Software Security 7

Sven Schäge

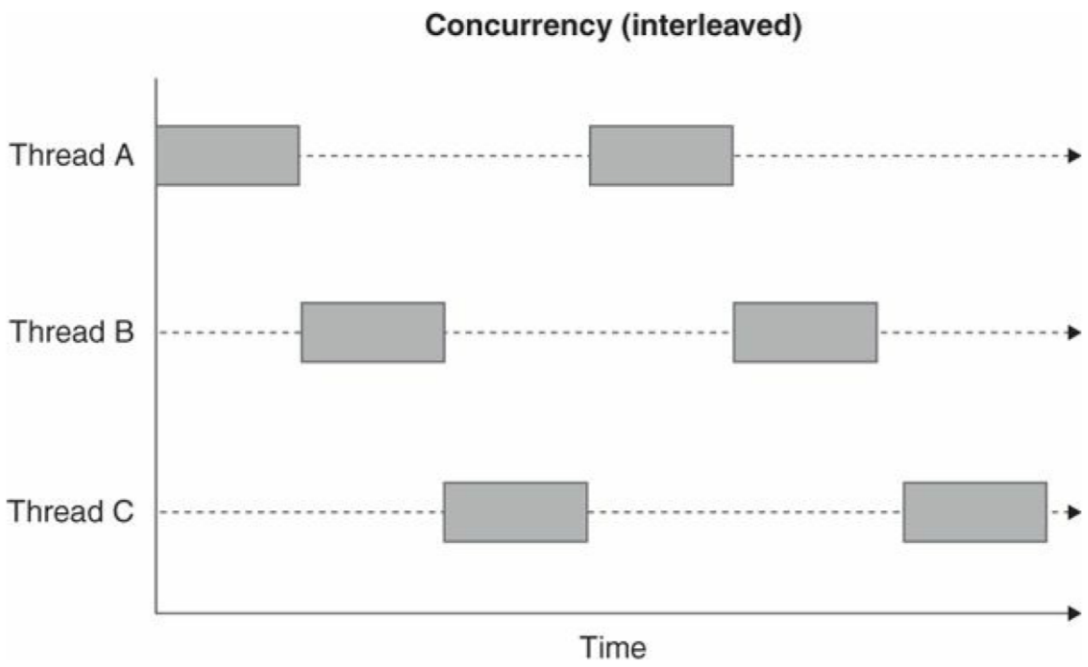
Learning Goals

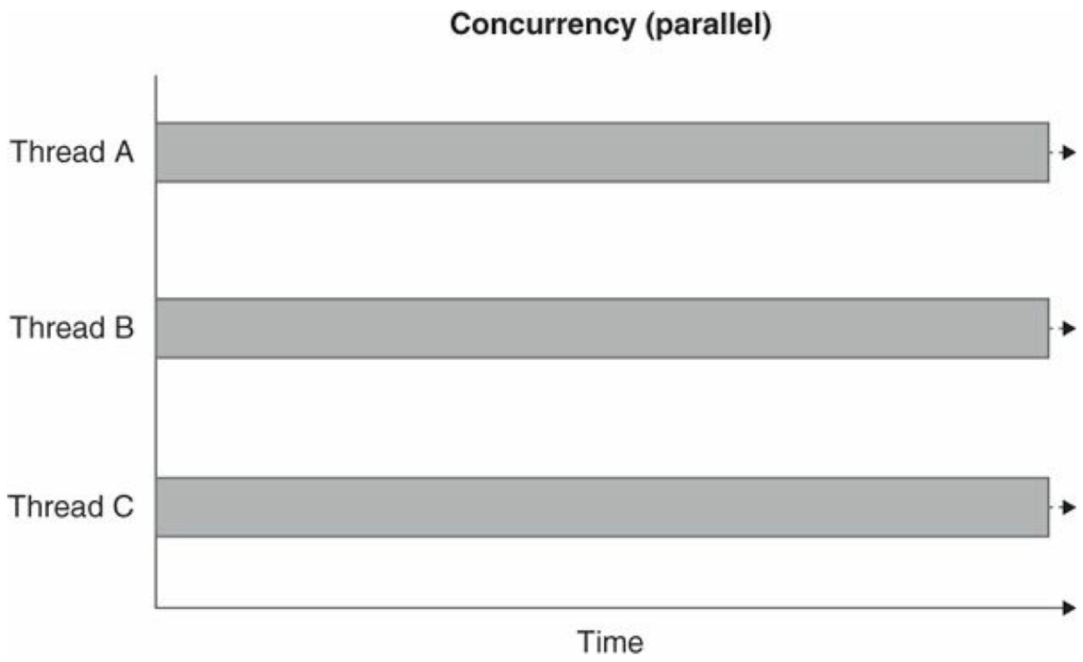
- Understand and quantify the benefits of concurrency in computing.
- Explain why concurrency problems can lead to vulnerabilities and be able to provide examples.
- Reflect on the definition of race conditions.
- Identify in various distributed settings application scenarios that require protection against concurrency problems.
- Reflect on the difficulty of finding concurrency problems and compare with other security vulnerabilities.
- Understand threads, their benefits, and dangers.
- Know key mechanisms in synchronization and their applications.
- Understand and apply the concept of bug depth of concurrency bugs.
- Explain techniques for finding concurrency bugs in order of increasing sophistication. Reflect on their advantages and drawbacks.
- Understand Dekker’s algorithm and its usage and be able to sketch how it can be generalized to more sophisticated token-based, non-token-based, and quorum-based algorithms.

Overview

- Introduction
- Threads
- Race Conditions
- Fixing: Task Synchronisation
- Critical Sections
- Access to the Same Ressource
- Dekker‘s Algorithm for Two Parties

Why Concurrency?





Robert Seacord, Secure Coding in C and C++

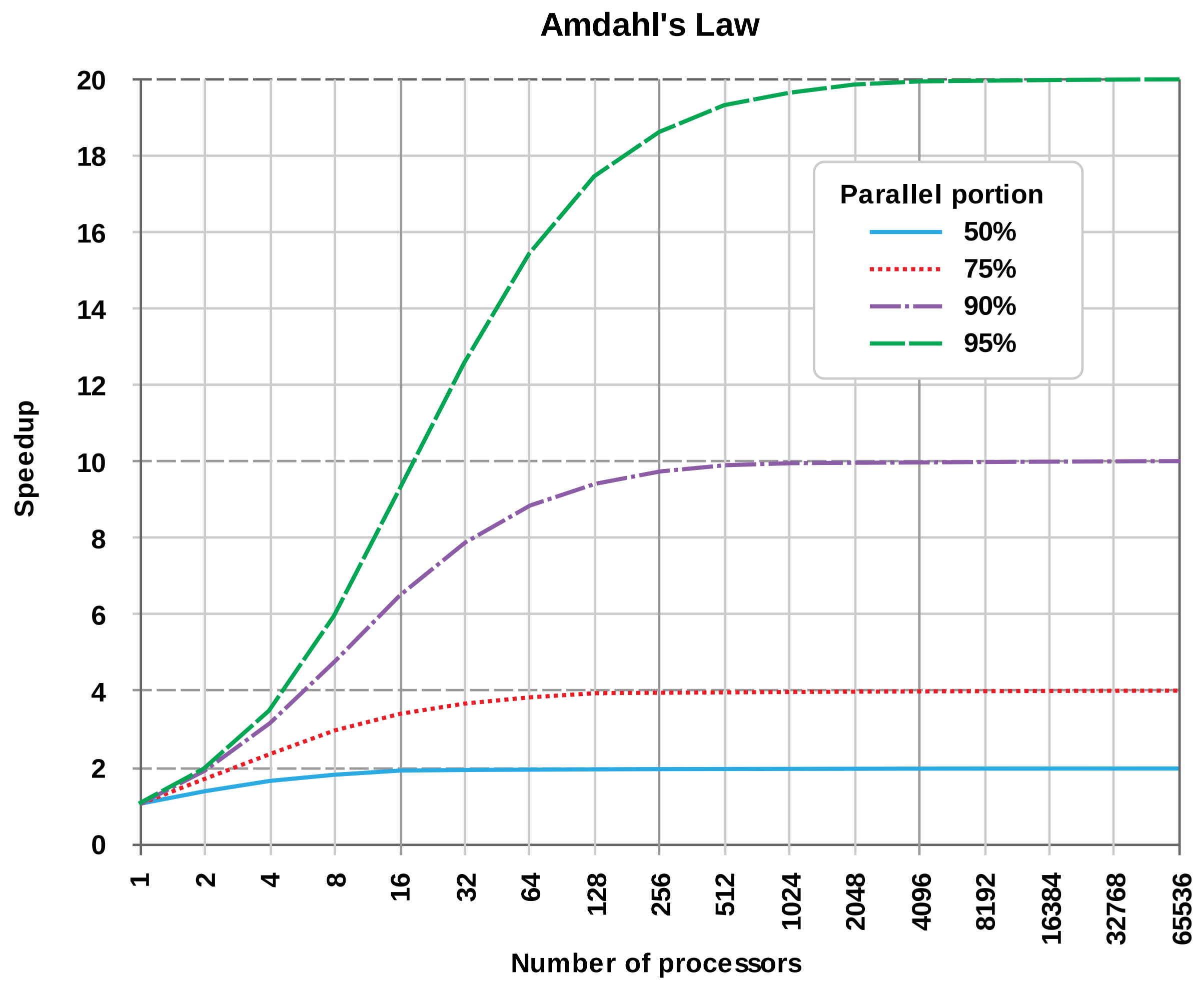
Amdahl‘s Law

- Let N be the number of processors
- Let P be the proportion of instructions that can be run in parallel

- Then the amount of speed-up achieved is at most

$$1/((1-P) + P/N)$$

Amdahl‘s Law


Amdahl's Law
Speedup
20
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors
Parallel portion
50%
75%
90%
95%

Concurrency Vulnerabilities

- ATM Example
- Assume Alice and Bob have shared account with a balance of EUR 100 and no credit line.
- Now they access the account independently at two distinct ATMs to retrieve the enitire money

1. Alice‘s ATM looks up whether the account does have entry EUR 100
2. Bob‘s ATM looks up whether the account does have entry EUR 100
3. Alice‘s ATM, now sure the transaction is valid, outputs EUR 100 and decreases the balance by EUR 100
4. Bob‘s ATM, now sure the transaction is valid, outputs EUR 100 and decreases the balance by EUR 100

Time-of-check-time-of-use (TOCTOU) Error

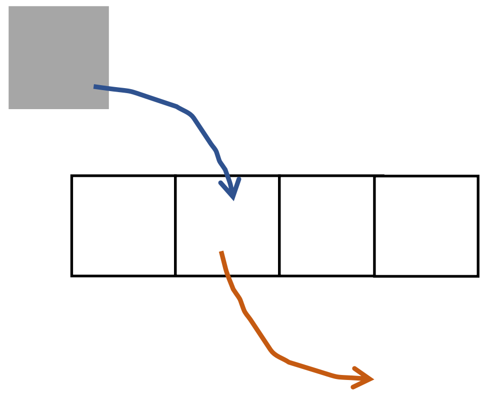
When Do we Face Concurrency Problems?

- Uncontrolled concurrency can lead to non-deterministic behaviour.
- A **race condition** occurs in any scenario in which two threads/processes can produce different behavior, depending on which thread completes first.
- Three conditions must be fulfilled:
  1. **Concurrency property**: At least two control flows must be executing concurrently.
  2. **Shared object property**: A shared race object must be accessed by both of the concurrent flows.
  3. **Change state property**: At least one of the control flows must alter the state of the race object.

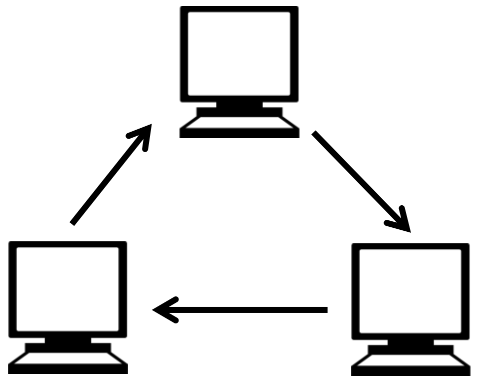
Concurrency Problems on Different Levels

- Threads (share the same memory)
- Interprocess-Communication
- Distributed Environments

Multithreading



Event-driven



Message passing

Thread

- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.
  - Scheduler typically: Operating System
- The multiple threads of a given process may be executed concurrently (via multithreading capabilities), **sharing resources** such as memory, while different processes do not share these resources.
  - Vulnerable to Race Conditions
- In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time.

Concurrency Vulnerabilities

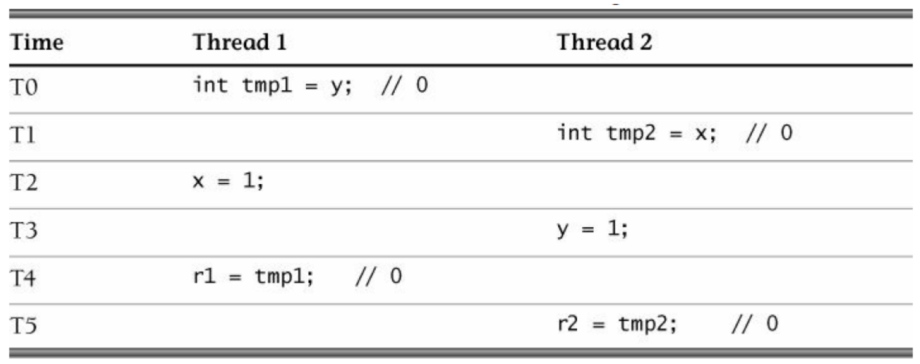
- Dekker‘s Example:

```
int x = 0, y = 0, r1 = 0, r2 = 0;
// Thread 1                   // Thread 2
 x = 1;                       y = 1;
 r1 = y;                      r2 = x;
```

Concurrency Vulnerabilities Race Conditions

- Dekker‘s Example:

```
int x = 0, y = 0, r1 = 0, r2 = 0;
// Thread 1                    // Thread 2
 x = 1;                        y = 1;
 r1 = y;                       r2 = x;
```



Time	Thread 1	Thread 2
T0	`int tmp1 = y;  // 0`	
T1		`int tmp2 = x;  // 0`
T2	`x = 1;`	
T3		`y = 1;`
T4	`r1 = tmp1;   // 0`	
T5		`r2 = tmp2;    // 0`

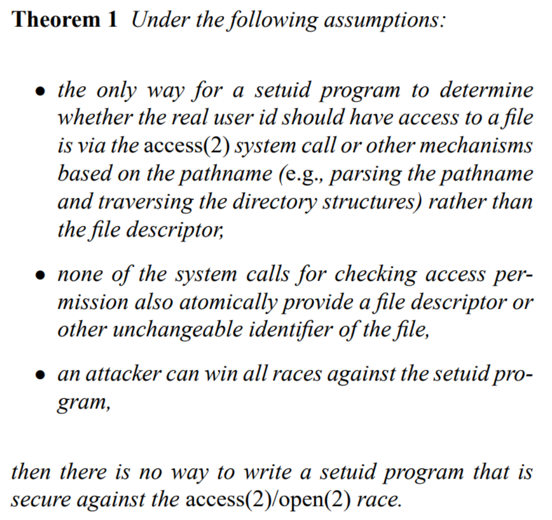
- Ideally, when both threads complete, both `r1` and `r2` are set to 1. However, the code lacks protection against race conditions. Therefore, it is possible for thread 1 to complete before thread 2 begins, in which case `r1` will still be 0.
- Likewise, it is possible for `r2` to remain 0 instead.
- However, there is a fourth possible scenario: both threads could complete with both `r1` and `r2` still set to 0!
- This scenario is possible only **because compilers and processors are allowed to reorder the events in each thread**, with the stipulation that each thread must behave as if the actions were executed in the order specified.

Race Window and Critical Section

- Race Window
  - Eliminating race conditions begins with identifying race windows. A race window is a code segment that accesses the race object in a way that opens a window of opportunity during which other concurrent flows could “race in” and alter the race object. Furthermore, a race window is not protected by a lock or any other mechanism.
- Critical Section
  - A race window protected by a lock or by a lock-free mechanism is called a critical section.

Why are Concurrency Problems Hard to Spot?

- Errors due to race conditions are hard to find or reproduce: Race conditions are particularly insidious because they are timing dependent and manifest sporadically. As a result, they are difficult to detect, reproduce, and eliminate and can cause errors such as data corruption or crashes.
  - Heisenbug
- Race conditions depend on the Meta-Context that is not immediately related to solving a programming task and often are non-local (effects that extend beyond the immediate scope of the code where the error originates)
  - **Race conditions result from runtime environments**, including operating systems that must control access to shared resources, especially through process scheduling. It is the programmer's responsibility to ensure that his or her code is properly sequenced regardless of how runtime environments schedule execution (given known constraints).
- Under certain conditions, Race Conditions are unavoidable…



Theorem 1 *Under the following assumptions:*

- *the only way for a setuid program to determine whether the real user id should have access to a file is via the* access(2) *system call or other mechanisms based on the pathname* (e.g., *parsing the pathname and traversing the directory structures) rather than the file descriptor,*
- *none of the system calls for checking access permission also atomically provide a file descriptor or other unchangeable identifier of the file,*
- *an attacker can win all races against the setuid program,*

then there is no way to write a setuid program that is secure against the access(2)/open(2) *race.*

D. Dean, A. J. Hu: Fixing Races for Fun and Profit: How to use *access(2)* (2004)

Examples C Code

The following code checks a file, then updates its contents.

```
struct stat *sb;
...
lstat("...",sb);   //gets info on some file and store it to sb
printf("stated file\n");
if (sb->st_mtimespec==...){ //compares time of last modification
print("Now updating things\n");
updateThings();
}
```

Synchronisation Mechanisms and Mutual Exclusion

- Synchronization in concurrent algorithms is crucial to ensure that multiple threads or processes can coordinate their actions and access shared resources without causing conflicts or unexpected behavior.
- Usually, the key problem is to solve the **Mutual Exclusion problem**.
  In this problem, there is a collection of asynchronous threads/processes, each alternately executing a critical and a noncritical section, that must be synchronized so that no two processes ever execute their critical sections concurrently.

Key Mechanisms in Synchronization

Locks-based Approaches:

- **Mutex (Mutual Exclusion):** A mutex is a synchronization primitive that provides exclusive access to a shared resource.
  Only one thread or process can hold the mutex at a time. Other threads attempting to acquire the mutex are blocked until the owning thread releases it.
- **Read-write locks** distinguish between read-only and write access. Multiple threads can hold the read lock simultaneously, but only one thread can hold the write lock.
- S**emaphores:** A semaphore is a synchronization object that maintains a count. It allows multiple threads to enter a critical section up to a specified limit. Semaphores can be binary (allowing only one or zero permits) or count-based.
- **Condition Variables:** Condition variables are used for signaling ordering between threads. They allow a thread to voluntarily release a lock and wait for a condition to be satisfied. When another thread signals that the condition is met, the waiting thread is awakened.

Atomic Operations:

- Atomic operations are indivisible and uninterruptible, ensuring that they are executed as a single, unbreakable unit.
  This is often used for simple operations on primitive data types, and many programming languages provide atomic operations.

Barrier Synchronization:

- Barriers are synchronization points where multiple threads wait until all threads have reached the barrier before any of them proceeds.
  They are often used in parallel computing as well.

Message Passing:

- Processes communicate through message passing, ensuring synchronization by exchanging messages.
  This can be done through various inter-process communication (IPC) mechanisms.

Finding Concurrency Bugs in Program

- Finding concurrency bugs requires finding
  - Inputs
  - **AND** Thread Schedules

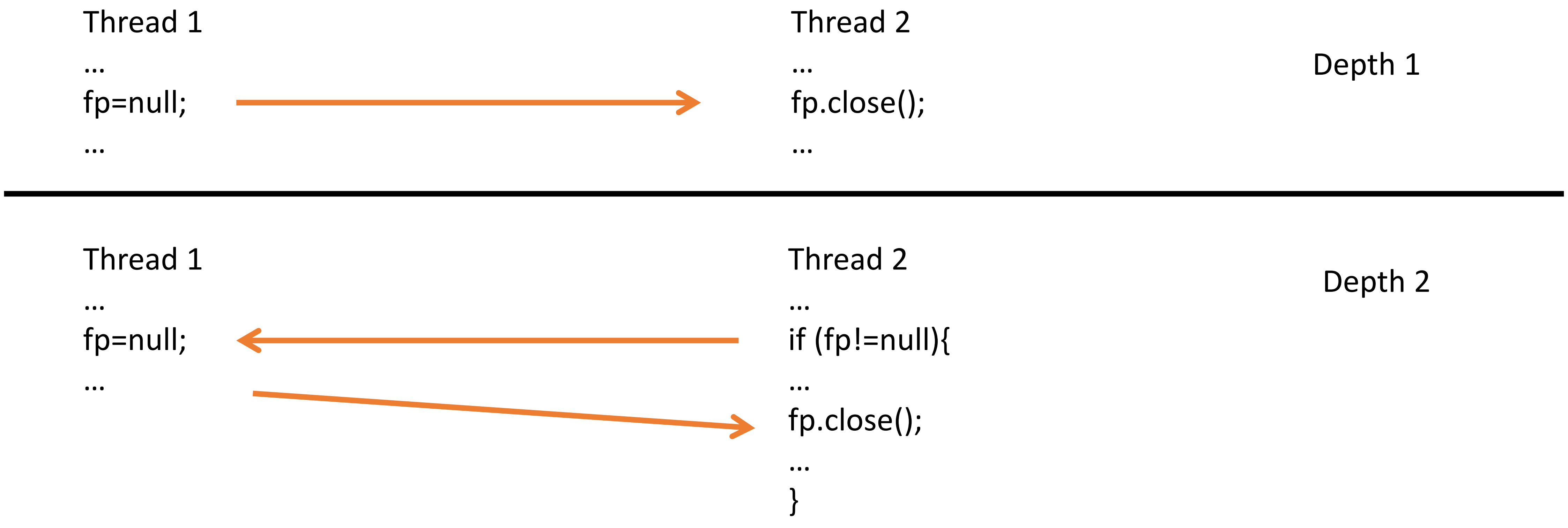
that trigger a bug!

Concurrency Bugs

- Non-Deadlocks
  - Atomicity Violation
    - The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)
    - Lock-based synchronization mechanisms can enforce atomicity
  - Order Violation
    - The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)
    - Condition Variables can fix these problems. The signal when a certain state has been reached and may allow code to only continue then.
- Deadlocks
  - Mutual exclusion: Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
  - Hold-and-wait: Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
  - No preemption: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
  - Circular wait: There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

Depth of Concurrency Bug

- Bug Depth: the number of ordering constraints a schedule has to satisfy to find the bug
- Consider filepointer fp that is declared to and points to some opened file. Closing fp=null is considered an error.



Thread 1

```
…
fp=null;
…
```

Thread 2

```
…
fp.close();
…
```

Depth 1

Thread 1

```
…
fp=null;
…
```

Thread 2

```
…
if (fp!=null){
…
fp.close();
…
}
```

Depth 2

Bug Depth

- Typically, concurrency bugs do not have high depth.
    - Most concurrency bugs will not have a large number of pre-requisites on the cocnurrency schedule to occur.
- High depth means that more conditions have to be fulfilled for the bug to trigger, making it generally less likely to occur.

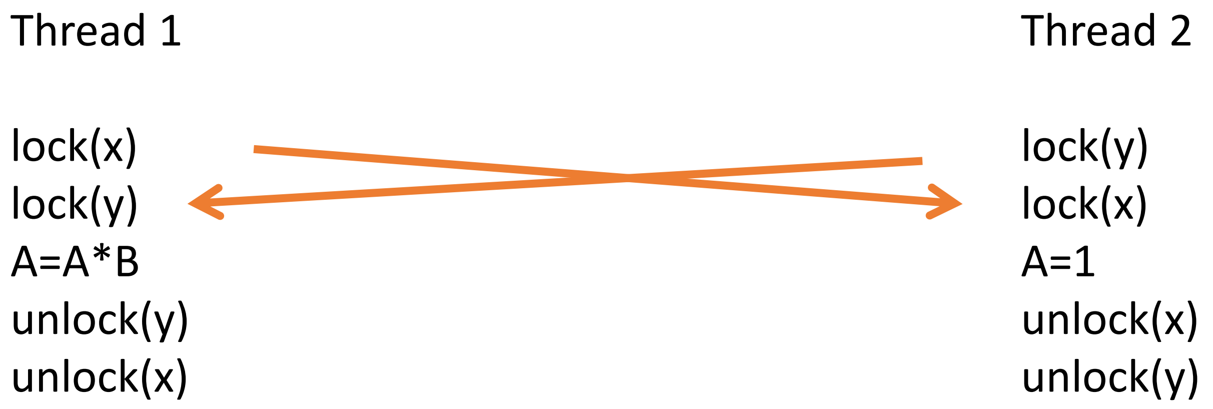
Depth of Concurrency Bug

- Assume that the lock() method acquires a lock on the specified variable while the unlock() method releases the lock on the specified variable.
- Locks are a means of enforcing mutual exclusion between threads: at most one thread can hold a lock on a given variable at any instant.
- A thread that attempts to acquire a lock that is held by another thread blocks and cannot execute any statements until the other thread releases the lock. Thread 1

Thread 1	Thread 2
lock(x)	lock(y)
lock(y)	lock(x)
A=A*B	A=1
unlock(y)	unlock(x)
unlock(x)	unlock(y)

Depth of Concurrency Bug

- Assume that the lock() method acquires a lock on the specified variable while the unlock() method releases the lock on the specified variable.
- Locks are a means of enforcing mutual exclusion between threads: at most one thread can hold a lock on a given variable at any instant.
- A thread that attempts to acquire a lock that is held by another thread blocks and cannot execute any statements until the other thread releases the lock.

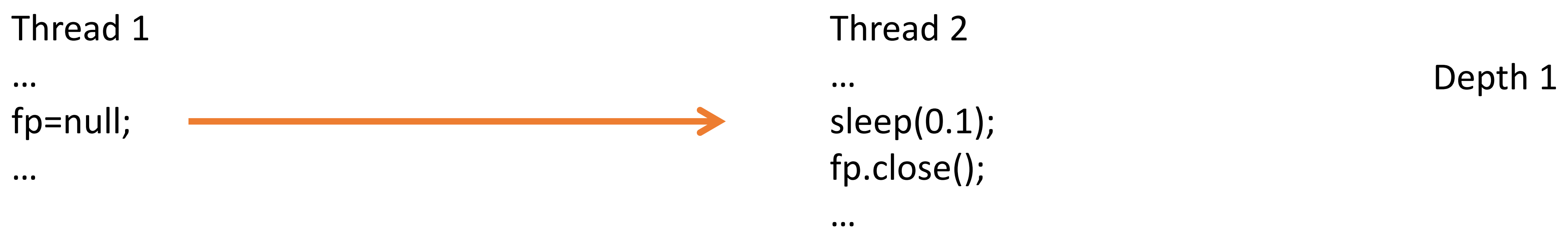


```
Thread 1                Thread 2

lock(x)                 lock(y)
lock(y)                 lock(x)
A=A*B                   A=1
unlock(y)               unlock(x)
unlock(x)               unlock(y)
```

Finding Concurrency Bugs – Central Idea

- Since Bugs Depend on the Concurrency Schedule which is under external control they are tricky to trigger logically
- Using delay statement like sleep() [~100ms] before statements can help to force a specific order into the execution



```
Thread 1
…
fp=null;
…
```

```
Thread 2
…
sleep(0.1);
fp.close();
…
```

- This will likely trigger the bug!

Concurrency Testing

- Using this central idea, we can now consider software tests for concurrency

1. Instrument the code with sleep() statements.
2. Perform fuzzing (dynamic testing)

- What remains to specify is how 1. should be performed.

Strategies for Concurrency Testing via Delays

- Probabilistic Analysis
  - Introduce delays at random and run many times
- Exhaustive testing
  - Each statement will either be preceeded by a delay or not
  - For n statements this will result in a search space of 2^n
  - Expensive
- Improved Exhaustive Testing
  - Introduce and concentrate on subset of statements: Preemption Points
    - Too few=> bugs not found
    - Too many=> computationally infeasible
    - Lock acquired, memory operations
-

Concurrency Invariants

- Idea: Concurrency Bugs occurs=>Some invariant has been violated before
- New idea: make sure that invariant is fullfiled
- Can also find non-triggered bugs
- =>Less expensive since we only have to consider program runs that reach to the violation of the invariant
- => Exact information on race conditions are not required!
- False positives

Invariants

- Lock-Sets: Any access to shared memory is protected by locks mechanism
- Data-races: Desireable: No concurrent read/write to same memory location
    - Very fragile to timings of thread executions (maybe by only a single instruction in one of the threats)
    - Testing not very pragmatic
- Happens-before-checking: No unordered reads and writes to the same memory location
    - Not: what is exactly racing at a given time
    - Collect happens-before-dependencies
        - Define if Thread X needs to be run before Thread Y
        - If there is no dependency between Thread X and Thread Y and they access the same memory, then they could theoretically race
    - More general than Lock-Sets Variant

Algorithms for Mutual Exclusion

Dekker‘s Algorithm

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Concurrency Control-Algorithms for Mutual Exclusion in Distributed Systems

- General Algorithms for Mutual Exclusion in Distributed Systems
  - Token-based
  - Non-token based
  - Quorum-based

- Many algorithms do not work if out-of-order execution is used on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread!

Required Properties

- Requirements of Mutual Exclusion Algorithm:
- No Deadlock: Two or more site should not endlessly wait for any message that will never arrive.
- No Starvation: Every site who wants to execute the critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute the critical section while other sites are repeatedly executing critical section
- Fairness: Each site should get a fair chance to execute the critical section. Any request to execute the critical section must be executed in the order they are made i.e. critical section execution requests should be executed in the order of their arrival in the system.
- Fault Tolerance: In case of failure, the system should be able to recognize the failure by itself in order to continue functioning without any disruption.

Token-based Algorithms

- Token Based Algorithm:
- A unique token is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence numbers to order requests for the critical section.
- Each requests for the critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach ensures Mutual exclusion as the token is unique

Non-token based Algorithm

- Non-token based approach:
- A site communicates with other sites in order to determine which sites should execute the critical section next. This requires the exchange of two or more successive round of messages among sites.
- This approach uses timestamps instead of sequence number to order requests for the critical section.
- When ever a site makes a request for the critical section, it gets a timestamp. The timestamp is also used to resolve any conflict between critical section requests.
- All algorithms which follows the non-token based approach maintain a logical clock. Logical clocks get updated according to Lamport's scheme

Lamport‘s Timestamps: Algorithms

- Sending
  - # event is known
  - time = time + 1;
  - # event happens
  - send(message, time);

- Receiving
  - (message, time_stamp) = receive();
  - time = max(time_stamp, time) + 1;

Lamports Timestamps

- Lamport timestamps are a method for ordering events in a distributed system to establish a partial ordering that reflects causality between events.
- The basic idea behind Lamport timestamps is to assign a unique timestamp to each event in the system. The timestamp is a pair consisting of a logical clock value and an identifier for the process that generated the event. The logical clock is used to order events, and it ensures that events that are causally related have a consistent ordering across the entire distributed system.

Lamport‘s Timestamps: Algorithms

1. Each process in the distributed system maintains a logical clock.
2. When a process generates an event (sending a message, receiving a message, etc.), it increments its logical clock.
3. The process includes its current logical clock value in the event.
4. When a process receives a message, it updates its logical clock to be greater than the maximum of its current logical clock and the timestamp received in the message, then increments the logical clock.

   By comparing Lamport timestamps, you can determine the ordering of events, even if they occurred in different processes. If the timestamp of event A is less than the timestamp of event B, then A causally precedes B. However, if the timestamps are equal, the events are concurrent.

Quorum-based Algorithm

- **Quorum based approach:**
- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion