Department of Electrical and Computer Engineering

Computer Engineering

Course ECSE-322B

Problem Set 10 - Solutions

April 6, 2008

1. Simulate the traffic at an intersection of two one-way streets using semaphore operations. In particular, the following rules should be satisfied:

> Only one car can be crossing at any given time
> When a car reaches an intersection and there are no cars approaching from the other street, it should be allowed to cross
> When cars are arriving from both directions, they should take turns to prevent indefinite postponement of either direction.
>
> Use diagrams to show how the semaphores would work.

Solution 1:

Semaphores:

Junction available = ja

add a favoured direction flag = fd.

Fd is set to ew if the east-west direction is favoured, it is set to ns if the north south direction is favoured.

The junction is a non-shareable resource so the basic loop must be:

North-South crossing.

East-West crossing

```
Loop:

    produce car
    wait(ja);
    Car crosses;
    signal(ja);
    go to loop;
```

```
Loop:

    produce car;
    wait(ja);
    Car crosses;
    signal(ja);
    go to loop;
```

This works if cars approach from both directions but not at the same time. If they approach simultaneously, we need a favoured direction (similar to the process in Dekker's Algorithm) which alternates between each direction. Thus if ew is the favoured direction, a car approaching on the north-south road (ns) would be stopped if there is already a car on the east-west approach(ew).

Add the favoured direction flag:

North-South crossing.

```
Loop:

    produce car
    increment number of ns cars
while (fd!= ns & number of ew cars!=0)
        do nothing;
    wait(ja);
    Car crosses;
    signal(ja);
    fd = ew
    decrement number of ns cars
    go to loop;
```

East-West crossing

```
Loop:

    produce car;
    increment number of ew cars
while (fd!=ew& number of ns cars !=0)
        do nothing;
    wait(ja);
    Car crosses;
    signal(ja);
        fd = ns
    decrement number of ew cars
    go to loop;
```

Ideally, the operation of the junction should be drawn out as a set of states to verify that the above will work.

Solution 2:

New solution to remove busy wait loop, and possibility of indefinite wait when favored direction is always occupied.

Idea
In general, let cars go in the favored direction. But if the favored direction is always occupied, let a car go in the non-favored direction at a certain rate.

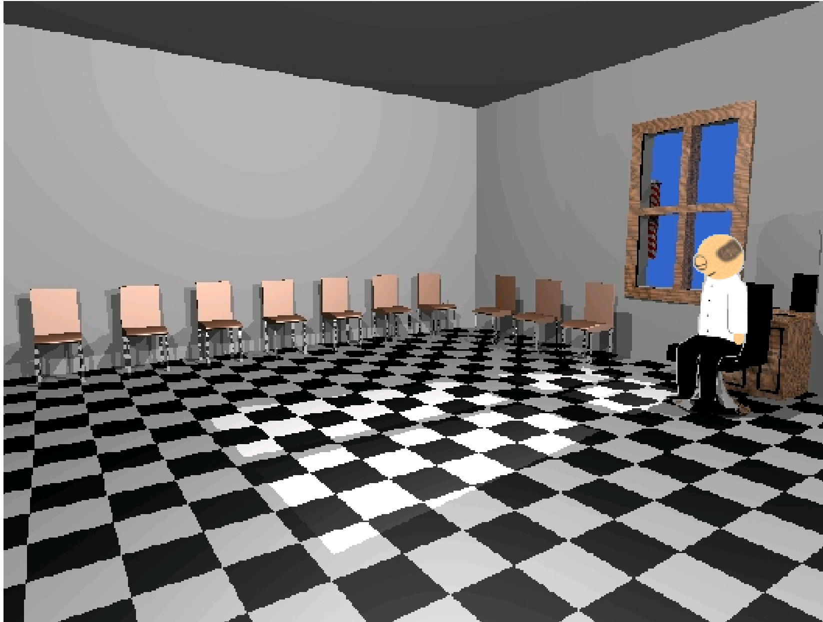
Description
ja: junction available, semaphore
fd_turns: *shared data*, number of consecutive turns in favored direction.
fd_car_count: number of waiting cars in favored direction
MAX_FD_TURNS: max number of times a car is allowed to go in the favored direction before the non-favored direction gets a turn.

Non-Favored Direction Process	**Favored Direction Process**

Loop:
 	
produce car;
 	
wait(ja);
while(0 < fd_turns < MAX_FD_TURNS
and
0 < fd_car_count)
 	
// let car go in favored direction, wait again
signal(ja);
wait(ja);
 	
end while
 	
car crosses;
fd_turns = 0;
signal(ja);
 	
go to Loop;
	Loop:
	
	produce car;
	increment fd_car_count;
	
	wait(ja);
	car crosses;
	increment fd_turns;
	signal(ja);
	
	decrement fd_car_count;
	
	go to Loop;

2. **Sleeping Barber Problem**:
A barbershop consists of a waiting room with *n* chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

Write the pseudo-code to coordinate the barber and the customers processes using semaphores.



Solution 1:

Identify processes
- The barber
- Each customer

Identify variables
- barber_state (busy, asleep)
- waiting (0, …, n) --- shared variable counts number of waiting customers
- customers (semaphore) --- indicates customers to be served
- next[i] (semaphore) --- indicates barber chair free for customer i
- done (semaphore) --- indicates customer has been shaved
- mutex (semaphore) --- update of shared variable

Identify events

- No customers, barber goes to sleep
- Customer enters, no chairs, customer leaves
- Customer enters, vacant chair, barber busy, customer takes free chair
- Customer enters, barber asleep, customer wakes barber

One possible solution (there are many):

Barber process:

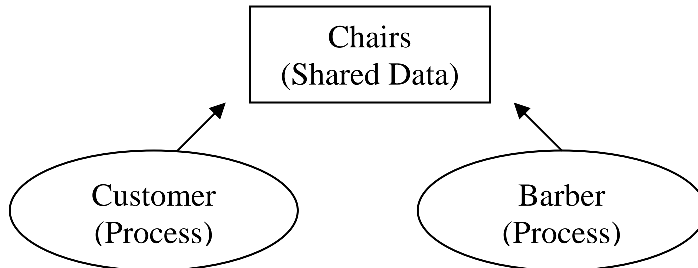
```
While(true)
        if(waiting==0)
                barber_state='asleep'; //go to sleep
                wait(customers);
        endif
        barber_state='busy';
        i=select_next_customer( ); //FCFS
        signal(next[i]);  //invite customer into barber chair
        wait(mutex);
        waiting--;
        signal(mutex);
        shave_customer( );
        signal(done);
endwhile
```

Customer process:

```
if(waiting==n)
        exit;       //no room, leave shop
endif
if(barber_state=='asleep')
        signal(customers);  //wake up barber
else
        wait(mutex);
        waiting++;
        signal(mutex);  //take a seat
endif
wait(next[i]);     //wait for barber to be free
wait(done);        //get shaved
exit;              //leave shop
```

Solution 2:

Simplified solution: only one piece of protected data, a ‘Chairs’ abstract data type. No need to protect all individual chairs, this is overly complicated and hard to verify that deadlock cannot occur.



Description

Chairs: abstract data type, represents a queue of all chairs.
Chairs.semaphore: semaphore used to restrict access to chairs.
Chairs.full(): returns true if all chairs are full.
Chairs.empty(): returns true if all chairs are empty
Chairs.add_customer(): adds a customer to the queue
Chairs.process_next_customer(): removes and processes next customer in queue

<table>
<tr><th>Customer Process</th><th>Barber Process</th></tr>
<tr><td>

```
if( Chairs.full() )
   exit;
else

   wait( Chairs.semaphore );
   Chairs.add_customer( me );
   signal( Chairs.semaphore );

end if;
```

</td><td>

```
while( true )

  if( Chairs.empty() )
    barber_state = ‘asleep’

 else
    wait( Chairs.semaphore );
    barber_state = ‘busy’;
    Chairs.process_next_customer();
    signal( Chairs.semaphore );

end while;
```

</td></tr>
</table>

3. What values can a semaphore have? Name the operations on a semaphore and describe what they do. Demonstrate that the current value of the semaphore is dependent on its starting value and the number of operations of each type done on it.

Answer:

A semaphore can have any non-negative integer value.
The two main operations on a semaphore are:

signal(s) (where s is the semaphore). This increments the value of s without allowing any interruption while doing so. i.e. it is an indivisible read-modify-write operation.

wait(s). This waits until the value of s is greater than zero and then decrements s without allowing for any interruptions in the operation, i.e. it is also an indivisible read-modify-write operation.

There is one more operation often used on a semaphore -

initialize (s, value) This set the semaphore, s, to a specific value, where ‘value’ is greater than or equal to zero.

Let the semaphore be, s. If the initial value is IV, the current value is CV, nw(s) is the number of wait operations performed on s and ns(s) is the number of signal operations performed on s, then:

CV = IV + ns(s) - nw(s) >= 0 (where >= means greater than or equal to)

If the initial value is 1 then

CV = 1 + ns(s) - nw(s) >= 0

If signal and wait operations occur in pairs, then the current value of a semaphore will always be either 0 or 1. If the initial value was 1, then the next value must be 0 and the above equation becomes

nw(s) <= ns(s) + 1 (where <= means less than or equal to)

(note that the above is more than the question asked for!)

4. Explain what is meant by *deadlock*. What are the necessary conditions for deadlock to occur?

Answer:

Deadlock is said to occur when one process is granted a resource but then needs another to complete its task. Simultaneously, a second process has been granted the second resource but needs the first to complete. Both processes cannot run because they each need a resource which is held by the other and neither process is willing to relinquish the resource it currently owns.

The necessary conditions for deadlock to occur are:

1. The resources involved must be unshareable
2. Processes hold the resources they have been allocated while waiting for new ones
3. Resources cannot be relinquished while being used
4. A circular chain of processes exists - each process currently holds resources being requested by the next process in the chain.