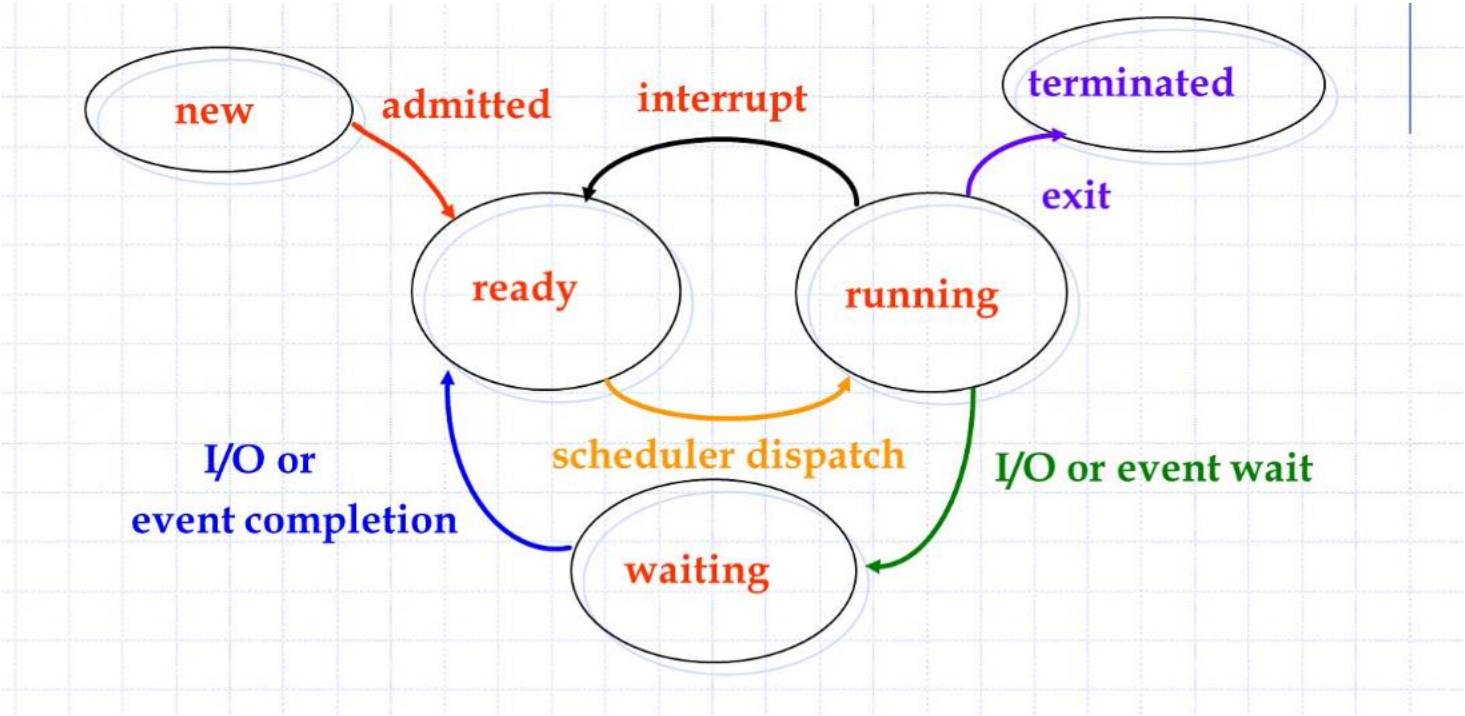
MP3　CPU scheduling

111062502

涂博允

Diagram of process state



New →Ready Trace code: Kernel::ExecAll()

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

利用for迴圈將程式一一讀取進來，接著透過Exec()執行程式，變數execfileNum則是已經讀取進來的程式數量。

每次呼叫 Exec() 時，會return讀取到的程式，並將執行結果存到ready queue變數 a 中。

當所有程式都執行完畢後，會呼叫currentThread->Finish()，結束currentThread的執行，讓其他thread有機會執行

Trace code: Kernel::Exec()

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

    return threadNum-1;
/*
```

為新的thread的space配置一塊新的addrspace

讓新的thread執行 ForkExecute() ，將 t[threadNum] ptr轉換為 void * 作為引數傳入。

回傳thread的index

Trace code: Thread::Fork()

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

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
    StackAllocate(func, arg);
```

Allocate a stack

```
    oldLevel = interrupt->SetLevel(IntOff);
```

ReadytoRun()必須在 interrupt 被 disable 的狀態下執行

```
    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
                                    // are disabled!
```

Put the thread on the ready queue

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

將interrupt恢復到 oldLevel 的原始狀態，以繼續執行之前可能被中斷的程式碼

Trace code: Thread::StackAllocate()

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

#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;  // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
                                // least 1 activation record
                                // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif

#ifdef ALPHA
    stackTop = stack + StackSize - 8;   // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif


#ifdef x86
    // 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

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

StackAllocate()的用途是為了一個新的thread配置一個Stack空間，用來存該thread在run時的local variable、call func. Return的addr.以及其他相關資訊。

一開始會先通過 AllocBoundedArray 函數配置一塊固定大小的stack空間，然後對不同的平台做一些特殊處理，以確保stack的正確配置。最後，將stack相關的資訊保存到一個叫做 machineState 的array中。

Trace code: Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff); 確保interrupt有開
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY); 將狀態設為ready
    readyList->Append(thread); 要排隊，因此放到readylist的末尾
}
```

Running→Ready Trace code: 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); 在一個for loop內 不斷的fetch and decode 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();
    }
}
```

Trace code: 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;
    }
}
```

CheckIfDue()檢查是否有在這段時間內待辦的的interrupts，如果有就處理該interrupt

如果前一個handler請求thread切換（yieldOnReturn 被 設為true），則轉成kernel mode並呼叫Yield()將目前的thread sleep掉並context switch到next thread

Trace code: 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);
}
```

從scheduler中找到下一個要執行的thread，並指派給 nextThread

若存在nextthread，則currentThread會放掉 CPU 使用權，之後 scheduler 會從ready queue找到下一個 thread 並執行，而且 currentThread 也會將自己放到 ready queue 的末端。之後當 currentThread 又排到 CPU 使用權時會 return 回當下放棄時的位址，並繼續執行之後的指令，最後將interrupt 的狀態設成之前的狀態。

Trace code: FindNextToRun ()

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

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

取得ready queue 最前面的thread並dequeue掉，且return取得的thread，若ready queue為空，則return null

Trace code: Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff); 確保interrupt有開
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY); 將狀態設為ready
    readyList->Append(thread); 要排隊，因此放到readylist的末尾
}
```

Trace code: Scheduler::Run()

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

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

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

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

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

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

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();       // check if thread we were running
                                // before this one has finished
                                // and needs to be cleaned up

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

透過finishing這個變數來判斷currentthread是否已完成，如果是因為timequantum等等的時間限制導致被迫讓出CPU(Thread::Yield())，也因為還尚未執行Thread::Finish()，因此finishing還是False，若thread成功完成的話，會呼叫Thread::Finish()，finishing就會被設為True，且thread會進入terminated state

若currentthread是user program，會有自己的addr. Space，因此在context switch前需將CPU register以及addr. Space 的state都儲存起來

判斷是否有overflow

Context switch就是在這之後處理，將nextthread的state切換成running，而Switch()的細節在Ready→Running時會更詳細解釋，此時也開始執行nextthread的program

因當時currentthread是被強迫放棄cpu的，因此此時要取回CPU使用權，並檢查是否有執行完成的thread還尚未ToBeDetroy

上面有提到若currentthread是user program，則需回復CPU register以及addr. Space 的state

Running→Waiting SynchConsoleOutput::PutChar()

```
void
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();   //lock -- only one writer at a time || Acquire -- wait until the lock is FREE, then set it to BUSY
    consoleOutput->PutChar(ch);
    waitFor->P();      //wait for EOF or a char to be available?
    lock->Release();   //Release -- set lock to be FREE, waking up a thread waiting in Acquire if necessary
}
```

Lock->Acquire() : 此部分是由semaphore實作的，目的是解決同步問題，取得該lock的thread才擁有使用權

consoleOutput->PutChar(ch) : 調用consoleOutput的PutChar函數， PutChar()主要功能是將單個字元寫入檔案或其他輸出裝置。在寫入之前，它會確保putBusy狀態是FALSE，並在寫入後設定putBusy狀態為TRUE，以避免同時執行多個PutChar操作。最後將這個interrupt排程，以在適當的時候執行後續的處理

該interrupt處理完成後會呼叫CallBack() ，最後也會執行waitFor->V()表示輸出完成。

waitFor->P() : 等到前面的waitFor->V()執行結束後，則可繼續執行(semaphore概念)

Lock->Release() : currentthread釋放lock，可提供給其他thread使用了

Trace code: 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);
}
```

Semaphore是atomic，因此要disable interrupt，
若有可用資源(value>0)，則使用該資源且將value-1，
若沒有可用資源(value==0)，則將currentthread放進queue中等待，以及將該thread put to sleep。
全部結束的最後將interrupt重啟

Trace code: List::Append(T)

```
template <class T>
SynchList<T>::SynchList()
{
    list = new List<T>;
    lock = new Lock("list lock");
    listEmpty = new Condition("list empty cond");
}
```

該list利用wait、signal來達到互斥存取的效果，例如:wait(P1)只能由signal(P1)來釋放，P2、P3是無法做到的

```
void Condition::Signal(Lock* conditionLock)
{
    Semaphore *waiter;

    ASSERT(conditionLock->IsHeldByCurrentThread());

    if (!waitQueue->IsEmpty()) {
        waiter = waitQueue->RemoveFront();
    waiter->V();
    }
}
```

```
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();
}
```

取得Lock後，把iten apend到list末端，結束後要做Signal讓其他thread wake up waiter，最後把lock釋放。

Trace code: 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";
    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);
}
```

If finishing=True，代表執行完成，currentthread會被block住，並且等到nextthread的出現以及呼叫CheckToBeDestroyed() ，否則會一直卡在此while loop內。
If finishing=False，代表執行失敗，currentthread放棄使用CPU，因此一樣將currentthread block住，並從ready queue中找出nextthread並執行。

Trace code: FindNextToRun ()

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

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

取得ready queue 最前面的thread並dequeue掉，且return取得的thread，若ready queue為空，則return null

Trace code: Scheduler::Run()

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

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

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

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

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

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

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();       // check if thread we were running
                                // before this one has finished
                                // and needs to be cleaned up

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

透過finishing這個變數來判斷currentthread是否已完成，如果是因為timequantum等等的時間限制導致被迫讓出CPU(Thread::Yield())，也因為還尚未執行Thread::Finish()，因此finishing還是False，若thread成功完成的話，會呼叫Thread::Finish()，finishing就會被設為True，且thread會進入terminated state

若currentthread是user program，會有自己的addr. Space，因此在context switch前需要將CPU register以及addr. Space 的state都儲存起

判斷是否有overflow

Context switch就是在這之後處理，將nextthread的state切換成running，而Switch()的細節在Ready→Running時會更詳細解釋，此時也開始執行nextthread的program

因當時currentthread是被強迫放棄cpu的，因此此時要取回CPU使用權，並檢查是否有執行完成的thread還尚未ToBeDetroy

上面有提到若currentthread是user program，則需回復CPU register以及addr. Space 的state

Waiting→Ready Trace code: 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);
}
```

Semaphore是atomic，因此要disable interrupt，並檢查是否有thread在queue中等待，若有則將該thread dequeue並放入ready queue中，使其之後能使用semaphore，並把value+1，最後將interrupt重啟。

Trace code: Scheduler::ReadyToRun()

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff); 確保interrupt有開
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY); 將狀態設為ready
    readyList->Append(thread); 要排隊，因此放到readylist的末尾
}
```

Running→Terminated Trace code: ExceptionHandler(ExceptionType) case SC_Exit

執行user program時，若需要system call時會呼叫此程式來處理exception，並且會根據exception的type來個別處理

case SC_Exit： 若程式執行完畢會呼叫此system call，exceptionHandler會來處理，該system call 會return userprogram 想要return的 value，並呼叫Thread::Finish()來結束該thread

Trace code: 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currentthread執行完後，就會呼叫此function

首先必須把interrupt關掉，之後把currentthread put to sleep

Trace code: 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";
    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);
}
```

If finishing=True，代表執行完成，currentthread會被block住，並且等到nextthread的出現以及呼叫CheckToBeDestroyed() ，否則會一直卡在此while loop內。
If finishing=False，代表執行失敗，currentthread放棄使用CPU，因此一樣將currentthread block住，並從ready queue中找出nextthread並執行。

Trace code: FindNextToRun ()

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

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

取得ready queue 最前面的thread並dequeue掉，且return取得的thread，若ready queue為空，則return null

Trace code: Scheduler::Run()

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

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

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

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

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

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

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();       // check if thread we were running
                // before this one has finished
                // and needs to be cleaned up

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

透過finishing這個變數來判斷currentthread是否已完成，如果是因為timequantum等等的時間限制導致被迫讓出CPU(Thread::Yield())，也因為還尚未執行Thread::Finish()，因此finishing還是False，若thread成功完成的話，會呼叫Thread::Finish()，finishing就會被設為True，且thread會進入terminated state

若currentthread是user program，會有自己的addr. Space，因此在context switch前需要將CPU register以及addr. Space 的state都儲存起來

判斷是否有overflow

Context switch就是在這之後處理，將nextthread的state切換成running，而Switch()的細節在Ready→Running時會更詳細解釋，此時也開始執行nextthread的program

因當時currentthread是被強迫放棄cpu的，因此此時要取回CPU使用權，並檢查是否有執行完成的thread還尚未ToBeDetroy

上面有提到若currentthread是user program，則需回復CPU register以及addr. Space 的state

Ready→Running Trace code: SWITCH(Thread*, Thread*)

Scheduler::FindNextToRun() 以及Scheduler::Run()由於前面重覆太多次，這邊就不重覆講解

```
/* 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    %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
```

Switch是由組合語言寫的，主要目的為儲存currentthread的cpu reg. 的state和addr. space的state，並回復newthread的cpu reg. 和addr. space的state

Step1: push eax到stack 當作currentthread的ptr

Step2: 儲存currentthread的cpu reg.的state 到stack ptr所指向的位置，並把return address from stack 存到PC storage

Step3: 把 newthread 的 stack ptr 存進 eax 當作newthread 的ptr

Step4: 藉由當初存到stack ptr所指向的位置的值，回復newthread的cpu reg. state 並從PC storage中取回return addr.

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

New → Running → Ready : 表示currentthread在第一次執行時被interrupt，並切換到newthread執行，newthread執行完畢後，會return currentthread當初執行switch的位址

Running → Waiting → Ready : 表示執行時，由於遇到I/O 中斷，被迫放到wait queue等待，直到I/0處理完畢後，會被放回ready queue

for loop in 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();
    }
}
```

For loop的目的就是為了無限循環不斷的fetch and decode instr.
OneTick()用來把時間往前推進，並檢查是否有interrupt

Implementation

2-1(a)、(b)

```
void Thread::RunningToWaiting() {
    double oldBurstTick = nextBurstTick;
    leaveRunning();
    nextBurstTick = 0.5 * burstTick + 0.5 * nextBurstTick;
    DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                        << "] update approximate burst time, from: [" << oldBurstTick
                        << "], add [" << burstTick << "], to [" << nextBurstTick << "]");
    burstTick = 0;
}

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
    //MP3
    currentThread->RunningToWaiting();
    // MP3 end
    queue->Append(currentThread);   // so go to sleep
    currentThread->Sleep(FALSE);
    }
    value--;              // semaphore available, consume its value

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

因為burst time是在waiting state計算，因此在此處計算(value==0，代表進入waiting ，thread put to sleep)

Implementation

2-1(c)

```
// Returns remain burst time
double getRemainBurstTick() const { return nextBurstTick - burstTick; }
```

```
static int ShortBurstTimeFirst(Thread *a, Thread *b){
    // less-burst-time thread is smaller
    if (a->getRemainBurstTick() > b->getRemainBurstTick()) {
        return 1;
    } else if (a->getRemainBurstTick() < b->getRemainBurstTick()) {
        return -1;
    } else {
        return 0;
    }
}

Scheduler::Scheduler()
{
    //readyList = new List<Thread *>;
    readyList_SJF = new SortedList<Thread *>(ShortBurstTimeFirst);
    preemtingThread = NULL;
    toBeDestroyed = NULL;
}
```

我在thread.h實作了getRemainBurstyTick() 用來計算running thread的 remain burst time (此處簡稱RBT)

並在scheduler.cc實作了burst time比較的部分
若thread a的RBT > thread b的RBT則return 1
若thread a的RBT = thread b的RBT則return 0
若thread a的RBT < thread b的RBT則return -1

Implementation

2-1(d)

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

    bool preempted = kernel->scheduler->Preempt();

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

CallBack()會計算時間，每次的time interval都會update一次，因此Preemt()在這邊實作，判斷是否要Preempt running thread

Implementation

2-2(a)

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

在debug.h定義const char dbgScheduler =z

```
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);
    // MP3
    thread->enterReady();
    DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue ");
    readyList_SJF->Append(thread);
    // MP3 end
}
```

ReadyToRun此處會把thread insert ready queue ，因此[A]Tick實作於此

Implementation

2-2(b)

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

    if (readyList_SJF->IsEmpty()) {
        return NULL;
    } else {
        Thread *thread = readyList_SJF->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                << "] is removed from queue ");
        return readyList_SJF->RemoveFront();
    }
}
```

從ready queue的首端找出下一個要執行的thread 因此[B]Tick 實作於此

2-2(c)

```
void Thread::RunningToWaiting() {
    double oldBurstTick = nextBurstTick;
    leaveRunning();
    nextBurstTick = 0.5 * burstTick + 0.5 * nextBurstTick;
    DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                        << "] update approximate burst time, from: [" << oldBurstTick
                                        << "], add [" << burstTick << "], to [" << nextBurstTick << "]");
    burstTick = 0;
}
```

approximate burst time我放在Thread::RunningToWaiting()內計算，因此[C]Tick實作於此

Implementation

2-2(d)

Whenever a context switch occurs without preemption: 代表遇到I/O interrupt 或是lock 、等待其他process釋放資源 等等因素，因為取得lock者會去做I/O以及未取得lock者會等待，都算是running->waiting所以我將[D]Tick實作於此處

```
void
SynchConsoleOutput::PutChar(char ch)
{
    //MP3
    DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getID()
                                << "] is now selected for execution, thread [" << oldThread->getID()
                                << "] starts IO, and it has executed [" << oldThread->getBurstTick() << "] ticks");
    //MP3 end
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

2-2(e)

```
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->ReadyToRunning();
DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getID()
                        << "] is now selected for execution, thread [" << oldThread->getID()
                        << "] is preempted, and it has executed [" << oldThread->getBurstTick() << "] ticks");
SWITCH(oldThread, nextThread);
```

當找到nextthread且要執行時，代表currentthread要被preempt
因此我將[E]Tick實作於SWITCH上方

Implementation

```
[os23s68@localhost test]$ ../build.linux/nachos -e hw3t1 -e hw3t2
hw3t1
hw3t2
1
return value:1
2
return value:2
^C
Cleaning up after signal 2
[os23s68@localhost test]$ ../build.linux/nachos -e hw3t1 -e hw3t2 -d z
hw3t1
hw3t2
[A] Tick [10]: Thread [1] is inserted into queue
[A] Tick [20]: Thread [2] is inserted into queue
[B] Tick [30]: Thread [1] is removed from queue
[E] Tick [30]: Thread [1] is now selected for execution, thread [0] is preempted, and it has executed [0] ticks
1[C] Tick [1162]: Thread [1] update approximate burst time, from: [0], add [1132], to [566]
[B] Tick [1162]: Thread [2] is removed from queue
[E] Tick [1162]: Thread [2] is now selected for execution, thread [1] is preempted, and it has executed [0] ticks
[A] Tick [1172]: Thread [1] is inserted into queue
[C] Tick [111184]: Thread [2] update approximate burst time, from: [0], add [110022], to [55011]
[B] Tick [111184]: Thread [1] is removed from queue
[E] Tick [111184]: Thread [1] is now selected for execution, thread [2] is preempted, and it has executed [0] ticks

[C] Tick [111194]: Thread [1] update approximate burst time, from: [566], add [10], to [288]
[A] Tick [111195]: Thread [1] is inserted into queue
[B] Tick [111195]: Thread [1] is removed from queue
[E] Tick [111195]: Thread [1] is now selected for execution, thread [1] is preempted, and it has executed [0] ticks
[A] Tick [111205]: Thread [2] is inserted into queue
return value:1
[B] Tick [331220]: Thread [2] is removed from queue
[E] Tick [331220]: Thread [2] is now selected for execution, thread [1] is preempted, and it has executed [220010] ticks
2[C] Tick [331230]: Thread [2] update approximate burst time, from: [55011], add [10], to [27510.5]
[A] Tick [331231]: Thread [2] is inserted into queue
[B] Tick [331231]: Thread [2] is removed from queue
[E] Tick [331231]: Thread [2] is now selected for execution, thread [2] is preempted, and it has executed [0] ticks

[C] Tick [331241]: Thread [2] update approximate burst time, from: [27510.5], add [10], to [13760.2]
[A] Tick [331242]: Thread [2] is inserted into queue
[B] Tick [331242]: Thread [2] is removed from queue
[E] Tick [331242]: Thread [2] is now selected for execution, thread [2] is preempted, and it has executed [0] ticks
return value:2
^C
Cleaning up after signal 2
```

Discuss

這次的作業相較於之前的MP1、MP2，我認為複雜度提升了許多，光是Trace code 的switch就花了不少時間複習組合語言，以及在實作時，各個state的切換中有非常多的細節需要注意，也因為此次的實作讓我對Diagram of process state的流程更清晰了