312505017 馮凡哲

312511057 林胤汶

Part.I Trace Code

1. New -> Ready

- **Kernel::ExecAll:**

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

Kernel::ExecAll 主要是使用 for 迴圈將 execfile 陣列的每一項傳入 Exec()執行,最後 currentThread 會呼叫 finish 來結束 NachOS。

- **Kernel::Exec:**

```
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;
```

Exec()創建一個新的 Thread 類別給要執行的 thread，並為其分配一個空間 (AddrSpace)，再使用 Fork 函式，將 ForkExecute 函式做為參數傳入，以實現新 Thread 的執行。
而 ForkExecute 函式中主要進行的就是使用 Load 函式，將要執行的程式載入 Memory 裡。

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

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

}
```

- **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);
}
```

首先宣告了兩個指標,型態分別是 kernal->interrupt 和 kernel->scheduler,再呼叫 StackAllocate(),傳入 ForkExecute 函式和 t[threadNum],最後將 interrupt 停用,最後呼叫 scheduler->ReadyToRun 將此 Thread 傳入 Ready List。

- **Thread::StackAllocate:**

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

首先配置 Stack 的空間,之後再針對不同的結構進行初始化。

```
#ifdef PARISC
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif
#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
    *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
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif
```

可以看到主要是進行確保 Stack 空間的操作。

```
#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
}
```

設置 machineState[]的部分,可以發現傳入的 ForkExecute 函式和 t[threadNum]被設置在 machineState[IntialPCState]和 machineState[InitialArgState]。

- **Scheduler::ReadyToRun:**

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

主要是將 thread 的狀態設為 ready,並將 thread 加入到 readyList。

2. Running -> Ready

- **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();
    }
}
```

主要就是使用 for 迴圈不斷呼叫 OneInstruction()來執行指令, 並用 OneTick()模擬 clock 的運作。

- **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 {
        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;
    }
}
```

首先判斷 status 是 UserMode 還是 SystemMode,以增加相對應的 Tick,再來先關閉 interrupt 以確保在處理目前的 interrupt 時不會被打斷,檢查完是否有待處理的 interrupt 後再將其重啟, 最後判斷 yieldOnReturn ,當 interrupt 處理程式返回時需要進行 context switch,則 yieldOnReturn 會設為 TRUE,若 yieldOnReturn 為 TRUE,則 status 會設成 SystemMode,接著執行 Yield()。

CheckIFDue 函式會先檢查是否有待處理的 interrupt,若無則 return FALSE,再來會檢查 interrupt 處理時間是否已到。若未到,因傳入函式的參數是 FALSE,所以不會做任何處理直接 return FALSE；若 interrupt 處理時間已到,則會進行一些相關的處理。

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

    ASSERT(level == IntOff);            // interrupts need to be disabled,
                                        // to invoke an interrupt handler
    if (debug->IsEnabled(dbgInt)) {
        DumpState();
    }
    if (pending->IsEmpty()) {           // no pending interrupts
        return FALSE;
    }
    next = pending->Front();

    if (next->when > stats->totalTicks) {
        if (!advanceClock) {            // not time yet
            return FALSE;
        }
        else {                  // advance the clock to next interrupt
            stats->idleTicks += (next->when - stats->totalTicks);
            stats->totalTicks = next->when;
            // UDelay(1000L); // rcgood - to stop nachos from spinning.
        }
    }

    DEBUG(dbgInt, "Invoking interrupt handler for the ");
    DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);

    if (kernel->machine != NULL) {
        kernel->machine->DelayedLoad(0, 0);
    }

    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;
}
```

- **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);
}
```

先關閉 interrupt,執行 FindNextTORun 取出下一個要執行的 Thread,再將目前執行的 Thread 放入 ReadyList,接著執行下一個 Thread,最後恢復 interrupt 的狀態。

- **Scheduler::FindNextToRun**

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

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

先檢查 readyList 是否為空,若否則回傳 Queue 中的下一個 Thread。

- **Scheduler::ReadyToRun**

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

將 thread 的狀態設為 ready,再把 thread 加入到 readyList。

- **Scheduler::Run**

```
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
```

先判斷 finishing,若為 FALSE,則表示上一個 Thread 還未完成,接著判斷前一個 Thread 的空間是否為空,若非空則呼叫 SaveUserState,再檢查員本的 Thread Stack 是否 overflow。

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

SaveUserState 將 ReadRegister 存入 userRegisters。

```
    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();
    }
}
```

再來將 currentThread 改成下一個 Thread,設定 state 並呼叫 SWITCH()進行 Thread 切換。
原本的 Thread 回來後,會呼叫 CheckToBeDestroyed()檢查是否有 Thread 要被清除,最後使用 RestoreUserState 恢復原本 Thread 的 State。

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

3. Running -> Waiting

- **SynchConsoleOutput::PutChar**

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

Synchconsole.cc 用來處理鍵盤和顯示器 I/O 的同步存取,PutChar()主要用來將字元寫入到顯示器,其中考慮同步問題,呼叫了 Acquire()來取得 Lock,Lock 主要是由 Semaphore 實現。

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

初始值設成 unlock。

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

Acquire 執行了 P(), 主要是在增加 value 的值 , 並將 lock 給目前的 Thread。

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

最後執行 Release(), 將 lock 釋出後執行 V(), V()主要是用來減少 value 的值。

```
void Lock::Release()
{
    ASSERT(IsHeldByCurrentThread());
    lockHolder = NULL;
    semaphore->V();
}
```

- **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);
}
```

首先將 interrupt 停用, 接著進入 while 迴圈判斷 value 是否為 0,若為 0 表示 semaphore 目前禁用,使用 Append 將目前 Thread 放入 queue,再用 Sleep 使目前 Thread 進入等待。

若 value 大於 0,則將 value 減 1,再恢復 interrupt 的狀態。

- **List::Append**

```
void List<T>::Append(T item)
{
    ListElement<T> *element = new ListElement<T>(item);

    ASSERT(!this->IsInList(item));
    if (IsEmpty())
    { // list is empty
        first = element;
        last = element;
    }
    else
    { // else put it after last
        last->next = element;
        last = element;
    }
    numInList++;
    ASSERT(this->IsInList(item));
}
```

功能類似 link list,將 item 加入到 List 的尾端。

- **Thread::Sleep**

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

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

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

    status = 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);
}
```

主要目的就是釋放 CPU,首先會將目前 Thread 的 status 設為 BLOCKED,接著進入 while 迴圈不斷尋找 readyList 中是否有下一個要執行的 Thread,若無則呼叫 idle(),若有則跳出迴圈執行下一個 Thread。

- **Scheduler::FindNextToRun**

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

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

主要就是檢查 ready 是否為空,若非空則回傳下一個 Thread。

- **Scheduler::Run**

```
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();
}
}
```

傳入的參數(finishing)仍然是 FALSE,表示原本的 Thread 並未完成,不需要被清除。

4. **Waiting -> Ready**

- **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);
}
```

主要是進行增加 value 的操作, 將 interrupt 停用, 如果有 Thread 在 ReadyQueue 中, 則將其設為準備狀態, 最後將 value 加 1,再恢復 interrupt 的狀態。

- **Scheduler::ReadyToRun**

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

將 thread 的狀態設為 ready,並將其加入到 readyList。

5. **Running -> Terminated**

- **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;
```

Thread 執行完後會經由此 systemcall，將 Thread 給結束。

- **Thread::Finish()**

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    DEBUG(dbgSche, "[B]Tick[" << kernel->stats->totalTicks << "]: Thread[" << ID << "] is removed from queue");

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

將 interrupt 停止後呼叫 Sleep()。

- **Thread::Sleep**

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

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

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

    status = 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);
}
```

與剛剛 Running -> Waiting 不同,這裡傳入 Sleep()的參數是 TRUE,表示 Thread 已經執行完。

- **Scheduler::FindNextToRun**

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

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

再經由 FindNextToRun 得到下一個要執行的 Thread。

- **Scheduler::Run**

```
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
```

傳入 Sleep 的參數最後會傳入 Run(),因為 finishing 為 TRUE,所以原本的 Thread 會被清除。

6. **Ready -> Running**

- **Scheduler::FindNextToRun**

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

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

若 readyList 非空,則回傳 List 中下一個 Thread。

- **Scheduler::Run**

```
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();
    }
}
```

保存完原本 Thread 的狀態後,執行 SWITCH。

- **SWITCH(Thread*, Thread*)**

SWITCH 由 Thread. h 宣告:

```
void ThreadRoot();

// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
}
```

在 switch.s 中實作:

```
SWITCH:
        sw      sp, SP(a0)              # save new stack pointer
        sw      s0, S0(a0)              # save all the callee-save registers
        sw      s1, S1(a0)
        sw      s2, S2(a0)
        sw      s3, S3(a0)
        sw      s4, S4(a0)
        sw      s5, S5(a0)
        sw      s6, S6(a0)
        sw      s7, S7(a0)
        sw      fp, FP(a0)              # save frame pointer
        sw      ra, PC(a0)              # save return address

        lw      sp, SP(a1)              # load the new stack pointer
        lw      s0, S0(a1)              # load the callee-save registers
        lw      s1, S1(a1)
        lw      s2, S2(a1)
        lw      s3, S3(a1)
        lw      s4, S4(a1)
        lw      s5, S5(a1)
        lw      s6, S6(a1)
        lw      s7, S7(a1)
        lw      fp, FP(a1)
        lw      ra, PC(a1)              # load the return address

        j       ra
        .end SWITCH
```

主要是作暫存器的 store 和 load, 暫存器編號定義在 switch.h。

```
/* Registers that must be saved during a context switch.  See comment above. */
#define    SP    0
#define    S0    4
#define    S1    8
#define    S2    12
#define    S3    16
#define    S4    20
#define    S5    24
#define    S6    28
#define    S7    32
#define    S8    36
#define    S9    40
#define    S10   44
#define    S11   48
#define    S12   52
#define    S13   56
#define    S14   60
#define    S15   64
#define    PC    68
```

- **depends on the previous process state :**

  若原本的 Thread 狀態是 BLOCKED,也就是在 waiting,則會執行 SWITCH 後續的指令。

```
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();
```

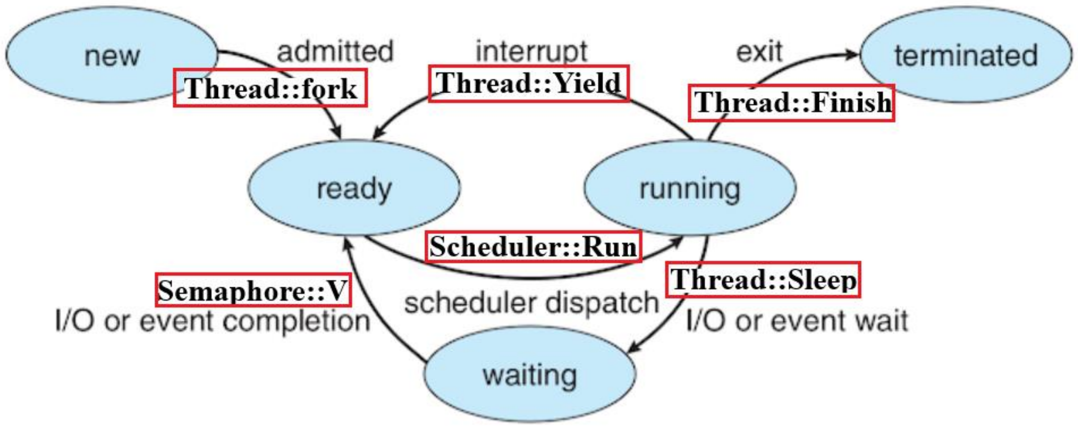
  若原本的狀態是 RUNNING,則可能因為 RR 排班,Thread 強制從 running 進到 ready, Thread 接收到一個 timer Interrupt，裡面的 CallBack 是 "YieldOnReturn",讓其執行 Yield。

```
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;
}
```

- **for loop in Machine::Run()**

```
for (;;) {
    OneInstruction(instr);
            kernel->interrupt->OneTick();
            if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
                    Debugger();
}
```

  這個迴圈不斷呼叫 OneInstruction()來執行指令, 並用 OneTick()模擬 clock 的運作, 而 OneTick()中則會用 CheckIfDue 定期檢查是否有 interrupt,這其中也包含 RR 排班的 timer Interrupt , 最後用 Yield 完成 context switch。



PartII. Implementation

In thread.h

Class Thread public:

```
char* name;
int   ID;
//int burstTime;
//int waitingTime;
int executionTime;
//int L3Time;
int priority;
int lastexecutionTime;
```

```
//void setBurstTime(int t) {burstTime = t;}
//void setWaitingTime(int t){waitingTime = t;}
void setExecutionTime(int t){executionTime = t;}
void setPriority(int p){priority = p;}
//void setL3Time(int t){L3Time = t;}
void setLastexecutionTime(int t){lastexecutionTime = t;}
//int getBurstTime(){return (burstTime);}
//int getWaitingTime(){return (waitingTime);}
int getExecutionTime(){return (executionTime);}
int getPriority(){return (priority);}
//int getL3Time(){return (L3Time);}
int getLastexecutionTime(){return (lastexecutionTime);}
```

在 class 中加入需要的參數及相關的設定函式。

In thread.cc

Thread::Thread(char* threadName, int threadID)

```
setExecutionTime(0);
```

初始化執行時間

In scheduler.cc

1. Scheduler::Scheduler()

```
L2ReadyList = new SortedList<Thread *>(PriorityCompare);
```

使用 SortedList 類別來存要執行的 threads

2. Scheduler::~Scheduler()

```
delete L2ReadyList;
```

刪除 Schedule

```
int PriorityCompare(Thread *a, Thread *b) {
    if(a->getPriority() != b->getPriority()){
        //cout << a->getName() << ": " << a->getPriority() <<
        //kernel->scheduler->Print();
        return a->getPriority() > b->getPriority() ? -1 : 1;
    }else{
        return a->getID() < b->getID() ? -1 : 1;
    }

    return 0;
}
```

使用以上程式做為比較依據。若優先度不同則大的優先，優先度相同 ID 小的優先。

3. Scheduler::ReadyToRun(Thread *thread)

```
DEBUG(dbgSche, "[A]Tick[" << kernel->stats->totalTicks << "]: Thread[" << thread->getID() << "] is inserted into queue");

thread->setStatus(READY);
L2ReadyList->Insert(thread);
```

將 thread 設定為 ready 並 insert(排序)到 Schedule。在執行此函式時會輸出 insert 的 debug 資訊。

4. Scheduler::FindNextToRun ()

```
if (!L2ReadyList->IsEmpty()){
    nextThread = L2ReadyList->RemoveFront();
    DEBUG(dbgSche, "[B]Tick[" << kernel->stats->totalTicks
        << "]: Thread[" << nextThread->getID() << "] is removed from queue");
    return nextThread;
} else {
    return NULL;
}
```

取出 Schedule 中最前面(優先度最大)的 thread。在執行此函式時會輸出 remove 的 debug 資訊。

5. Scheduler::Run (Thread *nextThread, bool finishing)

```
oldThread->setExecutionTime(oldThread->getExecutionTime()
                                + kernel->stats->totalTicks - oldThread->getLastexecutionTime());

DEBUG(dbgSche, "[D]Tick[" << kernel->stats->totalTicks << "]: Thread[" << nextThread->getID() <<
    "] is now selected for execution, thread[" << oldThread->getID() << "] is replaced, and it has executed [" <<
    oldThread->getExecutionTime() << "] ticks");     //12-7

nextThread->setLastexecutionTime(kernel->stats->totalTicks);

SWITCH(oldThread, nextThread);
```

在 switch 前新增計算執行時間的算式及 debug 輸出。計算方法為已執行 tick + 當下 tick - 上次執行 tick。

In kernel.cc

6. Kernel::Kernel(int argc, char **argv)

```
}else if (strcmp(argv[i], "-ep") == 0) {
    ASSERT(i + 2 < argc);
    execfile[++execfileNum]= argv[++i];
    threadPriority[execfileNum] = atoi(argv[++i]);
    if(threadPriority[execfileNum] > 149) {
        threadPriority[execfileNum] = 149;
    }
    if(threadPriority[execfileNum] < 0){
        threadPriority[execfileNum] = 0;
    }
    cout << execfile[execfileNum] << "\n";
    cout << "Priority = " << threadPriority[execfileNum] << "\n";
}
```

新增接收-ep 的指令並將優先度鎖定在 0~149。

7. void Kernel::ExecAll()

```
int a = Exec(execfile[i], threadPriority[i]);
```

新增接收優先度的欄位。

1. int Kernel::Exec(char* name, int priority)

```
t[threadNum]->setPriority(priority);
```

新增設定優先度。

In debug.h

```
const char dbgSche = 'z';        // scheduler
```

新增 debug flag。