Concurrent Bug Patterns and How to Test Them

Eitan Farchi
Yarden Nir
Shmuel Ur

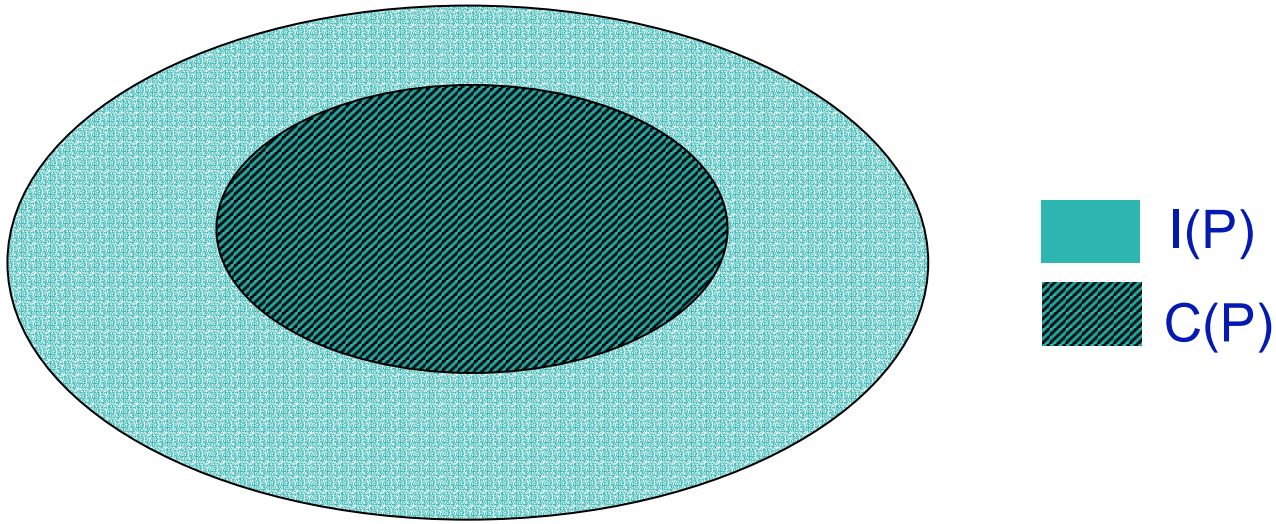
Outline

- Introduction
- Abstract categories of concurrent bugs
- Sub-categories of concurrent bugs
- Testing: how to reveal concurrent bugs

Categorizing concurrent bugs

- A concurrent event is a synchronization event or a memory access event
- An interleaving is a sequence of concurrent events that occurred in a specific execution of P (Java replay, Choi and Srinivasan, 1998)
- For a given program P
  - I(P) is the space of all possible interleavings of P
  - C(P) is the space of all correct interleavings of P

Categorizing concurrent bugs (continued)



- [I(P) – C(P)] is the set of erroneous interleavings
- Objective: characterize the gap between I(P) and C(P)

Example

- Given a two byte global variable X
- Two threads execute
  (X = 257) in parallel with (X = 0)

The programmer's intended results are 0 or 257. Thus, C(P) contains:

Thread 1	Thread 2
X = 257	
	X = 0

and

Thread 1	Thread 2
	X = 0
X = 257	

Example (continued)

- The programmer ignored the non-atomicity of the assignment operation
- A possible interleaving in I(P) – C(P) is (X[0] is the least significant byte):

Thread 1	Thread 2
	X[0] = 0
X[0] = 1	
X[1] = 1	
	X[1] = 0

- Result is 1

Outline

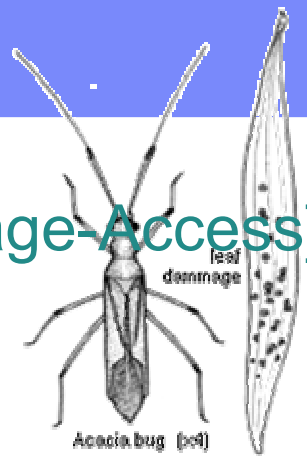
- Introduction
- Abstract categories of concurrent bugs
  - Explain why the I(P)-C(P) gap is created
- Sub-categories of concurrent bugs or
  - Bug tales
- Testing: how to reveal concurrent bugs

The gap is created when there is

1. A weak reality principle: the programmer incorrectly assumes that a code segment is protected
   - As in the first example
2. Denial: the programmer incorrectly assumes that an interleaving is impossible
   - Fork/join design pattern when the join is “implemented” using sleep()
3. Blocking: the programmer incorrectly assumes that a code segment will never block (i.e., wait for an event indefinitely)
   - A server assumes that incoming messages will arrive, but they never do

Weak reality bug tales [Not-Atomic]

- An operation is assumed to be atomic but is actually not
  - Source code operations often seem to the inexperienced programmer to be atomic when they are not
  - Example: x++



Weak reality bug tales (continued) [Two-Stage-Access]

Two stage access:

- We are given two tables
- To change a record in the second table, the first table is queried and then the second
- Each table is protected by a separate lock

lock [First query **key1 -> key2**]

window -> the tables can be changed here

lock [Second query **key2 -> record to be changed**]

Weak reality bug tales (continued) [Wrong/No-Lock]

Wrong lock or no lock

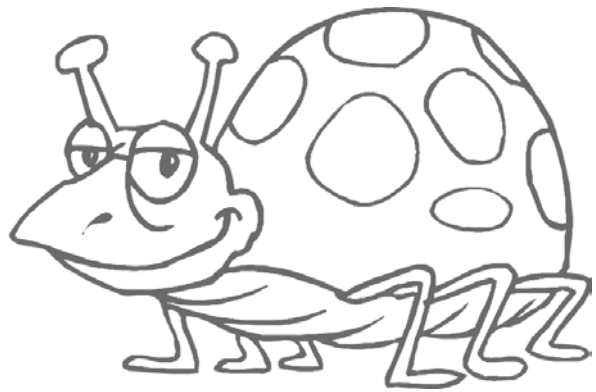
- Protection of thread one does not apply to thread two
- There is an access protocol that is not followed due to:
  - A new team member
  - An attempt to improve performance

Thread 1

```
Synchronized (o){
   x++;
}
```

Thread 2

```
x++;
```



Weak reality bug tales (continued) [Double-Check-Locking]

- Double-check locking
  - At object initialization time, the thread local copy of object fields is initialized but not written to the heap
  - Result: heap view is partially initialized while reference is not null
  - Source code level is misleading (looks atomic)
  - Bug pattern is well-documented on the internet

Denial bug tales [Initialization-Sleep]

- One example is adding sleep() statements to ensure that only the correct interleavings occur
  - As in the fork/join example, partial non-consistent results are used at the join stage

Denial bug tales [Lost-Notify]

- Losing notify: the notify is “lost” because it occurs before the thread executes the wait() primitive
  - The gap was created because the programmer didn’t think the notify would occur before the wait

Thread 1

```
Synchronized (o){
   o.wait();
}
```

Thread 2

```
synchronized (o){
    o.notifyAll();
}
```

Denial bug tales [Condition-For-Wait]

- Missing condition enclosing the wait
  - When returning from a wait the programmer forgets to check, or checks incorrectly if the reason for which he waited still holds
    - When returning from a wait with timeout the programmer assumes that a condition is met

Denial bug tales [Non-Commutative]

- Order of operations matter when you take into account interference and does not if the operations were atomic
  - A pool of some structure is handled by the system; a structure is accessed by the users iff its global reference is not null
  - When returning the structure to the pool
    - A global reference to the structure is set to null in a way that is viewable by other threads while keeping a local reference
    - Next, using the local reference, the fields of the structure are set to null and then the structure is reused
    - If the CPU is lost after the global reference is set to null the non consistent structure is not accessed
  - If the setting of the global reference is done after the setting of the structure fields and the CPU is lost in the middle other threads would access a non consistent structure

Denial bug tales [Unintentional-Different-Thread]

- A call to an API (typically a GUI API) is assumed to be in the same thread but is actually in a different thread causing the order of locking to sometimes change resulting in a deadlock

Blocking bug tales [Blocking-Critical-Section]

- Blocking critical section
  - In the design of a critical section protocol we assume that the thread executing the critical section will eventually exit
  - This assumption might be broken if the code is written by a third party or a different group

Blocking bug tales [Orphaned-Thread]

- The tale of the orphaned thread
  - A single master thread drives actions of other threads
  - Messages are put on the queue by the master thread and processed by the worker's threads
  - Abnormal termination of the master thread results in the remaining threads being orphaned
    - The system often blocks

Using ConTest to increase the probability that a concurrent bug occurs

- How can ConTest increase the probability that a concurrent bug occurs?
  - ConTest gains control of the execution before and after concurrent events
  - ConTest randomly chooses a thread and prevents its advancement until other threads stop advancing
  - The chosen thread executes:

```
otherAdvancing = true
While(otherAdvancing) { otherAdvancing = false; sleep(duration); }
```

  - Other threads execute: otherAdvancing = true;
- Why is the probability of finding the lost notify() bug increased?
  - When the chosen thread is the thread about to execute the wait(), the notify() will get lost

Summary

- The interleaving space can be used to categorized concurrent bugs
- In addition to the known deadlock bug, there are three types of concurrent bugs
  - Weak reality: non-protected code assumed to be protected
  - Denial: interleaving assumed to never occur
  - Blocking: blocking code assumed to be non-blocking
- Examples were given but more are needed
- Identifying and categorizing concurrent bugs motivates the creation of new testing techniques