CSD NOTES

UNIT 1.- INTRODUCTION TO CONCURRENT PROGRAMMING

1.-WHAT IS A CONCURRENT PROGRAM?

A **concurrent program** is a collection of activities (threads) that can run in parallel an cooperate to perform a common task.

It is possible to achieve concurrency by two means:

- **Logical parallelism**: one processor with multiprogramming.
- **Real parallelism**: several processors.

Advantages:

- Efficiency.
- Scalability.
- Communication management.
- Flexibility
- Minor semantic gap.

Disadvantages:

- Makes programming more difficult.
- Complex debugging (non-determinism).

2.-CONCURRENT PROGRAMMING IN JAVA

A **thread** is equal to the execution context composed of: JVM, the code and the data.

There are different options to create a thread in Java:

- Implementing Runnable interface (define **run()** method and you have to create a thread object with an object of the class as parameter).
- Extending Thread class (you only must create an object of the class).

A thread executions begins with **t.start()**.

Another methods of the class Thread:

Methods	What do they do?
delay(int n)	Suspends execution for n milliseconds
echo(String s)	Shows text in the screen
setName(String s)	Gives a name to the thread
getName()	Gets the name of the thread
sleep(int n)	Causes the suspension of the thread during n milliseconds
interrupt()	It reactivates a thread that was suspended
join()	Allows a thread to wait to the end of another thread.

currentThread()	Returns a reference to the thread object that is currently running
isAlive()	Returns TRUE if the thread has started and has not finished yet, FALSE otherwise
yield()	Voluntarily leaves the processor and goes to ready queue

QUICK NOTES:

- When the main thread finishes it doesn't end the execution of the children as it does in C language.
- We can use anonymous classes in order to create a thread.
- We have to use **java.util.concurrent** to use threads.

UNIT 2.- TASK SYNCHRONIZATION

1.-COMMUNICATION MECHANISMS

Concurrency implies **parallelism** and **cooperation** among threads. And **cooperation** requires:

- Communication (information interchange).
- Synchronization (setting certain order in specific cases).

The communication mechanisms are:

- Shared memory.
- Message exchange.

The problems caused by this communication are:

- Thread interferences (errors in shared data).
- Memory consistency errors (inconsistent views of what should be the same data).

This can be avoided with a **happens-before** relationship as:

- start().
- join().
- Synchronization mechanisms.

2.-EXECUTION MODEL AND RACE CONDITIONS

A **sentence** Is a sequence of atomic actions that carry out indivisible transformations.

An **atomic action** transforms the state and cannot be divided into smaller actions.

A **steady state** is a state that satisfies all the invariants of the object (consistent).

Normal sequence:

Steady state -> Intermediate states (inconsistent) -> Steady state

A **race condition** is an inconsistent modification of the shared memory.

3.-TYPES OF SYNCHRONIZATION

There are two types of synchronization:

- Mutual exclusion.
- Conditional synchronization.

A section of code is executed in **mutual exclusion** when only one thread can run this section at any time. It is required for shared variables/objects.

Conditional synchronization is fulfilled when a thread must delay its execution until a specific condition holds.

A **critical section** is a fragment of code that can cause race conditions, in other words, is a fragment that accesses to variables or objects shared by more than one thread.

A **lock** is an object with two states (open/closed) and two operations (open/close).

Using locks, we convert a critical section into an atomic action:

> Close lock -> Critical section -> Open lock

In Java, every object has an implicit associated lock. For that, you must label with *synchronized* all methods that are part of the critical section. All methods labelled with *synchronized* are executed in mutual exclusion among them.

Java provides two basic synchronization idioms:

- Synchronized methods (*synchronized* label).
- Synchronized statements (specifying the object that provides the intrinsic lock).

UNIT 3.- SYNCHRONIZATION PRIMITIVES

1.-MONITOR CONCEPT

A **monitor** is a class to define objects that can be safely shared between different threads.

Its methods are executed in mutual exclusion, for that, an entry queue is implemented for waiting there those threads that want to use the monitor when another thread is using it.

Solves synchronization as you can define waiting queues (condition variables) inside the monitor:

- c.wait() -> frees the monitor and waits at the condition c queue.
- c.notify() -> reactivates a thread that is waiting in the condition c queue.

2.-MONITOR VARIANTS

There are different models of monitors (N executes notify() / W waits in condition c):

- **Brich Hansen model**: thread N leaves the monitor. The sentence *notify* must be the last sentence of the method. Waterfall awake cannot be applied here.
- **Hoare model**: thread N waits in a special queue. The special queue has priority over the entry queue. Usually is used one "if" sentence with all the conditions.
- **Lampson-Redell model**: thread W waits in the entry. The condition must be rechecked.

If we use the implicit locks of the objects in java we have to use the following methods:

- wait(): wait on the waiting queue.
- notify(): reactivates one of the threads that waits on the waiting queue.
- notifyAll(): notifies all threads that are waiting.

The drawback is that Java only employs one condition variable per monitor.

3.-NESTED CALLS

Calling from one monitor to a method of another monitor can reduce concurrency and even cause deadlocks.

UNIT 4.- DEADLOCKS

1.-COFFMAN'S CONDITIONS

A **deadlock** occurs where there is a set of threads that cannot evolve because they are waiting to each other.

In order to detect a risk of deadlock, **Coffman's conditions** are used:

- **Mutual exclusion**: while a resource is assigned to a thread, other threads cannot use it.
- **Hold and wait**: resources are requested as needed, so we can have a resource assigned and request another not available (and thus wait for it).
- **No preemption**: an assigned resource can only be released by its owner (it cannot be expropriated).
- **Circular wait**: in the group of threads in deadlock, each of them is waiting for a resource that holds another of the group, and so on until closing the circle.

If all of them hold simultaneously, then there is a risk of deadlock. In case of deadlock, they are all true. In conclusion, they are necessary, but not enough conditions.

2.-GRAPHICAL REPRESENTATION

A RAG (Resource Allocation Graph or Wait-For Graph) is a graphical representation of the state of the system. The representation is as it follows:

- A **circle** represents a thread.
- A **rectangle** represents a resource and contains as much internal points as instances of the resource.
- **Assignment arista**: an assigned resource is represented with an arrow from a specific instance to the thread that employs it.
- **Request arista**: an unresolved request is represented with an arrow that goes from the thread to the resource.

A directed cycle in the RAG means a risk of deadlock. If all resources of the cycle have just 1 instance, then a deadlock occurs. If there is a safe sequence, then there is no deadlock.

3.-SOLUTION STRATEGIES

The solutions to the deadlock problem from best to worst are:

1. **Prevention**: design a system that breaks any Coffman's conditions.
2. **Avoidance**: using s RAG to dynamically consider every request.
3. **Detection and recovery**: *detection*, to monitor the system periodically, *recovery*, if there is a deadlock situation, then it aborts one or some of the activities involved in the deadlock.
4. **Ignore the problem**.

The easiest condition to break is circular wait.

UNIT 5.- OTHER SYNCHRONIZATION TOOLS

1.- INTRODUCTION

Limitations of the Java basic primitives, related with mutual exclusion:

- We cannot specify a maximum time of waiting when requesting the entry to the monitor.
- We cannot ask about the state of the monitor before requesting its entry.
- The tools that ensure mutual exclusion are oriented to blocks.
- We cannot extend its semantics.

Limitations of the Java basic primitives, related with conditional synchronization:

- There can be one single condition in each monitor.
- Java employs the Lampson and Redell variant.

2.- LOCKS

ReentrantLock is the class provided by Java that we use as lock. It implements a reentrant lock. It solves limitations of the *synchronized* label. Usual structure:

```
Lock x = new ReentrantLock();
x.lock();
try {
        //Critical section
} finally {
        x.unlock();
}
```

3.- CONDITION VARIABLES

The **Condition** interface allows declaring any number of condition variables inside a lock.

The **Lock** object acts as a factory for the condition variables related to it.

The method newCondition() of ReentrantLock class allows us generating all required waiting queues.

Methods of Condition interface:

- **await()**: suspends a thread in the condition.
- **signal()**: notifies the occurrence of the expected event to one of the threads who was waiting it.
- **signalAll()**: notifies the occurrence of the expected event to all threads waiting in the condition.

4.-CONCURRENT COLLECTIONS

Java includes Thread-safe versions of collections of objects as the **BlockingQueue** class.

The methods of BlockingQueue interface are:

	Methods	What do they do?
Inserting elements	add()	If there is not enough space, an exception is generated.
	offer()	If there is not enough space, it return false.
	put()	If there is not enough space, waits.
Extracting elements	take()	Extracts and deletes the first element. Waits if needed until there is any element to extract.
	poll()	Same as *take*, but if there are no elements in the queue waits as maximum as the specified interval.
	remove()	Deletes the specified object.
	peek()	Returns first element of the queue, but does not delete it.

5.-ATOMIC VARIABLES

Defines classes that support safe concurrent access to simple variables.

Primitive types: AtomicBoolean, AtomicInteger and AtomicLong.

References: AtomicReference.

6.-SYNCHRONIZATION: SEMAPHORES AND BARRIERS

The **Semaphore** class enables synchronization among thread:

- **acquire()**: blocks the thread until a permit is available, and the takes it.
- **release()**: adds a permit, potentially releasing a blocking acquirer.

They include a counter initialized in the creation of the semaphore.

The **CyclicBarrier** and **CountDownLatch** classes allow synchronization multiple executing threads.

The CyclicBarrir allow a group of threads to wait for each other. The barrier gets open when a certain number of threads reach the barrier:

- **await()** method of the barrier suspends a thread.
- When the last thread invokes await(), then the barrier gets open and all waiting threads are awaken.

The number of threads to be suspended is specified in the barrier constructor. It is **reusable** and allows to specify instructions to be done once the barrier gets opened.

The **CountDownLatch** allows suspending a group of threads, which remain waiting until the occurrence of some event that must be generated by another thread. There is a counter, which is initialized in the barrier constructor, once it arrives to zero, the barrier gets opened.

Methods of the CountDownLatch class:

- await(): threads get blocked in the barrier while it is closed.
- countdown(): decrements in one unit the value of the counter.

UNIT 6.-SYNCHRONIZATION IN REAL-TIME SYSTEMS

1.- INTRODUCTION TO RTS

A **real time system** (RTS) is a computing system that:

- Repeatedly interacts with its physical environment.
- Responds to the **stimuli** received from the environment within a **time interval**.
- For the correct behaviour of the system is not enough that the actions are correct, but they must be executed within the specified time interval.

The **time** in which actions are executed in the system is **significant**.

The response interval is characterized by a **deadline**:

- **Hard**: a response out of deadline is unacceptable.
- **Firm**: a response out of deadline is useless.
- **Soft**: a response out of deadline has little use (but can still be used).

Features of RTS:

- **Concurrency**. Controlled system components operate simultaneously. The RTS must attend them and generate control actions simultaneously.
- **Interaction with physical devices**. RTS interact with environment through various types of **non-conventional devices**. Software components that control the behaviour of these devices are, in general, **dependent of the particular system**.
- **Reliability and security**. A fail in the controller system might provoke that the controlled system behaves in a dangerous and non-economic way. It is important to ensure that if the controller system fails it must do it in a way that the controlled system remains in a safe state.
- **Temporal determinism**. Actions in certain time intervals. Essential that the temporal behaviour of the RTS is **deterministic**. The system must respond **correctly** in all situations. We must predict its behaviour in the **worst case**.

Not all techniques used to build other types of systems can be employed for hard real-time software.

Predictability is one of the main objectives in real-time systems. We must do a **schedulability analysis** of tasks in real-time system to predict whether tasks will meet their timing requirements.

A **scheduling paradigm** consists of:

- A **scheduling algorithm**, which determines the access order of tasks to system resources.
- An **analytical method** for calculating the temporal behaviour of the system. To verify that the requirements are guaranteed in all cases, we will always study the worst case. We need to know the duration of tasks in the worst case.

In RTS, **the Fixed-Priority Pre-emptive Scheduling** is the most popular Scheduling policy:

- The temporal behaviour is easier to understand and predict.
- Treatment with overloads is easy to predict.

- There are complete analytical techniques.
- It is required by standards of operating systems and concurrent languages.

Task model

Let us consider initially a simple task model:

- Static set of tasks, τ = {τ1, τ2..., τn}.
- All tasks are periodic, with **period** Ti.
- Tasks do not cooperate between them (they are independent).
- We know the **maximum execution time** of each task Ci.
- Each task has a **deadline** Di ≤ Ti.

Regarding the execution of tasks, we assume that:

- The scheduling policy is Priority Pre-emptive.
- The machine is dedicated to the application.
- Context switches have zero cost.
- Tasks cannot voluntarily suspend. Once a task is ready, it cannot delay its execution.

Two concepts help to build the worst case situation for independent periodic tasks:

- **Critical instant**: the response time in the worst case of all tasks is obtained if we activate them all at once.
- **Just check the first deadline**: after a critical instant, if a task meets its first deadline it will also meet all its subsequent deadlines.

Response time equation:

$$R_i = C_i + I_i \quad I_i^j = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

This can be solved by the following recurrence relation:

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Initial value is:

$$W_i^0 = C_i + \sum_{j \in hp(i)} C_j$$

This relation ends when:

$$W^{n+1} = w^n \text{ or}$$

$$W^{n+1} > D_i \text{ (the deadline is not met)}$$

2.- TASK SYNCHRONIZATION

One of the most important restrictions of the previous scheduling test is to assume that all tasks are independent. In most practical systems, tasks need to communicate between them:

- When communication is performed by sharing common data, then **synchronization** is necessary to avoid race conditions.
- We will assume that tasks only need **mutual exclusion** in the access to their critical sections, and that they protect them using a synchronization primitive equivalent to locks.

However, in real-time systems this solution is not valid because it can cause a situation called **priority inversion**, which invalidates the schedulability test.

Priority inversion occurs when a lower priority task is running before a higher priority task. The total time of blocking due to lower priority task is not bounded, since intermediate priority tasks can also be executed meanwhile.

This problem cannot be avoided, so we need to achieve that the blocking is bounded, is produced the east number of times as possible and is measurable.

The most important solutions are:

- **Priority inheritance**.
- **Immediate priority ceiling**.

Both cause that the priority inversion is based on the duration of one or more critical sections and not on the duration of complete tasks.

Priority ceiling is based on that each semaphore will be assigned a ceiling, equivalent to the highest priority of tasks that can use this semaphore. With this protocol, a task that accesses a semaphore immediately inherits the priority ceiling of the semaphore. The critical section is executed with the priority of the ceiling of the semaphore that protects it.

Properties of priority ceiling:

- Each task can be blocked at most once in each cycle. Moreover, if a task is blocked, it does it at the beginning of the cycle.
- There cannot be deadlocks.

The blocking factor B_i is the maximum time delay that a task i suffers due to all other tasks of lower priority.

We must also include the blocking effect of the lower priority tasks:

$$R_i = C_i + B_i + \sum_{j \in hp(i)}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

QUICK NOTES FROM TESTS:

- JAVA -> Lampson-Redell monitor.
- In Brinch Hansen model, the thread that calls to notify() doesn't get suspended, this call is made at the end as it finish its execution and leaves the monitor.