OS2022 HW2 - CPU Scheduling

Student: 游家權(r10942152)

Part1 - Sleep system call Implementation

Motivation:
Implement a sleep system call that can be called by user programs. The user should be able to decide how long they want the thread to be halted, and the thread needs to resume execution after the period of idling.

Implementation:
Step1: Define a new system call and pass the exception down to the system
I need to define a system call to allow the user to invoke sleep routine via system API.
In userprog/syscall.h, I define SC_Sleep as a new system call and the routine that will be invoked when the system call happens.

```
/* 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
//TODO add a new system call for Sleep()
#define SC_Sleep        12
```

In userprog/syscall.h, define Sleep()

```
/* Yield the CPU to another runnable thread, whether in this address space
 * or not.
 */
void ThreadYield();

void PrintInt(int number);      //my System Call
// TODO Declare sleep system call
void Sleep(int number);
```

In test/start.s, I also need to define Sleep() for the user program.

```
// TODO add code for SC_Sleep
Sleep:
        addiu $2,$0,SC_Sleep
        syscall
        j       $31
        .end Sleep

        .globl  Sleep
        .ent    Sleep
```

In userporg/expection.cc, I need to add SC_Sleep() for exception handler and pass exception to WaitUntil() in alarm.h and use it to implement the sleep function

```
case SC_Exit:
        DEBUG(dbgAddr, "Program exit\n");
        val=kernel->machine->ReadRegister(4);
        cout << "return value:" << val << endl;
        kernel->currentThread->Finish();
        break;
// TODO SC_sleep write sleep function
case SC_Sleep:
        val=kernel->machine->ReadRegister(4);
        // Define in thread/alarm.h
        kernel->alarm->WaitUntil(val);
        return;
```

Step2: Implement WaitUntil()

In threads/alarm.h, I need to define SleepThreadManager to help me keep track of the state of sleeping threads, and wake them up when the sleeping time is up.

I declare count_idle to count how many ticks the system has gone through since the start of the program. I use this counter to check whether or not it's time for threads to wake up.

Note that the exception handler will call WaitUntil() to fulfill sleeping function

```
// 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; }

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

    //TODO, declare manager to store sleep thread information
    SleepThreadManager sleepThreadManager;
    int count_idle;             // count system has idled for how long

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

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

In thread/alarm.h, I implement WaitUntil() and use sleepThreadManager to help me tackle with sleeping threads.

```
Alarm::Alarm(bool doRandom)
{
    timer = new Timer(doRandom, this);
    // TODO, init SleepThreadManager
    sleepThreadManager = SleepThreadManager();
}

// TODO implement Alarm::WaitUntil(x)
void Alarm::WaitUntil(int x){
    // Disable Interrput temporarily. Defined in machine/interrupt.cc
    IntStatus level_tmp = kernel->interrupt->SetLevel(IntOff);

    // Get current thread
    Thread* thread_cur = kernel->currentThread;

    // Sleep current thread
    sleepThreadManager.PutToSleep(thread_cur, x);

    // Enable Interrput
    kernel->interrupt->SetLevel(level_tmp);
}
```

In thread/alarm.h, the CallBack() will be invoked when a software tick sends an interrupt signal to the system. I decide to check the sleeping queue for every tick and increment the counter_idle to record the time elapse of the system .

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    // TODO, increment counter and check if threads need to wake up
    sleepThreadManager.CheckAndWakeUp();
    // TODO, if there's thread still sleeping, don't quit
    if (status == IdleMode && sleepThreadManager.SleepingThreadList.size() == 0)
 {      // is it time to quit?
        // TODO OS will quit, only if program has idled from a while
        count_idle++;
        if (!interrupt->AnyFutureInterrupts() and count_idle > 100) {
            timer->Disable();   // turn off the timer
        }
    } else {                    // there's someone to preempt
        count_idle = 0;         // Reset Idle Counter
        interrupt->YieldOnReturn();
    }
}
```

Step3: Define a data structure to manage sleeping threads

In threads/alarm.h, I need to define two classes to help me store sleeping thread information. SleepThread stores the status of the thread and its wake-up-time. The SleepThreadManager will manage a list of sleeping threads and check their status periodically. If count_int is bigger than thread's wakeUpTime, the manager will wake them up.

```
// TODO, define a data structure to store sleeping thread
class SleepThread{
    public:
        // Methods
        SleepThread(Thread* thread, int x, int count_int);
        Thread* thread;     // store thread that is sleeping

        // Variables
        int wakeUpTime;     // when to wake up the thread
};

// TODO, define a manager to cope with sleep threads
class SleepThreadManager{
    public:
        // Methods
        SleepThreadManager();
        void PutToSleep(Thread* thread, int x); // put thread into sleep list
        void CheckAndWakeUp(); // check sleep list and wake up threads

        // Variables
        std::list<SleepThread> SleepingThreadList; // store threads that are sle
eping
        int count_int; // interrupt counter, increment in CallBack()
};
```

In threads/alarm.cc, I need to implement all the functions defined in alarm.h.
In the constructor of SleepThread, I store the thread and set its wakeUpTime to current_tick plus user-specified ticks.
In PutToSleep(), I put the thread into halt and push the thread into a sleeping queue.
In CheckAndWakeUp(), I go through the whole sleeping queue and linearly search whether there's a thread that needs to be woken. Also, I increment the counter to make the ticks moves on.

```
//TODO, implement constructor of SleepThread
SleepThread::SleepThread(Thread* thread, int x, int count_int){
    this->thread = thread;
    this->wakeUpTime = count_int + x;
}

// TODO, put to threads to sleep list
void SleepThreadManager::PutToSleep(Thread* thread, int x){
    // Put this thread into sleep list
    SleepingThreadList.push_back(SleepThread(thread, x, count_int));
    // defined in threads/threads.cc
    thread->Sleep(false);
    cout << "Put thread into sleep when count_int = " << count_int << endl;
}

// TODO, Wake up threads if their wakeUpTime is up
void SleepThreadManager::CheckAndWakeUp(){
    // Increment interrupt counter
    count_int++;
    // Search thread that should be waken
    for (std::list<SleepThread>::iterator it = SleepingThreadList.begin(); it !=
 SleepingThreadList.end();){
        // Wake up threads
        if (count_int >= it->wakeUpTime){
            kernel->scheduler->ReadyToRun(it->thread);
            cout << "Woke up thread at count_int = " << count_int << endl;
            // pop out threads that are already waken
            it = SleepingThreadList.erase(it);
        }
        else {it++;}
    }
}

// TODO, constructor of SleepThreadManager
SleepThreadManager::SleepThreadManager(){
    count_int = 0;
}
```

Sleep System Call Experiment Result:

I define two test files and execute them together to help me test the sleep function. Both of the test programs will sleep for every 1000 ticks, but the second program will delay for 500 ticks; therefore, these two programs will execute in an interleaved fashion. I expect to see "11111" and "22222" showing right next to each other. This can prove the sleep system call works and the programs are woken up correctly.

In test/test_sleep1.c

```
#include "syscall.h"
main()
        {
            int i;
            for (i = 0; i < 3; i++){
                PrintInt(11111);
                Sleep(1000);
            }
        }
```

In test/test_sleep2.c

```
#include "syscall.h"
main()
        {
            int i;
            Sleep(500);
            for (i = 0; i < 3; i++){
                PrintInt(22222);
                Sleep(1000);
            }
        }
```

My Execute Result:

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code$ userprog/
nachos -e test/test_sleep1 -e test/test_sleep2
Total threads number is 2
Thread test/test_sleep1 is executing.
Thread test/test_sleep2 is executing.
Print integer:11111
Woke up thread at count_int = 500
Put thread into sleep when count_int = 500
Print integer:22222
Woke up thread at count_int = 1000
Put thread into sleep when count_int = 1000
Print integer:11111
Woke up thread at count_int = 1500
Put thread into sleep when count_int = 1500
Print integer:22222
Woke up thread at count_int = 2000
Put thread into sleep when count_int = 2000
Print integer:11111
Woke up thread at count_int = 2500
Put thread into sleep when count_int = 2500
Print integer:22222
Woke up thread at count_int = 3000
Put thread into sleep when count_int = 3000
return value:0
Woke up thread at count_int = 3500
Put thread into sleep when count_int = 3500
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 360000, idle 359622, system 190, user 188
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
```

Two programs sleep and resume as I expected.

Part2 - CPU Scheduling:

Motivation:

I want to implement **First-Come-First-Service(FCFS)** and **Shortest-Job-First(SJF)** for CPU scheduling in this part. By default, the kernel uses round robin scheduling. To achieve this, I need to implement scheduling algorithms in scheduler.cc by sorting the ready queue according to “start time” or “CPU burst time” of the programs. I also need to modify the definition of thread so that it can store its remaining CPU burst time. Lastly, I need to create my own test case in SelfTest() in thread.cc to validate the correctness of scheduling algorithms.

Implementation:

Step1: Allow user use arguments to decide which scheduler to use

In thread/main.cc, I define three arguments: -fcfs, -sjf, and -rr. Users can assign different algorithms by passing in arguments accordingly.

```
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;
        }

        // TODO, add argument that allow user to switch schedueling method
        else if (strcmp(argv[i], "-fcfs") == 0) {
            cout << "Using FCFS as CPU-schedueling method" << endl;
            scheduler_type = FCFS;
        }
        else if (strcmp(argv[i], "-sjf") == 0) {
            cout << "Using SJF as CPU-schedueling method" << endl;
            scheduler_type = SJF;
        }
        else if (strcmp(argv[i], "-rr") == 0) {
            cout << "Using RR as CPU-schedueling method" << endl;
            scheduler_type = RR;
        }


    }
    debug = new Debug(debugArg);
```

Step 2: Write test cases

I need to Implement SelfTest() in thread.cc to test my scheduling algorithm. In both of the test cases, I create three threads and give them different start time and burst time.

In thread/thread.cc

```
// TODO, write my own test case here to valid correctness of CPU-schedueler
void
Thread::SelfTest()
{
    DEBUG(dbgThread, "Entering Thread::SelfTest");

    char* thread_name[3]      = {"A", "B", "C"};
    // Test case 1
    int start_time[3] = {3, 2 ,1}; // For FCFS, should execute in "CBA" order
    int burst_time[3] = {5, 2, 3}; // For SJF, should execute in "BCA" order

    // Test case 2
    // int start_time[3] = {1, 2 ,3}; // For FCFS, should execute in "ABC" order
    // int burst_time[3] = {4, 1, 9}; // For SJF, should execute in "BAC" order

    // Init all thread
    Thread* t;
    int i = 0;
    for (i = 0; i < 3; i++){
        t = new Thread(thread_name[i]);
        t->set_attribute(start_time[i], burst_time[i]);
        t->Fork((VoidFunctionPtr) MyThread, (void *)NULL);
    }
}
```

In thread/thread.cc, I implement a function that can deduct thread's burst time for every tick to simulate what actually happens to the programs during execution.

```
//TODO, need to write another threadtest!!!
static void
MyThread()
{
    Thread* t = kernel->currentThread;
    while(t->burst_time > 0){
        cout << "*** thread " << t->name << " remain CPU burst time = " << t->burst_time << "\n";
        t->burst_time--;
        // I'm not sure about this
        kernel->interrupt->OneTick();
    }
}
```

Step3: Modify Thread Class

I need to modify the thread class to allow it to store start time and burst time. Also, it needs to initialize them correctly via set_attribute().

In thread/thread.h

```
    // TODO add burst_time and start_time to allow schedueler implement SJF and
FCFS
    void set_attribute(int start_time, int burst_time);
    int start_time; // For FCFS
    int burst_time; // For SJF
    char* name;
```

In thread/thread.cc, I implement a setter function here.

```
//TODO, set start_time and burst time in thread class
void
Thread::set_attribute(int st, int bt){
    start_time = st;
    burst_time = bt;
}
```

Step 4: Modify Scheduler

I Implement CPU scheduling algorithm in scheduler.cc and scheduler.h. I use SortedList defined in lib/list.cc to help me sort threads according to their attributes.

In thread/scheduler.h, I add FCFS and SJF as a new scheduler type and define FCFS_Compare and SJF_Compare as comparison functions for SortList.

```
//TODO, add FCFS
enum SchedulerType {
        FCFS,    // First come first serve
        RR,      // Round Robin
        SJF
        //Priority
};

// TODO, add helper function to define my schduler
int FCFS_Compare(Thread* a, Thread* b); // sort readylist with start time
int SJF_Compare (Thread* a, Thread* b); // sort readylist with burst time
```

In thread/scheduler.cc, I implement compare functions which will compare threads with their start time or burst time.

```
// TODO, implement FCFS compare function
int
FCFS_Compare(Thread* a, Thread* b){
    if (a->start_time > b->start_time){return 1;}
    else if (a->start_time < b->start_time){return -1;}
    else{return 0;}
}

// TODO, implement SJF compare function
int
SJF_Compare(Thread* a, Thread* b){
    if (a->burst_time > b->burst_time){return 1;}
    else if (a->burst_time < b->burst_time){return -1;}
    else{return 0;}
}
```

In thread/scheduler.cc, I allow the system to switch between scheduler and declare different implementation of ready queue.

```
// TODO, adjust comparision method in ready queue to implement different
// scheduling methods
Scheduler::Scheduler( SchedulerType scheduler_type )
{
        switch (scheduler_type){
            case FCFS:
                readyList = new SortedList<Thread*>(FCFS_Compare);
                break;
            case SJF:
                readyList = new SortedList<Thread*>(SJF_Compare);
                break;
            case RR:
                readyList = new List<Thread*>;
                break;
        }
        toBeDestroyed = NULL;
}
```

Step5: Pass Down Scheduler Type Argument

Scheduler type argument is the user-assigned argument in the command line. I need pass it down to the kernel to make the system change algorithm accordingly.

In thread/kernel.cc, pass scheduler_type to Scheduler()

```
// TODO, pass in argument for scheduler type
void
ThreadedKernel::Initialize(SchedulerType scheduler_type)
{
    stats = new Statistics();           // collect statistics
    interrupt = new Interrupt;          // start up interrupt handling
    scheduler = new Scheduler( 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();
}
```

In thread/main.cc, I declare a global variable to store scheduler type.

```
// global variables
KernelType *kernel;
Debug *debug;
//TODO Add a schedueler type global variable here
SchedulerType scheduler_type;
```

In thread/main.cc, pass the scheduler type down.

```
kernel = new KernelType(argc, argv);

// TODO, Pass in schedueler type
kernel->Initialize(scheduler_type);

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

CPU Scheduling Experiment Result:

In thread/thread.cc,. I define three threads with different start time and burst time.
In test case 1, I expect FCFS will execute the thread in “CBA” order and SJF will execute in “BCA” order, while in test case 2, I expect FCFS will execute in “ABC” order and SJF will execute in “BAC” order.

```
// TODO, write my own test case here to valid correctness of CPU-schedueler
void
Thread::SelfTest()
{
    DEBUG(dbgThread, "Entering Thread::SelfTest");

    char* thread_name[3]      = {"A", "B", "C"};
    // Test case 1
    int start_time[3] = {3, 2 ,1}; // For FCFS, should execute in "CBA" order
    int burst_time[3] = {5, 2, 3}; // For SJF, should execute in "BCA" order

    // Test case 2
    // int start_time[3] = {1, 2 ,3}; // For FCFS, should execute in "ABC" order
    // int burst_time[3] = {4, 1, 9}; // For SJF, should execute in "BAC" order

    // Init all thread
    Thread* t;
    int i = 0;
    for (i = 0; i < 3; i++){
        t = new Thread(thread_name[i]);
        t->set_attribute(start_time[i], burst_time[i]);
        t->Fork((VoidFunctionPtr) MyThread, (void *)NULL);
    }
}
```

Test case 1 Result:

FCFS: execute in “CBA” order as expected.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -fcfs
Using FCFS as CPU-scheduling method
*** thread C remain CPU burst time = 3
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
*** thread B remain CPU burst time = 2
*** thread B remain CPU burst time = 1
*** thread A remain CPU burst time = 5
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread A remain CPU burst time = 2
*** thread A remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

SJF: execute in “BCA” order as expected.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -sjf
Using SJF as CPU-scheduling method
*** thread B remain CPU burst time = 2
*** thread B remain CPU burst time = 1
*** thread C remain CPU burst time = 3
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
*** thread A remain CPU burst time = 5
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread A remain CPU burst time = 2
*** thread A remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

RR

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -rr
Using RR as CPU-scheduling method
*** thread A remain CPU burst time = 5
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread B remain CPU burst time = 2
*** thread B remain CPU burst time = 1
*** thread C remain CPU burst time = 3
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
*** thread A remain CPU burst time = 2
*** thread A remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Test Case 2 Result:

FCFS: execute in “ABC” order as expected.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -fcfs
Using FCFS as CPU-schedueling method
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread A remain CPU burst time = 2
*** thread A remain CPU burst time = 1
*** thread B remain CPU burst time = 1
*** thread C remain CPU burst time = 9
*** thread C remain CPU burst time = 8
*** thread C remain CPU burst time = 7
*** thread C remain CPU burst time = 6
*** thread C remain CPU burst time = 5
*** thread C remain CPU burst time = 4
*** thread C remain CPU burst time = 3
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

SJF: execute in “BAC” order as expected.

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -sjf
Using SJF as CPU-schedueling method
*** thread B remain CPU burst time = 1
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread A remain CPU burst time = 2
*** thread A remain CPU burst time = 1
*** thread C remain CPU burst time = 9
*** thread C remain CPU burst time = 8
*** thread C remain CPU burst time = 7
*** thread C remain CPU burst time = 6
*** thread C remain CPU burst time = 5
*** thread C remain CPU burst time = 4
*** thread C remain CPU burst time = 3
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

RR

```
kenyu@kenyu-VirtualBox:~/OS2022/hw1_thread_management/nachos-4.0/code/threads$ ./nachos -rr
Using RR as CPU-schedueling method
*** thread A remain CPU burst time = 4
*** thread A remain CPU burst time = 3
*** thread A remain CPU burst time = 2
*** thread B remain CPU burst time = 1
*** thread C remain CPU burst time = 9
*** thread C remain CPU burst time = 8
*** thread C remain CPU burst time = 7
*** thread C remain CPU burst time = 6
*** thread C remain CPU burst time = 5
*** thread C remain CPU burst time = 4
*** thread C remain CPU burst time = 3
*** thread A remain CPU burst time = 1
*** thread C remain CPU burst time = 2
*** thread C remain CPU burst time = 1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```