# Real-Time Systems Programming

## Summer-Semester 2002
## Lecture 8
## 3 May 2002

## Expressing Time

# *Overview*

1) Incorporating Time in RT Languages
   - Clocks
   - Delays
   - Timeouts

# *Real-Time Facilities*

- Measuring the Temporal Behavior
  - Accessing clocks
- Controlling the Temporal Behavior
  - Delaying processes until some future time
  - Programming timeouts
  - Scheduling ($\Rightarrow$ *Operating System level*)
- Expressing Requirements on the Temporal Behavior
  - Specifying rates of execution
  - Specifying deadlines
- Analyzing Temporal Behavior
  - Worst-Case Execution Time (WCET) Analysis

# *How does a Computer Tell the Time?*

- Direct access to the environment's time frame
  - E.g., GPS receiver
- Use of internal hardware clock that gives an adequate approximation to the passage of time in the environment
  - Cristal oscillator
- Sometimes a combination of both
  - External time receiver used to initialize/synchronize internal clock
  - E.g., internet node that gets its time from NTP
  - VCR gets time signal from TV
  - Wristwatch with DCF (UTC) receiver

# *Terminology*

- A ***clock***: counter that tells the time
- A ***clock tick***: periodic event incrementing the clock
- A ***timer***: performs an action at some point in time
- ***Clock time***: reading of clock
- ***Calendar time***: time according to some standard (UTC)
- A ***clock device***:
  - Clock
  - Timer queue (with pending expiration times)
  - Interrupt handler (to service timer expiries)

# *Clock Resolutions*

- ***Resolution*** of a clock: granularity of the clock expressed in physical time
- Technology today permits ***hardware clocks*** with nanosecond resolution
- Applications typically only have access to ***software clocks***, with a resolution that is orders of magnitude coarser (100s of microseconds ... milliseconds)
- OS kernel maintains sw clocks for each supported clock device

# *Low-Resolution Software Clocks*

- OS kernel programs clock device to raise periodic ***clock interrupts***
- On each clock interrupt, the OS
  1. increments the sw clock
  2. checks the clock's timer queue
  3. executes scheduler (***tick scheduling***)
- Thread gets current time by calling a ***get-time function*** (e.g. POSIX **`clock_gettime`**) that reads out clock counter value
- $\Rightarrow$ Resolution seen by the thread is given by the frequency of the clock interrupt (typical: 10 msec)

# *Higher-Resolution Software Clocks*

- Most OSs alternatively provide finer clock resolution:
- Again the clock device is programmed to raise periodic interrupts (***time service interrupts***)
- On each clock interrupt, the OS
  1. increments the sw clock
  2. checks the clock's timer queue
  3. executes the scheduler only every *n*-th time
- Typical resolutions: 100s of µsec ... msec
- Clock interrupt frequency bounded by
  - permissable overhead of clock interrupt (incl. scheduling)
  - jitter of clock interrupt execution time

# *High-Resolution Hardware Clocks*

- Can make hw clock directly available to application by mapping the hw clock into the address space of the application (e.g. on Pentium processor)
- However, OS may not make hw clock readable for sake of portability (e.g. to non-Pentium machines)
- Some OSs still use hw clock to improve clock resolution:
  - OS reads high-res hw clock at each time-service interrupt
  - When servicing get-time function call, the OS
    - reads the hw clock again and calculates delta to last reading
    - returns the current sw clock reading + the hw clock delta

# *Access to Time for the Programmer*

- Can have time primitives in the language or in the API
  - *Ada*: packages `Calendar` and `Real_Time`
  - *ANSI C*: `<time.h>` defines `clock_t` (clock time), `time_t` (calendar time)
  - *POSIX*: supports multiple clocks/timers
  - *Real-Time Java*: class `HighResolutionTime`, extended by `AbsoluteTime`, `RelativeTime`, `RationalTime`
- Can access time via device drivers

# *Ada: the* `Calendar` *Package*

- The package **`Ada.Calendar`** provides a sw clock
  - Low-resolution
  - Non-monotonic (includes leap seconds, leap years)
- A value of the private type **`Time`** is a combination of the date and the time of day
- The time of day is given in seconds from midnight
- Seconds are described in terms of a subtype **`Day_Duration`**
- Which is, in turn, defined by means of **`Duration`**

# Ada.Calendar I

```
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..86_400.0;

  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);
```

# Ada.Calendar II

```
  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:Duration; Right:Time) return Time;
  function "-"(Left:Time; Right:Duration) return Time;
  function "-"(Left:Time; Right:Time) return Duration;
  function "<"(Left,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 may be raised by Time_Of,
  -- Split, Year, "+" and "-"
private
  implementation-dependent
end Ada.Calendar;
```

# *Ada: Duration Type*

- Fixed point type **`Duration`** is one of the predefined scalar types and has a range which, although implementation dependent, must be at least -86400.0 .. +86400.0
- The value 86400 is the number of seconds in a day – *excluding leap seconds*
- The accuracy of **`Duration`** is also implementation dependent but the smallest representable value **`Duration'Small`** must not be greater than 20 msec
- Ada Reference Manual recommends that it is no greater than 100 µsec

# Example with 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;
```

# *Ada: the* `Real_Time` *package*

- The package `Ada.Real_Time` provides another clock
  - High-resolution (1 msec or better)
  - Monotonic
- This has a similar form to `Calendar` but is intended to give a finer granularity
- The range of `Time` (from the program's start-up) must be at least fifty years

# Calendar Time in C: <time.h>

```
typedef ... time_t;
struct tm {
  int tm_sec;   /* seconds after the minute – [0, 61] */
                /* 61 allows for 2 leap seconds */
  int tm_min;   /* minutes after the hour – [0, 59] */
  int tm_hour;  /* hour since midnight – [0, 23] */
  int tm_mday;  /* day of the month – [1, 31] */
  int tm_year;  /* years since 1900 */
  int tm_wday;  /* days since Sunday – [0, 6] */
  int tm_isdst; /* flag for alternate daylight savings time */
};

double difftime(time_t time1, time_t time2);
    /* subtract two time values */
time_t mktime(struct tm *timeptr);
    /* compose a time value */
time_t time(time_t *timer);
    /* returns the current time, no. of secs since 1/1/1970 */
    /* if timer is not null, timer is assigned time as well */
```

# *Limitations of C/UNIX*

- Only one real-time clock (**`ITIMER_REAL`**), accessed with `clock()`
- Limited clock resolution
- No detection of timer overruns
- Timer expirations can only trigger **`SIGALARM`**

⇒ *Enter POSIX !*

# POSIX Real-Time Clocks

```
typedef ... clockid_t;
#define CLOCK_REALTIME ...; /* clockid_t type */

struct timespec {
  time_t tv_sec;   /* number of seconds */
  long   tv_nsec;  /* number of nanoseconds */
};

int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
int clock_getcpuclockid(pthread_t_t thread_id, clockid_t, *clock_id);

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
/* nanosleep returns -1 if the sleep is interrupted by a */
/* signal. In this case, rmtp has the remaining sleep time */
```

# *POSIX Real-Time Clocks*

- POSIX can support many clocks, identified by **`clockid_t`**
- IEEE standard requires that at lease one clock is supported (**`CLOCK_REALTIME`**)
- Standard requires minimum resolution to be 20 msec
- **`tv_sec`** returns no. of seconds since Jan 1, 1970

# *Clocks in Real-Time Java*

- Similar to those in Ada
- **`java.lang.System.currentTimeMillis`** returns the number of milliseconds since 1/1/1970 GMT and is used by used by **`java.util.Date`**
- Real-time Java adds real-time clocks with high resolution time types
- The class **`HighResolutionTime`** is base class for
  - class **`AbsoluteTime`**
  - class **`relativeTime`**
  - class **`RationalTime`**

# RT Java: class HighResolutionTime

```
public abstract class HighResolutionTime implements
       java.lang.Comparable
{
  public abstract AbsoluteTime absolute(Clock clock,
                          AbsoluteTime destination);
  ...
  public boolean equals(HighResolutionTime time);

  public final long getMilliseconds();
  public final int getNanoseconds();

  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
}
```

# RT Java: class `AbsoluteTime`

```
public class AbsoluteTime extends HighResolutionTime
{
  // various constructor methods including
  public AbsoluteTime(AbsoluteTime T);
  public AbsoluteTime(long millis, int nanos);
  public AbsoluteTime absolute(Clock clock,
                               AbsoluteTime dest);
  public AbsoluteTime add(long millis, int nanos);

  public final AbsoluteTime add(RelativeTime time);
  ...
  public final RelativeTime subtract(AbsoluteTime time);
  public final AbsoluteTime subtract(RelativeTime time);
}
```

# RT Java: class RelativeTime

```
public class RelativeTime extends HighResolutionTime
{
  // various constructor methods including
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);
  public AbsoluteTime absolute(Clock clock,
                               AbsoluteTime destination);

  public RelativeTime add(long millis, int nanos);
  public final RelativeTime add(RelativeTime time);
  public void addInterarrivalTo(AbsoluteTime destination);
  public final RelativeTime subtract(RelativeTime time);
  ...
}

public class RationalTime extends RelativeTime
{ . . .}
```

# *RT Java: class* `Clock`

```
public abstract class Clock
{
  public Clock();
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolution);
}
```

# RT Java: Measuring Time

```
{
  AbsoluteTime oldTime, newTime;
  RelativeTime interval;
  Clock clock = Clock.getRealtimeClock();

  oldTime = clock.getTime();
  // other computations
  newTime = clock.getTime();

  interval = newTime.subtract(oldTime);

}
```

# *Delaying a Process*

- In addition to clock access, processes must also be able to delay their execution
  - either for some period of time (***relative delay***)
  - or until some point in future time (***absolute delay***)
- Example for relative delay in Ada:

```
Start := Clock; -- from calendar
 loop
   exit when (Clock - Start) > 10.0;
 end loop;
```

# *Delay Primitives*

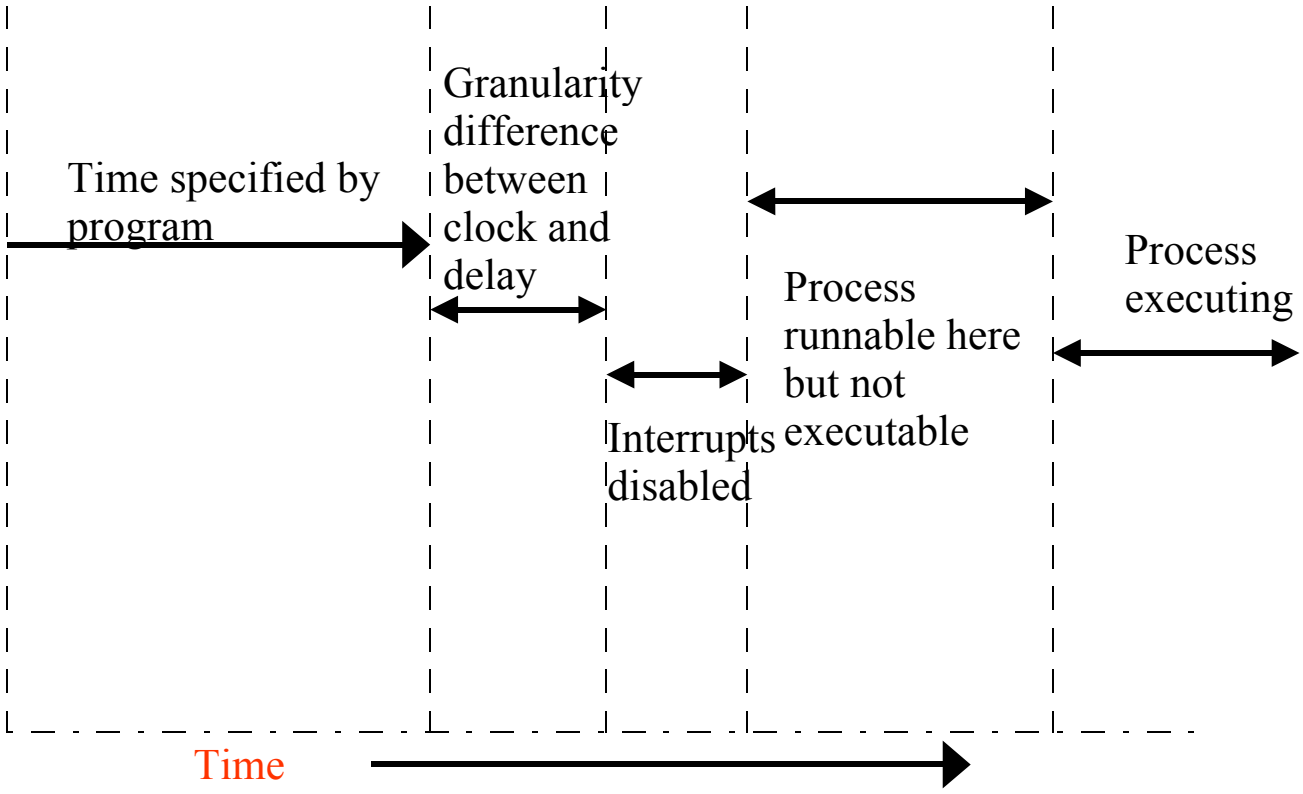
- To avoid busy-waits, most languages/OSs provide delay primitives
- *Ada*: delay statement
  ```
  delay 10.0;
  ```
- *C/POSIX*: **`sleep`** and **`nanosleep`**
- *Java*: **`sleep`**; RT Java provides a high resolution sleep
- ***Note***: granularity of delay and granularity of clock not always the same
- ***Note***: sometimes delays are also referred to as ***synchronous timeouts***

# *How long is the delay really?*

The actual time delay depends on

- the specified delay (relative or absolute) – but this is only the lower bound
- the granularity of the clock
- the processing overhead of the delay imposed by the OS
- whether interrupts are enabled when the time-out occurs
- whether a process is runnable then

# Delays


Time specified by program
Granularity difference between clock and delay
Interrupts disabled
Process runnable here but not executable
Process executing
Time

# *Relative vs. Absolute Delays*

- Specifying a 10 second relative delay in Ada:

```
START := Clock;
FIRST_ACTION;
delay 10.0 - (Clock - START);
SECOND_ACTION;
```

- The above delay may be prolonged by interrupts (**`delay`** is not atomic)
- An alternative using an absolute delay:

```
START := Clock;
FIRST_ACTION;
delay until START + 10.0;
SECOND_ACTION;
```

# *Relative vs. Absolute Delays*

- Like relative delays, absolute delays are accurate only in their lower bound
- RT Java: `sleep` can be relative or absolute
- POSIX: requires use of an absolute timer and signals

# *Drift*

- The time over-run associated with both relative and absolute delays is called the ***local drift*** and it it cannot be eliminated
- It is possible, however, to eliminate the ***cumulative drift*** that could arise if local drifts were allowed to superimpose

# Regular Activity

```
task T;
task body T is
begin
  loop
    Action;
    delay 5.0;
  end loop;
end T;
```

Cannot delay for less than 5 seconds

local and cumulative drift

# Periodic Activity

```
task body T is
  Interval : constant Duration := 5.0;
  Next_Time : Time;
begin
  Next_Time := Clock + Interval;
  loop
    Action;
    delay until Next_Time;
    Next_Time := Next_Time + Interval;
  end loop;
end T;
```

Will run on average every 5 seconds

local drift only

If `Action` takes 6 seconds, the delay statement will have no effect

# *Timeouts*

- A typical requirement on an RT system:
  - Must recognize, and act upon, non-occurence of some external event
- An ***(asynchronous) timeout***:
  - A restriction on the time a thread (or process) is prepared to wait for some event
- A ***timer*** is a means to implement a timeout
- The OS
  - provides one or more systemwide timers to be used by the threads (LINUX), or
  - allows threads to create their own timers (RT POSIX compliant systems, Windows NT, Solaris)

# *Timers*

- A timer contains
  - an expiration time
  - (optionally) a pointer to a handler that is executed when a timer event occurs
- A thread ***sets*** (or ***arms***) a timer when it asks the OS to give the timer a future expiration time
- A timer is ***canceled*** (or ***disabled***) if it is set to ***expire*** before the timer event
- Further functions allow to specify what action to perform if a timer expires
  - calling a function, waking up a thread, sending a message ...

# *Expiration Times*

- Expiration times can be
  - ***absolute*** (specific time instant)
  - ***relative*** (length of delay until expiration)
- Expirations can be
  - only once (***one-shot***; ***watchdog timer***)
  - several times (***periodic***)
- The granularity of time measured by the applications is the ***actual timer resolution***
- As with delays, the ***requested expiration time*** is only a lower bound on the ***actual expiration time***

# *Ada: Timeouts on Actions*

```
select
  delay 0.1;
then abort
  -- action
end select;
```

- ***Watchdog***: If an action takes too long, the triggering event will be taken and the action will be aborted
- Watchdogs are an effective way of catching ***run-away code***

# Ada: Imprecise Computation

```
declare
  Precise_Result : Boolean;
begin
  Completion_Time := ...
  --  compulsory part
  Results.Write(...); -- call to procedure in
                      -- external protected object
  select
    delay until Completion_Time;
    Precise_Result := False;
  then abort
    while Can_Be_Improved loop
      -- improve result
      Results.Write(...);
    end loop;
    Precise_Result := True;
  end select;
end;
```

# *RT Java: Timeouts on Actions*

- With Real-Time Java, timeouts on actions are provided by a subclass of `AsynchronouslyInterruptedException` called `Timed`

```
public class Timed
          extends AsynchronouslyInterruptedException
          implements java.io.Serializable
{
  public Timed(HighResolutionTime time) throws
               IllegalArgumentException;

  public boolean doInterruptible(Interruptible logic);

  public void resetTime(HighResolutionTime time);
}
```

# RT Java: Timers I

```
public abstract class Timer extends AsyncEvent
  {
    protected Timer(HighResolutionTimer time,
                    Clock clock,
                    AsyncEventHandler handler);

     public ReleaseParameters createReleaseParameters();
     public AbsoluteTime getFireTime();
     public void reschedule(HighResolutionTimer time);
     public Clock getClock();

     public void disable();
     public void enable();
     public void start(); // start the timer ticking
   }
```

# RT Java: Timers II

```
public class OneShotTimer extends Timer
{
  public OneShotTimer(HighResolutionTimer time,
                      AsyncEventHandler handler);
}

public class PeriodicTimer extends Timer
{
  public PeriodicTimer(HighResolutionTimer start,
         RelativeTime interval,
         AsyncEventHandler handler);
  public ReleaseParameters createReleaseParameters();
  public void setInterval(RelativeTime interval);
  public RelativeTime getInterval();
  public void fire();
  public AbsoluteTime getFireTime();
}
```

- POSIX does not directly support the ***Asynchronous Transfer of Control*** (***ATC***) as Ada and Java
  - Therefore, it is difficult to specify timeouts on actions
- POSIX does support Timers
  - relative or absolute times
  - delivers a signal upon expiration (`SIGALRM` by default)
- Signal is delivered to whole process
  - In the presence of multiple threads, this makes it difficult to identify which thread has overrun its deadline

# Timers in POSIX

```
#define TIMER_ABSTIME ...
struct itimerspec {
  struct timespec it_value;    /* first timer signal */
  struct timespec it_interval; /* subsequent intvls */
};

typedef ... time_t;
int timer_create(clockid_t clock_id,
                 struct sigevent *evp,
                 timer_t *timerid);
int timer_delete(timer_t timerid);

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value,
                  struct itimerspec *ovalue);
int timer_gettime(timer_t timerid,
                  struct itimerspec *value);

int timer_getoverrun (timer_t timerid);
```

# *Summary I*

- The programming language and/or the OS provide access to *clock times* and *calendar times* – of varying quality
- *Clock devices* consist of clocks + timers
- The quality of the clock accessible to the programmer depends on how the clock is maintained and how it is accessed
- Execution may be delayed by absolute or relative values – which provide lower bounds for the *actual* delay

# *Summary II*

- Timers can be one-shot or periodic
- Watchdogs can catch run-away code
- Watchdogs can be implemented directly by timeouts on actions

# To Go Further

- ***Incorporating Time into RT Languages***
  - ☞ Chapter 12 of [Liu 2000]
  - ☞ Chapter 12 of [Burns and Wellings 2001]

# *Announcement – Reminder*

The class on May 10
(Friday after "Himmelfahrt")
is ***cancelled***

There will also be ***no exercise*** on May 14

In lieu of the class, the following reading assignment:

1. P. J. Landin, The next 700 programming languages, *Communications of the ACM*, March 1966, http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ss02/rt-prog/papers/p157-landin.pdf
2. Sixto Ortiz Jr., The Battle over Real-Time Java, *IEEE Computer*, Vol. 32, No. 6; June 1999, pp. 13-15. http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ss02/emb-pl/papers/p13-ortiz.pdf

# *Problem Set 4 – Due: 16 May 2002*

1) Give a summary (2000 chars, +/- 500) of P. J. Landin, The next 700 programming languages*, Communications of the ACM*, March 1966. Do you agree with his criticism of the proliferation of programming languages? Do you think the situation has changed since then? **(2 pts)**
2) Java *and C* are case-sensitive; *Ada* is not. What are the arguments for and against case sensitivity?
   **(2 pts)**
3) Design and implement a program to measure the actual delay *d* of a call to `sleep(s)`. How does *d* vary as a function of *s* for a program written in ...
   a) *C* ? **(2 pts)**
   b) *Java* ? **(2 pts)**
   c) *Real-Time Java* ? **(2 pts)**
   d) What does the delay depend on? **(2 pts)**
   e) What does the accuracy of your measurement depend on? **(2 pts)**

***Note:*** For instructions on how to get started with RT Java with a Linux system, you can hava a look at http://www.informatik.uni-kiel.de/~kwi/programmierung/arbeit.html