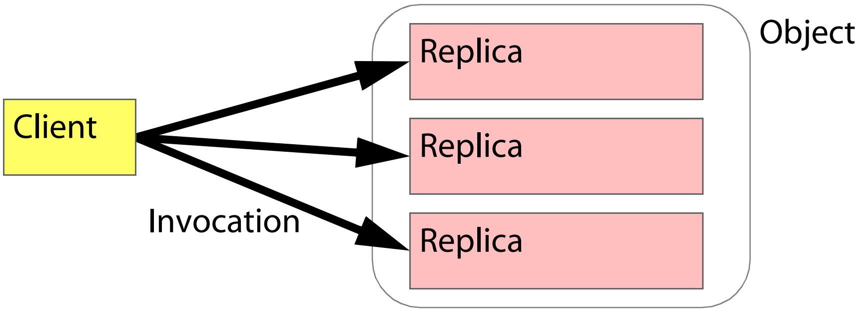
Franz J. Hauck
Institute of Distributed Systems, University of Ulm

Deterministic Scheduling for Replicated Systems

1 Active Replication

- Simultanous execution of invocation requests in all replicas



- Problem
  - Replicas have to be kept consistent!

1.1 Consistency of Replicas

- Requirement
  - all replicas have to reach the same internal state
- Deterministic execution of invocations
- Sources of non-determinism in replicas
  - different order of request processing
    - different order of incoming requests
    - different scheduling of worker threads
    - different arrival time of nested invocation responses
  - local invocation of non-deterministic operations or functions
    - e.g. `random()`, `getTimeOfDay()`, `getPID()` ...

1.1 Consistency of Replicas (2)

- Totally-ordered multicast
  - solves problems due to order of incoming requests or received responses
    - same order for all messages communicated to all replicas
- Mapping of non-deterministic operations to nested invocations
  - all replicas receive the same result of a single operation
- Focus of this talk:
  - **deterministic scheduling** of worker threads for concurrent invocation requests

2 Overview

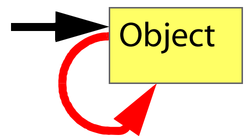
- Introduction
- Approaches to deterministic scheduling
  - sequential scheduling
  - SLT — Single Logical Thread
  - SAT — Single Active Thread
  - **ADETS/SAT** — SAT Extension for condition variables
  - LSA — Loose Synchronization Algorithm
  - PDS — Preemptive Deterministic Scheduling
  - **ADETS/MAT** — Multiple Active Threads
- Conclusion

3 Sequential Scheduling

- Sequential processing of invocation requests
  - next invocation is processed after the previous one was finished
- Disadvantage
  - bad utilisation of the CPU for nested invocations
    - undesired waiting time

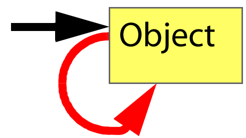
3 Sequential Scheduling

- Sequential processing of invocation requests
  - next invocation is processed after the previous one was finished
- Disadvantage
  - bad utilisation of the CPU for nested invocations
    - undesired waiting time
  - deadlock for self-invocations
    - execution at the own object blocks forever

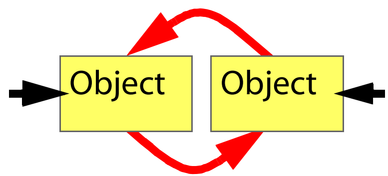


3 Sequential Scheduling

- Sequential processing of invocation requests
  - next invocation is processed after the previous one was finished
- Disadvantage
  - bad utilisation of the CPU for nested invocations
    - undesired waiting time
  - deadlock for self-invocations
    - execution at the own object blocks forever



  - deadlock for mutual invocations
    - two objects call each other and are stuck in a deadlock, as each object waits for the response of its request



3 Sequential Scheduling (2)

- Advantage
  - no additional communication for consistency
  - simple implicit coordination
- Standard in simple systems
  - e.g. GroupPac, OGS

3.1 Summary

	Seq
no additional communication	✔
no circular deadlocks	✖
no mutual deadlocks	✖
good CPU utilisation	✖

4 SLT — Single Logical Thread

- Detection of circular invocations
  - context information for each invocation
    - e.g., thread ID
  - return of the same context information identifies circular invocation
- Behaviour like in sequential scheduling, but
  - if current request processing is blocked due to a nested invocation, a circular request can be inserted
- ▲ Other problems persist
- Example
  - Eternal

4.1 Summary

	Seq	SLT
no additional communication	✔	✔
no circular deadlocks	✖	✔
no mutual deadlocks	✖	✖
good CPU utilisation	✖	✖

5 SAT — Single Active Thread

- Per invocation there is a single processing thread
  - in principle concurrent, but only one thread is allowed to run at each point in time
- Requires deterministic thread switches in each replica
- Coordination of concurrent threads
  - necessary for data consistence, even in non-replicated case
  - possible mechanisms for coordination:
    - Semaphores, Monitors, ...
- Idea: utilise coordination for consistency of replicas
  - i.e. deterministic thread switches at coordination points
- Example
  - Jimenez-Peris et al., Zhao et al.

5.1 Summary

	Seq	SLT	SAT
no additional communication	✔	✔	✔
no circular deadlocks	✖	✔	✔
no mutual deadlocks	✖	✖	✔
good CPU utilisation	✖	✖	✔
condition variables	✖	✖	✖

- Additional disadvantage
  - originally no algorithm for monitor-based coordination with condition variables
  - e.g. for Java-like coordination

6 ADETS/SAT — Single Active Thread

- ADETS = Aspectix Deterministic Thread Scheduling
  - ADETS/SAT = SAT-Scheduling with monitor-based coordination

6.1 Insertion: Java Coordination

- Binary Semaphores
  - each Java object is a binary semaphore
  - implicit locking and unlocking by `synchronized` statements

```
synchronized( obj ) {
    // do something nice
}
```

  - implicit locaking and unllocking by `synchronized` methods

```
synchronized void mymethod( int i ) {
    // do something nicer
}
```

- Condition variable
  - in Java there is one implicit condition variable per object/semaphore
  - `wait()`: thread releases lock and blocks for waiting
  - `notifiy()`: wakes up a one of the blocked threads,
    `notifiyAll()`: wakes up all blocked threads
  - Example: Bounded-Buffer

```
class BoundedBuffer { // ...
    synchronized int get() {
        while( /* buffer empty */ )
            wait();
        // take something out of buffer
        return something;
    }
    synchronized void put( int something ) {
        // put something into buffer
        notify();
        return;
    }
}
```

6.2 Deterministic Thread Switching

- Potential switching points
  - thread creation for processing a new requrest
  - thread termination
  - nested invocaton
  - reception of a response of a nested invocation
  - lock request
  - lock release
  - time slice end
  - priority changes
- Let’s exclude time slices and priority changes
  - all worker threads have same priority

6.2 Deterministic Thread Switching (2)

- Problem
  - request order is exactly defined, but not there arrival time
  - i.e., replicas can have made different progress
  - i.e., deterministic strategy has to decide the same regardless how far the local replica is

* Utilise coordination
  - non-deterministic processing of uncoordinated code sections is allowed
  - coordinated code section have to be executed in the same order in all replicas

- Schematic algorithm
  - if there is no thread running and a request comes in, a new thread will be started
  - if a thread is running an a request comes in, the request is enqueued in a message queue (MsgQueue)
  - if a thread has finished, a deterministic scheduling decision is made:
    - a new worker thread is started for a request from the MsgQueue

Up to now: **Sequential Scheduling**

6.3 ADETS/SAT Scheduling Algorithm

- Schematic algorithm
  - if there is no thread running and a request comes in, a new thread will be started
  - if a thread is running an a request comes in, the request is enqueued in a message queue (MsgQueue)
  - if a thread has finished, a deterministic scheduling decision is made:
    - a new worker thread is started for a request from the MsgQueue OR
    - process response in MsgQueue by waking up thread waiting for that response

  Up to now: **Sequential Scheduling**

  - if there is a nested invocation, the thread will be blocked and a deterministic scheduling decision is made
  - if there arrives a response for a nested invocation it is enqueued into MsgQueue

  Up to now: **SAT without coordination**
  Order of messages determines scheduling decisions

6.3 ADETS/SAT Scheduling Algorithm (2)

- Schematic algorithm (cont.)
  - if a thread `locks` a semaphore and
    - the semaphore is free, it will be locked
    - the semaphore is locked, the thread will be blocked and enqueed into a request queue (ReqQueue) and a deterministic scheduling decision is made
  - if a thread `unlocks` a semaphore, this will be registered;
    thread switching is delayed to the next scheduling decision to be made
  - extension of the scheduling decision:
    - if there is a lock request in ReqQueue and the lock is free, the lock will be granted and the thread is deblocked *(this choice has to be the first option)*

Up to now: **SAT with binary semaphores**

6.3 ADETS/SAT Scheduling Algorithm (3)

- Schematic algorithm (cont.)
  - if a thread calls `wait()` on a semaphore, the thread will be enqueued into a thread queue (WaitQueue) and blocked, the lock will be released and a scheduling decision is made
  - if a thread calls `notify()` or `notifyAll()`, the corresponding threads from WaitQueue are dequeued and enqueued in a deterministic order into ReqQueue

  Up to now: **SAT with binary semaphores and condition variables**

6.4 Summary

	Seq	SLT	SAT	A/SAT
no additional communication	✔	✔	✔	✔
no circular deadlocks	✖	✔	✔	✔
no mutual deadlocks	✖	✖	✔	✔
good CPU utilisation	✖	✖	✔	✔
condition variables	✖	✖	✖	✔
parallelism	✖	✖	✖	✖

- Disadvantage so far
  - no parallelism, i.e., no utilisation of multiprocessors and multi-core systems

7 LSA — Loose Synchronization Algorithm

- Leader follower model (Basile et al.)
  - a designated replica memorises scheduling decisions
    - e.g., lock granting order
  - designated replica sends out decision to all other replicas
  - other replicas decide not before leader has send its decisions
    - all lock request block in the beginning

- Disadvantage:
  - additional communication overhead
  - higher latency
  - intricate failure recovery

- Advantage
  - multiple threads can get different locks granted at the same time

7.1 Summary

	Seq	SLT	SAT	A/SAT	LSA
no additional communication	✔	✔	✔	✔	✖
no circular deadlocks	✖	✔	✔	✔	✔
no mutual deadlocks	✖	✖	✔	✔	✔
good CPU utilisation	✖	✖	✔	✔	✔
condition variables	✖	✖	✖	✔	✖
parallelism	✖	✖	✖	✖	✔
parallel lock granting					✔

8 PDS — Preemptive Deterministic Scheduling

- Round-based algorithm (Basile et al.)
  - fixed number of threads
  - in each round threads run until they terminate or request a lock
  - at the start of a new round:
    - lock requests are deterministically granted
    - new threads are started from a request queue until the fixed number is reached
  - several optimisations, e.g. in another version at most two locks can be granted within one round

8 PDS — Preemptive Deterministic Scheduling

- Disadvantage
  - fixed number of threads
  - if there are not enough requests, others have to wait (!)
  - otherwise the system can inject dummy requests
- Advantage
  - no additional messages
  - multiple threads can acquire different locks at the same round

8.1 Summary

	Seq	SLT	SAT	A/SAT	LSA	PDS
no additional communication	✔	✔	✔	✔	✖	✔
no circular deadlocks	✖	✔	✔	✔	✔	✔
no mutual deadlocks	✖	✖	✔	✔	✔	✔
good CPU utilisation	✖	✖	✔	✔	✔	✔
condition variables	✖	✖	✖	✔	✖	✖
parallelism	✖	✖	✖	✖	✔	✔
parallel lock granting					✔	✔
arbitrary thread number					✔	✖

9 ADETS/MAT — Multiple Active Threads

- Extension of the ADETS/SAT Algorithm for concurrent threads (Reiser et al.)
- Idea
  - primary thread behaves like the single thread of the ADETS/SAT algorithm
    - it can acquire locks
  - secondary threads can run concurrently and uncoordinated
    - but cannot acquire locks
  - switch from secondary to primary status is deterministic
  - a PrimaryCandidateQueue contains incoming requests sorted by the group communication system
- Coordination
  - like ADETS/SAT with Java like coordination with condition variable

9 ADETS/MAT — Multiple Active Threads (2)

* Advantage
  - no additional messages
  - arbitrary number of concurrent threads
    - can be mapped to multiple cores or processors

▲ Disadvantage
  - only one thread can aqcuire locks at a time
  - thread does only hand off primary status on termination or nested invocation
    - for some applications not relevant
    - for others fatal

9.1 Summary

	Seq	SLT	SAT	A/SAT	LSA	PDS	A/MAT
no additional communication	✔	✔	✔	✔	✖	✔	✔
no circular deadlocks	✖	✔	✔	✔	✔	✔	✔
no mutual deadlocks	✖	✖	✔	✔	✔	✔	✔
good CPU utilisation	✖	✖	✔	✔	✔	✔	✔
condition variables	✖	✖	✖	✔	✖	✖	✔
parallelism	✖	✖	✖	✖	✔	✔	✔
parallel lock granting					✔	✔	✖
arbitrary thread number					✔	✖	✔

9.1 Summary (2)

	Seq	SLT	SAT	A/SAT	A/LSA	A/PDS	A/MAT
no additional communication	✔	✔	✔	✔	✖	✔	✔
no circular deadlocks	✖	✔	✔	✔	✔	✔	✔
no mutual deadlocks	✖	✖	✔	✔	✔	✔	✔
good CPU utilisation	✖	✖	✔	✔	✔	✔	✔
condition variables	✖	✖	✖	✔	✔	✔	✔
parallelism	✖	✖	✖	✖	✔	✔	✔
parallel lock granting					✔	✔	✖
arbitrary thread number					✔	✖	✔

10 Conclusion

- Deterministic thread scheduling for active-replicated services/objects
  - introduction of available algorithms
- XtreemOS Virtual Nodes
  - contains implementations of all available algorithms
  - including extensions for Java-like coordination (monitor with at least one condition variable)
    - ADETS/PDS, ADETS/LSA
- Further Work
  - a new better algorithm is in the queue
  - adaptive and deterministic switch between different algorithms
    - optimisation of certain parameters: response time, throughput ...

10 Conclusion (2)

- Further application domains of deterministic schedulers
  - passive replication
    - failover sometimes based on outdated checkpoints
    - replay of missed invocation requests needs to be deterministic
  - debugging of non-interactive applications
    - e.g. long-running HPC applications

Questions?