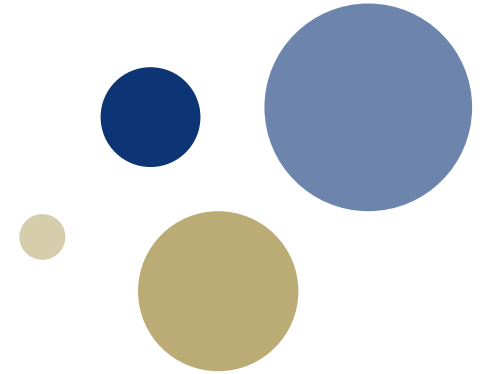




NTNU – Trondheim
Norwegian University of
Science and Technology



TTK4145 – Real-time Programming

Lecture 11 –

Atomic Actions and Asynchronous Transfer of Control

Example exam questions



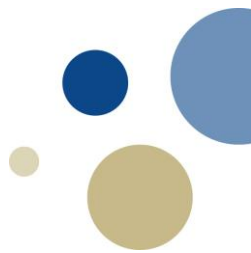
- From 2014:
 1. What is an Atomic Action? Which problem(s) is Atomic Actions meant to solve?
 2. Describe one way of implementing an Atomic Action.
 3. Operations for locking resources are always assumed to be atomic. Why is this so important?
- From 2010:
 1. What is the benefits by structuring the functionality of your program into Atomic Actions?
 2. Any drawbacks?

Learning goals: Atomic Actions



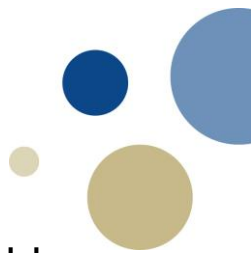
- A thorough understanding of the problems Atomic Actions are meant to solve and how these motivates the different aspects of Atomic Actions.
- Ability to use and implement Atomic Actions, including the mechanisms providing the start, side and end boundaries.
- Understanding the motivation for using Asynchronous Notification in Atomic Actions.
- A coarse knowledge of how the mechanisms for Asynchronous Notification in C/Posix, ADA and Java works.

Repetition (Transactions)



- Motivated by the need to implement fault tolerance in systems with multiple participants
- A transactions either completes successfully (commits) or if it fails (abort) *all* of its effects are undone (rolled back)
- Backwards error recovery is used to roll back when one or more participants fail during the transaction
- Avoids the domino effect by ensuring all participants store recovery points at the start of the transaction

Context



- In Real-Time systems, completely aborting an action might not be acceptable
 - Deadlines need to be met to ensure safe operation and/or the safety of people and equipment involved
 - Backwards Error Recovery might not be the best tool for the job
- Ideally the transactions would proceed even when errors occur, instead of completely aborting
 - Such actions would no longer be transactions according to formal definition (atomicity in ACID)
 - However, the framework for error confinement/assessment is still there
- We would like to create actions based on transactions, but utilizing Forward Error Recovery instead
- In order to do this, we must answer the following questions
 - What was the system doing when the error occurred?
 - What state is the system in after the error?
 - How do we inform all participants that an error occurred?
 - How do ensure that all participants engage in the error handling?
 - How do we recover from the error safely?



Atomic actions

The more (less) successful sibling of transactions

Definition of an atomic action



An action is atomic if the tasks performing it ...

1. ... are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the action
2. ... do not communicate with other processes while the action is being performed
3. ... can detect no state change except those performed by themselves and if they do not reveal their state changes until the action is complete
4. ... can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent

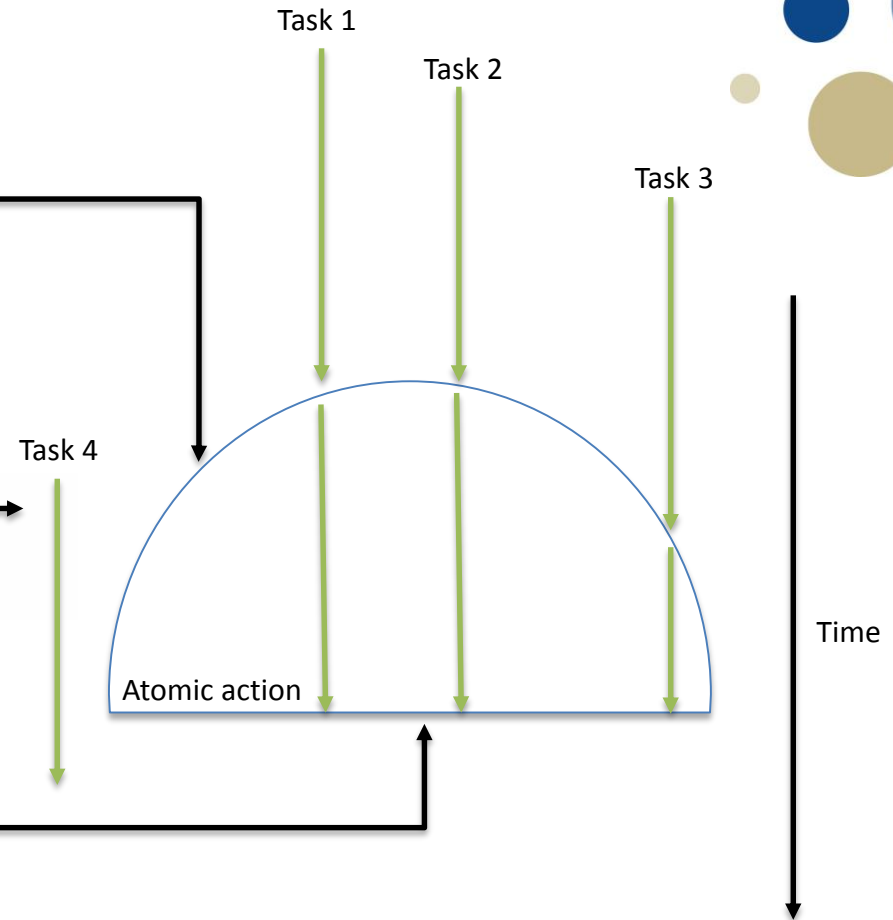
Requirements for atomic actions



- Well-defined boundaries
 - Should have a start, an end and a side boundary
- Indivisibility (isolation)
 - No exchange of information between the tasks inside the action and tasks outside
 - Tasks only leave the action when everyone is able and ready to leave
- Nesting
 - Can be nested as long as they don't overlap
 - Only strict nesting is allowed
- Concurrency
 - The outcome of running a collection of actions concurrently should be the same as serializing their executions

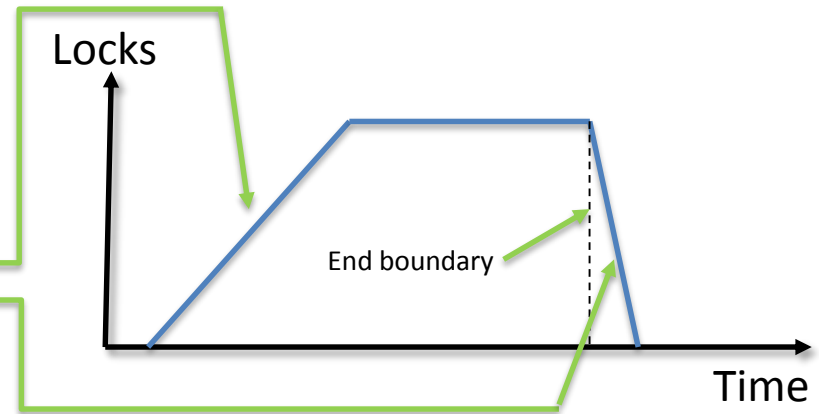
Boundaries

- Start boundary
 - Well-defined starting point for the action
 - Grants membership in the action when threads arrive
- Side boundary
 - No communication with non-participants on the outside of the action
 - Ensures error confinement
- End boundary
 - Well-defined end point for the action
 - Coordinates all the tasks finishing points in time
 - The point where the (trans)action manager decides to commit or abort



Implementation of boundaries

- Start boundary
 - Differentiate between two types of membership
 - Dynamic: the number of participants are unknown, new actions started by `startwork()`, new participants `join()` the action
 - Static: Hardcoded list of participants
- Side boundary
 - Lock the resources that are utilized in the action
 - Not just semaphores, more complex, the entire action is locked
 - Two-phase locking is used
 - Ensures no concurrent accessing of inconsistent resources
 - Growing phase, acquiring locks
 - Shrinking phase, releasing locks
- End boundary
 - Counting votes from all participants
 - Two-phase commit
 - Barrier that prevents participants from leaving
 - All participants need to wait for other participants to reach barrier
 - Barrier suspends participants until everyone has reached it



How do we implement an atomic action?



A multitude of design patterns available

1. Threads interacting with each other directly
 - Interaction can mean both communication and synchronization
 - In Go this is communication through channels
 - In Ada tasks are the building blocks for this
2. Servers that handle requests
 - A subset of the above
 - The implementation found in the upcoming Ada/Go examples
3. Threads that are created and terminated within the atomic action
 - Terminating threads often difficult and dangerous, why?
 - There can be unresolved dependencies leading to deadlock
 - The system state at termination might be unknown or inconsistent
 - Made easier with introduction of Lock Manager from last week
4. A number of procedures that are called by participating threads
 - A collection of actions we wish to have performed
 - Can be collected into a module, B&W uses classes for this
 - This will be our approach going forward

A resource manager in Go



```
func resourceManager(clients chan message, transactionManager chan message){
    myTmChannel = make(message);
    for {
        select {
            case message <- clients :
                switch message.request
                case "dowork1":
                    transactionManager <- myId, "joinTransaction", tid, myTmChannel;
                    // store recoverypoint (if not maintaining log)
                    // Do work of type 1 using requestData, keep track of errorstatus
                    replyChannel <- result;
                case "dowork2":
                    // ...

            case message <- myTmChannel
                switch message.request
                case "prepareToCommit":
                    transactionManager <- transactions[tid].errorStatus;
                case "commit":
                    // Delete recoverypoint, unlock any locks, delete transaction item from mapping
                case "abort"
                    // reset to recoverypoint, unlock any locks. delete transaction item from mapping
        }
    }
}
```

A transaction manager in Go

```
func transactionManager(clients chan message, resourceManagers chan message){
    for {
        select {
            case message <- clients :
                switch message.request
                    case "startWork":
                        // generate a transaction id, and reply
                    case "endWork":
                        // Send PrepareToCommit to all clients

            case message <- resourceManagers
                switch message.request
                    case "joinTransaction":
                        // Add the clientChannel to the list of channels to ask for prepareToCommit
                    case "ok":
                        // increment nOfVotes
                        // if nOfVotes == nOfParticipants, send commit/abort, depending on
                        // failFlag, to all.
                    case "fail"
                        // increment nOfVotes, set failFlag
                        // if nOfVotes == nOfParticipants, send abort to all.
                }
        }
    }
}
```

A resource manager in Ada



```
task body mytask is
  ...
begin
  loop
    select
      accept DoWork do
        ...
      exception
        // set ErrorFlag
      end DoWork;
    or
      accept PrepareToCommit(Vote: out Boolean) do
        // Vote = ErrorFlag
      end PrepareToCommit;
    or
      accept Commit do
        ...
      end Commit;
    or
      accept MyAbort do
        ...
      end MyAbort;
    end select;
  end loop;
end mytask;
```

Ada protected objects




- One of several interesting concurrency features of Ada
- As good as a synchronization primitive can be
- Protected objects are modules where the interface consists of
 - Functions
 - Procedure
 - Entries which can be protected by guards
- Guards
 - A test integrated in the language that opens or closes for execution
 - Covers both if statements and suspension of tasks
 - In our context, guards means blocking until guard changes its value
- As with tasks in Ada, protected objects are operated on under automatic mutual exclusion

Atomic action in Ada



```
package Action_X is  
    procedure Code_For_First_Task(--params); -- Role1  
    procedure Code_For_Second_Task(--params);-- Role2  
    procedure Code_For_Third_Task(--params); -- Role3  
end Action_X;
```

```
package body Action_X is  
  
    protected Action_Controller is  
        entry First;  
        entry Second;  
        entry Third;  
        entry Finished;  
    private  
        First_Here : Boolean := False;  
        Second_Here : Boolean := False;  
        Third_Here : Boolean := False;  
        Release : Boolean := False;  
end Action_Controller;
```

```
protected body Action_Controller is
  entry First when not First_Here is
  begin
    First_Here := True;
  end First;
  -- similarly for second, third

  entry Finished when Release or Finished'Count = 3 is
  begin
    if Finished'count = 0 then
      Release := False;
      First_Here := False;
      Second_Here := False;
      Third_Here := False;
    else
      Release := True;
    end if;
  end Finished;
end Action_Controller;
```

```
procedure Code_For_First_Task(--params) is  
begin
```

```
    Action_Controller.First;  
    -- acquire resources;  
    -- the action itself,  
    -- communication via resources  
    Action_Controller.Finished;  
    -- release resources
```

```
end Code_For_First_Task;
```

```
-- similar for second and third task
```

```
begin
```

```
-- any initialization of local resources
```

```
end Action_X;
```

Java monitors



- Java is object-oriented like C++
- Monitors are a mechanism to control concurrent access to an object
- They are implicitly declared with the keyword `synchronized` in front of any method
 - Any such methods will be called with mutual exclusion
 - Implemented as one mutex owned by all `synchronized` objects in java
 - Can lead to loss of concurrency
- `Wait()`
 - Can be called from inside `synchronized` method
 - Suspends the current accessing thread
 - Other threads can then access the same/other `synchronized` methods
- `Notify()/notifyAll()`
 - Awakens one thread from `Wait`, allowing it to resume
 - `notifyAll()` awakens all waiting threads, still must wait for lock in order

Atomic action in Java



```
public interface ThreeWayAtomicAction
{
    public void role1();
    public void role2();
    public void role3();
}
```

```
public class AtomicActionControl implements ThreeWayAtomicAction
{
    protected Controller Control;
    public AtomicActionControl() // constructor
    {
        Control = new Controller();
    }

    class Controller
    {
        protected boolean firstHere, secondHere, thirdHere;
        protected int allDone;
        protected int toExit;
        protected int numberOfParticipants;


        Controller()
        {
            firstHere = false;
            secondHere = false;
            thirdHere = false;
            allDone = 0;
            numberOfParticipants = 3;
            toExit = numberOfParticipants;
        }
    }
}
```



```
synchronized void first() throws InterruptedException
{
    while(firstHere) wait();
    firstHere = true;
}
```

```
synchronized void second() throws InterruptedException
{
    while(secondHere) wait();
    secondHere = true;
}
```

```
synchronized void third() throws InterruptedException
{
    while(thirdHere) wait();
    thirdHere = true;
}
```



```
synchronized void finished() throws InterruptedException
{
    allDone++;
    if(allDone == numberOfParticipants) {
        notifyAll();
    } else while(allDone != numberOfParticipants) {
        wait();
    }
    toExit--;
    if(toExit == 0) {
        firstHere = false;
        secondHere = false;
        thirdHere = false;
        allDone = 0;
        toExit = numberOfParticipants;
        notifyAll();
        // release processes waiting for the next action
    }
}
}
```

```
public void role1()
{
    boolean done = false;
    while(!done) {
        try {
            Control.first();
            done = true;
        }
        catch (InterruptedException e) { // ignore }
    }

    // .... perform action

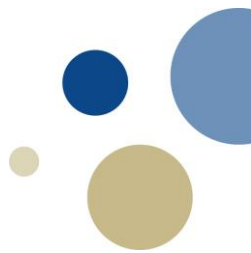
    done = false;
    while(!done) {
        try {
            Control.finished();
            done = true;
        } catch (InterruptedException e) { // ignore }
    }
};
```





```
public void role2()  
{  
    // similar to role1  
}  
  
public void role3()  
{  
    // similar to role1  
}  
}
```

How to distribute error information to all participants?



- Theme of the day is Forward Error Recovery
 - Need as much information as possible about what went wrong and when
 - Backwards Error Recovery you simply rollback, Forward Error Recovery need some information about the state of the system to work it after the error
- Additionally, we need a mechanism to notify other participants that they need to do error recovery. There are several possibilities:
 1. Extending the commit protocol
 - Instead of returning Boolean value at barrier, return error code
 - The action manager then contacts the other participants
 - For Real-Time purposes, this might be too slow for the same reasons Backwards Error Recovery is too slow
 - This does not handle all error modes
 2. Distribute error information when error occurs
 - Participants can routinely poll a global error variable, how are the other threads doing?
 - When an error is detected, react immediately
 - Conceptually better, but still might be too slow
 3. Asynchronous Transfer of Control
 - The other threads are interrupted in what they do, and error handling starts immediately
 - The solution we will work with



Asynchronous Transfer of Control

It's always better to ask for forgiveness than permission

Asynchronous notification



- Asynchronous notification is low-level thread interaction
 - There is no polling of thread status
 - Threads directly interfere with other threads execution
- Asynchronous notification comprises two different paradigms
- Asynchronous event handling (resumption)
 - The thread is suspended until the error is handled, then execution continues from where the error occurred
 - Hardware interrupts in embedded systems work this way
 - Not that useful, we would like to terminate the threads that are misbehaving to simplify error handling
- Asynchronous transfer of control (termination)
 - The thread is terminated and execution continues somewhere else
 - Exceptions using `try{}` and `catch{}` blocks are a good example
 - Powerful tool, but can lead to loss of maintainability, avoid if you can

C/ Real-Time POSIX Asynchronous Notification



- Asynchronous event handling
 - Modeled after hardware interrupts
 - Threads can generate signals and send to correct threads to notify about events
 - Signals can be handled, blocked or ignored by receiving thread
- Asynchronous transfer of control
 - No built-in support for this in the language
 - Can still dynamically create and cancel threads
 - Specifically, threads can be terminated asynchronously, but the thread controls the access to this
 - Asynchronous transfer of control can then be implemented using `Setjmp/longjmp`, along with termination
 - However, not obvious how you ensure atomicity in this case

setjmp / longjmp



```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");           // prints
    longjmp(buf,1);              // jumps back to where setjmp was called - making setjmp now
}

void first(void) {
    second();
    printf("first\n");           // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // when executed, setjmp returns 0
    } else {                    // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints
    }

    return 0;
}
```

Java Asynchronous Notification



- Asynchronous event handling
 - Can be implemented with `POSIXSignalHandler` class, identical to the standard POSIX signals
- Asynchronous transfer of control
 - Standard Java supports

```
public void interrupt() throws SecurityException  
public Boolean isInterrupted()
```

- Requires that the interruptible thread polls with `isInterrupted()`, not good for Real-Time applications
- Real-Time Java allows for exceptions being thrown to the interrupted thread instead of polling using `AsynchronouslyInterruptedException`
 - Exceptions are difficult to deal with in its basic form, gets way worse with multithreading
 - Like `setjmp/longjmp`, this is a very inelegant solution due to its implementation
 - The book has a long example using this, but it's not part of our curriculum

Ada Asynchronous Notification



- Asynchronous event handling
 - There is none
- Asynchronous Transfer of Control
 - Made easy through Ada's `Select then abort` statement

```
select
  delay 1.0 -- or any event.
then abort
  -- Do the real work
  -- Will be interrupted if it times out or
    the event happens.
end select;
```

- The normal execution of code happens inside the `abort` block
- This code execution will be aborted if `select` statement receives an event

Forward Error Recovery with Ada using Asynchronous Transfer of Control

```
package Action is  
    procedure T1(Params: Param); -- called by task 1  
    procedure T2(Params: Param); -- called by task 2  
    procedure T3(Params: Param); -- called by task 3  
  
    Atomic_Action_Failure : exception;  
end Action;
```

with Ada.Exceptions; **use** Ada.Exceptions;

package body Action **is**

type Vote_T **is** (Commit, Aborted);

protected Controller **is**

entry Wait_Abort(E: out Exception_Id);

entry Done;

procedure Cleanup (Vote: Vote_T);

procedure Signal_Abort(E: Exception_Id);

entry Wait_Cleanup(Result : out Vote_t);

private

 Killed : Boolean := False;

 Releasing_Cleanup : Boolean := False;

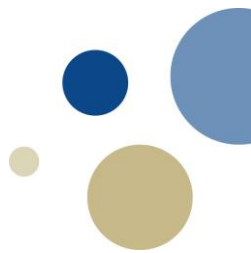
 Releasing_Done : Boolean := False;


 Reason : Exception_Id := Null_Id;

 Final_Result : Vote_t := Commit;

 Informed : Integer := 0;

end Controller;





```
protected body Controller is
  entry Wait_Abort(E: out Exception_id) when Killed is
  begin
    E := Reason; Informed := Informed + 1;
    if Informed = 3 then
      Killed := False;
      Informed := 0;
    end if;
  end Wait_Abort;

  entry Done when Done'Count = 3 or Releasing_Done is
  begin
    if Done'Count > 0 then Releasing_Done := True;
    else Releasing_Done := False; end if;
  end Done;

  procedure Cleanup(Vote: Vote_T) is
  begin
    if Vote = Aborted then
      Final_Result := Aborted;
    end if;
  end Cleanup;
```



```
procedure Signal_Abort(E: Exception_id) is
begin
    Killed := True;
    Reason := E;
end Signal_Abort;

entry Wait_Cleanup (Result : out Vote_T) when
    Wait_Cleanup'Count = 3 or Releasing_Cleanup is
begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
        Releasing_Cleanup := True;
    else
        Releasing_Cleanup := False;
        Final_Result := Commit;
    end if;
end Wait_Cleanup;

end Controller;
```



```
procedure T1(Params: Param) is
    X : Exception_Id;
    Decision : Vote_T;
begin
    select
        Controller.Wait_Abort(X);
        Raise_Exception(X);
    then abort
        begin
            -- code to implement atomic action
            Controller.Done;
            exception when E: others =>
                Controller.Signal_Abort(Exception_Identity(E));
        end;
    end select;

    exception when E: others =>
        -- code to handle exception
        if Handled_Ok then
            Controller.Cleanup(Commit);
        else
            Controller.Cleanup(Aborted);
        end if;
        Controller.Wait_Cleanup(Decision);
        if Decision = Aborted then
            raise Atomic_Action_Failure;
        end if;
    end T1;

    -- similarly for T2 and T3
end Action;
```