Programming Project Checkpoint 2

[112006234] [Justin Thiadi 程煒財]

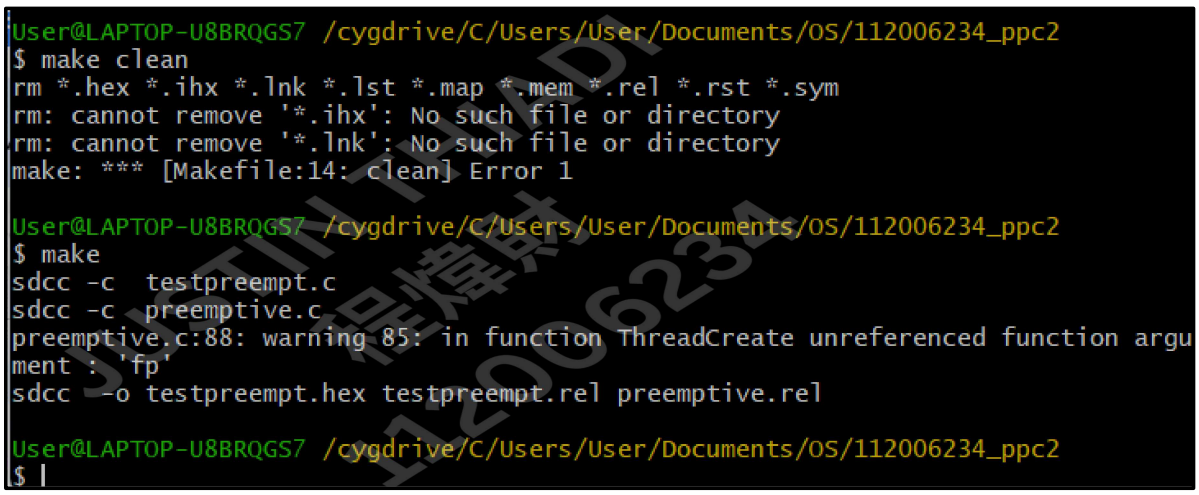
Task 1:

Turn in Screenshots showing compilation of your code using a modified Makefile (same as for cooperative except the file names are changed to the preemptive version). You should use the following two commands (Note: $ is the prompt displayed by the shell and is not part of the command that you type.) The first one deletes all the compiled files so it forces a rebuild if you have compiled before. The second one compiles it.

```
$ make clean
$ make
```

It should show actual compilation, warning, or error messages. Note that not all warnings are errors. The compiler should generate several `testpreempt.*` files with different extensions:

- the `.hex` file can be opened directly in EdSim51
- the `.map` file shows the mapping of the symbols to their addresses after linking

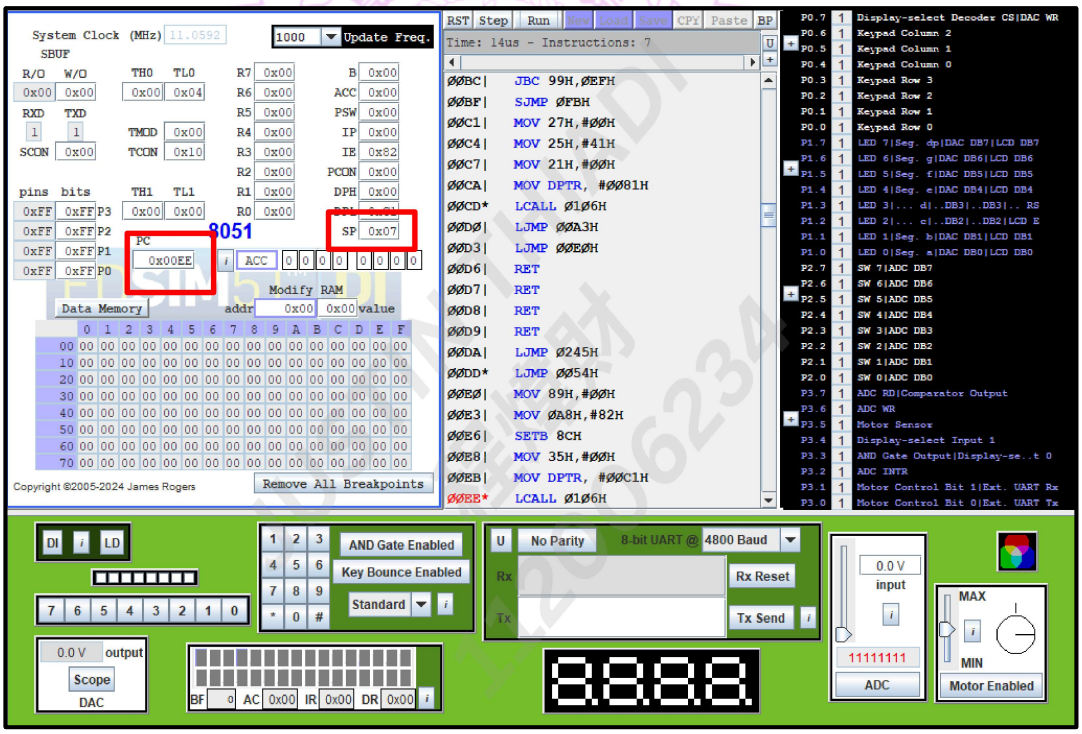


Task 2:

Look up the addresses for your symbols (i.e., functions, variables, etc) in the file testpreempt.map. Set one or more breakpoints in EdSim51's assembly code window after you have assembled it.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:   00000081  _Producer                       testpreempt
C:   000000A3  _Consumer                       testpreempt
C:   000000C1  _main                           testpreempt
C:   000000D3  __sdcc_gsinit_startup           testpreempt
C:   000000D7  __mcs51_genRAMCLEAR             testpreempt
C:   000000D8  __mcs51_genXINIT                testpreempt
C:   000000D9  __mcs51_genXRAMCLEAR            testpreempt
C:   000000DA  _timer0_ISR                     testpreempt
C:   000000E0  _Bootstrap                      preemptive
C:   00000106  _ThreadCreate                   preemptive
C:   0000018C  _ThreadYield                    preemptive
C:   000001E6  _ThreadExit                     preemptive
C:   00000245  _myTimer0Handler                preemptive
```

- Take one screenshot before each ThreadCreate call. Explain how the stack changes.



This is for ThreadCreate(main), here, before any ThreadCreate():
SP = 0x07 (default value after reset).

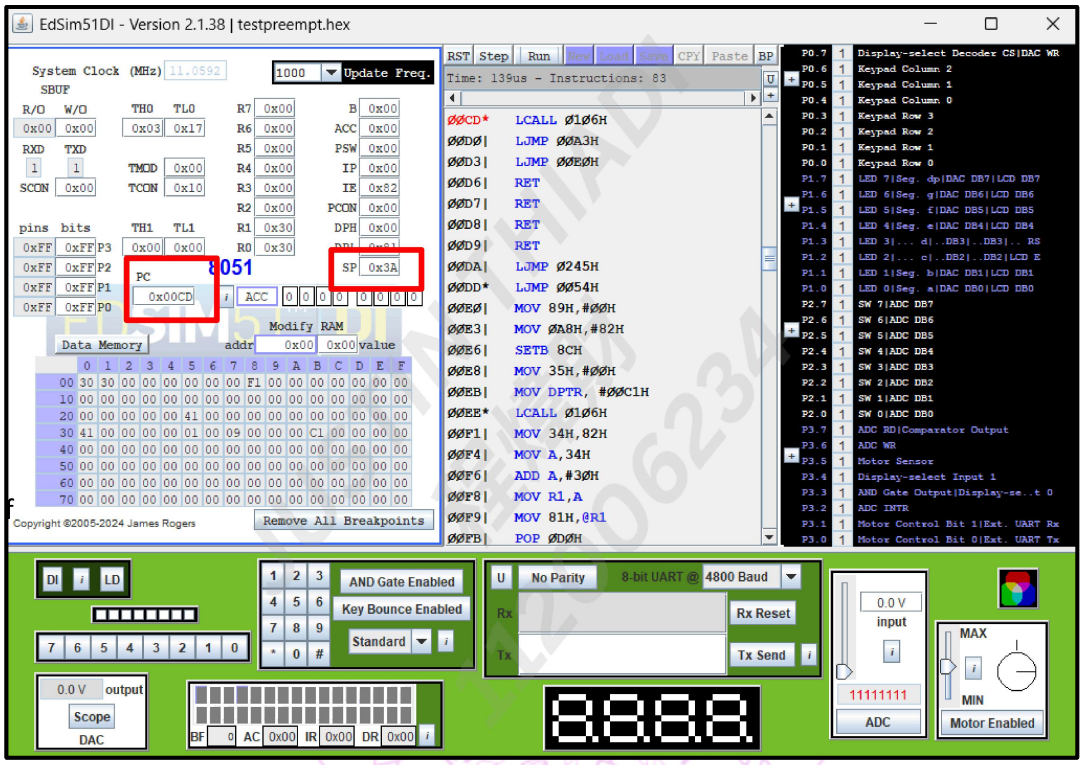
- Nothing has been pushed yet, so the stack is almost empty

To set up a new thread, ThreadCreate() pushes 7 things onto the stack:

- Return address of Function (so the thread starts here):
    - Low byte of address
    - High byte of address
- Registers:
    - ACC (A register)
    - B register
    - DPL (Data Pointer Low)
    - DPH (Data Pointer High)
    - PSW (to choose register bank for the thread)

When we call ThreadCreate(main):

- It assigns the next free stack space (in our case, 0x50 or 0x58).
- Sets SP to that value.
- Pushes fake return address of main.
- Pushes dummy register values (ACC, B, etc.).
- Saves this SP value into the thread table.
- Restores the original SP of the current thread.



This one is for ThreadCreate(Producer).
PC = 0x0CD: that's where LCALL 0x106 happens — it's calling ThreadCreate.

SP is currently 0x3A, meaning the next thread (after the main thread) is being set up — exactly what happens during ThreadCreate(Producer) if main was already created.

Before ThreadCreate(Producer) Call:

- SP = 0x3A
- The current thread (main) is running.

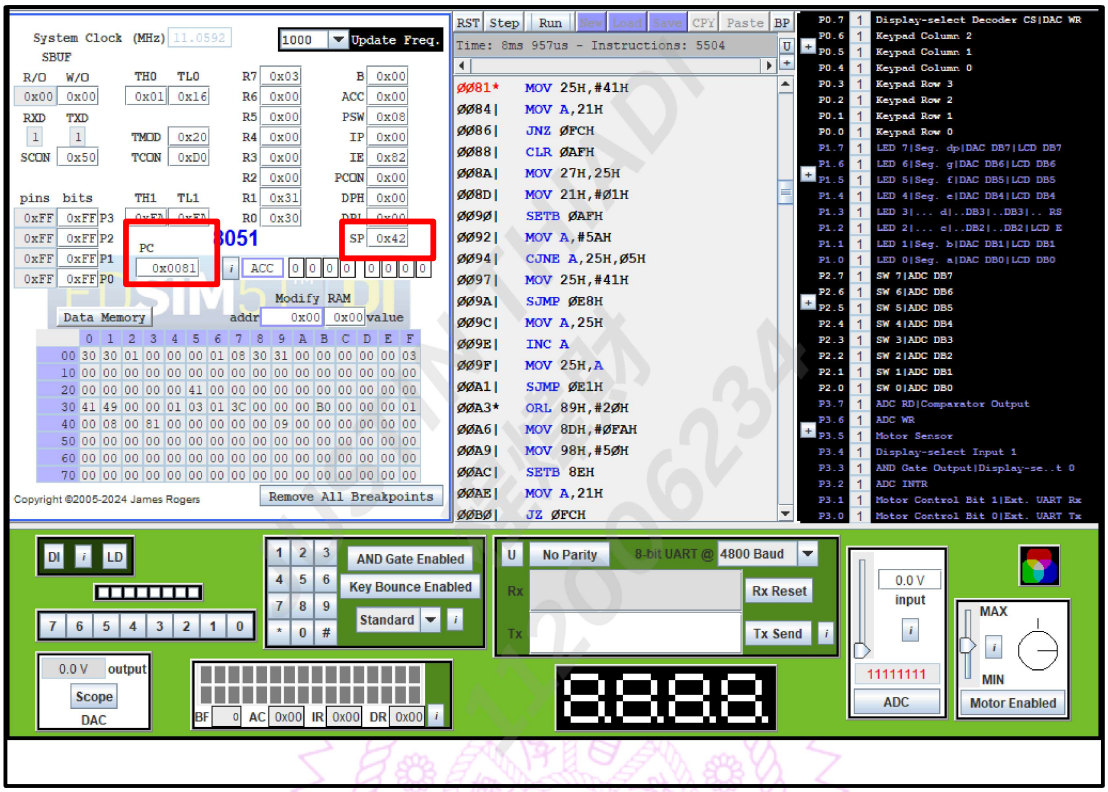
During ThreadCreate(Producer):

- The new thread's SP is set to 0x3C (next available stack space).
- The following values are pushed to the stack for the new thread:
  - Return address (low byte, high byte) of Producer — simulates a function call.
  - ACC (Accumulator) — initialized to 0x00.
  - B (Register B) — initialized to 0x00.
  - DPL (Data Pointer Low) — initialized to 0x00.
  - DPH (Data Pointer High) — initialized to 0x00.
  - PSW (Program Status Word) — set to 0x08 for Register Bank 1.

After ThreadCreate(Producer):

- SP = 0x3C — points to the top of the new stack for Producer.
- SP for the current thread (main) is restored — back to 0x3A.

- Take one screenshot when the Producer is running. How do you know?



Firstly, we can see that SP (Stack Pointer) = 0x42:

- This matches the starting stack address of the Producer thread.
- In the previous screenshot, ThreadCreate(Producer) set the Producer's SP to 0x3A.

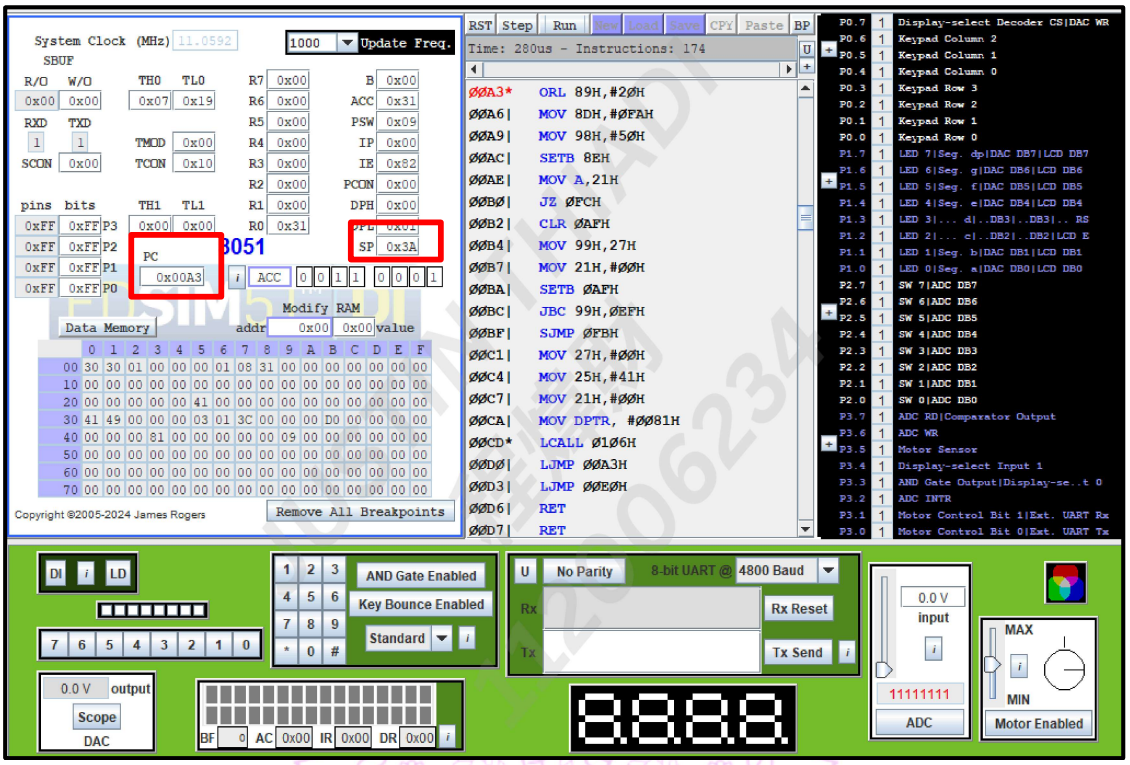
PC (Program Counter) = 0x0081:

- This address matches the starting point of the Producer function, as set up during ThreadCreate (refer to the screenshot showing the functions' addresses)

Register Values:

- ACC, B, DPL, DPH are all zero, indicating thread initialization, showing the producer thread is currently active and running

- Take one screenshot when the Consumer is running. How do you know?



First, we can see that SP (Stack Pointer) = 0x3A:

- This would be the starting stack address of the Consumer thread.
- Main and Consumer share the 0x3A range when Producer is at 0x42

PC (Program Counter) = 0x00A3:

- This is within the Consumer function's address range (refer to the screenshot showing the functions' addresses)
- When we keep on clicking STEP after this step, the instructions shown (ORL, MOV, SETB, etc.) match the Consumer initialization code, especially for UART settings

SFR Values:

- The SCON = 0x00 indicates the Consumer has just started.
    - The Consumer only sets SCON to 0x50 after 3 STEPs later, indicating a part of its initial setup.

- How can you tell that the interrupt is triggering on a regular basis?
    - Frequent Jumps to ISR Location:
        - The Program Counter (PC) repeatedly jumps to 0x000B, which is the Timer 0 interrupt vector (this can be observed when we set the BP to B)
        - From 0x000B, it jumps to the actual ISR at 0x00EA (_myTimer0Handler) -> refer to code
        - This jump is consistent between outputs (like before printing A, then B).
        - This repeated PC movement confirms that the Timer 0 interrupt is occurring periodically.

    - Regular Stack Changes:
        - The Stack Pointer (SP) consistently increases when the interrupt triggers (pushing values) and decreases when the ISR finishes (popping values)
            - We can observe this by clicking STEP several times to see:
                - When the interrupt triggers, the SP increases by several bytes. This is because the ISR pushes the return address and registers onto the stack.
                - After the ISR completes, the SP decreases, as it pops the values back.
        - This push-pop pattern repeats periodically, showing that the interrupt fires at regular intervals

    - No Program Stalling:
        - After each interrupt, the main program resumes without freezing or crashing.
        - This shows the system switches back and forth between the ISR and the main program, proving that the interrupt handling is regular