CS342301: Operating System
MP2: Multi-Programming

Team no. 46

Member Contribution:

陳敬中: Trace Code, Report, Implementation (50 %)

白宸安: Trace Code, Report, Implementation (50 %)

MP2

Part 1. Trace Code & Questions

1-1. threads/thread.cc

void Thread::Sleep(bool finishing) is responsible for suspending the current thread. It is called under two circumstances: 1. the current thread has finished; 2. the current thread has to wait. The argument “finishing” is set if Sleep() is called under the former circumstance. Sleep() first ensures that the calling thread is the one currently running and that the interrupts are disabled. The current thread is blocked from execution. Then, Sleep() calls the scheduler to find a ready thread to run. If there is none, the CPU idles until a scheduled interrupt is due or there is another thread in ready queue. Eventually, it calls kernel→scheduler→Run() to run the next ready thread and delete the current thread if it has finished.

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

Thread::StackAllocate(VoidFunctionPtr func, void *arg) is used to allocate and initialize a execution stack for a forked procedure to be executed. The stack is initialized with a stack frame for ThreadRoot. ThreadRoot calls Thread::Begin() to prepare for the forked thread to run, deleting all previous finished threads and enabling interrupts. Then, the machine executes the function pointed by “func” and arguments are passed through “arg.” Once the forked thread is done, it calls Thread::Finish() to end itself.

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

void Thread::Finish() is called when a forked thread has finished execution. It disables the interrupts because Sleep() works only when interrupts are turned off. “finishing” is true when it calls sleep to indicate that the calling thread has finished and is to be deleted. Note that Sleep is called to end a forked thread because the CPU is still running on that thread and its stack. We can only delete the thread after we have switched to another thread.

```
void Thread::Finish() {
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    Sleep(TRUE);        // invokes SWITCH
}
```

void Thread::Fork(VoidFunctionPtr func, void *arg) is used to fork a thread. It first calls StackAllocate() to allocate space for the forked thread. Then, it disables the interrupts so that it can call the scheduler to put the calling thread (”this”) to the ready queue.

```
void Thread::Fork(VoidFunctionPtr func, void *arg) {
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;
    StackAllocate(func, arg);
    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

1-2. userprog/addrspace.cc

AddrSpace::AddrSpace() is responsible for creating an address space for a user program. It sets up a sets up a page table to map virtual address to physical address. Here, the whole physical address space are allocated to whoever called AddrSpace(), so only one user program has access to the physical memory at a time. After initializing the page table, it also initializes the entire main memory to zero.

```
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);
}
```

bool AddrSpace::Load(char *filename) loads the user program into the physical memory. The user program is the binary code in the executable file specified by the “filename.” The NOFF header is first read from the file to handle the loading process. The NOFF header contains information of the code segment, data segment and the readonly data segment. The number of pages is calculated by dividing the size of the program by the size of a page, rounded up. Then, it ensures that the number of pages does not exceed the capacity of the address space. If everything is alright, it then loads the code and the data into the memory directly. Here, the virtual address space is identical to the physical address space. Note that in MakeFile, RDATA is defined.

```
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
}
```

void AddrSpace::Execute(char *fileName) executes the user program after it is loaded into the address space. It calls the currentThread→space to point to the address space, initializes the registers and page table registers, and calls the machine to run the program.

```
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
}
```

1-3. threads/kernel.cc

Kernel::Kernel(int argc, char **argv) interprets the command line arguments to determine flags for initialization. If we would like to run multiple programs at a time, we can use the “-e” flag. For each “-e” flag, the kernel adds a executable file to the execfile array.

```
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";
    }
    }
}
```

void Kernel::ExecAll() calls Exec() to execute all the user programs. And it calls Finish() to end the current thread and starts running all the process in the ready queue.

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

int Kernel::Exec(char *name) creates a new thread object, allocates the thread an address space and calls Fork() to execute the file concurrently.

```
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 ForkExecute(Thread *t) loads the program to the address space of the thread and executes the user program.

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

1-4. threads/scheduler.cc

Scheduler::ReadyToRun() sets a thread to ready and puts it to the ready queue. This assumes that interrupts are turned off.

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

Scheduler::Run() dispatches the CPU to the next ready thread for execution. It assumes that interrupts are turned off. If the current thread has finished, mark it to indicate it should be destroyed. If the current thread is a user program, save its states for future restoration. Then, it performs the context switch (SWITCH (in switch.s), machine dependent) and the next thread becomes running thread. After returning from the context switch, the old thread can be cleaned up or being restored for further execution.

```
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
    SWITCH(oldThread, nextThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    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();
    }
}
```

Questions :

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

In Kernel::Exec(), it first creates a new thread, and calls new AddrSpace() to allocate the memory space for a new thread.

```
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;
```

It then calls Fork to execute the thread. In Thread::Fork(), it will call the StackAllocate(func, arg). The function will initialize the execution stack for the thread. It will then disable interrupt, to put the thread into ready queue, for further CPU execution. Lastly, enable the interrupt.

```
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);
}
```

```
// thread/thread.cc
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    ...
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
    ...
}
```

“func” is the function to be execute.

Finally, in kernel.cc, the kernel will call ForkExecute() to execute the thread, by loading the object file into memory. It will also establish the pageTable for that thread.

```
void ForkExecute(Thread *t)
{
    /* allocate pageTable for this process */
    if ( !t->space->Load(t->getName()) ) {
      return; /* executable not found */
    }
    t->space->Execute(t->getName());
}
```

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

In AddrSpace::AddrSpace(), the page table of virtual memory corresponding to physical memory will be set at the beginning, and the entire address space will be zeroed out to initialize the memory content.

Then, in AddrSpace::Load(char *fileName), the user program will be loaded from file into memory, including

1. Initializing address space
2. Initializing code segment
3. Initializing data segment
4. Initializing read only data segment

fileName is a file containing object code, which should be loaded into memory.

How does Nachos create and manage the page table?

It uses the TranslationEntry class defined in translate.h to create and manage page table.

Then TranslationEntry *pageTable will be defined in addrspace.h.

In AddrSpace::AddrSpace(). The function will create address space for the user program. At the beginning, the page table of the virtual memory corresponding to the physical memory will be set, and the entire address space will be zeroed out to initialize the memory content.

```
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);
}
```

pageTable[i].virtualPage and pageTable[i].physicalPage will be the same i

pageTable[i].valid = TRUE; means this entry is not empty

pageTable[i].use = FALSE; means it has not been referenced yet

pageTable[i].dirty = FALSE; means it has not been written yet

pageTable[i].readOnly = FALSE; means this entry is not read only

Note : the implement here is still uniprogramming, we modify a multiprogramming version in our implement.

In AddrSpace::RestoreState().

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

If you want to restore the state of the previous thread, you will restore its pageTable.

How does Nachos translate addresses?

In AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite), the function will convert virtual address to physical address

```
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;

    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;

    ASSERT((*paddr < MemorySize));

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

    return NoException;
}
```

1. Calculate vpn (virtual page number), vpn = vaddr / PageSize;
2. Calculate offset, offset = vaddr % PageSize;
3. Assert that the VPN does not exceed numPages
4. View translation entry pte from pageTable[vpn]
5. Check read and write permissions
6. Physical frame number taken from pte
7. Check whether the page frame is too large
8. Set the use bit of pte to TRUE
9. If access is read-write, the dirty bit is set to TRUE

10. Finally, calculate *paddr = pfn*PageSize + offset;

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

There is a private variable in the thread class called

```
int userRegisters[NumTotalRegs];  // user-level CPU register state
```

This is used to store the registers state of the thread.

When initially creating a thread, status, machineState will be initialized first.

```
//----------------------------------------------------------------------
// Thread::Thread
//  Initialize a thread control block, so that we can then call
//  Thread::Fork.
//
//  "threadName" is an arbitrary string, useful for debugging.
//----------------------------------------------------------------------

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;   // not strictly necessary, since
          // new thread ignores contents
          // of machine registers
    }
    space = NULL;
}
```

StackAllocate will also be called to do some mahineStates settings when forking.

```
// thread/thread.cc
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    ...
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
    ...
}
```

Which object in Nachos acts the role of process control block

thread

In thread.h

```
// thread.h
//  Data structures for managing threads.  A thread represents
//  sequential execution of code within a program.
//  So the state of a thread includes the program counter,
//  the processor registers, and the execution stack.
```

PCB (process control block) contains some information related to process / thread, such as:

- process state
- program counter
- CPU register
- CPU scheduling info (e.g. priority)
- memory management info (e.g. base / limit register)
- and more

and class Thread has a lot of information that PCB should contain, such as:

- process state --> status
- memory management --> space
- CPU register --> machineState
- and more

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

When calling Thread::Fork(VoidFunctionPtr func, void *arg), it will call

```
scheduler->ReadyToRun(this);
```

This line will put the Thread that has allocated resources into the Ready Queue for future CPU scheduling execution.

It is called by the following path:

1. Kernel::Kernel(int argc, char **argv)
2. void Kernel::ExecAll()

3. int Kernel::Exec(char* name)
4. Thread::Fork()

Please look at the following code from urserprog/exception.cc and answer the question:

```
case SC_MSG:
    DEBUG(dbgSys, "Message received.\n");
    val = kernel->machine->ReadRegister(4);
    {
    char *msg = &(kernel->machine->mainMemory[val]);
    cout << msg << endl;
    }
    SysHalt();
    ASSERTNOTREACHED();
break;
```

In virtual memory, each virtual page is contiguous. However, the corresponding physical pages may not be contiguous.

In the code above, the message is assumed to be in a contiguous physical memory space, but in reality, this may not be the case. Therefore, if the message size is larger than a page, msg may read something that is in the physical memory space of another thread and an error may occur.

Part 2. Implementation

Implement

Originally, the AddrSpace() constructor is designed for uni-programming, so the virtual memory space is a one-to-one mapping to the entire physical memory space; that is, one single program takes up the entire physical memory space.

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = -1;  // indicate no physical page mapped
        pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
    // bzero(kernel->machine->mainMemory, MemorySize);
}
```

In our design, we initialize all entries of the page table by setting all the values to false. Note that valid is set to false and physical page is set to -1, because the mapping has not been done yet.

We introduced a data structure in kernel.h. PhysPagesQueue is a queue that contains the available physical page indices.

```
queue<int> PhysPagesQueue;
```

They are initialized when the kernel constructor is called.

```
Kernel::Kernel(int argc, char **argv)
{
    PhysPagesInUse[NumPhysPages] = {false};
    for (int i = 0; i < NumPhysPages; i++)
        PhysPagesQueue.push(i);
}
```

The ~AddrSpace() destructor releases the physical pages by pushing the physical page indices back to the PhysPagesQueue and setting the PhysPagesInUse of the pages to false. It then deletes the pageTable.

```
AddrSpace::~AddrSpace()
{
    // 要清掉使用使用中的physical page
    // numPages : virtual memory page有幾個
    // pageTable[i].physicalPage : virtual page對應到哪個的physical page
    for(int i=0; i<numPages; i++){
        kernel->PhysPagesQueue.push(pageTable[i].physicalPage);
    }
    delete pageTable;
}
```

The main implementations are in AddrSpace::Load(char *fileName).

```
// 改成在這裡面才建立pageTable
// 先知道獨進來的file有多大再決定要給他多少physical pages
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

    for (int i = 0; i < numPages; i++) {
        if (kernel->PhysPagesQueue.empty())
            ExceptionHandler(MemoryLimitException);
        int k = kernel->PhysPagesQueue.front();
        kernel->PhysPagesQueue.pop();
        // cout << "use " << k << "th physical pages" << endl;
        kernel->PhysPagesInUse[k] = true;                       // 把這個physical page設定為使用中
        pageTable[i].physicalPage = k;
        pageTable[i].valid = TRUE;                      // 剩下的設置照舊

        // 清空這塊page table的空間，要清空的physical page位置如下，一次清空一個PageSize的大小，相當於一次清空一個page
        bzero(kernel->machine->mainMemory + pageTable[i].physicalPage * PageSize, PageSize);
    }

    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);

        // 改成用對應的physical address
        unsigned int virtualAddr = noffH.code.virtualAddr;
        unsigned int physicalAddr;
        Translate(virtualAddr, &physicalAddr, 1);

        executable->ReadAt(
    &(kernel->machine->mainMemory[physicalAddr]),
      noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
      DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);

        // 改成用對應的physical address
        unsigned int virtualAddr = noffH.initData.virtualAddr;
        unsigned int physicalAddr;
        Translate(virtualAddr, &physicalAddr, 1);

        executable->ReadAt(
```

```
    &(kernel->machine->mainMemory[physicalAddr]),
      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);

        // 改成用對應的physical address
        unsigned int virtualAddr = noffH.readonlyData.virtualAddr;
        unsigned int physicalAddr;
        Translate(virtualAddr, &physicalAddr, 1);

        executable->ReadAt(
    &(kernel->machine->mainMemory[physicalAddr]),
      noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

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

First, we need to know the size of the code, init data and readonly data, and then we can decide how many pages are needed to store them.

In this part, we perform the virtual to physical memory mapping and update the page table. Since the available physical pages are stored in a queue called PhysPagesQueue, we use it to find the next available physical page for our virtual page. If the queue is already empty, then the memory limit exception occurs. Otherwise, the physical page at the front is mapped to the current virtual page.

```
for (int i = 0; i < numPages; i++) {
    if (kernel->PhysPagesQueue.empty())
        ExceptionHandler(MemoryLimitException);
    int k = kernel->PhysPagesQueue.front();
    kernel->PhysPagesQueue.pop();
    // cout << "use " << k << "th physical pages" << endl;
    kernel->PhysPagesInUse[k] = true;                       // 把這個physical page設定為使用中
    pageTable[i].physicalPage = k;
    pageTable[i].valid = TRUE;                      // 剩下的設置照舊

    // 清空這塊page table的空間，要清空的physical page位置如下，一次清空一個PageSize的大小，相當於一次清空一個page
    bzero(kernel->machine->mainMemory + pageTable[i].physicalPage * PageSize, PageSize);
}
```

The following is the code, init data, read only data segment part.

With the pageTable, a virtual page can be mapped to a physical page.

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

```
    // 改成用對應的physical address
    unsigned int virtualAddr = noffH.code.virtualAddr;
    unsigned int physicalAddr;
    Translate(virtualAddr, &physicalAddr, 1);
    cout << "In code segment : " << endl;
    cout << "physicalAddr = " << physicalAddr << endl;

    executable->ReadAt(
&(kernel->machine->mainMemory[physicalAddr]),
  noffH.code.size, noffH.code.inFileAddr);
}
```

Translate() converts noffH.code.virtualAddr into the according physicalAddr, and then the code segment is loaded into mainMemory[physicalAddr].

init data and read only data follow the same procedure.

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

    // 改成用對應的physical address
    unsigned int virtualAddr = noffH.initData.virtualAddr;
    unsigned int physicalAddr;
    Translate(virtualAddr, &physicalAddr, 1);
    cout << "In data segment : " << endl;
    cout << "physicalAddr = " << physicalAddr << endl;

    executable->ReadAt(
&(kernel->machine->mainMemory[physicalAddr]),
  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);

        // 改成用對應的physical address
        unsigned int virtualAddr = noffH.readonlyData.virtualAddr;
        unsigned int physicalAddr;
        Translate(virtualAddr, &physicalAddr, 1);
        cout << "In read only data segment : " << endl;
        cout << "physicalAddr = " << physicalAddr << endl;

        executable->ReadAt(
    &(kernel->machine->mainMemory[physicalAddr]),
      noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif
```

Before the thread is deleted, we added a line to release the address space.

```
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        // releasing address space
        delete toBeDestroyed->space;
        delete toBeDestroyed;
        toBeDestroyed = NULL;
    }
}
```