OS MP3 Report

Team Member:

110062204 呂宜嫺: trace code，report 1.5 - 1.6，以及implementataion 2-3

110062239 侯茹文: trace code，report 1.1 - 1.4 & Implementation，以及implementataion 2-1、2-2

Trace Code Part

1.1 New -> Ready

- **Kernel::ExecAll()**

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

> 逐行執行execfile中的程式(call Exec()接續處理)，並在結束時call Finish()。

- **Kernel::Exec(char*)**

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

    return threadNum-1;
}
```

> 利用name以及threadID為目標程式建立一個新的thread，並存入陣列t。先進入AddrSpace()設定page的初始狀態，並進入fork進行後續的運作(ForkExecute)，最後回傳一個threads的編號。

- **Thread::Fork(VoidFunctionPtr, void*)**

```
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt; //獲取了現在的interrupt & scheduler
(以pointer的形式)
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg); // call StackAllocate() 初始化一些項目

    oldLevel = interrupt->SetLevel(IntOff); //oldlevel儲存了現在的interrupt
status，並將現在的status設為IntOff
    scheduler->ReadyToRun(this); //call ReadyToRun() 準備進入ready

    (void) interrupt->SetLevel(oldLevel); //跑完可以回到原本的status了
}
```

> fork一個新的thread，並進行一些初始化的操作以及利用StackAllocate()分配空間。
>
> 並且將interrupt狀態儲存後設為IntOff，以防後續操作(如switch)被interrupt影響。而後進入ReadyToRun()，會將此thread放入ready queue，並將其狀態設為ready，代表準備可以跑了。
>
> 結束後回到此function時，就可以將interrupt狀態設回來了。

- **Thread::StackAllocate(VoidFunctionPtr, void*)**

```
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int)); //分配空間給stread

#ifdef PARISC
    stackTop = stack + 16;  //由stack定義stacktop的位置
    stack[StackSize - 1] = STACK_FENCEPOST; //用來檢測是否溢出
#endif

... //基於不同的系統定義stackTop

#ifdef PARISC  //儲存不同數據的地址，並根據系統不同有不同的操作
    machineState[PCState] = PLabelToAddr(ThreadRoot); //first frame
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin) //current thread的起
始位置
    machineState[InitialPCState] = PLabelToAddr(func); //func 是要被fork的函數
    machineState[InitialArgState] = arg; //函數中會需要的參數
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish); //current thread的
結束位置
#else //因為不同架構分開定義
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
```

```
#endif
}
```

> 為了現在的new thread分配空間。先利用AllocBoundedArray()獲取空間，並計算stackTop(空間的boundary)。
>
> 而後根據現在的current thread初始化machine state (PCB)。

- **Scheduler::ReadyToRun(Thread*)**

```
Scheduler::ReadyToRun (Thread *thread)
{
   ASSERT(kernel->interrupt->getLevel() == IntOff); //確保不會被interrupt

   thread->setStatus(READY); //將status設為ready，準備進入ready queue

   readyList->Append(thread); //將現在的threads放入ready queue
}
```

> thread是可以進入ready queue的狀態了，將狀態設為ready並放入ready queue。

1.2 Running -> Ready

- **Machine::Run()**

```
Machine::Run()
{
    Instruction *instr = new Instruction;
    kernel->interrupt->setStatus(UserMode);
    for (;;) {

        OneInstruction(instr); //執行用戶指令
        kernel->interrupt->OneTick(); //因為一個instruction已經被執行了，所以要加一個
tick

    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
        Debugger();
    }
}
```

> 模擬用戶執行程序，在一開始就被kernel調用、不會返回。

- **Interrupt::OneTick()**

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

    if (status == SystemMode){  //根據當前的狀態(user or kernel)推進時間
        ...
    }

    ChangeLevel(IntOn, IntOff);  //關閉當前中斷
    CheckIfDue(FALSE); //檢查是否有應該被觸發的interrupt
    ChangeLevel(IntOff, IntOn);

    if (yieldOnReturn) { //是否需要context switch

    yieldOnReturn = FALSE;
    status = SystemMode;

    kernel->currentThread->Yield(); //switch到下一個在ready queue中的thread
    status = oldStatus;
    }
}
```

> 推進時間，並確認當下是否需要處理interrupt，以及判斷是否需要進行context switch。若是有需要處理的，都會call到相應function進行處理。

- **Thread::Yield()**

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

    ASSERT(this == kernel->currentThread);

    nextThread = kernel->scheduler->FindNextToRun(); //取ready list中第一個
    if (nextThread != NULL) {
    kernel->scheduler->ReadyToRun(this); //將currentThread的狀態設回ready，並放入
ready list
    kernel->scheduler->Run(nextThread, FALSE); //false表示old
thread(currentThread)還未執行完畢
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

> 將current thread放回ready queue(running->ready)，將next thread開始run(ready->run)。

- **Scheduler::FindNextToRun()**

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

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

> 看ready list是否有需要執行的，沒有就返回null；有就返回最上面的，並將其從ready list移除。

- **Scheduler::ReadyToRun(Thread*)**

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

    thread->setStatus(READY);
    readyList->Append(thread); //放入ready list，因為不需要排列所以用Append()
}
```

> 將thread放入ready list，並將狀態設為ready(run->ready)。

- **Scheduler::Run(Thread*, bool)**

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

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

    if (finishing) { //如果上一個threads(currentThreads)已經結束，就可以destroy (這是上面finish提到的要請cpu call destroy) 然後會在CheckToBeDestroyed()中被發現然後刪掉

         ASSERT(toBeDestroyed == NULL);
     toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) { //存現在threads的狀態，執行完新的以便換回來
        oldThread->SaveUserState();
    oldThread->space->SaveState();
    }

    oldThread->CheckOverflow(); //check stack overflow (stack overflow: 是否把
```

```
threads固定的stack空間用完了)

    kernel->currentThread = nextThread; //換成run下一個threads
    nextThread->setStatus(RUNNING);


    SWITCH(oldThread, nextThread); //switch到下一個thread，跑完後回到原本的

    ASSERT(kernel->interrupt->getLevel() == IntOff); //跑switch的threads是不能
interrupt的

    CheckToBeDestroyed(); //看是否有跑完的threads要被刪掉

    if (oldThread->space != NULL) { //恢復原本狀態 除非threads已經被destroy
        oldThread->RestoreUserState();
    oldThread->space->RestoreState();
    }
}
```

> 暫時停止執行現在的thread(或是現在的thread已經結束了)，並switch到下一個thread運作，然後回來。

1.3 Running -> Waiting

- **SynchConsoleOutput::PutChar(char)**

```
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire(); //把下面的鎖住，保證只有一個thread在執行(同步化)
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

> 因為要保證 console 不能被 interrupt，所以lock並且call waitFor->P()。

- **Semaphore:😛()**

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

    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) { //是否可以訪問resource
    queue->Append(currentThread); //讓現在的thread sleep直到resource
available(running->waiting)
```

```
        currentThread->Sleep(FALSE); //在現在的thread進入waiting時
        }

        value--; //獲得了resource，因此--

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

> 判斷是否有resource可以使用，如果沒有就進入waiting state，直到有resource可以使用。
>
> 而若是有，就需要將value--，代表有一個資源被占用。

- **List::Append(T)**

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

    ASSERT(!IsInList(item));
    if (IsEmpty()) {
    first = element;
    last = element;
    } else {
    last->next = element;
    last = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

> 將T的放入 List。List 的結構為單向的 linked list，此作為 queue (FIFS) 使用。

- **Thread::Sleep(bool)**

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

    ASSERT(this == kernel->currentThread); //確保kernel 中的thread 是this(現在這個)
    ASSERT(kernel->interrupt->getLevel() == IntOff); //透過getLevel()獲取現在的
interrupt狀態，而IntOff是用來確認是不是"現在所有interrupt"都被禁用
    // IntOff也是最高級別的interrupt，為了確保某些狀態下不會被interrupt影響


    status = BLOCKED; //將status設為block(waiting)

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) { //判斷接下
來有沒有要跑的
```

```
        kernel->interrupt->Idle();//如果暫時沒有要跑的，就進入idle狀態
    }
    kernel->scheduler->Run(nextThread, finishing);  //run接下來的thread
}
```

> 當現在的thread結束工作或是在等待時，會進入sleep的狀態。
>
> 判斷接下來是否有新的thread要執行，若是無就在idle狀態，而若是此時有新的thread，就會離開sleep的狀態，繼續執行下一個thread。

- **Scheduler::FindNextToRun()**

> 在ready list中尋找需要跑的，並回傳給sleep()，如果沒有就會回傳NULL，並進入Idle狀態。

- **Scheduler::Run(Thread*, bool)**

> 如果ready list 裡面有東西(脫離idle狀態)，就近到run跑。

1.4 Waiting -> Ready

> 在 Interrupt 進行 I/O 後進行 callback，最後將 SynchConsoleOutput 的 resource 釋出

- **Semaphore::V()**

```
{
    Interrupt *interrupt = kernel->interrupt;

    IntStatus oldLevel = interrupt->SetLevel(IntOff); //保證atomic

    if (!queue->IsEmpty()) {
    kernel->scheduler->ReadyToRun(queue->RemoveFront()); //讓正在等待 resource 的
thread 進到 ready (waiting->ready)
    }
    value++; //有離開waiting的，資源++

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

> 如果有正在等待 resource 釋出 (先前呼叫過 P()) 的 thread，因資源釋出，將其狀態設為 ready。

- **Scheduler::ReadyToRun(Thread*)**

> 將thread放入ready list，並將狀態設為ready (waiting->ready)。

1.5 Running -> Terminated

- **ExceptionHandler(ExceptionType) case SC_Exit**

```
case SC_Exit:
    DEBUG(dbgAddr, "Program exit\n");
            val=kernel->machine->ReadRegister(4); //讀取return value
            cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
```

> 當呼叫 Exit 的 System call 時會到這裡，call finish。

- **Thread::Finish()**

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);    // 使目前 Thread 從 running -> terminate
}
```

> 由running->terminate，接下來會進入sleep()，傳入true代表現在的thread已經執行完畢，之後在Run()不需要再另外存現在的狀態，並直接透過switch切換到下一個thread。

- **Thread::Sleep(bool)**

```
void Thread::Sleep (bool finishing)
{
    ...
    status = BLOCKED;    // state: terminate in this case (finishing ==
true)
    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);
}
```

> 處理terminate，並檢查接下來是否有需要跑的thread。

- **Scheduler::FindNextToRun()**

  > 尋找在 ready queue 中的 thread 作為 nextThread

- **Scheduler::Run(Thread*, bool)**

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

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

    ...
    SWITCH(oldThread, nextThread);

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

> 因 finishing == true，會將此 thread 進行 destroy

1.6 Ready -> Running

- **Scheduler::FindNextToRun() / Scheduler::Run(Thread*, bool)**

> 在 context switch 之前一定會經過這兩個 functions
>
> 分別可能會在 Thread::Yield 或 Thread::Sleep 中被執行
>
> 目的為尋找要 switch 的 thread (在 ready queue 中)
>
> 以及進行 SWITCH() 前需要的處理 (將下一個 thread 設為 currentThread 並切換 state 為 running

- **SWITCH(Thread*, Thread*) / depends on the previous process state / for loop in Machine::Run()**

```
/* 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    // when SWITCH() is called, PC will switch to here
```

```
    .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax (pointer to start
function)
        movl    4(%esp),%eax            # move pointer to t1 (4(%esp)) into eax
        movl    %ebx,_EBX(%eax)         # save registers (to prevent affecting by
other thread after switching)
        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
        # 以上為將 t1 在執行檔案用的 stack 中資料存入 memory
        # 從這裡開始將 t2 所需資料從 memory 搬到執行檔案用的 stack
        # 並將 PC 移到 t2 應該要開始的位置
        # 如果是第一次 Run 此 thread 會到 ThreadRoot
        # 否則會回到上一次 context switch 前的位置 (Scheduler::Run 的 SWITCH() 之後)
        movl    8(%esp),%eax            # move pointer to t2 (8(%esp)) 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
        # will return to the address where t2 should start
        ret
```

> 進行 machine level 的指標處理，將 register 中的資料進行 switch (t1 -> t2)，也將 return 後的位置變為 t2 應該要開始 thread 的位置
>
> 若 t2 是第一次執行 (new->ready)，會從 ThreadRoot 開始
>
> 若原本為 running (running->ready)，會依 (Scheduler::Run -> Thread::Yield -> Interrupt::OneTick -> Machine::Run) 的路徑回到原本執行 instruction 的位置
>
> 若原本為 waiting (waiting->ready)，會回到原本開始 waiting (呼叫 Thread::Sleep(FALSE)) 的位置 (e.g. 若是由 Semaphore:: P() 呼叫，則由 Sleep 回到 P)

```
        .globl  ThreadRoot    // 在 StackAllocate 中設定，使第一次 Switch 到該
Thread 時會從此開始執行
        .globl  _ThreadRoot

    /* 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()
*/
_ThreadRoot:
ThreadRoot:
        pushl   %ebp        // 在 StackPointer-4 的位置 push 此 Thread 的 base
pointer
        movl    %esp,%ebp   // move value in StackPointer to base pointer of this
thread
        pushl   InitialArg  // push InitialArg to stack for InitialPC function

        call    *StartupPC  //
        call    *InitialPC  // these address has been defined in
Thread::StackAllocate
        call    *WhenDonePC //

        # NOT REACHED
        movl    %ebp,%esp
        popl    %ebp
        ret
```

> 在第一次 Swtich 進入這個 thread 時，會進行以上 instructions，因為切換到這個 thread 之後的 PC 是指到 ThreadRoot。
>
> InitialPC () => Kernel::ForkExecute(Thread*)
>
> InitialArg (InitialPC routine 的參數) => 目標 thread
>
> StartupPC (在 thread 開始時呼叫) => Thread::ThreadBegin
>
> WhenDonePC (在 thread return 時呼叫) => Thread::ThreadFinish

Implementation

kernel.cc

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

> 增加一個"-ep"的判斷，用來讀取command line及priority，並將priority存到陣列中。

```
int priority[10]; //MP3
```

> kernel.h也需要建一個存被讀取的priority的陣列。

---

thread.cc

```
public: // 因為那些變數都是private，需要透過public function來access
    void setPriority(int newnum) { priority = newnum; }
    int getPriority() { return priority; }
    void setBurstTime(double newnum) { burstTime = newnum; }
    double getBurstTime() { return burstTime; }
    void setTotalTime(double newnum) { totalTime = newnum; }
    double getTotalTime() { return totalTime; }
    void setRunningBurstTime(double newnum) { runningBurstTime = newnum; }
    double getRunningBrustTime() { return runningBurstTime; }
    void setInRunningState(double newnum) { inRunningState = newnum; }
    double getInRunningState() { return inRunningState; }
    void setAgingTime(int newnum) { agingTime = newnum; }
    int getAgingTime() {return agingTime; }
    void setInReadyState(double newnum) { inReadyState = newnum;}
    double getInReadyState() { return inReadyState; }

  private: //MP3
    int priority; // 目前thread的priority
    double burstTime; // 預估算出的burst time
    double totalTime; // 目前cpu的burst time
    double runningBurstTime; // 目前cpu的burst time，與totalTime一樣，但totalTime的
更新較為頻繁，擔心出問題所以另外記錄一個。
    int agingTime; //紀錄aging tick，每1500 ticks會更新priority的
    double inReadyState; //進入ready state的質間
```

> 首先現在thread.h宣告這些變數，這些是用來記錄thread中的一些執行細項。
>
> 而這些會在Thread::Thread()進行初始化的動作，全部設為0。

```
Thread::Yield ()
{
    ...
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) { //確認有找到下一個要做的thread再繼續計算(才能確定這個
thread要結束)
    //MP3
    double nowRunningTime = kernel->currentThread->getRunningBrustTime() + kernel-
```

```
>stats->totalTicks - kernel->currentThread->getInRunningState(); //計算現在cpu burst(上一次running到現在的時間)
    kernel->currentThread->setTotalTime(nowRunningTime); //設定時間

    kernel->currentThread->setRunningBurstTime(nowRunningTime); //設定時間

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

    kernel->scheduler->ReadyToRun(this);
    nextThread->setInRunningState(kernel->stats->totalTicks); //在進入running state之前設定inRunningState的時間(紀錄現在時間)
    kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

> 這部分在處理由running進入waiting前，我們需要紀錄這一段burst time的時間，並將其加到runningBurstTime(紀錄waiting前的運作時間)，也更新totalTime。

```
Thread::Sleep (bool finishing)
{
    //MP3

    double nowRunningTime = kernel->currentThread->getRunningBrustTime() + kernel->stats->totalTicks - kernel->currentThread->getInRunningState(); //計算目前的時間
    double oldBurst = kernel->currentThread->getBurstTime(), t = nowRunningTime; //更新時間
    double newBurst = t * 0.5 + oldBurst * 0.5; //算出新的burst time(ti)

    kernel->currentThread->setTotalTime(0); //歸零時間，因為已經要進入IO了

    if (!finishing) DEBUG('z', "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << kernel->currentThread->getID() << "] update approximate burst time, from: [" << oldBurst << "], add [" << t << "], to [" << newBurst << "]");

    kernel->currentThread->setBurstTime(newBurst); //將計算好的burst time紀錄，讓下一次執行可以用來計算L1 priority
    kernel->currentThread->setRunningBurstTime(0);//歸零時間，因為已經要進入IO了

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    nextThread->setInRunningState(kernel->stats->totalTicks); //找到的nextThread要進入running state了，紀錄進入running state時間

    DEBUG('z', "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getID() << "] is now selected for execution, thread [" << kernel-
```

```
>currentThread->getID() << "] is replaced, and it has executed [" << t << "] ticks");finishing);
}
```

> current thread進入wait之前(running -> wait)，先更新這一輪的時間資訊。並且設定next thread進入running的時間，因為它也要進入running了。

---

scheduler.cc

```
private: //宣告L1, L2, L3的陣列
    SortedList<Thread *> *readyL1; //MP3
    SortedList<Thread *> *readyL2;
    List<Thread *> *readyL3;

    Thread *toBeDestroyed;

  public:
    void aging(); //計算是否需要aging
    bool PreemptiveCompare(); //比對是否需要進行preemprive
```

> 先在scheduler.h宣告三個list，其中因為L3不需要sorted，因此用一般的list宣告就好了。
>
> 並且宣告兩個會用到的functuon--aging()以及PreemptiveCompare()，用來計算是否需要aging以及preemptive。

```
Scheduler::Scheduler() //MP3
{
    readyL1 = new SortedList<Thread *>(compareL1);
    readyL2 = new SortedList<Thread *>(compareL2);
    readyL3 = new List<Thread *>;

    toBeDestroyed = NULL;
}

Scheduler::~Scheduler()
{
    delete readyL1;
    delete readyL2;
    delete readyL3;
}
```

> 在Scheduler constructer中宣告三個需要的list，而L1 L2因為需要sort，所以需要有compare的部分，如下。
>
> 上述三的list也需要再Scheduler被delete。

```
int compareL1(Thread *a, Thread *b) //MP3
{
    //分別計算出現在的remain burst time
    double aRemain = a->getBurstTime() - a->getRunningBrustTime();
    double bRemain = b->getBurstTime() - b->getRunningBrustTime();

    if(aRemain > bRemain) return -1;
    else if(aRemain < bRemain) return 1;
    else return 0;
}

int compareL2(Thread *a, Thread *b) //MP3
{
    //利用priority來比較
    if(a->getPriority() > b->getPriority()) return -1;
    else if(a->getPriority() < b->getPriority()) return 1;
    else return 0;
}
```

> 上述是用來sort L1及L2的compare function，一個使用remaining burst time計算，一個使用priority。

```
Scheduler::PreemptiveCompare()
{
    if(kernel->currentThread->getPriority() >= 100){
        if(!readyL1->IsEmpty()){
        //計算並比較兩者的remaining burst time
            Thread* first = readyL1->Front();
            double first_remain = first->getBurstTime() - first-
>getRunningBrustTime();
            double now_remain = kernel->currentThread->getBurstTime() - kernel-
>currentThread->getRunningBrustTime();

            if(first_remain < now_remain) return true;
        }
    }
    else if(kernel->currentThread->getPriority() < 100 && kernel->currentThread-
>getPriority() >= 50){
        if(!readyL1->IsEmpty()){ //L2只會被L1打斷
            return true;
        }
    }
    return false; //不會被打斷，繼續執行
}
```

> 用來計算現在的thread是否需要被preemptive，首先先判斷是否是L1中的在執行，如果是就檢查current thread以及list中最前面的remaining burst time，如果後者較小，就需要進行preemptive，回傳-1。
>
> 而判斷是L2範圍內的，就需要檢查現在L1內是否有thread在，如果有就應該被其打斷。

```
Scheduler::aging() //MP3
{
    //遍歷三個list，並一個一個檢查是否有thread需要aging
    //因為操作相似，我就以L1為主要說明

    if(!readyL1->IsEmpty()){
        ListIterator<Thread *> *iter = new ListIterator<Thread*>(readyL1); //獲取ready的pointer

        for(; !iter->IsDone(); iter->Next()){ //進行遍歷檢查
            Thread* now = iter->Item(); //正在被檢查的thread

            if((kernel->stats->totalTicks - now->getAgingTime()) >= 1500 && now->getPriority() < 149){ //確認現在的thread aging值是否到達1500，而如果現在它的priority已經149了也不需要在aging

                now->setAgingTime(kernel->stats->totalTicks);  //更新aging time為現在時間，重新計算1500 ticks
                int oldP = now->getPriority(),
                newP = (oldP + 10 > 149) ? 149 : oldP + 10; //更新現在的priority(+10)，如果超過149就設為149

                DEBUG('z', "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] changes its priority from [" << oldP << "] to [" << newP << "]");
                now->setPriority(newP); //設為新的priority

            }
        }
    }
    if(!readyL2->IsEmpty()){
        ListIterator<Thread *> *iter = new ListIterator<Thread*>(readyL2);

        for(; !iter->IsDone(); iter->Next()){
            Thread* now = iter->Item();

            if((kernel->stats->totalTicks - now->getAgingTime()) >= 1500){
                now->setAgingTime(kernel->stats->totalTicks);
                int oldP = now->getPriority(),
                newP = (oldP + 10 > 149) ? 149 : oldP + 10;

                DEBUG('z', "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] changes its priority from [" << oldP << "] to [" << newP << "]");
                now->setPriority(newP);

                DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is removed from queue L[2]");
                readyL2->Remove(now); //因為L2有可能往L1前進，所以先remove

                if(now->getPriority() >= 100) {
                    DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is inserted into queue L[1]");
```

```
                    readyL1->Insert(now); //priority超過100的就insert進入L1
                }
                else {
                    DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << now->getID() << "] is inserted into queue L[2]");
                    readyL2->Insert(now); //priority小於100的就insert回L2
                }
            }
        }
    }
    if(!readyL3->IsEmpty()){
        ListIterator<Thread *> *iter = new ListIterator<Thread*>(readyL2);

        for(; !iter->IsDone(); iter->Next()){
            Thread* now = iter->Item();

            if((kernel->stats->totalTicks - now->getAgingTime()) >= 1500){
                now->setAgingTime(kernel->stats->totalTicks);
                int oldP = now->getPriority(),
                newP = (oldP + 10 > 149) ? 149 : oldP + 10;

                DEBUG('z', "[C] Tick [" << kernel->stats->totalTicks << "]: Thread 
[" << now->getID() << "] changes its priority from [" << oldP << "] to [" << newP 
<< "]");
                now->setPriority(newP);

                DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread 
[" << now->getID() << "] is removed from queue L[3]");
                readyL3->Remove(now);

                if(now->getPriority() >= 50) {
                    DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << now->getID() << "] is inserted into queue L[2]");
                    readyL2->Insert(now); //prioriy超過50的就insert進入L2
                }
                else {
                    DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: 
Thread [" << now->getID() << "] is inserted into queue L[3]");
                    readyL3->Append(now); //如果小於50就照舊，因為L3不用sort，所以用
Append()
                }
            }
        }
    }
}
```

> 遍歷三個list檢查是否有thread需要被更新priority，還需要另外確認是否有thread超過149，又或是因為priority的提高而需要換list。

```
Scheduler::ReadyToRun (Thread *thread)
{
```

```
    ...
    thread->setStatus(READY);
    thread->setAgingTime(kernel->stats->totalTicks); //因為進入了ready，將aging
time設為現在，用來判斷aging時間

    //根據現在的priority去分配需要進入哪一個list
    if(thread->getPriority() < 50){
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[3]");
        readyL3->Append(thread);
    }
    else if(thread->getPriority() >= 100){
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[1]");
        readyL1->Insert(thread);
    }
    else{
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[2]");
        readyL2->Insert(thread);
    }
}
```

> 因為進入ready state，所以需要更新aging的值(需要用來判斷進入ready state多久來提升priority)。
>
> 而後依據其priority來分配它要進入哪一個list(L1 or L2 or L3)。

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

    //由L1開始判斷是否為空，如果不是就直接回傳第一個thread
    if(!readyL1->IsEmpty()){
        Thread *ret = readyL1->RemoveFront();
        DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
ret->getID() << "] is remove from queue L[1]");
        return ret;
    }
    else if(!readyL2->IsEmpty()){
        Thread *ret = readyL2->RemoveFront();
        DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
ret->getID() << "] is remove from queue L[2]");
        return ret;
    }
    else if(!readyL3->IsEmpty()){
        Thread *ret = readyL3->RemoveFront();
        DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
ret->getID() << "] is remove from queue L[3]");
        return ret;
    }
    else return NULL; //L1 L2 L3皆為空，回傳NULL
}
```

> 需要回傳ready list中下一個要被執行的值，因此由priority最高的來判斷，如果為空就接續往下，全都為空就回傳NULL；而若是不為空，就會回傳該list的front，也就是list中priority最高的(除了L3不是最高)。

---

alarm.cc

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

    //先進行aging，判斷priority是否需要被更改
    Scheduler *scheduler = kernel->scheduler;
    scheduler->aging();

    if (status != IdleMode) {

        if(scheduler->PreemptiveCompare()){  //進行preemptive compare判斷L1是否需要進行context switch, L1需要判斷remain burst time, L2需要判斷L1現在是否為空
            interrupt->YieldOnReturn(); //需要context switch 就會call YieldOnReturn()來改變currentThread
        }
        if(kernel->currentThread->getPriority() < 50){ //如果是L3就只需判斷現在是否執行了100 ticks

            kernel->currentThread->setTotalTime(kernel->currentThread->getRunningBrustTime() + kernel->stats->totalTicks - kernel->currentThread->getInRunningState()); //先更新現在運作的total tick
            if(kernel->currentThread->getTotalTime() >= 100) interrupt->YieldOnReturn(); //如果運作超過100 ticks就可以call YielrOnReturn()
        }
    }
}
```

> 這一部分有兩個功能: 檢查aging以及判斷是否需要context switch。Aging的部分就call之前寫在scheduler中的aging function。
>
> 判斷context switch的部分分成三個list去判斷，L1 L2在preemepriveCompare()中，前者透過檢查list->front()的remaining burst time來判斷是否有需要context switch(如果L1->Front()比較小就會context switch)。而L2只需判斷L1現在是否有thread，有就需要讓給他執行，也需要context switch。
>
> 而L3我直接在callback中判斷，先更新它的total tick(因為只在yield()中更新這邊無法得知現在的運行多久了)，然後再判斷是否>100(spec_v2更新的條件)，如果>100就會call YieldOnReturn。