Operating Systems: Solutions to 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? Sol. The variable "top"

   (b) How could the race condition be fixed? Sol. By making the update of this variable a critical section. The code line of $top++$ should be proceeded by a synchronization such as wait(mutex) followed by signal(mutex) where mutex is a binary semaphore initialized to 1. Similarly with $top--$ using the same mutex semaphore.

```
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;
}
```

Note however that the program above will not work even if top is protected by a synchronization primitive, as by the example below:

```
push(item) {
  if (top < SIZE) {
    stack[top] = item;
    request mutex;
    top++;
    release mutex;
```

```
  }
  else ERROR
}
```

Assume two threads, T1 and T2.

(a) T1 calls push() when $top = SIZE - 1$. While T1 executes, it is de-scheduled after executing the instruction "if ($top < SIZE$)".

- Since $top < SIZE$ is true, T1 has entered the if condition.

(b) Then T2 is scheduled, which also calls push(), but T2 is de-scheduled after returning from push(). In this case $top = SIZE$

(c) Once T1 is re-scheduled, $top < SIZE$ is false, nonetheless T1 executes $stack[top] = item$;

To make this program to work we need to treat push() and pop() as each a critical section (removing the critical section on top). However, for this to work, we must use the same synchronization primitive, thus only one push() or one pop() can run at a time, which could be an inconvenient in parallel computer architectures.

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

Sol: Updating the variable highestBid is a critical section. We can ensure the update is atomic by inserting a mutex into the program before and after the instruction $highestBid = amount$;. However, like the previous question, this progrm will not work properly because a thread $T1$ can be de-scheduled just after entering the if condition, then a thread $T2$ may make highestBid larger than amount in thread $T1$. When $T1$ is re-scheduled, if (amount > highestBid) is no longer true, but this will be ignored by thread $T1$ as it is already in the if condition.

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? Sol: Yes it is as explained in class

```
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.

   Sol: It is not useful. As the return value is received, there is no guarantee it hasn't been changed in the meantime by another process. So

if getValue() is called and the semaphore is free in which case the calling process would not have waited using the wait() function, or the semaphore become busy after the value returned by getValue() in which case the process wait the same as if it had called wait().

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?

   Sol: Need a binary semaphore mutex initialized at 0. The instruction wait(mutex) is executed before opB, and the instruction signal(mutex) is executed after the instruction opA.

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? Sol. a given process is no more than one iteration ahead of the other
   (b) the semaphores S and Q are both initially 0? Sol. Deadlock occurs
   (c) one semaphore is initialized to 0 and the other one to 1? Sol. The processes proceed in strict alternation

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

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

Sol: Either process might execute its wait statement first. The semaphores ensure that a given process is no more than one iteration ahead of the other. If one semaphore is initially 1 and the other 0, the processes proceed in strict alternation. If both semaphores are initially 0, a deadlock occurs.

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

Sol: s1 = 0; s2 = 0; s3 = 0

P_1	P_2	P_3
	wait(s2)	wait(s3)
print(Y)	print(O)	print(U)
signal(s2)	signal(s3)	signal(s1)
wait(s1)	wait(s2)	wait(s3)
print(ARE)	print(OK)	print(NOW)
signal(s2)	signal(s3)	

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

Sol: Need 2 binary semaphores s_1 and s_2 initialized to 0

P_1	P_2	P_3
wait(s_1)	wait(s_2)	OP3
		signal(s_1)
		signal(s_2)
OP1	OP2	

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? Sol. 0, 1, 2 and 3

(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. Sol: From the semantic of the code, either y+z is performed with the initial value of the variables or after they have been modified by process P_2. Using semaphores we can select either way by having process P_2 blocked so that P_1 executes first, thus x = 0 or block process P_1 until process P_2 has executed, then x = 3.

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

Sol: We can use two binary semaphores s_1 and s_2 initialized to 0

P_1	P_2
print(A);	print(E);
signal(s_1);	
	wait(s_1);
print(B);	print(F);
	signal(s_2);
wait(s_2);	
print(C);	print(G);

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

    Sol: Need two binary semaphores s_1 and s_2 initialized to 1

    T_1 = while true wait(s_1); print Y; signal(s_2);
    T_2 = while true wait(s_2); print Z; signal(s_1);