# 312505017 馮凡哲
# 312511057 林胤汶

## Part.I Multi-Queue:

1. **L1(preemptive SJF):**

首先我們在thread.h新增設置預估burst time、剩餘burst time、總執行時間的函式和變數，並在thread.cc設置初始值。

```
void setBurstTime(float t) {burstTime = t;}
float getBurstTime(){return (burstTime);}
void setRemain(float t)  {Remain = t;}
float getRemain(){return (Remain);}
void setTotalExe(float t)  {TotalExeTime = t;}
float getTotalExe(){return (TotalExeTime);}
```

```
setBurstTime(0);
setRemain(0);
setTotalExe(0);
```

根據作業規定" Reset T and update the approximated burst time when the thread becomes waiting state. "，我們在Thread::Sleep()更新burst time及總執行時間。

```
status = BLOCKED;

this->setTotalExe(kernel->stats->totalTicks - this->getLastexecutionTime() + this->getYieldTime());
float prevBurstTime = this->getBurstTime();
float newBurstTime = 0.5 * prevBurstTime + 0.5 * this->getTotalExe();

this->setYieldTime(0);

if(this->getTotalExe() != 0){
    this->setBurstTime(newBurstTime);
```

總執行時間(TotalExe)的算法是讓目前系統的tick減掉thread開始執行的時間點(LastexecutionTime)再加上YieldTime。
最後加上了確保執行時間不為零的判斷，防止thread在沒有真正執行的情況下burst time被稀釋。
LastexecutionTime在Scheduler::Run重置。

```
nextThread->setLastexecutionTime(kernel->stats->totalTicks);
```

因為作業規定" Stop accumulating T when the thread becomes ready state, and resume accumulating T when the thread moves back to the running state."，只要thread變成ready state就要停止計算執行時間，而除了sleep以外，yield也會讓thread進入Ready Queue，使其變成ready state，所以我們在Thread::Yield計算thread因為yield而中止的執行時間(YieldTime)，並且在Sleep計算總執行時間後將YieldTime歸零。
YieldTime的算法是用目前系統的tick減掉thread開始執行的時間點(LastexecutionTime)。

```
this->setYieldTime(kernel->stats->totalTicks - this->getLastexecutionTime());
```

無論讓thread停止執行的是Sleep()或Yield()，最後thread都會經由ReadyToRun()回到Ready Queue，因此我們在ReadyToRun()判斷thread的權重讓他們分配進不同的Queue。

```cpp
void Scheduler::ReadyToRun(Thread *thread)
{
    int List;
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

        if(thread->getPriority() >= 100 && thread->getPriority() <= 149){
            if( !kernel->scheduler->L1ReadyList->IsInList(thread) ){
                L1ReadyList->Insert(thread);
                List = 1;
            }
        }
        else if ( (thread->getPriority() >= 50 && thread->getPriority() <= 99) ){
            if( !L2ReadyList->IsInList(thread) ){
                L2ReadyList->Insert(thread);
                List = 2;
            }
        }
        else if ( (thread->getPriority() >= 0 && thread->getPriority() <= 49) ){
            if( !L3ReadyList->IsInList(thread) ){
                L3ReadyList->Append(thread);
                List = 3;
            }
        }
```

Thread在進入不同的Queue(L1、L2、L3)時，會經過不同的方法排序。

```cpp
Scheduler::Scheduler()
{
    L1ReadyList = new SortedList<Thread *>(BurstTimeCompare);
    L2ReadyList = new SortedList<Thread *>(PriorityCompare);
    L3ReadyList = new List<Thread *>;

    toBeDestroyed = NULL;
}

//----------------------------------------------------------------------
// Scheduler::~Scheduler
//  De-allocate the list of ready threads.
//----------------------------------------------------------------------

Scheduler::~Scheduler()
{
    //delete readyList;
    delete L1ReadyList;
    delete L2ReadyList;
    delete L3ReadyList;
}
```

SJF排序的比較方法定義在scheduler.cc，在比較之前會先計算剩餘的執行時間(Remain)，再以此進行比較。

```
int BurstTimeCompare(Thread* a, Thread* b){
    a->setRemain(a->getBurstTime() - a->getTotalExe());
    b->setRemain(b->getBurstTime() - b->getTotalExe());

    if(a->getRemain() != b->getRemain()){
        if (a->getRemain() < b->getRemain()){
            return -1;
        }
        else{
            return 1;
        }
    }else{
        return a->getID() < b->getID() ? -1 : 1;
    }
    return 0;
}
```

讓thread進入yield的時機點設置在Alarm::CallBack中。

```
if ( thread->getID() > 0 && status != IdleMode && (priority>99 && kernel->scheduler->checkRemain())){
    interrupt->YieldOnReturn();
}
```

我們另外定義了checkRemain()函式，用來檢測是否有剩餘時間(Remain) 比目前thread的更小的thread在Queue中，因為Queue已經排序過，Remain最小的會在Queue的第一項，所以只須與第一項比較即可。

```
int Scheduler::checkRemain(){
    Thread *thread = kernel->currentThread;
    ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(L1ReadyList);

    if( !L1ReadyList->IsEmpty() ){
        if(iter1->Item()->getRemain() < thread->getRemain()){
            return 1;
        }
    }
    return -1;
}
```

最後thread從Read Queue取出時，會依序從L1、L2、L3挑選thread。

```
Thread *
Scheduler::FindNextToRun ()
{
    int List;
    Thread *nextThread;
    Thread *oldThread = kernel->currentThread;
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if( !L1ReadyList->IsEmpty() ){
        nextThread = L1ReadyList->RemoveFront();
        List = 1;
    }else if ( !L2ReadyList->IsEmpty() ){
        nextThread = L2ReadyList->RemoveFront();
        List = 2;
    }else if ( !L3ReadyList->IsEmpty() ){
        nextThread = L3ReadyList->RemoveFront();
        List = 3;
    }else {
        return NULL;
    }
```

2. **L2(non-preemptive priority):**

在thread.h定義設置priority的函式。

```
void setPriority(int p){priority = p;}
int getPriority(){return (priority);}
```

因為是non-preemptive，所以不考慮Yield的情形，只需處理Sleep後的Waiting Queue的排序方法。

```
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    Q1 = new SortedList<Thread *>(BurstTimeCompare);
    Q2 = new SortedList<Thread *>(PriorityCompare);
    Q3 = new List<Thread *>;

}

//----------------------------------------------------------------
// Semaphore::Semaphore
//  De-allocate semaphore, when no longer needed.  Assume no one
//  is still waiting on the semaphore!
//----------------------------------------------------------------

Semaphore::~Semaphore()
{
    delete Q1;
    delete Q2;
    delete Q3;

}
```

ReadyToRun一樣也需要分配thread到相對應的Queue。

```
void Scheduler::ReadyToRun(Thread *thread)
{
    int List;
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

        if(thread->getPriority() >= 100 && thread->getPriority() <= 149){
            if( !kernel->scheduler->L1ReadyList->IsInList(thread) ){
                L1ReadyList->Insert(thread);
                List = 1;
            }
        }
        else if ( (thread->getPriority() >= 50 && thread->getPriority() <= 99) ){
            if( !L2ReadyList->IsInList(thread) ){
                L2ReadyList->Insert(thread);
                List = 2;
            }
        }
        else if ( (thread->getPriority() >= 0 && thread->getPriority() <= 49) ){
            if( !L3ReadyList->IsInList(thread) ){
                L3ReadyList->Append(thread);
                List = 3;
            }
        }
```

priority比較方法的定義。

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

要從Queue取出thread時，會依序從L1、L2、L3挑選。

```
Thread *
Scheduler::FindNextToRun ()
{
    int List;
    Thread *nextThread;
    Thread *oldThread = kernel->currentThread;
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if( !L1ReadyList->IsEmpty() ){
        nextThread = L1ReadyList->RemoveFront();
        List = 1;
    }else if ( !L2ReadyList->IsEmpty() ){
        nextThread = L2ReadyList->RemoveFront();
        List = 2;
    }else if ( !L3ReadyList->IsEmpty() ){
        nextThread = L3ReadyList->RemoveFront();
        List = 3;
    }else {
        return NULL;
    }
```

3. **L3(RR):**
   RR的ReadyToRun和FindNextToRun也使用一樣的方法排序及分配。
   Alarm::CallBack採用原本的設定，thread每100 ticks會Yield一次。

```
if ( thread->getID() > 0 && status != IdleMode && (priority < 50 )){
    interrupt->YieldOnReturn();
}
```

4. **Aging:**
   Aging的部分，一樣是在Alarm::CallBack每100 ticks檢查Waiting Time以判斷是否要更新priority。

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

   更新函式的部分，我們是使用定義在list.h的ListIterator，檢查L1、L2、L3中所有的thread，更新Waiting Time並檢查是否超過1500，若超過則更新priority，並且將Waiting Time歸零。

```
void Scheduler::updatePriority()
{
    int oldPriority;
    int newPriority;

    ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(L1ReadyList);
    ListIterator<Thread *> *iter2 = new ListIterator<Thread *>(L2ReadyList);
    ListIterator<Thread *> *iter3 = new ListIterator<Thread *>(L3ReadyList);
    Statistics *stats = kernel->stats;

    // L1
    for( ; !iter1->IsDone(); iter1->Next() ){
        ASSERT( iter1->Item()->getStatus() == READY);
        iter1->Item()->setWaitingTime(iter1->Item()->getWaitingTime()+TimerTicks);
        if(iter1->Item()->getWaitingTime() >= 1500 && iter1->Item()->getID() > 0 ){
            oldPriority = iter1->Item()->getPriority();
            newPriority = oldPriority + 10;
            if (newPriority > 149){
                newPriority = 149;
            }
        iter1->Item()->setPriority(newPriority);
        iter1->Item()->setWaitingTime(0);
        }
    }
```

```
// L2
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();
        newPriority = oldPriority + 10;
        if (newPriority > 149){
            newPriority = 149;
        }
        iter2->Item()->setPriority(newPriority);
        iter2->Item()->setWaitingTime(0);
    }
}
// L3
for( ; !iter3->IsDone(); iter3->Next() ){
    ASSERT( iter3->Item()->getStatus() == READY);
    iter3->Item()->setWaitingTime(iter3->Item()->getWaitingTime()+TimerTicks);
    if( iter3->Item()->getWaitingTime() >= 1500 && iter3->Item()->getID() > 0 ){
        oldPriority = iter3->Item()->getPriority();
        newPriority = oldPriority + 10;
        if (newPriority > 149){
            newPriority = 149;
        }
        iter3->Item()->setPriority(newPriority);
        iter3->Item()->setWaitingTime(0);
    }
}
```

另外在thread開始執行後，根據作業規定”When the thread turns into running state, the waiting time should be reset.”，在Scheduler::Run中thread變成running state時也要將Waiting Time歸零。

```
nextThread->setStatus(RUNNING);        // nextThread is now running
nextThread->setWaitingTime(0);
```

以及” When the thread turns back into ready state, the priority should be reset to init priority.”， 在thread變成ready state時要將priority重置。
在thread.h定義設置初始priority的函數，並且在Kernel::Exec設定priority。

```
void setInitPriority(int t) {initPriority = t;}
int getInitPriority(){return initPriority;}
```

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

    return threadNum-1;
```

最後在ReadyToRun中thread變成ready state時還原priority。

```
thread->setStatus(READY);
thread->setPriority(thread->getInitPriority());
```

※備註:作業給的waiting time限制是1500 ticks，但我們自己測試Aging Test 要將限制改成1000 ticks才會有效果，測資thread的waiting time最多只會到1100 ticks。

# Part.II Debug:

A. 寫在ReadyToRun，當有thread進入Queue中就會觸發。

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

B. 寫在FindNextToRun，當有thread從Queue取出就會觸發。

```
DEBUG(dbgSche, "[B]Tick[" << kernel->stats->totalTicks
    << "]: Thread[" << nextThread->getID() << "] is removed from queue L[" << List << "]");
```

C. 寫在我們定義的updataPriority函式裡，thread的priority被更新就會觸發。

```
DEBUG(dbgSche, "[C]Tick[" << Tick << "]: Thread [" << iter1->Item()->getID()
<< "] changes its priority from ["<<oldPriority<<"] to ["<<newPriority<<"]");
```

D. 寫在Sleep，當預估burst time更新就會觸發。

```
DEBUG(dbgSche, "[D]Tick[" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
<< "] update approximate burst time, from: ["<< prevBurstTime << "], add [" << this->getTotalExe()
<< "], to [" << newBurstTime << "]");
```

E. 寫在Run的SWITCH前，當new thread和old thread切換時就會觸發。

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