Department of Computer Science and Engineering
University of Dhaka.

CSE-3211.
Operating System Lab

Assignment-1
Investor-Producer Synchronization Problem

Submitted by:
Mustafizur Rahman(22)

Submission Date: 30/10/2019.

Submitted to:
Dr.Mosaddek Hossain Kamal Tushar.
Professor,Department of Computer Science and Engineering,
University of Dhaka.
Dr.Md Mamun-or-Rashid.
Professor,Department of Computer Science and Engineering,
University of Dhaka.

Theory :

<u>Critical region:</u>
A critical region is a region of codes where shared resources are accessed. Uncontrolled access to critical region results in race condition. As a result, it creates concurrency problem. To solve this type of concurrency issue, we need to synchronize different threads or processes who want to access critical regions.

<u>Semaphore:</u>
To solve concurrency issue, this technique was first introduced by dijkstra in 1965. This technique has two primitives which are atomic.

1. P(): known as down or wait
2. V(): known as up or signal

Semaphore definition:

```
typedef struct {
        Int count;
        Struct process *L;
}semaphore;
```

Semaphore operations are defined as:

```
P(S):
        S.count-- ;
        If ( S.count < 0 ){
                        Add this process to S.L;
                        Sleep ;
        }
```

```
V(S):
    S.count++ ;
    If ( S.count <= 0 ){
            Remove a process P from S.L;
            wakeup(P) ;
    }
```

Part 1: Questions & Answers :

Thread Questions:

1. What happens to a thread when it exits (i.e., calls thread_exit())? What about when it sleeps?

thread_exit() - thread.c

=> Decrease the reference pointer to the current working directory that the thread is on.
=> Destroy the address space of the thread.
=> Virtual file system for the thread is cleared (VFS).
=> Set priority level high - disables all interrupts. (splhigh())
=> Mark the thread as a zombie and processor picks up another thread.

wchan_sleep(struct wchan *wc) -thread.c

=> Thread is switched to sleep mode and processor picks up another thread.
=> Yield the cpu to another process, and go to sleep, on the specified wait channel WC.

2. What function(s) handle(s) a context switch?

=> thread_switch() -thread.c

3. How many thread states are there? What are they?

thread.h

=> S_RUN - running
=> S_READY - ready to run
=> S_SLEEP - sleeping
=> S_ZOMBIE - zombie; existed but not yet deleted

4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?

=> Current working thread will not be interrupted until interrupts are re-enabled, meaning the code section between disabling and re-enabling interrupts will be executed as if atomically.
=> Calling cpu_irqoff (cpu.c) sets interrupt enable mode bit.
=> Calling splx disables or enables interrupts and adjust the current set priority levels and returns old spl level. Refer to spl.h, for example, spl = splhigh() sets IPL to the highest value, and disables all interrupts.

5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

=> Call to wchan_wakeone (thread.c) wakes up one thread by putting it in CPU's renqueue. When thread_switch happens, the woken up thread may run again.
=> Calling thread_make_runnable (thread.c) will make a thread runnable again.

Scheduler Questions:

6. What function is responsible for choosing the next thread to run?

=> thread_switch() swaps in the next thread on the CPU's run queue.
=> schedule() reshuffles the threads around on the run queue.
=> thread_consider_migration() migrates threads across CPU's run queues in case on imbalance

7. How does that function pick the next thread?

=> thread_switch() simply picks the next thread on the run queue. But we are responsible for implementing schedule().

8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

=> hardclock() calls both schedule() and thread_consider_migration() periodically.

Synchronisation Questions:

9. Describe how thread_sleep() and thread_wakeup() are used to implement semaphores. What is the purpose of the argument passed to thread_sleep() ?

=> TO synchronize sleeping and waiting calls so that no thread will sleep forever.
=> The argument that is passed in is the address of the object (in this case, semaphore) the sleeping thread is associated with.
This is required so that when thread_wakeup is called on the same semaphore, it can selectively wake up only the threads associated with that particular semaphore.

10. Why does the lock API in OS/161 provide lock_do_i_hold() , but not lock_get_holder() ?

=> Because locks have to be released by the same thread that acquires them (and thereby prevent malicious actions).

Part 2: Investor Producer Synchronization Problem:

Problem definition:

In this problem, a customer requests the producer to supply some items to accomplish their every day’s need; by accepting the request, the producer loan money from the bank and use to produce the items. Bank finance the business and get the money back from the producer with a service charge. The producer decides the item price combining bank service charges and profit.
Customer pays before the consumption of the requested items. Then, they finish shopping and back home and sleep. On the other hand, bank and producer evaluate the business at the end of the day.

Proposed Solution:

To solve this problem, first we will try to implement the functions of customer and producer threads and try to identify critical sections and then providing semaphores in right orders to solve synchronization problem and to avoid deadlock.

Function Implementations:

Customer thread :

1. Order_item ():
   In this function, we allocated memory one by one and inserted orders in req_serv_item linked list.
2. Consume_item():
   In this function, a customer consumes the orders he/she has placed at a time and updates it’s customer spending amount.

3. end_shopping():
   In this function, before going to home each customer decreases a variable (stopped_customer) which we maintained to see whether all the customers went home or not.

Producer thread :

1. take_order():
   In this function, each producer takes one-third items at a time and changes each orders order_type as taken for production.
2. calculate_loan_amount():
   In this function, each producer computes the loan amount need from bank which is item_price*item_quantity.
3. loan_request():
   In this function, a producer chooses a bank randomly and takes loan from the bank i.e updates the bank_account's variables.
4. produce_item():
   In this function, each producer updates each item's price with added product_profit for producer and bank_interest for bank.
5. serve_order():
   In this function, each producer updates order_type as SERVICED so that a customer can consume the order.
6. loan_reimburse():
   In this function, each producer repays the loan it took from the bank to produce the item and updates bank_account's variables. Besides, it updates producer_income with product_profit.

Identifying critical regions and use of semaphores :

1. In order_item() function, the code segment which inserts into req_serv_item linked list is a critical region(CR). Because, when more then one customer try to insert into linked list it will create a concurrency problem. Therefore, before going to insert into linked list

we used P() to lock the CR and after inserting N_item_type at a time we used V() to unlock the CR. Besides, in consume_item() function we looped through the req_serv_item to see which orders are serviced. It is also a CR. In both cases, we used the same semaphore.

Semaphore initialization: **mutex = sem_create("mutex", 1);**

2. In take_order() function, No two producer produce order for same item. Therefore, to implement that we used P() before going to the code segment that reads the req_serv_item linked list to get the requested item and after taking the requested item we used V() to unlock the code segment.
   Semaphore initialization: `TO_empty = sem_create("TO_empty",1);`

3. In take_order() function each producer should wait for a order being placed by a customer. To implement this, we used a semaphore which is initialized as 0 and we used P() as soon as we enter into take_order() function and V() is used as soon as a customer places a order in order_item() function. Besides, when a customer ends shopping it also signals take_order() function using V() so that producer thread doesn’t end up in deadlock.

   Semaphore initialization: `TO_full = sem_create("TO_full",0);`

4. We maintained a variable stopped_customer which was initialized with NCUSTOMER. Each time a customer ends shopping decreases the value of the variable by 1. So that producer thread knows whether all the customer went home or not. As it is a critical region, therefore we used a semaphore to solve the concurrency problem.

   Semaphore initialization: `CLA = sem_create("CLA",1);`

5. In loan_request() and loan_reimburse() function we need to access bank_account array. Both the code segment is critical region. Therefore, to solve the concurrency issue we used a semaphore in both the cases.

   Semaphore initialization: `WFP = sem_create("WFP",1);`

6. We maintained an array (service_arr[]) where i'th index of the array will tell us how many orders of i'th customer is serviced. As soon as a producer serves a order in serve_order() function service_arr's index for that customer is incremented by 1. As this is a critical region, therefore we used a semaphore to solve the concurrency issue.

   Semaphore initialization: `mutex3 = sem_create("mutex3",1);`

7. As each customer should wait till it receives all the orders it has placed before going to place a new order. To implement this, we used an array of semaphores where i'th semaphore is used for i'th customer. Besides, all the semaphores are initialized with 0. So, as soon as a customer enters into consume_item() it waits until it is signalled by serve_order() function. serve_order() function signals corresponding customer if it's service_arr's corresponding index is equal to N_ITEM_TYPE.

   Semaphore_initialization:

```
for (int i = 0; i < NCUSTOMER; i++)
{
  CONSUME_ITEM_SEM[i] =sem_create("CONSUME_ITEM_SEM",0);
}
```