Understand Synchronize

(simultaneous,synchronous,concurrent,multiprogramming) - same time

synchronize (sync - verb):

---

*.Sync, short for "synchronize," is a verb for making things work together.
*.When you synchronize things, you make them happen at the same time. If you have rhythm, you can synchronize your dance moves with the beat of the music. If not, stay off the dance floor.

***.synchronization -(noun)** - the relation that exists when things occur at the same time.
***.synchronisation** - the relation that exists when things occur at the same time.
*.in common use, “synchronization” means making two things happen at the same time.
*.in computer systems,synchronization is a little more general; it refers to relationships among events-any number of events,and any kind of relationship(beoe,during,after).
*.conputer programmers are often concerned with **synchronization constraints**, which are requirements pertaining to the order of events.
*.

Serialization:

---

*.Event A must happen before Event B.

Mutual exclusion:

---

*.Events A and B must not happen at the same time.

Concurrent (adj) - occurring or operating at the **same time**.

---

*.concurrent - (ఉభయ)
*.*synonyms*: simultaneous , synchronous.
*.occuring or existing at the same time or having the same period or phase.
*.Concurrent means happening at the same time, as in two movies showing at the same theater on the same weekend.
*.concurrent describes two or more things happening at the same time.
*.A prisoner who is serving two concurrent five-year sentences will serve those prison terms together, meaning that he'll probably get out of jail in five years rather than ten.

simultaneously (adv) - at the same instant

---

*.at the same time.

*.Use the adverb simultaneously to describe actions that occur at the same time. You are reading this sentence and simultaneously learning a new word!.

concurrent execution (noun)

---

*synonyms: multiprogramming.
*.the execution of two or more computer programs by a single computer.

Good topic:
pls read on how synchronization is weired?

*.one possibility is that the computer is *parallel*, meaning that it has *multiple processors* running at the *same time*.In that case it is not easy to know if a statement on one processor is executed before a statement on another.

*.Another possibility is that a single processor is running multiple threads of execution. A thread is a sequence of instructions that execute sequentially. If there are multiple threads, then the processor can work on one for a while, then switch to another, and so on.

*.In general the programmer has no control over when each thread runs; the operating system (specifically, the scheduler) makes those decisions. As a result,again, the programmer can't tell when statements in different threads will be executed.

*.For purposes of synchronization, there is no difference between the parallel model and the multithread model. The issue is the same—within one processor (or one thread) we know the order of execution, but between processors (or threads) it is impossible to tell.

A **real world** example might make this clearer. Imagine that you and your friend Bob live in different cities, and one day, around dinner time, you start to wonder who ate lunch first that day, you or Bob. How would you find out? Obviously you could call him and ask what time he ate lunch. But what if you started lunch at 11:59 by your clock and Bob started lunch at 12:01 by his clock? Can you be sure who started first? Unless you are both very careful to keep accurate clocks, you can't.

Computer systems face the same problem because, even though their clocks are usually accurate, there is always a limit to their precision. In addition,most of the time the computer does not keep track of what time things happen.

There are just too many things happening, too fast, to record the exact time ofeverything.

Puzzle: Assuming that Bob is willing to follow simple instructions, is there any way you can guarantee that tomorrow you will eat lunch before Bob?

Key logic:*(on synchronization)*

---

*.When we talk about concurrent events, it is tempting to say that they happen at the same time, or simultaneously. As a shorthand, that's fine, as long as you remember the strict definition:

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Sometimes we can tell, after the program runs, which happened first, but often not, and even if we can, there is no guarantee that we will get the same result the next time.

Non-deterministic:

---

*.Concurrent programs are often **non-deterministic**, which means it is not possible to tell, by looking at the program, what will happen when it executets.

Because the two threads run concurrently, the order of execution depends on the scheduler.Non-determinism is one of the things that makes concurrent programs hard to **debug**.These kinds of ***bugs*** are almost impossible to find by **testing;** they can only be avoided by careful programming.

Shared variables:

---

*.Most of the time, most variables in most threads are **local** , meaning that they belong to a single thread and no other threads can access them. As long as that's true, there tend to be few synchronization problems, because threads just don't interact.

But usually some variables are **shared** among two or more threads; this is one of teh ways threads interact with each other.

Thread communication:

For example,
One way to communicate information between threads is for one thread to read a value written by another thread.

If the threads are *unsynchronized,* then we cannot tell by looking at the program whether the reader will see the value the writer writes or an old value that was already there. Thus many applications enforce the constraint that the reader should not read until after the writer writes. This is exactly the **serialization** problem.

Concurrent writes:

---

*.if x is a shared variable accessed by two writers.
*.it depends on the **order** in which the statements are executed, called the **execution path.**
*.Answering question like theseis an important part of **concurrent** programming:
What paths are possible and what are the possible effects? Can we prove that a given(desirble) effect is necessary or that an (undesirable) effect is impossible?

Concurrent updates:

---

*.An update is an operation that reads the value of a variable, computes a new value based on the old value, and writes the new value.

*.	Thread A	Thread B
	Count++	count++

At first glance, it is not obvious that there is a **synchronization** problem here. There are only two **execution paths** , and they yield the same result.

The problem is that these operations are translated into **machine language** before **execution.** The problem is more obvious if wer rewrite the code with a temporary variable, temp.

Thread A	Thread B
Temp = count	temp = count
Count = temp+1	count = temp+1

Now consider the following execution ptah

a1<b1<b2<a2

Assuming that the initial value of x is 0, what is its final value? Because both threads read the same initial value, they write the same value.

This kind of problem is subtle because it is not always possible to tell,looking at a high-level program, which operations are performed in a single step and which can be interrupted. In fact, some computers provide an increment
instruction that is implemented in hardware and cannot be interrupted. An operation that cannot be interrupted is said to be **atomic**.

So how can we write concurrent programs if we don't know which operations are ***atomic***? One possibility is to collect specific information about each operation on each hardware platform. The *drawbacks* of this approach are obvious.

The most common alternative is to make the conservative assumption that *all updates and all writes are not atomic*, and to use synchronization constraints to control concurrent access to shared variables.

The most common constraint is **mutual exclusion**, or **mutex**

Mutual exclusion *guarantees* that *only one thread* accesses a shared variable at a time, eliminating the kinds of synchronization errors in this section.

Code Sharing:-

---

*.In the embedded software, many subprograms such as interrupt service routines are invocated by multiple tasks concurrently, and these programs may be called recurrsively,which require such programs to be *reentrant*.

A *reentrant* function can safely be called from mulitple threads simultaneously, the execution of the function can be interrupted, and it can be called again, e.g., by another task (program), whithout affecting each other. That is, the function can be re-entered while it is already Running.

If a function contains static variables or accesses global data, then it is *not reentrant*.

The static variables of a function maintain their values between invocations of the function. When concurrent multiple tasks invoke this function, a race condition occurs. The sam reace condition may take place when multiple tasks invoke the same function.Each attempts to modify that global variable, which is not protected.

A function is considered *reentrant* if the function cannot be changed while in use.Reentrant code avoids race conditions by removing references to global variables and modifiable static data.In other word, all data with a reentrant function must be **atomic** data requirement, a reentrant function should work only on the data provided to it by the caller, should

not call any non-reentrant functions, and should not return any address to a static or global data.

*.in the following example, neither functions *f1* nor *f2* are reentrant.

```
int global = 0;
int f1()
{
      static s1 =1;
      global += 1;
      s1 += 1;

      return global + s1;
}

Int f2()
{
      Return f1()+1;
}
```

The function f1 depends on a global variable global and a static variable s1.Thus, if two tasks execute it and access these two variables concurrently, then the result will vary depending on the timing of the execution due to the race condition. Hence, f1 is not reentrant, and neither is f2 because it calls the non-reentrant function f1.

```
int f3(int i)
{
      int a;
      a = i + 1;

      return a;
}
int f4(int i)
{
      return f4(i) + 1;
}
```

Now both f3 and f4 are re-entrant functions at this time, because the f3 function only uses either atomic variables or the data provided by the caller and the f4 function calls the reentrant function f3.

NOTE:-

If any non-reentrant function needs to share the common data in the concurrent execution, mutual exclusion needs to be enforced. The semaphore for critical sections is one of the common practices.

Rendezvous:

---

*.rendezvous
- రెండెజౌస్
- Reṇḍejaus (pronounce)
- పరస్పర అంగీకారంతో ఏర్పచిన సమావేశము
- a secret rendezvous (especially between lovers)
- a date; usually with a member of the opposite sex
- a meeting arranged in advance
- Spot,get together,meet,appointment,date,

*.Thread A has to wait for Thread B and vice versa.
*.
Thread A	Thread B
Statement a1	statement b1
Statement a2	statement b2

*.We want to ***guarantee*** that a1 happens before b2 and b1 happens before a2.
*. a1 b2 , b1 a2
*.This **synchronization problem** has a name; it's a **rendezvous**.
*.The idea is that two threads rendezvous at a point of execution, and neither is allowed to proceed until both have arrived.

barrier

Src: Wiki
In parallel computing, a **barrier** is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

link:www.embeddedlinux.org.cn

q).why can't we use synchronization with waitpid/pthread_join ?

ans).

***.**

The pthread_join() call provides synchronization for threads similar to
That which waitpid() provides for processes, suspending its caller until
Another thread exits.

*.But we use coarse methods to synchronize

1. Mutex
2. Conditional variables

*.one process or thread stalled until the others caughtup and finished.

*.rather than blocking the execution o entire routine and thread in which it executes. Using finer synchronization techniques, threads can spend Less time waiting on each other and more time accomplishing the tasks For which they were designed.

[multithreading code in linux]