CS4212 – Compiler Design October 19, 2012
School of Computing
National University of Singapore

Problem Set 5

Semester 1, 2012/13

Due: ***October 28, 23:59*** ***Marks: 8***

Submission: In IVLE, in the cs4212 workbin, you will find a folder called `“Homework submissions”.` In that folder, you will find 1 ***new subfolder:*** **`PS5P01`**, where you are expected to submit your solution.

The authors' ***names and matriculation numbers should appear clearly in a comment*** at the top of the file representing your submission. Teams may have at most 2 members. Each team should have a single submission for each problem; there is ***no need for multiple submissions of the same solution*** from each of the contributors. Please remember that for team submissions, each team member receives 75% of the marks awarded to the solution.

In this problem set, we shall consider the problem of generating *pseudo-concurrent* code. To keep things simple, we shall add the pseudo-concurrency feature to our first toy language, the one implemented by `comp-stmt.pro` in Lecture 4. This language is extended with a concurrency construct represented by the operator || (resemblant of the parallel notation in geometry). For simplicity, we shall assume that in each program, this operator is only used once, to create dynamically two threads. When the threads reach their join point, the program terminates. Here is an example program that uses this operator:

```
i = 0 ; j = 0 ; a = 0 ;
{ % thread 0
  while i < 10 do {
    a = a + 1 ;
    i = i + 1 ;
  }
} || { % thread 1
  while j < 10 do {
    a = a - 1 ;
    j = j + 1 ;
  }
}
```

In this program each thread executes a while loop. The two threads have concurrent access to variable **`a`**, thus leading to a race condition. The final value of **`a`** can be any value between -10 and 10. The program has an initial sequential part, which initializes the variables, and, in compliance with the above specification, creates 2 threads, and terminates at their *join point*.

By *join point*, we mean that the two threads will wait for each other; the one that reaches first will block till the other thread arrives as well.

The following is a possible translation of the program above. The code was obtained by making changes to the AL code produced for the sequential fragments of the program above by the `comp-stmt.pro` toy compiler, and adding a few calls to a pseudo-concurrency API that will be described below. The file has been uploaded to the workbin under the name **`p.s`**.

```
               .text
               .globl _entry
_entry:        # initial sequential segment
               pushl %ebp
               movl %esp,%ebp
               pushl $0
               popl %eax
               movl %eax,i    # i = 0
               pushl %eax
               popl %eax
               pushl $0
               popl %eax
               movl %eax,j    # j = 0
               pushl %eax
               popl %eax
               pushl $0
               popl %eax
               movl %eax,a    # a = 0
               pushl %eax
               popl %eax

               # intialize the thread system
               call init_threads

               # second thread begins at L6
               movl $L6,%eax
               call set_second_thread

               # from this point on, we have two threads
               # and any call to cosw will switch to the other
               # thread

               # code of first thread

               jmp L2
L3:
               pushl a
               call cosw  # randomly inserted context switch
               pushl $1
               popl %ebx
               popl %eax
               addl %ebx,%eax
               pushl %eax
               popl %eax
               movl %eax,a
               pushl %eax
               popl %eax
               pushl i
               call cosw  # randomly inserted context switch
               pushl $1
               popl %ebx
               call cosw  # randomly inserted context switch
               popl %eax
               addl %ebx,%eax
               pushl %eax
               popl %eax
               movl %eax,i
               call cosw  # randomly inserted context switch
               pushl %eax
               popl %eax
L2:
               pushl i
```

```
                pushl $10
                popl %eax
                call cosw  # randomly inserted context switch
                popl %ebx
                cmpl %eax,%ebx
                jge L0
                pushl $1
                jmp L1
L0:
                pushl $0
L1:
                popl %eax
                cmpl $0,%eax
                jne L3
                jmp join  # end of first thread
                          # jump to the join point

L7:       # second thread begins here
                pushl a
                pushl $1
                popl %ebx
                call cosw  # randomly inserted context switch
                popl %eax
                subl %ebx,%eax
                pushl %eax
                call cosw  # randomly inserted context switch
                popl %eax
                movl %eax,a
                pushl %eax
                popl %eax
                call cosw  # randomly inserted context switch
                pushl j
                pushl $1
                popl %ebx
                call cosw  # randomly inserted context switch
                popl %eax
                call cosw  # randomly inserted context switch
                addl %ebx,%eax
                pushl %eax
                popl %eax
                movl %eax,j
                pushl %eax
                popl %eax
L6:
                pushl j
                pushl $10
                call cosw  # randomly inserted context switch
                popl %eax
                popl %ebx
                cmpl %eax,%ebx
                jge L4
                call cosw  # randomly inserted context switch
                pushl $1
                jmp L5
L4:
                pushl $0
L5:
                popl %eax
                call cosw  # randomly inserted context switch
                cmpl $0,%eax
                jne L7


join:           # join point, both threads will arrive here at different points
                call single_thread
                # single thread at this point, normal return from procedure
                movl %ebp,%esp
                popl %ebp
                ret
```

```
        .data
        .globl __var_area

        __var_area:
i:      .long 0
a:      .long 0
j:      .long 0

        .globl __var_ptr_area
        __var_ptr_area:
i_ptr:  .long i_name
a_ptr:  .long a_name
j_ptr:  .long j_name

__end_var_ptr_area:     .long 0

        .globl __var_name_area
        __var_name_area:
i_name:  .asciz "i"
a_name:  .asciz "a"
j_name:  .asciz "j"
```

To implement concurrency, the code makes use of the following API:

`init_threads:` procedure that initializes the thread system. Will create a main thread, for the current stream of sequential execution.

`set_second_thread:` procedure that creates a second thread. Needs the start address of this thread in the register `%eax`.

`cosw:` procedure that performs a context switch. When called, it will not immediately return to the current thread. Instead, it will save the state of the current thread, restore the state of the "other" thread, and resume it. The call is perfectly symmetric, each call leading to executing a fragment of the "other" thread.

`single_thread:` procedure that should be called by each thread at its join point. The main thread will wait for the second thread, terminate it, and then continue execution in a sequential fashion.

This API has been implemented in a separate file called **`threads.s`**. The file is included here for completeness, though you should not need to understand the details of this file in order to be able to use the API. However, the comments in the code should make the approach easy to understand. The code should be compiled with the command:

```
gcc –o p p.s threads.s runtime-stmt.c
```

The text of **Problem 1** follows after the code.

```
# Pseudo-concurrency API for PS4

        .text

# Procedure that initializes the thread system
# In this simple version, simply saves the stack pointer into
# the thread slot for the main thread

        .globl init_threads
init_threads:       # threads is an array of thread stack pointers
                    # the main thread is at threads[0]
```

```
        movl %esp, threads
        ret


# Procedure to set up the second thread
# Takes into %eax the address of the desired thread
# Sets up a stack for the second thread and the
#  entry in threads[1]
# Again, we take advantage of the simplicity of our approach
#  here: we're expecting that registers do not contain anything
#  important at the point of setting up a second thread (except
#  for %eax)
        .globl set_second_thread

set_second_thread:
        movl threads+4,%ebx
        subl $4,%ebx
        movl %eax,(%ebx)   # thread address into 'ret' field of stack
        subl $28,%ebx      # space for saving registers
        movl %ebx,threads+4
        ret


# Context switch procedure. Calls to this procedure can be
# inserted _anywhere_ in the body of a thread to trigger a
# context switch. There is NO REQUIREMENR that this call
# should be placed only between statements. The more frequent
# the calls, the finer the granularity of the concurrency

        .globl cosw
cosw:
        push %ebp # save all registers
        push %esi
        push %edi
        push %ebx
        push %ecx
        push %edx
        push %eax
        movl curr_thread,%eax # save %esp in slot of current thread
        movl %eax,%ebx        #  i.e. threads[curr_thread] = %esp
        shll $2,%ebx
        addl $threads,%ebx
        movl %esp,(%ebx)      # toggle thread number
        xorl $1,%eax          #  i.e. curr_thread ^= 1
        movl %eax,curr_thread
        movl %eax,%ebx       # load %esp from thread slot
        shll $2,%ebx         #   i.e. %esp = threads[curr_thread]
        addl $threads,%ebx
        movl (%ebx),%esp
        popl %eax            # restore registers of new thread
        popl %edx
        popl %ecx
        popl %ebx
        popl %edi
        popl %esi
        popl %ebp
        ret                 # This will resume the new thread

# Procedure to join the two threads into one
# The threads will set a flag to one, and then
# wait on each other's flag so as to make sure
# that they both have completed. Once that is achieved
# the main thread will simply no longer context-switch
# into the second thread, effectively terminating it.

        .globl single_thread
single_thread:
        movl curr_thread,%eax # find flag location for current thread
        movl %eax,%ebx
        shll $2,%eax
        addl $single_flag1,%eax
        movl $1,(%eax)        # set current flag

        # this loop will end up being executed by second thread
        # it appears to be an infinite loop, but it only executes
        # when main thread context-switches here -- this will
        # stop happening when main thread reaches join point
```

```
wait_main_thread_join:
        cmpl $0,%ebx               # check if this is the main thread
        jz  wait_snd_thread_join   # and jump to loop of main thread
        call cosw                  # if second thread, context-switch
        jmp wait_main_thread_join  # to allow main thread to complete
                                   # context switch will return into
                                   # second thread only if second thread
                                   # reached join point first -- repeated
                                   # context switches will allow main
                                   # thread to complete

        # this loop will be executed only by the main thread
        # it waits for the second thread to complete by checking its
        # flag and context-switching to it to allow it to progress
        # once single_flag2 is set, main thread stops context-switching
        # effectively completing the second thread

wait_snd_thread_join:
        cmpl $1,single_flag2
        jz joined
        call cosw
        jmp wait_snd_thread_join

        # the two threads are joined at this point
        # and the procedure can simply return,
        # allowing sequential execution of the main program
        # from this point on

joined:
        ret

        .data
        .globl curr_thread
curr_thread:      # holds the id of the currently running thread
         .long 0  # either 0 or 1, toggled by the context switch procedure

        .globl threads
threads:            # array of two elements, holding saved stack
                    # pointers for each thread
         .long 0    # first elem will be set by init_threads
         .long thread2_stack # second elem can be statically initialized
                             # with buffer allocated for second stack

        .globl single_flag1
        .globl single_flag2
single_flag1:       # two flags allowing the threads to synchronize
         .long 0
single_flag2:
         .long 0
        .globl thread2_start_stack
        .globl thread2_stack

thread2_start_stack: # buffer for stack of second thread
         .space 1000

thread2_stack:
         .long 0
```

Problem 1

Add the concurrency feature described in this problem set to the toy compiler of comp-stmt.pro. Your compiler should accept the || operator, as described in the specification (i.e. a single instance of the operator is allowed in the code, and the code terminates at the join point of the two threads). Your implementation should generate into the assembly language code appropriate calls to the threads API described above, according to the model given in the example. The calls to the context switch routine **`cosw`** should be as random as possible, and definitely not confined to the boundaries between statements. The threads API provided in the **`threads.s`** file becomes part of the runtime, and needs to be added to the compilation command.