Programming Project Checkpoint 3

[112006234] [Justin Thiadi 程煒財]

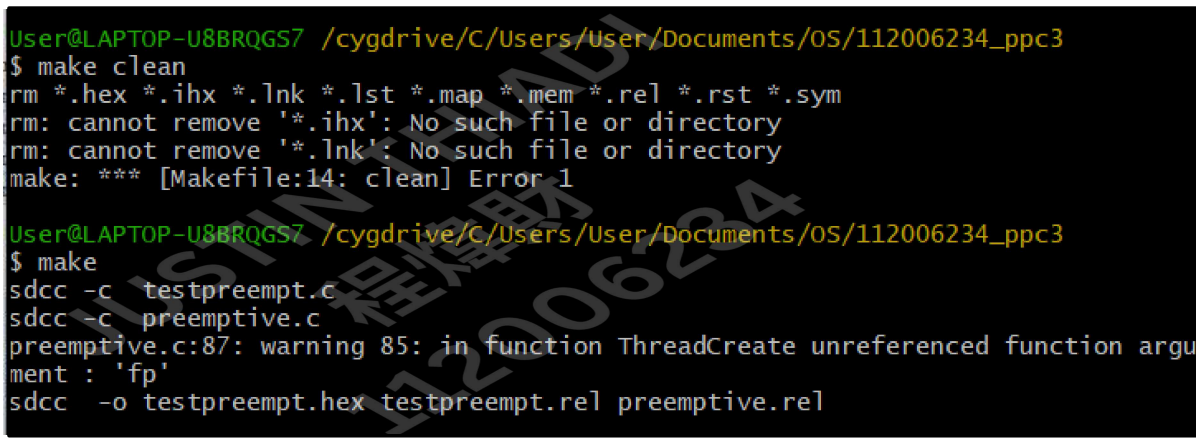
Task 1:

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



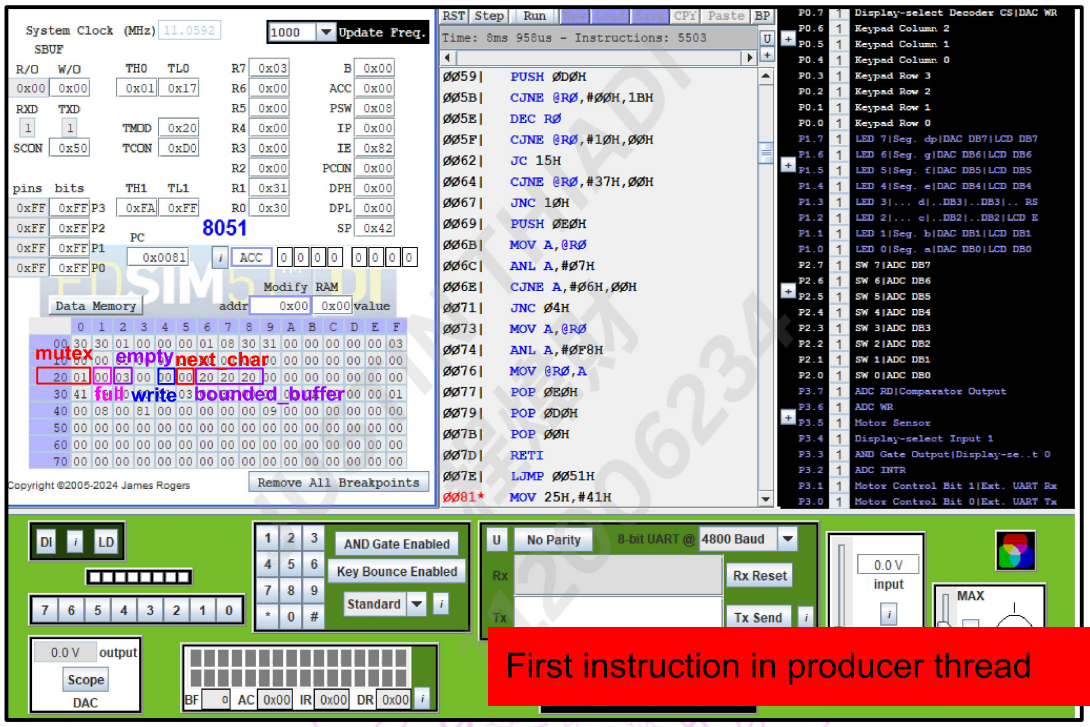
Task 2:

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
00000081	_Producer	testpreempt
000000BF	_Consumer	testpreempt
000000F9	_main	testpreempt
00000117	__sdcc_gsinit_startup	testpreempt
0000011B	__mcs51_genRAMCLEAR	testpreempt
0000011C	__mcs51_genXINIT	testpreempt
0000011D	__mcs51_genXRAMCLEAR	testpreempt
0000011E	_timer0_ISR	testpreempt
00000124	_Bootstrap	preemptive
0000014A	_ThreadCreate	preemptive
000001D0	_ThreadYield	preemptive
0000022A	_ThreadExit	preemptive
00000289	_myTimer0Handler	preemptive

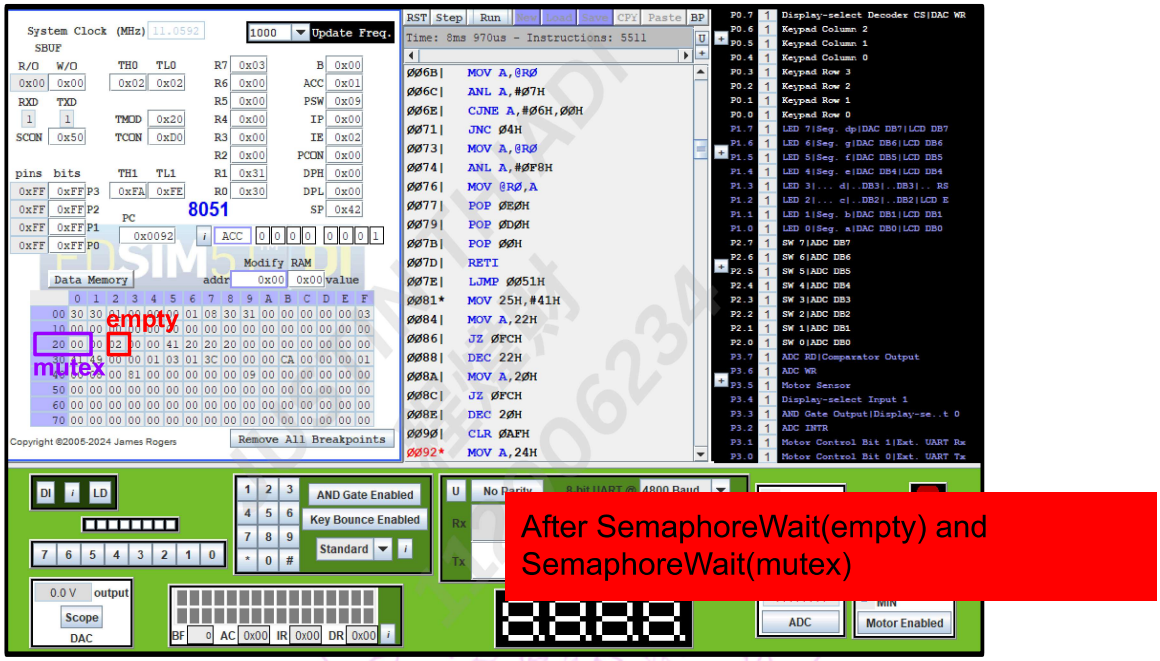
Value	Global	Global Defined In Module
00000000	.__.ABS.	
00000020	_mutex	testpreempt
00000021	_full	testpreempt
00000022	_empty	testpreempt
00000023	_read	testpreempt
00000024	_write	testpreempt
00000025	_next_char	testpreempt
00000026	_bounded_buffer	testpreempt
00000030	_saved_SP	preemptive
00000034	_current_thread	preemptive
00000035	_valid_thread	preemptive
00000036	_new_thread	preemptive
00000037	_old_SP	preemptive

- Take screenshots when the Producer is running and show semaphore changes.



- The Producer is checking if there's space in the buffer
- Notice that at this point, the value in bounded in buffer is 0x20, indicating space in ASCII, which is consistent with the code
- *empty* is a **counting semaphore** initialized to 3 (as buffer has 3 slots)
- → meaning all buffer slots are available

- The Producer must **wait (block)** if *empty* == *0*, but here *empty* = *3*, so it proceeds. In other words, before the Producer puts anything into the buffer, it must **claim a slot** by decrementing empty. If *empty* == *0*, it would block.
    - Here, the Producer is **about to** decrement empty, to indicate that **1 slot will be reserved** for producing



- Writes 'A' into buffer at index write= 0
  - next_char = 'A' (0x41)
  - Producer stores 'A' at bounded_buffer[write], which is 0x26
  - write++ happens next

- **Semaphore State**:
  - *empty* already decreased to 2 → since the count of available empty slots decreases
  - The *semaphoreWait(mutex)* locks the critical section by taking the mutex (value becomes 0).
  - *full* is not yet incremented

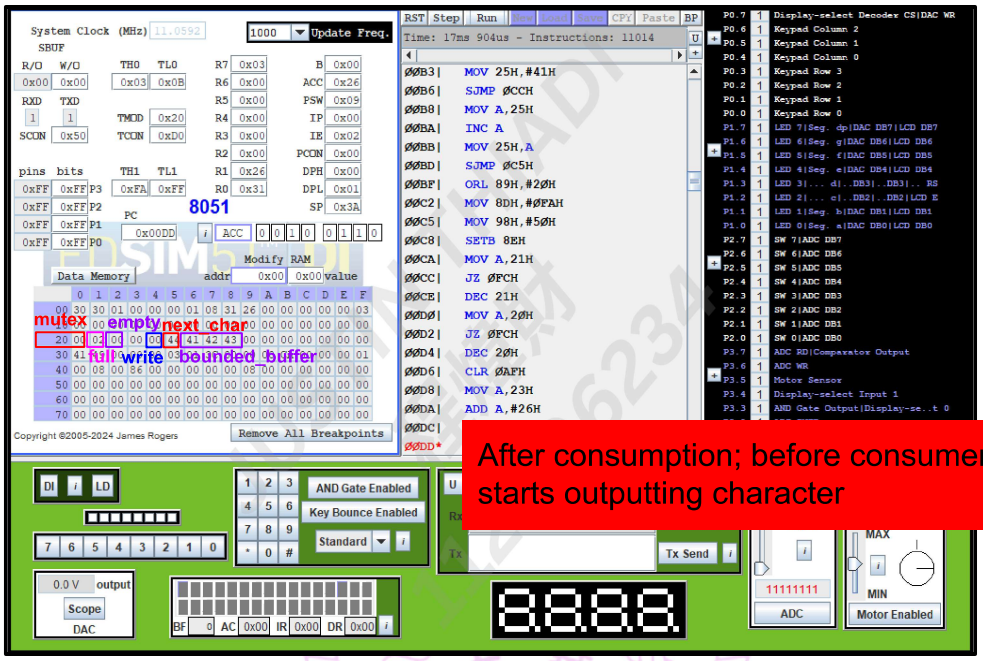
This happens as in the code, the producer is inside the critical section. It has claimed the slot (*empty*--) and is now writing 'A'. If we consider a step after the screenshot above, we can say that there are no semaphore changes here — just using the slot it reserved earlier.

- Updates *next_char* = 'B'
- Here, producer finishes the critical section:
- Buffer has 'A' at [0], *write* = 1
- *next_char*++ from 'A' → 'B'
- About to **signal** that **new data** is **ready**

Semaphore State:
After critical section:

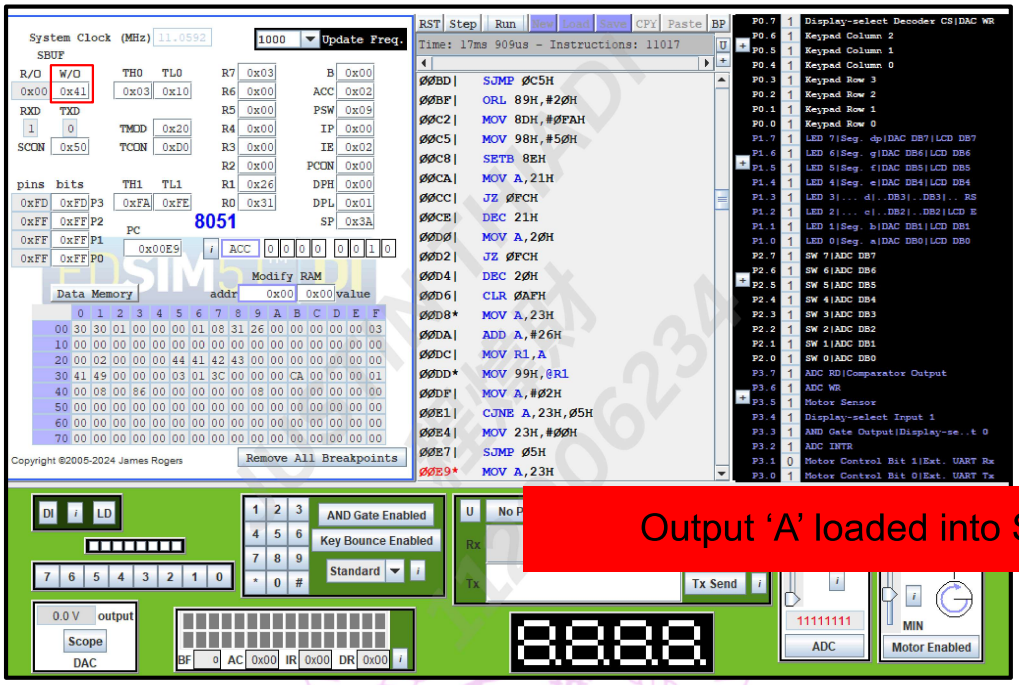
- *mutex* is released
- *full*++, making *full* = *1* a step after this, indicating 1 full slot in the buffer
- Once the Producer is done with writing to the buffer and updating internal state (like write and next_char), it signals that one more item is available → SemaphoreSignal(full);

- Take screenshots when the Consumer is running and show semaphore changes

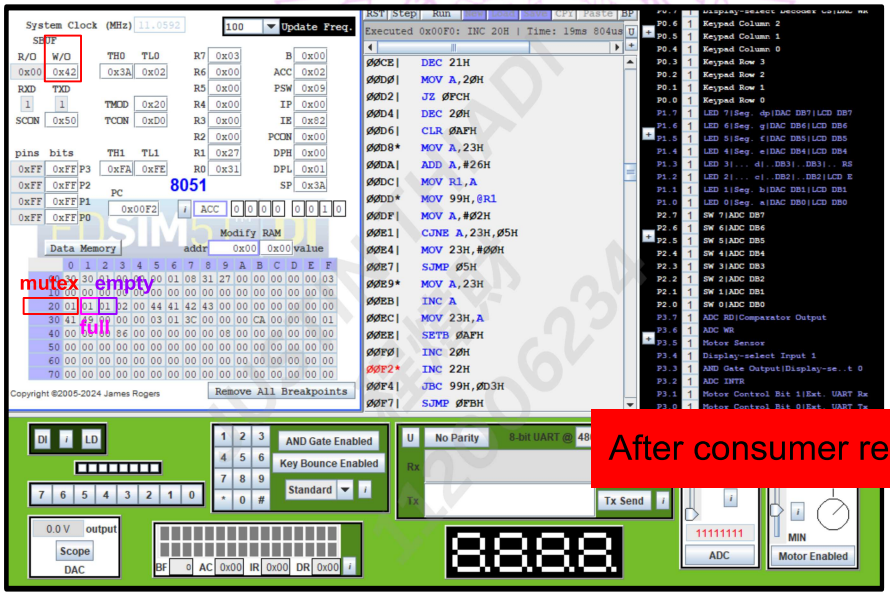


- Consumer reads 'A' from shared_buffer[read]
- Consumer has passed wait(full) (so full--)
- Now inside critical section
- Reads value from buffer at *read*= 0, which is 'A'

- Semaphore State:
  - *full = 2* → it was 3, now decreased due to SemaphoreWait(full)
  - *empty* still 0 (until later)
  - *mutex* currently held by Consumer

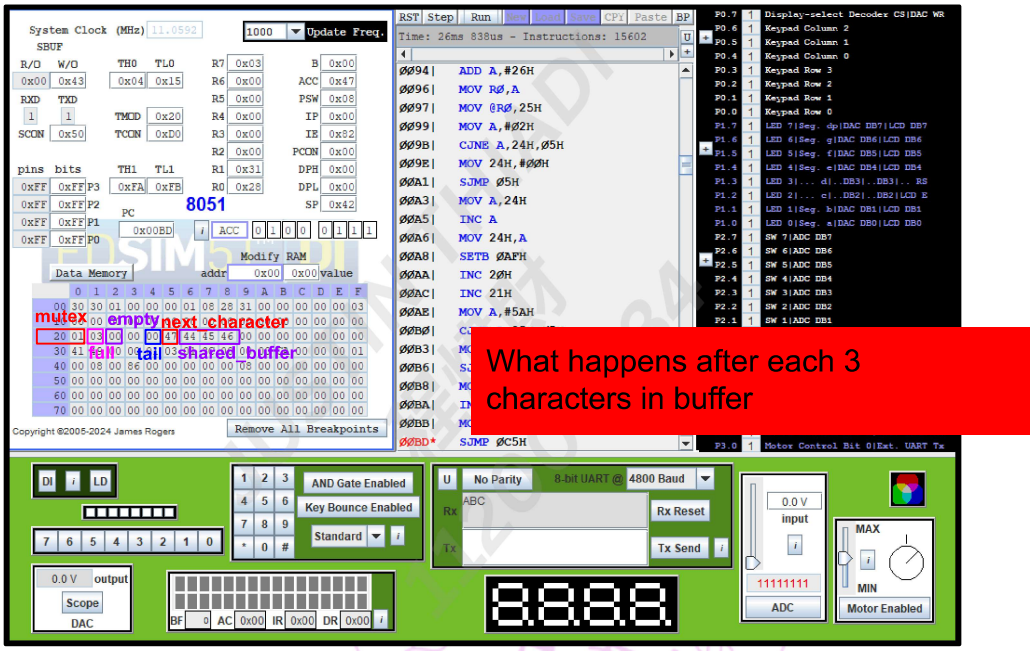


'A' has been written to SBUF



Semaphore State:

- *mutex*++ → releasing critical section, since after consuming ‘A’, it exits critical section
- *empty*++ → one slot is now free, since one slot is available again after ‘A’ consumed
- *full*– → same logic as why empty++



- Producer prepared next character 'D'
- Buffer is full: 'B' at [1], 'C' at [2]
- write wrapped around to 0
- next_char = 'D'

- **Semaphore State:**
  - empty = 0 → buffer is full, can't produce more
  - full = 3
  - Producer would now block on wait(empty) if it continues

Producer has filled the entire buffer with 'A', 'B', 'C', so now, it's preparing 'D' but **cannot write** unless Consumer has consumed something. On the next loop, it will **block** at *SemaphoreWait(empty)*, explaining *empty == 0*.