Concurrent Computing 5: More Asynchrony

1. Introduction
2. Terminology
3. Synchronization
4. C and Concurrency
5. **More Asynchrony**
   - **other sources of asynchrony**
   - **signals and asynchrony**
   - **reentrant vs. thread-safe**
6. Event-Driven Programming

Other Sources of Asynchrony

The chief source of complexity in writing concurrent programs is the uncertainty (and variability) in how operations in concurrent threads of execution will end up being *interleaved*.

The most obvious cause for this uncertainty is the freedom that the *OS scheduler* has to suspend any thread at any point and start another thread executing.

We mentioned earlier that with a high-level programming language like C, uncertainty is compounded by the fact that even simple C statements may *not* be *atomic*, and the atomicity of a C statement can vary.

Other Sources of Asynchrony (contd.)

Unfortunately, there are even more reasons why the order in which individual program operations may occur or be interleaved is uncertain.

The most common additional sources of order uncertainty are:

- CPUs using a **superscalar architecture**
- **optimizing compilers**
- computers that contain *multiple CPUs/cores*

Other Sources: Superscalar Architecture

Superscalar architecture CPUs implement **instruction-level parallelism** within each single CPU/core.

That is, multiple machine-level instructions (or components of an instruction) are simultaneously being computed across multiple **functional units** in each core.

To obtain the utmost speed/efficiency from the functional units, most superscalar CPUs support **out-of-order execution**:
the execution of machine instructions can be *reordered* as long as **data dependencies** are not violated.

This means that even if we know the order of the machine instructions in a thread, we cannot know the order in which the CPU will actually execute the instructions.

Other Sources: Superscalar Arch. (contd.)

This leads to additional possible interleavings of the machine instructions that comprise a concurrent program.

Most modern CPUs use a superscalar architecture—including Pentium and Core CPUs.

Other Sources of Asynchrony (contd.)

An **optimizing compiler** is one that can adjust the **object code** it produces for a program so as to *minimize* ("optimize") one or more *costs*, such as execution speed, executable size.

The produced code must of course implement the required semantics of the program!

Optimizing compilers can adjust object code by reordering instructions, removing instructions, combining instructions, and so forth.

This means that the instructions in the executable may be very different in order and even structure from what would be expected based on the source code.

Other Sources: Compiler Optimization

For example, consider the C function fragment:

```
  ...
  i++;
  j++;
  i++;
  return i;
}
```

An optimizing compiler might produce code more like:

```
  ...
  i += 2;
  return i;
}
```

Other Sources: Compiler Optimization (contd.)

Compiler optimizations can result in reordered instructions in threads, and thus unexpected interleavings of instructions in concurrent programs.

However, it is possible to **disassemble** the object code produced by an optimzing compiler to see the resulting machine-level code.

For example, you could use `objdump`:

```
objdump -S prog
```

(compile with the `-g` flag)

The optimization option for GCC is `-O`:

```
gcc -Wall -g -O2 -oprog program.c
```

Other Sources: Multiple CPUs

Modern computers often contain multiple CPUs and/or multiple cores, and such machines can provide true computation parallelism with concurrent programs.

However, because these machines can have multiple threads running at exactly the same time, the programmer is faced with an even larger set of possible relative orderings of the operations in different threads.

For example, every single machine instruction in every thread could end up being interleaved (if there are more CPUs/cores than threads).

Thus, the use of multi-CPU/core systems greatly increases the likelihood that an error-producing order of operations will occur in concurrent programs that contain a race condition bug.

Other Sources: Multiple CPUs (contd.)

Note that in modern multi-CPU/core computers (as apposed to **clusters**), the CPUs/cores all share the same memory (RAM).

While it appears that this architecture could lead to race conditions when multiple units all try to access memory at once, each unit will lock the **memory bus** when accessing memory, eliminating this possibility.

Programmers do not have to worry about race conditions occurring below the level of control they have over program operations.

Signals and Asynchrony/Atomicity

While we are focusing on issues in concurrent programming here, similar problems involving *asynchrony and atomicity* of operations can arise even in *single-thread programs*.

In such programs, the source of these issues is **signals**.

Signals are a type of **software interrupt** that informs a process that some **event** has occurred.

Signals that are generated by the process itself may be received **synchronously**—e.g., a `SIGPFE` due to an instruction that attempts *division by zero*.

Most signals, though, can be received by a program at any point during its execution—i.e., they can be received **asynchronously**.

Signals and Asynchrony/Atomicity (contd.)

Examples of signals that can be *received asynchronously* even in a *single process* program include those from:

- alarms/timers
- the user (e.g., `^-c` on keyboard or `kill` command)
- **signal-driven I/O**

When a program has **registered** a **handler** for one or more signals, the primary program can be interrupted at any point, the handler run, and then the primary program resumed.

Such programs must be designed so that interruptions can occur at any point (while the handler is registered) without affecting the correctness of the results.

Handlers and Reentrant Functions

To be safe and reliable, signal handlers must be **reentrant**.

A handler is **nonreentrant** if it accesses/modifies data used by other program components or other invocations of itself.

This can result in *race conditions*: incorrect results can be produced if the handler is invoked at particular points in the primary code or is invoked multiple times.

To be **reentrant**, a function must modify only data that is *local* (unique) to each *execution instance* of the function.

Reentrant functions cannot modify (or return pointers to) any data that is shared among multiple calls to the function, such as global or `static` variables, files, and so forth.

Handlers and Reentrant Functions

Signal handlers may be nonreentrant because the they call *library functions* that are nonreentrant.

SUS/POSIX refer to *reentrant* functions as **async-signal-safe functions**.

SUS/POSIX define a set of functions that must be safe to call in signal handlers, known as the **async-signal-safe functions** (see "`man 7 signal`").

Some *C library functions* are *not reentrant* because they modify statically allocated data (e.g., `fgetc()`, `malloc()`) or because they return pointers to statically allocated memory (e.g., `strtok()`, `gethostbyname()`).

Most of the *C standard I/O library* functions are *not async-signal-safe functions!*

Reentrant vs. Thread-Safe Functions (contd.)

The concept of *reentrant functions* is different from that of **thread-safe functions** (see "`man 7 pthreads`").

Many reentrant functions are also **thread-safe function**, but this is not necessarily the case.

E.g., if a reentrant function reads from a file that may be modified by another thread, the reentrant function is *not* thread-safe.

Being thread-safe does *not imply anything* about a function being reentrant!

A *nonreentrant* function can be made *thread-safe* by using a *mutex* to limit simultaneous execution of the function or limit simultaneous access to shared resources (like global variables).

Reentrant vs. Thread-Safe Functions (contd.)

SUS/POSIX library functions/syscalls that are *not thread-safe* are also *not async-signal-safe*, such as: `asctime()`, `ftw()`, `gethostbyname()`, `rand()`, `strtok()`, etc.

However, there are a number of *thread-safe functions* that are *not async-signal-safe*, such as: `fgetc()`, `fopen()`, `malloc()`, `mount()`, etc.

Most of the latter are C library functions, but there are a few system calls as well.