MP3_report_36

109062233蘇裕恆：

50%trace code 50% implementation

109062320朱季葳：

50%trace code 50% implementation

Trace Code

1-1. New→Ready

KERNEL::EXECALL()

這裡有一個for loop去iterate through execfile裡面的檔案並call Kernel::Exec()去執行他們，然後在執行完所有檔案之後，把現在在CPU的thread結束執行（call Finish()）。

KERNEL::EXEC(CHAR*)

在這邊，程式create一個thread並allocate address space給他(create new AddrSpace)，再將原本的execute的東西pass到Fork裡面讓Fork call StackAllocate去分空間給thread的動作，最後將threadnum++ 並將create的thread number 回傳。

THREAD::FORK(VOIDFUNCTIONPTR, VOID*)

在這裡程式invoke傳進來的(*func)(arg)，在這裡的意思不是call那個函式的意思，而是可以讓caller(Fork)以及callee(ForkExecute)可以concurrent的執行。
而在這個部分他call了StackAllocate()去allocate空間給thread(因為每個thread都有自己的stack，不與同個process內的其他threads共用)，如同我們在Kernal::Exec()的部分所解釋的一樣，然後去disable interrupt（來避免在ReadyToRun()的時候被打斷），並call ReadyToRun()把這個thread加到ready list裡面，最後再把interrupt enable回來。

THREAD::STACKALLOCATE(VOIDFUNCTIONPTR, VOID*)

```
306   void
307   Thread::StackAllocate (VoidFunctionPtr func, void *arg)
308   {
309       stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
310
311   #ifdef PARISC
312       // HP stack works from low addresses to high addresses
313       // everyone else works the other way: from high addresses to low addresses
314       stackTop = stack + 16;  // HP requires 64-byte frame marker
315       stack[StackSize - 1] = STACK_FENCEPOST;
316   #endif
317
318   #ifdef SPARC
319       stackTop = stack + StackSize - 96;  // SPARC stack must contains at
320                                           // least 1 activation record
321                                           // to start with.
322       *stack = STACK_FENCEPOST;
323   #endif
324
325   #ifdef PowerPC // RS6000
326       stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
327       *stack = STACK_FENCEPOST;
328   #endif
```

基本上他會先將一個stack 用AllocateBoundedArray()創造出來

並會依據不同的ISA來做判斷 以本次平台為例 使用UNAME -A(會印出系統的所有的資訊) 可以發現使用的架構為X86

```
x86_64
[os22team36@localhost ~]$ uname -a
Linux localhost.localdomain 3.10.0-1160.45.1.el7.x86_64 #1 SMP Wed Oct 13 17:20:51 UTC 2021 x86_64 x86_64 x86_64
GNU/Linux
```

因此stacktop等地配置為下。

```
#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
```

最後將machineState的配置設定完成並即結束整個StackAllocate的流程，而machineState會在之後進行context switch的時候派上用場。

```
350   #ifdef PARISC
351       machineState[PCState] = PLabelToAddr(ThreadRoot);
352       machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
353       machineState[InitialPCState] = PLabelToAddr(func);
354       machineState[InitialArgState] = arg;
355       machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
356   #else
357       machineState[PCState] = (void*)ThreadRoot;
358       machineState[StartupPCState] = (void*)ThreadBegin;
359       machineState[InitialPCState] = (void*)func;
360       machineState[InitialArgState] = (void*)arg;
361       machineState[WhenDonePCState] = (void*)ThreadFinish;
362   #endif
363   }
```

SCHEDULER::READYTORUN(THREAD*)

```
56    void
57    Scheduler::ReadyToRun (Thread *thread)
58    {
59        ASSERT(kernel->interrupt->getLevel() == IntOff);
60        DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
61        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
62        thread->setStatus(READY);
63        readyList->Append(thread);
64    }
```

這邊將這個thread的status設為ready (但還沒有跑)

並將這個thread加到Scheduler::readyList裡面(call

Append(thread)把這個thread加到list的尾巴。

1-2. Running→Ready

MACHINE::RUN()

讓machine開始跑起來，基本上一個thread會有一個。並且不停的執行 one instruction(找指令並執行他們，詳細的實作細節在MP1的report有提到) & one tick

INTERRUPT::ONETICK()

會有兩種事物可以call到oneTick

1. interrupts are re-enable

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

    // interrupt handlers are prohibited from enabling interrupts
    ASSERT((now == IntOff) || (inHandler == FALSE));

    ChangeLevel(old, now);          // change to new state
    if ((now == IntOn) && (old == IntOff)) {
    OneTick();              // advance simulated time
    }
    return old;
}
```

2. a user instruction is called

在這個function裡面主要有下列幾項功能

1. 依據當前的mode(system mode or user mode)去將clock advance
2. call CheckIfDue()確認是否有interrupt即將發生，有的話就執行它
3. 確認會不會需要把現在正在使用CPU的thread的使用權yield出去(yieldOnReturn == true)，因為系統是time-sharing system，所以到了一定的時間，Timer就會raise interrupt，然後call Alarm::CallBack() (因為timer是Alarm的private member，所以call到的CallBack()是Alarm::CallBack()) ，而在這個function裡面會在有runnable thread在ready list裡的時候call YieldOnReturn()，而YieldOnReturn()會把yieldOnReturn設為ture。
4. 如果yieldOnReturn == true，call Yield()把現在正在使用CPU的thread加回ready list的尾巴，並Run下一個thread

THREAD::YIELD()

Yield 會先檢查目前的thread一不一樣。並執行另一個thread。他會先首先禁用interrupt。然後，他會call findNextToRun找到要執行的下一個thread ，並將現在的thread加回到ready list，再Run next thread。最後，將interrupt設回原本(上述的步驟為critical section)。

SCHEDULER::FINDNEXTTORUN()

他會check readyList是否為empty，如果是的話就會直接return NULL。
不是empty則代表有要執行的thread，就會return那個thread，並且將他在readyList刪除。

SCHEDULER::READYTORUN(THREAD*)

詳情請見1-1 Scheduler::ReadyToRun(Thread*)

- **ReadyToRun**

SCHEDULER::RUN(THREAD*, BOOL)

這個function的主要目的是為了Run傳進來的NextThread，而這裡主要分為兩種情況

- 在原本的thread還沒有跑完的情況下切到下一個thread(此處的情況)
- 原本的thread已經finish了(finishing = true)。

而假設oldThread是User program，我們就需要call Thread::SaveUserState()將他的registers記錄起來，因為user program會有兩組CPU register，分別是給user mode跟kernel mode，而透過SaveUserState()可以只用一組CPU register來實作，因為每當有context switch發生的時候都會把該user program的CPU register的值存下來。
接下來確認oldThread確認有沒有overflow，然後將currentThread設為next thread 並將他的Status設為running。
下一步，call switch，詳情請見Switch.s 把Oldthread換回來(基本上就是將整個stack pointer與其他pointer的位置調換掉)
最後call CheckToBeDestroyed()來將已經finish(toBeDestroyed!=NULL)的thread delete掉。
假如oldThread沒有被delete掉的話，將CPU register的資料以及PageTable, PageTableSize restore回去。

switch

The code first saves the current value of the eax register on the stack, so that it can be used as a pointer to the first thread (t1) being switched from. It then saves the values of the other registers (ebx, ecx, edx, esi, edi, and ebp) from the first thread on the stack, as well as the stack pointer. It also saves the saved value of eax from the first thread on the stack.
Next, the code gets the return address from the stack and saves it in the pc storage for the first thread. It then moves the pointer to the second thread (t2) into the eax register, and restores the saved value of eax for the second thread into the eax register. It then restores the values of the other registers and the stack pointer from the second thread.
Finally, it restores the return address from the second thread and copies it over the return address on the stack, and returns from the function. This completes the thread switch, allowing the second thread to start executing.

simple conclusion

簡單來說，running->ready要被trigger到需要一些interrupt。而因為整個NachOS 是time sharing system，因此我們會需要使用onetick來處理時間與在特定時間trigger yield來保證不會有單一個thread佔用CPU太久(CPU Protection)。
大致上我們可以看到run-> ready必須要yield。而在裡面會先將interrupt關掉並且找到要找的。並執行switch來模擬context switch...

1-3. Running→Waiting (Note: only need to consider console output as an example)

1-3 與 1-4 會先大致上講過，接著會統一有一個overall review來描述整個putchar

SynchConsoleOutput::PutChar(char)

這個是user call 並在thread被執行的一個程序。主要來說就是要將一個character寫到console display 並會讓thread 變成waiting state。
就像是其他critical part一樣，他會需要lock來確保一次只會有一個thread來執行(在這邊是等到該thread拿到lock時才會

繼續執行以下的critical section)，並在處理完output(call consoleOutput->PutChar(ch)來將characters寫到一個file，然後把這個I/O interrupt加入排程)和synchronization(call P())以後將lock release掉。
整體來說，他完美的解決了同步的問題。

SEMAPHORE::P()

它是一個處理同步化的atomic function，而因為semaphore裡面的data(value)會被多個thread存取到，所以要把會用到value的地方包起來(critical section)。
然後在CS裡面，如果（value == 0）的話，代表現在沒有可以用的semaphore，所以在把currentThread加進去semaphore的waiting queue之後，就讓這個thread put to sleep。
反之，當while loop break的時候(value > 0)，把value−並取得該semaphore的使用權。

LIST T::APPEND(T)

他就跟一般的list差不多，但比較不一樣的是他在放進去之前會先檢查這個物品是否已經在list裡面。如果沒有，就用one way link list的方式將它放到最後面。

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);
}
```

在這裡他會先把currenThread block掉，然後call FindNextToRun( )，並於return值為NULL(代表readyList裡面沒有thread)的時候把CPU idle掉，直到readyList裡面有thread為止，而這個thread只能是做完I/O之後從waiting state回到ready state的thread，因為在idle()裡面程式會call checkIfDue()來確保pending interrupts有被處理到，而如果沒有pending interrupt的話，他則會return false，然後idle()

會Halt掉整個kernel。而值得注意的點是，sleep是不能被打斷的，所以無論在任何的情況下call sleep，都要先確保interrupt有被disable掉。

SCHEDULER::FINDNEXTTORUN()

find next thread in ready list to run
詳情請見1-2 Scheduler::FindNextToRun()

SCHEDULER::RUN(THREAD*, BOOL)

run the new thread。
詳情請見1-2 Scheduler::Run(Thread*, bool)

simple conclusion

我們大致上可以將整個putchar分為兩種類型
1. 沒有別人要印
2. 有人要印

如果是後者的話，我們就會遇到以上的過程。因為I/O必須為critical，在nashos裡面我們使用的是lock & semaphore 來達到的。

1-4. Waiting→Ready (Note: only need to consider console output as an example)

SEMAPHORE::V()

在handle console output的interrupt的過程中，在checkIfDue()裡面會先確認有沒有可執行的pending interrupt，然後call該interrupt的callback()去handle interrupt，而在這裡會先call到ConsoleOutput::CallBack()，而在這個function裡面會再call到SynchConsoleInput::CallBack()，而在這裡面會call到這個function。詳細的code如下面幾張圖所示

- SynchConsoleInput::SynchConsoleInput(char *inputFile)

```
SynchConsoleInput::SynchConsoleInput(char *inputFile)
{
    consoleInput = new ConsoleInput(inputFile, this);
    lock = new Lock("console in");
    waitFor = new Semaphore("console in", 0);
}
```

在這裡會new 一個ConsoleInput並把自己（SynchConsoleInput object傳入）

- ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)

```
ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
{
    if (writeFile == NULL)
    writeFileNo = 1;                // display = stdout
    else
        writeFileNo = OpenForWrite(writeFile);

    callWhenDone = toCall;
    putBusy = FALSE;
}
```

在這裡會把callWhenDone設為toCall(SynchConsoleInput object)

- ConsoleOutput::CallBack()

```
void
ConsoleOutput::CallBack()
{
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

在這裡會call到callWhenDone->CallBack()，所以會對應到SynchConsoleInput::CallBack()

- SynchConsoleInput::CallBack()

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

在這裡會call到Semaphore::V()
以上就是call到Semaphore::V()所會經過的路徑。
而這個Semaphore::V()與Semaphore::P()對稱，在這個function中也會處理同步化問題（理由在1-3Semaphore::P()有提到過了）。
首先，他會先將interrupt disable (同 P() 因為是一個 atomic function ）。接著把Semaphore::queue最前面的thread加回ready list，然後再把value+1來free掉這個semaphore給其他thread，最後將interrupt enable回來。

SCHEDULER::READYTORUN(THREAD*)

同之前講過的 將readylist append上這個thread。
詳情請見1-1 Scheduler::ReadyToRun(Thread*)

overall review on the critical section

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

我們在putchar裡面會將lock傳給這個thread。那同時會call到consoleOutput的putchar。這邊會們會call write將字母寫過去。

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

同時間putBusy = True 這樣子如果其他thread也想要putChar就會產生alert。接著我們會把一個interrupt傳到scheduler裡面。

這邊他會先處理putchar的waitfor->P()等待到resource夠了以後，就會將整個lock release

(此發生在下個clk)

```
void
ConsoleOutput::CallBack()
{
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

在此我們會call一個callback並且將putBusy設為false。

同時會叫到semaphore::v()

大致上synchronize就是這樣

1-5. Running→Terminated (Note: start from the Exit system call is called)

ExceptionHandler(ExceptionType) case SC_Exit

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

在這裡程式call了finish()來把currentThread finish掉

THREAD::FINISH()

在Thread::Finish()中，他將interrupt disable掉並且call Sleep() (因為Sleep 需要再interrupt 的status為off)，而在這裡的註解中還有提到為什麼不直接de-allocate掉thread，而是call sleep之後再間接de-allocate掉thread。這是因為在這個階段程式可能還正在跑這個thread或者還在stack上，所以在之後檢查到程式正在跑其他thread的或者這個thread finish的時候才會de-allocate掉這個thread(詳情請見Scheduler::Run()的解說）。

```
170   void
171   Thread::Finish ()
172   {
173       (void) kernel->interrupt->SetLevel(IntOff);
174       ASSERT(this == kernel->currentThread);
175
176       DEBUG(dbgThread, "Finishing thread: " << name);
177       Sleep(TRUE);                // invokes SWITCH
178       // not reached
179   }
180
```

THREAD::SLEEP(BOOL)

把該thread put to sleep然後find next thread in ready list to run。
詳細解說請見1-3 Thread::Sleep(bool)。

SCHEDULER::FINDNEXTTORUN()

find next thread in ready list to run
詳情請見1-2 Scheduler::FindNextToRun()

SCHEDULER::RUN(THREAD*, BOOL)

詳情請見1-2 Scheduler::Run(Thread*, bool)，而此處的狀況與1-2的不太一樣，在這裡thread已經finish了，所以CheckToBeDestroyed()會把他delete掉，而詳細情形在1-2都

有提到所以就不再贅述了。

1-6. Ready→Running

SCHEDULER::FINDNEXTTORUN()

find next thread in ready list to run
詳情請見1-2 Scheduler::FindNextToRun()

SCHEDULER::RUN(THREAD*, BOOL)

在這裡有兩種情況

1. 因為context switch所以在ready list的thread的state由Ready轉為Run
2. 前一個thread跑完了，所以run next thread
   此兩種情況的不同會造成oldThread的delete與否，詳細的情形可以在1-2 Scheduler::Run(Thread*, bool)看到

SWITCH(THREAD*, THREAD*)

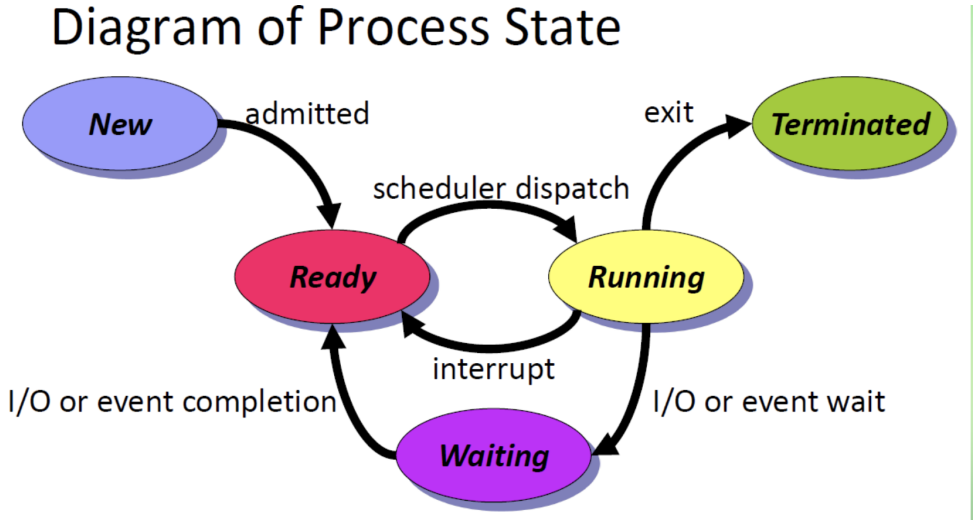
1. 在switch.h那邊會定義好一些在switch.s所對應到的registers，而我們在Thread::StackAllocate(VoidFunctionPtr, void*)的解說中有提到「machineState會在之後進行context switch的時候派上用場」，而在switch.h的定義可以和StackAllocate相對應，如下圖

```
/* These definitions are used in Thread::AllocateStack(). */
#define PCState         (_PC/4-1)
#define FPState         (_EBP/4-1)
#define InitialPCState  (_ESI/4-1)
#define InitialArgState (_EDX/4-1)
#define WhenDonePCState (_EDI/4-1)
#define StartupPCState  (_ECX/4-1)

#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx
```

2. 在switch.S的ThreadRoot裡面，會call到machine state裡面initial的function(by setting program counter to the value of registers，而在switch.h有定義好那些function對應到的registers)
3. 執行完new thread的initial function之後，透過movl指令把old thread的data存起來，然後restore new thread的data，使得新的thread可以開始跑。

(DEPENDS ON THE PREVIOUS PROCESS STATE, E.G., [NEW,RUNNING,WAITING]→READY)



由上圖，我們可以得知只有三種state可以進到ready state:New, Waiting, Running，而我們將針對這三種previous state進行ready->running的case解說。

1. New -> Ready:詳細情形可以從1-1的解說看到
2. Running -> Ready:詳細情形可以從1-2的解說看到
3. Waiting -> Ready:詳細情形可以從1-4的解說看到
   而第2,3都是因為做了context switch所以才被中斷的，所以程式要去restore他們的state還有一些value像是program counter等等的並繼續執行他們，而第1的部分只需要從頭開始執行就好了。

FOR LOOP IN MACHINE::RUN()

讓machine開始跑起來並執行program的instructions，詳情在1-2 Machine::Run()可以看到

Implementation

CODE/LIB/DEBUG.H

```
// The pre-defined debugging flags are:

const char dbgAll = '+';        // turn on all debug messages
const char dbgThread = 't';     // threads
const char dbgSynch = 's';      // locks, semaphores, condition va
const char dbgInt = 'i';        // interrupt emulation
const char dbgMach = 'm';       // machine emulation
const char dbgDisk = 'd';       // disk emulation
const char dbgFile = 'f';       // file system
const char dbgAddr = 'a';       // address spaces
const char dbgNet = 'n';        // network emulation
const char dbgSys = 'u';                // systemcall
const char dbgTraCode = 'c';
const char dbgMP3 = 'z'; // add MP3 109062233
```

在這裡我們為了配合2-3所要求的功能，所以在pre-defined debugging flag加上**dbgMP3 = 'z'**

CODE/THREAD/KERNEL.*

- kernel.h
  我們在這邊為了配合thread跟所輸入的初始priority，所以新增一個可以傳入thread name還有priority的Exec()如下圖

```
int Exec(char* name);
int Exec(char* name , int thread_priority); // MP3 109062233 modified
```

因為每個thread都有自己的priority，所以我們新增一個紀錄priority的array如下圖

```
// add MP3 109062233
int thread_priority[10];
```

- **kernel.cc** (http://kernel.cc)
  我們首先在kernel的constructor裡面仿照command line argument -e加入command line argument -ep，並加入對每個thread priority的設置。

```
// begin adding
else if (strcmp(argv[i], "-ep") == 0){
    execfile[++execfileNum]= argv[++i];
    thread_priority[execfileNum]= atoi(argv[++i]);
    cout << execfile[execfileNum] << "has the priority of " << thread_priority[execfileNum] <<"\n" ;
    // check priority is valid
    ASSERT(thread_priority[execfileNum] >= 0 && thread_priority[execfileNum] <= 149);
} // add MP3 109062233
```

然後在ExecAll()的部分call到Exec(char* name , int thread_priority)

```
void Kernel::ExecAll()
{
    // modified MP3 109062233
    /*for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i]);
    }*/
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i] , thread_priority[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

而在Exec(char* name , int thread_priority)之中，相較於原本的，我們在fork之前call了set priority來設置thread的priority，而在thread的constructor中只有單純把priority initialize為0，並在這裡才set priority

```
// MP3 modified by 109062320
// MP3 109062233 revised the exec function to fit priority
int Kernel::Exec(char* name , int priority)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->setPriority(priority);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);

    threadNum++;

    return threadNum-1;
}
```

CODE/THREAD/SCHEDULER.*

- scheduler.h
  我們在這裡把原本的ready list註解掉，並加入SortedList L1, L2還有L3，因為L1以及L2都有自己排序的規則所以使用sortedList，而round robin則沒有，時間到了就打斷換thread輪流執行。

```
// List<Thread *> *readyList;  // queue of threads that are ready to run,
    // but not running
SortedList<Thread *> *L1_list;
SortedList<Thread *> *L2_list;
List<Thread *> *L3_list;

// end adding MP3 109062233
```

而在這裡我們額外加入了三個成員函數，分別是用於call aging的update_priority, 在priority updated之後將L2和L3的thread的priority都檢查一遍並作重新排序的scheduling，以及將L1, L2的List做re-sort的Re_Sort_List。

```
void update_priority(); // MP3 109062233
void scheduling();// MP3 109062320
void Re_Sort_List();
```

- **scheduler.cc** (http://scheduler.cc)
  - comparison functions
    - sjf:依據spec的更新，我們在比較之後，將approximated remaining burst time較低的thread排在比較前面。
    - prior_cmp:將priority較高的thread擺在比較前面
  - Schedular()
    - initialize L1, L2, L3
  - update_priority()
    - 先檢查各個list是否為empty，不是的話再iterate through list中的每個thread，然後call他們各自的aging function(aging function定義在thread的class底下)
  - scheduling()
    - (for threads in L2, L3)
      如果有符合進入L1的資格(priority > 99)，則做dequeue並insert入L1的list
    - (for threads in L3)
      如果有符合進入L2的資格(priority > 49 && priority < 100)，則做dequeue並insert入L2的list
    - every time remove thread from a list:
      加入[B]的debug message
    - every time insert thread to a list:
      加入[A]的debug message
  - re_sort_list()
    因為同一個list裡的thread的approximated remaining burst time以及priority可能會因為aging或者經過一些change state的過程之後會改變，而我們在改變之後沒有把他remove掉並重新insert進去，所以可能導致list裡面的排序並非我們所想要的，因此在每次FindNextToRun()被call到時，在裡面會先call re_sort_list()才做找next thread並return他的動作。
  - ReadyToRun()
    根據傳進來的thread的Priority去把它insert相對應的List裡面，並加入[A]的debug message，然後再set last_ready_time(開始在ready state等待的時間)以便之後做aging。

- FindNextToRun()
  如同在re_sort_list()所提到的，會先call到他，然後先在L1裡面找thread，如果L1是empty就換找L2，最後才輪到L3，如果都沒有thread的話就return NULL，而在找non-empty list的時候因為都排序好了，所以直接call RemoveFront()並加入[B]的debug message，還有set last_run_time。
- Run()
  在這裡我們只有在switch回old thread讓他繼續跑的時候有更新他的last_run_time，還有加入[E]的debug message

CODE/THREAD/THREAD.*

- thread.h
  加入五個變數，分別紀錄

1. burst time
2. accumulated execution time
3. exec_priority(the priority of the thread)
4. last_ready_time(進入ready state的時間)
5. last_run_time(進入run state 的時間)

```
double burst_time ;
double accumulated_time;//accumulated execution time
int exec_priority;
int last_ready_time;
int last_running_time;
```

加入set/get以上變數的函式以及aging的函式

```
void setBurstTime(double t){
  burst_time = t;
}
double getBurstTime(){
  return burst_time;
};
void setPriority(int p){
  exec_priority = p;
}
int getPriority(){
  return exec_priority;
};
int getReadyTick(){ return last_ready_time; };
void setReadyTick(int t){ last_ready_time = t;}
int getRunTick(){ return last_running_time; };
void setRunTick(int t){ last_running_time = t;}
double getAccumulatedTime(){ return accumulated_time; };
void setAccumulatedTime(int t){ accumulated_time = t;}
/* end adding MP3109062233*/
/* added MP3 109062320*/
void aging(int currentTime);
```

重啟 Visual Studio Cod

- **thread.cc** (http://thread.cc)
  - Thread()
    加入那些新加入的變數的初始化值
  - Yield()
    計算現在時間跟last_run_time相差多少，然後加進去accumulate_time來累進已經執行的時間。
  - Sleep()
    先update accumulated_time，再依據spec給的公式計算approximated_burst_time(accumulated_time/2 + burst_time/2)，並在計算完之後reset accumulated_time為0，因為這時CPU burst已經結束了。
  - aging()
    計算waiting_time(current tick - last_ready_time)，並用一個while loop在waiting_time>1500時，做以下計算：
  - priority + 10
  - waiting_time - 1500
  - last_ready_time + 1500
    最後再check priority有沒有超出合法範圍(149)，然後在超出合法範圍時設為149，並在priority有做改變的時候印出[C]的debug message。

CODE/THREADS/ALARM.CC

當status != IdleMode時，做以下的功能：

1. call update_priority()做aging(細節在#### code/thread/scheduler.*有詳盡說明)
2. call scheduling() to re-schedule the threads(細節在#### code/thread/scheduler.*有詳盡說明)
3. call Re_sort_list() 因為priority有改動，thread也可能跑到其他list裡面，所以要再re-sort一次。(細節在#### code/thread/scheduler.*有詳盡說明)
4. 依據currentThread的priority來決定是否要被preempt，以下為被preempt的條件
    - priority > 99:當L1的first element的remaining approximate burst time小於current thread的remaining approximate burst time
    - 99 > priority > 49:當L1 is non-empty
    - priority <= 49
        - L1 is non-empty
        - L2 is non-empty
        - L3 is non-empty