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

Programming Project Checkpoint 3

What to submit: One zip file named <studentID>-`ppc3.zip` (replace <studentID> with your own student ID). It should contain:

- one PDF file named **`ppc3.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 Makefile from CheckPoint2.
    - File named `ppc3.pdf` that includes the screenshots and explanations as instructed below.

For this programming project checkpoint, you are to implement (1) semaphore with busy wait for your preemptive multithreading and (2) test your code using the **classical bounded-buffer** example.

You should make a new directory and copy the source files from the previous checkpoint. No need to create new files. Instead, first add the semaphores in `preemptive.h` (and `preemptive.c` if necessary). Second, use the semaphore primitives to write the revised producer/consumer example.

1. [40 points] preemptive.h (and preemptive.c if necessary): semaphores

Add a semaphore API to `preemptive.h` and `preempt.c` if necessary, depending on whether you implement it as a macro or as a function. A macro simply defines the inline assembly code and gives you precise control over the implementation; so we will assume a macro implementation, although you may also choose code implementation.

The basic semaphore API to implement consists of

```
#define SemaphoreCreate(s, n)  // create a counting semaphore s
                               // that is initialized to n
```

```
#define SemaphoreWait(s)        // do (busy-)wait() on semaphore s
#define SemaphoreSignal(s)      // signal() semaphore s
```

You will need to fill in the macro definitions. We use the `Semaphore` prefix in front of the `wait()` and `signal()` functions to prevent name conflict.

1.1 SemaphoreCreate(s, n)

`SemaphoreCreate()` simply initializes the semaphore *s* (an "integer", but we will just use a signed char) to the value *n*. In the busy-waiting version, that is all it takes. You can write this as a function in C if you want, since it is just a simple assignment. You can also write it as a macro of inlined assembly if you want, but you can do that optimization later.

If you want to write this as a function in C, you will need to pass the pointer to the semaphore, so that the semaphore variable can be changed. (The textbook's example passes the semaphore or lock variable *by reference* in C++ style as indicated by the & operator in front of the formal parameter name) . However, it is not recommended, also because passing multiple parameters including pointers can get complicated in SDCC.

[**Extra credit**] If you are doing the non-busy-waiting version, then you will want to initialize any associated data structures, like the list of threads that are blocked on this semaphore.

1.2 SemaphoreSignal(s)

`SemaphoreSignal()` is very simple for the busy-waiting version of semaphore: simply increment the semaphore variable. It is recommended that you write this in inlined assembly, because you can do this as an atomic operation (i.e., single assembly instruction), namely `INC`. The operand of the assembly instruction can refer to a global variable name defined in C by prepending it with an underscore (_). For example, if you declare in C

```
char sem;
```

then you can increment sem in 8051 assembly by

```
INC _sem
```

However, if you define it as a macro argument, e.g., `#define SemaphoreSignal(s)`, the symbol `_s` in your macro definition won't expand `s` into the actual name of the semaphore like `_sem`, but `_s` is preprocessed just another identifier (i.e., left as-is)! Fortunately, C macro preprocessor includes the `##` operator, which lets you concatenate two symbols together. For convenience, you can define another macro for converting a C name into an assembly name:

```
#define CNAME(s) _ ## s
```

This way, if you write `CNAME(mutex)`, it gets macro-expanded into `_mutex`, which is exactly what you want.

[**Extra Credit**] The non-busy-waiting version actually does more: it also needs to go through the waiting list and pick one process to wake up.

1.3 SemaphoreWait(s)

This one is a bit more work, but still not too difficult. The pseudocode looks like

```
Wait(S) :
   while (S <= 0) { } // busy wait
   S--;
```

One thing to notice is that since the value of `S` is expected to drop below zero, do not declare the semaphore as an unsigned!

You could write this in C, but SDCC might not compile it to the right code. Look at the `.asm` file to see how it compiles the code. If you can get it working properly in C, then that is fine. However, it is recommended that you write assembly code for this.

The assembly code looks something like this:

```
#define SemaphoreWaitBody(S, label) \
    { __asm \
label: ;; top of while-loop \
       ;; read value of _S into ACC (where S is semaphore) \
       ;; conditionally jump(s) back to label if ACC <= 0  \
       ;; fall-through to drop out of while-loop \
           dec  CNAME(S) \
      __endasm; }
```

The list of conditional branch instructions in 8051 are

- `JZ  label        ;; jump if accumulator is zero`
- `JNZ label      ;; jump if accumulator is not zero`
- `JB bit, label    ;; jump if bit is 1`
- `JNB bit, label   ;; jump if bit is 0`

Hint: if `ACC.7` bit (the sign bit) is 1, then the number in the accumulator is negative.

However, there is one more issue, which is with the **assembly label**. Because we are defining it as a macro, and we (can) use multiple semaphores in the same code, we need to give each macro-expanded instance of inlined assembly code a different label, so the instruction can jump to the proper label. However, we don't want the caller to have to come up with a unique label each time they call `SemaphoreWait`. Fortunately, this problem can

be solved by using C preprocessor's `__COUNTER__` name, which generates a new integer literal each time it is used. SDCC reserves the labels in the form of 1$, 2$, … up to 100$ for the user's code without conflict with the assembler's labels. So we can define `SemaphoreWait` to pass a unique label based on `__COUNTER__` to `SemaphoreWaitBody` each time. Just be sure you concatenate `__COUNTER__` with the `$` to form the label.

[**Extra Credit**] The non-busy-waiting version will need to put the thread to a list of waiting processes, instead of busy waiting. But if the thread does not have to wait then it just does the decrement. Note that at the time of getting waken, the semaphore value can still be negative! So this means you don't keep testing the `while (s <= 0);` when you are waken, you just decrement the `s` anyway, even if it is still negative!

2. [40 points] `testpreempt.c`

Your `testpreempt.c` can be based on that for CheckPoint2. You are to replace the test case with the bounded-buffer (3-deep) instead of the single buffer that you have been using.

2.1 Process Synchronization

We will use the "Bounded Buffer", as shown in Chapter 7's first classical synchronization problem (slides 3-4). Make sure you understand how this works before proceeding.

Declare your three semaphores named `mutex`, `full`, and `empty` at known addresses. Add code to create and initialize them accordingly.

Declare a 3-deep char buffer, and you need to keep track of the head and tail for this circular queue -- either as index into the array or as pointer. Initialize them accordingly -- in your code, not in the global declaration.

2.2 Producer

The producer "produces" one character at a time starting from `'A'` to `'Z'` and start over again. It loops forever

```
  SemaphoreWait(empty);
  SemaphoreWait(mutex);
  // add the new char to the buffer
  SemaphoreSignal(mutex);
  SemaphoreSignal(full);
```

You are to write the code to add the new char to the buffer and generate the next char outside. Note: keep the critical section as short as possible!

2.3 Consumer

The consumer consumes one char at a time from the buffer but otherwise should behave the same as before. That is, it needs to loop forever

```
SemaphoreWait(full);
SemaphoreWait(mutex);
// remove the next char from the buffer
SemaphoreSignal(mutex);
SemaphoreSignal(empty);
```

Similarly, keep the critical section as short as possible!

3. [20 points] Screenshots

3.1 [2 points] Screenshots for compilation

Turn in a screenshot showing compilation of your code using the Makefile (same as CheckPoint 2). 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 screenshots when the Producer is running and show semaphore changes.
- Take screenshots when the Consumer is running and show semaphore changes.