SYSTEM CALL & CPU SCHEDULING

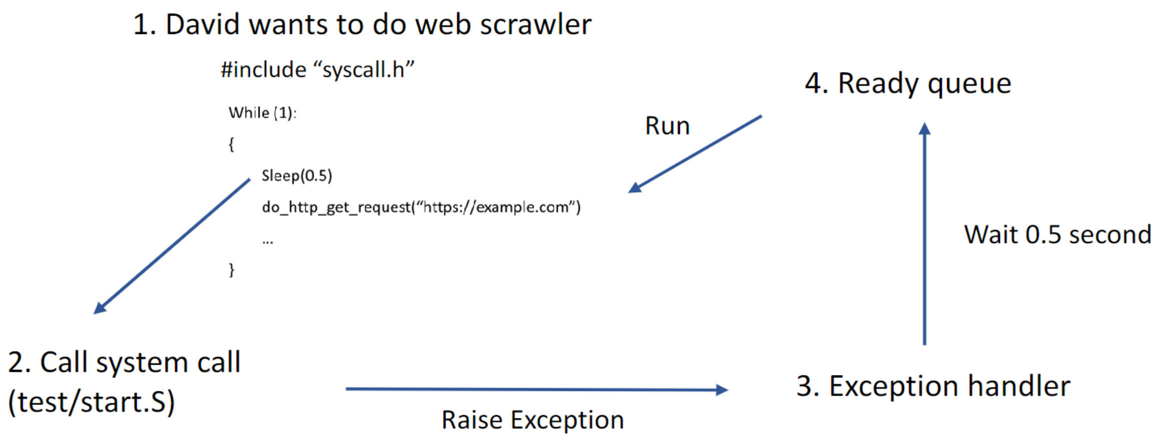
2nd Project of Operating System

Author
Name: Man-Chen Hsu
Student ID: R11942135
Due: November 23th, 2023

Contents

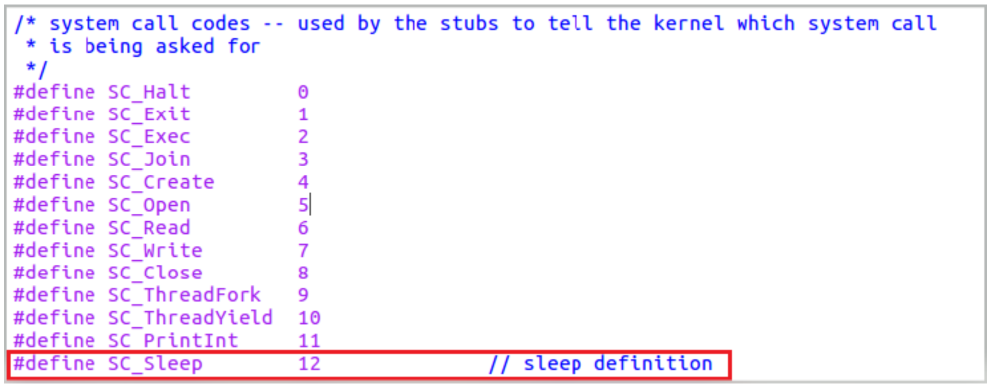
1 System Call

1.1 Plan



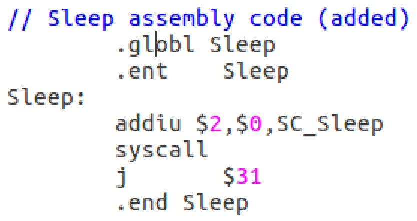
From the block diagram of the slide, it is obvious what the process is. First, we should define the system call in `../userprog/syscall.h`. Also, we have to add an assembly code associated with ”Sleep” in `../test/start.s`, so that it can help make system call to the Nachos kernel. Then, calling system call will raise Exception. Therefore, we have to add a new case to handle system call in ”ExceptionHandler” (`../userprog/exception.cc`). The instruction `kernel->alarm->WaitUntil()` should be called when a thread is going to sleep, which is implemented in `../threads/alarm.cc`. Additionally, we should first create a sorted list to store the threads in waking up order. The faster the thread wakes up, the frontaler it should be put into the sorted list. The instruction `kernel->alarm->CallBack()` should be called periodically to check whether there is any thread going to wake up. If a thread wakes up, we should put it into ready queue. With the help of iterator, we could check whether there is any thread going to wake up. Until all the processes are done, we should turn off the timer to avoid from looping forever.

1.2 Code



```
/* system call codes -- used by the stubs to tell the kernel which system call
 * is being asked for
 */
#define SC_Halt         0
#define SC_Exit         1
#define SC_Exec         2
#define SC_Join         3
#define SC_Create       4
#define SC_Open         5
#define SC_Read         6
#define SC_Write        7
#define SC_Close        8
#define SC_ThreadFork   9
#define SC_ThreadYield  10
#define SC_PrintInt     11
#define SC_Sleep        12              // sleep definition
```

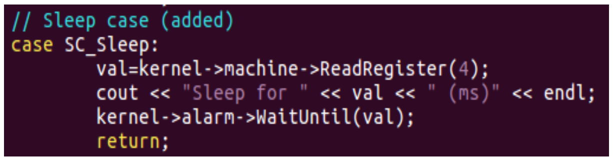
First, "Sleep" is one kind of system call. We should define the system call "Sleep" in `../userprog/syscall.h`, if we want to implement "Sleep". Also we need to give a stub number to the system call, since it is used to tell the kernel which system call is being asked for. The way to add the definition of "sleep" is same as that of other system call.



```
// Sleep assembly code (added)
        .globl Sleep
        .ent    Sleep
Sleep:
        addiu $2,$0,SC_Sleep
        syscall
        j       $31
        .end Sleep
```

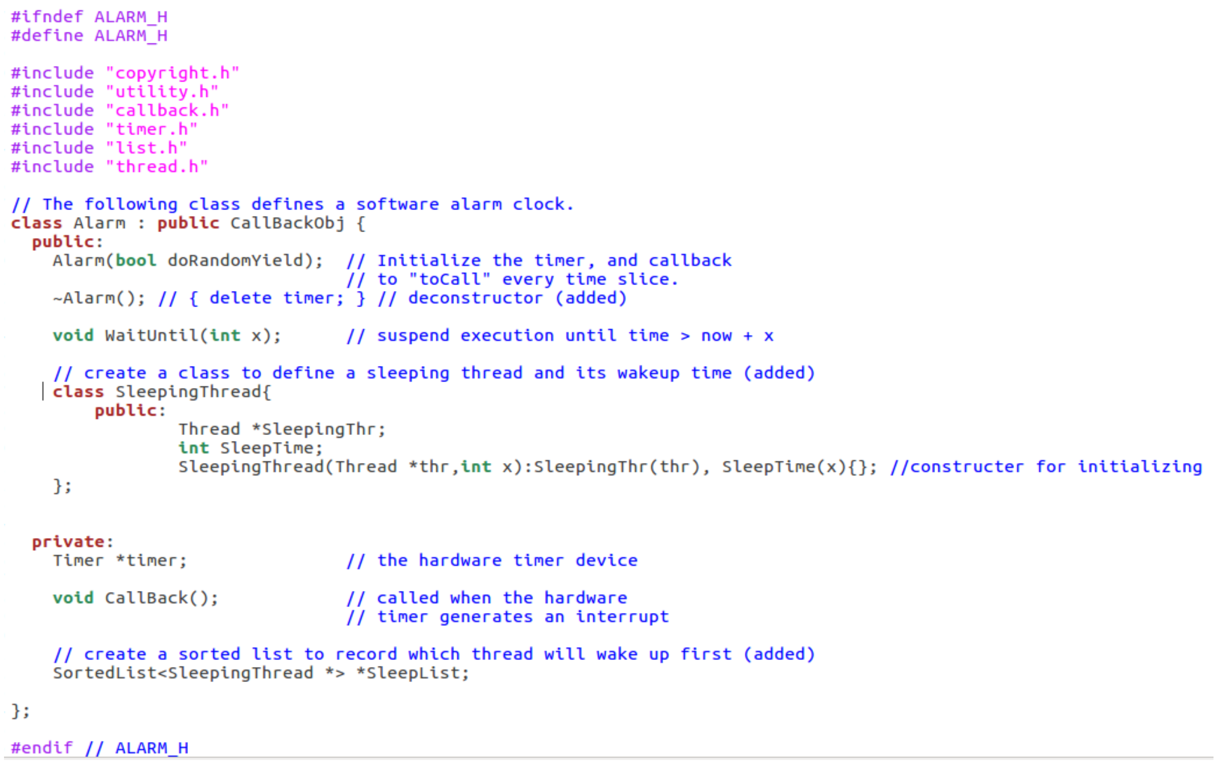
Since an assembly language stub stuffs the system call code into a register, we should add assembly code associated with "Sleep" in `../test/start.s`. Then the assembly language assist to make system calls to the Nachos kernel.

The way to add an assembly code for "Sleep" is same as other system call. The stub places the code for the system call into register r2, and leaves the arguments to the system call alone (in other words, arg1 is in r4, arg2 is in r5, arg3 is in r6, arg4 is in r7). And the return value would be in r2. In system call stub, `.globl` is an assembler directive which tells the assembler that the `main` symbol will be accesible from outside the current file (i.e. it can be referenced from other files). And, `.ent` is a debugger (pseudo) operation that marks the entry of `main`.



```
// Sleep case (added)
case SC_Sleep:
        val=kernel->machine->ReadRegister(4);
        cout << "Sleep for " << val << " (ms)" << endl;
        kernel->alarm->WaitUntil(val);
        return;
```

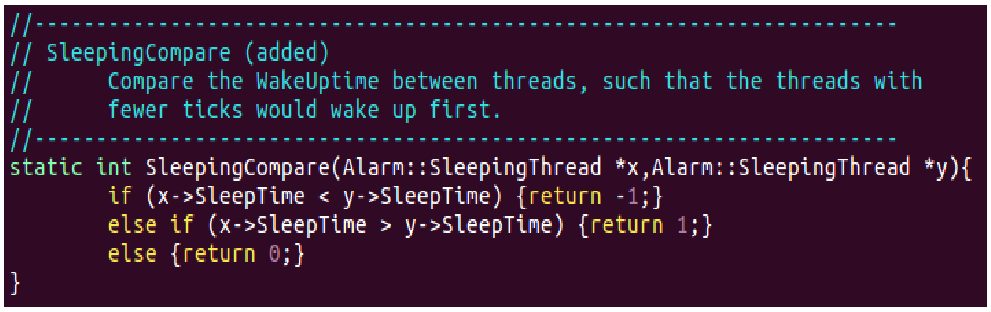
However, calling system call will raise Exception. That is, the real entry point into the Nachos kernel from user program is `../userprog/exception.cc`. As a result, we have to add a case associated with "sleep" here. According to the slides of the homework, we have to use `kernel->alarm->WaitUntil(int x)`, which will suspend execution until time ⩾ now+x. It is obvious that we need the service of kernel, when we implement "Sleep" (call system call).



```
#ifndef ALARM_H
#define ALARM_H

#include "copyright.h"
#include "utility.h"
#include "callback.h"
#include "timer.h"
#include "list.h"
#include "thread.h"

// The following class defines a software alarm clock.
class Alarm : public CallBackObj {
  public:
    Alarm(bool doRandomYield);  // Initialize the timer, and callback
                                // to "toCall" every time slice.
    ~Alarm(); // { delete timer; } // deconstructor (added)

    void WaitUntil(int x);      // suspend execution until time > now + x

    // create a class to define a sleeping thread and its wakeup time (added)
    class SleepingThread{
        public:
                Thread *SleepingThr;
                int SleepTime;
                SleepingThread(Thread *thr,int x):SleepingThr(thr), SleepTime(x){}; //constructer for initializing
    };


  private:
    Timer *timer;               // the hardware timer device

    void CallBack();            // called when the hardware
                                // timer generates an interrupt

    // create a sorted list to record which thread will wake up first (added)
    SortedList<SleepingThread *> *SleepList;

};

#endif // ALARM_H
```

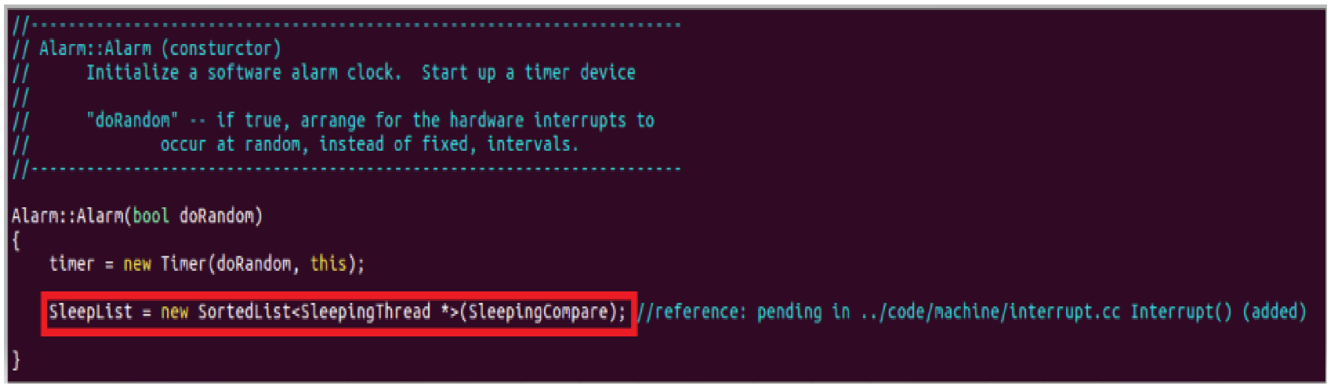
`WaitUntil(int x)` is defined in `../threads/alarm.h`, where there are data structures for a software alarm clock. But `../threads/alarm.h` is not completely implemented. In fact, we make use of a hardware timer device to generate an interrupt every X time ticks. We can find that the timer has been initialized. Also, `WaitUntil(int x)` has been set up. First, I create a class to define a sleeping thread and its wakeup time. The data members of this class include the sleeping thread called `*SleepingThr` and the total sleep time called SleepTime. Then we need to initialize the object through constructor. As mentioned in "Plan", we define a sorted list `*SleepList` to record which thread will wake up first. The way to create a sorted list could be found in `../lib/list.h` or `../machine/interrupt.h` or `../machine/interrupt.h`. Since we create a sorted list, the destructor has to be modified. Here, we just define the destructor. Later in `../threads/alarm.cc`, we will implement it specifically.

Now we have to modify `../threads/alarm.cc`. That is, we need to add some function, since it includes routines to use a hardware timer device to provide a software alarm clock.



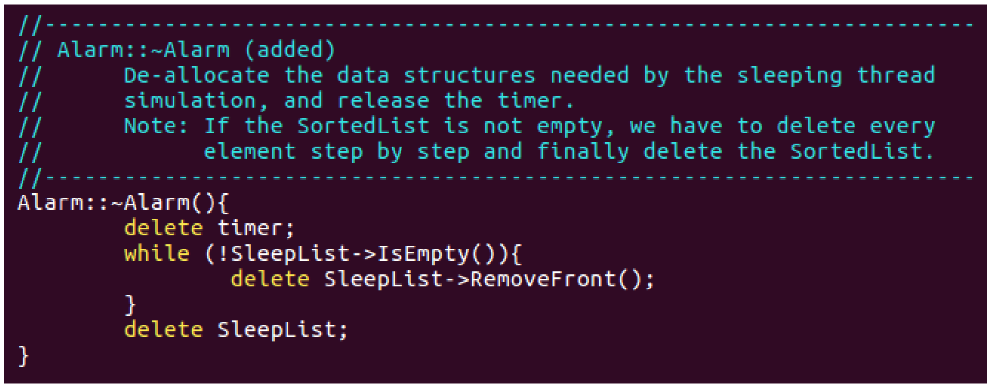
```
//----------------------------------------------------------------------
// SleepingCompare (added)
//      Compare the WakeUptime between threads, such that the threads with
//      fewer ticks would wake up first.
//----------------------------------------------------------------------
static int SleepingCompare(Alarm::SleepingThread *x,Alarm::SleepingThread *y){
        if (x->SleepTime < y->SleepTime) {return -1;}
        else if (x->SleepTime > y->SleepTime) {return 1;}
        else {return 0;}
}
```

First, I add a function `SleepingCompare`. When there are more than one thread, the duration of sleep should first compared. If a thread with less duration doesn't wake up, then other threads will certainly not wake up.



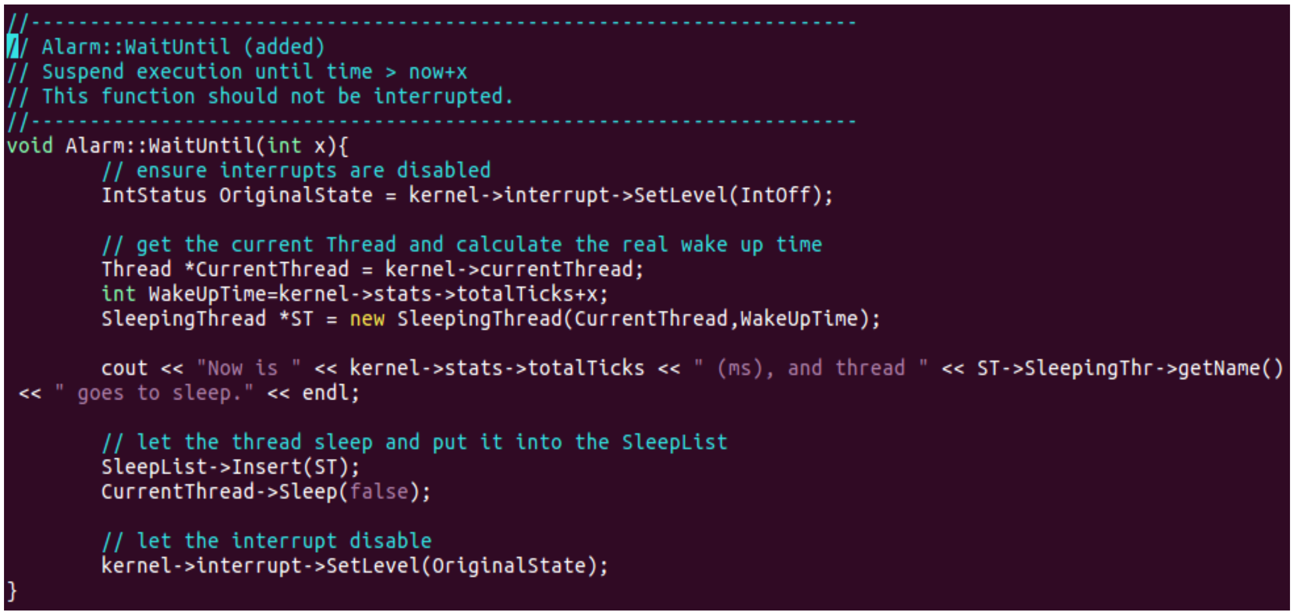
```
//----------------------------------------------------------------------
// Alarm::Alarm (consturctor)
//      Initialize a software alarm clock.  Start up a timer device
//
//      "doRandom" -- if true, arrange for the hardware interrupts to
//              occur at random, instead of fixed, intervals.
//----------------------------------------------------------------------

Alarm::Alarm(bool doRandom)
{
    timer = new Timer(doRandom, this);

    SleepList = new SortedList<SleepingThread *>(SleepingCompare);//reference: pending in ../code/machine/interrupt.cc Interrupt() (added)
}
```

For Initialization, we allocate memory dynamically for the sorted list `SleepList`. Since all types to be inserted onto a sorted list must have a "Compare", we set up `SleepingCompare` previously.



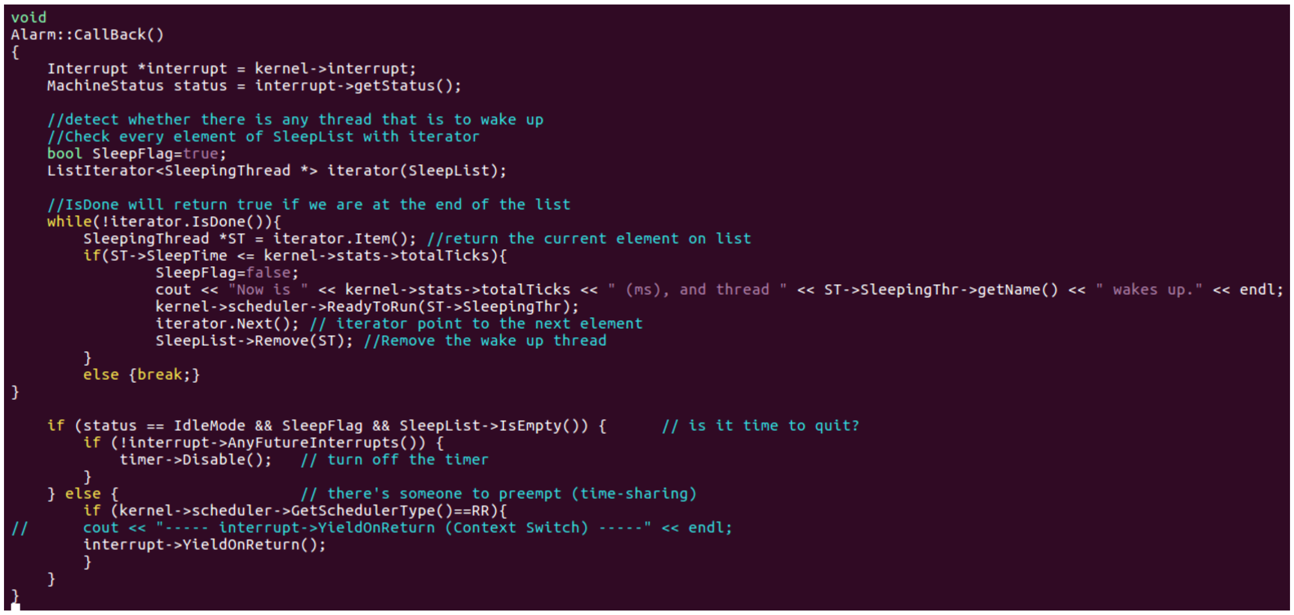
```
//----------------------------------------------------------------------
// Alarm::~Alarm (added)
//      De-allocate the data structures needed by the sleeping thread
//      simulation, and release the timer.
//      Note: If the SortedList is not empty, we have to delete every
//            element step by step and finally delete the SortedList.
//----------------------------------------------------------------------
Alarm::~Alarm(){
        delete timer;
        while (!SleepList->IsEmpty()){
                delete SleepList->RemoveFront();
        }
        delete SleepList;
}
```

If all process of sleep have been done, we should de-allocate the data structures needed by the sleeping thread simulation, and release the timer. It is worth mentioning that we have to delete every element of the sorted list step by step and then finally delete the whole sorted list, if the sorted list is not empty. Since the function `IsEmpty()` is in `../lib/list.h`, we must include it in headfile.



```
//----------------------------------------------------------------------
// Alarm::WaitUntil (added)
// Suspend execution until time > now+x
// This function should not be interrupted.
//----------------------------------------------------------------------
void Alarm::WaitUntil(int x){
        // ensure interrupts are disabled
        IntStatus OriginalState = kernel->interrupt->SetLevel(IntOff);

        // get the current Thread and calculate the real wake up time
        Thread *CurrentThread = kernel->currentThread;
        int WakeUpTime=kernel->stats->totalTicks+x;
        SleepingThread *ST = new SleepingThread(CurrentThread,WakeUpTime);

        cout << "Now is " << kernel->stats->totalTicks << " (ms), and thread " << ST->SleepingThr->getName()
 << " goes to sleep." << endl;

        // let the thread sleep and put it into the SleepList
        SleepList->Insert(ST);
        CurrentThread->Sleep(false);

        // let the interrupt disable
        kernel->interrupt->SetLevel(OriginalState);
}
```

In `interrupt.h`, there are data structures to emulate low-level interrupt hardware. The hardware provides a routine (SetLevel) to enable or disable interrupts. `IntStatus IntOff` stands for disabled and `IntStatus IntOn` stands for enabled. When member function `Alarm::WaitUntil` is implemented, it should not be interrupted. Therefore, we have to disabled the interrupt in the way mentioned above. Also, there may be multiple threads going to sleep. So we have to get the current thread and its calculate its real wake up time. Then create SleeppingThread `*ST` to record all the information. Then put it into the sorted list. Since the sleep starts, the parameter is set to be false (not finishing).

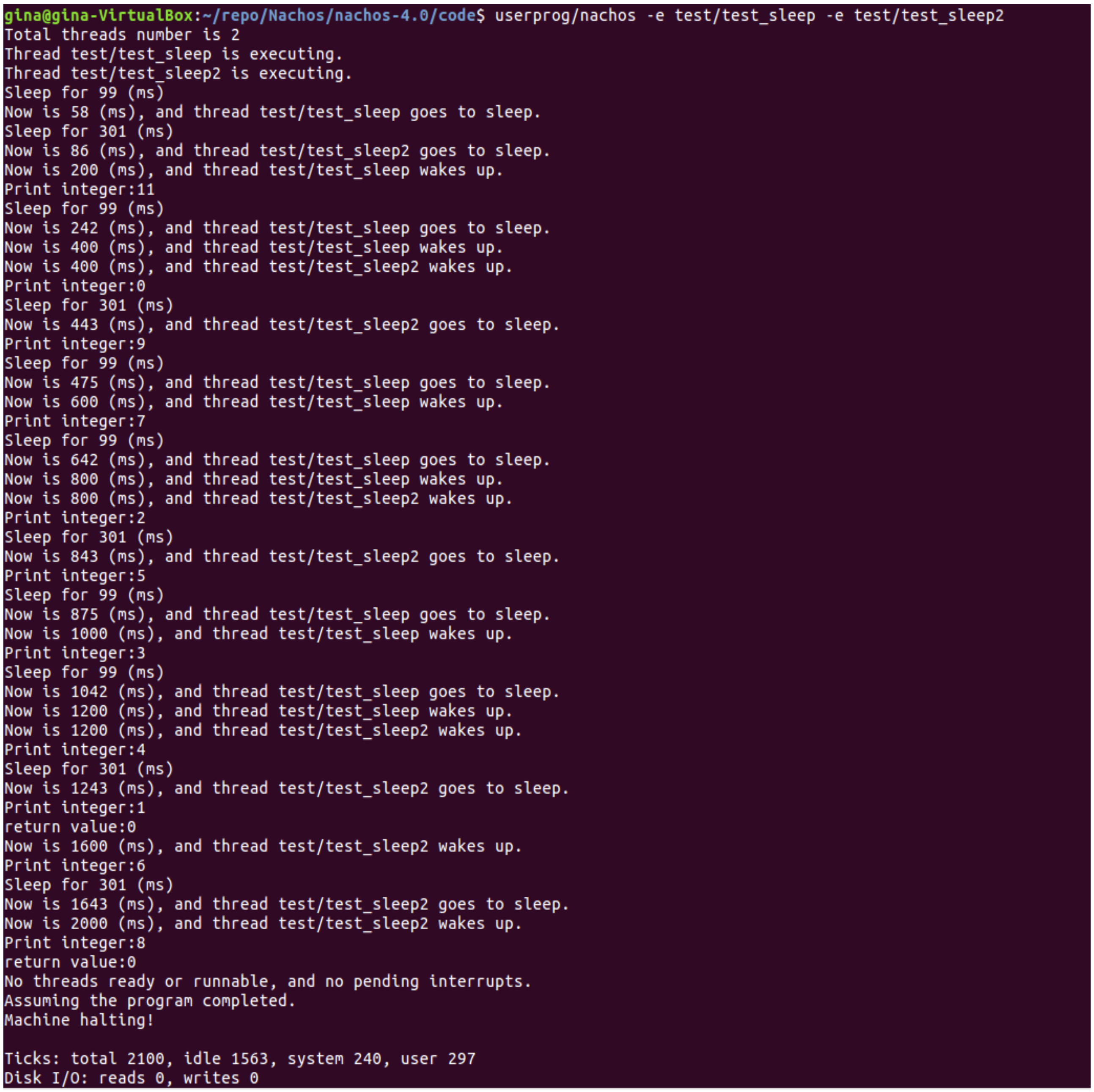


```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    //detect whether there is any thread that is to wake up
    //Check every element of SleepList with iterator
    bool SleepFlag=true;
    ListIterator<SleepingThread *> iterator(SleepList);

    //IsDone will return true if we are at the end of the list
    while(!iterator.IsDone()){
        SleepingThread *ST = iterator.Item(); //return the current element on list
        if(ST->SleepTime <= kernel->stats->totalTicks){
                SleepFlag=false;
                cout << "Now is " << kernel->stats->totalTicks << " (ms), and thread " << ST->SleepingThr->getName() << " wakes up." << endl;
                kernel->scheduler->ReadyToRun(ST->SleepingThr);
                iterator.Next(); // iterator point to the next element
                SleepList->Remove(ST); //Remove the wake up thread
        }
        else {break;}
}

    if (status == IdleMode && SleepFlag && SleepList->IsEmpty()) {      // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
        }
    } else {                    // there's someone to preempt (time-sharing)
        if (kernel->scheduler->GetSchedulerType()==RR){
//      cout << "----- interrupt->YieldOnReturn (Context Switch) -----" << endl;
        interrupt->YieldOnReturn();
        }
    }
}
```

Member function `Alarm::CallBack` is a software interrupt handler for the timer device. The timer device is set up to interrupt the CPU periodically (once every TimerTicks). This routine is called each time there is a timer interrupt, with interrupts disabled. With the help of iterator, we could check whether every element of the sorted list is going to wake up. If the `SleepTime` of a `SleepingThread *ST` is less equal to the present ticks, then the thread wakes up. And we put the thread to ready queue. Also, we will remove the thread from the sorted list. Note that instead of calling `Yield()` directly (which would suspend the interrupt handler, not the interrupted thread which is what we want to context switch), we set a flag so that once the interrupt handler is done, it will appear as if the interrupted thread calls Yield at the point it is interrupted.

To keep from looping forever, we have to check if there is nothing on the ready list, and there are no other pending interrupts. If the status is set to be idle and there is now no element to wake up and the sorted list empty and interrupt is not implementing, then we can turn off the timer. And we also ensure there's someone to preempt.

1.3 Result



```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code$ userprog/nachos -e test/test_sleep -e test/test_sleep2
Total threads number is 2
Thread test/test_sleep is executing.
Thread test/test_sleep2 is executing.
Sleep for 99 (ms)
Now is 58 (ms), and thread test/test_sleep goes to sleep.
Sleep for 301 (ms)
Now is 86 (ms), and thread test/test_sleep2 goes to sleep.
Now is 200 (ms), and thread test/test_sleep wakes up.
Print integer:11
Sleep for 99 (ms)
Now is 242 (ms), and thread test/test_sleep goes to sleep.
Now is 400 (ms), and thread test/test_sleep wakes up.
Now is 400 (ms), and thread test/test_sleep2 wakes up.
Print integer:0
Sleep for 301 (ms)
Now is 443 (ms), and thread test/test_sleep2 goes to sleep.
Print integer:9
Sleep for 99 (ms)
Now is 475 (ms), and thread test/test_sleep goes to sleep.
Now is 600 (ms), and thread test/test_sleep wakes up.
Print integer:7
Sleep for 99 (ms)
Now is 642 (ms), and thread test/test_sleep goes to sleep.
Now is 800 (ms), and thread test/test_sleep wakes up.
Now is 800 (ms), and thread test/test_sleep2 wakes up.
Print integer:2
Sleep for 301 (ms)
Now is 843 (ms), and thread test/test_sleep2 goes to sleep.
Print integer:5
Sleep for 99 (ms)
Now is 875 (ms), and thread test/test_sleep goes to sleep.
Now is 1000 (ms), and thread test/test_sleep wakes up.
Print integer:3
Sleep for 99 (ms)
Now is 1042 (ms), and thread test/test_sleep goes to sleep.
Now is 1200 (ms), and thread test/test_sleep wakes up.
Now is 1200 (ms), and thread test/test_sleep2 wakes up.
Print integer:4
Sleep for 301 (ms)
Now is 1243 (ms), and thread test/test_sleep2 goes to sleep.
Print integer:1
return value:0
Now is 1600 (ms), and thread test/test_sleep2 wakes up.
Print integer:6
Sleep for 301 (ms)
Now is 1643 (ms), and thread test/test_sleep2 goes to sleep.
Now is 2000 (ms), and thread test/test_sleep2 wakes up.
Print integer:8
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2100, idle 1563, system 240, user 297
Disk I/O: reads 0, writes 0
```

From the above picture, we could see the printing result is same as the provided result from TA. The test file test_sleep sleeps for 99 (ms), and the other sleeps for 301 (ms). When the process of sleep is done, the files would print the number in descending and ascending way, respectively. There is an interesting phenomenon. When the sleep time is up, it will not wake up immediately. For example, test_sleep goes to sleep initially at 58 (ms). And we can predict that it should wake up at 157 (ms) (58+99). In reality, it wake up at 200 (ms). The result is that the threads wake up only at every 200 (ms). It is a little out of expectation.

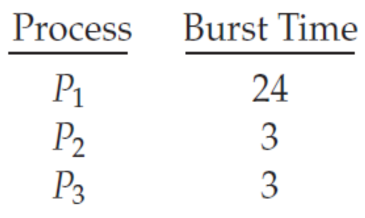
1.4 Problem

1. Why would the result is different from our prediction?
   A: In file `../machine/timer.h`, we could find the answer. A hardware timer generates a CPU interrupt every X milliseconds. This means it can be used for implementing time-slicing, or for a thread go to sleep for a specific period of time. This time, we use the timer to implement "Sleep". X is set to be 100. However, there is no thread waking up at every 100 (ms). Therefore, threads wake up only every 200 (ms).

2 CPU Scheduling

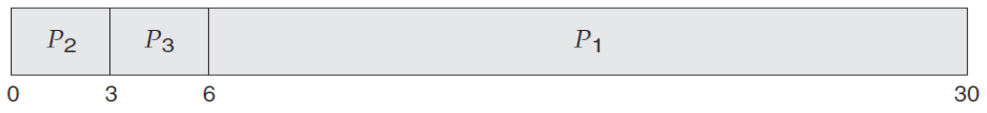
2.1 Concepts

- FCFS (First Come, First Served Scheduling)
  - With this scheme, the process that requests the CPU first is allocated the CPU first.
  - e.g.



Process	Burst Time
P_1	24
P_2	3
P_3	3

    If the process arrive in the order P2,P1,P3, we can get the result shown in the following Gantt chart,

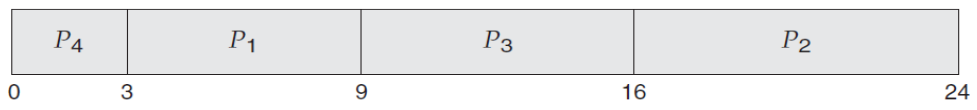


- SJF (Shortest-Job-First Scheduling)
  - The algorithm associates with each process and the length of its next CPU burst.

- That is, it is assigned to the process that has the smallest next burst CPU burst, when the CPU is available.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- e.g.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

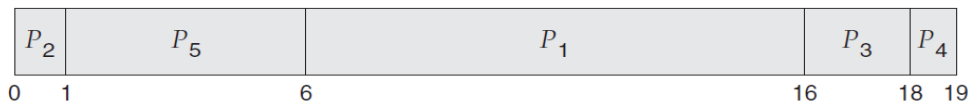
Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



- Priority Scheduling
  - A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
  - Equal-priority processes are scheduled in FCFS order.
  - Some systems use low numbers to represent low priority; others use low numbers for high priority. This time, we assume that low numbers represent high priority.
  - An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.
  - e.g.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

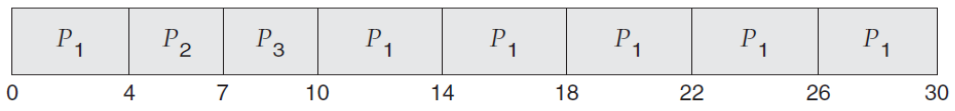
Using priority scheduling, we would schedule these processes according to the following Gantt chart:



- RR (Round-Robin Scheduling)
  - The RR scheduling algorithm is designed especially for timesharing systems.
  - It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
  - e.g.

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



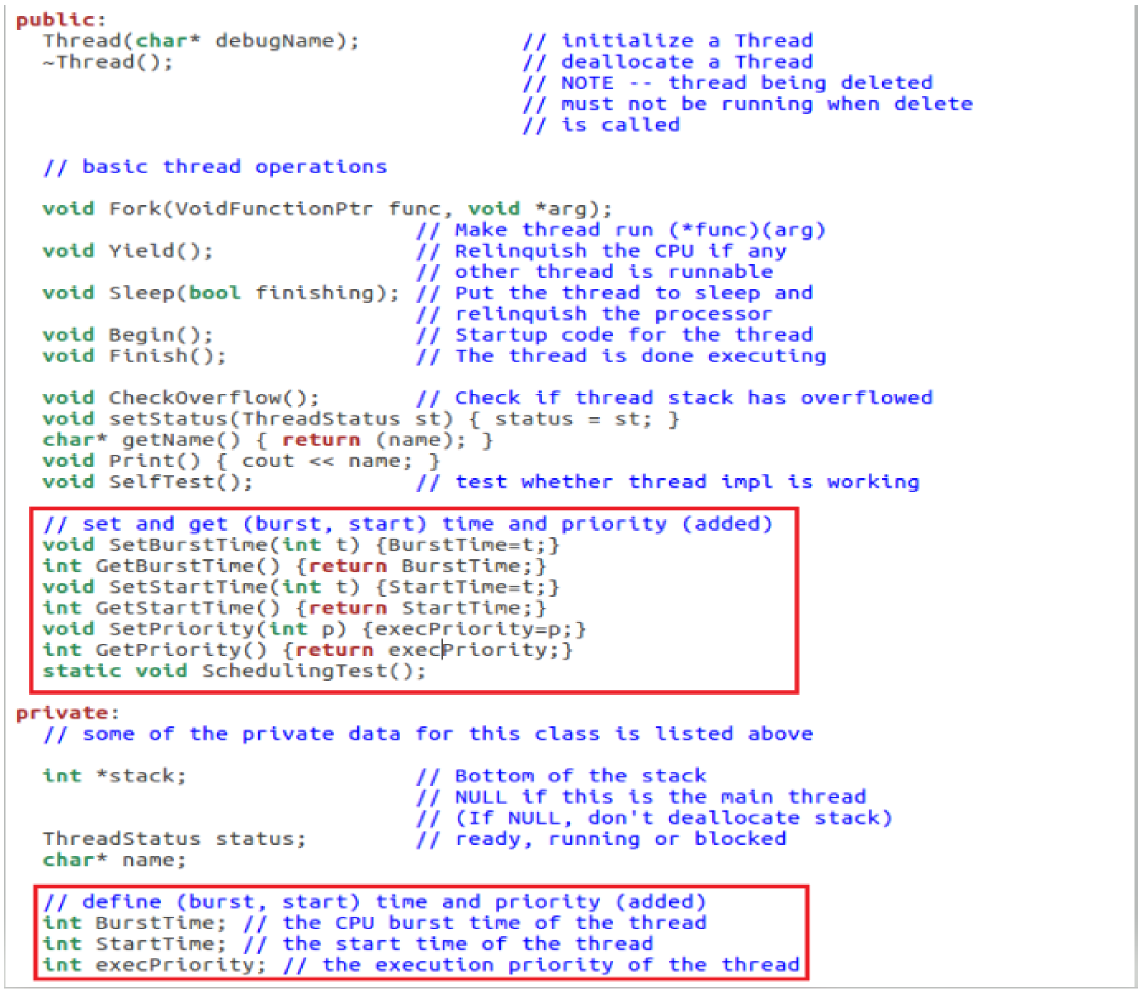
2.2 Plan

To implement multi-programming (multi-threads for Nachos), we design several scheduling algorithms to achieve this goal. So, we should modify files in threads.

First, the parameters (burst time, arrive time, priority) should be defined in `../threads/thread.h`. We also have to define functions so that we could easily get and set the parameters. Then we should add codes in `../threads/kernel.cc` for the kernel to read which type we use. In `../threads/scheduler.h` all the types we would like to implement should be enumerated. Then we create sorted lists or lists based on the concepts mentioned above. Since RR scheduling must satisfy context switch , we have to add this case in `../threads/alarm.cc` to let the threads preempt.

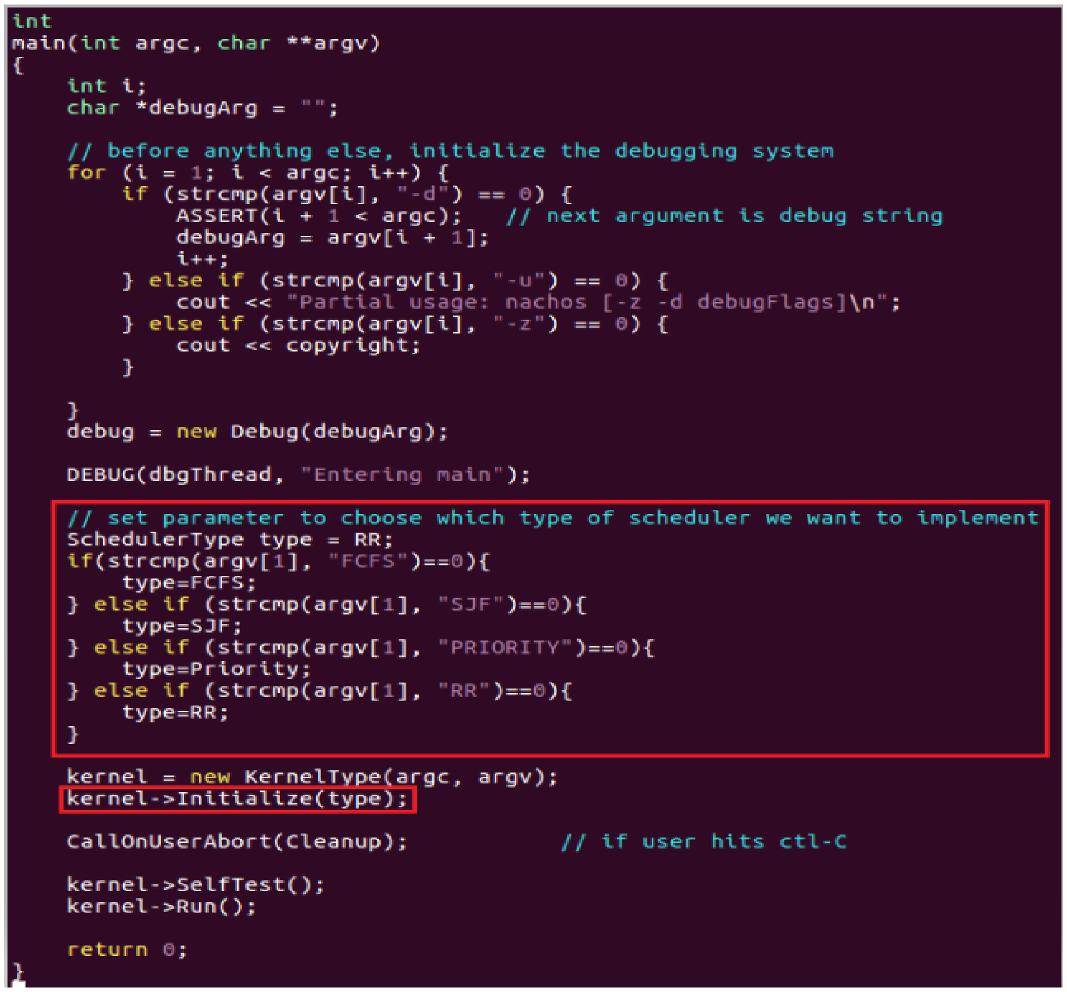
For the test file, I would like to create three threads and assign their burst time, arrival time and priority randomly. Then add the test function into `ThreadedKernel::Selftest` in `../threads/kernel.cc`, we could test our scheduling algorithms.

2.3 Code



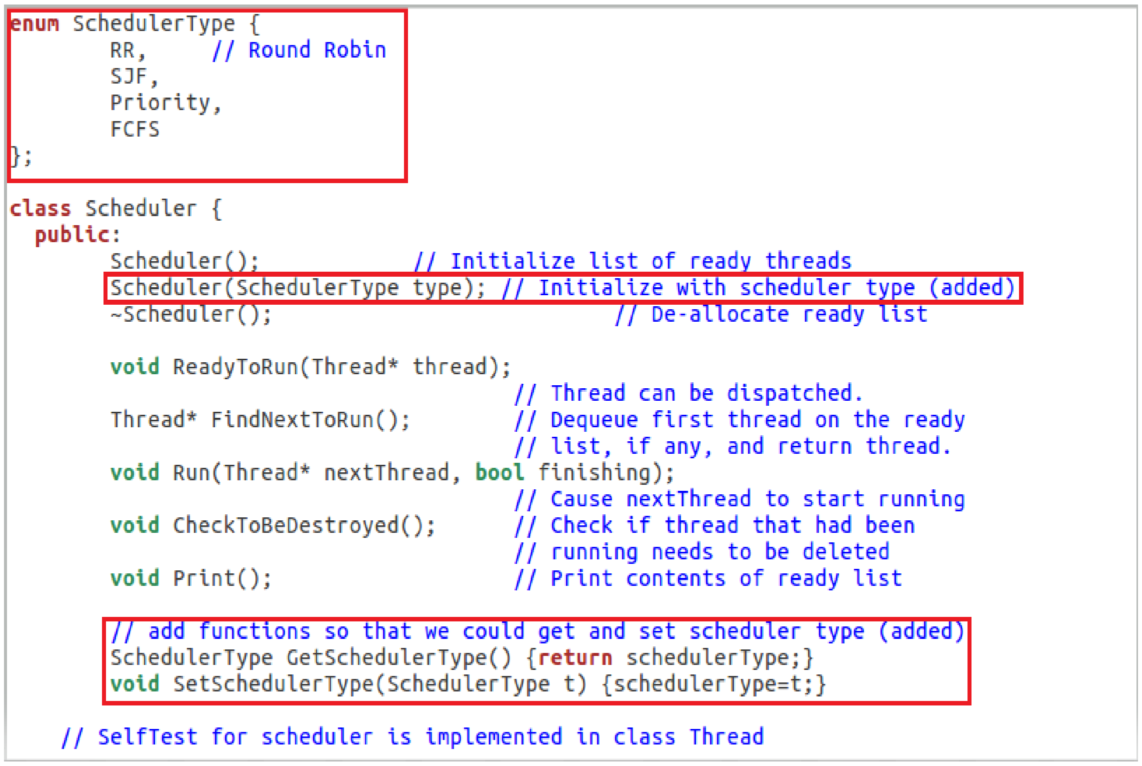
```
public:
  Thread(char* debugName);             // initialize a Thread
  ~Thread();                           // deallocate a Thread
                                       // NOTE -- thread being deleted
                                       // must not be running when delete
                                       // is called

  // basic thread operations

  void Fork(VoidFunctionPtr func, void *arg);
                               // Make thread run (*func)(arg)
  void Yield();                // Relinquish the CPU if any
                               // other thread is runnable
  void Sleep(bool finishing);  // Put the thread to sleep and
                               // relinquish the processor
  void Begin();                // Startup code for the thread
  void Finish();               // The thread is done executing

  void CheckOverflow();        // Check if thread stack has overflowed
  void setStatus(ThreadStatus st) { status = st; }
  char* getName() { return (name); }
  void Print() { cout << name; }
  void SelfTest();             // test whether thread impl is working

  // set and get (burst, start) time and priority (added)
  void SetBurstTime(int t) {BurstTime=t;}
  int GetBurstTime() {return BurstTime;}
  void SetStartTime(int t) {StartTime=t;}
  int GetStartTime() {return StartTime;}
  void SetPriority(int p) {execPriority=p;}
  int GetPriority() {return execPriority;}
  static void SchedulingTest();
private:
  // some of the private data for this class is listed above

  int *stack;                  // Bottom of the stack
                               // NULL if this is the main thread
                               // (If NULL, don't deallocate stack)
  ThreadStatus status;         // ready, running or blocked
  char* name;

  // define (burst, start) time and priority (added)
  int BurstTime; // the CPU burst time of the thread
  int StartTime; // the start time of the thread
  int execPriority; // the execution priority of the thread
```

As mentioned before, the three most important parameters for scheduling algorithms are burst time, arrival time and priority, respectively. In `../threads/thread.h`, there are data structures for managing threads. A thread represents sequential execution of code within a program. So the state of a thread originally includes the program counter, the process registers and the execution stack. To implement multi-threads, we should first define the parameters in this file. Also, we add several functions so that we could easily set and get all these parameters.



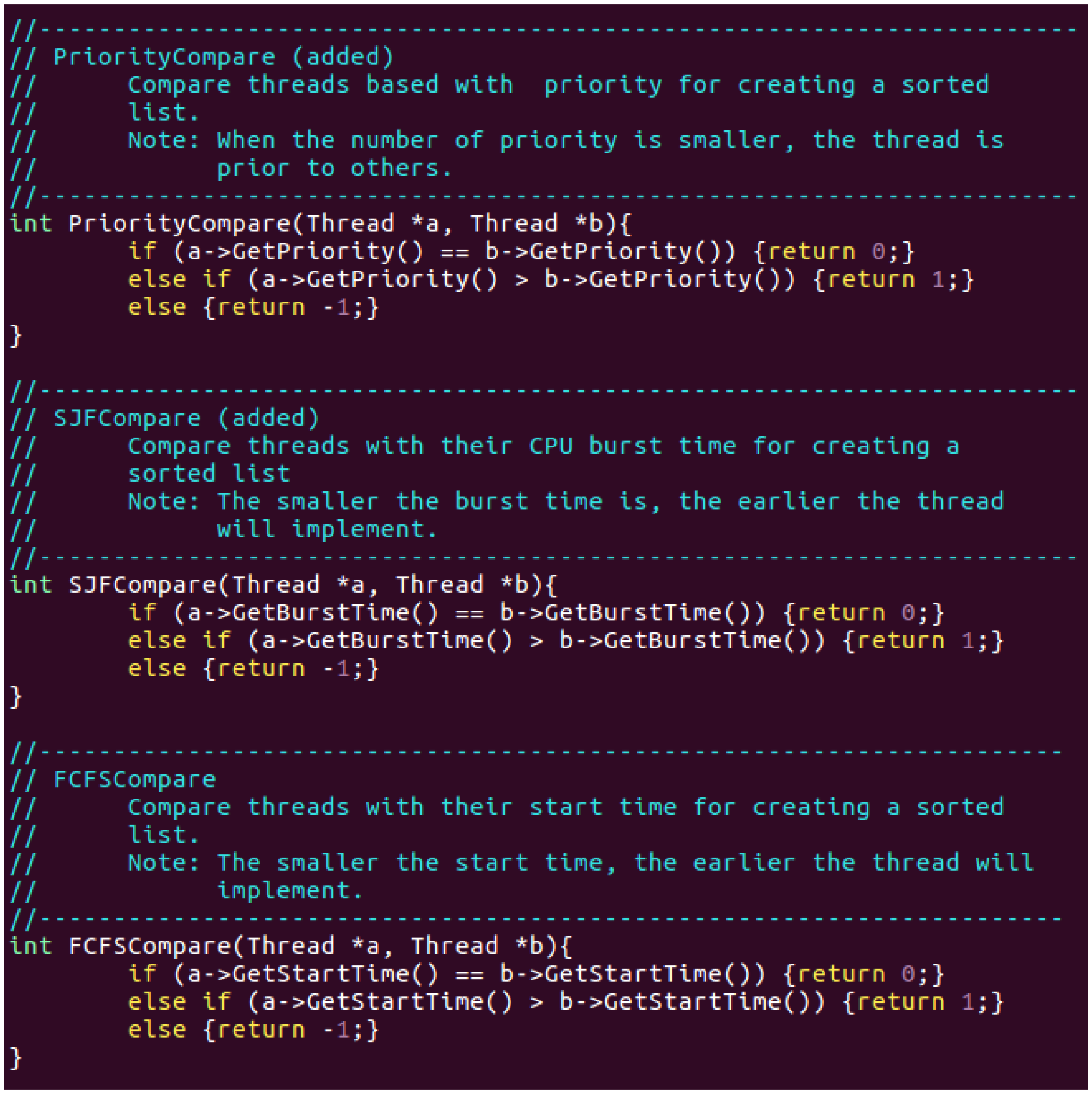
```
int
main(int argc, char **argv)
{
    int i;
    char *debugArg = "";

    // before anything else, initialize the debugging system
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-d") == 0) {
            ASSERT(i + 1 < argc);   // next argument is debug string
            debugArg = argv[i + 1];
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            cout << "Partial usage: nachos [-z -d debugFlags]\n";
        } else if (strcmp(argv[i], "-z") == 0) {
            cout << copyright;
        }

    }
    debug = new Debug(debugArg);

    DEBUG(dbgThread, "Entering main");

    // set parameter to choose which type of scheduler we want to implement
    SchedulerType type = RR;
    if(strcmp(argv[1], "FCFS")==0){
        type=FCFS;
    } else if (strcmp(argv[1], "SJF")==0){
        type=SJF;
    } else if (strcmp(argv[1], "PRIORITY")==0){
        type=Priority;
    } else if (strcmp(argv[1], "RR")==0){
        type=RR;
    }

    kernel = new KernelType(argc, argv);
    kernel->Initialize(type);

    CallOnUserAbort(Cleanup);           // if user hits ctl-C

    kernel->SelfTest();
    kernel->Run();

    return 0;
}
```

Since there are more than one algorithm, we should add codes for the kernel to read which scheduling algorithm we have chosen. In `../threads/kernel.cc`, `ThreadedKernel::ThreadedKernel()` interpret command line arguments in order to determine flags for the initialization. But we can also see the comments in `../threads/main.cc`. So, I add the instructions directly in `../threads/main.cc`. Here, "argc" is the number of command line argument and "argv" is an array of strings, one for each command line argument. We compare the second element of ”argv” with the scheduling algorithm. Then, the kernel could know which type we want to implement. Don’t forget to add a parameter for Initialize, since we have implemented other algorithms.

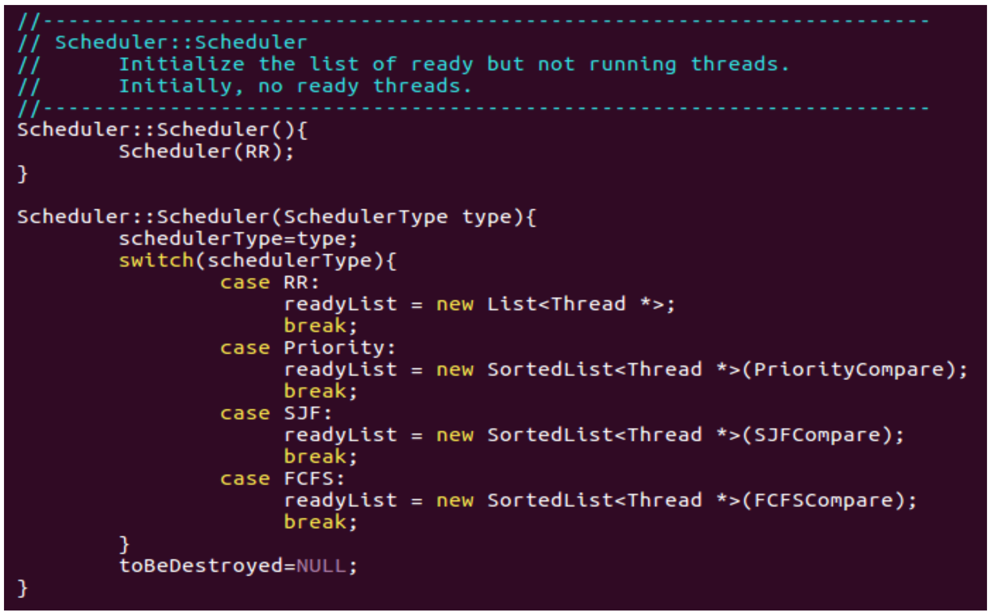


```
enum SchedulerType {
        RR,     // Round Robin
        SJF,
        Priority,
        FCFS
};

class Scheduler {
  public:
        Scheduler();            // Initialize list of ready threads
        Scheduler(SchedulerType type); // Initialize with scheduler type (added)
        ~Scheduler();                           // De-allocate ready list

        void ReadyToRun(Thread* thread);
                                        // Thread can be dispatched.
        Thread* FindNextToRun();        // Dequeue first thread on the ready
                                        // list, if any, and return thread.
        void Run(Thread* nextThread, bool finishing);
                                        // Cause nextThread to start running
        void CheckToBeDestroyed();      // Check if thread that had been
                                        // running needs to be deleted
        void Print();                   // Print contents of ready list

        // add functions so that we could get and set scheduler type (added)
        SchedulerType GetSchedulerType() {return schedulerType;}
        void SetSchedulerType(SchedulerType t) {schedulerType=t;}

    // SelfTest for scheduler is implemented in class Thread
```

The original codes in `../threads/scheduler.h` has enumerated several scheduler type (RR, SJF, Priority). Actually, the only type implemented is RR. Therefore, we have to write algorithms for other type in `../threads/scheduler.cc`. Also, I implement a new type ”FCFS”. Since there are more than one algorithm, we should give parameter to scheduler to initialize. Here, I add functions such that I can easily get the scheduler type which I have chosen.

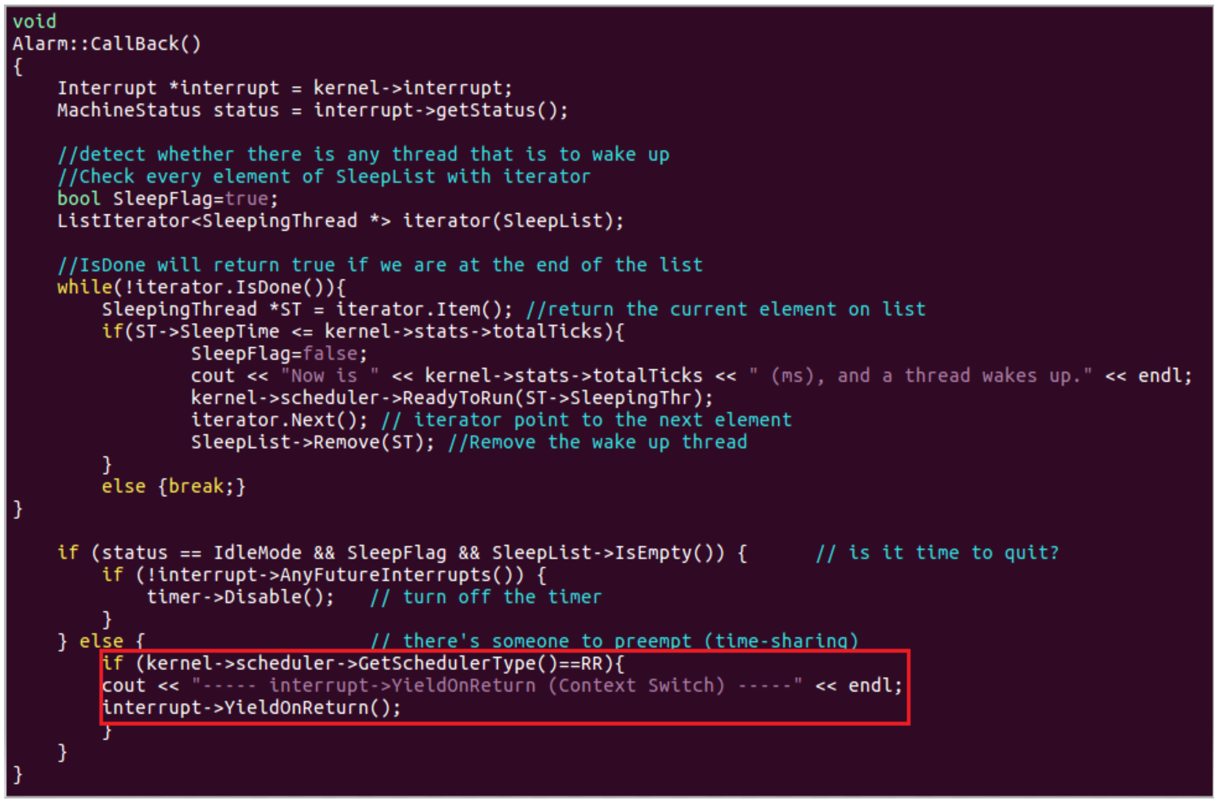

```
//-----------------------------------------------------------------------
// PriorityCompare (added)
//      Compare threads based with  priority for creating a sorted
//      list.
//      Note: When the number of priority is smaller, the thread is
//            prior to others.
//-----------------------------------------------------------------------
int PriorityCompare(Thread *a, Thread *b){
        if (a->GetPriority() == b->GetPriority()) {return 0;}
        else if (a->GetPriority() > b->GetPriority()) {return 1;}
        else {return -1;}
}

//-----------------------------------------------------------------------
// SJFCompare (added)
//      Compare threads with their CPU burst time for creating a
//      sorted list
//      Note: The smaller the burst time is, the earlier the thread
//            will implement.
//-----------------------------------------------------------------------
int SJFCompare(Thread *a, Thread *b){
        if (a->GetBurstTime() == b->GetBurstTime()) {return 0;}
        else if (a->GetBurstTime() > b->GetBurstTime()) {return 1;}
        else {return -1;}
}

//----------------------------------------------------------------------
// FCFSCompare
//      Compare threads with their start time for creating a sorted
//      list.
//      Note: The smaller the start time, the earlier the thread will
//            implement.
//----------------------------------------------------------------------
int FCFSCompare(Thread *a, Thread *b){
        if (a->GetStartTime() == b->GetStartTime()) {return 0;}
        else if (a->GetStartTime() > b->GetStartTime()) {return 1;}
        else {return -1;}
}
```

In `../threads/scheduler.cc`, I first create sorted lists for the algorithms. For "FCFS", we have to compare the threads with the arrival time. The thread which arrives earlier will first be implemented. For "SJF", the burst time of the threads should be compared. The shorter the burst time of the thread is, the earlier it would be implemented. For "Priority", the thread should be first implemented, if its value of priority is smaller. For "RR", every thread would be implemented in turn in a short period of time. So it is necessary to implement context switch.



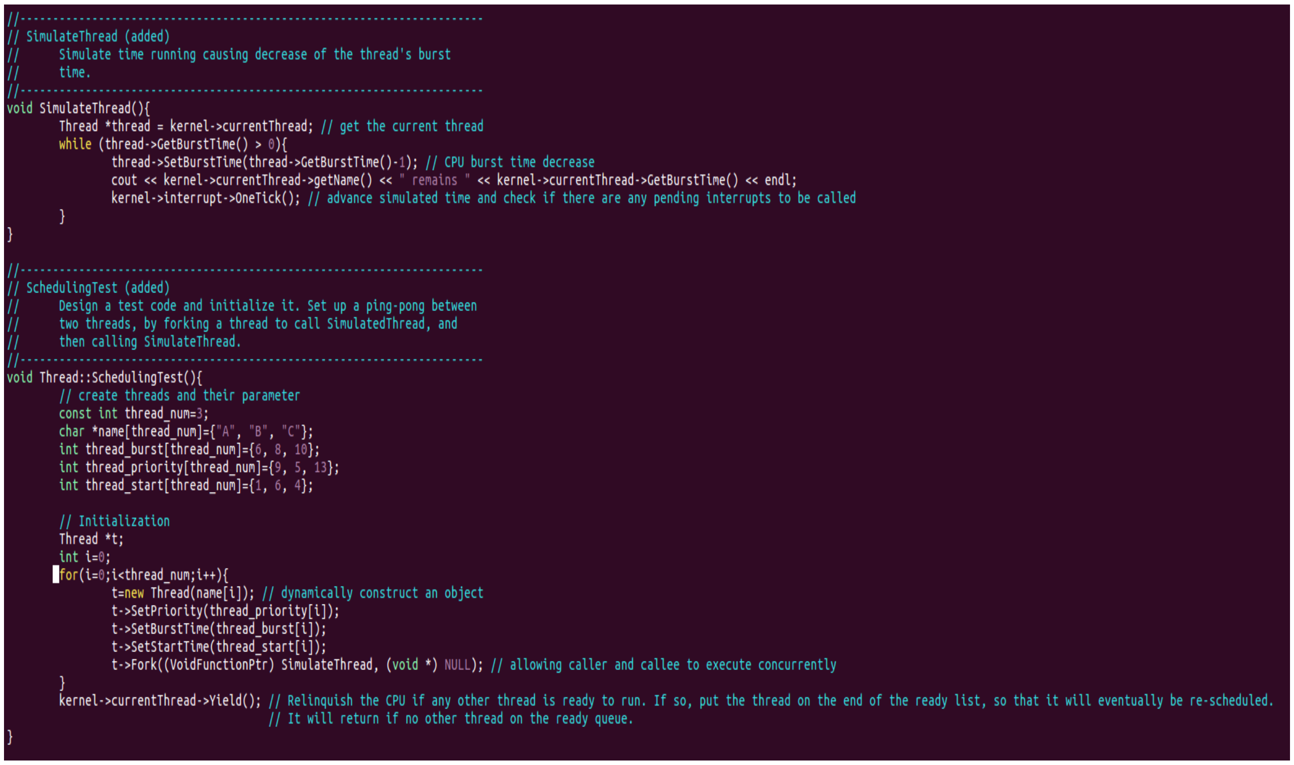
```
//----------------------------------------------------------------------
// Scheduler::Scheduler
//      Initialize the list of ready but not running threads.
//      Initially, no ready threads.
//----------------------------------------------------------------------
Scheduler::Scheduler(){
        Scheduler(RR);
}

Scheduler::Scheduler(SchedulerType type){
        schedulerType=type;
        switch(schedulerType){
                case RR:
                     readyList = new List<Thread *>;
                     break;
                case Priority:
                     readyList = new SortedList<Thread *>(PriorityCompare);
                     break;
                case SJF:
                     readyList = new SortedList<Thread *>(SJFCompare);
                     break;
                case FCFS:
                     readyList = new SortedList<Thread *>(FCFSCompare);
                     break;
        }
        toBeDestroyed=NULL;
}
```

Then, initialize the scheduler. If the user doesn't assign the scheduler type, the system will set it Round-Robin (RR) directly. Also, we have to initialize scheduler with parameter, because there are now many types to choose. For "RR" case, we just create a list and put the threads into the list. For the other cases, we should create sorted list based on the criteria mentioned above.



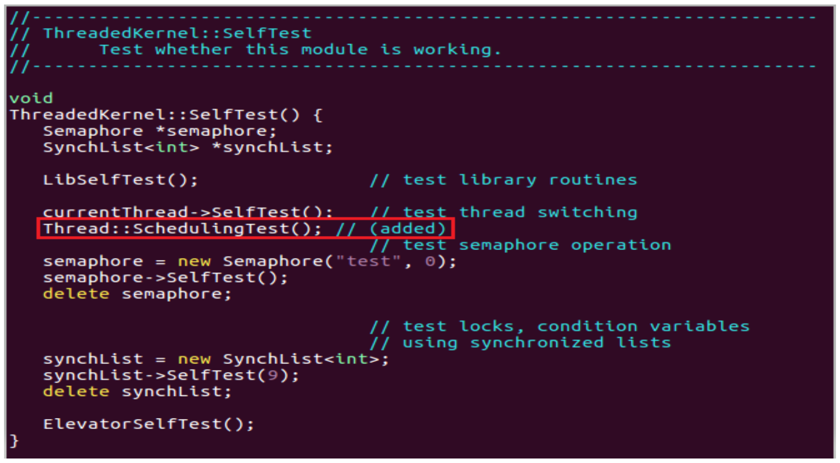
```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    //detect whether there is any thread that is to wake up
    //Check every element of SleepList with iterator
    bool SleepFlag=true;
    ListIterator<SleepingThread *> iterator(SleepList);

    //IsDone will return true if we are at the end of the list
    while(!iterator.IsDone()){
        SleepingThread *ST = iterator.Item(); //return the current element on list
        if(ST->SleepTime <= kernel->stats->totalTicks){
                SleepFlag=false;
                cout << "Now is " << kernel->stats->totalTicks << " (ms), and a thread wakes up." << endl;
                kernel->scheduler->ReadyToRun(ST->SleepingThr);
                iterator.Next(); // iterator point to the next element
                SleepList->Remove(ST); //Remove the wake up thread
        }
        else {break;}
}

    if (status == IdleMode && SleepFlag && SleepList->IsEmpty()) {      // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
        }
    } else {                    // there's someone to preempt (time-sharing)
        if (kernel->scheduler->GetSchedulerType()==RR){
        cout << "----- interrupt->YieldOnReturn (Context Switch) -----" << endl;
        interrupt->YieldOnReturn();
        }
    }
}
```

For ”RR”, there is someone to preempt at any time. So I add the instructions like the red block in the picture, and I can know when the context switch happens.



```
//----------------------------------------------------------------------
// SimulateThread (added)
//      Simulate time running causing decrease of the thread's burst
//      time.
//----------------------------------------------------------------------
void SimulateThread(){
        Thread *thread = kernel->currentThread; // get the current thread
        while (thread->GetBurstTime() > 0){
                thread->SetBurstTime(thread->GetBurstTime()-1); // CPU burst time decrease
                cout << kernel->currentThread->getName() << " remains " << kernel->currentThread->GetBurstTime() << endl;
                kernel->interrupt->OneTick(); // advance simulated time and check if there are any pending interrupts to be called
        }
}

//----------------------------------------------------------------------
// SchedulingTest (added)
//      Design a test code and initialize it. Set up a ping-pong between
//      two threads, by forking a thread to call SimulatedThread, and
//      then calling SimulateThread.
//----------------------------------------------------------------------
void Thread::SchedulingTest(){
        // create threads and their parameter
        const int thread_num=3;
        char *name[thread_num]={"A", "B", "C"};
        int thread_burst[thread_num]={6, 8, 10};
        int thread_priority[thread_num]={9, 5, 13};
        int thread_start[thread_num]={1, 6, 4};

        // Initialization
        Thread *t;
        int i=0;
        for(i=0;i<thread_num;i++){
                t=new Thread(name[i]); // dynamically construct an object
                t->SetPriority(thread_priority[i]);
                t->SetBurstTime(thread_burst[i]);
                t->SetStartTime(thread_start[i]);
                t->Fork((VoidFunctionPtr) SimulateThread, (void *) NULL); // allowing caller and callee to execute concurrently
        }
        kernel->currentThread->Yield(); // Relinquish the CPU if any other thread is ready to run. If so, put the thread on the end of the ready list, so that it will eventually be re-scheduled.
                                        // It will return if no other thread on the ready queue.
}
```

From the slides of homework 2, we need to design a test code. In `../threads/thread.cc`, I first create three threads and all its parameters. Then import all these parameters into the threads respectively. To allow caller and callee to execute concurrently, the instruction ”Fork” should be used. It would call ”SimulatedThread”, then the CPU burst time of the current thread would decrease as time passes.

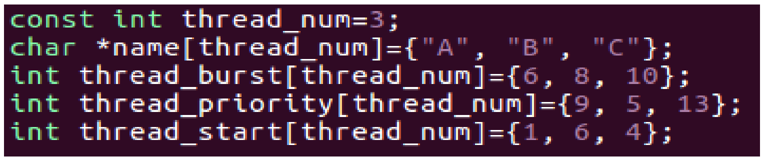


```
//----------------------------------------------------------------------
// ThreadedKernel::SelfTest
//      Test whether this module is working.
//----------------------------------------------------------------------

void
ThreadedKernel::SelfTest() {
   Semaphore *semaphore;
   SynchList<int> *synchList;

   LibSelfTest();                   // test library routines

   currentThread->SelfTest();       // test thread switching
   Thread::SchedulingTest(); // (added)
                                    // test semaphore operation
   semaphore = new Semaphore("test", 0);
   semaphore->SelfTest();
   delete semaphore;

                                    // test locks, condition variables
                                    // using synchronized lists
   synchList = new SynchList<int>;
   synchList->SelfTest(9);
   delete synchList;

   ElevatorSelfTest();
}
```

Lastly, we should add `Thread::SchedulingTest` into `ThreadedKernel::Selftest` in `../threads/kernel.cc` to test whether this module is working.

2.4 Result

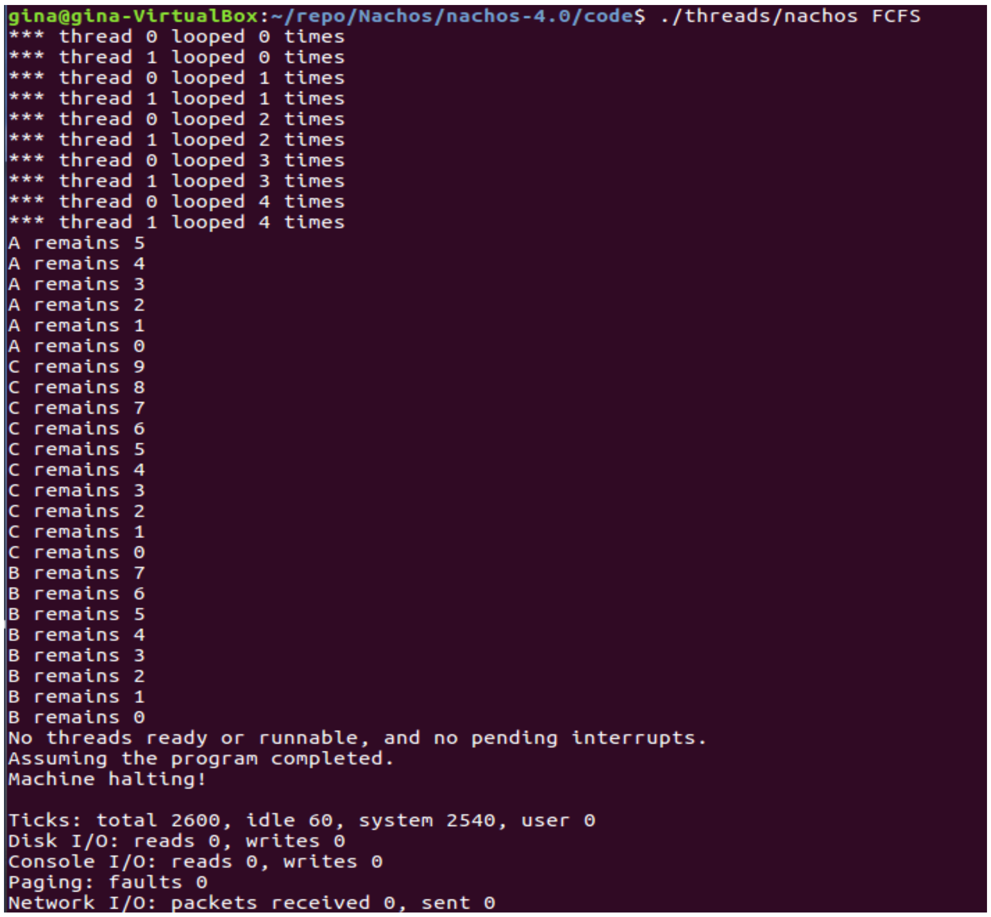
The parameters are set as the following pictures:



```
const int thread_num=3;
char *name[thread_num]={"A", "B", "C"};
int thread_burst[thread_num]={6, 8, 10};
int thread_priority[thread_num]={9, 5, 13};
int thread_start[thread_num]={1, 6, 4};
```

2.4.1 FCFS

For FCFS, we can predict that the order of implementation will be A, C, B. Now, implement it on Nachos,

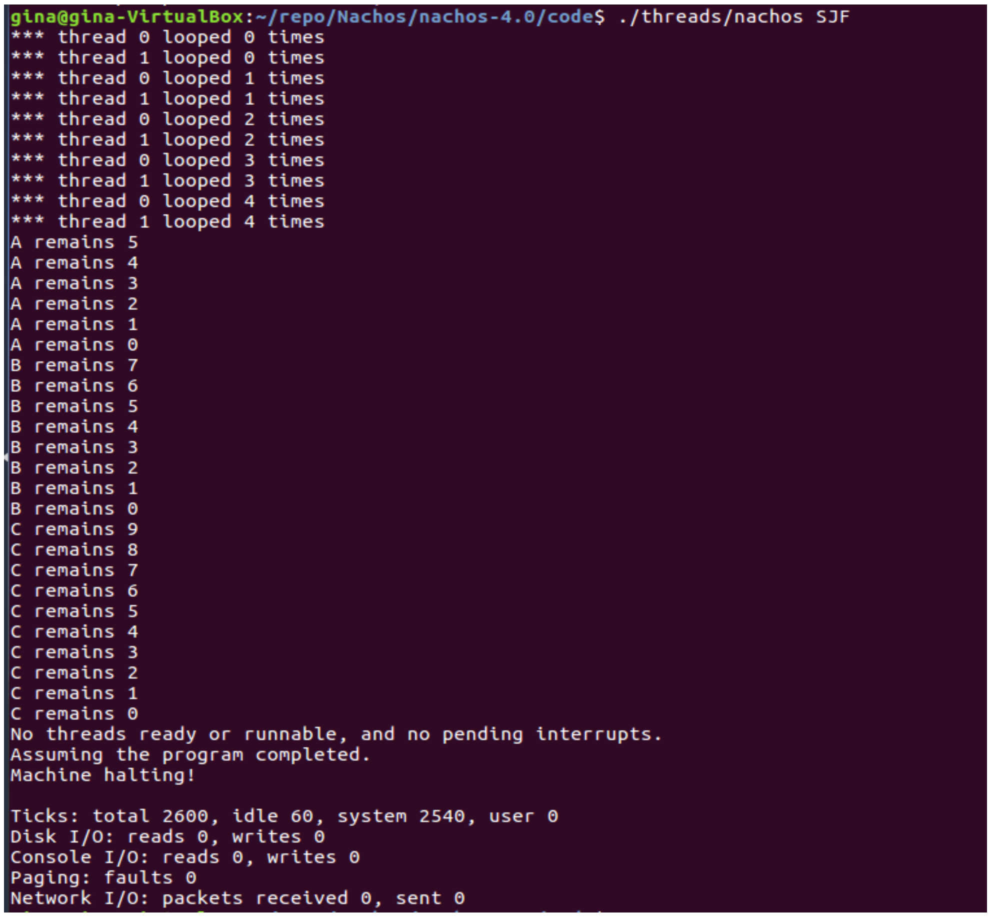


```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code$ ./threads/nachos FCFS
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
A remains 5
A remains 4
A remains 3
A remains 2
A remains 1
A remains 0
C remains 9
C remains 8
C remains 7
C remains 6
C remains 5
C remains 4
C remains 3
C remains 2
C remains 1
C remains 0
B remains 7
B remains 6
B remains 5
B remains 4
B remains 3
B remains 2
B remains 1
B remains 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 60, system 2540, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

The result is same as our prediction. Every thread will finish until its burst time gets to 0.

2.4.2 SJF

For SJF, we can predict that the order of implementation will be A, B, C. Now, implement it on Nachos,

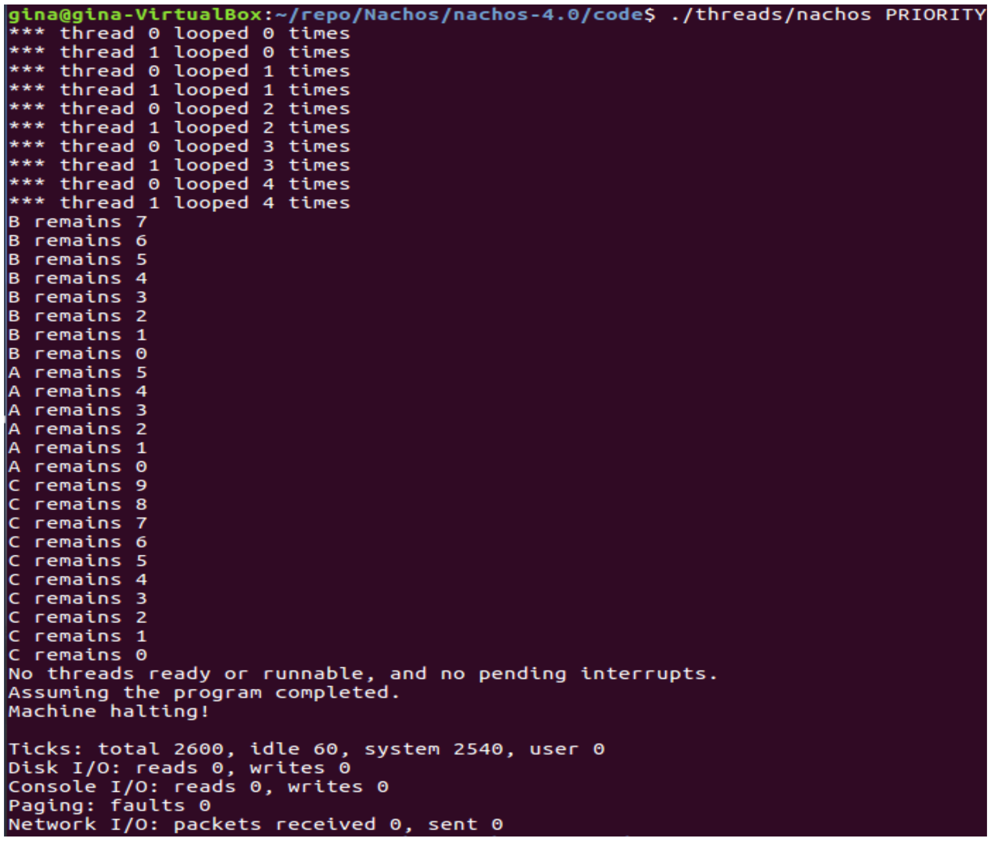


```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code$ ./threads/nachos SJF
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
A remains 5
A remains 4
A remains 3
A remains 2
A remains 1
A remains 0
B remains 7
B remains 6
B remains 5
B remains 4
B remains 3
B remains 2
B remains 1
B remains 0
C remains 9
C remains 8
C remains 7
C remains 6
C remains 5
C remains 4
C remains 3
C remains 2
C remains 1
C remains 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 60, system 2540, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

The result is same as our prediction. Every thread will finish until its burst time gets to 0.

2.4.3 Priority

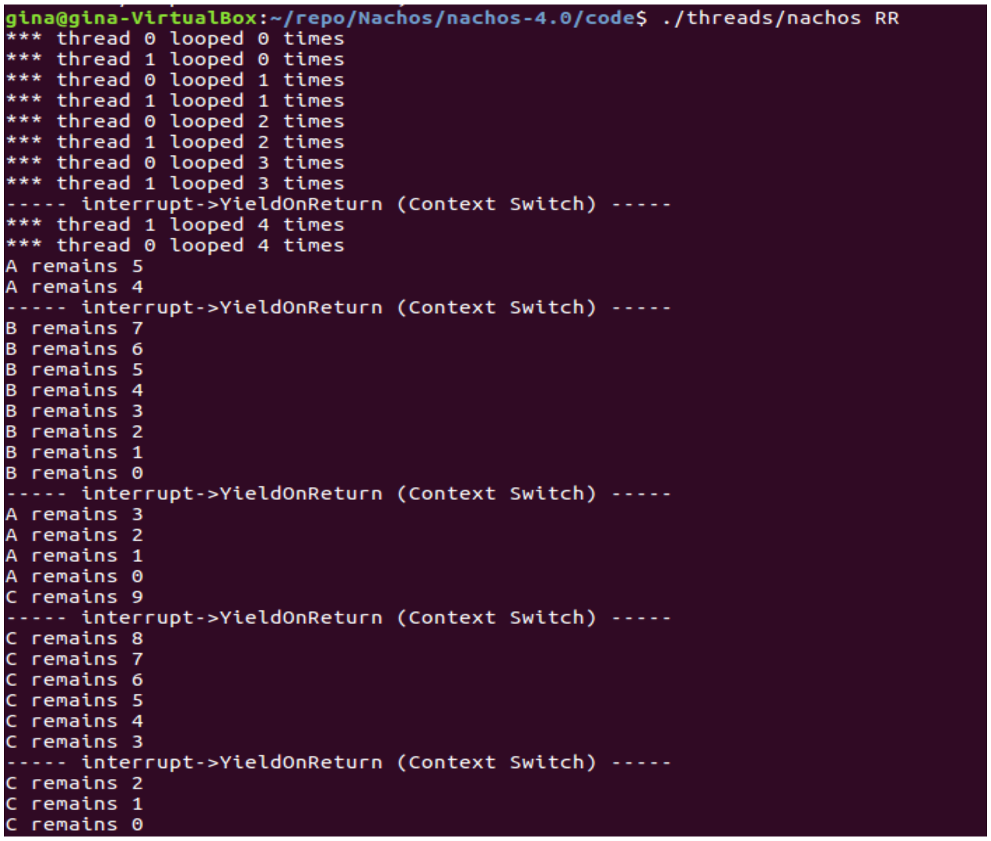
For Priority, I let low numbers represent high priority. So we can predict that the order of implementation will be B, A, C.



```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code$ ./threads/nachos PRIORITY
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
B remains 7
B remains 6
B remains 5
B remains 4
B remains 3
B remains 2
B remains 1
B remains 0
A remains 5
A remains 4
A remains 3
A remains 2
A remains 1
A remains 0
C remains 9
C remains 8
C remains 7
C remains 6
C remains 5
C remains 4
C remains 3
C remains 2
C remains 1
C remains 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 60, system 2540, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

The result is same as our prediction. Every thread will finish until its burst time gets to 0.

2.4.4 RR



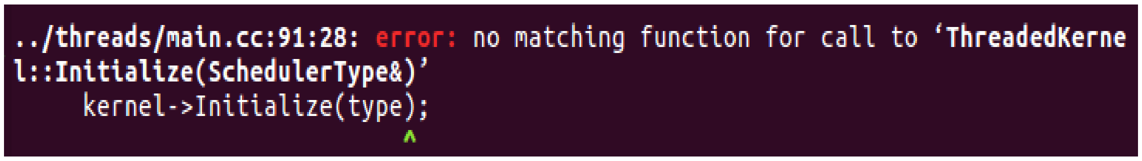
```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code$ ./threads/nachos RR
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
----- interrupt->YieldOnReturn (Context Switch) -----
*** thread 1 looped 4 times
*** thread 0 looped 4 times
A remains 5
A remains 4
----- interrupt->YieldOnReturn (Context Switch) -----
B remains 7
B remains 6
B remains 5
B remains 4
B remains 3
B remains 2
B remains 1
B remains 0
----- interrupt->YieldOnReturn (Context Switch) -----
A remains 3
A remains 2
A remains 1
A remains 0
C remains 9
----- interrupt->YieldOnReturn (Context Switch) -----
C remains 8
C remains 7
C remains 6
C remains 5
C remains 4
C remains 3
----- interrupt->YieldOnReturn (Context Switch) -----
C remains 2
C remains 1
C remains 0
```

Context switch would happen at any time.

2.5 Problem

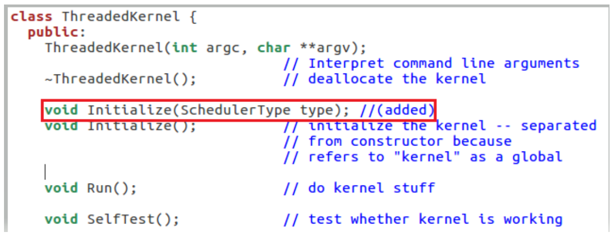
1. What would happen if we don't assign the scheduler type for `kernel->Initialize()` in `../threads/main.cc`?
   A: segmentation fault

2. How to solve the problem in the following picture?

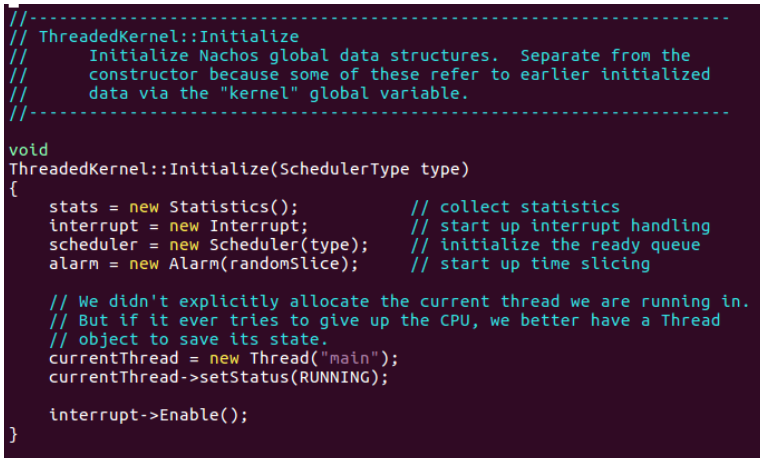


```
../threads/main.cc:91:28: error: no matching function for call to 'ThreadedKerne
l::Initialize(SchedulerType&)'
    kernel->Initialize(type);
                           ^
```

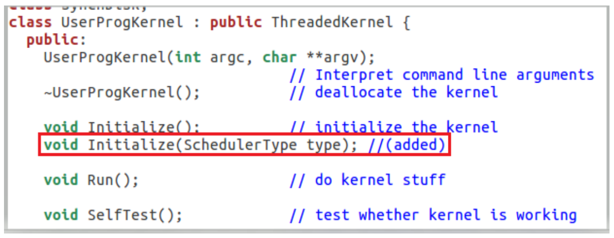
   A: File `../threads/main.h` includes `../threads/kernel.h`, `../userprog/userkernel.h` and `../network/netkernel.h`. Since we have modified Initialization in `../threads/main.cc`, we also need to modify Initialization in the head files and `../threads/kernel.cc`, `../userprog/userkernel.cc` and `../network/netkernel.cc`.



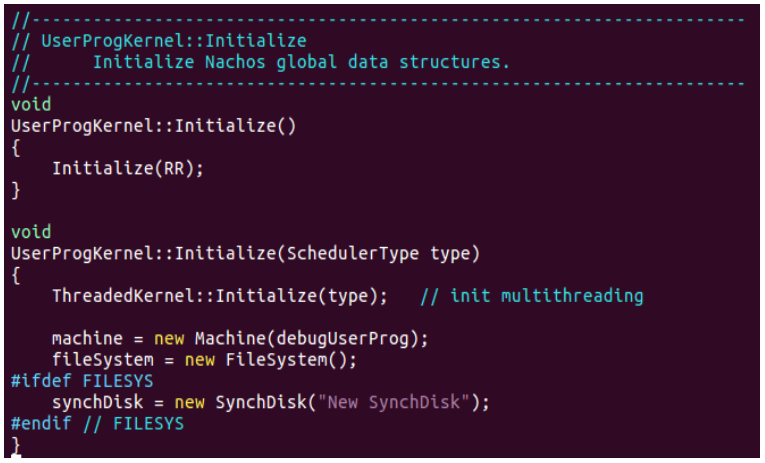
```
class ThreadedKernel {
  public:
    ThreadedKernel(int argc, char **argv);
                                // Interpret command line arguments
    ~ThreadedKernel();          // deallocate the kernel

    void Initialize(SchedulerType type); //(added)
    void Initialize();          // initialize the kernel -- separated
                                // from constructor because
                                // refers to "kernel" as a global

    |
    void Run();                 // do kernel stuff

    void SelfTest();            // test whether kernel is working
```



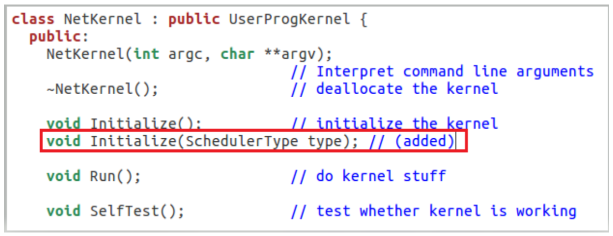
```
//----------------------------------------------------------------------
// ThreadedKernel::Initialize
//      Initialize Nachos global data structures.  Separate from the
//      constructor because some of these refer to earlier initialized
//      data via the "kernel" global variable.
//----------------------------------------------------------------------

void
ThreadedKernel::Initialize(SchedulerType type)
{
    stats = new Statistics();           // collect statistics
    interrupt = new Interrupt;          // start up interrupt handling
    scheduler = new Scheduler(type);    // initialize the ready queue
    alarm = new Alarm(randomSlice);     // start up time slicing

    // We didn't explicitly allocate the current thread we are running in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.
    currentThread = new Thread("main");
    currentThread->setStatus(RUNNING);

    interrupt->Enable();
}
```

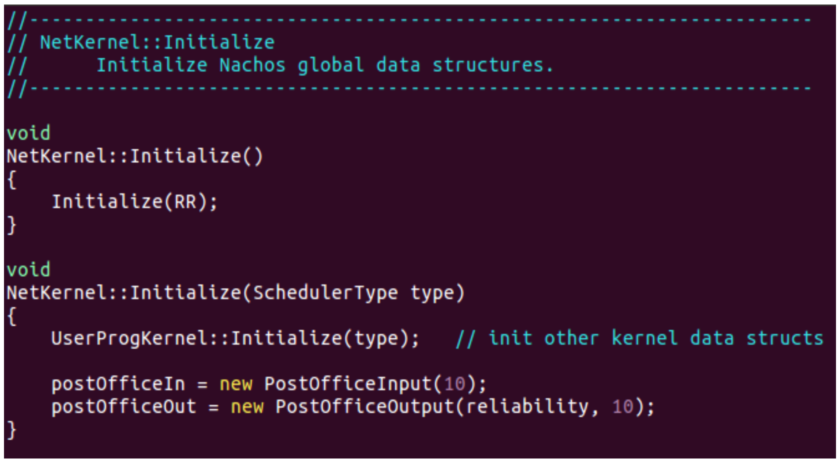


```
class UserProgKernel : public ThreadedKernel {
  public:
    UserProgKernel(int argc, char **argv);
                                // Interpret command line arguments
    ~UserProgKernel();          // deallocate the kernel

    void Initialize();          // initialize the kernel
    void Initialize(SchedulerType type); //(added)

    void Run();                 // do kernel stuff

    void SelfTest();            // test whether kernel is working
```



```
//----------------------------------------------------------------------
// UserProgKernel::Initialize
//      Initialize Nachos global data structures.
//----------------------------------------------------------------------
void
UserProgKernel::Initialize()
{
    Initialize(RR);
}

void
UserProgKernel::Initialize(SchedulerType type)
{
    ThreadedKernel::Initialize(type);   // init multithreading

    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS
}
```


```
class NetKernel : public UserProgKernel {
  public:
    NetKernel(int argc, char **argv);
                                    // Interpret command line arguments
    ~NetKernel();                   // deallocate the kernel

    void Initialize();              // initialize the kernel
    void Initialize(SchedulerType type); // (added)

    void Run();                     // do kernel stuff

    void SelfTest();                // test whether kernel is working
```


```
//----------------------------------------------------------------------
// NetKernel::Initialize
//      Initialize Nachos global data structures.
//----------------------------------------------------------------------

void
NetKernel::Initialize()
{
    Initialize(RR);
}

void
NetKernel::Initialize(SchedulerType type)
{
    UserProgKernel::Initialize(type);   // init other kernel data structs

    postOfficeIn = new PostOfficeInput(10);
    postOfficeOut = new PostOfficeOutput(reliability, 10);
}
```

3 Question

In the line 60 of `Machine::Run()` , `kernel->interrupt` will be first set as UserMode (void setStatus(MachineStatus st) status = st; in `interrupt.h`). Then we get into an infinite loop and execute `Machine::OneInstruction(Instruction *instr)` (in `../machine/mipssim.cc:116`). There is a `case OP_SYSCALL` (in `../machine/mipssim.cc:557`) representing system call. If we call system call, `RaiseException(SyscallException, 0);` will be implemented. Also, there is an instruction `kernel->interrupt->Onetick()` in the loop to advance simulated time, check whether there is any pending interrupt ready to fire and may cause context switch (`Interrupt::YieldOnReturn()`).

`Machine::RaiseException(ExceptionType which, int badVAddr)` is implemented in `../machine/machine.cc`. The parameter `which` represents the cause of the kernel trap and `badVAddr` is the virtual address causing the trap. First, we will put the virtual address into the register. Then, finish anything in progress through `DelayedLoad(0,0)` . Now, we will be trapped into SystemMode through `kernel->interrupt->setStatus(SystemMode)` . `ExceptionHandler(which)` will let interrupts be enabled at this point.

`ExceptionHadler(ExceptionType which)` is defined in

`../userprog/exception.cc`. This time, we are going to implement system call ”Sleep”. Thus, `kernel->alarm->WaitUntil(val)` will be called. In `Alarm::WaitUntil`, ”Sleep” will be implemented. Also, interrupt will be disabled and `OneTick()` will help us advance time ticks.

When the value of totalTicks is equal to 100, `Interrupt::OneTick()` will implement `CheIfDue(FALSE)`. In `CheIfDue(FALSE)`, `next` will first be set as `pending->Front()` to catch timer interrupt. Then, load the value of `DelayedLoad(0,0)` through `kernel->machine->DelayedLoad(0,0)`. Set `inHandler` as TRUE to set up the state of processing interrupt. Then we will get into while loop to process all the interrupts that is time up. In the loop, `next` will first be set as `pending->RemoveFront()` to process the first element in the pending list. Then `next->callOnInterrupt->CallBack()` will call `Timer->CallBack()` and also `Alarm->CallBack()`. In `Alarm::CallBack()`, timer will be disabled if the status is IdleMode and there is no one in pending list. `Interrupt->YieldOnReturn` may also be implemented.

4 References

- co-work with R11942140 Yu-Wei, Chang
- Work from Predecessor
- Work from Predecessor
- Work from Predecessor
- Work from Predecessor