Lecture 3: Scenarios, races, locks, semaphores

© K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

22 January 2020

Counting sequentially to two

First, pseudo-code showing how to sequentially increment a *counter* twice. (Java would also need *boilerplate* code to declare *classes*, *instances*, etc. In other languages, the code below might be closer to the actual code).

Listing 1: Sequential counter: increment twice

```
1 int counter=0;                    //global variable
2 void increment() {                //global procedure
3   int cnt = counter;
4   counter = cnt + 1;
5   }
6 increment();
7 increment();
```

The value of *counter* at the end is 2, every time that code snippet runs.

- Are there set-ups in which to check such statements by computer?
- Yes. One such is to use *scenarios*.

Scenario = list of commands and state *before* each runs

```
1 int counter=0;                    //global variable
2 void increment() {                //global procedure
3     int cnt = counter;
4     counter = cnt + 1;
5 }
6 increment();
7 increment();
```

#	pc	local state (cnt)	global state (counter)
1	6	none	0
2	3	$\perp$ (=undefined)	0
3	4	0	0
4	5	0	1
5	7	none	1
6	3	$\perp$	1
7	4	1	1
8	5	1	2
9	done	none	2

Adding concurrency

Now, we revisit the example by introducing *concurrency*:

Suppose the two calls to method *increment* can be executed *concurrently*.

(In Java, this is done using *threads*. The details will be in the tutorial on concurrency in Java. Here, understand the concurrency via pseudo-code).

The idea is that:

- There are two independent *processes* (execution units), *t* and *u*
- We do not know the *order of execution* of the commands of *t* and *u*; a *scheduler* beyond our control *interleaves* these commands as it wishes
- The variable *counter* is *shared* between *t* and *u*

Incrementing concurrently - one abbreviated scenario

```
1 void increment() {                    //global procedure
2     int cnt = counter;
3     counter = cnt + 1;
4 }
```

int counter=0	shared variable
process t	process u
increment();	increment();

#	pc	local state cnt_t	local state cnt_u	global state counter
1	t2	$\perp$	$\perp$	0
2	u2	0	$\perp$	0
3	u3	0	0	0
4	t3	0	0	1
5	done	none	none	1

We've skipped t4, u4 and the calls to *increment*.

The sequence t2, t3, u2, u3 will produce *counter*=2.

Non-deterministic interleaving!. Are other values possible?

Reasons for using concurrency

WHY do we need concurrent programming in the first place?

- *abstraction: separate* naturally different tasks, never mind when to execute them (E.g., individuals in epidemic simulation)
- *responsiveness*: be *responsive* to user, executing different tasks independently (e.g., browse one page while another loads)

A separate matter is to speed things up, performance.

- We look at this in the multi-core part of the course (Lab 3).
- *split a complex task* into subtasks, and assign each its own processor (e.g., compute all prime numbers up to 1 billion)

Concurrency vs. parallelism

Principles of *concurrent* programming

vs.

Principer för *parallell* programmering

The Swedish *parallell* covers both *concurrent* and *parallel*.

OK, so how do *concurrent* and *parallel* differ?

Concurrency vs. parallelism

- You can have *concurrency without physical parallelism*: operating systems running on single-processor single-core systems.
- Parallelism is mainly about *speeding up* computations by taking advantage of multiple processing units (read cards and print lines).

concurrency: nondeterministic composition of independently executing units (*potentially physically* parallel).

- There may be more or fewer processes than processors.
- The program is designed to meet any scheduling.

parallelism: (*physical* parallelism) on multiple processing units.

- There may be more or fewer processes than processors.
- The program aims is designed for speed-up.
- Synchronisation as in concurrent programming is useful to avoid working entirely by *timimg*.
  - As in the old unit record example, CDR, CPU and LPT

A few frames from the Carlo/Sandro Lecture 01, frame 22 - 60 (including many overlays) if you wish to see the same things look in Java.

Suggestion: Look at the process states, and do the rest off-line.

Race conditions

Concurrent programs are *nondeterministic*:

- executing multiple times the same concurrent program with the same inputs may lead to *different execution traces*
- this is a result of the nondeterministic *interleaving* of each thread's trace to determine the overall program trace
- in turn, the interleaving is a result of the *scheduler*'s decisions

A *race condition* is a situation where the correctness of a concurrent program depends on the specific execution

The *concurrent counter* example has a race condition:

- in some executions the final value of *counter* is 2 (correct),
- in some executions the final value of *counter* is 1 (wrong).

Race conditions can greatly *complicate debugging*!

Data races vs. race conditions

A *data race* occurs when two concurrent threads

- access a shared memory location,
- at least one access is a *write*,
- the threads use *no explicit synchronisation* mechanism to protect the shared data.

Not every race is a data race

- race conditions can occur even without shared memory access
  - file systems, network access. These are shared resources, but part of the underlying OS support.

Not every data race is a race

- the data race might not affect the result
  - e.g., if two threads write the same value to shared memory

The mutual exclusion problem

The *mutual exclusion* problem is a fundamental synchronization problem.

- Arises whenever multiple threads share access to a common resource.
- *critical section* (CS): the part of a program that accesses the shared resource (typically, a shared variable)
- *mutual exclusion property*: at most one thread is in its CS at any given time

The *mutual exclusion problem*: devise a *protocol*, satisfying the mutual exclusion property, for accessing a *shared resource*.

- Updating a shared variable *consistently* is an *instance*.

The phrase "mutual exclusion" is abbreviated *mutex*.

The mutex problem - simplifications

Simplifications to present solutions in a uniform way:

- the critical section is an arbitrary *block* of code
- threads *continuously* try to enter the critical section
  - Typically, endless loop of non-critical (NCS) and critical section (CS)
- threads spend a *finite* amount of *time* in the critical section
  - Cannot die or loop in a CS
- we *ignore* what the threads do *outside* their their critical sections
  - Can die or loop in NCS

Mutex - classic schematic program

Reminder:

- *mutex property*: at most one thread is in its CS at any given time
- *mutex problem*: devise a protocol to access a shared resource, while satisfying the mutex property.

shared variables	
process t	process u

while true {
NCS;
entry protocol;
CS;
exit protocol;
};
	while true {
	NCS;
	entry protocol;
	CS;
	exit protocol;
	}

What's a *good solution* to the mutual exclusion problem?

A fully satisfactory solution is one that achieves three properties:

1. *Mutual exclusion*: at most one thread is in its CS at any given time
2. *Freedom from deadlock*: if one or more threads try to enter their respective CS's, some thread will eventually succeed
3. *Freedom from starvation*: every thread that tries to enter its CS will eventually succeed

(Note that freedom from starvation implies freedom from deadlock.)

A good solution should also work for an *arbitrary number* of threads sharing the same memory.

Deadlocks

Under the following conditions:

- A mutex protocol provides *exclusive access* to shared resources to one thread at a time.
- Threads that try to access the resource when it is not available will have to block and *wait*.

Mutually dependent waiting conditions may arise, causing a situation called a *deadlock*.

A *deadlock* is the situation where a group of threads wait forever because each of them is waiting for resources that are held by another thread in the group (circular waiting)

Note that this makes sense only when we have *processes* (abstractions supported by a run-time system) with *blocked* and *ready* states.

At a lower abstraction level, "hardware processes" are sometimes informally said to "deadlock". Each is looping until something happens, and that will never happen. It is better to call such situations *livelock*.

Deadlock: examples

- Two polite people waiting either side of a narrow doorway: "you first, you first".
  - Each is *waiting* for the other to go through the doorway.
    - ★ The scheduler marks both *blocked*.
    - ★ The run-time support will wake up a person when the *event* "other person walks through door" happens. — But that will never happen.
- What happens if two greedy people wait either side of a narrow doorway going "me first, me first"?
  - They cause a *livelock*, not visible to the run-time system.
  - They both remain ready and keep getting scheduled, but get nowhere.
- You and I share a pencil and writing pad. I grab the pencil and wait for the pad, you do the opposite.

The dining philosophers

The *dining philosophers* is a classic synchronization problem introduced by Dijkstra. It illustrates the problem of deadlocks using a colorful metaphor (by Hoare).

- Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers.
- Each philosopher alternates between thinking (*non-critical section*) and eating (*critical section*).
- In order *to eat*, a philosopher needs to pick up the *two forks* that lie to the philopher's left and right.
- Since the forks are *shared*, there is a *synchronization* problem between philosophers (*threads*).

Deadlocking philosophers

An *unsuccessful attempt* at solving the dining philosophers problem:

```
                    the shared forks
process p[i]
while true {
        think;
        pick up left fork, then right fork;
        eat;
        put down left fork, then right fork;
        }
```

This protocol *deadlocks* if all philosophers get their left forks, and wait forever for their right forks to become available.

The Coffman conditions

Necessary conditions for a deadlock to occur:

1. *Mutual exclusion*: threads may have exclusive access to the shared resources.
2. *Hold and wait*: a thread that may request one resource while holding another resource.
3. *No preemption*: resources cannot forcibly be released from threads that hold them.
4. *Circular wait*: two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

Avoiding deadlocks requires to *break* one or more of these conditions.

Breaking a circular wait

A solution to the dining philosophers problem that *avoids deadlock* by avoiding a *circular wait*: everyone follows the previous deadlocking protocol except the 5th philosopher, who picks up their right for first.

Ordering shared resources and forcing all threads to acquire the resources in order is a common measure to avoid deadlocks. So always grab pen first, then pad.

Atomic philosophers

A solution to the dining philosophers problem that *avoids deadlock* by breaking *hold and wait* (and thus *circular wait*): pick up both forks at once (*atomic* operation).

This protocol avoids deadlocks, but it may introduce *starvation*: a philosopher may never get a chance to pick up the forks.

Starvation

No deadlocks means that the system makes *progress as a whole*.
However, some individual thread may still make no progress because it is treated *unfairly* in terms of access to shared resources.

Starvation is the situation where a thread is
perpetually denied access to a resource it requests.

Avoiding starvation requires an additional assumption about the *scheduler*.

Fairness

Starvation is the situation where a thread is *perpetually denied* access to a resource it requests.

Avoiding starvation requires the *scheduler* to

"give every thread a chance to execute".

Weak fairness: if a thread *continuously requests* (that is, requests without interruptions) access to a resource, then access is granted eventually (or infinitely often).

Strong fairness: if a thread *requests* access to a resource infinitely often, then access is granted eventually (or infinitely often).

Applied to a *scheduler*:

- request = a thread is ready (enabled)
- fairness = every thread has a chance to execute

Tools: model checkers

The model checker SPIN

- Checks Promela programs. It checks assertions and more general LTL formulas, showing where they fail to hold.
  - A single state is needed to disprove a safety property. You said "this bad thing won't happen". Well, here is where it does, and a path to get there.
  - A loop of states is needed to disprove a liveness property. You said "this good thing will eventually happen". Well, here is a loop where I can get stuck and where the good thing never happens.

Finish with Carlo/Sandro Lecture 2, frame 25 (Locks) onwards.