Manisa Celal Bayar University

Comp. Eng. Depart. CSE3124 Operating Systems Spring (2019–2020)

Catering Scheduling Problem

TERM PROJECT

İbrahim Caner Kartal - 160315007

Mert Dumanlı - 160315002

Scenario

In this project, we will provide a service order for 10 guests participating in the party arrangement we host as a scenario. We will have 3 kinds of food or drink to offer to our guests and we will ensure that each guest eats at least 1 of them. Food or beverage types are as follows.

- Börek
- Cake
- Drink

There are a limited number of items in stock from each unit. There are a total of 15 for Cake and 30 from the others. Of course, the waiter waiting at the desk is waiting at the desk to present these materials to the guests.

The maximum capacity rate for the three plates on each table is 5 pieces. Therefore, restrictions should be introduced that will allow food sharing for each guest. Each guest can take a maximum of 5 pastries and drinks, and 2 slices of cake.

For this reason, in order to apply restrictions, every part purchase step of the guests is checked and placed in the sleep queue for synchronization. The most important stage in the program is synchronization. In the case of coming to the table at the same time and taking one of the food or drinks, there may be a decrease or queue problem. Waiter, who is always at the desk to welcome the guests, is responsible for each plate. It is activated when any of the 3 plates is emptied or 1 unit is left and adds enough to fill the plate from the stock. Since the plate capacities are 5 units, each time the waiter is activated, 4 or 5 units of stock are reduced from the related food or drink. In order to come to the random table at the time of start, all guests start the program as a thread during sleep for random periods.

It will be a good solution to use threads as each instance will be shown as a process in the programming section. Let's talk about the thread working logic.

Multi-Threading

It would be nice if we could focus our attention on performing only one action at a time and performing it well, but that's usually difficult to do. The human body performs a great variety of operations in parallel—or, as we'll say throughout this chapter, concurrently. Respiration, blood circulation, digestion, thinking and walking, for example, can occur concurrently, as can all the senses—sight, touch, smell, taste and hearing.

Computers, too, can perform operations concurrently. It's common for personal computers to compile a program, send a file to a printer and receive electronic mail messages over a network concurrently. Only computers that have multiple processors can truly execute multiple instructions concurrently. Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once. Today's multicore computers have multiple processors that enable computers to perform tasks truly concurrently. Multicore smartphones are starting to appear.

Historically, concurrency has been implemented with operating system primitives available only to experienced systems programmers. The Ada programming language— developed by the United States Department of Defense—made concurrency primitives widely available to defense contractors building military command-and-control systems. However, Ada has not been widely used in academia and industry.

Java Concurrency

Java makes concurrency available to you through the language and APIs. Java programs can have multiple threads of execution, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory. This capability is called multi-threading.

Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several thread states—illustrated in the UML state diagram in Figure 1. Several of the terms in the diagram are defined in later sections. We include this discussion to help you understand what's going on "under the hood" in a Java multithreaded environment. Java hides most of this detail from you, greatly simplifying the task of developing multithreaded applications.

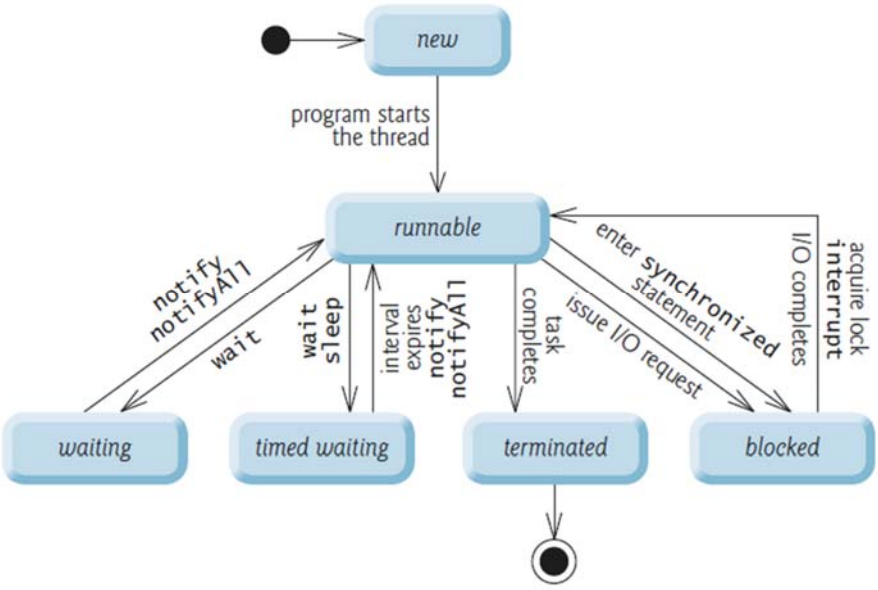

Figure 1. Thread life-cycle UML state diagram

New and Runnable States

A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread, which places it in the runnable state. A thread in the runnable state is considered to be executing its task.

Waiting State

Sometimes a runnable thread transitions to the waiting state while it waits for another thread to perform a task. A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.

Timed Waiting State

A runnable thread can enter the timed waiting state for a specified interval of time. It transitions back to the runnable state when that time interval expires or when the event it's waiting for occurs. Timed waiting and waiting threads cannot use a processor, even if one is available. A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it's waiting for another thread to perform a task. Such a thread returns to the runnable state when it's notified by another thread or when the timed interval expires—whichever comes first. Another way to place a thread in the timed waiting state is to put a runnable thread to sleep. A sleeping thread remains in the timed waiting state for a designated period of time (called a sleep interval), after which it returns to the runnable state. Threads sleep when they momentarily do not have work to perform.

Blocked State

A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the blocked thread transitions to the runnable state, so it can resume execution. A blocked thread cannot use a processor, even if one is available.

Terminated State

A runnable thread enters the terminated state (sometimes called the dead state) when it successfully completes its task or otherwise terminates (perhaps due to an error). In the UML state diagram of Figure 1, the terminated state is followed by the UML final state (the bull's-eye symbol) to indicate the end of the state transitions.

Operating-System View of the Runnable State

At the operating system level, Java's runnable state typically encompasses two separate states (Figure 2). The operating system hides these states from the Java Virtual Machine (JVM), which sees only the runnable state. When a thread first transitions to the runnable state from the new state, it's in the ready state.

A ready thread enters the running state (i.e., begins executing) when the operating system assigns it to a processor—also known as dispatching the thread. In most operating systems, each thread is given a small amount of processor time—called a quantum or time slice—with which to perform its task. Deciding how large the quantum should be being a key topic in operating systems courses. When its quantum expires, the thread returns to the ready state, and the operating system assigns another thread to the processor. Transitions between the ready and running states are handled solely by the operating system. The JVM does not "see" the transitions—it simply views the thread as being runnable and leaves it up to the operating system to transition the thread between ready and running. The process that an operating system uses to determine which thread to dispatch is called thread scheduling and is dependent on thread priorities.

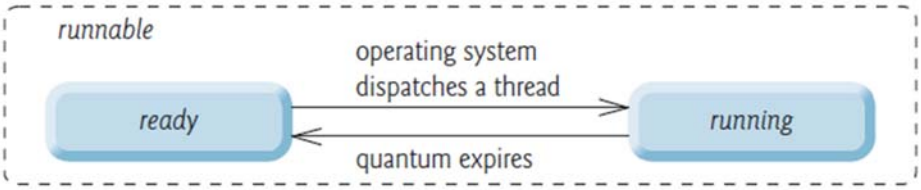

Figure 2. Operating System's internal view of Java's runnable state

Thread Priorities and Thread Scheduling

Every Java thread has a thread priority that helps determine the order in which threads are scheduled. Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. Nevertheless, thread priorities cannot guarantee the order in which threads execute.

It's recommended that you do not explicitly create and use Thread objects to implement concurrency, but rather use the Executor interface. The Thread class does contain some useful static methods, which you will use later in the chapter.

Most operating systems support time slicing, which enables threads of equal priority to share a processor. Without time slicing, each thread in a set of equal-priority threads runs to completion (unless it leaves the runnable state and enters the waiting or timed waiting state, or gets interrupted by a higher-priority thread) before other threads of equal priority get a chance to execute. With time slicing, even if a thread has not finished executing when its quantum expires, the processor is taken away from the thread and given to the next thread of equal priority, if one is available.

An operating system's thread scheduler determines which thread runs next. One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there's more than one highest-priority thread, ensures that all such threads execute for a quantum each in round-robin fashion. This process continues until all threads run to completion.

When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread (an operation known as preemptive scheduling). Depending on the operating system, higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such indefinite postponement is sometimes referred to more colorfully as starvation. Operating systems employ a technique called aging to prevent starvation—as a thread waits in the ready state, the operating system gradually increases the thread's priority, thus ensuring that the thread will eventually run.

Thread Synchronization

When multiple threads share an object and it's modified by one or more of them, indeterminate results may occur (as we'll see in the examples) unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it's unclear which thread's update takes effect. When this happens, the program's behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won't. In either case, there'll be no indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed. This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion.

Monitors

A common way to perform synchronization is to use Java's built-in monitors. Every object has a monitor and a monitor lock (or intrinsic lock). The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time. Monitors and monitor locks can thus be used to enforce mutual exclusion. If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock before proceeding with the operation. Other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock, at which point the blocked threads may attempt to acquire the lock and proceed with the operation.

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement. Such code is said to be guarded by the monitor lock; a thread must acquire the lock to execute the guarded statements. The monitor allows only one thread at a time to execute statements within synchronized statements that lock on the same object, as only one thread at a time can hold the monitor lock. The synchronized statements are declared using the synchronized keyword:

```
synchronized ( object )
{
   statements
} // end synchronized statement
```

where object is the object whose monitor lock will be acquired; object is normally this if it's the object in which the synchronized statement appears.

If several synchronized statements are trying to execute on an object at the same time, only one of them may be active on the object—all the other threads attempting to enter a synchronized statement on the same object are placed in the blocked state.

When a synchronized statement finishes executing, the object's monitor lock is released and one of the blocked threads attempting to enter a synchronized statement can be allowed to acquire the lock to proceed. Java also allows synchronized methods. Before executing, a non-static synchronized method must acquire the lock on the object that's used to call the method. Similarly, a static synchronized method must acquire the lock on the class that's used to call the method.

Coding

We will examine the coding part as 4 main topics. These will consist of main, guest, table, waiter parts, respectively. Finally, we will consider the outcome.

Main Part

```
public class PartyScheduling {

   static int []trays = {0,0,0};
   static int []total = {30,15,30};//total stock
   static int []min0 = {0,0,0,0,0,0,0,0,0,0};//Borek counter for each guest
   static int []min1 = {0,0,0,0,0,0,0,0,0,0};//Cake counter
   static int []min2 = {0,0,0,0,0,0,0,0,0,0};//Drink counter

    public static void main(String[] args) {
       System.out.println("Welcome to the Organization.");
        Table q = new Table();
       new Waiter(q);
       new Guest0(q);
       new Guest1(q);
       new Guest2(q);
       new Guest3(q);
       new Guest4(q);
       new Guest5(q);
       new Guest6(q);
       new Guest7(q);
       new Guest8(q);
       new Guest9(q);
    }
    }
```

As you can see, the main part of the program is used only for variable definition and thread triggering. The reason why the variables are static is that they will be processed until the end of the program.

Guest Part

```
public class Guest0 implements Runnable {

    Table q;

    public Guest0(Table q) {
        this.q = q;
        Thread t0 = new Thread(this, "Guest0");
        t0.start();
    }

    public void run() {
        while (true) {
            Random rand = new Random();
            int x = rand.nextInt(3);
            int y = rand.nextInt(100) + 1000;
            int z = rand.nextInt(500);
            try {
                Thread.sleep(z);
            } catch (Exception e) {
            }
            if (x == 0 && min0[0] < 5) {
                if (30 - q.sum(min0) > q.zeros(min0) || min0[0] == 0) {
                    if (trays[0] > 0) {
                        q.get(0);
                        System.out.println("Guest0 took a borek.");
                        min0[0] = min0[0] + 1;
                        try {
                            Thread.sleep(y);
                        } catch (Exception e) {
                        }
                    }
                    if (trays[0] == 0) {
                        try {
                            Thread.sleep(y);
                        } catch (Exception e) {
                        }
                    }
                }
            }
```

I think it would be enough to examine only one example for 10 identical guest classes. The Runnable class, which we first noticed in the class definition, is a class that we define as inherit for thread usage. We create an instance named q to use the methods we have defined for the table.

In the constructor structure that enables us to reach methods again for each created guest, the thread definition of that guest is made. Endless while loops are written for the guests who will work continuously within the run structure, which is the thread working method. Two of the 3 random values discarded were used for sleep time. Thus, those who wake up at random time for each guest created can attack plates. The third random number is the variable that determines from which plate the guest will receive food. Then the plates are checked in the if else structures to ensure that guests receive the food.

Table Part

```
public class Table {

    int num;

    public synchronized void put(int num, int x) {
        this.num = num;
        trays[x] = trays[x] + num;
        total[x] = total[x] - num;
        switch (x) {
            case 0:
                if (num == 1) {
                    System.out.println((char) 27 + "[32m" + "Waiter Put : " + num
                } else {
                    System.out.println((char) 27 + "[32m" + "Waiter Put : " + num
                }
                break;
            case 1:
                if (num == 1) {
                    System.out.println((char) 27 + "[31m" + "Waiter Put : " + num
                } else {
                    System.out.println((char) 27 + "[31m" + "Waiter Put : " + num
                }
                break;
            default:
                if (num == 1) {
                    System.out.println((char) 27 + "[34m" + "Waiter Put : " + num
                } else {
                    System.out.println((char) 27 + "[34m" + "Waiter Put : " + num
                }
                break;
        }

    }

    public synchronized void get(int q) {
        trays[q] = trays[q] - 1;
    }

    public synchronized int sum(int[] q) {
        int total = 0;
        for (int a = 0; a < 10; a++) {
            total = total + q[a];
        }
        return total;
    }
```

This class was created to perform the actions of guests and waiters in general. The functions of table food and plate filling actions for waiter are described here. The reason for the instance created in the guest class is the functions in this class.

Waiter Part

```
public class Waiter implements Runnable {

    Table q;

    public Waiter(Table q) {
        this.q = q;
        Thread t = new Thread(this, "Waiter");
        t.start();
    }

    public void run() {
        System.out.println((char) 27 + "[32m" + "*" + "Total number
        System.out.println((char) 27 + "[31m" + "*" + "Total number
        System.out.println((char) 27 + "[34m" + "*" + "Total number
        while (true) {
            Random rand = new Random();
            int y = rand.nextInt(100) + 1000;
            if (total[0] > 0) {
                if (trays[0] == 0) {
                    if (total[0] >= 5) {
                        q.put(5, 0);
                    } else if (total[0] < 5) {
                        q.put(total[0], 0);
                    }
                } else if (trays[0] == 1) {
                    if (total[0] >= 4) {
                        q.put(4, 0);
                    } else if (total[0] < 4) {
                        q.put(total[0], 0);
                    }

                }
            }
            if (total[1] > 0) {
                if (trays[1] == 0) {
                    if (total[1] >= 5) {
                        q.put(5, 1);
                    } else if (total[1] < 5) {
                        q.put(total[1], 1);
                    }
                } else if (trays[1] == 1) {
                    if (total[1] >= 4) {
                        q.put(4, 1);
                    } else if (total[1] < 4) {
                        q.put(total[1], 1);
                    }
```

Waiter, which is a thread just like the guests, works in an infinite loop to make continuous plate control at the desk. Unlike the guests, waiter, which is quite similar to the guest class, never sleeps and uses the table class to fill plates. Of course, if else structures,

it constantly checks for 3 different plates and if any of them are 0 or 1, it adds to the relevant plate by reducing the required amount from the stock.

Result

```
Guest9 took a glass of drink.
Waiter Put : 4 glasses of drink on drink tray.
Guest6 took a slice of cake.
Guest7 took a borek.
Guest5 took a glass of drink.
Guest2 took a glass of drink.
Guest8 took a borek.
Guest3 took a glass of drink.
Guest1 took a borek.
Waiter Put : 3 boreks on borek tray.
Guest7 took a borek.
Guest0 took a glass of drink.
Waiter Put : 2 glasses of drink on drink tray.
Guest4 took a glass of drink.
Guest5 took a glass of drink.
Guest9 took a glass of drink.
Guest8 took a borek.
Guest2 took a borek.
Guest1 took a borek.
ORGANIZATION HAS BEEN FINISHED
Who did that?
#0Guest= Borek: 1 Slices of Cake: 2 Glasses of Drink: 4
#1Guest= Borek: 4 Slices of Cake: 1 Glasses of Drink: 4
#2Guest= Borek: 4 Slices of Cake: 1 Glasses of Drink: 3
#3Guest= Borek: 3 Slices of Cake: 2 Glasses of Drink: 3
#4Guest= Borek: 5 Slices of Cake: 1 Glasses of Drink: 2
#5Guest= Borek: 2 Slices of Cake: 2 Glasses of Drink: 3
#6Guest= Borek: 1 Slices of Cake: 1 Glasses of Drink: 3
#7Guest= Borek: 4 Slices of Cake: 2 Glasses of Drink: 1
#8Guest= Borek: 4 Slices of Cake: 2 Glasses of Drink: 3
#9Guest= Borek: 2 Slices of Cake: 1 Glasses of Drink: 4
MADE BY
Mert DUMANLI    160315002
İbrahim Caner KARTAL    160315007
BUILD SUCCESSFUL (total time: 11 seconds)
```

The program shows each step on the console step by step and the result is quite satisfactory. I think this study helps us to understand multithreading in general terms. Finally, thank you for your time.

References:

- http://campus.murraystate.edu/academic/faculty/wlyle/325/Chapter26.pdf
- https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)