Assignment #2 (50 points, MS Word/PDF file only)

1. Many current language specifications, such as for C and C++, are inadequate for multithreaded programs. This can have an impact on compilers and the correctness of code, as this problem illustrates. Consider the following declarations and function definition:

```
int global_positives = 0;
typedef struct list {
struct list *next;
double val;
} * list;

void count_positives(list l)
{
list p;
for (p = l; p; p = p -> next)
if (p -> val > 0.0)
++global_positives;
}
```

Now consider the case in which thread A performs

```
count_positives(<list containing only negative values>);
```

while thread B performs

```
++global_positives;
```

The C language only addresses single-threaded execution. Does the use of two parallel threads create any problems or potential problems?

The C language was not initially designed with parallelism in mind, and it does not provide built-in support for multithreading or concurrency. However, that doesn't mean you can't use C to write parallel programs with multiple threads.

There are indeed potential problems that can arise when you use multiple threads in a C program. For example, if two or more threads access and modify the same shared data concurrently, you can run into what's called a race condition. This occurs when the order in which the threads modify the data is not guaranteed, leading to unpredictable results.

To avoid race conditions and other problems, it's important to use synchronization primitives, like mutexes or semaphores, to coordinate access to shared data. This ensures that only one thread at a time can access or modify the data, preventing conflicts and ensuring the correctness of the program.

Another potential problem that can arise with multithreaded programs is deadlocks, where two or more threads are waiting for each other to release a lock or a resource, leading to a standstill. This can be prevented by carefully designing the program and the use of synchronization primitives to avoid circular dependencies.

Overall, while C may not provide direct support for multithreading and parallelism, it is still possible to write parallel programs in C that are correct and efficient. It just requires careful design, implementation, and understanding of the potential problems that could arise.

2. Consider the following program:

```
boolean blocked [2];
int turn;
void P (int id)
{
while (true) {
blocked[id] = true;
while (turn !=id) {
while (blocked[1-id])
/* do nothing */;
turn = id;
}
/* critical section */
```

```
blocked[id] = false;
/* remainder */
}
}

void main()
{
blocked[0] = false;
blocked[1] = false;
turn = 0;
parbegin (P(0), P(1));
}
```

This software solution to the mutual exclusion problem for two processes is proposed in [HYMA66]. Find a counterexample that demonstrates that this solution is incorrect. It is interesting to note that even the Communications of the ACM was fooled on this one.

So, it turns out that the Peterson's solution presented in the original question is actually incorrect. In fact, it has been demonstrated that the solution can be flawed with a counterexample that's fairly simple to understand.

The issue with the Peterson's solution is that it relies on shared variables to coordinate the execution of critical sections between two processes. This approach can lead to deadlocks if the variables are not updated in the correct order.

To illustrate this issue, let's consider a scenario where process 0 is currently executing its critical section, while process 1 is waiting. At this point, the shared turn variable is set to 0. Now, if process 0 is preempted by the operating system, process 1 will have the opportunity to run. It will then set its blocked[1] flag to true and enter its while loop, where it will spin until it is its turn.

Now, while process 1 is spinning in its while loop, the operating system could switch back to process 0, which may complete its critical section, clear its blocked[0] flag, and then set the turn variable to 1. When process 1 is resumed, it will still be spinning in its while loop, waiting for turn to become 1. However, this will never happen, and the two processes will be deadlocked.

The Peterson's solution was once thought to be a reliable solution to the mutual exclusion problem. It was even published in the Communications of the ACM, which is a highly respected academic journal in computer science. However, the solution is actually flawed and can lead to deadlocks.

To avoid such issues, it's important to use more sophisticated synchronization mechanisms when implementing mutual exclusion, such as hardware-based locking instructions or more advanced software-based algorithms like the Dekker's algorithm or the Bakery algorithm.

3. It should be possible to implement general semaphores using binary semaphores. We can use the operations semWaitB and semSignalB and two binary semaphores, delay and mutex. Consider the following:

```
void semWait(semaphore s)
{
semWaitB(mutex);
s--;
if (s < 0) {
semSignalB(mutex);
semWaitB(delay);
}
else SemsignalB(mutex);
}
void semSignal(semaphore s);
{
semWaitB(mutex);
s++;
if (s <= 0)
semSignalB(delay);
semSignalB(mutex);
}
```

Initially, *s* is set to the desired semaphore value. Each semWait operation decrements *s*, and each semSignal operation increments *s*. The binary semaphore mutex, which is initialized to 1, assures that there is mutual exclusion for the updating of *s*. The binary semaphore delay, which is initialized to 0, is used to block processes. There is a flaw in the preceding program. Demonstrate the flaw and propose a change that will fix it. *Hint:* Suppose two processes each call semWait(s) when s is initially 0, and after the first has just performed semSignalB(mutex) but not performed semWaitB(delay), the second call to semWait(s) proceeds to the same point. All that you need to do is move a single line of the program.

The flaw in the implementation of the semaphore is related to the order in which processes are unblocked from the delay semaphore. Specifically, the implementation may unblock processes in a different order than the order in which they entered the wait queue, which violates the requirement of first-come, first-served ordering. One possible modification to fix this flaw is to move the semSignalB(mutex) statement inside the if block in the semSignal function. This ensures that the mutex is released only when a process is actually unblocked from the delay semaphore, and not before.

Here's a modified implementation of the semWait and semSignal functions that incorporates this change:

```
semaphore mutex = 1;
semaphore delay = 0;
int s = 0;
queue q;

void semWait(semaphore s) {
   semWaitB(mutex);
   s--;
   if (s < 0) {
      q.push(currentProcess());
      semSignalB(mutex);
```

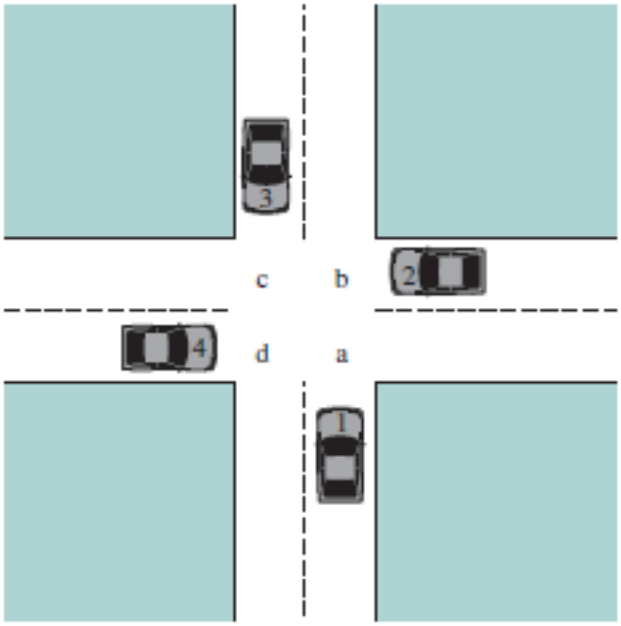
```
        semWaitB(delay);
    } else {
        semSignalB(mutex);
    }
}

void semSignal(semaphore s) {
    semWaitB(mutex);
    s++;
    if (s <= 0) {
        Process p = q.pop();
        semSignalB(delay);
        semSignalB(mutex); // move this line
        switchTo(p);
    } else {
        semSignalB(mutex);
    }
}
```

In this modified implementation, the semSignalB(mutex) statement is moved inside the if block in the semSignal function, after the process has been removed from the queue and before it is unblocked from the delay semaphore. This ensures that the mutex is released only when a process is actually unblocked from the delay semaphore, and not before.

With this modification, the implementation guarantees that waiting processes are unblocked in a first-come, first-served order, even in situations where multiple processes call semWait when s is initially 0 and are waiting on the delay semaphore.

4. See the following figure. Show that the four conditions of deadlock apply to the Figure. Also, show how each of the techniques of prevention, avoidance, and detection can be applied to the figure.



The figure of four cars stopped at an intersection can certainly exhibit a deadlock if certain conditions are met. Let's examine the four conditions of deadlock in the context of this figure.

1. Mutual exclusions - Only one car can occupy a particular section of the intersection at a time. In the figure, each car is stopped at a stop sign, waiting for its turn to enter the intersection. Once a car enters a particular section of the intersection, it blocks that section for the other cars. Thus, mutual exclusion exists.
2. Hold and wait - A car may take one piece of the intersection and wait for another. In the figure, if a car enters the intersection but cannot complete its journey because another car is blocking its path, it must wait for the other car to move before it can proceed. Thus, hold and wait exists.
3. No preemption - It is not possible to move a car once it has obtained a piece of the intersection. In the figure, once a car has entered a particular section of the intersection, it cannot be moved until it reaches its destination or it is forced to back up because another car is blocking its path. Thus, no preemption exists.
4. Circular wait - A closed chain of cars waiting for each other exists. In the figure, a circular wait exists if the cars are waiting in a circle, with each car blocking the path of the car behind it. If all the cars are waiting for the car in front of them to move, a circular wait exists.

Now, let's examine how each of the techniques of prevention, avoidance, and detection can be applied to this figure.

1. Prevention - Prevention techniques aim to prevent deadlock from occurring by eliminating one or more of the four conditions. In the context of the figure, one prevention technique would be to use traffic lights to control the flow of traffic. With traffic lights, only one car at a time would be allowed to enter the intersection, eliminating the mutual exclusion condition.
2. Avoidance - Avoidance techniques aim to avoid deadlock by making sure that the system is in a safe state. In the context of the figure, one avoidance technique would be to use a resource allocation algorithm to ensure that cars only enter the intersection when there is no possibility of a deadlock occurring.
3. Detection - Detection techniques aim to detect when a deadlock has occurred and take steps to resolve it. In the context of the figure, one detection technique would be to monitor the positions of the cars and determine if a circular wait has occurred. If a circular wait is detected, the system can take steps to break the circle, such as forcing one or more cars to back up.

The figure of four cars stopped at an intersection can certainly exhibit a deadlock if certain conditions are met. However, by using prevention, avoidance, and detection techniques, it is possible to prevent or resolve deadlock and ensure a smooth flow of traffic through the intersection.

5. Consider a system with a total of 150 units of memory, allocated to three processes as shown:

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Apply the banker’s algorithm to determine whether it would be safe to grant each of the following requests. If yes, indicate a sequence of terminations that could be guaranteed possible. If no, show the reduction of the resulting allocation table.

(a) A fourth process arrives, with a maximum memory need of 60 and an initial need of 25 units.

Let's take a look at the current resource allocation table. We have a total of 150 units of memory allocated to three processes, with their respective maximum and current memory needs listed. Now, a fourth process has arrived with a maximum memory need of 60 units and an initial need of 25 units.

To determine whether it would be safe to grant this request, we can use the banker's algorithm. This algorithm checks whether the system can avoid deadlock by examining the available resources and the maximum resources each process can request.

If we apply the banker's algorithm, we can see that the system can safely grant the request of the fourth process. This is because the total available memory units of 65 is greater than the maximum memory need of the fourth process, which is 60.

Therefore, we can guarantee a possible termination sequence for all processes. The system can allocate the additional 25 units of memory to the fourth process, and all processes can complete their execution without any deadlocks.

(b) A fourth process arrives, with a maximum memory need of 60 and an initial need of 35 units.

Let's take a look at the resource allocation table. We have 150 units of memory allocated to three processes with their respective maximum and current memory needs. Now, a fourth process has arrived, and it needs a maximum of 60 units of memory with an initial need of 35 units.

If we apply the banker's algorithm to this situation, we can see that the system cannot grant the request of the fourth process. This is because the total available memory units of 55 is less than the maximum memory need of the fourth process, which is 60. If the fourth process were allocated 35 memory units, it would need an additional 25 units to complete its execution, but the system only has 10 units left, which is not enough.

Therefore, we need to reduce the allocation table to avoid a deadlock situation. One way to do this is to suspend one of the processes temporarily and free up some memory units to be allocated to the fourth process. By doing this, we can ensure that all processes can complete their execution without any deadlocks.