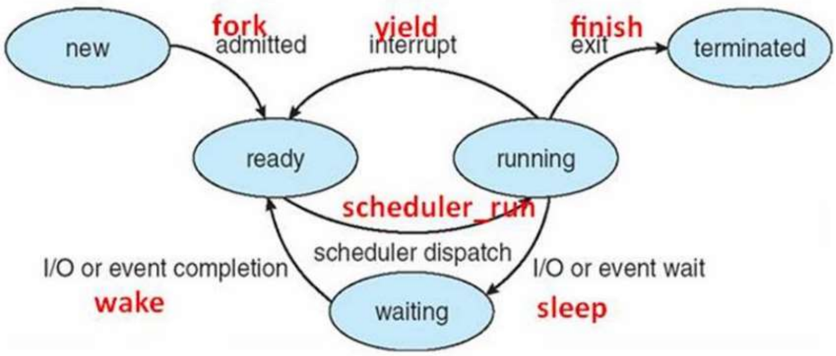
Operating System

Homework3

Cheng-Ying Tsai

1. Trace code：Explain the purposes and details of the following 6 code paths



1-0. Preface

thread/main.cc 裡的 main() function，Bootstrap OS kernel，接受參數，由 strcmp 判斷，初始化後，最後 kernel->Run()

thread/main.cc

```
int
main(int argc, char **argv)
{
    int i;
    char *debugArg = "";

    // before anything else, initialize the debugging system
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-d") == 0) {
            ASSERT(i + 1 < argc);   // next argument is debug string
            debugArg = argv[i + 1];
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            cout << "Partial usage: nachos [-z -d debugFlags]\n";
        } else if (strcmp(argv[i], "-z") == 0) {
            cout << copyright;
        }

    }
    debug = new Debug(debugArg);

    DEBUG(dbgThread, "Entering main");

    kernel = new KernelType(argc, argv);
    kernel->Initialize();

    CallOnUserAbort(Cleanup);           // if user hits ctl-C

    kernel->SelfTest();
    kernel->Run();

    return 0;
```

1-1. New→Ready

1-1. New→Ready

```
Kernel::ExecAll()
        ↓
Kernel::Exec(char*)
        ↓
Thread::Fork(VoidFunctionPtr, void*)
        ↓
Thread::StackAllocate(VoidFunctionPtr, void*)
        ↓
Scheduler::ReadyToRun(Thread*)
```

接下來 trace code 的部分，因為我的 NachOS 檔案與作業不太相同，ExecAll()與 Exec(char*)的部分是寫在一起的，是寫在 UserProgKernel::Run()裡。

UserProgKernel::Run()

```
void
UserProgKernel::Run()
{

        cout << "Total threads number is " << execfileNum << endl;
        for (int n=1;n<=execfileNum;n++)
                {
                t[n] = new Thread(execfile[n]);
                t[n]->space = new AddrSpace();
                t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
                cout << "Thread " << execfile[n] << " is executing." << endl;
                }

    ThreadedKernel::Run();
}
```

透過 for loop 執行每個檔案，完成後 call Finish()結束 NachOS。

(1) 先 new thread

Thread::Thread(char* threadName)

```
Thread::Thread(char* threadName)
{
    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
    }
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
```

(2) 再分配一個 Address Space 給 thread，並配置 PageTable

AddrSpace::AddrSpace()

```
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);
}
```

以上還沒有將 Program Load 到 Memory，故由 t[n]->Fork 來做到，另外，ForkExecute 這個 function pointer 是 Thread::Begin()之後要馬上執行的程式。

ForkExecute(Thread* t)

```
void
ForkExecute(Thread *t)
{
        t->space->Execute(t->getName());
}
```

AddrSpace::Execute(char* filename)

```
void
AddrSpace::Execute(char *fileName)
{
    if (!Load(fileName)) {
        cout << "inside !Load(FileName)" << endl;
        return;                         // executable not found
    }

    //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"
}
```

補充：

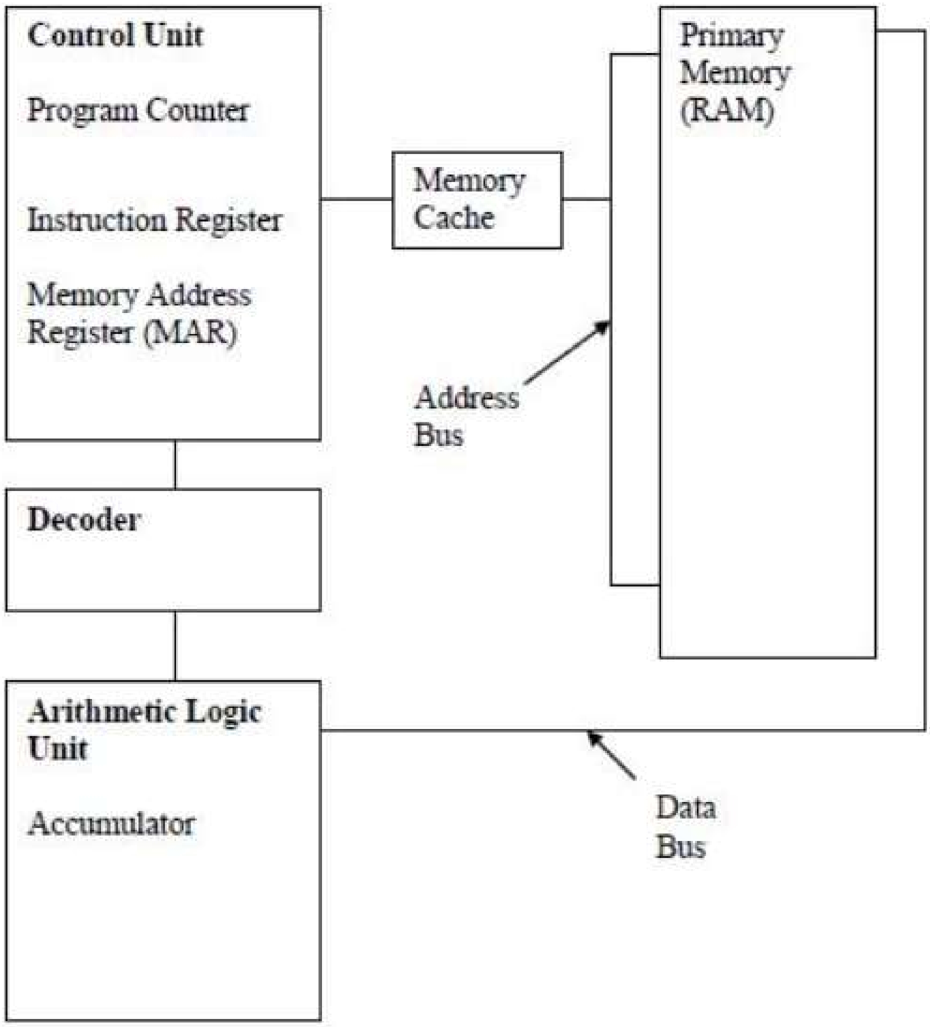
由 switch.s 完成：

(1) **thread::Begin()**

```
void
Thread::Begin ()
{
    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Beginning thread: " << name);

    kernel->scheduler->CheckToBeDestroyed();
    kernel->interrupt->Enable();
}
```

(2) ForkExecute()，將 Program Load 到 memory，接著就是框起來的那三個步驟，也就是說在新的 thread 拿到控制權後，透過 Machine 會抓取 Program Counter 存放指令的 Address 來做 decode，接著 ForkExecute 透過 AddrSpace::Execute(char* filename) 讀取 Program，並設定初始暫存器的值，然後 load Page Table，Machine->Run()就是靠 OneInstruction(instr)、OneTick()不斷 Loop 讀取指令，以下為 CPU 的流程圖。


Control Unit
Program Counter
Instruction Register
Memory Address
Register (MAR)
Memory
Cache
Primary
Memory
(RAM)
Address
Bus
Decoder
Arithmetic Logic
Unit
Accumulator
Data
Bus

Thread::Fork(VoidFuntionPtr func, void* 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);
}
```

Fork()過程：

(1) 宣告兩個指標，interrupt 與 scheduler

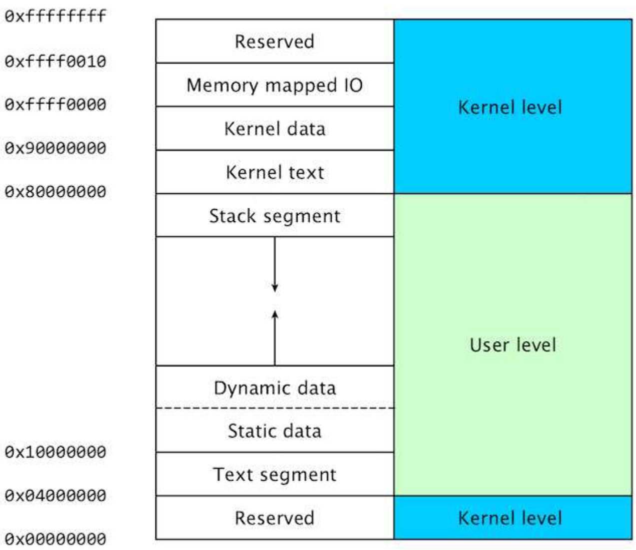
(2) 讓 InStatus Oldlevel 表示舊的 thread

(3) 由 StackAllocate()配置 stack 與 machinstate

(4) 結束 StackAllocate()，disable interrupt，並把 thread 丟入到 ready Queue，scheduler->ReadyToRun(this)

補充：

堆疊為高位址堆到低位址



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

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

    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
machineState[PCState] =(void *)ThreadRoot;
machineState[StartupPCState] = (void *)ThreadBegin;
machineState[InitialPCState] = (void *)func;
machineState[InitialArgState] = (void *)arg;
machineState[WhenDonePCState] = (void *)ThreadFinish;
```

StackAllocate(func, arg)過程：

(1) 分配一個 array，此時 stack 就是指向頂部 low address

(2) 讓 stackTop 指向底部 high address，-4 可以在 safe side

(3) 讓 stack 的第一個元素為 ThreadRoot

(4) *stack = STACK_FENCEPOS 表示再往上一個，防止存取越界

(5) 設定 machineState，ThreadRoot 將使用 func(arg)，ThreadBegin

Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY);
    readyList->Append(thread);
}
```

將 thread 設定成 READY，Append 進 readyList

1-2. Running->Ready

1-2. Running→Ready

```
Machine::Run()
      ↓
Interrupt::OneTick()
      ↓
Thread::Yield()
      ↓
Scheduler::FindNextToRun()
      ↓
Scheduler::ReadyToRun(Thread*)
      ↓
Scheduler::Run(Thread*, bool)
```

表示有 interrupt 發生，與 CPU 排程有關。

NachOS 透過 Machine::Run()與 Interrupt::OneTick()模擬 user program 於 MIPS 的每一個 clock 執行的過程，然而 interrupt 的發生需要依靠 Thread::Yield()來達成。

Machine::Run()

```
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();
    }
}
```

(1) 設定成 UserMode，用到 system call 時就會切換

(2) 無窮迴圈，不斷一行一行讀，OneTick 模擬每個 clock 進行

(3) 若在 OneInstruction 時，有 system call 時，就 RaiseException，切換成 System Mode，呼叫 ExceptionHandler，作業一

```
case OP_SYSCALL:
  RaiseException(SyscallException, 0);
  return;
  break;
```

```
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);

    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);                  // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
//      cout << "entering system mode...\n";
    ExceptionHandler(which);            // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
//      cout << "entering user mode...\n";
}
```

但這裡發現 NachOS 沒有實作 ThreadYield 的 system call，Running->Ready 不是因為這個原因，故以下為 NachOS 真正處理 interrupt 的過程。

Interrupt::OneTick()

```
void
Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

// advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {                                    // USER_PROGRAM
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
                                // (interrupt handlers run with
                                // interrupts disabled)
    CheckIfDue(FALSE);          // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts
    if (yieldOnReturn) {        // if the timer device handler asked
                                // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;            // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

(1) 判斷是哪個 mode，並增加相對應的 Ticks

(2) turn off interrupt，確保下個指令執行是 Atomic

(3) CheckIfDue 檢查是否有下一個到期的 pending Interrupt，並執行，其中 next 的 callback()會讓 interrupt 的 yieldOnReturn 設成 True

(4) 當 yieldOnReturn 為真，則進行 context switch，先將 yieldOnReturn 設為 FALSE，改成 SystemMode，並釋放 kernel 目前的 thread，切換成原來的 mode，通常為 user mode

Interrupt::CheckIfDue()

```
bool
Interrupt::CheckIfDue(bool advanceClock)
{
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;

   inHandler = TRUE;
   do {
       next = pending->RemoveFront();    // pull interrupt off list
       next->callOnInterrupt->CallBack();// call the interrupt handler
       delete next;
   } while (!pending->IsEmpty()
               && (pending->Front()->when <= stats->totalTicks));
   inHandler = FALSE;
   return TRUE;
```

Alarm::CallBack()

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    bool woken = _bedroom.MorningCall();

    kernel->currentThread->setPriority(kernel->currentThread->getPriority() - 1);

    if (status == IdleMode && !woken && _bedroom.IsEmpty()) {   // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
        }
    } else {                    // there's someone to preempt
        if (kernel->scheduler->getSchedulerType() == RR ||
            kernel->scheduler->getSchedulerType() == Priority){
        cout << "======== interrupt->YieldOnReturn() ========" << endl;
        interrupt->YieldOnReturn();
        }
    }
}
```

這裡小小整理一下，也就是說 Hardware Timer 會定期發一個 Interrupt 來呼叫 YieldOnReturn()，當 OneTick()看到 yieldOnReturn 被設成 True，那麼就會執行 Yield()，以下來談談 Yield()。

Thread::Yield()

```
void
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

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

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

(1) 拒絕 interrupt

(2) 由 scheduler 找出下一個 nextThread

(3) 不為空，將目前執行的 Thread 放入 ready queue

(4) 執行 Run(nextThread)，context switch 就達成

(5) 恢復可以 interrupt 的狀態

Scheduler::FindNextToRun()

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

若 readyList 不為空，Deque()

Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY);
    readyList->Append(thread);
}
```

將 thread 設定成 READY，Append 進 readyList

Scheduler:: Run()

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    if (finishing) {    // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

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

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

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

 SWITCH(oldThread, nextThread);
```

switch.s 裡為實作

```
 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();
 }
```

Context switch 在此進行

(1) 如果 finishing 為 true，表示上一個 thread 執行完畢，此時讓

toBeDestroyed 指向上一個 thread

(2) 保存上一個 thread 的 user state，就是 user program 對應到的 register set，存入 thread class 的 userRegisters[NumsTotalRegisters]，接著保存 AddrSpace 的 states

Thread::SaveUserState()

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

AddrSpace::SaveState()

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

Thread::CheckOverflow()

```
void
Thread::CheckOverflow()
{
    if (stack != NULL) {
#ifdef HPUX                     // Stacks grow upward on the Snakes
        ASSERT(stack[StackSize - 1] == STACK_FENCEPOST);
#else
        ASSERT(*stack == STACK_FENCEPOST);
#endif
    }
}
```

(3) 將現在的 thread 改為要執行的 nextThread，並呼叫 SWITCH，switch.s 實作，而定義的巨集與參數在 thread.h、switch.h

(4) nextThread 執行完後，由 CheckToBeDestroyed()來 delete，注意記得要指向空指標，防止 Dangling Pointer 的問題

Scheduler::CheckToBeDestroyed()

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

(5) 恢復 oldThread 相關的 States

1-3. Running→Waiting

1-3. Running→Waiting (Note: only need to consider console output as an example)

```
SynchConsoleOutput::PutChar(char)
↓
Semaphore::P()
↓
SynchList<T>::Append(T)
↓
Thread::Sleep(bool)
↓
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
```

通常是因為 I/O 或 event 的 interrupt 造成，由 Sleep()來 Block running thread。

Kernel->Initialize()包含以下兩行程式碼，stdin 與 stdout 為 arg

```
synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
```

這裡我們先來看一下 SynchConsoleOutput 的建構子

SynchConsoleOutput::SynchConsoleOutput(char* outputFile)

```
SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
    lock = new Lock("console out");
    waitFor = new Semaphore("console out", 0);
}
```

ConsoleOutput:: ConsoleOutput(char* writeFile, …)

```
ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
{
    if (writeFile == NULL)
        writeFileNo = 1;                        // display = stdout
    else
        writeFileNo = OpenForWrite(writeFile);

    callWhenDone = toCall;
    putBusy = FALSE;
}
```

可以看到 lock 與 waitFor 的初始值設定，另外，注意 SynchConsoleOutput 與 ConsoleOutput 皆公有繼承自 callback，然而 callback 這個類別我們可以看作成一個介面，故此時 toCall 這個 argument，就是 SynchConsoleOutput，這個類。

SynchConsoleOutput::PutChar(char ch)

```
void
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

這裡我們先來看一下 lock 的建構子

Lock::Lock(char* debugNmae)

```
Lock::Lock(char* debugName)
{
    name = debugName;
    semaphore = new Semaphore("lock", 1);  // initially, unlocked
    lockHolder = NULL;
}
```

可以發現底層是由 semaphore 建立起來的

Semaphore::Semaphore(char* debugName, int initialValue)

```
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List<Thread *>;
}
```

(1) Lock->Acquire()，hold 現在的 thread

Lock::Acquire()

```
void Lock::Acquire()
{
    semaphore->P();
    lockHolder = kernel->currentThread;
}
```

(2)

ConsoleOutput::PutChar(char ch)

```
void
ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}
```

writeFileNo 在建構子時設成了 1，WriteFile 會將 1 個 char 寫上 stdout，putBusy 為 True，正在輸出，會擋掉其他的 thread。

Semaphore::P()

```
void
Semaphore::P()
{
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {                 // semaphore not available
        queue->Append(currentThread);    // so go to sleep
        currentThread->Sleep(FALSE);
    }
    value--;                        // semaphore available, consume its value

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

(1) Disable interrupt

(2) 檢查 value 是否等於 0，成立表示沒有資源，將等待的 current thread append 進 queue，並呼叫 Sleep(FALSE)

SynchList<T>::Append(T)

這其實就只是 single linked list 的 C++樣板函式而已。

Thread::Sleep(bool finishing)

```
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);
}
```

(1) 將狀態設成 BLOCKED

(2) 從 ready queue 尋找下一個要執行的 thread，有的話就 Run()，沒有的話 Idle()

Scheduler::FindNextToRun()

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

若 readyList 不為空，Deque()，return 要執行的 thread 指標

Scheduler:: Run()

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    if (finishing) {    // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
    }

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

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

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

 SWITCH(oldThread, nextThread);

 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();
 }
```

這裡收到的 finishing 為 False，表示還沒完成，因為被 BLOCK 住了，倘若為 True，表示執行完畢，讓 toBeDestroyed 指向上一個 Thread，也就是 old Thread，接下來的過程上一節有介紹過。

1-4. Waiting→Ready

1-4. Waiting→Ready (Note: only need to consider console output as an example)

```
Semaphore::V()
      ↓
Scheduler::ReadyToRun(Thread*)
```

PutChar 會將 ConsoleOutput 本身餵入 interrupt pending list，之後執行 Callback()，根據 trace code 發現 Callback()就是 waitFor->V()

```
void
ConsoleOutput::CallBack()
{
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}

void
SynchConsoleOutput::CallBack()
{
    waitFor->V();
}
```

也就是說當 Console Time (1 tick)過去，interrupt 發生，waitFor->V()

Semaphore::V()

```
void
Semaphore::V()
{
    Interrupt *interrupt = kernel->interrupt;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

(1) 檢查 Semaphore 的 List<Thread* > *queue 是否為空，並 deque，

找出準備從 BLOCKED 變成 READY 的 thread

(2) 找出後，將 semaphore value++，也就是釋放資源讓 p()可以用

(3) re-enable interrupts

Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY);
    readyList->Append(thread);
}
```

1-5. Running→Terminated

1-5. Running→Terminated (Note: start from the Exit system call is called)

```
ExceptionHandler(ExceptionType) case SC_Exit
↓
Thread::Finish()
↓
Thread::Sleep(bool)
↓
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
```

表示執行完所有該執行的 instruction，資源釋放，因為 thread 不能 delete 自己，因為自己正在 run，故需要下一個 thread 來 delete，此時就會呼叫 Finish()，Finish()會呼叫 Sleep(True)，current thread 會 BLOCKED，然後藉由 1-3 介紹的 toBeDestroyed()來 delete，若沒有下一個 thread，直接 Halt()結束 NachOS。

ExceptionHandler(ExceptionType) case SC_Exit

```
case SC_Exit:
        DEBUG(dbgAddr, "Program exit\n");
        val=kernel->machine->ReadRegister(4);
        cout << "return value:" << val << endl;
        kernel->currentThread->Finish();
        break;
```

可能為執行到 return 0，或是說一個 instruction decode 後，有呼叫此 system call (Exit(0))，經由 start.s 與 syscall.h 將控制權交給 ExceptionHandler，以下為此 system call。

syscall.h

```
/* This user program is done (status = 0 means exited normally). */
void Exit(int status);
```

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
}
```

Thread::Sleep(bool finishing)

```
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);
}
```

將 current thread 變成 BLOCKED，由 scheduler 找出 ready queue 下一個 pop 的 thread，若無，Idle()這個函式會自己判斷是否該 Advance Clock 或是 Halt()，若有呼叫 scheduler->Run 下一個 thread。

Scheduler::FindNextToRun()

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

Pop 下一個 thread。

Scheduler:: Run()

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    if (finishing) {    // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

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

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

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

SWITCH(oldThread, nextThread);

 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();
 }
```

(1) Context switch，這裡只要注意 finishing 為 True，表示上一個 thread 完成，toBedestroyed = oldthread

(2) 保存 oldthread 的狀態，SaveUserState()，與保存 Address Space，SaveState()

(3) 將現在要執行的 currentThread 指向 nextThread，設置狀態為 RUNNING，呼叫 SWITCH()，交換 thread

(4) 呼叫 CheckToBeDestroyed()，檢查是否有 thread 要被 delete，因為前面說 finishing 為 True 的話，會 delete 掉。

Scheduler::CheckToBeDestroyed()

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

(5) delete 掉的話 if 不成立，不需復原。