OS Homework 3 Report

Team21

葉宥忻 109062301

陳禹勳 109062134

contribution: 一起

1-1. New -> Ready

Kernel::ExecAll()

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

呼叫Kernel::Exec()去個別生成thread來執行kernel中所有的execfile，執行完後呼叫currentThread的Finish()，也就是結束掉main thread

Kernel::Exec(char*)

```cpp
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;
```

新增一個thread給要執行的user program，再make一個新的pageTable給這個thread，接著呼叫Thread::Fork來做allocate stack，把thread放進ready queue等操作，最後回傳這個thread的thread number

Thread::Fork(VoidFunctionPtr, void*)

```cpp
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);
}
```

先做StackAllocate()，allocate空間給要執行的thread，接下來disable interrupt，把thread放進ready queue，回復interrupt原本的狀態

Thread::StackAllocate(VoidFunctionPtr, void*)

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

create一塊pagesize*2+size的空間，保護前後兩個page，來偵測有沒有overflow，回傳第二個 page，也就是第一個可以用的page

```
#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
```

把stackTop移動到stack+StackSize-4的位置，然後--stackTop後指向threadRoot，當我們switch到這個process的時候，就會從Threadroot開始run

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

將ThreadRoot、ThreadBegin、func procedure、arg等pointer，存進machineState這個register，之後 要呼叫這些procedure或使用argument就從這些register找

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);
}
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

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();
    }
}
```

當user program開始執行的時候，準備好一個Instruction的data structure，並把status設成user mode，在for迴圈裡 OneInstruction一個一個讀並執行指令，每個指令執行完後呼叫OneTick()增加simulated time，同時檢查有沒有interrupt需要處理

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;
    }
}
```

看現在是user mode還是kernel mode增加simulated time，先disable interrupt，用 CheckIfDue()去檢查並處理interrupt，再enable interrupt回來，最後檢查yieldOnReturn是否為TRUE，若為TRUE就呼叫Yield()，做完再把yieldOnReturn設回FALSE，yieldOnReturn會根據以下的步驟被設為TRUE。

```
    alarm = new Alarm(randomSlice); // start up time slicing
```

在kernel::Initialize會create一個alarm object

```
Alarm::Alarm(bool doRandom)
{
    timer = new Timer(doRandom, this);
}
```

alarm會create一個time，並把doRandom和自己當作參數

```
Timer::Timer(bool doRandom, CallBackObj *toCall)
{
    randomize = doRandom;
    callPeriodically = toCall;
    disable = FALSE;
    SetInterrupt();
}
```

Time把傳進來的alarm object設為callBack object callPeriodically，doRandom決定是否random arrange interrupt，把disable設為FALSE，最後呼叫SetInterrupt()

```
void        You, 2 weeks ago • mp2
Timer::SetInterrupt()
{
    if (!disable) {
        int delay = TimerTicks;

        if (randomize) {
            delay = 1 + (RandomNumber() % (TimerTicks * 2));
        }
        // schedule the next timer device interrupt
        kernel->interrupt->Schedule(this, delay, TimerInt);
    }
}
```

接下來根據random決定delay interval，然後schedule這個timer device的interrupt，傳入timer自己和delay的時間

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

    DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time = " << when);
    ASSERT(fromNow > 0);

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

create一個toOccur interrupt，會傳入我們前面傳的timer object(toCall)和interrupt發生的時間(when)，然後insert到PendingInterrupt中

```cpp
321 bool
322 Interrupt::CheckIfDue(bool advanceClock)
323 {
324     PendingInterrupt *next;
325     Statistics *stats = kernel->stats;
326 
327     ASSERT(level == IntOff);        // interrupts need to be disabled,
328                     // to invoke an interrupt handler
329     if (debug->IsEnabled(dbgInt)) {
330     DumpState();
331     }
332     if (pending->IsEmpty()) {       // no pending interrupts
333     return FALSE;
334     }
335     next = pending->Front();
336 
337     if (next->when > stats->totalTicks) {
338         if (!advanceClock) {        // not time yet
339             return FALSE;
340         }
341         else {              // advance the clock to next interrupt
342         stats->idleTicks += (next->when - stats->totalTicks);
343         stats->totalTicks = next->when;
344         // UDelay(1000L); // rcgood - to stop nachos from spinning.
345     }
346     }
347 
348     DEBUG(dbgInt, "Invoking interrupt handler for the ");
349     DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);
350 
351     if (kernel->machine != NULL) {
352         kernel->machine->DelayedLoad(0, 0);
353     }
354 
355     inHandler = TRUE;
356     do {
357         next = pending->RemoveFront();    // pull interrupt off list
358         DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack, " << stats->totalTicks);
359         next->callOnInterrupt->CallBack();// call the interrupt handler
360         DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->CallBack, " << stats->totalTicks);
361     delete next;
362     } while (!pending->IsEmpty()
363             && (pending->Front()->when <= stats->totalTicks));
364     inHandler = FALSE;
365     return TRUE;
366 }
```

當OneTick或Idle呼叫CheckIfDue，而且時間到了interrupt該執行的時間時，會在359行呼叫interrupt的CallBack()，也就是timer的callback

```
void
Timer::CallBack()
{
    // invoke the Nachos interrupt handler for this device
    callPeriodically->CallBack();

    SetInterrupt(); // do last, to let software interrupt handler
                    // decide if it wants to disable future interrupts
}
```

呼叫callPeriodically(Alarm)的CallBack，同時再set一個新的interrupt

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

    if (status != IdleMode) {
    interrupt->YieldOnReturn();
    }
}

```

如果Machine的status不是IdleMode，呼叫interrupt的YieldOnReturn

```
Interrupt::YieldOnReturn()
{
    ASSERT(inHandler == TRUE);
    yieldOnReturn = TRUE;
}
```

把yieldOnReturn設為TRUE

Thread::Yield()

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

    ASSERT(this == kernel->currentThread);

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

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

當currentThread要把CPU的使用權交給其他thread，並且currentThread還沒finish，之後還要執行，所以用FindNextToRun找有沒有其他thread要run，有的話呼叫 ReadyToRun()把currentThread放回ready queue，然後呼叫scheduler run去做context switch

Scheduler::FindNextToRun()

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

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

回傳下一個要做的thread，若無則回傳NULL

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);
}
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

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());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

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

呼叫SWITCH停止oldThread開始nextThread，完成switch

```cpp
    // 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();
    }
}
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-3. Running -> Waiting

SynchConsoleOutput::PutChar(char)

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

1. lock -> Acquire(): 避免和其他thread同時output，透過lock::Acquire和semaphore::P去檢查和搶資源

2. consoleOutput -> PutChar(): 把char寫進display

3. waitFor->P(): 等待後面呼叫了CallBack()中的waitFor->V()表示output結束

4. lock->Release(): thread釋出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);
}
```

先disable interrupt，因為85到89行確認value跟減value的動作不能被中斷，若value=0，代表有其他thread在用，所以這個thread被block住了，所以把它放進waiting queue，然後讓他sleep，若value>0，表示resource空下來了，所以用value--去搶這個resource，然後再回復interrupt的狀態。

List<T>::Append(T)

```cpp
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));
}
```

List是一個link list的結構，append會把新的item接在last的尾端，numInList++
Thread::Sleep(bool)

```cpp
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";
    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);
}
```

把thread status設為BLOCKED，如果ready queue是空的，呼叫Interrupt::Idle去檢查有沒有interrupt要處理，如果ready queue有thread在等待，把CPU從A thread交給ready queue的第一個 thread，A thread可能是因為已經做完或在等synchronization variable，若是前者，A thread之後會被刪掉；若是後者之後會把A thread重新放回ready queue，未來會叫醒A thread

Scheduler::FindNextToRun()

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

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

回傳下一個要做的thread，若無則回傳NULL

Scheduler::Run(Thread*, bool)

```cpp
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());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

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

呼叫SWITCH停止oldThread開始nextThread，完成switch

```cpp
    // 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();
    }
}
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-4. Waiting -> Ready

Semaphore::V()

```
103 void
104 Semaphore::V()
105 {
106     DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
107     Interrupt *interrupt = kernel->interrupt;
108 
109     // disable interrupts
110     IntStatus oldLevel = interrupt->SetLevel(IntOff);
111 
112     if (!queue->IsEmpty()) {  // make thread ready.
113     kernel->scheduler->ReadyToRun(queue->RemoveFront());
114     }
115     value++;
116 
117     // re-enable interrupts
118     (void) interrupt->SetLevel(oldLevel);
119 }
```

先disable interrupt，因為85到89行確認value跟加value的動作不能被中斷，檢查waiting queue有沒有waiter，有的話把waiter放進ready queue，增加value來釋出資源，然後回復interrupt狀態

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);
}
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

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();
    break;
```

CPU收到SC_exit的exception type，故結束currentThread

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
}
```

當一個thread做完就會呼叫，先Disable interrupt，實作方式是呼叫Sleep裡面傳TRUE，讓currentThread Sleep，因為現在還沒完成switch，所以還不能刪除這個thread，Sleep的參數TRUE代表這個Thread已經結束了，之後才會在Scheduler::CheckToBeDestroyed將這個Thread刪除

Thread::Sleep(bool)

```cpp
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";
    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);
}
```

把thread status設為BLOCKED，如果ready queue是空的，呼叫Interrupt::Idle去檢查有沒有interrupt要處理，如果ready queue有thread在等待，把CPU從A thread交給ready queue的第一個 thread，A thread可能是因為已經做完或在等synchronization variable，若是前者，A thread之後會被刪掉；若是後者之後會把A thread重新放回ready queue，未來會叫醒A thread

Scheduler::FindNextToRun()

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

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

回傳下一個要做的thread，若無則回傳NULL

Scheduler::Run(Thread*, bool)

```cpp
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());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

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

呼叫SWITCH停止oldThread開始nextThread，完成switch

```cpp
    // 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();
    }
}
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-6. Ready -> Running

Scheduler::FindNextToRun()

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

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

回傳下一個要做的thread，若無則回傳NULL

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());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

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

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
    // 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();
    }
}
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

SWITCH(Thread*, Thread*)

```
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
```

```
342 _SWITCH:
343 SWITCH:
344         movl    %eax,_eax_save          # save the value of eax
345         movl    4(%esp),%eax            # move pointer to t1 into eax
346         movl    %ebx,_EBX(%eax)         # save registers
347         movl    %ecx,_ECX(%eax)
348         movl    %edx,_EDX(%eax)
349         movl    %esi,_ESI(%eax)
350         movl    %edi,_EDI(%eax)
351         movl    %ebp,_EBP(%eax)
352         movl    %esp,_ESP(%eax)         # save stack pointer
353         movl    _eax_save,%ebx          # get the saved value of eax
354         movl    %ebx,_EAX(%eax)         # store it
355         movl    0(%esp),%ebx            # get return address from stack into ebx
356         movl    %ebx,_PC(%eax)          # save it into the pc storage
357
358         movl    8(%esp),%eax            # move pointer to t2 into eax
359
360         movl    _EAX(%eax),%ebx         # get new value for eax into ebx
361         movl    %ebx,_eax_save          # save it
362         movl    _EBX(%eax),%ebx         # retore old registers
363         movl    _ECX(%eax),%ecx
364         movl    _EDX(%eax),%edx
365         movl    _ESI(%eax),%esi
366         movl    _EDI(%eax),%edi
367         movl    _EBP(%eax),%ebp
368         movl    _ESP(%eax),%esp         # restore stack pointer
369         movl    _PC(%eax),%eax          # restore return address into eax
370         movl    %eax,4(%esp)            # copy over the ret address on the stack
371         movl    _eax_save,%eax
372
373         ret
374
375 #endif // x86
```

先把%eax暫存到_eax_save，然後把t1 pointer存到%eax，346-352行把register存到t1的stack中，353-354行把%eax的restore後也存進t1的stack，355-356行把return address存進pc storage中。

358行把t2 pointer存到%eax。

360-361行把t2的%eax放到%ebx後暫存到_eax_save，362-369行從t2的stack restore register的值。

370把t2 return address存到%esp+4

371 restore t2 %eax的值。

373 ret 會回到%esp，下一個執行的指令是esp+4

depends on the previous process state, [New,Running,Waiting]→Ready

```
/* void ThreadRoot( void )
**
** expects the following registers to be initialized:
**      eax     points to startup function (interrupt enable)
**      edx     contains inital argument to thread function
**      esi     points to thread function
**      edi     point to Thread::Finish()
*/
```

New:會從Thread root開始做

```
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();
}
```

Running:代表t2原本是在跑Machine::Run的OneTick,因為yield被放進了ready queue,context switch回來後會從之前的program counter的instruction繼續run

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

Waiting:t2原本是在Semaphore::P等待資源而被放進了waiting queue,接下來又因為資源可以用了而被放進ready queue,所以context switch回來後,會從Semaphore::P的while繼續run

Implementation

debug.h

```
// MP3
const char z = 'z';
```

新增一個debug flag ‘z’

stats.h

```
// MP3
// const int ConsoleTime =   100; // time to read or write one character
const int ConsoleTime = 1;
```

kernel.cc

```
#endif // FILESYS_STUB
    // postOfficeIn = new PostOfficeInput(10);
    // postOfficeOut = new PostOfficeOutput(reliability);
```

```
    // delete postOfficeIn;
    // delete postOfficeOut;
```

```
        // MP3
        else if (strcmp(argv[i], "-ep") == 0){
            execfile[++execfileNum]= argv[++i];
            priorities[execfileNum] = atoi(argv[++i]);
        }
```

把要執行的檔案放進execfile中，並給它priority

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        // int a = Exec(execfile[i]);
        You, 21 minutes ago • Uncommitted changes
        // MP3
        int a = Exec(execfile[i], priorities[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

```
// MP3
// int Kernel::Exec(char* name)
int Kernel::Exec(char* name, int priority)
{
    // t[threadNum] = new Thread(name, threadNum);
    // MP3
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->setPriority(priority);

    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
```

create好thread之後，設定他的initital priority

Kernel.h

```
    // MP3
    int Exec(char* name, int priority);
```

```
// MP3        You, 1 s
int priorities[10];
```

Scheduler.h

```
    // MP3
    int Aging();
    // Thread* CheckPreemptive();
    bool CheckPreemptive();

    // SelfTest for scheduler is implemented in class Thread

  private:

    List<Thread *> *readyList;  // queue of threads that are ready to run,
        // but not running
    // MP3
    SortedList<Thread *> *L1;
    SortedList<Thread *> *L2;
    List<Thread *> *L3;
```

Scheduler.cc

```
Scheduler::Scheduler()
{
    readyList = new List<Thread *>;
    // MP3
    L1 = new SortedList<Thread *>(L1Compare);
    L2 = new SortedList<Thread *>(L2Compare);
    L3 = new List<Thread *>;

    toBeDestroyed = NULL;
}
```

在scheduler新增L1, L2, L3三個queue，L1和L2是sortedList，因為要根據remain time和priority來排序，在create sortedlist時，要把compare的function傳進去(L1Compare, L2Compare)。

```
Scheduler::~Scheduler()
{
    delete readyList;
    // MP3
    delete L1;
    delete L2;
    delete L3;
}
```

```cpp
// MP3
int
L1Compare(Thread *t1, Thread *t2) {
    // smaller burst time first
    double t1ApproxRemainTime = t1->getApproxRemainTime();
    double t2ApproxRemainTime = t2->getApproxRemainTime();

    if (t1ApproxRemainTime > t2ApproxRemainTime){
        return 1;
    }
    else if (t1ApproxRemainTime < t2ApproxRemainTime){
        return -1;
    }
    else {
        return 0;
    }
}
```

L1Compare比較的是approxRemainTime，approxRemainTime較小的優先

```cpp
// MP3
int
L2Compare(Thread *t1, Thread *t2) {
    // higher priority first
    int t1Priority = t1->getPriority();
    int t2Priority = t2->getPriority();

    if (t1Priority < t2Priority) {
        return 1;
    }
    else if (t1Priority > t2Priority) {
        return -1;
    }
    else {
        return 0;
    }
}
```

L2Compare比較的是priority，priority較大的優先

```
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);
    // MP3
    int threadPriority = thread->getPriority();

    thread->setStartWaitingTime(kernel->stats->totalTicks);

    if (threadPriority > 149) {
        // out of range
        DEBUG(z, "Thread priority is out of range");
    }
    else if (threadPriority >= 100) {
        // L1 (100 - 149)
        L1->Insert(thread);
        thread->InsertedIntoQueue(1);
    }
    else if (threadPriority >= 50) {
        // L2 (50 - 99)
        L2->Insert(thread);
        thread->InsertedIntoQueue(2);
    }
    else if (threadPriority >= 0) {
        // L3 (0 - 49)
        L3->Append(thread);
        thread->InsertedIntoQueue(3);
    }
    else {
        // out of range
        DEBUG(z, "Thread priority is out of range");
    }

```

當要將thread放進ready queue時，我們先將start waiting time設為現在，然後根據thread的priority用Insert(thread)把thread放進queue，InsertedIntoQueue()則設定thread的queuelevel並print出debug訊息

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

    // if (readyList->IsEmpty()) {
    //  return NULL;
    // } else {
    //  return readyList->RemoveFront();
    // }
    // MP3
    if (!L1->IsEmpty()) {···
    else if (!L2->IsEmpty()) {···
    else if (!L3->IsEmpty()) {···
    else {
        return NULL;
    }
}
```

FindNextToRun依序從L1, L2, L3尋找ready thread

```
    if (!L1->IsEmpty()) {
        Thread *currThread = kernel->currentThread;
        Thread *nextThread = L1->RemoveFront();

        nextThread->setCpuStartTime(kernel->stats->totalTicks);
        nextThread->setTotalWaitingTime(0);

        nextThread->RemovedFromQueue();
        currThread->ContextSwitch(nextThread->getID());

        return nextThread;
    }
```

假設L1 queue中有next thread，則把它從L1 queue中remove，然後將cpuStartTime設為現在，totalWaitingTime重置為0，RemovedFromQueue()和ContextSwitch()會print出debug訊息，最後回傳next thread

```cpp
// MP3
int
Scheduler::Aging()
{
    Thread *thread;
    ListIterator<Thread *> *iterator;

    if (!L1->IsEmpty()) {···

    if (!L2->IsEmpty()) {···

    if (!L3->IsEmpty()) {···
}
```

```cpp
    if (!L1->IsEmpty()) {
        iterator = new ListIterator<Thread *>(L1);
        for (; !iterator->IsDone(); iterator->Next()) {
            thread = iterator->Item();

            // Increase total waiting time
            bool doAging = thread->IncreaseTotalWaitingTime();
            thread->setStartWaitingTime(kernel->stats->totalTicks);

            // Check whether over 1500
            if (doAging) {
                thread->ChangePriority();
            }

        }
    }
```

```cpp
    if (!L2->IsEmpty()) {
        iterator = new ListIterator<Thread *>(L2);
        for (; !iterator->IsDone(); iterator->Next()) {
            thread = iterator->Item();

            // Increase total waiting time
            bool doAging = thread->IncreaseTotalWaitingTime();
            thread->setStartWaitingTime(kernel->stats->totalTicks);

            // Check whether over 1500
            if (doAging) {
                thread->ChangePriority();
            }

            if (thread->getPriority() > 99) {
                L2->Remove(thread);
                L1->Insert(thread);

                thread->RemovedFromQueue();
                thread->InsertedIntoQueue(1);
            }
        }
    }
```

```
    if (!L3->IsEmpty()) {
        iterator = new ListIterator<Thread *>(L3);
        for (; !iterator->IsDone(); iterator->Next()) {
            thread = iterator->Item();

            // Increase total waiting time
            bool doAging = thread->IncreaseTotalWaitingTime();
            thread->setStartWaitingTime(kernel->stats->totalTicks);

            // Check whether over 1500
            if (doAging) {
                thread->ChangePriority();
            }

            if (thread->getPriority() > 49) {
                L3->Remove(thread);
                L2->Insert(thread);

                thread->RemovedFromQueue();
                thread->InsertedIntoQueue(2);
            }
        }
    }
}
```

Aging會對所有queue中的thread去做檢查看是否要等待超過1500 ticks，有的話就增加10 priority，L2和L3中的thread需要再確認queue level是否可以升級了

```cpp
// MP3
bool
Scheduler::CheckPreemptive()
{
    Thread *currThread = kernel->currentThread;
    int currThreadLevel = currThread->getQueueLevel();

    if (currThreadLevel == 1) {
        if (!L1->IsEmpty()) {
            if (L1Compare(currThread, L1->Front()) == 1) {
                return TRUE;
            }
        }
    }
    else if (currThreadLevel == 2) {
        if (!L1->IsEmpty()) {
            return TRUE;
        }
    }
    else {
        if (!L1->IsEmpty()){
            return TRUE;
        }
        if (!L2->IsEmpty()) {
            return TRUE;
        }
        if (!L3->IsEmpty()) {      You, 1 second ago • Uncommi
            return TRUE;
        }
    }

    return FALSE;
}
```

1. 如果現在的thread是L1，只有L1的thread可以preemptive，所以拿他的remain time去和現在L1的第一個thread比，決定是否可以preemptive
2. 如果現在的thread是L2，只有L1的thread可以preemptive
3. 如果現在的thread是L3，所有thread都可以preempt

alarm.cc

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

    // MP3
    // calculate burst and remain time
    kernel->currentThread->IncreaseAccuTicks();
    kernel->currentThread->IncreaseCpuBurstTime();
    kernel->currentThread->UpdateApproxRemainTime();
    kernel->currentThread->setCpuStartTime(kernel->stats->totalTicks);    // Because we have to keep accumulate burst time

    // do aging and then check preemptive
    kernel->scheduler->Aging();
    // nextThread = kernel->scheduler->CheckPreemptive();

    if (status != IdleMode) {
        if (kernel->scheduler->CheckPreemptive()) {
            interrupt->YieldOnReturn();
        }
    }
}
```

在time slice到時，會呼叫CallBack()，在裡面計算accuTicks、cpuBurstTime、approxRemainTime，接著呼叫aging。最後確認thread是否可被其他thread搶，若可以則會呼叫YieldOnReturn()會把yieldOnReturn設為true，然後在OneTick()就會呼叫Yield()

thread.h

```
    // MP3
    int priority;
    int queueLevel;

    int startWaitingTime;
    int totalWaitingTime;

    double cpuStartTime;
    double cpuBurstTime;     // T
    double approxBurstTime;
    double approxRemainTime;

    double accuTicks;
```

在class Thread中新增幾項data，如下:

int priority:該thread的priority

int queueLevel:該thread在哪個ready queue

int startWaitingTime: 進入Waiting state的時間

int totalWaitingTime: 在waiting等了多少時間

double cpuStartTime: 進入Ready state的時間

double cpuBurstTime: 進入RuningState的時間

double approxBurstTime: 估計會進入RuningState的時間

double approxRemainTime: 估計所需要的剩餘時間

double accuTicks: 最後一次的連續cpuBurstTime

```
    int getPriority() { return priority; }
    int getQueueLevel() { return queueLevel; }
    int getStartWaitingTime() { return startWaitingTime; }
    int getTotalWaitingTime() { return totalWaitingTime; }

    double getCpuStartTime() { return cpuStartTime; }
    double getCpuBurstTime() { return cpuBurstTime; }
    double getApproxBurstTime() { return approxBurstTime; }
    double getApproxRemainTime() { return approxRemainTime; }

    void setPriority(int initPriority) { priority = initPriority; }
    void setStartWaitingTime(int newTime) { startWaitingTime = newTime; }
    void setTotalWaitingTime(int newTime) { totalWaitingTime = newTime; }

    void setCpuStartTime(double newTime) { cpuStartTime = newTime; }
    void setCpuBurstTime(double newTime) { cpuBurstTime = newTime; }
    void setAccuTicks(double newTick) { accuTicks = newTick; }
```

定義一些get和set function

Thread.cc

```
Thread::Thread(char* threadName, int threadID)
{
    ID = threadID;
    name = threadName;
    // MP3
    priority = 0;
    queueLevel = 0;

    totalWaitingTime = 0;
    startWaitingTime = 0;

    cpuStartTime = 0;
    cpuBurstTime = 0;
    approxBurstTime = 0;
    approxRemainTime = 0;

    accuTicks = 0;
```

Thread::Thread把剛才新加的data初始化為0

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

    ASSERT(this == kernel->currentThread);

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

    // // MP3
    // // nextThread = kernel->scheduler->FindNextToRun();
    // // calculate burst and remain time
    // this->IncreaseAccuTicks();
    // this->IncreaseCpuBurstTime();
    // this->UpdateApproxRemainTime();
    // this->setCpuStartTime(kernel->stats->totalTicks);   // Because we have to keep accumulate burst time

    // // do aging and then check preemptive
    // kernel->scheduler->Aging();
    // nextThread = kernel->scheduler->CheckPreemptive();
    nextThread = kernel->scheduler->FindNextToRun();

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

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

Thread::Yield()

current thread要將讓出cpu給其他thread，呼叫FindNextToRun()去找下一個thread

```cpp
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;

    // MP3
    this->IncreaseAccuTicks();
    this->IncreaseCpuBurstTime();
    // Only update approximate burst time from running to waiting
    if (!finishing) {
        this->UpdateApproxBurstTime();
    }

    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();  // no one to run, wait for an interrupt
    }

    // clear cpu burst time from running to waiting
    this->setCpuBurstTime(0);


    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

Thread::Sleep()

先計算accuTick、cpuBurstTime、approxBurstTime，然後因為thread從runnning移到了waiting，故要重設cpuBurstTime為0

```cpp
// MP3
void
Thread::InsertedIntoQueue(int newQueueLevel)
{
    queueLevel = newQueueLevel;

    DEBUG(z, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] is inserted into queue L[" << queueLevel << "]");
}
```

Thread::InsertedIntoQueue()當有thread priority改變，以致從一個queue跑到另一個 queue時呼叫(如L2->L1，就會insert到L1)

```cpp
// MP3
void
Thread::RemovedFromQueue()
{
    DEBUG(z, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] is removed from queue L[" << queueLevel << "]");
}
```

Thread::RemovedFromQueue()當有thread priority改變，以致從一個queue跑到另一個 queue時呼叫(如L2->L1，就會從L2移出)

```cpp
void
Thread::ChangePriority()
{
    int oldPriority = this->priority;
    int newPriority = min(oldPriority + 10, 149);

    priority = newPriority;

    DEBUG(z, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] changes its priority from [" << oldPriority << "] to [" << newP
}
```

將priority增加10，上限是149

```cpp
// MP3
void
Thread::IncreaseCpuBurstTime()
{
    cpuBurstTime = cpuBurstTime + (kernel->stats->totalTicks - cpuStartTime);
}
```

累積cpuBurstTime

```cpp
// MP3
bool
Thread::IncreaseTotalWaitingTime()
{
    totalWaitingTime = totalWaitingTime + (kernel->stats->totalTicks - startWaitingTime);

    if (totalWaitingTime >= 1500) {
        totalWaitingTime = totalWaitingTime - 1500;
        return TRUE;
    }
    else {
        return FALSE;
    }
}
```

累積totalWaitingTime，如果等了超過1500 Ticks，則減去1500並回傳true，代表要增加priority

```
Thread::UpdateApproxBurstTime()
{
    double newApproxBurstTime = 0.5 * cpuBurstTime + 0.5 * approxBurstTime;

    DEBUG(z, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "

    // update approxBurstTime and reset approxRemainTime
    approxBurstTime = newApproxBurstTime;
    approxRemainTime = newApproxBurstTime;
}
```

當thread從running to waiting，用公式計算新的approxBurstTime和approxRemainTime

```
// MP3
void
Thread::ContextSwitch(int newThreadId)
{
    DEBUG(z, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << newThreadId << "] is no
}
```

```
// MP3
void
Thread::UpdateApproxRemainTime()
{
    approxRemainTime = approxBurstTime - cpuBurstTime;
}
```

更新approxRemainTime

```
// MP3
void
Thread::IncreaseAccuTicks()
{
    accuTicks = accuTicks + (kernel->stats->totalTicks - cpuStartTime);
}
```

累積AccuTicks

心得

part 1 因為跟前兩次有蠻多地方重疊，所以code trace比較快，也比較快理解thread是如何在各個state切換。

part 2 則讓我們學會multy queueLevel scheduling是如何在OS上實作的，像是non-preemtive跟preemptive在實作上的差異，以及SJF如何實現的等等。