

Operating System Project 3-2

何孟遠

R11222007

December 28, 2023

For the given code, the execution result on my virtual machine is like the figure below, ACBDB.

```
A: remaining 2
A: remaining 1
A: remaining 0
C: remaining 6
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
B: remaining 8
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
D: remaining 2
D: remaining 1
D: remaining 0
B: remaining 0
```

However, since the round-robin scheduling should act like FCFS initially, the result here is **wrong**. If the results are correct, the output should be like ABCDBDBDBD.

To know what is happening during the execution, I add the debug flag and output the ready queue every time a thread is running. The results are in the figures below.


```
ABCD
A: remaining 2
A: remaining 1
A: remaining 0
Ready list contents:
BCD
Ready list contents:
CD
C: remaining 6
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
Ready list contents:
DB
Ready list contents:
B
B: remaining 8
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
Ready list contents:
D
D: remaining 2
D: remaining 1
D: remaining 0
Ready list contents:
B
B: remaining 0
List is empty
List is empty
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

```
Ready list contents:
ABCD
Switching from: main to: A
Beginning thread: A
Deleting thread: main
A: remaining 2
A: remaining 1
A: remaining 0
Finishing thread: A
Sleeping thread: A
Ready list contents:
BCD
Switching from: A to: B
Beginning thread: B
Deleting thread: A
Yielding thread: B
Ready list contents:
CD
Putting thread on ready list: B
Switching from: B to: C
Beginning thread: C
C: remaining 6
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
Finishing thread: C
Sleeping thread: C
Ready list contents:
DB
Switching from: C to: D
Beginning thread: D
Deleting thread: C
Yielding thread: D
```

From the figures, we can see that Thread B and Thread D are yielding immediately after the previous thread is finished and move to the end of the ready queue. As a result, we saw outputs of Thread C after the outputs of Thread A instead of Thread B. Furthermore, I noticed that Thread B and D yield several times even at the end of the whole program.

```
B: remaining 1
Yielding thread: B
Putting thread on ready list: B
Switching from: B to: D
Now in thread: D
D: remaining 2
D: remaining 1
D: remaining 0
Finishing thread: D
Sleeping thread: D
Switching from: D to: B
Now in thread: B
Deleting thread: D
B: remaining 0
Finishing thread: B
Sleeping thread: B
```

From the above analysis, I know that the problems relate to the too-early yielding. Track down the whole process of the scheduling test, I found out the only code we manually set the yielding and the criteria for RR scheduling is the function `Alarm::CallBack()` in `alarm.cc`:



```
void Alarm::CallBack() {
    ...
    else {          // there's someone to preempt
        if(kernel->scheduler->getSchedulerType() == RR){
            interrupt->YieldOnReturn();
        }
    }
}
```

Indeed, if we comment on the above line and run the code, we will get results the same as FCFS, which is ABCD. The next problem hence becomes where is the correct place to set the yielding.

Recall the concept of RR scheduling, the spirit of RR scheduling is the **quantum time**, which is the time limit for a single process. The kernel always runs the first process in the ready queue as FCFS does, but if a process executes longer than the time limit, the process will be suspended and moved to the end of the ready queue, which is yielding.

Check out the files of alarm.cc, scheduling.cc, threads.cc, and interrupt.cc, the yielding (interrupt->YieldOnReturn()) happens when we create the test cases or every 100 ticks, which is quite strange since the yielding should happen according to the tick the thread used.

```
Finishing thread: A
Sleeping thread: A
Switching from: A to: B
Beginning thread: B
Deleting thread: A
    interrupts: off -> on
== Tick 100 ==
    interrupts: on -> off
Time: 100, interrupts off
Pending interrupts:
Interrupt handler timer, scheduled at 100
End of pending interrupts
Invoking interrupt handler for the
timer at time 100
Scheduling interrupt handler the timer at time = 200
    interrupts: off -> on
    interrupts: on -> off
Yielding thread: B
```

To set the new quantum time, we need to set the yielding in the place where the test cases run, which is the threadBody() in thread.cc. We couldn't move interrupt->YieldOnReturn() to here since it will cause the kernel abortion due to the incorrect flag yieldOnReturn, nor set the flag Scheduler::yieldOnReturn to TRUE here since it is the private variance. Fortunately, we have the function Thread::Yield() that could directly switch contexts.

The code after modification is like the figure below.

```
void threadBody() {
    Thread *thread = kernel->currentThread;
    int quantum = 4;
    int q = 0;
    while (thread->getBurstTime() > 0) {
        q++;
        thread->setBurstTime(thread->getBurstTime() - 1);
        kernel->interrupt->OneTick();
        cout << kernel->currentThread->getName() << ": remaining " <<
            kernel->currentThread->getBurstTime() << endl;
        if (q % 4 == 0 && kernel->scheduler->getSchedulerType() == RR) {
            kernel->currentThread->Yield();
        }
    }
}
```

For the test cases we build and the quantum time is set to 4, the expected result should be: ABCDBCB. The result is listed in the below-left figure.

```
A: remaining 2
A: remaining 1
A: remaining 0
B: remaining 8
B: remaining 7
B: remaining 6
B: remaining 5
C: remaining 6
C: remaining 5
C: remaining 4
C: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
C: remaining 2
C: remaining 1
C: remaining 0
B: remaining 0
List is empty
List is empty
```

```
A: remaining 7
A: remaining 6
A: remaining 5
A: remaining 4
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
C: remaining 7
C: remaining 6
C: remaining 5
C: remaining 4
D: remaining 7
D: remaining 6
D: remaining 5
D: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
List is empty
List is empty
```

The result is the same as I expected. If I modify the burst time of the test cases to 8 for all, I should expect the result ABCDABCD. Finally, the result listed in the above-right figure is the same as I expected.