1) Producer consumer problem

Give a possible implementation for the producer/consumer problem with semaphors, assuming that the buffer in which the data is written away is now finite (when buffer is full, producer waits)

solution:

producer	consumer
while(true)	while(true)
produce	wait(n)
wait(e)	wait(s)
wait(s)	take
append	signal(s)
signal(s)	signal(e)
signal(n)	consume

2) Concurrency

Consider the following code:

```
const int n = 50;
int tally;
void total() {
    int count;
    for (count = 1; count <= n; count++) {
        tally=tally+1;
    }
}

void main() {
    tally = 0;
    parbegin(total(),total());  RUN IN PARALLEL
    cout << tally << endl;
}
```

1) what is the minimum and maximum value that the variable tally can assume, at the end of the program?

solution:

consider that `tally=tally+1` is not an atomic instruction so it consists of read, increment, write

- max: 100
- min: 2

P_1: read(0)

P_2: read(0), 49 iterations, write(49)

P_1: increment, write(1)

P_2: read(1)

P_1: read(1), 49 iterations, write(50)

P_2: increment, write(2)

2) Suppose N parallel instances of total are executed, what changes?

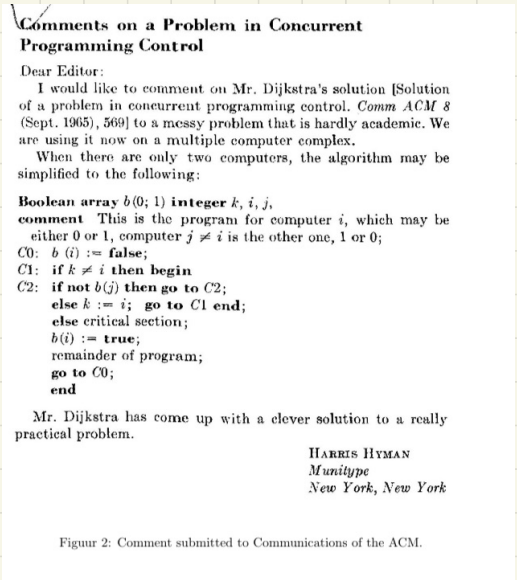
solution:

- max: 50·N
- min 2:

P_1: read(0)
P_2: read(0), 49 it., write(49)
P_1: increment, write(1)
P_2: read(1)
P_1: read(1), 49 it., write 50
$P_3 \dots N$: read(0), 50 it., write(50)
P_2: increment, write(2)

3) Paper review

Paper review is one of the many activities of researchers. The purpose of this is to 'check' a paper, including its correctness, before it can be published in a magazine, or presented at a conference. The following comment was submitted to the fairly highly regarded journal Communications of the ACM in 1965. Review this comment, in Figure 2, for correctness. (Tip: find an output tracking order, where two processes gain access to the critical section at the same time)



Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

Boolean array $b(0; 1)$ **integer** k, i, j,
comment This is the program for computer i, which may be either 0 or 1, computer $j \neq i$ is the other one, 1 or 0;
$C0$: $b\ (i)$:= **false**;
$C1$: **if** $k \neq i$ **then begin**
$C2$: **if not** $b(j)$ **then go to** $C2$;
else $k := i$; **go to** $C1$ **end**;
else critical section;
$b(i)$:= **true**;
remainder of program;
go to $C0$;
end

Mr. Dijkstra has come up with a clever solution to a really practical problem.

HARRIS HYMAN
Munitype
New York, New York

Figuur 2: Comment submitted to Communications of the ACM.

the described algorithm is:

```
b[i]=0
while(k!=i){
    while(!b[j]){}; //busy wait
    K=i
}
<critical section>
b[i]=1
```

two processes can access critical section at the same time when a process switch occurs just before K=i

4) Binary semaphores

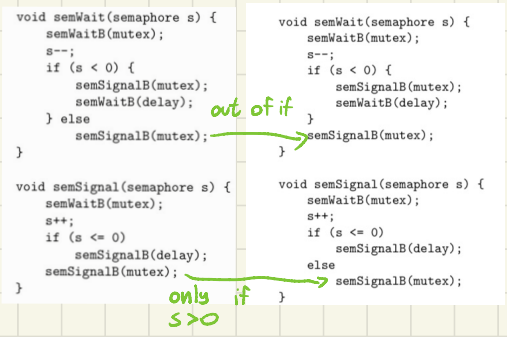
```
void semWait(semaphore s) {
    semWaitB(mutex);
    s--;
    if (s < 0) {
        semSignalB(mutex);
        semWaitB(delay);
    } else
        semSignalB(mutex);
}

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

This is a possible implementation, which realizes counting semaphores using binary ones.

There still is a problem with this code, find it and fix it

solution:

suppose two processes start semWait. Both are interrupted just before semWaitB(delay).
In addition, suppose that two other processes start semSignal and both go trough this function completely
Now semSignalB(delay) has been executed twice. However these are binary semaphores so when the first two processes continue and execute semWaitB(delay), only one of them will be able to continue
The other has to wait unjustly.



5) Mutual exclusion

Given the code answer the questions

```
1  globale variabelen:
2  int turn = -1;
3  bool busy = false;
4
5  thread_i:
6      while(turn != i) {
7        while(busy) {
8          turn = i;
9        }
10       busy = true;
11     }
12   }
13
14   <critical section>
15
16   busy = false;
```

1) is mutual exclusion guaranteed for the critical section? explain.
2) Can a deadlock happen?
3) Is starvation possible? why?

1) Thread i first demands his turn, then it checks whether no one else is already busy, if not he says that he's busy himself, then checks whether no one else has claimed the turn in the meantime. This order ensures that a thread can only leave the outer loop when it's his turn and busy==true. This means no one can get out of the inner loop if someone has gotten out of the outer.

Furthermore, if we assume that several threads have gotten out of the inner loop but still have to get out of the outer loop, only 1 of those can get out of the outer; namely the one who performed the last [turn = i], all the others go to line 7 because it's not their turn, so back in the loop

2) Deadlock is possible. for example:
Suppose T_1 switches to T_2 when T_1 is between line 10 and 11 (in other words busy=true just happened) and T_2 is just behind line 8, T_2 will then run turn=2 and then check if busy=true
That is true so the thread will stay in the loop.
If we then switch back to T_1 it will check if turn!=1, which is true because T_2 has set it to 2.
So T_1 is inside the outer loop, and then goes back in the inner loop (busy is true)
Both threads are now in the inner loop, this will remain until another thread sets busy=false

3) Starvation can occur.
Suppose T_1 switches to T_2, when T_1 is just behind line 10 (immediately after the inner loop)
Now there is nothing stopping other threads from modifying the turn value and surpass T_1, even tho they arrived later

6) Model of an amusement park

Consider an amusement park's attraction: a 1-person cart through a haunted house.
The attraction has n carts on a fixed track (they cannot overtake) and only the full carts drive around; the others wait for passengers in the station
The m visitors walk around for a while, arrive at the haunted house, possibly wait for a cart in a FIFO queue, get in the first available one, ride, get out and do smth else.

If we both model visitors and the carts as processes that have to synchronize (using semaphores) we should prevent that:

- a cart leaves when it's not full
- a cart leaves when a passenger is getting in/out
- passengers cannot leave during the ride

Given the global variables

```
cars_avail = semaphore initialized on 0;
passenger_safe[] = array of n semaphores initialized on 0;
ride_done[] = array of n semaphores initialized on 0;
```

code for the carts (complete)

```
car_j:
while (turned_on) {
  signal(cars_avail)
  <pick up human i>
  wait(passenger_safe[j])
  <drive around with human i>
  signal(ride_done[j])
  <drop off human i>
  wait(passengers_safe[j])
}
```

code for the passengers (incomplete)

```
human_i:
  ...
  <wander around in park>
  ...
  <get into car j>
  ...
  <get scared>
  ...
  <get out of car j>
  ...
  <complete visit to park>
  ...
```

- complete the code for passengers (only known inst. for sem.)

```
human_i:
  <wander around in park>
  wait(cars_avail)
  <get into car j>
  signal(passenger_safe[j])
  <get scared>
  wait(ride_done[j])
  <get out of car j>
  signal(passenger_safe[j])
  <complete visit to park>
```

- what changes if carts can take K people?

  modify the code. Suppose people cannot communicate and they don't know how many passengers are already in the cart.

```
cars_avail = semaphore initialized on 0;
passenger_safe[] = array of n semaphores initialized on 0;
ride_done[] = array of n semaphores initialized on 0;

car_j:
while (turned_on) {
  for (int p = 0; p < k; p++) {
    signal(cars_avail)
  }
  <pick up human i..i+k>
  for (int p = 0; p < k; p++) {
    wait(passenger_safe[j])
  }
  <drive around with human i..i+k>
  for (int p = 0; p < k; p++) {
    signal(ride_done[j])
  }
  <drop off human i..i+k>
  for (int p = 0; p < k; p++) {
    wait(passenger_safe[j])
  }
}

human_i:
  <wander around in park>
  wait(cars_avail)
  <get into car j>
  signal(passenger_safe[j])
  <get scared>
  wait(ride_done[j])
  <get out of car j>
  signal(passenger_safe[j])
  <complete visit to park>
```

← indicates if the passenger is either on the cart or in the cart

at the start K seats available

picks up K people

the passengers of j are on board

the ride is finished for K passengers

← passSafe[j]=0 after they entered

- Are there boundaries to K and m?

  if K>m the few passengers that want to take a ride would wait forever for a cart to be full