Time Handling in Programming Language

Rajesh Gupta
University of California, San Diego
System Characteristics

• Complexity in function (and in size)
• Concurrent control of separate components
  – devices operate in parallel
• Facilities to interact with special purpose hardware
  – need to program devices in a reliable and abstract way
• High reliability and safety
• Guaranteed response times
  – predictability is important; sometimes more so than efficiency
  – Here we address programming language facilities for handling time. This is followed later by scheduling.
Outline

• The notion of time in programming
• Time facilities
  – access to a clock
  – delays and timeouts
  – timing requirements
  – temporal scopes
• Language support for time
Real Time Facilities

• Generally RT facilities in programming are built on top of concurrency mechanisms
• There are three parts to handling of time in a programming language
  1. Interfacing with “time”:
     – E.g., access clock so that passage of time can be measured
     – Delaying processes until some future time
     – Programming timeouts for recognizing non-occurrence
  2. Representing timing requirements
     – Execution deadlines, rates
  3. Satisfying timing requirements
The Notion of Time

• Primary purpose of time is to ensure ordering of events
  – Temporal versus Causal Ordering
• Topology of time
  – Equate passage of time with a “real” line
  – Time is: Transitive, Linear, Irreflexive, Dense
    • Transitivity: \( \forall x, y, z : (x < y \text{ and } y < z) \Rightarrow x < z \)
    • Linearity: \( \forall x, y : x < y \text{ or } y < x \text{ or } x = y \)
    • Irreflexivity: \( \forall x : \text{not } (x < x) \)
    • Density: \( \forall x, y : x < y \Rightarrow \exists z : (x < z < y) \)
1 Access to a clock

• One of two ways:
  – Direct access to environmental time
  – An internal approximation
• Direct access to environmental time
  – Environment supplies a regular interrupt that is clocked internally
  – Or use a global time: UTC, IAT signals
    • (GPS uses UTC service)
  – From the pov of a programming language, time access can be by
    • A clock primitive in the language, or
    • Via a device driver for the internal clock, external clock or radio receiver
  – Let us consider some examples…
Time Standards

• **Universal Time (UT0)**
  – Defined in 1884. Mean solar time at Greenwich meridian
    • Second: 1/86400 of a mean solar day
  – **UT1**: Correction of UT0 for polar motion
  – **UT2**: Correction of UT1 for variation in earth speed

• **Atomic**
  – Second: 9 192 631 770 period of radiation corresponding to the transition between two levels of the ground state of the caesium-133 atom
    • Accurate to 1 in 10^13 or 1 clock error in 300,000 years
  – **International Atomic Time (IAT)**

• **Coordinated Universal Time (UTC)**
  – IAT clock synchronized to UT2 by the additional of occasional leap ticks
  – Maximum difference between UT2 and UTC is kept < 0.5 seconds.
Example: ANSI C

- ANSI C provides a standard library for interfacing with ‘calendar’ time
  - Basic time type “time_t”
  - Several routines for manipulating objects of type time_t
  - (Interface to ANSI C dates and time)

```c
typedef ... time_t;
struct tm {
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday;
    int tm_isdst;
};
Double difftime (time_t time1, time_t time2);
Time_t mktime(struct tm *timeptr); // compose a time value
Time_t time(time_t *timer); // returns the current time
```

- “tv_sec” = number of seconds since 1/1/1970
  - In case of CLOCK_REALTIME, tv_nsec long integer <10^9
POSIX

- Allows many clocks to be supported by an implementation
- Each clock has its own identifier (of type `clockid_t`)
- IEEE standard requires: at least one clock be supported, `CLOCK_REALTIME`
- Minimum resolution of `CLOCK_REALTIME` is 50 Hz or 20 ms
- C interface to POSIX clocks

```c
#define CLOCK_REALTIME
typedef struct timespec {
    time_t tv_sec;
    long tv_nsec;
} timespec;

typedef ... clockid_t;
int clock_settime (clockid_t clock_id, const struct timespec *tp);
int clock_gettime (clockid_t clock_id, struct timespec *tp);
int clock_getres (clockid_t clock_id, struct timespec *res);
int nanosleep (const struct timespec *rqtp, struct timespec *rmtl);
```
Example: Times in OCCAM2

- Any occam2 process can obtain value of the “local” clock by reading from a TIMER
  - Each process must use a distinct TIMER
- Reading from a TIMER is similar to channel read
  - Except that it can not block
  - Single integer for clock value
  - Relative value, need >1 readings

<table>
<thead>
<tr>
<th>granularity</th>
<th>range (approximately)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 microsecond</td>
<td>71.6 minutes</td>
</tr>
<tr>
<td>100 microsecond</td>
<td>119 hours</td>
</tr>
<tr>
<td>1 millisecond</td>
<td>50 days</td>
</tr>
<tr>
<td>1 second</td>
<td>136 years</td>
</tr>
</tbody>
</table>

Timer Granularities

```plaintext
TIMER clock:
INT Time:
SEQ
  clock ? Time    -- read time

TIMER clock:
INT old, new, interval:
SEQ
  clock ? old
  -- other computations
  clock ? new
  interval := new MINUS old
```
Example: Clock packages in ADA

• Library package called “Calendar” and optional real-time facility
• This package implements an ADT for “time”
  – Provides function “clock” for reading time
  – Type “duration” is a predefined fixed point real
    • Both its accuracy and its range are implementation dependent, though...
    • Covers at least the number of seconds in a day with a granularity of less than 20 ms
• Library package “Real_Time”
  – Similar to “Calendar” but finer granularity
    • “Tick” less than 1 ms; the range of “Time” (from the epoch that represents the program’s startup) at least 50 years.
package Ada.Calendar is

    type Time is private;

    subtype Year_Number is Integer
        range 1901 .. 2099;
    subtype Month_Number is Integer
        range 1 .. 12;
    subtype Day_Number is Integer
        range 1 .. 31;
    subtype Day_Duration is Duration
        range 0.0 .. 86400.0;

    function Clock return Time;

    function Year(Date:Time) return Year_Number;
    function Month(Date:Time) return Month_Number;
    function Day(Date:Time) return Day_Number;
    function Seconds(Date:Time) return Day_Duration;

    procedure Split(Date:in Time;
        Year:out Year_Number;
        Month:out Month_Number;
        Day:out Day_Number;
        Seconds:out Day_Duration);

    function Time_Of(Year:Year_Number
        Month:Month_Number;
        Day:Day_Number; Seconds:Day_Duration := 0.0)
        return Time;

    function "+"(Left:Time; Right:Duration)
        return Time;

    function "-"(Left:Time; Right:Duration)
        return Time;

    function "="(Left:Time; Right:Time)
        return Boolean;

    function "<"(Left,Right:Time) return Boolean;
    function ">">"(Left,Right:Time) return Boolean;
    function ">="(Left,Right:Time) return Boolean;

    Time_Error:exception;
    -- TIME_ERROR is raised by TIME_OF,split,"+",
    -- and "-"

private
    -- implementation dependent
end Ada.Calendar;

declare
    Old_Time, New_Time : Time;
    Interval : Duration;
begin
    Old_Time := Clock;
    -- other computations
    New_Time := Clock;
    Interval := New_Time - Old_Time;
end;
Real_Time is defined to be monotonic

Not so with Calendar (leap years etc).
2 Delays and Timeouts

• Processes must be able to delay their execution for a
  – Relative time
  – Or until some time in the future

• Relative delays
  – Obvious thing is to “busy wait”
    
    ```
    Start := Clock; -- from calendar
    loop
      exit when (Clock - Start) > 10.0;
    end loop;
    ```

  – To avoid this, language must provide a delay primitive
    • ADA: delay 10.0;
    • POSIX: sleep system call, or nanosleep using
      CLOCK_REALTIME
Delay guarantees

- Only that the process is made runnable after the period has expired
  - The actual delay may vary (due to other processes)
- Also, the granularity of delay and the granularity of clock are not necessarily the same
  - Plus there interrupt inhibitions that affect clock time
Programming Absolute Delays

• If a delay to an absolute time is needed then
  – Either the programmer calculate the period to delay, or
  – An additional primitive is needed.

• Example:

  ```pascal
  Start := Clock;
  First_Action;
  delay 10.0 - (Clock - Start);
  Second_Action;
  ```

• But this may not achieve the desired result
  – Delay will have to be an **uninterruptible** action

• ADA remedy: “delay until”

  ```pascal
  Start := Clock;
  First_Action;
  delay until Start + 10.0;
  Second_Action;
  ```

• Accurate only in its lower bound.
Drift

• Drift is the time over-run in programming delays
  – Local drift
  – Cumulative drift
• While local drift may not be eliminated, it is possible to eliminate the cumulative drift
• Example: if two Actions are 7.5 sec apart, the next ones will be delayed only 6.5 sec

```plaintext
declare
  Next : Time;
  Interval : constant Duration := 7.0;
begin
  Next := Clock + Interval;
  loop
    Action;
    delay until Next;
    Next := Next + Interval;
  end loop;
end;
```
OCCAM2 and POSIX

- OCCAM2 supports only absolute delays
  - Use the keyword “AFTER”
    - \( G \) is the number of TIMER updates per second

```
SEQ
Clock ? Now
Clock ? AFTER now PLUS (10 * G)
```

- POSIX
  - Absolute delays possible using an absolute timer and waiting for the signal generated when it expires.
Programming Timeouts

• A timeout is a restriction on the time a process is prepared to wait for a communication
• Closely tied to communication and synchronization schemes available
  – By definition these require “waiting” (for communication, conditions)
• Two important aspects in the communication & synchronization schemes:
  1. Mutually exclusive access to a critical section (mutex)
  2. Condition synchronization
• Communication and synchronization (C&S) based on
  1. Shared Variables: 
    – objects that more than one process has access to
  2. Message Passing
    – Explicit exchange of data between two processes
Shared Variable C&S and Timeouts

• Mutually exclusive access to a critical section:
  – When a process attempts access to a shared critical region, it is blocked if another process is already active
    • But this blocking is bounded by the time taken to execute the code of the section and the number of other processes that also wish to execute the critical section
    • Blocking time can be analyzed and estimated (later in RT scheduling)
  ► Hence, no explicit timeout associated with attempted entry

• On the other hand, blocking with condition synchronization is not easily bounded
  – E.g., a process attempting to place data in a full buffer must wait..
  ► Therefore, it is important to allow a process to opt out of waiting for condition synchronization
S.V. Condition Synchronization Facilities

- The condition synchronization facilities that can be used for timeouts
  - Semaphores
    - a non-neg. var that can only be updated by 2 procedures (wait, signal)
  - Conditional critical regions
  - Condition variables in monitors and mutexes
  - Entries in protected objects
- (not covered in this class).
- Example: a process could suspend itself on the semaphore CALL with a timeout value to 10 seconds.
  
  ```
  wait CALL noLongerThan 10 : 200
  ```
- POSIX supports semaphores but does not provide an explicit timeout option
  - But “wait” is interruptible by a POSIX signal
- With POSIX condition variables, the it is possible to associate timeouts (through specific system calls)
- Protected objects in Ada can have associated timeouts (as with any task entry call)
  - Covered in message passing based C&S.
Message Passing C&S and Timeouts

- Inherently blocking, needs explicit programming of timeouts
- In Ada, you can do that by programming a task that delays itself for the timeout period
- If the controller accepts this second task before the normal call then a timeout has occurred.

```ada
task Controller is
  entry Call(T : Temperature);
private
  entry Timeout;
end Controller;
```

```ada
task body Controller is
  task Timer is
    entry Go(D : Duration);
    end Timer;
    -- other declarations
  task body Timer is
    Du : Duration;
  begin
    accept Go(D : Duration) do
      Timeout_Value := D;
    end Go;
    delay Timeout_Value;
    Controller.Timeout;
  end Timer;
begin
  loop
    Timer.Go(10.0);
    select
    accept Call(T : Temperature) do
      New_Temp := T;
      end Call;
    or
    accept Timeout;
    -- action for timeout
  end select;
  -- other actions
end loop;
end Controller;
```

Controller task in ADA that accepts a new temperature setting.

*Question: what happens after first timeout?*
Better Alternative using “Selective Waits”

• Use delay as an alternative to a call
  – Special form of a select statement
  – A “timed entry call”
    • A timeout specified with the call being accepted
• Both relative and absolute delays are possible
  – Delay or delay_until

```vhdl
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
        end Call;
    or
      delay 10.0;
      -- action for timeout
      end select;
      -- other actions
    end loop;
end Controller;
```
Timeout “Events”

• A timeout can be treated as an “event”
  – If asynchronous events are supported then these events can be used to change control flow in processes
    • SIGNALs in POSIX are mainly asynchronous events

• Example:
  – Asynchronous Transfer of Control (ATC) facility in Ada to catch ‘runaway code’
    ```
    select
      delay 0.1;
    then abort
      -- action
    end select;
    ```

• Timeouts can also be used to support “imprecise computations”
  – Compulsory and optional part
  – Associate a timeout with the optional part.
Example

Begin
Completion_time := …
-- compulsory part
Results.Write(…); -- call to procedure in external protected object

Select
  delay until Completion_time;
then abort
  loop
    --- improve result
    Results.Write(…);
  end loop;
End select;
End;
Specifying Timing Requirements

• Types of timing requirements
  – Interval bounds
  – Throughput bounds
• Captured as binary relations among operations
  – E.g., Timing Graph
• Timing requirements capture
  – As a part of function or independently
• Types of checks
  – Consistency
  – Feasibility, Satisfiability
  – Satisfiable by an implementation
  – Transformations to ensure satisfiability
• Example: Timing Graphs
Temporal Scopes

- A TS identifies a collection of statements with an associated timing constraint
- Components of a TS
  - Deadline
  - Min delay
  - Max delay
  - Max execution time
  - Max elapsed time
- TS can be
  - Periodic
    - Sample data, execute control loop and have explicit deadlines
  - Aperiodic
    - From asynchronous events outside of the embedded computer
    - Response time requirements

Maximum execution time = a + b + c
TS and Deadlines

- Periodic, aperiodic, sporadic
- Hard, Soft, Interactive, Firm
  - Hard if it fails on missed deadlines
  - Interactive strive to achieve adequate response time
  - A non-hard process with fixed rigid deadline (that is, missed deadline is useless (but not harmful)) is firm
Specifying Temporal Scopes

- Often use attributes and tags
  - Example: hardware modeling languages

- Time constraints as
  - max or min times for IDLE
  - Or end of TS by some deadline
    - Expressed either as absolute time
    - Or as execution time since the start of the TS
    - Or elapsed time since the start of the temporal scope
  - Example: a process sampling a data
    - Sampling done at the start of the TS, TS contains processing (buffering) for this data and deadline at the end of TS ensures that the process can loop around in time.

```plaintext
process periodic_P;
...
begin
loop
  IDLE
  start of temporal scope
  ...
  end of temporal scope
end;
end;
```
TS in Aperiodic Processes

• **Typically:**

```c
process aperiodic_P;
...
begin
  loop
    wait for interrupt
    start of temporal scope
    ...
    end of temporal scope
  end;
end;
```

• **When a data passes through a number of processes**
  – **Called a Real-time Transaction**

• **Deadlines on a RT transactions requires suitable partitioning of the time budget and scheduling of the processes**
  – In general, a very difficult problem.
Fault Tolerance and Timing Requirements

- In a hard RT system, a missed deadline must trigger error recovery mechanisms

- Source of error
  - Inaccurate timing estimates
  - Inadequate schedulability analysis
  - Error in schedulability analysis
  - System working outside its design parameters

- Some systems may be designed to anticipate such situations and incorporate mode changes
  - Environmental change may obviate the need for some computation
  - One way to handle is through recovery mechanisms
Forward Error Recovery

- If the RT is aware of deadlines then
  - It can recognize situations in which deadlines are missed
  - May be able to predict such situations
  - Provide mechanisms to inform the application that a “missed deadline” event has occurred
- Exceptions provide a natural means of such a dialogue
- The handlers must be able to make important distinctions in error sources
  - Overrun deadlines, overrun execution times, timeouts on communications, other non-temporal error conditions
- Error recovery may require extension of the original deadline and continue execution of the original block
  - Resumption is generally preferred over the termination model.
Event-based Reconfiguration

- With applications prone to mode changes, a process (client) may want to change the control flow in another (server) process while the computation is ongoing.
- This reconfiguration may require that the server
  - Immediately return the best result so far
  - Change to a quicker (may be less accurate) algorithm
  - Forget the current computation and be ready to undertake new computation “restart without reload”
- These responses may be captured by the type of the asynchronous event (exception) raised
  - To do this, the language must support asynchronous event handling
    - E.g., except, deactivate, kill
    - Ada: abort, ATC (deactivate)
Backward Error Recovery

- BER involves acceptance tests
- Therefore, one can include timing requirements as a part of the acceptance tests
  - The runtime system may fail on an acceptance test if the deadline has been overrun
  - Example: “dialogues”
    - Specify multiple alternative dialogues within a timeout interval
    - So a process tries these alternatives within a given deadline, and if it does not succeed the execution does not move forward
Example

task body deadline_example is
begin
loop
...
time := calculated.time_to.deadline;
slack := calculate.time_for.degraded.algorithm;
restore := state.restoration.time;
timeout.value := time - (slack + restore);
--
SELECT
dialog.1;
TIMEOUT timeout_value
  -- sequence of statements to recover
  -- from missed deadline
ELSE
  fail;
END SELECT;
end loop;
end deadline_example;

Not a real language.