Final_Report_<23>

1102 Operating System , Programming Project - Nachos Topic: Scheduling

Danny 黃鼎元 107030031, Davis 周志偉 109006201

I. New→Ready

userprog/userkernel.cc UserProgKernel::InitializeAllThreads()

```
 UserProgKernel::InitializeAllThreads()
{
   for (int i = 1; i <= execfileNum; i++){
      int a = InitializeOneThread(execfile[i], threadPriority[i], threadRemainingBurstTime[i]);
   }
   // After InitializeAllThreads(), let the main thread be terminated so that we can start to run
our thread.
// Use a for loop to run all (total amount = execfileNum) files, and use InitializeOneThread to
execute it.
   currentThread->Finish();
//After InitializeOneThread is successfully executed -> current thread calls Finish to end
}
```

-
    - **Use a for loop to run all (total amount = execfileNum) files, and use InitializeOneThread to execute it. After InitializeOneThread is successfully executed -> current thread calls Finish to end**

userprog/userkernel.cc UserProgKernel:: InitializeOneThread(char*, int, int)

```
 UserProgKernel::InitializeOneThread(char* name, int priority, int burst_time)
{
//The InitializeOneThread function will initialize the new thread
//When traversing each input file, give a new thread class to the file to be executed, and
create space for it

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

//The newly created thread will call fork, the purpose is to allocate the stack and load the
program into the memory
```

```
    threadNum++;
    return threadNum - 1;

//Here it will be initialized first, and then set run time (use the function in thread.h)

}
```

-
    - **The InitializeOneThread function will initialize the new thread. When traversing each input file, give a new thread class to the file to be executed, and create space for it. The newly created thread will call fork, the purpose is to allocate the stack and load the program into the memory. Here it will be initialized first, and then set each run time (use the function in thread.h)**

threads/thread.cc Thread::Fork(VoidFunctionPtr, void*)

```
 void
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);

// In the Fork function, the StackAllocate function will be called to allocate the stack and set the machine state

    StackAllocate(func, arg);

// After StackAllocate is executed, disable interrupt first, and put the set thread into the ready queue to be executed

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

-
    - **In the Fork function, the StackAllocate function will be called to allocate the stack and set the machine state. After StackAllocate is executed, disable interrupt first, and put the set thread into the ready queue to be executed.**

threads/thread.cc Thread::StackAllocate(VoidFunctionPtr, void*)

```
 void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
   //  Use the stack variable to store the Array, pointing to its top
   stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
  //…//
   // StackTop points to the whole from the head + size-4. The bottom of the array, -4 is to
ensure boundary safety
   stackTop = stack + StackSize -4;   // -4 to be on the safe side!
   *(--stackTop) = (int) ThreadRoot;
//the top of the stack points to the root of the thread
//so that the thread can be directly retrieved from the stack when the thread is SWITCH.
   *stack = STACK_FENCEPOST;
  //…//
   // Set the machine state, corresponding to switch.h/s, which is equal to the Register used
by HOST

   machineState[PCState] =(void *)ThreadRoot;
   machineState[StartupPCState] = (void *)ThreadBegin;
   // The two registers will record what was just passed in (&ForkExecute, t[threadNum])
                 // which will be used later when Machine::Run() is executed
   machineState[InitialPCState] = (void *)func;
   machineState[InitialArgState] = (void *)arg;
   machineState[WhenDonePCState] = (void *)ThreadFinish;
#endif
}
```

-
    - **Use the stack variable to store the Array, pointing to its top. StackTop points to the whole from the head + size-4. The bottom of the array, -4 is to ensure boundary safety. the top of the stack points to the root of the thread. So that the thread can be directly retrieved from the stack when the thread is SWITCH. Set the machine state, corresponding to switch.h/s, which is equal to the Register used by HOST. The two registers will record what was just passed in (&ForkExecute, t[threadNum]). which will be used later when Machine::Run() is executed.**
- threads/scheduler.cc Scheduler::ReadyToRun(Thread*)
    - **DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName()). First set the thread status to Ready, then put it in the ready queue. Since our is preemptive SJF, when putting a new thread into ready queue, we have to check wether preemption or not. if the burst time is relatively large, use YieldOnReturn() to put the current The right to use yield, and then change to a new thread.**

II. Ready→Running

threads/scheduler.cc Scheduler::FindNextToRun()

```
 Scheduler::FindNextToRun ()
{
        ASSERT(kernel->interrupt->getLevel() == IntOff);
    Statistics* stats = kernel->stats;
    ListIterator<Thread *> *Fisrt = new ListIterator<Thread *>(ReadyQueue);
        if (ReadyQueue->IsEmpty()) {
                return NULL;
        }
        else {
        DEBUG(dbgSJF,"[R] Tick[" <<stats->totalTicks<<"]:"<<" Thread
["<<Fisrt->Item()->getID()<<"] is removed from queue readyQueue");
                return ReadyQueue->RemoveFront();
        }
}
```

-
    - **First confirm that the status of the interrupt is IntOff , so interrupt will not occur then use the listIterator to know the first item of each list . Then make sure that it's not empty and return RemoveFront(). If there are no ready threads return null.**
- threads/scheduler.cc Scheduler::Run(Thread*, bool)
    - **In Run() we execute the context switch, but must first store the current state of the current thread by putting its register into the PCB, and check if the old thread had an undetected stack overflow. After that we set the status of nextThread to Running, and call the context switch. In the end, we run CheckToBeDestory() to see if the program has finished running, and delete it if it has finished running.**
- threads/switch.s SWITCH(Thread*, Thread*)
    - **Store the value of old thread into eax first, then overwrite the value of new thread to old thread, and finally store the value in register back to new thread to make context switch.**
- machine/mipssim.cc Machine::Run()
    - **The thread returns to the running state, forming a context switch loop. Load the new instruction then start to execute the loop, by use OneInstruction to run the instruction in the thread, and entering Onetick. After each simulated instruction is completed, with totaltick reaching runUntilTime, enter the debugger to debug the user program.**

III. Running→Ready

- machine/mipssim.cc Machine::Run()
    - **The thread returns to the running state, forming a context switch loop. Load the new instruction then start to execute the loop, by use OneInstruction to run the instruction in the thread, and entering Onetick. After each simulated instruction is completed, with totaltick reaching runUntilTime, enter the debugger to debug the user program.**

machine/interrupt.cc Interrupt::OneTick()

```
 void
Interrupt::OneTick()
{
   MachineStatus oldStatus = status;
   Statistics *stats = kernel->stats;

// advance simulated time
//updating the total ticks of the program,
//check whether it is currently in user or system mode
//(total ticks for the two modes should be calculated separately)
   if (status == SystemMode) {
      stats->totalTicks += SystemTick;
         stats->systemTicks += SystemTick;
   } else {                                    // USER_PROGRAM
         stats->totalTicks += UserTick;
         stats->userTicks += UserTick;
   }
   DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");
   ChangeLevel(IntOn, IntOff);          // this will disable interrupt
   CheckIfDue(FALSE); //check if there is an interrupt that is about to expire, and execute it.
   ChangeLevel(IntOff, IntOn);

                //During the process, yieldOnReturn will be set to "true", indicating that the
timer device requires context switch
   //Before calling yield, first set status to system mode to give up control and do context
switch
          //After that, the current thread will call yield to give up the control.
                //After the program is executed, it will switch back to the user mode at the
beginning,
                //and return to machine run to continue the unfinished work.

                if (yieldOnReturn) {
         yieldOnReturn = FALSE;
         status = SystemMode;                  // yield is a kernel routine
         kernel->currentThread->Yield();
         status = oldStatus;

   }
```

}

-
    - **First updating the total ticks of the program, and check whether it is currently in user or system mode (total ticks for the two modes should be calculated separately). ChangeLevel(IntOn, IntOff) will disable interrupt. CheckIfDue(FALSE) will check if there is an interrupt that is about to expire, and execute it. During the process, yieldOnReturn will be set to "true", indicating that the timer device requires context switch.Before calling yield, first set status to system mode to give up control and do context switch.After that, the current thread will call yield to give up the control. After the program is executed, it will switch back to the user mode at the beginning, and return to machine run to continue the unfinished work.**

threads/thread.cc Thread::Yield()

```
 void
Thread::Yield ()  //new one
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Yielding thread: " << name);
    //<TODO>
    // 1. Put current_thread in running state to ready state
    // 2. Then, find next thread from ready state to push on running state
    // 3. After resetting some value of current_thread, then context switch

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

    if (nextThread != NULL) {
    DEBUG(dbgSJF,"<YS>
Tick["<<kernel->stats->totalTicks<<"]:"<<"Thread["<<nextThread->getID()<<"] is now
selected for execution, thread[" << kernel->currentThread->getID()<<"],"<<"is replaced, and
it has executed ["<<kernel->currentThread->getRunTime()<<"] ticks");
    kernel->scheduler->Run(nextThread, FALSE); // context switch
    }

    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

-
    - **Disable interrupt first. The thread that is executing now has to give up the right to use it so first set the status of the current thread to ready**

and use ReadyToRun() to add the current thread to the waiting queue where it should be. Then use FindNextToRun() to find the next thread. If there is nextThread, execute it.

threads/scheduler.cc Scheduler::FindNextToRun()

```
 Scheduler::FindNextToRun ()
{
        ASSERT(kernel->interrupt->getLevel() == IntOff);
   Statistics* stats = kernel->stats;
   ListIterator<Thread *> *Fisrt = new ListIterator<Thread *>(ReadyQueue);
        if (ReadyQueue->IsEmpty()) {
                return NULL;
        }
        else {
      DEBUG(dbgSJF,"[R] Tick[" <<stats->totalTicks<<"]:"<<" Thread
["<<Fisrt->Item()->getID()<<"] is removed from queue readyQueue");
                return ReadyQueue->RemoveFront();
        }
}
```

-
    - **First confirm that the status of the interrupt is IntOff , so interrupt will not occur then use the listIterator to know the first item of each list . Then make sure that it's not empty and return RemoveFront(). If there are no ready threads return null.**
- threads/scheduler.cc Scheduler::Run(Thread*, bool)
    - **In Run() we execute the context switch, but must first store the current state of the current thread by putting its register into the PCB, and check if the old thread had an undetected stack overflow. After that we set the status of nextThread to Running, and call the context switch. In the end, we run CheckToBeDestory() to see if the program has finished running, and delete it if it has finished running.**

IV. Running→Waiting

- userprog/exception.cc ExceptionHandler(ExceptionType) case SC_PrintInt
    - **Get the output from register4 and call SynchConsoleOutput::PutInt()**
- userprog/synchconsole.cc SynchConsoleOutput::PutInt()
    - **Use sprintf to convert from a number to a string. Before performing I/O, lock the lock to prevent other read and write from entering, and use the do while loop to output the content.**

- machine/console.cc ConsoleOutput::PutChar(char)

    - **In PutChar(), first obtain the mutex (lock) of the device, then perform I/O (PutChar()), wait until the end of I/O and issue an interrupt instruction (waitFor) before releasing the mutex. If set putBusy to true, putchar is in progress, other processes cannot use it, and then put this behavior into the schedule. ConsoleTime indicates how long it will take to execute.**

threads/synch.cc Semaphore::P()

```
 void
Semaphore::P()
{
   Interrupt *interrupt = kernel->interrupt;
   Thread *currentThread = kernel->currentThread;

   // first disable interrupts to ensure mutual exclusiveness,
                 // the processor is not preemptive by other processors during access to
shared variables
   IntStatus oldLevel = interrupt->SetLevel(IntOff);

                 //In P(), it is the wait mechanism of the semaphore.
                 //When the value is greater than zero, it means that there are resources
available;
                 // when the value is equal to zero, it means that there are currently no
resources
   while (value == 0) {
         queue->Append(currentThread);
                 //the thread will be added to the waiting queue (Append()) of the semaphore
         currentThread->Sleep(FALSE);
                 //let it enter the waiting queue sleep (Sleep()).

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

    - **First disable interrupts to ensure mutual exclusiveness, the processor is not preemptive by other processors during access to shared variables. In P(), it is the wait mechanism of the semaphore.When the value is greater than zero, it means that there are resources available. when the value is equal to zero, it means that there are currently no resources. The thread will be added to the waiting queue (Append()) of the semaphore. Let it enter the waiting queue sleep (Sleep()).**
- threads/synchlist.cc SynchList<T>::Append(T)

- **First lock the lock to ensure that the access list is mutual exclusive, and store the thread in the waiting_list. Depend on the queue the thread may be added at the front or the end. If the queue is empty, the added thread will be in the first element; if the queue is not empty, it will be queued to the end of the queue. Release the lock at the end.**

- threads/thread.cc Thread::Sleep(bool)

  - **In sleep(), first set the current thread is either finished or is blocked. Then check if there are other threads to be executed (FindNextToRun()), the next thread cannot be current, then update the ApproximateBurstTime, by using the formula. For SJF we use the previous burst time plus the burst time of the current thread divide by two, to get the approximate burst time.If there is other threads to be executed add it to the schedule (Run).**

threads/scheduler.cc Scheduler::FindNextToRun()

```
 Scheduler::FindNextToRun ()
{
        ASSERT(kernel->interrupt->getLevel() == IntOff);
   Statistics* stats = kernel->stats;
   ListIterator<Thread *> *Fisrt = new ListIterator<Thread *>(ReadyQueue);
        if (ReadyQueue->IsEmpty()) {
                return NULL;
        }
        else {
      DEBUG(dbgSJF,"[R] Tick[" <<stats->totalTicks<<"]:"<<" Thread
["<<Fisrt->Item()->getID()<<"] is removed from queue readyQueue");
                return ReadyQueue->RemoveFront();
        }
```

  - **First confirm that the status of the interrupt is IntOff , so interrupt will not occur then use the listIterator to know the first item of each list . Then make sure that it's not empty and return RemoveFront(). If there are no ready threads return null.**

- threads/scheduler.cc Scheduler::Run(Thread*, bool)

  - **In Run() we execute the context switch, but must first store the current state of the current thread by putting its register into the PCB, and check if the old thread had an undetected stack overflow. After that we set the status of nextThread to Running, and call the context switch. In the end, we run CheckToBeDestory() to see if the program has finished running, and delete it if it has finished running.**

V. Waiting→Ready

- threads/synch.cc Semaphore::V()
    - **In V(), when the thread finishes I/O and issues an interrupt signal, the OS will put it back into the ready queue (ReadyToRun()) and release the resources (value) occupied by it.**
- threads/scheduler.cc Scheduler::ReadyToRun(Thread*)
    - **DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName()). First set the thread status to Ready, then put it in the ready queue. Since our is preemptive SJF, when putting a new thread into ready queue, we have to check wether preemption or not. if the burst time is relatively large, use YieldOnReturn() to put the current The right to use yield, and then change to a new thread.**

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

- userprog/exception.cc ExceptionHandler(ExceptionType) case SC_Exit
    - **In SC_Exit, when the CPU register points (ReadRegister()) to the system call that ends the program, it will end the thread (Finish()).**
- threads/thread.cc Thread::Finish()
    - **In Finish(), the signal to end the thread (TRUE) is passed to Sleep().**
- threads/thread.cc Thread::Sleep(bool)
    - **In sleep(), first set the current thread is either finished or is blocked. Then check if there are other threads to be executed (FindNextToRun()), the next thread cannot be current, then update the ApproximateBurstTime, by using the formula. For SJF we use the previous burst time plus the burst time of the current thread divide by two, to get the approximate burst time.If there is other threads to be executed add it to the schedule (Run).**

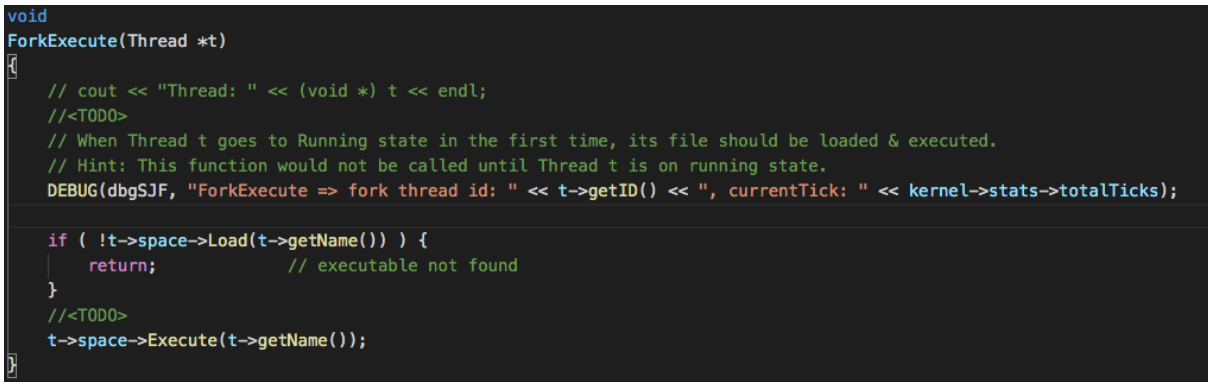
threads/scheduler.cc Scheduler::FindNextToRun()

```
 Scheduler::FindNextToRun ()
{
        ASSERT(kernel->interrupt->getLevel() == IntOff);
   Statistics* stats = kernel->stats;
   ListIterator<Thread *> *Fisrt = new ListIterator<Thread *>(ReadyQueue);
        if (ReadyQueue->IsEmpty()) {
                return NULL;
        }
        else {
      DEBUG(dbgSJF,"[R] Tick[" <<stats->totalTicks<<"]:"<<" Thread
["<<Fisrt->Item()->getID()<<"] is removed from queue readyQueue");
                return ReadyQueue->RemoveFront();
        }
}
```

- threads/scheduler.cc (continued)
    - **First confirm that the status of the interrupt is IntOff , so interrupt will not occur then use the listIterator to know the first item of each list . Then make sure that it's not empty and return RemoveFront(). If there are no ready threads return null.**
- threads/scheduler.cc Scheduler::Run(Thread*, bool)
    - **In Run() we execute the context switch, but must first store the current state of the current thread by putting its register into the PCB, and check if the old thread had an undetected stack overflow. After that we set the status of nextThread to Running, and call the context switch. In the end, we run CheckToBeDestory() to see if the program has finished running, and delete it if it has finished running.**

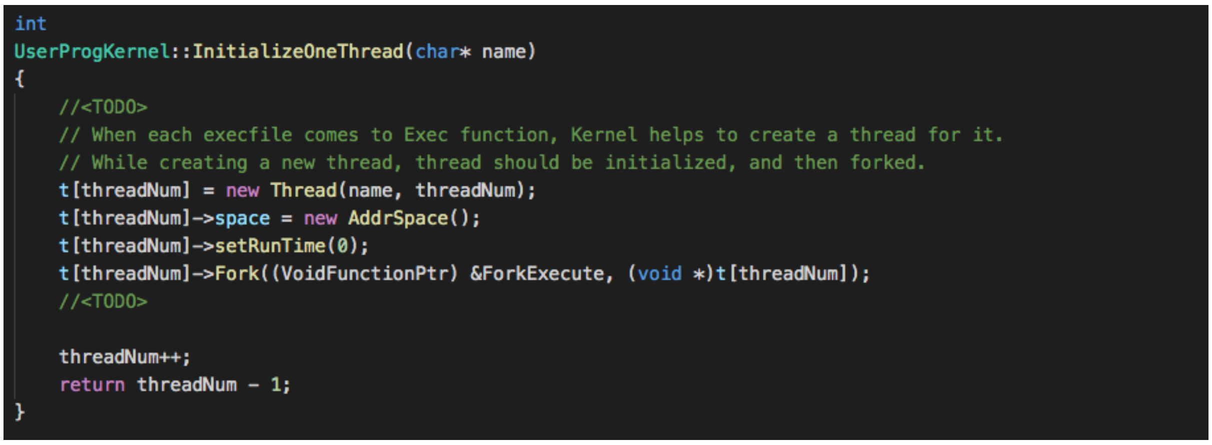
Part2: Implement

Userprog/userkerel.cc



```
void
ForkExecute(Thread *t)
{
    // cout << "Thread: " << (void *) t << endl;
    //<TODO>
    // When Thread t goes to Running state in the first time, its file should be loaded & executed.
    // Hint: This function would not be called until Thread t is on running state.
    DEBUG(dbgSJF, "ForkExecute => fork thread id: " << t->getID() << ", currentTick: " << kernel->stats->totalTicks);

    if ( !t->space->Load(t->getName()) ) {
        return;             // executable not found
    }
    //<TODO>
    t->space->Execute(t->getName());
}
```

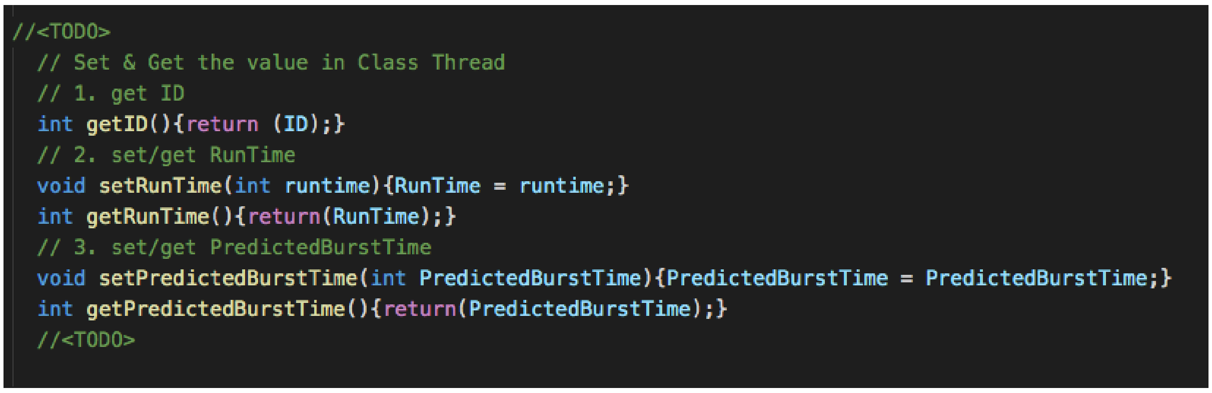
- Thread enters the running state for the first time so it is necessary to load and execute it.



```
int
UserProgKernel::InitializeOneThread(char* name)
{
    //<TODO>
    // When each execfile comes to Exec function, Kernel helps to create a thread for it.
    // While creating a new thread, thread should be initialized, and then forked.
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->setRunTime(0);
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    //<TODO>

    threadNum++;
    return threadNum - 1;
}
```

-initialize the thread and then fork().

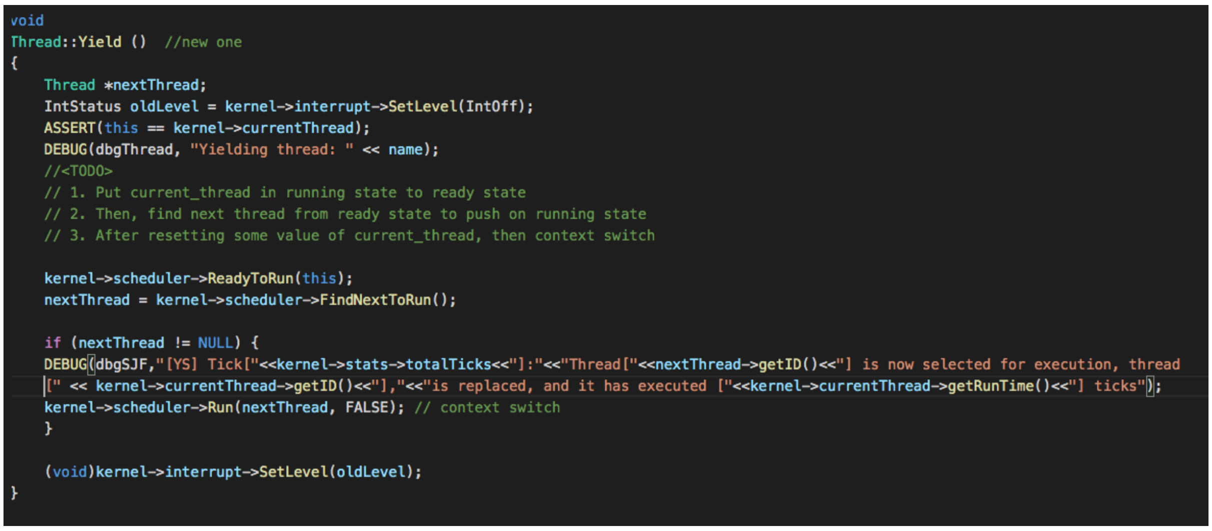
threads/thread.h



```
//<TODO>
  // Set & Get the value in Class Thread
  // 1. get ID
  int getID(){return (ID);}
  // 2. set/get RunTime
  void setRunTime(int runtime){RunTime = runtime;}
  int getRunTime(){return(RunTime);}
  // 3. set/get PredictedBurstTime
  void setPredictedBurstTime(int PredictedBurstTime){PredictedBurstTime = PredictedBurstTime;}
  int getPredictedBurstTime(){return(PredictedBurstTime);}
  //<TODO>
```

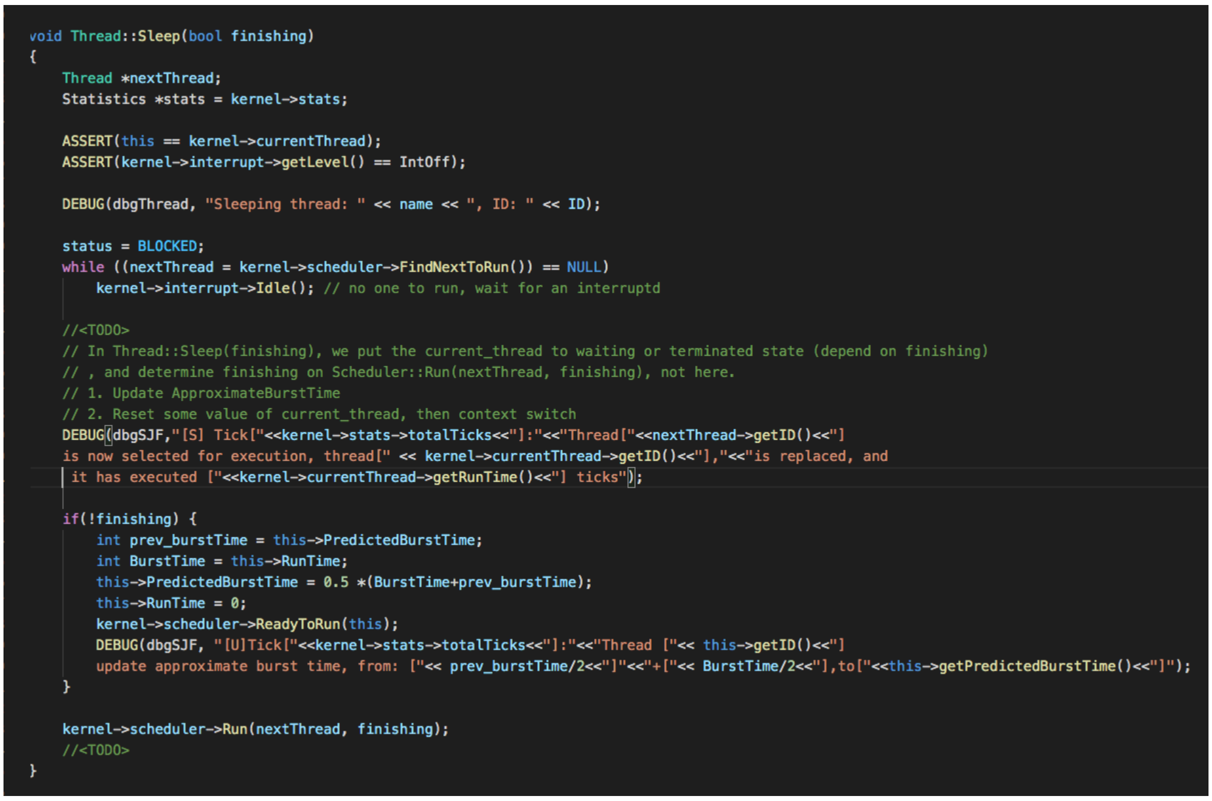
Define member function for later use.

threads/thread.cc



```
void
Thread::Yield ()  //new one
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Yielding thread: " << name);
    //<TODO>
    // 1. Put current_thread in running state to ready state
    // 2. Then, find next thread from ready state to push on running state
    // 3. After resetting some value of current_thread, then context switch

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

    if (nextThread != NULL) {
    DEBUG(dbgSJF,"[YS] Tick["<<kernel->stats->totalTicks<<"]:"<<"Thread["<<nextThread->getID()<<"] is now selected for execution, thread
    [" << kernel->currentThread->getID()<<"],"<<"is replaced, and it has executed ["<<kernel->currentThread->getRunTime()<<"] ticks");
    kernel->scheduler->Run(nextThread, FALSE); // context switch
    }

    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

- **In Yield(), the current thread is put back into the ready queue. FindNextToRun() helps locate the next thread to be run.**

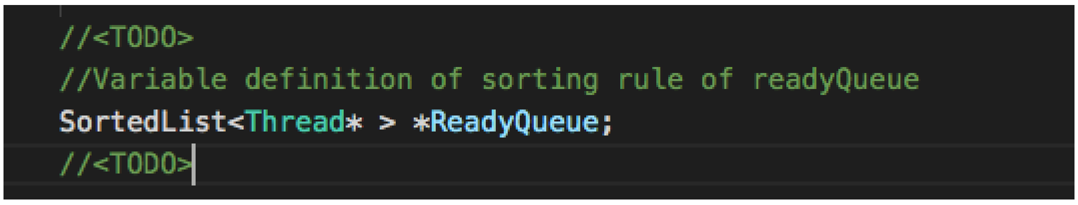


```
void Thread::Sleep(bool finishing)
{
    Thread *nextThread;
    Statistics *stats = kernel->stats;

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

    DEBUG(dbgThread, "Sleeping thread: " << name << ", ID: " << ID);

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle(); // no one to run, wait for an interruptd

    //<TODO>
    // In Thread::Sleep(finishing), we put the current_thread to waiting or terminated state (depend on finishing)
    // , and determine finishing on Scheduler::Run(nextThread, finishing), not here.
    // 1. Update ApproximateBurstTime
    // 2. Reset some value of current_thread, then context switch
    DEBUG(dbgSJF,"[S] Tick["<<kernel->stats->totalTicks<<"]:"<<"Thread["<<nextThread->getID()<<"]
    is now selected for execution, thread[" << kernel->currentThread->getID()<<"],"<<"is replaced, and
     it has executed ["<<kernel->currentThread->getRunTime()<<"] ticks");

    if(!finishing) {
        int prev_burstTime = this->PredictedBurstTime;
        int BurstTime = this->RunTime;
        this->PredictedBurstTime = 0.5 *(BurstTime+prev_burstTime);
        this->RunTime = 0;
        kernel->scheduler->ReadyToRun(this);
        DEBUG(dbgSJF, "[U]Tick["<<kernel->stats->totalTicks<<"]:"<<"Thread ["<< this->getID()<<"]
        update approximate burst time, from: ["<< prev_burstTime/2<<"]"<<"+["<< BurstTime/2<<"],to["<<this->getPredictedBurstTime()<<"]");
    }

    kernel->scheduler->Run(nextThread, finishing);
    //<TODO>
}
```

-In Sleep(), the current thread will go to either the waiting or terminated state.

We check the variable finishing to see which state it goes next. If it goes to waiting, then we print the updated burst time.

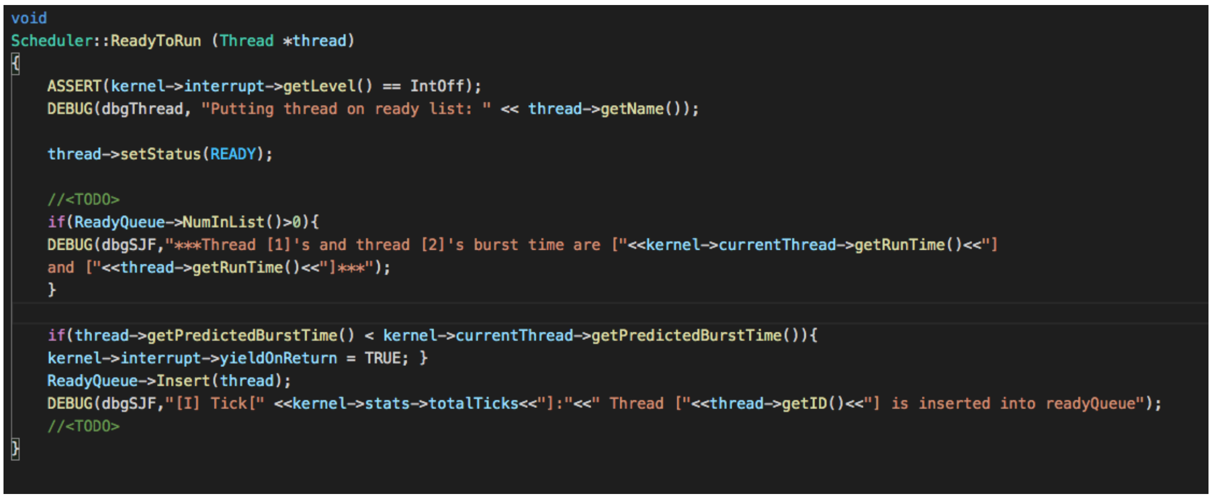
threads/scheduler.h



```
//<TODO>
//Variable definition of sorting rule of readyQueue
SortedList<Thread* > *ReadyQueue;
//<TODO>
```

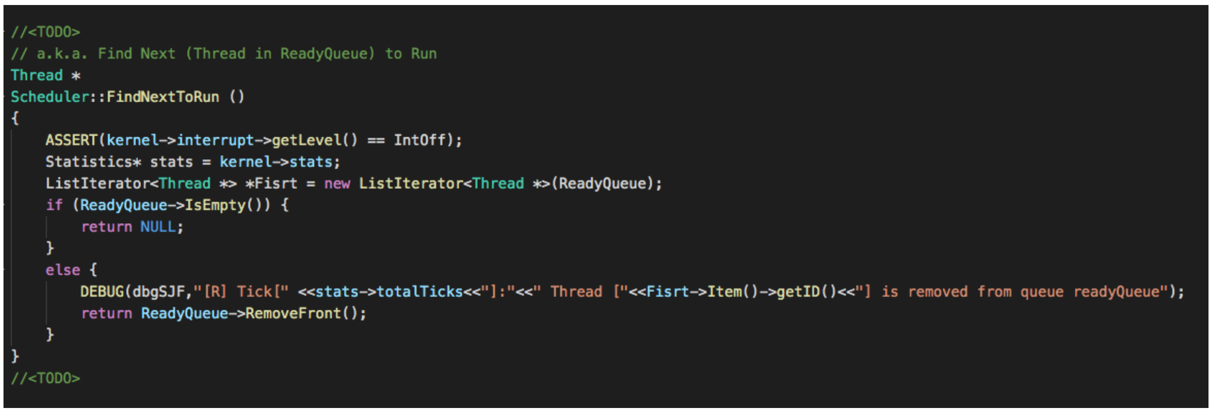
-define the readylist we use.

threads/scheduler.cc



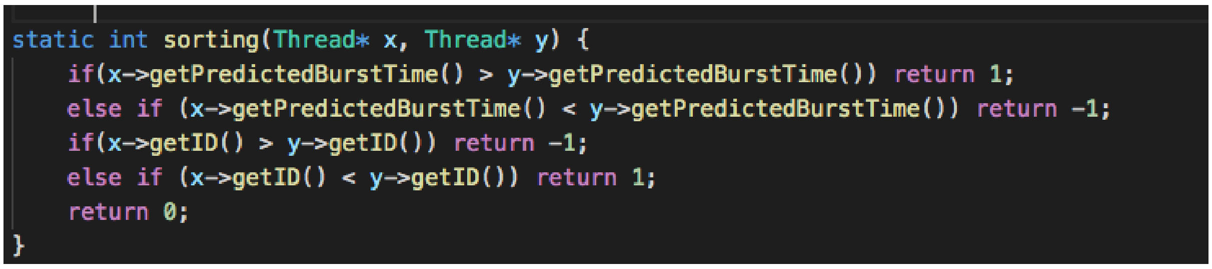
```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY);

    //<TODO>
    if(ReadyQueue->NumInList()>0){
    DEBUG(dbgSJF,"***Thread [1]'s and thread [2]'s burst time are ["<<kernel->currentThread->getRunTime()<<"]
    and ["<<thread->getRunTime()<<"]***");
    }

    if(thread->getPredictedBurstTime() < kernel->currentThread->getPredictedBurstTime()){
    kernel->interrupt->yieldOnReturn = TRUE; }
    ReadyQueue->Insert(thread);
    DEBUG(dbgSJF,"[I] Tick[" <<kernel->stats->totalTicks<<"]:"<<" Thread ["<<thread->getID()<<"] is inserted into readyQueue");
    //<TODO>
}
```

If the ready queue is not empty, we check the burst time of the threads inside the ready queue. The thread with the shorter burst time is executed next.



```cpp
//<TODO>
// a.k.a. Find Next (Thread in ReadyQueue) to Run
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    Statistics* stats = kernel->stats;
    ListIterator<Thread *> *Fisrt = new ListIterator<Thread *>(ReadyQueue);
    if (ReadyQueue->IsEmpty()) {
        return NULL;
    }
    else {
        DEBUG(dbgSJF,"[R] Tick[" <<stats->totalTicks<<"]:"<<" Thread ["<<Fisrt->Item()->getID()<<"] is removed from queue readyQueue");
        return ReadyQueue->RemoveFront();
    }
}
//<TODO>
```

If the ready queue is empty null otherwise take the list from the front thread.

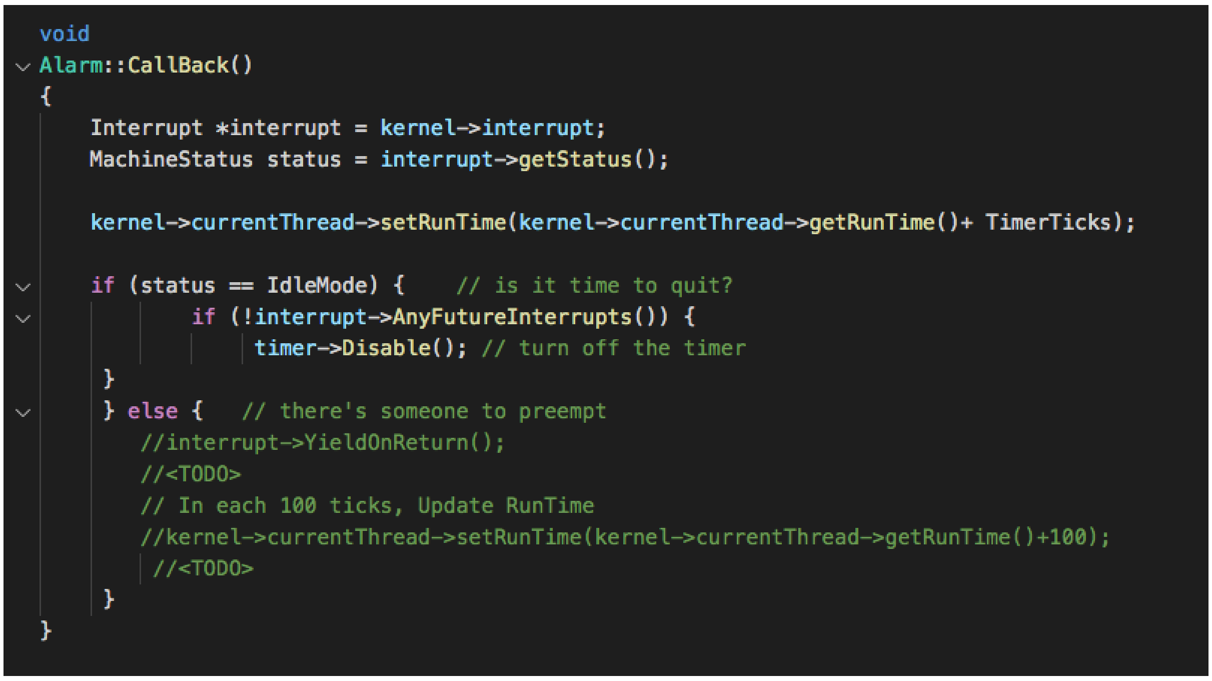


```cpp
static int sorting(Thread* x, Thread* y) {
    if(x->getPredictedBurstTime() > y->getPredictedBurstTime()) return 1;
    else if (x->getPredictedBurstTime() < y->getPredictedBurstTime()) return -1;
    if(x->getID() > y->getID()) return -1;
    else if (x->getID() < y->getID()) return 1;
    return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

compare function

- **Because return -1 means x will be in front, so we return -1**
- **when x has less prediction time or same prediction time but bigger ID and vice versa. Indicates that x is to be compared first.**

threads/Alarm.cc



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

    kernel->currentThread->setRunTime(kernel->currentThread->getRunTime()+ TimerTicks);

    if (status == IdleMode) {    // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable(); // turn off the timer
     }
     } else {   // there's someone to preempt
        //interrupt->YieldOnReturn();
        //<TODO>
        // In each 100 ticks, Update RunTime
        //kernel->currentThread->setRunTime(kernel->currentThread->getRunTime()+100);
         //<TODO>
     }
}
```

-Add 100 ticks to the running thread.

Output

```
[I] Tick[10]: Thread [1] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [0] and [0]***
[I] Tick[20]: Thread [2] is inserted into readyQueue
[R] Tick[30]: Thread [2] is removed from queue readyQueue
[S] Tick[30]:Thread[2] is now selected for execution, thread[0],is replaced, and it has
executed [0] ticks
Switching from: 0 to: 2
ForkExecute => fork thread id: 2, currentTick: 40
[AddrSpace::Load over] Tick [40]: Thread [2]
[AddrSpace::Execute over] Tick [40]: Thread [2]
2[R] Tick[69]: Thread [1] is removed from queue readyQueue
[S] Tick[69]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [0] ticks
[U]Tick[69]:Thread [2] update approximate burst time, from: [0]+[0],to[0]
Switching from: 2 to: 1
[I] Tick[79]: Thread [2] is inserted into readyQueue
ForkExecute => fork thread id: 1, currentTick: 79
[AddrSpace::Load over] Tick [79]: Thread [1]
[AddrSpace::Execute over] Tick [79]: Thread [1]
[R] Tick[98]: Thread [2] is removed from queue readyQueue
[S] Tick[98]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [0] ticks
[U]Tick[98]:Thread [1] update approximate burst time, from: [0]+[0],to[0]
Switching from: 1 to: 2
[I] Tick[108]: Thread [1] is inserted into readyQueue


2[R] Tick[643]: Thread [1] is removed from queue readyQueue
[S] Tick[643]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [600] ticks
[U]Tick[643]:Thread [2] update approximate burst time, from: [0]+[300],to[300]
Switching from: 2 to: 1
[I] Tick[644]: Thread [2] is inserted into readyQueue
[R] Tick[644]: Thread [2] is removed from queue readyQueue
[S] Tick[644]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [0] ticks
[U]Tick[644]:Thread [1] update approximate burst time, from: [0]+[0],to[0]
Switching from: 1 to: 2
[I] Tick[654]: Thread [1] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [0] and [0]***
[I] Tick[664]: Thread [2] is inserted into readyQueue
[R] Tick[664]: Thread [1] is removed from queue readyQueue
[YS] Tick[664]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [0] ticks
```

```
Switching from: 2 to: 1
1[R] Tick[674]: Thread [2] is removed from queue readyQueue
[S] Tick[674]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [0] ticks
[U]Tick[674]:Thread [1] update approximate burst time, from: [0]+[0],to[0]
Switching from: 1 to: 2
[I] Tick[684]: Thread [1] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [0] and [0]***
[I] Tick[684]: Thread [2] is inserted into readyQueue
[R] Tick[684]: Thread [1] is removed from queue readyQueue
[YS] Tick[684]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [0] ticks
Switching from: 2 to: 1


1[R] Tick[1229]: Thread [2] is removed from queue readyQueue
[S] Tick[1229]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [600] ticks
[U]Tick[1229]:Thread [1] update approximate burst time, from: [0]+[300],to[300]
Switching from: 1 to: 2
[I] Tick[1239]: Thread [1] is inserted into readyQueue


[R] Tick[1754]: Thread [1] is removed from queue readyQueue
[S] Tick[1754]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [500] ticks
[U]Tick[1754]:Thread [2] update approximate burst time, from: [150]+[250],to[400]
Switching from: 2 to: 1
[I] Tick[1764]: Thread [2] is inserted into readyQueue


1[R] Tick[2299]: Thread [2] is removed from queue readyQueue
[S] Tick[2299]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [500] ticks
[U]Tick[2299]:Thread [1] update approximate burst time, from: [150]+[250],to[400]
Switching from: 1 to: 2
[I] Tick[2300]: Thread [1] is inserted into readyQueue
[R] Tick[2300]: Thread [1] is removed from queue readyQueue
[S] Tick[2300]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [0] ticks
[U]Tick[2300]:Thread [2] update approximate burst time, from: [200]+[0],to[200]
Switching from: 2 to: 1
[I] Tick[2310]: Thread [2] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [100] and [100]***
[I] Tick[2320]: Thread [1] is inserted into readyQueue
[R] Tick[2320]: Thread [2] is removed from queue readyQueue
[YS] Tick[2320]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [100] ticks
```

```
Switching from: 1 to: 2
2[R] Tick[2330]: Thread [1] is removed from queue readyQueue
[S] Tick[2330]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [0] ticks
[U]Tick[2330]:Thread [2] update approximate burst time, from: [100]+[0],to[100]
Switching from: 2 to: 1
[I] Tick[2340]: Thread [2] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [100] and [100]***
[I] Tick[2340]: Thread [1] is inserted into readyQueue
[R] Tick[2340]: Thread [2] is removed from queue readyQueue
[YS] Tick[2340]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [100] ticks
Switching from: 1 to: 2


2[R] Tick[2885]: Thread [1] is removed from queue readyQueue
[S] Tick[2885]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [500] ticks
[U]Tick[2885]:Thread [2] update approximate burst time, from: [50]+[250],to[300]
Switching from: 2 to: 1
[I] Tick[2895]: Thread [2] is inserted into readyQueue
***Thread [1]'s and thread [2]'s burst time are [100] and [100]***
[I] Tick[2895]: Thread [1] is inserted into readyQueue
[R] Tick[2895]: Thread [2] is removed from queue readyQueue
[YS] Tick[2895]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [100] ticks
Switching from: 1 to: 2


2[R] Tick[3440]: Thread [1] is removed from queue readyQueue
[S] Tick[3440]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [600] ticks
[U]Tick[3440]:Thread [2] update approximate burst time, from: [150]+[300],to[450]
Switching from: 2 to: 1
[I] Tick[3450]: Thread [2] is inserted into readyQueue


[R] Tick[3965]: Thread [2] is removed from queue readyQueue
[S] Tick[3965]:Thread[2] is now selected for execution, thread[1],is replaced, and it has
executed [600] ticks
[U]Tick[3965]:Thread [1] update approximate burst time, from: [200]+[300],to[500]
Switching from: 1 to: 2
[I] Tick[3975]: Thread [1] is inserted into readyQueue


return value:2
[R] Tick[4502]: Thread [1] is removed from queue readyQueue
```

```
[S] Tick[4502]:Thread[1] is now selected for execution, thread[2],is replaced, and it has
executed [500] ticks
Switching from: 2 to: 1
1[I] Tick[4513]: Thread [1] is inserted into readyQueue
[R] Tick[4513]: Thread [1] is removed from queue readyQueue
[S] Tick[4513]:Thread[1] is now selected for execution, thread[1],is replaced, and it has
executed [0] ticks
[U]Tick[4513]:Thread [1] update approximate burst time, from: [250]+[0],to[250]
Switching from: 1 to: 1


1[I] Tick[5059]: Thread [1] is inserted into readyQueue
[R] Tick[5059]: Thread [1] is removed from queue readyQueue
[S] Tick[5059]:Thread[1] is now selected for execution, thread[1],is replaced, and it has
executed [600] ticks
[U]Tick[5059]:Thread [1] update approximate burst time, from: [125]+[300],to[425]
Switching from: 1 to: 1


return value:1
```