Report Checkpoint 1

Jonathan Sutedjo 鄭安良 111006207

1. testcoop.c: main + test case for cooperative multithreading

```
__data __at (0x36) char SharedBuffer;
__data __at (0x37) int Buffer_Availability;
```

For this part, I initialize 2 global variables where the SharedBuffer is used to store the char value of ‘A’ to ‘Z’ and the Buffer_availability is used to tell if the producer has produced new character.

```
/* [TODO for this function]
 * the producer in this test program generates one characters at a
 * time from 'A' to 'Z' and starts from 'A' again. The shared buffer
 * must be empty in order for the Producer to write.
 */
void Producer(void)
{
    /*
     * [TODO]
     * initialize producer data structure, and then enter
     * an infinite loop (does not return)
     */
    SharedBuffer = 'A';
    while (1)
    {
        /* [TODO]
         * wait for the buffer to be available,
         * and then write the new data into the buffer */
        if(Buffer_Availability){
            ThreadYield();
        }
        else{
            Buffer_Availability += 1;
            ThreadYield();
            if(SharedBuffer == 'Z'){
                SharedBuffer = 'A';
            }
            else{
                SharedBuffer += 1;
            }
        }
    }
}
```

In the producer part, it is responsible for generating the characters in a sequential order from ‘A’ to ‘Z’ and storing it in the shared buffer. This will run in an infinite loop so that the buffer is empty before writing a new character. When the buffer is unavailable, it will yield execution to other thread using ThreadYield(). When it reach ‘Z’, I make it so that the producer cycles back to ‘A’.

```
/* [TODO for this function]
 * the consumer in this test program gets the next item from
 * the queue and consume it and writes it to the serial port.
 * The Consumer also does not return.
 */
void Consumer(void)
{
    /*
     * [TODO]
     * initialize Tx for polling
     */
    TMOD = 0x20;
    TH1 = (char)-6;
    SCON = 0x50;
    TR1 = 1;
    TI = 1;

    while (1)
    {
        /*
         * [TODO]
         * wait for new data from producer
         * write data to serial port Tx,
         * poll for Tx to finish writing (TI),
         * then clear the flag
         */

        while (Buffer_Availability == 0) {
            ThreadYield();
        }

        while (!TI){
        }
        SBUF = SharedBuffer;
        TI = 0;
        Buffer_Availability -= 1;
        ThreadYield();
    }
}
```

In the consumer function, it continues to retrieve character from the shared buffer and writes them to the serial port. First, I initialize the serial port, then if the buffer is empty, the consumer yields control to other threads. After sending a character, it decreases the buffer availability and yields execution. The function operates in an infinite loop.

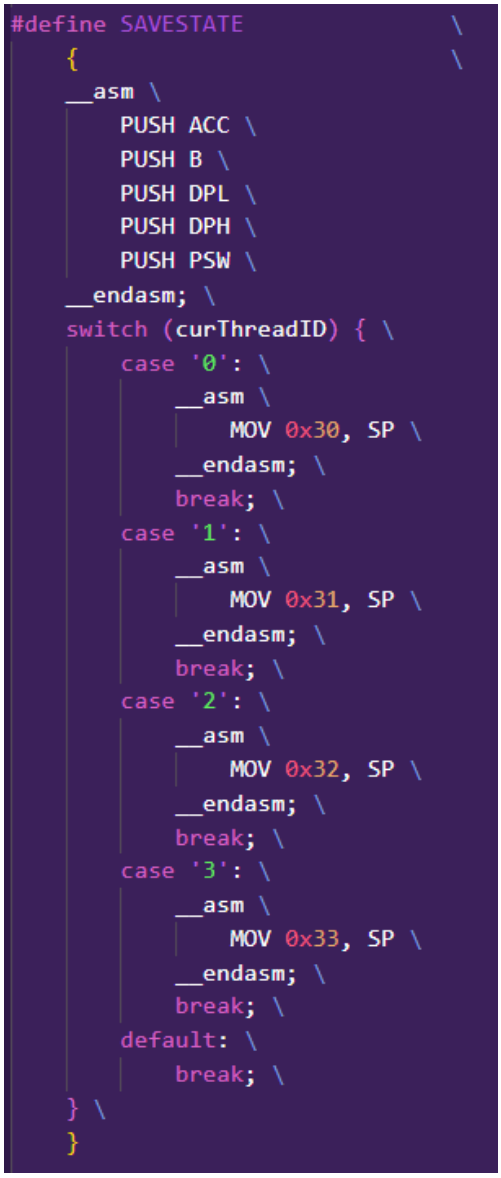
```c
void main(void)
{
    /*
     * [TODO]
     * initialize globals
     */

    SharedBuffer = ' ';
    Buffer_Availability = 0;

    /*
     * [TODO]
     * set up Producer and Consumer.
     * Because both are infinite loops, there is no loop
     * in this function and no return.
     */
    ThreadCreate(Producer);
    Consumer();
}

void _sdcc_gsinit_startup(void)
{
    __asm
        LJMP _Bootstrap
    __endasm;
}
```

In the main function I initialize global variable like the shared buffer and buffer availability. It is use as a bootstrap thread, so that it create the producer thread first then it create the consumer thread.

2. Cooperative Thread

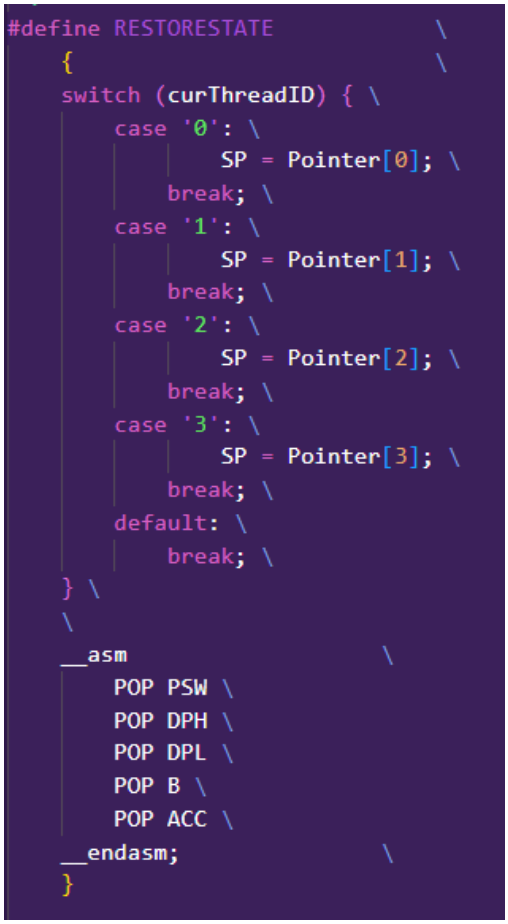
```c
__data __at (0x30) char Pointer[MAXTHREADS];
__data __at (0x3C) ThreadID newThreadID;
__data __at (0x34) int ValidBitMap;

__data __at (0x21) ThreadID curThreadID;
__data __at (0x22) char tempSP;
__data __at (0x23) char newSP;
```

In the cooperative.c file, I declare static global variables for various purposes. The first is an array to store the starting SP addresses for each thread. Two ThreadID variables are used: one for the currently running thread (curThreadID) and another for the newly assigned thread (newThreadID). The curThreadID ranges from '0' to '3', representing each thread. The ValidBitMap is a 4-bit map that tracks the availability of each thread. The tempSP variable temporarily holds the current SP value before creating a new thread, while newSP specifies the starting address for the newly assigned thread.



```
#define SAVESTATE                \
    {                            \
    __asm \
        PUSH ACC \
        PUSH B \
        PUSH DPL \
        PUSH DPH \
        PUSH PSW \
    __endasm; \
    switch (curThreadID) { \
        case '0': \
            __asm \
                MOV 0x30, SP \
            __endasm; \
            break; \
        case '1': \
            __asm \
                MOV 0x31, SP \
            __endasm; \
            break; \
        case '2': \
            __asm \
                MOV 0x32, SP \
            __endasm; \
            break; \
        case '3': \
            __asm \
                MOV 0x33, SP \
            __endasm; \
            break; \
        default: \
            break; \
    } \
    }
```

The SAVESTATE saves the state of the current thread by pushing the key CPU registers (ACC, B, DPL, DPH, and PSW) onto the stack and storing the stack pointer into the array indexed by the current thread ID.



```
#define RESTORESTATE                \
    {                               \
    switch (curThreadID) { \
        case '0': \
                SP = Pointer[0]; \
            break; \
        case '1': \
                SP = Pointer[1]; \
            break; \
        case '2': \
                SP = Pointer[2]; \
            break; \
        case '3': \
                SP = Pointer[3]; \
            break; \
        default: \
            break; \
    } \
    \
    __asm                       \
        POP PSW \
        POP DPH \
        POP DPL \
        POP B \
        POP ACC \
    __endasm;                   \
    }
```

The RESTORESTATE restores the state of the current thread by loading the saved stack pointer from the array and popping the key CPU registers (PSW, DPL, DPH, B, and ACC) from the stack.

```
void Bootstrap(void)
{
    /*
     * [TODO]
     * initialize data structures for threads (e.g., mask)
     *
     * optional: move the stack pointer to some known location
     * only during bootstrapping. by default, SP is 0x07.
     *
     * [TODO]
     *     create a thread for main; be sure current thread is
     *     set to this thread ID, and restore its context,
     *     so that it starts running main().
     */

    ValidBitMap = 0b0000;
    Pointer[0] = 0x3F;
    Pointer[1] = 0x4F;
    Pointer[2] = 0x5F;
    Pointer[3] = 0x6F;
    curThreadID = ThreadCreate(main);
    RESTORESTATE;
}
```

For the bootstrap function, I initialize the thread valid bitmap and then save the stack pointers for each thread. After that I creates a thread for the main() function, setting it as the current thread, and restores its context to begin execution. This function is use as the starting point for the cooperative multitasking system so that the data structures needed are ready and the main thread is initialized correctly.

```
if ((ValidBitMap & 0b1111) == 0b1111) {
    return -1;
}

if ((ValidBitMap & 0b0001) == 0b0000) {
newThreadID = '0';
ValidBitMap |= 0b0001;
newSP =  Pointer[0];
}
else if ((ValidBitMap & 0b0010) == 0b0000) {
newThreadID = '1';
ValidBitMap |= 0b0010;
newSP =  Pointer[1];
}
else if ((ValidBitMap & 0b1000) == 0b0000) {
newThreadID = '2';
ValidBitMap |= 0b0100;
newSP =  Pointer[2];
}
else if ((ValidBitMap & 0b1000) == 0b0000) {
newThreadID = '3';
ValidBitMap |= 0b1000;
newSP =  Pointer[3];
}

__asm
MOV 0x22, SP
MOV SP, 0x23
PUSH DPL
PUSH DPH
MOV A, 0x00
PUSH A
PUSH A
PUSH A
PUSH A
__endasm;
```

```
    switch (newThreadID) {
    case '0':
            PSW = 0b00000000;
        __asm
            PUSH PSW
            MOV 0x30, SP
        __endasm;
        break;
    case '1':
            PSW = 0b00001000;
        __asm
            PUSH PSW
            MOV 0x31, SP
        __endasm;
        break;
    case '2':
        PSW = 0b00010000;
        __asm
            PUSH PSW
            MOV 0x32, SP
        __endasm;
        break;
    case '3':
        PSW = 0b00011000;
        __asm
            PUSH PSW
            MOV 0x33, SP
        __endasm;
        break;
    default:
        break;
    }

    SP = tempSP;

    return newThreadID;
}
```

ThreadCreate function here first creates a new thread by assigning an available thread ID from the bitmap, updating the bitmap to mark the thread as in use, and setting up its stack pointer. It saves the current stack pointer temporarily, sets the new stack pointer for the thread, and initializes the thread's stack with default values and the function pointer fp as the return address. After initializing the thread, it restores the previous stack pointer and returns the newly created thread ID. If no more threads can be created, it returns -1. So here we first check if there are any empty threads by verifying that the ValidBitmap is not equal to 0b1111, indicating at least one thread is available. Each bit in the threadBitmap represents the state of a thread, with the first bit corresponding to the first thread, and so on. I assign the first available thread to newThreadID and update the threadBitmap using a bitwise OR operation to mark the thread as occupied. Next, I save the current stack pointer (SP) into tempSP and switch to the new thread's stack pointer (newSP). Then I initialize the stack of the new thread by pushing zeros for the ACC, B, DPL, and DPH registers. Finally, I set the PSW register according to the thread's ID, push it onto the stack, save the stack pointer to the shared Pointer array, and restore the previous SP value to maintain continuity.

```
void ThreadYield(void)
{
    SAVESTATE;
    do
    {
        /*
         * [TODO]
         * do round-robin policy for now.
         * find the next thread that can run and
         * set the current thread ID to it,
         * so that it can be restored (by the last line of
         * this function).
         * there should be at least one thread, so this loop
         * will always terminate.
         */
        curThreadID = (curThreadID == '3') ? '0' : curThreadID + 1;

        // Check if the thread is valid using a switch statement
        switch (curThreadID) {
            case '0':
                if ((ValidBitMap & 0b0001) == 0b0001) {
                    break; // Exit loop if thread 0 is runnable
                }
                continue; // Move to the next thread
            case '1':
                if ((ValidBitMap & 0b0010) == 0b0010) {
                    break; // Exit loop if thread 1 is runnable
                }
                continue;
            case '2':
                if ((ValidBitMap & 0b0100) == 0b0100) {
                    break; // Exit loop if thread 2 is runnable
                }
                continue;
            case '3':
                if ((ValidBitMap & 0b1000) == 0b1000) {
                    break; // Exit loop if thread 3 is runnable
                }
                continue;
        }
        break;

    } while (1);
    RESTORESTATE;
}
```

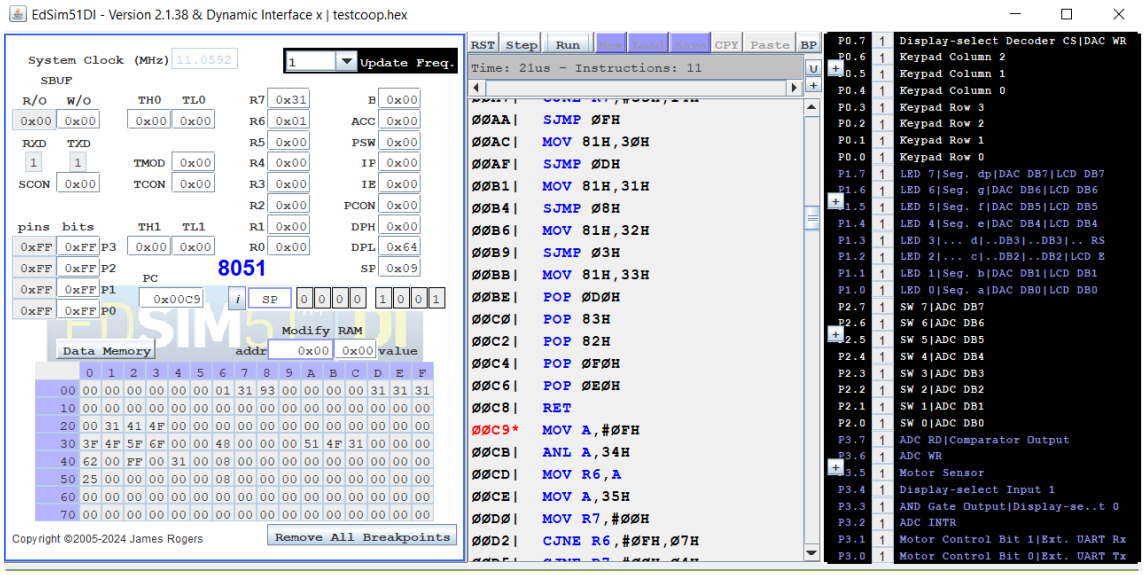
The ThreadYield function here save the current thread state and select the next runnable thread based on the bitmap. We loop through the available thread in order, then update the current thread ID to the next valid one, and also restoring the new thread state.

```c
void ThreadExit(void)
{
    /*
     * clear the bit for the current thread from the
     * bit mask, decrement thread count (if any),
     * and set current thread to another valid ID.
     * Q: What happens if there are no more valid threads?
     */
    switch (curThreadID) {
        case '0':
            ValidBitMap &= 0b1110; // Clear bit 0
            break;
        case '1':
            ValidBitMap &= 0b1101; // Clear bit 1
            break;
        case '2':
            ValidBitMap &= 0b1011; // Clear bit 2
            break;
        case '3':
            ValidBitMap &= 0b0111; // Clear bit 3
            break;
        default:
            return;
    }

    do {
        curThreadID = (curThreadID == '3') ? '0' : curThreadID + 1;

        switch (curThreadID) {
            case '0':
                if ((ValidBitMap & 0b0001) == 0b0001) {
                    break; // Exit loop if thread 0 is valid
                }
                continue;
            case '1':
                if ((ValidBitMap & 0b0010) == 0b0010) {
                    break; // Exit loop if thread 1 is valid
                }
                continue;
            case '2':
                if ((ValidBitMap & 0b0100) == 0b0100) {
                    break; // Exit loop if thread 2 is valid
                }
                continue;
            case '3':
                if ((ValidBitMap & 0b1000) == 0b1000) {
                    break; // Exit loop if thread 3 is valid
                }
                continue;
            default:
                break;
        }
        break;
    } while (1);

    RESTORESTATE;
}
```

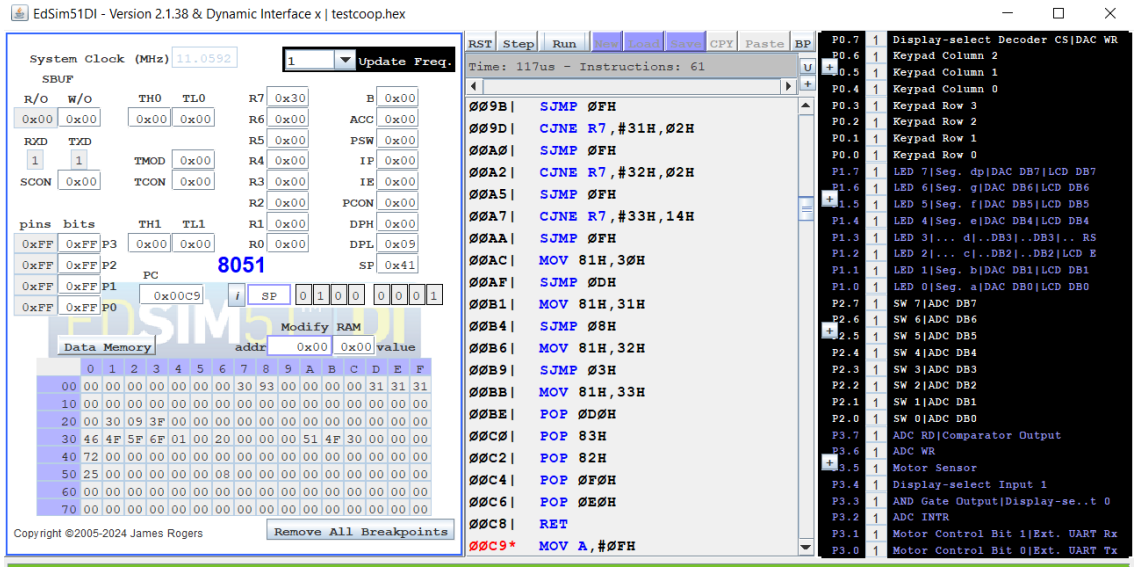
For the thread exit, it allows a thread to terminate by clearing its bit in the thread bitmap, marking it as inactive. It then finds the next valid thread and updates the current thread ID. After identifying the next thread, it restores the new thread's context and never returns, ensuring the termination to be done and switching to the next thread.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:    00000009  _Producer                      testcoop
C:    00000036  _Consumer                      testcoop
C:    00000064  _main                          testcoop
C:    00000075  __sdcc_gsinit_startup          testcoop
C:    00000079  __mcs51_genRAMCLEAR            testcoop
C:    0000007A  __mcs51_genXINIT               testcoop
C:    0000007B  __mcs51_genXRAMCLEAR           testcoop
C:    0000007C  _Bootstrap                     cooperative
C:    000000C9  _ThreadCreate                  cooperative
C:    00000179  _ThreadYield                   cooperative
C:    0000024E  _ThreadExit                    cooperative
```

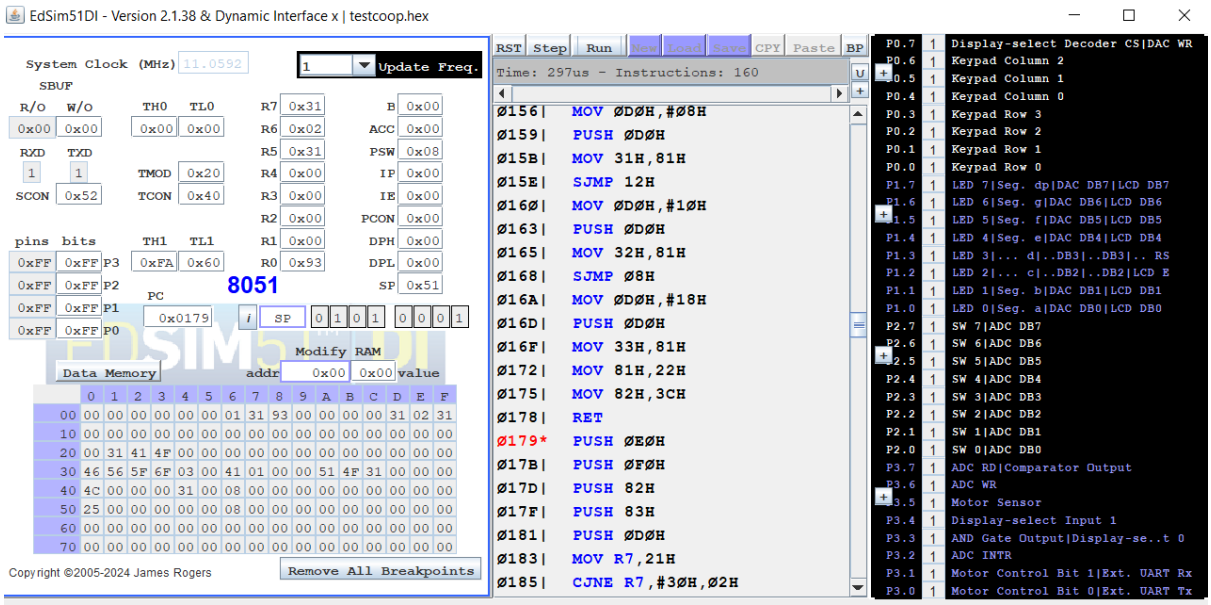
In the map file, it is listed that the _ThreadCreate is in the PC C9.



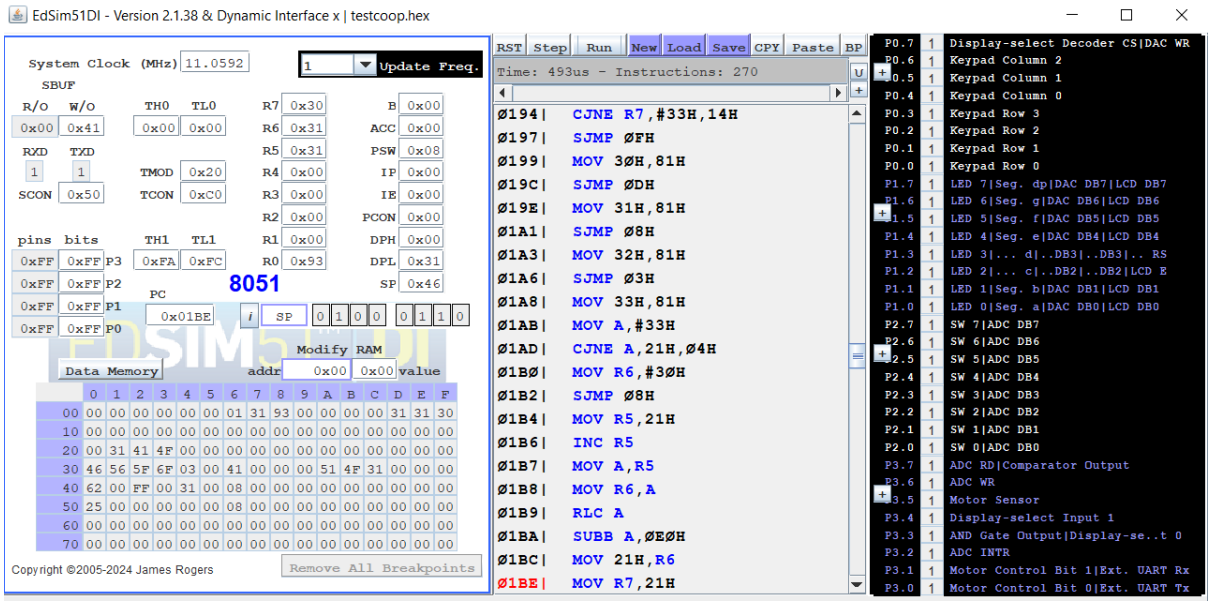
Here, above is the condition before we call the ThreadCreate for the main. We can see that the SP is now 0x09.



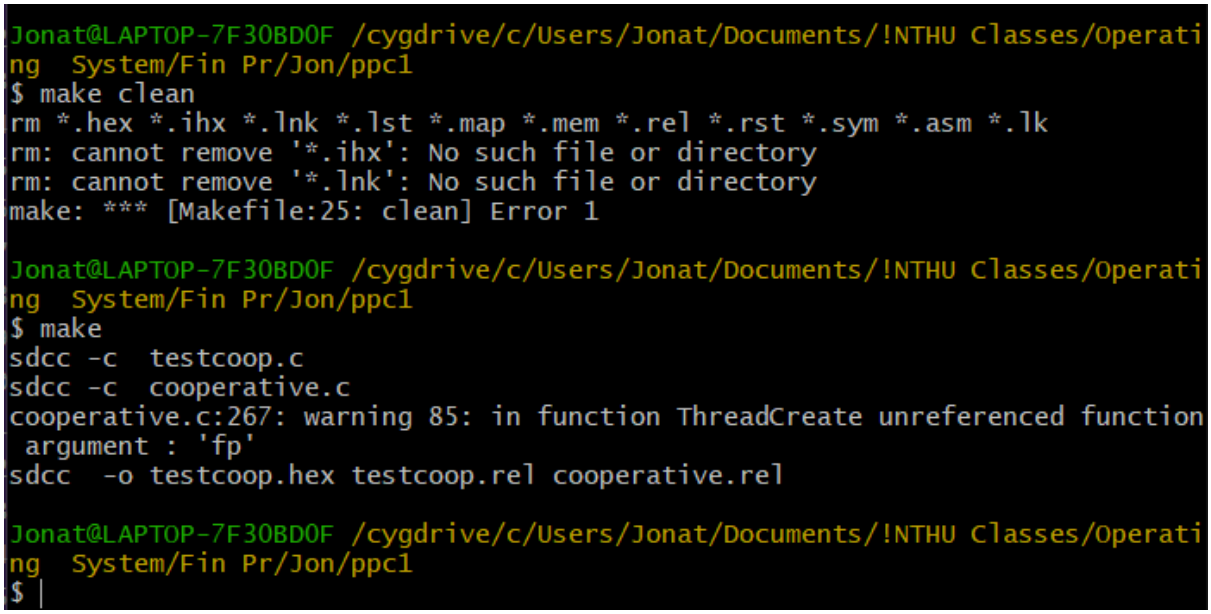
Then above is the condition before the ThreadCreate for the producer. The SP here is now 0x41, that means right now the current thread is the first thread.



This is a screenshot showing the producer running. We know the producer is active because the SP value is 0x51, which is within the range of 0x50-0x5F assigned to the producer thread. This confirms that the producer is currently running.



This is a screenshot showing the consumer running. We know the consumer is active because the SP value is 0x46, which is within the range of 0x40-0x4F assigned to the consumer thread. This confirms that the consumer is currently running.



This is when I use the commands make and make clean to compile the testcoop.c and cooperative.c files into testcoop.rel and cooperative.rel files, which are then linked and compiled to produce the testcoop.hex file.

The command make clean will delete all file from the make command.