Concurrent Computing 2: Terminology

1. Introduction
2. **Terminology**
   - **terminology summary**
   - **asynchrony**
   - **race conditions**
   - **deadlock**
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Terminology

Synchronous (program/operations): the order in which program operations are executed is predictable.

Asynchronous/asynchrony: (some) program operations do *not* occur in a predictable order or at predictable times relative to one another.

Synchronization: operations/mechanisms that make certain that specific operations in separate threads of execution occur in the proper order or do not occur at the same time.

Serialization: enforcing one particular order among operations that might otherwise occur in various orders (i.e., asynchronously); can be both a positive as when done to avoid **race conditions** (below) and a negative as when it causes a program to block when it could do useful work.

Terminology (contd.)

Race condition: a logic error in a concurrent program, where incorrect results can be obtained if the operations in different threads of execution get scheduled to occur in a particular order (or to not occur in a particular order).

Atomicity: a program operation is said to be **atomic** if it cannot be interrupted and cannot encounter state changes during its instruction sequence.

Shared resource: a program object that can be simultaneously accessed by multiple threads (e.g., a variable, an open file).

Critical section: a program fragment that accesses a *shared resource* that should not be simultaneously accessed by multiple threads.

Terminology (contd.)

Mutual exclusion: refers to both the need to avoid multiple threads simultaneously entering a *critical section* and the OS or program constructs that enforce this.

Mutex: shortened form of "mutual exclusion," but typically refers to an OS/program mechanism for enforcing mutual exclusion among threads; i.e., a *synchronization* mechanism.

Busy wait (**spinlock**): *synchronization* method where a thread of execution effectively sits in a *loop* repeatedly testing for some condition (e.g., if a variable's value is non-zero) before proceeding; generally undesirable as it can waste CPU cycles and lead to **resource starvation** for other threads.

Terminology (contd.)

Deadlock: situation where two or more threads of execution *make no progress* because each is waiting for the other(s) to take an action; most commonly it is because each is waiting for the other(s) to release *shared resources* before they can proceed.

Livelock: similar to deadlock in that two or more processes make no progress, only difference being that the processes are not suspended but are running (e.g., two processes each try to break a deadlock by taking an action that still thwarts the other).

(Resource) Starvation: a thread of execution is perpetually denied resources required to be able to run (e.g., CPU cycles).

Terminology (contd.)

Priority Inversion: type of **starvation** where a *low-priority* thread *acquires a shared resource* before a *high-priority* thread, so the high-priority thread cannot run, but the low-priority thread never gets run so it cannot release the shared resource.

Semaphore: mechanism for controlling access to a shared resource, effectively an integer representing the number of clients that may currently be allowed to access the resource.

Lock: *synchronization* method for *mutual exclusion*, usually used to prevent multiple threads of execution from simultaneously accessing a *shared resource* (in *critical sections*).

Terminology (contd.)

Monitor: basically a *mutex* plus a mechanism for *signaling* threads when a condition is met that will allow access.

Condition Variables: signaling mechanism similar to **signals** but designed to work at the thread level in conjunction with **mutexes** to avoid need for *busy waiting*; basically what is required for a *monitor* (above).

Barrier: synchronization mechanism that forces all threads to stop at a point until every one reaches that point.

Memory Barrier (or **Fence**): machine instruction that enforces memory access ordering constraints.

Terminology (contd.)

Reentrant: code whose execution can be interrupted and run again ("*re-entered*") without affecting the results from the interrupted execution when/if it is resumed; not specifically related to concurrency, also relevant to the use of **signal handlers** and to **recursive** code.

Nonreentrant: code that is *not reentrant*; typically this is because the code is capable of modifying data that is *not local to each invocation* (e.g., `static` or global variables).

Thread-Safe Function: function that can be invoked from multiple simultaneous OS threads without affecting its semantics (as compared with a single invocation).

Asynchrony

Consider the following *single-thread* program:

```
int main()
{
  int x, y;
  ...read in the initial values for x & y...
  x = x + y;
  y = x - y;
  if (x > y)
    printf("x larger than y");
  else
    printf("y larger than x");
}
```

Asynchrony (contd.)

Program overview:

- assume `x` and `y` are read in from the terminal or a file
- `x` and `y` are then updated as shown
- their values are compared and the appropriate message printed

The operations in such a *single-thread* program are **synchronous**:

- we know that `x` will have its value changed before `y` does
- we don't know which `printf` statement will be executed (that will depend on the initial values assigned to `x` and `y`), but we know that the `if` test will occur after both `x` and `y` have been updated
- the program will always print the same message given the same inital values for `x` and `y`

Asynchrony (contd.)

Now consider a similar but *multithreaded* (*Pthreads*) program:

```
//Global variables:
int x, y;

//Function to be run in second Pthread:
void *update_y(void *arg)
{
  y = x - y;
  return NULL;  //(Needed due to Pthread calling convention.)
}

int main()
{
  ...read in the initial values for x & y...
  pthread_create(...,update_y,...);
  x = x + y;
  if (x > y)
    printf("x larger than y");
  else
    printf("y larger than x");
}
```

Asynchrony (contd.)

Multithreaded program overview:

- `x` and `y` have been defined as *global variables* so that they are accessible in every **scope** (i.e., every subroutine)
- a global variable is *shared* by all Pthreads since all Pthreads share address space (including the *static* memory segment)
- a new Pthread must run a function, and this function must take a single pointer argument and have pointer return value
- the `pthread_create()` call creates a second Pthread, running the function `update_y()`
- `x` is updated in the initial/main Pthread, `y` is updated in the *second Pthread*

Asynchrony (contd.)

Now various operations are **asynchronous**:

- we don't know whether `x` or `y` will have its value changed first, since this will depend on which Pthread is run first (following `pthread_create()`)
- while `x` will definitely be updated before the `if` test, we cannot be sure whether `y` will
- as a result, the output will not only depend on the initial values assigned to `x` and `y` (in the missing code), but on the *order* in which the two Pthreads get scheduled to run
- multi-CPU/core machines may even intermingle the **machine instructions** that implement the assignment statements
- since Pthreads may be scheduled differently on each execution, the program's output might vary from run to run, even with the same initial values for `x` and `y`

Race Conditions

Because the output of the above multithreaded program depends on the order the OS threads get scheduled to run as well as the input values for `x` and `y`, the program suffers from a **race condition**.

Race conditions are one of the most common *logic errors/bugs* that occur in concurrent programs.

Eliminating the race condition in the above program requires that we introduce **synchronization mechanisms** to ensure that:

- `y` is updated in the second thread only *after* `x` has been updated in the first/main thread
- the `if` condition in the first thread is tested only *after* `y` has been updated in the second thread

Race Conditions (contd.)

Let's examine another, less obvious race condition example.

Consider the following C code fragments:

```
//Globals:
int i = 0;

//Fragment #1:
i = 1;
i++;

//Fragment #2:
i = 3;
i += 1;
```

Assume that the two fragments occur in *separate threads* that are executing *concurrently*.

Race Conditions (contd.)

If fragment #1 runs before #2 then we expect `i` to end up with *value 4*, while if fragment #2 runs before #1 we expect `i` to end up with *value 2*.

However, scheduler decisions about the two threads could cause the fragments' statements to be *interleaved* in six different orders, leading to other possible values for `i`:

- `i=1; i++; i=3; i+=1;` `i` is 4
- `i=3; i+=1; i=1; i++;` `i` is 2
- `i=1; i=3; i++; i+=1;` `i` is 5
- `i=1; i=3; i+=1; i++;` `i` is 5
- `i=3; i=1; i++; i+=1;` `i` is 3
- `i=3; i=1; i+=1; i++;` `i` is 3

Race Conditions (contd.)

If the correct/intended value for `i` is either 2 or 4, then this program suffers from a *race condition* bug that can cause `i` to have an incorrect value of either 3 or 5.

The source of the race condition here is that we *cannot* assume that sequences of *multiple program statements* in a thread of execution will get executed **atomically**.

In fact, we will see (below) that we cannot even assume that *individual C statements* like `i++` will get executed atomically!

If we need to have one fragment get executed in its entirety before the other begins execution, then we will have to use **synchronization mechanisms** in our code to ensure that is the only order in which execution can occur.

Race Conditions (contd.)

One way that we could prevent the fragments' statements from being interleaved is by preventing either from running if the other has started running but not yet completed.

In other words, consider each fragment as a **critical region**, and enforce **mutual exclusion**.

We might imagine that this could be accomplished by using a shared `int` variable, `lock`, where if `lock` is 1 (one) it means it is safe for a fragment's critical region to begin execution, while if it is 0 (zero) it means it is not safe and the thread should wait.

This is the basic concept behind a **mutex** or **binary semaphore**.

Race Conditions (contd.)

So we modify the fragments as follows:

```
//Fragment #1:
while(lock != 1);
lock = 0;
i = 1;
i++;
lock = 1;

//Fragment #2:
while(lock != 1);
lock = 0;
i = 3;
i += 1;
lock = 1;
```

Race Conditions (contd.)

The idea here is that each fragment will wait for the other to finish before it proceeds, since each will sit in a `while` loop testing for `lock`'s value to become 1 (one).

One problem with this approach is that the `while` loop checking `lock` represents a **busy wait** or **spinlock**, so wastes CPU cycles.

A second and even more serious problem is that this approach *completely fails to eliminate the race condition*, as both processes can still end up modifying `i` concurrently.

Since testing `lock` and changing its value are *not done atomically*, we can end up with an execution sequence such as:
thread #1: `... while(lock!=1);`
thread #2: `while(lock!=1); lock=0; i=3;`
thread #1: `lock=0; i=1; i++;`
thread #2: `i+=1;...`

Race Conditions (contd.)

This race condition cannot be eliminated when using ordinary variables in a high-level programming language due to the lack of an *atomic* **test-and-set operator**.

What is required are special *OS system calls* that implement appropriate **synchronization mechanisms**.

Since the OS controls the scheduling of operations across the CPUs/cores, it can provide effective synchronization mechanisms by controlling the execution of processes/threads, blocking CPU interrupts, locking access to memory, and so forth.

All modern OS's provide synchronization mechanisms that can be used to eliminate the race condition in the sample code (e.g., **mutexes** and/or **semaphores**).

Deadlock

Another potential flaw in cocurrent programs is the possibility for **deadlock**.

Deadlock will typically involve a *race condition*, where threads deadlock only under particular interleavings of operations.

Deadlock can occur when two (or more) threads need multiple shared resources to do their processing and each is able to acquire a subset of the resources; to proceed, each thread needs resources held by the other.

This is called a **Coffman deadlock** or **resource deadlock**.

Resource deadlock can often be avoided by making sure that all threads try to acquire common resources in the same order, or by extending the critical sections of the threads to include all resource acquisition.

Deadlock (contd.)

Another common type of deadlock is **communication deadlock**: threads send messages to each other for synchronization, but the messages can get lost.

For example, suppose one thread must send a message to a second to get it to process data, after which the first thread then waits to hear back from the second that it is done.

If either of the two messages is lost, the two threads will end up deadlocked.

This situation can occur when sending **signals**:
if a process has not yet been setup to receive a signal when the signal is received, the signal will probably be discarded, meaning a subsequent `pause()` could then cause the process to sleep forever.