**Assignment 1 Report**

CIS 657 Principles of Operating Systems

Name: Xiaoqian Huang

SUID: 878174727

NetId: xhuang62

1. **Time Consuming**
2. Analyze the problem, determine specifications, create the design: 3 days
3. Implement the design: 12 hours
4. Test/debug the program: 8 hours (make the program compiled and run, fix logical errors, fix segmentation fault errors, print and analyze large amount of output data to debug).
5. **Design/Solution for each requirement**
6. Task 1

In task 1 we are asked to simulate some IO events.

Then I created an IoEvent class (ioevent.[h|cc]) to simulate this event. When an event is created, a random wait time will be set according to the type of the IO events: 0-2000 ticks for write operation and 10000-20000 for read opration. And when the IO interrupt is raised, the process will call the oprateIO() to indicate the simulated event is finished.

To simulate the process of io event, I use IoThread() (threadtest.cc) to perform the operations:

1. create IO event.
2. insert the event to a global io enent list which is declared in kernel.h.
3. set alarm (ioalarm.[h|cc]) to schedule the io interrupt according to the waiting time generated by the event.
4. sleep the thread until the scheduled time. It will be woken up by ioalarm.
5. when is woken up, it will print out the information about this ioalarm.
6. if all the io events are raised and the threads are finished, the system will be halted.

To separate up the usage of time interrupt and io interrupt, I created ioalarm.[h|cc] and iotimer.[h|cc] to set up and raise the event at the specific time. Once the time is up, an interrupt will process iotimer’s CallBcak() function, and iotimer will call ioalram’s CallBack() function which is exactly the IO interrupt handler mentioned in the requirement.

When the IO interrupt handler is called, it will check the on due interrupt and scheduled it to run. So the on due thread is put in ready list and waiting to run.

Above all, the IO events are simulated in nachos.

1. Task 2

In task 2, we are required to change the schedule algorithm from RR to CFS as well as change the data structure of ready list from list to red-black tree.

For CFS, we need to calculate and keep updating of virtual runtime once the situation of ready list changed.

**Scheduling**

To calculate the virtual runtime, I use the following equation:

VirtualRunTime(new) = VirtualRunTime(old) + Current thread Weight / Total Weight of threads in ready list \* Time slice + Decay;

As a result, I need to record the following attributes:

1. record the virtual run time in the thread.
2. allow the caller to set weight of a thread.
3. generated a random decay (0-50) for calculation
4. record of total weight of all the threads that are scheduled in ready list which means I need to keep track of the change of total weight in ready list. So when a thread is scheduled to run and put in ready list or remove from ready list, the total weight will be changed dynamically. To be fair, when the total weight is changed, all the threads stored in ready list who have the old total weight must be updated as well as the the virtual runtime.

Specifically, the virtual runtime updated because of the change of total weight must be rollback for the previous virtual runtime and recalculated. So the previous virtual runtime should be recorded in thread.

To make the data updated correctly, in ReadyToRun() in scheduler.cc, I first update the total weight, then update the whole tree’s virtual runtime (due to the change of total weight), and update the new comer’s virtual runtime (due to the new turn) and then insert the new thread in the ready list.

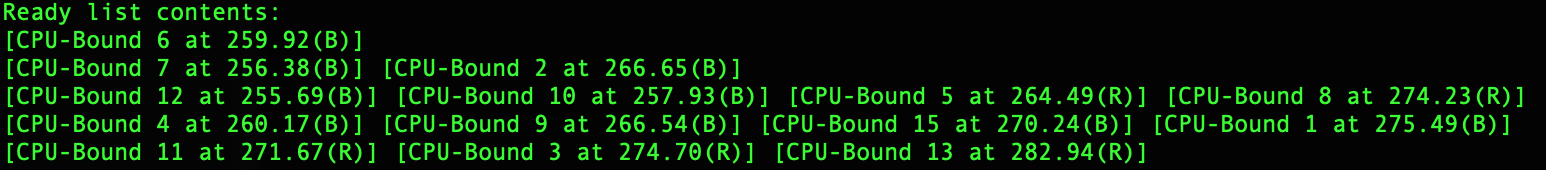
Similar, for FindNextToRun() in scheduler.cc, when a thread is removed to run, the total weight also changed. So I need to update the total weight and then virtual runtime in the whole tree.

Also, to keep all the threads having fair ticks to run, I need to update the time interrupt dynamically according to the total number running in parallel (assuming all the threads have same weight).

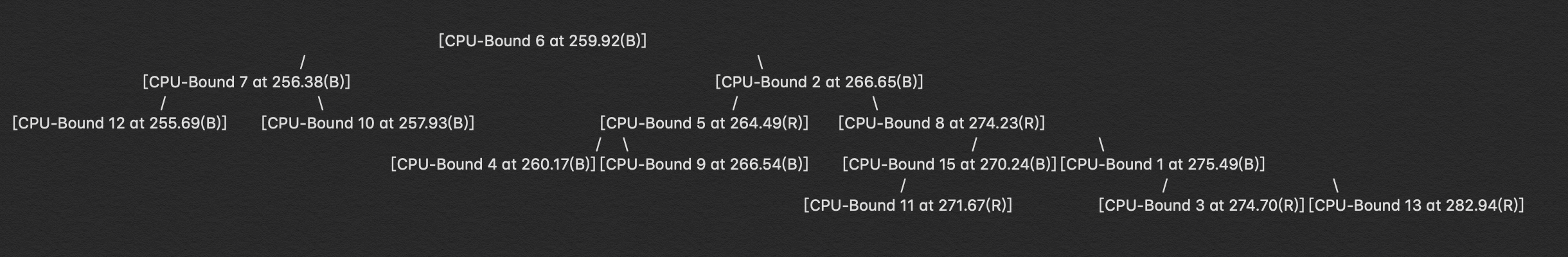
**Data Structure**

I use red-black tree to replace the original list for ready list. So I can get a binary search tree ordered by virtual runtime.

In my program, I will print out content of ready list in Breadth-First-Search way once a thread is added to ready list. So it will look like this:



It’s the output of following tree which is obviously the red-black tree we want.



To retrieve the thread with smallest virtual runtime, we just need to first find the left-most child in the tree, or the smallest node is the root itself.

**Time Slice**

As mentioned above, to keep all the thread has fair CPU tick time, I need to update the scheduled time interrupt when scheduling next thread to run.

So in Alarm.[h|cc] and Timer.[h|cc] I overload the setInterrupt() method, allowing scheduler to set time interrupt dynamically. In addition, to maintain the correct schedule of time in the CallBack() method in timer.cc, a last interrupt time will be recorded and set to trigger next time interrupt.

In my program, I make the time quantum to 1000 so the divided slice for each thread would not be so tiny. Also, to make the output accurately, I change the SystemTick in stats.h to 1 so the time interrupt can be raised at more accurate time.

That means, if I have 15 threads running in total, then every thread has 1000/15 = 66 ticks to run.

1. Task 3

To test the IO event simulation in Task1, I created pure IO-Bound thread.

To test the CFS algorithm as well as the red-black tree in Task2, I created pure CPU-Bound thread which has only oneTick() operations to simulate the computation of CPU.

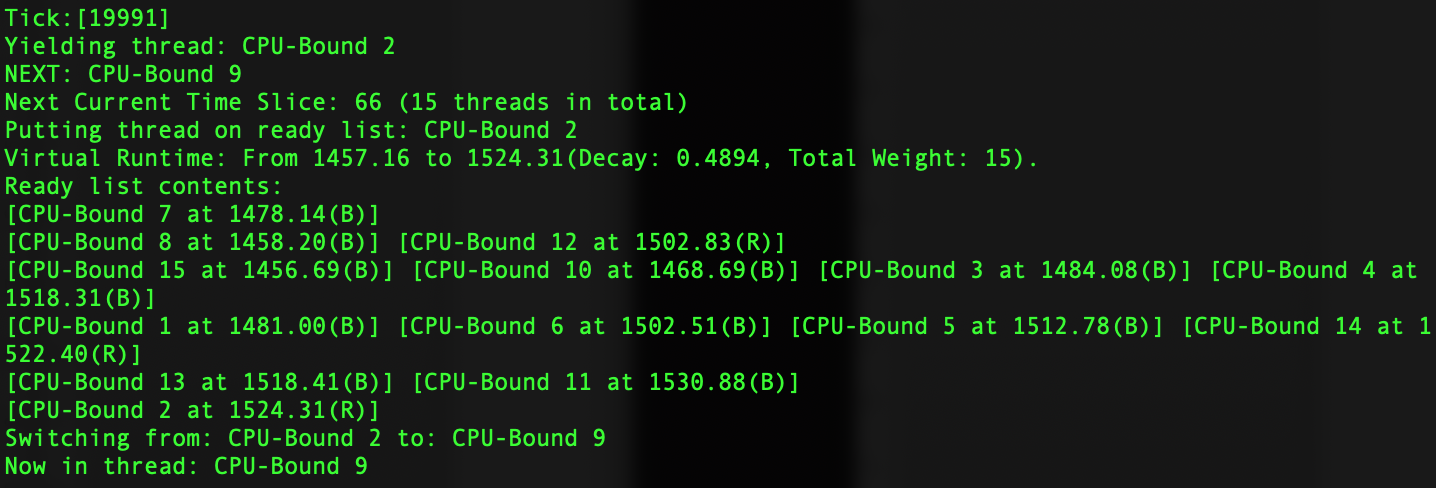
Also, I have MIX thread who is mixed up of CPU computation and IO operations.

I created 5 pure IO threads, 1 MIX thread and 15 CPU threads, which is 21 threads running in total.

That means 6 IO events will be scheduled to raise interrupt and run. When all 6 events are finished, the system will halt.

I printed out the total tick number of current slice, then which is the next thread to be run, the full information of the new inserted thread and the content of red-black tree.

If we run it in –d t debug mode, we can see the thread changing, which indicating everything is running in the expected way. (more explanation detailed in testing part)



**3. Implementation of the solution**

In threadtest.cc

#include "kernel.h"

#include "main.h"

#include "thread.h"

#include "ioevent.h"

#include "ioalarm.h"

void

IoThread(IoAlarm \*ioAlarm) // launch IO operation

{

Statistics \*stats = kernel->stats;

IoEvent \*newEvent = new IoEvent(rand()%2, rand()%20, kernel->currentThread);

newEvent->setCompletionTime(newEvent->getWaitingTime() + stats->totalTicks); // set completion time

cout << "IO Event from [" << kernel->currentThread->getName() << "] is created and added to queue ( interrupt at " << newEvent->getCompletionTime() << " ticks ). \n";

kernel->ioEventQueue->Insert(newEvent);

ioAlarm->SetAlarm(newEvent->getCompletionTime(), newEvent->getType());

kernel->interrupt->SetLevel(IntOff);

kernel->currentThread->Sleep(FALSE);

if (kernel->interrupt->getLevel() == IntOff) {

kernel->interrupt->SetLevel(IntOn);

}

newEvent->operateIo();

cout << "Total ticks: " << kernel->stats->totalTicks << "\n";

cout << "IO Event from [" << kernel->currentThread->getName() << "] is finished ( scheduled completed at " << newEvent->getCompletionTime() << " ticks ). \n";

if (kernel->getTotalFinishedIoThreadNum() >= 6 ) { // terminate the system when all the io events finished

kernel->interrupt->Halt();

}

kernel->currentThread->Finish();

}

void

CPUThread(int which) { // simulate the cpu computation

for(int i = 0; i < which; i++) {

if (kernel->interrupt->getLevel() == IntOff) {

kernel->interrupt->SetLevel(IntOn);

}

kernel->interrupt->OneTick();

}

}

void

MIXThread(IoAlarm\* ioAlarm) { // mix cpu and io

CPUThread(50);

IoThread(ioAlarm);

CPUThread(300);

}

char\* getThreadName(const char\* type, int i) { // build the thread name to identify each thread

int intLen = 1, reminder = 0;

char\* num = new char[10];

reminder = i % 10;

num[0] = ' ';

num[1] = (char)(reminder + '0');

while (i / 10 > 0) {

intLen++;

i = i / 10;

reminder = i % 10;

num[intLen] = (char)(reminder + '0');

}

// reverse string

int j = intLen;

for (int s = 1; s <= (intLen + 1) / 2; s++) {

char temp = num[s];

num[s] = num[j];

num[j] = temp;

j--;

}

int totalLen = strlen(type) + intLen;

char \*n\_str = new char[totalLen + 1];

strcpy(n\_str, type);

strcat(n\_str, num);

return n\_str;

}

void

ThreadTest()

{

srand((unsigned int)time(0));

IoAlarm \*ioAlarm = new IoAlarm(FALSE);

Thread \*t;

int CPUCycles = 10000;

t = new Thread("IO-Bound 1");

t->setWeight(1);

t->Fork((VoidFunctionPtr)IoThread, (void \*)ioAlarm);

t = new Thread("IO-Bound 2");

t->setWeight(1);

t->Fork((VoidFunctionPtr)IoThread, (void \*)ioAlarm);

t = new Thread("IO-Bound 3");

t->setWeight(1);

t->Fork((VoidFunctionPtr)IoThread, (void \*)ioAlarm);

t = new Thread("IO-Bound 4");

t->setWeight(1);

t->Fork((VoidFunctionPtr)IoThread, (void \*)ioAlarm);

t = new Thread("IO-Bound 5");

t->setWeight(1);

t->Fork((VoidFunctionPtr)IoThread, (void \*)ioAlarm);

t = new Thread("MIX-Thread");

t->setWeight(1);

t->Fork((VoidFunctionPtr)MIXThread, (void \*)ioAlarm);

t = new Thread("CPU-Bound 1");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 2");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 3");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 4");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 5");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 6");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 7");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 8");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 9");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 10");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 11");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 12");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 13");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 14");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

t = new Thread("CPU-Bound 15");

t->setWeight(1);

t->Fork((VoidFunctionPtr)CPUThread, (void \*)CPUCycles);

}

ioevent.h

#pragma once

#ifndef IOEVENT\_H

#define IOEVENT\_H

#include "thread.h"

class IoEvent {

public:

IoEvent(int ioType, int parameter, Thread \*callingThread) {

\_ioType = ioType;

\_parameter = parameter;

\_callingThread = callingThread;

\_setWaitingTime();

}

int getWaitingTime();

void startProcessing(int currentTime);

int getCompletionTime();

void setCompletionTime(int time);

void operateIo();

void CallBack(); // interrupt handler

Thread\* getCallingThread();

int getType(); // 0: write

private:

void \_setWaitingTime();

int \_ioType; // 0: write, 1: read

Thread \*\_callingThread;

char\* \_buffer;

int \_parameter; // size of buffer (counting by char)

int \_waitingTime;

int \_completionTime; // random waiting time + starting running time

int \_executionTime = 0;

};

#endif // IOEVENT\_H

ioevent.cc

#include "ioevent.h"

void IoEvent::\_setWaitingTime(){ // generate waiting time randomly

if (\_ioType == 0) { // write, small waiting time, 0-2000

\_waitingTime = rand() % 2001;

}

else { // read, large waiting time, 10000-20000

\_waitingTime = rand() % 10001 + 10000 ;

}

}

int IoEvent::getWaitingTime() {

return \_waitingTime;

}

int IoEvent::getCompletionTime() {

return \_completionTime;

}

void IoEvent::startProcessing(int completionTime) {

\_completionTime = completionTime;

}

void IoEvent::setCompletionTime(int time) {

\_completionTime = time;

}

void IoEvent::operateIo() {

if (\_ioType == 0) {

cout << "=========== Write Operation Finished! =========== \n";

}

else {

cout << "=========== Read Operation Finished! ============ \n";

}

}

Thread\* IoEvent::getCallingThread() {

return \_callingThread;

}

// io interrupt handler

void IoEvent::CallBack() {

cout << "Call back to ioevent: completion time[" << \_completionTime << "]\n";

}

int IoEvent::getType() {

return \_ioType;

}

iotimer.h

#ifndef IOTIMER\_H

#define IOTIMER\_H

#include "copyright.h"

#include "utility.h"

#include "callback.h"

#include "kernel.h"

#include "../machine/interrupt.h"

// The following class defines a hardware timer.

class IoTimer : public CallBackObj {

public:

IoTimer(bool doRandom, CallBackObj \*toCall);

// Initialize the timer, and callback to "toCall"

// every time slice.

virtual ~IoTimer() {}

void Disable() { disable = TRUE; }

// Turn timer device off, so it doesn't

// generate any more interrupts.

void SetInterrupt(int TotalTicks, int type); // cause an interrupt to occur in the

private:

bool randomize; // set if we need to use a random timeout delay

CallBackObj \*callOnDue; // call back to ioalarm

bool disable; // turn off the timer device after next

// interrupt.

void CallBack(); // called internally when the hardware

// timer generates an interrupt

// the future after a fixed or random

// delay

Statistics \*stats;

};

#endif // IOTIMER\_H

iotimer.cc

#include "copyright.h"

#include "iotimer.h"

#include "main.h"

#include "sysdep.h"

//----------------------------------------------------------------------

// Timer::Timer

// Initialize a hardware timer device. Save the place to call

// on each interrupt, and then arrange for the timer to start

// generating interrupts.

//

// "doRandom" -- if true, arrange for the interrupts to occur

// at random, instead of fixed, intervals.

// "toCall" is the interrupt handler to call when the timer expires.

//----------------------------------------------------------------------

IoTimer::IoTimer(bool doRandom, CallBackObj \*toCall)

{

randomize = doRandom;

callOnDue = toCall;

disable = FALSE;

stats = kernel->stats;

}

//----------------------------------------------------------------------

// Timer::CallBack

// Routine called when interrupt is generated by the hardware

// timer device. Schedule the next interrupt, and invoke the

// interrupt handler.

//----------------------------------------------------------------------

void

IoTimer::CallBack()

{

// invoke the Nachos interrupt handler for this device

callOnDue->CallBack();

//SetInterrupt(); // do last, to let software interrupt handler

// decide if it wants to disable future interrupts

}

//----------------------------------------------------------------------

// Timer::SetInterrupt

// Cause a timer interrupt to occur in the future, unless

// future interrupts have been disabled. The delay is either

// fixed or random.

//----------------------------------------------------------------------

void

IoTimer::SetInterrupt(int TotalTicks, int type)

{

if (!disable) {

int delay = TimerTicks;

if (randomize) {

delay = 1 + (RandomNumber() % (TimerTicks \* 2));

}

// schedule the next io interrupt

if (type == 0) { // write event

kernel->interrupt->Schedule(this, TotalTicks - stats->totalTicks, IoIntWrite);

}

else {

kernel->interrupt->Schedule(this, TotalTicks - stats->totalTicks, IoIntRead);

}

}

}

rb\_tree\_nachos.h

// rb\_tree\_nachos.h

// data structure for storing thread entities in ready queue

#pragma once

#ifndef RB\_TREE\_NACHOS\_H

#define RB\_TREE\_NACHOS\_H

#include "debug.h"

#include "list.h"

#include "thread.h"

static const char \* NodeNames[] = { "R", "B", "DB" };

enum Color { RED, BLACK, DOUBLE\_BLACK };

struct Node

{

Thread\* data;

int color;

Node \*left, \*right, \*parent;

explicit Node(Thread\*);

};

class RBTree

{

private:

Node \*root;

public:

void rotateLeft(Node \*&); // left rotate the subtree

void rotateRight(Node \*&); // right rotate the subtree

void fixInsertRBTree(Node \*&); // insert adjust

void fixDeleteRBTree(Node \*&); // delete adjust

void inorderBST(Node \*&); // print the tree by in order sequence

void preorderBST(Node \*&); // print the tree by pre order sequence

int getColor(Node \*&); // get the color

void setColor(Node \*&, int); // set the color

Node \*minValueNode(Node \*&); //Find the node with minimum value

Node \*minValueNode(); // public

Node \*maxValueNode(Node \*&); //Find the node with maximum value

Node \*getANode(); // get a node from the tree

Node\* insertBST(Node \*&, Node \*&); // insert node

Node\* deleteBST(Node \*&, Thread\*); // delete node

int getBlackHeight(Node \*); // get black height of the tree

bool isEmpty(); // if the tree is empty

void printTree();

public:

RBTree();

void insertValue(Thread\*);

void deleteValue(Thread\*);

void merge(RBTree);

void inorder();

void preorder();

};

#endif //RED\_BLACK\_TREE\_RBTREE\_H

rb\_tree\_nachos.cc

#include "rb\_tree\_nachos.h"

Node::Node(Thread \*data) {

this->data = data;

color = RED;

left = right = parent = NULL;

}

RBTree::RBTree() {

root = NULL;

}

int RBTree::getColor(Node \*&node) {

if (node == NULL)

return BLACK;

return node->color;

}

void RBTree::setColor(Node \*&node, int color) {

if (node == NULL)

return;

node->color = color;

}

Node\* RBTree::insertBST(Node \*&root, Node \*&ptr) {

if (root == NULL)

return ptr;

if (ptr->data->getVirtualRunTime() < root->data->getVirtualRunTime()) {

root->left = insertBST(root->left, ptr);

root->left->parent = root;

}

else if (ptr->data->getVirtualRunTime() > root->data->getVirtualRunTime()) {

root->right = insertBST(root->right, ptr);

root->right->parent = root;

}

return root;

}

void RBTree::insertValue(Thread\* n) {

Node \*node = new Node(n);

root = insertBST(root, node);

fixInsertRBTree(node);

}

void RBTree::rotateLeft(Node \*&ptr) {

Node \*right\_child = ptr->right;

ptr->right = right\_child->left;

if (ptr->right != NULL)

ptr->right->parent = ptr;

right\_child->parent = ptr->parent;

if (ptr->parent == NULL)

root = right\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = right\_child;

else

ptr->parent->right = right\_child;

right\_child->left = ptr;

ptr->parent = right\_child;

}

void RBTree::rotateRight(Node \*&ptr) {

Node \*left\_child = ptr->left;

ptr->left = left\_child->right;

if (ptr->left != NULL)

ptr->left->parent = ptr;

left\_child->parent = ptr->parent;

if (ptr->parent == NULL)

root = left\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = left\_child;

else

ptr->parent->right = left\_child;

left\_child->right = ptr;

ptr->parent = left\_child;

}

void RBTree::fixInsertRBTree(Node \*&ptr) {

Node \*parent = NULL;

Node \*grandparent = NULL;

while (ptr != root && getColor(ptr) == RED && getColor(ptr->parent) == RED) {

parent = ptr->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node \*uncle = grandparent->right;

if (getColor(uncle) == RED) {

setColor(uncle, BLACK);

setColor(parent, BLACK);

setColor(grandparent, RED);

ptr = grandparent;

}

else {

if (ptr == parent->right) {

rotateLeft(parent);

ptr = parent;

parent = ptr->parent;

}

rotateRight(grandparent);

swap(parent->color, grandparent->color);

ptr = parent;

}

}

else {

Node \*uncle = grandparent->left;

if (getColor(uncle) == RED) {

setColor(uncle, BLACK);

setColor(parent, BLACK);

setColor(grandparent, RED);

ptr = grandparent;

}

else {

if (ptr == parent->left) {

rotateRight(parent);

ptr = parent;

parent = ptr->parent;

}

rotateLeft(grandparent);

swap(parent->color, grandparent->color);

ptr = parent;

}

}

}

setColor(root, BLACK);

}

void RBTree::fixDeleteRBTree(Node \*&node) {

if (node == NULL)

return;

if (node == root) {

root = NULL;

return;

}

if (getColor(node) == RED || getColor(node->left) == RED || getColor(node->right) == RED) {

Node \*child = node->left != NULL ? node->left : node->right;

if (node == node->parent->left) {

node->parent->left = child;

if (child != NULL)

child->parent = node->parent;

setColor(child, BLACK);

delete (node);

}

else {

node->parent->right = child;

if (child != NULL)

child->parent = node->parent;

setColor(child, BLACK);

delete (node);

}

}

else {

Node \*sibling = NULL;

Node \*parent = NULL;

Node \*ptr = node;

setColor(ptr, DOUBLE\_BLACK);

while (ptr != root && getColor(ptr) == DOUBLE\_BLACK) {

parent = ptr->parent;

if (ptr == parent->left) {

sibling = parent->right;

if (getColor(sibling) == RED) {

setColor(sibling, BLACK);

setColor(parent, RED);

rotateLeft(parent);

}

else {

if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {

setColor(sibling, RED);

if (getColor(parent) == RED)

setColor(parent, BLACK);

else

setColor(parent, DOUBLE\_BLACK);

ptr = parent;

}

else {

if (getColor(sibling->right) == BLACK) {

setColor(sibling->left, BLACK);

setColor(sibling, RED);

rotateRight(sibling);

sibling = parent->right;

}

setColor(sibling, parent->color);

setColor(parent, BLACK);

setColor(sibling->right, BLACK);

rotateLeft(parent);

break;

}

}

}

else {

sibling = parent->left;

if (getColor(sibling) == RED) {

setColor(sibling, BLACK);

setColor(parent, RED);

rotateRight(parent);

}

else {

if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {

setColor(sibling, RED);

if (getColor(parent) == RED)

setColor(parent, BLACK);

else

setColor(parent, DOUBLE\_BLACK);

ptr = parent;

}

else {

if (getColor(sibling->left) == BLACK) {

setColor(sibling->right, BLACK);

setColor(sibling, RED);

rotateLeft(sibling);

sibling = parent->left;

}

setColor(sibling, parent->color);

setColor(parent, BLACK);

setColor(sibling->left, BLACK);

rotateRight(parent);

break;

}

}

}

}

if (node == node->parent->left)

node->parent->left = NULL;

else

node->parent->right = NULL;

delete(node);

setColor(root, BLACK);

}

}

Node\* RBTree::deleteBST(Node \*&root, Thread\* data) {

if (root == NULL)

return root;

if (data->getVirtualRunTime() < root->data->getVirtualRunTime())

return deleteBST(root->left, data);

if (data->getVirtualRunTime() > root->data->getVirtualRunTime())

return deleteBST(root->right, data);

if (root->left == NULL || root->right == NULL)

return root;

Node \*temp = minValueNode(root->right);

root->data = temp->data;

return deleteBST(root->right, temp->data);

}

void RBTree::deleteValue(Thread\* data) {

Node \*node = deleteBST(root, data);

fixDeleteRBTree(node);

}

void RBTree::inorderBST(Node \*&ptr) {

if (ptr == NULL)

return;

inorderBST(ptr->left);

cout << "[" << ptr->data->getName() << " at " << ptr->data->getVirtualRunTime() << "(" << NodeNames[ptr->color] << ")]";

inorderBST(ptr->right);

}

void RBTree::inorder() {

inorderBST(root);

}

void RBTree::preorderBST(Node \*&ptr) {

if (ptr == NULL)

return;

cout << ptr->data << " " << ptr->color << endl;

preorderBST(ptr->left);

preorderBST(ptr->right);

}

void RBTree::preorder() {

preorderBST(root);

cout << "-------" << endl;

}

Node \*RBTree::minValueNode(Node \*&node) {

Node \*ptr = node;

while (ptr->left != NULL)

ptr = ptr->left;

return ptr;

}

Node \* RBTree::minValueNode(){

minValueNode(root);

}

Node\* RBTree::maxValueNode(Node \*&node) {

Node \*ptr = node;

while (ptr->right != NULL)

ptr = ptr->right;

return ptr;

}

int RBTree::getBlackHeight(Node \*node) {

int blackheight = 0;

while (node != NULL) {

if (getColor(node) == BLACK)

blackheight++;

node = node->left;

}

return blackheight;

}

void RBTree::merge(RBTree rbTree2) {

Thread\* temp;

Node \*c, \*temp\_ptr;

Node \*root1 = root;

Node \*root2 = rbTree2.root;

int initialblackheight1 = getBlackHeight(root1);

int initialblackheight2 = getBlackHeight(root2);

if (initialblackheight1 > initialblackheight2) {

c = maxValueNode(root1);

temp = c->data;

deleteValue(c->data);

root1 = root;

}

else if (initialblackheight2 > initialblackheight1) {

c = minValueNode(root2);

temp = c->data;

rbTree2.deleteValue(c->data);

root2 = rbTree2.root;

}

else {

c = minValueNode(root2);

temp = c->data;

rbTree2.deleteValue(c->data);

root2 = rbTree2.root;

if (initialblackheight1 != getBlackHeight(root2)) {

rbTree2.insertValue(c->data);

root2 = rbTree2.root;

c = maxValueNode(root1);

temp = c->data;

deleteValue(c->data);

root1 = root;

}

}

setColor(c, RED);

int finalblackheight1 = getBlackHeight(root1);

int finalblackheight2 = getBlackHeight(root2);

if (finalblackheight1 == finalblackheight2) {

c->left = root1;

root1->parent = c;

c->right = root2;

root2->parent = c;

setColor(c, BLACK);

c->data = temp;

root = c;

}

else if (finalblackheight2 > finalblackheight1) {

Node \*ptr = root2;

while (finalblackheight1 != getBlackHeight(ptr)) {

temp\_ptr = ptr;

ptr = ptr->left;

}

Node \*ptr\_parent;

if (ptr == NULL)

ptr\_parent = temp\_ptr;

else

ptr\_parent = ptr->parent;

c->left = root1;

if (root1 != NULL)

root1->parent = c;

c->right = ptr;

if (ptr != NULL)

ptr->parent = c;

ptr\_parent->left = c;

c->parent = ptr\_parent;

if (getColor(ptr\_parent) == RED) {

fixInsertRBTree(c);

}

else if (getColor(ptr) == RED) {

fixInsertRBTree(ptr);

}

c->data = temp;

root = root2;

}

else {

Node \*ptr = root1;

while (finalblackheight2 != getBlackHeight(ptr)) {

ptr = ptr->right;

}

Node \*ptr\_parent = ptr->parent;

c->right = root2;

root2->parent = c;

c->left = ptr;

ptr->parent = c;

ptr\_parent->right = c;

c->parent = ptr\_parent;

if (getColor(ptr\_parent) == RED) {

fixInsertRBTree(c);

}

else if (getColor(ptr) == RED) {

fixInsertRBTree(ptr);

}

c->data = temp;

root = root1;

}

return;

}

bool RBTree::isEmpty() {

if (root == NULL) {

return TRUE;

}

return FALSE;

}

void RBTree::printTree() { // list

if (root == NULL) {

return;

}

List<Node\*> \*rbqueue = new List<Node\*>();

rbqueue->Append(root);

while (rbqueue->NumInList() != 0) {

int size = rbqueue->NumInList();

for (int i = 0; i < size; i++) {

Node \*n = rbqueue->Front(); // remove front

if (n != NULL) {

if (n->left != NULL) {

rbqueue->Append(n->left);

}

if (n->right != NULL) {

rbqueue->Append(n->right);

}

printf("[%s at %.2f(%s)] ", n->data->getName(), n->data->getVirtualRunTime(), NodeNames[n->color]);

}

rbqueue->RemoveFront();

}

printf("\n");

}

}

Node\*

RBTree::getANode() {

if (minValueNode() != root) {

return minValueNode();

}

else {

return maxValueNode(root);

}

}

In Thread.h

class Thread {

private:

…

double getVirtualRunTime(); // get virtual run time

void UpdateVirtualRunTime(); // update the virtual run time before next schedule

void setWeight(double weight); // set weight for the thread

double getWeight(); // get thread weight

void UpdateInTree(); // update when in ready list

private:

…

double VirtualRunTime;

double Weight;

double Decay;

};

In Thread.cc

//----------------------------------------------------------------------

// Thread::getVirtualRunTime

// Get virtual time

//----------------------------------------------------------------------

double

Thread::getVirtualRunTime(){

return VirtualRunTime;

}

//----------------------------------------------------------------------

// Thread::UpdateVirtualRunTime

// Update virtual run time for thread switching

//----------------------------------------------------------------------

void

Thread::UpdateVirtualRunTime() {

ASSERT(kernel->getTotalWeight() != 0);

cout << "Virtual Runtime: From " << VirtualRunTime;

VirtualRunTime = VirtualRunTime + Weight / kernel->getTotalWeight() \* TimerTicks + Decay;

cout << " to " << VirtualRunTime << "(Decay: " << Decay << ", Total Weight: " << kernel->getTotalWeight() << ").\n";

}// update the virtual run time before next schedule

//----------------------------------------------------------------------

// Thread::UpdateInTree

// Update the virtual run time when total weight chenged but not run yet

//----------------------------------------------------------------------

void

Thread::UpdateInTree() {

if (kernel->getLastUpdateWeight() != 0) {

int original = VirtualRunTime - Weight / kernel->getLastUpdateWeight() \* TimerTicks - Decay;

VirtualRunTime = original + Weight / kernel->getTotalWeight() \* TimerTicks + Decay;

}

}

//----------------------------------------------------------------------

// Thread::setWeight

// Set thread weight

//----------------------------------------------------------------------

void

Thread::setWeight(double weight) {

Weight = weight;

}

//----------------------------------------------------------------------

// Thread::getWeight

// Get thread weight

//----------------------------------------------------------------------

double

Thread::getWeight() {

return Weight;

}

In Scheduler.h

class Scheduler {

public:

…

private:

RBTree \*readyList; // queue of threads that are ready to run,

// but not running

…

int LastSwitchTick;

void UpdateTree(); // update the whole tree

List<Thread\*> \*tempList; // temp list

int threadNum;

};

In Scheduler.cc

//----------------------------------------------------------------------

// Scheduler::ReadyToRun

// Mark a thread as ready, but not running.

// Put it on the ready list, for later scheduling onto the CPU.

//

// "thread" is the thread to be put on the ready list.

//----------------------------------------------------------------------

void

Scheduler::ReadyToRun (Thread \*thread) // who is next to run

{

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

DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

thread->setStatus(READY);

// update virtual run time

kernel->setTotalWeight(kernel->getTotalWeight() + thread->getWeight()); // update totalweight

thread->UpdateVirtualRunTime();

UpdateTree(); // update the whole tree.

threadNum++;

kernel->setCurrentTimeSlice(threadNum);

// arrange rdylist

readyList->insertValue(thread);

Print();

}

//----------------------------------------------------------------------

// Scheduler::FindNextToRun

// Return the next thread to be scheduled onto the CPU.

// If there are no ready threads, return NULL.

// Side effect:

// Thread is removed from the ready list.

//----------------------------------------------------------------------

Thread \*

Scheduler::FindNextToRun ()

{

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

if (readyList->isEmpty()) {

return NULL;

} else {

Thread \*next = readyList->minValueNode()->data;

readyList->deleteValue(next);

kernel->setTotalWeight(kernel->getTotalWeight() - next->getWeight()); // update weight

UpdateTree(); // update the whole tree.

cout << "NEXT: " << next->getName() << "\n";

// change time interrupt

cout << "Next Current Time Slice: " << kernel->getCurrentTimeSlice() <<" ("<< threadNum <<" threads in total)\n";

kernel->alarm->UpdateNextInterrupt(kernel->stats->totalTicks + kernel->getCurrentTimeSlice());

threadNum--;

kernel->setCurrentTimeSlice(threadNum);

return next;

}

}

//----------------------------------------------------------------------

// Scheduler::Print

// Print the scheduler state -- in other words, the contents of

// the ready list. For debugging.

//----------------------------------------------------------------------

void

Scheduler::Print()

{

cout << "Ready list contents:\n";

readyList->printTree();

}

//----------------------------------------------------------------------

// Scheduler::getLastTick

// get last context switch ticks

//----------------------------------------------------------------------

int

Scheduler::getLastTick() {

return LastSwitchTick;

}

//----------------------------------------------------------------------

// Scheduler::UpdateTree

// Wpdate the whole tree when the total weight changes

//----------------------------------------------------------------------

void

Scheduler::UpdateTree() {

Thread \*t;

while (!readyList->isEmpty()) { // clear tree

t = readyList->getANode()->data;

tempList->Append(t);

readyList->deleteValue(t);

}

RBTree \* old = readyList;

readyList = new RBTree();

while (!tempList->IsEmpty()) { // update all the nodes

t = tempList->RemoveFront();

t->UpdateInTree();

readyList->insertValue(t);

}

delete old;

}

Kernel.h

class Kernel {

public:

…

SortedList<IoEvent\*> \*ioEventQueue; // io event queue

double getTotalWeight(); // get total weight in ready list

void setTotalWeight(double newWeight); // set total weight in ready list

int getTotalFinishedIoThreadNum(); // get total finished io thread number

void setTotalFinishedIoThreadNum(int num); // set total finished io thread number

int getLastUpdateWeight(); // get last update weight to restore virtual time

int getCurrentTimeSlice();

void setCurrentTimeSlice(int threadNum);

private:

…

char \*consoleOut; // file to send console output to

double totalWeight;

int TotalFinishedIoThread;

int lastUpdateWeight;

…

};

Kernel.cc

double

Kernel::getTotalWeight() {

return totalWeight;

}

void

Kernel::setTotalWeight(double newWeight) {

lastUpdateWeight = totalWeight;

totalWeight = newWeight;

}

int

Kernel::getTotalFinishedIoThreadNum() {

return TotalFinishedIoThread;

}

void

Kernel::setTotalFinishedIoThreadNum(int num) {

TotalFinishedIoThread = num;

}

int

Kernel::getLastUpdateWeight() {

return lastUpdateWeight;

}

int

Kernel::getCurrentTimeSlice(){

return currentTimeSlice;

}

void

Kernel::setCurrentTimeSlice(int threadNum){

if(threadNum > 0){

currentTimeSlice = TimerTicks / threadNum;

}else{

currentTimeSlice = TimerTicks;

}

}

Alarm.h

void UpdateNextInterrupt(int nextTime);

Alarm.cc

void

Alarm::UpdateNextInterrupt(int nextTime){

// cancel last interrupt

kernel->interrupt->clearTimeInterrupt();

// set the updated interrupt

timer->SetInterrupt(nextTime);

}

Timer.h

#ifndef TIMER\_H

#define TIMER\_H

#include "copyright.h"

#include "utility.h"

#include "callback.h"

// The following class defines a hardware timer.

class Timer : public CallBackObj {

public:

Timer(bool doRandom, CallBackObj \*toCall);

// Initialize the timer, and callback to "toCall"

// every time slice.

virtual ~Timer() {}

void Disable() { disable = TRUE; }

// Turn timer device off, so it doesn't

// generate any more interrupts.

void SetInterrupt(int nextTime);

private:

bool randomize; // set if we need to use a random timeout delay

CallBackObj \*callPeriodically; // call this every TimerTicks time units

bool disable; // turn off the timer device after next

// interrupt.

void CallBack(); // called internally when the hardware

// timer generates an interrupt

void SetInterrupt(); // cause an interrupt to occur in the

// the future after a fixed or random

// delay

int LastTime;

};

#endif // TIMER\_H

Timer.cc

#include "copyright.h"

#include "timer.h"

#include "main.h"

#include "sysdep.h"

//----------------------------------------------------------------------

// Timer::Timer

// Initialize a hardware timer device. Save the place to call

// on each interrupt, and then arrange for the timer to start

// generating interrupts.

//

// "doRandom" -- if true, arrange for the interrupts to occur

// at random, instead of fixed, intervals.

// "toCall" is the interrupt handler to call when the timer expires.

//----------------------------------------------------------------------

Timer::Timer(bool doRandom, CallBackObj \*toCall)

{

randomize = doRandom;

callPeriodically = toCall;

disable = FALSE;

LastTime = 0;

SetInterrupt(LastTime + TimerTicks);

}

//----------------------------------------------------------------------

// Timer::CallBack

// Routine called when interrupt is generated by the hardware

// timer device. Schedule the next interrupt, and invoke the

// interrupt handler.

//----------------------------------------------------------------------

void

Timer::CallBack()

{

// invoke the Nachos interrupt handler for this device

callPeriodically->CallBack();

SetInterrupt(LastTime + kernel->getCurrentTimeSlice());// do last, to let software interrupt handler

// decide if it wants to disable future interrupts

}

//----------------------------------------------------------------------

// Timer::SetInterrupt

// Cause a timer interrupt to occur in the future, unless

// future interrupts have been disabled. The delay is either

// fixed or random.

//----------------------------------------------------------------------

void

Timer::SetInterrupt()

{

if (!disable) {

int delay = TimerTicks;

if (randomize) {

delay = 1 + (RandomNumber() % (TimerTicks \* 2));

}

// schedule the next timer device interrupt

kernel->interrupt->Schedule(this, delay, TimerInt);

}

}

void

Timer::SetInterrupt(int nextTime)

{

kernel->interrupt->Schedule(this, nextTime - kernel->stats->totalTicks, TimerInt);

LastTime = nextTime;

}

**4. Testing**

1. Method to run my tests.

Run:

./**nachos -K**

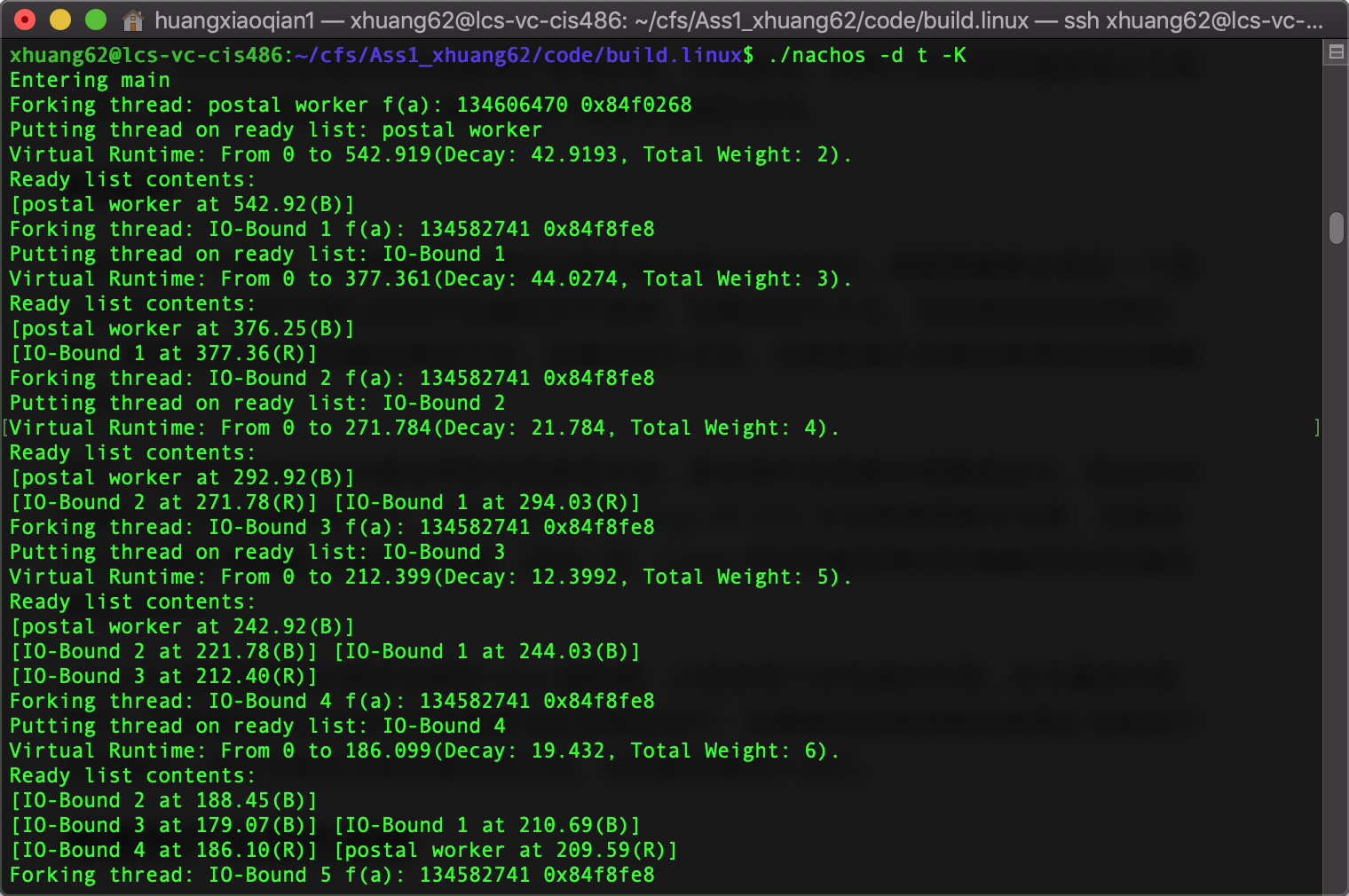
./**nachos –d t –K**  (used in my snapshots)

There are 5 pure IO threads, 1 MIX thread and 15 CPU threads in total.

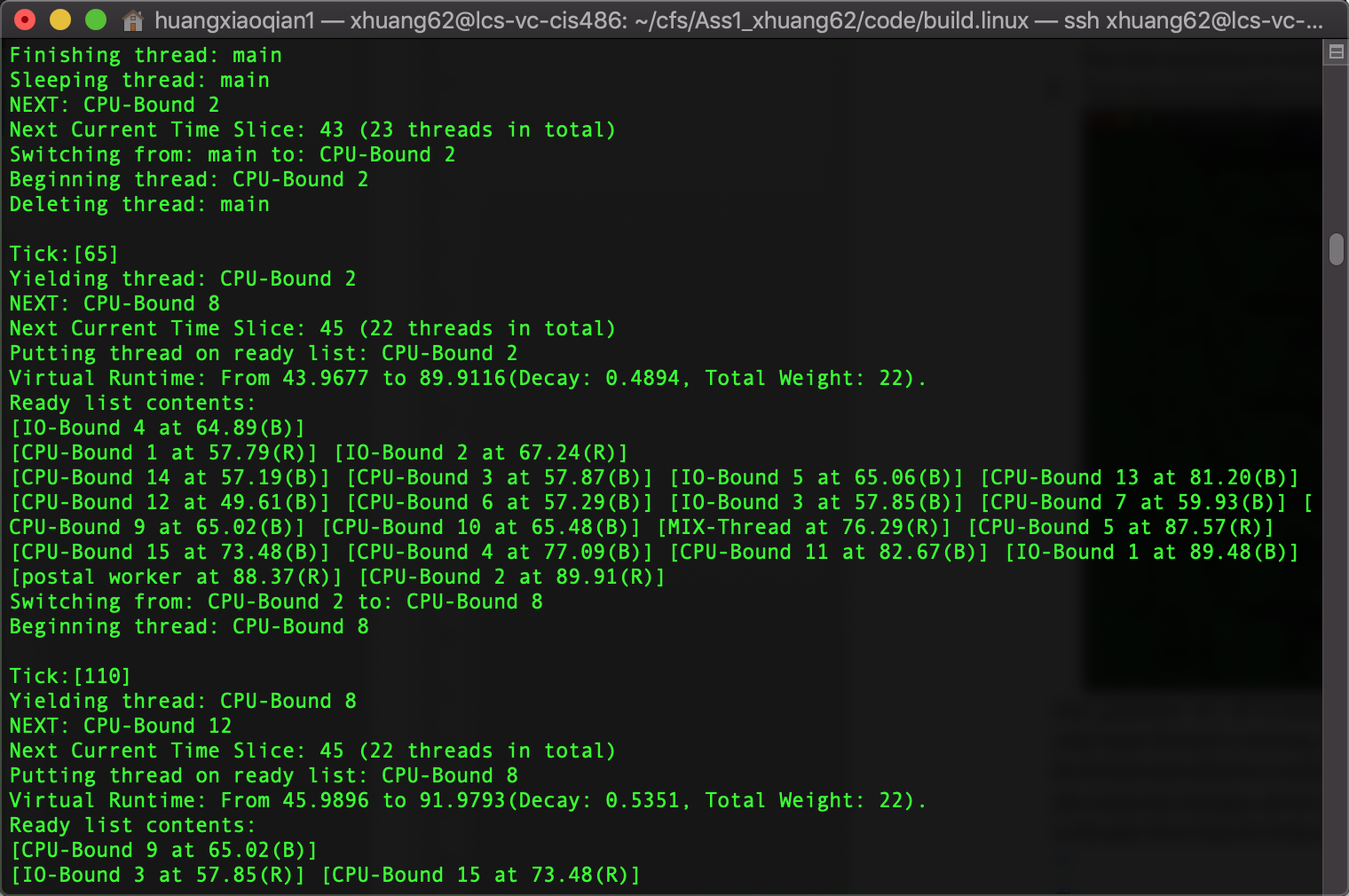
Then 6 IO events will be scheduled to raise interrupt and run. When all 6 events are finished, the system will halt.

The time quantum of scheduling is 1000 ticks.

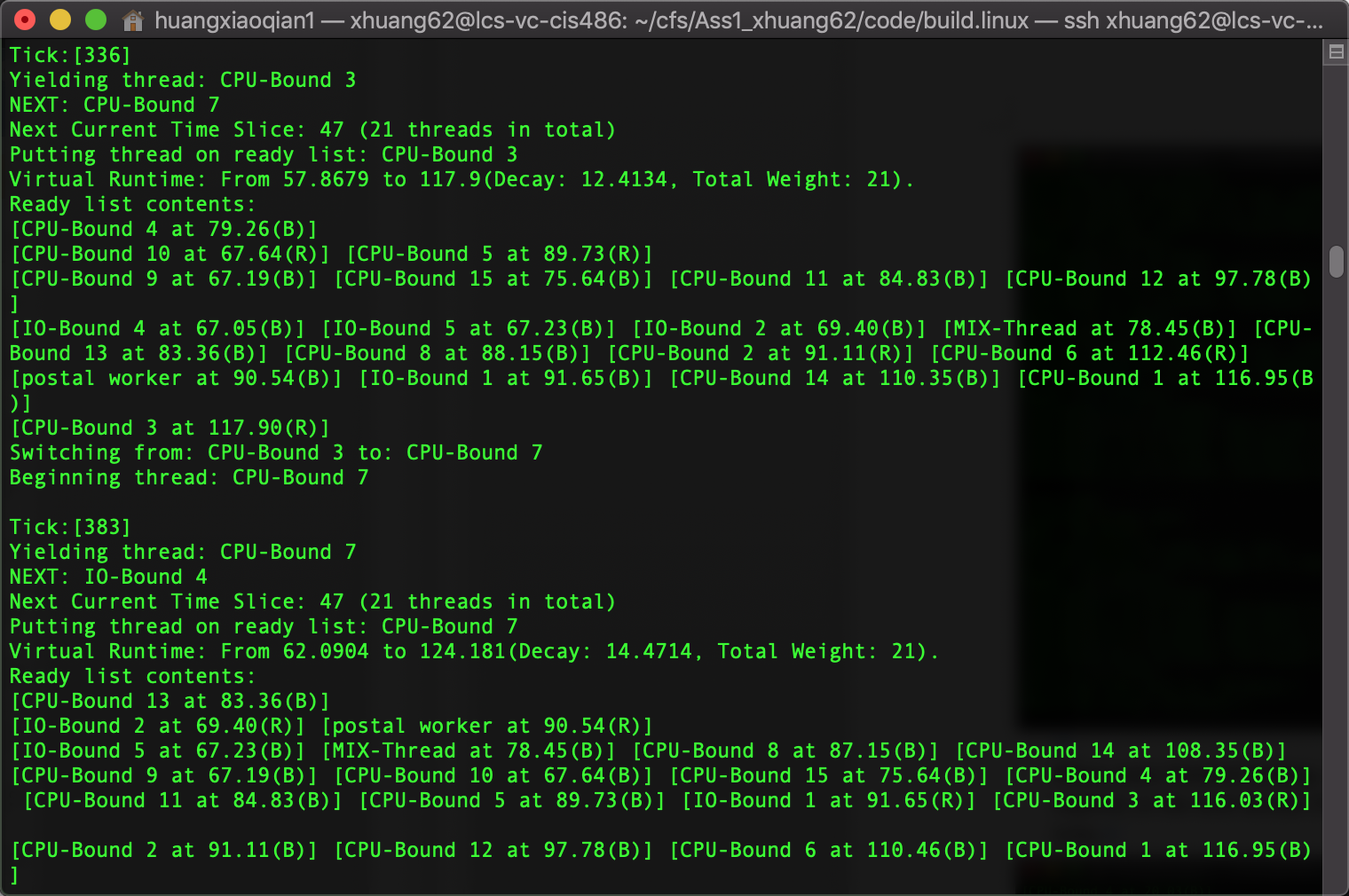
1. Test case and snapshots (the snapshots are taken in –d t debug mode)

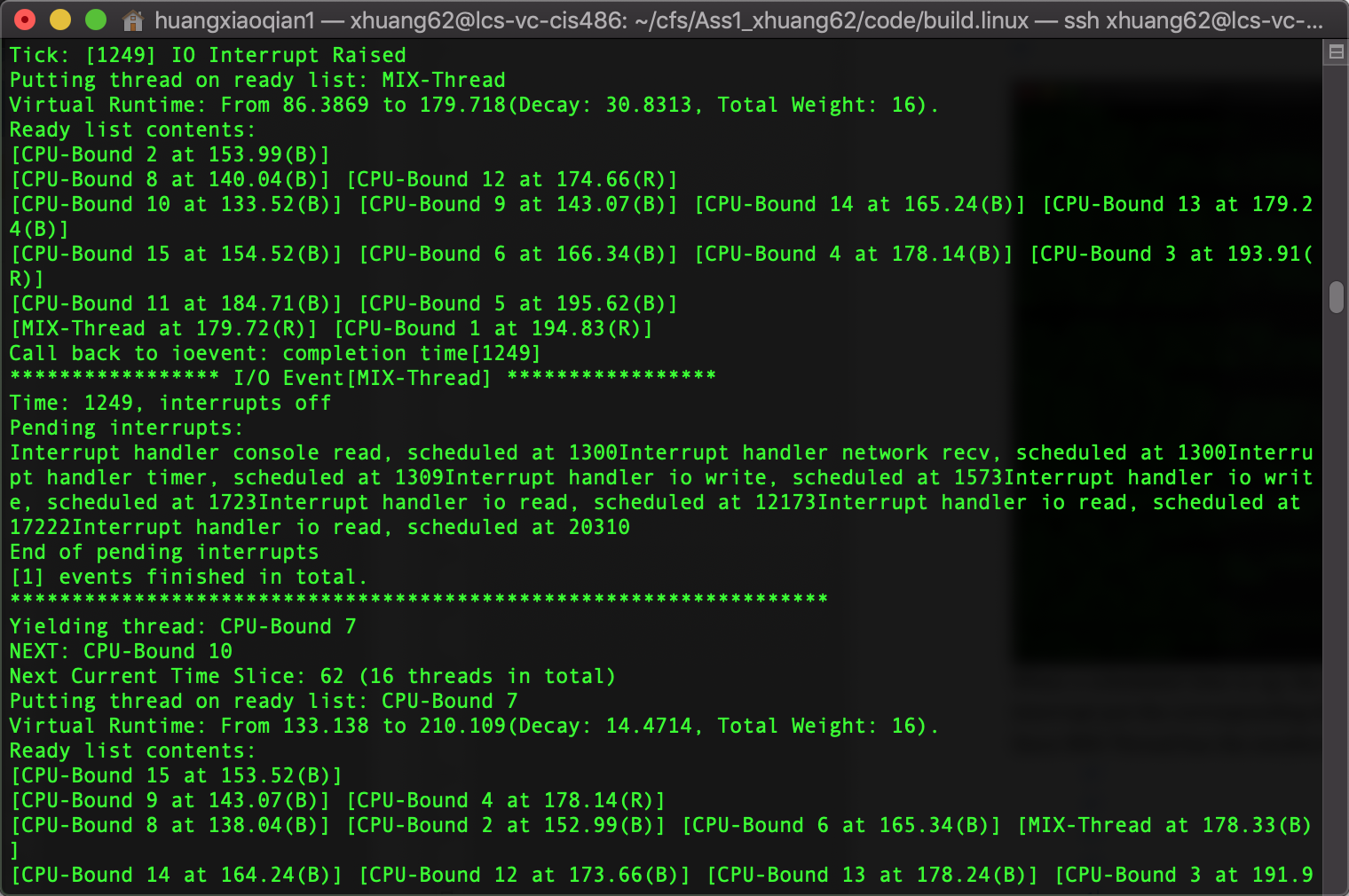


Use “./nachos –d t –K” to launch the program in debug mode with thread information. Since only main thread is running, the time slice now is 1000. So all the 21 threads that I set will be forked and added to ready list. Then main yield the CPU when it is finished at 65 ticks. Every time the ready list changes, all the virtual run time nodes in the red-black tree (the tree is printed in Breath-First-Search Order).



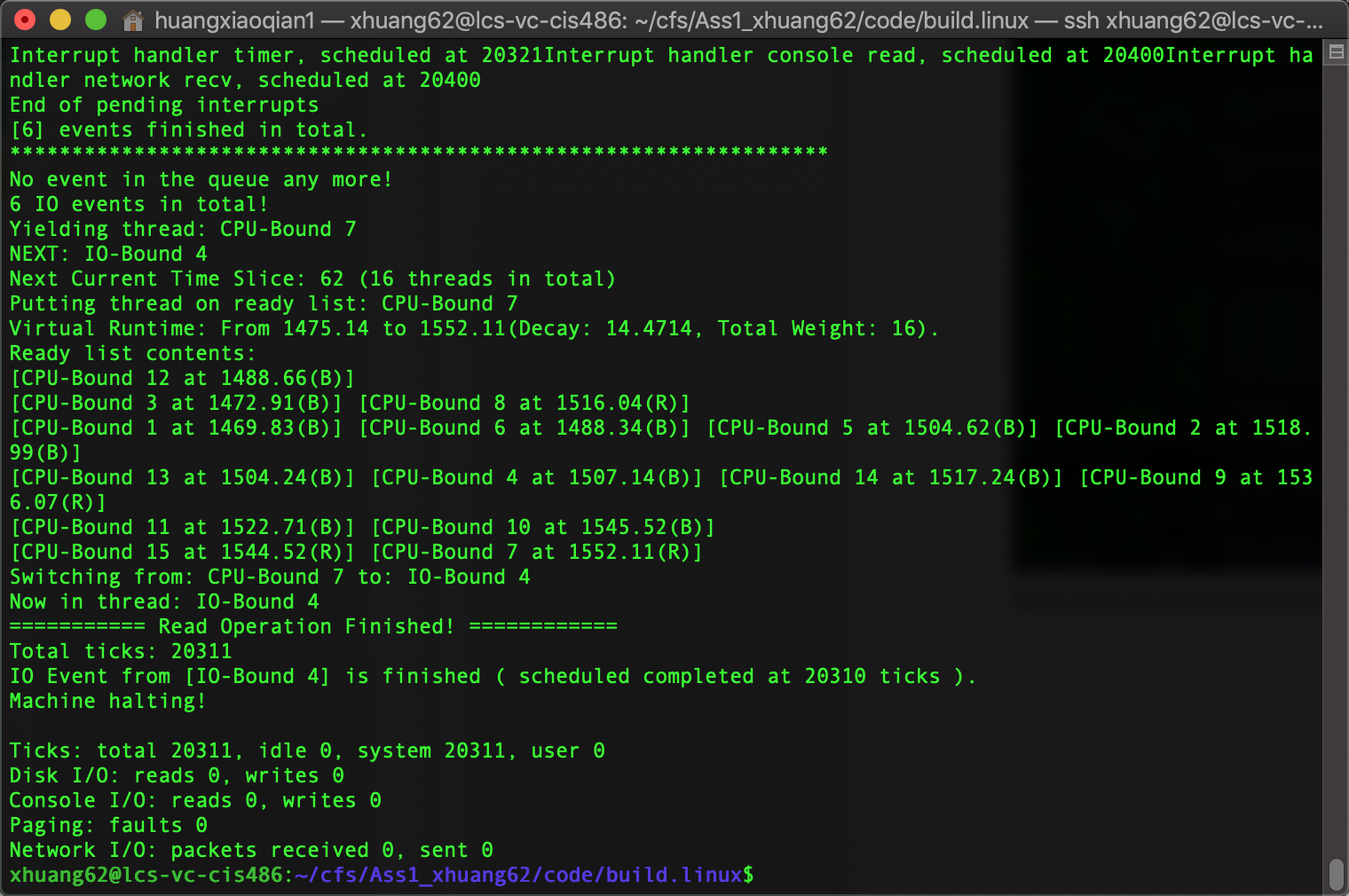
The main thread yield at 65 ticks, and now there are 22 threads in total which makes every thread can have 45 ticks to run. So the thread with smallest virtual time CPU-Bound 2 will start running at 65 immediately and end at 110 ticks.





When a scheduled time is up, the io interrupt will raised (1249 in this case). Then the this io interrupt put the corresponding thread(MIX-Thread) into ready list.

According to the information printed by pending interrupt queue, the next time interrupt, and remaining io interrupts are all scheduled as expected.



When all the io event finished (6 here), the process will halt.