UTA

## Race Analysis

- Introduction
- The Lockset Algorithm
- Message Race Detection



UTA

## Race Conditions

Many factors, such as unpredictable process scheduling and variations in message latencies, can contribute to the non-deterministic behavior of concurrent programs.

These factors can be informally referred to as race conditions or just races. The activity of identifying races is referred to as race analysis.

UTA

## General & Data Races

There are two different types of races that capture different kinds of bugs.

General races cause nondeterministic execution and are failures in programs intended to be deterministic.

Data races cause non-atomic execution of critical sections and are failures in programs that access and update shared data in critical sections.



UTA

## Examples

```
Process 1
/* deposit */
amount = read_amount ();
P (mutex);
balance = balance + amount;
interest = interest + rate * balance;
V (mutex);

Process 2
/* withdraw */
amount = read_amount ();
P (mutex);
if (balance < amount)
    printf ("NSF");
else
    balance = balance – amount;
    interest = interest + rate * balance;
V (mutex);
```

```
Process 1
/* deposit */
amount = read_amount ();
P (mutex);
balance = balance + amount;
interest = interest + rate * balance;
V (mutex);

Process 2
/* withdraw */
amount = read_amount ();
if (balance < amount)
    printf ("NSF");
else
    balance = balance – amount;
    interest = interest + rate * balance;
```

UTA

## Race Analysis

- Introduction
- The Lockset Algorithm
- Message Race Detection



UTA

## Locking Discipline

A locking discipline is a programming policy that ensures the absence of data races.

For example, a simple locking discipline is to require that every variable shared between threads is protected by a mutual exclusion lock.

The lockset algorithm is to ensure that all shared-memory accesses follow the simple locking discipline.

UTA

## The Algorithm (1)

For each shared variable v, the algorithm maintains the set C(v) of candidate locks as follows:

- When a variable is initialized, C(v) contains all possible locks;
- When a variable is accessed, C(v) is replaced with the intersection of C(v) and the set of locks held by the current thread.
- If C(v) becomes empty, it signals that there is no lock that consistently protects v. (Otherwise, there is at least one lock that remains in C(v).)



UTA

## The Algorithm (2)

```
for each shared variable v, initialize CandidateLocks(v) to the set of all locks
On each read or write access to v by thread T
    CandidateLocks(v) = CandidateLocks(v) ∩ LocksHeld (T)
    if (CandidateLocks(v) == {}) issue a warning.
```

UTA

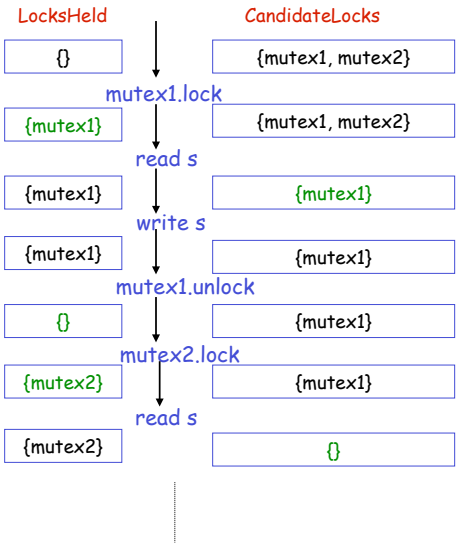
# Example (1)

| Thread 1 | LocksHeld(Thread1) | CandidateLocks(s) |
|---|---|---|
| | { } | { mutex1, mutex2} |
| mutex1.lock(); | { mutex1 } | { mutex1, mutex2 } |
| s = s + 1; | { mutex1 } | { mutex1, mutex2} |
| mutex1.unlock(); | { } | { mutex1 } |
| mutex2.lock(); | { mutex2 } | { mutex1 } |
| s = s + 2; | { mutex2 } | { } |
| mutex2.unlock(); | { } | { } |



UTA

# Example (2)



UTA

## False Alarms

The simple locking discipline is strict, in the sense that some common programming practices violate this discipline and are still free from data races:

- Initialization: Shared variables are frequently initialized without a lock.
- Read-shared Data: Some shared variables are written during initialization and are read-only thereafter.



UTA

## Initialization

There is no need for a thread to lock out others if no other can possibly hold a reference to the data being accessed.

To avoid false alarms caused by unlocked writes during initialization, we delay the refinement of a location's candidate set until after it has been initialized.

However, there is no easy way of knowing initialization is complete. Any solution?

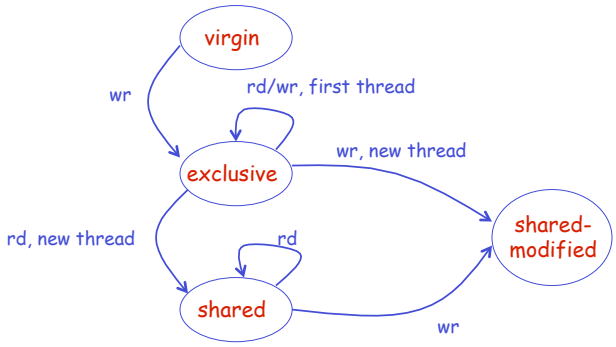
## Read-shared Data

Since simultaneous reads of a shared variable by multiple threads are not races, there is also no need to protect variable if it is read-only.

To avoid false alarms caused by unlocked read-sharing for such data, we report races only after an initialized variable has become write-shared by more than one thread.



## State Transitions



UTA

## Race Analysis

- Introduction
- The Lockset Algorithm
- Message Race Detection



UTA

## Asynchronous vs. Synchronous

In an asynchronous message-passing program, each process uses non-blocking send and blocking receive operations to send and receive messages.

This is in contrast with synchronous message-passing, where blocking send and blocking receive operations are used.

UTA

## Communication Scheme (1)

A communication scheme may impose certain restrictions on the relationship between the order in which messages are sent and the order in which messages are received.

The behavior of an asynchronous message-passing program depends on its underlying communication scheme.



UTA

## Communication Scheme (2)

The fully asynchronous scheme allows that messages are received out of order.

The FIFO scheme requires that messages exchanged between any two processes be received in the same order as they are sent.

The causal scheme requires that all the messages are received in the same order as they are sent.

## SR-sequence

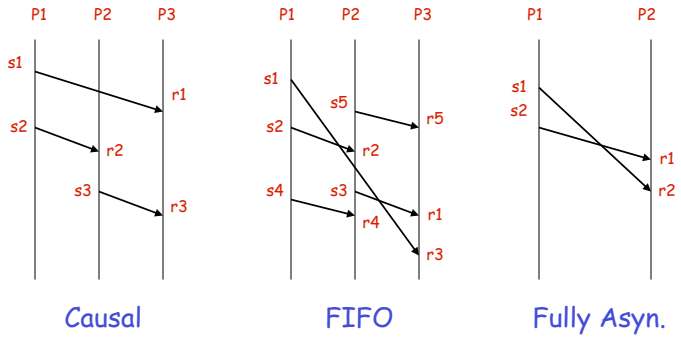
An execution of an asynchronous message-passing program exercises a sequence of send and receive events, or an SR-sequence.

Formally, an SR-sequence is defined as a triple (S, R, Y), where S is a set of send events, R a set of receive events, and Y a set of synchronizations.



## Example

Causal ⊂ FIFO ⊂ Fully Asyn.




Causal FIFO Fully Asyn.

UTA

## Message Race (1)

Informally, there exists a message race between two send events if the messages sent by the two events may be received by the same receive event.

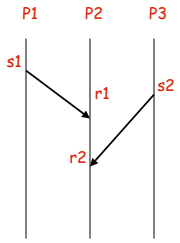
Given an SR-sequence, we can perform race analysis, and determine, for each receive event, the set of send events that may send messages to this receive event.



UTA

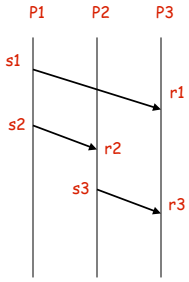
## Message Race (2)

Message race shall be independent from program implementation.

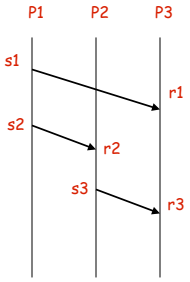


## Message Race (3)

Message race depends on the underlying communication scheme.




Causal




FIFO or Asyn.



## Message Race Detection (1)

Let P be an asynchronous message-passing program consisting of processes P1, P2, ..., and Pn. Let Q be the SR-sequence exercised by one execution of P.

Let r is a receive event in Q. Assume that r receives the message sent by a send event s in Q. A send event s' is in the race set, race(r), of r if the message sent by s' can be received by r in another execution.

Race analysis of Q is to determine race(r) for every receive event r in Q.

## Message Race Detection (2)

Assume that the fully asynchronous communication scheme is used. Let r be a receive event of Pi. A send event s of Pj is in race(r) if the following two conditions are met:

- the message sent by s is destined to Pi.
- r does not happen before s
- if s is received by r', then r must happen before r'.



## Message Race Detection (3)

Assume that the FIFO communication scheme is used. Let r be a receive event of Pi. A send event s of Pj is in race(r) if the following three conditions are met:

- the message sent by s is destined to Pi.
- r does not happen before s
- if s is received by r', then r must happen before r'.
- there does not exist another send event s' of Pj such that s' happens before s and either s' satisfies the above two conditions or is synchronized with r.

## Message Race Detection (4)

Assume that the Causal communication scheme is used. Let r be a receive event of Pi. A send event s of Pj is in race(r) if the following three conditions are met:

- the message sent by s is destined to Pi.
- r does not happen before s
- if s is received by r', then r must happen before r'.
- there does not exist another send event s' of any process such that s' happens before s and either s' satisfies the above two conditions or is synchronized with r.



## Message Race Detection (5)