OS Team22 MP3 Report
成員：郭蕙綺、趙仰生
貢獻：50% 50%

1. Trace code: Explain the purposes and details of the following 6 code paths to understand how nachos manages the lifecycle of a process (or thread) as described in the Diagram of Process State in our lecture slides

1-1. New→Ready：

Kernel::ExecAll()、Kernel::Exec(char*)：主要在執行execfile（此陣列是存取每個字串第一個字元的記憶體位址），用一個for-loop去呼叫Exec()並將execfile[i]傳入，而Exec()這個函數主要在建立Thread並將其存到“t“陣列（PCB的角色），並new一個addrspace給Thread的space，再呼叫t[threadNum]->Fork()，最後每個execfile[i]執行完畢後呼叫currentThread->Finish()。

Thread::Fork()、Thread::StackAllocate()：Fork()去呼叫StackAllocate()去allocate space給Thread並初始化記憶體，再呼叫scheduler->ReadyToRun()。

Scheduler::ReadyToRun()：將Thread的Status設成Ready，並且將Thread Append到readyList上。

1-2. Running→Ready：

Machine::Run()：讓kernel去開始usermode，為了模擬user-level的計算(離開kernel-level)， 在無限迴圈裡，執行一個user的instruction(呼叫OneInstruction())，並且送出onetick的interrupt。

Interrupt::OneTick()：主要模擬時間並檢查是否有任何待處理的interrupt被called，當Timer要求context switch時呼叫currentThread->Yield()。

Thread::Yield()、Scheduler::ReadyToRun()、Scheduler::Run()：Yield()呼叫scheduler->FindNextToRun()去找有沒有其他Thread在readyList裡，當有其他Thread在readyList時，呼叫scheduler->ReadyToRun()讓currentThread放棄CPU，並將currentThread放到readyList的最後面，在之後還可以被

re-scheduled，最後呼叫scheduler->Run()將CPU交給下一個Thread並將其State設為Running，將原本的state存起來。

1-3. Running→Waiting：

SynchConsoleOutput::PutChar()、Semaphore::P()、SynchList<T>::Append(T)：先用lock鎖住，呼叫consoleOutput->PutChar()將字串顯示在顯示器上，並且呼叫waitFor->P()，等到semaphore value > 0，就可以將其遞減，因為檢查此value與遞減必須是自動完成的，所以需要在檢查此value之前，將interrupt disable掉，當value為0時，代表semaphore還不能使用，所以呼叫Append(currentThread)將currentThread Append到queue上，然後呼叫Sleep()直到semaphore value > 0。

Thread::Sleep(bool)、Scheduler::FindNextToRun()、Scheduler::Run(Thread*, bool)：先將status設成BLOCKED，後呼叫scheduler->FindNextToRun()去找有沒有其他Thread在ready queue裡，如果沒有thread需要執行，那就必須idle CPU直到下一個I/O interrupt發生，當有其他Thread在ready queue時，呼叫scheduler->Run()將CPU交給下一個Thread並將其State設為Running，將原本的state存起來。

1-4. Waiting→Ready：

Semaphore::V()：先disable interrupt後確認queue的狀態，如果不是empty則呼叫scheduler->ReadyToRun(queue->RemoveFront())，最後re-enable interrupts。

Scheduler->ReadyToRun(queue->RemoveFront())：

呼叫queue->RemoveFront()回傳queue的第一個Thread並刪除，ReadyToRun()則將Thread放入ready queue。

1-5. Running→Terminated：

ExceptionHandler(ExceptionType) case SC_Exit：呼叫Finish()來終止目前的Thread。

Thread::Finish()：通知當前Thread的ThreadRoot已完成工作，並呼叫Sleep()。

Thread::Sleep(bool)、Scheduler::FindNextToRun()、Scheduler::Run(Thread*, bool)：先將status設成BLOCKED，後呼叫scheduler->FindNextToRun()去找有沒有其他Thread在ready queue裡，如果沒有thread需要執行，那就必須idle CPU直到下一個I/O interrupt發生，當有其他Thread在ready queue時，呼叫scheduler->Run()將CPU交給下一個Thread並將其State設為Running，將原本的state存起來。

1-6. Ready→Running：

FindNextToRun去找到下一個要執行的thread，但如果是NULL的話，代表沒有thread需要執行，那就必須idle CPU直到下一個I/O interrupt發生，如果不是NULL的話，代表有thread需要執行，就呼叫Scheduler::Run()，將CPU Dispatch 給下一個thread，並將其State設成Running，將原本thread的state存起來，接著呼叫SWITCH做context switch，這個.S檔有兩個routines(ThreadRoot和SWITCH)支持很多architectures，包括DEC MIPS、DEC Alpha、SUN SPARC、HP PA-RISC、Intel 386、IBM RS6000。

先說ThreadRoot，它是所有thread運行的入口，這個routine先將frame pointer(fp)清空，然後呼叫startup procedure、將InitialArg移到a0(r4，也就是argument registers)上、並呼叫main procedure、最後再呼叫clean up procedure。

再來是SWITCH，它負責thread之間的切換，這個routine首先處理old thread，它先save old stack pointer、然後save所有callee-save register(包括s0~s7，也就是r16~r23)、接著save frame pointer(fp)還有save return address(pc)。再來處理new thread，它先load new stack pointer、然後load所有callee-save register(包括s0~s7，也就是r16~r23)、接著load frame pointer(fp)還有load return address(pc)。最後jump到new thread的pc上，如此一來達到交換執行thread的效果。

回到Scheduler::Run()這裡，檢查如果前一個thread已經完成了，就將其清空，最後，如果有address space需要被restore，就將其restore。

2. Implementation

在這次的作業中，我們共修改了在Thread檔案夾裡的alarm.cc, alarm.h, kernel.cc, kernel.h, scheduler.cc, scheduler.h, thread.cc, thread.h 及在lib檔案夾中的debug.h，以下會一一說明。

1. kernel.h:

- class kernel:我們在private創造一個priority陣列，紀錄讀取file的priority。

```
int priority[10];
```

2. kernel.cc:

- Kernel::Kernel():我們在這裡增加一個“-ep”的判斷，用來讀取command line的檔案及其priority，將priority存到陣列中。

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

- Kernel::Exec():我們在Thread創建時，利用setPriority()將該Thread的Priority設好。

```
t[threadNum]->setPriority(priority[threadNum]);
```

3. thread.h:

- class Tread:我們在private創accumExec(為了算thread當次在running跑的tick)、priority、predict(用來記錄predict time)、execTime(用來記錄該Thread到waiting或terminate前累積的CPU burst time)、lastTime(紀錄一次完整的CPU burst time)、agingCount(該Thread在ready queue等待的時間)、comeReady(紀錄該Thread進來ready queue的時間點)，而所有public的set function是用來設定值的，所有get function是用來讀取值的。

```
private:
  // some of the privat
  double   accumExec;
  int   priority;
  double   predict;
  double   execTime;
  double   lastTime;
  int   agingCount;
  double   comeReady;
```

```
void setPriority(int input) { priority = input; }
int getPriority() { return priority; }
void setPredict(double input) { predict = input; }
double getPredict() { return predict; }
void setExecTime(double input) { execTime = input; }
double getExecTime() { return execTime; }
void setLastTime(double input) { lastTime = input; }
double getLastTime() { return lastTime; }
void setAgingCount(int input) { agingCount = input; }
int getAgingCount() { return agingCount; }
void setAccumExec(double input) { accumExec = input; }
double getAccumExec() { return accumExec; }
void setComeReady(double input) { comeReady = input; }
double getComeReady() { return comeReady; }
```

4. thread.cc:

- Thread::Thread():在constructor將剛剛在thread.h創的變數初始化。
- Thread::Yield():由於在running_state轉換到ready_state會經過Yield()，因此我們在此累加該Thread的execTime，計算方法：之前累積的execTime + 現在的時間 – 當初進到running state的時間；也在此算出該Thread的accumExec(剛剛執行的時長)，計算方法：現在的時間 – 當初進到running state的時間。

```
kernel->currentThread->setExecTime(kernel->currentThread->getExecTime()
+ kernel->stats->totalTicks - kernel->scheduler->getcomingRun());
kernel->currentThread->setAccumExec(kernel->stats->totalTicks - kernel->scheduler->getcomingRun());
```

- Thread::Sleep():由於在running_state轉換到waiting_state及running_state轉換到terminated_state時會經過sleep，因此我們一樣在此更新execTime、accumExec，計算方法均與Yield()相同；由於轉到waiting_state及terminated_state意即做完一次CPU burst time，我們在此更新LastTime(完整的CPU burst time)，而LastTime也就是累積到現在的execTime，還需更新predict，計算方法為spec附的公式，最後需將execTime設為0，重新累加CPU burst time，我們也在此呼叫aging，更新agingCount使FindNextToRun()能依循更新後的priority去找。

```
kernel->currentThread->setExecTime(kernel->currentThread->getExecTime()
 + kernel->stats->totalTicks - kernel->scheduler->getcomingRun());

if (finishing == false){
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<kernel->currentThread->getID()<<" update approximate burs
    <<", add "<<kernel->currentThread->getExecTime()<<", to "<<kernel->currentThread->getPredict() / 2 + kernel->current
}

kernel->currentThread->setAccumExec(kernel->stats->totalTicks - kernel->scheduler->getcomingRun());
                Loading...
kernel->currentThread->setLastTime(kernel->currentThread->getExecTime());

kernel->currentThread->setPredict(kernel->currentThread->getPredict() / 2 + kernel->currentThread->getLastTime() / 2);

kernel->currentThread->setExecTime(0);

kernel->scheduler->aging();
```

5. scheduler.h:

- class Scheduler:我們在private根據spec要求，建立型態為SortedList的MultiLevelList1(use preemptive SJF)、MultiLevelList2(use non-preemptive priority)，及型態為List的MultiLevelList3(use round robin)，及comingRun(用來記錄進入running state的時間點)；在public建

立：getcomingRun()用來讀取private的comingRun、aging()用來呼叫agingCheck()、agingCheck()確認在ready list的Threads是否需要aging、Preemtive()用來確認是否需preempt。

```
SortedList<Thread *> * MultiLevelList1;
SortedList<Thread *> * MultiLevelList2;
List<Thread *> * MultiLevelList3;
double comingRun;
```

```
double getcomingRun() { return comingRun; }
void aging();
void agingCheck(List<Thread *>*list);
bool Preemtive();
```

6. scheduler.cc:

- Scheduler::Scheduler():我們在constructor初始化三個list及comingRun。

```
MultiLevelList1 = new SortedList<Thread *>(SJFCompare);
MultiLevelList2 = new SortedList<Thread *>(PriorityCompare);
MultiLevelList3 = new List<Thread *>;
comingRun = 0;
```

- Scheduler::~Scheduler():在destructor加上三個list的delete。

```
delete MultiLevelList1;
delete MultiLevelList2;
delete MultiLevelList3;
```

- Scheduler::aging():這個function用來呼叫agingCheck並將三個list傳入。

```
agingCheck(MultiLevelList1);
agingCheck(MultiLevelList2);
agingCheck(MultiLevelList3);
```

- int SJFCompare():這個function用在MultiLevelList1(sortedList)的判斷(利用job time決定該thread在list的位置)。

```
int SJFCompare(Thread *a, Thread *b) {
    if(a->getPredict()/2 + a->getLastTime()/2 == b->getPredict()/2 + b->getLastTime()/2)
        return 0;
    return (a->getPredict()/2 + a->getLastTime()/2) > (b->getPredict()/2 + b->getLastTime()/2) ? 1 : -1;
}
```

- int PriorityCompare():這個function用MultiLevelList2(sortedList)的判斷(利用priority決定該thread在list的位置)。

```
int PriorityCompare(Thread *a, Thread *b) {
    if(a->getPriority() == b->getPriority())
        return 0;
    return (a->getPriority() > b->getPriority()) ? -1 : 1;
}
```

- Scheduler::agingCheck():我們利用agingCheck()做aging的確認，首先先將傳入的List建成ListIterator的型態，方便之後將list裡的Thread一一做確認，接著利用for-loop檢查List裡面的所有Thread，我們先更新AgingCount，更新方法：先前累積的AgingCount + 現在的時間 - 進來ready list的時間，並在加完後將進入ready list的時間設為現在，避免下次進來時重複計算，算完後確認thread的等待時間已經超過1500 ticks且priority不是149時將其priority + 10，且將等待時間-1500，而這裡要注意的是必須判斷加過的priority是否超過149，如果超過149需設為149(因為valid的priority為0~149)，接著處理該Thread的位置，我們的做法是直接從list裡面remove該thread，再利用更新後的priority將其插入正確的list。

```
void Scheduler::agingCheck(List<Thread *> *list){
    ListIterator<Thread*> *iter = new ListIterator<Thread*>((List<Thread*>*)list);
    for( ; iter->IsDone() != true; iter->Next()){
        Thread* now = iter->Item();
        if (now!=kernel->currentThread) {
            now->setAgingCount(now->getAgingCount() + kernel->stats->totalTicks - now->getComeReady());
            now->setComeReady(kernel->stats->totalTicks);
        }
        int oriPriority = now->getPriority();
        if(now->getAgingCount() >= 1500 && oriPriority != 149){
            now->setAgingCount(now->getAgingCount()-1500);
            now->setPriority(now->getPriority() + 10);
            if(now->getPriority() > 149) now->setPriority(149);
            DEBUG('z', "[C] Tick ["<<kernel->stats->totalTicks << "]: Thread ["<< now->getID() << "] changes its priority from [" << oriP
            list->Remove(now);
            if(now->getPriority() > 99){
                MultiLevelList1->Insert(now);
                if(list != MultiLevelList1){ // L2->L1
                    DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is removed from queue L[2]
                    DEBUG('z',"[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is inserted into queue L[1
                }
            } else if(now->getPriority() > 49){
                MultiLevelList2->Insert(now);
                if(list != MultiLevelList2){
                    DEBUG('z',"[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is inserted into queue L[2
                    DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << now->getID() << "] is removed from queue L[3
                }
            } else {
                MultiLevelList3->Append(now);
            }
        }
    }
}
```

- Scheduler::Preemtive():這個function用來檢查現在的Thread需不需要被preemptive，如果現在的Thread的priority為100~149，則需檢查在MultiList1第一個Thread的predict Job time是否較低，如果較低則需preempt(SJF)，回傳true；如果現在的Thread的priority為50~99，則需檢查MultiList1有沒有Thread，如果有則需Preempt(MultiList1 priority>=100)，回傳true，如果不是以上兩種狀況則回傳false。

```
bool Scheduler::Preemtive(){
    if (kernel->currentThread->getPriority() >= 100 && kernel->currentThread->getPriority() <= 149 ){
        if (!MultiLevelList1->IsEmpty()){
            Thread * first_thread = MultiLevelList1->Front();
            double cur_job = kernel->currentThread->getPredict();
            double first_job = first_thread->getPredict();
            if(first_job < cur_job) return true;
        }
    }
    else if (kernel->currentThread->getPriority() >= 50 && kernel->currentThread->getPriority() <= 99){
        if (!MultiLevelList1->IsEmpty()) return true;
    }
    return false;
}
```

- Scheduler::ReadyToRun():ReadyToRun表示Thread剛放回readyList，當Thread是從New_state->Ready_state時需將等待時間設為0，接著需設該thread進入ready list的時間，方便後面計算agingCount，接著將Thread根據放入List，priority=100~149放入MultiLevelList1，priority=50~99放入MultiLevelList2，priority=0~49放入MultiLevelList3。

```
if(thread->getStatus() == JUST_CREATED){
    thread->setAgingCount(0);
}
thread->setStatus(READY);
thread->setComeReady(kernel->stats->totalTicks);

if(thread->getPriority()>=100 && thread->getPriority()<=149 ){
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<thread->getID()<<" is inserted into queue L1\n");
    MultiLevelList1->Insert(thread);
}
else if(thread->getPriority()>=50 && thread->getPriority()<=99){
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<thread->getID()<<" is inserted into queue L2\n");
    MultiLevelList2->Insert(thread);
}
else if(thread->getPriority()>=0 && thread->getPriority()<=49){
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<thread->getID()<<" is inserted into queue L3\n");
    MultiLevelList3->Append(thread);
}
```

- Scheduler::FindNextToRun():FindNextToRun是用來找尋下一個將進入running_state的Thread，由於MultiLevelList1為priority最高的ready list，所以我們先檢查MultiLevelList1，如果有Thread在裡面，則呼叫removeFront(拿到在MultiLevelList1 job time最低的Thread並將其移出MultiLevelList1)，並將其Thread設為nextThread，還要將RoundRobin設為false，表示不適用此演算法；如果MultiLevelList1是空的，我們則檢查MultiLevelList2，如果有Thread，則呼叫removeFront，並將priority最高的Thread設為nextThread，也將RoundRobin設為false；最後則檢查MultiLevelList3，如果有Thread在裡面，則呼叫removeFront，並將該

Thread設為nextThread，RoundRobin設為true；如果三個List皆為空，則return NULL，表示沒有正在等待被執行的Thread。

```
if(MultiLevelList1->IsEmpty() == false){
    kernel->alarm->setRoundRobin(false);
    next_thread = MultiLevelList1->RemoveFront();
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<next_thread->getID()<<" is removed from queue L1, pri = "<<next_thread->ge
}
else if(MultiLevelList2->IsEmpty() == false){
    kernel->alarm->setRoundRobin(false);
    next_thread = MultiLevelList2->RemoveFront();
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<next_thread->getID()<<" is removed from queue L2, pri = "<<next_thread->ge
}
else if(MultiLevelList3->IsEmpty() == false){
    kernel->alarm->setRoundRobin(true);
    next_thread = MultiLevelList3->RemoveFront();
    DEBUG('z',"Tick "<<kernel->stats->totalTicks<<": Thread "<<next_thread->getID()<<" is removed from queue L3, pri = "<<next_thread->ge
}
else return NULL;
```

7. alarm.h:

- class Alarm:我們在private設一個roundRobin用來記錄是否適用於Thread的演算法是否為Round Robin，並在public設一個setRoundRobin()的function，讓scheduler能改到Alarm的private值。

```
bool roundRobin; void setRoundRobin(bool input){ roundRobin = input; }
```

8. alarm.cc:

- Alarm::CallBack():由於Spec提到aging及preemption可以延遲到下一次timer alarm interval，於是我們在此呼叫aging，確認有沒有需要被aging的Thread；需不需要preempt則是先判斷是否為IdleMode，不是的話才呼叫scheduler的Preemtive()，判斷是否需執行preempt，如果需要則呼叫interrupt的YieldOnReturn()，或者是現在的Thread適用於roundRobin的話，也需呼叫interrupt的YieldOnReturn()。

```
kernel->scheduler->aging();

if (status != IdleMode){
    if (kernel->scheduler->Preemtive() == true || roundRobin == true) interrupt->YieldOnReturn();
}
```

3. FeedBack:

這次作業在trace code時就更加了解NachOS在CPU scheduling的運作，也了解到每個state轉換需經過哪些程序，對後面的implement非常有幫助，在implement時遇到許多小問題(紀錄agingCount的時機、計算CPU burst time的位置、紀錄execution Time的位置.....)，非常感謝助教在討論區為我們解答，幫助我們釐清很多疑問，在整個implement完畢後更清楚CPU scheduling的運作及其演算法該如何使用，也注意到更多小地方，可能一個時間沒紀錄好將影響整個CPU scheduling的運作，這次作業實作第五章所學，收穫許多！