Implementation of Real-Time Systems

Real-Time Requirements

- A real-time specification will include certain time constraints, such as:
  - Relative to a designated event (or to some absolute time, which can be regarded as an event):
    - An event should not occur until a given time has elapsed.
    - An event should not occur after a given time has elapsed.
    - A certain computational activity must take place before an event.
  - Responding to one kind of event takes priority over responding to another.

How do processors deal with such constraints?

- Responding to external events:
  - **Polling**: Processor cycles through a series of tests for external flags raised by events. The time of a cycle can be determined by counting machine cycles and knowing the clock rate.
  - **Interrupts**: Raising a flag causes an interrupt, which directs the processor to an appropriate ISR (Interrupt Service Routine).

Timing:

- Processors incorporate a clock running at a known rate. Therefore it is simple to construct an **interval timer**, which counts the number of ticks elapsed:
  \[
  \text{elapsed time} = \frac{\# \text{ of ticks}}{\text{clock rate}} \quad \text{(in ticks per sec)}
  \]

Example: Interval Timers in Solaris

- #include <sys/time.h>

- int getitimer(int which, struct itimerval *value);

- int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
Example: Interval Timers in Solaris

- struct itimerval is defined to include:
  - struct timeval it_value; /* current value */
  - time until next expiration
  - struct timeval it_interval; /* timer interval*/
  - time to reload when interval expires
- struct timeval includes:
  - time_t tv_sec /* seconds */
  - long tv_nsec /* nanoseconds */

Example: More Interval Timers in Solaris

- int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid);

  where clock_id is one of:
  - CLOCK_REALTIME (wall clock)
  - CLOCK_VIRTUAL (user CPU usage)
  - CLOCK_PROF (user and system CPU)

Example: More Interval Timers in Solaris

- 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);

Example microprocessor widely-used for real-time
Intel 8051 (MCS-51 family)

- 40-pin package
- 8-bit word length
- 1 instruction = 2 bytes
- 4k bytes ROM (set at factory per user spec.)
- 128 bytes RAM (on chip)
- 64K RAM addressing limit
- 32 I/O lines
- 2 16-bit timers, which work in various modes
- 5 interrupt sources (2 external)
- 1 duplex serial port

Intel 8051

- Each processor cycle has 6 states x 2 phases = 12 ticks.
- Timers are incremented once per cycle.
- Timers can also be used as event counters.

Intel 8051

- Events that can trigger interrupt:
  - Timer 0 Overflow.
  - Timer 1 Overflow.
  - Reception/Transmission of Serial Character.
  - External Event 0.
  - External Event 1.
  - Each type of interrupt has a different ISR address.
  - Interrupts can be disabled individually.
  - Interrupt priorities are set in registers.
Intel 8051
Interrupt Sequence

- On interrupt:
  - The current Program Counter is saved on the stack, low-byte first (yes, there is a stack-pointer register).
  - Interrupts of the same and lower priority are blocked.
  - In the case of Timer and External interrupts, the corresponding interrupt flag is cleared.
  - Program execution transfers to the corresponding interrupt handler vector address.
  - The Interrupt Handler Routine executes.

- On return:
  - Two bytes are popped off the stack into the Program Counter to restore normal program execution.
  - Interrupt status is restored to its pre-interrupt status.

Intel 8051

- Programming: C or Assembler
- Many derivatives, by several vendors
- Extension: 8052 (additional capabilities)

Real-Time Operating Systems

- INTEGRITY (Green Hills Software)
- QNX (QNX Software Systems)
- RT-Mach (CMU)
- RTMX O/S (Open BSD + Realtime extensions)
- Solaris (Sun)
- Spring Kernel (U. of Massachusetts)
- VRTX (Mentor Graphics)
- VxWorks (Wind River Systems)
  - and many, many others
- Some links:

Requirements for RTOS's

- Multi-threading with:
  - priorities
  - preemption (including of the kernel itself)
  - programmable scheduling
  - predictable thread-switching latency
  - synchronization and mutual exclusion mechanism that have predictable delay
  - priority inheritance mechanism
  - Timers that can trigger events
  - Interrupt handlers (also predictable)

Examples from VxWorks

- VxWorks processes are similar to threads in that they all share a common memory space.
- VxWorks provides time-sliced (round-robin) process scheduling, with preemption based on priority.
- The scheduler can be disabled on a per-task basis by calling taskLock() and taskUnlock()
Examples from VxWorks

- Task spawning is currently not Posix:
  - `taskSpawn(name, priority, options, stacksize, main, arg1, …, arg10)`
    - create and activate a new thread
  - `taskOptionsSet()`,
  - `taskSuspend()`, `taskResume()`,
  - `taskRestart()`, `taskDelay()`
  - `nanosleep()`

Examples from VxWorks

- Semaphores have a timeout option:
  - If no one posts the semaphore within the specified timeout period, the waiting task continues anyway (without decrementing the value)
  - For a non-binary semaphore, `semFlush()` wakes up all waiting processes.
  - Semaphore queue can be specified as either Priority Queue or FIFO.
  - Priority inheritance is an option with Priority queue semaphores.

Examples from VxWorks

- Posix semaphores are also available
- Message queues and pipes are available
- Watchdog timers:
  - Run for a settable length of time.
  - If not cancelled within that time, a settable function is called.

Examples from VxWorks

- Message queues and pipes are available
- Watchdog timers:
  - Run for a settable length of time.
  - If not cancelled within that time, a settable function is called.

A tale of Priority Inversion in VxWorks

- "What really happened on Mars", etc.

RT case study

Structure of DiCK (Didactic C Kernel)

<table>
<thead>
<tr>
<th>System call layer</th>
<th>process management</th>
<th>machine layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>creation termination</td>
<td>communication synchronization</td>
<td>utility services</td>
</tr>
<tr>
<td>scheduling</td>
<td>dispatching</td>
<td></td>
</tr>
<tr>
<td>creation termination</td>
<td>communication synchronization</td>
<td>utility services</td>
</tr>
</tbody>
</table>
Robot Control Example

- Puma 560 robot arm with wrist force/torque sensor
- CCD camera
- Arm has to exert desired forces on the object surface and follow the surface contour using visual feedback from a camera, e.g. in a deburring operation.

Robot Arm with Deburring Tool

Robot Control

- Organization:
  - Two servo loops communicating via “cabs”
  - Low-level loop:
    - image acquisition
    - force reading
    - robot control
  - High-level loop:
    - scene analysis
    - surface reconstruction

Process Structure

CAB = Cyclical Asynchronous Buffer

- One-to-many communication channel
- At each instant contains the latest message or data inserted
- Message is not consumed; it remains in the CAB until over-written.
- No blocking on either writing or reading.

CAB (2)

- If receiver is faster than sender, the same message may be reread multiple times.
- If sender is faster, a message can be overwritten before it has been read.
- The structure of a CAB guarantees the semantics of reading an integral datum or message, as opposed to a hybrid.
- The inner workings reveal multiple buffers.
Examples where CABs are Useful

- **Graphics**: Multiple updates to the screen per physical refresh: If update is slow, don't want to see stale information; just get the latest.
- **Vision**: Multiple frame updates per use if processing is slow.
- **Sensing**: Device constantly produces new information whether or not requested.

CAB Primitives (1)

- Insert a message:
  - Task must reserve a buffer from the CAB
  - Copy message into the buffer
  - Put the buffer into the CAB structure as most-recent
  - Code:
    
    ```
    buf_pointer = reserve(cab_id);
    ... put message into *buf_pointer ...
    putmes(buf_pointer, cab_id);
    ```

CAB Primitives (2)

- Get a message:
  - Get the pointer to the most recent message
  - Use data in the message
  - Release the pointer
  - Code:
    
    ```
    mes_pointer = getmes(cab_id);
    ... use message from *mes_pointer
    unget(mes_pointer, cab_id);
    ```

CAB Primitives (3)

- The maximum number of buffers is specified in the open_cab primitive.
- This should be 1 more than the number of tasks that write to the CAB.

Robotic Application Processes

- **Sensor process**: periodically reads force/torque sensor into “force” CAB.
  - Must have guaranteed execution time: 20 ms. period. Missing causes instability.
- **Vision process**: periodically reads CCD buffer and computes next direction. Data put in “path” CAB.
  - Must have period of 80 ms.
  - Missing deadline causes tracking to fail.

Robotic Application Processes

- **Robot control process**: Computes target points for arm. Hybrid position/force control moves end-effector along a direction tangential to the object surface, applying a force normal to the surface all the while.
  - 28 ms period, imposed by the communication protocol of the robot
  - Missed deadline could cause gouging or breakage.
Robotic Application Processes

- **Surface representation process**: based on force/torque data and the direction being pursued.
- Soft task with period 60 ms.

---

Program (2)

```c
#include "di ck.h"
#define TICK 1.0 /* system tick (1 ms) */
#define T1 20.0 /* period for force */
#define T2 80.0 /* period for vision */
#define T3 28.0 /* period for control */
#define T4 80.0 /* period for display */
#define WCET1 0.300 /* execute time for force */
#define WCET2 4.780 /* execute time for vision */
#define WCET3 1.183 /* execute time for control */
#define WCET4 2.230 /* execute time for display */
cab fdata; /* CAB for force data */
cab angle; /* CAB for path angles */
proc force; /* force sensor acquisition */
proc vision; /* camera acquisition & processing */
proc control; /* robot control process */
proc display; /* robot trajectory display */
```

Program (3)

```c
proc main()
{
  ini_system(TICK);
  fdata = open_cab("force", 3*sizeof(float), 3);
  angle = open_cab("path", sizeof(float), 3);
  create(force, HARD, PERIODIC, T1, WCET1);
  create(vision, HARD, PERIODIC, T2, WCET2);
  create(control, HARD, PERIODIC, T3, WCET3);
  create(display, HARD, PERIODIC, T4, WCET4);
  activate_all();
  while( sys_clock() < LIFETIME ) /* do nothing */ ;
  end_system();
}
```

Program (4)

```c
proc force()
{
  float* fvect;
  while( 1 )
  {
    fvect = reserve(fdata);
    read_force_sensor(fvect);
    putmes(fvect, fdata);
    end_cycle();
  }
}
```

Program (5)

```c
proc control()
{
  float *fvect, *alfa;
  float x[6];
  while( 1 )
  {
    fvect = getmes(fdata);
    alfa = getmes(angle);
    control_law(fvect, alfa, x);
    send_robot(x);
    unget(fvect, fdata);
    unget(alfa, angle);
    end_cycle();
  }
}
```
Languages & Models for Real-Time Systems

Why look at Languages?

- A language can be considered to be a large class of specific applications.
- If the language is properly implemented, the applications are implemented as a consequence.

Why Models?

- A model is an abstract version of the solution to a problem.
- Models are often prologues to being able to express the application in a specific language.

Ada 95 Language Features Pertinent to Real-Time and Parallelism (see http://www.adahome.com/ada95/rtf-loc.html)

- Tasking (threads), dynamic priorities
- accept statement: task interacts with environment
- select statement: respond to alternate requests (non-deterministically)
- delay option: specifies minimum time delay
- when clause: part of case statement
Ada speak

- entry: declaration of an interaction point with a task type
- entry-call: `taskName.entryName(...)params...`
- accept: code for servicing an entry & rendezvous
- select: code for choosing among entries (implicitly queued) and delays.

Non-determinism using Select

```ada
task Protected_Variable is
  entry Read(X: out Item);
  entry Write(X: in Item);
end;
PV: Protected_Variable;

loop
  select
    accept Read(X: out Item) do
      X := V;
    end;
  or
    accept Write(X: in Item) do
      V := X;
    end;
  or
    terminate;
end select;
end loop;
end Protected_Variable;
```

Non-determinism + Termination

```ada
task body Protected_Variable is
  V: Item;
begin
  accept Write(X: in Item) do
    V := X;
  end;
loop
  select
    accept Read(X: out Item) do
      X := V;
    end;
  or
    accept Write(X: in Item) do
      V := X;
    end;
  or
    terminate;
end select;
end loop;
end Protected_Variable;
```

Semaphore in Ada

```ada
protected type Semaphore (Start_count :Integer := 1) is
  entry wait;
  procedure signal;
  function Count return integer;
private
  Current_count :integer := start_count;
end;
protected
  body Semaphore is
    procedure signal is
      Current_Count := Current_count + 1;
    end;
    entry
      wait when Current_count > 0 is
        Current_Count := Current_count - 1;
      end;
    end Semaphore;
```

"Unconditional" Alternative

```ada
task multiple_entry is
  entry increment;
  entry decrement;
end multiple_entry;

task body multiple_entry is
  i : integer;
begin
  select
    accept increment do
      i := i + 1;
    end;
  or
    accept decrement do
      i := i - 1;
    end;
  else
    perform_some_processing;
  end select;
end;
```

Diagrammatic Model of Typical Ada Processing by Loop

- Order of joining queue can be non-deterministic
- Possibly-looping "server" task
- Order of service can be non-deterministic
- Client tasks
- Non-determinism using Select
- Semaphore in Ada
- "Unconditional" Alternative
"when" conditions

```plaintext
task body Buffering is
N: constant := 8; -- for instance
A: array (1 .. N) of Item;
I, J: Integer range 1 .. N := 1;
Count: Integer range 0 .. N := 0;
begin
loop
select
when Count < N =>
  accept Put(X: in Item) do
  A(I) := X;
  end;
  I := I mod N+1; Count := Count+1;
or
when Count > 0 =>
  accept Get(X: out Item) do
    X := A(J);
  end;
  J := J mod N+1; Count := Count-1;
en end select;
end loop;
```

"when" conditions: readers/writers

```plaintext
task body Count is
Readers: Integer := 0;
begin
  accept Write(X: in Item) do
    V := X;
  end;
loop
select
  accept Start;
    Readers := Readers+1;
  or
  accept Stop;
    Readers := Readers-1;
  or
  when Readers = 0 =>
    accept Write(X: in Item) do
      V := X;
    end;
en end select;
end loop;
end Count;
```

Delay Alternative

```plaintext
select
  accept Read( ... ) do
    ...
  end;
or
  accept Write( ... ) do
    ...
  end;
or
delay 10*Minutes; -- time out statements
end select;
```

Autonomous Periodic Tasks
(one loop for each task)

```plaintext
task body Periodic_Task is
Next_Time: Calendar.Time := Task_Start_Time;
begin
  delay (Task_Start_Time - Calendar.Clock);
  loop
    -- do work
    Next_Time := Next_Time + Task_Period;
    delay (Next_Time - Calendar.Clock);
  end loop;
end Periodic_Task;
```

Problem with Previous Approach

- **Delay jitter:**
  - A task can be preempted between reading clock and starting its delay, making the delay end later than planned.

- **Cumulative effect is that deadlines can be missed.**

- **A better approach is to use a central dispatcher task.**

Dispatcher Model

```plaintext
task body Periodic_Dispatcher is
begin
  loop
    accept Clock_Interrupt;
    loop
      -- Determine which task to activate next
      select
        Selected_Task.Activate;
      else
        -- Handle missed deadline
        end select;
    end loop;
  end loop;
end Periodic_Dispatcher;
```

```
```
Better Model: Use builtin \textit{delay\_until}

\begin{verbatim}
  task body Periodic_Task is
    Next_Time : Calendar.Time := Task_Start_Time;
    begin
      delay_until(Task_Start_Time);
      loop
        -- do work
        Next_Time := Next_Time + Task_Period;
        begin
          delay_until(Next_Time);
          exception
            when Calendar.Time_Error => -- handle missed deadline;
          end;
        end loop;
    end Periodic_Task;
\end{verbatim}

Priority Issues

\begin{itemize}
  \item To get Ada to use the Priority Ceiling Protocol:
    \begin{verbatim}
    pragma Locking_Policy(Ceiling_Locking);
    \end{verbatim}
  \item Recall that PCP guarantees:
    \begin{itemize}
      \item Execution of a high-priority task can be delayed by at most one lower priority task per call.
    \end{itemize}
\end{itemize}

Sources

\begin{itemize}
  \item A lot of these examples are from:
    \begin{verbatim}
    \end{verbatim}
  \item See also:
    \begin{itemize}
      \item http://www.sei.cmu.edu/publications/documents/89.reports/89.tr.022.html
    \end{itemize}
\end{itemize}