Operating System

Homework2

Cheng-Ying Tsai

1. Explain how a thread goes through in NachOS and other detail.

threads/thread.h

以下為 thread 的 data structure

```
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.
};
```

threads/kernel.h

以下為 Kernel 的 data structure

```
class Kernel {
  public:

   void ExecAll();

   int Exec(char* name);        // create a thread and allocate the resource

   Thread* getThread(int threadID){ return t[threadID];}

  private:
    Thread* t[10];
    char* execfile[10];
    int execfileNum;
    int threadNum;
```

若我們想要執行一個程式：

(1) 創造一個 thread

(2) 賦予一個空間

(3) 透過 Fork 載入要執行的程式碼

(4) thread 數量-1

```
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){
    if (!t->space->Load(t->getName())){return;}
    t->space->Execute(            ));
}
```

可以發現呼叫 addrspace.cc 裡的 load 函式載入 memory 中，最後呼叫 Execute()

一個 thread 要執行需要下面三個步驟

userprog/AddrSpace.h

```
void
AddrSpace::Execute()
{

    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"
}
```

不考慮 Content switch，當一個 Thread 要執行時的三個步驟：

(1) InitReigisters()　// 初始化 user registers

(2) RestoreState()　// 載入這個程式對應的 page table

(3) kernel->machine->Run()

```
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);
}
```

從 Fork()可知，StackAllocate()，初始化各種不同 type 的 Kernel Registers (machine state)，初始化 stack，並將 thread 放入 ready queue

```
#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
```

透過 machineState[InitialPCState] = (void*) func，讓原先的 function pointer 成為未來 Program counter 要執行的程式

```
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;
}
```

threads/kernel.cc

```
void Kernel::ExecAll(){
    for (int i = 0; i < execfileNum; i++){
        int a = Exec(execfile[i]);
    }
    currentThread->Finish();
}
```

Main thread 也就是 kernel 去執行所有要執行的程式

另外 Finish()的部分，釋放 Thread 空間步驟如下

```
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name);

    Sleep(TRUE);                                    // invokes SWITCH
    // not reached
}

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);
}
```

在 sleep()中，確認 interrupt 是否為 off，若是則將 thread block，並判斷是否有下一個 thread 要 run，若無進入 idle()，若有 De Allocate 上個 thread

machine/mipssim.cc

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
          Debugger();
    }
}
```

模擬 thread 執行的解碼過程，Onetick()模擬 CPU Clock 往前跑的情形。

大致過程整理如下： (1) New a thread and AddrSpace (2) Fork()讓 thread 載入欲執行的程式碼 (3) Fork 先 StackAllocate()做 stack 初始化，讓原先的 function pointer 成為未來 program counter 要執行的程式 (4) set intoff 也就是不准 interrupt (5) scheduler->ReadyToRun(this)，放入 Ready Queue 準備讓 CPU 執行 (6) CPU scheduler 未來會從 Ready Queue 中 load 要執行的 thread，並讀取 program counter

2. Implement page table in NachOS

尚未 implement multiprogramming 的結果如下

```
chenging@chenging-VirtualBox:~/nachos/nachos-4.0/code/test$ ../userprog/nachos -
e test1 -e test2
Total threads number is 2
Thread test1 is executing.
Thread test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
Print integer:7
Print integer:8
Print integer:9
```

原因為，當兩個程式匯入時，操作到相同區域的 code segment。

每一個 process 都有 Addrspace，也就是 paging 的概念，

pageTable 對應到 pageTable[i].physicalPage

以下 Addrspace 的建構子

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
        pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
        pageTable[i].physicalPage = i;
//      pageTable[i].physicalPage = 0;
        pageTable[i].valid = TRUE;
//      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);
}
```

框起來的地方表示紀錄 logical address 對應的 physicalPage，這樣才會 mapping 到不同區域。

以下 Addrspace 的除構子

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

   delete pageTable;
}
```

Process 執行成功後，除構，釋放資源，將標記使用的部分設定為 0

根據我們 trace code 的結果，可以看出要在 addrspace::load load

我們要執行的 file，

```
bool
AddrSpace::Load(char *fileName)

 for (unsigned int i = 0, j = 0; i < numPages; i++){
     pageTable[i].virtualPage = i;
     while (j < NumPhysPages && AddrSpace::usedPhyPage[j] == TRUE)
         j++;
     AddrSpace::usedPhyPage[j] = TRUE;
     pageTable[i].physicalPage = j;
     pageTable[i].valid = TRUE;
     pageTable[i].use = FALSE;
     pageTable[i].dirty = FALSE;
     pageTable[i].readOnly = FALSE;
 }
```

利用 usedPhyPage 來記錄是否被使用，若沒有就可以放入 table 中。

```
// then, copy in the code and data segments into memory
        if (noffH.code.size > 0) {
                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) {
        executable->ReadAt(
                &(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/
PageSize].physicalPage*PageSize + (noffH.code.virtualAddr%PageSize)]),
                        noffH.initData.size, noffH.initData.inFileAddr);
    }
```

有了區域，決定程式的進入點，也就是 main memory address。

首先，virtualAddr 除以 PageSize 得到是第幾個 page，利用 pageTable

查出是實體第幾頁，在乘上實體記憶體，接著找出這一頁的甚麼位置，也就是 offset 的部分，virtualAddr mod PageSize。

以下為 multiprogramming 結果

```
chenging@chenging-VirtualBox:~/nachos/nachos-4.0/code$ ./userprog/nachos -e test
/test1 -e test/test2
Total threads number is 2
Thread test/test1 is executing.
Thread test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
```