

CSE 411

Fall 1999

Solution for Homework #1

Nan Huo, Yiqiong Wu and Anirudh Modi

September 13, 1999

1.
 - (3.5a of the text) That VAX/VMS operating system has many distinct wait states which is better than having just one (as in Figure 3.5 of the text) or two (as in Figure 3.15 of the text). The reason is that if all *blocked* processes are placed in one blocked queue, when an event occurs, the operating system must scan the entire blocked queue to search for the processes waiting on that event. Since VAX/VMS operating system may have so many blocked processes, it would be more efficient to have several wait states, one for each event. Then, when the event occurs, such as I/O completion, a resource can be freed and the appropriate process can be moved to the *ready* state.
 - (3.5b of the text) I/O activities are much slower than computation. Therefore, when memory holds multiple processes, the processor can move to another process while one process is waiting. When all of the processes in memory are waiting for I/O, the operating system has to swap some blocked processes out to the disk onto swapped-out state. So the difference between a resident process and a swapped-out one is whether or not it is in the main memory. Those wait states only have one version because they may only be in one state for the above situation. Pagefault wait state is waiting for the page not in main memory, which does not use any memory, so it need not be swapped out. Other page wait states and common event wait states may use some memory, but since they involve the use of it, it would not be applicable in swapped-out state. Resource wait is waiting for miscellaneous system resources, which means it could wait a long time for the resource to be freed, therefore it would not be reasonable to keep it in memory. So it may only have swapped-out version and access the resource when the resource frees up.

2. (5.5 of the text) *Counterexample*: Assume that the thread P(1) executes ahead of P(0) and reaches line 11. Then thread P(0) starts and reaches line 8. At this instant, for P(0), turn=0 and id=0 (since P(1) has not yet executed line 12), which causes it to skip the while loop and go to the *critical section*. Later, when P(1) executes line 12, it makes turn equal to id, thus skipping the while loop and entering the *critical section*. Thus, both P(0) and P(1) have entered the critical section simultaneously, making the solution to the *mutual exclusion* problem incorrect.

```
| 1| var  blocked: array [0..1] of boolean;
| 2|     turn: 0..1;
| 3|
| 4| procedure P (id: integer);
| 5| begin
| 5|     repeat
| 7|         blocked[id] := true;
| 8|         while turn <> id do /* breakpoint 1 - P(0) here */
| 9|             begin
|10|                 while blocked[1-id] do;
|11|                     /* breakpoint 2 - P(1) here */
|12|                 turn := id;
|13|             end;
|14|         < critical section >
|15|         blocked[id] := false;
|16|         < remainder >
|17|     until false
|18| end;
```

3. Skipped (will not be graded)
4. Since the *cnotify* operation only moves a thread from the *blocked state* to the *ready state* and lets the scheduler determine which thread will run next, therefore, it causes the effect that the notified process might not be the next to use the critical section. The Java machine is a user process and all of the support of multi-threading is done at the user level. The Java interpreter will handle the scheduling decisions, hence it will give monitor the preference to the thread that is awakened. Thus, Java machine is more certain that the notified process will be the next to use the critical section.
5. *Solution*: People select their food at the same time. Whenever they want to buy any frozen food (i.e., ice-cream) on impulse, they go to a checkout counter and reserve one before they actually carry the frozen food item. The maximum number of registers that can be reserved for frozen food is 5 (one less than the number of cashiers). This is because whenever someone wants to pay for any non-frozen food, the case where all cashiers sit idle waiting for their frozen food customer is avoided.

```

Program grocery_store;
var   reserved: semaphore(:=5);
      register: semaphore(:=6);
      ready:   (* binary*) semaphore(:=0);
procedure customer;
begin
  /* arrive at store*/
  frozen := false
  /* select food. if want to buy frozen food on
     impulse (i.e., ice-cream), then set frozen as true */
  if frozen then
    begin
      P(reserved);
      P(register);
      /* continue to select food*/
      V(ready);
      /* purchase food at check-out counter */
      V(reserved);
    end
  else
    begin
      /* continue to select food */
      P(register);
      V(ready);
      /* purchase food at check-out counter */
    end
  /* leave the store */
end
procedure cashier;
begin
  repeat
    P(ready);
    /* accept pay */
    V(register);
  forever
end

begin(* main program *)
  parbegin
    customer;...      /* equal to the number of customers */
    cashier; cashier; cashier; cashier; cashier; cashier;
  parend
end

```

Alternatively, one can keep the ice-cream (which may be bought on impulse) at the checkout counter itself (if it is possible; as done in Wal-mart), so that the customer does not have to wait after he has decided to buy it (since he is already at the checkout counter). Thus, the checkout counters can be reserved for the customers who want to buy frozen food as soon as they enter the grocery store, thus removing the last constraint and further simplifying things.

6. *Attempt 1:*

- *Cause of the error:* The program does not satisfy the constraint that a CO_2 molecule must be produced any time there are atleast two Oxygen atoms and one Carbon atom present. If there are a lot of Carbon atoms present, some of them may just grab one Oxygen atom and form CO pairs.
- *Solution to solve the problem:* Apply a semaphore `cMutualExclu` only to Carbon atoms for mutual exclusion, where `cMutualExclu` is initially equal to 1.

```

oReady()
{
O -> V();
C -> P();
}

cReady()
{
cMutualExclu -> P();
O -> P();
O -> P();
cMutualExclu -> V();
makeCO2();
C -> V();
C -> V();
}

```

Attempt 2:

- *Cause of the error:* For Mesa semantics, the problem is that the waking processes may not update the values of the shared variables in time. For example, assume that there are two Oxygen atoms already in the system, and then two Carbon atoms arrive at the same time. The first Carbon atom will call `makeCO2()` because it sees the two Oxygen atoms, while the second atom will then wake up and sit in the *Ready* queue after the first Carbon atom. Therefore, the second Carbon atom will still see that there are two Oxygen atoms in the blocked state because the `oReady` functions have not decremented the value of the variable O_s and the value of O_s is still 2.
- *Solution to solve the problem:* Use a *signal* to decrement the values of O_s and C_s . e.g., let $O_s = O_s - 2$ in `cReady()`.