MP3_report_58

Team Members & Contributions

- 108062119 鄭幃謙
- 108062133 劉廷宜

Work	Contributions
trace code	鄭幃謙、劉廷宜
implementation	鄭幃謙、劉廷宜
debug	鄭幃謙
report	劉廷宜

Trace Code

1-1. New → Ready

過程

1. main.cc創立kernel → main thread也隨之創出並設為RUNNING
2. main.cc呼叫ExecAll()將所有exeFile丟入Exec()執行
3. Exec()會給exeFile創建新的TCB、配置AddrSpace，上述完成後呼叫Fork()
4. Fork()會呼叫StackAllocate()把&Forkexecute跟t[threadNum]配置給Stack並初始化
5. 接下來Fork()會呼叫ReadyToRun()將t[threadNum]設為READY並放到readyList中

> t[threadNum]到時候從readyList拿出來時
> stack中放置的&ForkExecute會幫忙把t[threadNum]所需的東西
> （例如code segment、data segment）
> 搬移到memory，讓t[threadNum]可以順利運行
> ＆ForkExecute最後會呼叫AddrSpace::Execute()
> Execute()會呼叫Machine::Run()以執行t[threadNum]

- **threads/main.cc**
  創建一個kernel，並對其初始化

```
// global variables
Kernel *kernel;
int main(int argc, char **argv){
    kernel = new Kernel(argc, argv);
    kernel->Initialize();
    ...
    kernel->ExecAll();
}
```

- **threads/kernel.cc/Kernel::Kernel()**
  在constructor會先設定一些參數如randomSlice等
  但主要是處理command line instruction
  當argv[i]為-e時，代表argv[i+1]為exe檔
  將argv[i+1]放入execfile陣列中、更新execfileNum

```
for (int i = 1; i < argc; i++) {
    ...
    else if (strcmp(argv[i], "-e") == 0) {
        execfile[++execfileNum] = argv[++i];
        cout << execfile[execfileNum] << "\n";
    }
    ...
}
```

- **threads/kernel.cc/Kernel::Initialze()**
  currentThread存在於Kernel object中
  創建名為main的thread，並將其assign給currentThread
  更新threadNum並將currentThread的狀態設為RUNNING

```
void Kernel::Initialize() {
    ...
    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);
    ...
}
```

- **threads/kernel.cc/Kernel::ExecAll()**
  在constructor將所有command line instructions處理完後
  execfile[]和execfileNum都已更新完畢
  threads/main.cc會呼叫ExecAll()
  ExecAll()會將所有execfile[]的檔案丟進Exec()
  當所有execfile都執行完後
  currentThread(main thread)就會結束

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

  - **threads/threads.cc/Thread::Finish()**
    因為等等會呼叫Sleep()

而Sleep()假定Interrupt是disabled的
所以先將Interrupt disabled
再來利用ASSERT確認現在這個thread是否為currentThread
呼叫Sleep(TRUE)將currentThread給destroy

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

    Sleep(TRUE);
}
```

- **threads/threads.cc/Thread::Sleep()**
  傳入的bool finishing為TRUE
  利用ASSERT確認這個thread是否為currentThread和Interrupt是否為disabled
  status = BLOCKED;
  將在等待的thread的status改為BLOCKED
  kernel->scheduler->Run(nextThread, finishing);
  使用while迴圈從readyList找到nextThread
  ・若無：呼叫Idle()，裡面會判斷要advance clock或是halt
  ・若有：呼叫Run()來執行nextThread

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

- **machine/interrupt.cc/Interrupt::Idle()**
  透過CheckIfDue()檢查是否有任何pending interrupt
  ・若有：回到scheduler
  ・若無：呼叫Halt()來停止machine

```
void Interrupt::Idle()
{
    status = IdleMode;
```

```
        if (CheckIfDue(TRUE)) {
            status = SystemMode;
            return;
        }
        Halt();
    }
```

- **threads/kernel.cc/Kernel::Exec()**
  t[threadNum] = new Thread(name, threadNum);
  給Exefile創建新的Thread object（類似TCB）
  並將其存於Thread *t[]中
  t[threadNum]->space = new AddrSpace();
  簡單配置一個AddrSpace給剛創建的t[threadNum]
  page table的創建在MP2時搬到Load()
  t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void)t[threadNum])
  透過Fork()可以完成stack的配置跟初始化
  將t[threadNum]傳入ForkExecute()執行
  ForkExecute()中會呼叫Load()將t[threadNum]所需的東西搬移到memory

```
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;
}
```

- **threads/thread.cc/Thread::Fork(VoidFunctionPtr, void)**
  宣告Interrupt及Scheduler指標來使用NachOS的module
  呼叫**StackAllocate()**幫thread建立stack和設定MachineState
  將Interrupt disabled後呼叫scheduler將thread放進readyList
  （因為ReadyToRun()假定Interrupt是disabled的）
  完成後再將Interrupt設為原本的status

```
void Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
```

```
    (void) interrupt->SetLevel(oldLevel);
}
```

- **threads/kernel.cc/Kernel::ForkExecute(VoidFunctionPtr, void)**
  將thread *t會使用到的code segment跟data segment利用`Load()`放入memory
  若Load()失敗就會回傳FALSE：ForkExecute() return
  若Load()成功就會呼叫Execute：執行Thread *t

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

- **userprog/addrspace.cc/AddrSpace::Execute()**
  將currentThread的位置指向這個thread的address space
  初始化register跟設定page table register後
  呼叫`machine->Run()`來執行這個thread

```
void AddrSpace::Execute(char *fileName) {
    kernel->currentThread->space = this;

    this->InitRegisters();
    this->RestoreState();

    kernel->machine->Run();

    ASSERTNOTREACHED();
}
```

- **machine/interrupt.cc/Interrupt::SetLevel()**
  將原本Interrupt的status記錄下來
  不產生錯誤的話有三種情況
  (1) now == IntOff / inHandler == FALSE
  這代表interrupt handler沒有在運行，我們之後要把它disabled
  (2) now == IntOff / inHandler == TRUE
  這代表interrupt handler在運行，我們之後要把它disabled
  (3) now == IntOn / inHandler == FALSE
  這代表interrupt handler沒有在運行，我們之後要把它enabled
  檢查完有無錯誤後
  將Interrupt的status由old改成now
  如果Interrupt是在enabled的狀態就呼叫OneTick()推進時間
  之後return原來Interrupt的status

```
IntStatus Interrupt::SetLevel(IntStatus now)
{
    IntStatus old = level;

    ASSERT((now == IntOff) || (inHandler == FALSE));

    ChangeLevel(old, now);
    if ((now == IntOn) && (old == IntOff)) {
        OneTick();
    }
    return old;
}
```

- **machine/interrupt.cc/Interrupt::ChangeLevel()**
  改變Interrupt的status
  由old改成now

```
void Interrupt::ChangeLevel(IntStatus old, IntStatus now)
{
    level = now;
}
```

- **Thread::StackAllocate(VoidFunctionPtr, void)**
  StackAllocate()會幫傳進來的func跟arg配置跟初始化stack
  （下面的code只放了x86的部分做說明）
  stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
  使用AllocBoundedArray()allocate一個array作為stack
  stack會安排在兩個page中間以管理thread stack overflow進而達成保護
  stack pointer指向這個array的位置(low address)
  stackTop = stack + StackSize - 4;
  讓StackTop指向stack的底部(high address)
  確保安全會再減四
  ***(--stackTop) = (int) ThreadRoot;**
  讓stack的第一個元素設為ThreadRoot的address
  這樣在SWITCH時可以直接從stack取出ThreadRoot的位址來執行
  ***stack = STACK_FENCEPOST;**
  將STACK_FENCEPOST放在stack的頂部
  主要是偵測stack有無overfolw
  最後是設定MachineState的部分
  由於NachOS是跑在Host上的VM
  MachineState可以視為NachOS給Host使用的register

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

```
    ...
    #ifdef x86
        stackTop = stack + StackSize - 4;
        *(--stackTop) = (int) ThreadRoot;
        *stack = STACK_FENCEPOST;
    #endif
    #else
        machineState[PCState] = (void*)ThreadRoot;
        machineState[StartupPCState] = (void*)ThreadBegin;
        machineState[InitialPCState] = (void*)func;
        machineState[InitialArgState] = (void*)arg;
        machineState[WhenDonePCState] = (void*)ThreadFinish;
    #endif
 }
```

- **threads/scheduler.cc/Scheduler::ReadyToRun()**
  先確認Interrupt是否有disabled以免出現Interrupt打斷的情況
  將thread的state設為READY
  將其放入readyList中

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

1-2. Running → Ready

> thread由RUNNING到READY的狀況：
> interrupt occurs (e.g. timer, process with higher priority)

- **machine/mipssim.cc/Machine::Run()**
  Machine::Run()會在**AddrSpace::Execute()**被呼叫
  會先創建新的Instruction object來裝新的instruction
  將系統設為UserMode
  反覆執行**OneInstruction()**對instruction decode和利用**OneTick()**模擬每個clock的執行

```
void Machine::Run()
{
    Instruction *instr = new 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();
    }
}
```

- **machine/mipssim.cc/Machine::OneInstruction()**
  會在裡面對instruction進行decode()
  再對照instruction的opCode進行相應的處理

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

- **Instruction class**

  ```
  class Instruction {
      public:
          void Decode();
          unsigned int value;
          char opCode;
          char rs, rt, rd;
          int extra;
  };
  ```

- **machine/interrupt.cc/Interrupt::OneTick()**
  (1) 判斷是在SystemMode還是UserMode增加totalTicks跟對應的執行Ticks
  (2) 將Interrupt disabled達成不被interrupt打斷(atomic)
  CheckIfDue()會檢查是否有到期的pending interrupt
  檢查完再把interrupt enabled
  (3) yieldOnReturn代表我們是不是要做context switch
  若為TRUE則先將yieldOnReturn設回FALSE
  如果不這樣做的話之後的thread看到yieldOnReturn為TRUE會有錯誤
  切換到SystemMode並對currentThread(main thread)呼叫Yield()
  完成後將status換回原本的mode（通常是UserMode）
  再回到Machine::Run()的無窮迴圈中被呼叫

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

```
    // (1)
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
    } else {
    stats->totalTicks += UserTick;
    stats->userTicks += UserTick;
    }

    // (2)
    ChangeLevel(IntOn, IntOff);
    CheckIfDue(FALSE);
    ChangeLevel(IntOff, IntOn);

    // (3)
    if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

- **machine/interrupt.cc/Interrupt::CheckIfDue()**
  next = pending->RemoveFront();
  next即為OneTick()中的yieldOnReturn
  在這邊會將yieldOnReturn設為TRUE

  ```
  inHandler = TRUE;
  do {
      next = pending->RemoveFront();
      next->callOnInterrupt->CallBack();
  } while (!pending->IsEmpty()
          && (pending->Front()->when <= stats->totalTicks));
  ```

- **threads/thread.cc/Thread::Yield()**
  YieldOnReturn為TRUE就代表要進行context switch
  切換thread、釋出CPU資源
  IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
  因為希望switch到readyList front的thread時為atomic
  所以先將Interrupt設為disabled
  nextThread = kernel->scheduler->FindNextToRun();
  scheduler呼叫FindNextToRun()來找到nextThread
  若nextThread不為NULL
  就將現在正在運行的thread利用ReadyToRun()放到ReadyList的尾巴
  呼叫Run()來運行得到的nextThread
  (void) kernel->interrupt->SetLevel(oldLevel);
  上述都完成後再將Interrupt設回原本的level（通常為enabled）

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

    ASSERT(this == kernel->currentThread);

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

- **threads/scheduler.cc/Scheduler::FindNextToRun()**
  檢查readyList是否為空
  ・是：回傳NULL
  ・否：dequeue得到nextThread並回傳

- **threads/scheduler.cc/Scheduler::Run()**
  在**Yield()**中呼叫kernel->scheduler->Run(nextThread, FALSE)
  將oldThread指向currentThread
  確保Interrupt disabled再進行以下操作
  (1) finishing為FALSE
  代表oldThread還沒執行完，不需要destroy
  (2) 若oldThread->space != NULL
  將oldThread的UserState保存於TCB
  接著保存oldThread address space的state
  (3) 檢查oldThread的stack是否overflow
  (4) 將kernel執行的currentThread改為nextThread
  並把nextThread的status設為RUNNING
  完成上述後呼叫SWITCH()進行context switch
  (5) 呼叫CheckToBeDestroyed()檢查是否有thread要被刪掉
  (6) 將oldThread的userRegister和addreess space的page table都回復原狀

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

    // (1)
    if (finishing) {
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
    }

    // (2)
```

```
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    // (3)
    oldThread->CheckOverflow();

    // (4)
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);

    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // (5)
    CheckToBeDestroyed();

    // (6)
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

- **threads/thread.cc/Thread::SaveUserState()**
  在context switch時
  會把現在運行CPU的user program的CPU state給存起來
  （在這邊是儲存運行user code部分的register）
  int userRegisters[NumTotalRegs]; // user-level CPU register state
  定義在thread class中

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

- **userprog/addrspace.cc/AddrSpace::SaveState()**

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

- **threads/thread.cc/Thread::CheckOverflow()**
  用stack的頂部是否為STACK_FENCEPOST來檢查是否產生stack overflow

```
void Thread::CheckOverflow()
{
    if (stack != NULL) {
        ASSERT(*stack == STACK_FENCEPOST);
    }
}
```

- **threads/scheduler.cc/Scheduler::CheckToBeDestroyed()** toBeDestroyed是之前設定指向要delete的thread
  （**Scheduler::Run()**中finishing為TRUE時會指向oldThread）
  若toBeDestroyed != NULL：delete toBeDestroyed（刪掉完成的thread） 完成後將toBeDestroyed設回NULL

  > 在NachOS中 當一個thread執行完成後它並不能自己把自己刪掉
  > （因為自己還在跑）
  > 所以需要下一個要執行的thread來將自己刪除

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

- **threads/thread.cc/Thread::RestoreUserState()**
  將userRegister一一寫回CPU register中

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

- **userprog/addrspace.cc/AddrSpace::RestoreState()**

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

1-3. Running → Waiting

> thread由RUNNING到WAITING的狀況：
> interrupt occurs (e.g. I/O)

- userprog/synchconsole.cc/SynchConsoleOutput::PutChar()
  lock->Acquire();
  由於不能兩個thread同時在做output(I/O)
  所以呼叫lock->Acquire()
  讓這個thread能擁有lock
  consoleOutput->PutChar(ch);
  將字元輸出
  waitFor->P();
  代表需要資源，如果沒有資源的話就會困在P()中的迴圈直到資源被釋出
  SynchConsoleOutput::CallBack()中會呼叫waitFor->V()來釋出資源使waitFor->P()順利執行
  lock->Release();
  全部字元都成功輸出後
  呼叫lock->Realease()來釋放lock

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

  - class Lock
    可以從Lock的定義看出Lock的實作是由Semaphore完成
    lock->lockHolder代表現在持有lock的thread

    ```
    class Lock {
      public:
        Lock(char* debugName);      // initialize lock to be FREE
        ~Lock();            // deallocate lock
        char* getName() { return name; }

        void Acquire();
        void Release();

        bool IsHeldByCurrentThread() {
                return lockHolder == kernel->currentThread; }

      private:
        char *name;
        Thread *lockHolder;
        Semaphore *semaphore;
    };
    ```

- **Semaphore**
  (1) A tool to generalize the synchronization problem -> A record of how many units of a particular resource are available
  (2) accessed only through 2 atomic ops: wait & signal

- **class Semaphore**

```
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    char* getName() { return name;}

    void P(); // wait
    void V(); // signal
    void SelfTest();

  private:
    char* name;
    int value;
    List<Thread *> *queue;
   };
```

- **threads/synch.cc/Lock::Aquire()**
  呼叫semaphore->P()來完成取得lock的動作
  若是沒有資源，就會一直被困在P()的while迴圈直到資源被釋出
  若是取得lock成功，就會將lockHolder改為kernel->currentThread

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

- **machine/console.cc/ConsoleOutput::PutChar()**
  ASSERT(putBusy == FALSE);
  putBusy代表現在是否有putChar在進行
  因為一次只能一個threadu進行putChar的動作
  所以若putBusy == TRUE就會報錯
  （已經有別的thread在輸出了這個thread現在不能去輸出）
  WriteFile(writeFileNo, &ch, sizeof(char));

  > writeFileNo初始化時就已經設定為1，所以一次只能輸出一個字元

  利用WriteFile()將字元輸出
  putBusy = TRUE;
  將putBusy設為TRUE表示正在putChar

以免別的thread也要putChar而產生錯誤
kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt); 將ConsoleOutput放進pending interrupt list
在這邊ConsoleTime為1 tick
所以當1 tick過去，console write interrupt就會發生
進而執行ConsoleOutput->CallBack()

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

- **machine/interrupt.cc/Interrupt::Schedule()**
  將之後要被call back的object和它要被執行的時間放到pending interrupt list中

```
void Interrupt::Schedule(CallBackObj *toCall, int fromNow,
IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall,
when, type);

    ASSERT(fromNow > 0);

    pending->Insert(toOccur);
}
```

- **machine/console.cc/ConsoleOutput::CallBack()**
  會更新kernel->stats->numConsoleCharsWritten
  呼叫**callWhenDone->CallBack()**

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

- **userprog/synchconsole.cc/SynchConsoleOutput::CallBack()**
  waitFor是在class SynchConsoleInput中的Semaphore
  主要功能就是在等待CallBack

在SynchConsoleOutput::CallBack()中只呼叫了**waitFor->V()**
會釋放資源讓資源數量加一

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

- **threads/synch.cc/Semaphore:: P()**
  List<Thread*> *queue定義在class Semaphore中
  用來儲存呼叫P()而等待資源釋出的thread
  IntStatus oldLevel = interrupt->SetLevel(IntOff);
  先將Interrupt disabled來符合P()為atomic的條件
  while (value == 0)
  若value == 0：代表現在沒有資源，將這個thread放到queue並put to sleep
  若value != 0：代表有資源，將資源給這個thread使用並將資源數量減一
  (void) interrupt->SetLevel(oldLevel);
  將interrupt設回原本的狀態（通常為enabled）

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

    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {
        queue->Append(currentThread);
        currentThread->Sleep(FALSE);
    }
    value--;

    (void) interrupt->SetLevel(oldLevel);
}
```

- **lib/list.cc/List::Append(T)**
  就是實作single linked list

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

    ASSERT(!IsInList(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));
    }
```

- **threads/thread.cc/Thread::Sleep()**
  是在**Semaphore:: P()**中當value == 0時被呼叫
  傳入的bool finishing為FALSE
  利用ASSERT確認這個thread是否為currentThread和
  Interrupt是否為disabled
  status = BLOCKED;
  將在等待的thread的status改為BLOCKED
  kernel->scheduler->Run(nextThread, finishing);
  使用while迴圈從readyList找到nextThread
  ・若無：呼叫Idle()，裡面會判斷要advance clock或是halt
  ・若有：呼叫Run()來執行nextThread

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

- **threads/scheduler.cc/Scheduler::FindNextToRun()**
  檢查readyList是否為空
  ・是：回傳NULL
  ・否：dequeue得到nextThread並回傳

- **threads/scheduler.cc/Scheduler::Run()**
  將oldThread指向currentThread
  確保Interrupt disabled再進行以下操作
  (1) finishing為FALSE
  代表oldThread還沒執行完，不需要destroy
  (2) 若oldThread->space != NULL

將oldThread的UserState保存於TCB
接著保存oldThread address space的state
(3) 檢查oldThread的stack是否overflow
(4) 將kernel執行的currentThread改為nextThread
並把nextThread的status設為RUNNING
完成上述後呼叫SWITCH()進行context switch
(5) 呼叫CheckToBeDestroyed()檢查是否有thread要被刪掉
(6) 將oldThread的userRegister和addreess space的page table都回復原狀

1-4. Waiting → Ready

過程

1. SynchConsoleOutput::PutChar()被呼叫
2. ConsoleOutput::PutChar()被呼叫
3. waitFor->P()被呼叫
4. 當PutChar()完成時SunchConsoleOutput::CallBack()被呼叫
5. waitFor->V()被呼叫
6. 資源順利被釋放，waitFor->P()順利執行下去

- **threads/synch.cc/Semaphore::V()**
  if (!queue->IsEmpty())
  如果queue中不為空
  dequeue下一個thread，將status從BLOCKED轉變為READY
  value++;
  釋放資源，將資源的數量加一
  (void) interrupt->SetLevel(oldLevel);
  將Interrupt變為原本的狀態（通常為enabled）

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

    if (!queue->IsEmpty()) {
    kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

- **threads/scheduler.cc/Scheduler::ReadyToRun()**
  先確認Interrupt是否有disabled以免出現Interrupt打斷的情況
  將thread的state設為READY
  將其放入readyList中

1-5. Running → Terminated

> thread由RUNNING到TERMINATED的狀況：
> 代表整個thread的指令都已經執行完成

- **ExceptionHandler(ExceptionType) case SC_Exit**
  kernel->currentThread->Finish();
  結束currentThread

```
case SC_Exit:
    val = kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
```

- **threads/threads.cc/Thread::Finish()**
  因為等等會呼叫Sleep()
  而Sleep()假定Interrupt是disabled的
  所以先將Interrupt disabled
  再來利用ASSERT確認現在這個thread是否為currentThread
  呼叫Sleep(TRUE)將currentThread給destroy

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

    Sleep(TRUE);
}
```

- **threads/thread.cc/Thread::Sleep()**
  是在**Thread::Finish()**被呼叫
  傳入的bool finishing為TRUE
  利用ASSERT確認這個thread是否為currentThread和Interrupt是否為disabled
  status = BLOCKED;
  將在等待的thread的status改為BLOCKED
  kernel->scheduler->Run(nextThread, finishing);
  使用while迴圈從readyList找到nextThread
  ・若無：呼叫Idle()，裡面會判斷要advance clock或是halt
  ・若有：呼叫Run()來執行nextThread

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

- **threads/scheduler.cc/Scheduler::FindNextToRun()**
  檢查readyList是否為空
  ・是：回傳NULL
  ・否：dequeue得到nextThread並回傳

- **threads/scheduler.cc/Scheduler::Run()**
  將oldThread指向currentThread
  確保Interrupt disabled再進行以下操作
  (1) finishing為TRUE
  上一個thread已經執行完
  將toBeDestroyed指向oldThread
  (2) 若oldThread->space != NULL
  將oldThread的UserState保存於TCB
  接著保存oldThread address space的state
  (3) 檢查oldThread的stack是否overflow
  (4) 將kernel執行的currentThread改為nextThread
  並把nextThread的status設為RUNNING
  完成上述後呼叫SWITCH()進行context switch
  (5) 呼叫CheckToBeDestroyed()將oldhread刪掉
  (6) 將oldThread的userRegister和addreess space的page table都回復原狀

1-6. Ready → Running

- **threads/scheduler.cc/Scheduler::FindNextToRun()**
  檢查readyList是否為空
  ・是：回傳NULL
  ・否：dequeue得到nextThread並回傳
- **threads/scheduler.cc/Scheduler::Run()**
  將oldThread指向currentThread
  確保Interrupt disabled再進行以下操作
  finishing有可能為TRUE也可能為FALSE
  (1) finishing判斷上一個thread是否執行完
  若執行完的話就將toBeDestroyed指向oldThread
  (2) 若oldThread->space != NULL
  將oldThread的UserState保存於TCB
  接著保存oldThread address space的state
  (3) 檢查oldThread的stack是否overflow
  (4) 將kernel執行的currentThread改為nextThread
  並把nextThread的status設為RUNNING

完成上述後呼叫SWITCH()進行context switch
(5) 呼叫CheckToBeDestroyed()檢查是否有thread要被刪掉
(6) 將oldThread的userRegister和addreess space的page table都回復原狀

- **SWITCH(Thread*, Thread*)**
  · SWITCH是透過switch.h的輔助和thread.h的外部宣告
  · 實作是在switch.S
  (以下只針對x86架構做說明)

  - **threads/switch.h**
    宣告register的位置
    之後switch.S會使用到

```
#ifdef x86

#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

#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
```

  - **threads/thread.h**
    Scheduler::Run()會呼叫這邊定義的SWITCH
    透過extern的宣告和編譯器的輔助使得x86組合語言可以與c語言互相呼叫

```
extern "C" {
void ThreadRoot();
void SWITCH(Thread *oldThread, Thread *newThread);
}
```

- **threads/switch.S**
  在**StackAllocate()**就已經把未來函式的address放到Host CPU使用的register了
  eax, ebx, ecx, edx, esi, edi, ebp, esp等都是x86組合語言中CPU上的通用暫存器的名稱，是32位的暫存器
  如果用C語言來解釋，可以把這些暫存器當作變數看待
  switch.S中的register value
  - **ecx: points to start function**
    對應到C的(void*)ThreadBegin()
  - **edx: contains initial argument to thread function**
    對應到C的(void*)arg
  - **esi: points to thread function**
    對應到C的(void*)func
    （也就是ForkExecute）
  - **edi: points to Thread::Finish()**
    對應到C的(void*)ThreadFinish()
  - **esp: stores the new PC value**
    對應到C的(void*)ThreadRoot

  ThreadRoot的部分：
  call *StartupPC
  對應Thread::ThreadBegin()
  call *InitialupPC
  對應Kernel::ForkExecute()
  call *WhenDonePC
  對應Thread::ThreadFinish()
  SWITCH的部分：

  > 先處理oldThread t1

  ```
  movl    %eax,_eax_save
  movl    4(%esp),%eax
  ```

  1. 將eax原本的值存起來
  2. 將指向t1的pointer的地址存到eax

  ```
      movl    %ebx,_EBX(%eax)
      movl    %ecx,_ECX(%eax)
      movl    %edx,_EDX(%eax)
      movl    %esi,_ESI(%eax)
      movl    %edi,_EDI(%eax)
      movl    %ebp,_EBP(%eax)
  ```

  1. 將CPU registers的值存回t1的stack

```
    movl    %esp,_ESP(%eax)
```

1. 將stack pointer(esp)存回t1的stack

```
movl    _eax_save,%ebx
movl    %ebx,_EAX(%eax)
```

1. 取出原本存在eax的值（現在存在_eax_save）存到ebx
2. 再藉由ebx將原本存在eax的值存到t1的stack

```
movl    0(%esp),%ebx
movl    %ebx,_PC(%eax)
```

1. 將t1的return address存入ebx
   (這樣下次SWITCH到t1才可以知道要去哪裡繼續執行)
2. 再藉由ebx將return address存到t1的PC storage

> 再處理 newThread t2

```
movl    8(%esp),%eax
movl    _EAX(%eax),%ebx
movl    %ebx,_eax_save
```

1. 將指向t2的pointer的地址存到eax
2. 將t2的stack中eax的值存入ebx
3. 再藉由ebx將t2的stack中eax的值存入_eax_save

```
movl    _EBX(%eax),%ebx
movl    _ECX(%eax),%ecx
movl    _EDX(%eax),%edx
movl    _ESI(%eax),%esi
movl    _EDI(%eax),%edi
movl    _EBP(%eax),%ebp
```

1. 將t2的stack中的資料放到CPU registers中

```
movl    _ESP(%eax),%esp
```

1. 將指向t2的stack pointer存放到esp

```
movl    _PC(%eax),%eax
movl    %eax,4(%esp)
movl    _eax_save,%eax
```

1. 把t2的return address從t2的stack取出放入eax
2. 再藉由eax將t2的return address放入4(%esp)
3. 最後將_eax_save存回eax

```
ret
```

set CPU program counter to the memory address pointed by the value of register esp
也就是把CPU的program counter的值設為t2的return address
完成後就開始t2的執行

- **for loop in Machine::Run()**
  下一個thread（也就是SWITCH過去的t2）開始執行
  將系統設為UserMode
  反覆執行OneInstruction()對instruction decode和利用OneTick()模擬每個clock的執行

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

Implementation

2-1

- **threads/thread.h**
  在class Thread中新增以下東西
  approximateBurstTime、totalBurstTime、 remainingBurstTime、priority 如字面意思
  startTime用來記錄thread開始執行的時間
  execTime紀錄執行時間（用在debug）

ageTime用來算aging的時間
UndateBurstTime(bool)和Aging()之後會詳述

```
public:
    void UpdateBurstTime(bool toReady);
    void Aging(void);
    double approximateBurstTime, totalBurstTime, remainingBurstTime,
startTime, execTime, ageTime;
    int priority;
```

- **threads/thread.cc**
    - **Thread constructor**
      對在thread.h新增的variables初始化

      ```
      approximateBurstTime = 0.0;
      remainingBurstTime = 0.0;
      totalBurstTime = 0.0;
      startTime = 0.0;
      execTime = 0.0;
      ageTime = 0.0;
      priority = 0;
      ```

    - **Thread::Yield()**
      UpdateBurstTime(true)
      true表示之後這個thread進入ready state
      把ReadyToRun()放到FindNextToRun()前面
      以免有priority較低的thread先被執行

      ```
      UpdateBurstTime(true);
      kernel->scheduler->ReadyToRun(this);
      nextThread = kernel->scheduler->FindNextToRun();
      if (nextThread != NULL) {
          kernel->scheduler->Run(nextThread, FALSE);
      }
      ```

    - **Thread::Sleep()**
      呼叫UpdateBurstTime(false)

      ```
      UpdateBurstTime(false);

      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
      {
          kernel->interrupt->Idle();
      ```

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

- **Thread::UpdateBurstTime(bool)**

  execTime = kernel->stats->totalTicks-startTime;
  記錄執行時間
  totalBurstTime += execTime;
  將totalBurstTime加上剛剛算的執行時間
  若toReady == true：代表thread要進入ready
    remainingBurstTime更新為max(0, t_i-T)
  若toReady == false：代表thread沒有要進入ready
    approximateBurstTime = 0.5 * totalBurstTime + 0.5 * approximateBurstTime;
    approximateBurstTime更新為0.5 * T + 0.5 * t_{i-1}
    remainingBurstTime = approximateBurstTime;
    remainingBurstTime更新為0.5 * T + 0.5 * t_{i-1}
    totalBurstTime = 0.0;
    totalBurstTime歸零

```
void
Thread::UpdateBurstTime(bool toReadys) {
    execTime = kernel->stats->totalTicks-startTime;
    totalBurstTime += execTime;
    if (toReady)
        remainingBurstTime = max(0.0, approximateBurstTime -
totalBurstTime);
    else {
        approximateBurstTime = 0.5 * totalBurstTime + 0.5 *
approximateBurstTime;
        remainingBurstTime = approximateBurstTime;
        totalBurstTime = 0.0;
    }
}
```

- **Thread::Aging()**

  將priority增加10（不能超過149）
  ageTime增加1500（做了一次age再重新算）

```
void
Thread::Aging(void) {
    if (priority < 149) {
        priority = min(149, priority + 10);
    }
    ageTime += 1500;
}
```

- **threads/scheduler.h**
  在class Scheduler中新增以下東西
  UpdatePriority()即如文字所述會更新thread的priority
  L1, L2, L3即為三種level的queue

```
public:
    void UpdatePriority(void);
    SortedList<Thread *> *L1;
    SortedList<Thread *> *L2;
    List<Thread *> *L3;
```

- **threads/scheduler.cc**
  - **comparator**
    使用於SortedList
    L1用remainingBurstTime排序
    L2用priority排序
    L3因為是NachOS內建的round robin所以不用再寫一個comparator

```
static int
L1Compare (Thread *t1, Thread *t2)
{
    if (t1->remainingBurstTime > t2->remainingBurstTime) return
1;
    else if (t1->remainingBurstTime < t2->remainingBurstTime)
return -1;
    else return 0;
}

static int
L2Compare (Thread *t1, Thread *t2)
{
    if (t1->priority > t2->priority) return -1;
    else if (t1->priority < t2->priority) return 1;
    else return 0;
}
```

  - **Scheduler constructor**
    L1和L2利用剛剛寫的comparator建立SortedList
    L3建立普通的List即可

```
L1 = new SortedList<Thread *>(L1Compare);
L2 = new SortedList<Thread *>(L2Compare);
L3 = new List<Thread *>;
```

- **Scheduler destructor**
  將L1, L2, L3都刪除

```
delete L1;
delete L2;
delete L3;
```

- **Scheduler::ReadyToRun()**
  根據thread的priority插入對應的queue
  更新ageTime

```
if (thread->priority >= 100) {
    L1->Insert(thread);
}
else if (thread->priority >= 50) {
    L2->Insert(thread);
}
else {
    L3->Append(thread);
}
thread->ageTime = kernel->stats->totalTicks;
```

- **Scheduler::FindNextToRun()**
  依順序L1 → L2 → L3找到要run的thread

```
if (!L1->IsEmpty()) {
    return L1->RemoveFront();
}
else if (!L2->IsEmpty()) {
    return L2->RemoveFront();
}
else if (!L3->IsEmpty()) {
    return L3->RemoveFront();
}
else
    return NULL;
```

- **Scheduler::Run()**
  在SWITCH前後分別紀錄nextThread, oldThread的startTime

```
nextThread->startTime = kernel->stats->totalTicks;
SWITCH(oldThread, nextThread);
oldThread->startTime = kernel->stats->totalTicks;
```

- **Scheduler::Print()**
  修改為 print L1, L2, L3

```
cout << "queue L1 contents:\n";
L1->Apply(ThreadPrint);
cout << "queue L2 contents:\n";
L2->Apply(ThreadPrint);
cout << "queue L3 contents:\n";
L3->Apply(ThreadPrint);
```

- **Scheduler::UpdatePriority()**
  遍歷L1, L2, L3並更新需要age的thread
  把thread從queue裏remove後再放入ready queue

```
void
Scheduler::UpdatePriority()
{
    ListIterator<Thread *> *iter1 = new ListIterator<Thread *>
(L1);
    ListIterator<Thread *> *iter2 = new ListIterator<Thread *>
(L2);
    ListIterator<Thread *> *iter3 = new ListIterator<Thread *>
(L3);

    for (; !iter1->IsDone(); iter1->Next()) {
        if (kernel->stats->totalTicks - iter1->Item()->ageTime >
1500) {
            iter1->Item()->Aging();
            L1->Remove(iter1->Item());
            ReadyToRun(iter1->Item());
        }
    }
    for (; !iter2->IsDone(); iter2->Next()) {
        if (kernel->stats->totalTicks - iter2->Item()->ageTime >
1500) {
            iter2->Item()->Aging();
            L2->Remove(iter2->Item());
            ReadyToRun(iter2->Item());
        }
    }
    for (; !iter3->IsDone(); iter3->Next()) {
        if (kernel->stats->totalTicks - iter3->Item()->ageTime >
1500) {
            iter3->Item()->Aging();
            L3->Remove(iter3->Item());
            ReadyToRun(iter3->Item());
        }
    }
}
```

- **threads/alarm.cc/Alarm::CallBack()**
  呼叫UpdatePriority()來更新priority
  若thread為L1或L3可呼叫YieldOnReturn()進行preempt

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    Thread *thread = kernel->currentThread;

    kernel->scheduler->UpdatePriority();

    if (status != IdleMode)
        if (kernel->currentThread->priority >= 100 || kernel-
>currentThread->priority < 50)  //L1 || L3
            interrupt->YieldOnReturn();
}
```

2-2

- **threads/kernel.h**
  在class kernel新增以下來存priority

```
int priority[10];
```

- **threads/kernel.cc**
  - **Kernel constructor**
    新增command line argument -ep 來讀進priority

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

  - **Kernel::ExecAll()**
    Exec()傳入priority

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

- **Kernel::Exec()**
  argument新增priority
  初始化priority

  ```
  int Kernel::Exec(char *name, int priority) {
      t[threadNum] = new Thread(name, threadNum);
      t[threadNum]->priority = priority;
      t[threadNum]->space = new AddrSpace();
      t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void
  *)t[threadNum]);
      threadNum++;

      return threadNum - 1;
  }
  ```

2-3

- **lib/debug.h**
  新增debugging flag z

  ```
  const char dbgMP3 = 'z';
  ```

- **[A]**

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

      if (thread->priority >= 100) {
          DEBUG(dbgMP3, "[A] Tick [" << kernel->stats->totalTicks << "]:
  Thread [" << thread->getID() << "] is inserted into queue L1");
          L1->Insert(thread);
      }
      else if (thread->priority >= 50) {
          DEBUG(dbgMP3, "[A] Tick [" << kernel->stats->totalTicks << "]:
  Thread [" << thread->getID() << "] is inserted into queue L2");
          L2->Insert(thread);
      }
      else {
          DEBUG(dbgMP3, "[A] Tick [" << kernel->stats->totalTicks << "]:
  Thread [" << thread->getID() << "] is inserted into queue L3");
          L3->Append(thread);
      }
  ```

```
    thread->ageTime = kernel->stats->totalTicks;
}
```

- **[B]**

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

    if (!L1->IsEmpty()) {
        DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << L1->Front()->getID() << "] is removed from queue L1");
        return L1->RemoveFront();
    }
    else if (!L2->IsEmpty()) {
        DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << L2->Front()->getID() << "] is removed from queue L2");
        return L2->RemoveFront();
    }
    else if (!L3->IsEmpty()) {
        DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << L3->Front()->getID() << "] is removed from queue L3");
        return L3->RemoveFront();
    }
    else
        return NULL;
}
```

```
void
Scheduler::UpdatePriority()
{
    ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(L1);
    ListIterator<Thread *> *iter2 = new ListIterator<Thread *>(L2);
    ListIterator<Thread *> *iter3 = new ListIterator<Thread *>(L3);

    for (; !iter1->IsDone(); iter1->Next())
    {
        if (kernel->stats->totalTicks - iter1->Item()->ageTime > 1500)
        {
            iter1->Item()->Aging();
            DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks << 
"]: Thread [" << iter1->Item()->getID() << "] is removed from queue 
L1");
            L1->Remove(iter1->Item());
            ReadyToRun(iter1->Item());
        }
    }
    for (; !iter2->IsDone(); iter2->Next())
```

```
    {
        if (kernel->stats->totalTicks - iter2->Item()->ageTime > 1500)
        {
            iter2->Item()->Aging();
            DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks <<
"]: Thread [" << iter2->Item()->getID() << "] is removed from queue
L2");
            L2->Remove(iter2->Item());
            ReadyToRun(iter2->Item());
        }
    }
    for (; !iter3->IsDone(); iter3->Next())
    {
        if (kernel->stats->totalTicks - iter3->Item()->ageTime > 1500)
        {
            iter3->Item()->Aging();
            DEBUG(dbgMP3, "[B] Tick [" << kernel->stats->totalTicks <<
"]: Thread [" << iter3->Item()->getID() << "] is removed from queue
L3");
            L3->Remove(iter3->Item());
            ReadyToRun(iter3->Item());
        }
    }
}
```

- **[C]**

```
void
Thread::Aging(void) {
    if (priority < 149) {
        int oldP = priority;
        priority = min(149, priority + 10);
        DEBUG(dbgMP3, "[C] Tick[" << kernel->stats->totalTicks << "]:
Thread [" << this->getID() << "] changes its priority from [" \
            << oldP << "] to [" << priority << "]");
    }
    ageTime += 1500;
}
```

- **[D]**

```
void
Thread::UpdateBurstTime(bool toReady) {
    execTime = kernel->stats->totalTicks-startTime;
    totalBurstTime += execTime;
    if (toReady)
        remainingBurstTime = max(0.0, approximateBurstTime -
totalBurstTime);
    else {
```

```
        double oldT = approximateBurstTime;
        approximateBurstTime = 0.5 * totalBurstTime + 0.5 *
approximateBurstTime;
        remainingBurstTime = approximateBurstTime;
        totalBurstTime = 0.0;
        DEBUG(dbgMP3, "[D] Tick[" << kernel->stats->totalTicks << "]:
Thread [" << this->getID() << "] update approximate burst time, from:
["
            << oldT << "], add [" << approximateBurstTime-oldT << "],
to [" << approximateBurstTime << "]");
    }
}
```

- **[E] threads/scheduler.cc/Scheduler::Run()**

```
nextThread->startTime = kernel->stats->totalTicks;
    DEBUG(dbgMP3, "[E] Tick[" << kernel->stats->totalTicks << "]:
Thread [" << nextThread->getID() << "] is now selected for execution,
thread [" \
        << oldThread->getID() << "] is replaced, and it has executed
[" << (int)(oldThread->execTime) << "] ticks");
    SWITCH(oldThread, nextThread);
    oldThread->startTime = kernel->stats->totalTicks;
```

Feedback

- **鄭幃謙**
  trace code的部分跟之前有重疊到沒有花那麼多時間
  但這次的implementation比較複雜，debug花了很多時間
  而且因為spec沒有sample input的output所以其實還是不確定是不是正確的Q Q
  而且aging試不出來完全不知道有沒有成功實作
  有跟同學對過結果相同希望是對的Q A Q
- **劉廷宜**
  因為上次的表現太爛
  （trace code太隨便都沒有trace到很精熟、助教問的問題我都不會）
  所以這次決定把要trace的code搞得很熟，至少題目不會完全回答不出來
  trace code到眼睛快花了
  把report大致完成後就跟鄭幃謙討論其中的一些盲點
  之後就開始implement Multi-level feedback queue
  另外也想說換成markdown來打報告雖然很累
  但是report出來易讀性提升讓我很開心
  希望自己可以再進步
  大家都辛苦了Q A Q