* read only data */
#endif
    Segment uninitData;      /* uninitialized data segment --
                 * should be zero'ed before use
                 */
} NoffHeader;
```

program/noff.h

noffH 是上面這個，是一種 data structures 用來分三個 Segment，Segment 分三個部分

```
 *     Data structures defining the Nachos Object Code Format
 *     Basically, we only know about three types of segments:
 * code (read-only), initialized data, and unitialized data
```

```
typedef struct segment {
  int virtualAddr;      /* location of segment in virt addr space */
  int inFileAddr;     /* location of segment in this file */
  int size;         /* size of segment */
} Segment;
```

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

Openfile 去 call readat

```
Int OpenFile::ReadAt(char *into, int numBytes, int position)
```

Filesys/Openfile.cc

去讀一部分的 file，從傳入的 position 開始，並回傳實際讀到的 byte 數量

```
//  "into" -- the buffer to contain the data to be read from disk
//  "numBytes" -- the number of bytes to transfer
//  "position" -- the offset within the file of the first byte to be
```

```
There is no guarantee the request starts or ends on an even disk sector
boundary; however the disk only knows how to read/write a whole disk
sector at a time.  Thus:

For ReadAt:
   We read in all of the full or partial sectors that are part of the
   request, but we only copy the part we are interested in.
```

在我們讀取的時候，disk 只會處理整個 sector，因此在 readat 的時候就算讀到了整個 sector，我也只會要我有興趣的部分。

```
 bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
```

裡面最主要的是這一行，把前面處理好的 sector 的資料現在在 buf 裡面，然後把要的資料丟到 into 裡面，然後看幾個 byte

```
    if ((noffH.noffMagic != NOFFMAGIC) &&
          (WordToHost(noffH.noffMagic) == NOFFMAGIC))
          SwapHeader(&noffH);
```

繼續上面的 load 的部分，這一個部分確認了一個 object code file

```
#define NOFFMAGIC 0xbadfad    /* magic number denoting Nachos
```

不符合的話就處理 swapheader，這部分 byte 的 endian 有關係，怕 file 的讀法跟 nachos 的不一樣

```
//----------------------------------------------------------------------
// SwapHeader
//  Do little endian to big endian conversion on the bytes in the
//  object file header, in case the file was generated on a little
//  endian machine, and we're now running on a big endian machine.
//----------------------------------------------------------------------
```

```
    ASSERT(noffH.noffMagic == NOFFMAGIC);
```
一樣是在確認 noffH.noffMagic 這個數字

繼續 load 的部分，確保 load 的不會超出 Numphyspages

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

void AddrSpace::Execute(char* fileName) /addrspace

```
/ AddrSpace::Execute
/  Run a user program using the current thread
/
/      The program is assumed to have already been loaded into
/      the address space
/
```

```
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;
                                // the address space exits
                                // by doing the syscall "exit"
}
```
現在的 thread 的 space 直接換成 this(addrspace)

然後初始化 register value，跟讀取 table register，讓 user program run 下去，然後就一直 run 下去，除非有 system call “exit”

```
//----------------------------------------------------------------------
// AddrSpace::InitRegisters
//  Set the initial values for the user-level register set.
//
//  We write these directly into the "machine" registers, so
//  that we can immediately jump to user code.  Note that these
//  will be saved/restored into the currentThread->userRegisters
//  when this thread is context switched out.
//----------------------------------------------------------------------
```
AddrSpace::InitRegisters()

初始化 user level Register，直接將 register 寫到 mechine 就可以快速地到我們的 code 了

```
for (i = 0; i < NumTotalRegs; i++)
    machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);
    machine->WriteRegister(NextPCReg, 4);
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
```

有一般 cpu 裡面的 register，然後 PC Reg, NextPCReg,StackReg 都回到要的 program 所要的

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

AddrSpace:RestoreState()
在 Context switch 的時候，要找回原本的 page Table 不然對應的 physical address 會不一樣

```
kernel->machine->Run();      // jump to the user progam
```

這個在上一份作業解釋過了，因為 register 也都就位了，所以就可以 run 了

Thread::Sleep (bool finishing)

目前的 thread 可能因為做完了，或是有什麼東西需要 waiting，所以要先放棄掉，去做 ready 裡面的 thread，之後的 thread 可能做完了一些事讓這個放棄的 thread 又可以運作了，就會被放到 ready queue 中了
如果 ready 沒事可以做就會"Interrupt::Idle"
但這邊先假定 interrupt disable 了，這樣才可以從 ready 裡面拿 thread

```
NOTE: we assume interrupts are already disabled, because it
is called from the synchronization routines which must
disable interrupts for atomicity.   We need interrupts off
so that there can't be a time slice between pulling the first thread
off the ready list, and switching to it.
```

```
ASSERT(this == kernel->currentThread);
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

確認要睡覺的 thread 是現在 kernel 裡面的 thread
確認 interrupt 的狀況，才可以進行 sleep

```
    status = BLOCKED;
```
將這個 thread 的狀態設為 Blocked

```
while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle();   // no one to run, wait for an interrupt
}
```
nextThread 從 kernel 的 scheduler 裡面拿下一個，就是之前設成 ready 加入到 scheduler 裡面的 readylist 裡面，有的話把他 pop 出來

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```
如果沒有下一個就讓他 interrupt -> idle

```
kernel->scheduler->Run(nextThread, finishing);
```
去 run 下一個 thread

Void Scheduler::Run (Thread *nextThread, bool finishing)

這個在 Scheduler，將原本的 thread 停掉，並將下一個 thread 布置好

```
    Note: we assume the state of the previously running thread has
  already been changed from running to blocked or ready (depending).
```
在 sleep 裡面有將原本的 state 設 block 了

nextThread 是下一個要 run 的 thread

finishing 是原本跑的 thread，看是不是做完結束了，如果是結束的要把他 destory 掉

```
    Thread *oldThread = kernel->currentThread;
```
oldthread 紀錄原本的 thread

```
    ASSERT(kernel->interrupt->getLevel() == IntOff);
```
確認 interrupt disable

```
if (finishing) {    // mark that we need to delete current thread
    ASSERT(toBeDestroyed == NULL);
 toBeDestroyed = oldThread;
}
```
原本的 thread 如果是結束的，需要 destory 掉，這邊是將他記在 toBeDestroyed

```
    CheckToBeDestroyed();    // check if thread we were running
```

在後面的時候有 call 這個去 destory 原本的 thread，如果 toBeDestroyed 有的話，等 destory(delete)之後，toBeDestroyed 設回 null

```
void Scheduler::CheckToBeDestroyed(){
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
    toBeDestroyed = NULL;
    }
}
```

```
if (oldThread->space != NULL) { // if this thread is a user program,
    oldThread->SaveUserState();      // save the user's CPU registers
oldThread->space->SaveState();
}
```

記好原本的 thread 的狀態像是 register、page table 或是 state

```
void
Thread::SaveUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
    userRegisters[i] = kernel->machine->ReadRegister(i);
}
```

thread.cc

```
void AddrSpace::SaveState()
{}
```

現在被說不需要記住東西

```
oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow
```

Chech 原本的 thread 有沒有 overflow 的情況發生

如果得到奇怪的結果(seg faults)可能就是這邊有出問題，因為我們的 complier 沒有做檢查，要處理的話，我們可能要在 stack 的 size 做處理

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

把 kernel 的 thread 轉到 nextThread 上面，並將他 state 設為 running

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

在 SWITCH.s 裡面，整理來說 就是將原本的 thread 的 register 上面的東西存好，然後將下一個 thread 的東西載入下來

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
```

```
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack
 into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage
        movl    8(%esp),%eax            # move pointer to t2 into eax
        movl    _EAX(%eax),%ebx        # get new value for eax into ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax        # restore return address into eax
        movl    %eax,4(%esp)    # copy over the ret address on the stack
        movl    _eax_save,%eax
        ret
```

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

AddrSpace:RestoreState() 前面有了

Thread::RestoreUserState() 在下面，敘述說這是在處理 user code 的，並不是 executing kernel code

```
void Thread::RestoreUserState(){
    for (int i = 0; i < NumTotalRegs; i++)
    kernel->machine->WriteRegister(i, userRegisters[i]);
}
```

如果沒有刪除的話 就要再回來，因為一個 thread 無法停掉自己，會需要別的 thread 然後 call finish，因此前面有描述一個副作用是 kernel->currentThread becomes nextThread.

```
Side effect:
 The global variable kernel->currentThread becomes nextThread.
```

Actually switch to the next thread by invoking *Switch()*. After *Switch* returns, we are now executing as the new thread. Note, however, that because the thread being switched to previously called *Switch* from *Run()*, execution continues in *Run()* at the statement immediately following the call to *Switch*.

If the previous thread is terminating itself (as indicated by the *threadToBeDestroyed* variable), kill it now (after *Switch()*). As described in Section 3, threads cannot terminate themselves directly; another thread must do so. It is important to understand that it is actually another thread that physically terminates the one that called *Finish()*.

上面打得有點雜，解釋助教要的**at least** cover answer for the questions

- How Nachos allocates the memory space for new thread(process)?

Thread::Exec 裡面 call Fork

Thread::Fork 裡面 call Stackallocate

Stackallocate 裡面 call AllocBoundedArray

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

主要是在 AddrSpace::Load(char *fileName)

裡面用 openfile 打開之後，使用 readat，把要的資料讀出來，讀的時候 disk 只會給你 sector 上的資料，還要把要的部分 copy 出來

- How Nachos creates and manages the page table?

在 Exec()裡面

t[threadNum]->space = new AddrSpace();

AddrSpace::AddrSpace() 去 create page table

Page table 的 manage 跟 TranslationEntry 裡面的變數有關係，在創建使用的時候都會去設定裡面的 page table 裡面的變數

- How Nachos translates address?

```
// Translate virtual address _vaddr_
// to physical address _paddr_. _mode_
// is 0 for Read, 1 for Write.
ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);
```

```
TranslationEntry *pte;
int              pfn;
unsigned int     vpn    = vaddr / PageSize;
unsigned int     offset = vaddr % PageSize;
```

Vpn 先用 vaddr /pagesize ，offset 是 vaddr%pagesize

pte = &pageTable[vpn]; 取得 vpn 位置的 page table 值

pfn = pte->physicalPage; 就是 visual address 對應到的值

```
if(vpn >= numPages) {
    return AddressErrorException;
}
```

vpn 不能夠大於他 visual 的限制

```
if(isReadWrite && pte->readOnly) {
    return ReadOnlyException;
}
```
如果是要寫的話，但 pte 是只能夠 read 一樣 error

```
if (pfn >= NumPhysPages) {
    DEBUG(dbgAddr, "Illegal physical page " << pfn);
    return BusErrorException;
}
```
pte 是 physicalpage 位置，不能夠超過

```
    pte->use = TRUE;           // set the use, dirty bits
    pte->dirty = TRUE;
```
這兩個，控制 page table 裏面的變數

```
    *paddr = pfn*PageSize + offset;
```
取得妳要的最後的值

```
    return NoException;
```
代表轉換沒問題

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

Thread::StackAllocate 裡面負責的是最一開始的初始化

```
#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
}
```

但如果是每次 run 之前的話

AddrSpace::InitRegisters()

```
for (i = 0; i < NumTotalRegs; i++)
    machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);
    machine->WriteRegister(NextPCReg, 4);
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
```

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

```
// The following class defines a "thread control block" -- which
// represents a single thread of execution.
//
//  Every thread has:
//     an execution stack for activation records ("stackTop" and "stack")
//     space to save CPU registers while not running ("machineState")
//     a "status" (running/ready/blocked)
//
//  Some threads also belong to a user address space; threads
//  that only run in the kernel have a NULL address space.

class Thread {
```

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

Thread::Exec 裡面 call Fork

fork 最後 call scheduler->ReadyToRun(this);

ReadyToRun 將 thread 加入到 readylist

(c) Code Implements

1. PageTable iniytial

原本 AdderSpace.cc 裡，由於在未更改前會在一宣告 AdderSpace 物件時，initial 就會創建一個與 memory 一樣大的 pageTable，造成不必要的浪費，但事實上我們只需要創建一個與讀取到程式一樣大小的 pageTable 就行了，因此可以先將 AdderSapce::AddrSpace()裡的 code 挪至 AddrSpace::Load()中，等取得 program size 後再進行 pageTable 的創建。

AdderSpace::AddrSpace()

```
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);
    */
}
```

AdderSpace::Load()

```
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    pageTable = new TranslationEntry[numPages];
    for (int i = 0, index = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;
        while(index<NumPhysPages && UsedPhysPages[index]==TRUE)index++;
        UsedPhysPages[index] = TRUE;
        pageTable[i].physicalPage = index;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

    ASSERT(numPages <= NumPhysPages);       // check we're not trying
```

上圖是對 physicalPage 進行分配以及 PageTable 的 initial，用一 while 迴圈逐一查看有沒有能使用的 physicalPage 控間，有的話就抓起來使用，並在 PageTable 中做映射位置的紀錄，並在該 PageTable 標記 valid = True, use = False, dirty = Fasle, readOnly = False。

另外，我在 kernel.h 中宣告 static UsedPhysPages 來記錄 physical memory 的使用狀況，供每個 process 參考。用 static 是避免重複宣告造成資料錯誤。

`ASSERT(numPages <= NumPhysPages);` 是用來判斷程式大小是否超過記憶體大小。

2. 更改讀取位置

接下要處理的是將 `code segments` 與 `data segments` 讀取的動作更改:

原本在假設 virtualMemory 大小等於 physicalMamory 時，只會放入檔案紀錄的 virtualMemory，但經過上步驟我們開始針對不同程式使用 pageTable，我們開始需要換算映射的正確 physicalMemory 位置。正確位置為:

Page_entry 所對應的 physicalPage * PageSize(offsets) + location in physicalPage_th

= pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize +
noffH.code.virtualAddr % PageSize (for code segments)

= pageTable[noffH.data.virtualAddr/PageSize].physicalPage * PageSize +
noffH.code.virtualAddr % PageSize (for ata segments)

```
if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
            executable->ReadAt(
        &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize + (noffH.code.virtualAddr%PageSize
            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[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage * PageSize + (noffH.code.virtualAddr%Page
            noffH.initData.size, noffH.initData.inFileAddr);
    }
```

每次執行結束後，需要對 physicalPage 進行 free，以供後面程式使用:

```
AddrSpace::~AddrSpace()
{
    for(int i = 0; i < numPages; i++) {
        UsedPhysPages[pageTable[i].physicalPage] = FALSE;
    }
   delete pageTable;
}
```