

What is meant by the terms safety properties & liveness properties when applied to concurrent programs?

Safety properties refer to requirements that something should never happen, such as a failure of mutual exclusion or condition synchronization. Liveness properties require that something *does* happen eventually, such as a process entering its critical section. A concurrent program must satisfy both these properties in order to be deemed 'correct'.

Define the notion of a semaphore & explain the behaviour of the primitive operations provided on semaphores

A semaphore is an integer value which can only take positive values. It can be seen as representing the amount of available resource. To use the resource, the semaphore is decremented (if there is resource available, i.e. the semaphore > 0) and after use, incremented. These operations are usually defined as P and V:

```
P {
    if (s > 0) sem--;
    else suspend;
}
V {
    sem++;
    wake up;
}
```

The waiting can be done using busy-waiting (inefficient) or blocking. Mutual exclusion can be implemented using a binary semaphore (one that only takes values of 0 or 1).

Define the notion of a *monitor* and briefly explain how synchronization within a monitor is achieved

A monitor is an abstract data type representing a shared resource. It has four components:

- A set of private variables to represent the resource
- A set of monitor procedures that provide the public interface to the monitor
- A set of condition variables
- Initialization code

Only one monitor procedure can run at a time, so this gives the required mutual exclusion. Conditional variables are used to implement condition synchronization, using the monitor primitive functions, wait(v) which makes the process wait until v is fulfilled and notify(v) which wakes up a process waiting for v.

Explain the advantages & disadvantages of semaphore-based & monitor-based solutions to concurrent programming problems

Both type of solutions, correctly implemented, ensure the desired safety and liveness properties. Semaphores can work for an arbitrary number of processes in a vastly simpler way than, for example, Dekker's algorithm. They can be implemented using blocking rather than busy-waiting which is more efficient. However, they are low-level constructs – omitting a single operation may lead to deadlock or violation of mutual exclusion. They are unstructured - the synchronization code is scattered about the program whenever a call is made to the shared resource. Also, they confuse conceptually distinct operations, implementing mutual exclusion and condition synchronization with the same primitive.

Monitors encapsulate the shared resource, keeping the resource bundled together with the operations that manipulate it. This is a higher level of solution and programmatically much neater, providing a public interface (that guarantees the desired concurrent properties) to the resource. Monitors can also handle an arbitrary number of processes and use blocking instead of busy-waiting.

Both methods provide the same power. What we gain with monitors is a higher level of abstraction.

Define the notion of a module as used in distributed processing implementations of concurrency & briefly explain how mutual exclusion and condition synchronization can be achieved in a module

A module contains a number of processes and local/exported procedures. Exported procedures can be called (through message passing) by remote processes. The results of these procedures are also returned through message passing (returned objects must be serialisable). The module contains a header (with the signature of each exported procedure) and a body with the implementations of the procedures. Mutual exclusion can be implemented either like in a monitor when only one procedure can run at one time, or by semaphores. Conditional synchronization is implemented through condition variables.

Explain the difference between Remote Procedure Calls and Extended Rendezvous

Using RPC, a new process is created to handle each call to a procedure. Extended rendezvous services each request using a single process. I.e., RPC handles a number of processes concurrently whereas ERV services each call one after the other. In this latter case, the server process is blocked until there is a pending call to service.

Define the notion of an atomic action and explain why atomic actions are important in concurrency programming

Atomic actions are those that make indivisible state changes – any intermediate state of the system is not seen by other processes. Fine-grained atomic actions are those that are implemented at a machine level. Different processors have different ranges of built-in atomic actions. Some of the more common ones are in/decrement, test-and-set and exchange. These can be used to build coarse-grained atomic actions – series of actions made uninterruptible through methods such as interrupt disabling or defining a mutual exclusion protocol. Atomic actions are vital in concurrent programming – if, for example, we test a condition variable (such as a lock) and perform some action depending on its value, it is vital that this test and corresponding action be done atomically, otherwise the condition may change between the test and the start of the action.

What is a thread in Java

A thread is a single sequential flow of control within a program. A number of threads can be running in the same JVM, communicating through a shared memory space. In Java, a thread is an instance of the Thread class (or an implementation of the Runnable interface).

Briefly describe the Java Memory Model and outline the guarantees it provides for implementing Mutual Exclusion protocols

Each Java thread maintains a 'working memory'. The memory model specifies when these values are reconciled with main memory:

- Atomicity: all read and writes to fields other than 'long' and 'double' are atomic. A value other than long or double will hold the initial value or a value written by some thread – not necessarily the most recent one.
- Visibility: values are flushed to main memory on termination of the thread.
- Ordering: the ordering of events can be changed by the compiler. With respect to the operating thread, the ordering appears correct but with respect to other threads, almost anything could happen.