Australian
National
University

EXPLORE
ATOMIC ACTIONS

Lab week 6 for Real-Time and Embedded Systems 2019

Uwe R. Zimmer

This exercise will add a new dimension to your real-time programming repertoire: **Asynchronous transfer of control**. This feature is frequently required in real-time system as it enables to "interrupt" activities based on time or events. As an asynchronous interruption is a complex and potentially highly dangerous operation, we need to learn how to handle this feature with grace and control.

> *Side remark:* The term "**interrupt**" commonly implies asynchronism, thus I will mostly omit the "asynchronous". Nevertheless keep in mind that there are multiple levels of asynchronism in computer systems. There is always a granularity at which flows of control in a clocked system can change. The finest level on a single CPU system would be given by the CPU-clock frequency itself, but asynchronous transfer of control between processes will usually require a few more cycles of code execution inside the runtime-environment or real-time operating system before an asynchronous event can take effect.

The common approach among real-time programming languages is to allow a process to mark a code region in which the process is prepared to be interrupted by a pre-defined set of events (which includes timing events). A common, simple form of this would be for instance to add a time limit to a request. The allowed event here would be the time-out, and the code section would be the task being suspended in a waiting queue. This is still comparatively easy to handle, as this "code section" can rather easily be interrupted.

The more general form of **asynchronous transfer of control** will include the execution and potential abortion of more general code sections. This is provided for instance in Ada by this statement:

```
select
   <entry-call | delay until <time>>
   [ … statements … ]
then abort
   … statements …
end select;
```

Depending on the availability of the entry which is called, or the passing of the defined deadline (either of those events will be denoted as a "**trigger**"), the executed statements will vary.

a. If the **trigger is going through** and can be completed: the optional *statements following the trigger are executed* and the select statement is completed (the abortable part is never started).

b. If the **trigger is blocked** or requeued to a blocked entry: the statements in the *abortable part are executed* and one the following two cases will apply:

   i. If the **abortable part completes before the trigger is completed**, an attempt is made to revoke the triggering statement. The select statement is completed after the cancelled or completed triggering statement.

   ii. If the **trigger is completed before the abortable part is completed**, the abortable part is stopped, the optional *statements following the trigger are executed* and the select statement is completed.

Thus one of the statement sequence could be executed completely while the other one will never be started, yet the challenging case is when the abortable sequence is started, yet then aborted on trigger, and the process continues its execution after the trigger statements.

> *Side remark:* There is also some control over the granularity of the interruption, as some statements inside the abortable part can themselves be non-abortable, and thus will need to be completed before the transfer of control can happen.

Atomic Actions

While asynchronous transfer of control has many applications in real-time systems, one form which this assignment is focusing on is **atomic actions**. A simple, informal way to define an atomic action is to state that:

Definition 1: An atomic action is performed in full or not at all.

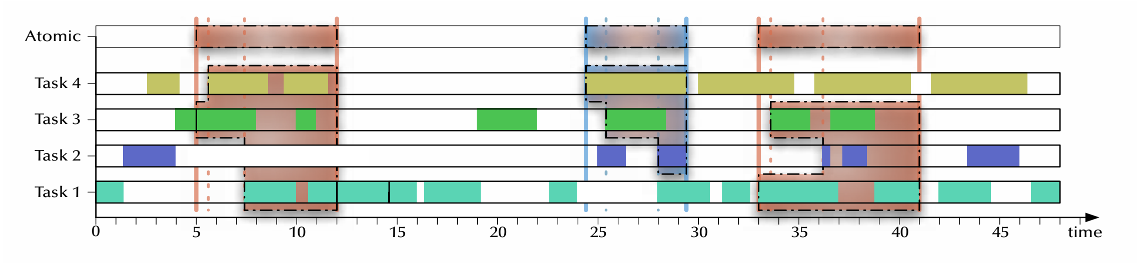
While this is easy to define, it is somewhat harder to implement. Two aspects will give us work: Firstly, not all partially executed statement sequences can be undone. Secondly, an atomic action can involve any number of concurrent operations, which implies that those concurrent operations need to be able to potentially abort each other, if the atomic operation is declared failed and everything need to be undone.

Another way of looking at an atomic action would be:

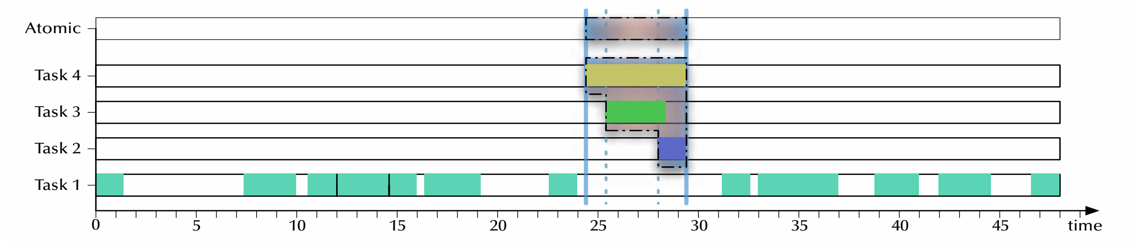
Definition 2: An atomic action is indivisible and instantaneous.

This definition focuses on the aspect of observability of an atomic action. Neither progression inside the atomic action can be observed from the outside, nor can the atomic action observe any events outside of its realm. Events which do not provide an observable time-line are logically considered instantaneous (similar to the zero-delay assumption in synchronous programming languages).

As you can see in the Figure below, atomic actions start with the first task joining in, while all tasks are held inside the atomic action until all the work is done, and then released at once.



Alternatively if anything goes wrong, all tasks have to be informed about the failure and all tasks will attempt to either roll back (full undo) or achieve another consistent state. In the above Figure, the tasks do already exist, join into an atomic action, and still exist (going their individual ways again) after the end of an atomic action. In the code which you see on course web-site, the tasks inside the atomic action and created especially for this purpose and destroyed after finalization of the atomic action. Thus this would look like in the Figure below.



Exercise 1: Flight surface controller

As an example given on the course web-site, a system where flight control surfaces need to be moved in synchrony as well as under their own specific timing constraints. To achieve that all individual movement procedures are defined as part of the same atomic action and parametrized with individual timing constraints. If anything goes wrong (which you should of course provoke as part of your experimentation), all parts of the atomic action need to be informed about it. Either by means of interrupting what they are currently doing (if they have not completed their operations yet) or calling them back from their “I completed but just wait here for everybody to complete, so that I can get out of the atomic action”-state. Either way all need to be informed that all they did was in vain and they need to go to fail-safe positions instead.

Carefully experiment with the code given and try to reconstruct the control flow paths in each case as precisely as possible. You might want to draw a few diagrams and become perfectly clear about your findings. Explain your findings to your tutor.

As you probably already noticed, creating and destroying tasks is not how it’s done in the real-time world (for obvious reasons) and so you should find a better way of doing it which allows for existing tasks to join an atomic action instead of creating dedicated tasks. While the concept itself is not hard, the practical implementation can still be challenging. Yet it will give you the necessary basic experience to handle time constraints in concurrent systems.

I’d expect a lot of questions and we are happy to help you into it. Just do the basic analysis of the code first (as far as you get), and we will then assist you to clarify the finer details and discuss your idea how to transform the example into a version which allows for existing tasks to join an atomic action.

Submit your improved `Atomic_Actions.zip` file on the *SubmissionApp* under “Lab 6 Atomic Actions“ for code inspection by your colleagues and by us.