CMSC 412 (shankar, exam 1, oct 27, 1994) Name: ______________________

Total points 50. Total time 75 mins. Closed book. There are 7 problems over 5 pages.

1. [7 pts] The following 3 processes implement a mutual exclusion algorithm.

```
cobegin
  P_A : loop                 P_B : loop                 P_C : loop
          entry_code                 entry_code                 entry_code
          critical_section  ||       critical_section  ||       critical_section
          exit_code                  exit_code                  exit_code
        end                        end                        end
coend
```

Assume the following:

- entry_code and exit_code use busy-waiting (e.g. with test-and-set loops).
- Execution time of entry_code is 0 if no process in its critical_section (e.g. the lock is free).
- Execution time of critical_section, exit_code, and loop branch is 10 ms.
- The 3 processes are run on a single CPU using round-robin scheduling with quantum size of 5 ms.

(a) What is the throughput, i.e. number of critical sections executed per second.

(b) What can you say about the waiting time of a process, i.e. the time taken to execute entry_code.

2. [5 pts] Using semaphores, modify the following so that at any time at most three of the processes, $P_1, P_2, \cdots, P_n$, are executing their T_i's.

```
cobegin
  P_1 : loop           P_2 : loop                 P_n : loop
          S_1;                 S_2;                       S_n;
          T_1;    ||           T_2;     || ... ||         T_n;
        end                  end                        end
coend
```

3. [5 pts] Is the following following a correct solution to the 2-process mutual exclusion problem? Justify your answer.

flag : array [0..1] of { true, false }. Initialized to false.
turn : 0..1. Initialized to 0.

cobegin

P_0: loop
    flag[0] $\leftarrow$ true ;
    while *turn* = 1 do
      { while *flag*[1] do skip ;
        turn $\leftarrow$ 0 } ;
    critical section ;
    flag[0] $\leftarrow$ false ;
  end
	$\|$	P_1: loop
		    flag[1] $\leftarrow$ true ;
		    while *turn* = 0 do
		      { while *flag*[0] do skip ;
		        turn $\leftarrow$ 1 } ;
		    critical section ;
		    flag[1] $\leftarrow$ false ;
		  end

coend

4. [8 pts] You have to solve the 4-process mutual exclusion problem by repeatedly applying Peterson's 2-process mutual exclusion solution. Assume the following constructs for Peterson's solution:

- variables *turn*, *flag*[0], *flag*[1]. /* of appropriate type and initial value */
- entry_code(*turn*, *flag*[0]) /* executed by process 0 */
- entry_code(*turn*, *flag*[1]) /* executed by process 1 */
- exit_code(*turn*, *flag*[0]) /* executed by process 0 */
- exit_code(*turn*, *flag*[1]) /* executed by process 1 */

Using the above constructs, solve the 4-process mutual exclusion problem, i.e. supply the variables and code for the boxes below. Your solution must not use any other synchronization primitives (such as test-and-sets, disabling/enabling interrupts, semaphores, etc.).

```
[      ]
cobegin
  A : loop                    B : loop                    C : loop                    D : loop
        [      ]                    [      ]                    [      ]                    [      ]
        critical_section   ||       critical_section   ||       critical_section   ||       critical_section
        [      ]                    [      ]                    [      ]                    [      ]
      end                         end                         end                         end
coend
```

Hint: think of a binary tree with four leaves representing A, B, C, and D.

5. [5 pts] In project 2, the top of the stack of a ready process contained the address of `dispatch ()`, and you had the following:

```
void scheduler ()
 { ...
   /* get next ready process,
      put it in run queue,
      update \_SP and \_SS to process's stack */
      asm retf  /* go to dispatcher */
 }

void dispatch ()
 { asm {
    mov sp, bp
    pop bp
    pop bp
    pop di
    ...
  }
```

If you replaced the `asm retf` in the scheduler by `dispatch ()`, what changes would be required elsewhere?

6. [5 pts] In project 2, you had

`void interrupt system_service ( sbp, sdi, ssi, sds, ses, sdx, ...  , proc, argc, argv )`

What is the purpose of specifying all these parameters in the definition of `system_service`?

7. [15 pts] You have to solve a variation of the readers-writers problem, in which multiple writers can write at the same time. Specifically, there are readers and writers. Multiple reads at the same time are allowed. Multiple writes at the same time are allowed. A read and a write at the same time is not allowed.

Provide a solution using semaphores with the following properties:

- no busy waiting.
- starvation-freedom (i.e. a continuous stream of readers does not starve writers, and vice versa) is desirable but not compulsory (you will lose only some points).
- you cannot use process ids and you cannot have a separate semaphore for every process.

Below is a skeleton program for you to build upon by supplying code for the boxes and perhaps introducing more variables. You are also welcome to disregard this skeleton and come up with something else.

nr_active : integer initialized to 0.	/* number of reads currently executing */
nr_wait : integer initialized to 0.	/* number of readers currently waiting */
r_sem : semaphore initialized to 0.	/* readers wait here */
nw_active : integer initialized to 0.	/* number of writes currently executing */
nw_wait : integer initialized to 0.	/* number of writers currently waiting */
w_sem : semaphore initialized to 0.	/* writers wait here */

Readers execute the following:	Writers execute the following:
loop	loop
$\boxed{}$;	$\boxed{}$;
Read operation ;	Write operation ;
$\boxed{}$;	$\boxed{}$;
end	end

SOLUTION TO EXAM 1 CMSC 412 (shankar): Fall 94

1. [7 pts]

Assume without loss of generality that round-robin queue has processes in the order A, B, and C. Here is the evolution starting from time 0.

A executes critical_section till time	5 ms	A waited 0 ms
B executes entry_code till time	10 ms	B starts wait at 5 ms
C executes entry_code till time	15 ms	C starts wait at 10 ms
A executes critical_section, exit_code till time	20 ms	A completes
B executes critical_section till time	25 ms	B waited 15 ms
C executes entry_code till time	30 ms	
A executes entry_code till time	35 ms	A starts wait at 30 ms
B executes critical_section, exit_code till time	40 ms	B completes
C executes critical_section till time	45 ms	C waited 30 ms
A executes entry_code till time	50 ms	
B executes entry_code till time	55 ms	B starts wait at 50 ms
C executes critical_section, exit_code till time	60 ms	C completes

and so on

(a) [5 pts] one critical_section executed every 20 ms, or equivalently 50 per second.

(b) [2 pts] average waiting time is 30 ms (ignoring initial transient).

2. [5 pts] Introduce the following: a semaphore, say $gate$, initialized to 3; $P(gate)$ before every T_i; and $V(gate)$ after every T_i.

3. [5 pts] No. It allows both processes to enter CS as follows.

P_1 executes - $flag[1] \leftarrow$ true
- while $turn = 0$ do
- while $flag[0]$ do skip
(P_1 at $turn \leftarrow 1$)

P_0 executes - $flag[0] \leftarrow$ true
- while $turn = 1$ do
(P_0 at CS)

P_1 executes - $turn \leftarrow 1$
- while $turn = 0$ do
(P_1 at CS)

Can also have starvation as follows:

(1) P_0 executes - $flag[0] \leftarrow$ true ; ... ; enters critical_section
(2) P_1 executes - $flag[1] \leftarrow$ true ; enters while $flag[0]$ do skip
(3) P_0 executes - $flag[0] \leftarrow$ false ; branches back ; $flag[0] \leftarrow$ true ; critical_section

Steps (2), (3), (2), (3), ... can repeat endlessly. Each process is getting a share of CPU, but P_1 starves.

4. [8 pts] Introduce three sets of variables, say, $turn_X$, $flag_X$ $turn_Y$, $flag_Y$ $turn_Z$, $flag_Z$

cobegin

A: loop	B: loop	C: loop	D: loop
entry_code($turn_X, flag_X[0]$)	entry_code($turn_X, flag_X[1]$)	entry_code($turn_Y, flag_Y[0]$)	entry_code($turn_Y, flag_Y[1]$)
entry_code($turn_Z, flag_Z[0]$)	entry_code($turn_Z, flag_Z[0]$)	entry_code($turn_Z, flag_Z[1]$)	entry_code($turn_Z, flag_Z[1]$)
critical_section ‖	critical_section ‖	critical_section ‖	critical_section
exit_code($turn_Z, flag_Z[0]$)	exit_code($turn_Z, flag_Z[0]$)	exit_code($turn_Z, flag_Z[1]$)	exit_code($turn_Z, flag_Z[1]$)
exit_code($turn_X, flag_X[0]$)	exit_code($turn_X, flag_X[1]$)	exit_code($turn_Y, flag_Y[0]$)	exit_code($turn_Y, flag_Y[1]$)
end	end	end	end

coend

[3 points for recognizing three applications of Peterson's algorithm, 2 points for declaring the variables, 3 points for the entry and exit code.]

5. [5 pts] Calling `dispatch ()` instead of using `retf` places the return address (a location within scheduler) on the stack, after which `dispatch ()` puts its frame on the stack.

The simplest fix is to modify `dispatch ()` so that it removes the frame (as is currently done), then removes the next 4 words on stack (skipping return address and address of `dispatch` on stack), and continue as before (restore registers).

A more natural fix is to not have the address of `dispatch ()` on top of a ready process's stack. `proc_start` and `defer` should not put address of `dispatch ()` on stack. The beginning of `dispatch ()` should be modified to remove the frame (as is currently done) and the return address to scheduler.

6. [5 pts] To access variables that are already on the process's stack, and have the compiler generate the addressing code.

7. [15 pts]

Add semaphore *mutex* initialized to 1 to protect the variables.

Readers execute the following:

loop
  P($mutex$) ;
  if $nw\_active > 0$ or $nw\_wait > 0$ then
    { $nr\_wait \leftarrow nr\_wait + 1$;
     V($mutex$) ; P(r_sem) }
  else { $nr\_active \leftarrow nr\_active + 1$;
     V($mutex$) }
  Read operation ;
  P($mutex$) ;
  $nr\_active \leftarrow nr\_active - 1$;
  if $nr\_active = 0$ and $nw\_wait > 0$ then
    { for $i = 1$ to nw_wait do V(w_sem) ;
     $nw\_active \leftarrow nw\_wait$; $nw\_wait \leftarrow 0$; }
  V($mutex$) ;
end

Writers execute the following:

loop
  P($mutex$) ;
  if $nr\_active > 0$ or $nr\_wait > 0$ then
    { $nw\_wait \leftarrow nw\_wait + 1$;
     V($mutex$) ; P(w_sem) }
  else { $nw\_active \leftarrow nw\_active + 1$;
     V($mutex$) }
  Write operation ;
  P($mutex$) ;
  $nw\_active \leftarrow nw\_active - 1$;
  if $nw\_active = 0$ and $nr\_wait > 0$ then
    { for $i = 1$ to nr_wait do V(r_sem) ;
     $nr\_active \leftarrow nr\_wait$; $nr\_wait \leftarrow 0$; }
  V($mutex$) ;
end

[2 points for *mutex*, 4 for updating *nr_active* and *nw_active* (consitently), 3 for updating *nr_wait* and *nw_wait*, 2 for P(r_sem) and P(w_sem), 4 for P(r_sem) and P(w_sem).]

Common mistakes: Using more than one *mutex*. Doing only one V when last reader (or writer) leaves (causes readers and writers to alternate in heavy load, readers can be left stranded if no writers show up, ...).