PPC4 Report 110062120 高小榛

1. Code explanation

(1) test3thread.c
In SemaphoreCreate, the function is to create a semaphore with initial value n.

First, I create global variable Cur_ID, Mutex, Full, Empty, Word1, Word2, Head, Tail and Buffer. The meaning of these variable is define in this picture.

```
// Global variables for shared buffer and threading
__data __at (0x25) ThreadID Cur_ID;  // Current thread ID
__data __at (0x36) char Mutex;       // Mutex semaphore for mutual exclusion
__data __at (0x37) char Full;        // Semaphore indicating buffer is full
__data __at (0x38) char Empty;       // Semaphore indicating buffer is empty
__data __at (0x39) char Word1;       // Character to be produced
__data __at (0x3A) char Word2;
__data __at (0x3B) char Head;        // Head index for buffer
__data __at (0x3C) char Tail;        // Tail index for buffer
__data __at (0x3D) char Buffer[3] = {' ', ' ', ' '}; // Circular buffer
```

In the Producer1 function, I assign first word produce with 'A', then in the while loop, I use SemaphoreWait and SemaphoreSignal to design. First wait Empty until buffer has empty space, then we can enter critical section with wait for Mutex. In the critical section, first add new character to buffer, update tail index then update character. Finally leave the critical section with signal mutex, then signal buffer is full.

In the Producer2 function, I do the similar things but change the character from "A~Z" to "0~9".

In Consumer function, first I initialize Tx for polling using the information in class. , change the TMOD to TMOD |= 0x20. Then we should wait for new data from producer, write data to serial port Tx, poll for Tx to finish writing (TI), then finally clear the flag. So first we should wait for data in buffer with wait Full, then wait for mutex to enter critical section. In the critical section, we can do thing that mention above. Wait for Tx to be ready, send character from buffer, clear Tx interrupt flag then circular increment of head. Finally exit critical section then signal buffer is empty.

In main function, I create 3 semaphore Mutex, Full and Empty. Then initialize head index and tail index. Use ThreadCreate function to create producer1 thread and producer2 thread then store with cur_id. Finally, use assembly to set up stack pointer, call consumer function.

Additionally, I add timer0_ISR to let ISR call my routine timer.

(2) preemptive.c

First, I create the static global SP0 ~ SP3, Map, Cur_ID, SP_old, SP_new, New_ID and Next_ID. Then I define 2 macro save state and restore state in a similar way. In savestate we should saving the current thread context. So first push ACC, B register, Data pointer registers (DPL, DPH), PSW with assembly. Then save SP into the saved Stack Pointers array with if-else block to check which place we should store the SP into it. In restorestate, I first assign SP to the saved SP from the saved stack pointer array, then pop the registers PSW, data pointer registers, B reg, and ACC.

In bootstrap function, I clear thread bitmap indicating no threads are running and initialize stack pointer. I initialize timer and interrupts for preemption with set TMOD = 0, IE = 0x82 to enable timer 0 interrupt, and set TR0 = 1 to start running timer0. Set Next_ID to '1' initially, this means that After the main thread (thread 0), the next thread to run will be thread 1. Then I create a thread for main and store with Cur_ID and restore the content to run main function.

In myTimer0Handler function, first I use EA=0 to disable interrupts, then save the current thread state. To perform fairness, I use the if-else block to determine which thread should run. First, If the current thread is not thread 0, then we should switch to thread 0 for execution next. Else if current thread is 0, we change the next thread based on Next_ID. Then toggle Next_ID for the next interrupt to switch to another thread. Finally restore the state, re-enable interrupts then return.

In ThreadCreate function, first disable interrupt, check to see we have not reached the max #threads and return -1 if no available thread ID. Then initialize a New_ID, run a if-else block to check each thread ID (0 to 3) to find an available one. For each thread ID, perform bitwise AND with Map and check if the result is 0. If 0, it means the thread ID is available for use. If Thread I is available: first set New_ID to 'i', then update Map to indicate Thread I is now in use, finally set SP_new to the starting stack location for Thread i. Then save the current SP in SP_old, set SP to the new thread's starting stack location. Then push the return address (fp) onto the stack for the new thread. Then initialize the registers to 0 for the new thread by set A to 0 then push into each register. Then set up the PSW for the new thread and save its SP, and PSW setup depends on the thread ID to use the correct register bank. Finally restore the original SP from SP_old, re-enable interrupt, and return new_ID.

In ThreadYield function, I use Round-Robin to find the next thread to run. First cycle through thread IDs and select the next valid thread, then break out of the loop if a valid thread is found, finally restore state.

In ThreadExit function, I clear the current thread from the thread bitmap and set up for context switch, then modify the thread bitmap to indicate the current thread is no longer valid. After dealing with Map, also cycle through thread IDs and select the next valid thread, then break out of the loop if a valid thread is found, finally restore state.

(3) preemptive.h

In preemptive.h, I add some definition according to the document. First, CNAME and LABELNAME is to concatenating symbols, it can add _ in front and $ in tail. Then I define SemaphoreSignal is to signal a semaphore. SemaphoreWaitBody do lots of things. First check semaphore value, decoded wait if it is 0, finally decrement semaphore. Last I define SemaphoreWiat, it will use SemaphoreWaitBody to wait a semaphore.

2. Fairness

(1) Round-Robin Scheduling: The scheduler cycles through the threads, allowing each to run in turn. This is evident in the myTimer0Handler function, where threads are switched based on the Cur_ID and Next_ID variables. This approach ensures that every thread gets a chance to run, preventing any single thread from monopolizing the CPU.

(2) Role of Next_ID: The variable Next_ID plays a crucial role in determining the next thread to be scheduled. It alternates between threads, ensuring that after the main thread (ID 0) executes, the next thread (either ID 1 or 2) gets scheduled. This mechanism ensures a balanced rotation between the main thread and other threads.

(3) Consideration of Thread Starvation: The current implementation effectively mitigates the risk of thread starvation. Given the equal distribution of execution time among threads, each thread is guaranteed to be scheduled. This prevents any thread from being indefinitely deprived of CPU time.

3. Typescript

(1) Makefile

```
CC = sdcc
CFLAGS = -c
LDFLAGS =
#--stack-after-data --stack-loc 0x39 --data-loc 0x20

C_OBJECTS = test3thread.rel preemptive.rel

all: test3thread.hex

test3thread.hex:    $(C_OBJECTS) $(ASM_OBJECTS)
                $(CC) $(LDFLAGS) -o test3thread.hex $(C_OBJECTS)

clean:
    rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym

%.rel:  %.c    preemptive.h Makefile
    $(CC) $(CFLAGS) $<
```

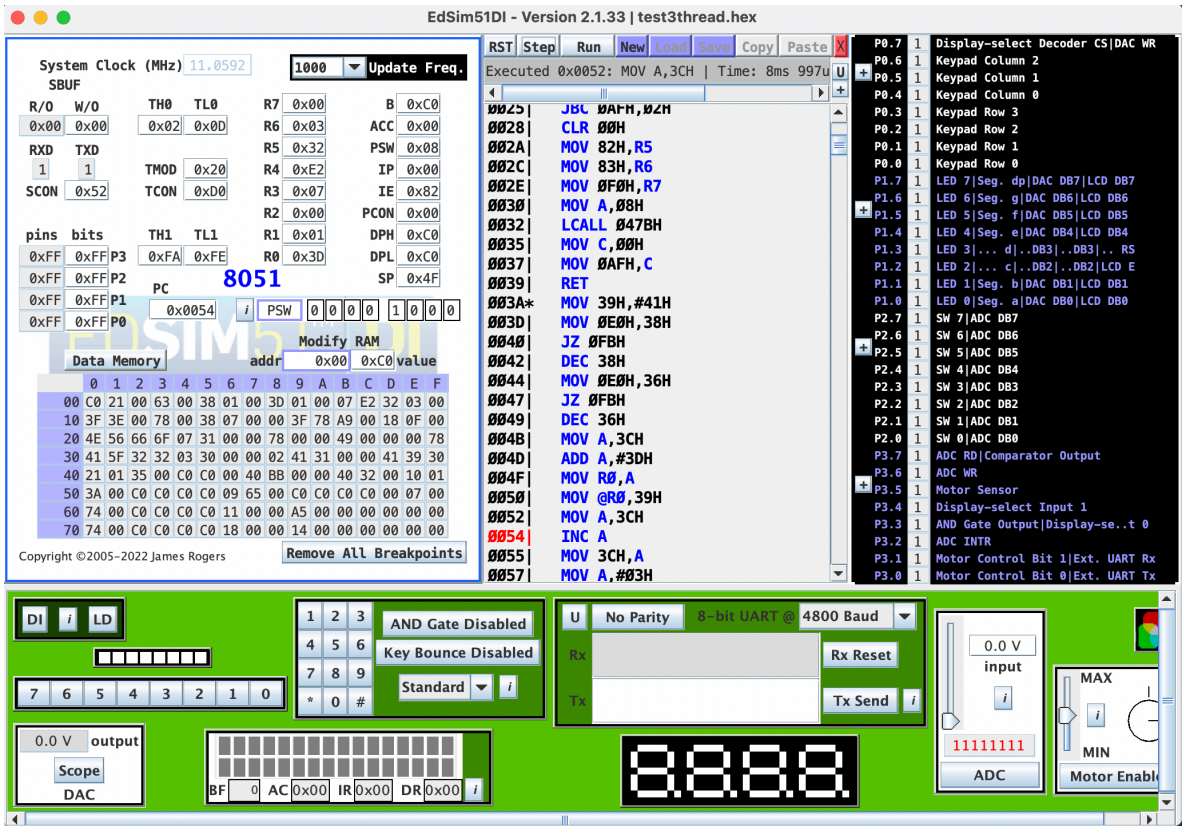
(2) Screenshot

```
sunny@xiaozhendeAir 110062120-ppc4 % make
sdcc -c  test3thread.c
sdcc -c  preemptive.c
preemptive.c:326: warning 85: in function ThreadCreate unreferenced function arg
ument : 'fp'
sdcc  -o test3thread.hex test3thread.rel preemptive.rel
sunny@xiaozhendeAir 110062120-ppc4 % make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym
rm: *.ihx: No such file or directory
rm: *.lnk: No such file or directory
make: *** [clean] Error 1
```

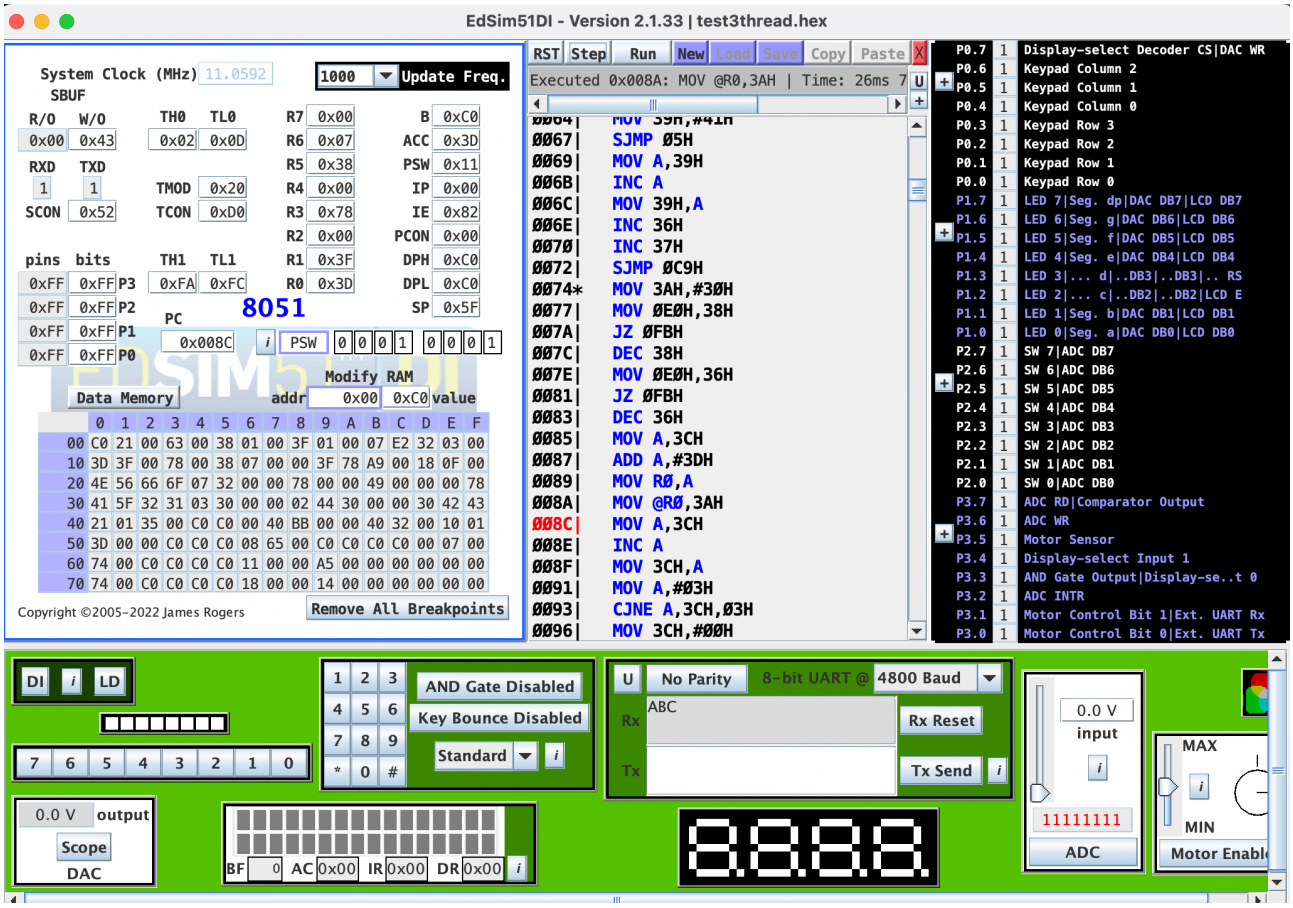
4. Screenshots and explanation

(1) Producer running

Producer1 is at 003A according to map.

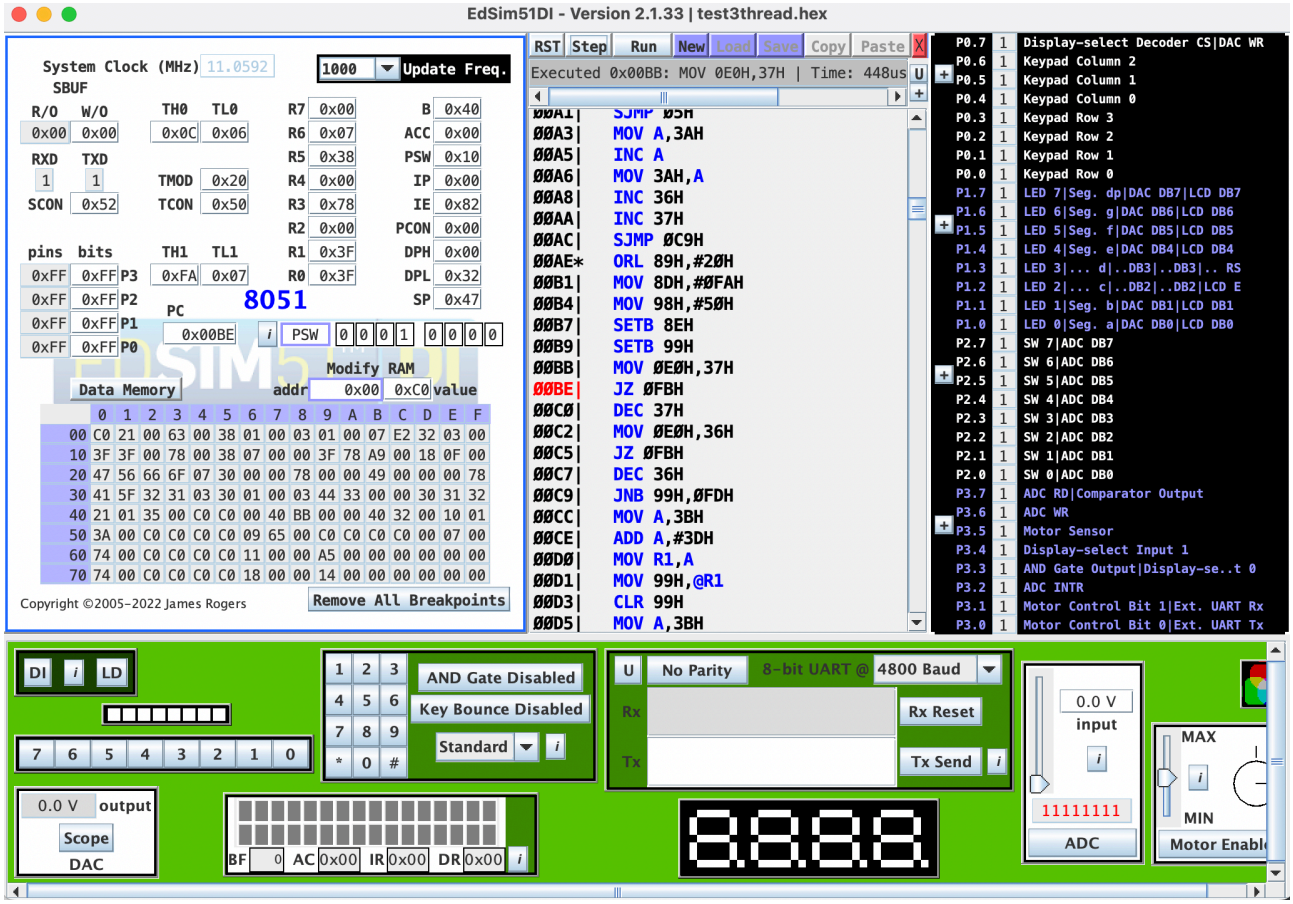


Producer2 is at 0074 according to map.



(2) Consumer running

Consumer is at 00AE according to map.



(3) UART output

The UART output is fair because the output of producer1 and producer2 is equal.