OS Final Project Report – CheckPoint 4

106072237 Huang Tsai-Yin

1. Codes

// preemptive.h

```
#include <string.h>
#include <math.h>
#ifndef __PREEMPTIVE_H__
#define __PREEMPTIVE_H__

#define MAXTHREADS 4
#define CNAME(s) _ ## s // concatenate _ with S
#define name(s) s ## $ // concatenate $ with the value generated by the counter


#define SemaphoreCreate(s, n) \ // to assign value n to variable s
            { \
                s = n; \
            }

#define SemaphoreWait(s) \ // to wait till unlocked
   { \
         SemaphoreWaitBody(s, __COUNTER__); \
   }

#define SemaphoreWaitBody(S,c) \
            { \
                __asm \
   name(c): MOV _ACC, CNAME(S) \ // move the value in to the register
                JZ name(c) \ // go back to the loop if value is 0
                JB ACC.7, name(c) \ // go back to the loop if value is less than 0
                DEC    CNAME(S) \ // if the value is bigger than 0, decrease by one
                __endasm; \
            }

#define SemaphoreSignal(s) \
            { \
                __asm \
                    INC CNAME(s) \ // increase the value by one
```

```
            __endasm; \
        }

typedef char ThreadID;
typedef void (*FunctionPtr)(void);

ThreadID ThreadCreate(FunctionPtr);
void ThreadExit(void);
void myTimer0Handler(void);

#endif

// preemptive.c
#include <8051.h>
#include "preemptive.h"

__data __at (0x30) char MAX[0x04]; // to store the SP of each thread
__data __at (0x34) char THREADID; // current thread identification
__data __at (0x35) char THREADVALID = 0x00; // value to indicate which thread is used
__data __at (0x36) char Otid; // temporary variable used in ThreadCreate in order not to messed up
                                the thread identification of main
__data __at (0x37) char tmp_zero = 0x00; // the variable pushed into the thread when creating
__data __at (0x22) char change_produser; // the flag to indicate the producer produces

#define SAVESTATE   \
        {   \
            __asm   \
            PUSH ACC    \
            PUSH B  \
            PUSH DPL    \
            PUSH DPH    \
            PUSH PSW    \
            __endasm;   \
            MAX[THREADID] = 0x47 + THREADID * 0x10;   \// this is originally SP, but somehow the
                                                        value messed up during the
                                                        calculation, so I recorded the value it
                                                        should be and directly assigned to it.
        }
```

```
#define RESTORESTATE \
            {   \
              SP = 0x47 + THREADID * 0x10;    \// so is this.
              __asm \
                POP PSW \
                POP DPH \
                POP DPL \
                POP B \
                POP ACC \
              __endasm; \
              }

extern void main(void);

void Bootstrap(void) {
        THREADVALID = 0x00;
        SP = 0x07;

        TMOD = 0x00; // timer 0 mode 0
        IE = 0x82; // enable timer 0 interrupt,
        TR0 = 0x01; // start running timer0

        THREADID = ThreadCreate(main); // this thread will generate producer1 and producer2,
                                                after generating, itself will become consumer thread
        RESTORESTATE;
        return;
}

ThreadID ThreadCreate(FunctionPtr fp) {
        char SPtmp = SP; // to store the SP of main to prevent from messing up

        if (THREADVALID < 0x03){ // to see if there are any free threads
                                I originally use THREADVALID as a bit map to record which
                                thread is used. (i.e. THREADVALID is initialize to 0(0000), and if a
                                thread is generated, it will become 1(0001), and if another
                                thread is generated, it will be 3(0011), next thread will be
                                7(0111)).
```

However, I now use THREADVALID in another way which is that it is initialize to 0. If a thread is generated, it will become 1, if another thread is generated, it will become 2.
In this way, I no longer need to check THREADVALID to assign value to Otid individually. Also, I can update THREADVALID simply by adding one.
The reason why all my numbers are in hexadecimal is because I was originally all using decimal, but they sometimes don't work. After debugging with the TA, I found out that hexadecimal is the most reliable way to implement this project. Hence, I changed all my numbers into hexadecimal.

```
            Otid = THREADVALID; // THREADVLID is same with the new thread identification
            THREADVALID += 0x01; // meaning decrease one thread usable
        }
        else
            return -0x01; // if running out of threads, return -1
        SP = 0x40 + 0x10 * Otid; // I used to assign SP individually depends on the value of Otid.
                                 However, I found this way more efficient and less error when
                                 compiling to assembly
        PSW = 0x08 * Otid; // this is also changed due to the same reason above
        __asm// there should be a lot fractions of assembly in this function. However, I combined
                them to make it cleaner and more manageable.
            PUSH DPL
            PUSH DPH
            PUSH _tmp_zero
            PUSH _tmp_zero
            PUSH _tmp_zero
            PUSH _tmp_zero
            PUSH PSW
        __endasm;
        MAX[Otid] = SP; // save the current SP into the array
        SP = SPtmp; // restore the SP saved at the beginning of this function
        return Otid; // return the created thread's identification
}

void myTimer0Handler(void) {
          EA = 0x00; // disable the interrupt
          SAVESTATE;
```

```
        if(THREADID != 0x00){ // if current thread is for producing
          change_produser = THREADID; // record the current thread
          THREADID = 0x00; // and change the thread to consuming
        }
        else if (change_produser == 0x01) // if the current thread is for consuming, check what is
                                              the next producer, 0x01 is producer1 and 0x02 is
                                              producer2
          THREADID = 0x02;
        else
          THREADID = 0x01;
        RESTORESTATE;
        EA = 0x01; // enable the interrupt
        __asm
          reti
        __endasm;
}

void ThreadExit(void) {
        EA = 0x00;
        if(THREADID != Otid) // to change the current thread id into the one that is usable
           THREADID += 0x01;
        else
           THREADID = 0x00;
        THREADVALID -= 0x01; // meaning one more thread available
        RESTORESTATE;
        EA = 0x01;
}

// test3threads.c
#include <8051.h>
#include "preemptive.h"

__data __at (0x38) char i; // variable to store alphabet for producer1
__data __at (0x39) char j; // variable to store number for producer2
__data __at (0x3A) char mutex;
__data __at (0x3B) char full;
__data __at (0x3C) char empty;
__data __at (0x3D) char buffer[0x03];
```

```
__data __at (0x20) char front = 0x00; // since I implemented the three deep buffers as a cycled
                                      queue, it requires pointer to indicate the position of the
                                      data stored, front pointer is for consuming, back pointer is
                                      for producing, both of them increase when activating.
__data __at (0x21) char back = 0x00;
__data __at (0x23) char counter; // I implemented the project in a way that each producer will
                                 generate three character, so I use this counter to count the
                                 amount of character each one produced
__data __at (0x24) char flag_first; // When I completed this project, I found out that the output isn't
                                    the way I expected to be. I expected it to be like
                                    ABC012DEF345, instead, it is ABCABC012DEF345, so I use this
                                    variable to consume the character that I don't want.

void Producer1(void) {
        while (0x01) {
                SemaphoreWait(empty);
                SemaphoreWait(mutex);
                if(counter < 0x03){ // If producer1 generates less than three characters, it can
                                    continue
                  buffer[back] = i; // to store the value in the right position
                  counter += 0x01; // record the amount of character generated
                  if(back < 0x02) // update the pointer
                     back += 0x01;
                  else
                     back = 0x00;
                 if(i < 'Z') // update the character that is gong to be generate next
                     i += 0x01;
                  else
                     i = 'A';
                }
                SemaphoreSignal(mutex);
                SemaphoreSignal(full);
        }
}

void Producer2(void) { // the whole implementing way is similar to producer1
        while (0x01) {
                SemaphoreWait(empty);
```

```
                SemaphoreWait(mutex);
                if(counter < 0x06 && counter > 0x02){ // if counter is more than 2 and less than
                                                      6, producer2 can generate
                    buffer[back] = j;
                    counter += 0x01;
                    if(back < 0x02)
                        back += 0x01;
                    else
                        back = 0x00;
                    if(j < '9')
                        j += 0x01;
                    else
                        j = '0';
                    if(counter == 0x06) // if the counter equals to 6, reset the counter, so
                                            producer2 can no longer produce and producer1 can
                        counter = 0x00;
                }
                SemaphoreSignal(mutex);
                SemaphoreSignal(full);
        }
}

void Consumer(void) {
        TMOD |= 0x20;
        TH1 = 0xFA;
        SCON = 0x50;
        TR1 = 0x01;
        while (0x01) {
                SemaphoreWait(full);
                SemaphoreWait(mutex);
                if(flag_first > 0x02){ // to consume the character I don't want it to produce
                SBUF = buffer[front];
                if(front < 0x02)
                    front += 0x01;
                else
                    front = 0x00;
                while(TI == 0x00){}
                TI = 0x00;
```

```
                        }
                        else
                            flag_first += 0x01; // after three loops, the consumer can assign value to SBUF
                        SemaphoreSignal(mutex);
                        SemaphoreSignal(empty);
                }
}

void main(void) {
                SemaphoreCreate(mutex, 0x01); // initialize variables
                SemaphoreCreate(empty, 0x03);
                SemaphoreCreate(full, 0x00);
                i = 'A';
                j = '0';
                counter = 0x00;
                flag_first = 0x00;
                ThreadCreate(Producer1); // create producers
                ThreadCreate(Producer2);
                Consumer(); // change main thread into consumer thread
}

void _sdcc_gsinit_startup(void) {
        __asm
                        ljmp    _Bootstrap
        __endasm;
}

void _mcs51_genRAMCLEAR(void) {}
void _mcs51_genXINIT(void) {}
void _mcs51_genXRAMCLEAR(void) {}

void timer0_ISR(void) __interrupt(0x01) {
        __asm
                        ljmp _myTimer0Handler
        __endasm;
}
```

2. Answers in checkpoints

   What do you get when spawning threads indifferent order?

   When under the situation of not implementing any way of making the thread produce equally, and thread changing part in myTimer0Handler is like below

   If (THREADID < Otid)

   THREADID += 0x01;

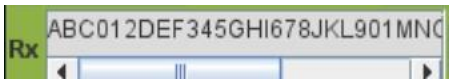
   else

   THREADID = 0x00

   The first spawned thread would keep generating and the other would starve.

   Show your fairness of producers and ways to implement

   My goal is to produce three characters by each thread, and below is my output



   The ways I implement it is to change myTimer0Handler into the logic that

   Is it a consumer

   -> yes -> change to the producer that is not the previous one

   -> no -> change to a consumer

   On the other hand, I also implement the producer to pass a function that producer1 can only produce when counter is less than three and the producer2 can only produce when the counter is less than six and bigger than two.

3. Screenshots

   0x34 THREADID

   0x3A mutex

   0x3B full

   0x3C empty

   Producer1

	0	1	2	3	4	5	6	7	8	9	A	B	C
00	6F	30	00	02	00	11	02	10	3F	31	00	00	00
10	3F	32	00	00	00	00	00	00	00	00	00	00	00
20	00	00	02	03	03	00	00	00	00	00	00	00	00
30	47	57	67	00	01	03	02	00	4A	34	01	03	00

   THREADID = 1

   Mutex = 1

   Full = 3

   Empty = 0

Producer2

	0	1	2	3	4	5	6	7	8	9	A	B	C
00	6F	30	00	02	00	11	02	20	3F	31	00	00	00
10	3F	32	00	00	00	00	00	36	00	00	00	00	00
20	00	00	01	00	03	00	00	00	00	00	00	00	00
30	47	57	67	00	02	03	02	00	4A	37	01	03	00

THREADID = 2

Mutex = 1

Full = 3

Empty = 0

Consumer

	0	1	2	3	4	5	6	7	8	9	A	B	C
00	6F	3D	00	02	00	11	02	00	3F	31	00	00	00
10	3F	32	00	00	00	00	00	00	00	00	00	00	00
20	01	00	02	00	03	00	00	00	00	00	00	00	00
30	47	57	67	00	00	03	02	00	47	34	00	02	00

THREADID = 0

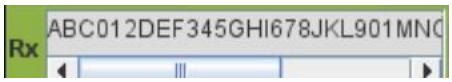
Mutex = 0

Full = 2

Empty = 0

UART output

Fair version



Unfair version