CS 342302 Operating Systems
Fall Semester 2024
Prof. Pai H. Chou

Programming Project Checkpoint 1

What to submit: One zip file named <studentID>-`ppc1.zip` (replace <studentID> with your own student ID). It should contain:

- one PDF file named **ppc1.pdf** for Part 3 of this checkpoint. It should explain how your code works. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID.
- (Parts 1 and 2) Turn in the source files for cooperative multithreading to be compiled using SDCC and target EdSim51.
    - `testcoop.c`, which contains the startup code and sets up the producer-consumer example.
    - `cooperative.c`, `cooperative.h`, which contains the cooperative multithreading code.
- (Part 3) Turn in the screenshots for compiling your code using the provided Makefile.
    - File named ppc1.pdf that includes the screenshots and explanations as instructed below.

For this programming project checkpoint, you are to write a cooperative multithreading package (as presented during week 6) and a test case based on the single-buffer producer-consumer example. Cooperative multithreading means the code for each thread needs to explicitly do a thread-yield (not to be confused with Python's generator-yield) in order to switch to another thread, if any.

For this assignment, we will use a relatively simple threads API, as defined in cooperative.h.

Data types

- `typedef char ThreadID;` where a `ThreadID` is a type for thread ID. It is represented by a single byte, which is the native word size for 8051.
- `typedef void (*FunctionPtr)(void);` a `FunctionPtr` is a type for a function pointer, for the purpose of attaching code to a thread. SDCC compiles it to a 2-byte pointer (assumed to be code space).

API

- `ThreadID ThreadCreate(FunctionPtr);` This is the primary API for creating (and starting) a thread, and return its thread ID. Once created, the thread is eligible to run.
- `void ThreadYield(void);` This is the API for a thread to yield its control to another thread, if any. Later, it can resume control immediately after the `ThreadYield()` statement.

- `void ThreadExit(void);` This is the code for a thread to exit itself either explicitly or implicitly so that the resources can be recycled for another thread. For now we will not use it, but we will use it in a later assignment.

This cooperative threads package makes the following assumptions:

- Use of the four register banks on 8051 for each of the four threads. `R0`-`R7` gets mapped to addresses 0x00-0x07, 0x08-0x0F, 0x10-0x17, 0x18-0x1F, depending on `PSW` bits <4:3>.
- Use of four 16-byte stacks at locations 0x40-0x4F, 0x50-0x5F, 0x60-0x6F, and 0x70-0x7F for threads 0-3.
- manually allocated memory in the range 0x20-0x3F

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

Write the code for testcoop.c, which sets up the startup code and defines `main()` to set up the producer and consumer. Here is the template for testcoop.c; Look for the `[TODO]` in the comments for the code to fill out.

```
C/C++
/*
 * file: testcoop.c
 */
#include <8051.h>
#include "cooperative.h"

/*
 * [TODO]
 * declare your global variables here, for the shared buffer
 * between the producer and consumer.
 * Hint: you may want to manually designate the location for the
 * variable.  you can use
 *        __data __at (0x30) type var;
 * to declare a variable var of the type
 */

/* [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)
     */
    while (1)
    {
        /* [TODO]
         * wait for the buffer to be available,
         * and then write the new data into the buffer */
    }
}

/* [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
     */
    while (1)
    {
        /*
         * [TODO]
         * wait for new data from producer
         */

        /*
         * [TODO]
         * write data to serial port Tx,
         * poll for Tx to finish writing (TI),
         * then clear the flag
         */
    }
}

/* [TODO for this function]
 * main() is started by the thread bootstrapper as thread-0.
 * It can create more thread(s) as needed:
 * one thread can act as producer and another as consumer.
 */
void main(void)
{
    /*
     * [TODO]
     * initialize globals
```

```
     */

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

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

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

2. cooperative.c

Write the code that implements the cooperative multithreading API above. In addition to the API functions, you need to define the thread bootstrapping function.

```C/C++
#include <8051.h>

#include "cooperative.h"

/*
 * [TODO]
 * declare the static globals here using
 *        __data __at (address) type name; syntax
 * manually allocate the addresses of these variables, for
 * - saved stack pointers (MAXTHREADS)
 * - current thread ID
 * - a bitmap for which thread ID is a valid thread;
 *   maybe also a count, but strictly speaking not necessary
 * - plus any temporaries that you need.
 */
```

```
/*
 * [TODO]
 * define a macro for saving the context of the current thread by
 * 1) push ACC, B register, Data pointer registers (DPL, DPH), PSW
 * 2) save SP into the saved Stack Pointers array
 *   as indexed by the current thread ID.
 * Note that 1) should be written in assembly,
 *     while 2) can be written in either assembly or C
 */
#define SAVESTATE               \
    {                           \
        __asm                   \
        /*your code here*/      \
        __endasm;               \
    }

/*
 * [TODO]
 * define a macro for restoring the context of the current thread by
 * essentially doing the reverse of SAVESTATE:
 * 1) assign SP to the saved SP from the saved stack pointer array
 * 2) pop the registers PSW, data pointer registers, B reg, and ACC
 * Again, popping must be done in assembly but restoring SP can be
 * done in either C or assembly.
 */
#define RESTORESTATE            \
    {                           \
        __asm                   \
        /*your code here*/      \
        __endasm;               \
    }

/*
 * we declare main() as an extern so we can reference its symbol
 * when creating a thread for it.
 */

extern void main(void);

/*
 * Bootstrap is jumped to by the startup code to make the thread for
 * main, and restore its context so the thread can run.
 */

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().
     */
}

/*
 * ThreadCreate() creates a thread data structure so it is ready
 * to be restored (context switched in).
 * The function pointer itself should take no argument and should
 * return no argument.
 */
ThreadID ThreadCreate(FunctionPtr fp)
{
    /*
     * [TODO]
     * check to see we have not reached the max #threads.
     * if so, return -1, which is not a valid thread ID.
     */
    /*
     * [TODO]
     *     otherwise, find a thread ID that is not in use,
     *     and grab it. (can check the bit mask for threads),
     *
     * [TODO] below
     * a. update the bit mask
          (and increment thread count, if you use a thread count,
           but it is optional)
       b. calculate the starting stack location for new thread
       c. save the current SP in a temporary
          set SP to the starting location for the new thread
       d. push the return address fp (2-byte parameter to
          ThreadCreate) onto stack so it can be the return
          address to resume the thread. Note that in SDCC
          convention, 2-byte ptr is passed in DPTR.  but
          push instruction can only push it as two separate
          registers, DPL and DPH.
       e. we want to initialize the registers to 0, so we
          assign a register to 0 and push it four times
          for ACC, B, DPL, DPH.  Note: push #0 will not work
          because push takes only direct address as its operand,
          but it does not take an immediate (literal) operand.
```

```
       f. finally, we need to push PSW (processor status word)
          register, which consist of bits
           CY AC F0 RS1 RS0 OV UD P
          all bits can be initialized to zero, except <RS1:RS0>
          which selects the register bank.
          Thread 0 uses bank 0, Thread 1 uses bank 1, etc.
          Setting the bits to 00B, 01B, 10B, 11B will select
          the register bank so no need to push/pop registers
          R0-R7.  So, set PSW to
          00000000B for thread 0, 00001000B for thread 1,
          00010000B for thread 2, 00011000B for thread 3.
       g. write the current stack pointer to the saved stack
          pointer array for this newly created thread ID
       h. set SP to the saved SP in step c.
       i. finally, return the newly created thread ID.
     */
}

/*
 * this is called by a running thread to yield control to another
 * thread.  ThreadYield() saves the context of the current
 * running thread, picks another thread (and set the current thread
 * ID to it), if any, and then restores its state.
 */

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.
         */
    } while (1);
    RESTORESTATE;
}

/*
 * ThreadExit() is called by the thread's own code to terminate
 * itself.  It will never return; instead, it switches context
 * to another thread.
```

```
 */
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?
     */
    RESTORESTATE;
}
```

3. screenshots

3.1 Screenshots for compilation

Turn in a screenshot showing compilation of your code using the provided `Makefile`. 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 `testcoop.*` 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

3.2 Screenshots and explanation

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

- Take one screenshot before each `ThreadCreate` call. Explain how the stack changes.
- Take one screenshot when the `Producer` is running. How do you know?
- Take one screenshot when the `Consumer` is running. How do you know?

Grading Policy:

- Parts 1 and 2: (70%) Make sure your code can be compiled and executed successfully; otherwise, you will receive 0 points. No partial credit will be given.

- Part 3: (30%) Be sure to include all the required screenshots and explanations in your PDF; otherwise, points may be deducted at the TA's discretion.