Ekrem Kocaguneli

CS550 HW#1

Question 1: The online notes present a correct solution to the mutual exclusion problem for N processes by Eisenberg and McGuire. No analysis is given. Provide an analysis showing that each of Dijkstra's four conditions is met by this algorithm.

Answer 1: The four conditions of Dijkstra that are to be satisfied are: Mutual exclusion, progress,deadlock and bounded waiting. Below is the analysis of Eisenberg and McGuire algorithm w.r.t. these 4 conditions:

- **Mutual Exclusion is satisfied:** The entrance to critical section is governed by 2 variables (*flag* and *turn*) and the entry protocol. Before claiming the turn variable we should have set our state (i.e. flag[i]) to ACTIVE. After setting our state to ACTIVE, we are allowed to claim *turn* variable only if there are no other ACTIVE processes before us and if the process currently holding the process has the IDLE state. If those conditions are satisfied, then we claim the turn, so that only one process at a time is in allowed to enter critical section. When we gain the *turn* variable, the progress that used to hold it is in its EXIT section, giving the turn to us, i.e. no two processes can have the *turn* variable at the same time.
- **Progress requirement is satisfied:** Even if a process finds that all the processes before it are IDLE, it sets its state to ACTIVE only tentatively and makes another check before going into the critical section. So if there are other processes that changed their state to ACTIVE in the mean time, then the process understands that there are competing processes for the critical section. This allows another process that started its scan later but that belongs before the tentatively ACTIVE process to be able to enter the critical section, so that all the processes can progress.
- **Deadlock is not possible:** There is always one process that has the *turn* variable (even when there are no ACTIVE processes asking for the resource, the process that used the turn variable the last time will keep it) and whenever the flag of the process that has the turn variable is IDLE and there is a process whose flag became ACTIVE, this ACTIVE process will eventually be discovered and the turn variable will be passed to the new active process, so there will not be any deadlocks.
- **Bounded waiting is satisfied:** When a process "*p"* with the flag[p]=ACTIVE exists, then no process whose index is not between "*turn"* and "*p"* can enter critical section. On the other hand any process whose index is between "*turn"* and "*p"* can enter its critical section, because system always progresses and *"turn"* variable becomes closer to *"p"*. At the end "*turn"* becomes "*p"* or there are no ACTIVE processes between "*turn"* and "*p"*, whose result is that process p is allowed to enter the critical section; so that process p does not wait indefinitely.

Question 2: Write a monitor providing procedures for customers and for the restaurant host to carry out this system as effectively as possible.

Answer 2:

The implementation of my monitor provides procedures to seat customers to large or small tables or to send them to another restaurant.

The amount of condition variables in my implementation is 3: large_table_free represents that there is a free large table, small_table_free represents there is a free small table and customer_present represents that there is a customer waiting in the line.

The customer is implemented as a record with 3 fields: request, reservation and next. Request is an integer variables saying how many seats are requested by each customer. Reservation is an integer that can be 1 (if reservation was made) or 0 (if there was not a reservation). Next is a pointer that points to next customer in line, so the line is basically a linked list of customers. If there is only one customer in line then the next pointer is null. If we want to put a customer (with reservation) to the front of the line, we make its pointer to point to the first customer in line. If we want to put a customer to the end of the line, we update the last customer's pointer (which was null) to point to the new customer.

The large and small tables are implemented as arrays of size 20 and 5 respectively and they are treated as circular buffers (similar to producer-consumer problem). The variables large_in, small_in and large_out, large_in give the index of the next position for seating-in and seating-out customers (if any) from large and small tables. Variables large_count and small_count give the number of customers seated at large and small tables respectively.

MONITOR:

DATA:

```
condition large_table_free, small_table_free, customer_present;
record customer: /* record with 2 fields */
        int customer.request; /* tells how many seats are requested */
        int reservation; /* 1 if reservation was made, 0 otherwise */
        pointer customer.next; /* points to next customer in line */
int customer_count;           /* keeps number of customers in line */
anytype large_tables[20], small_tables[5];
int large_in, large_out, large_count;
int small_in, small_out, small_count;
```

INITIALIZATION:

```
large_in = 0;large_out = 0; large_count = 0;
small_in = 0;small_out = 0; small_count = 0;
customer_count = 0;
```

PROCEDURES:

```
void seat_in_customer(record customer) {

        /* see if customer's request is suitable for our restaurant */
        if (customer.request > 8)      /* if customer can't sit */
            customer_count = customer_count -1;/*decrement counter*/

        elseif (customer.request)> 4 /*if customer asks large table*/
                /* if no large table is available, wait for one */
                if (large_count >= 20)large_table_free.wait;

                /* seat in the customer */
                large_tables[large_in] = customer;
                large_in = large_in + 1 mod 20;
                large_count = large_count + 1;
                customer_count = customer_count - 1;
                customer_present.signal; /*announce presence of
                                           a new customer*/


        else /*if customer asks small table*/
                /* if no small table is available, wait for one */
                if (small_count >= 5)small_table_free.wait;

                /* seat in the customer */
                small_tables[small_in] = customer;
                small_in = small_in + 1 mod 5;
                small_count = small_count + 1;
                customer_count = customer_count - 1;
                customer_present.signal; /*announce presence of
                                           a new customer*/

}


/*procedure to seat-out customer from a large table*/
void seat_out_customer_large_table(void) {

        record customer;

        /* if there are no customers sitting, then wait for one */
        if (customer_count <= 0) customer_present.wait;

        /* get the next customer to be seated-out of the restaurant
                from a large table*/
        customer = large_tables[large_out];
        large_out = large_out + 1 mod 20;
        large_count = large_count - 1;

        /* announce that there is free space in large tables */
        large_table_free.signal;

        /* delete the customer record to free its space in memory */
        delete(customer);
}
```

```
/*procedure to seat-out customer from a small table*/
void seat_out_customer_small_table(void) {

        record customer;

        /* if there are no customers sitting, then wait for one */
        if (customer_count <= 0) customer_present.wait;

        /* get the next customer to be seated-out of the restaurant
                from a small table*/
        customer = small_tables[small_out];
        small_out = small_out + 1 mod 5;
        small_count = small_count - 1;

        /* announce that there is free space in small tables */
        large_table_free.signal;

        /* delete the customer record to free its space in memory */
        delete(customer);
}
```

CUSTOMER :

```
/*assumption is that customer keep coming at a certain interval */
/*      and that properties of the customer (reservation and    */
/*      request)are assigned at random                          */

repeat {

        /* make a customer come to the restaurant */
        customer = customer_come(); /*this function produces a
                                        random customer*/

        /* put it in the buffer */
        seat_in_customer(customer);

} until done;
```

RESTAURANT HOST:

```
/* assumption is that customer are seated-out of the restaurant */
/* a certain interval randomly from small and large tables      */
repeat {

        /* generate a random variable from 0 to 1 */
        whichTable = rand();
        if (whichTable <= 0.5)
                seat_out_customer_small_table();
        else
                seat_out_customer_large_table();

} until done;
```

Question 3: Show that: 1) The current state is not deadlocked but it is unsafe. 2) If P1 requests 2 more units of R3, then P1, P3, and P4 will be deadlocked.

Answer 3: Below are the 3 tables for 5 processes and 4 resource types. I designed the tables in such a way that resource types other than R3 are in abundance amounts and processes cannot go into deadlock or an unsafe state due to these resources. The only resource type that can result in a deadlock or an unsafe state is R3 (hence R3 is highlighted in all tables).

In table ***Allocation***, the current amount of resources held by each process is listed under the column "Has". The amount of maximum request each process can make is listed under the columns "Max".

In table ***Request***, the current requests of each process from each resource is given.

In table ***Available***, the availability of the resources is given.

1. **Why current state is unsafe?** An unsafe state is a state which may lead to a deadlock situation, but which may as well execute to completion.
    a. **The deadlock Situation:** With the current state of the tables, there is the danger of deadlock; that is, if scheduler executes P1's request of 2 units of R3 first, then P1 will get the available 1 unit of R3 and start waiting on P3 or P4 to release the 1 unit of R3 that they are holding. However, also P3 and P4 are requesting another 1 unit of R3 to complete, so they will also start waiting and will not release their resource of R3 before completion. Since P1, P3 and P4 will wait for 1 unit of R3 and since no process is able to release any resource of R3, there will be infinite waiting, hence the deadlock.
    b. **The completion of unsafe state:** As we saw in previous paragraph, the current state is prone to deadlock. However, if scheduler prefers a different execution order of between P1, P3 and P4; then the deadlock may be avoided. For example if the 1 unit of available R3 is granted to either P3 or P4 first, they will be able to complete (since they need only 1 more unit of R3 to complete) and release their held R3 resources. Then there will be 2 units of R3, which is available for P1 to complete. So possible execution orders between P1, P3 and P4 that can avoid deadlock are:
        - P3, P4, P1
        - P4, P3, P1
        - P3, P1, P4
        - P4, P1, P3

Allocation								
	R1		R2		R3		R4	
	Has	Max	Has	Max	Has	Max	Has	Max
P1	1	2	1	2	1	3	1	2
P2	1	2	1	2	1	2	1	2
P3	1	2	1	2	0	0	1	2
P4	1	2	1	2	0	0	1	2
P5	1	2	1	2	1	2	1	2

Request				
	R1	R2	R3	R4
P1	1	1	2	1
P2	1	1	0	1
P3	1	1	1	1
P4	1	1	1	1
P5	1	1	0	1

Available			
R1	R2	R3	R4
10	10	1	10