MP2 Multi-Programming

111062502

涂博允

Trace code: threads/thread.cc Thread::Sleep()

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread; 指向下一條thread的位址

    ASSERT(this == kernel->currentThread); 目前的thread就是當前kernel的thread，若不是就abort()
    ASSERT(kernel->interrupt->getLevel() == IntOff);目前interrupt的getlevel是Off狀態

    DEBUG(dbgThread, "Sleeping thread: " << name);
    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);

    status = BLOCKED; 目前的thread狀態設為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);
}
```

持續檢查是否有其他thread可以執行，如果沒有則進入Idle 狀態，直到interrupt的發生後，會再次檢查是否有其他thread可以執行。

當有其他thread可以執行時，會選擇nextthread去執行，且可以在 finishing 設為 true 時結束currentthread的執行。

Trace code: threads/thread.cc 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
                                        // least 1 activation record
                                        // to start with.
    *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
}
```

StackAllocate()的用途是為了一個新的thread配置一個Stack空間，用來存該thread在run時的local variable、call func. Return的addr.以及其他相關資訊。

一開始會先通過 AllocBoundedArray 函數配置一塊固定大小的stack空間，然後對不同的平台做一些特殊處理，以確保stack的正確配置。最後，將stack相關的資訊保存到一個叫做 machineState 的array中。

Trace code: threads/thread.cc 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
}
```

用來將currentThread的狀態設置為end，並且讓其他thread有機會執行。

Trace code: threads/thread.cc 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);
}
```

Allocate a stack

Put the thread on the ready queue

將interrupt恢復到 oldLevel 的原始狀態，以繼續執行之前可能被中斷的程式碼

Trace code: userprog/addrspace.cc AddrSpace::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);
}
```

新增一個pagetable，然後將virtual page與physical page都map到同一個，並作初始化。

Trace code: userprog/addrspace.cc AddrSpace::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;
                    // the address space exits
                    // by doing the syscall "exit"
}
```

將currentthread的addr. space與caller link起來

初始化暫存器以及load 執行的程式所需的page table

程式執行

Trace code: userprog/addrspace.cc AddrSpace::Load()

```
bool
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);

#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);

// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
    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);
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
    DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

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

開啟可執行檔案

如果檔案不存在，回傳 FALSE

讀取檔案標頭

如果檔案標頭的 magic number 不符合，則進行轉換

計算要載入的程式大小

Trace code: threads/kernel.cc 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";
        }
    }
}
```

設置系統預設值

-rs，將randomSlice設定為TRUE，然後取下一個參數設定randomseed。
RandomInit是一個初始化亂數產生器的函數。

debugUserProg設為TRUE

表示要執行某個可執行文件，將argv[i+1]的值設為execfile中的一個元素，execfileNum用於紀錄目前要執行的程序數量。並在控制台上顯示該可執行文件的名稱。

表示要指定輸入的控制台文件，將argv[i+1]的值設為consoleIn變數，表示從這個文件讀取輸入

指定控制台輸出文件名

Disk format

列出所有可用的命令

Trace code: threads/kernel.cc Kernel::ExecAll()

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

利用for迴圈將程式一一讀取進來，接著透過Exec()執行程式，變數execfileNum則是已經讀取進來的程式數量。

每次呼叫 Exec() 時，會return讀取到的程式，並將執行結果存到變數 a 中。

當所有程式都執行完畢後，會呼叫currentThread->Finish()，結束currentThread的執行，讓其他thread有機會執行

Trace code: threads/kernel.cc Kernel::Exec()

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace(); 為新的thread的space配置一塊新的addrspace
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]); 讓新的thread執行 ForkExecute() ，將
                                                                              t[threadNum] ptr轉換為 void * 作為引數傳入。
    threadNum++;

    return threadNum-1; 回傳thread的index
/*
```

Trace code: threads/kernel.cc Kernel::ForkExecute()

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) {  檢查執行檔是否可被讀取
        return;             // executable not found
    }

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

}
```

Trace code: threads/scheduler.cc Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff); 確保interrupt有開
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY); 將狀態設為ready
    readyList->Append(thread); 要排隊，因此放到readylist的末尾
}
```

Trace code: threads/scheduler.cc Scheduler::Run()

將nextThread切換到CPU上運行，同時將當前thread從CPU切換下去

```
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());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    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();
    }
}
```

Implementation：userprog/addrspace.cc Page table building

預設的nachOS只會執行一個可執行檔，也因此在addrspace創建的page table會直接涵蓋整個physical page，所以不需要做memery mapping，但若要執行多個可執行檔時，會發現單一個thread不該持有整個physical page，
因此將建立page table的時間往後移，在load executable file後，確認file的header大小後，才建立page table。

```
    numPages = divRoundUp(size, PageSize);
    ASSERT(numPages < kernel->usedPhyPage->numUnused());

    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;

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

checkAndSet() 會return一個空的 physical page 的 page number，如果整個physical page 都已經被使用，return -1

Implementation : threads/kernel.h put the data structure

在kernel.cc中額外增加class UsePhyPage用於記錄Page有無被使用過

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

Implementation : NachOS executable file

在NachOS操作系统中，當需要執行一個可執行檔時，系統需要將該檔案從disk中load到內存中，以便程式可以在CPU上運行。每個可執行檔都被分成四個部分：header、code、initData 、 readonlyData。
在load可執行檔之前，NachOS會先讀取可執行檔的header，以獲取metadata。

```
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;

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;
```

Implementation : NachOS modify file loading

Multithread的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(External Fragmentation問題)，因此資料把page 一個一個load 進 memory ，而不能整段 segment load 進 memory

```
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;
    inFilePosition = 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 + inFilePosition);

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

```
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;
    inFilePosition = 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 + inFilePosition);

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

```
#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;
        inFilePosition = 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 + inFilePosition);

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

Question

- How does Nachos allocate the memory space for a new thread(process)?
- How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?
- How Nachos initializes the machine status (registers, etc) before running a thread(process)

在NachOS中，要執行一個程式時，會先建立一條thread，並作一些初始化，再建立一個新的Addrspace給該條thread，在此之後thread會呼叫fork()。
此時Fork接收到ForkExecute(現實要執行的程式)的FunctionPtr，且這個 thread會把 所要執行的function和argument丟進 StackAllocate

```
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;
/*
```

```
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);
}
```

```
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
                                        // least 1 activation record
                                        // to start with.
    *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
}
```

machineState[InitialPCState]=(void*)func;代表func是之後要執行的程式，然後先將interrupt 功能關掉，在ReadyToRun()把要執行的threa放入ready queue ，因為CPU會load Ready裡的程式、讀取其PC值。

machineStates主要都是在此時初始化

Question

- How does Nachos create and manage the page table?
- How does Nachos translate addresses?

translate.h裡面有定義TranslationEntry ，例如管理page table會用到valid、dirty等等，class UsedPhyPage 用來記錄Page有無使用過， ，在addrspace.cc中，建立pageTable 後利用Translate() 將 virtual addr 轉換成physical addr

```
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    if (tlb == NULL) {        // => page table => vpn is index into table
    if (vpn >= pageTableSize) {
        DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
        return AddressErrorException;
    } else if (!pageTable[vpn].valid) {
        DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
        return PageFaultException;
    }
    entry = &pageTable[vpn];
    } else {
        for (entry = NULL, i = 0; i < TLBSize; i++)
            if (tlb[i].valid && (tlb[i].virtualPage == ((int)vpn))) {
        entry = &tlb[i];              // FOUND!
        break;
        }
    if (entry == NULL) {              // not found
            DEBUG(dbgAddr, "Invalid TLB entry for this virtual page!");
            return PageFaultException;      // really, this is a TLB fault,
                        // the page may be in memory,
                        // but not in the TLB
    }
    }
```

```
entry->use = TRUE;       // set the use, dirty bits
if (writing)
entry->dirty = TRUE;
*physAddr = pageFrame * PageSize + offset;
ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
DEBUG(dbgAddr, "phys addr = " << *physAddr);
return NoException;
```

Question

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

PCB (process control block) 包含和 process / thread 相關的一些資訊，例如 :

- Process Number
- Program Counter
- Process State
- CPU register
- CPU Scheduling Information
- Memory Management Information

在class Thread中，能看到很多類似PCB的東西

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    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
                // NOTE -- thread being deleted
                // must not be running when delete
                // is called

    // basic thread operations

    void Fork(VoidFunctionPtr func, void *arg);
                 // Make thread run (*func)(arg)
    void Yield();        // Relinquish the CPU if any
            // other thread is runnable
    void Sleep(bool finishing); // Put the thread to sleep and
            // relinquish the processor
    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:
    // some of the private data for this class is listed above

    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   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
                 // Allocate a stack for thread.
            // Used internally by Fork()

// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    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.
};
```

Question

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

在建好thread的stack後，呼叫scheduler->ReadyToRun(this)，這會把該條thread放入ready queue中，未來CPU scheduler就會安排執行。

```
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);
}
```