Operating Systems

Project Checkpoint 4

111062109 黃庭曜

Table of Contents

§1 Screenshots for compilation

§1.1 Command "`make clean`"

```
huangtingyao@crazytingyao ppc4 % make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym *.asm *.lk
rm: *.ihx: No such file or directory
rm: *.lnk: No such file or directory
make: *** [clean] Error 1
```

Fig. 1: "`make clean`" command execution result

§1.2 Command "`make`"

```
huangtingyao@crazytingyao ppc4 % make
sdcc -c  test3threads.c
test3threads.c:88: warning 158: overflow in implicit constant conversion
sdcc -c  preemptive.c
preemptive.c:216: warning 85: in function ThreadCreate unreferenced function argument : 'fp'
preemptive.c:274: warning 158: overflow in implicit constant conversion
sdcc  -o test3threads.hex test3threads.rel preemptive.rel
```

Fig. 2: "`make`" command execution result

§2 Screenshots and explanation

§2.1 Address list

The address list of all functions is shown in **Fig. 3**.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:   00000014  _Producer1                      test3threads
C:   00000062  _Producer2                      test3threads
C:   000000B0  _Consumer                       test3threads
C:   000000FA  _main                           test3threads
C:   00000115  __sdcc_gsinit_startup           test3threads
C:   00000119  __mcs51_genRAMCLEAR             test3threads
C:   0000011A  __mcs51_genXINIT                test3threads
C:   0000011B  __mcs51_genXRAMCLEAR            test3threads
C:   0000011C  _timer0_ISR                     test3threads
C:   00000120  _Bootstrap                      preemptive
C:   00000149  _myTimer0Handler                preemptive
C:   000001C5  _ThreadCreate                   preemptive
C:   00000239  _ThreadYield                    preemptive
C:   0000028D  _ThreadExit                     preemptive
```

Fig. 3: Function address list

§2.2 Producer1

The `Producer1`'s code address is between 0x14 and 0x60. The screenshot for the first `Producer1` call is shown in **Fig. 4**.

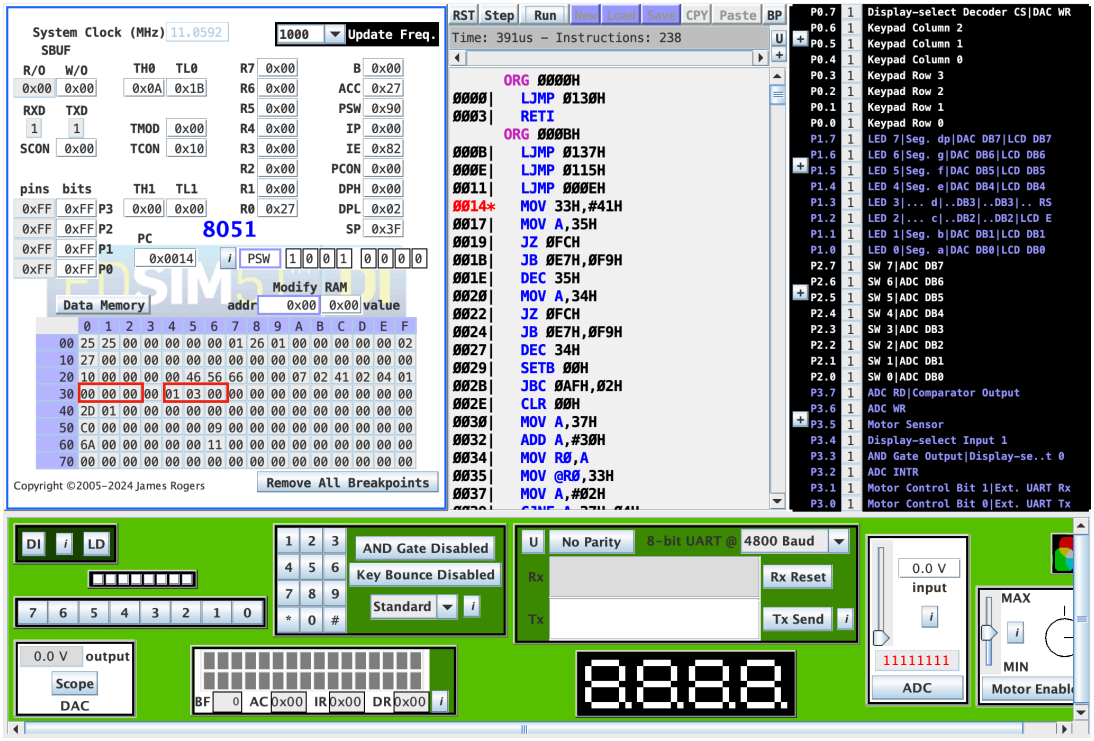

Fig. 4: Screenshot for first `Producer1` call

The semaphore values of `mutex`, `empty`, and `full` are stored in 0x34, 0x35, and 0x36, respectively. For the first run, since the value of `empty` is 3, `Producer1` will enter the critical part and push three characters into the buffer (the buffer ranges from 0x30 to 0x32). Then, the values of `mutex`, `empty`, and `full` will be changed to 1, 0, and 3, respectively. The code for `Producer1` is shown in **Fig. 5**, and the screenshot before the `Producer1` was preempted is shown in **Fig. 6**.

```c
#define LABEL(x) x ## $
#define L(x) LABEL(x)
void Producer1(void)
{
    currentChar = 'A';
    while (1){
        SemaphoreWaitBody(empty,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        buffer[producer_ptr] = currentChar;
        producer_ptr = (producer_ptr == 2) ? 0 : producer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
        currentChar = (currentChar == 'Z') ? 'A' : currentChar + 1;
    }
}
```

Fig. 5: Code for `Producer1`

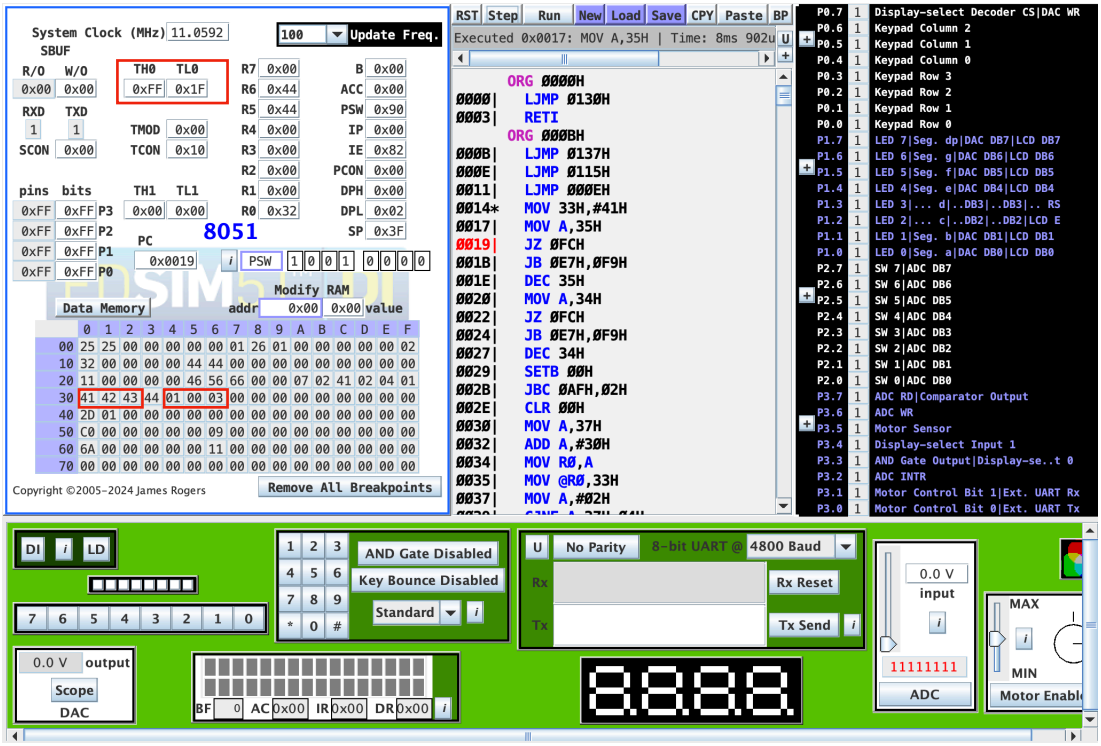

Fig. 6: Screenshot before the `Producer1` was preempted

§2.2 Producer2

The `Producer2`'s code address is between 0x62 and 0xAE. The screenshot for the first `Producer2` call is shown in **Fig. 7**.

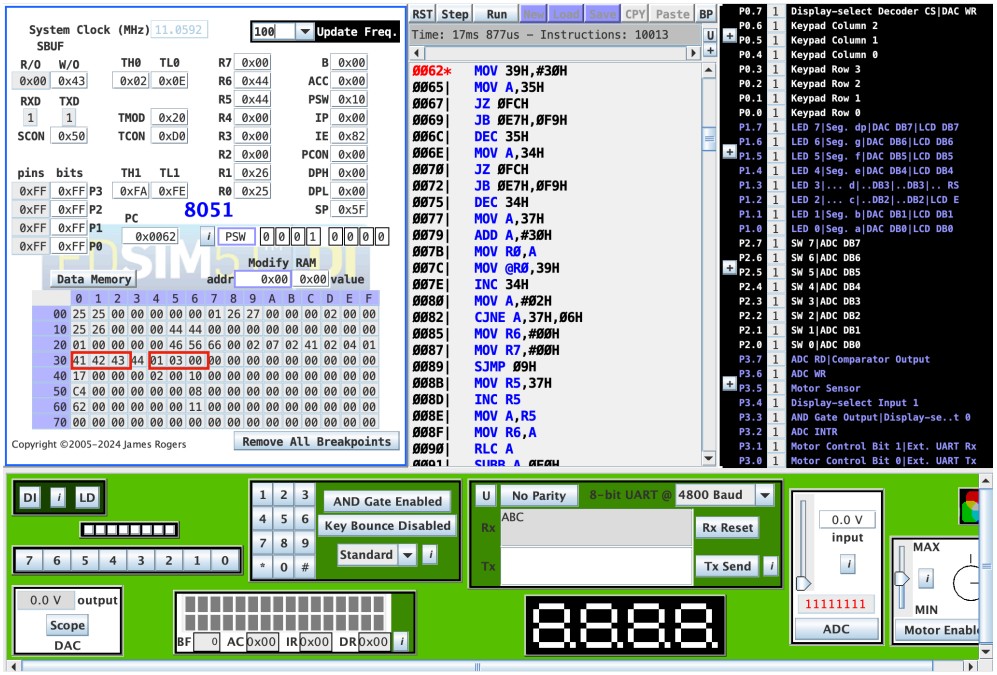

Fig. 7: Screenshot for first `Producer2` call

Since I have changed the scheduling policy, for the first run, the value of `empty` is 3, the `Producer2` will enter the critical part and push three numbers into the buffer. Then, the values of `mutex`, `empty`, and `full` will be changed to 1, 0, and 3, respectively. The screenshot before the `Producer2` was preempted is shown in **Fig. 8**, and the code for `Producer2` is shown in **Fig. 9**.

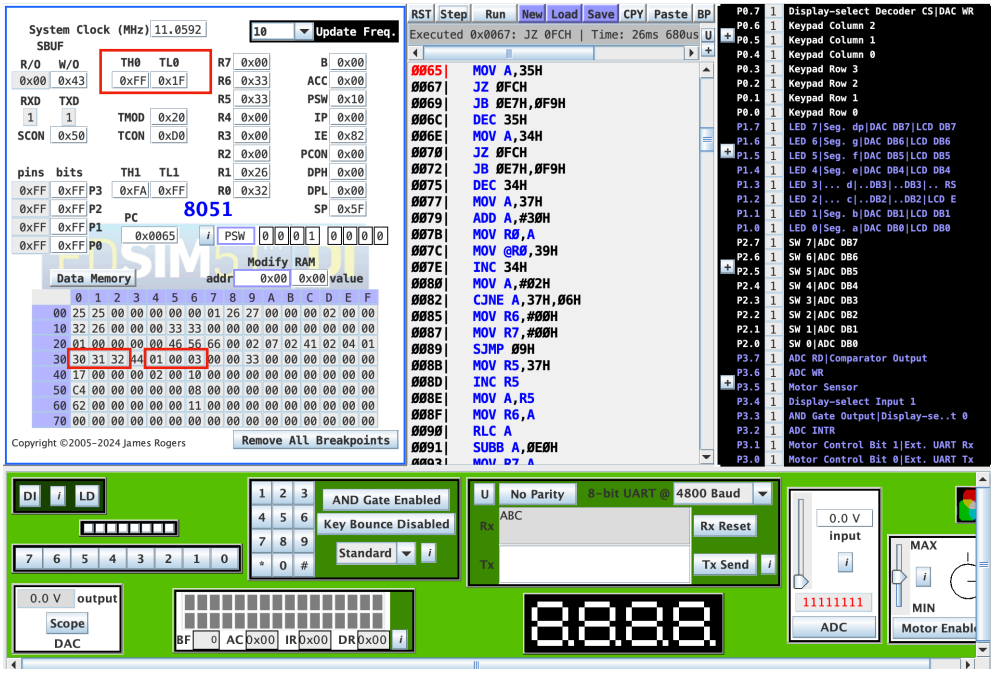

Fig. 8: Screenshot before the `Producer2` was preempted

```
void Producer2(void)
{
    currentNum = '0';
    while (1){
        SemaphoreWaitBody(empty,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        buffer[producer_ptr] = currentNum;
        producer_ptr = (producer_ptr == 2) ? 0 : producer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
        currentNum = (currentNum == '9') ? '0' : currentNum + 1;
    }
}
```

Fig. 9: Code for `Producer2`

§2.4 Consumer

The `Consumer`'s code address is between 0xB0 and 0xF8. The screenshot for the first `Consumer` call is shown in **Fig. 10**.

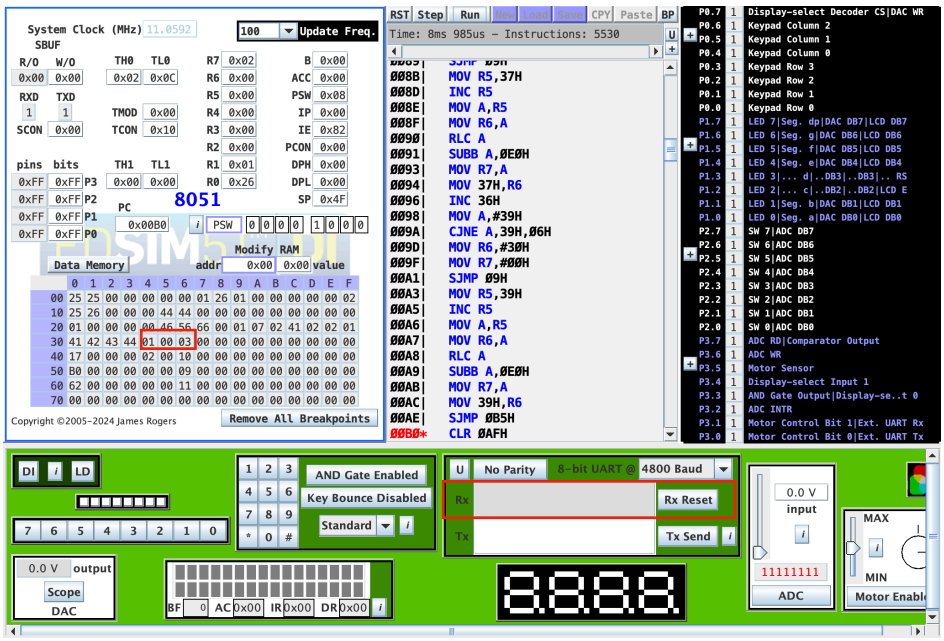

Fig. 10: Screenshot for first `Consumer` call

Since I have changed the scheduling policy, for the first run, the buffer is full of characters, and the `Consumer` will enter the critical part and pop the buffer to the SBUF, then it starts to wait for the RX transmission to complete. If completed, the `Consumer` pops another item in the buffer. After all items are popped, the `Consumer` waits for the timer to preempt it. The screenshot before the `Consumer` was preempted is shown in **Fig. 11**. The code is the same as the consumer in checkpoint 3.

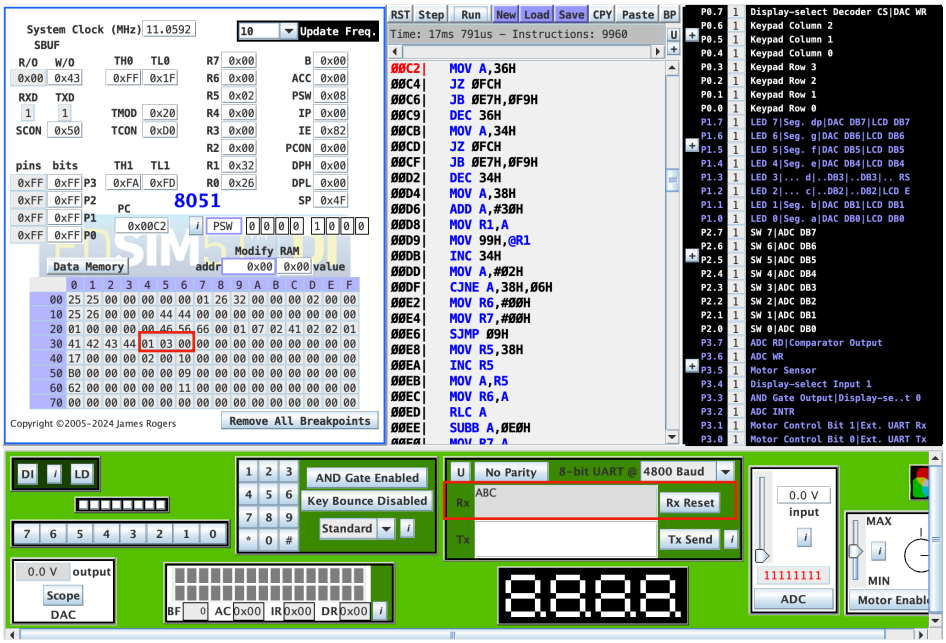

Fig. 11: Screenshot before the `Consumer` was preempted

§2.5 Unfairness

If we perform the round-robin scheduling, unfairness of the two producers will occur. For example, Thread 0 is assigned to `Procuder1`, Thread 1 to `Procuder2`, and Thread 2 to `Consumer`. The round-robin scheduling will execute the threads with the sequence `Procuder1` → `Procuder2` → `Consumer` → `Procuder1`. However, since the timer is long enough for `Producer1` to fill in all buffers, `Procuder2` will be stuck at the semaphore. After the `Consumer` pops all items in the buffer, `Producer1` will fill in all buffers again, and `Procuder2` will never have a chance to push anything in the buffer. The result is shown in **Fig. 12**.

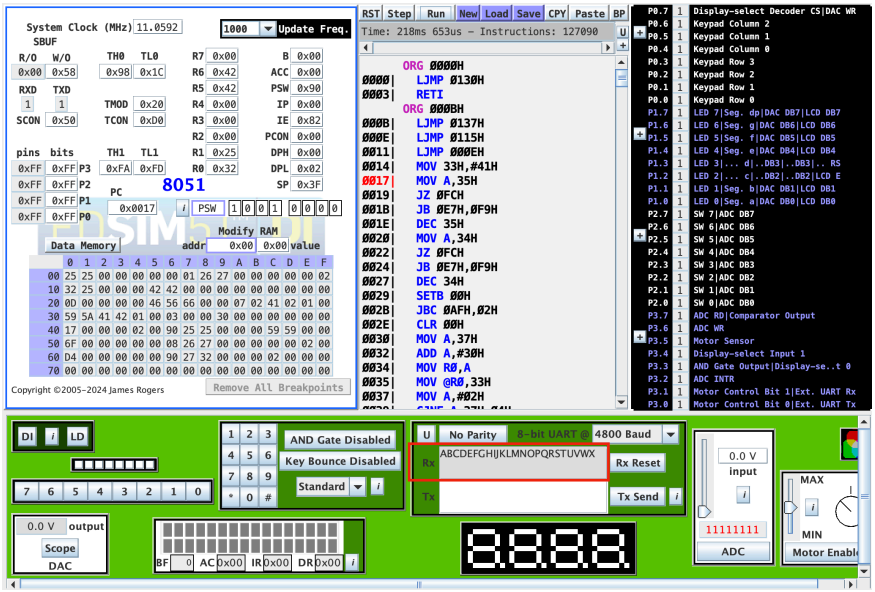

Fig. 12: Unfair condition for `Procuder2`

Reversely, if Thread 0 is assigned to `Procuder2`, Thread 1 to `Procuder1`, and Thread 2 to `Consumer`. The `Procuder1` will never have a chance to push anything in the buffer. The result is shown in **Fig. 13**.

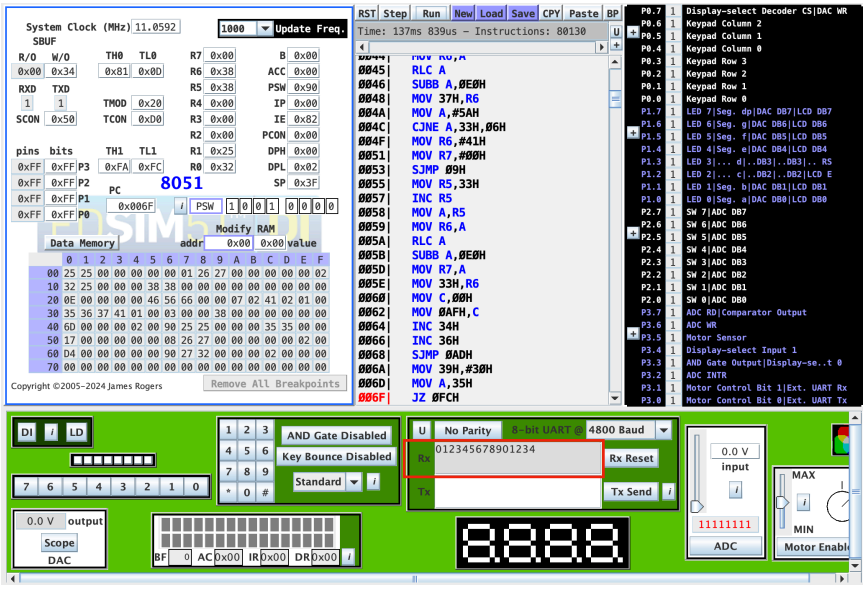

Fig. 13: Unfair condition for `Procuder1`

To solve this, I changed the round-robin scheduling policy. By defining the direction of the thread scheduling, the thread number will first count up till Thread 3, then the direction will be reversed and start to count down till Thread 0. The execution sequence will be:
Thread 0→Thread 1→Thread 2→Thread 3→Thread 2→Thread 1→Thread 0→Thread 1…
Then, assign `Procuder1` to Thread 0, `Procuder1` to Thread 2, and `Consumer` to Thread 1. The scheduling will execute the threads with the sequence `Procuder1 → Consumer → Procuder2 → Consumer → Procuder1`… Which gives fairness to both Producers. The result is shown in **Fig. 14**, and the code is shown in **Fig. 15**.

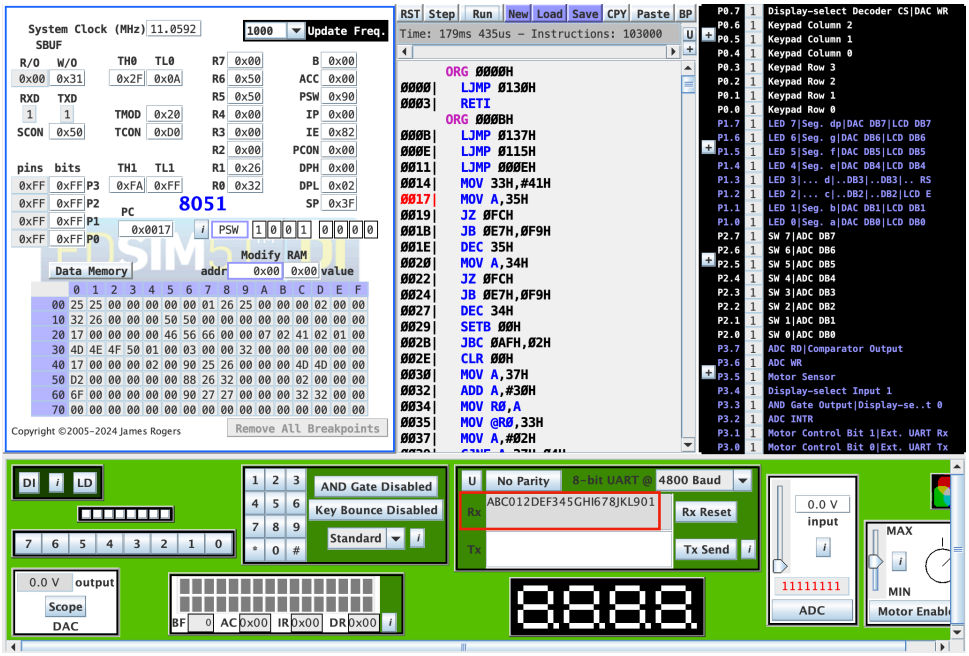

Fig. 14: Unfairness resolved with the new scheduling

```c
void myTimer0Handler(void) {
    EA = 0; // don't do __critial
    SAVESTATE;
    do{
        if(clockwise){
            clockwise = !(currentThread == 3);
            if(currentThread == 3) currentThread = 2;
            else currentThread++;
        }
        else {
            clockwise = (currentThread == 0);
            if(currentThread == 0) currentThread = 1;
            else currentThread--;
        }
        temp = 1 << currentThread;
        if (threadMask & temp){
            break;
        }
    } while (1);
```

Fig. 15: Code for new scheduling