

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.

```

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

```

__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 create a thread for the main() function, setting it as the current thread, and restore its context to begin execution. This function is used 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.


```

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;
        }
    } while (1);

    RESTORESTATE;
}

```

```

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;
    }
} while (1);

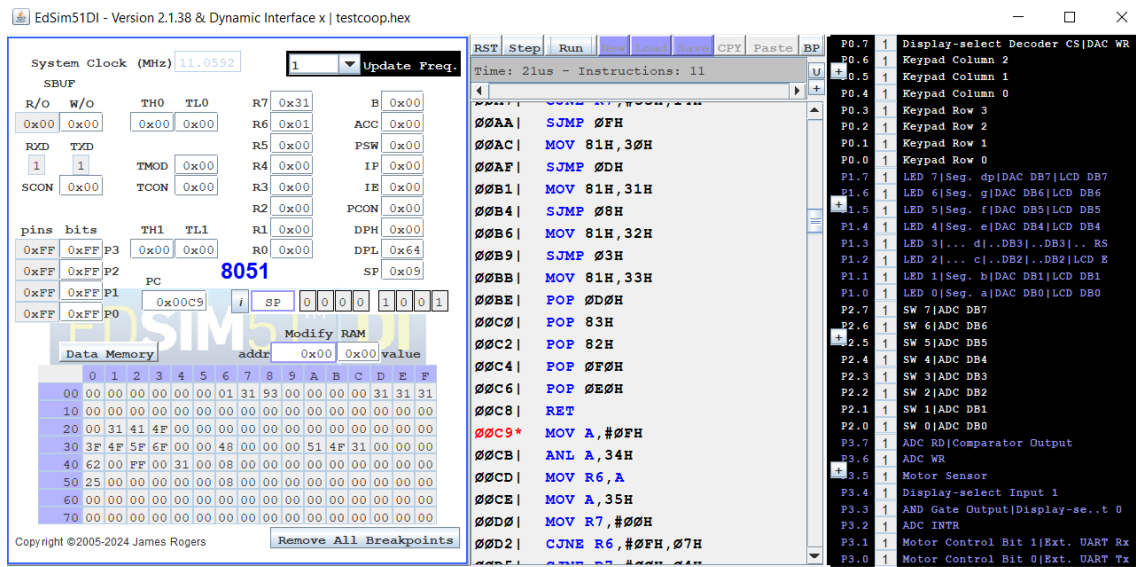
RESTORESTATE;
}

```

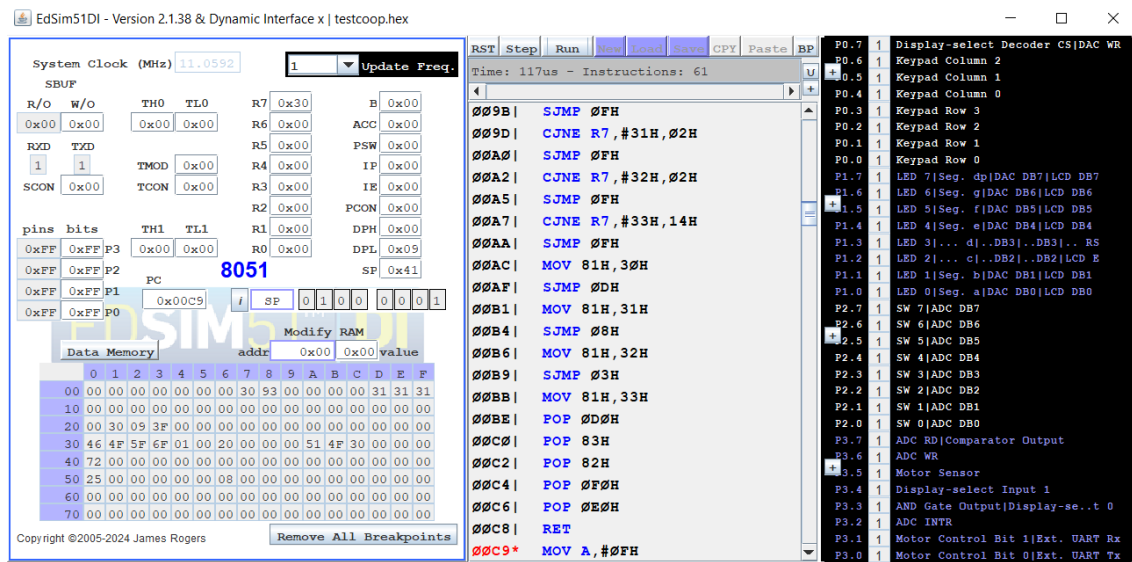
For the thread exit, it allows a thread to terminate by clearing its bit in the thread bitmap, marking it as inactive. It then finds the next valid thread and updates the current thread ID. After identifying the next thread, it restores the new thread's context and never returns, ensuring the termination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

EdSim51DI - Version 2.1.38 & Dynamic Interface x | testcoop.hex

System Clock (MHz) 11.0592 1 Update Freq.

SBUF

R/O	W/O	TH0	TL0	R7	0x31	B	0x00
0x00	0x00	0x00	0x00	R6	0x02	ACC	0x00
RxD	TxD	TMOD	0x20	R5	0x31	PSW	0x08
1	1	TCON	0x40	R4	0x00	IP	0x00
SCON	0x52	TH1	TL1	R3	0x00	IE	0x00
		PC	0x0179	R2	0x00	PCON	0x00
				R1	0x00	DPH	0x00
				R0	0x93	DPL	0x00
						SP	0x51

8051

Modify RAM

addr	0x00	0x00	value													
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	00	00	00	00	00	00	00	01	31	93	00	00	00	00	31	02
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	31	41	4F	00	00	00	00	00	00	00	00	00	00	00	00
30	46	56	5F	6F	03	00	41	01	00	00	51	4F	31	00	00	00
40	4C	00	00	00	31	00	08	00	00	00	00	00	00	00	00	00
50	25	00	00	00	00	00	08	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Copyright ©2005-2024 James Rogers Remove All Breakpoints

RST Step Run New Load Save CPY Paste BP

Time: 297us - Instructions: 160

```

0156 | MOV 0D0H,#08H
0159 | PUSH 0D0H
015B | MOV 31H,81H
015E | SJMP 12H
0160 | MOV 0D0H,#10H
0163 | PUSH 0D0H
0165 | MOV 32H,81H
0168 | SJMP 08H
016A | MOV 0D0H,#18H
016D | PUSH 0D0H
016F | MOV 33H,81H
0172 | MOV 81H,22H
0175 | MOV 82H,3CH
0178 | RET
0179* | PUSH 0E0H
017B | PUSH 0F0H
017D | PUSH 82H
017F | PUSH 83H
0181 | PUSH 0D0H
0183 | MOV R7,21H
0185 | CJNE R7,#30H,02H

```

P0.7 1 Display-select Decoder CS|DAC WR
P0.6 1 Keypad Column 2
P0.5 1 Keypad Column 1
P0.4 1 Keypad Column 0
P0.3 1 Keypad Row 3
P0.2 1 Keypad Row 2
P0.1 1 Keypad Row 1
P0.0 1 Keypad Row 0
P1.7 1 LED 7|Seg. dp|DAC DB7|LCD DB7
P1.6 1 LED 6|Seg. g|DAC DB6|LCD DB6
P1.5 1 LED 5|Seg. f|DAC DB5|LCD DB5
P1.4 1 LED 4|Seg. e|DAC DB4|LCD DB4
P1.3 1 LED 3|... d|..DB3|..DB3|.. RS
P1.2 1 LED 2|... c|..DB2|..DB2|LCD E
P1.1 1 LED 1|Seg. b|DAC DB1|LCD DB1
P1.0 1 LED 0|Seg. a|DAC DB0|LCD DB0
P2.7 1 SW 7|ADC DB7
P2.6 1 SW 6|ADC DB6
P2.5 1 SW 5|ADC DB5
P2.4 1 SW 4|ADC DB4
P2.3 1 SW 3|ADC DB3
P2.2 1 SW 2|ADC DB2
P2.1 1 SW 1|ADC DB1
P2.0 1 SW 0|ADC DB0
P3.7 1 ADC RD|Comparator Output
P3.6 1 ADC WR
P3.5 1 Motor Sensor
P3.4 1 Display-select Input 1
P3.3 1 AND Gate Output|Display-se..t 0
P3.2 1 ADC INTR
P3.1 1 Motor Control Bit 1|Ext. UART Rx
P3.0 1 Motor Control Bit 0|Ext. UART Tx

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.

EdSim51DI - Version 2.1.38 & Dynamic Interface x | testcoop.hex

System Clock (MHz) 11.0592 1 Update Freq.

SBUF

R/O	W/O	TH0	TL0	R7	0x30	B	0x00
0x00	0x41	0x00	0x00	R6	0x31	ACC	0x00
RxD	TxD	TMOD	0x20	R5	0x31	PSW	0x08
1	1	TCON	0xC0	R4	0x00	IP	0x00
SCON	0x50	TH1	TL1	R3	0x00	IE	0x00
		PC	0x01BE	R2	0x00	PCON	0x00
				R1	0x00	DPH	0x00
				R0	0x93	DPL	0x31
						SP	0x46

8051

Modify RAM

addr	0x00	0x00	value													
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	00	00	00	00	00	00	01	31	93	00	00	00	00	31	31	30
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	31	41	4F	00	00	00	00	00	00	00	00	00	00	00	00
30	46	56	5F	6F	03	00	41	00	00	00	51	4F	31	00	00	00
40	62	00	FF	00	31	00	08	00	00	00	00	00	00	00	00	00
50	25	00	00	00	00	00	08	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Copyright ©2005-2024 James Rogers Remove All Breakpoints

RST Step Run New Load Save CPY Paste BP

Time: 493us - Instructions: 270

```

0194 | CJNE R7,#33H,14H
0197 | SJMP 0FH
0199 | MOV 30H,81H
019C | SJMP 0DH
019E | MOV 31H,81H
01A1 | SJMP 08H
01A3 | MOV 32H,81H
01A6 | SJMP 03H
01A8 | MOV 33H,81H
01AB | MOV A,#33H
01AD | CJNE A,21H,04H
01B0 | MOV R6,#30H
01B2 | SJMP 08H
01B4 | MOV R5,21H
01B6 | INC R5
01B7 | MOV A,R5
01B8 | MOV R6,A
01B9 | RLC A
01BA | SUBB A,0E0H
01BC | MOV 21H,R6
01BE | MOV R7,21H

```

P0.7 1 Display-select Decoder CS|DAC WR
P0.6 1 Keypad Column 2
P0.5 1 Keypad Column 1
P0.4 1 Keypad Column 0
P0.3 1 Keypad Row 3
P0.2 1 Keypad Row 2
P0.1 1 Keypad Row 1
P0.0 1 Keypad Row 0
P1.7 1 LED 7|Seg. dp|DAC DB7|LCD DB7
P1.6 1 LED 6|Seg. g|DAC DB6|LCD DB6
P1.5 1 LED 5|Seg. f|DAC DB5|LCD DB5
P1.4 1 LED 4|Seg. e|DAC DB4|LCD DB4
P1.3 1 LED 3|... d|..DB3|..DB3|.. RS
P1.2 1 LED 2|... c|..DB2|..DB2|LCD E
P1.1 1 LED 1|Seg. b|DAC DB1|LCD DB1
P1.0 1 LED 0|Seg. a|DAC DB0|LCD DB0
P2.7 1 SW 7|ADC DB7
P2.6 1 SW 6|ADC DB6
P2.5 1 SW 5|ADC DB5
P2.4 1 SW 4|ADC DB4
P2.3 1 SW 3|ADC DB3
P2.2 1 SW 2|ADC DB2
P2.1 1 SW 1|ADC DB1
P2.0 1 SW 0|ADC DB0
P3.7 1 ADC RD|Comparator Output
P3.6 1 ADC WR
P3.5 1 Motor Sensor
P3.4 1 Display-select Input 1
P3.3 1 AND Gate Output|Display-se..t 0
P3.2 1 ADC INTR
P3.1 1 Motor Control Bit 1|Ext. UART Rx
P3.0 1 Motor Control Bit 0|Ext. UART Tx

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.

```

Jonat@LAPTOP-7F30BD0F /cygdrive/c/Users/Jonat/Documents/!NTHU Classes/Operati
ng System/Fin Pr/Jon/ppc1
$ make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym *.asm *.lk
rm: cannot remove '*.ihx': No such file or directory
rm: cannot remove '*.lnk': No such file or directory
make: *** [Makefile:25: clean] Error 1

Jonat@LAPTOP-7F30BD0F /cygdrive/c/Users/Jonat/Documents/!NTHU Classes/Operati
ng System/Fin Pr/Jon/ppc1
$ make
sdcc -c testcoop.c
sdcc -c cooperative.c
cooperative.c:267: warning 85: in function ThreadCreate unreferenced function
argument : 'fp'
sdcc -o testcoop.hex testcoop.rel cooperative.rel

Jonat@LAPTOP-7F30BD0F /cygdrive/c/Users/Jonat/Documents/!NTHU Classes/Operati
ng System/Fin Pr/Jon/ppc1
$ |

```

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.