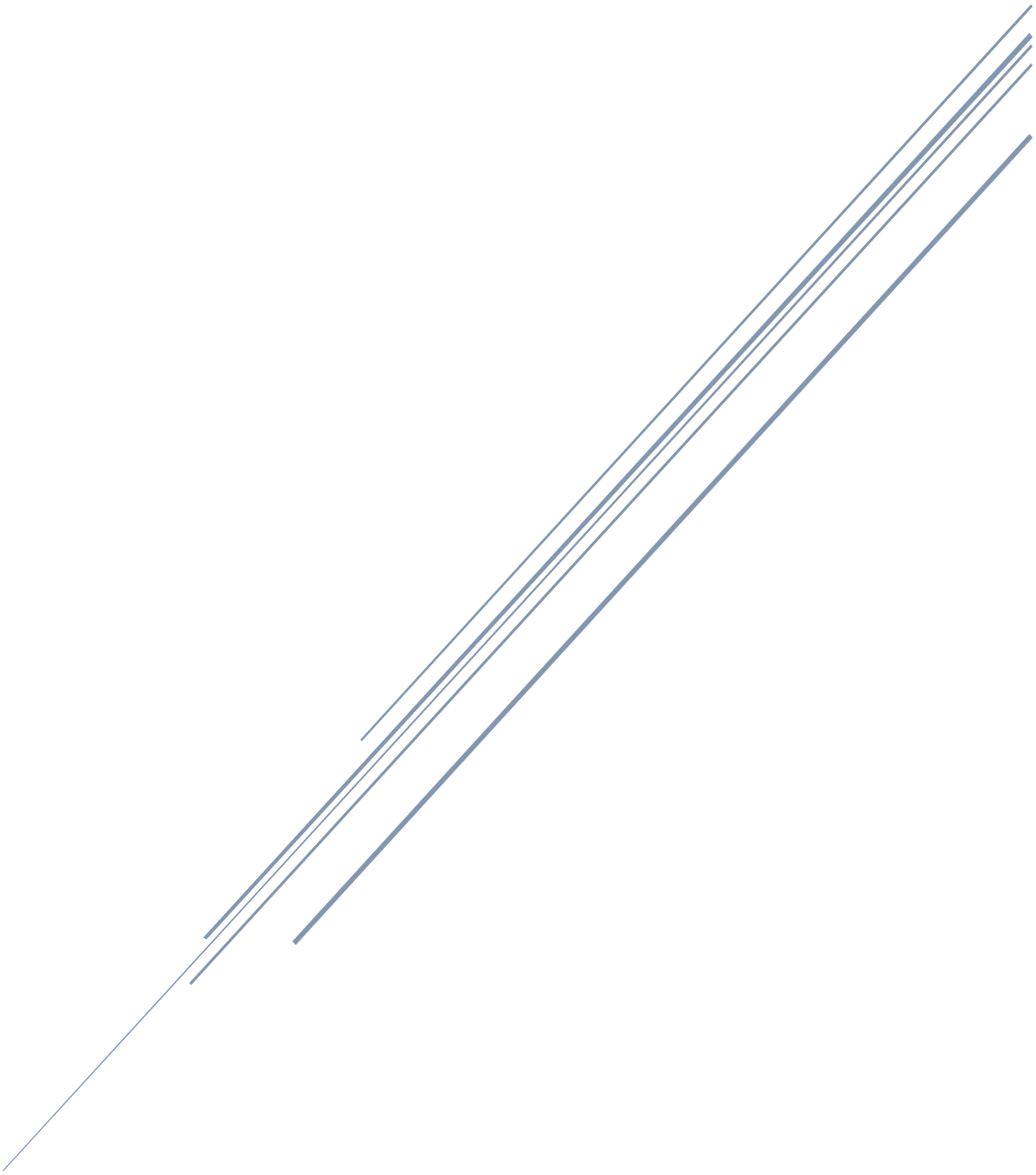
MP3: CPU SCHEDULING

Team 21

Operating System



➢ Team Contibution:

Trace Code、Code Implement: 107062134 樊明膀

Trace Code、Code Implement: 107062333 湯睿哲

➢ Trace code

1-1.New→Ready

void Kernel::ExecAll()

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

這個 function 裡面 call 了一個 for 迴圈，做 Exec，execfileNum 是要做的數量，做完迴圈之後讓現在的 thread 進到 finish
currentThread 是目前，掌握著 CPU 的 thread

```
Kernel::Kernel(int argc, char **argv)
{
    debugUserProg = TRUE;
} else if (strcmp(argv[i], "-e") == 0) {
    execfile[++execfileNum]= argv[++i];
    cout << execfile[execfileNum] << "\n";
```

execfileNum 的數字的處理在 kernel 的創建的時候，在處理 command 的時候，如果 command 有-e 的話數量就會增加，並且使 execfile 對應到輸入的字上面，對應到的是-e 後面的字串。

```
char*    execfile[10];
```

Execfile 是 char*的型別，用來存儲字串，在 kernel.h 裡面

int Kernel::Exec(char* name)

```
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;
```

在 Exec 裡面第一步先創一個新的 Thread
這裡的 name 是當初 execfile 裡面的字串，也就是-e 後面那一個

```
Thread* t[10];
```

T 是 Thread 的 pointer

```
currentThread = new Thread("main", threadNum++);
```

Threadnum 在 Kernel::Initialize()會增加

第二行則是創建了 AddrSpace

```
AddrSpace *space;      // User code this thread is running.
```

space 是 AddrSpace 的 pointer，AddrSpace 是一個資料的結構，用來記錄使用者 memory 程式的狀況，後面會解釋
第三是 thread 去 call 了 fork，可以讓 thread run，後面會在解釋
最後是 threadnum 的數量增加

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

exec 第三行 Thread 呼叫了 Fork，使 thread 可以運作，允許 caller and callee 的運作同時進行

```
void Fork(VoidFunctionPtr func, void *arg);
        // Make thread run (*func)(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);
}
```

傳了一個 function(ForkExecute)跟一個 pointer(thread*)進去，然後取了 kernel 的 interrupt,scheduler，以及創一個 IntStatus

```
IntStatus SetLevel(IntStatus level);
        // Disable or enable interrupts
    // and return previous setting.
```

可以停止或起動 interrupt 但這邊是停止，並將回傳前一個的設定存在 oldLevel 裡面

```
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;
}
```

到最後面的時候在加到 readyqueue 的後面，有在將他設回 oldlevel

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

thread.cc

啟動且設置好 stack，func 是要 fork 的 procedure，*arg 是要給 procedure 的 parameter

```
// Size of the thread's private execution stack.
// WATCH OUT IF THIS ISN'T BIG ENOUGH!!!!!
const int StackSize = (8 * 1024); // in words
```

StackSize 的大小

```
char *
AllocBoundedArray(int size)
{
#ifdef NO_MPROT
    return new char[size];
#else
    int pgSize = getpagesize();
    char *ptr = new char[pgSize * 2 + size];

    mprotect(ptr, pgSize, 0);
    mprotect(ptr + pgSize + size, pgSize, 0);
    return ptr + pgSize;
#endif
}
```

sysdep.cc

設置 stack 大小的 function

pgSize 就是 page 的 size，stack 的大小是 page*2+StackSize(上面有查到)

mprotect 就是將這一塊 memory 保護起來不要讓別人用

fork 最後 call scheduler-> ReadyToRun (this);

this 就是這個 thread 使他加入到 ready queue，ReadyToRun 把這個 thread 標記成 ready，並加入到 readylist

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

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);
}
```

```
machineState[PCState] = (void*)ThreadRoot;
machineState[StartupPCState] = (void*)ThreadBegin;
machineState[InitialPCState] = (void*)func;
machineState[InitialArgState] = (void*)arg;
machineState[WhenDonePCState] = (void*)ThreadFinish;
```

在 StackAllocate 的最後設置了 machineState 等等的東西

```
extern "C" {
// First frame on thread execution stack;
//     call ThreadBegin
//   call "func"
//   (when func returns, if ever) call ThreadFinish()
void ThreadRoot();
```

Threadroot 是 thread 在執行時候的框架 從 begin 到 func 直到結束 call finish 裡面有初始化需要用到的 register，以便於傳給 threadRoot 值，第一組的 register 是給 threadroot 用的，第二組是 thread 這個 object 裡面用的在 StackAllocte()也就是上面那邊

```
The following are the initial registers we need to set up to
 * pass values into ThreadRoot (for instance, containing the procedure
 * for the thread to run).  The first set is the registers as used
 * by ThreadRoot; the second set is the locations for these initial
 * values in the Thread object -- used in Thread::StackAllocate().
 */
#define InitialPC   s0
#define InitialArg  s1
#define WhenDonePC  s2
#define StartupPC   s3

#define PCState     (PC/8-1)
#define FPState     (S6/8-1)
#define InitialPCState  (S0/8-1)
#define InitialArgState (S1/8-1)
#define WhenDonePCState (S2/8-1)
#define StartupPCState  (S3/8-1)
```

Func 的 pointer 是 forkExecute 這一個，arg 就是那個 thread 的 pointer

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

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

}
```

thread/kernel.cc

1-2. Running→Ready(yield)

Machine::Run()

就是要 run 一個 user 的指令

```
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);
```

創造需要 decode 的 instruction 的 Storage，中間的 debug 不用理他，debug 主要是告訴你說，你現在執行順序中的 ins 是哪一個，然後時間是怎麼樣
Kernel ->interrupt->setStatus(Usermode)kernel 用 interrupt 把 mode 設成 usermode 避免之後在執行程式的時候，使用到了 kernel 才可以用的指令，保護電腦，使 user 不能隨意的更改一些東西。

```
for (;;) {
DEBUG(dbgTraCode, "In Machine::Run(), into O
    OneInstruction(instr);
DEBUG(dbgTraCode, "In Machine::Run(), return

DEBUG(dbgTraCode, "In Machine::Run(), into O
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return
if (singleStep && (runUntilTime <= kernel->s
    Debugger();
}
```

```
const char dbgTraCode = 'c';
```

在 for 迴圈裡面 call instruction 用 OneInstruction()使他一直往下面執行，然後算 oneTick()這個跟 call back function 的時間有關，會影響 instruction 的排序問題，後面會再解釋

Machine::OneInstruction
就是在處理一個普通的 user 的指令
但如果中間有發生 interrupt 的情形的話，就會啟動到 interrupt handler，如果要回到這個 function 的話一樣是要經過 run()這個 function 避免出錯
還有就是如果有 interrupt 在之前還是會把變數那些的好好的存好在 mem or reg，而在下次回來的時候，就回 restart 上次的 instruction，並取得正確的數值。

```
#ifdef SIM_FIX
    int byte;       // described in Kane for LWL,LWR,...
#endif
```

這個會跟 mips 的 instruction 的讀法有關等等，用 SIM_FiX 這個 flag 是否 def 去決定讀法做修正，後面也常常會有類似的方法

```
// Fetch instruction
if (!ReadMem(registers[PCReg], 4, &raw))
return;          // exception occurred
instr->value = raw;
instr->Decode();
```

從 mem 取得 instruction
Instr->Decode() 是 decode the binary representation of the instruction

```
unsigned int value; // binary representation of the instruction

char opCode;     // Type of instruction.  This is NOT the same as the
                 // opcode field from the instruction: see defs in mips.h
char rs, rt, rd; // Three registers from instruction.
int extra;       // Immediate or target or shamt field or offset.
                 // Immediates are sign-extended.
```

Instr 的結構是這個樣子 value 記下 raw 的值，也就是 memory 對應到的 pcreg 的值，然後解析之後確認是什麼指令(opcode) reg 的位置在哪(rs,rt,rd)或是 immd or 額外的 bit(extra)

```
switch (instr->opCode) {
```

根據 opcode 去做相對應的動作(case)，動作裡面也些也會因為 SIM_FIX 而讀寫的方式不太一樣

```
// Do any delayed load operation
DelayedLoad(nextLoadReg, nextLoadValue);

// Advance program counters.
registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
                                            // are jumping into lala-land
registers[PCReg] = registers[NextPCReg];
registers[NextPCReg] = pcAfter;
```

Delayedload 跟 interrupt 或處理 exception 那些有關係
然後下面就是要將跑過的指令替換掉了 存下跑過的原因是為了 debug 然後把下一個要跑的，替換過來

Interrupt::OneTick()

兩種情況會啟動這個 onetick，一個是 user instruction 執行的時候，另一個是 interrupt 又遇到 interrupt
這個功能主要是確認是不是有 pending interrupt 需要執行

```
advance simulated time
if (status == SystemMode) {
    stats->totalTicks += SystemTick;
stats->systemTicks += SystemTick;
} else {
stats->totalTicks += UserTick;
stats->userTicks += UserTick;
}
```

確認好時間，用 mode 判別

```
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;
}
```

Changelevel 是因為避免有在 handler 遇到 interrupt 又插入的情況

Interrupt::CheckIfDue(bool advanceClock)

這個功能主要是拿來判斷說有沒有排程中的 interrupt 需要發生的，有的話就讓它發生

```
"advanceClock" -- if TRUE, there is nothing in the ready queue,
    so we should simply advance the clock to when the next
    pending interrupt would occur (if any).
```

advanceClock ，如果是的話就是現在沒有東西 ready，而我們要提前 clock 讓下一個 interrupt 發生，但 OneTick 裡面的 advanceClock==false

```
  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.
  }
  }
```

CheckIfDue 裡面去判斷說時間的條件有沒有滿足，以及有沒有前面說的 advanceClock 的要求，如果時間沒到的話 就會 return False

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

看一下 machine 有沒有在忙

```
inHandler = TRUE;
do {
    next = pending->RemoveFront();     // pull interrupt off list
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack,
    next->callOnInterrupt->CallBack();// call the interrupt handler
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->Cal
delete next;
} while (!pending->IsEmpty()
        && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
return TRUE;
```

說明正在處理 ishandler=true 等做完在設成 false，然後中間 do while 處理第一筆後，如果後面的時間一樣>=when(interrupt 要發生的時間)的話就繼續做，while 裡面的動作就是一一的清空 pending 裡面的名單，然後 call 對應的 interrupt handler。

這邊因為是要 print out 所以會是對應到 output

yieldOnReturn 是應對 handler 的一些要求 switch context

```
// Interrupt::YieldOnReturn
//  Called from within an interrupt handler, to cause a context switch
//  (for example, on a time slice) in the interrupted thread,
//  when the handler returns.
//
//  We can't do the context switch here, because that would switch
//  out the interrupt handler, and we want to switch out the
//  interrupted thread.
voidInterrupt::YieldOnReturn()
{
    yieldOnReturn = TRUE;
}
```

如果判斷出來是 true 的話，就 call 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);
}
```

可以看到說，找下一個 thread 藉由 FindNextToRun，如果有抓到的話，就將自己現在的 thread 排進 ready queue 然後，run nextThread

```
Thread *Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

有抓到的話，因為要 pop 所以 removeFront

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

將現在自己的這個 thread 設成 ready 然後，將他加入到 ready queue

```
void Scheduler::Run (Thread *nextThread, bool finishing)
```

留到 1-6. Ready→Running 的時候說

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

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

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

threads/synch.cc

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

Lock 裡面有用 semaphore->P(); 這個時候，semaphore 的 value 應該等於 1，call 完這個之後會變成零(lock->Acquire())

waitFor->P()就是在等 putchar 的時間

```
Semaphore *waitFor;    // wait for callBack
```

Putchar 一樣先 lock 然後裡面 call 了 console.cc 的 Putchar()

```
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);
}
```

因為前面 lock->Acquire()的關係，所以這時候的 value 是等於零的，因此這個 queue(waiting queue) Append 現在的 Thread 然後使這個 Thread 進入 sleep

這邊的 sleep false 的意思是 這個程式還沒有 finish，true 才是 finish 的

```
List<Thread *> *queue;
        // threads waiting in P() for the value to be > 0
```

這邊的 list 就是一般的 list append 的操作

```
template <class T>
void List<T>::Append(T item)
{
    ListElement<T> *element = new ListElement<T>(item);
    if (IsEmpty()) {        // list is empty
    first = element;
    last = element;
    } else {            // else put it after last
    last->next = element;
    last = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

```
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);
}
```

在說，現在的這個 thread status 先設成 block
找到下一個要 run 的 Thread 如果找不到的話就 idle()這是一個 interrupt，因為會用到 sleep 往往都是要等 I/O 所以 current Thread 也不能繼續
kernel->scheduler->FindNextToRun() 在前面的 1-2 解釋過了
kernel->scheduler->Run(nextThread, finishing); 的部份等到 1-6

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

這邊接續到前面的 SynchConsoleOutput::PutChar 裡的 Lock->Release()

```
void Lock::Release()
{
    ASSERT(IsHeldByCurrentThread());
    lockHolder = NULL;
    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);
}
```

因為等到 putchar waitfor->P()結束了之後，IO 就做完了那這個 Thread 就應該要回到 ready 的狀態，從 waiting 到 ready queue，然後 value 增加，因為 lock 釋放

那 ReadyToRun 在前面的 1-1 的 Thread::Fork 那邊有解釋過了

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

```
case SC_Exit:
                    val=kernel->machine->ReadRegister(4);;
            kernel->currentThread->Finish();
            break;
```

kernel->currentThread->Finish(); 主要就注意這一條

```
void Thread::Finish ()
{
    Sleep(TRUE);                // invokes SWITCH
    // not reached
}
```

跟 1-3 那邊的 sleep 那邊不同的是，這邊的 sleep 傳進去 true

```
void Thread::Sleep (bool finishing)
{

    kernel->scheduler->Run(nextThread, finishing);
}
```

回到 sleep 主要注意到裡面的這個 run，這時候 finishing 傳進去的是 true

那 run 這邊一樣等到 1-6 說明

1-6. Ready→Running

可以看到說有三個時候要會有這個情況的發生，current Thread 從 run 到 waiting、run 到 terminate 以及 run 到 yield
因此都 call 到了 Scheduler::FindNextToRun() 這個在前面解釋過了
之後就是進入到 run 的狀態

Void Scheduler::Run (Thread *nextThread, bool finishing)

這個在 Scheduler，將原本的 thread 停掉，並將下一個 thread 布置好

```
    Note: we assume the state of the previously running thread has
already been changed from running to blocked or ready (depending).
```

在 sleep 裡面有將原本的 state 設 block 了
nextThread 是下一個要 run 的 thread
finishing 是原本跑的 thread，看是不是做完結束了，如果是結束的要把他 destory 掉

```
    Thread *oldThread = kernel->currentThread;
```

oldthread 紀錄原本的 thread

```
    ASSERT(kernel->interrupt->getLevel() == IntOff);
```

確認 interrupt disable

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

原本的 thread 如果是結束的，需要 destory 掉，這邊是將他記在 toBeDestroyed

```
    CheckToBeDestroyed();    // check if thread we were running
```

在後面的時候有 call 這個去 destory 原本的 thread，如果 toBeDestroyed 有的話，等 destory(delete)之後，toBeDestroyed 設回 null

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

因此像 1-5 的部分因為 finishing 傳進來的是 true 所以之後就會被 destroy

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

記好原本的 thread 的狀態像是 register、page table 或是 state

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

thread.cc

```
void AddrSpace::SaveState()
{}
```

現在被說不需要記住東西

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

Chech 原本的 thread 有沒有 overflow 的情況發生
如果得到奇怪的結果(seg faults)可能就是這邊有出問題，因為我們的 complier 沒有做檢查，要處理的話，我們可能要在 stack 的 size 做處理

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

把 kernel 的 thread 轉到 nextThread 上面，並將他 state 設為 running

```
    SWITCH(oldThread, nextThread);
```

在 SWITCH.s 裡面，整理來說 就是將原本的 thread 的 register 上面的東西存好，然後將下一個 thread 的東西載入下來

簡單的來說就是 register 的互換
其中的 eax ebx 就只是位置

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
```

```
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
```
eax 的值存下來
```
        movl    4(%esp),%eax            # move pointer to t1 into eax
```
stack pointer+4 存到 eax
```
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
```
這邊上面都是做 copy 的挪移
```
        movl    _eax_save,%ebx          # get the saved value of eax
```
原先 eax 的值給到 exb
```
        movl    %ebx,_EAX(%eax)         # store it
```
原先 eax 的值給到 eax(已經 stack pointer+4)
```
        movl    0(%esp),%ebx    # get return address from stack into ebx
```
stack pointer+0 的值給 ebx
```
        movl    %ebx,_PC(%eax)          # save it into the pc storage
```
傳到 pc 的 storage
```
        movl    8(%esp),%eax            # move pointer to t2 into eax
        stack pointer+8 的給到 eax
        movl    _EAX(%eax),%ebx        # get new value for eax into ebx
```
此時 eax(stack pointer+8)的 eax 給到 exb
```
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax         # restore return address into eax
```
取回下一個 thread 的 register 的資料
```
        movl    %eax,0(%esp)    # copy over the ret address on the stack
```
將現在的 eax(前面加 stack+8 的)變成現在的 esp
```
        movl    _eax_save,%eax 取回原本的 ebx 也是(stack pointer+8)的 eax
```

```
        ret
```

set CPU program counter to the memory address pointed by the value of register esp

ecx: points to startup function

- 對應到C的(void*)ThreadBegin (interrupt enable)
- 裡面會做kernel->interrupt->Enable();

edx: contains inital argument to thread function

- 對應到C的(void*)arg;

esi: points to thread function

- 對應到C的(void*)func (其實就是ForkExecute))

edi: point to Thread::Finish()

- 對應到C的(void*)ThreadFinish

esp (組語執行到最後，esp裡面會存放新Thread的PCState的值)

- 對應到C的(void*)ThreadRoot;

這個部分跟前面所設的 machinestate 那邊有關係

那上面的 switch 結束了之後，如果又回到這個 function 的話，就是因為又輪到這個 Thread 了
那這邊會有前面的 toBeDestroyed 的標示，如果要的話就會用到

```
Thread::~Thread()
{
    ASSERT(this != kernel->currentThread);
    if (stack != NULL)
    DeallocBoundedArray((char *) stack, StackSize * sizeof(int));
}
```

前面確認了 this thread 不是 kernel 現在的 thread 因為要刪除的 thread 不能夠刪掉自己

那如果沒刪除的話就是下面這樣

```
if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
oldThread->space->RestoreState();
}
```

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

AddrSpace:RestoreState()
在 Context switch 的時候，要找回原本的 page Table 不然對應的 physical address 會不一樣

Thread::RestoreUserState() 在下面，敘述說這是在處理 user code 的，並不是 executing kernel code

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

如果沒有刪除的話 就要再回來，因為一個 thread 無法停掉自己，會需要別的 thread 然後 call finish，因此前面有描述一個副作用是 kernel->currentThread becomes nextThread.

```
Side effect:
 The global variable kernel->currentThread becomes nextThread.
```

Actually switch to the next thread by invoking *Switch()*. After *Switch* returns, we are now executing as the new thread. Note, however, that because the thread being switched to previously called *Switch* from *Run()*, execution continues in *Run()* at the statement immediately following the call to *Switch*.

If the previous thread is terminating itself (as indicated by the *threadToBeDestroyed* variable), kill it now (after *Switch()*). As described in Section 3, threads cannot terminate themselves directly; another thread must do so. It is important to understand that it is actually another thread that physically terminates the one that called *Finish()*.

那回來的這個 thread 不管之前的狀況怎麼樣[New,Running,Waiting]，因為 call scheduler-> ReadyToRun (this);或是其他地方等等都會將 status 設成 ready，那 terminate 的就是剛剛拿去交給下一個 thread 刪掉了，接下來就會因為這個 instruction 做完了回去到 machine::run 裡面的迴圈繼續做

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

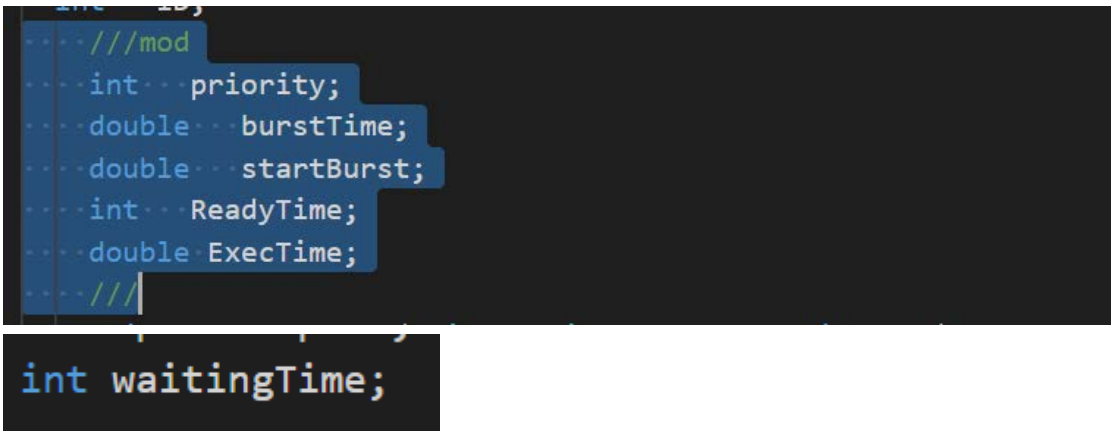
```
    for (;;) {
            OneInstruction(instr);
    kernel->interrupt->OneTick();
    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
        Debugger();
    }
}
```

繼續往下做 instruction

➢ Code_Implements:

● Thread.h & Thread.cc

1. 先在 Thread.h 裡新增一些變數，用來存取每個 thread 的 priority, 發生 busrt 的時間點(Startburst), 最近一次放入 queue 的時刻 (ReadyTime),完成 thread 所需的時間(burstTime)以及在 ready queue 裡等待的時間(waitingTime)。



2. 並宣告其變數取值的 functions(get…)以及更新的 functions (set…)，並於 Thread.cc 進行撰寫，其中 setBurstTime(...)裡的 sb 代表 startBurstTime, b 代表目前的 burstTime, 依據 t(i) = t(i-1)*0.5 + ExecutionTime*0.5 = b*0.5 + (kernel->stats->totalticks - sb)*0.5 計算出新的 burstTime。

   Thread.h:

```
void setPriority(int new_priority);
void setReadyTime();
void setBurstTime(double sb, double b);
void setStartBurst();
void setExecTime(double t);
int getPriority();
int getBurstTime();
int getReadyTime();
int getStartBurst();
int getExecTime();
```

(附註: 由於在 deadline 前臨時修改 code，所以先將 waitingTime 放在 publice，以方便直接取用)

3. Spec 中提示, 當從 running state 進入 waiting state 時，需要更新 burstTime，因此我在 Thread::Sleep()裡會進行 BurstTime 的更新

```
status = BLOCKED;
///更新busrt_time
this->setBurstTime(this->getStartBurst(),this->getBurstTime());
```

- # Scheduler, 流程

1. 為應付 multi-feedback-queue,我們需要對 scheduler 進行修改, 因此在我先宣三個 queue 分別對應 L1 (preemptive SJF), L2(non-preemptive priority)以及 L3(Round Robin)於 scheduler.h 裡:

```
private:
  //List<Thread *> *readyList;  // queue of threads that are ready to run,
      // but not running
  SortedList<Thread *>* readyListL1; //SJF
  SortedList<Thread *>* readyListL2; //non-pre-pri
  List<Thread *>* readyListL3; // RR
  Thread *toBeDestroyed;  // finishing thread to be destroyed
        // by the next thread that runs
```

而在 scheduler 的建構式中，由於 L1 以及 L2 分別需要對 BurstTime 以及 Priority 做比較，因此需要分別 define 其 compare function:

```
static int comp_burst_time(Thread* T1, Thread* T2){
    int result = 0;
    if(T1->getBurstTime()>T2->getBurstTime())result = 1;
    else if(T1->getBurstTime()<T2->getBurstTime())result = -1;
    else{
        if(T1->getID() > T2->getID())result = 1;
        else if(T1->getID() < T2->getID())result = -1;
    }
    return result;
}

static int comp_priority(Thread* T1, Thread* T2){
    int result = 0;
    if(T1->getPriority() > T2->getPriority())result = -1;
    else if(T1->getPriority() < T2->getPriority())result = 1;
    else{
      if(T1->getID() > T2->getID())result = 1;
      else if(T1->getID() < T2->getID())result = -1;
    }
    return result;
}
```

2. 根據 spec 要求，若 thread 的 waiting time 超過 1500ticks 時，需要進行 aging (即 update priority)，因此我們可以在每次 alarm callback 觸發的時候(每 100ticks)進行 Aging()，並在每次 aging 的時候先讓 waitingTime + 100(因為一次 alarm 會 call 一次 Aging(), 而 alarm callback 每 100ticks 一次)，再來判別總 waitingTime 是否超過 1500，若是則 update priority，並讓 waitingTime -1500。

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    kernel->scheduler->Aging();
    if(status != IdleMode && (!kernel->scheduler->readyListL1->IsEmpty() || kernel->currentThread->getPriority()<50 || kernel->currentThrea
        interrupt->YieldOnReturn();
    }
}
```

(被切掉的部分: kernel->currentThread->getPriority()>99)
Note:由於 L2 並不會動內部 threads 進行互相 preemptive，因此藉由判別 L1 是否有 thread 或目前 thread 不屬於 L2 來決定是否需要進入 YeildOnReturn()的 function 進行 preemptive(即 call Thread->Yield()來實現)。

而 aging 的實際 implements 如下，可以利用 list iteration 來遍力所有已排程到 queue 裡面的 threads，檢查他的 waiting time 是否超過 1500，若是，則讓他的 priority +10, 並在檢查該 thread 因為新的 prioruty 是否需要搬動到別的 queue。(以 queue L2 為例)

```
for(; !l2->IsDone(); l2->Next()){
    Thread *curThread = l2->Item();
    curThread->waitingTime+=100;
    if(curThread->waitingTime >= 1500){
        curThread->waitingTime -= 1500;
        curThread->setReadyTime(); /// reflash wiating time = 0
        int old_priority = curThread->getPriority();
        int new_priority = curThread->getPriority()+10;
        if(new_priority >= 100){ ///move to L1 from L2
            //cout<<"L2Aging!\n";
            curThread->setPriority(new_priority);
            kernel->scheduler->readyListL2->Remove(curThread);
            DEBUG('z',"[B] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["<<curThread->getID()<<"] is removed from queue L["<<2<<"]")
            kernel->scheduler->readyListL1->Insert(curThread);
            DEBUG('z',"[A] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["<<curThread->getID()<<"] is inserted into queue L["<<1<<"]"
        }
        else curThread->setPriority(new_priority); /// remain in L2
        //DEBUG('z',"[C] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["<<curThread->getID()<<"] changes its priority from ["<<old_pr
    }
}
```

3. 接下來，在排程的部分，可以修改 scheduler::ReadyToRun()，依據傳入的 thread 的 priority，放入對應的 queue，並記錄它放入 queue 的時間點(setReadyTime)

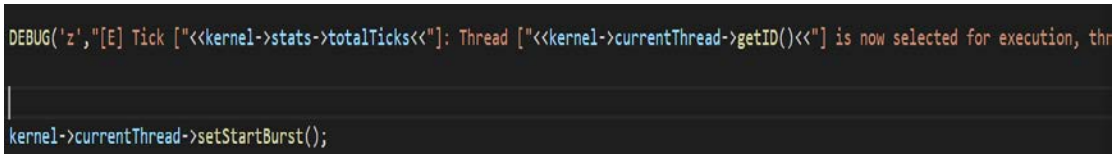
```
///mod
int insert_level;
if (thread->getPriority() <= 49) { ///L3
    readyListL3->Append(thread);
    insert_level = 3;
}
else if (thread->getPriority() >= 50 && thread->getPriority() <= 99) { ///L2
    readyListL2->Insert(thread);
    insert_level = 2;
}
else if (thread->getPriority() >= 100 && thread->getPriority() <= 149) { ///L1
    readyListL1->Insert(thread);
    insert_level = 1;
}
DEBUG('z',"[A] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["<<thread->getID()<<"] is inserted into queue L["<<insert_level<<"]");
///
```

4. 而 FindNextToRun()是用來找下一個要執行的 Thread，會呼叫的原因有 Run 完當前 thread, 發生 thread Yeild 時以及, 以及 thread sleep 時，做法是先後從 L1, L2, L3(優先順序:L1>L2>L3)挑出一個還未被執行(即處於 ready 狀態)的 thread，並回傳給 caller。

```
///priority L1>L2>L3
int remove_level;
Thread* removed_thread;
if (!readyListL1->IsEmpty()) {
    removed_thread = readyListL1->RemoveFront();
    remove_level = 1;
}
else if (!readyListL2->IsEmpty()) {
    removed_thread = readyListL2->RemoveFront();
    remove_level = 2;
}
else if (!readyListL3->IsEmpty()) {
    removed_thread = readyListL3->RemoveFront();
    remove_level = 3;
}
else{
    return NULL;
}
DEBUG('z',"[B] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["<<removed_thread->getID()<<"] is removed from queue L["<<remove_level<
return removed_thread;
```

5. Thread:: Yield():
   當 preemptive 發生時，thread 之間要進行切換先後的動作，因此需藉由 Yield()先藉由 FindNextToRun()來取得下一個可執行的 thread，並對當前 thread 進行 preemptive(即把當前 thread 利用 readyToRun 放入對應的 queue 並進入 waiting, 並把欲執行的 thread 利用 Run(thread,False)進入執行動作)，

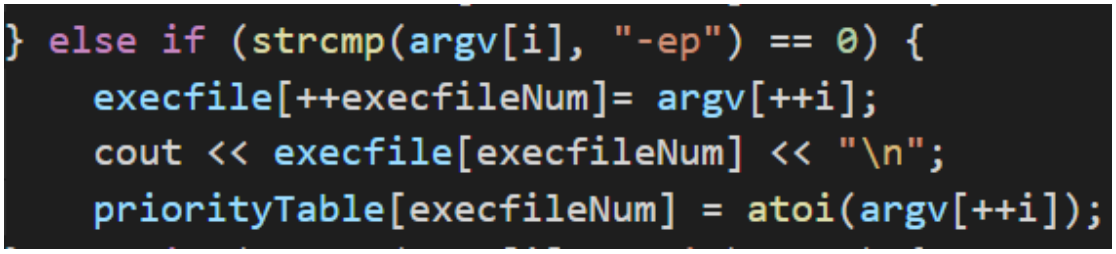
6. 最後，在 scheduler::RUN()裡，我們需要將 StartBurstTime 更新成當前 totalticks，並記錄下來即可。



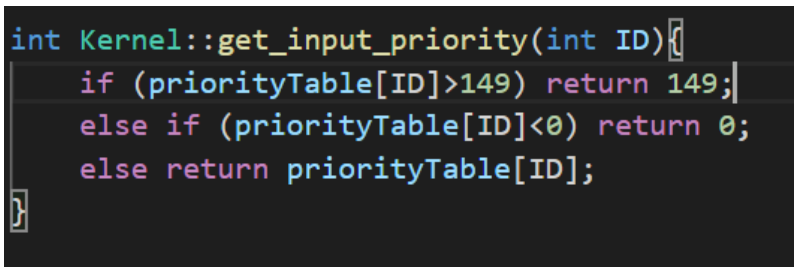
7. 最後，當 Thread 進入 Sleep()時(Waiting State)，需進行 burstTime 的更新，並把 waitingTime 歸零。

- Define -ep:

1. 這部分只需要在 kernel.cc 中的建構式中多加一條 else if 判別即可:



   值得一提的是，由於我是在 thread 的建構式做 initial，但讀取流的 arguments 會在 kernel.cc 中收到，因此我可以宣告一個陣列 priorityTable，用來存取輸入檔名的 ID 對應的輸入 priority 值，並在 thread 建構式中呼叫 kernel->get_input_priority()來取得其剛開始對應的 priority。



- 加入 Debug('z',...)訊息

需要加入 debug 訊息的地方為以下:
[A]: 加在每次 call list.insert()的下方
[B]: 加在每次 call list.RemoveFront()的下方
[C]: 加在每次 Aging 裡判別 waitingTime>=1500 為真的情況下，call 完 thread->setPriority()的下方
[D]: 直接加在 thread->setBurstTime(double sb, double b)裡面算完新的 burstTime 下方即可

- [E]加在 scheduler->RUN()裡得到下一個欲執行的 thread(next_thread)下方即可，