MP3_Report_Team11

Team Member

- 資工大三 108062213 顏浩昀
- 電資大三 108061250 吳長錡

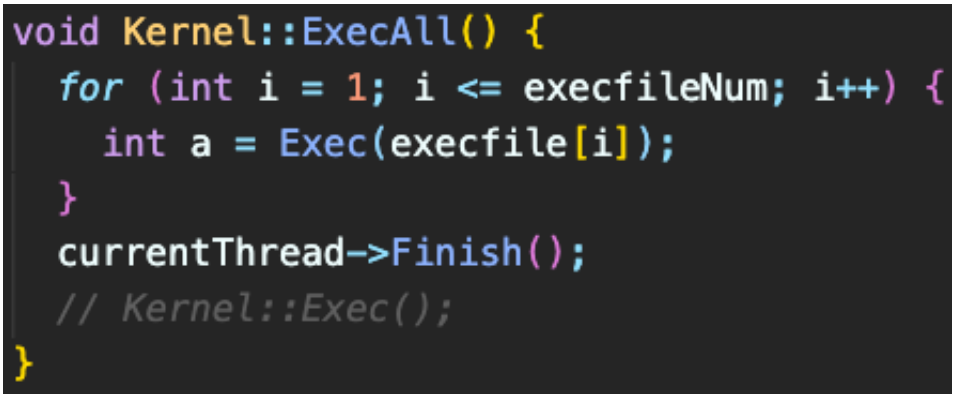
Contributions

Work	Member
Trace code	顏浩昀&吳長錡
Implement	顏浩昀&吳長錡
Debug	顏浩昀&吳長錡
report（TraceCode）	吳長錡
report（Implement）	顏浩昀

PartII-1 Trace code

1.1 New -> Ready

Kernel::ExecAll()



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

在 ExecAll() 中就可以看到會一個一個去執行所有的程式，在執行完後執行 Finish()

```
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    Sleep(TRUE);                                // invokes SWITCH
    // not reached
}
```

於是在 Finish() 會呼叫 Sleep()

Kernel::Exec(char*)

```
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;
```

從Kernel::Exec中我們可以知道要執行執行一個程式，我們需要

1. 新增一個Thread
2. 給一個AddrSpace
3. 透過Fork()載入要執行的程式碼
4. 紀錄Thread數量+1，並會傳現在是第幾個Thread

Thread::Fork(VoidFunctionPtr, void*)

```
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()` 的部分，我們發現Fork還有執行 `StackAllocate` 、 `SetLevel` 與 `ReadyToRun` ，用來執行：

1. Allocate a excution stack and init it.
2. 將interrupt設為IntOff
3. 將要Thread放入readyList中

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

Thread::StackAllocate(VoidFunctionPtr, void*)

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

1. `AllocBoundedArray` 會回傳一個stack的指標，指向一段合法可以access的記憶體(此thread的stack)

```
#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
```

2. stackTop指向stack+StackSize-4的位址
3. stackTop-1的地方指向ThreadRoot
4. 使得之後去access stackTop-1時會從ThreadRoot進去
5. stack指標(thread的stack的bottom的pointer) 指向 STACK_FENCEPOST

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

6. 最後讓machineState各個特定element去存特定的 function pointer
7. 為了之後context switch的時候，能讓kernel正確執行特定的function及正確找到這個thread的stack

Scheduler::ReadyToRun(Thread*)

```
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);
}
```

1. 檢查interrupt是否已經disabled
2. 將thread的狀態設為Ready
3. 將此thread放入readyList

1.2 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 (;;) {
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
                      << "== Tick " << kernel->stats->totalTicks << " ==");
    OneInstruction(instr);
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
                      << "== Tick " << kernel->stats->totalTicks << " ==");

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

1. 無窮迴圈FetchInstruction()

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;
  }
}
```

1. 遞增stats裡面所記錄的totalTicks，順便判斷是在SystemMode還是UserMode，增加對應的執行Ticks
2. Disable Interrupt (確保下一條指令執行是Atomic的)
3. CheckIfDue會檢查是否有下一條已經到期的pending Interrupt，並執行它
4. 執行完CheckIfDue後，yield的Flag被設置為True，進入迴圈
5. yieldOnReturn必須先恢復False (不然下一個Thread如果看到Flag為True，可能會有BUG)

6. 這邊切換到SystemMode，並執行Yield()，細節見下一節
7. Thread2執行完畢後回來，切換為oldStatus(通常是切回UserMode)，因為之後又要回到Machine::Run()的迴圈內了

Thread::Yield()

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

  ASSERT(this == kernel->currentThread);

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

  kernel->scheduler->Aging();
  kernel->scheduler->ReArrangeThreads();

  if (kernel->currentThread->InWhichQueue() == 3) {
    nextThread = kernel->scheduler->Scheduling();
    if (nextThread != nullptr) {
      kernel->scheduler->AddToQueue(this, this->priority);
      kernel->scheduler->Run(nextThread, FALSE);
    }
  }

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

1. Yield的目的就是要切換Thread來執行
2. 首先Disable Interrupt (Yield的過程不容許打斷)

3. 從readyList找出nextThread
4. 將目前執行的Thread放回readyList (run -> ready)
5. Run nextThread
6. 將Interrupt Level恢復原本

Scheduler::FindNextToRun()

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

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

1. 檢查readList是否為空
2. 否的話就DeQueue並Return下一條Thread

Scheduler::ReadyToRun(Thread*)

```
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);
}
```

1. 將準備要執行的Thread的Status設置為Ready，並放入readyList

Scheduler::Run(Thread*, bool)

```
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();
    }
}
```

1. Run基本上就是執行下一條Thread，而Context Switch在此進行
2. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)
3. 保存oldThread的UserState，存入Thread
4. Check if the old thread had an undetected stack overflow
5. 將kernel所執行的currentThread改為準備要執行的Thread，並設置Status為Running，接著呼叫SWITCH()進行線程切換
6. 呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
7. 因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
8. 將oldThread的相關states都恢復原狀

1.3 Running -> Waiting

SynchConsoleOutput::PutChar(char)

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

1. 由於console(stdout)為一個互斥存取的物件(不能同時有兩個Thread在做輸出)，故先lock- >Acquire()

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

2. putBusy設為True代表正在putChar，如有其他Thread想同時輸出，ASSERT(putBusy == FALSE)就會報錯
3. PutChar的最後會將ConsoleOutput本身餵進去interrupt pending list (之後會執行CallBack)

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

4. 當ConsoleTime (1 tick)過去，console write interrupt發生，執行ConsoleOutput->Callback( )
5. CallBack裡面又呼叫了callWhenDone->CallBack( )，而先前提過，callWhenDone就是SynchConsoleOutput
6. allWhenDone->CallBack( )，可以發現這邊其實只做了waitFor->V()這個動作 (waitFor是一個Semaphore Class)
7. 由於putChar完成後，console write intterupt發生，最後讓semaphore++ (因為 waitFor- >V( )的關係)
8. 這邊的 waitFor->P( )將會成功執行，讓semaphore–
9. 最後呼叫lock->Release()來釋放lock

Semaphore::P()

```
void
Semaphore::P()
{
    DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
    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semaphore value是否 > 0，若無，while(value==0)成立
3. 將等待semaphore的currentThread Append進queue裡

SynchList< T >::Append(T)

```
template <class T>
void
SynchList<T>::Append(T item)
{
    lock->Acquire();        // enforce mutual exclusive access to the list
    list->Append(item);
    listEmpty->Signal(lock);    // wake up a waiter, if any
    lock->Release();
}
```

1. Append a item into Single Linked List

Thread::Sleep(bool)

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

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

  DEBUG(dbgThread, "Sleeping thread: " << name);
  DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: "
                    << name << ", " << kernel->stats->totalTicks);

  status = BLOCKED;
  // cout << "debug Thread::Sleep " << name << "wait for Idle\n";

  if (!finishing) {
    this->update_ti(kernel->stats->totalTicks);
  }

  while ((nextThread = kernel->scheduler->Scheduling()) == NULL) {
    kernel->interrupt->Idle(); // no one to run, wait for an interrupt
  }

  // 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. 讓等待Semaphore的currentThread變成Blocked的status
2. 接著scheduler從readyList找出下一條要執行的Thread
3. 若無(NULL)，則呼叫Idle( )，裡面會判斷是否該Advance Clock或者直接Halt程式
4. 若有(nextThread)，則呼叫scheduler->Run讓nextThread執行

Scheduler::FindNextToRun()

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

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

1. 檢查readList是否為空
2. 否的話就DeQueue並Return下一條Thread

Scheduler::Run(Thread*, bool)

```
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();
    }
}
```

1. Run基本上就是執行下一條Thread，而Context Switch在此進行
2. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)
3. 保存oldThread的UserState，存入Thread
4. Check if the old thread had an undetected stack overflow
5. 將kernel所執行的currentThread改為準備要執行的Thread，並設置Status為Running，接著呼叫SWITCH()進行線程切換
6. 呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
7. 因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
8. 將oldThread的相關states都恢復原狀

1-4. Waiting -> Ready

Semaphore::V()

```
void
Semaphore::V()
{
    DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
    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. Interrupt已經在呼叫V( )之前就被Disable了，確保Semaphore的操作是Atomic的
2.

Scheduler::ReadyToRun(Thread*)

```
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);
}
```

1. 將準備要執行的Thread的Status設置為Ready，並放入readyList
2. 檢查 Semaphore的 " List<Thread *> *queue " ，並從裡面DeQueue，找出準備從BLOCKED狀態變到READY的Thread
3. emaphore value++
4. 還原interrupt Level (通常是Enable)

1.5 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();
```

1. 執行Finish()

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
}
```

1. Finish裡面再度呼叫Sleep
2. Sleep的參數為True，最後會再被接力傳進Run( )裡面，代表Thread Finishing

Thread::Sleep(bool)

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

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

  DEBUG(dbgThread, "Sleeping thread: " << name);
  DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: "
                    << name << ", " << kernel->stats->totalTicks);

  status = BLOCKED;
  // cout << "debug Thread::Sleep " << name << "wait for Idle\n";

  if (!finishing) {
    this->update_ti(kernel->stats->totalTicks);
  }

  while ((nextThread = kernel->scheduler->Scheduling()) == NULL) {
    kernel->interrupt->Idle(); // no one to run, wait for an interrupt
  }

  // 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. 讓等待Semaphore的currentThread變成Blocked的status
2. 接著scheduler從readyList找出下一條要執行的Thread
3. 若無(NULL)，則呼叫Idle( )，裡面會判斷是否該Advance Clock或者直接Halt程式
4. 若有(nextThread)，則呼叫scheduler->Run讓nextThread執行

Scheduler::FindNextToRun()

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

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

1. 檢查readList是否為空
2. 否的話就DeQueue並Return下一條Thread

Scheduler::Run(Thread*, bool)

```
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();
    }
}
```

1. Run基本上就是執行下一條Thread，而Context Switch在此進行
2. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)
3. 保存oldThread的UserState，存入Thread
4. Check if the old thread had an undetected stack overflow
5. 將kernel所執行的currentThread改為準備要執行的Thread，並設置Status為Running，接著呼叫SWITCH()進行線程切換
6. 呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
7. 因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
8. 將oldThread的相關states都恢復原狀

1.6 Ready -> Running

Scheduler::FindNextToRun()

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

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

1. 檢查readList是否為空
2. 否的話就DeQueue並Return下一條Thread

Scheduler::Run(Thread*, bool)

```
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();
    }
}
```

1. Run基本上就是執行下一條Thread，而Context Switch在此進行
2. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)
3. 保存oldThread的UserState，存入Thread
4. Check if the old thread had an undetected stack overflow
5. 將kernel所執行的currentThread改為準備要執行的Thread，並設置Status為Running，接著呼叫SWITCH()進行線程切換
6. 呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
7. 因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
8. 將oldThread的相關states都恢復原狀

SWITCH(Thread*, Thread*)

1. SWITCH主要是透過switch.h的輔助 (define macro)
2. 以及thread.h內的外部宣告(extern)

3. 最後在switch.s裡面使用組合語言實作

```
#ifdef x86

/* the offsets of the registers from the beginning of the thread object */
#define _ESP     0
#define _EAX     4
#define _EBX     8
#define _ECX     12
#define _EDX     16
#define _EBP     20
#define _ESI     24
#define _EDI     28
#define _PC      32

/* These definitions are used in Thread::AllocateStack(). */
#define PCState         (_PC/4-1)
#define FPState         (_EBP/4-1)
#define InitialPCState  (_ESI/4-1)
#define InitialArgState (_EDX/4-1)
#define WhenDonePCState (_EDI/4-1)
#define StartupPCState  (_ECX/4-1)

#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx

#endif // x86
```

4. 這邊宣告一些register的位置，為了讓switch.s取用

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

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

5. scheduler::Run裡面會呼叫這邊定義的SWITCH
6. 透過extern的宣告以及compiler的輔助，使得x86組語能夠與C語言互相呼叫

```
/* 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.
*/
        .comm   _eax_save,4

        .globl  SWITCH
    .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into 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
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage

        movl    8(%esp),%eax            # move pointer to t2 into eax

        movl    _EAX(%eax),%ebx         # get new value for eax into ebx
        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
        movl    %eax,4(%esp)            # copy over the ret address on the stack
        movl    _eax_save,%eax

        ret

#endif // x86
```

7. 將 t1 (舊thread) 的所有相關 registers 保存起來 (需配置一塊空間於Memory)
8. 將 t2 (新thread) 的所有相關 registers從Memory裡面的對應位置 Load 進 CPU registers 裡面
9. ret -set CPU program counter to the memory address pointed by the value of register esp

(depends on the previous process state, e.g., [New,Running,Waiting]→Ready)

1. 當執行到這邊時，表示控制權又回到Thread 1這邊了
    - 可能是Thread 2那邊又作了一個Context Switch回來
    - 或者它成功finish了
    - 或它等待I/O所以被BLOCKED了
2. 目前Program Counter記載的指令仍然在 Run( ) 函式內
3. 此時Thread 1的Case可能有幾種情形
    - Thread 1 (Old Thread)原先是BLOCKED (waiting) status，然後Run的finishing參數為True
    - Thread 1 原先是 BLOCKED，但finished參數為False
    - Thread 1 原先就是 Ready狀態 (比如說RR排班，被SWITCH回來)

for loop in Machine::Run()

1. Test Program基本上都是在UserMode下執行的
2. 用一個無窮迴圈反覆抓取User Program的程式碼並Decode (透過 OneInstruction)，然後用OneTick來模擬每個Clock的執行

PartII-2 Implement

kernel.h

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

新增threadPriority變數，用來記錄每個thread的priority，因為要配合thread數量，故與變數t同為大小是10的array。

kernel.cc (http://kernel.cc)

```
} else if (strcmp(argv[i], "-e") == 0) {
  execfile[++execfileNum] = argv[++i];
  threadPriority[execfileNum] = 0; // default
  cout << execfile[execfileNum] << "\n";
} else if (strcmp(argv[i], "-ep") == 0) {
  execfile[++execfileNum] = argv[++i];
  threadPriority[execfileNum] = atoi(argv[++i]); // init
```

這邊的更動是要將thread初始的priority存入threadPriority中。依照"-e"的方式新增"-ep"的條件並加對應的priority存入。

thread.h

```
// add var
double ti;
double last_ti; // t_i-1
int T;          // True ticks
int priority;
int CPU_start_time;
int CPU_end_time;
int ready_queue_wait_time; // total time in ready queue
int enter_ready_time;
// add fn
int InWhichQueue(); // return this thread in which level of ready queue
void set_start_wait_time(int time) { this->enter_ready_time = time; }
// int get_wait_time() { return this->ready_queue_wait_time; }
void record_start_time(int cpu_start_time) {
  this->CPU_start_time = cpu_start_time;
}
void update_ti(int cpu_end_time);
```

thread.cc (http://thread.cc)

```
int Thread::InWhichQueue() {
  if (this->priority >= 100)
    return 1;
  if (this->priority >= 50)
    return 2;
  return 3;
}
```

InWhichQueue是判斷當時該thread屬於哪一個queue。根據spec 2-1-c，可以知道當priority在100～149屬於L1，50～99屬於L2，0～49屬於L3。

```
void Thread::update_ti(int cpu_end_time) {
  this->CPU_end_time = cpu_end_time;
  this->T += this->CPU_end_time - this->CPU_start_time;
  this->ti = 0.5 * (this->T) + 0.5 * (this->last_ti);

  DEBUG(dbgZ, "[D]  Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << this->ID
                << "} ] update approximate burst time, from: [ {"
                << this->last_ti << "}], add [ {" << this->T
                << "} ], to [ {" << this->ti << "}]");

  this->last_ti = this->ti;
  this->T = 0;
}
```

update_ti是配合spec 2-1-d所需要的function。根據spec的公式得到ti(burst time)

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

  ASSERT(this == kernel->currentThread);

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

  kernel->scheduler->Aging();
  kernel->scheduler->ReArrangeThreads();

  if (kernel->currentThread->InWhichQueue() == 3) {
    nextThread = kernel->scheduler->Scheduling();
    if (nextThread != nullptr) {
      kernel->scheduler->AddToQueue(this, this->priority);
      kernel->scheduler->Run(nextThread, FALSE);
    }
  }

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

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

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

  DEBUG(dbgThread, "Sleeping thread: " << name);
  DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: "
                    << name << ", " << kernel->stats->totalTicks);

  status = BLOCKED;
  // cout << "debug Thread::Sleep " << name << "wait for Idle\n";

  if (!finishing) {
    this->update_ti(kernel->stats->totalTicks);
  }

  while ((nextThread = kernel->scheduler->Scheduling()) == NULL) {
    kernel->interrupt->Idle(); // no one to run, wait for an interrupt
  }

  // 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);
}
```

執行sleep function代表即將要進入WAIT state(finish==false)，這時候要更新burst time(ti)，因此呼叫update_ti。
另外，原本nextThread是根據FindNextToRun()，現在改為根據我們去排程的Scheduling()去尋找下一個要做的thread，同樣如果沒有nextThread就執行Idle()。

scheduler.h

```
class Scheduler {
public:
  Scheduler();  // Initialize list of ready threads
  ~Scheduler(); // De-allocate ready list

  void ReadyToRun(Thread *thread);
  // Thread can be dispatched.
  Thread *FindNextToRun(); // Dequeue first thread on the ready
                           // list, if any, and return thread.
  void Run(Thread *nextThread, bool finishing);
  // Cause nextThread to start running
  void CheckToBeDestroyed(); // Check if thread that had been
                             // running needs to be deleted
  void Print();              // Print contents of ready list

  // add fn
  void Aging();
  Thread *Scheduling();
  void AddToQueue(Thread *thread, int priority);
  void ReArrangeThreads();
  void CheckPreempt(Thread *thread);

  // SelfTest for scheduler is implemented in class Thread

private:
  List<Thread *> *readyList; // queue of threads that are ready to run,
                             // but not running
  Thread *toBeDestroyed; // finishing thread to be destroyed
                         // by the next thread that runs

  // add var
  SortedList<Thread *> *L1;
  SortedList<Thread *> *L2;
  List<Thread *> *L3;
};
```

增加L1, L2, L3 Queue，並且L1, L2為SortedList，L3為一般的List。另外也增加一些function，會在之後詳細解釋。

scheduler.cc (http://scheduler.cc)

```
static int L1_compare(Thread *x, Thread *y) {
  if (x->ti < y->ti)
    return -1;
  if (x->ti > y->ti)
    return 1;
  if (x->ti == y->ti)
    return 0;
}
//----------------------------------------------------------------------
// L2_compare
//----------------------------------------------------------------------
static int L2_compare(Thread *x, Thread *y) {
  if (x->priority < y->priority)
    return 1;
  if (x->priority > y->priority)
    return -1;
  return 0;
}
```

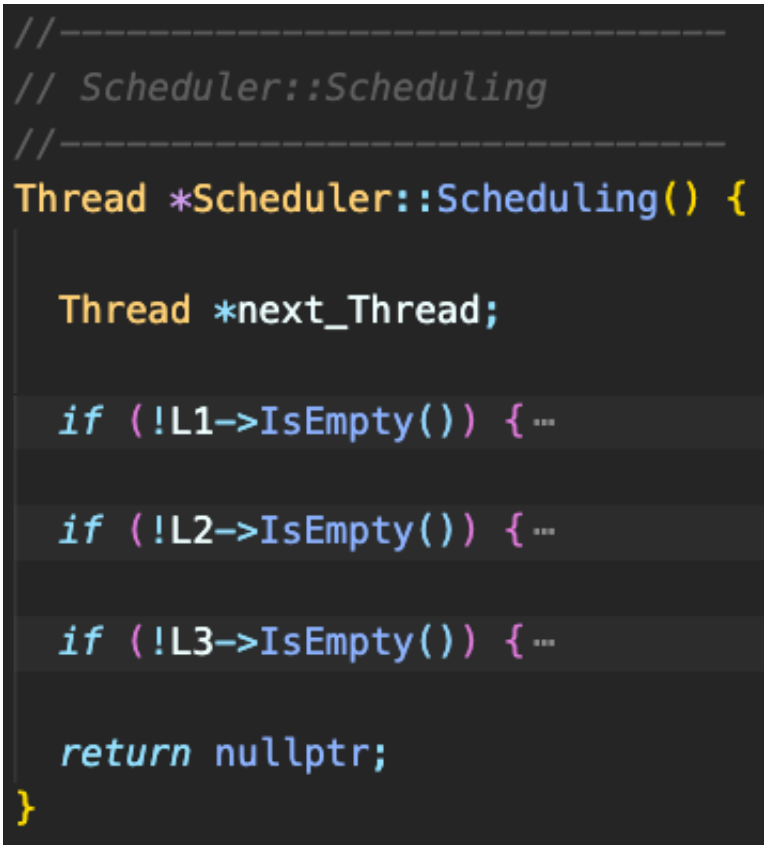
這邊是L1及L2的排序規則。在L1的部分是根據burst time，L2則是根據priority。

```
Scheduler::Scheduler() {
  readyList = new List<Thread *>;
  toBeDestroyed = NULL;

  L1 = new SortedList<Thread *>(L1_compare);
  L2 = new SortedList<Thread *>(L2_compare);
  L3 = new List<Thread *>;
}
```

初始化L1, L2, L3。

```
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);
  thread->set_start_wait_time(kernel->stats->totalTicks);

  if (thread->priority >= 100) {
    // thread->setStatus(READY);
    L1->Insert(thread);
    DEBUG(dbgZ, "[A] Tick [ {"
                    << kernel->stats->totalTicks << "} ]: Thread [ {"
                    << thread->getID()
                    << "} ] is inserted into queueL[ {1} ]");
  } else if (thread->priority >= 50) {
    // thread->setStatus(READY);
    L2->Insert(thread);
    DEBUG(dbgZ, "[A] Tick [ {"
                    << kernel->stats->totalTicks << "} ]: Thread [ {"
                    << thread->getID()
                    << "} ] is inserted into queueL[ {2} ]");
  } else {
    // thread->setStatus(READY);
    L3->Append(thread);
    DEBUG(dbgZ, "[A] Tick [ {"
                    << kernel->stats->totalTicks << "} ]: Thread [ {"
                    << thread->getID()
                    << "} ] is inserted into queueL[ {3} ]");
  }
}
```

根據priority的規則，將當前的thread加入應去的Queue之中。



```
//---------------------------------------
// Scheduler::Scheduling
//---------------------------------------
Thread *Scheduler::Scheduling() {

  Thread *next_Thread;

  if (!L1->IsEmpty()) {…

  if (!L2->IsEmpty()) {…

  if (!L3->IsEmpty()) {…

  return nullptr;
}
```

```
if (!L1->IsEmpty()) {
  next_Thread = L1->RemoveFront();
  next_Thread->record_start_time(kernel->stats->totalTicks);
  DEBUG(dbgZ, "[B] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is removed from queue L[ {1} ]");
  DEBUG(dbgZ, "[E] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is now selected for execution, thread [ {"
                << kernel->currentThread->getID()
                << "} ] is replaced, and it has executed [ {"
                << kernel->stats->totalTicks -
                     kernel->currentThread->CPU_start_time
                << "} ] ticks ");
  return next_Thread;
}

if (!L2->IsEmpty()) {
  next_Thread = L2->RemoveFront();
  next_Thread->record_start_time(kernel->stats->totalTicks);
  DEBUG(dbgZ, "[B] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is removed from queue L[ {2} ]");
  DEBUG(dbgZ, "[E] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is now selected for execution, thread [ {"
                << kernel->currentThread->getID()
                << "} ] is replaced, and it has executed [ {"
                << kernel->stats->totalTicks -
                     kernel->currentThread->CPU_start_time
                << "} ] ticks ");
  return next_Thread;
}
```

```
if (!L3->IsEmpty()) {
  next_Thread = L3->RemoveFront();
  next_Thread->record_start_time(kernel->stats->totalTicks);
  DEBUG(dbgZ, "[B] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is removed from queue L[ {3} ]");
  DEBUG(dbgZ, "[E] Tick [ {"
                << kernel->stats->totalTicks << "} ]: Thread [ {"
                << next_Thread->getID()
                << "} ] is now selected for execution, thread [ {"
                << kernel->currentThread->getID()
                << "} ] is replaced, and it has executed [ {"
                << kernel->stats->totalTicks -
                     kernel->currentThread->CPU_start_time
                << "} ] ticks ");
  return next_Thread;
}
```

Scheduling的規則是從L1 Queue開始查看，有東西就拿出來當作nextThread，沒有就去找L2 Queue；同樣L2 Queue如果沒東西就找L3 Queue。

```
void Scheduler::Aging() {
  Thread *thread;
  int totalTicks = kernel->stats->totalTicks;

  if (!L1->IsEmpty()) {…

  if (!L2->IsEmpty()) {…

  if (!L3->IsEmpty()) {…
}
```

```
if (!L1->IsEmpty()) {
  ListIterator<Thread *> *iterator;
  iterator = new ListIterator<Thread *>(L1);
  while (!iterator->IsDone()) {
    thread = iterator->Item();
    thread->ready_queue_wait_time += totalTicks - thread->enter_ready_time;
    thread->enter_ready_time = totalTicks;

    if (thread->ready_queue_wait_time >= 1500) {
      if (thread->priority + 10 >= 149) {
        DEBUG(dbgZ, "[C] Tick [ {"
                    << kernel->stats->totalTicks
                    << "} ]: Thread [ {"
                    << iterator->Item()->getID()
                    << "} ] changes its priority from [ {"
                    << iterator->Item()->priority
                    << "} ] to [ {149} ]");
        thread->priority = 149;
      } else {
        DEBUG(dbgZ, "[C] Tick [ {"
                    << kernel->stats->totalTicks
                    << "} ]: Thread [ {"
                    << iterator->Item()->getID()
                    << "} ] changes its priority from [ {"
                    << iterator->Item()->priority << "} ] to [ {"
                    << iterator->Item()->priority + 10 << "} ]");
        thread->priority += 10;
      }
      thread->ready_queue_wait_time -= 1500;
    }
    iterator->Next();
  }
}
```

```
if (!L2->IsEmpty()) {
  ListIterator<Thread *> *iterator;
  iterator = new ListIterator<Thread *>(L2);
  while (!iterator->IsDone()) {
    thread = iterator->Item();
    thread->ready_queue_wait_time += totalTicks - thread->enter_ready_time;
    thread->enter_ready_time = totalTicks;

    if (thread->ready_queue_wait_time >= 1500) {
      DEBUG(dbgZ, "[C] Tick [ {"
                  << kernel->stats->totalTicks << " } ]: Thread [ {"
                  << iterator->Item()->getID()
                  << " } ] changes its priority from [ {"
                  << iterator->Item()->priority << " } ] to [ {"
                  << iterator->Item()->priority + 10 << " } ]");
      thread->priority += 10;
      thread->ready_queue_wait_time -= 1500;
    }
    iterator->Next();
  }
}
if (!L3->IsEmpty()) {
  ListIterator<Thread *> *iterator;
  iterator = new ListIterator<Thread *>(L3);
  while (!iterator->IsDone()) {
    thread = iterator->Item();
    thread->ready_queue_wait_time += totalTicks - thread->enter_ready_time;
    thread->enter_ready_time = totalTicks;

    if (thread->ready_queue_wait_time >= 1500) {
      thread->priority += 10;
      thread->ready_queue_wait_time -= 1500;
    }
    iterator->Next();
  }
}
```

Aging做的事情是更動每個thread的priority。根據spec 2-1-g，當waiting time超過1500ticks就要進行Priority update。因此我們要看過每個thread，根據每個thread去更新priority。

首先是L1的部分，因為priority上限是149，因此如果priority update後（+10）超過149，就維持priority = 149，否則就將priority+10。

L2, L3則不需要考慮priority上限的問題，只要waiting time超過1500ticks就將

priority+10。
最後，剛剛那些有超過1500ticks的，要將waiting time-1500，才能再下一個1500 ticks再次更新priority。

```
void Scheduler::ReArrangeThreads() {
  Thread *move_thread;
  ListIterator<Thread *> *iter3; // L3
  iter3 = new ListIterator<Thread *>(L3);
  while (!iter3->IsDone()) {…

  ListIterator<Thread *> *iter2; // L2
  iter3 = new ListIterator<Thread *>(L2);
  while (!iter2->IsDone()) {…
}
```

```
while (!iter3->IsDone()) {
  move_thread = L3->RemoveFront();
  DEBUG(dbgZ, "[B] Tick [{"
                << kernel->stats->totalTicks << "}]: Thread [{"
                << move_thread->getID()
                << "}] is removed from queue L[{3}]");
  if (move_thread->priority >= 100) {
    L1->Insert(move_thread);
    DEBUG(dbgZ, "[A] Tick [{"
                  << kernel->stats->totalTicks << "}]: Thread [{"
                  << move_thread->getID()
                  << "}] is inserted into queue L[{1}]");
    this->CheckPreempt(move_thread);
  } else if (move_thread->priority >= 50) {
    L2->Insert(move_thread);
    DEBUG(dbgZ, "[A] Tick [{"
                  << kernel->stats->totalTicks << "}]: Thread [{"
                  << move_thread->getID()
                  << "}] is inserted into queue L[{2}]");
    this->CheckPreempt(move_thread);
  } else {
    L3->Append(move_thread);
    DEBUG(dbgZ, "[A] Tick [{"
                  << kernel->stats->totalTicks << "}]: Thread [{"
                  << move_thread->getID()
                  << "}] is inserted into queue L[{3}]");
  }

  iter3->Next();
}
```

```
while (!iter2->IsDone()) {
  if (iter2->Item()->priority >= 100) {
    move_thread = L2->RemoveFront();
    DEBUG(dbgZ, "[B] Tick [ {"
                  << kernel->stats->totalTicks << "} ]: Thread [ {"
                  << move_thread->getID()
                  << "} ] is removed from queue L[ {2} ]");
    L1->Insert(move_thread);
    DEBUG(dbgZ, "[A] Tick [ {"
                  << kernel->stats->totalTicks << "} ]: Thread [ {"
                  << move_thread->getID()
                  << "} ] is inserted into queue L[ {1} ]");
    this->CheckPreempt(move_thread);
  }
  iter2->Next();
}
```

ReArrange做的事情是將更新Priority後的thread，如果不在屬於該Queue者，移動至符合的Queue。

當更新後的Priority超過49，就要移出L3的thread，根據新的Priority移到L2 or L1。
同理，超過99的thread則要從L2移動到L1。

```
void Scheduler::CheckPreempt(Thread *thread) {
  // current thread is L3 and L1/L2 has thread added.
  if (kernel->currentThread->InWhichQueue() == 3 &&
      (thread->InWhichQueue() == 2 || thread->InWhichQueue() == 1)) {
    kernel->currentThread->T +=
        kernel->stats->totalTicks - kernel->currentThread->CPU_start_time;
    Thread *nextThread = this->Scheduling();
    // this->AddToQueue(kernel->currentThread, kernel->currentThread->priority);
    this->ReadyToRun(kernel->currentThread);
    this->Run(nextThread, FALSE);
  } else if (kernel->currentThread->InWhichQueue() == 2 &&
             thread->InWhichQueue() == 1) {
    kernel->currentThread->T +=
        kernel->stats->totalTicks - kernel->currentThread->CPU_start_time;
    Thread *nextThread = this->Scheduling();
    // this->AddToQueue(kernel->currentThread, kernel->currentThread->priority);
    this->ReadyToRun(kernel->currentThread);
    this->Run(nextThread, FALSE);
  } else if (kernel->currentThread->InWhichQueue() == 1 &&
             (thread->InWhichQueue() == 1 &&
              thread->ti < kernel->currentThread->ti)) {
    kernel->currentThread->T +=
        kernel->stats->totalTicks - kernel->currentThread->CPU_start_time;
    Thread *nextThread = this->Scheduling();
    // this->AddToQueue(kernel->currentThread, kernel->currentThread->priority);
    this->ReadyToRun(kernel->currentThread);
    this->Run(nextThread, FALSE);
  }
}
```

最後是check preempt的部分。check preempt有三種情形分別是：
1.current thread是L3，這時L1/L2有thread加入
2.current thread是L2，且L1有thread加入
3.current thread是L1，且L1有更小burst time的thread傳入
而內部做的事情是更新T、找到nextThread、把current thread放入該去的Queue、switch to nextThread。

Scheduler::Run()

```
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".

nextThread->ready_queue_wait_time = 0;

SWITCH(oldThread, nextThread);
```

set waiting time to 0