CONCURRENCY

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

In fact, concurrency is essential in modern programming:

Web sites must handle multiple simultaneous users.

Mobile apps need to do some of their processing on servers ("in the cloud").

Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we're getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces.

In Programming Concurrency is a thread of control in a program is the sequence of program points reached as control flows through the program.

Categories of Concurrency:

1. Physical concurrency - Multiple independent processors (multiple threads of control).

2. Logical concurrency - The appearance of physical concurrency is presented by timesharing one processor (software can be designed as if there were multiple threads of control).

- Coroutines provide only quasiconcurrency. Reasons to Study Concurrency

1. It involves a new way of designing software that can be very useful--many real-world

situations involve concurrency.

2. Computers capable of physical concurrency are now widely used.

Design Issues for Concurrency

1. How is cooperation synchronization provided?

2. How is competition synchronization provided?

3. How and when do tasks begin and end execution?

4. Are tasks statically or dynamically created?

Methods of Providing Synchronization

1. Semaphores

2. Monitors

3. Message Passing

Semaphores

Semaphores (Dijkstra - 1965).

- A semaphore is a data structure consisting of a counter and a queue for storing task descriptors.

- Semaphores can be used to implement guards on the code that accesses shared data structures.

- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra).

- Semaphores can be used to provide both competition and cooperation synchronization

Example

wait(aSemaphore)

if aSemaphore's counter > 0 then

Decrement aSemaphore's counter

else

Put the caller in aSemaphore's queue

Attempt to transfer control to some

ready task

(If the task ready queue is empty,

deadlock occurs)

end

Example

release(aSemaphore)

if aSemaphore's queue is empty then

Increment aSemaphore's counter

else

Put the calling task in the task ready

queue

Transfer control to a task from

aSemaphore's queue

end

Monitors

It is a synchronization technique that enables threads to mutual exclusion and the wait() for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

- Competition Synchronization with Monitors:

- Access to the shared data in the monitor is limited by the implementation to a single process at a time; therefore, mutually exclusive access is inherent in the semantic definition of the monitor.

- Multiple calls are queued.

Cooperation Synchronization with Monitors:

- Cooperation is still required - done with semaphores, using the queue data type and the built-in operations, delay (similar to send) and continue (similar to release).

- delay takes a queue type parameter; it puts the process that calls it in the specified queue and removes its exclusive access rights to the monitor's data structure.

- Differs from send because delay always blocks the caller.

- continue takes a queue type parameter; it disconnects the caller from the monitor, thus freeing the monitor for use by another process.

-It also takes a process from the parameter.

-queue (if the queue isn't empty) and starts it.

-Differs from release because it always has some effect (release does nothing if the queue is empty).

Message Passing

In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send-off messages, and incoming messages to each module are queued up for handling

Example:

- a shared buffer.

- Encapsulate the buffer and its operations in a task.

- Competition synchronization is implicit in the semantics of accept clauses.

- Only one accept clause in a task can be active at any given time.

Java Threads

Competition Synchronization with Java Threads:

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution.

- If only a part of a method must be run without interference, it can be synchronized.

- Cooperation Synchronization with Java Threads:

- The wait and notify methods are defined in Object, which is the root class in Java, so all objects inherit them.

- The wait method must be called in a loop.

Example - the queue.