1-1. New→Ready

```
Kernel::ExecAll()
        ↓
Kernel::Exec(char*)
        ↓
Thread::Fork(VoidFunctionPtr, void*)
        ↓
Thread::StackAllocate(VoidFunctionPtr, void*)
        ↓
Scheduler::ReadyToRun(Thread*)
```

Kernel::ExecAll()

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

目的:執行每一支程式。

解釋:OS把要執行的數個程式名稱丟到Exec()裡面，有幾個檔案就做幾次，執行完之後結束。

Kernel::Exec(char* name)

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

    return threadNum-1;
```

目的:執行一支程式。

解釋:為這支新程式創建一個Thread，給Thread記憶體空間，接著Thread fork。

Thread::Fork(VoidFunctionPtr func, void* arg)

```
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
                                    // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

目的:把這個thread變成是可讓CPU執行的thread。

解釋:首先把kernel的interrupt和scheduler抓到當前的thread, 再來開創stack的空間給這個thread, 都準備好之後, 把這個thread放到ready queue裡面。而在scheduler執行ReadyToRun過程中, 要把interrupt關閉, 結束之後再回到之前的interrupt狀態。

Thread::StackAllocate(VoidFunctionPtr func, void* arg)

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

```
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

目的:register state and stack initialization for this thread

解釋:創建記憶體空間給這個thread, 並設定此thread的初始狀態以便此thread進入running state時, 能夠讓CPU正常執行。

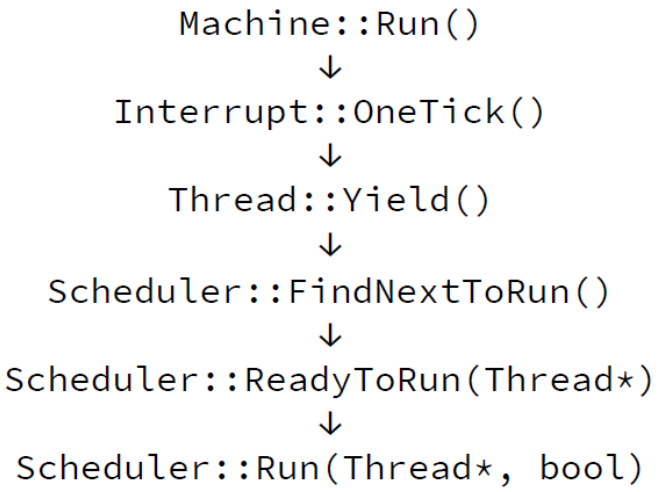
Scheduler::ReadyToRun(Thread* thread)

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

目的:讓thread加到ready queue裡面, 才能排程進入running state。

解釋:設定thread的狀態為ready, 並把thread加到ready queue裡面。

1-2. Running→Ready



Machine::Run()
目的: 抓取一個instruction，並不斷執行
解釋: one instruction在無限迴圈執行抓取instruction

```
OneInstruction(instr);
```

Interrupt::OneTick()
目的: 事先模擬時間，計算user mode和system mode的tick。
解釋:First part of OneTick is to calculate ticks of system mode and user mode.

```
if (status == SystemMode) {
    stats->totalTicks += SystemTick;
stats->systemTicks += SystemTick;
} else {
stats->totalTicks += UserTick;
stats->userTicks += UserTick;
}
```

The second part of OneTick() is to check any interrupts are ready to happen. We turn off interrupts firstly since interrupt handlers run with interrupts disabled. Then, we check for any interrupts are scheduled to happen. Finally we re-enable interrupts. YieldOnReturn variable is the symbol of context switch. If so, do context switch and turn it off. Then we do context switch and change the control of current thread to another thread. And the mode turns back to the original mode.

Thread::Yield()

目的:釋放CPU執行權，給下一隻thread

```
nextThread = kernel->scheduler->FindNextToRun();
if (nextThread != NULL) {
kernel->scheduler->ReadyToRun(this);
kernel->scheduler->Run(nextThread, FALSE);
}
(void) kernel->interrupt->SetLevel(oldLevel);
```

解釋: Find the next thread to execute, and put “this” thread to the ready queue, and finally run the next thread and set the interrupt level to the old level.

Scheduler::FindNextToRun()

目的: 找到下一個執行的thread

解釋: if readylist is not empty, return the first element in the ready list. else return NULL

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

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

Scheduler::ReadyToRun(Thread*)

目的: 增加一個thread到readylist

解釋: 把thread的status設成ready，並在readylist加一個thread

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

Scheduler::Run(Thread*, bool)

目的: Dispatch the CPU to nextThread. Save the state of the old thread,
and load the state of the new thread, by calling the machine
dependent context switch routine, SWITCH.

解釋: We are going to do context switch, if current thread are going to be finished, we should delete it. If the thread is a user program, we should save its states in CPU registers. And we check if the deleted stack is stack overflow. And the current thread is switched to the next thread. And we set it status to RUNNING. We switch the old thread and new thread in the next part. And we check if the thread we are running before this one has finished and needs to be deleted. If there is an address space, restore it.

```
Thread *oldThread = kernel->currentThread;

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

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

if (oldThread->space != NULL) { // if this thread is a user program,
    oldThread->SaveUserState();     // save the user's CPU registers
oldThread->space->SaveState();
}

oldThread->CheckOverflow();         // check if the old thread
                    // had an undetected stack overflow

kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running
```

```
SWITCH(oldThread, nextThread);

// we're back, running oldThread

// interrupts are off when we return from switch!
ASSERT(kernel->interrupt->getLevel() == IntOff);

DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

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

if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
oldThread->space->RestoreState();
```

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

```
SynchConsoleOutput::PutChar(char)
↓
Semaphore::P()
↓
SynchList<T>::Append(T)
↓
Thread::Sleep(bool)
↓
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
```

SynchConsoleOutput::PutChar(char)

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

目的:把一個字元display到console上。

解釋:由於I/O不可同時進行，所以要等拿到lock之後才可以做I/O。另外為了不要同時改到Semaphore的value，所以寫了waitFor->P()，確定沒人用P()之後，再把lock的資源釋放回系統。

Semaphore::P()

```
void
Semaphore::P()
{
    DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {        // semaphore not available
    queue->Append(currentThread);   // so go to sleep
    currentThread->Sleep(FALSE);
    }
    value--;            // semaphore available, consume its value

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

目的:分配資源給需要I/O的Thread。

解釋:首先關閉interrupt, 接著判斷現在是否有資源, 若沒有則讓thread加到waiting list, 等到再次輪到他時, Semaphore再看看有沒有資源可以給他;若有資源則將I/O資源減1, 如此一來這個Thread就可以執行I/O。

SynchList<T>::Append(T)

```
template <class T>
void
SynchList<T>::Append(T item)
{
    lock->Acquire();        // enforce mutual exclusive access to the list
    list->Append(item);
    listEmpty->Signal(lock);    // wake up a waiter, if any
    lock->Release();
}
```

目的:管控I/O資源。

解釋:首先為了不要同時跟其他function使用list, 這裡有monitor在管控, 所以一開始先拿lock, 拿到之後才能append item到list, 做完後signal任何想access list的thread, 通知那些thread現在可以access list了, 接著release lock。

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);
}
```

目的:把目前的thread suspend, 讓其他thread先使用CPU。

解釋:首先檢查要sleep的是不是current thread, 以及是否已關閉interrupt。接下來把thread state設成waiting, 並找ready queue裡面是否有在等待CPU的thread, 如果沒有就等到有, 有了之後就讓給他CPU。

Scheduler::FindNextToRun()

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

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

目的:在ready queue找一個可以跑的thread。

解釋:首先檢查是否已關閉interrupt。接下來看看ready queue有沒有thread，有的話回傳那個thread，並將那個thread從queue中移除，不然就回傳NULL。

Scheduler::Run(Thread*, bool)

目的:把目前正在執行的thread（舊的thread），換成另一個準備要使用CPU的thread（新的thread）。

解釋:

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

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

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

首先看看目前舊的thread是否已做完該做的工作，做完就紀錄給toBeDestroyed，下面的CheckToBeDestroyed會刪除這個thread。

```
if (oldThread->space != NULL) { // if this thread is a user program,
    oldThread->SaveUserState();     // save the user's CPU registers
oldThread->space->SaveState();
}

oldThread->CheckOverflow();         // check if the old thread
                    // had an undetected stack overflow
```

接下來幫舊的thread儲存CPU register狀態，並且看看stack有無overflow。

```
kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running

DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

把currentThread換成新的thread，並且將新的thread的state換成running。

```
SWITCH(oldThread, nextThread);

// we're back, running oldThread

// interrupts are off when we return from switch!
ASSERT(kernel->interrupt->getLevel() == IntOff);

DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

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

if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
oldThread->space->RestoreState();
}
```

執行context switch，結束後檢查interrupt是否是關閉狀態，接著看看舊的thread是否要被移除，最後把新的thread的CPU register state load回來，使得新的thread可以回到之前做到一半的地方繼續做下去。

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

```
Semaphore::V()
      ↓
Scheduler::ReadyToRun(Thread*)
```

Semaphore::V()

目的：increment the value and release the shared sources.

解釋：Disable interrupts firstly and let add the first item in the queue into ready queue. Finally re-enable interrupts.

```
// disable interrupts
IntStatus oldLevel = interrupt->SetLevel(IntOff);

if (!queue->IsEmpty()) {  // make thread ready.
kernel->scheduler->ReadyToRun(queue->RemoveFront());
}
value++;

// re-enable interrupts
(void) interrupt->SetLevel(oldLevel);
```

Scheduler::ReadyToRun(Thread*)
目的: 增加一個thread到readylist
解釋: 把thread的status設成ready，並在readylist加一個thread

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

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

```
ExceptionHandler(ExceptionType) case SC_Exit
                ↓
        Thread::Finish()
                ↓
      Thread::Sleep(bool)
                ↓
   Scheduler::FindNextToRun()
                ↓
Scheduler::Run(Thread*, bool)
```

ExceptionHandler(ExceptionType) case SC_Exit

目的:呼叫System Call並關閉程式。

解釋:We can observe the return status throught register4 and the current thread calling Thread::Finish().

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

Thread::Finish()

目的:關掉interrupts並進入Sleep狀態。

解釋：We turn off all the interrupts firstly since sleep happens under disabled interrupts and call Sleep and thus invokes a context switch.

```
(void) kernel->interrupt->SetLevel(IntOff);
ASSERT(this == kernel->currentThread);

DEBUG(dbgThread, "Finishing thread: " << name);
Sleep(TRUE);                // invokes SWITCH
// not reached
```

Thread::Sleep(bool)

目的：等待下一個I/O interrupt的發生

解釋：持續讓CPU在idle狀態，並尋找下一個可以執行的thread

```
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);
```

Scheduler::FindNextToRun()
目的: 找到下一個執行的thread
解釋: if readylist is not empty, return the first element in the ready list. else return NULL

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

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

Scheduler::Run(Thread*, bool)
目的: Dispatch the CPU to nextThread. Save the state of the old thread,
and load the state of the new thread, by calling the machine
dependent context switch routine, SWITCH.
解釋: We are going to do context switch, if current thread are going to be finished, we should delete it. If the thread is a user program, we should save its states in CPU registers. And we check if the deleted stack is stack overflow. And the current thread is switched to the next thread. And we set it status to RUNNING. We switch the old thread and new thread in the next part. And we check if the thread we are running before this one has finished and needs to be deleted. If there is an address space, restore it.

```
Thread *oldThread = kernel->currentThread;

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

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

if (oldThread->space != NULL) { // if this thread is a user program,
    oldThread->SaveUserState();     // save the user's CPU registers
oldThread->space->SaveState();
}

oldThread->CheckOverflow();         // check if the old thread
                    // had an undetected stack overflow

kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running
```

```
SWITCH(oldThread, nextThread);

// we're back, running oldThread

// interrupts are off when we return from switch!
ASSERT(kernel->interrupt->getLevel() == IntOff);

DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

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

if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
oldThread->space->RestoreState();
```

1-6. Ready→Running

Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
↓
SWITCH(Thread*, Thread*)
↓
(depends on the previous process state, e.g., [New,Running,Waiting]→Ready)
↓
for loop in Machine::Run()

Scheduler::FindNextToRun()
目的: 找到下一個執行的thread
解釋: if readylist is not empty, return the first element in the ready list. else return NULL

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

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

Scheduler::Run(Thread*, bool)

目的: Dispatch the CPU to nextThread. Save the state of the old thread, and load the state of the new thread, by calling the machine dependent context switch routine, SWITCH.

解釋: We are going to do context switch, if current thread are going to be finished, we should delete it. If the thread is a user program, we should save its states in CPU registers. And we check if the deleted stack is stack overflow. And the current thread is switched to the next thread. And we set it status to RUNNING. We switch the old thread and new thread in the next part. And we check if the thread we are running before this one has finished and needs to be deleted. If there is an address space, restore it.

```
Thread *oldThread = kernel->currentThread;

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

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

if (oldThread->space != NULL) { // if this thread is a user program,
    oldThread->SaveUserState();     // save the user's CPU registers
oldThread->space->SaveState();
}

oldThread->CheckOverflow();         // check if the old thread
                                    // had an undetected stack overflow

kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running
```

```
 SWITCH(oldThread, nextThread);

 // we're back, running oldThread

 // interrupts are off when we return from switch!
 ASSERT(kernel->interrupt->getLevel() == IntOff);

 DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

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

 if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
 oldThread->space->RestoreState();
```

SWITCH(Thread*, Thread*)

for loop in Machine::Run()

1. Alarm.cc:

   a. Timer interrupt時做aging

   b. 修改YieldOnReturn的條件，使timer interrupt可以做Aging和context switch signal

   We do aging at first line of alarm.cc.

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

    // perform aging algorithm per 100 ticks
    kernel->scheduler->Aging();
```

   Then, we judge the layer of thread priority and modify the condition of doing context switch, like the picture below. There are three cases to do context switch.

   1. L1 in cpu VS L1 in queue.
   2. L3 in cpu VS L3, L2, L1 in queue.
   3. L2 in cpu VS L1 in queue.

```
ASSERT(kernel->currentThread->priority>=0 && kernel->currentThread->priority <=149);
int thread_priority = kernel->currentThread->priority;
if(thread_priority>=100 && thread_priority<=149){
    level = 1;
}
else if(thread_priority>=50 && thread_priority<=99){
    level = 2;
}
else if(thread_priority>=0 && thread_priority<=49){
    level = 3;
}

bool context_switch = false;
if(level == 1 && !(kernel->scheduler->L1->IsEmpty())){ // L1 in cpu VS L1 in queue.
    context_switch = true;
}
if(level == 3){ // L3 in cpu VS L3, L2, L1 in queue.
    context_switch = true;
}
if(level == 2 && !(kernel->scheduler->L1->IsEmpty())){ // L2 in cpu VS L1 in queue.
    context_switch = true;
}
```

If the requirement is satisfied, we do a context switch.

```
if(context_switch){
    interrupt->YieldOnReturn();
}
```

And we define 3 queues to represent multi-level queue in scheduler.cc.

```
    List<Thread *> *L1;  // queue of threads that are ready to run,
    | | // but not running
    List<Thread *> *L2;
    List<Thread *> *L3;
```

2. 使thread可以讀到我們在command line寫在-ep後面的priority。

    a. thread.h

    b. thread.cc

    c. kernel.cc (Kernel::Exec())

    d. kernel.h

    We let the command line to read the argument of "-ep" and priority by modifying these files.

    1. thread.cc
    2. kernel.cc(Kernel::Exec())
    3. kernel.h
    4. thread.h

    We add an option to read "-ep" argument at thread.cc like the picture below. Then we modify Kernel::Exec() in kernel.cc to construct the basic information of a thread. We also modify kernel.h and thread.h to let a thread to have new attributes.

```
else if (strcmp(argv[i], "-ep") == 0) {
  execfile[++execfileNum]= argv[++i];
  cout << execfile[execfileNum] << "\n";
  DEBUG('z', "" << execfile[execfileNum] << "\n");
  char* temp = argv[++i];
  execfile_priority[execfileNum]= atoi(temp);
  cout << execfile_priority[execfileNum] << "\n";
```

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

    return threadNum-1;
```

3. scheduling queue

   a. scheduler.h (constructor & destructor)

We define new functions like Aging and Aging_for_one_queue to the scheduling in scheduler.h. And we modify the constructor and destructor of scheduler.

```
void Aging();
void Aging_for_one_queue(List<Thread *> *queue, int level);
```

```
Scheduler::Scheduler()
{
    L1 = new List<Thread *>;
    L2 = new List<Thread *>;
    L3 = new List<Thread *>;
    toBeDestroyed = NULL;
}
```

```
Scheduler::~Scheduler()
{
    delete L1;
    delete L2;
    delete L3;
}
```

4. put thread into ready queue

In scheduler::ReadyToRun() function, we determine the layer of the event by its priority and put new events into ready queue and update the start tick of the event.

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    if(thread->priority >= 100){ // TODO: we should ensure that the priorities of all thread <= 149 and >=0 (revision will start from
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[1]");
        L1->Append(thread);
    }
    else if(thread->priority >= 50){
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[2]");
        L2->Append(thread);
    }
    else{
        DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[3]");
        L3->Append(thread);
    }
    thread->start_aging_tick = kernel->stats->totalTicks;
}
```

5. take thread out of ready queue

   a. scheduler::FindNextToRun()

In scheduler::FindNextToRun(), we determine what to do corresponding to each condition. And we finally return next event to run.

```
if(!L1->IsEmpty()){
    // SJF
    Thread_return = L1->Front();
    it = new ListIterator<Thread *>(L1);
    for(; !it->IsDone(); it->Next()){
        itThread = it->Item();
        if(Thread_return->predicted_burst_time > itThread->predicted_burst_time){
            Thread_return = itThread;
        }
    }
    DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << Thread_return->getID() << "] is removed from queue L[1]");
    L1->Remove(Thread_return);
}
```

In the first case, we are going to implement SJF scheduling, so we do linear search to find the smallest approximate burst time event. And finally we remove that event from the ready queue.

```
else if(!L2->IsEmpty()){
    // np-p
    Thread_return = L2->Front();
    it = new ListIterator<Thread *>(L2);
    for(; !it->IsDone(); it->Next() ){
        // cout << "Thread " << itThread->getID() << " priority: " << itThread->priority << endl;
        itThread = it->Item();
        if(Thread_return->priority < itThread->priority){
            Thread_return = itThread;
        }
    }
    DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << Thread_return->getID() << "] is removed from queue L[2]");
    L2->Remove(Thread_return);
}
```

In the second case, we do similar things as the first case, not only finding the highest priority but also remove that event from the ready queue.

```
else if(!L3->IsEmpty()){
    Thread_return = L3->Front();
    DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << Thread_return->getID() << "] is removed from queue L[3]");
    L3->Remove(Thread_return);
}
```

In the third case, we do round robin scheduling, so just remove the first event from the queue from the ready queue.

```
else if(!L3->IsEmpty()){
    Thread_return = L3->Front();
    DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << Thread_return->getID() << "] is removed from queue L[3]");
    L3->Remove(Thread_return);
}
```

6. manage cpu burst time for every thread

   a. Scheduler::Run() (start_cpu_tick)

   b. Thread::Thread (constructor initialization)

   c. Thread::Sleep (update predicted CPU burst time)

   We manage burst time of every thread by aging.

   First, We assert the interrupts are all off, save the old thread's user state and update the currentthread.

```
void
Scheduler::Run(Thread *nextThread, bool finishing) // we should not revis
{
    Thread *oldThread = kernel->currentThread;

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

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

    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();     // save the user's CPU registers
        oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();         // check if the old thread
                                        // had an undetected stack overflow

    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running
```

Secondly, we make it run, restore the oldthread's state and update start_cpu_tick.

```
SWITCH(oldThread, nextThread);
//cout << "Switch finished\n";
// we're back, running oldThread

// interrupts are off when we return from switch!
ASSERT(kernel->interrupt->getLevel() == IntOff);

oldThread->start_cpu_tick = kernel->stats->totalTicks; // record new initial tick
// cout << "Now threadID = " << oldThread->getID() << endl;
// cout << "Now tick = " << oldThread->start_cpu_tick << endl;
DEBUG('z', "[F] Tick [" << oldThread->start_cpu_tick << "]: Thread [" << oldThread->getID() << "] ");

DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

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

if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
oldThread->space->RestoreState();
}
```

We add new variables like priority, predicted_burst_time, start_cpu_tick, real_burst_time, start_aging_tick and aging token these new attribute to the thread.

```
priority = 0;
predicted_burst_time = 0.0;
start_cpu_tick = 0;
real_burst_time = 0.0;
start_aging_tick = 0;
aging_token = 0;
```

In the Thread::Sleep() function, we update the predicted_burst_time and check status of the kernel, when it is not idle, run the next thread.

```
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;

    // returns when it's time for us to run
    double old_predicted_burst_time = predicted_burst_time;
    real_burst_time = kernel->stats->totalTicks - start_cpu_tick;
    predicted_burst_time = ((double)(real_burst_time/2)) + (predicted_burst_time/2);
    DEBUG('z', "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID() << "] update app

    // reset variable
    real_burst_time = 0.0;
    //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
    }
    kernel->scheduler->Run(nextThread, finishing);
```