Concurrent Computing 4: C & Concurrency

1. Introduction
2. Terminology
3. Synchronization
4. **C and Concurrency**
   - **atomicity in C**
   - **concurrency and language elements**
   - **debugging concurrent programs**
5. More Asynchrony
6. Event-Driven Programming

Atomicity in C

When doing concurrent programming in a **high-level programming languge** like C, it is important to understand that each single language statement can be compiled/translated into *multiple* **machine instructions** (which are what ultimately get executed).

While each *machine instruction* is **atomic**, because a single *C statement* may be translated into multiple machine instructions, C statements are generally *not atomic* (or at least they cannot be assumed to always be atomic).

Consider an `int` variable `i` and the *increment* statement: `i++;`

On a modern CPU architecture, this statement might be compiled as a *single* machine instruction, but it might be compiled as *multiple* machine instructions.

Atomicity in C (contd.)

Compilation choices will depend on factors such as:

- the CPU architecture (e.g., Pentium vs. SPARC vs. ARM)
- the compiler plus the selected compiler architecture and optimzation options
- the variable's storage class (static, auto, etc.)
- how variables are used in the program

Intel x86 CPUs support an *integer addition* instruction, `add`, that can perform binary additions with the following:

- two **registers**
- a register and a memory location
- a constant and a register
- a constant and a memory location

Atomicity in C (contd.)

The C increment statement `i++;` can be implemented by adding the constant `0x1` to either a register or a memory location containing variable `i`'s value, so the statement can be translated into a *single machine instruction*, e.g.:

```
add  $0x1,%eax  # add constant one to i stored in register EAX
```

However, this is not what will always be done by a compiler, because of factors like:

- adding with registers is much faster than adding with memory addresses
- x86 CPUs have a limited number of *registers*, so not all variables can be held in registers

Atomicity in C (contd.)

As a result, a single C increment statement could be implemented as multiple x86 instructions, e.g.:

```
mov  0x804961c,%eax  # copy i's value from memory to register EAX
add  $0x1,%eax       # add one to i's value in EAX
mov  %eax,0x804961c  # copy i's incremented value to memory
```

Here, the C increment statement is obviously *not atomic*.

If another thread is accessing the variable, a *race condition* can result: only one or two of the above instructions are executed in a thread, that thread is suspended, a second thread runs and modifies `i`'s memory, that second thread is suspended, and the remaining of the above instructions are executed.

Atomicity in C (contd.)

The question of atomicity of C statements and *even individual machine instructions* becomes more complicated in the case of computers with multiple CPUs/cores!

In such systems, multiple threads may literally be running at the very same time, potentially trying to access/modify exactly the same memory locations.

Clearly, it is risky to assume that C statements even as simple as incrementing an `int` are atomic operations.

Instead, one should make use of **synchronization mechanisms** to ensure that race conditions cannot occur.

Atomicity in C (contd.)

While use of such mechanisms will tend to *slow* program execution, a program that has the potential of producing incorrect results is an *incorrect/buggy program*!

Old student programmer joke:
programmer1: "Hah, my program runs twice as fast as yours."
programmer2: "But mine always produces the correct answer!"

Question: Whose program do you want to use?

C and Concurrency

Prior to C11, the C language did not itself support concurrent programming.

Instead, *system calls* had to be used to create and control concurrent processes or OS threads.

C11 added support for writing *multithreaded* programs:

- threads control: `thrd_create()`, `thrd_join()`, etc.
- mutexes: `mtx_init()`, `mtx_trylock()`, etc.
- condition variables: `cnd_init()`, `cnd_signal()`, etc.
- thread-specific storage: `tss_create()`, `tss_set()`, etc.
- atomic data types: `atomic_int`, `ATOMIC_VAR_INIT()`, etc.
- fences: `atomic_thread_fence()`, etc.

Concurrency and C Language Elements

C99 and earlier versions do have some language elements that are relevant to concurrent programming:

- the `volatile` declaration keyword
- the `sig_atomic_t` data type
- **sequence points**
- **inline assembly**

volatile

The C keyword **volatile** is a **type qualifier**.

It can be used in declarations of variables and some other objects.

E.g., `volatile int flag;`

Basically, its use informs the compiler that the object's value might change independently of (local) code statements.

This means the compiler must *not* "optimize" code involving the object.

Its inclusion in C is to deal with:

- **memory-mapped devices**
- the `setjmp()`, `longjmp()`, etc. instructions
- `sig_atomic_t` variables in *signal handlers*

volatile (contd.)

`volatile`'s meaning appears to imply that it should also help with multithreaded programs, where one OS thread might modify a variable that another OS thread is also manipulating.

However, `volatile` is of almost *no value* in improving the safety of multithreaded programs:

- it has zero effect on the *atomicity* of operations on variables
- it has zero effect on the order of non-`volatile` variables
- while it does prevent optimizations to the `volatile` variables, one would have to declare every variable in a thread as `volatile` to help safety, which corresponds to using no optimization

sig_atomic_t

The C *data type* `sig_atomic_t` is a special *integer* type that guarantees **atomic** *access and modification* operations.

Specifically, this means that you can:

- atomically evaluate a variable's value
- atomically set its value to a constant

Note that it does *not* guarantee that *incrementing* a variable will be atomic!

The **Gnu C library** (**glibc**) documentation says this:
"In practice, you can assume that int is atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these assumptions are true on all of the machines that the GNU C library supports...."

volatile sig_atomic_t

Since standard *signal handlers* cannot be passed arguments, it is common to exchange information with a handler via *global variables*.

However, because the primary program may be interrupted at any point (*asynchronously*) to run a handler (and a handler can be interrupted to run another handler), global variables must be used carefully.

To provide as much safety as possible, global signal handler variables should generally be declared as: "`volatile sig_atomic_t`"

Sequence Points

A **sequence point** in a programming language is point at which all **side-effects** of previous evaluations shall be complete and *no* side effects of subsequent evaluations shall have taken place.

The C standard defines the sequence points in C programs—see Wikipedia's "Sequence point" entry if you want more details.

The importance of this for concurrent programming is that a C programmer must be aware that *sub-expressions* in statements may not be guaranteed to be executed in a particular order.

Sequence Points (contd.)

The C99 standard contains the following example:

```
int sum;
char *p;
sum = sum * 10 - '0' + (*p++ = getchar());
```

The expression statement is *grouped* as if written:

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

However the *execution order* can differ:

- the increment of `p` can occur at any time between the previous sequence point and the next sequence point (the `;`)
- the call to `getchar()` can occur at any point prior to the need for its returned value

Inline Assembly Code

If ultimate control over operations at the machine level are required, it is possible to include **assembly code** statements in C programs.

GCC supports **inline assembly** via the `asm()` call.

Here is an example call from the GCC documentation:

```
asm ("cmoveq %1,%2,%[result]"
     : [result] "=r"(result)
     : "r" (test), "r"(new), "[result]"(old));
```

Debugging Concurrent Programs

Concurrent programs are subject to all the bugs/errors that occur in single-thread programs—*plus* a new set of bugs/errors (e.g., *race conditions*).

The parallel and asynchronous nature of operations in concurrent programs makes it significantly more difficult for humans to find errors in concurrent programs.

Unfortunately, the kind of *testing* that programmers are used to doing with single-thread programs is also *much less effective* with concurrent programs.

It is not enough to show that a concurrent program produces the correct output for every input (class), it must be shown to do this under *every possible interleaving of its threads*.

Debugging Concurrent Programs (contd.)

However, simply running a concurrent program *repeatedly* (even *many* times) *cannot guarantee* that every possible interleaving of thread operations will get tested.

The OS, number of CPUs/cores, and current load can have a big impact on which interleavings occur when testing.

On one machine certain orderings may be unlikely to occur, while on a different machine those orderings could be most likely.

Even different OS versions may perform differently: e.g., some Linux kernel versions run the parent first after a `fork()` while others run the child first.

It is not uncommon for student code to immediately fail when graded, despite the students claiming they tested it "many times."

Debugging Concurrent Programs (contd.)

The only way a programmer has of forcing certain orderings is by the (temporary) insertion of *synchronization primitives* or `sleep()` and related calls in strategic locations.

Obviously, this is going to be extremely tedious and time consuming.

Concurrent programs are also much more likely than single-thread programs to suffer from *"hangs"* (e.g., **deadlock**).

Tools such as **GDB**, **strace**, and **ltrace** will be critical to identify what calls are blocking when the hang occurs, and to trace the order of calls leading to the hang.

Of course, the particular ordering of calls that lead to the hang may occur infrequently and so may be difficult to replicate.

Standard Synchronization Problems

The difficulty of debugging and testing means that it is critical to design concurrent programs to be *correct from the start!*

Most concurrent programs will involve one or more of a set of standard **synchronization problems**.

There are many sources programmers can consult to find the **design patterns** that correctly solve each problem.

Patterns will specify the ordering of program operations and the application of synchronization mechanisms that are required to avoid race conditions and deadlock.

Thus, the first step in implementing a concurrent program is to identify which synchronization problems apply to the program.

Standard Synchronization Problems (contd.)

Among the standard synchronization problems are:

- **producers consumers** variations:
  - bounded/finite buffer version
  - unbounded/infinite buffer version
  - single vs. multiple producers
  - single vs. multiple consumers
- **dining philosophers**
- **readers writers**
- **cigarette smokers**
- **barbershop** (**sleeping barber**)
- **dining savages** (obviously named some time ago :)