MP3 Team19

Kernel.*

我們在kernel.h新增了一個array來抓當作參數輸入的優先度，並把原本的array t改成20來處理L3 aging的測資(同時有16個以上的Thread在L3裡面等待)

```
int ThreadPriority[20];    //kernel.h line72
Thread *t[20]              //kernel.h line75
```

在kernel::kernel新增了"-ep"這個指令

```
else if (strcmp(argv[i], "-ep") == 0) {    //kernel.cc line61~66
    execfile[++execfileNum] = argv[++i];
    ThreadPriority[execfileNum] = atoi(argv[++i]);
    cout << execfile[execfileNum] << "\n"
         << "Priority: " << ThreadPriority[execfileNum] << "\n";
}
```

在Kernel::ExecAll呼叫kernel::Exec時多傳了Thread的優先度進去，在kernel::Exec創立Thread時順便初始化burstTime、waitTime、exeTime跟Priority

```
void Kernel::ExecAll()            //kernel.cc line272~295
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i], ThreadPriority[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}

int Kernel::Exec(char* name, int priority)
{
    threadNum++;
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->SetBurstTime(0);
    t[threadNum]->SetWaitTime(0);
    t[threadNum]->SetExeTime(0);
    t[threadNum]->SetPriority(priority);


    t[threadNum]->space = new AddrSpace(usedPhysicalPage);
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);

    return threadNum-1;
}
```

Thread.*

在thread.h新增了8個method和5個variable。 我們原本實作Round Robin的方法利用了在Thread::Yield exeTime會歸0的特性，後來看到討論區說Yield不能改burstTime(表示也不能改exeTime)之後，就新增了一個L3time，他會跟著exeTime一起加，但是在yield會歸0

```
void SetPriority(int p);
int GetPriority();

void SetBurstTime(int t);
int GetBurstTime();

void SetExeTime(int t);
int GetExeTime();

void SetWaitTime(int t);
int GetWaitTime();

int Priority;               // priority
int WaitTime;               // time wainting in ready queue
int BurstTime;              // next expect execution time
int ExeTime;                // processed time in CPU
int L3time;                 // for L3 Round-Robin
```

Thread.cc要改的部分只有會放棄CPU的Yield跟Sleep

Yield不能更新burstTime，所以exeTime也不能更新，所以在Thread::Yield需要印出來的資訊只有哪個Thread被挑選出來執行與currentThread被換掉

```
void Thread::Yield ()
{
    // cout << "Thread " << this->getID() << " yield\n";
    Thread *nextThread;
    Statistics *stats = kernel->stats;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

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

    // put itself to ready queue
    kernel->scheduler->ReadyToRun(this);

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

    if (nextThread != NULL)
    {
        cout << "Tick[" << stats->totalTicks
             << "]: Thread[" << nextThread->getID()
             << "] is now selected for execution\n"
             << "Tick[" << stats->totalTicks
             << "]: Thread[" << this->getID()
             << "] is replaced, and it has executed ["
             << this->GetExeTime() <<"] ticks\n";

        this->L3time = 0;

        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

Sleep因為會進入waiting queue，所以要更新burstTime和exeTime

```
void 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();
    }
    // returns when it's time for us to run

    // burstTime setup
    int prev_burstTime = this->GetBurstTime();
    Statistics *stats = kernel->stats;
    cout << "Tick[" << kernel->stats->totalTicks
         << "]: Thread[" << kernel->currentThread->getID()
         << "] has changed its burstTime to "
         << 0.5*prev_burstTime + 0.5*this->GetExeTime()
         << " Ticks\n";

    this->SetBurstTime(0.5*prev_burstTime+0.5*this->GetExeTime());


    // printing information
    cout << "Tick[" << stats->totalTicks
         << "]: Thread[" << nextThread->getID()
         << "] is now selected for execution\n"
         << "Tick[" << stats->totalTicks
         << "]: Thread[" << this->getID()
         << "] is replaced and it has executed ["
         << this->GetExeTime() << "] ticks\n";

    this->SetExeTime(0);
    this->L3time = 0;
    kernel->scheduler->Run(nextThread, finishing);
}
```

Scheduler.*

我們在scheduler.h新增了2個sorted linked-list(定義在libs/list.cc)當作L1和L2，一個不需要sorted的linked-list當作L3，還有一個boolean值判斷目前是不是正在做aging

```
SortedList<Thread *> *L1queue;
SortedList<Thread *> *L2queue;
List<Thread *> *L3queue;
bool aging;
```

依L1、L2、L3的順序搜尋下一個可以使用CPU的Process

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

    if(!L1queue->IsEmpty()){
        cout << "Tick[" << stats->totalTicks << "]: Thread["
             << L1queue->Front()->getID()
             << "] is removed from queue L[1]\n";
        return L1queue->RemoveFront();
    }
    else if(!L2queue->IsEmpty()){
        cout << "Tick[" << stats->totalTicks << "]: Thread["
             << L2queue->Front()->getID()
             << "] is removed from queue L[2]\n";
        return L2queue->RemoveFront();
    }
    else if(!L3queue->IsEmpty()){
        cout << "Tick[" << stats->totalTicks << "]: Thread["
             << L3queue->Front()->getID()
             << "] is removed from queue L[3]\n";
        return L3queue->RemoveFront();
    }
    else
        return NULL;
}
```

接下來Scheduler::ReadyToRun分成L1 queue、L2 queue、L3 queue三個部分說明，三個queue都要做的是把傳進來的process state設成ready，然後重設waitTime

L1 queue

1~9行是Sorted list的compare function，sorted list用這個function作為sorting的依據 15~17行把spec要求的資訊print出來 20~24行因為L1是Preemptive SJF，有process進入時就必須讓現在正在執行的process把CPU Yield出來。判斷ID>1是因為想要避免Process 0(NachOS本體)和Process 1(postal worker)在執行中被user創造的process中斷，判斷currentThread是不是在queue裡面是因為currentThread->Yield()也會call scheduler::ReadyToRun來把正在執行中的Process放進ready queue，如果不判斷currentThread是不是已經被加進去了，Yield和ReadyToRun就會一直互相呼叫

```
static int LOneCompare (Thread* x,Thread *y){
    if(x->GetBurstTime() > y->GetBurstTime()) return 1;
    else if(x->GetBurstTime() < y->GetBurstTime()) return -1;
    else{
        if(x->getID() < y->getID()) return -1;
        else return 1;
    }
    return 0;
}

Scheduler::ReadyToRun
{
    if(thread->GetPriority() >= 100 && thread->GetPriority() <= 150)
    {
        if (!kernel->scheduler->L1queue->IsInList(thread))
        {
            cout << "Tick[" << stats->totalTicks
                 << "]: Thread[" << thread->getID()
                 << "] is inserted into queue L[1]\n";

            L1queue->Insert(thread);
            if (kernel->currentThread->getID() > 1
                && !L1queue->IsInList(kernel->currentThread))
            {
                kernel->interrupt->yieldOnReturn = true;
            }
        }
    }
}

```

L2 queue

因為我們實作Non-preemptive，所以當有Process進入queue裡面時不需要Yield 1~9行一樣是sorted list的compare function 11~17行把process放進L2 queue內

```
static int LTwoCompare (Thread* x,Thread *y){
    if(x->GetPriority() > y->GetPriority()) return -1;
    else if(x->GetPriority() < y->GetPriority()) return 1;
    else{
        if(x->getID() < y->getID()) return -1;
        else return 1;
    }
    return 0;
}

Scheduler::ReadyToRun
{
    else if(thread->GetPriority() >= 50
            && thread->GetPriority() <= 99)
    {
        cout << "Tick[" << stats->totalTicks << "]: Thread["
             << thread->getID()
             << "] is inserted into queue L[2]\n";
        L2queue->Insert(thread);
    }
}

```

L3 queue

L3 queue我們使用alarm.cc內define的Timer來判斷是否要做yield，Timer的功能是它每過一個你設定的delay之後會來呼叫Alarm::CallBack，要把Round-Robin的Time quantum設定成100個tick的話，設delay的時候就要去看這個process做的時間離100還有多久，因為Yield會佔掉一個tick，所以在currentThread做了99個tick的時候就要去call Yield，不然exeTime會算成101個ticks

```
void Timer::SetInterrupt()
{
    if (!disable) {
        int delay;
        Thread* t = kernel->currentThread;
        if (t->L3time < 99 && t->GetPriority() < 50)
            delay = TimerTicks - t->L3time - 1;
        else
            delay = TimerTicks;
        if (randomize) {
            delay = 1 + (RandomNumber() % (TimerTicks * 2));
        }
       kernel->interrupt->Schedule(this, delay, TimerInt);
    }
}

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

    if (status != IdleMode && kernel->currentThread->Priority<50)
    { //remove L1,L2
        if (kernel->currentThread->L3time >= 99){
            interrupt->YieldOnReturn();
        }
    }
}

Scheduler::ReadyToRun
{
    else
    {
        cout << "Tick[" << stats->totalTicks << "]: Thread["
             << thread->getID()<<"] is inserted into queue L[3]\n";
        L3queue->Append(thread);
    }
}
```

Aging

因為OneTick負責執行user給的instruction，我們在Interrupt::OneTick計算currentThread的exeTime和在waiting queue等待的其他process。 L2和L3因為有經過aging後跑到別的queue的可能，所以我們加完優先度之後會把它從目前的queue移除並呼叫ReadyToRun來決定它應該被分到哪個queue。L1因為不會再去別的queue了，所以我們直接更改它的優先度

```
void Interrupt::OneTick
{
    Scheduler *schedule  = kernel->scheduler;
    schedule->IncreaseWaitTime();

    kernel->currentThread->SetExeTime(
        kernel->currentThread->GetExeTime()+1);

    kernel->currentThread->L3time++;
}

void
Scheduler::IncreaseWaitTime()
{

    ListIterator<Thread *> *iter1 =
        new ListIterator<Thread *>(L1queue);

    ListIterator<Thread *> *iter2 =
        new ListIterator<Thread *>(L2queue);

    ListIterator<Thread *> *iter3 =
        new ListIterator<Thread *>(L3queue);

    Statistics *stats = kernel->stats;
    int oldpriority;
    //cout<<"In IncreaseWaitTime\n";
    //L1
    for(;!iter1->IsDone();iter1->Next()){
        iter1->Item()->SetWaitTime(iter1->Item()->GetWaitTime()+1);
        if(iter1->Item()->GetWaitTime() >= PeriodToAging){
            aging = true;
            oldpriority = iter1->Item()->GetPriority();
            iter1->Item()->SetPriority(oldpriority+Aging);
            cout << "Tick[" << stats->totalTicks
                 << "]: Thread[" <<iter1->Item()->getID()
                 << "] changes its priority from [" << oldpriority
                 << "] to [" <<iter1->Item()->GetPriority()<<"]\n";
            iter1->Item()->SetWaitTime(0);
        }
    }
    //L2
    for(;!iter2->IsDone();iter2->Next()){
        iter2->Item()->SetWaitTime(iter2->Item()->GetWaitTime()+1);
        if(iter2->Item()->GetWaitTime() >= PeriodToAging){
            aging = true;
            oldpriority = iter2->Item()->GetPriority();
            iter2->Item()->SetPriority(oldpriority+Aging);
            cout << "Tick["<<stats->totalTicks
                 << "]: Thread["<<iter2->Item()->getID()
                 << "] changes its priority from [" << oldpriority
                 << "] to [" << iter2->Item()->GetPriority()<<"]\n";

            L2queue->Remove(iter2->Item());
            cout << "Tick[" << stats->totalTicks << "]: Thread["
                 << iter2->Item()->getID()
                 << "] is removed from queue L[2]\n";
            ReadyToRun(iter2->Item());
        }
    }
    //L3
    for(;!iter3->IsDone();iter3->Next()){
        iter3->Item()->SetWaitTime(iter3->Item()->GetWaitTime()+1);
        if(iter3->Item()->GetWaitTime() >= PeriodToAging
           && iter3->Item()->getID() > 1){
            aging = true;
            oldpriority = iter3->Item()->GetPriority();
            iter3->Item()->SetPriority(oldpriority+Aging);
            cout << "Tick[" << stats->totalTicks << "]: Thread["
                 << iter3->Item()->getID()
                 << "] changes its priority from [" << oldpriority
                 << "] to [" << iter3->Item()->GetPriority()<<"]\n";
            L3queue->Remove(iter3->Item());
            cout << "Tick[" << stats->totalTicks << "]: Thread["
                 << iter3->Item()->getID()
                 << "] is removed from queue L[3]\n";
            ReadyToRun(iter3->Item());
        }
    }
}
```

貢獻度

鍾昀誼：Kernel.* Thread.* Report 陳麒懋：Scheduler.* Interrupt.* Alarm.* Timer.*