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

Programming Project Checkpoint 2

What to submit: One zip file named <studentID>-`ppc2.zip` (replace <studentID> with your own student ID). It should contain:

- one PDF file named **ppc2.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 preemptive multithreading to be compiled using SDCC and targets EdSim51.
    - `testpreempt.c`, which contains the startup code and sets up the producer-consumer example.
    - `preemptive.c`, `preemptive.h`, which contains the cooperative multithreading code.
- (Part 3) Turn in the screenshots for compiling your code using your own modified Makefile (based on CheckPoint 1's Makefile)
    - File named `ppc2.pdf` that includes the screenshots and explanations as instructed below.

For this programming project checkpoint, you are to write a **preemptive** multithreading package and a test case based on the single-buffer producer-consumer example. Preemptive multithreading means the code for each thread does not need to explicitly do a thread-yield (not to be confused with Python's generator-`yield`) in order to switch to another thread, if any. Of course, it can also do `ThreadYield()` if it wants.

Your preemptive.h should be mostly the same as your `cooperative.h`, except any macro names involving `__COOPERATIVE_H__` should be replaced with `__PREEMPTIVE_H__`. Also, depending on your implementation, you may or may not use a separate thread for the thread manager. If so, then you may want to define your `MAXTHREADS` accordingly. All other API should be the same. Actually, the `ThreadYield()` function should be available for preemptive threads, too, because even though the OS can always preempt a thread, a thread should still be able to voluntarily give up control.

1. [25 points] `testpreempt.c`: main + test case for preemptive multithreading

Write the code for `testpreempt.c`, which sets up the startup code and defines `main()` to set up the producer and consumer. It can be based on `testcoop.c` from your CheckPoint 1.

1.1 Producer and Consumer

These should be mostly the same as those from `testcoop.c`. The differences are

- They should not call `ThreadYield()`. Instead, rely on the preemptive mechanism to context switch.
- where they access the shared variables, SDCC suggests that you can surround the code fragment using the `__critical { }` construct, to ensure that the two shared vars are accessed atomically. It can be like

  ```
  __critical {
        // your code to be executed atomically
  }
  ```

  but you should to examine the assembly output to see if it does what you expect. If not, then you may need to temporarily disable interrupts by

  `EA = 0;`

  and reenable interrupts by

  `EA = 1;`

- In your `Consumer()` code, where it sets `TMOD`, you might want to do
  - `TMOD |= 0x20;`

  instead of `TMOD = 0x20`, because `TMOD` is also assigned by the (modified) `Bootstrap` code to set up the timer interrupt in timer-0 for preemption. This way, it preserves the `Bootstrap` code's setting.

1.2 ISR

As explained in lecture, the ISR should be defined in the same file as `main()` in order for SDCC to generate the proper code for ISR. So, include the following lines at the bottom of your `testpreempt.c`:

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

void timer0_ISR(void) __interrupt(1) {
        __asm
                ljmp _myTimer0Handler
        __endasm;
}
```

This allows the ISR to call your routine named `myTimer0Handler` (defined in `preemptive.c`) to handle the actual interrupt itself.

2. [55 points] `preemptive.c`

Your `preemptive.c` can be based on `cooperative.c` that you did for CheckPoint1. Make sure the cooperative version works properly before attempting this part. In addition, you may also want to see these slides for more details on the timer and its use in preemptive scheduling.

2.1 SAVESTATE and RESTORESTATE

Check to make sure your assembly code saves registers before trashing them. It can be the same as the cooperative version.

2.2 Bootstrap()

In `Bootstrap`, you can set up Timer 0 to cause preemption. (Timer 1 is already used by UART). If you want to use Timer 0 in mode 0, you can add this code

```
    TMOD = 0;  // timer 0 mode 0
    IE = 0x82;  // enable timer 0 interrupt; keep consumer polling
                // EA  -  ET2  ES  ET1  EX1  ET0  EX0
    TR0 = 1; // set bit TR0 to start running timer 0
```

before you create the initial thread and context switch to it.

2.3 ThreadCreate(), ThreadYield(), ThreadExit()

These functions are largely the same as the cooperative version, except we want to make sure it is executed atomically. To do this, you could try using the `__critical` construct just before the `{ }` for the entire function's body and check if SDCC generates the code you expect; or you may set and clear `EA` bit before and after the function body code.

2.4 myTimer0Handler()

This is the new routine you need to write to be the ISR for Timer0, which serves the purpose of preemption. A straightforward implementation would be to copy the code for `ThreadYield()` but instead of relying on the compiler to generate `RET` instruction to return to a function (or subroutine) call, you need to put in the `RETI` assembly instruction to return from the interrupt (after all, it is invoked as an interrupt service routine).

However, depending on how you write the `ThreadYield()`, if you write parts of it in C between `SAVESTATE` and `RESTORESTATE`, it is likely to use registers (especially `R0` and `R1` if it needs to use pointers to IRAM). Because it only saves the bank numbers but not copying the register values, any code that modifies `R0` - `R7` will trash them and their values cannot be restored by `RESTORESTATE`.

One solution is for you to insert code to preserve the value of any such registers by copying them to registers that have been saved (e.g., `B`, `DPH`, `DPL`, etc., or your designated memory locations) after `SAVESTATE`, and copy them back to those registers before the `RESTORESTATE`. This is the quickest way to get working code.

Another solution, which may be more robust, is to reserve one thread just for the thread manager, so that you always switch context to this thread (which runs outside the ISR), and it has its own register set and stack, so that trashing `R0`-`R7` is not an issue. It also has the advantage of keeping the ISR short, especially if the scheduler itself gets more complex. However, it also means using one thread, so the user has now one fewer thread.

3. [20 points] Screenshots

3.1 [2 points] Screenshots for compilation

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

3.2 [18 points] Screenshots and explanation

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.

- 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?
- How can you tell that the interrupt is triggering on a regular basis?

Additional Hints

- Many of the bugs have to do with the use of registers by the compiler as its temporaries. You should look at the .asm file to see how code gets compiled. You notice that any indexing operation (e.g., `savedSP[currentThread] = ..`), SDCC generates code that uses register `r0` and possibly `r1` to hold the address or offset. You may need to save and restore those registers. As shown in class, after you have done `PUSH` of all those registers such as `ACC`, `B`, `DPH`, `DPL` etc, you can reuse them or use them as temporary for saving other registers (`R0`-`R7`) that are used.
- Of course, you can always push register values onto stack and pop them off, as long as you are still using the same stack.
- In case of shift operation, SDCC actually makes uses of `ar7` and `ar6` (especially if you use any shift operations, like `<<` or `>>` in C, or AND operator). Note that they are two names defined by SDCC-generated to be idata addresses 7 and 6, respectively. That is, they are hardwired to registered R7 and R6 in bank 0, rather than your current bank. You need to identify these and save accordingly.
- In SDCC, if you are declaring several variables of the same type with your own assigned locations, you should declare them on separate lines, instead of just specifying one address for them. For example, **don't do this**:

```
__data __at (0x35) char a, b, c;  /* the locations of b, c may not be defined! */
```

  instead, do them separately, so you can be sure they are :

```
__data __at (0x35) char a;
__data __at (0x36) char b;
__data __at (0x37) char c;
```

- If your cooperative code works but your preemptive version doesn't, chances are your code has one of these register problems that you didn't fix. Go back and examine the .asm files in your cooperative version and make those fixes with registers. Make sure your fix to the cooperative code doesn't break anything. Once they work, then transfer those fixes to the preemptive version.

Advanced Topics

In case you are interested in why SDCC generates code using `ar7` (direct address 7, defined in the `.asm` code by the assembler directive for constant declaration `ar7 = 7`) instead of `r7` (register 7), 8051 instructions like `ANL` (“AND logical”) are limited to the following combinations (see the data sheet for details):

```
ANL A, Rn
ANL A, direct
ANL A, @Ri
ANL A, #data
ANL direct, A
ANL direct, #data
```

But 8051 instruction set does not allow

```
ANL Ri, ____
```

of any sort. That is why SDCC (assuming you don’t make use of register bank) uses `ar7` (direct address) instead of `r7` (register).