Lecture 18 — Reentrancy, Inlining, HLL

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 22, 2019

Part I

Reentrancy

Non-Reentrant C Function

The trivial example of a non-reentrant C function:

```
int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}
```

Why is this non-reentrant?

How can we make it reentrant?

Reentrancy

$\Rightarrow$ A function can be suspended in the middle and **re-entered** (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially “no data races”.

Moot point if the function only modifies local data, e.g. `sin()`.

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```c
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example—Explained (a trace)

```
call swap(&a, &b);
 t = *x;                      // t = 3 (a)
 *x = *y;                     // a = 4 (b)
 call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    t = *x;                   // t = 1 (x)
    *x = *y;                  // x = 2 (y)
    *y = t;                   // y = 1
 *y = t;                      // b = 1

Final values:
a = 4, b = 1

Expected values:
a = 4, b = 3
```

Reentrancy Example, Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

   @\alert{s = t}@;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
   @\alert{t = s}@;  // restore global variable
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example, Fixed—Explained (a trace)

```
call swap(&a, &b);
s = t;                      // s = UNDEFINED
t = *x;                     // t = 3 (a)
*x = *y;                    // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    s = t;                  // s = 3
    t = *x;                 // t = 1 (x)
    *x = *y;                // x = 2 (y)
    *y = t;                 // y = 1
    t = s;                  // t = 3
*y = t;                     // b = 3
t = s;                      // t = UNDEFINED

Final values:
a = 4, b = 3

Expected values:
a = 4, b = 3
```

Interrupt Handling

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important.

Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted, or we pause and resume.

Either way, if the code is not reentrant we will run into problems.

Thread Safe vs. Reentrant

Let us also draw a distinction between thread safe code and reentrant code.

A thread safe operation is one that can be performed from more than one thread at the same time.

On the other hand, a reentrant operation can be invoked while the operation is already in progress, possibly from within the same thread.

Or it can be re-started without affecting the outcome.

If a function is not reentrant, it may not be possible to make it thread safe.

And furthermore, a reentrant function cannot call a non-reentrant one (and maintain its status as reentrant).

Thread Safe Non-Reentrant Example

```c
int length = 0;
char *s = NULL;

// Note: Since strings end with a 0, if we want to
// add a 0, we encode it as "\0", and encode a
// backslash as "\\".


// WARNING! This code is buggy — do not use!
void AddToString(int ch)
{
  EnterCriticalSection(&someCriticalSection);
  // +1 for the character we're about to add
  // +1 for the null terminator
  char *newString = realloc(s, (length+1) * sizeof(char));
  if (newString) {
    if (ch == '\0' || ch == '\\') {
      AddToString('\\'); // escape prefix
    }
    newString[length++] = ch;
    newString[length] = '\0';
    s = newString;
  }
  LeaveCriticalSection(&someCriticalSection);
}
```

Analysis of This Example

Is it thread safe? Sure—there is a critical section protected by the mutex `someCriticalSection`.

But is is re-entrant? Nope.

The internal call to `AddToString` causes a problem because the attempt to use `realloc` will use a pointer to `s`.

That is no longer valid because it got stomped by the earlier call to `realloc`.

Functional Programming

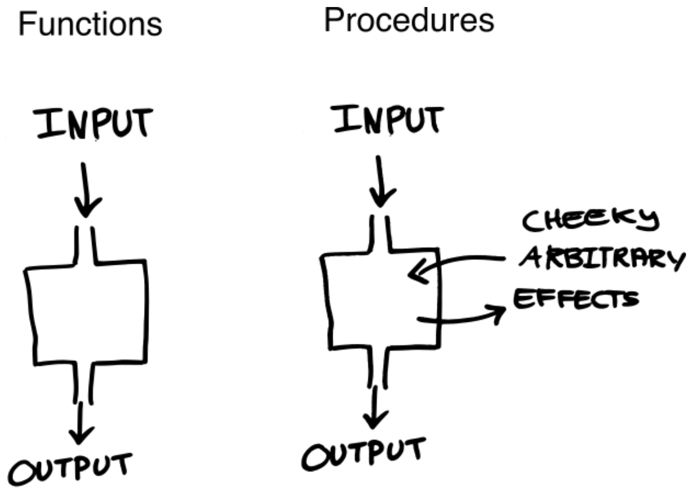
Interestingly, functional programming languages (NOT procedural like C) such as Scala and so on, lend themselves very nicely to being parallelized.

Why?

Because a purely functional program has no side effects and they are very easy to parallelize.

Any impure function has to indicate that in its function signature.

Functional Programming



`https://www.fpcomplete.com/blog/2017/04/pure-functional-programming`

Joel on Functional

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable.

— Joel Spolsky

OOP: `getBrain().damage()`

Object oriented programming kind of gives us some bad habits in this regard.

We tend to make a lot of `void` methods.

In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs.

If a function returns nothing, what does it do?

For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

Functional Programming in Your C++

`algorithms` has been part of C++ since C++11.
C++17 (not well supported yet) introduces parallel and vectorized `algorithms`;
  you specify an execution policy, compiler does it:
  (`sequenced`, `parallel`, or `parallel_unsequenced`).

Some examples of algorithms:
  `sort`, `reverse`, `is_heap`...

Other examples of algorithms:
  `for_each_n`, `exclusive_scan`, `reduce`

If you know functional programming (e.g. Haskell), these are:
  `map`, `scanl`, `fold1/foldl1`.

Side Effects

Side effects are sort of undesirable (and definitely not functional), but not necessarily bad.

Printing to console is unavoidably making use of a side effect, but it’s what we want.

We don’t want to call print reentrantly, because two threads trying to write at the same time to the console would result in jumbled output.

Or alternatively, restarting the print routine might result in some doubled characters on the screen.

Purity

The notion of purity is related to side effects.

A function is *pure* if it has no side effects and if its outputs depend solely on its inputs.

Pure functions should be implemented as thread-safe and reentrant.

Previous Example: thread-safety

Is the previous reentrant code also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}
```

Consider two calls: `swap(a, b)`, `swap(c, d)` with
  `a = 1, b = 2, c = 3, d = 4`.

Previous Example: thread-safety trace

```
global: t

/* thread 1 */                        /* thread 2 */
a = 1, b = 2;
s = t;    // s = UNDEFINED
t = a;    // t = 1
                                      c = 3, d = 4;
                                      s = t;    // s = 1
                                      t = c;    // t = 3
                                      c = d;    // c = 4
                                      d = t;    // d = 3
a = b;    // a = 2
b = t;    // b = 3
t = s;    // t = UNDEFINED
                                      t = s;    // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

Reentrancy vs Thread-Safety (1)

- Re-entrant does not always mean thread-safe (as we saw)
  - But, for most sane implementations, it is thread-safe

Ok, but are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

Are **thread-safe** functions reentrant? Nope. Consider:

```
int f() {
    lock();
    // protected code
    unlock();
}
```

Recall: **Reentrant functions can be suspended in the middle of execution and called again before the previous execution completes.**

`f()` obviously isn't reentrant. Plus, it will deadlock.

Interrupt handling is more for systems programming, so the topic of reentrancy may or may not come up again.

Summary of Reentrancy vs Thread-Safety

Difference between reentrant and thread-safe functions:

Reentrancy

- Has nothing to do with threads—assumes a **single thread**.
- Reentrant means the execution can context switch at any point in in a function, call the **same function**, and **complete** before returning to the original function call.
- Function's result does not depend on where the context switch happens.

Thread-safety

- Result does not depend on any interleaving of threads from concurrency or parallelism.
- No unexpected results from multiple concurrent executions of the function.

Another Definition of Thread-Safe Functions

"A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved."

Past Example of an Exam Question

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

- Is the above code thread-safe?
- Write some expected results for running two calls in parallel.
- Argue these expected results always hold, or show an example where they do not.

Part II

Good Practices

Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

No overhead… sounds like better performance…
let's inline everything!

Inlining in C++

Implicit inlining (defining a function inside a class definition):

```
class P {
public:
    int get_x() const { return x; }
...
private:
    int x;
};
```

Explicit inlining:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

The Other Side of Inlining

One big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors).

Just from this your performance may go down easily.

Compilers on Inlining

Inlining is merely a suggestion to compilers.
They may ignore you.

For example: n

- taking the address of an “inline” function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

From a Usability Point-of-View

Debugging is more difficult (e.g. you can’t set a breakpoint in a function that doesn’t actually exist).

- Most compilers simply won’t inline code with debugging symbols on.
- Some do, but typically it’s more of a pain.

Library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Introduction

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.
Common C sights:

- **#define macros**
- **void***

C++11 has made major strides towards readability and efficiency (it provides light-weight abstractions).

Outline

1 Sorting

2 Vectors vs. Lists

Goal

Sort a bunch of integers.

In **C**, usually use qsort from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

- A fairly ugly definition (as usual, for generic C functions)

How ugly? `qsort` usage

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - *((int*)b));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

- This looks like a nightmare, and is more likely to have bugs.

C++ sort

C++ has a sort with a much nicer interface[1]...

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

[1]...nicer to use, after you get over templates (they're useful, I swear).

C++ `sort` Usage

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor.
By default, `sort` uses `operator`$<$ on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

Timing Various Sorts

[Shown: actual runtimes of `qsort` vs `sort`]

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

Custom Sort

[Shown: actual runtimes of custom sort vs `sort`]

- The C++ version is still faster (although it's close).
- However, this is quickly going to become a maintainability nightmare.
  - Would you rather read a custom sort or 1 line?
  - What (who) do you trust more?

Lesson

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Lecture Fun

Let’s throw Java-style programming (or at least collections) into the mix and see what happens.

Outline

1 Sorting

2 Vectors vs. Lists

Vectors vs. Lists: Problem

1. Generate **N** random integers and insert them into (sorted) sequence.

Example: 3 4 2 1

- 3
- 3 4
- 2 3 4
- 1 2 3 4

2. Remove **N** elements one at a time by going to a random position and removing the element.

Example: 2 0 1 0

- 1 2 4
- 2 4
- 2
-

For which **N** is it better to use a list than a vector (or array)?

Complexity

- **Vector**
  - Inserting
    - $O(\log n)$ for binary search
    - $O(n)$ for insertion (on average, move half the elements)
  - Removing
    - $O(1)$ for accessing
    - $O(n)$ for deletion (on average, move half the elements)
- **List**
  - Inserting
    - $O(n)$ for linear search
    - $O(1)$ for insertion
  - Removing
    - $O(n)$ for accessing
    - $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality

[Shown: actual runtimes of vectors and lists]

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory.
  On 64 bit machines:
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

Performance Tips: Bullets

- Don’t store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.

Summary (Reentrancy)

- Reentrancy vs thread-safety.
- Inlining: limit your inlining to trivial functions:
  - makes debugging easier and improves usability;
  - won't slow down your program before you even start optimizing it.
- Tell the compiler high-level information but think low-level.

Programming for Performance with the Compiler

- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—
  different hardware needs different optimizations.