What is retrency lock and concept in general

**Re-entrant locking**

A reentrant lock is one where a process can claim the lock multiple times without blocking on itself. It's useful in situations where it's not easy to keep track of whether you've already grabbed a lock. If a lock is non re-entrant you could grab the lock, then block when you go to grab it again, effectively deadlocking your own process.

Reentrancy in general is a property of code where it has no central mutable state that could be corrupted if the code was called while it is executing. Such a call could be made by another thread, or it could be made recursively by an execution path originating from within the the code itself.

If the code relies on shared state that could be updated in the middle of its execution it is not re-entrant, at least not if that update could break it.

**A use case for re-entrant locking**

A (somewhat generic and contrived) example of an application for a re-entrant lock might be:

* You have some computation involving an algorithm that traverses a graph (perhaps with cycles in it). A traversal may visit the same node more than once due to the cycles or due to multiple paths to the same node.
* The data structure is subject to concurrent access and could be updated for some reason, perhaps by another thread. You need to be able to lock individual nodes to deal with potential data corruption due to race conditions. For some reason (perhaps performance) you don't want to globally lock the whole data structure.
* You computation can't retain complete information on what nodes you've visited, or you're using a data structure that doesn't allow 'have I been here before' questions to be answered quickly.  
    
  An example of this situation would be a simple implementation of Dijkstra's algorithm with a priority queue implemented as a binary heap or a breadth-first search using a simple linked list as a queue. In these cases, scanning the queue for existing insertions is O(N) and you may not want to do it on every iteration.

In this situation, keeping track of what locks you've already acquired is expensive. Assuming you want do the locking at the node level a re-entrant locking mechanism alleviates the need to tell whether you've visited a node before. You can just blindly lock the node, perhaps unlocking it after you pop it off the queue.

**Re-entrant mutexes**

A simple mutex is not re-entrant as only one thread can be in the critical section at a given time. If you grab the mutex and then try to grab it again a simple mutex doesn't have enough information to tell who was holding it previously. To do this recursively you need a mechanism where each thread had a token so you could tell who had grabbed the mutex. This makes the mutex mechanism somewhat more expensive so you may not want to do it in all situations.

IIRC the POSIX threads API does offer the option of re-entrant and non re-entrant mutexes.

How is ***safely*** defined?

If a program can be ***safely executed concurrently***, does it always mean that it is reentrant?

What exactly is the common thread between the six points mentioned that I should keep in mind while checking my code for reentrant capabilities?

Also,

1. Are all recursive functions reentrant?
2. Are all thread-safe functions reentrant?
3. Are all recursive and thread-safe functions reentrant?

While writing this question, one thing comes to mind: Are the terms like **reentrance** and **thread safety**absolute at all i.e. do they have fixed concrete definations? For, if they are not, this question is not very meaningful.

**1. How is *safely* defined?**

Semantically. In this case, this is not a hard-defined term. It just mean "You can do that, without risk".

**2. If a program can be safely executed concurrently, does it always mean that it is reentrant?**

No.

For example, let's have a C++ function that takes both a lock, and a callback as a parameter:

typedef void (\*MyCallback)() ;

NonRecursiveMutex mutex ;

void myFunction(MyCallback f)

{

lock(mutex) ;

f() ;

unlock(mutex) ;

}

At first sight, this function seems Ok... But wait:

int main(int argc, char \* argv[])

{

myFunction(myFunction) ;

return 0 ;

}

If the lock on mutex is not recursive, then here's what will happen:

1. main will call myFunction
2. myFunction will acquire the lock
3. myFunction will call myFunction
4. the 2nd myFunction will try to acquire the lock, fail and wait for it to be released
5. Deadlock.
6. Oops...

Ok, I cheated, using the Callback thing. But it's easy to imagine more complex pieces of code having a similar effect.

**3. What exactly is the common thread between the six points mentioned that I should keep in mind while checking my code for reentrant capabilities?**

You can *smell* a problem if your function has/gives access to a modifiable persistent resource, or has/gives access to a function that *smells*.

(*Ok, 99% of our code should smell, then... See last section to handle that...*)

So, studying your code, one of those points should alert you:

1. The function has a state (i.e. access a global variable, or even a class member variable)
2. This function can be called my multiple threads, or could appear twice in the stack while the process is executing (i.e. the function could call itself, directly or indirectly). Function taking callbacks as parameters *smell* a lot.

Note that non-reentrancy is viral : A function that could call a possible non-reentrant function cannot be considered reentrant.

Note, too, that C++ methods *smell* because they have access to this, so you should study the code to be sure they have no funny interaction.

**4.1. Are all recursive functions reentrant?**

No.

In multithreaded cases, a recursive function accessing a shared resources could be called by multiple threads at the same moment, resulting in bad/corrupted data.

In singlethreaded cases, a recursive function could use a non-reentrant function (like infamousstrtok), or use global data without handling the fact the data is already in use. So you function is recursive because it calls itself directly or indirectly, but it can still be *recursive-unsafe*.

**4.2. Are all thread-safe functions reentrant?**

In the example above, I showed how an apparently threadsafe function was not reentrant. Ok I cheated because of the Callback parameter. But then, there are multiple ways to deadlock a thread by having it acquire twice a non-reccursive lock.

**4.3. Are all recursive and thread-safe functions reentrant?**

I would say "yes" if by "recursive" you mean "recursive-safe".

If you can guarantee that a function can be called simultaneously by multiple threads, and can call itself, directly or indirectly, without problems, then it is reentrant.

The problem is evaluating this guarantee... ^\_^

**5. Are the terms like reentrance and thread safety absolute at all i.e. do they have fixed concrete definations?**

I believe they have, but then, evaluating a function is thread-safe or reentrant can be difficult. This is why I used the term *smell* above: You can find a function is not reentrant, but it could be difficult to be sure a complex piece of code is reentrant

**6. An example**

Let's say you have an object, with one method that need to use a resources:

struct MyStruct

{

P \* p ;

void foo()

{

if(this->p == NULL)

{

this->p = new P() ;

}

// Lots of code, some using this->p

if(this->p != NULL)

{

delete this->p ;

this->p = NULL ;

}

}

} ;

The first problem is that if somehow this function is called recursively (i.e. this function calls itself, directly or indirectly), the code will probably crash, because this->p will be deleted at the end of the last call, and still probably be used before the end of the first call.

Thus, this code is not *recursive-safe*.

We could use a reference counter to correct this:

struct MyStruct

{

size\_t c ;

P \* p ;

void foo()

{

if(c == 0)

{

this->p = new P() ;

}

++c ;

// Lots of code, some using this->p

--c ;

if(c == 0)

{

delete this->p ;

this->p = NULL ;

}

}

} ;

This way, the code becomes recursive-safe... But it is still not reentrant because of multithreading issues: We must be sure the modifications of c and of p will be done atomically, using a **recursive** mutex (not all mutexes are recursive):

struct MyStruct

{

mutex m ; // recursive mutex

size\_t c ;

P \* p ;

void foo()

{

lock(m) ;

if(c == 0)

{

this->p = new P() ;

}

++c ;

unlock(m) ;

// Lots of code, some using this->p

lock(m) ;

--c ;

if(c == 0)

{

delete this->p ;

this->p = NULL ;

}

unlock(m) ;

}

} ;

And of course, this all assumes the lots of code is itself reentrant, including the use of p.

And the code above is not even remotely [exception-safe](http://en.wikipedia.org/wiki/Abrahams_guarantees), but this is another story... ^\_^

**7. Hey 99% of our code is not reentrant!!**

It is quite true for spaghetti code. But if you partition correctly your code, you will avoid reentrancy problems.

**7.1. Make sure all functions have NO state.**

They must only use the parameters, their own local variables, other functions without state, and return copies of the data if they return at all.

**7.2. Make sure your object is "recursive-safe".**

An object method has access to this, so it shares a state with all the methods of the same instance of the object.

So, make sure the object can be used at one point in the stack (i.e. calling method A), and then, at another point (i.e. calling method B), without corrupting the whole object. Design your object to make sure that upon exiting a method, the object is stable and correct (no dangling pointers, no contradicting member variables, etc.).

**7.3. Make sure all your objects are correctly encapsulated.**

No one else should have access to their internal data:

// bad

int & MyObject::getCounter()

{

return this->counter ;

}

// good

int MyObject::getCounter()

{

return this->counter ;

}

// good, too

void MyObject::getCounter(int & p\_counter)

{

p\_counter = this->counter ;

}

Even returning a const reference could be dangerous if the use retrieves the address of the data, as some other portion of the code could modify it without the code holding the const reference being told.

**7.4. Make sure the user knows you object is not thread-safe**

Thus, the user is responsible to use mutexes to use an object shared between threads.

The objects from the STL are designed to be not thread-safe (because of performance issues), and thus, if a user want to share a std::string between two threads, the user must protect its access with concurrency primitives;

**7.5. Make sure you thread-safe code is recursive-safe**

This means using recursive mutexes if you believe the same resource can be used twice by the same thread.

What is meant by thread safe code

Does it mean that two threads can't change the undelying data simultaneously? or does it mean that the given code component will run with unpredictable results when more than one thread are running it?

*Thread safety is a computer programming concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.*

...

*There are a few ways to achieve thread safety:*

**Re-entrancy**

*Writing code in such a way that it can be partially executed by one task, reentered by another task, and then resumed from the original task. This requires the saving of state information in variables local to each task, usually on its stack, instead of in static or global variables.*

**Mutual exclusion**

*Access to shared data is serialized using mechanisms that ensure only one thread reads or writes the shared data at any time. Great care is required if a piece of code accesses multiple shared pieces of data—problems include race conditions, deadlocks, livelocks, starvation, and various other ills enumerated in many operating systems textbooks.*

**Thread-local storage**

*Variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be reentrant.*

**Atomic operations**

*Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter what other threads access it. Atomic operations form the basis of many thread locking mechanisms.*

**read more :**

<http://en.wikipedia.org/wiki/Thread_safety>

What is the difference between the concepts of **"Code Re-entrancy"** and **"Thread Safety"**? As per the link mentioned below, a piece of code can be either of them, both of them or neither of them.

[Reentrant and Thread safe code](http://encyclopedia.thefreedictionary.com/reentrant+code)

I was not able to understand the explaination clearly. Help would be appreciated.

Re-entrant code has no state in a single point. You can call the code while something is executing in the code. If the code uses global state, one call can conceivably overwrite the global state, breaking the computation in the other call.

Thread safe code is code with no race conditions or other concurrency issues. A race condition is where the order in which two threads do something affects the computation. A typical concurrency issue is where a change to a shared data structure can be paritially completed and left in an incosistent state. In order to avoid this, you have to use concurrency control mechanisms such as semaphores of mutexes to ensure that nothing else can access the data structure until the operation is completed.

For example, a piece of code can be non re-entrant but thread-safe if it is guarded externally by a mutex but still has a global data structure where the state must be consistent for the entire duration of the call. In this case, the same thread could initiate a call-back into the procedure while still protected by an external coarse-grained mutex. If the call-back occured from within the non re-entrant procedure the call could leave the data structure in a state that could break the computation from the caller's point of view.

A piece of code can be re-entrant but non thread-safe if it can make a non-atomic change to a shared (and sharable) data structure that could be interrupted in the middle of the update leaving the data structure in an incosistent state. In this case another thread accessing the data structure could be affected by the half-changed data structure and either crash or perform an operation that corrupts the data.

What is reentrant kernel

**Kernel Re-Entrance**

If the kernel is not re-entrant, a process can only be suspended while it is in user mode. Although it could be suspended in kernel mode, that would still block kernel mode execution on all other processes. The reason for this is that all kernel threads share the same memory. If execution would jump between them arbitrarily, corruption might occur.

A re-entrant kernel enables processes (or, to be more precise, their corresponding kernel threads) to give away the CPU while in kernel mode. They do not hinder other processes from also entering kernel mode. A typical use case is IO wait. The process wants to read a file. It calls a kernel function for this. Inside the kernel function, the disk controller is asked for the data. Getting the data will take some time and the function is blocked during that time. With a re-entrant kernel, the scheduler will assign the CPU to another process (kernel thread) until an interrupt from the disk controller indicates that the data is available and our thread can be resumed. This process can still access IO (which needs kernel functions), like user input. The system stays responsive and CPU time waste due to IO wait is reduced.

This is pretty much standard for today's desktop operating systems.

**Kernel pre-emption**

Kernel pre-emption does not help in the overall throughput of the system. Instead, it seeks for better responsiveness.

The idea here is that normally kernel functions are only interrupted by hardware causes: Either external interrupts, or IO wait cases, where it voluntarily gives away control to the scheduler. A pre-emptive kernel instead also interrupts and suspends kernel functions just like it would interrupt processes in user mode. The system is more responsive, as processes e.g. handling mouse input, are woken up even while heavy work is done inside the kernel.

Pre-emption on kernel level makes things harder for the kernel developer: The kernel function cannot be suspended only voluntarily or by interrupt handlers (which are somewhat a controlled environment), but also by any other process due to the scheduler. Care has to be taken to e.g. avoid deadlocks: A thread locks resource A but needing resource B is interrupted by another thread which locks resource B, but then needs resource A.

**Take my explanation of pre-emption with a grain of salt.** I'm happy for any corrections.

**All Unix kernels are reentrant.** This means that several processes may be executing in Kernel Mode at the same time. Of course, on uniprocessor systems, only one process can progress, but many can be blocked in Kernel Mode when waiting for the CPU or the completion of some I/O operation. For instance, after issuing a read to a disk on behalf of a process, the kernel lets the disk controller handle it and resumes executing other processes. An interrupt notifies the kernel when the device has satisfied the read, so the former process can resume the execution.

One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called reentrant functions . But a reentrant kernel is not limited only to such reentrant functions (although that is how some real-time kernels are implemented). Instead, the kernel can include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time.

If a hardware interrupt occurs, a reentrant kernel is able to suspend the current running process even if that process is in Kernel Mode. This capability is very important, because it improves the throughput of the device controllers that issue interrupts. Once a device has issued an interrupt, it waits until the CPU acknowledges it. If the kernel is able to answer quickly, the device controller will be able to perform other tasks while the CPU handles the interrupt.

Now let's look at kernel reentrancy and its impact on the organization of the kernel. A kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.

In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves the kernel control paths :

A process executing in User Mode invokes a system call, and the corresponding kernel control path verifies that the request cannot be satisfied immediately; it then invokes the scheduler to select a new process to run. As a result, a process switch occurs. The first kernel control path is left unfinished, and the CPU resumes the execution of some other kernel control path. In this case, the two control paths are executed on behalf of two different processes.

The CPU detects an exception-for example, access to a page not present in RAM-while running a kernel control path. The first control path is suspended, and the CPU starts the execution of a suitable procedure. In our example, this type of procedure can allocate a new page for the process and read its contents from disk. When the procedure terminates, the first control path can be resumed. In this case, the two control paths are executed on behalf of the same process.

A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled. The first kernel control path is left unfinished, and the CPU starts processing another kernel control path to handle the interrupt. The first kernel control path resumes when the interrupt handler terminates. In this case, the two kernel control paths run in the execution context of the same process, and the total system CPU time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.

An interrupt occurs while the CPU is running with kernel preemption enabled, and a higher priority process is runnable. In this case, the first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf of the higher priority process. This occurs only if the kernel has been compiled with kernel preemption support.

The [kernel](http://en.wikipedia.org/wiki/Kernel_%28computing%29) is the core part of an operating system that interfaces directly with the hardware and[schedules](http://en.wikipedia.org/wiki/Scheduling_%28computing%29) processes to run.

Processes call kernel functions to perform tasks such as accessing hardware or starting new processes. For certain periods of time, therefore, a process will be executing kernel code. A kernel is called*reentrant* if more than one process can be executing kernel code at the same time. "At the same time" can mean either that two processes are actually executing kernel code concurrently (on a multiprocessor system) or that one process has been interrupted while it is executing kernel code (because it is waiting for hardware to respond, for instance) and that another process that has been scheduled to run has also called into the kernel.

A reentrant kernel provides better performance because there is no [contention](http://en.wikipedia.org/wiki/Lock_%28computer_science%29#Granularity) for the kernel. A kernel that is not reentrant needs to use a [lock](http://en.wikipedia.org/wiki/Lock_%28computer_science%29) to make sure that no two processes are executing kernel code at the same time.

n UNIX systems we know malloc() is a non-reentrant function (system call). Why is that?

Similarly, printf() also is said to be non-reentrant; why?

I know the definition of re-entrancy, but I wanted to know why it applies to these functions. What prevents them being guaranteed reentrant?

malloc and printf usually use global structures, and employ lock-based synchronization internally. That's why they're not reentrant.

The malloc function could either be thread-safe or thread-unsafe. Both are not reentrant:

1. Any thread-unsafe function is not reentrant (reentrant functions are thread-safe by definition). Malloc operates on a global heap, and it's possible that two different invocations of malloc that happen at the same time, return the same memory block. (The 2nd malloc call should happen before an address of the chunk is fetched, but the chunk is not marked as unavailable). This violates the postcondition of malloc, so this implementation would not be re-entrant.
2. To prevent this effect, a thread-safe implementation of malloc would use lock-based synchronization. However, if malloc is called from signal handler, the following situation may happen:
3. malloc(); //initial call
4. lock(memory\_lock); //acquire lock inside malloc implementation
5. signal\_handler(); //interrupt and process signal
6. malloc(); //call malloc() inside signal handler
7. lock(memory\_lock); //try to acquire lock in malloc implementation
8. // DEADLOCK! We wait for release of memory\_lock, but

// it won't be released because the original malloc call is interrupted

This situation won't happen when malloc is simply called from different threads. Indeed, the reentrancy concept goes beyond thread-safety and also requires functions to work properly **even if one of its invocation never terminates**. That's basically the reasoning why any function with locks would be not re-entrant.

The printf function also operated on global data. Any output stream usually employs a global buffer attached to the resource data are sent to (a buffer for terminal, or for a file). The print process is usually a sequence of copying data to buffer and flushing the buffer afterwards. This buffer should be protected by locks in the same way malloc does. Therefore, printf is also non-reentrant.