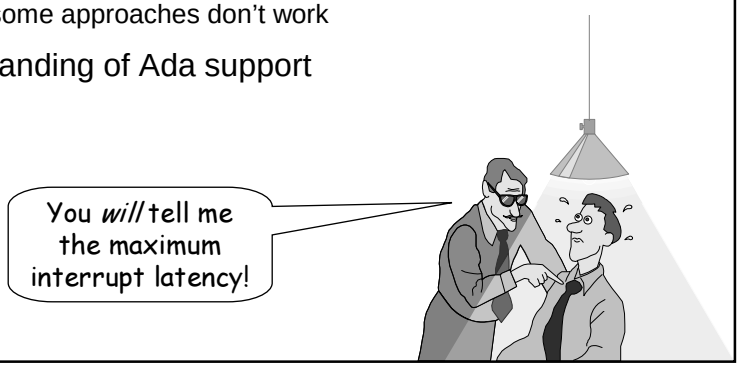
Real-Time Programming in Ada

Patrick Rogers

Software Arts & Sciences
http://www.classwide.com
and
The University of York
Real-Time Systems Group

Tutorial Goals

- Appreciation of modern approaches for *hard* real-time
  - Advantages
  - Risks
- Understanding of scheduling foundation theory
  - How it works
  - Why some approaches don't work
- Understanding of Ada support

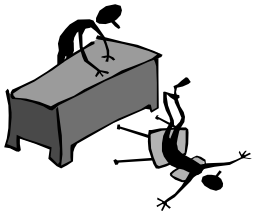


Tutorial Agenda

- Definitions
- Deadline Scheduling Theory
- Support for Deadline Scheduling
- Low-Level Tasking Control
- User-Defined Schedulers

Real-Time Systems

- Timeliness is now a *functional* requirement
  - Right answer at the wrong time is the wrong answer
- Various kinds of timing requirements
  - Minimum interval after initiation
  - Maximum interval after initiation
  - Deadline prior to next periodic initiation
  - others
- Cardiac pacemaker example
  - Too early or too late can cause fibrillation

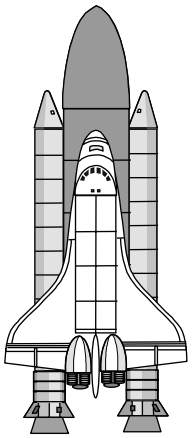


Deadlines

- Hard Deadline
  - Critical : missing the deadline has serious effects
  - System fails if missed
  - Pacemaker example
- Soft Deadline
  - Important, but not individually critical
  - Cumulative misses may become intolerable
  - Airline reservation system example
- Firm Deadline
  - Combination of soft and hard deadlines
  - Can be missed occasionally but no benefit from being late
  - Hospital patient ventilator example
    - Short soft deadline and longer hard deadline

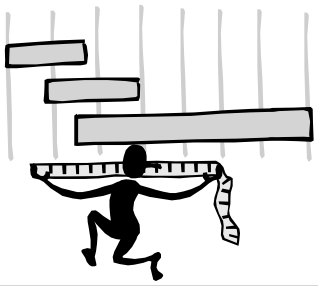
Classes of Systems

- Hard Real-Time (HRT) Systems
  - Dominated by hard deadlines
  - Fails if hard deadlines missed
  - “Best effort” is not sufficient
- Soft Real-Time Systems
  - Not dominated by hard deadlines
    - But may have some
  - Not critical to miss some deadlines
  - “Best effort” is sufficient
- Combinations
  - Shuttle has both depending on phase
    - Ascent Phase
    - On-Orbit Phase



Timing Analysis Goal

- **Reliably meeting deadline requirements**
  - Guaranteeing deadlines are met
  - Guaranteeing what happens if deadlines are missed
- *Executing with utmost speed is not the point*
- Contrast with goal of concurrent programming
  - Throughput



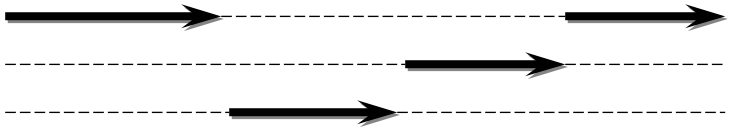
Timing Analysis Benefits

- Cheaper than manual analysis based on trial-and-error
- Worst-case timing behaviour calculated off-line
- Worst-case memory usage calculated off-line
- Amenable to “what-if” scenario evaluations without having to construct the system
- Provides evidence in support of certification

"Finite Progress Assumption"

- Process execution interleaves with other processes
- Designer cannot assume that a given process is executing at any given point in time
- Designer can only assume that process are making progress at some finite rate

Threads of Control



Determinism

- The ability to exactly determine which processes will be executing at any given point in time
- "Nondeterminism" is obviously the lack of determinism
  - Implies that effects of execution are not repeatable
- Finite Progress Assumption says processes are inherently nondeterministic

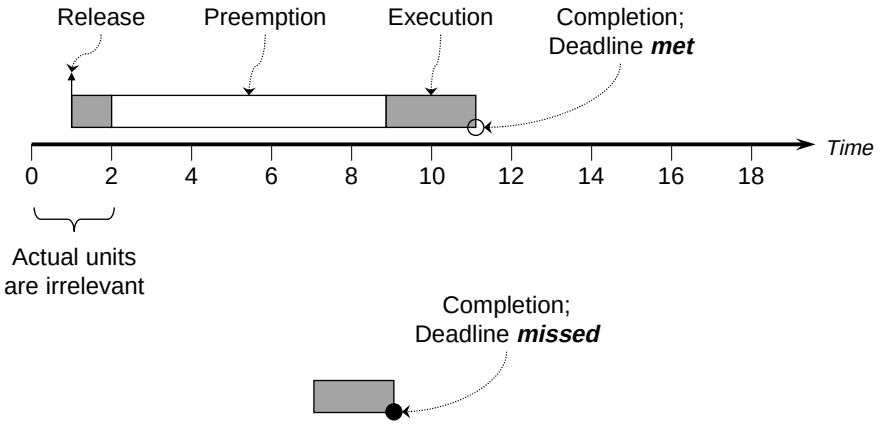
Schedulers

- Schedule resource usage
  - Multiple runnable processes implies contention
    - Processes contend for processors
  - Typically fewer processors than processes
- Remove nondeterminism from resource management
  - Finite Progress Assumption is not sufficient
  - Impossible to guarantee deadlines otherwise

Priority-Based Schedulers

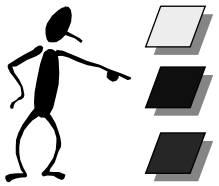
- Highest-priority runnable process is dispatched
- Invoked by state changes
  - Newly runnable process may have higher priority
- Preemptive Schedulers
  - Immediately switch to the higher priority process
- Non-preemptive Schedulers
  - Await completion of the currently executing process
- Advantages
  - Preemptive schedulers allow more responsive behavior
  - Non-preemptive schedulers impose less overhead
    - Potentially fewer context switches

Process Symbology



Tutorial Agenda

- Definitions
- ***Deadline Scheduling Theory***
  - Preference Scheduling
  - Deadline Scheduling
  - Concluding Remarks
- Support for Deadline Scheduling
- Low-Level Tasking Control
- User-Defined Schedulers

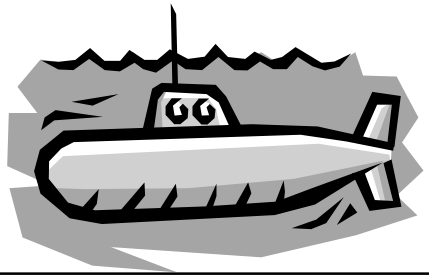


Preference Scheduling

- Designer assigns priority according to importance
  - More important threads are allocated more urgent priority
  - Termed “semantic importance”
- Scheduler always runs the highest-priority process
  - More urgent threads run in preference to less urgent threads
  - Thus the term “preference”
- Typical preemptive priority-based scheduling approach
  - Simple implementation

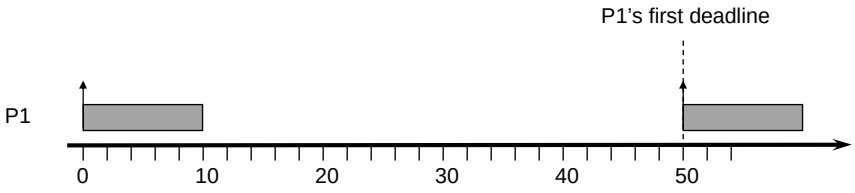
Preference Scheduling Inadequacy

- Can only guarantee deadline for most urgent process
  - Assuming deadline can be met at all
- Might not handle *sets* of *otherwise schedulable* tasks
  - Due to priority assignments based upon semantic importance
- Example: two tasks on a nuclear submarine
  - Process P1 is the nuclear reactor controller
  - Process P2 is a washing machine controller

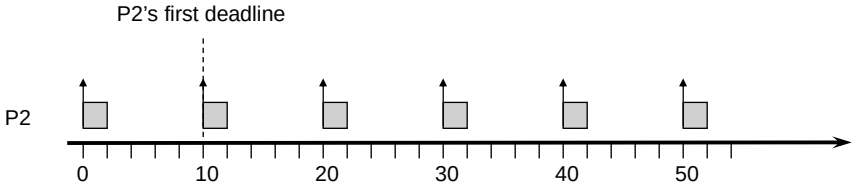


Preference Scheduling Example

- P1 has period of 50 & executes 10 time units per period

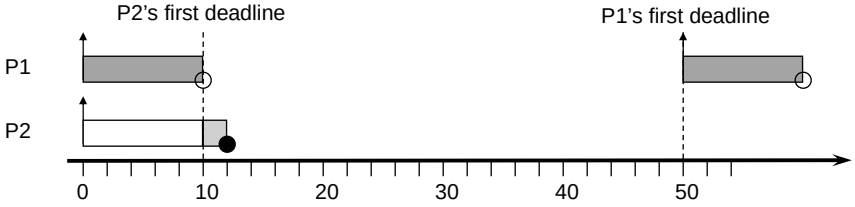


- P2 has period of 10 & executes 2 time units per period

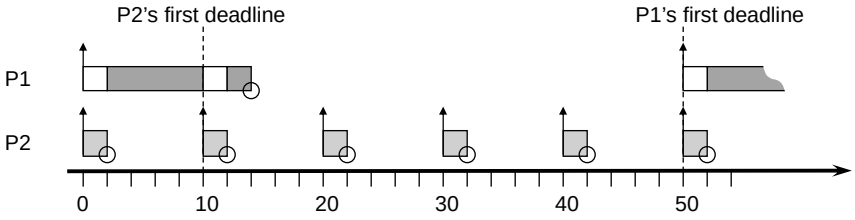


Preference Scheduling Example

- If P1 has higher priority P2 will miss its deadline



- Both would meet deadlines if P2 had higher priority

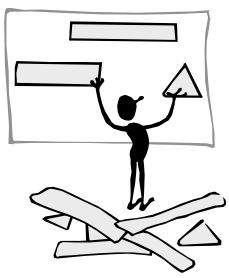


Preference Scheduling Summary

- Doesn't guarantee deadlines for entire process set
- Inadequate for deadline reliability

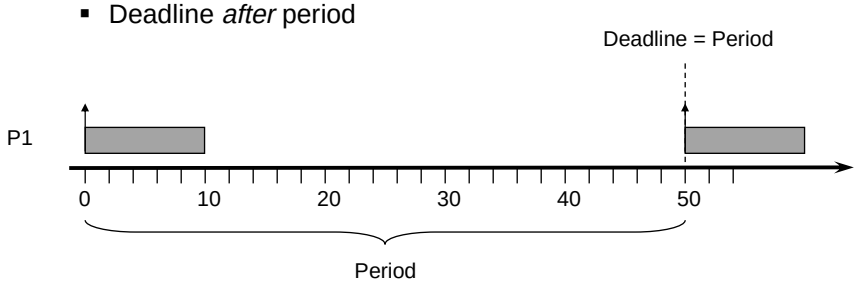
Tutorial Agenda

- Definitions
- ***Deadline Scheduling Theory***
  - Preference Scheduling
  - ***Deadline Scheduling***
  - Concluding Remarks
- Support for Deadline Scheduling
- Low-Level Tasking Control
- User-Defined Schedulers



Deadline Scheduling

- Maps individual deadlines to individual processes
  - Deadline *equal to* period
  - Deadline *before* period
  - Deadline *after* period



- Uses *deterministic* scheduling schemes

Scheduling Schemes

- Different schemes address different deadline mappings
- Each scheme defines how to:
  - Assign priorities to processes
  - Test for schedulability
- Two classes of priority assignment algorithms
  - "Fixed" if made prior to execution
  - "Dynamic" if made at run-time
- Analysis of schedulability is pass/fail

Schedulability Analysis

- Test to determine if *all* tasks will meet deadlines
  - Not just most semantically important task
- An analytical process
  - *Not trial-and-error*
  - Deadline reliability is determined prior to execution
- Currently supported for single processor targets
- Situations Beyond Scope
  - Multiple Processors
    - Becoming better understood
  - Multiple Programs
    - Feasible with some contortions
  - Distributed Systems
    - Policies based on determinism will not suffice

Optimal Scheduling Schemes

- Guarantee deadlines will be met if at all possible
  - If it can be done at all, an “optimal” scheme can do it
- Fixed-Priority Schemes
  - Rate Monotonic Scheduling (RMS)
    - Now usually termed Rate Monotonic *Analysis* (RMA)
  - Deadline Monotonic
- Dynamic-Priority Schemes
  - Earliest Deadline First
  - Least Slack Time
- Not all schemes are equally desirable in all situations

Processes Characteristics

☺ Periodic
- Time-triggered initiation per period
- Characterized by period and execution time
- Execution time can be average or worst-case

☹ Aperiodic
- Event-triggered initiation (i.e., random)
- May also have deadlines, relative to initiation
- *Cannot be analyzed without further information*

☺ Sporadic
- Aperiodic but with *known minimum duration between initiations*
  - Example: A sensor with a sample rate no greater than N
- Allows worst-case analysis for aperiodic processes

Schedulable Processes Set Models

- First Tier
  - No blocking
  - Periodic processes only
- Second Tier
  - Addition of sporadic processes
- Third Tier
  - Above plus blocking allowed

Tier 1 Model

- All processes are non-blocking
  - Run without synchronization or direct communication
  - “Competing” versus “cooperative” thread architectures
- Fixed in number
  - Referred to as N
- Strictly periodic
  - Minimum release time is referred to as T
- Fixed computation time per process
  - Referred to as C or WCET (worst case execution time)
- Deadline equals period
  - Must complete execution before next invocation
  - Referred to as D
  - Hence D = T

Rate Monotonic Analysis

- Optimal when deadline equals period
- Priorities are fixed and unique per process
- Priorities assigned according to process periods
  - Shorter period => higher priority
- Priorities are not a function of semantic importance
  - Recall washing machine and nuclear reactor controller example
  - Semantic importance is a factor when deadlines are missed

Why Unique Priorities?

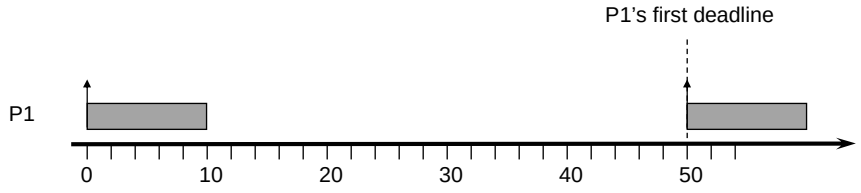
- Reduces blocking
  - Otherwise, worst-case is prior execution by all other threads!
- Result is better schedulability

Max Waiting Times Example

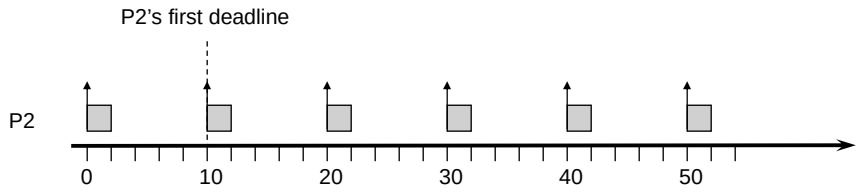
Task	Exec. Time	***Equal Priorities***	***Distinct Priorities***	Assigned Priority
T1	10	50	0	High
T2	20	40	10	Medium
T3	30	30	30	Low

RMA Submarine Example

- P1 has period of 50 & executes 10 time units per period

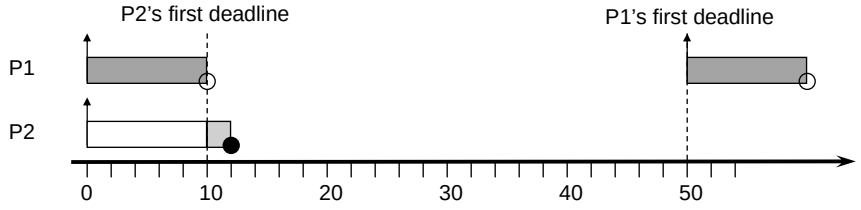


- P2 has period of 10 & executes 2 time units per period

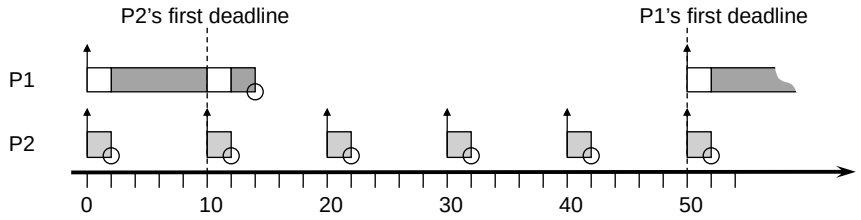


RMA Submarine Example

- If P1 has higher priority P2 will miss its deadline



- Both meet deadlines with RMA (P2 has higher priority)



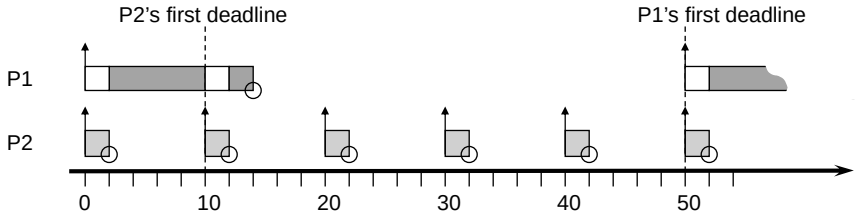
Utilization Test

- Schedulability test
  - Optimal scheme priority assignments
  - Priority-based preemptive scheduler
- A simple true/false condition
  - Left-hand side is the calculated utilization for the task set
  - Right-hand side is the utilization bound for that process set

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) < N(2^{1/N} - 1)$$

N is total number of processes
C_i is computation time for process i
T_i is period for process i

Submarine Analysis Example



$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) < N(2^{1/N} - 1)$$

$$\left(\frac{10}{50}+\frac{2}{10}\right) < 2(2^{1/2} - 1)$$

$$0.4 \quad < \quad 0.828$$

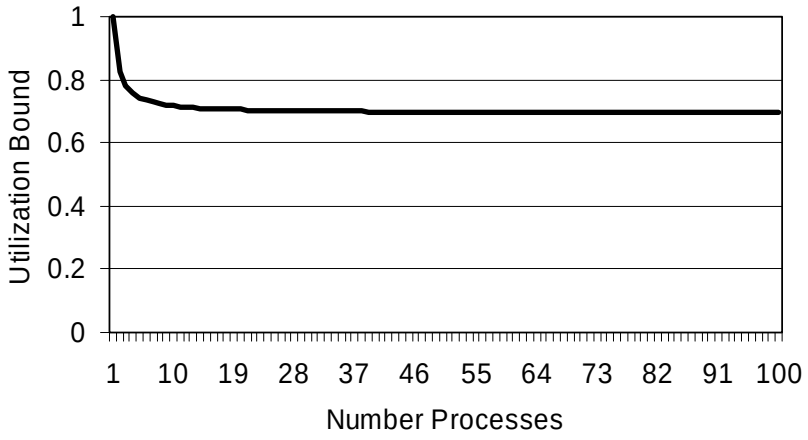
Schedulable

Role of Performance

- Poor performance increases values of C
- Larger values of C reduce schedulability
  - Perhaps beyond what is possible
- So performance *is* very important
- But performance provides no guarantees

Utilization Test Guarantee

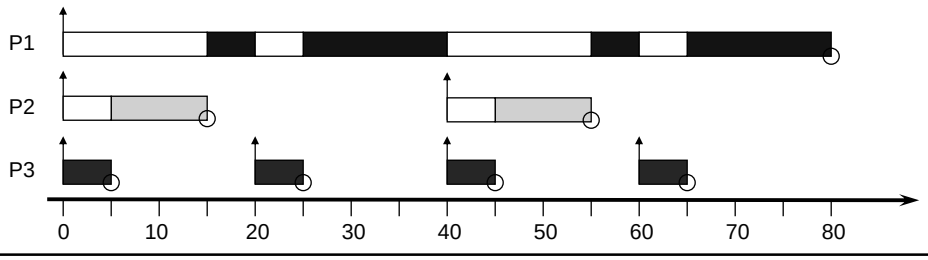
- Right hand side asymptotically approaches 0.693
- *Any* utilization below 69.3% will guarantee *all* deadlines



Limitations of Utilization Test

- Restricted applicability
- May give false negatives (only)
  - Example: schedulable at 100% utilization (versus 78%)

	C / *T*
P1 :	40/ 80
P2 :	10/ 40
P3 :	5 / 20



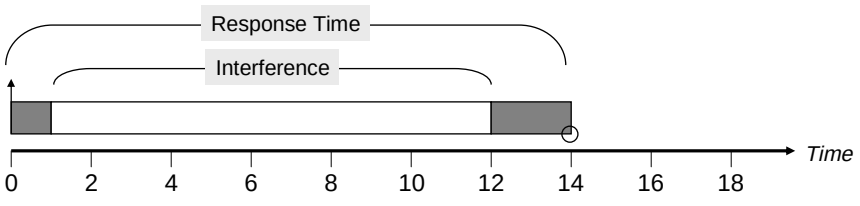
Response Time Analysis

- No false negatives
- Applicable to aperiodics and all deadline relationships
  - Basic form shown first for Tier 1 Model
- Approach
  - Compute response time for each process
    - Completion time - release time
  - Compare each response time to associated deadline

Response Time

- Response Time = Computation Time + Interference
  - Computation time is known
- Symbology for process i
  - Response Time R_i
  - Computation Time C_i
  - Interference Time I_i

$$R_i = C_i + I_i$$



Transient Overloads

- Temporary situation in which not all deadlines met
- Caused by use of average execution time for analysis
  - May provide better schedulability
  - Worst case may be so pessimistic that utilization suffers
- Scheduling must remain stable during overloads
  - *Semantically important* processes must still meet deadlines!

RMA & Overloads

- Remains stable
  - Lower-priority processes miss deadlines before others
  - Effects thus determined analytically
  - Why RMA is preferred over Earliest Deadline First
- Priority assignments may not match importance!
  - Assigned by period rather than semantic importance
  - Reactor controller might have lowest priority

Period Transformation

- Make important processes have shorter periods
  - Thus higher priority, last to miss deadlines
- Approach is to break execution time into pieces
  - Not physically breaking it up
  - Each piece thus has less execution time
  - Unlike other processes
    - Different pieces of same process do different things per release
- Two implementation choices
  - Insert a delay into source code, one per piece
  - Have RTS schedule as different pieces
    - Requires no change to source code

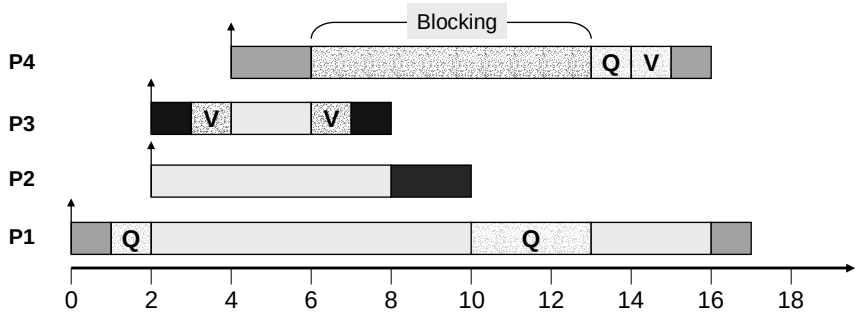
```
loop
   -- code for piece #1
   delay N;
   -- code for piece #2
   delay M;
end loop;
```

Tier 2 Model

- Addition of aperiodic & sporadic processes
  - No fixed period for some processes
- Earliest Deadline First is an optimal approach
  - ***But cannot predict which deadlines will be missed***
- RMA adaptation
  - Provide a periodic process to serve aperiodic processes
    - Allocated maximum time without perturbing scheme
  - Essentially polling
    - Polling is incompatible with nature of aperiodic events
    - Server may not be ready when events occur
    - Too many events for server to handle
  - Polling compromise : bandwidth-preserving algorithms

Tier 3 Model

- "Cooperating" processes must occasionally block
  - Communication
  - Synchronization
    - Condition synchronization
    - Mutual exclusion
- Example: mutual exclusion for resources Q and V



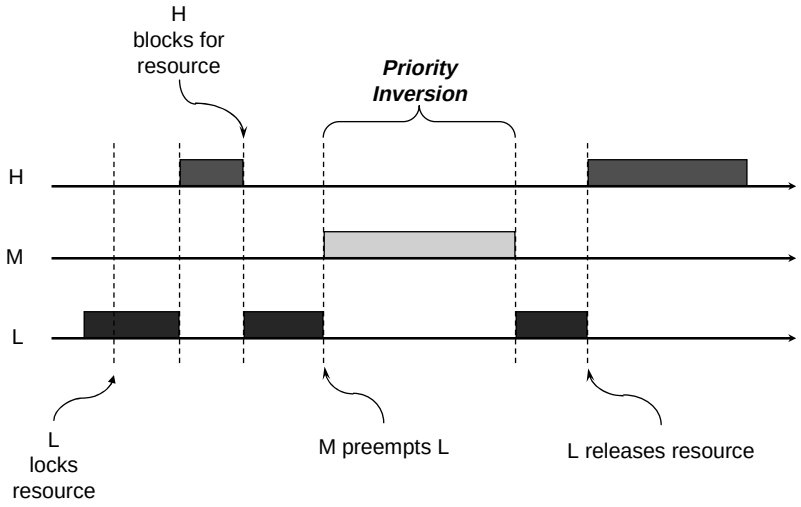
Blocking

- Response time includes worst-case blocking time
  - Hence analysis requires determination of blocking time
- "Priority Inversion" leads to unbounded blocking
  - Recall priorities are fixed by scheduling scheme
    - In order to achieve schedulability
    - Manually changing priority at run-time is not consistent
  - *A higher-priority task can be blocked by lower-priority tasks an unbounded number of times*
- Schedulability with unbounded blocking is intractable
  - Thus need a way to bound the blocking

Priority Inversion Example

- Given
  - High priority process "H"
  - Medium priority process "M"
  - Low priority process "L"
- Scenario
  - Assume L has mutually exclusive access to resource
    - Protected object, semaphore, rendezvous, etc.
  - H then tries to access resource, and is blocked
    - This blocking is correct
  - M then becomes eligible, and is not accessing resource
  - M preempts L, since higher priority than L
  - Thus, M is preempting H unproductively

Priority Inversion Example



Mars Pathfinder Example

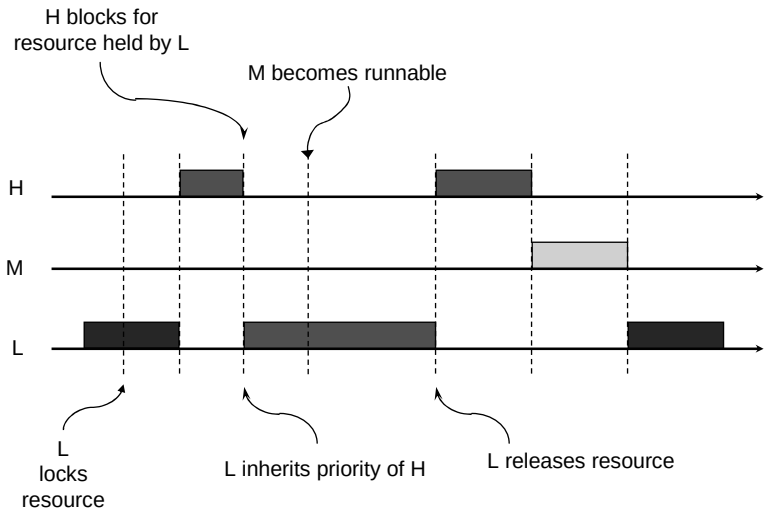
- A critical task missed a deadline under heavy load
  - While in-flight towards Mars
- Cause was priority inversion
- System reset in response
  - Did not cause overall project failure
- Ground team enabled “priority inheritance” in-flight
  - System then executed correctly
- Reference
  - http://redwood.snu.ac.kr/~sshong/courses/99-fall-rtsys/authoritative_account.html

View Pictures

Priority Inheritance

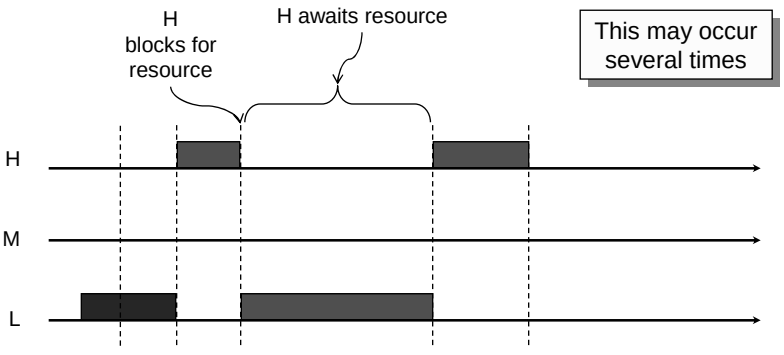
- Prevents priority inversions caused by fixed priorities
- *RTS dynamically changes blocking process’s priority*
  - Not programmer
- When H is blocked by L, L's priority is raised to that of H
  - Thus L will execute in preference to M
- Provides a (rather high) upper bound for worst-case
  - Number of resources to be locked
  - But schedulability can be determined
- Transitive inheritance implies high overhead
  - T1 waits for T2, which is waiting for T3, which is waiting ...
  - Can be expensive if fully implemented

Priority Inheritance Example



Priority Inheritance Bounds

- Higher-priority task might be blocked for *each* resource
  - Might need to lock several resources per activity
- Transient blocking may occur as well
- Result is pessimistic schedulability analysis

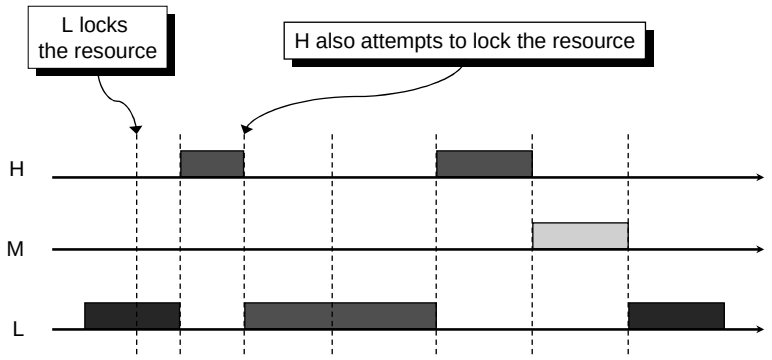


Ceiling Priority Protocols

- Reduce blocking by lower priority processes that share resources with higher priority processes
- Approach is to prioritize *resources* too
  - Synchronization mechanisms, i.e. things causing blocking
- All resources have a “ceiling priority”
  - Maximum of processes that share the resource
- Client processes inherit the ceiling priority
- Only the run-time system changes priorities
  - Not by source code
- Various protocols are available

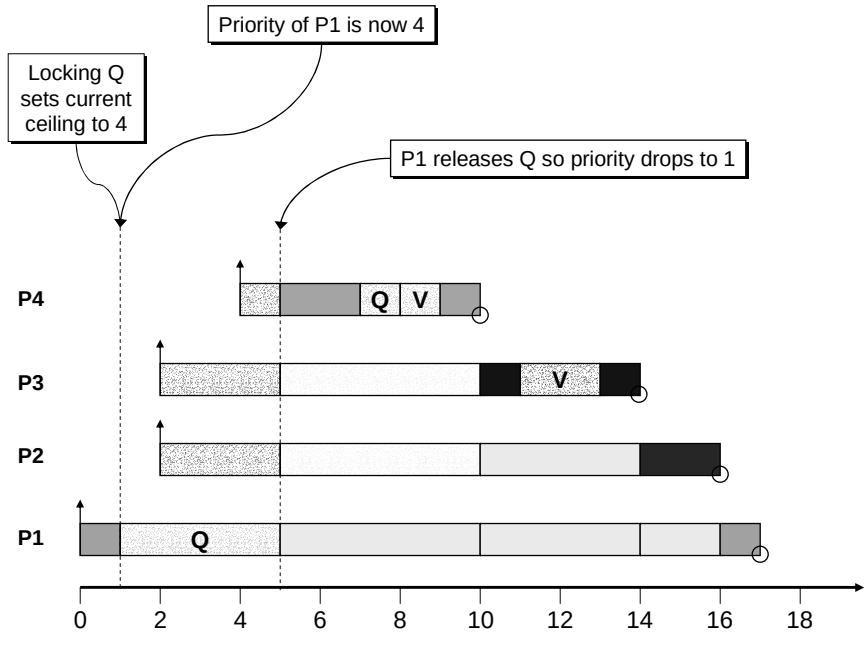
Assigning Resource Priorities

- *The urgency of a resource is at least that of any process that might be waiting for it*
- Thus resource priority should be the max of its clients
  - Both L and H are clients of the resource so priority is that of H



Immediate Ceiling Priority Protocol

- Each process has a static priority as usual
  - Assigned by some optimal scheme
- Each resource has a static “ceiling” priority
  - Max of all clients that access it (attempt to lock/unlock it)
- Each process has a dynamic priority
  - Max of static priority and *ceiling priorities inherited from any resources it has locked*



Ceiling Protocols' Benefits

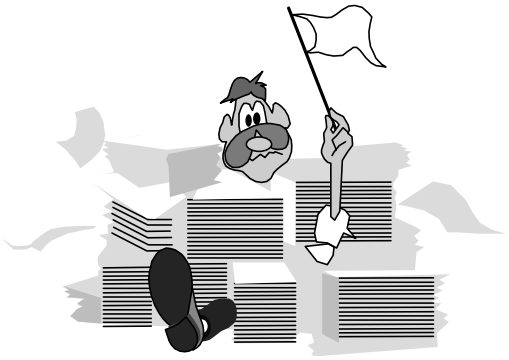
- Higher priority processes are blocked at most once per iteration by lower priority processes
  - Even though locking multiple resources may be required
- Deadlocks are prevented
  - By prioritization of resources integrated with scheduler
  - Major benefit of FIFO is prevention of deadlocks
    - FIFO is incompatible with deterministic scheduling
- Transient blocking is prevented
- Locking is provided automatically by priorities
  - Processes are blocked if they are not allowed to lock resource
  - Extremely efficient
  - Ensures mutual exclusion without actual lock mechanism
    - Only true on uniprocessors

Tutorial Agenda

- Definitions
- ***Deadline Scheduling Theory***
  - Preference Scheduling
  - Deadline Scheduling
  - ***Concluding Remarks***
- Support for Deadline Scheduling
- Low-Level Tasking Control
- User-Defined Schedulers

Topics Not Covered

- Device Drivers
- System Overheads
- Mode Changes' Impact on Ceiling Priority Assignments
- Cooperative Scheduling
- Release Jitter
- Arbitrary Deadlines



Further Reading

- Real-Time Systems and Programming Languages
  - Alan Burns & Andy Wellings
  - Addison-Wesley, 2001
  - 3rd edition, ISBN 0-201-72988-1
  - Errata: http://www.cs.york.ac.uk/rts/RTSbookThirdEdition/errata.html
- Meeting Deadlines in Hard Real-Time Systems
  - Loïc Briand & Dan Roy
  - IEEE Computer Society Press, 1999
  - ISBN 0-8186-7406-7
- A Practitioner's Handbook for Real-Time Analysis
  - Mark Klein et al
  - Kluwer Academic Publishers, 1993
  - ISBN 0-7923-9361-9

Tutorial Agenda

- Definitions
- Deadline Scheduling Theory
- ***Support for Deadline Scheduling***
  - Priority Subtypes
  - Task Priorities
  - Priority Scheduling
  - Priority Ceiling Locking
  - Entry Queuing Policies
  - Preemptive Abort
- Low-Level Tasking Control
- User-Defined Schedulers

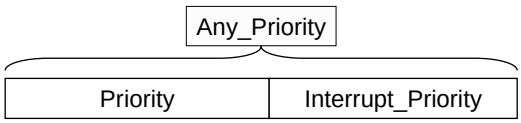
Real-Time Annex

- Core leaves some tasking semantics loosely defined
  - No priority semantics
  - No scheduling policy
- Advantages
  - No burden for non-real-time vendors
  - Doesn't preclude multiprocessors
- Non-determinism precludes schedulability analysis
- *Real-Time Annex defines appropriate environment*
  - Priority mechanism and semantics
  - Deterministic scheduling policy
  - Specific semantics for various constructs

Priority Subtypes

- Declared in package System
- Have implementation-defined ranges

```
subtype Any_Priority      is Integer      range implementation-defined;
subtype Priority          is Any_Priority range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last;
```



Task Priorities

- Base Priority
  - The intrinsic priority of the task
  - Set by programmer
    - Priority and Interrupt_Priority pragmas
    - Call to procedure Dynamic_Priorities.Set_Priority
    - By default value
- Active Priority
  - Dynamic priority used for queuing and dispatching
  - Max of base priority and inherited priority at any instant
  - Manipulated by run-time system

Pragma Priority

- Parameter specifies base priority for tasks
  - Per task type (as for Ada 83)
  - Per task object
- Parameter values need not be static

```
task type T is
   pragma Priority( N );
   ...
end T;
```

```
task type T( Urgency : System.Priority ) is
   pragma Priority( Urgency );
   ...
end T;

Worker1 : T( Urgency => 10 );
Worker2 : T( Urgency => 7 );
```

Default Task Priority

- Constant defined in package System
  - Equal to one-half non-interrupt priority

```
Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

- Setting individual priorities is meaningful
  - Need not set priority for *every* task if setting *any* task's priority
- Not needed with deterministic scheduling schemes
  - Priority assigned for *every* task

Scheduling Policy

- Predefined policy implements deterministic scheduling
  - Preemptive, priority-based scheduling
  - Priority Inheritance
  - Immediate Ceiling Priority Protocol
- Implementations may define additional policies
- Controlled by a pragma
  - pragma Task_Dispatching_Policy(policy_identifier)
  - Policy_identifiers select policies
  - Applies to entire partition
- Undefined unless pragma appears

FIFO_Within_Priorities

- Predefined policy identifier for Task_Dispatching_Policy
- Selects preemptive priority-based scheduling
  - Highest active priority task is chosen to execute
  - Tasks are queued behind other equal-priority tasks
- Requires priority inheritance and priority ceiling protocol
  - Specified via another pragma
  - No executable created if both pragmas not specified
    - A post-compilation rule

Priority Inheritance

- Various activities cause one task to block another
  - A task being activated inherits priority of activator
  - Rendezvous accepting task inherits caller priority if higher
  - During a protected operation a task inherits PO's priority
- Vendor must document max RTSE priority inversion

Resource Locking Policy

- Specified via pragma
  - pragma Locking_Policy(*policy*_identifier)
- Required for protected operations to have priority
- Specifies interactions between task and PO priorities
  - Relationships between task priorities and ceiling priorities
  - State of a task when it executes a protected action
  - How calling a protected operation affects a task's active priority
- Predefined policy specifies ICPP
- Implementations may define other policies

Ceiling_Locking

- Predefined policy identifier for pragma Locking_Policy
- Every PO has a ceiling priority specified
  - Via pragmas Priority or Interrupt_Priority
  - If no such pragma appears ceiling is Priority'Last
- *Tasks in protected actions inherit the ceiling of the PO*
- *Caller priorities must not be greater than PO's ceiling*
  - Follows from priority ceiling protocol concepts
  - Design error otherwise
  - Program_Error is raised if violated
- Assumes servers are implemented as protected objects
  - Calls to entries of server tasks are not checked

Integrated Interrupt Handling

- One overall integrated priority range
  - Subtypes for interrupt and non-interrupt levels
  - No overlap between subtype ranges
- One integrated priority semantics
  - Protected objects contain interrupt handlers
  - Protected objects can have priority
  - Tasks and protected objects work under same priority rules
- Protected Objects have interrupt priority specified
  - Parameter to pragma Interrupt_Priority
    - Parameter specifies specific priority
  - No parameter defaults to highest interrupt priority value

Prioritized Activities

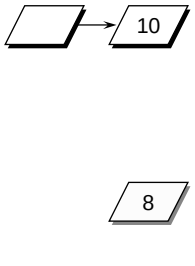
- Scheduling makes processor management deterministic
  - Highest-priority process gets processor
- Other activities must also be deterministic
  - Overall task management when semantics allow choices
  - Communication
  - Hardware bus control
  - et cetera
- Ada supports deterministic task management
  - Entry queuing
  - Task abort statements
  - Dynamic dispatching for OOP

Entry Queuing Policies

- Allow control over how entry queues are managed
  - Order in which calling tasks are queued
  - Choice among open alternatives in select statements
  - Choice among queued entry callers in a protected object when multiple barrier conditions are true
- Controlled by a pragma
  - pragma Queuing_Policy(policy_identifier)
- The policy_identifier is one of:
  - FIFO_Queuing
  - Priority_Queuing
  - An implementation-defined policy_identifier
- FIFO_Queuing is the default policy

Using Priorities of Callers

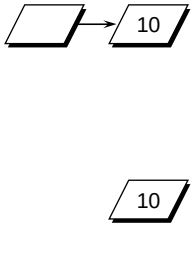
- Select statements with multiple open alternatives use the priorities of the callers to choose the alternative to accept




```
select
   accept Service1 do
      ...
   end Service1;
or
   accept Service2 do
      ...
   end Service2;
or
   accept Service3 do
      ...
   end Service3;
end select;
```

Using Textual Order

- Select statements with multiple open alternatives with equal priority callers use the textual order of the alternatives to choose the alternative to accept




```
select
   accept Most_Urgent do
      ...
   end Most_Urgent;
or
   accept Urgent do
      ...
   end Urgent;
or
   accept Least_Urgent do
      ...
   end Least_Urgent;
end select;
```


Using Textual Order for Delays

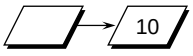
- If two or more delay alternatives have expired, the textual order is used to choose the sequence of statements to execute

```
select
   accept Call do
      ...
   end Call;
or
   delay M;
   -- more urgent statements ...
or
   delay N;
   -- less urgent statements ...
end select;
```

Multiple delays may have expired by the time this task is once again eligible for execution

Caller Priorities with Barriers

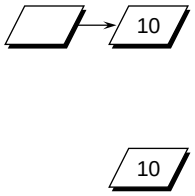
- If two or more of a protected object's barrier conditions become true with callers waiting, the call with the highest priority is chosen




```
protected body Bounded_Buffer is

   entry Put( Input : in Datum ) when Count < Size is
   begin
      ...
   end Put;

   entry Get( Output : out Datum ) when Count > 0 is
   begin
      ...
   end Get;

end Bounded_Buffer;
```


Textual Order for Barriers

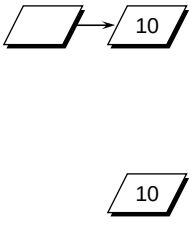
- If caller priorities are equal, entry textual order is used




```
protected body Bounded_Buffer is

   entry Put( Input : in Datum ) when Count < Size is
   begin
      ...
   end Put;

   entry Get( Output : out Datum ) when Count > 0 is
   begin
      ...
   end Get;

end Bounded_Buffer;
```

Using Family Member Index

- If one or more entry barrier conditions of an entry family are true and the callers have the same priority, then the call on the entry with the lowest family index is chosen



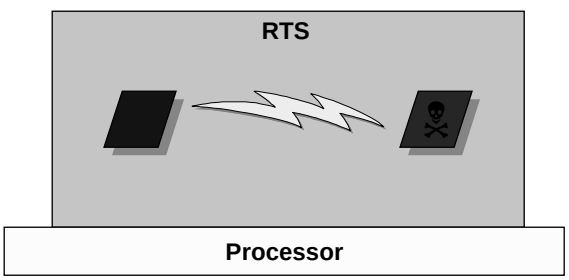

```
type Criticality is ( Critical, Important, Valued );

protected Manager is
   entry Grant( Criticality ) ( R : in out Resources );
end Manager;

entry Grant( Critical ) ( R : in out Resources ) when Critical-Ready is
   ...
end Grant;

entry Grant( Important ) ( R : in out Resources ) when Important-Ready is
   ...
end Grant;

entry Grant( Valued ) ( R : in out Resources ) when Valued-Ready is
   ...
end Grant;
```


Preemptive Abort

- Not required in Core
  - Semantics defined in RM 9.8
- Required by Real-Time Annex for uniprocessor systems
- Aborted construct must complete at the first point outside of an abort-deferred region
  - Protected actions
  - Waiting for an entry call to complete
  - Waiting for dependent tasks to terminate
  - Initialization, Finalization or assignment of controlled objects

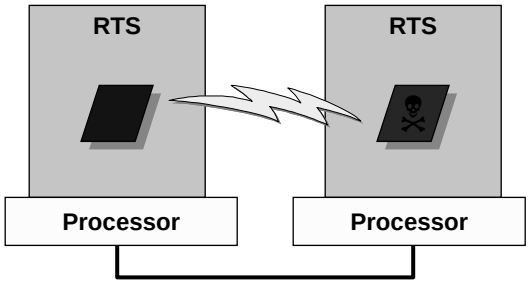
Preemptive Abort & Uniprocessors

- Reasonable
  - Run-time system has immediate opportunity to kill target task
- Required documentation
  - Execution time of abort statement
  - Any conditions causing aborting task to be preempted



Preemptive Abort & Multiprocessors

- May be difficult
  - Target task may be occupying processor
- Required documentation
  - Conditions delaying effect beyond that of single CPU systems
  - Additional communication delays that affect the immediacy



Tutorial Agenda

- Definitions
- Deadline Scheduling Theory
- Support for Deadline Scheduling
- ***Low-Level Tasking Control***
  - Synchronous Task Control
  - Asynchronous Task Control
- User-Defined Schedulers

Low-Level Tasking Control

- Facilities available for extreme efficiency situations
  - When protected objects are still too slow
  - Note that any similar mechanism will also be too slow
    - Such as semaphores
- Harder to maintain than higher-level abstractions
- More error-prone than higher-level abstractions
- Use by user-defined schedulers etc. is more reasonable
  - Thus not directly used by applications
  - Not spread throughout source
- Two predefined facilities
  - Synchronous Task Control
  - Asynchronous Task Control

Synchronous Task Control

- Provides primitives to construct synchronization mechanisms and two-stage suspend operations
- Package exports a Suspension_Object type
  - Values are “true” and “false”, initially “false”
  - Such objects are awaited by one task but set by other tasks

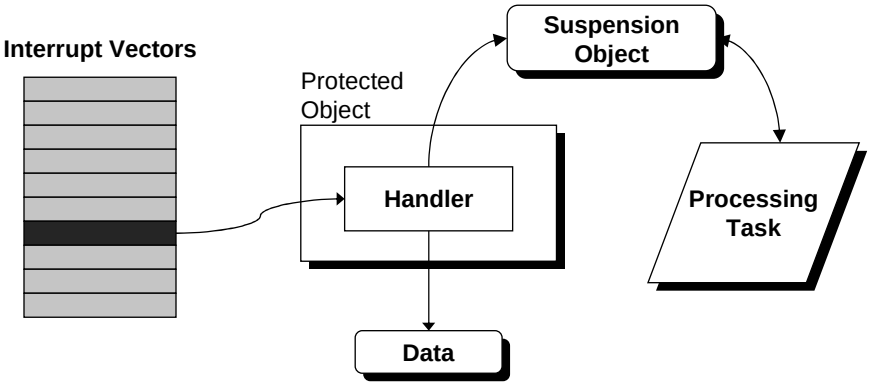
```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True( S : in out Suspension_Object );
   procedure Set_False( S : in out Suspension_Object );
   procedure Suspend_Until_True( S : in out Suspension_Object );
   function Current_State( S : Suspension_Object ) return Boolean;
private
   ...
end Ada.Synchronous_Task_Control;
```



```
with Ada.Synchronous_Task_Control;
package body Demo is

   package STC renames Ada.Synchronous_Task_Control;
   ...

   Items_Available : STC.Suspension_Object;

   task Consumer;
   task Producer;

   task body Consumer is
   begin
      ...
      STC.Suspend_Until_True( Items_Available );
      ...
   end Consumer;

   task body Producer is
   begin
      ...
      STC.Set_True( Items_Available );
      ...
   end Producer;

end Demo;
```

Synchronous Task Control

- Useful with interrupt handlers when speed is paramount
  - Task calls Suspend_Until_True
  - Handler calls Set_True



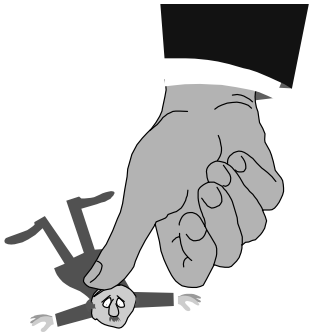
Asynchronous Task Control

- Tasks can unilaterally suspend and resume other tasks
  - Type-independent task identification facility shown later

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
   procedure Hold( T : in Ada.Task_Identification.Task_Id );
   procedure Continue( T : in Ada.Task_Identification.Task_Id );
   function Is_Held( T : Ada.Task_Identification.Task_Id ) return Boolean;
private
   ...
end Ada.Asynchronous_Task_Control;
```

Asynchronous Task Control

- Held tasks have lower priority than conceptual idle task
  - Thus held tasks cannot be dispatched
- Error conditions
  - Tasking_Error is raised if the operand task is terminated
  - Program_Error raised if Null_Task_Id used as actual parameter



Asynchronous Task Control

- Being “held” is deferred when inheriting priorities
  - Executing protected actions
  - While inside a rendezvous
  - During activation
- Thus not absolutely asynchronous
  - Extreme performance justification also means those other constructs probably aren’t used anyway...
- Note deferral represents potential distributed cost
  - If implementation checks after each point to see if now “held”
- A conforming implementation need not implement it
  - If “infeasible” on the intended target

Tutorial Agenda

- Definitions
- Deadline Scheduling Theory
- Support for Deadline Scheduling
- Low-Level Tasking Control
- ***User-Defined Schedulers***
  - Systems Programming Annex Facilities
    - Type-Independent Task Identification
    - Per-Task Data
  - Real-Time Systems Annex Facilities
    - Dynamic Priorities
    - High Resolution Clock
    - Delay Accuracy
  - Periodic Scheduler example

Type-Independent Task Identification

- A means of identifying tasks independent of type
  - Generalized abstractions no longer need type visibility
  - Servers can identify callers even though model is asymmetric
- Support is provided by a package and two attributes

```
package Ada.Task_Identification is
   type Task_Id is private;
   function "="( Left, Right : Task_ID ) return Boolean;
   function Current_Task return Task_ID;
   procedure Abort_Task( T : in out Task_ID );
   ...
private
   ...
end Ada.Task_Identification;
```

  - T'Identity returns the task_id of task T
  - E'Caller returns the task_id of the current caller of entry E
    - Not to be used outside an entry body or accept statement

Type-Independent Task Identification

- Tasking model is asymmetric
  - Caller specifies name of task or protected object to call
  - Called unit doesn't know the identities of callers
  - Similar to the public telephone system
  - Necessary for flexibility and reuse of called units
- E'Caller is similar to the "caller-id" telephone service

```
entry Grant( R : in out Resource ) when True is
begin
   if Previous_Caller /= Grant'Caller then
      Previous_Caller := Grant'Caller;
      ...
   end if;
end Grant;
```


Caller Id Example

```
with Ada.Task_Identification;
use  Ada.Task_Identification;

package Secure_Resource_Allocation is

   protected Controller is
      entry Allocate;
      procedure Free;
   private
      Allocated         : Boolean := False;
      Current_Owner : Task_Id := Null_Task_Id;
   end Controller;

   Not_Allocated : exception;
      -- raised by procedure Free

end Secure_Resource_Allocation;
```

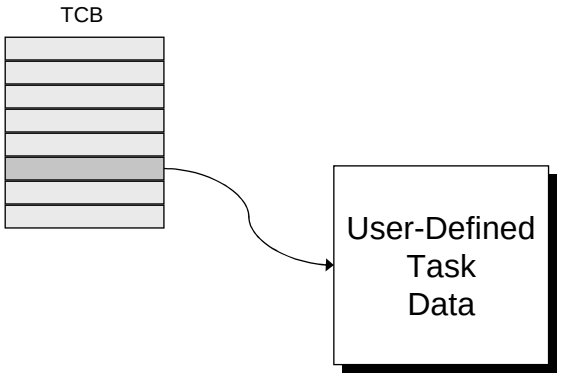
```
package body Secure_Resource_Allocation is


   protected body Controller is

      entry Allocate when not Allocated is
      begin
         Allocated := True;
         Current_Owner := Allocate'Caller;
      end Allocate;

      procedure Free is
      begin
         if Current_Task /= Current_Owner then
            raise Not_Allocated;
         else
            Allocated := False;
            Current_Owner := Null_Task_Id;
         end if;
      end Free;

   end Controller;


end Secure_Resource_Allocation;
```


Type-Independent Task Identification

- Bounded error to call Current_Task in entry bodies
  - Implementation approach may be to let last task inside do all
- Bounded error to call Current_Task in interrupt handlers
  - Implementation need not take time to store id of interrupted task
  - Implementation might directly vector to the the handler
- Current_Task can indicate the environment task
- E'Caller can indicate the environment task
  - That is, when the main subprogram calls an entry

Per-Task Data

- Provides system-level access to task representation
- Users can essentially add to the Task Control Block
  - For example, debugging data or any other "system-level" data



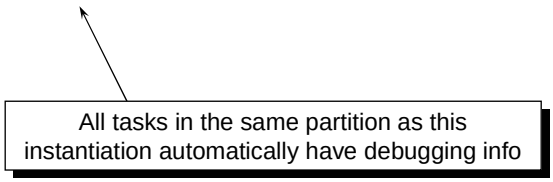
Per-Task Data Model

- Model based on elaboration of an instance of a generic
  - Applies to all tasks in the partition containing the instantiation
  - Application tasks do not declare variables directly

```
with Ada.Task_Identification;  use  Ada.Task_Identification;

generic
   type Attribute is private;
   Initial_Value : in Attribute;
package Ada.Task_Attributes is
   type Attribute_Handle is access all Attribute;
   function Value( T : Task_Id := Current_Task ) return Attribute;
   function Reference( T : Task_Id := Current_Task ) return Attribute_Handle;
   procedure Set_Value(    Val : in Attribute;  T : in Task_Id := Current_Task );
   procedure Reinitialize( T : Task_Id := Current_Task );
end Ada.Task_Attributes;
```

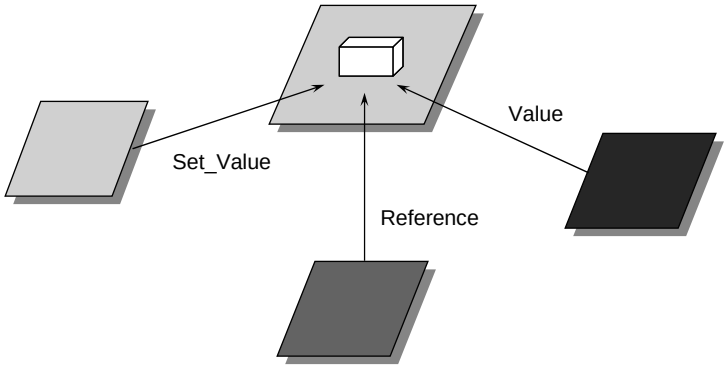
Per-Task Data Model Example

```
type Debugging_Info is
   record
      ...
   end record;

Default : Debugging_Info := ( ... );

package Debugging_Attributes is
   new Ada.Task_Attributes( Debugging_Info, Default );
```



Per-Task Data Atomicity

- Calls to exported routines are atomic
- Getting access value is atomic, not dereferencing it
  - Function Reference returns access value designating attribute



Dynamic Priorities

- A task's base priority may be set or queried at run-time
  - Current task is the default parameter
  - Program_Error is raised if Null_Task_Id is passed
  - Execution is erroneous is the task no longer exists

```
with System;
with Ada.Task_Identification;

package Ada.Dynamic_Priorities is

   procedure Set_Priority( Priority : in  System.Any_Priority;
                           T        : in  Ada.Task_Identification.Task_Id :=
                                          Ada.Task_Identification.Current_Task );

   function Get_Priority( T : in  Ada.Task_Identification.Task_Id :=
                                  Ada.Task_Identification.Current_Task )
      return System.Any_Priority;

private
   ...
end Ada.Dynamic_Priorities;
```


Dynamic Priorities Semantics

- Integrated with standard task dispatching
  - Always moves task to end of queue associated with new priority
- Integrated with priority queuing policy
  - If blocked, the task is moved to the new position in entry queue
- Integrated with priority ceiling model
  - Bounded error to raise waiting task's priority above PO's ceiling
  - Effects of error, if any, are localized to the task calling the PO
    - Not the task that raised the priority
    - Rationale: the calling task would have gotten the error if the priority had been too high when the call was made

High-Resolution Clock

- An alternative to Ada.Calendar and type Duration
  - No backward clock jumps due to time zones, leap-years, etc.
- Probably uses same time source as Calendar.Clock
  - But with better resolution

```
package Ada.Real_Time is

   type Time is private;
   ...
   type Time_Span is private;
   ...
   function Clock return Time;

   function "+"( Left : Time;  Right : Time_Span ) return Time_Span;
   function "-"( Left : Time; Right : Time ) return Time_Span;
   ...
private
   ...
end Ada.Real_Time;
```


High-Resolution Clock Requirements

- Type Time is a private type
  - Similar to Ada.Calendar.Time
  - Correspondence to a standard time reference is not required
    - Presumably corresponds to system initialization
- Implementation requirements for type Time
  - Must be able to represent at least 50 years
  - Must have a resolution of not more than 1 millisecond
- A 32-bit machine is assumed
  - A 32-bit counter for seconds and a 32-counter for ticks-per-second
- Full range & granularity not required
  - May not be possible on smaller machines
  - But must be documented

Type Time_Span

- Defined as a private type
  - Similar to type Duration but with higher resolution
- Modeled as an implementation-defined integer
  - Values are K*Time_Unit where K is of this hidden integer type
  - Time_Unit is smallest time representable by type Time
    - Expressed in seconds
    - Not greater than 20 microseconds
  - Not required to be implemented as an integer but probably will
- Time_Span_First is the smallest Time_Span value
  - No greater than -3600 seconds
- Time_Span_Last is the largest Time_Span value
  - No less than 3600 seconds
- Thus room for a two-microsecond resolution

Composing Time_Span Values

- Composition functions are provided
  - To_Time_Span takes a value of type Duration
    - Input thus limited by resolution of type Duration
  - Nanoseconds takes a value of type Integer
  - Microseconds takes a value of type Integer
  - Milliseconds takes a value of type Integer
- Decomposition function is also provided
  - To_Duration takes a Time_Span value

```
function Nanoseconds ( NS  : Integer ) return Time_Span;
function Microseconds ( US  : Integer ) return Time_Span;
function Milliseconds  ( MS  : Integer ) return Time_Span;
```

Delay Accuracy

- RT Annex specifies additional requirements
  - Relative delay statements
  - Absolute delay statements
  - Delay alternatives in select statements
- Guarantees for function Ada.Real_Time.Clock that:
  - Relative delays are always “forward” in time
  - Absolute delays are always “forward” in time
- Specifies that zero or negative delay values do not block
  - RM D.2.2(11) says will cause rescheduling
- When a delay appears in a selective accept alternative, the selection of an entry call to accept is attempted, regardless of the delay value

Periodic Scheduler Example

```
with Ada.Task_Identification;
use  Ada.Task_Identification;

with Ada.Real_Time;
use  Ada.Real_Time;

package Periodic_Scheduler is

   procedure Schedule( This_Task : in Task_Id;
                       Period    : in Time_Span;
                       Epoch     : in Time );

   procedure Await_Schedule;

   procedure Release_All;

   procedure Await_Release;

end Periodic_Scheduler;
```

Periodic Scheduler Example

```
task body Worker is
begin
   Periodic_Scheduler.Await_Release;
   loop
      Periodic_Scheduler.Await_Schedule;
      -- application-specific actions here...
   end loop;
end Worker;
```


Periodic Scheduler Example

```
with Ada.Task_Attributes;
with Blocking;

package body Periodic_Scheduler is

   Ready : Blocking.Event( Blocking.Down );

   type Scheduling is
      record
         Period          : Time_Span := Time_Span_Zero;
         Next_Scheduling : Time      := Time_Last;
      end record;

   Default : Scheduling; -- necessary for instantiation

   package Scheduling_Parameters is
      new Ada.Task_Attributes( Scheduling, Default );

   ...

end Periodic_Scheduler;
```

Periodic Scheduler Example

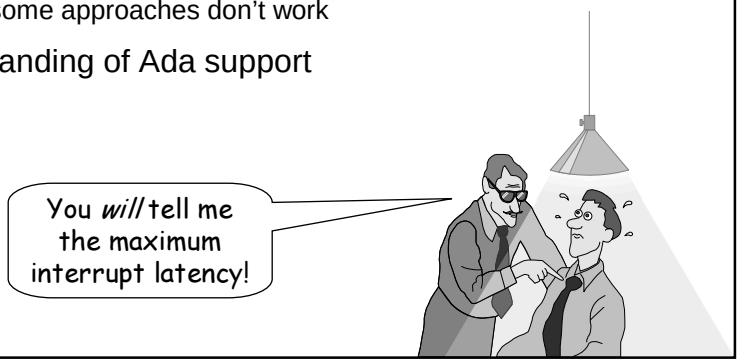
```
procedure Schedule( This_Task  : in Task_Id;
                    Period     : in Time_Span;
                    Epoch      : in Time ) is
begin
   Scheduling_Parameters.Set_Value( Scheduling'(Period,Epoch), This_Task );
end Schedule;


procedure Await_Schedule is
   This_Task  : Scheduling := Scheduling_Parameters.Value;
   Next_Time : Time;
begin
   Next_Time := This_Task.Period + This_Task.Next_Scheduling;
   Scheduling_Parameters.Set_Value( Scheduling'(This_Task.Period,Next_Time) );
   delay until Next_Time;
end Await_Schedule;
```


Periodic Scheduler Example

```
procedure Await_Release is
begin
   Ready.Wait( Blocking.Up );
end Await_Release;


procedure Release_All is
begin
   Ready.Set;
end Release;
```

Tutorial Goals Review

- Appreciation of modern approaches for *hard* real-time
  - Advantages
  - Risks
- Understanding of scheduling foundation theory
  - How it works
  - Why some approaches don't work
- Understanding of Ada support



Contacting Me

Feel free to send questions to me via email

Patrick Rogers

progers@classwide.com
(281)648-3165
http://www.classwide.com

Evaluations

- Your opinions are important!
- Please provide feedback on the form provided