First question

- A cruising boat can carry up to 80 passengers. These passengers can embark or debark through a narrow gangway that can accommodate one person at a time.

- Add the required semaphore calls to the following two functions to ensure that the boat nor its gangway will never be overloaded. (30 points)
First question

- semaphore boat = 80;
  semaphore gangway = 1;

- embark()
  
  ________;
  ________;
  ________;
  walk();
  ________;
  ________;

} //embark

- debark()
  
  ________;
  ________;
  ________;
  walk();
  ________;
  ________;

} //debark
Let us put first the mutex calls

- semaphore boat = 80;
- semaphore gangway = 1;

- embark(){
  ________;
  P(&gangway);
  walk();
  V(&gangway);
  ________;
} //embark

- debark(){
  ________;
  P(&gangway);
  walk();
  V(&gangway);
  ________;
} //debark
Then the two other calls

- semaphore boat = 40;
  semaphore gangway = 1;

- embark(){
  P(&boat);
  P(&gangway);
  walk();
  V(&gangway);
  ________;
}

//embark

debark(){
  ________;
  P(&gangway);
  walk();
  V(&gangway);
  V(&boat);
}

//debark
Second question

Consider the following System V.4 scheduler:

<table>
<thead>
<tr>
<th>ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 3</td>
</tr>
</tbody>
</table>

Which priority levels will be visited by a process that starts at level 1, waits for 20 seconds, requests 80ms of CPU time and does an I/O request (2×5 points)
Answer

<table>
<thead>
<tr>
<th>ts quantum</th>
<th>ts tqexp</th>
<th>ts slpret</th>
<th>ts maxwait</th>
<th>ts lwait</th>
<th>LVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 3</td>
</tr>
</tbody>
</table>

- Which priority levels will be visited by a process that starts at level 1, waits for 20 seconds, requests 80ms of CPU time and does an I/O request (2×5 points)

- Goes from level 1 to level 2 because ts_lwait
- Goes from level 2 to level 3 because ts_slpret
Third question

- What is the *main disadvantage* of the *round-robin* CPU scheduling policy?
Third question

- What is the main disadvantage of the round-robin CPU scheduling policy?
  - It causes too many context switches at medium to high CPU loads
    - Because we want to keep a good response time for interactive requests
Third question

- Of all *four necessary conditions for deadlocks*, which one is the *easiest to deny*?
Third question

- Of all four necessary conditions for **deadlocks**, which one is the **easiest to deny**?

  - **Circular wait**
Third question

- What is the main advantage of **datagrams** over **virtual circuits** and **streams**?
Third question

- What is the main advantage of datagrams over virtual circuits and streams?

  - Much lower overhead
Third question

- How can you simulate a *blocking receive* using only *non-blocking primitives*?
Third question

- How can you simulate a **blocking receive** using only **non-blocking primitives**?

  - Using a busy wait:
    ```c
    while(receive(mbox, buffer, nbytes) == NO_MSG);
    ```
Third question

- Why do *notify* primitives are *safer* to use than the older *signal* primitives?
Third question

- Why do notify primitives are safer to use than the older signal primitives?

  - Because a notify call never interrupts the procedure calling it
Third question

- What is the best way to *prevent starvation* in scheduling policies implementing variable priorities?
Third question

What is the best way to prevent starvation in scheduling policies implementing variable priorities?

- We should increase the priorities of processes that have waited for too long in the ready queue.
Fourth question

- Consider the following solution to the mutual exclusion problem and explain when it fails (5 points) and what happens then. (5 points)
Fourth question

- shared int reserved[2] = {0, 0}; // global var
- void enter_region(int pid) { // 0 or 1
  int other;
  other = 1 - pid; // pid of other process
  reserved[pid] = 1; // reserve
  while (reserved[other]); // busy wait
  reserved[pid] = 1; // reserve
} // enter_region

- void leave_region(int pid) {
  reserved[pid] = 0;
} // leave_region
Fourth question

- shared int reserved[2] = {0, 0}; // global var
- void enter_region(int pid) { // 0 or 1
  int other;
  other = 1 - pid; // pid of other process
  reserved[pid] = 1; // reserve
  while (reserved[other]); // busy wait
  reserved[pid] = 1; // reserve
} // enter_region

- void leave_region(int pid) {
  reserved[pid] = 0;
} // leave_region

The two critical lines
Fourth question

- shared int reserved[2] = {0, 0}; // global var
- void enter_region(int pid) { // 0 or 1
  int other;
  other = 1 - pid; // pid of other process
  reserved[other] = 1; // reserve
  while (reserved[other]); // busy wait
  reserved[pid] = 1; // reserve
} // enter_region

- void leave_region(int pid) {
  reserved[pid] = 0;
} // leave_region

Reserve then check: risk of deadlock
No tie-breaker
Final answer

- Solution will cause a **deadlock** when two processes attempt to enter the critical section in lockstep.
Fifth question

Consider the function

```c
void badexchange(int *pa, int *pb) {
    *pa = *pb;
    *pb *= *pa;
} // badexchange
```

and assume the following calling sequence:

```c
alpha = 3; beta = 5;
badexchange (&alpha, &beta);
```
Fifth question

What will be the value of alpha and beta after the call assuming that the call was:

- A conventional procedure call? (5 points)
  alpha = __________ beta = __________

- A remote procedure call? (5 points)
  alpha = __________ beta = __________
The hard case

Given

void badexchange(int *pa, int *pb) {
  *pa = *pb;
  *pb *= *pa;
} // badexchange

and the calling sequence

alpha = 3; beta = 5
badexchange (&alpha, &beta);

a conventional call would return

alpha = 5 and beta = 25
The easy case

- Given
  - `void badexchange(int *pa, int *pb) {
    *pa = *pb;
    *pb *= *pa;
  } // badexchange`

  and the calling sequence
  - `alpha = 3; beta = 5;
    badexchange (&alpha, &beta);`

  an RPC would return
  - `alpha = 5 and beta = 25`
Sixth question

- What is the **main advantage** of the *all or nothing* RPC model over the *at most once* model? (5 points)

- What is its **sole disadvantage**? (5 points)
Sixth question

- What is the **main advantage** of the **all or nothing** RPC model over the **at most once** model? (5 points)
  - It eliminates the risk of **partial executions** of the request
  - **Crucial for financial transactions.**

- What is its **sole disadvantage**? (5 points)
  - Its **high overhead**