Operating Systems: Exercises on processes synchronization

1. The pseudocode below illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm is used in a concurrent environment, answer the following questions:

   (a) What data have a race condition?

   (b) How could the race condition be fixed?

```
push(item) {
  if (top < SIZE) {
    stack[top] = item;
    top++;
  }
  else ERROR
}

pop() {
  if (!empty()) {
    top = top - 1;
    return stack[top];
  }
  else ERROR
}
is empty() {
  if (top == 0) return true;
  else return false;
}
```

2. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
  if (amount > highestBid)
    highestBid = amount;
}
```

   Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring

3. The compare and swap() instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example below presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of Node elements with top representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
   value_t data;
   struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
   Node *old_node;
   Node *new_node;

   new_node = malloc(sizeof(Node));
   new_node->data = item;

   do {
      old_node = top;
      new_node->next = old_node;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);
}

value_t pop() {
   Node *old_node;
   Node *new_node;

   do {
      old_node = top;
      if (old_node == NULL)
        return NULL;
      new_node = old_node->next;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);

   return old_node->data;
}
```

4. Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

   if ($getValue(\&sem) > 0$)
   wait(&sem);

   Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

5. A process B must do an operation opB() only after a process A has done operation opA(). How can you guarantee this using semaphores?

6. (a) Give a brief description of how an atomic adder should work.

   (b) Describe how the code below implements an atomic adder in the context where this code is used by several threads

```
Bool cas(int *p, int old, int new){
  if (*p ≠ old) return false;
  *p = new;
  return true;
}
int add(int *p, int a){
  done = false;
  while (not done) {
  value = *p;
  done = cas(p, value, value + a);
  }
  return value + a;
}
int main(){
  int *p, a;
  add(*p,a);
  return 0;
}
```

7. What happens in the following pseudocode if

   (a) the semaphores S and Q are both initially 1?

   (b) the semaphores S and Q are both initially 0?

   (c) one semaphore is intialized to 0 and the other one to 1?

```
 Process 1                    Process 2
for ( ; ; ) {                for ( ; ; ) {
   wait(S);                    wait(Q);
   print(a);                   print(b);
   signal(Q);                  signal(S);
}                             }
```

8. Consider the following 3 processes that run concurrently

P_1	P_2	P_3
print(Y)	print(O)	print(U)
print(ARE)	print(OK)	print(NOW)

Use semaphores and semaphore initializations such that the result printed is Y O U ARE OK NOW

9. Consider the following 3 processes that run concurrently

P_1	P_2	P_3
OP1	OP2	OP3

You want the processes to have the following behaviors: Process P_3 must execute its operation OP3 before any of the two other processes. Then processes P_1 and P_2 may execute their operation in any order. Use semaphores to impose this behavior, don't forget to provide the initial values of the semaphores

10. Consider the following two processes that run concurrently and where initially $y = z = 0$

P_1	P_2
int x;	y = 1;
x = y + z;	z = 2;

(a) What are the possible final values for x?

(b) Is it possible, using semaphore, to have only two values for x? If so, list the two values and explain how you can get them.

11. Consider the following two processes that run concurrently

P_1	P_2
print(A);	print(E);
print(B);	print(F);
print(C);	print(G);

Insert semaphores to satisfy the following properties. Don't forget to provide the initial values of the semaphores

(a) Print A before F

(b) Print F before C

12. Consider the following two threads:

T_1 = while true print Y
T_2 = while true print Z

Add semaphores such that at any moment the number of Y or Z differs by at most 1. The solution should allow strings such as: YZZYYZZYZY