Operating System 2

Project Documentation

2022/2023

	Member Name	**Member ID**
Member #1	فاطمة احمد حفظي علي	202000628
Member #2	روان حسام محمد إبراهيم	202000330
Member #3	منى جمال محروس احمد	202000949
Member #4	اسلام جمال قرني على	201900134
Member #5	سندس سليم محمد سليم	202000406
Member #6	سيد عبدالناصر مهدي	201900361
Member #7	مريم عبدالمجيد سمير	202000879
Member #8	رؤى شريف ربيع عبد الغني	202000344

Project Brief

Multiple Sleeping Barber Problem This problem is based on a hypothetical barbershop with k barbers. When there are no customers, all barbers sleep in their chairs. If any customer enters, he will wake up the barber and sit in the customer chair. If there are no chairs empty, they wait in the waiting queue.

Solution Pseducode

Barber function:

```
Start
  While the barber is sleeping    //wait (Customers);
        If there are no customers in the waiting room
              The barber is still sleeping in the barber's chair
        Else
              Pick up the first customer from the waiting chair to the barber's chair
              Then barber cut hair  //cutHair();
              if barber has another customer in waiting chair //signal(barber);
                    pick up the next customer
              else
                    the barber go to sleep until another customer arrive
stop
```

customer function:

```
start
    customer arrive the barber shop
    //wait(mutex);
    If the customer number is greater than the waiting chair number
    // if (count== n+1)
```

```
          The customer will leave the barber shop // signal(mutex); leave();
      Else
          Waiting in a waiting chair //count+=1; signal(mutex);
          go to barber to get cut hair on their turn , customer will follow FIFO
          //signal(customer);
             wait(barber);
            getCuthair ();
    Then leave the shop // wait(mutex);
        count-=1;
        signal(mutex);
stop
```

Examples of Deadlock

- Deadlock is possible if thread_one acquires first_mutex and thread_two acquires second_mutex.
- Thread_one then waits for second_mutex and thread_two waits for first_mutex.

♦ **ReentrantLock**

- provides fair locks, when lock is fair - first lock is obtained by longest-waiting thread in java.
- Constructor to provide fairness - ReentrantLock(boolean fair)
- Creates an instance of ReentrantLock.
- When fair is set true, first lock is obtained by longest-waiting thread.
- If fair is set false, any waiting thread could get lock
- Provide tryLock() method. If lock is held by another thread then method return false in java.

boolean tryLock()

- Acquires the lock if it is not held by another thread and returns true. And sets lock hold count to 1.
- If current thread already holds lock then method returns true.

And increments lock hold count by 1.

- If lock is held by another thread then method return false.

♦ **Deadlock Characterization**

Deadlock can arise if **four** conditions hold at the same time(simultaneously) :

1. Mutual exclusion: only one process at a time can use a resource
2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
3. No preemption: Resources cannot be preempted, a resource can be released only by the process holding it, after that process has completed its task
4. Circular wait: there exists a set {P0 , P1 , …, Pn } of waiting processes such that P0 is waiting for a resource that is held by P1 , P1 is waiting for a resource that is held by P2 , …, Pn–1 is waiting for a resource that is held by Pn , and Pn is waiting for a resource that is held by P0

Solution

How did solve deadlock

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Invalidate one of the four necessary conditions for deadlock :

- ♦ **Mutual Exclusion** – not required for sharable resources (e.g., readonly files); must hold for non-sharable resources
- ♦ **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

• First protocol: Require process to request and be allocated all its resources before it begins execution, impractical for most applications due to the dynamic nature of requesting resources.

• Second protocol: allow the process to request resources only when the process has none allocated to it. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

• Both these protocols have two main disadvantages.

• First , resource utilization may be low, since resources may be allocated but unused for a long period .For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration.

• Second , starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

- ♦ **No Preemption –**

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

Preempted resources are added to the list of resources for which the process is Waiting

Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting .

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions.

It can not generally be applied to such resources as mutex locks and semaphores, the type of resources where deadlock occurs most commonly.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering.

However, establishing a lock ordering can be difficult, especially on a system with hundreds—or thousands—of locks.

Sleeping Barber Solution

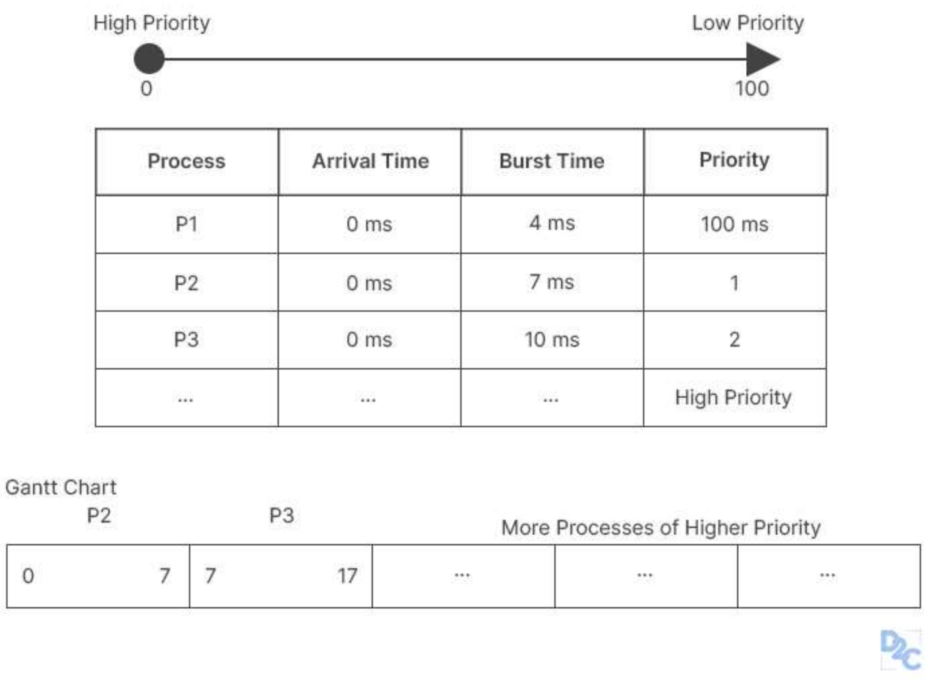
int counts = 0 //number of customers;

mutex = Semaphore(1);

```
customer = Semaphore(0);
barber = Semaphore(0);
    • void customer (void){
 wait(mutex);
 if (counts==n+1) {
 signal(mutex);
 leave();
 }
 counts +=1;
 signal(mutex);
signal(customer);
 wait(barber);
getHairCut();
wait(mutex);
counts -=1;
 signal(mutex); }
void barber (void){ wait(customer);
signal(barber);
 cutHair(); }
```

Absence of deadlock: For this scenario, the deadlock will occur if the customer ends up waiting for the barber and the barber ends up waiting for the customer to arrive. To handle this problem my code has used reentrant locks and after a thread acquires a lock it sleeps for few milliseconds and then release the lock.[4] The code ensures that each thread releases the lock after performing the critical section. I have also used try-catch blocks to handle exceptions. Using reentrant locks, the critical section which is inside the locks can only be accessed by one thread at a time.

Examples of starvation



Process	Arrival Time	Burst Time	Priority
P1	0 ms	4 ms	100 ms
P2	0 ms	7 ms	1
P3	0 ms	10 ms	2
...	...	...	High Priority

In the example, the process P2 is having the highest priority and the process P1 is having the lowest priority. In general, we have a number of processes that are in the ready state for its execution. So, as time passes, if only that processes are coming in the CPU that are having a higher priority than the process P1, then the process P1 will keep on waiting for its turn for CPU allocation and it will never get CPU because all the other processes are having higher priority than P1. This is called Starvation.

Common causes of starvation

In a Starvation, a lower priority process may wait forever if higher priority processes constantly monopolize the processor. Since the low-priority programs are not interacting with anything, it becomes impossible for Starvation to cause a deadlock.

It is a fail-safe method to get out of a deadlock, making it much more important how it affects the system as a whole.

It may occur if there are not enough resources to provide to every process as required.

If a random selection of processes is used then a process may wait for a long time because of non-selection.

In this problem, the barbershop has one barber, one barber chair, and a waiting room with chairs for the customers.

Barber

- The barber sleeps when there is no customer in the waiting room. Therefore, when a customer arrives, he should wake up the barber
- As we've seen from the figure, customers can enter the waiting room and should wait for whether the barber is available or not

Customer

- If other customers keep coming while the barber is cutting a customer's hair, they sit down in the waiting room
- Customers leave the barbershop if there is no empty chair in the waiting room

Solution of Starvation

The solution to this problem includes three semaphores .

First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting).

Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute.

In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.

- **If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping.**

- **At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleep**s.

Starvation Solution to sleeping barber

- The solution to this problem includes three semaphores .

- First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting).

- Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

- When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.

- If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer’s semaphore this wakes up the barber if he is sleeping.

- At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

Absence of Starvation: For this scenario problem of starvation will occur if the customers don’t follow any order for getting a haircut, as some won’t get a haircut even though even after waiting for a long time. To handle this problem in my code I have inserted the customers in a linked list which follows the first in first out property. So, every time a customer sits in a waiting room, they will be selected by the barber in first come first serve basis. We could have also used other data structures like a stack, but the linked list seems like the best choice for this scenario

Real world application

Sleeping Barber problem in call centre

The problem:

If a company doesn't have enough employees in its call center to keep up with calls from customers, it creates a sleeping barber problem.

The same type of sleeping barber problem can be seen with various types of processing situations involving computer technology that is in need of some sort of upgrade in order to handle an increase in demand. For example, if the call center for a company is inundated with inbound calls from customers who want help now, but the number of customer service representatives is not sufficient to keep up with the demand, then a portion of those customers will abandon their calls and possibly seek a relationship with a competitor. In like manner, within the processing systems of a computer network, if the resources devoted to the timely processing of tasks are insufficient for the number of tasks involved, some of those processes will incur what is known as a time-out, and possibly even terminate. The end result is a loss of efficiency that can in turn slow down other processes that were scheduled to follow the ones that timed out.

The solution:

There is no one right way to managing a sleeping barber problem. Depending on the configuration of the computer equipment used, there may be ways to implement new approaches that help to reallocate resources so that tasks may be managed with greater efficiency. For example, reconfiguring the auto attendant services for a call center could result in saving time by automatically routing the next call in the waiting queue to a representative without the need for that representative to manually pick up another pending line. Just as adding additional barbers to a shop make it easier to handle customers seek a haircut, adding more resources in terms of inbound lines and customer service workstations to handle the call volume would also make it easier to handle the volume without triggering a lot of delays. In like manner, expanding resources on a network to support additional functions and processes may also help to minimize a sleeping barber problem and allow the system to operate more efficiently.

Real world application implementation

Customer Class :

```
package sleeping_barbers_shop;

import java.util.Date;

public class Customer implements Runnable{

    int cust_Id;
    Date inTime;

    Call_center callCenter;

    public Customer(Call_center callCenter) {

        this.callCenter = callCenter;
    }

    public int getCustomerId()
{
        return cust_Id;
    }

    public Date getInTime() {
        return inTime;
    }

    public void setcustomerId(int cust_Id) {
        this.cust_Id = cust_Id;
    }

    public void setInTime(Date inTime) {
        this.inTime = inTime;
    }

    public void run()
{
        goForAnswerTheCall();
    }
    private synchronized void goForAnswerTheCall()
{

        callCenter.add(this);
    }
}
```

Cust_representatives class :

```
package sleeping_barbers_shop;


public class Cust_representatives implements Runnable{
     Call_center callCenter;
    int representativesId;

    public Cust_representatives(Call_center callCenter, int
representativesId) {

        this.callCenter = callCenter;
        this.representativesId = representativesId;
    }

    public void run() {

        while(true) {

            callCenter.AnswerTheCall(representativesId);
        }
    }
}
```

CallCenter_run class :

```
package sleeping_barbers_shop;

import static java.util.concurrent.TimeUnit.SECONDS;
import java.util.Date;
import java.util.Random;
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class CallCenter_run {

    public static void main(String[] args) throws
InterruptedException {
      int num_cust_representatives, cust_Id=1,
num_customers, num_Chairs;

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of customer
service representatives (K):");
        num_cust_representatives=sc.nextInt();

        System.out.println("Enter the number of waiting
room"
                + " chairs(c):");
        num_Chairs=sc.nextInt();

        System.out.println("Enter the number of
customers:");
    num_customers=sc.nextInt();

        ExecutorService exec =
Executors.newFixedThreadPool(12);
        Call_center callCenter = new
Call_center(num_cust_representatives,
num_Chairs);
        Random r = new
Random();
```

```
        System.out.println("\ncallCenter opened with "
                +num_cust_representatives+" customer service
representatives(s)\n");

        long startTime  =
System.currentTimeMillis();

        for(int i=1; i<=num_cust_representatives;i++)
{

            Cust_representatives barber = new
Cust_representatives(callCenter, i);
            Thread thbarber = new Thread(barber);
            exec.execute(thbarber);
        }

        for(int i=0;i<num_customers;i++)
{
            Customer customer = new Customer(callCenter);
            customer.setInTime(new Date());
            Thread thcustomer = new Thread(customer);
            customer.setcustomerId(cust_Id++);
            exec.execute(thcustomer);
            System.out.println("\n**************************
****************");

            try {

                double val = r.nextGaussian() * 2000 +
2000;
                int millisDelay = Math.abs((int)
Math.round(val));
                Thread.sleep(millisDelay);

            }
            catch(InterruptedException iex) {
            }

        }
        exec.shutdown();

        exec.awaitTermination(12,
SECONDS);

        long elapsedTime = System.currentTimeMillis() -
startTime;

        System.out.println("\ncallCenter shop closed");
        System.out.println("\nTotal time elapsed in seconds
for serving "+num_customers+" customers by "
                +num_cust_representatives+" customer service
representatives with "+num_Chairs+
                " chairs in the waiting room is: "
                +TimeUnit.MILLISECONDS
                .toSeconds(elapsedTime));
        System.out.println("\nTotal customers:
"+num_customers+
                "\nTotal customers served:
"+callCenter.getTotalAnswerTheCall()
                +"\nTotal customers lost:
"+callCenter.getCustomerLost());

        sc.close();

    }

}
```

Call_center class :

```java
package sleeping_barbers_shop;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
public class Call_center {
     private final AtomicInteger TotalAnswerTheCall = new
AtomicInteger(0);
    private final AtomicInteger customersLost = new
AtomicInteger(0);
    int nchair, noOfRepresentatives,
availableRepresentatives;
    List<Customer> listCustomer;

    Random r = new Random();

    public Call_center(int noOfRepresentatives, int
noOfChairs){

        this.nchair =
noOfChairs;

        listCustomer = new
LinkedList<>();
        this.noOfRepresentatives =
noOfRepresentatives;
        availableRepresentatives = noOfRepresentatives;
    }

    public AtomicInteger getTotalAnswerTheCall() {

        TotalAnswerTheCall.get();
        return TotalAnswerTheCall;
    }

    public AtomicInteger getCustomerLost() {

        customersLost.get();
        return customersLost;
    }

    public void AnswerTheCall(int representativesId)
    {
        Customer customer;
        synchronized (listCustomer)
{


            while(listCustomer.size()==0) {

                System.out.println("\nRepresentatives
"+representativesId+" is waiting "
                        + "for the customer and sleeps in
his chair");

                try {

                    listCustomer.wait();

                }
                catch(InterruptedException iex) {

                    iex.printStackTrace();
                }
            }

            customer =
(Customer)((LinkedList<?>)listCustomer).poll();

            System.out.println("Customer
"+customer.getCustomerId()+
                    " finds the representatives asleep and
wakes up "
                    + "the representatives
"+representativesId);
        }

        int millisDelay=0;
        try {

            availableRepresentatives--
;

            System.out.println("Representatives
"+representativesId+" AnswerTheCall of "+
                    customer.getCustomerId()+ " so customer
sleeps");

            double val = r.nextGaussian() * 2000 +
4000;
            millisDelay = Math.abs((int)
Math.round(val));
            Thread.sleep(millisDelay);

            System.out.println("\nCompleted Answering call
of "+
                    customer.getCustomerId()+" by
Representatives " +
                    representativesId +" in "+millisDelay+ "
milliseconds.");

            TotalAnswerTheCall.incrementAndGet();


            if(listCustomer.size()>0)
{
                System.out.println("Representatives
"+representativesId+
                        " wakes up a customer in the
"
                        + "waiting room");
            }

            availableRepresentatives++;
        }
        catch(InterruptedException iex) {

            iex.printStackTrace();
        }

    }

    public void add(Customer customer) {

        System.out.println("\nCustomer
"+customer.getCustomerId()+
                " enters through the entrance door in the
the shop at "
                +customer.getInTime());

        synchronized (listCustomer) {

            if(listCustomer.size() == nchair) {

                System.out.println("\nNo chair available "
                        + "for customer
"+customer.getCustomerId()+
                        " so customer leaves the shop");

              customersLost.incrementAndGet();

                return;
            }
            else if (availableRepresentatives > 0)
{


                ((LinkedList<Customer>)listCustomer).offer(c
ustomer);
                listCustomer.notify();
            }
            else
{
```

```
                ((LinkedList<Customer>)listCustomer).offer(c
ustomer);

                System.out.println("All Representatives(s)
are busy so "+
                        customer.getCustomerId()+
                        " takes a chair in the waiting
room");

                if(listCustomer.size()==1)
                    listCustomer.notify();
            }
        }
    }

}
```