

Today's Agenda

- Quick Review
- Finish **The Critical Section Problem**
- **Replay Shared Variables**

Quick Review

- What are the three requirements for the critical section problem?

Tracing/Replay for Shared Variables

- Introduction
- RW-Sequence
- Tracing and Replay
- Implementation Issues

Tracing and Replay

Replay is to repeat the behavior of a previous execution of a concurrent program.

Tracing is to capture necessary information during a concurrent execution such that it can be repeated later in a controlled fashion.

Challenges

There are a number of challenges to address for tracing and replay:

- ❑ What exactly should be replayed?
- ❑ What information should be captured?
- ❑ How should such information be encoded?

SYN-sequence

A **SYN-sequence** is a sequence of synchronization operations, i.e. operations that are performed on shared variables or synchronization constructs.

A concurrent execution can be repeated, in terms of producing the same results, if the same **SYN-sequence** is replayed.

SYN-sequence (cont'd)

A SYN-sequence can be **object-based** or **thread-based**.

An object-based SYN-sequence is associated with a shared **object**, consisting of all synchronization operations that are executed on that object.

A thread-based SYN-sequence is associated with a **thread**, consisting of all synchronization operations that are executed by that thread.

SYN-sequence (cont'd)

Replay based on SYN-sequences can be accomplished by inserting additional control into programming objects.

The alternative is to record thread schedules, i.e., we record information about how context switches are performed.

Replay based on **thread schedules** often involve modifications to the thread scheduler, which is usually more difficult to implement, but can be more efficient.

Tracing/Replay for Shared Variables

- Introduction
- **RW-sequence**
- Tracing and Replay
- Implementation Issues

Definition

A SYN-sequence for a shared variable is a sequence of read and write operations, called a **RW-sequence**.

Each **write** operation on a shared variable creates a new version of the variable.

An execution can be replayed by ensuring each thread **reads** and **writes** the same variable versions that were recorded in the execution trace.

Encoding

For each variable, we keep track of its versions: a **write** operation increases the version number; a **read** operation keeps the version number unchanged.

In a RW-sequence, a read event is encoded as **(ID, version)**, where **ID** is the **ID** of the thread that executes the read event, and **version** is the current version number.

Encoding (cont'd)

A write event is encoded as **(ID, version, total reads)**, where **ID** is the **ID** of the thread that executes the write operation, **version** is the old version number, and **total reads** is the number of read operations that read the old version of the variable.

Encoding (cont'd)

There are two important points to be made about the definition of RW-sequence:

- We record the **order** in which read and write operations are performed, not the **values** that are read and written.
- There is no need to record any **type** information.

Example

<u>T1</u>	<u>T2</u>	<u>T3</u>
temp = s;	temp = s;	s = s + 1;

Assume that s is initialized to 0. A possible RW-sequence of s is:

(3, 0)
 (1, 0)
 (3, 0, 2)
 (2, 1)

Tracing/Replay for Shared Variables

- Introduction
- RW-sequence
- Tracing and Replay
- Implementation Issues

Tracing and Replay

During program tracing, a **RW-sequence** is recorded for each shared variable.

During program replay, we make sure that each read/write operation reads/writes the same version of shared variable as recorded in a RW-sequence.

Instrumentation

Read and write operations are instrumented by adding routines to control the start and end of the operation:

read

```
startRead(x)
read the value of x
endRead(x)
```

write

```
startWrite(x)
write the value of x
endWrite(x)
```

startRead & endRead

```
startRead (x)
  if (mode == trace) {
    ++ x.activeReaders;
    RWSeq.record(ID, x.version);
  }
  else {
    readEvent r = RWSeq.nextEvent (ID);
    myVersion = r.getVersion ();
    while (myVersion != x.version) delay;
  }
}
```

```
endRead (x)
  ++ x.totalReaders;
  -- x.activeReaders;
```

startWrite & endWrite

startWrite (x)

```
if (mode == trace) {
    while (x.activeReaders > 0) delay;
    RWSeq.record(ID, x.version, x.totalReaders);
}
else {
    writeEvent w = RWSeq.nextEvent (ID);
    myVersion = w.getVersion ();
    while (myVersion != x.version) delay;
    myTotalReaders = w.getTotalReaders ();
    while (x.totalReaders < myTotalReaders) delay;
}
```

endWrite (x)

```
x.totalReaders == 0;
++ x.version;
```

CREW

If two or more readers read a particular version during tracing, then they must **read** the same version during replay, but these **read** operations can be in any order.

In our solution, readers are only **partially ordered** with respect to write operations, while write operations are **totally ordered** for each shared variable.

This is known as **Concurrent Reading and Exclusive Writing (CREW)** policy.

C++ Implementation

The course pack has shown a template class `sharedVariable` that can be used to trace and replay C++ programs.

The only change involved is to wrap each variable as an instance of `sharedVariable`.

`int turn;` \Rightarrow `sharedVariable<int> turn;`

Java Implementation

More changes are necessary for Java programs, due to the lack of the features of template class and operator overloading.

However, such changes can be made automatically, without leaving developers with too much burden.