**OS HW03 GROUP 18**

**Part 1:Trace Code**

1. **Explain following path**
   1. **New – Ready**
2. **Kernel::ExecAll() -**

void Kernel::ExecAll()

{

for (int i=1;i<=execfileNum;i++) {

int a = Exec(execfile[i]);

}

currentThread->Finish();

}

*ExecAll()* 會對所有的 execfile (要執行的指令或檔案) 做Exec()，然後呼叫 *Finish()* 來終止執行 *ExecAll()* 的 Thread。

1. **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;

}

*Exec()* 會先創造一條新的 Thread 給要執行的程式，之後呼叫*AddrSpace()* 給 Thread 一個對應的address space，最後在透過 Fork() 讀取要執行的程式 ( Fork() 透過 VoidFunctionPtr 指向要使用的 function ForkExecute() )，並累加 total thread 的數量。

1. **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);

oldLevel = interrupt->SetLevel(IntOff);

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

(void) interrupt->SetLevel(oldLevel);

}

*Fork()* 會使Thread 和kernel 裡的物件都指到同一個 interrupt 和 scheduler，然後再利用*StackAllocate()* 分配和初始化stack，而 function pointer func 這邊會呼叫 Kernel::ForkExecute，arg是要傳給procedure的參數 (這邊為某一個 Thread)，然後因 *ReadyToRun()* 的要求，將 interrupt 設為 IntOff，之後再執行*ReadyToRun()*，將目前 this 這個 Thread 物件加入 Ready\_List 中，之後再還原 interrupt 的 Level。

1. **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

\*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

}

*Thread::StackAllocate() 會經由 Fork()呼叫*，它有各種的def要去對不同的device做設置(PARISC，SPARC，PowerPC，decmips，alpha，x86，PARISC)，他會分配以及初始化 stack。

1. **Scheduler::ReadyToRun()**

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);

}

而在*ReadyToRun()*，它會將 thread 的 status 設為 ready，然後再將 ready 的thread 放入ready\_list，等待後續執行.

* 1. **Running – Ready**

1. **Machine::Run();**

void Machine::Run()

{

Instruction \*instr = new Instruction; //storage for decoded instruction

kernel->interrupt->setStatus(UserMode); //transfer control to user mode

for (;;)

{

OneInstruction(instr);

kernel->interrupt->OneTick();

if (singleStep && (runUntilTime <= kernel->stats->totalTicks))

Debugger();

}

}

*Run()* 用來模擬當程式啟動 ， kernel 會將系統設定為 user mode，並執行 *OneInstruction()* 來運行程式的指令，並且使用 *OneTick()* 來模擬 CPU 裡的 clock。

1. **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;

    }

}

當進行 *OneTick()* 時會先累加 TotalTick，接著會檢查是否有 pending 的 interrupt，並且會先將 interrupt 設定為 disable，並執行 *CheckIfDue()*，然後再將 interrupt 重設為 enable，之後若 timer device ask for context switch，則會執行下面的 if statement，呼叫 *Yield()* 來取得ready queue 裡的下一個 thread，並進行 context switch。

1. **Thread::Yield()**

void

Thread::Yield() {

Thread \*nextThread;

IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

nextThread = kernel->scheduler->FindNextToRun();

if (nextThread != NULL) {

kernel->scheduler->ReadyToRun(this);

kernel->scheduler->Run(nextThread, FALSE);

}

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

}

Yield() 會呼叫 FindNextToRun() 來取得 ready queue 裡的下一個 thread，若 ready queue 裡是空的會 call ReadyToRun() 將目前執行的 thread 放回 ready queue 裡面，並且在執行此。

1. **Scheduler::FindNextToRun**

Thread \*

Scheduler::FindNextToRun ()

{

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

    if (readyList->IsEmpty()) {

        return NULL;

    } else {

        return readyList->RemoveFront();

    }

}

*FindNextToRun()* 會從 ReadyList 中找到下一個要執行的 Thread

1. **Scheduler::ReadyToRun -**

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);

}

而在*ReadyToRun()*，它會將 thread 的 status 設為 ready，然後再將 ready 的thread 放入ready\_list，等待後續執行.

1. **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());

SWITCH(oldThread, nextThread);

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

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

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

if (oldThread->space != NULL) { // if there is an address space

oldThread->RestoreUserState(); // to restore, do it.

oldThread->space->RestoreState();

}

}

若 finishing 為 true 代表此 currentThread 之前有呼叫過 *Finish()*，這邊用 toBeDestroyed 記錄此要刪除的 thread，接著準備做 content switch，會先儲存 oldThread 的資訊，並將 nexThread 的 status 設為 RUNNING 並且 assign 給 currentThread (即將此 thread 放到 CPU上跑)，接這便呼叫 *SWITCH()* 進行 context switch，而 content switch 完成後並不會執行下一行程式，因為CPU現在的控制權在 nextThread 手上，而當CPU的控制權又 content switch 回原本的 thread 時，才會做 *SWITCH()* 後續的程式碼，其會讀取原本的 Thread 的資訊並繼續後續指令的執行，而*CheckToBeDestroyed()* 會刪除 toBeDestroyed 中記錄要刪除的 thread

* 1. **Running – Waiting**

1. **SynchConsoleOutput::PutChar**

void SynchConsoleOutput::PutChar(char ch) {

lock->Acquire();

consoleOutput->PutChar(ch);

waitFor->P();

lock->Release();

}

使用*lock->Acquire()*; 去lock 資源。然後去呼叫 thread去使用資源，其他thread會被block直到lock被release。之後呼叫*PutChar(ch)*去傳遞參數ch，然後再利用*waitfor ->P()*去讓後續的字元去等待。

1. **Semaphore::P()**

void

Semaphore::P()

{

    Interrupt \*interrupt = kernel->interrupt;

    Thread \*currentThread = kernel->currentThread;

    // disable interrupts

    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {        // semaphore not available

    queue->Append(currentThread);   // so go to sleep

    currentThread->Sleep(FALSE);

    }

    value--;            // semaphore available, consume its value

    (void) interrupt->SetLevel(oldLevel);

}

解決threads的同步問題。號誌的value > 0則value遞減，檢查value值和遞減不可中斷。當value==0時，行程會被擋住，thread無法執行。

1. **List::Append**

template <class T>

void List<T>::Append(T item)

{

    ListElement<T> \*element = new ListElement<T>(item);

    ASSERT(!this->IsInList(item));

    if (IsEmpty())

    { // list is empty

        first = element;

        last = element;

    }

    else

    { // else put it after last

        last->next = element;

        last = element;

    }

    numInList++;

    ASSERT(this->IsInList(item));

}

將 item 放到 linklist 中的最後一個位置。

1. **Thread::Sleep()**

void

Thread::Sleep (bool finishing){

Thread \*nextThread;

ASSERT(this == kernel->currentThread);

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

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

status = BLOCKED;

while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)

kernel->interrupt->Idle(); // no one to run, wait for an interrupt

// returns when it's time for us to run

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

}

只有 currentThread 可以呼叫此 *Thread::Sleep()*，然後將 currentThread 的 status 設為 BLOCKED，準備做 content switch，接著跑 *FindNextToRun()*，從 Ready\_List 中找到下一個要進入 CPU 的 nextThread，並執行 *Run()*，若Ready\_list裡面沒有任何 element就不會有任何的執行，等待interrupt，其中的 argument finishing 若為 true 表示此 thread 執行完成了，等下在後續的 function 中會刪除此 thread。

1. **Scheduler::FindNextToRun()**

Thread \*

Scheduler::FindNextToRun ()

{

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

    if (readyList->IsEmpty()) {

        return NULL;

    } else {

        return readyList->RemoveFront();

    }

}

*FindNextToRun()* 會從 ReadyList 中找到下一個要執行的 Thread。

1. **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

    SWITCH(oldThread, nextThread);

   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();

    }

thread尚未執行完成時要切換到另一個thread執行，需要保存原本thread當下狀態，並載入下一個要執行的thread狀態。若舊thread的程序已完成，則刪除。

* 1. **Waiting – Ready**

1. **Semaphore::V**

void

Semaphore::V()

{

    Interrupt \*interrupt = kernel->interrupt;

    // disable interrupts

    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.

    kernel->scheduler->ReadyToRun(queue->RemoveFront());

    }

    value++;

    // re-enable interrupts

    (void) interrupt->SetLevel(oldLevel);

}

用以增加信號值，執行V()，訊號標S的值會被增加。結束離開臨界區段的行程，將會執行V()。當訊號標S不為負值時，先前被擋住的其他行程，將可獲准進入臨界區段。

1. **Scheduler::ReadyToRun()**

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);

}

而在*ReadyToRun()*，它會將 thread 的 status 設為 ready，然後再將 ready 的thread 放入ready\_list，等待後續執行.

* 1. **Running – Terminated**

1. **ExceptionHandler(ExceptionType)**

void

ExceptionHandler(ExceptionType which)

{

        case SC\_Exit:

            DEBUG(dbgAddr, "Program exit\n");

            val=kernel->machine->ReadRegister(4);

            cout << "return value:" << val << endl;

            kernel->currentThread->Finish();

            break;

}

ExceptionHandler根據 ExceptionType which 使用case SC\_Exit，然後對於SC\_Exit，它輸出register(4)的終止呼叫currentThread->*Finish()*來終止執行的thread。

1. **Thread::Finish()**

void Thread::Finish ()

{

(void) kernel->interrupt->SetLevel(IntOff);

Sleep(TRUE);

}

thread已經執行完畢，當forked程序結束時被呼叫，呼叫Sleep()使thread 變為blocked狀態。

1. **Thread::Sleep**

Thread::Sleep (bool finishing)

{

    Thread \*nextThread;

    ASSERT(this == kernel->currentThread);

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

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

    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);

}

s

只有 currentThread 可以呼叫此 *Thread::Sleep()*，然後將 currentThread 的 status 設為 BLOCKED，準備做 content switch，接著跑 *FindNextToRun()*，從 Ready\_List 中找到下一個要進入 CPU 的 nextThread，並執行 *Run()*，若Ready\_list裡面沒有任何 element就不會有任何的執行，等待interrupt，其中的 argument finishing 若為 true 表示此 thread 執行完成了，等下在後續的 function 中會刪除此 thread。

1. **Scheduler::FindNextToRun**

Thread \*

Scheduler::FindNextToRun ()

{

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

    if (readyList->IsEmpty()) {

        return NULL;

    } else {

        return readyList->RemoveFront();

    }

}

*FindNextToRun()* 會從 ReadyList 中找到下一個要執行的 Thread。

1. **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

    SWITCH(oldThread, nextThread);

   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();

    }

若 finishing 為 true 代表此 currentThread 之前有呼叫過 *Finish()*，這邊用 toBeDestroyed 記錄此要刪除的 thread，接著準備做 content switch，會先儲存 oldThread 的資訊，並將 nexThread 的 status 設為 RUNNING 並且 assign 給 currentThread (即將此 thread 放到 CPU上跑)，接這便呼叫 *SWITCH()* 進行 context switch，而 content switch 完成後並不會執行下一行程式，因為CPU現在的控制權在 nextThread 手上，而當CPU的控制權又 content switch 回原本的 thread 時，才會做 *SWITCH()* 後續的程式碼，其會讀取原本的 Thread 的資訊並繼續後續指令的執行，而*CheckToBeDestroyed()* 會刪除 toBeDestroyed 中記錄要刪除的 thread

* 1. **Ready – Running**
     1. **Scheduler::FindNextToRun**

Thread \*

Scheduler::FindNextToRun ()

{

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

    if (readyList->IsEmpty()) {

        return NULL;

    } else {

        return readyList->RemoveFront();

    }

}

*FindNextToRun()* 會從 Ready\_List 中找到下一個要執行的 Thread

* + 1. **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

    SWITCH(oldThread, nextThread);

   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();

    }

若 finishing 為 true 代表此 currentThread 之前有呼叫過 *Finish()*，這邊用 toBeDestroyed 記錄此要刪除的 thread，接著準備做 content switch，會先儲存 oldThread 的資訊，並將 nexThread 的 status 設為 RUNNING 並且 assign 給 currentThread (即將此 thread 放到 CPU上跑)，接這便呼叫 *SWITCH()* 進行 context switch，而 content switch 完成後並不會執行下一行程式，因為CPU現在的控制權在 nextThread 手上，而當CPU的控制權又 content switch 回原本的 thread 時，才會做 *SWITCH()* 後續的程式碼，其會讀取原本的 Thread 的資訊並繼續後續指令的執行，而*CheckToBeDestroyed()* 會刪除 toBeDestroyed 中記錄要刪除的 thread

* + 1. **SWITCH(Thread\*, Thread\*)**

SWITCH:

movl %eax,\_eax\_save # save the value of eax

movl 4(%esp),%eax # move pointer to t1 into eax

movl %ebx,\_EBX(%eax) # save registers

movl %ecx,\_ECX(%eax)

movl %edx,\_EDX(%eax)

movl %esi,\_ESI(%eax)

movl %edi,\_EDI(%eax)

movl %ebp,\_EBP(%eax)

movl %esp,\_ESP(%eax) # save stack pointer

movl \_eax\_save,%ebx # get the saved value of eax

movl %ebx,\_EAX(%eax) # store it

movl 0(%esp),%ebx # get return address from stack into ebx

movl %ebx,\_PC(%eax) # save it into the pc storage

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

movl \_EAX(%eax),%ebx # get new value for eax into ebx

movl %ebx,\_eax\_save # save it

movl \_EBX(%eax),%ebx # retore old registers

movl \_ECX(%eax),%ecx

movl \_EDX(%eax),%edx

movl \_ESI(%eax),%esi

movl \_EDI(%eax),%edi

movl \_EBP(%eax),%ebp

movl \_ESP(%eax),%esp # restore stack pointer

movl \_PC(%eax),%eax # restore return address into eax

movl %eax,4(%esp) # copy over the ret address on the stack

movl \_eax\_save,%eax

ret

SWITCH(Thread\*, Thread\*) 目的: 先把 %eax (register) 的內容 ''暫存在一個 data section (\_eax\_save)，然後把 %eax register 先拿來存指向 t1 的位址 (t1 stack 的起始位址)，接著一系列將 cpu registers ( %ebx, %ecx ,…, % )的''內容''存回 t1 的 stack。剩下最後一個位置 (4(%eax)) 還沒存到本來該存的內容(暫存在 \_eax\_save) 將其存回去。

* + 1. **(depends on the previous process state, e.g.,[New, Running, Waiting]→Ready**

**New → Ready :**表示該執行緒第一次進入ready queue。

**Ready → Running → Ready:** 就緒是指在該thread執行過程中觸發interrupt，將CPU的控制權交給另一個thread。

**Running → Waiting → Ready:** ready queue 意味者oldThread完成了它的工作並等待IO資源。

**Part 2:Implementation**

1. **Detail of your implementation**

**a. thread.h**

class Thread {

public :

void setBurstTime(int t) {burstTime = t;}

void setWaitingTime(int t) {waitingTime = t;}

void setExecutionTime(int t) {executionTime = t;}

void setPriority(int p) {priority = p;}

void setL3Time(int t){L3Time = t;}

int getBurstTime(){return (burstTime);}

int getWaitingTime(){return (waitingTime);}

int getExecutionTime(){return (executionTime);}

int getPriority(){return (priority);}

int getL3Time(){return (L3Time);}

private :

int burstTime;

int waitingTime;

int executionTime;

int L3Time;

int priority;

在 thread.h 新增 brustTime , waitTime , executionTime , priority 等參數，並且定義了一些 set 和 get 這些參數的 function。

**b. scheduler.h**

class Scheduler {

public :

void updatePriroty();

private :

SortedList<Thread \*> \*L2ReadyList;

在 scheduler 中不用原本的 List 而該用 SortedList (因要實作 priority queue)，並且定義*updatePriority()* 來實作當 thread 太久沒有執行時會增加 priority。

**c. scheduler.cc**

void Scheduler::updatePriority()

{

ListIterator<Thread \*> \*iter2 = new ListIterator<Thread\*>(L2ReadyList);

Statistics \*stats = kernel->stats;

int oldPriority;

int newPriority;

for(;!iter2->IsDone();iter2->Next())

{

ASSERT(iter2->Item()->getStatus()==READY);

iter2->Item()->setWaitingTime(iter2->Item()->getWaitingTime()+TimerTicks);

if(iter2->Item()->getWaitingTime() >=1500 && iter2->Item()->getID()>0)

{

oldPriority = iter2->Item()->getPriority();

DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter2->Item()->getID()<< "] changes its priority from [" << oldPriority << "] to ["<< newPriority << "]");

newPriority = oldPriority + 10;

if(newPriority>149)

{

newPriority = 149;

}

iter2->Item()->setPriority(newPriority);

L2ReadyList->Remove(iter2->Item());

ReadyToRun(iter2->Item());

}

}

}

會使用*updatePriority()* 來實作當 thread 太久沒有執行時會增加 priority，當 thread 沒有執行到時，會增加 WaitingTime 的時間，而當 WaitingTime 大於 1500 時會增加此 thread 的 priority。

(註:在 scheduler.cc 中因為我們改使用 SortedList 而非 List，故裡面的 List 都要改成 SortedList)

因為在 sortedList中要根據 priority 來進行排列 ，故在 scheduler.cc 中在定義一個比大小的 function，用以給 SortedList 中的 *insert()* function 所需的 function pointer。

static int compareL2(Thread \*t1, Thread \*t2)

{

if(t1->getPriority()> t2->getPriority()){

return -1;

}else if(t1->getPriority() < t2->getPriority()){

return 1;

} else {

return t1->getID() < t2->getID() ? -1:1;

}

return 0;

}

Scheduler::ReadyToRun (Thread \*thread)

{

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

DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

thread->setStatus(READY);

if(thread->getPriority() >= 0 && thread->getPriority() <= 149)

{

if(!L2ReadyList->IsInList(thread))

{

DEBUG('z' , "[A] Tick [" << kernel->stats->totalTicks << "]:Thread [" << thread->getID()<< "] is inserted into queue");

L2ReadyList->Insert(thread);

}

}

}

在 *ReadyToRun()* 中原本會使用 *append()* 將thread 加到 ready queue 的最後面，而在這邊改使用 *insert()* 根據 thread 的 priority 大小加入到 ready queue 裡面

**d. Alarm.cc**

void

Alarm::CallBack()

{

Interrupt \*interrupt = kernel->interrupt;

MachineStatus status = interrupt->getStatus();

kernel->scheduler->updatePriority();

Thread \*thread = kernel->currentThread;

thread->setExecutionTime(thread->getExecutionTime()+ TimerTicks);

thread->setL3Time(thread->getL3Time()+ TimerTicks);

if(kernel->currentThread->getID()>0 && status != IdleMode && kernel->currentThread->getPriority() >=149)

{

interrupt->YieldOnReturn();

}

if (status != IdleMode) {

interrupt->YieldOnReturn();

}

}

**當 timer interrupt 產生時會呼叫此 function，故我們會在此呼叫先前定義的 updatePriority()，且還會更新 thread 的 executionTime**

**e. Kernel.h**

class Kernel {

private:

Thread\* t[51];

int threadPriority[51];

char\* execfile[51];

新增 threadPriority 來記錄不同的執行檔的 priority

**f. Kernel.cc**

Kernel::Kernel(int argc, char \*\*argv)

{

else if (strcmp(argv[i],"-ep") == 0) {

execfile[++execfileNum]= argv[++i];

threadPriority[execfileNum] = atoi(argv[++i]);

if(threadPriority[execfileNum]>149){

threadPriority[execfileNum] = 149;

}

if(threadPriority[execfileNum]<0){

threadPriority[execfileNum] = 0;

}

}

}

新增了本作業所需的 -ep 指令，會將 execution file 的名字記錄起來外，還會接收 priority 的大小並記錄起來，其中我們還做了防呆機制，priority 會限定在我們定義的範圍內。

在 *Exec()* function 中新增了 priority 的 argument，並且在此 function 中進行參數的初始化，如 WaitingTime , Priority ,etc.

int Kernel::Exec(char\* name,int priority)

{

t[threadNum] = new Thread(name, threadNum);

t[threadNum]->setBurstTime(0);

t[threadNum]->setWaitingTime(0);

t[threadNum]->setExecutionTime(0);

t[threadNum]->setPriority(priority);

DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]:Thread [" << threadNum << "] changes its priority from [0] to ["<< priority << "]");

t[threadNum]->space = new AddrSpace();

t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void \*)t[threadNum]);

threadNum++;

return threadNum-1;

}

**g. Adding message**

**1. Whenever a process is inserted into a priority queue.**

**Scheduler::ReadyToRun()**

void

Scheduler::ReadyToRun (Thread \*thread)

{

DEBUG('z' , "[A] Tick [" << kernel->stats->totalTicks << "]:Thread [" << thread->getID()<< "] is inserted into queue");

L2ReadyList->Insert(thread);

}

呼叫*Insert()* function 時因出所需的 DEBUG message

**2. Whenever a process is removed from a queue.**

**Scheduler::FindNextToRun ()**

Thread \*

Scheduler::FindNextToRun ()

{

Thread \*a = L2ReadyList->RemoveFront();

DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]:Thread [" << a->getID()<< "] is removed from queue");

return a;

}

當使用 *RemoveFront()* 取出 ready queue 裡的 element 時印出 DUBUG message。

**3. Whenever a process change its scheduling priority.**

**Kernel::Exec()**

int Kernel::Exec(char\* name,int priority)

{

t[threadNum]->setBurstTime(0);

t[threadNum]->setWaitingTime(0);

t[threadNum]->setExecutionTime(0);

t[threadNum]->setPriority(priority);

DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]:Thread [" << threadNum << "] changes its priority from [0] to ["<< priority << "]");

}

當一開始呼叫 *Exec()* 時會初始化設定 thread 的 priority 故這邊會印出一次 DEBUG message。

**Scheduler::updatePriority()**

void Scheduler::updatePriority()

{

iter2->Item()->setWaitingTime(iter2->Item()->getWaitingTime()+TimerTicks);

if(iter2->Item()->getWaitingTime() >=1500 && iter2->Item()->getID()>0)

{

oldPriority = iter2->Item()->getPriority();

DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter2->Item()->getID()<< "] changes its priority from [" << oldPriority << "] to ["<< newPriority << "]");

newPriority = oldPriority + 10;

iter2->Item()->setPriority(newPriority);

}

}

}

當 process 超過一定的等待時間後會增加此 process 的 priority 故此時也會印出 DEBUG message

**4. Whenever a context switch occurs**

**Scheduler::Run()**

void

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

{

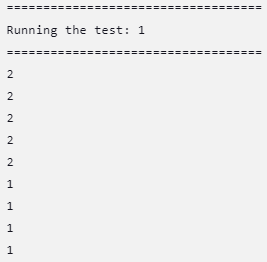
DEBUG('z', "[D] Tick [" << kernel->stats->totalTicks << "]:Thread [" << nextThread->getID()<< "] is now selected for execution, thread [" << oldThread->getID() << "] is replaced, and it has executed ["<< oldThread->getBurstTime() << "] ticks");

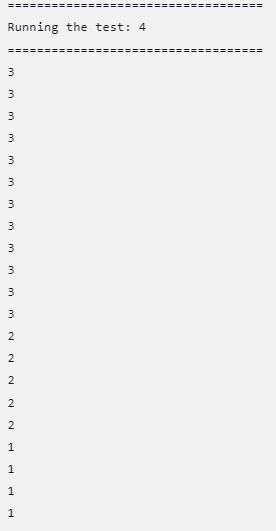
SWITCH(oldThread, nextThread);

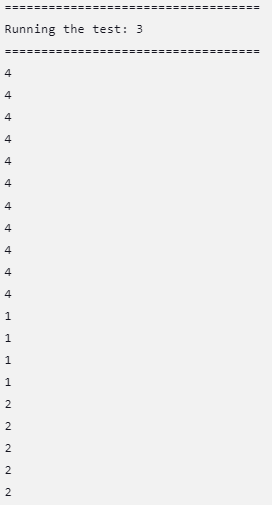
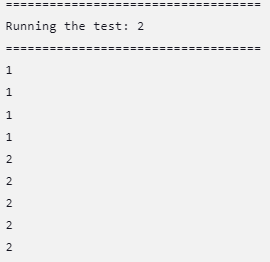
}

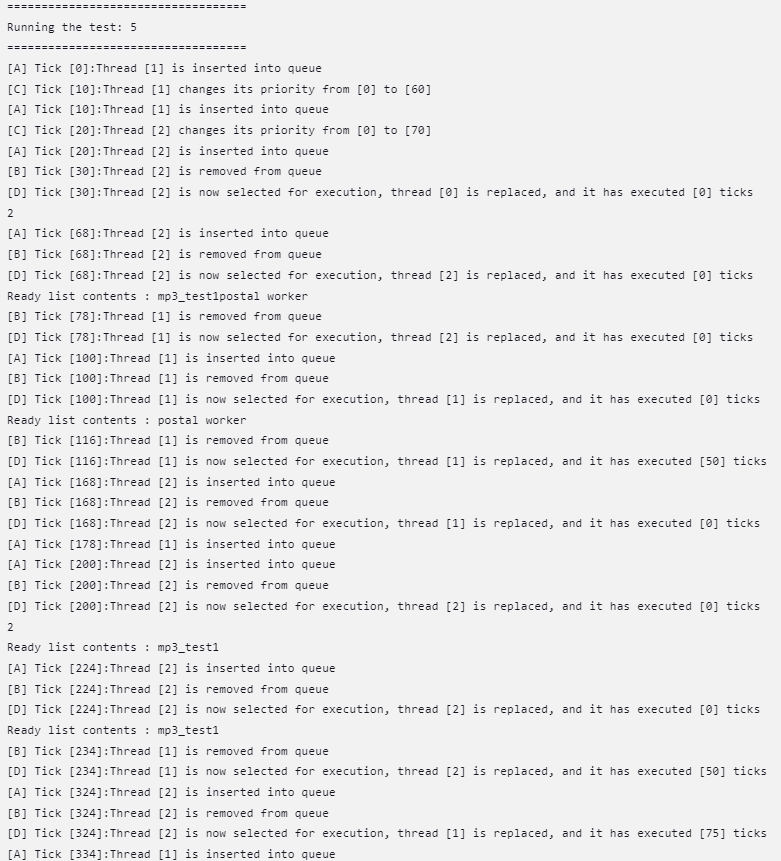
當呼叫 *SWITCH()* 時即產生 context switch，故印出 DEBUG message。

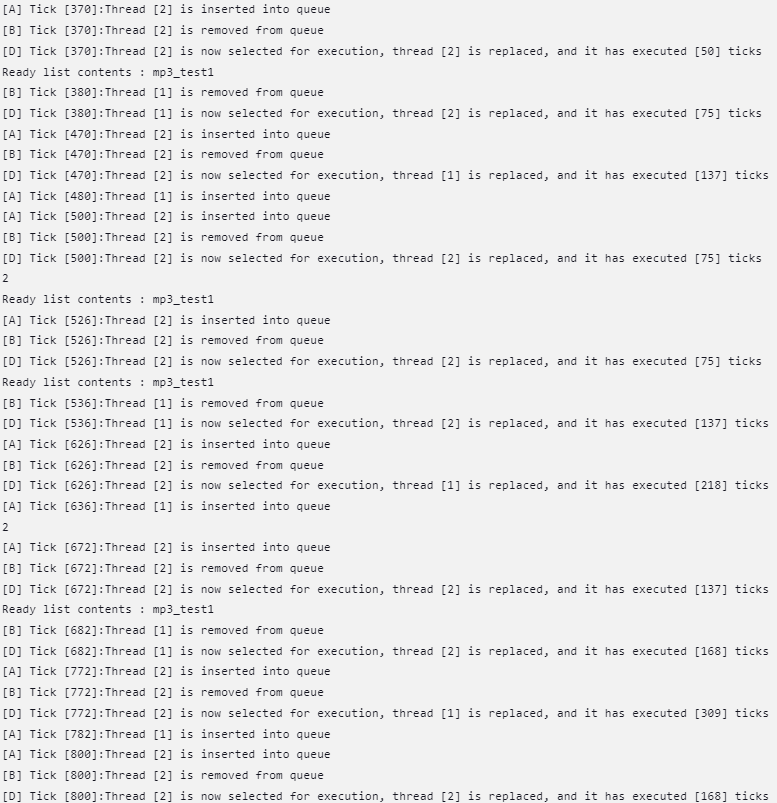
**輸出結果**

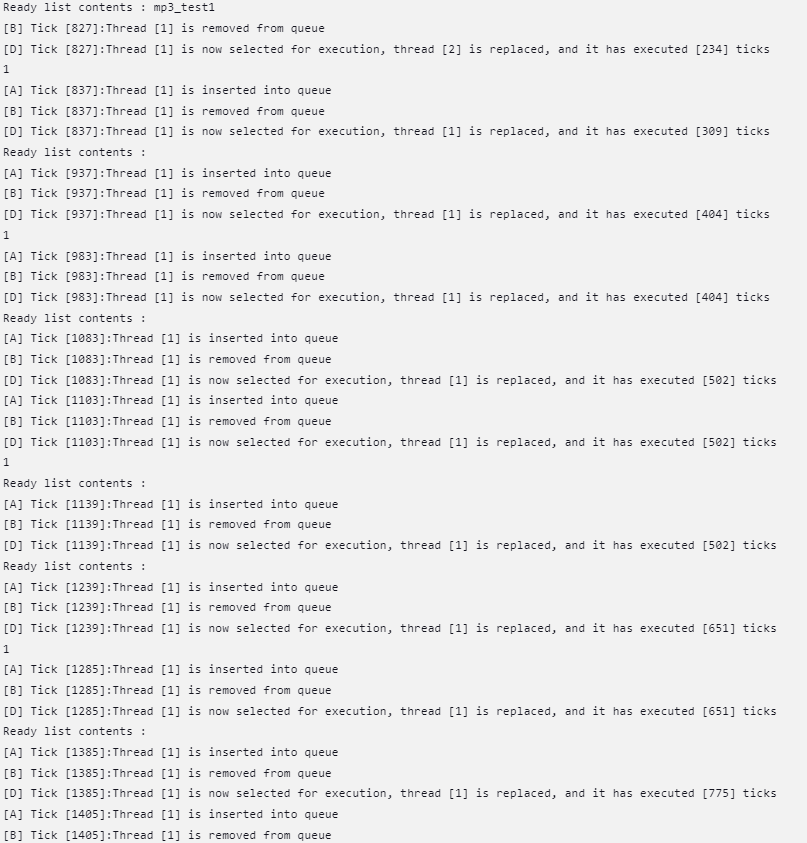
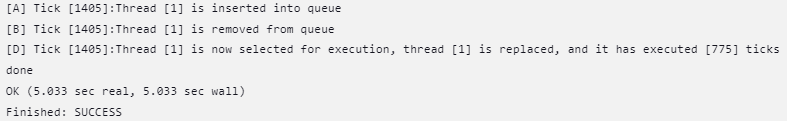
****

****

****

****

****

****

**Part 3:Contribution**

1. **Describe details and percentage of each member’s contribution.**

|  |  |  |
| --- | --- | --- |
|  | 吳孟儒 | 張世傑 |
| Part1. | 50 % | 50 % |
| Part2. | 50 % | 50 % |