

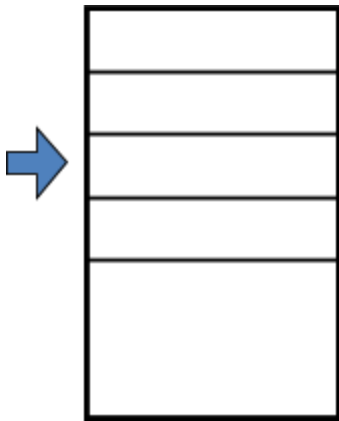
UNIT-1

SEQUENTIAL PROGRAMMING:

Sequential programming is writing an application as a series of steps. Modern compiler might rearrange the steps for faster execution. Almost anything with the exception of IO or communication could be considered sequential, even steps in a thread could be considered sequential.

All programs are sequential in that they execute a sequence of instructions in a pre-defined order:

$x = x + 1$



There is a *single thread* of execution or control

Sequential Program:

P;

Q;

R;

$x = 1$; // P

$y = x + 1$; // Q

$x = y + 2$; // R

For every possible execution (e) of this program, P must precede

Q and Q must precede R.

forall e: P ® Q ® R

The “®” operator means precedes or happens before. P ®Q means that

P must begin before Q begins and further P must finish before Q finishes

ie. there is no overlap in the execution of the instructions making up P & Q.

If each component P & Q is made up of several instructions then:

forall e: p1 ® p2 ... ® pm ® q1 ® q2 ... ® qn

There is a total ordering of the instructions making up P & Q.

It is clear that the final values of the variables in the example program depend on the order that statements are executed in. In general, given the same input data, a sequential program will always execute the same sequence of instructions and it will always produce the same results. Sequential program execution is deterministic.

The sequential paradigm has the following two characteristics:

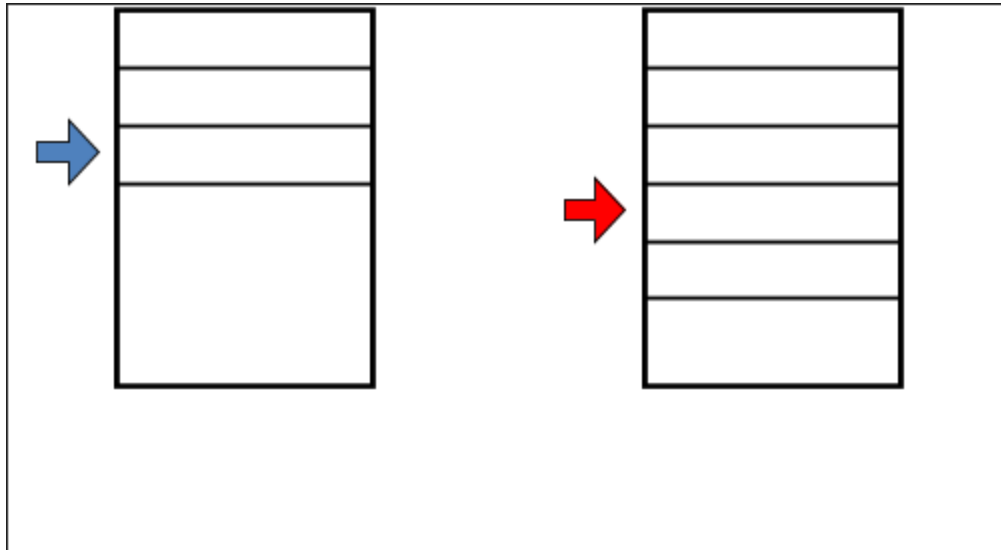
- The textual order of statements specifies their order of execution;
- Successive statements must be executed without any overlap (in time) with one another.
- Neither of these properties applies to concurrent programs.

CONCURRENT PROGRAMMING:

A **concurrent program** is one consisting of two or more processes — threads of execution or control

$x = x + 1$

$y = x$



Each *process* is itself a sequential program.

It is often useful to be able to do several things at once:

- **when *latency* (responsiveness) is an issue**, e.g., server design, cancel buttons on dialogs, etc.;
- **when you want to *parallelise*** your program, e.g., when you want to distribute your code across multiple processors;
- **when your program consists of a number of *distributed parts***, e.g., client–server designs.

Concurrent designs can still be effective even if you only have a single processor:

Consider a client–server system for file downloads (e.g. BitTorrent, FTP)

- without concurrency

- it is impossible to interact with the *client* (e.g., to cancel the download or start another one) while the download is in progress
- the *server* can only handle one download at a time—anyone else who requests a file has to wait until your download is finished
- with concurrency

the user can interact with the client while a download is in progress (e.g., to cancel it, or start another

More examples of concurrency

- **GUI-based applications:** e.g., javax.swing
- **Mobile code:** e.g., java.applet
- **Web services:** HTTP daemons, servlet engines, application servers
- **Component-based software:** Java beans often use threads internally
- **I/O processing:** concurrent programs can use time which would otherwise be wasted waiting for slow I/O
- **Real Time systems:** operating systems, transaction processing systems, industrial process control, embedded systems etc.
- **Parallel processing:** simulation of physical and biological systems, graphics, economic forecasting etc.

Advantages of concurrent programs

- Reactive programming—User can interact with applications while tasks are running, e.g., stopping the transfer of a big file in a web browser.

- Availability of services—Long-running tasks need not delay short-running ones, e.g., a web server can serve an entry page while at the same time processing a complex query.
- Parallelism—Complex programs can make better use of multiple resources in new multi-core processor architectures, SMPs, LANs or WANs, e.g., scientific/engineering applications, simulations, games, etc.
- Controllability—Tasks requiring certain preconditions can suspend and wait until the preconditions hold, then resume execution transparently.

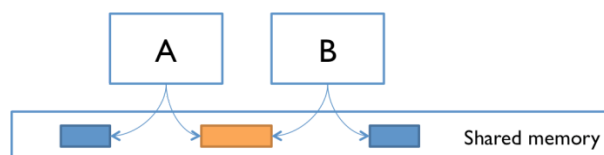
Disadvantages of concurrent programs

- Safety—«Nothing bad ever happens»—Concurrent tasks should not corrupt consistent state of program
- Liveness—«Anything ever happens at all »—Tasks should not suspend and indefinitely wait for each other (deadlock).
- Non-determinism—Mastering exponential number of interleaving due to different schedules.
- Resource consumption—Threads can be expensive. Overhead of scheduling, context-switching, and synchronization.—Concurrent programs can run slower than their sequential counterparts even with multiple CPUs!

Two Models for Concurrent Programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

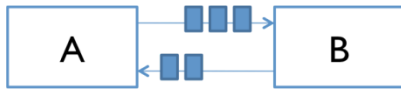
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



Other examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common file system with files they can read and write.
- A and B might be two threads in the same Java program (we'll explain what a thread is below), sharing the same Java objects.

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. Examples include:



- A and B might be two computers in a network, communicating by network connections.
- A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like `ls | grep` typed into a command prompt.

RACE CONDITION:

A race condition means that the correctness of the program (the satisfaction of post conditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say “A is in a race with B.”

Some interleaving of events may be OK, in the sense that they are consistent with what a single, no concurrent process would produce, but other interleaving produce wrong answers – violating post conditions or invariants.

- Race conditions
 - When correctness of result (postconditions and invariants) depends on the relative timing of events

These ideas connect to our three key properties of good software mostly in bad ways. Concurrency is necessary but it causes serious problems for correctness. We'll work on fixing those problems.

- **Safe from bugs.** Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
- **Easy to understand.** Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design in such a way that programmers don't have to think about that.
- **Ready for change.** Not particularly relevant here.

SYNCHRONIZATION PRIMITIVES

Synchronization primitives are simple software mechanisms provided by a platform (e.g. operating system) to its users for the purposes of supporting thread or process synchronization. They're usually built using lower level mechanisms (e.g. atomic operations, memory barriers, spinlocks, context switches etc).

Mutex, event, conditional variables and semaphores are all synchronization primitives. So are shared and exclusive locks. Monitor is generally considered a high-level synchronization tool. It's an object which guarantees mutual exclusion for its methods using other synchronization primitives (usually exclusive locks with condition variables to support waiting and signaling). In some contexts when monitor is used as a building block it is also considered a synchronization primitive.

Mutex class

The `System.Threading.Mutex` class, like `Monitor`, grants exclusive access to a shared resource. Use one of the `Mutex.WaitOne` method overloads to request the ownership of a mutex. Like `Monitor`, `Mutex` has thread affinity and the thread that acquired a mutex must release it by calling the `Mutex.ReleaseMutex` method.

Unlike `Monitor`, the `Mutex` class can be used for inter-process synchronization. To do that, use a named mutex, which is visible throughout the operating system. To create a named mutex instance, use a `Mutex` constructor that specifies a name. You also can call the `Mutex.OpenExisting` method to open an existing named system mutex.

Semaphore and SemaphoreSlim classes

The `System.Threading.Semaphore` and `System.Threading.SemaphoreSlim` classes limit the number of threads that can access a shared resource or a pool of resources concurrently. Additional threads that request the resource wait until any thread

releases the semaphore. Because the semaphore doesn't have thread affinity, a thread can acquire the semaphore and another one can release it.

SemaphoreSlim is a lightweight alternative to Semaphore and can be used only for synchronization within a single process boundary.

On Windows, you can use Semaphore for the inter-process synchronization. To do that, create a Semaphore instance that represents a named system semaphore by using one of the Semaphore constructors that specifies a name or the Semaphore.OpenExisting method. SemaphoreSlim doesn't support named system semaphores.

EventWaitHandle, AutoResetEvent, ManualResetEvent, and ManualResetEventSlim classes

The System.Threading.EventWaitHandle class represents a thread synchronization event.

A synchronization event can be either in an unsignaled or signaled state. When the state of an event is unsignaled, a thread that calls the event's WaitOne overload is blocked until an event is signaled. The EventWaitHandle.Set method sets the state of an event to signaled.

Monitor class

The System.Threading.Monitor class grants mutually exclusive access to a shared resource by acquiring or releasing a lock on the object that identifies the resource. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and the Monitor.Enter method waits until the lock is released. The Enter method acquires a released lock. You can also use the Monitor.TryEnter method to specify the amount of time during which a thread attempts to acquire a lock. Because the Monitor class has thread affinity, the thread that acquired a lock must release the lock by calling the Monitor.Exit method.

You can coordinate the interaction of threads that acquire a lock on the same object by using the Monitor.Wait, Monitor.Pulse, and Monitor.PulseAll methods.

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. Condition variables are user-mode objects that cannot be shared across processes.

Condition variables enable threads to atomically release a lock and enter the sleeping state. They can be used with critical sections or slim reader/writer (SRW) locks. Condition variables support operations that "wake one" or "wake all"

waiting threads. After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

Condition variable function	Description
InitializeConditionVariable	Initializes a condition variable.
SleepConditionVariableCS	Sleeps on the specified condition variable and releases the specified critical section as an atomic operation.
SleepConditionVariableSRW	Sleeps on the specified condition variable and releases the specified SRW lock as an atomic operation.
WakeAllConditionVariable	Wakes all threads waiting on the specified condition variable.
WakeConditionVariable	Wakes a single thread waiting on the specified condition variable.

Critical section is not a synchronization primitive. It's a part of an execution path that must be protected from concurrent execution in order to maintain some invariants. You need to use some synchronization primitives to protect critical section.