Virtual University	**Assignment No. 05 Solution** **SEMESTER SPRING 2012** **CS604- Operating System**	**Total Marks: 20**

Marks: 20

Question 1: [Marks 5]

1st Possible Solution:

```
int waiting = 0;  /* femaleclient  waiting */
semaphore femaleclient = 0;
semaphore shabnam = 0;
semaphore mutex = 1;
int SOFAS=n;

void shabnam ( )
{
  while (TRUE)
 {
     wait(femaleclient);
     wait(mutex);
     /* go to sleep if no femaleclient */
     waiting--; /* one less waiting */
     signal(shabnam);
     signal(mutex);
     Do_makeup();
   }
}
```

Code for femaleclient

```
void femaleclient ( )
{
  /* no free sofas? leave */
 wait(mutex);
  if (waiting < SOFAS) {
      waiting++; /* sit down */
      signal(femaleclient);
      signal(mutex);
      wait(shabnam);
           get_ makeup ();
  } else {
      signal(mutex);
  }
}
```

2nd Possible Solution:

Initialization of Semaphores:

```
room_size = 1                          //room_size is fixed to one
mutex = Semaphore(1)                   //confirms that no one using room
room_full = Semaphore(0)               //initially no client in the room
room_empty = semaphore(n)
local event
```

1. Since, we know that only one client can enter the makeup room at a time, so we declare a variable **room_size** and set it values to **1** i.e., **room_size = 1**.
2. **mutex** provides exclusive access to the makeup room
3. When **room_full** is positive, it indicates that some client is under the process of makeup in the room. When it is negative, it indicates that the room is empty
4. When the client leaves the room it should signal **room_empty**. When a patient arrives it should decrement **room_empty** , at which point it might block until the client signals
5. **event** is a local variable, which in this context means that each thread has its own version

Client_comes in:

```
room_empty.wait()        //check if there is a space for client
mutex.wait()             //make sure no other client is in the room
        room.add(event)  //client enter the room
mutex.signal()           //ok for others to go into room for makeup
room_full.signal() }     //tell other clients, there is a client in room
```

Client_wants_to_enter makeup room:

```
event = waitForEvent()

room_empty.wait()        //check if there is a space for client
mutex.wait()             //make sure no other client is in the room
    room.add(event)      //client enter the room
mutex.signal()           //ok for others to go into room for makeup
room_full.signal()       //tell other clients, there is a client in room
```

Client leaves makeup room:

```
room_full.wait()         //check if there's a client
mutex.wait()             //make sure no one else is in the room
        event = room.get() //client exits from the room
mutex.signal()           //next client turn
room_empty.signal()      //tell other clients that the room is empty

event.process()
```

1. In **"client_wants_to_enter"**, the client doesn't have to get exclusive access to the room until it gets an **event**

2. In **"client_leaves"**, the room operation is protected by mutex, but before the client leaves the room, it has to decrement **room_full**
3. In order to avoid deadlock, **"client_wants_to_enter"** and **"client_leaves"** check availability before getting the mutex. For best performance, they release the mutex before signaling

3rd Possible Solution:

```
# include <stdio.h>
# include <conio.h>
# include "prototypes.h"
# define SOFAS 5
typedef int semaphore;

semaphore        femaleclient=0;
semaphore        shabnam=0;
semaphore        mutex=1;
int               waiting=0;

void main()
{
do
{
if(femaleclient)
{
femaleclient ();
shabnam ();
}
}
while
}

Void shabnam (void)
{
while(TRUE)
{
femaleclient --;
Mutex--;
Waiting= waiting - 1;
shabnam ++;
Mutex++;
makeup ();
}
}

Void femaleclient (void)
```

```
{
mutex--;
if (waiting < SOFAS)
{
waiting = waiting + i;
femaleclient ++;
mutex ++;
shabnam --;
get_ makeup ();
}
else
{
mutex;
}
}

makeup ()

{
printf("shabnam is ready to do makeup");
}

get_ makeup ()

{
printf("femaleclient get makeup");
}
```

Question 2: [Marks 15]

a) Number of bits needed for 'p' **$\log_2 32$ = 5 bits** **[3 Marks]**
b) Number of bits needed for 'f' **$\log_2 64$ = 6 bits** **[3 Marks]**
c) Number of bits needed for 'd' **$\log_2 16$ = 4 bits** **[3 Marks]**
d) Size of logical address i.e. Number of bits needed to uniquely identify a page in this address space of 16 pages. **[3 Marks]**

If no. of pages = 32
Then, Number of bits needed to uniquely identify a page will be
p,d = 5 + 4 = 9 bits
But If no. of pages = 16
Then, Number of bits needed to uniquely identify a page will be
p,d = 4 + 4 = 8 bits

e) Logical address in bits for (18, 10). Where 'p' and 'd' are 18 and 10 respectively. **[3 Marks]**

P = 18 = 10010
D = 10 = 1010

So logical address will be 100101010