SYSC 5701
Operating System Methods for Real-Time Applications

Real-Time Languages

Winter 2014

Languages for Real-Time Systems

- survey of existing real-time features
  – W. A. Halang, A. D. Stoyenko, ``Comparative Evaluation of High Level Real Time Programming Languages,'' *Real-Time Systems*, Volume 4, Number 2, pp. 365 – 382, December 1990
- **conventional elements**
  – bit processing
  – re-entrant procedures
  – I/O capabilities

Halang & Stoyenko

- **processes/tasking**
  – hierarchy
  – run-time statistics available
  – scheduling strategies
  – priorities – static and/or dynamic
  – IPC: messaging
- **synchronization**
  – semaphores
  – “other” – IPC (e.g. mutex, condition variables, rendezvous)
  – resource reservation & allocation strategy
  – resource statistics available

Halang & Stoyenko

- **events**
  – interrupt handling
  – enable/disable interrupt
  – asynchronous event signaling
- **timing**
  – date/time available
  – cumulative run-time available
  – timed scheduling support
  – timed synchronization/IPC services
- **run-time verification**
  – exceptions

Halang & Stoyenko Suggest Additional Desirable Features

- application-oriented synchronization constructs
  – e.g. monitors
- surveillance of:
  – occurrences of events within time windows
  – occurrences of event sequences
  – timeout of resource claims
- availability of current task and resource states
- inherent prevention of deadlocks

Halang & Stoyenko: Additional Desirable Features

- feasible scheduling algorithms
  – analyzable schedulability
- early detection and handling of transient overloads
- accurate real time
- exact timing of operations
- dynamic configuration for fault recovery
- event recording/tracing

sounds like a lot to ask of a language!

Conventional Approach

- separate **language** and **kernel**
- procedural language
  - gets close to machine when needed
  - e.g. C, assembly language
    - how many of desired features are supported in language directly?
- most of real-time support is in the kernel!
- application program makes "calls" to kernel primitives (i.e. invokes primitives)

Conventional: Kernel

- kernel primitives are not part of the language
- program compiled using kernel interface library to hide details of invocation
- compiled object file linked with kernel object file to create a load module (static binding)
- sometimes compiled object is converted to load module – assumes that kernel is pre-loaded in target – dynamic binding mechanism (hidden by interface library)

Conventional Problems ☹

- resulting application is kernel/platform dependent
  - "virtual machine" abstraction is too low-level
  - portability, platform evolution ?? ☹
    - E.G. FreeRTOS vs. μC/OS
- multiple tool vendors: compiler (IDE), kernel
  - compatibility issues – more details! ☹

Languages to Address Hard Real-Time

- **Ada (1983, 1995, 2005, 2012)**
  - DoD
- **Pearl** (not PERL) **(early 1970'S !!)**
  - extinct? (Germany, control systems)
- **Real-Time Euclid (1986)** – research project
  - extends Euclid (U. of T. early 80's)
- **Real-Time Java (2000)** - time permitting

Ada 83

- include kernel-like functionality (and support) in language
  - multitasking & multiprocessing
  - motivation: US DoD **software engineering** concerns
    - **too many** different platforms/kernel/language combinations – unmanageable
    - **goal**: single software platform (language + kernel) for applications
    - **goal**: decrease (hide) hardware dependence
- very ambitious ! !

Ada 83

- Pascal-like syntax & strongly-typed
- **package**
  - modularity & information hiding
  - black-box modules → software engineering roots!
- **task** concurrent process
- packages and tasks have separate **specification** (interface) and **body** (implementation)
- **interrupts** are integral part of language

procedure syntax Ada 83

- **in** param's: set by caller
- **out** param's: set by callee → reply!
- **inout** param's: initial value set by caller, returned value set by callee
- argument list fixed by syntax at **compile time**
- e.g.: procedure DeQ (aList: **inout** integer_list;
  x: **out** integer)
- Functions? Yes, but **no side effects**

Package

- **specification**: define publicly-visible artifacts that may be accessed (imported) from outside of the package
  - e.g. data objects, object types, procedures, task + entries
- **body**: implementation of exported artifacts
  - may include hidden declarations and definitions (used in implementation)
  - hidden artifacts cannot be directly accessed from outside of package

Spec is different from .h files !!!

Package

- can separately compile specification and body
- s/w eng roots!
- "packaging" is a common object-oriented concept
- Ada83 does not implement "modern" OO features
  - e.g. inheritance

Stack Package Example

"--" = comment

in/out parameters

- - stack package for integer variables
- - SPECIFICATION
package INTEGER_STACK **is**
  type STATUS is (OK, UNDERFLOW, OVERFLOW);
  procedure PUSH (E : **in** INTEGER; FLAG : **out** STATUS);
  procedure POP (E : **out** INTEGER; FLAG : **out** STATUS);
end INTEGER_STACK;

- This is all that is public for "user" of package to see

Stack Example

Pascal-like type and data declarations

HIDDEN: array implementation of stack

```
- - BODY
package body INTEGER_STACK is
    SIZE : constant INTEGER :=  10;
    SPACE : array (1 . . SIZE) of INTEGER;
    INDEX :  INTEGER range  0 . . SIZE :=  0;

procedure PUSH ( E : in INTEGER; FLAG: out STATUS ) is
begin
  if  INDEX  = SIZE  then  FLAG := OVERFLOW
    else          INDEX  :=  INDEX + 1;
                  SPACE( INDEX )  :=  E ;
                  FLAG  :=  OK;
  endif;
end PUSH;
```


Stack Example

```
procedure POP ( E : out INTEGER; FLAG: out STATUS ) is
begin
   . . .
end POP;
end INTEGER_STACK;
```

- user calls:

  STACK . PUSH (ELEMENT, STAT);

  STACK . POP (ELEMENT, STAT);

Ada Tasks

- spec/body syntax similar to packages
- **rendezvous** IPC (send/receive/reply)
- tasks accessed by calls to rendezvous **entry**s
  - like a procedure call for task IPC ports
  - similar syntax to procedures (in/out params)
- task **accepts** entry
  - **blocked** if no caller waiting
- caller is **blocked** until accepted by task
- caller released when task finishes entry

Example: Character Buffer Task

```
- - character BUFFER SPEC
task BUFFER is
   entry READ ( C : out CHARACTER );
   entry WRITE ( C : in CHARACTER );
end BUFFER;
```

- This (and maybe some documentation) is all that is given to programmers that use the buffer task

Single Char Buffer Example

```
task body BUFFER is
POOL : CHARACTER;
begin
   loop
      accept WRITE ( C : in CHARACTER ) do
         POOL := C;
      end;
      accept READ ( C : out CHARACTER ) do
         C := POOL;
      end;
   endloop;
end BUFFER;
```

local variable: persistent over life of task (POOL : CHARACTER)

accept WRITE then accept READ

reply! (C := POOL)

release caller (end;)

Single Char Buffer Example

- producer task calls:
  BUFFER . WRITE(CHAR);
- consumer task calls:
  BUFFER . READ(CHAR);
- What happens if reader calls before a character is in the buffer?

Selective Accept

- Single statement allows receiver to receive from more than one entry

```
select
      accept entryA
or
      accept entryB
or
      …
end select;
```


Multiple Character Buffer Example

- Specification: Same as version 1!!

```
- - character BUFFER SPEC
task BUFFER is
   entry READ ( C : out CHARACTER );
   entry WRITE ( C : in CHARACTER );
end BUFFER;
```

Multiple Char Buffer Example

task body BUFFER is
- - character buffer (BUFF) is a circular FIFO list
- - in a static array
B_SIZE : constant INTEGER = 100;
BUFF : array (1 . . B_SIZE) of CHARACTER;
IN_INDEX, OUT_INDEX :
INTEGER range 1 . . B_SIZE := 1;

Multiple Char Buffer Example

```
begin
 loop
  select      - - accept whichever one is ready!
   accept WRITE ( C : in CHARACTER ) do
          BUFF(IN_INDEX)  := C;          in rendezvous
     end;
     IN_INDEX := IN_INDEX  mod  B_SIZE + 1     outside of rendezvous
    or
   accept  READ ( C : out CHARACTER ) do
          C  :=  BUFF(OUT_INDEX);
     end;
     OUT_INDEX := OUT_INDEX mod B_SIZE + 1
  end select ;
end loop;
end BUFFER;
```

Multiple Char Buffer Example

- **selective accept**: receive from either Producer or Consumer – whenever called
- "**mutex**" (sequential) in Buffer task – no interference at BUFF!
- **what if** consumer calls when BUFF empty
  - nothing to consume? should block? exception?
- **what if** producer calls when BUFF full
  - no space to store? should block? exception?

GUARDS

- conditional closing of entries during select
- **guard**: boolean condition
  - when **true**: entry **open** – will be selected
  - when **false**: entry **closed** – will not be selected
- Guard evaluated each time containing "select" is executed

Syntax:

when boolean_condition_true =>

Modifications to Buffer Task Body

```
COUNT :  INTEGER range 1 . . B_SIZE  := 0 ;
begin
loop
  select                                   guard against full condition
   when   COUNT  <  B_Size   =>
    accept WRITE ( C : in CHARACTER ) do
            BUFF(IN_INDEX)   :=  C;
        end;
        IN_INDEX := IN_INDEX  mod  B_SIZE + 1;
        COUNT  := COUNT + 1;
```


Modifications to Buffer Task Body

```
  or                                       guard against empty condition
   when   COUNT  >  0   =>
   accept  READ ( C : out CHARACTER ) do
             C  :=  BUFF(OUT_INDEX);
        end;
        OUT_INDEX := OUT_INDEX mod B_SIZE + 1;
        COUNT  := COUNT – 1;
  end select;
end loop;
```

Some Variations

not necessarily a call to RCV_TASK

- **sender** can **select** alternative **action** if receiver is not ready to accept entry:

```
select
  RCV_TASK . RNDZVOUS; -- do rendezvous
or
  RCV_NOT_READY_proc;  -- call procedure
end select ;
```


Delay

- **delay** – an “else” alternative in select
- for receivers:

```
select
      . . .            - - selective accepts as before
or
      delay   T ;
       - - no callers in time T
      do_DELAY_processing ;
end  select ;
```


Delay

- delay – can also expand sender’s options

```
select
  RCV_TASK.RNDZVOUS;
or
  delay  T;
  TIMED_OUT_proc;
end select ;
```

- if sender not accepted within time T – then call is aborted, and sender performs TIMED_OUT_proc

Ada 95

- improvements based on 10 years of trying ☺
- enhance to include classical O-O features
- improved tasking:
  - more efficient IPC
  - more predictable operation
  - improved interrupt handling

Ada 95

- decomposed standard from “all-or-nothing” to core + annexes
  - Ada83 – compiler did everything, or was not certified
  - Ada95 – compiler must support minimum core and then annexes as desired
    - allowed more efficient compilers (profiles)

Some Interesting/Relevant Annexes

Systems Programming

- machine operations
- interrupt support
- user-defined allocation/finalization
- shared variable control
- task identification (vs. global names)

Some Interesting/Relevant Annexes

Real-Time Systems

- priorities – static and dynamic
  - interrupt and task priorities
- task dispatching
  - run-until-blocked/completed
  - preemption
- ceiling priorities
- entry queuing facilities → done by tasks!
  - forward calls to different entries
  - requeue entries

Some Interesting/Relevant Annexes

Real-Time Systems (con't)

- mutex and synchronization
  - protected types (monitors?)
- can configure for simpler tasking models
- e.g. max. number of tasks
  max. number of entries per task
  max. stack space

Some Interesting/Relevant Annexes

Interrupts

- **binding**: associates an **interrupt procedure** with an interrupt
- supports communication between interrupt procedures and other objects
  - bound procedures can modify shared variables that are guarding entries of "protected" objects

Interrupt Example (static binding)

```
use INTERRUPT_MANAGEMENT;
protected Timer is
   entry WAIT_FOR_TICK;
   procedure Handle_Timer_Interrupt;
   pragma ATTACH_HANDLER(
      Handle_Timer_Interrupt , TIMER_INTERRUPT_ID );

private Tick_Occurred : BOOLEAN  := FALSE;
```

creates interrupt procedure binding

binding mechanism is compiler specific

Interrupt Example

```
protected body Timer is
   entry WAIT_FOR_TICK
   when  Tick_Occurred  is
     Tick_Occurred := FALSE;
      - - external task leaves and does once-a-tick stuff
end WAIT_FOR_TICK;

procedure Handle_Timer_Interrupt is
begin
   Tick_Occurred  :=  TRUE;
end Handle_Timer_Interrupt;

end Timer;
```

entry guarded – called by external task

Guards are re-evaluated after every execution of an entry or procedure on a protected object

Ada 2005 (10 more years)

- Additional dispatching models
  - Including: Non-preemptive, and EDF
  - Round robin ☹
- Timing events
  - Define handlers ... Interrupt-like
- Execution time monitoring
  - cumulative run-time (not wall-clock time)
- Ravenscar profile – deterministic!
  - Subset for safety critical systems

Ada 1012 (7 more years)

- Programming contracts → pre- and post-conditions (assertions)
- Task affinities → map onto multicore architectures
- Task-safe queues → more efficient synchronized structures

What about Other Languages that Include “Time”?

- PEARL
- Real-Time Euclid
- Real-Time Java

Pearl (not PERL)

Process and **E**xperiment **A**utomation **R**ealtime **L**anguage

- Germany – early 70’s – collaboration between researchers and industry – motivated by engineering issues in real-time control systems
- overview:
  - procedural aspects: Pascal-like, strong typing
  - allows direct hardware access
  - modules: import and export lists
  - separate compilation

Pearl (con’t)

- includes process definition (**tasks**)
- activation: time and/or event-related
- missing:
  - schedulability analysis provisions
  - structured exception handlers
  - unstructured (semaphore-like) process synchronization

Time in PEARL

- additional data types:
  - **clock** value = time instance
  - **duration** value = time interval

scheduling time-constrained behaviour:

- **simple schedules**: based on temporal events or interrupt occurrence

Pearl Schedules

- (BNF) syntax:

simple-event-schedule ::=
at *clock-expr* **|** **after** *duration-expr* **|** **when** *int-name*

- periodic schedules: start period stop

schedule ::=
at *clock-expr* **all** *duration-expr* **until** *clock-expr*
| after *duration-expr* **all** *duration-expr* **during** *duration-expr*
| when *int-expr* **all** *duration-expr*
{ **until** *clock-expr* **|** **during** *duration-expr* }

Pearl
Task State Transition Control

- tasks are either dormant, ready, running or suspended
- Programs must force task state changes
- dormant to ready (or running):
  [*schedule*] **activate** *task-name* [**priority** *positive-int*]
- ready (or running) to dormant:
  terminate *task-name*
- ready (or running) to suspended:
  suspend *task-name*

Pearl
Task State Transition Control (con't)

- suspended to ready (or running):
  [*simple-event-schedule*] **continue** *task-name*
- running to suspend then back to ready (or running):
  simple-event-schedule **resume**
- all synchronization is tightly controlled
  - Programs explicitly mange task states
  - No general semaphore mechanism
    - Could semaphores be built using forced transitions in a monitor-like structure?

Pearl refs

eds. GI-working group 4.4.2
PEARL 90, Language report, Version 2.2
Technical report GI (1998)
http://www.irt.uni-hannover.de/pearl/pub/report.pdf

D. Stoyenko, ``*Real-Time Euclid: Concepts Useful for the Further Development of PEARL,'' in Proceedings* PEARL 90 --- *Workshop uber*

Realzeitsysteme, W. Gerth and P. Baacke (Eds.), In-for-ma-tik-Fach-be rich-te 262, pp. 12 -- 21, Berlin-Heidelberg-New York: Springer-Verlag, 1990

Real-Time Euclid

- research project – U. of Toronto
  Euclid → Turing → Real-Time Euclid
- Stoyenko (PhD. 1987)
- schedulability analysis
- some academic application
  - no industry experience (as of 1995)

Real-Time Euclid

Features:

- **procedural** aspects – Pascal-like, strongly-typed
- **processes**: run concurrently
  - each process is sequential
  - statically allocated
  - program terminates when all processes terminate
- **modules**: package data together with processes and subprograms (procedures & functions) that use the data

Real-Time Euclid

- can **import/export** subprograms, types and constants
  - cannot export modules or variables
- each module can contain an **initially** section
  - executed before any processes (in program) are run
  - allows convenient initialization of program
- **monitors**: allow only one active process inside
  - wait/signal on condition variables (Hoare, 1974)

Real-Time Euclid

Time?

- **time** managed as “real time” value!
- program defines time increment:
- **RealTimeUnit**(timeInSeconds)
  - e.g. RealTimeUnit(25e-3)
    - one real time unit = 25 milliseconds
- function Time : returns elapsed time from startup in real time units
  - e.g. Time = 10 → 10 * 25e-3

Real-Time Euclid

Constraints to make schedulability analysis possible

- **no dynamic** data structures (e.g. heap)
  - allocation/deallocation time bound at compile-time
  - memory needed for a subprogram to be called and executed is bound
  - can guarantee at compile-time that system has enough memory for processes to execute
- **bounded loops**
  - maximum number of iterations fixed at compile-time

Real-Time Euclid

Constraints to make schedulability analysis possible (con’t)

- looping construct:

for [**decreasing**] *id*: *compIntExpr*. . *compIntExpr*
declarations and statements
[**invariant** *BoolExpr*]
end for

(variable name) (BNF snytax: **[** *optional* **]**) (compile-time integer values!)

- can also terminate loop (early) using:

exit [**when** *BoolExpr*]

  - but must still **have max. iteration bound** !!

Real-Time Euclid

Constraints to make schedulability analysis possible (con’t)

- **no recursion**
- can **analyse** subprogram call trees (a priori)
  - determine memory required (local variables, stack)
  - determine execution times

Side Note: MISRA C

- There other “constraint standards” to make C more deterministic, predictable and analyzable
  → e.g. MISRA C Guidelines
- MISRA = Motor Industry Software Reliability Assoc.
- Rules for programming in C
  - code safety, portability and reliability
- Target: embedded systems programmed in ISO C
- Also a set of guidelines for MISRA C++

Real-Time Euclid

Processes

- static
- can be declared to be periodic or aperiodic
- **activation by**: time, other processes, interrupts
- syntax:

process *id* : *activationInfo*
[*importList*]
[*exceptionHandler*]
declarations and statements
end *id*

Processes (con't)

condition variable or interrupt

time info

condition variable and/or timed

- forms of **activation info**:
  - **aperiodic**:
    - **atEvent** *conditionId* *frameInfo*
  - **periodic**:
    - **period** *frameInfo* **first activation** *timeOrEvent*
- **frame info**: scheduling time frame (e.g. period)
  - **frame** *compIntExpr*
    - absolute frame
  - **relative frame** *compIntExpr*
    - relative to frames of other processes

Processes (con't)

- **timeOrEvent**:
  (first activation of periodic process)
  - **atTime** *compIntExpr*
  - **atEvent** *conditionId*
  - **atTime** *compIntExpr* **or** **atEvent** *conditionId*
- scheduling constraints:
  - **deadline** = frame
  - cannot activate more than once per frame

Condition Variables

- similar to semaphore, but no "counter":
  - **wait**: always block in queue
  - **signal**: always unblocks from the queue
- two types: inside monitor and outside monitor
- inside monitors:
  - used for synchronization when data must be shared
  - programmer responsible for ensuring mutex!!

Condition Variables (con't)

- deferred signal form: (for inside monitor only!)
  - unblocked process is ready to execute in monitor but must wait for mutex turn
  - caller remains running in monitor
- outside monitors:
  - used for synchronization without sharing data

Condition Variables (con't)

only for inside monitor form

- syntax:

  var *conditionId* :
  [**deferred**] **condition** [**atLocation** *intAddress*]
  noLongerThan *compIntExpr* : *timeoutReason*
- ***intAddress***: allows an interrupt to be the signal mechanism
  - i.e. performing the signal is part of the ISR
- **noLongerThan**: max. block time – if time out, then *timeoutReason* is passed to exception handler

Condition Variables (con't)

- **signal**: **signal** *conditionId*
- **wait**: **wait** *conditionId*
  [**noLongerThan** *compIntExpr* : *timeoutReason*]
  - if timebound not specified – uses condition variable's time bound and *timeoutReason*
  - if in monitor and timeout occurs: after processing by exception handler, process is outside monitor and must re-queue if monitor access is desired
- **broadcast**: **broadcast** *conditionId*
  - for outside monitor condition variables
  - signals all processes in queue simultaneously

Real-Time Euclid

Exception Handling

passed to processId's exception handler

kill: **kill** *processId* : *killReason*

- termination (done, dead, caput: no reactivation)
  - part of program is shut down
  - (possibly) raise an exception and terminate
  - if "victim" (processId) is idle – i.e. completed frame and not ready, then no exception raised and victim is terminated
  - process can kill self

Real-Time Euclid

Exception Handling (con't)

deactivate: **deactivate** *processId* : *deactivateReason*
terminate process in the current frame of the victim (possibly self)

- reactivated in next frame
- used for fault recovery in a frame
- if victim not idle – exception raised in victim

except: **except** *processId* : *exceptReason*
raise an exception in processId and continue

- no effect on ready-to-run status

Real-Time Euclid

Exception Handling (con't)

exception handler:

handler (*exceptionReason*)
  exceptions (*exceptionNumber [: maxRaised]*
    { *, exceptionNumber [:maxRasied]* } **)**
  [*importList*]
  declarations and statements
end handler

Real-Time Euclid

Exception Handling (con't)

- some default exception handlers are built-in
  - programmer can replace/override defaults with specific handlers
  - e.g. divide by zero
- each process has an associated exception handler
- when an exception is raised to the process – the handler is "ready"
- if no exceptions raised – handler has no effect

Real-Time Euclid

Exception Handling (con't)

Ready:

- if the process is running, the handler is executed like a software interrupt in the context of the process
- if process is not running, then handler is invoked when process begins to run (unless process was idle while **kill**ed or **deactivate**d)
- handler has priority in process's context

Real-Time Euclid

Schedulability Analysis

- uses techniques similar to hard real-time analysis discussed previously (but more comprehensive!)
- built tools to support analysis
- two parts: front end & back end

front end:

- extracts timing and calling info from compilation units
- execution times of individual statements, subprograms and process bodies
  - does not account for process contention (blocking)
  - gives lower bounds on execution times

Schedulability Analysis (con't)

back end:

- maps system onto a real-time model – includes:
  - platform dependent (h/w) characteristics
  - process contention
  - uses front-end info + analysis of model to arrive at worst-case response times
  - solves for worst-case schedulability

Real-Time Euclid

Schedulability Analysis (con't)

What if resulting processes are not schedulable?

- **front-end info** may help to identify pure processing bottlenecks – candidates for optimization
- **back-end info** may help to highlight contention hot-spots – may need some redesign to eliminate

RT-Euclid refs

"Real-Time Euclid: A Language for Reliable Real-Time Systems", E. Kligerman et al, IEEE Transactions on Software Engineering SE-12(9):941-949 (Sept 1986)