r06921048_project2_report

電機碩二　李友岐

Part 1: System Call - Sleep()

1. Motivation and the problem analysis :

Based on the principle of cpu scheduling, at one time, there is a thread executing while the other threads don't. For the threads not executing, we need to put them into waiting queue. As a result, we should write a function named Sleep(), which adds thread to the waiting queue and context-switch with the thread which should wake up.

2. What's your plan to deal with the problem (high-level) :

To add a thread into waiting queue, we can use data structure like list. Besides, we should know when to wake up this thread. Thus, we define a struct containing two members, which are the pointer to this thread and the time to wake it up, respectively. To wake up the threads, we should check the waiting queue to see whether there is a thread sleeping enough time, and then do context-switch. As a result, we define a class to be the place where threads sleep. This class has two data members, which are the waiting queue and the current time, respectively. The waiting queue collects the struct we mention above.

3. You can including some important code segments and comments :

threads/alarm.h:

```
struct Wait{
     int time;
     Thread* thread;

};


class WaitRoom {
    public:
        WaitRoom(){ curInterrupt=0;};
        void addToRoom( int x, Thread *t);
        bool wakeUp();
        bool isEmpty(){ return waitList.size()==0;}
    private:

        int curInterrupt;
        std::list<Wait> waitList;
};
```

Wait and WaitRoom are the struct and class we describe in last section. WaitRoom has three member functions and two private data members. The waiting queue is named as waitList and implemented by std::list.

threads/alarm.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;
        delete waitRoom;
    }

    void WaitUntil(int x);      // suspend execution until time > now + x
  private:
    Timer *timer;               // the hardware timer device
    WaitRoom *waitRoom;
    void CallBack();            // called when the hardware
                // timer generates an interrupt
};
```

In class Alarm, we add a data member called waitRoom, which is a object of our new defined class.

threads/alarm.cc:

```
void WaitRoom::addToRoom(int x, Thread* t) {
    struct Wait waitingThread={  curInterrupt + x , t };
    waitList.push_back( waitingThread);
    t->Sleep(false);
}
```

```
bool WaitRoom::wakeUp() {
    curInterrupt++;
    std::list<Wait>::iterator it;

    for( it = waitList.begin(); it != waitList.end(); it++ ) {
        if(curInterrupt >= it->time) {

            kernel->scheduler->ReadyToRun(it->thread);
            cout << it->thread->getName()<<" is awake" << endl;
            it = waitList.erase(it);
            return true;
        }

    }
    return false;
}
```

We define addToRoom() to put the thread into waiting queue and make them sleep. In addition, we define wakeUp() to check whether there is a thread in waiting queue ready to run. If finding one, then we remove this thread from waiting queue and return true. If we can't find one after traversing the list, then return false.

threads/alarm.cc:

```
void Alarm::WaitUntil(int x) {
    IntStatus iniLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;
    cout <<t->getName()<< " go to sleep" << endl;


    waitRoom->addToRoom(x, t);
    kernel->interrupt->SetLevel(iniLevel);
}
```

The WaitUntil() will be called when a thread going to sleep. we do it by calling addToRoom().

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

    bool check = !waitRoom->wakeUp() & waitRoom->isEmpty();

    if (status == IdleMode and check ) {// is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
        }
    }
    else {                      // there's someone to preempt

           if(kernel->scheduler->getSchedulerType() == RR) {
                 cout << "============= interrupt============ ===" << endl;
                 interrupt->YieldOnReturn();
           }
    }
}
```

We call the CallBack() to check which thread should wake up, which is done by calling wakeUp().

4. Experiment result and some discussion:

test/test.c:

```
#include "syscall.h"

main() {

      int i;
      for (i=0;i<5;i++){
          Sleep (3000000);
          PrintInt(25);

      }

}
```

3 million (ms) = 3 thousand (s)

sample command (move to /nachos-4.0/code/ first, the command part is after $):

```
/nachos-4.0/code$ ./userprog/nachos -e test/test
```

./userprog/nachos -e test/test

result:

```
Total threads number is 1
Thread test/test is executing.

Sleep time: 3000.0000 s
test/test go to sleep
test/test is awake
Print integer:25

Sleep time: 3000.0000 s
test/test go to sleep
test/test is awake
Print integer:25

Sleep time: 3000.0000 s
test/test go to sleep
test/test is awake
Print integer:25

Sleep time: 3000.0000 s
test/test go to sleep
test/test is awake
Print integer:25

Sleep time: 3000.0000 s
test/test go to sleep
test/test is awake
Print integer:25
return value:0
```

It has five iterations totally, and printing 25 at the end of each iteration. Moreover, the thread sleep in every iteration and the sleep time is 3000s. These phenomenon are same as what we define in test/test.c. The thread would go to sleep due to calling WaitUntil() and then awake after some time due to calling wakeUp().

test/sleep.c:

```c
#include "syscall.h"

main() {
    int i;
    for(i = 1; i < 4; i++) {
        Sleep(i*1000);
        PrintInt(i);
    }
    return 0;
}
```

1000 ms = 1s

sample command (move to /nachos-4.0/code/ first, the command part is after $)

```
/nachos-4.0/code$ ./userprog/nachos -e test/sleep
```

./userprog/nachos -e test/sleep

result:

```
Total threads number is 1
Thread test/sleep is executing.

Sleep time: 1.0000 s
test/sleep go to sleep
test/sleep is awake
Print integer:1

Sleep time: 2.0000 s
test/sleep go to sleep
test/sleep is awake
Print integer:2

Sleep time: 3.0000 s
test/sleep go to sleep
test/sleep is awake
Print integer:3
return value:0
```

It has three iterations totally, and printing i at the end of ith iteration. Moreover, the thread sleep in every iteration and the sleep time is i*1.0000 s. These phenomenon are same as what we define in test/sleep.c. The thread would go to sleep due to calling WaitUntil() and then awake after some time due to calling wakeUp(). In summary, our program is correct.

Part 2: CPU Scheduling

1. Motivation and the problem analysis :

We want to perform different cpu scheduling algorithm. As a result, we implement four scheduling algorithm, including Round Robbing (RR), Shortest Job First (SJF), Priority, First Come First Serve (FCFS)。

2. What's your plan to deal with the problem (high-level) :

We should add four scheduling cases into scheduler, which includes RR, SJF, FCFS and Priority. For each case, we sort the thread list by corresponding algorithm. The order of FCFS and RR is same as the order of the threads arrive. For SJF, the threads are sorted by their cpu burst time. For Priority, all threads have a given priority number, and we sort the list by it. In our implementation, only RR is preemptive, while the other three are non-preemptive. We don't need to do anything else after sorting the thread list for three non-preemptive cases. For RR, we should handle the situation that the interrupt occurs during the execution of a thread. To deal with it, we need to update the remaining cpu burst time of this thread and put it into waiting queue.

3. You can including some important code segments and comments :

threads/main.cc:

```
if(strcmp(argv[1], "RR") == 0) {
    Stype = RR;
}
else if (strcmp(argv[1], "SJF") == 0) {
    Stype = SJF;
}
else if (strcmp(argv[1], "PRIORITY") == 0) {
    Stype = Priority;
}
else if (strcmp(argv[1], "FCFS") == 0) {
    Stype = FCFS;
}
```

Add four cases in main(), we can specify which scheduling algorithm we want in command line by doing so.

threads/scheduler.cc:

```
int SJFCmp(Thread *a, Thread *b) {
    if(a->getBurstTime() == b->getBurstTime()){
        return 0;
    }
    else if ( a->getBurstTime() > b->getBurstTime() ){
        return 1;
    }
    else {
        return -1;
    }
}
int PriorityCmp(Thread *a, Thread *b) {
    if(a->getPriority() == b->getPriority()){
        return 0;
    }
    else if ( a->getPriority() > b->getPriority() ){
        return 1;
    }
    else {
        return -1;
    }
}
```

```
int FCFSCmp(Thread *a, Thread *b) {
    return 1;
}


int RRCmp(Thread *a, Thread *b) {
   return 1;
}
```

We define the sorting algorithm for four different scheduling cases. SJF is sorted by cpu burst time and Priority is sorted by priority number. For FCFS and RR, the order is same as the order of the threads arrive, which is the initial list order.

threads/scheduler.cc:

```
Scheduler::Scheduler(SchedulerType type)
{
    schedulerType = type;
    switch(schedulerType) {
            case RR:
                readyList = new SortedList<Thread *>(RRCmp);
                break;
            case SJF:
                readyList = new SortedList<Thread *>(SJFCmp);
                break;
            case Priority:
                readyList = new SortedList<Thread *>(PriorityCmp);
                break;
            case FCFS:
                readyList = new SortedList<Thread *>(FCFSCmp);
    }
    toBeDestroyed = NULL;
}
```

We add four cpu scheduling types into Scheduler. For each type, the thread list is sorted by the corresponding algorithm.

threads/thread.cc:

```
void
threadBody() {
    Thread *thread = kernel->currentThread;
    int count=0;

    while (thread->getBurstTime() > 0) {
        int time=thread->getBurstTime();
        thread->setBurstTime(time-1);
        count++;
        kernel->interrupt->OneTick();
        printf(" %s(%d)", thread->getName(),count);
        printf("  ,%s time remain:%d\n", thread->getName(), thread->getBurstTime());


    }
    cout<<thread->getName()<<" is finished,  ";
    printf("%s burst time:%d,  ", thread->getName(), count);
    printf("%s priority:%d\n", thread->getName(), thread->getPriority());

}
```

We obtain the current thread by kernel and then execute it by a while loop. For each iteration, we decrease the cpu burst time by 1. When the burst time becomes zero, this thread is finished.

4. Experiment result and some discussion:

We design two test cases to prove the correctness of our program. To execute it, first move to /code. The sample command for each cpu scheduling algorithm is as follows.

RR (move to /nachos-4.0/code/ first, the command part is after $):

```
/nachos-4.0/code$ ./threads/nachos RR
```

./threads/nachos RR

FCFS (move to /nachos-4.0/code/ first, the command part is after $):

```
/nachos-4.0/code$ ./threads/nachos FCFS
```

./threads/nachos FCFS

SJF (move to /nachos-4.0/code/ first, the command part is after $):

```
/nachos-4.0/code$ ./threads/nachos SJF
```
./threads/nachos SJF

Priority (move to /nachos-4.0/code/ first, the command part is after $):

```
/nachos-4.0/code$ ./threads/nachos PRIORITY
```
./threads/nachos PRIORITY

test case 1:

```
#define threadNum 5
void
Thread::SchedulingTest()
{

    char *threadName[threadNum] =   {"P1", "P2", "P3","P4","P5"};
    int threadPriority[threadNum] = {  4 ,   5 ,   1 ,  3 ,  2 };
    int threadBurst[threadNum] =    {  2 ,   5 ,   3 ,  4 ,  2 };
```

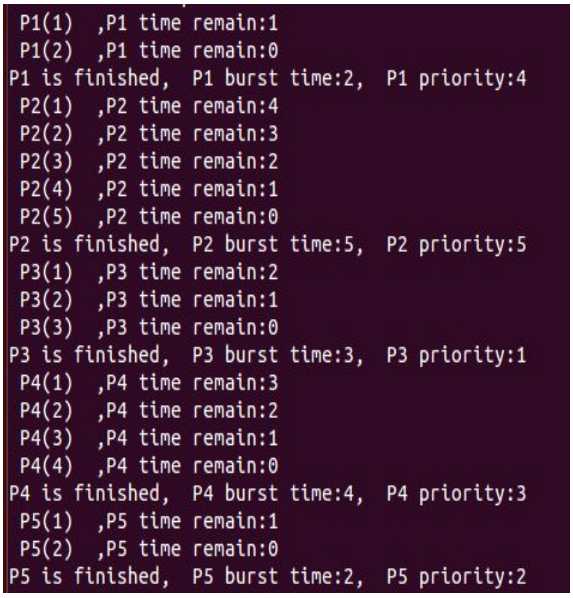
The order of FCFS and RR (sorted by arrival order): P1, P2, P3, P4, P5
The order of SJF (sorted by burst time): P1, P5, P3, P4, P2
The order of Priority (sorted by priority number): P3, P5, P4, P1, P2

RR:

```
============ interrupt============ ===
 P2(1)  ,P2 time remain:4
 P2(2)  ,P2 time remain:3
 P2(3)  ,P2 time remain:2
 P2(4)  ,P2 time remain:1
 P2(5)  ,P2 time remain:0
P2 is finished,  P2 burst time:5,  P2 priority:5
 P3(1)  ,P3 time remain:2
 P3(2)  ,P3 time remain:1
============ interrupt============ ===
 P4(1)  ,P4 time remain:3
 P4(2)  ,P4 time remain:2
 P4(3)  ,P4 time remain:1
 P4(4)  ,P4 time remain:0
P4 is finished,  P4 burst time:4,  P4 priority:3
 P5(1)  ,P5 time remain:1
 P5(2)  ,P5 time remain:0
P5 is finished,  P5 burst time:2,  P5 priority:2
============ interrupt============ ===
 P1(1)  ,P1 time remain:1
 P1(2)  ,P1 time remain:0
P1 is finished,  P1 burst time:2,  P1 priority:4
 P3(3)  ,P3 time remain:0
P3 is finished,  P3 burst time:3,  P3 priority:1
============ interrupt============ ===
```

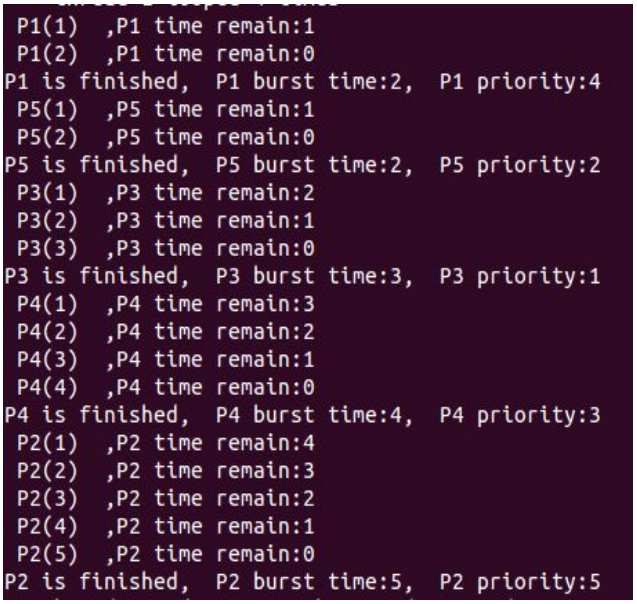
There is an interrupt occuring when P1 first execute so we see P2 show up at the beginning instead of P1. Although P1 and P3 didn't finish in their first execution because of interrupt, they completed successfully in their second execution. The remaining cpu burst time for all threads are updated correctly during whole process. As a result, the result is absolutely correct.

FCFS:



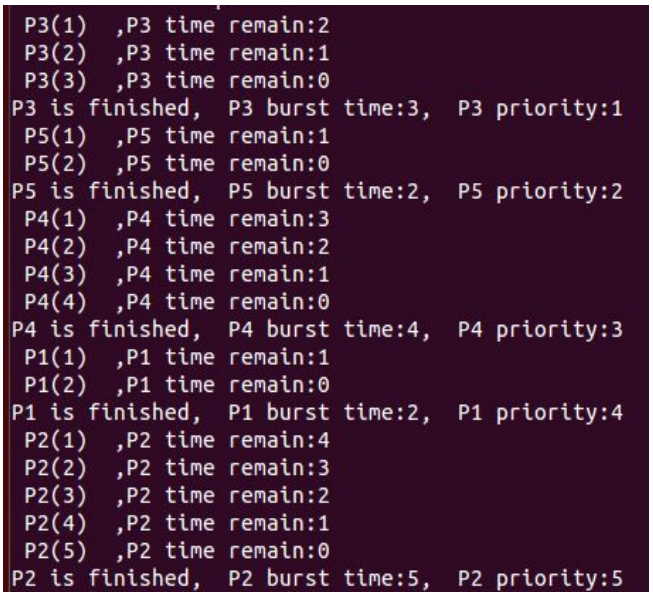
The order of thread is same as the FCFS order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.

SJF:



The order of thread is same as the SJF order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.

Priority:



The order of thread is same as the Priority order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.

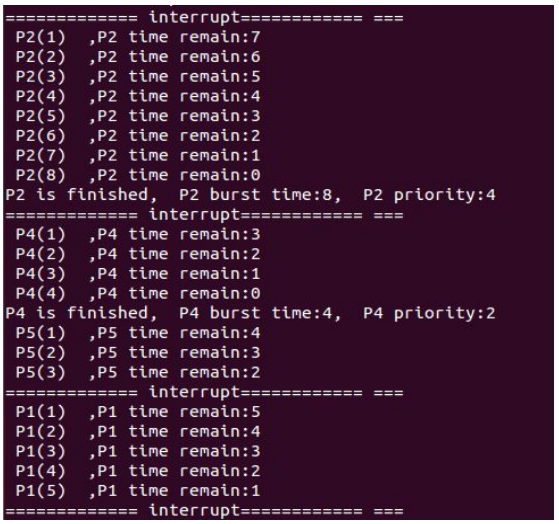
test case 2:

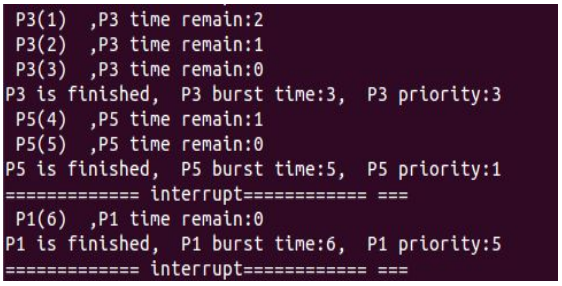
```
#define threadNum 5
void
Thread::SchedulingTest()
{

    char *threadName[threadNum] =   {"P1", "P2", "P3","P4","P5"};
    int threadPriority[threadNum] = {  5 ,   4 ,   3 ,  2 ,  1 };
    int threadBurst[threadNum] =    {  6 ,   8 ,   3 ,  4 ,  5 };
```

The order of FCFS and RR (sorted by arrival order): P1, P2, P3, P4, P5
The order of SJF (sorted by burst time): P3, P4, P5, P1, P2
The order of Priority (sorted by priority number): P5, P4, P3, P2, P1

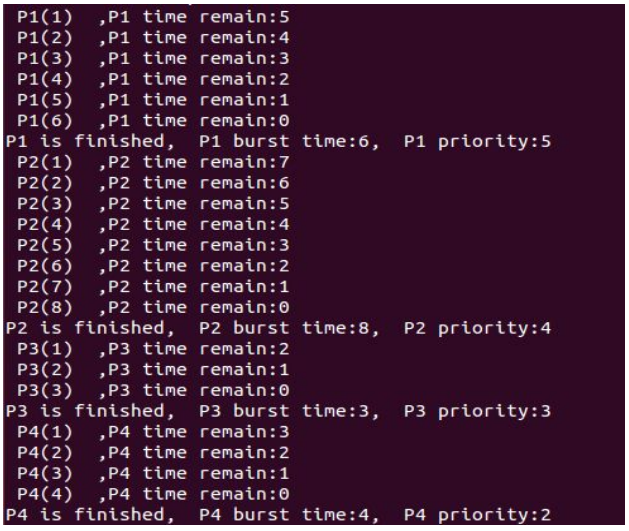
RR:





There is an interrupt occuring when P1, P3, P5 first execute so we see P2 show up at the beginning instead of P1, and P4 show up below P2 rather than P3. Although P3 and P5 didn't finish in their first execution because of interrupt, they completed successfully in their second execution. On the other hand, P1 complete in its third execution. The remaining cpu burst time for all threads are updated correctly during whole process. As a result, the result is absolutely correct.

FCFS:



```
 P5(1)  ,P5 time remain:4
 P5(2)  ,P5 time remain:3
 P5(3)  ,P5 time remain:2
 P5(4)  ,P5 time remain:1
 P5(5)  ,P5 time remain:0
P5 is finished,  P5 burst time:5,  P5 priority:1
```

The order of thread is same as the FCFS order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.

SJF:

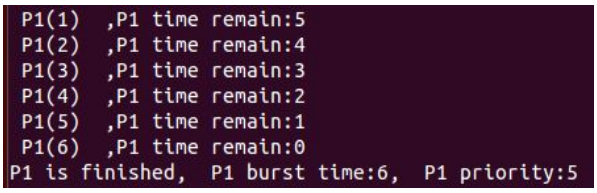
```
 P3(1)  ,P3 time remain:2
 P3(2)  ,P3 time remain:1
 P3(3)  ,P3 time remain:0
P3 is finished,  P3 burst time:3,  P3 priority:3
 P4(1)  ,P4 time remain:3
 P4(2)  ,P4 time remain:2
 P4(3)  ,P4 time remain:1
 P4(4)  ,P4 time remain:0
P4 is finished,  P4 burst time:4,  P4 priority:2
 P5(1)  ,P5 time remain:4
 P5(2)  ,P5 time remain:3
 P5(3)  ,P5 time remain:2
 P5(4)  ,P5 time remain:1
 P5(5)  ,P5 time remain:0
P5 is finished,  P5 burst time:5,  P5 priority:1
 P1(1)  ,P1 time remain:5
 P1(2)  ,P1 time remain:4
 P1(3)  ,P1 time remain:3
 P1(4)  ,P1 time remain:2
 P1(5)  ,P1 time remain:1
 P1(6)  ,P1 time remain:0
P1 is finished,  P1 burst time:6,  P1 priority:5
```

```
 P2(1)  ,P2 time remain:7
 P2(2)  ,P2 time remain:6
 P2(3)  ,P2 time remain:5
 P2(4)  ,P2 time remain:4
 P2(5)  ,P2 time remain:3
 P2(6)  ,P2 time remain:2
 P2(7)  ,P2 time remain:1
 P2(8)  ,P2 time remain:0
P2 is finished,  P2 burst time:8,  P2 priority:4
```

The order of thread is same as the SJF order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.

Priority:

```
 P5(1)  ,P5 time remain:4
 P5(2)  ,P5 time remain:3
 P5(3)  ,P5 time remain:2
 P5(4)  ,P5 time remain:1
 P5(5)  ,P5 time remain:0
P5 is finished,  P5 burst time:5,  P5 priority:1
 P4(1)  ,P4 time remain:3
 P4(2)  ,P4 time remain:2
 P4(3)  ,P4 time remain:1
 P4(4)  ,P4 time remain:0
P4 is finished,  P4 burst time:4,  P4 priority:2
 P3(1)  ,P3 time remain:2
 P3(2)  ,P3 time remain:1
 P3(3)  ,P3 time remain:0
P3 is finished,  P3 burst time:3,  P3 priority:3
 P2(1)  ,P2 time remain:7
 P2(2)  ,P2 time remain:6
 P2(3)  ,P2 time remain:5
 P2(4)  ,P2 time remain:4
 P2(5)  ,P2 time remain:3
 P2(6)  ,P2 time remain:2
 P2(7)  ,P2 time remain:1
 P2(8)  ,P2 time remain:0
P2 is finished,  P2 burst time:8,  P2 priority:4
```



The order of thread is same as the Priority order we describe above. The remaining cpu burst time for all threads are updated correctly during whole process. Therefore, the result is completely correct.