“Real-Time Programming”

Prof. Dr. E. Plödereder

Chapter 1
Introduction

1 Introduction

1.1 What Is a Real-Time System?

Oxford Dictionary of Computing:

Any system in which the time at which output is produced is significant... The lag from input time to output time must be sufficiently small for acceptable timeliness.

Put differently:

Responses by a real-time system must meet deadlines imposed by the environment in which it operates.

The scale of deadlines depends heavily on the environment or application:

- meteorological forecast: measured in hours
- interactive systems: measured in seconds
- embedded systems: often measured in milli- and microseconds

An **embedded system** is a system in which the information processing component, and in particular the software, is part of a larger engineering system and is interfaced with physical devices monitored and controlled by the software.

Deadlines come in different flavors:

"hard deadline": the deadline must never be missed.

"firm deadline": a response after the deadline is useless.

"soft deadline": the value of the response progressively decreases after the deadline, or the deadline may be missed occasionally.

Correspondingly, we distinguish

- **hard real-time systems**: systems that have (among others) hard deadlines with significant requirements on timeliness
- **soft real-time systems**: systems without hard deadlines

Along a different dimension (consequences of failure), we also distinguish

- **safety-critical (real-time) systems:** systems whose failure to perform can cause disastrous results (loss of life, major economical or environmental damage)

This course is primarily about programming embedded real-time systems in any of these categories.

1.2 General Requirements on Embedded Real-Time Systems

Embedded real-time systems are or need to be

- often large and complex (e.g. avionics systems)
- extremely reliable (e.g. switching networks)
- extremely safe (e.g. medical equipment)
- verifiable ("rigorously prove that the code implements the design")
- validatable ("show that the software meets user requirements and needs")
- adaptable to changes in the usage environment
  - as part of system evolution and maintenance
  - as part of execution ("mode changes")
- often highly efficient to meet deadlines (e.g. in vehicle control)
- concurrently managing multiple devices (e.g. in process control)
- often (highly) constrained in their resources (memory size, CPU power)

The **last three characteristics** are quite specific to real-time systems; the others are, in principle, shared with software in general but must be taken much more seriously in real-time system applications.

These general requirements translate into the following “musts” for embedded systems:

- functionally correct
- well designed and engineered
- often capable of executing for indefinite time
- deterministic and predictable in results
- meeting the deadlines
- fault-tolerant and fail-safe where needed
- verifiable and validatable at acceptable cost

This course is about programming paradigms and language constructs that support these requirements and are particularly relevant to real-time systems.

For a more comprehensive treatment of process issues in the development of real-time systems and of overall design issues, see the course

“Software Engineering for Real-Time Systems”.

We assume that you are familiar with general principles of good software design and implementation (e.g. encapsulation, information hiding, modularization, interface specifications, type models, architectural models, etc., etc., ...).

1.3 Correctness of Algorithms - a Few Remarks

Is the following function correct ?

```
function factorial(N: Integer) return Integer is
    begin
        if N = 0   then return 0;
                   else return N * factorial(N-1);
        end if;
    end factorial;
```

This question is meaningless without additional information. In fact, the function is (almost) correct for the following specification: “Calculate 0 with as much computational effort as needed for the calculation of the factorial of N”.

What does the following procedure do ?

```
procedure Insert(L: in out List; E: Element);
```

The signature conveys some intuitive notion but significant details remain unknown:

- Where in the list will the element be inserted?
- What happens if E is already in the list?
- Can there be an overflow of the list?
- Can parallel processors call Insert on the same list without first synchronizing with each other?
- etc.

hence: ***Specification of functionality is extremely important***
(although it should be redundant to the implementation)

because ... ***verification of correctness of the implementation***
is impossible without a (hopefully) redundant specification.

Let's try again. Does the following function correctly compute the factorial of N?

```
function factorial(N: Integer) return Integer is
  -- computes factorial of N; 0!=1 and N!=N*(N-1)!
begin
    if N = 0   then return 1;
               else return N * factorial(N-1);
    end if;
end factorial;
```

Yes, it does but ....

... calling “factorial(-1)” causes disaster (infinite recursion, crashing the system). The implementation has a precondition: N must not be negative.

In some languages, preconditions on value ranges can be expressed by typing mechanisms, e.g., CARDINAL in Modula, NATURAL in Ada, causing invalid calls to be detected at compile-time or at run-time with associated recovery mechanisms. In general, though, programming languages provide not enough expressive power to describe all preconditions formally.

Nevertheless: ***Documenting the Preconditions is very important to avoid errors***

A subprogram specification should consist of

1. its signature (name, parameter types, result type)
2. description of functionality
3. description of preconditions
4. description of behavior when preconditions are violated

By comparing this information with the implementation, we can argue the correctness of the implementation.

Note: Proper encapsulations often make it easy to prove that the precondition is always satisfied.

Reprise... here is the (almost) correct factorial function in Ada:

```
function factorial(N: Natural) return Natural is
  -- computes factorial of N; 0!=1 and N!=N*(N-1)!
begin
    if N = 0   then return 1;
               else return N * factorial(N-1);
    end if;
end factorial;
```

(To be honest: There still is the overflow problem when the result of factorial exceeds the target architecture‘s integer range. More about this issue later in the course.)

1.4 The Role of Programming Languages

As a coding tool:

- to communicate instructions to a computer ("Mindset Assembler")
- ... more easily using better mnemonics, but knowing pretty well the actual instructions generated by a compiler and executed on the computer ("Mindset C")
- to program a fairly abstract machine with powerful operations, trusting a compiler to model this machine efficiently on an actual computer ("Modern Mindset")

As a specification and checking tool:

- to reflect and enforce program architecture in the code
- to discover program design and coding errors as early as possible
- to discover remaining errors at run-time rather than obtain incorrect and unpredictable results from the execution

As a communication medium:

- to communicate algorithms to other people, e.g. for validation or verification

Real-Time Programming

Chapter 2
Deterministic Program Behavior

2 Deterministic Program Behavior

Given the same input, a program should yield the same predictable result on each execution. This is called **deterministic program behavior**.

For real-time systems with deadlines, a program operating on the same input should have deterministic behavior also with respect to the amount of execution time and space needed.

Unfortunately, there are factors that interfere with the guarantee of deterministic behavior. They are caused by

- language semantics
- language implementation by compiler and run-time support
- hardware properties
- parallelism in the program

The software engineer for real-time and safety-critical systems should be aware of these factors and minimize or eliminate their effects.

2.1 Impact of Language Semantics

Language semantics are not completely deterministic.

Consider the following example:

```
declare
        X: integer;
        function f return integer is
        begin
           X := X + 1;
           return X;
        end f;

        function g return integer is
        begin
           X := X * 2;
           return X;
        end g;

     begin
        X := 1;
        PUT (f + X * g);
end;
```

What is the output of this program ?

For most languages, there are several legitimate results for this example, as these languages do not prescribe a left-to-right evaluation of expressions.

2.1.1 Language vs. Execution Semantics

Let S_{PL} : Program $\times$ Input $\rightarrow$ Output
be the semantics of the programming language PL

Let S_C be the respective mapping obtained by the execution of the program compiled by compiler C

Ideally: $\forall$ P, I : $S_{PL}(P, I) = A$ iff $S_C(P, I) = A$

Reality: $\exists$ P, I : $S_{PL}(P, I) = A_1 \vee A_2 \vee \ldots A_n$
admissible impl.-dependent alternatives

Consequence:

$S_{C_1}(P, I) = A_2$! nevertheless both

$S_{C_2}(P, I) = A_5$! compilers are correct

Correctness condition for compilers:

$$\forall\ P, I :\ S_C(P, I) = A_{P,I,C} \ \Rightarrow\ A_{P,I,C} \in S_{PL}(P, I)$$

Although execution of the compiled program will always yield the same result, the presence of implementation-dependent alternatives renders this result non-predictable from the source code; it may even change when the same program is recompiled.

Lesson:

Be aware of implementation-dependent language behavior.

Reference manuals for language standards should identify under which circumstances behavior is implementation-dependent. ("How-to"-books rarely do and trying it out on the compiler will not help at all.) Avoid these circumstances from ever arising in your programs by means of style restrictions that are easy to check.

For the previous example:

Draconian rule: *No side-effects allowed in functions.*

Moderate rule: *No function call with side-effects is allowed as part of a more complicated expression.*

Other “infamous” implementation dependencies:

- precise statement order
- certain permissions for optimizations in the language
- any execution time or space guarantees
- order of parameter passing
- representation of types
- accuracy of floating-point and fixed-point operations

These items may affect predictability on a given system, or only across target systems, i.e. portability. They may also affect program correctness.

2.2 Impact from Language Implementation and Run-Time Support

Predictability of worst-case time and space requirements of programs will play an important role in later chapters. Apart from global complexity estimates (e.g. constant, linear, quadratic, cubic complexity), time and space requirements cannot be accurately derived from the source code. It is the binary code that determines the actual time and space requirements.

Consequently, code generation strategies in the compiler and algorithms in the run-time support can have a major impact on satisfying time and space requirements.

Not all compilers and run-time systems are suitable for real-time programming. In fact, for hard real-time systems, few are.

Know Thy Compiler !

Among the facts to know are:

- memory management schemes, in particular any implicit use of heap memory (usually a disqualifying property for hard real-time systems)
- process management schemes (see later)
- worst-case time requirements of any run-time call
- nature and degree of (and control over) code optimization

2.3 Impact of Hardware Properties

Timing characteristics of the executing code will obviously depend on the speed of the hardware. However, the hardware may introduce considerable non-determinacy. Timing behavior may vary for each execution of the same code. Some influencing factors are:

- **data caches:** a given LOAD operation may take 1 or 2 cycles ("cache hit") or as many as 20-50 cycles ("cache miss"), depending on the state of the cache.

  Traditional remedies:
  - turning the cache off (or having none)
  - controlling the cache explicitly (if possible)
  - estimating on the basis of 100% cache misses

- **instruction pipeline and caches:** the time to retrieve the next instruction from memory after a branch may vary significantly.

- **“out-of-order” execution; superscalar architectures:** the hardware may execute instructions in other than the sequential order of the binary code. (These architectures have not significantly penetrated the hard real-time market yet.)

Program results may also be influenced by hardware

- most notably, by **floating-point accuracy**
- by sizes of numeric types

  (Luckily, this aspect is deterministic for any given architecture.)

Real-Time Programming

Chapter 3
Memory Management and Estimation

3 Memory Management and Estimation

No system should exhaust available memory. Reliability requirements on most embedded systems require proof that available memory will suffice even under the most demanding operating conditions. Also, many embedded systems must be capable of running reliably for indefinite duration.

A worst-case estimate of memory requirements is necessary and obviously needs to be below the amount of available memory.

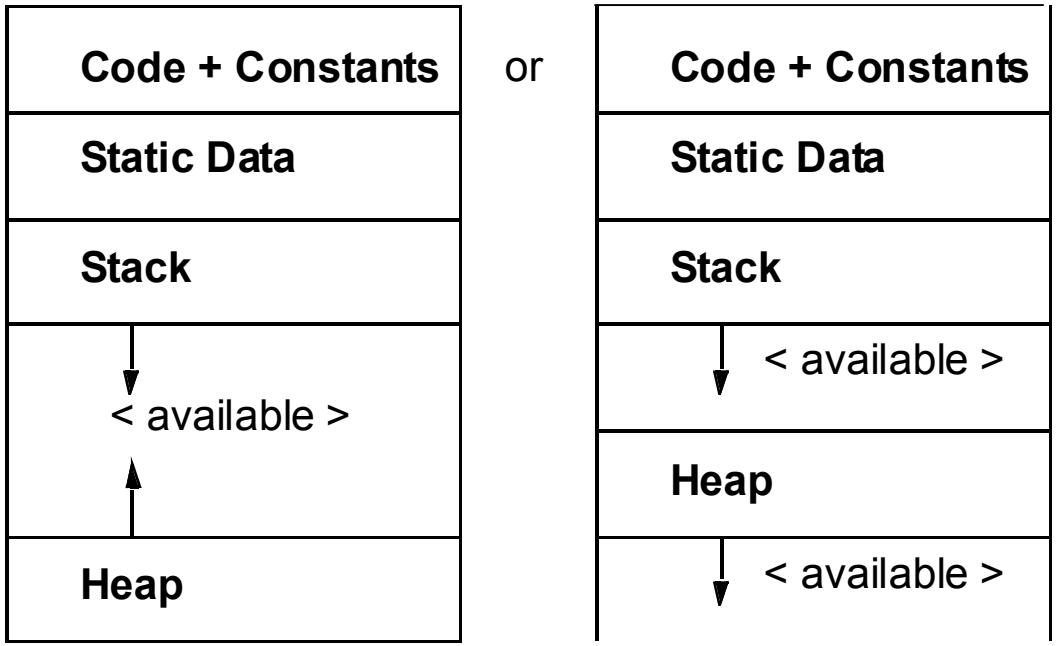
Many embedded systems operate under heavy limitations on available memory.

Programs generally need memory for

- code segments for routines and other control constructs
- constants
- static data (i.e. global variables)
- local and administrative data of routines
  (-> stack management)
- data whose lifetime is not limited to particular control constructs, i.e. routines (-> heap management)

For code segments and constants, ROM (read-only memory) suffices. On embedded systems, these are often stored in EPROMs. For the other data, RAM (random-access memory) is needed.

Generic storage management (without threads)



For code and for the static data, diagnostics by the linker/compiler establish the exact memory requirements (presuming that the size of all data types is known at link-time).

The stack is managed at run-time by (compiler-generated) code as part of the protocol of subprogram calls. For each call, an activation record of the subprogram is pushed onto the stack and popped again when the call returns. The maximum size of the stack needs to be determined.

The heap is managed by the run-time system or else by explicit user-provided code, reacting to allocation and deallocation requests. Its maximum size (and more) needs to be determined.

3.1 Determination of the Maximum Stack Size

In the following, we assume that the size of all data types is statically known. (If this is not the case, one needs to substitute worst-case sizes for exact sizes in the estimations.)

Diagnostics of the compiler (or else examination of the generated code) will tell the exact size of the activation record for a routine to be pushed on the stack for a call of the routine.

Based on a **call graph** of an application, we can estimate the maximum stack size.

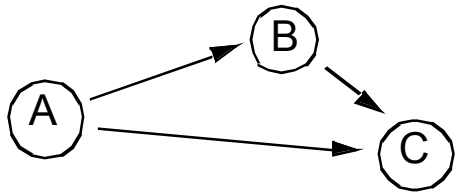
3.1.1 The Call Graph

The call graph represents the overall hierarchy of subprogram calls arising from the execution of a main program.

The nodes of the call graph represent the subprograms. The edges of the call graph connect each subprogram to the subprograms called by it.

Example:

```
proc A is begin ... call B;  ... call C; end A;
proc B is begin ... call C; ... end B;
proc C is begin ... end C;
```



Using the call graph, an estimate of the maximum stack size can be obtained fairly easily:

- attribute the nodes with the size of the activation record for the subprogram
- starting at the node for the main program, find the path through the graph that maximizes the sum of the sizes along its nodes; this is the maximum stack size

The size obtained in this way is either exact or an overestimation. The latter is the case if this particular call nesting can never arise due to the control logic in the program.

Complication: Cycles in the graph will make paths in the call graph infinite. (Cycles can only arise from recursive calls, either directly or indirectly.)

Solutions:

- forbid recursion (this restriction is often found in highly constrained embedded systems)
- establish a proven upper bound for the recursion depth, multiply the cost of the cycle by this upper bound and use the result in the calculations as the fixed cost of the cycle on the path.

Refinements of this scheme are possible, if necessary, by including the control flow graph of the subprograms in the path calculation (i.e. use branch predicate information).

3.2 Heap Management

While stack management ensures that memory for all local data of a subprogram is made available for each call and released after each call on the subprogram, sometimes there is a need to request memory during the execution of the program and not release it for reuse automatically with the end of the subprogram that issued the request.

This is particularly true for the **implementation of pointer or reference semantics** in programming languages where memory for the designated objects is explicitly allocated by operators like “new” or functions like “malloc”. In accordance with the language semantics, this memory cannot be allocated on the stack. It is allocated from the **heap**.

If a program needs to run reliably for an indefinite period of time, it is clear that it will eventually run out of memory if allocations are made continuously without ever deallocating the requested memory for reuse.

Heap management is concerned with the handling of allocations and the subsequent deallocation of heap memory that is no longer needed.

Heap management is provided as part of the run-time support of a language. **The "standard" implementations of heap management are extremely unsuitable and dangerous for hard real-time systems.**

Moreover, pointer semantics are a significant source of very dangerous coding errors. Many guidelines for hard real-time or safety-critical systems forbid the use of pointer types altogether.

3.2.1 Heap Organization

The heap is generally subdivided into blocks of memory. There are two competing approaches with quite different characteristics: the blocks are either of some fixed common size or of sizes that are different and dynamically determined. Both approaches follow the same basic principle:

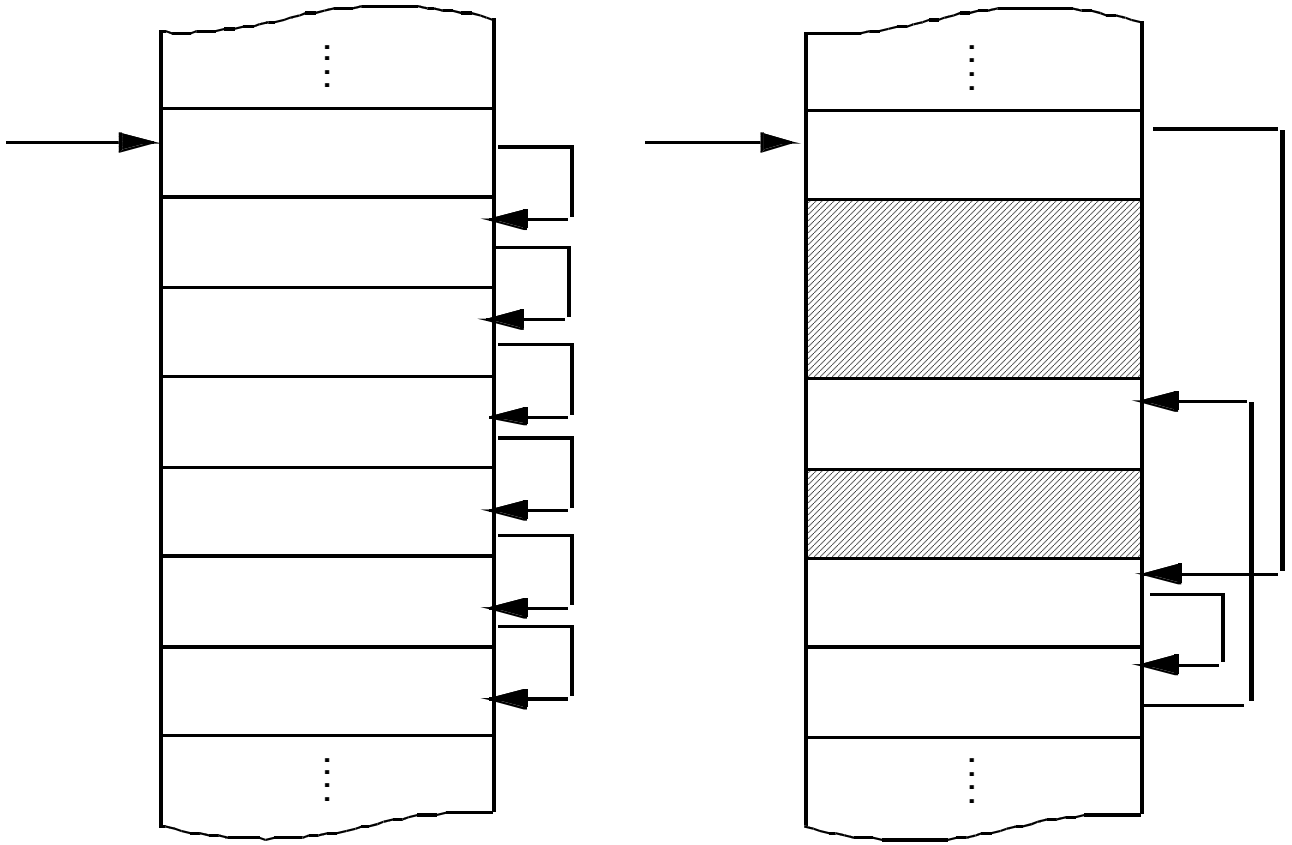
- available ("free") blocks are tracked by heap management either by threading them on a so-called "freelist" or by keeping a separate bitvector over the heap, distinguishing free and occupied blocks
- upon allocation, a block of suitable size is taken from the freelist (or bitmarked as occupied) and returned to the caller
- upon deallocation, the deallocated block is pushed on the freelist (or bitmarked as free)

- if the heap is full, an attempt might be made to obtain more memory from the operating system (if there is one). Alternatively, the heap can grow until it tries to overlap the stack or reaches the bounds of available memory resulting in a fatal error situation.

The following explanations are formulated in the terminology of a freelist. The alternative bitvector implementation of keeping track of free blocks is quite analogous. Significant differences will be pointed out specifically.

3.2.2 (Sub-)Heap Organization for Elements of Equal Size

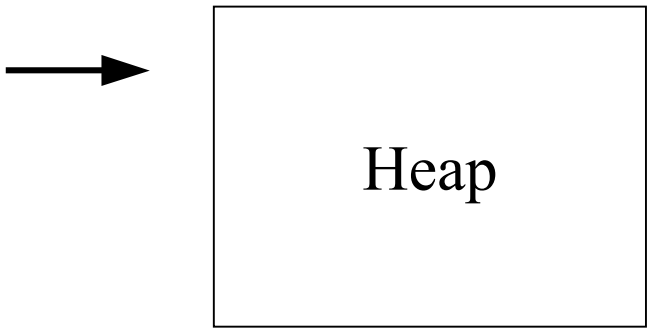
The heap gets subdivided into equally sized blocks.



- allocation is easy (“pop from freelist”).
- deallocation is easy, presuming one knows which elements are no longer needed (“push onto list”); note, however, the “holes” in the physical layout

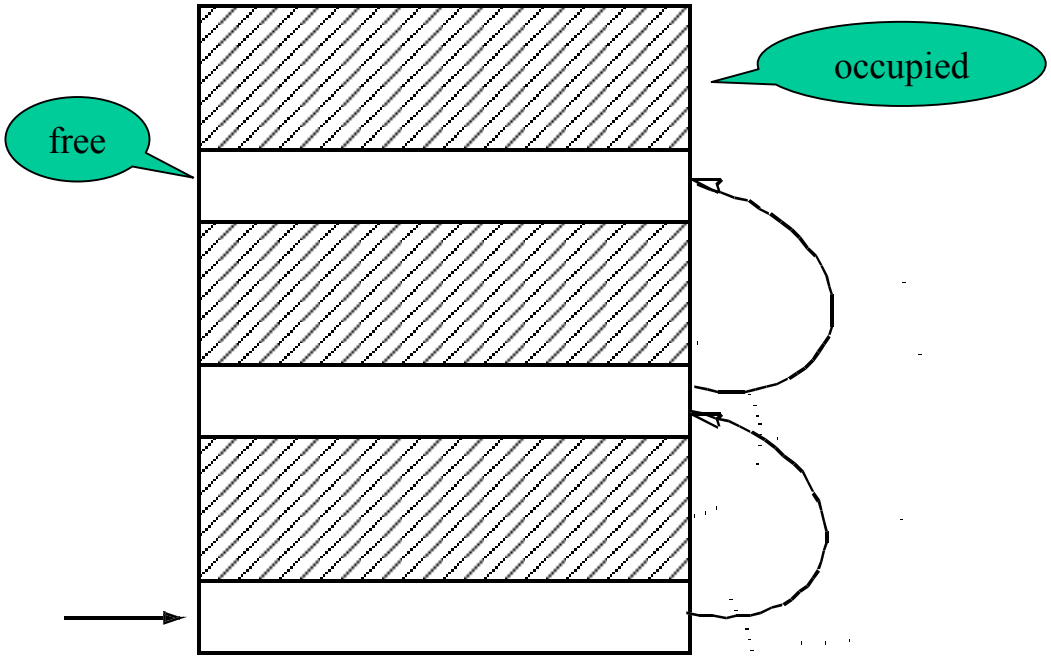
3.2.3 Heap Organization for Elements of Different Sizes

- The heap is initially one block of memory.



- allocations are done by “carving” the requested amount of from a suitably large block (see various strategies below) on the freelist

- Any remainders and deallocations are kept on the freelist which also records the size of the free blocks.
- Sometimes later …



... none of the free blocks is sufficiently large to accommodate the request (although the total free area might well be sufficiently large)

=> fragmentation of the heap!

Fragmentation has two consequences: first, it wastes memory; second and more importantly, the search times for a matching free block degenerate, as many small blocks are looked at and rejected.

- "Solution" 1: Try to avoid or minimize fragmentation by some suitable allocation strategy
- Solution 2: Move free blocks to create a contiguous free area for further allocations ("compactification")

Various allocation strategies have been proposed and extensively researched:

- **best-fit method:** select block with the least amount of left-over
  => very small, unusable blocks pollute the freelist
  => prohibitively costly to search for the best fit

- **first-fit method:** select first block with sufficient size
  => progressively smaller left-overs accumulate at the beginning of the list (search times progressively degenerate)

- **next-fit method**: freelist is a cyclic list; its “beginning” rotates around the list; otherwise like first-fit
  => distributes the small left-overs across the freelist (prolongs, but does not avoid, the point of degeneration)

- **buddy method:** search for a precise fit; if not available, split a block of (at least) twice the size; typically powers of two are used for the block sizes.
  => higher probability to reuse the left-over

- **worst-fit method:** use the block with the highest amount of left-over

- and so on .....

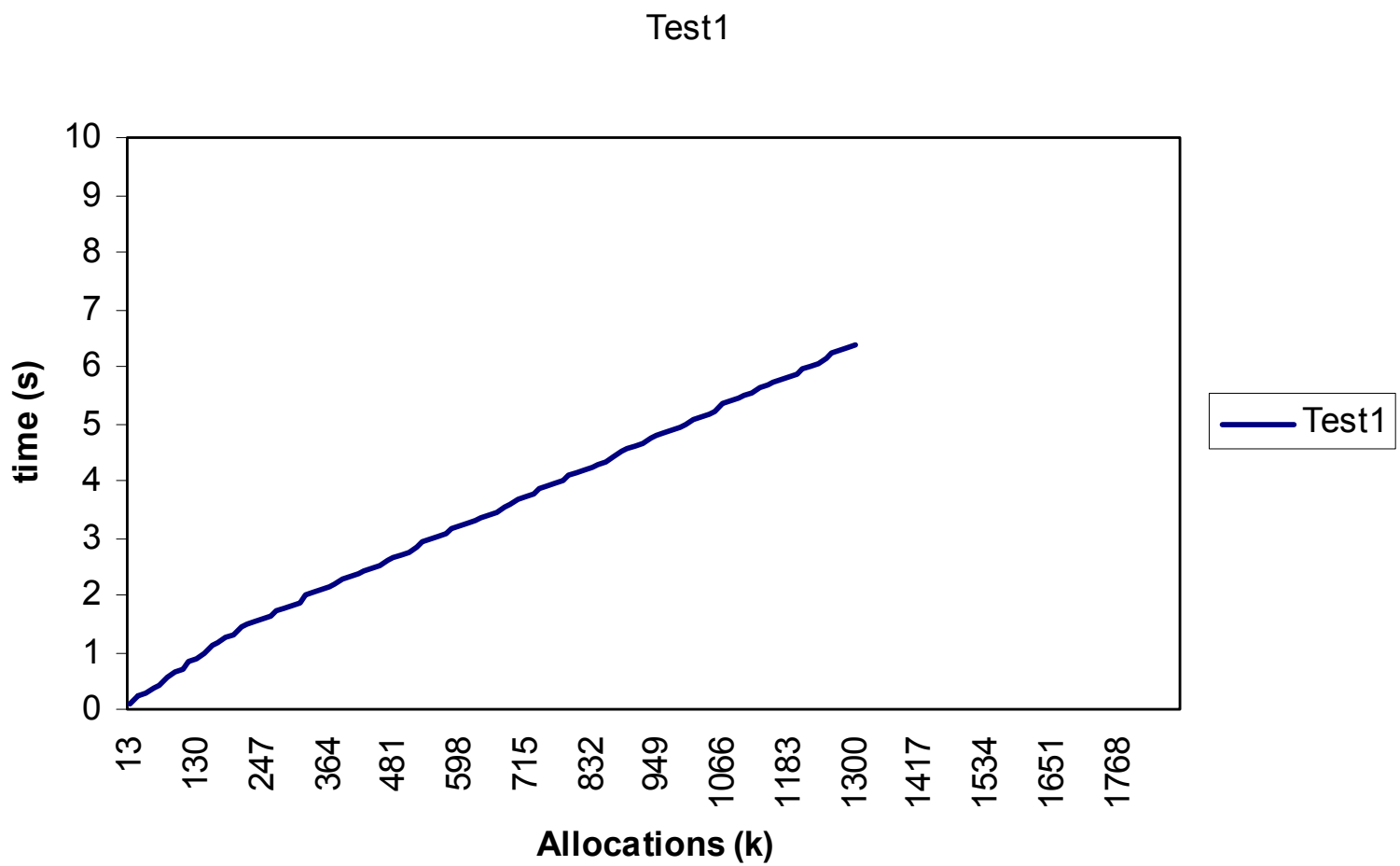
For all these variants:

- there are examples of the superiority of each strategy over all others !
- there is "Knuth's Law", stating that, in steady state, on average the heap will contain one free block (small and unusable) for every 2 occupied blocks

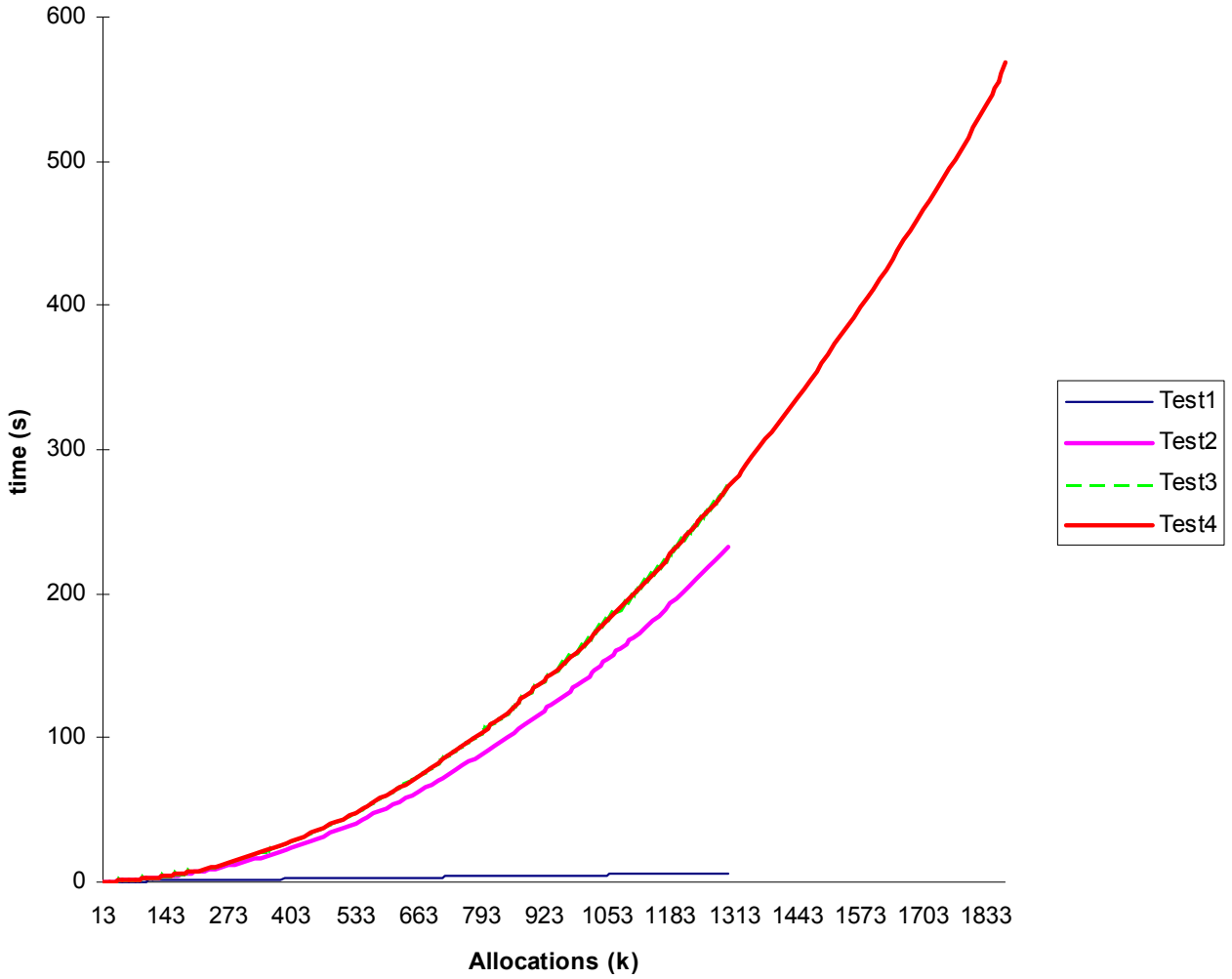
$\Rightarrow$ **Fragmentation cannot be avoided by a "clever" allocation strategy**

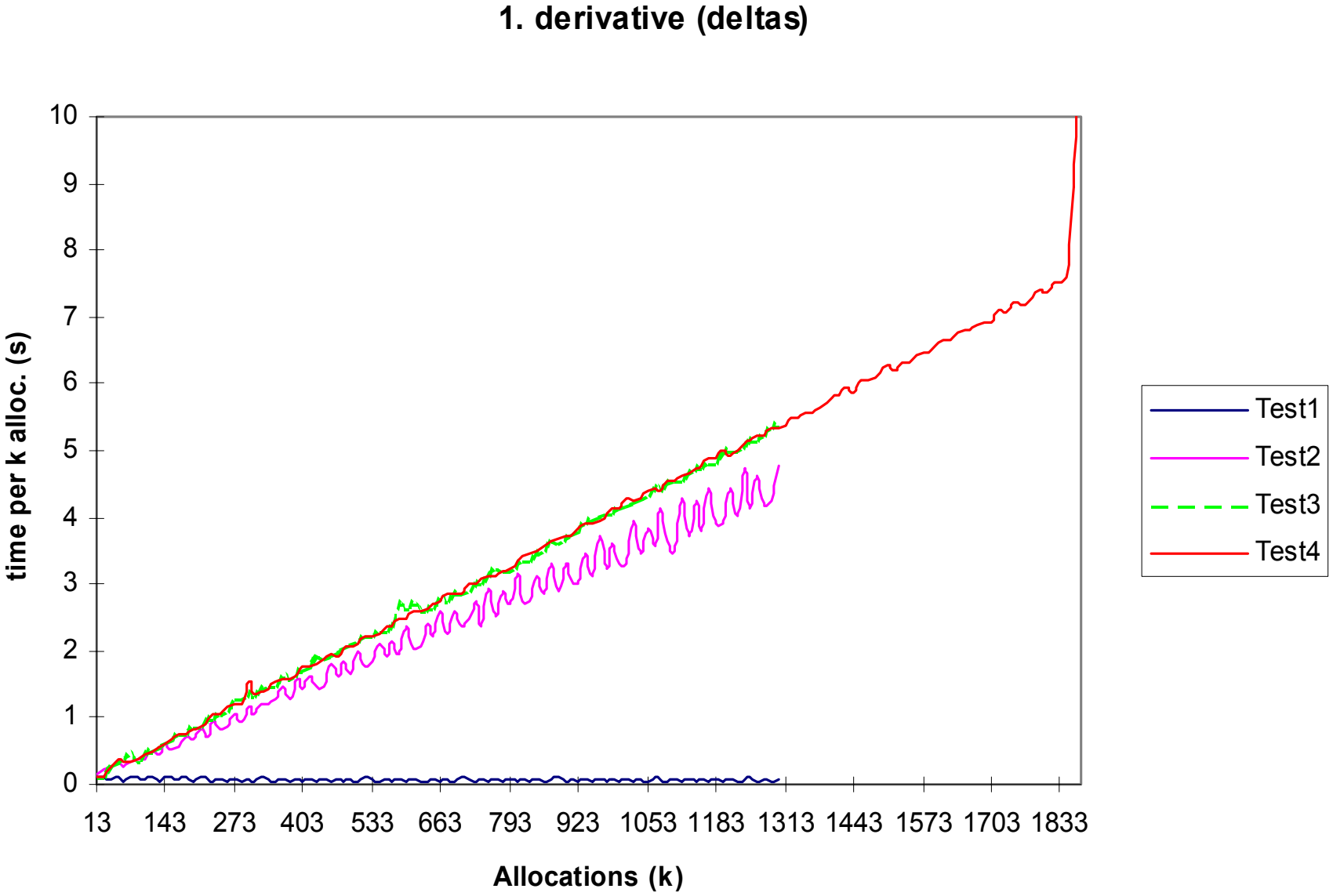
(empirically: next-fit or first-fit are our best choices)

Execution time in the absence of fragmentation



Execution time in the presence of fragmentation

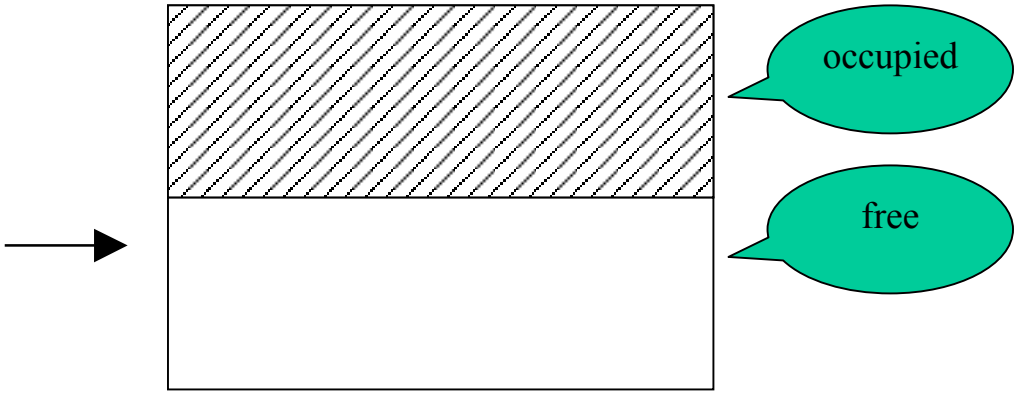



1. derivative (deltas)
time per k alloc. (s)
10
9
8
7
6
5
4
3
2
1
0
13
143
273
403
533
663
793
923
1053
1183
1313
1443
1573
1703
1833
Allocations (k)
Test1
Test2
Test3
Test4

What about "Solution 2" (compactifying the heap) ...?

- **partial compactification**: merge physically adjacent blocks
  - expensive with a freelist: since physically adjacent blocks are unlikely to be adjacent in the list, one would need to keep a freelist sorted by addresses, making allocations or deallocations expensive
  - simple (actually automatic) with a bitvector implementation

- **full compactification**: move all the occupied blocks to a contiguous area (which leaves a compactified free block)



  - very difficult: all pointers to moved areas need to be adjusted to point to the new locations (more about this problem below)

  - for real-time systems an added question: when to do it without interfering with deadlines?

3.2.4 Implicit Garbage Collection

In the previous subchapters, we assumed that deallocation is done explicitly by the user code. Such deallocations are very error-prone, as we shall see. On the other hand, if they are not made before the last pointer to the structure is lost, the memory can no longer be deallocated explicitly, resulting in "garbage". Hence, the idea evolved to not require the user to deallocate heap objects that are no longer needed. Instead, the heap management should “figure out” which heap objects are no longer needed, because they are no longer reachable via pointer structures, and deallocate them. This process is termed **Garbage Collection (GC)**.

Nota bene: **Garbage collection does not free the programmer from worrying about allocated data structures**. If they are still accessible by some forgotten pointers, they will not be garbage collected, even if they are no longer used. **Pointers that will no longer be used must be set to NULL** if garbage collection is supposed to collect the garbage.

Standard GC approach:

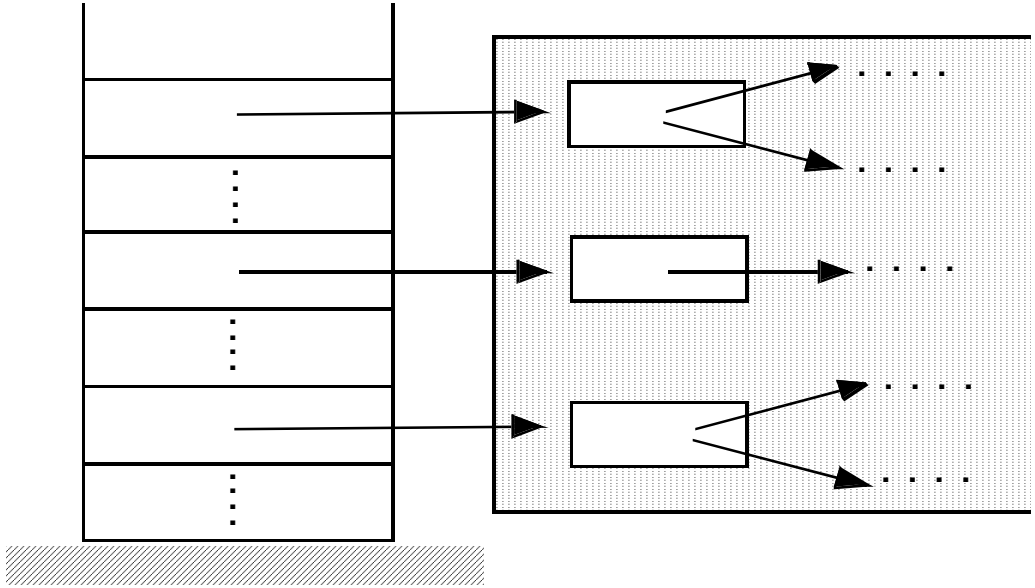
Phase 1: Mark all reachable heap objects

Phase 2: Compactify the heap, moving all the heap objects marked in phase 1

Standard problems:

- Non-deterministic interruption of the normal program by any implicit garbage collection scheme
- Identification of all pointers to reachable objects is quite difficult

Starting point are all pointers in global and local data on the stack. Typically:



Then the heap objects reachable from the stack can be marked. The process continues recursively to identify heap objects reachable via pointers within the already marked heap objects.

The overall process is an expensive breadth- or depth-first search over all pointer structures in the program.

There is a **significant technical issue** to be resolved before Garbage Collection can actually be made to work:

How does heap management know which data on the stack represent pointers? Also, how does heap management find the pointers within the heap objects?

=> A prerequisite for Garbage Collection is run-time knowledge about the nature and structure of the stored data!

Compilers for languages with static name and type binding semantics have no reason to create run-time descriptors that would identify and describe the types for the data on which the code operates. In order to enable GC, these descriptors would have to be created occupying significant amounts of memory.

- for efficiency reasons, such languages do not postulate Garbage Collection ...
- and for the same reason, their heap management does not support full compactification of the heap

In object-oriented languages, there is need for type descriptors of classes. Also, (almost) all class instances need to be allocated on the heap. Many such languages have therefore decided to require Garbage Collection.

3.2.5 Heap Management in Real-Time Systems

A continuous real-time system will not be able to cope with heap fragmentation due to unpredictable and degenerating time consumption by allocation.

A hard real-time system will not be able to cope with unpredictable interruptions by implicit garbage collection or lock-outs during compactification.

Any system relying on garbage collection will be (by today's techniques) impossible to verify with respect to staying within a limit of available heap memory.

The options that remain are few in number:

Heap utilization during a start-up phase of the system but no further allocations or deallocations during operation is acceptable (since the total amount of allocated heap memory can be measured by testing or even static analysis).

Heap management utilizing a substructure of equally sized blocks is acceptable in principle. Validation is needed that the number of occupied blocks stays below the amount of available memory at all times.

Multiple such heaps can be utilized when allocations of a few different but known sizes are needed. The above validation then needs to be applied to each such “sub”-heap.

3.2.6 Dangers of Explicit Deallocations

When deallocations are made explicitly by the programmer, there is always the danger of deallocating while there are still pointers to the deallocated object. These are called “dangling references”.

Example:

```
p := NEW(…);  -- allocation
q := p;
DISPOSE(p);  -- deallocation
...
q.comp := 10;    -- can do arbitrary damage, including
                 -- destruction of the freelist !!!!
```

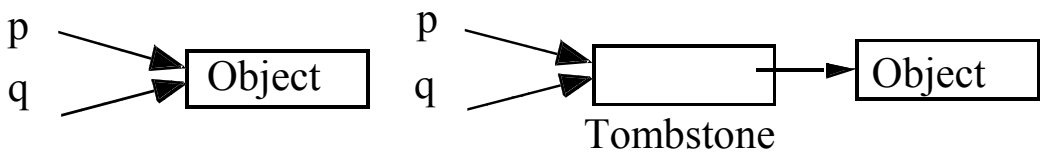
There are methods to counter this danger:

"tombstones“ , "key and lock“, "reference counting“

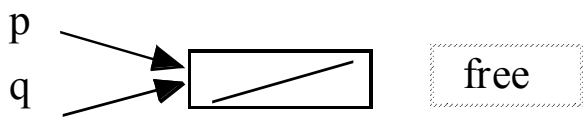
Prerequisite: no (legal) address arithmetic in the PL!

„Tombstone” Method

Idea: instead of now



„Dispose” frees the object and sets tombstone to NULL.
Dispose(q):



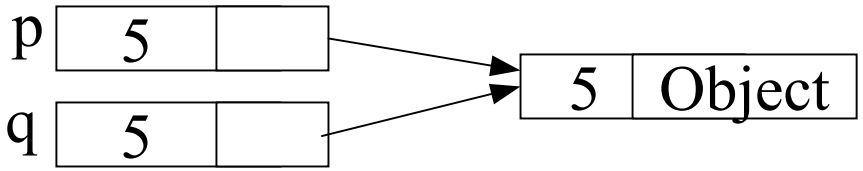
however:

- object access now requires double dereferencing
- freeing the tombstones is difficult

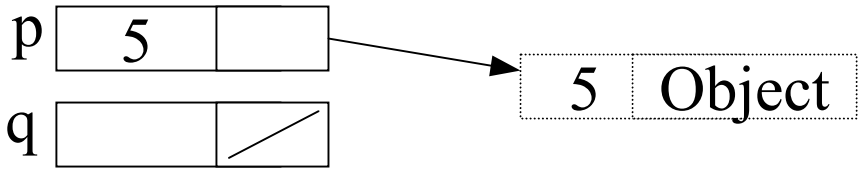
„Key-and-Lock" Method

Idea:

- heap objects have „locks" that are unique and are assigned during allocation.
- pointers have „keys" that are set equal to the lock for the result pointer of an allocation.
- access is legal only when key = lock

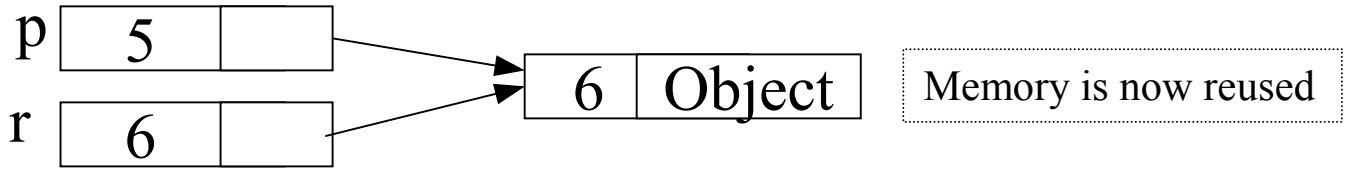


Dispose (q):



Still „available" until memory is actually reused …

New (r):



"p.all" is now diagnosed as illegal since key is unequal lock

however:

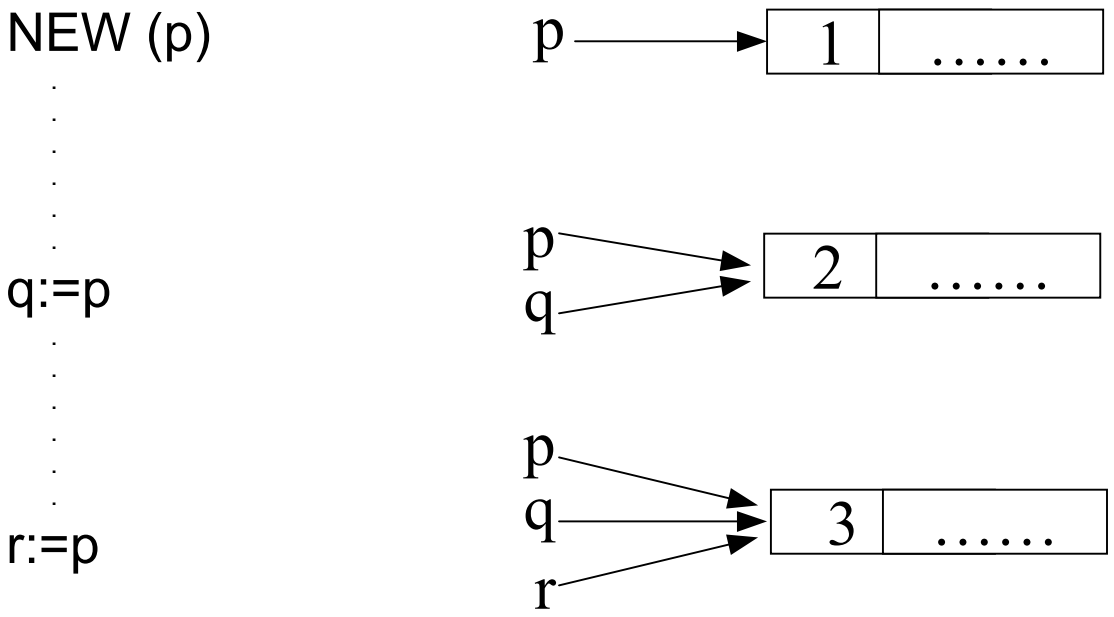
- additional space for key and lock
- problems for any compactification scheme
- cost of checks for every access

3.2.7 Reference Counting Method

This method will prevent dangling references and also assist Garbage Collection by causing the deallocation of heap memory at the moment when the last pointer to a heap object is about to disappear.

Each heap object contains a counter for the number of pointers pointing to the object

E.g.



The heap storage for the object is freed when and only when the counter reaches the value 0.

The counters obviously need to be maintained to correctly reflect their intended semantics.

- initialized upon allocation
- upon assignments "p := q" and other pointer-copying constructs:
  - incrementing the counter of q.all (unless q=NULL)
  - decrementing the counter of the old p.all (unless p=NULL)
- if the counter is decremented to 0, freeing the object
- upon exit from a block or subprogram, decrementing the counters of x.all for all locally declared pointer objects x.
- upon freeing an object, decrementing the counters of x.all for all pointer components x of the object.

One disadvantage: data structures with cyclic pointer structures (and structures reachable from there) will not be freed when they become inaccessible. The cycles need to be broken explicitly.

3.2.8 Mark/Release Method

Yet another method to prevent the accumulation of "garbage" is the so-called "mark/release" method:

Heap memory is allocated in a strictly contiguous and ascending (or descending) address order. There are no deallocations of individual heap objects. Two operations are provided on the heap:

Mark: remembers (possibly on a stack of marks) the current "top" address of the heap.

Release: "deallocates" the entire heap section above the (most recent) mark by making the heap above the mark available for allocation again.

Advantages:

- Very fast and constant allocation and deallocation time
- no garbage left after the release operation
- no fragmentation; maximum memory usage easy to measure

Disadvantages:

- Extremely serious danger of “dangling reference” after a release operation
- garbage left allocated within the mark/release cycle, increasing maximum heap requirement
- (in practice, usable only in a phase or cycle oriented system in which no state information is carried on the heap across phases or cycles)

3.3 Final Words on Memory and Run-Time Estimation

- The memory estimates MUST BE conservative, i.e. cover the worst case.
- There MUST NOT BE a "memory leak", i.e. repeated heap allocations during system operation that are not eventually paired with a deallocation. ("Once only" allocations as part of system initializations are ok.)
- There MUST NOT BE fragmentation of the heap.
- For memory estimates, all but heap size can be well approximated by static analysis of the compiled code, given a few coding restrictions.

- The run-time estimates MUST BE conservative if hard deadlines are involved. (For soft deadlines, very unlikely cases in the worst-case analysis may be ignorable.)

- The unsatisfactory state of the art in heap size and in run-time estimation is that the estimates are obtained by measurements during extensive testing that tries to illicit the worst-case behavior and by adding a large safety margin to the measured consumption.

3.3.1 Recap on Heap Management

- Risk: Fragmentation
  - Only allocation of equally sized blocks, or
  - No implicit or explicit deallocation without compactification
- Risk: Garbage
  - Have an automated Garbage Collection (GC), or
  - Have Reference Counting and manually break cycles, or
  - Make sure to explicitly deallocate before loosing last pointer/reference to heap block
- Risk: Memory Leakage
  - Without GC: make sure to deallocate (ALL) objects
  - With GC or Ref.counting: set (ALL) unneeded pointers/references to Null (holds for references held in API data structures as well !!!)
- (Further Risks: Programming Errors, Deadlines vs. GC)

Real-Time Programming

Chapter 4

Fault Tolerance

4 Fault Tolerance

A few relevant terms:

“Reliability” of a system: “a measure of the success with which the system conforms to some authoritative specification of its behavior.” (Randell et al.)

“Failure” of a system: “When the behavior of a system deviates from that which is specified for it, this is called a failure.”

A famous measure of reliability is the “**Mean Time to Failure**” metric, i.e. the mean time elapsed from system start to the occurrence of the first failure of the system.

Failures refer to the external behavior of a system. They are the result of internal **“errors”** in the system. An error arises when the computation reaches an internal state that was not anticipated in the design of the system. The internal, technical cause for an error is termed a “**fault**”.

As a system might be composed of subsystems, failure of a subsystem may result in a fault within the system (and possibly its failure).

“Fault tolerance” is the ability of a system to handle internal errors and faults in such a way that failure of the system is avoided or at least less severe in its consequences.

4.1 The Nature of Failures

- "Value Failures"
  - an incorrect result is delivered (most often a software design or coding error)
  - a service is called with arguments outside the range of values specified for this service (constraint error; often an error in requirements analysis)
- "Timing Failures": services are delivered...
  - too late (performance failure)
  - (too early) (often indistinguishable from an omission failure)
  - never (omission failure)
- "Commission Failures"
  - a service is rendered that was not expected
- combinations of the above

Some of the resulting **failure modes** of the system are:

- **fail late**: deadlines are being missed
- **fail silent**: a service and all subsequent services are never rendered
- **fail stop**: like fail silent, but service requestors can detect that the (sub)system will no longer respond
- **fail uncontrolled**

4.2 Nature of Faults

Faults can be categorized as follows, for example:

- **transient faults:** typically induced by external influences, e.g. heat or radiation, that cause a system component to fail for the duration of the external influence. The failure, in turn, causes a predictable fault in some other component. After the external influence "normalizes", the fault disappears.
- **permanent faults:** faults that persist and are easily reproducible until the fault is repaired. Examples are most software design and coding errors or permanent hardware damage.
- **intermittent faults:** faults that arise sometimes but not in a fully predictable fashion.

(N.B. This classification is slightly different from the one given by Burns.)

Of these, intermittent faults are the most difficult to discover and to prevent.

There are essentially two ways of dealing with faults:

- **fault prevention** by fault avoidance and fault removal
  (-> development process)

- **fault tolerance** by reacting to arising faults at run-time

4.3 Methods of Fault Prevention

Fault avoidance:

- use only the most reliable components and production techniques suitable for your product system and its intended environment
- adhere to a well-defined development process
- use proven, rigorous methods for specification, design, coding, test, etc.
- utilize a good software engineering environment
- perform a risk analysis on the overall system!
- be paranoid

Fault removal:

- verification and validation
- code reviews
- (system testing)
- “Cleanroom” techniques (-> Harlan Mills et al.)

Remember Djikstra:

"Testing can only show the presence of errors, never their absence."

Despite all the above, some failure causes will remain in the system.

Fault tolerance will need to be built into the system to deal with the remaining potential of faults.

4.4 Levels of Fault Tolerance

Based on the system's behavior in the presence of faults, we can distinguish at least three levels of fault tolerance:

- **full fault tolerance:** the system will continue to operate in the presence of faults without significant loss of functionality or performance.
- **graceful degradation:** the system will continue to operate in the presence of faults, albeit with a partial degradation in functionality or performance.
- **fail safe:** the system will cease to operate in the presence of faults, however only after ensuring that its external effects are such that the system environment is left in a state in which the potential of damage to life, health, and property is minimized.

The nature of faults has a major impact on the level of fault tolerance that is achievable. Many faults will imply that, at most, graceful degradation can be achieved.

Equally, the nature of faults has a major impact on which measures can actually increase fault tolerance of the system (and which are perfectly useless for a given class of faults).

Along this dimension, one needs to distinguish faults

- caused by hardware failure
- caused by software design and coding errors
- caused by errors in the requirements analysis

A further issue is how a fault is actually detected.

4.5 Fault Tolerance by Redundancy

Redundancy implies that system components are replicated (in various forms), so that failure of one component instance can be compensated by properly functioning alternate components.

Two strategies can be considered:

a) duplicated components are available in case the primary component fails (backup strategy). Usually, the duplicating components are concurrently active, but their results are ignored until the primary component fails.

b) all components are concurrently active and their results are compared; the majority result wins. (voting strategy)

For the backup strategy, the fault detection needs to occur within the primary component. The voting strategy can be used to detect faults by considering the component(s) with the minority results to be faulty.

The voting strategy is considerably more complicated to implement, as the voting implies additional functionality in the system. For some of the issues in voting, see Burns and Wellings.

Redundancy can be applied both to the hardware and the software components of a system.

4.5.1 Effects of Hardware Redundancy

For the voting strategy, an often practiced method is **Triple Modular Redundancy (TMR)**, consisting of three identical instances of the replicated component, plus circuitry to realize the voting.

Effects of TMR:

- faults in the hardware that affect only one of the instances can be masked out (full fault tolerance)
- transient faults that affect the instances differently can be detected
- TMR is useless to discover or tolerate
  - systemic faults in the hardware (hardware design or production errors)
  - any software-related faults
  - any faults caused by errors in the requirement specification

Heterogeneous redundancy (hardware of identical functionality but different design) could detect and mask some additional faults but is rarely practiced because of substantive technical problems in the implementation of voting.

Redundancy for a backup strategy has essentially the same effects, except that faults cannot be discovered by virtue of the redundancy. It is much simpler to implement, however.

Nota Bene: Hardware redundancy as the only fault tolerance mechanism unrealistically assumes absence of software faults. (“Text-book example: ARIANE”)

4.5.2 Software Redundancy: N-Version Programming

Identical replication of software in analogy to TMR would be a perfectly useless exercise. Instead, a useful analogy can be derived from a heterogeneous hardware redundancy:

The software for a particular functionality is implemented in multiple versions, developed by teams in total isolation from each other. (The reasonable assumption is that not all teams will introduce the same faults in their version. Use of different programming languages or different development environments can provide additional protection against tool-induced faults.)

The N versions of the software are incorporated in the operational system; voting or backup strategies are applied to obtain the “right” result. For a backup strategy, a variant is that the backup version operates with less stringent requirements, e.g. on accuracy of numeric results.

Effects of N-Version Programming:

- multiplied development cost
- multiplied execution resource requirements
- many faults caused by software design and coding can be detected and masked
- many transient hardware errors can be detected and masked
- (quite) useless against errors in the requirement specification

Because of cost, N-version programming is rarely used, except for localized backup strategies, avoiding timing faults.

4.6 Fault Tolerance by Means of Software

Four tasks are to be accomplished (Anderson and Lee):

1. **Fault detection:** discover that an error has occurred.
2. **Damage confinement and assessment:** discover how much corruption of the system has already occurred. (The fault may manifest itself only some time after its original cause, e.g. on memory corruption.)
3. **Error recovery:** fix any corruptions to allow for continued operation of the system.
4. **Fault treatment and continued service:** if possible at all, take measures to avoid the recurrence of the error. Log the fault for maintenance. Continue the services, possibly in degraded form.

With the exception of a few faults that are detected by the hardware (e.g., address violations, divisions by zero, overflows), all these tasks will require explicit code to be incorporated in the operational software system.

For another subset of faults, higher-order programming languages will provide for implicit detection code generated by the compiler or incorporated in the run-time system. This can ease the task of the software designer greatly, as these faults are “automatically detected” from his or her viewpoint. The downside is that this detection code causes execution overhead in those places where the application is guaranteed to not cause the respective fault.

4.6.1 Error Detection by Software

There is a large repertoire of error detection mechanisms that can be employed:

- **hardware traps and implicit checks** imposed by a programming language (**-> exceptions**).
- **reversal checks:** for functions that have an easily computable inverse, check that the computed result matches the argument.
- **data coding checks:** for data, validity can be checked by means of checksums and similar redundancy mechanisms.
- **constraint checks:** check that data (input and output) are within the range postulated by the specification - in some languages subsumed by implicit checks.
- **precondition checks:** check that the preconditions for a service are satisfied. In some languages, this is explicitly supported by **"assertions"**, Boolean expressions that when evaluated to false indicate a fault.

- **plausibility checks:** using application knowledge, check that a computed result is within an interval that can be reasonably expected at this point, e.g. for the current state of the system or based on the degree of change compared to previous results.
- **structural checks:** check for violations of structural properties of data, e.g. tree-ness, absence of cycles in data structures.
- **timing checks:** check that a service is rendered timely for deadlines (-> **"watchdog timers"**) or check predictively whether sufficient time remains until the deadline.

The big challenges of error detection by software:

- There must be an overall strategy in the system and software design defining how responsibilities for error detection and handling are allocated to software modules (e.g. “does the caller of a service check preconditions or does the service?”)
- There should be a predictable, uniform style of performing the necessary checks (e.g. “the return status of a service must always be checked”)
- Where possible, checking code and operational code should be easily distinguishable.
- For fault tolerance, the job doesn’t end with the detection of the error. The error also needs to be handled to prevent failures of the system. Developing an overall handling strategy as well as detailed responses to individual error situations may be much more difficult than the design of the system in normal operation mode.

Error detection and general fault tolerance introduced into a system as an afterthought, i.e. not reflected in the original design, is rarely (never?) successful.

The job gets considerably easier if good architectural principles of software engineering are adhered to, in particular good modularization with well-specified interfaces and unbroken encapsulations.

Caveat: Do not forget the possibility of hardware failures and faults in your analysis of possible error situations.

4.6.2 Damage Confinement and Assessment

... is, to a large extent, a design activity of understanding the possible causes that lead to a detected error and assessing the amount of damage that might have been done between the occurrence of the cause and the detection of the error. E.g. a value domain fault may have "polluted" any number of other results before being detected; a memory corruption may have corrupted more than is apparent upon error detection.

Damage confinement tries to minimize the effects of faults. A good design rule is obviously: **"Detect and handle early!"** Further, the design can incorporate firewalls across which the propagation of fault effects is minimized or made completely impossible. Examples of firewalls are:

- **modular encapsulations**, used only via well-defined and checked interfaces, help to minimize pollution effects.

- Hardware or run-time support for **separate address spaces** assigned to different software components eliminate memory corruptions across these address spaces. Other **protection mechanisms** on resources (read and write privileges, dependent on system or component mode) can equally limit damage propagation.

- **atomic actions** (“transactions”) which guarantee that, once their state changes are initiated, no other activity can interact with these changes until they are completed.
  (-> **synchronization mechanisms** in real-time languages)

Damage assessment tries to discover the actual damage done by a fault. Many of the error detection mechanisms can be applied to the system state to discover corruptions. However, if there is any likelihood that damage might be overlooked or, if discovered, could not be reasonably repaired, this effort might not be worth it. In this case,

go for a lesser degree of fault tolerance instead!

A fail-safe reaction or degenerate mode of operation is often vastly preferable to continued operation with corrupted state.

4.6.3 Error Recovery

With a fault detected and damage assessed, error recovery transforms the corrupted state of computation to one in which operation can be continued, possibly in temporarily or permanently degraded mode. Two overall strategies apply:

- **forward recovery:** the state is fixed by selective updates to achieve a well-defined state that allows continuation. A typical language mechanism for forward recovery is the **raising and handling of exceptions**. Operation continues after these fixes at a point determined by the exception model of the language. This is either the point where the fault occurred ("resumption" model) or a point after an enclosing construct with the applicable exception handler ("recovery" model).

- **backward recovery:** at certain user-defined points in program execution **(“recovery points”)**, a copy of the system state is taken and stored **(“checkpointing”)**. Upon detection of a fault, the system is restored to the state of a recovery point and the operation continues at this point. Variations of the theme are **“hot restart”,** with a single recovery point taken after system initialization, or **“incremental checkpointing”,** retaining only interesting portions of the state.

4.6.4 Assessment of Forward Recovery

- To “fix the state” to be the intended one, sufficient redundant information must be retained at run-time to allow for such restoration; this must be preplanned in the software design.
- To fix the state to be only a safe state for continuation, requirements on information retention at run-time are usually much less demanding.
- In order to assess and modify the state, its components must be visible to the exception handling code. This need may break or weaken encapsulation principles.
- Care needs to be taken that the recovery does not lead to an omission failure causing faults in other components (e.g. omission of a signal in code omitted as part of the recovery semantics after exception handling).

- The mechanism works regardless of the nature of the fault.

- Exception handling as a mechanism (if competently implemented by the compiler and run-time system) is very efficient.

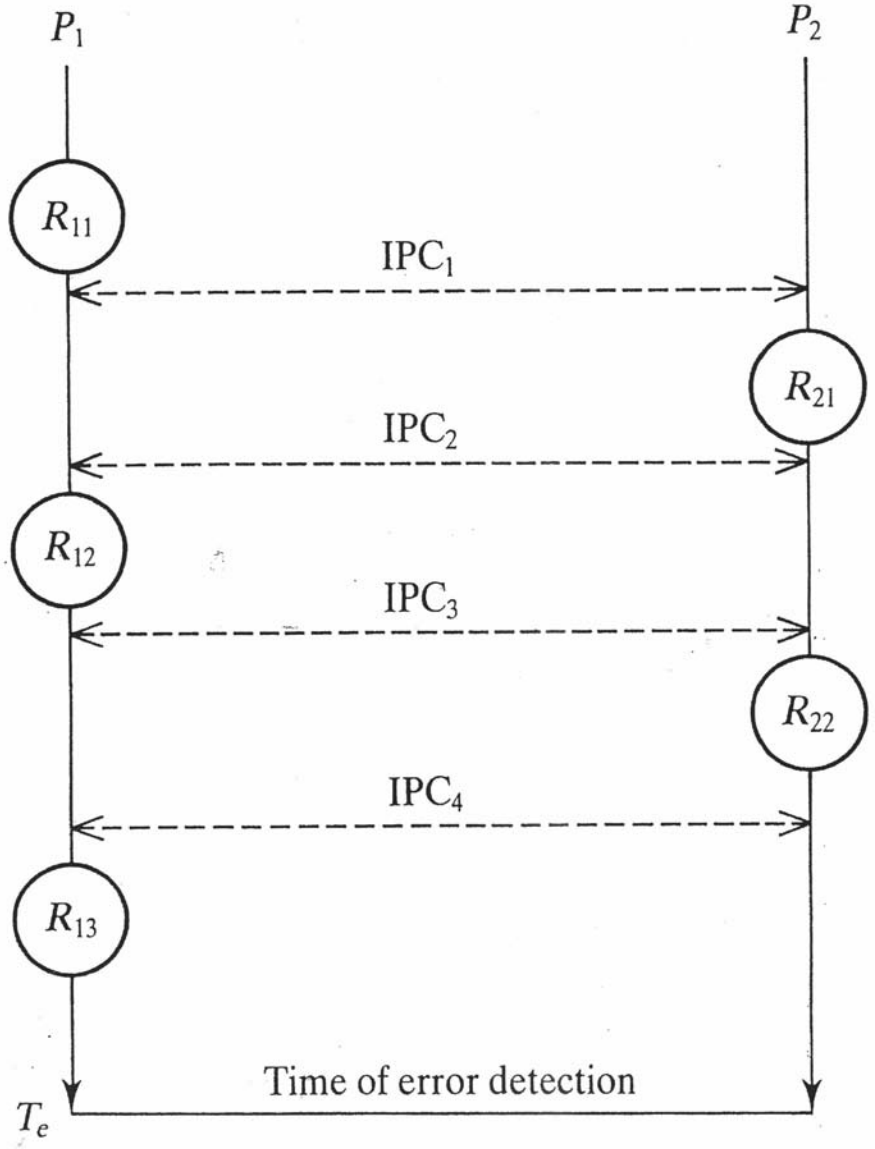
- A mandatory verification activity is to show that any exception potentially raised will indeed be handled. An unhandled exception will cause the executable to unexpectedly end its execution. For the usual exception models (Ada, C++, Java), such verification is possible by induction proof over the Call Graph, augmented with information about exceptions potentially raised and exceptions handled by the respective subprograms. There must not be a path from the root to a potential exception in a subprogram without a matching handler in a subprogram on the path.

4.6.5 Assessment of Backward Recovery

- Checkpointing and restoring full states is a very expensive operation.
- Encapsulations can remain fully intact, as the restoration is done in toto and in its details implicitly from the viewpoint of the program.
- Restoration and retry will not work for permanent faults as the same fault will be encountered again.
- There is the danger of creating commission faults if any external behavior has been effected since the last recovery point. These effects will be repeated.
- There is the danger of creating artificial omission faults, unless related components that have interacted with the given component since the last recovery point are induced to repeat these actions as well. This may be physically impossible (e.g. for sensor data).

- The last two issues make backward recovery very difficult if multiple processes are involved - they all would need to be reset to a recovery point that is common to all of them. (On distributed systems, this is infeasible.)

- For recovery points that are associated with individual processes, see the discussion of the **domino effect** in Burns, p.114, potentially causing a cascade of recoveries.



Domino effect:

Consider what needs to happen when a fault occurs in P1 and compare it to the events if the fault is in P2.

You need to take care of commission or omission faults caused solely by the recovery to the check point.

4.6.6 Idioms of Exception Handling

Full recovery:

The problem can be fixed locally within the given specification.

```
 ...
 ...
 ...
begin
        A := B + C;
exception
        when Constraint_Error =>
                A := Integer'Last;
end;
```

(Limited) Retry:

An idiom to deal with transient or intermittent faults.

```
for Attempt in 1..10 loop
        begin
                Get(Sensor, Data);
                exit;                          -- if reading is successful
        exception
                when SENSOR_INPUT_ERROR =>
                        if Attempt < 10
                        then Reset(Sensor); -- restoring state
                        else raise FATAL_SENSOR_ERROR;
                        end if;
        end;
end loop;
```

Non-specific Forward Recovery:

An idiom of desperation, feasible only if the state to be restored is simple enough and there is no alternative reaction available.

```
        loop
        begin
                Get_Sensor_Data (Data);
                Control_Airplane (Data);
        exception
                when others => Reinitialize_Cycle_State;
                        -- „skip this cycle”, but KEEP FLYING!!
        end;
end loop;
```

Note the danger that, if the fault is permanent, this loop will be infinite and cause a “fail silent” failure for any enclosing system.

Degenerated or Alternative Behavior:

An idiom to be able to respond to a fault by some algorithm with an alternative algorithm of sufficiently similar or degenerate behavior:

```
begin
        Find_Very_Accurate_Solution;
exception
        when Numeric_Instability | Not_Enough_Time =>
                Find_Sufficiently_Accurate_Solution;
end;
```

"Last Wishes":

While the fault cannot be handled locally but will be handled at a higher level, certain actions may be nevertheless needed locally to prevent subsequent faults or to enable a fail-safe shut-down.

Example for preparing a fail-safe shutdown:

```
Valve: Gaspump_Valve := Identify_Valve(Signal);
begin
        Open(Valve);   -- can raise Valve_Did_Not_Open
        while Sensor_Indicates_Flow loop
          Measured_Quantity := Sensor_Reading;
          Communicate(Monitor, Measured_Quantity);
                -- can raise a number of communication exceptions
          delay 0.01;
        end loop;
        Close(Valve);
        exception
              when Valve_Did_Not_Open =>
                    Communicate(Monitor, 0);  raise;
              when others =>
                    Close(Valve);  raise;
end;
```

While this code is now fail-safe with respect to ensuring a closed valve if the flow cannot be communicated to the monitor, there is a subtle problem here if both faults happen to combine. The fix is left to the reader. (It illustrates how „paranoid“ one has to be.)

Another example of last wishes follows in the next subsection.

4.6.7 Caveats of Exception Handling

The following code is extremely "bad news”:

```
procedure Produce;
-- deposits data in global buffer A at position I
-- raises Constraint_Error, if I violates A’s index bounds
```

```
procedure Produce is
        X: data_ptr := new data;
begin
        fill_and_process_data(X);
        get_lock(A.lock); -- get a lock on the global variable A
        A.contents(I) := X.comp;
        release_lock(A.lock);
        Free(X);
end;
```

If the Constraint_Error occurs, the buffer A will remain locked (-> subsequent deadlock faults) and the memory allocated for the data is never freed (-> subsequent out-of-memory faults possible).

First attempt at improvement:

```
procedure Produce is
        X: data_ptr := new data;
begin
        fill_and_process_data(X);
        get_lock(A.lock);   -- get a lock on the global variable A
        A.contents(I) := X.comp;
        release_lock(A.lock);
        Free(X);
exception
     when Constraint_Error => -- "last wishes", then propagate
                release_lock(A.lock); Free(X);
                raise;
end;
```

The improvement helps but it ignores the situation of an exception being raised by ’fill_and_process_data’

Second improvement:

```
procedure Produce is
        X: data_ptr := new data;
begin
        fill_and_process_data(X);
        get_lock(A.lock);   -- get a lock on the global variable A
        A.contents(I) := X.comp;
        release_lock(A.lock);
        Free(X);
exception
  when Constraint_Error =>      -- "last wishes", then propagate
        release_lock(A.lock); Free(X);
        raise;
  when others => -- last wishes for other exceptions
        Free(X);
        raise Producer_Failed;
end;
```

If ’fill_and_process_data’ cannot raise Constraint_Error, this version will satisfy the need for last wishes. Note, however, that the exception behavior of the specification needs to be refined. This should be done during software design, not as an afterthought prompted by test failures.

The example also illustrates another design issue: Why doesn’t the caller of Produce ensure that ’I’ is within index bounds? This would generally obviate the need for such selective recovery, unless one needs to be paranoid about transient hardware faults.

Another interesting exercise:
reconsider the example if "fill_and_process_data" can
raise Constraint_Error. What can go wrong?

Depending on language and compiler, the following code is also bad news:

```
...
State := 1;
A := B/C;
State := 2;
B := C/E;
State := 3;
C := E/F;
State := 4;
exception
        when Constraint_Error =>
                case State is
                        when 1 => ... -- B/C failed
                        when 2 => ... -- C/E failed
                        when 3 => ... -- E/F failed
                        when others => ... -- utter surprise
                end case;
```

This code may result in "utter surprise". For efficiency reasons, languages allow reordering of operations within an exception frame as long as all data dependencies in the frame, excluding the handlers, are satisfied.

Therefore, when an implicit exception is raised, do not assume anything about the precise location of the cause of the exception within the frame. For the same reason, do not abuse implicit exceptions as intended control structures.

As an aside: Without very special specifications (-> volatility of the variable), you cannot assume that an assignment to a variable, followed by yet another assignment without intervening use of the variable, will be made at all by the compiled code.

4.7 Finally, the Ideal Component (Burns)…

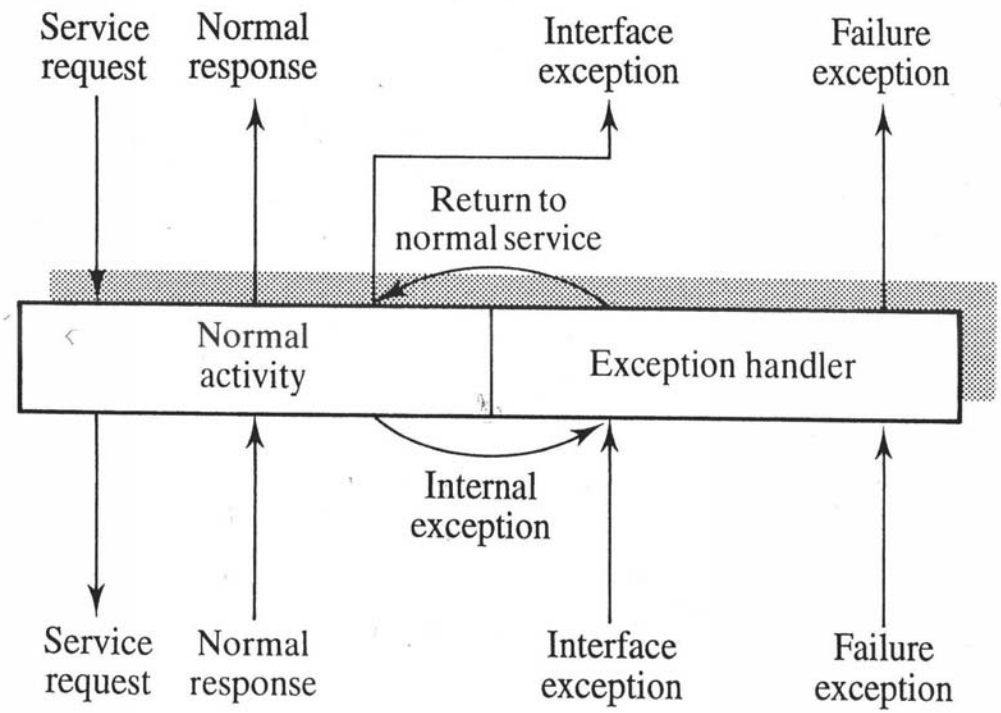


Figure 5.8 An ideal fault-tolerant component.

Real-Time Programming

Chapter 5
Scheduling - “Take One”

5 Scheduling - “Take One”

The tasks performed by a real-time system can be initiated

- by explicit instructions in executing software, i.e. by synchronous program control.
- in response to (asynchronous) events caused by other software components or hardware, i.e. in an event-driven system.

Many of today’s real-time systems perform a combination of event-driven and program-driven tasks.

Many tasks of a real-time system need to take place repeatedly and with a certain regular frequency. These are the **periodic** tasks. Others might take place irregularly in response to certain system states or events. These are the **aperiodic** or **sporadic** tasks.

Scheduling is responsible for causing the right tasks to be executed at the right time, meeting any deadlines associated with these tasks.

For simple program-driven systems with certain characteristics of the tasks, scheduling can be achieved by a **cyclic executive** requiring little beyond sequential programming.

For other systems, more elaborate scheduling schemes are necessary that involve the notion of **preemptable**, **logically concurrent** units of execution, such as **processes**, **threads or tasks** in the execution environment.

5.1 Cyclic Executives

Cyclic executives are fairly common in real-time systems consisting of a few periodic tasks, e.g. monitoring a few gauges and controlling equipment, such as valves or wing surfaces, accordingly. (Historically, cyclic executives are the oldest form of schedulers.)

For cyclic executives, certain prerequisites apply:

- the system has a fixed number of tasks
- all tasks are periodic with known periods (e.g. 10, 100 Hz)
- worst-case execution times are known for each task
- the deadline of each task is equal to its period, i.e. it must be finished before it needs to be started again; early result delivery is allowed
- the tasks are independent of each other

The last two prerequisites can be weakened in practice (tighter deadlines, ordering of tasks) at the cost of more constraints on building the cyclic executive.

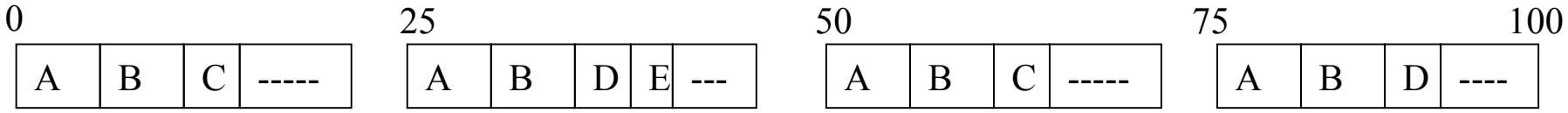
Scheduling in a cyclic executive is entirely the responsibility of the program designer. He/she needs to pre-plan the order of executing the tasks. The typical design paradigm is to subdivide a timeline of fixed length (the **major cycle**) into smaller portions (the **minor cycles**) and then to allocate the task executions within the minor cycles such that

- they are executed with their required period
- even for their worst-case execution time (**WCET**), they will finish before the end of the minor cycle

For this to work, **all task periods must be a multiple of the minor cycle period.**

Example:

Task	Period	WCET
A	25 ms	10 ms
B	25 ms	8 ms
C	50 ms	5 ms
D	50 ms	4 ms
E	100 ms	2 ms



```
start_up;

loop
        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_C;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_D;
        procedure_for_E;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_C;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_D;

end loop;
```

Some observations on cyclic executives:

- Programming is straightforward: essentially the tasks are plain procedures called in sequence of the schedule (plus code to wait for the start of each minor cycle).
- The strict sequentiality allows uncomplicated communication among the tasks via shared global variables (unlike the process models discussed later), as long as the right order of task invocations is observed in the schedule.
- The cyclic executive “proves” schedulability of the tasks by the fact that it was possible to construct it.
- The **really hard part is to come up with the schedule**: this is a bin-packing problem and as such NP-hard, i.e. cannot be optimally solved except for trivial cases.
- Adding a task **during maintenance** (or worsening a WCET) may require the **development of an entirely different schedule.**

- If a task with a long period is present, either the major cycle needs to be at least as long or artificial secondary schedules need to be introduced (i.e. a procedure that calls the real procedure for the task only every N major cycles).
- If the worst-case execution time (WCET) of a task exceeds the minor cycle, it needs to be broken up artificially into multiple tasks, often affecting maintainabilty.
- If the task periods do not allow for a sizable minor cycle, many tasks will have this problem. (A very large common divisor for the task periods is required in practice.)
- The actual interval of two invocations of a task may vary from the specified period by as much as the length of the minor cycle. On average the period will be obeyed, though.
- **Sporadic tasks cannot be sensibly accommodated**.

5.2 Waiting for an Event

The cyclic executive contains code that waits for a minor cycle to start. This requires a (hard- and software) clock with suitably fine resolution for the application, a **real-time clock.**

If 'current_time' is a function that returns the current value of the real-time clock, such waiting can be implemented as follows:

```
start_up ===  next_cycle_time := current_time + minor_cycle_time;


wait_for_minor_cycle_begin ===
          while current_time < next_cycle_time loop
                  null; -- a busy-wait loop
          end loop;
          next_cycle_time := next_cycle_time + minor_cycle_time;
```

(To be precise: code to protect against the clock “roll-over” also needs to be added; the time for the failing comparison and the assignment needs to be subtracted from the minor cycle time.)

This style of waiting for an event, e.g. for the clock reaching a certain value, is called busy waiting, since it keeps the processor busy doing nothing but repeatedly checking. Normally busy waiting is not advisable, as the processor might be put to good use for other tasks. However, in cyclic executives it is fine, since nothing else is scheduled to be done, anyway.

For non-busy waiting and for other disadvantages of busy waiting, see later chapters.

From a different perspective, repeatedly querying a device (i.e. the clock) for needed input is called **polling** the device. Other strategies in later chapters avoid repeated queries or polling by busy-wait.

5.3 The notion of process, thread, or task

In order to relieve the program designer from the arduous task of sequentializing entire systems by hand and also to be able to map programs onto multiple processors, some notion of (logically) concurrently executing units is needed. Informally, such units are referred to as **processes**. Various programming languages explicitly support such units; alternatively, support for processes is present in most operating systems and real-time kernels.

Unix has had a major influence on the terminology. Today, we typically distinguish the following concurrent units:

- **Process:** an executable program, communicating with other processes by mechanisms provided by operating systems or kernels (e.g. message passing, remote procedure call). Within a process, there may be multiple concurrent threads of control.

- **Thread:** part of an executable program communicating with other threads within the same program/process possibly by the use of shared memory, e.g. variables that can be concurrently accessed by multiple such threads.

- **Task:** A term specific to some programming languages (e.g. Ada) for a concurrent unit in the language, conceptually very close to a thread (although not necessarily mapped to a thread in the Unix sense).

On a single processor, the concurrency is strictly logical, as only one concurrent unit can execute at any given time. On a multi-processor or a distributed system, truly parallel execution of these units becomes possible.

There is considerable debate on whether concurrency should be supported by programming languages (and their run-time systems) or should be the domain of real-time operating systems (RTOS) and kernels.

In favor of RTOS and kernels:

- ability to program the concurrent units in different languages
- likely mismatches of the programming language's concurrent units to the RTOS/kernel units and of the RTOS/kernel assumptions about concurrency within processes
- simpler programming language
- replaceable kernel implementations

In favor of programming languages:

- embedded systems might not have an RTOS
- more readable and maintainable programs
- compiler consistency checking of task interactions
- RTOS/kernel primitives are often slower than PL run-time support
- if the compiler is unaware of concurrency, the generated code may not be suitable for parallel execution (in particular, for thread preemption)
- a compiler can occasionally optimize task interactions considerably

(My personal opinion: definitely in favor of the programming language, mainly because of the last three technical reasons)

5.4 States of Processes, Threads, and Tasks

Concurrent units can be in any one of the following states during run-time:

- **(non-existing)**
- **created:** the unit now exists
- **initialized:** the unit is now ready for interactions
- **runnable:** given a processor, the unit could execute
- **running:** the unit is currently executing
- **blocked:** the unit is waiting for
  - a timing event
  - a resource currently locked by a(nother) unit
  - an event to be caused by external devices or another concurrent unit

- **(completed, ready for termination)**: two special states in connection with termination semantics of some concurrency models
- **terminated**

A direct transition from running to runnable state of a concurrent unit caused by a scheduler is called a **preemption**.

When there is a change of the unit executing on a processor, a **context switch** is necessary to save the state of the vacating unit if needed for later continuation and to (re)instate the state of the unit now to be executing.

The run-time system/OS/kernel tracks the state of the units, schedules them for execution and causes the context switches.

5.5 Process Scheduling

Unless noted otherwise, we use the term “process” in the following to also encompass threads and tasks.

As there are usually less processors available than processes to be executed, the execution of the processes must be multiplexed over the available processors (often a single one).

Scheduling is responsible for the assignment of processors to processes. There are (at least) four different schemes applied:

- **non preemptive scheduling**: once a process has been assigned a processor, it runs to completion or until it executes an action that causes it to be become blocked. This scheme is generally inappropriate for real-time and most other systems.

- **time-sliced scheduling:** typically the default on non real-time operating systems. Processes get the processor for a limited time; if the time slice is used up, the process is preempted and the processor is given to the next process. This scheme is inappropriate for most real-time systems, as deadline guarantees are very difficult to establish.

- **priority-based preemptive scheduling**: at any time, if a higher-priority process can run (and no processor is idle), the execution of any lower-priority process will be immediately suspended and the processor given to the higher-priority process. This is the usual scheme for real-time scheduling.

- **deferred preemption, cooperative dispatching:** in principle like priority-based preemption; however, preemption is not immediate, but rather at a subsequent point in execution that is more “convenient” for the preempted process.

5.6 Priority-based Preemptive Scheduling

The application designer is no longer responsible for establishing a schedule, since scheduling is done by the run-time system. (From now on, the term “run-time system” is to be understood to encompass the various implementation forms “operating system” or “kernel” as well).

The explicit “chopping” of tasks often necessary in cyclic executives to fit subtasks into the minor cycles has an equivalent in the preemption and later resumption of the processes controlled by the scheduler in the run-time system. Also, while a process is waiting for an event, e.g. a timer event, the scheduler is able to utilize the processor for other processes. The producer of the run-time system needs to implement these capabilities.

Now the application designer is responsible to assign priorities to the processes so that the processes meet their deadlines.

Luckily, there are systematic ways of accomplishing this job, foremost by "**rate- or deadline-monotonic priority assign-ment**". As part of assigning the priorities, a **schedulability analysis** is now needed to prove that the tasks will meet their deadlines.

Initially we assume a system consisting only of processes that exhibit the characteristics enumerated for cyclic executives (only independent periodic processes with deadlines equal to period). Also, the formulas are presented assigning zero cost to system overhead, in particular context-switching time. We will then gradually weaken these assumptions.

All the analyses start at the **critical instant**, a time in execution when all processes are ready to run, a situation sure to arise eventually with periodic independent processes.

5.6.1 Rate-monotonic Priority Assignment

Each process is assigned a unique priority based on its period using the simple rule

The shorter the period, the higher the priority.

Theory tells us that, if any process set can be scheduled to meet its deadlines (equal to the periods) with any fixed priority assignment scheme under priority-based preemptive scheduling, then it can also be scheduled with a rate-monotonic priority assignment.

Example:

Task	Period	Priority
A	25 ms	5
B	60 ms	3
C	42 ms	4
D	105 ms	1
E	75 ms	2

N.B. "1" is by convention the lowest priority.

Note that a cyclic executive would be practically incapable of accommodating these periods.

We now need a schedulability analysis to determine whether this process set is schedulable to meet its deadlines ...

5.6.2 Utilization-based Schedulability Analysis

This is a very simple test of a sufficient (but not necessary) criterion for schedulability.

With the following notations:

N = number of processes in the set

C_i = WCET of the i-th process

T_i = Period of the i-th process

the test requires that

$$\sum_{i=1}^{N} (C_i / T_i) \leq N * (2^{1/N} - 1)$$

Note that the right term of the comparison depends only on N. The following table gives some of the values of this term

N	Utilization bound (%)
1	100.0
2	82.8
3	78.0
4	75.5
5	74.3
10	71.8
∞	69.3

For the following simple example:

Process	Period	WCET	Priority	Utilization
A	50 ms	12 ms	1	0.24
B	40 ms	10 ms	2	0.25
C	30 ms	10 ms	3	0.33..

the utilization C_i / T_i is easily computed. Applying the equation

$$\sum_{i=1}^{N} (C_i / T_i) \leq N * (2^{1/N} - 1)$$

results in 82.3% compared to a bound of 78.0%. The test fails.

Schedulability is not guaranteed by this test.

Drawing the time lines of these three tasks shows that in fact process A would not meet its deadline. If process B were to consume only 8 ms, overall utilization would drop to 77.3% which is sufficient to guarantee schedulability of the process set.

For the following simple example:

Process	Period	WCET	Priority	Utilization
A	80 ms	40 ms	1	0.50
B	40 ms	10 ms	2	0.25
C	20 ms	5 ms	3	0.25

the overall utilization is 100% and, hence, the test fails. Yet, a time-line analysis shows that the process set can be scheduled so that it meets all deadlines (barely). Hence, the test is a sufficient but not a necessary criterion.

A short assessment of utilization-based schedulability analysis:

- A simple test that can be evaluated by hand even for fairly large N.
- If successful, meeting the deadlines is guaranteed.
- It is a conservative test, i.e. a process set may be schedulable even though the test failed.
- The test (criterion) is applicable only if the processes are independent of each other (or if there is a guarantee that no process is ever blocked other than by waiting on the event controlling its period timing).
- Sporadic processes cannot be handled easily by this test.

- For interesting values of N and a successful test, the achievable overall utilization is somewhat disappointing (especially since the WCET also introduces some slack). The processor will be idle at least 25-30% of the time.

- The test merely provides a yes/no answer; it does not tell us about worst-case response times of processes other than that they are below the respective periods.

And so we look for more precise criteria with less restrictions

and more informative answers ...

5.6.3 Response Time Analysis

This analysis determines the worst-case response time of each process and then compares it with the process deadline to establish whether the deadline is met.

The presented analysis establishes a sufficient and necessary criterion for schedulability, i.e. if the response time of any process exceeds its deadline, the process set cannot be scheduled under priority-based preemptive scheduling.

The response time R of the process of the highest priority is easily determined. It is equal to its WCET (C), since the process cannot be preempted and will always run to completion.

That is,

$$R_1 = C_1$$

The general response time of the i-th process is given by:

$$R_i = C_i + I_i$$

where I_i stands for the maximum interference that the i-th process can experience in any time interval [t, t+R_i). Note the open-end interval denoted by ")". Such interference comes from higher-priority processes executing during this interval, thus preventing the processor from being available to the i-th process.

Within the duration R_i the j-th process (of higher priority) will be released times $\lceil R_i / T_j \rceil$, where the ceiling function is applied to the quotient and T_j denotes the period of the j-th process.

Thus, the maximum interference by the j-th process is

$$\lceil R_i / T_j \rceil \times C_j$$

Example: if R_i is 90ms, T_j is 40 ms, and C_j is 8 ms, the j-th process can be released 3 times, at times t, t+40, and t+80 and cause maximum interference of 3 x 8 ms = 24 ms.

The overall interference on the i-th process is the sum over the interference caused by all processes of higher priority than the i-th process:

$$I_i = \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times C_j$$

where hp(i) are the indices of processes with higher priority.

Substituting this value in the response time formula, we obtain

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times C_j$$

This is a fixed-point equation. We are looking for the smallest R_i that satisfies the equation.

Computationally this can be solved by the recurrence relation

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil w_i^n / T_j \rceil \times C_j$$

iterated upon until the w_i stabilizes or until $w_i > D_i$, where D_i is the relative deadline of the i-th process (currently $D_i = T_i$). The latter case indicates that the response time will exceed the deadline of the process, in which case the schedulability test has failed.

The w_i^n series corresponds to extending a window over a time line in which the process executions are mapped out. Interference may force us to extend this window to accommodate any remaining computation time of the process, hopefully before the deadline or period of the process is reached.

(If we are interested in response time even though the deadline is failed, e.g. for soft deadlines D, D < T later on, we can extend the window up to T rather than D.)

Example:

Process	Period	WCET	Priority
A	7 ms	3 ms	3
B	12 ms	3 ms	2
C	20 ms	5 ms	1

R_A = 3 ms (trivially)

R_B = 6 (suffering interference of 3 ms from A; one iteration over the recurrence relation suffices to obtain a value that balances the equation)

R_C = 20 (going through a sequence of 11, 14, 17, 20 before the recurrence relation stabilizes - barely in time)

The actual scheduling sequence is the following:

0-3	3-6	6-7	7-10	10-12	12-14	14-17	17-18	18-20
A	B	C	A	C	B	A	B	C

The example that failed the utilization-based test because of its 100% overall utilization succeeds in the response time test with the values noted in the following table:

Process	Period	WCET	Priority	Utilization	Response
A	80 ms	40 ms	1	0.5	80 ms
B	40 ms	10 ms	2	0.25	15 ms
C	20 ms	5 ms	3	0.25	5 ms

Assessment of the response time analysis as presented:

- it produces values for the worst-case response times for each periodic task and, as a side benefit, provides a necessary and sufficient criterion for schedulability.

- the computation isn't quite as easy as utilization-based analysis; for large N, it is worth having a program for it.

- so far, it is afflicted with many simplifying assumptions

Within the framework of these assumptions (primarily the assumption that no process is ever blocked other than on its period), there is in fact a much easier necessary and sufficient schedulability test, as described below. However, for response time analysis, these assumptions can be eliminated; for the simple algorithm below, they cannot.

Under the given assumptions, priority-based preemptive scheduling is an activity that is fully deterministic in process choice and the time of context switches. Consequently, we can draw a timeline, such as the one on the previous page, and map out the sequence of executing processes, including the preemptions, as they will be scheduled in the running system. This algorithm is exact (up to WCET slack), very simple and efficient. The above mathematics are but a less illustrative method to arrive at the same results.

5.6.4 Inclusion of Sporadic Processes

To include sporadic processes in the response analysis, we need to know the minimum inter-arrival interval of each such process. The average inter-arrival interval is also useful. (A sporadic process is an aperiodic process for which such values have been reliably established.) We will use these values as substitutes for T, the period of these tasks, even though sporadic processes do not really have a defined period.

Sporadic processes are often the result of an event that needs urgent handling. Thus, the priority is high and the deadline might be short, almost certainly much shorter than the average inter-arrival interval. Thus, relative deadlines D of such processes, so far set equal to the period T, should now be less than T. Anticipating a result of the next section, we assign higher priorities to the processes with shorter deadlines, respectively.

We then include the sporadic processes in the response time analysis with a period T equal to their respective minimum inter-arrival time, thus simulating a worst-case scenario. As a result, we obtain the worst-case response times of all processes for a situation at run-time in which the sporadic processes are scheduled at their highest frequency at a time coinciding with the critical instant of the system.

If successful, the check will guarantee that all deadlines will be met even at the busiest times.

(Unlike the critical instant for periodic processes, this worst-case scenario may never happen in the running system.)

As sporadic processes may exhibit “burst” characteristics, i.e. show up with a high frequency for a short period of time while being infrequent otherwise, the minimum inter-arrival interval as value of T may yield very pessimistic response times for lower-priority tasks. Combined with the conservative effects of WCET, a process set that passes the schedulability test may, in actual operation, yield rather low processor utilization.

If the analyzed system contains processes with soft deadlines, a more pragmatic approach to response time analysis and schedulability can be considered, based on the following rules:

1. All **processes with hard deadlines** must be schedulable using C = WCTE and T = **minimal** inter-arrival intervals of sporadic processes. For other processes, we allow the missing of deadlines in this response analysis.
2. **All processes** must be schedulable using **average** inter-arrival times of sporadic processes (and possibly even using C = average execution time).

Under these rules, there may be soft deadlines that are occasionally missed. This condition is called a **transient overload.**

Rule 1 guarantees that, even under transient overload, no hard deadlines are missed. Rule 2 ensures that the overload will indeed be transient.

NB: If transient overloads are allowed and the missing of soft deadlines causes fault tolerance mechanisms to set in, care must be taken in testing the system and in determining the minimum inter-arrival times of processes that might be started during transient overloads. There may be "negative feedback", i.e. bursts of sporadic processes trying to handle the faults and exacerbating the overload.

5.7 Deadline-monotonic priority assignment

For D = T, rate monotonic priority assignment is optimal. However, D = T often is not a realistic reflection of the deadlines imposed on processes. Much more often, more stringent deadlines D, D < T, are imposed by the application on both periodic and sporadic processes.

Intuitively, it would be plausible that tasks with the most demanding deadlines get the highest priority.

This leads us to a refinement of rate-monotonic priority to **deadline-monotonic priority order** of processes (**DMPO**).

The assignment scheme is equally simple:

The shorter the relative deadline, the higher the priority.

The response time analysis of Section 5.6.3 remains unchanged (it fails on D rather than T, anticipating the now apparent difference).

For D = T, assignment by DMPO is the same as rate-monotonic priority assignment. For D < T, however, DMPO is superior to rate-monotonic priority assignment and, in fact, **equal or superior to any other fixed priority scheme**. For the interesting proof of the latter fact, see Burns, 13.8.1., p. 414.

The following example shows the priority assignment by DMPO and the computed response times:

Process	Period	Deadline	WCET	Priority	Response	(RMA)
A	20 ms	5 ms	3 ms	4	3 ms	10 ms
B	15 ms	7 ms	3 ms	3	6 ms	7 ms
C	10 ms	10 ms	4 ms	2	10 ms	4 ms
D	20 ms	20 ms	3 ms	1	20 ms	20 ms

The “(RMA)” column shows the response times obtained from a rate-monotonic priority assignment. Here process A would miss its deadline.

Assessment of DMPO:

- under the assumption of independent processes (or, less stringently, the assumption that no process blocking occurs other than on period timing), DMPO delivers the optimal priority assignment for a preemptive scheduling based on fixed priorities
- with some minor adjustments to the formulas, system overhead for scheduling and context switches can be included in the calculation

but the bad news is....

- unfortunately, the assumption of non-blocking processes is unrealistic for many applications in which the processes need to access shared resources or wish to interact or communicate in other ways.

We will return to the implications of blocking behavior on scheduling in Section “ Scheduling – „Take Two” ”...

Real-Time Programming

Chapter 6
Concurrency

6 Concurrency

6.1 Revisiting Deterministic Behavior

In Chapter 2 we identified a number of causes for non-deterministic functional behavior of programs, e.g. rounding effects, order of evaluation in the presence of side-effects, and other situations in which the language semantics are not deterministic. While the behavior of the program is no longer fully predictable at the source level then (and may change as the program is recompiled), in almost all cases it is nevertheless deterministic at the execution level.

Concurrency adds another dimension to non-determinism. If concurrent activities may have influence on each other‘s functional behavior, **results may differ depending on timing characteristics** of these activities. Unless these influences are properly controlled by means of synchronization, the functional behavior becomes **non-deterministic at both the language and the execution level.** Sometimes non-deterministic results within a bounded range represent "don’t care" elements of program design and hence are acceptable outcomes, e.g. we may not care in which order certain service requests are honored, but we do care that they are eventually handled. More often, non-deteministic results are the result of unintentional coding errors, resulting in intermittent faults. Some examples will follow...

On the other hand, bounded **non-determinism of timing characteristics** is often desirable or even essential for the application. Deadlines establish the bounds.

6.2 Fine-Grained Parallelism

1. collateral construct ',' (Algol68):

S_1, S_2, S_3

The three statements can be executed in any order or in parallel, while ";" in

S_1; S_2; S_3;

implies sequential execution.

Caution: Data dependencies! e.g.

```
n := 7; n := n + 1, n := 17
```

is non-deterministic: n = 8 or 18 or 17

The statements

```
n := n + 1, m := m + 2
```

are a deterministic use of the collateral construct. Note that its functional behavior is now indistinguishable from ";".

2. „cobegin" or "par"-construct:

```
cobegin        -- Semantics as for the collateral construct
    S1;   S2;   S3;
coend;
```

While fine-grained parallelism is not (yet) a common paradigm in real-time programming, these simple examples illustrate an important principle:

Whenever there is concurrent modification of data shared by concurrent units (more generally: of any shared resource), non-determinacy results.

Such situations are called ***race conditions***, as the outcome depends on which concurrent unit "gets there first".

6.3 Coarse-Grained Parallelism

Parallelism comes in three flavors:

pseudo-parallel:

multiple program units are concurrently activated but at any point in time only one unit is executing. This is not real parallelism, merely a sequential simulation of concurrency. If supported at the language level, control is explicitly passed between these units by using constructs of the appropriate semantics. (=> Coroutines)

physically parallel:

multiple units are executing concurrently.

=> important implications for the compiler or kernel implementer (or any other implementer of basic synchronisation and communication primitives)

=> for the user of a language with such primitives, equivalent to logically parallel execution

logically parallel:

multiple units may be executing concurrently. The implementation may be on the basis of pseudo-parallel or physically parallel execution.

Concurrent program units:

- Coroutines -- only pseudo-parallel (↾ Simula)
- specially designated procedures (↾ Modula, PL/I)
- special constructs: „tasks" (↾ Ada)
- (main) procedures as OS processes or threads

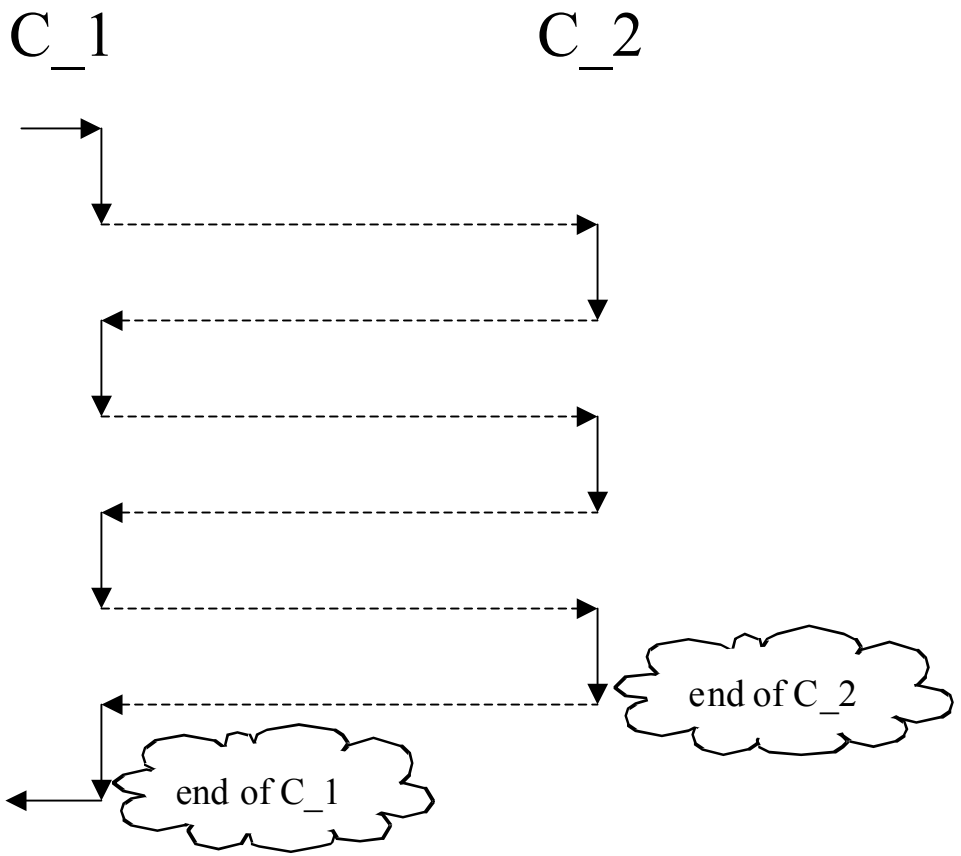
6.3.1 Coroutines

Idea:

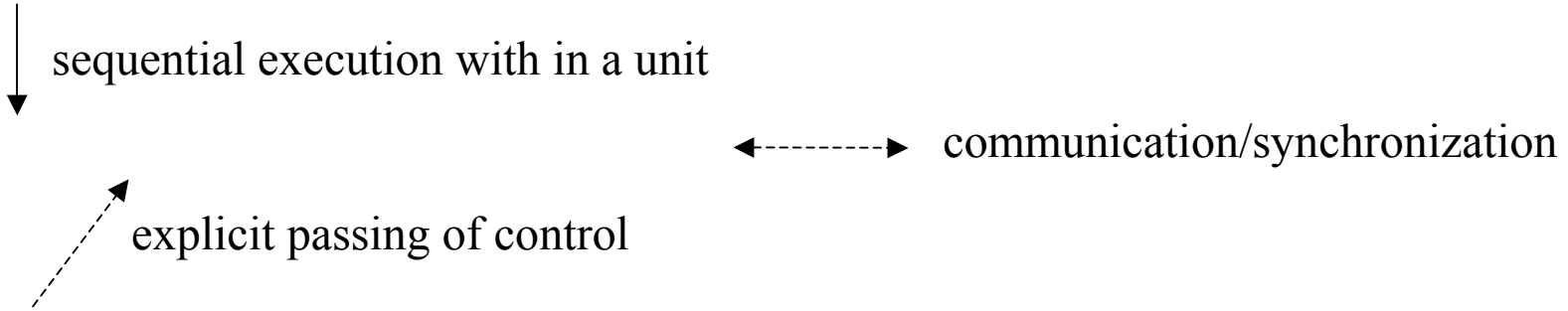
- coroutines operate as symmetrical units that, after their initial activating call, pass control back and forth among each other

- when control is passed to a coroutine, it continues at the point where it last relinquished control

- the coroutine ends when it reaches the end of its body (there is no requirement that it ever will reach its end)

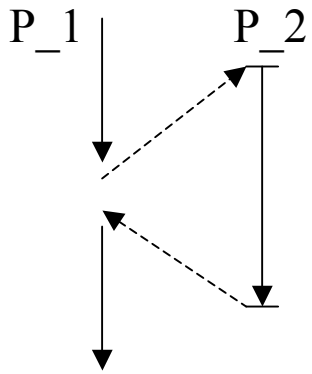

C_1
C_2
end of C_2
end of C_1

legend:

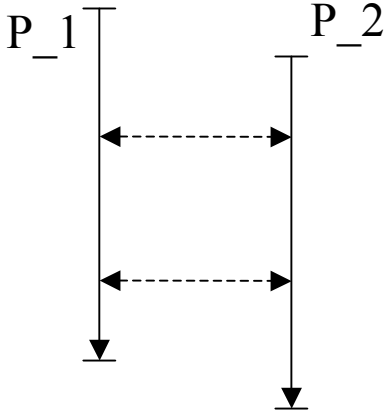


The coroutine principle is different from ...

subprogram principle



process principle



6.3.2 Coroutines in SIMULA 67

operations:

- *detach*: suspends execution of coroutine and returns to caller (after initialization)

- *resume*(C): suspends execution of coroutine and continues the execution of the coroutine C at the point of its most recent suspension

Example (symmetric producer-consumer problem)

```
class producer (consumerptr);
      ref (consumer) consumerptr;
            begin
                  integer stuff;
                  detach;
                  while true do
                        begin
                        .
                        .       **** produce stuff  ****
                        .
                        buf := stuff;
                        resume(consumerptr);
                        end
            end;
```

```
class consumer (producerptr);
        ref (producer) producerptr;
                begin
                        integer value;
                        detach;
                        while true do
                              begin
                              value := buf;
                              .
                              .       ****  consume value  ****
                              .
                              resume(producerptr);
                              end
                end;

        **** Master-Unit ****
                begin
                        integer buf;
                        ref (producer) prod1;
                        ref (consumer) cons1;
                        prod1 := new producer(cons1);
                        cons1 := new consumer(prod1);
                        resume(prod1)
                end;
```

6.3.3 Implementation of Coroutines

Observation: *One* stack of activation records isn't enough. Reason:

- Upon leaving a coroutine, one cannot pop the stack and reclaim the space for the activation record, since the locals must still be there when the coroutine is resumed. Only when the coroutine ends can we reclaim its activation record.

- The coroutine can activate other program units, e.g. normal sub-programs which require a stack.

therefore .....

- when switching control between coroutines, the state of execution (register contents, etc.) and particularly the code address where to continue the coroutine when resumed needs to be saved; the state of the coroutine that is resumed needs to be restored. This is closely related to a context switch among processes or threads.

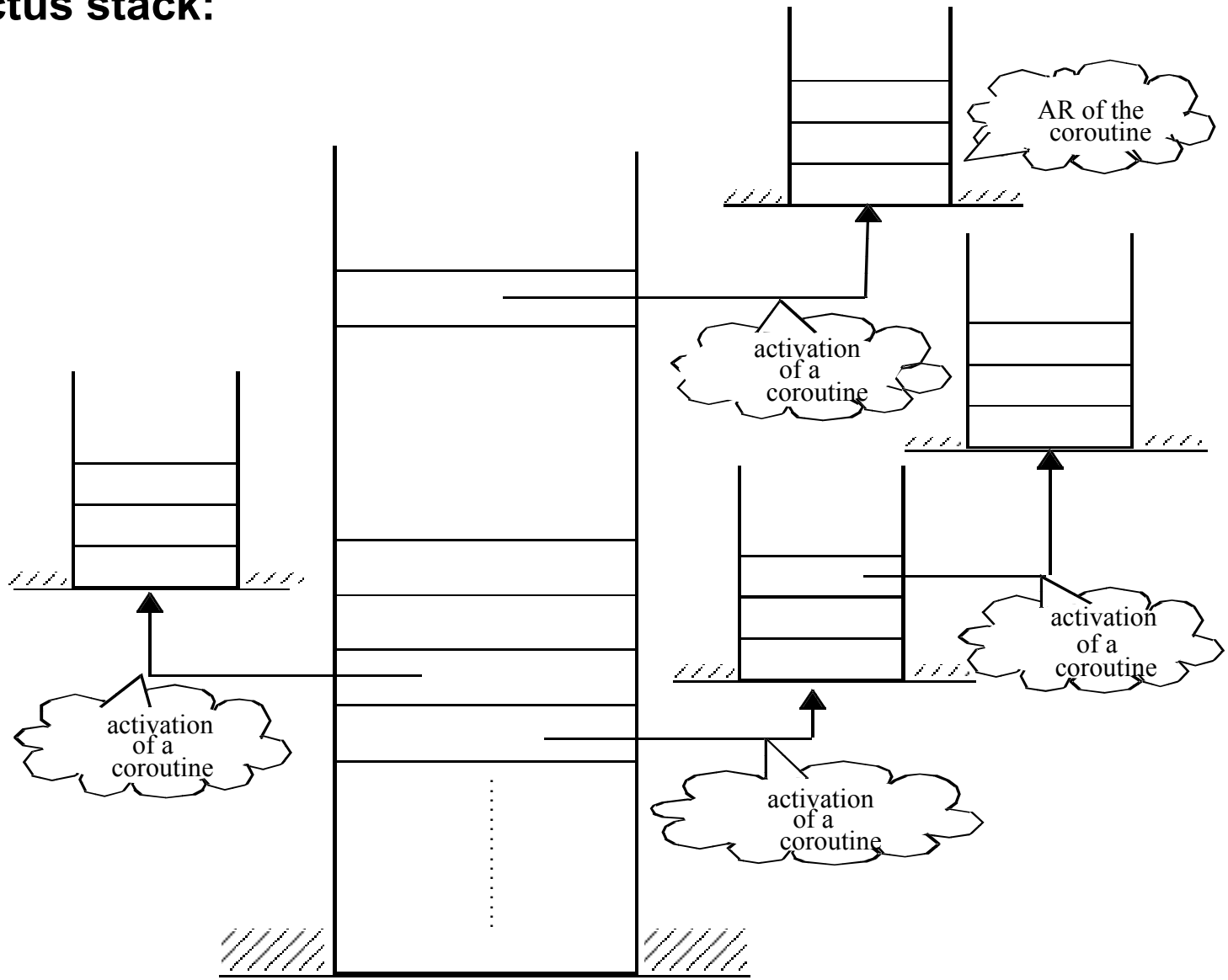
- secondly, we must find an alternative to the "one stack, one heap" model of memory management.

The answer is a so-called "**cactus stack**".

While this is a nice mental model, we have to deal with the fact of linear memory space. The cactus stack needs to be linearized. Problem: We need to know the worst-case size of each arm of the cactus in order to safely map it into linear space.

This problem is not unique to coroutines. It is common to all implementations of concurrent units. The only variation is in the shape of the cactus.

The cactus stack:



6.3.4 Assessment of Coroutines

- no physical parallelism possible
- if viewed as logically parallel, then "scheduling" is part of the co-routine design, since each resume identifies the coroutine that continues execution
- no opportunity to use CPU cycles when the code is waiting for an event to happen
- no notion of periodic or deadline-oriented processing
- difficult to understand the flow of control when many coroutines are involved ("spaghetti flow")
- difficult to model asymmetric communication behavior, e.g. "mailbox" models (see later)

- no race conditions, e.g., for accessing shared data
- simple concept
- an early approach to encapsulate data and state information that needs to be preserved from one "call" (i.e. resume) to the next

6.4 Routines as Concurrent Units

6.4.1 Fork and Join

The UNIX "fork-and-join" paradigm, shown here in pseudo code:

```
function F(...) return T is ....
function G(...) return T2 is
        begin
                ...
                C := fork F(...);
                ...
                J := join C;
                ...
        end;
```

Between "fork" and "join C", functions F and G will execute concurrently. At the join point, G waits for F if F is not yet completed. If F completed earlier, it waits until it can deliver its result to G.

"fork" copies the entire process environment at the fork point for the execution of F, so that F and G cannot communicate via shared variables but can access their own copies of the data.

"fork-and-join" is quite reminiscent of the "cobegin...coend" (or "par") construct, where N concurrent units can be started and joined at the "coend". However, these constructs do not provide separated environments for the execution of their concurrent components. "fork-and-join" is much less structured than "cobegin ... coend"; there is no syntactically enforced guarantee that the join wasn't forgotten (making F wait forever).

6.4.2 Unix Threads

Unix threads are very similar to "fork-and-join". "fork" is now the "pthread_create" function, "join" is the "pthread_join" function.

The major difference is that the data environment is not copied, so that the thread creator and the new thread operate on shared data, either directly accessible or passed via parameters.

There are minor differences, mostly in terms of some very useful additional interfaces to influence threads (stack size and other attributes can be set that influence thread behavior, unique identification of threads, abortion and asynchronous failure signals, "detached" (i.e. non-joinable) execution, and so on).

See Burns, p. 190, for the standardized interfaces.

A couple of caveats:

1. Processes that are forked by a thread contain a copy of only the creating thread, not of the entire process. (POSIX standard)
2. If a process (main program) terminates, all the threads in it terminate automatically. A main program, used only to start up multiple threads, needs to join the threads before it ends.

```
int main() {
  p_thread xp, xy;
        arg X,Y;
        void mycode(arg *a) {...};
        ... /* set the arguments X and Y */
        pthread_create(&xp, 0, (void *)mycode, &X);
        pthread_create(&xy, 0, (void *)mycode, &Y);
            /* without the joins, xp and xy would terminate now */
        pthread_join(xp, (void **)&result);
        pthread_join(xy, (void **)&result);
}
```

6.5 Language-supported Concurrent Units

Apart from language implementation considerations, (ab)using routines for expressing concurrent units is a poor software engineering choice. Consider:

```
procedure P is
begin
    while true loop
        ... -- complicated code
    end loop;
end P;
```

For a regular subprogram, this is highly suspicious code. Is the infinite loop really intended? However, for a periodic task, this code is a very frequent paradigm.

Then, in turn, can we call this interface P by a regular subprogram call as well? The language will not hinder us to do so. Of course, the call will never end and return to the caller. Do we have to read the code of all subprograms to decide whether they are intended as concurrent or as sequential units?

(Ugggh, get serious! Yet such is life in C and RTOS)

So, even if no semantics beyond creation of a process or thread for the execution of the subprogram body are provided by the language, it is a good idea to identify the concurrent units as such, e.g. by replacing the keyword ’procedure’ by ’process’. This is done for example in Modula.

(If you are required to use a language that doesn’t, PLEASE write a comment for the subprogram specification to this effect.)

6.5.0 Java Threads

Java threads are semantically very similar to Unix threads. Threads are instances of classes that implement the „runnable“ interface. The thread code is the „run“ method of the class.

```
class PrimeThread extends Thread {  // or „implements runnable“

   long minPrime;  // declarations comparable to thread-specific state

   PrimeThread(long minPrime) {  // constructor initializes the state
       this.minPrime = minPrime; }

   public void run() {  // the code to be executed as a thread  ...
                      }
 }

PrimeThread p = new PrimeThread(143); // creates the object and thread
p.start();  // starts the thread executing ‚run‘
```

6.5.1 Ada Tasks

Ada provides explicit constructs for concurrent units which are called "**tasks**" (or, more precisely, task objects).

```
procedure P is
    task A;     -- a task declaration
    task B;     -- another one
    task body A is
            -- local declarations go here
    begin
            -- the statements that the task executes;
            -- they may contain other task declarations
    end A;
    task body B is ... end B;
begin   -- here A and B are activated, i.e. start executing
        -- ... while P also executes its statements logically in parallel
end P;  -- here P waits until A and B have completed
```

A and B are called task objects. The above is actually an abbreviated syntax for declaring a **task type** and an object of this type. If we want more than one task object of this task type, we need to declare a task type.

A task type declaration ....

```
task type MyTask;      -- a task type declaration

task body MyTask is
                -- local declarations go here
        begin
                -- the statements that any task of this type executes;
                -- they may contain other task declarations
        end MyTask;
```

... and its usages:

```
type Rec is record
        arg: some_type;
        actor: My_Task;  -- this is fine!
         end record;
type Task_Ptr is access MyTask; -- fine too

A,B: MyTask;   -- two tasks of this type
Arr: array(1..10) of MyTask; -- 10 such tasks
My_Rec, Your_Rec: Rec; -- each containing a task
P: Task_Ptr; -- not yet designating a task

begin -- 14 tasks are activated here

...
P := new MyTask;   -- immediately activated here
...
A := B; -- illegal, no assignment for tasks
P := Q; -- o.k., access type assignment
```

Aside:

```
task type MyTask;    (and task MyTask;)
```

will normally be written as

```
task type MyTask is
   ...   -- certain declarations for the communication
         -- interface of the task
end MyTask;
```

Global tasks:

```
package P is
  task A,B;
end P;

package body P is
        task body A is ... end A;
        task body B is ... end B;
begin -- here A and B are activated
...
end P;
```

6.5.2 The Master Concept (for Awaiting Termination)

For each task object, a "master" is defined by the language

- The master of a locally declared task object is the block, subprogram, task or entry body, or the accept statement in which the declaration occurs.
- The master of a global task is a conceptual environment task whose body calls the main program.
- The master of an allocated task (via "new") is the master in which (the ultimate ancestor of) the access type of the result returned by the allocator is declared.

Each master awaits the termination of the tasks of which it is a master before completing.
Tasks terminate when they reach the end of their body, or are aborted, or have indicated their willingness to terminate jointly with all other tasks of their master (see "select" later).

6.5.3 Implementing Periodic Tasks in Ada

To achieve periodic execution, we need to await timing events. Busy-waits are out of the question.

Ada provides two simple primitives to initiate and await timing events: "**delay**" and "**delay until**". Also, there are interfaces to the Ada.Calendar.Clock and to the Ada.Real_Time.Clock.

"**delay**" suspends the task for at least the duration of its argument. This is a **relative delay**. It delays "at least" that long because the process, when unblocked, might not be scheduled immediately. (A higher priority process might execute.)

```
package ART renames Ada.Real_Time;
Cycle_Length: constant ART.Time_Span := ART.Milliseconds(25);
Start, Next_Time: ART.Time;
use type ART.Time; -- to make "-" visible
begin
        Start := ART.Clock;   -- get the current time
        loop
        delay ART.To_Duration(Cycle_Length - (ART.Clock - Start));
                Start := ART.Clock;
                ... -- the actions to be performed in each period
        end loop;
```

This code tries to adjust for differences in execution time in each period by delaying only for the rest of the period. Because of the effects of the **local drift** caused by the (unavoidable) "at least" semantics, we might see actual execution start times at

0, 27, 54, 80, 110, 135 ...

Note the monotonous increase of the difference versus the desired but generally unachievable sequence

0, 25, 50, 75, 100, 125, ...

To prevent this effect of a **cumulative drift** of the schedule, Ada provides "**delay until**" to achieve a delay until an **absolute** time value.

Our example might read as follows:

```
Next_Time := ART.Clock;
loop
delay until Next_Time;
Next_Time := Next_Time + Cycle_Length;
   ... -- the actions to be performed in each period
end loop;
```

Although we will still experience the unavoidable local drift, there is no cumulative drift anymore, since the local drift will be compensated by the Next_Time calculation. We might see the task scheduled at:

0, 27, 51, 80, 100, 127, ...

Now we have all the means at hand to implement independent tasks, using the schedulability analysis of Chapter to ensure that the dead-lines are met ...

... but, unfortunately, many tasks require communication with other tasks or at least a sychronization with them.

And so we enter a new and difficult chapter...

Real-Time Programming

Chapter 7

Communication and Synchronization

7 Communication and Synchronization

7.1 The Problem

With the means provided by the previous chapters, we might be tempted to program:

```
type Item_Type is ...
type Item_Array is array(Positive range <>)  of Item_Type;

Data : Item_Array(1..Buffer_Size);
Count: Natural := 0;   -- counting the filled elements
In_Index, Out_Index: Positive := 1;

task type Producer;
task type Consumer;
```

```
task body Producer is
        Item: Item_Type;
begin
        ... -- produce an item here
        loop
          if Count >= Buffer_Size then
                delay 0.5; -- Buffer is currently full
          else
                Data(In_Index) := Item;
                In_Index := In_Index mod Buffer_Size + 1;
                Count := Count + 1;
                ... -- produce a new item
          end if;
        end loop;
end Producer;
```

```
task body Consumer is
        Item: Item_Type;
begin
        loop
          if Count = 0 then
                delay 0.5; -- Buffer is currently empty
          else
                Item := Data(Out_Index);
                Out_Index := Out_Index mod Buffer_Size + 1;
                Count := Count - 1;
                        ... -- consume the item
          end if;
        end loop;
end Consumer;
```

```
MacD: Producer;
Teen: Consumer;
-- or maybe even ...
Cooks: array(1..5) of Producer;
Gang: array(1..10) of Consumer;
```

Of course, this code is utterly disastrous causing a variety of intermittent faults. Enumerating the race conditions and resulting errors is left to the reader.

While we are very interested in exploiting the non-determinism caused by items being produced and consumed concurrently at varying speed (and in balancing the system by adding more producers if needed), the non-determinism in the accessing of the global data will cause havoc in the execution of these tasks. We need to ensure consistency of these data by synchronizing the accesses by the tasks.

7.2 Atomic Actions

Our problem would disappear if the three statements dealing with shared resources in the tasks were executed as one single, indivisible instruction which, unfortunately, is not possible. Still, we call this activity an **atomic action** and will ensure that its instructions behave as if they were executed in this way. The code that achieves the effects of the atomic action is called a **critical region**.

The indivisibility of an atomic action does NOT imply a priori that its execution cannot be interleaved or be concurrent with the execution of another task. In real-time systems, this would break the notion of preemptive scheduling and response analysis, as atomic actions could take a fairly long time to complete and could not be preempted.

A more appropriate definition of an atomic action is that it implies that no task can read or modify any interim state of the shared resources modified in an atomic action of another task and vice versa (concurrent read-only accesses are not a problem).

As we will see later, in some clearly isolated cases we will have to make an atomic action truly indivisible, i.e. it cannot be preempted or interrupted.

Our main programming mechanisms to provide the guarantee of atomicity of actions are low- and high-level sychronization constructs.

7.3 Synchronization Primitives

7.3.1 Semaphores

There are two kinds of semaphores used for different purposes:

- **binary semaphore** ("**mutex**", for mutual exclusion)
- general or **counting semaphore** (for counting free or occupied limited resources)

A **binary semaphore** can be viewed as an abstract data type represented by a record structure

```
type Mutex is record
                        Flag : Boolean;
                        Queue : queue_of_process_id;
        end record
```

There are two operations, P and V, available for mutexes with the following semantics:

```
P(M)    ≡       if not M.Flag then
                      M.Flag := true;
                else enqueue the calling process on M.Queue and switch
                     context to another process

V(M)    ≡       if not Empty(M.Queue) then
                      dequeue a process from M.Queue
                      (and make it runnable)
                else M.Flag := false;
```

These operations can be used to guarantee exclusive access to a resource shared by concurrent units.

The resource is originally made available either by initializing the M.Flag to “false” or by executing V(M). The flag indicates whether another process holds the exclusive resource.

Example:

```
May_I : Mutex;
        ...
                P(May_I);
                Data(In_Index) := Item;
                In_Index := In_Index mod Buffer_Size + 1;
                Count := Count + 1;
                V(May_I);
```

With this modification to our producer task and an identical one to the consumer task, we achieve that the tasks will no longer be able to cause quite such random havoc. The buffer accesses are now **atomic actions**, i.e. all subactions happen before the action can occur again. There still is a problem on a full or empty buffer, though...

A counting semaphore is a generalization of the mutex, replacing the flag by a counter:

```
type Semaphore(Init: Natural) is record
                Count : Natural := Init;
                Queue : queue_of_process_id;
        end record
```

The P and V operations have the following semantics:

```
P(S) ≡ if S.Count > 0 then
         S.Count := S.Count - 1
         else enqueue the calling process on S.Queue, etc.

V(S) ≡ if not Empty(S.Queue) then
                dequeue a process from S.Queue, etc.
        else S.Count := S.Count + 1
```

The initialization of the semaphore S needs to set the count to the maximum number of free resources.

An alternative formulation of P and V for general semaphores (with "Count: Integer := Init;") could easily count the number of queued processes:

```
P(S)  =  S.Count := S.Count - 1;
         if S.Count < 0 then
            enqueue the calling process on S.Queue, etc.

V(S)  =  S.Count := S.Count + 1;
         if S.Count <= 0 then
            dequeue a process from S.Queue, etc.
```

For negative S.Count, -S.Count is the number of processes in the queue.

We can now improve on the somewhat unsatisfactory earlier solutions that our producers wake up every 0.5 seconds to check whether they can deliver their products (and might be misinformed, too, by the race condition on checking “count”) ....

```
type Item_Type is ...
type Item_Array is array(Positive range <>)      of Item_Type;

May_I: Mutex;
Space_Available: Semaphore(Buffer_Size);
Products_Available: Semaphore(0);

Data : Item_Array(1..Buffer_Size);
In_Index, Out_Index: Positive := 1;

task type Producer;
task type Consumer;
```

```
task body Producer is
        Item: Item_Type;
begin
        -- produce an item here
        loop
                P(Space_Available);
                P(May_I);
                Data(In_Index) := Item;
                In_Index := In_Index mod Buffer_Size + 1;
                V(May_I);
                V(Products_Available);
                        ...  -- produce a new item
        end loop;
end Producer;
```

```
task body Consumer is
        Item: Item_Type;
begin
        loop
                P(Products_Available);
                P(May_I);
                Item := Data(Out_Index);
                Out_Index := Out_Index mod Buffer_Size + 1;
                V(May_I);
                V(Space_Available);
                        ... -- consume the item
        end loop;
end Consumer;
```

We now have an orderly solution that will work as expected.

Or won't it? The semaphores are, after all, shared resources of the tasks. Have we simply pushed the problem by one level without actually solving it?

And indeed we have, but at least we have minimized the amount of code to be protected against interference.

"Kernel implementers to the rescue ..."

7.3.2 Implementation of Semaphores

It is clear that the semaphore semantics as given will work correctly only if they are executed as actions that cannot be interfered with by actions of concurrent units. They need to be **atomic actions** (without the help of semaphores).

On a uniprocessor (i.e. for pseudo-parallel execution) this is not very hard to do:

- first, the scheduler must not preempt the running task during these atomic actions. They are **non-preemptible operations**. (Thus, "*immediate* preemption" in scheduling isn't quite true but has a tightly bounded latency.)
- second, any external interrupts (or at least those that have any opportunity to interfere with the atomic action) must be disabled for the duration of the atomic action.

These conditions guarantee that no other execution will interfere with an atomic action currently executing.

On a multiprocessor system (i.e. for physical parallelism) these conditions are insufficient, as multiple units might execute P or V operations in parallel. An even more fundamental mechanism in part supported by the hardware is necessary. The latter is a mechanism to query and change an addressable memory location in one indivisible operation at a level at which the hardware sequentializes such operations.

On some machines this is a "**test-and-set**(address)" operation with the following semantics:

```
if contents(address) = 0 then
                    contents(address) := 1; return 0;
        else        return 1;
```

On other machines, a “**swap**” operation is used that stores a new value and returns the old one. “swap(address, 1)” is equivalent to the test-and-set operation.

Resetting the value to 0 is a simple “store” instruction.

The value toggled in this way is called a **lock**. A similarity to the semantics of the flag in a binary semaphore and its role in the P operation is obvious.

The hardware mechanisms can be used to implement the flag manipulations of a binary semaphore indivisibly. If the result of the hardware operation is 1 (=true), the process will be queued. The V operation opens the lock again or dequeues another process while retaining the lock for that process.

For a general semaphore, there is generally no similar direct hardware support. On a multiprocessor, its operations need to be made atomic by means of an additional lock for the semaphore. Since the durations of the P or V operations are very short, we can employ busy-waiting on the lock (rather than queuing as for a semaphore) upon entry to the semaphore operations. A lock with busy-waiting is called a "**spin-lock**".

```
while swap(lock,1) loop null; end loop;
```

The lock is released when the semaphore operation ends.

N.B. The use of spin-locks should be strictly reserved for the language or kernel implementer.

7.3.3 Deadlocks, Livelocks and Starvation

Does the order of P operations matter?

Consider our producer task with interchanged P operations:

```
task body Producer is
        Item: Item_Type;
begin
        -- produce an item here
        loop
                P(May_I);
                P(Space_Available);
                Data(In_Index) := Item;
                In_Index := In_Index mod Buffer_Size + 1;
                V(May_I);
                V(Products_Available);
                        ... -- produce a new item
        end loop;
end Producer;
```

While seemingly innocent at first glance, this is actually a fatal error. The first time no space is available in the buffer, the producer task will be queued by “P(Space_Available)" while having exclusive access to the buffer manipulation. As a consequence, no other task will be able to access the buffer, no space will be freed by consumers, and the system will come to a halt, since all tasks will be queued waiting for their turn to proceed. Such a situation in which a group of processes is blocked in a way in which none of them can possibly proceed in the future is called a “**deadlock**”.

An analogous situation in which a group of processes is spin-locked in the same way is called a “**livelock**”. If anything, it is even worse than the already fatal deadlock, since the processes are taking away CPU resources from as yet unaffected processes.

Since the two causes can combine, i.e. some spin-locked and some blocked processes, we use the term “deadlock” in all cases from here on to describe the situation.

Deadlocks can arise whenever

1. resources are exclusively assigned
2. the resource requests by a requestor are made one after another
3. once acquired, a resource is not given back when another needed resource cannot be acquired
4. there is a cycle in a graph whose nodes are the resources and whose edges represent the order of the requests by each requestor

All four conditions must hold to enable a deadlock. Deadlock avoidance strategies usually try to ensure that one of the above conditions is never satisfied.

Yet another cause for a fatal error quite similar to deadlock is the accidental omission of a V operation. For a mutex, it will make all future P operations on the mutex wait indefinitely. For a counting semaphore, it will decrease the number of available resources permanently with each execution of the matching P operation, until the P operations start waiting indefinitely.

Detecting deadlocks before they happen is a very complicated problem. A highly disciplined and well-designed use of all synchronization primitives is required to avoid deadlocks. An excellent rule is to always request resources in a globally preestablished order. However, even this property may be very hard to verify. A verified pairing of P and V operations is, of course, equally essential and can be quite difficult.

The P and V operations do not inherently specify which task on the semaphore queue to dequeue in a V operation. Such a strategy might be FIFO (first-in-first-out) which one might consider the “fair” choice. This is known as the **FIFO Queuing Strategy** (for all kinds of process queues, e.g. the queue of runnable processes).

However, when scheduling is already done by priority order to meet deadlines, it makes sense that higher-priority tasks are dequeued earlier than low-priority tasks, even if they were queued later. In this case, the queue would usually be implemented as a queue sorted by priority. This is known as the **Priority Queuing Strategy (PQS)**.

Especially with PQS, it can be the case that a low-priority process may be queued indefinitely or for an exceedingly long time because there are always higher-priority processes arriving and gaining the resource first. This is known as **starvation** or **lockout** of the process.

A workable real-time system must be free of deadlocks, livelocks, and starvation. This is the so-called **lifeness** property of the system and is to be verified.

7.3.4 Condition Variables

Another primitive mechanism for synchronization, quite similar to mutexes (and equally difficult to program) and often used as a replacement for counting semaphores, is the concept of a condition variable. Condition variables also have two operations, usually called “wait” and “signal”. In some models, there is “broadcast” as a third operation. Quite often, condition variables are also referred to as signals and the operations as signaling operations (not to be confused with POSIX signals).

For a condition variable C, a call on “wait(C)” will unconditionally suspend the calling process and queue it on this condition variable.

A call on “signal(C)” will free one of processes queued on C if there is one. The queuing discipline chooses among multiple candidates.

A call on “broadcast(C)” will make all processes currently queued on C runnable again, i.e. they can proceed as soon as they can be scheduled for execution.

The major difference between a condition variable and a mutex is that the “wait” is unconditional and when no process is queued on the condition variable, a “signal” has no effect at all. (The permission to proceed will not be “saved” for the next process executing a “wait”, as would be the case for a mutex.)

As the condition under which “wait” gets called can reasonably be presumed to be a shared resource among tasks, many models of this primitive allow or require “wait” to be called with a mutex as its second argument. The semantics are for “wait” to call V on the mutex (i.e. release the resource) and for any process resumed by “signal” to call P on the mutex (i.e. contend for the resource again before proceeding).

This is the model in C, C++ (and their standardized run-time interfaces) and, in a considerably more dangerous and inefficient variation, in Java and its API. Examples in the syntax of these languages will be shown later.

Our example turns into:

```
May_I: Mutex;
Space_Available, Products_Available: condition;
Count: Natural := 0;   -- counting the filled elements
...
task body Producer is
        Item: Item_Type;
begin
        -- produce an item here
        loop
                P(May_I);
                while Count >= Buffer_Size loop
                        Wait(Space_Available, May_I);
                end loop;
                Data(In_Index) := Item;
                In_Index := In_Index mod Buffer_Size + 1;
                Count := Count + 1;
                Signal(Products_Available);
                V(May_I);
                ...  -- produce a new item
        end loop;
end Producer;
```

```
task body Consumer is
        Item: Item_Type;
begin
        loop
                P(May_I);
                while Count = 0 loop
                        Wait(Product_Available, May_I);
                end loop;
                Item := Data(Out_Index);
                Out_Index := Out_Index mod Buffer_Size + 1;
                Count := Count - 1;
                Signal(Space_Available);
                V(May_I);
                ... -- consume the item
        end loop;
end Consumer;
```

7.3.5 Synchronization Primitives in Programming Languages

Most real-time languages provide semaphores and possibly condition variables as the most primitive form of synchronization. They vary in small details such as the names of the P and V or signaling operations and in whether or not they restrict semaphores to binary semaphores:

- ALGOL 68 (“down” and “up”)
- C/Posix (“sem_wait”, “sem_post”, “cond_wait”, “cond_signal”)
- Ada (“Wait”, “Signal” in Semaphore_Package)
- PL/I via „events” (~ condition variables)

  P-Operation ↑ WAIT (. . . event(s) . . .)

  V-Operation → COMPLETION (. . . event . . .)
- Modula-2 Extensions (“wait” and “send”/”signal”)
- Java (“wait” and “notify“/notify_all” for the equivalent of condition variables)

Programming with semaphores and signals is quite difficult:

- the order of P or wait operations is important to avoid deadlocks
- omission of a P or wait operation causes loss of synchronization and, hence, quite unpredictable results
- omission of a V or signal operation causes deadlock or at least permanent loss of resources

Modern languages with concerns for real-time programming also provide better and higher-level synchronization constructs which should be preferred over semaphores as a much safer and more understandable alternative. To a very large extent, their implementation ultimately maps transparently onto the use of binary semaphores and condition variables.

7.4 Structured Communication/Synchronization

- „conditional critical regions” (Hoare, Hansen) <77
- monitors (Concurrent Pascal, Modula, Mesa) ~77
- Protected Types (Ada) ~92
- Rendezvous (Occam, Ada) ~79

7.4.1 Conditional Critical Regions

Idea: For each non-sharable resource, identify the access code syntactically as a “critical region” protected by an implicit mutex. Evaluate conditions for access under protection of the mutex. If the condition calls for delay, free the mutex before suspending (thus avoiding a deadlock). Reactivate when the condition becomes true, reacquire the mutex, and (possibly (*) – see later –) proceed.

```
procedure receivemessage(var olditem: Message;
                         var buffer: MessageBuffer);
begin
        region buffer await buffer.size > 0 do
                buffer.size := buffer.size - 1;
                olditem := buffer.items[buffer.front];
                buffer.front := buffer.front mod capacity + 1
        od
end;

procedure sendmessage(newitem: Message;
                      var buffer: MessageBuffer);
begin
        region buffer await buffer.size < capacity do
                buffer.size := buffer.size + 1;
                buffer.items[buffer.tail] := newitem;
                buffer.tail := buffer.tail mod capacity + 1
        od
end;
```

Issues:

1. How and when is C re-evaluated for suspended processes? Note that C is usually different in different regions for the same resource

2. Note the race condition at point (*) !
   e.g. between C = true and P(X_s), another process can have successfully executed *"P(X_s); C := false; V(X_s)"*

   ⇒ re-evaluation of „await C" is indeed necessary!

   ⇒ danger of starvation even for FIFO queues!

Answers on 1:

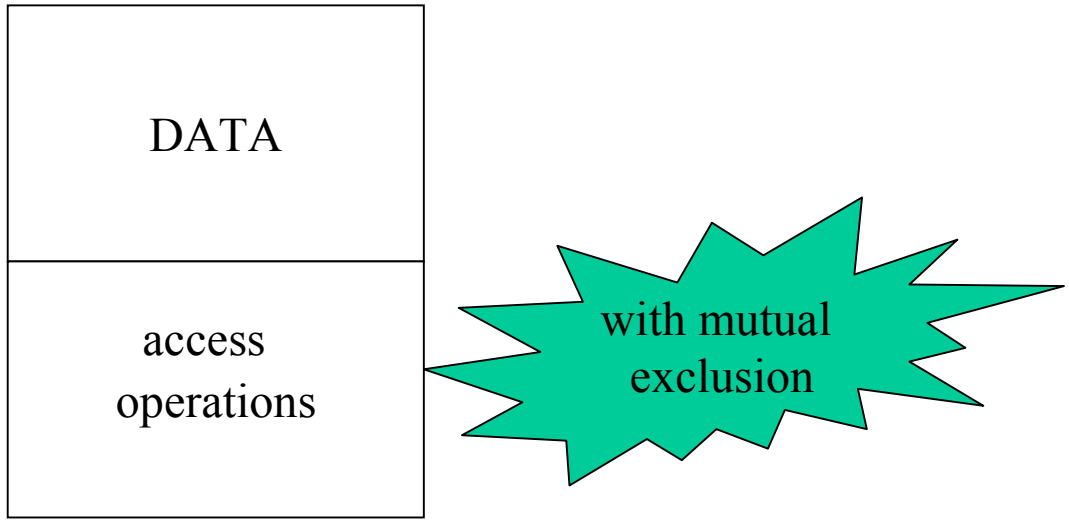
- spin-locks as implementation -- possible only for physical parallelism and even then highly wasteful
- waking up suspended processes "blindly" every time a region is left in order to check their condition C -- this implies a lot of context switching, reacquiring the resource, and re-suspending if C is still false.

Very few languages support conditional critical regions, although they are definitely an improvement over programming with semaphores. However, the critical code is distributed all over the application. Also, the protected data should not be accessible outside of these regions.

Shortly after conditional critical regions were proposed in the literature, the concept of “monitors” was developed that focused on these concerns and was adopted in many languages...

7.4.2 Monitors

Idea: provide an abstract data type with access operations that are synchronized so that they cannot be executed in parallel.



The abstract data type is associated with an implicit mutex M. On entry to an access operation, P(M) is called, on exit V(M) is called. The encapsulation mechanism ensures that all accesses to the data are done via the operations. The critical code is centralized in the definition of the monitor.

Yet, sometimes operations additionally need to wait on some conditions to be satisfied (as in our running example). The answer in most languages that support monitors is to combine monitors with condition variables or equivalent mechanisms.

Below we present the Concurrent Pascal version where in lieu of condition variables the suspension queue is exposed and manipulated. The models are isomorphic: “delay” maps to “wait”, “continue” to “signal”, with the monitor mutex being an implicit argument to the calls.

Monitors in Concurrent Pascal notation (but less restrictive semantics in allowing multiple processes to wait):

```
type MONI = monitor
                ... variable declarations for the monitored data
                SENDER, RECEIVER : queue;
                ...
                procedure entry ONE ....
                procedure entry TWO ....
                ...
        begin ... initialization of data in monitor ... end;
```

```
var BUFFER : MONI
```

- A statement

  ```
  init BUFFER
  ```

  initializes the buffer.

- monitored data can be changed only by operations of the monitor, e.g.

  ```
  BUFFER.ONE . . . .
  ```

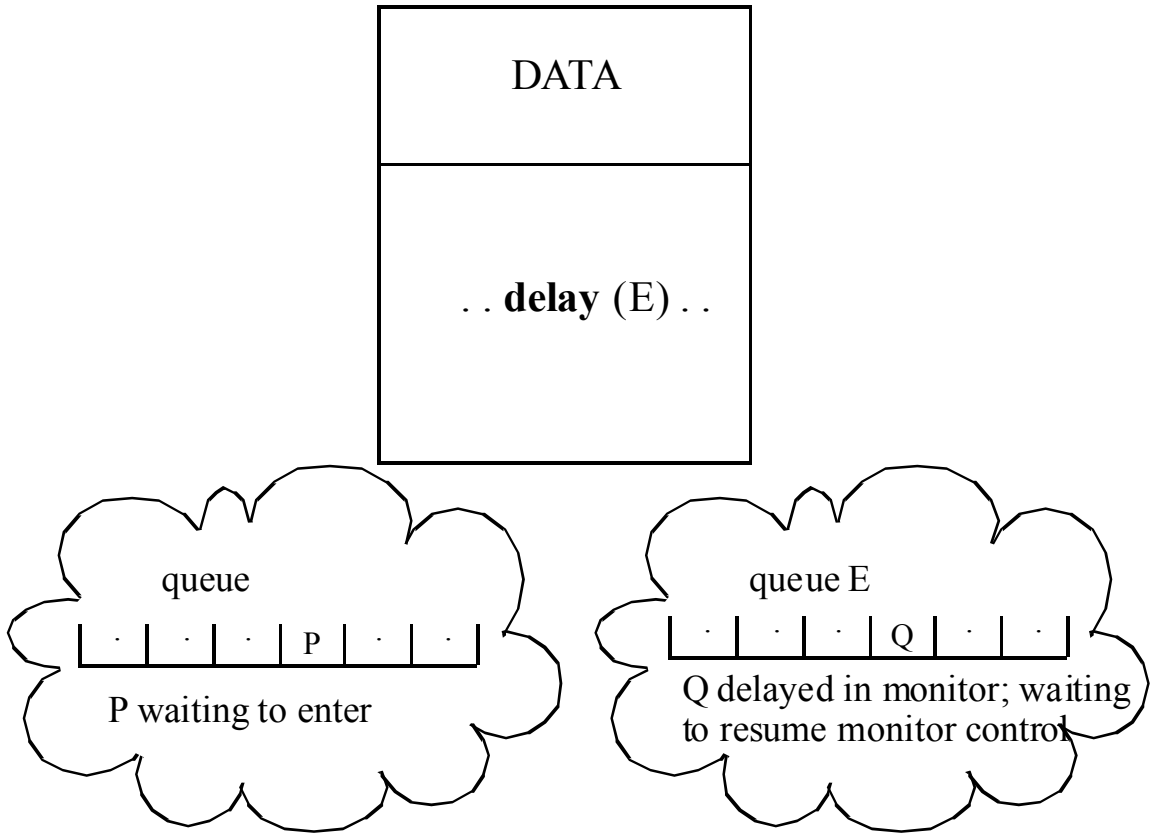
- the coordination between the access operations ONE and TWO is by means of

  - two variables of type **queue** (as data of the monitor)
  - two built-in operations **delay**(...) **continue**(...)

delay (SENDER)	queues the calling process on the SENDER queue and relinquishes the monitor
continue (SENDER)	dequeues one of the processes on the SENDER queue and passes control over the monitor to it, while ending the current call on a monitor operation

(analogously for RECEIVER)

We have two kinds of queues in monitors:



Example:

```
type databuf =
monitor
        const   bufsize = 100;
        var     buf: array[1..bufsize] of integer;
                next_in, next_out: 1..bufsize;
                filled: 0..bufsize;
                sender_q, receiver_q: queue;

        procedure entry insert(item: integer);
        procedure entry remove(var item: integer);

begin
        filled := 0;
        next_in := 1;
        next_out := 1
end;
```

... see next page

... see next page

```
procedure entry insert(item: integer);
    begin
            if filled = bufsize then delay(sender_q);
            buf[next_in] := item;
            next_in := (next_in mod bufsize) + 1;
            filled := filled + 1;
            continue(receiver_q)
    end;

 procedure entry remove(var item: integer);
    begin
            if filled = 0 then delay(receiver_q);
            item := buf[next_out];
            next_out := (next_out mod bufsize) + 1;
            filled := filled - 1;
            continue(sender_q)
    end;
```

```
type producer = process(buffer: databuf);
    var stuff : integer;
begin
        cycle
            -- produce stuff --
            buffer.insert(stuff)
        end
end;
```

```
type consumer = process(buffer: databuf);
    var stored_value : integer;
begin
        cycle
          buffer.remove(stored_value);
          -- consume stored_value --
        end
end;
```

```
var new_producer : producer;
    new_consumer : consumer;
    new_buffer : databuf;

begin
init new_buffer, new_producer(new_buffer),
     new_consumer(new_buffer)
end;
```

Some issues with monitors that are generally made the responsibility of the programmer:

1. Is it guaranteed that the condition that caused a “wait” is actually false when “signal” releases the suspended process? For efficiency‘s sake, the languages do not enforce re-evaluation. Also, there is a subtle race condition, in case the released process has to recompete for the monitor (not the case in Concurrent Pascal but in POSIX).

2. Evaluation of the condition for a “wait” ought to be the very first action in the monitor operation or else the monitored operation cannot really be regarded as an atomic action. This remains unenforced by the languages.

3. Signaling ought to be the last action in the monitor operation for the same reason as 2. and also because otherwise the caller and the resumed process are concurrently within the monitor. The languages have different approaches to deal with this issue:
   - enforcement of this restriction
   - signal implies “return”
   - the released process must compete for the exclusive access to the monitor (in which case we definitely need a „while“ on the condition, not an „if“)

For programming in a language that only provides semaphores and condition variables as the synchronization primitives, it is nevertheless a very good idea to structure the management of shared resources by means of monitors built “manually” using the available primitives but keeping the above issues in mind.

Example: A monitor expressed in C/Posix

```
#include pthreads.h
#define BUF_SIZE 10
typedef struct {
        pthread_mutex_t    mutex; /*monitors the struct */
        pthread_cond_t     buffer_not_full;  /* two condition variables */
        pthread_cond_t     buffer_not_empty;
        int      count, first, last; /* the monitored data */
        int      buf(BUF_SIZE);
}  buffer;

void put(int item, buffer *B) {
        pthread_mutex_lock(&B->mutex);
        while (&B->count == BUF_SIZE)  /* while or if ? */
                pthread_cond_wait(&B->buffer_not_full,  &B->mutex);
        /* the real buffer operations go here /*
        pthread_mutex_unlock(&B->mutex);
        pthread_cond_signal(&B->buffer_not_empty);
}
```

```
void get(int *item, buffer *B) {
        pthread_mutex_lock(&B->mutex);
        while (&B->count == 0)
            pthread_cond_wait(&B->buffer_not_empty, &B->mutex);
        /* the real buffer operations go here /*
        pthread_mutex_unlock(&B->mutex);
        pthread_cond_signal(&B->buffer_not_full);
}
```

In expressiveness and safety, this may actually be a step backwards from mutex and counting semaphores (in this specific instance where the condition is tied to a count). However, the monitor concept lends systematic structure to the synchronization template to be applied.

„Monitors“ expressed in Java

- Java provides „synchronized methods“ on objects, which implicitly generate a P operation upon call and a V operation upon return from the method on an implicit semaphore associated with each object. Also, critical regions can be synchronized on an object.

- To ensure that shared data are not accessed without synchronization, the data components need to be made **private**.

- The equivalent of a condition variable needs to be constructed by a signaling capability („wait“, „notify“, „notifyAll“) of each object that needs to be accessed under synchronization; „wait“ releases the semaphore on the object and reacquires it when notified. The synchronization is enforced **by a runtime check.**

- „wait“ing within a (transitively) synchronized method on an object other than the synchronized object causes deadlock.

A „monitor“ expressed in Java:

```
class Buffer {
   static final int BufferSize = 20;
   private int InIndex = 0;
   private int OutIndex = 0;
   private int Count = 0;
   private Item Data[] = new Item[BufferSize];

   public Buffer() {} // constructor

   public synchronized void Put(Item X) throws InterruptedException  {
      while (Count >= BufferSize) {wait();}       // Buffer is currently full
      Data[InIndex] = X;   InIndex = (InIndex + 1) % BufferSize;  Count++;
      notifyAll();       // a broadcast notifying all threads „wait“ing on this buffer object
   }

   public synchronized Item Get() throws InterruptedException  {
       Item X;
       while (Count == 0) {wait();}      // Buffer is currently empty
       X = Data[OutIndex];  OutIndex = (OutIndex+1) % BufferSize;  Count--;
       notifyAll();      // a simple „notify“ here and above can/will cause occasional deadlock !!!
       return X;
   }
```

The following code will deadlock on the first call to „wait“:

```
class DeadlockingBuffer {
   ... data components as before, additionally ...
   private Object SpaceAvailable := new Object();
   private Object ItemAvailable := new Object();

   public synchronized void Put(Item X) throws InterruptedException  {
      while (Count >= BufferSize) {
            synchronized(SpaceAvailable) {SpaceAvailable.wait();}   // Buffer is currently full
      Data[InIndex] = X;   InIndex = (InIndex + 1) % BufferSize;  Count++;
      synchronized(ItemAvailable){ItemAvailable.notify();}
   }

   public synchronized Item Get() throws InterruptedException  {
       Item X;
       while (Count == 0) {
            synchronized(ItemAvailable) {ItemAvailable.wait();} // Buffer is currently empty
       X = Data[OutIndex];  OutIndex = (OutIndex+1) % BufferSize;  Count--;
      synchronized(SpaceAvailable){SpaceAvailable.notify();}
      return X;
   }
```

Assessment of the previous monitor templates:

- Java „synchronized“ is much better than C/C++ semaphores for programming mutual exclusion.
- Correct positioning of the „wait/notify“ calls is essential.
- The condition of a „wait“ must be reevaluated after notification.
- None of the models allows concurrent reading of shared data.
- The tie-in of semaphore and „wait“ semantics to a single object in Java is very unfortunate, since it forces a „notifyAll“ waking up all „wait“ing threads, even though the notification was intended for only one thread. This is **extremely inefficient**, since N context-switches occur with N-1 threads waiting again. A simple „notify“ can/will cause intermittent deadlocks !!
- In Java, a forgotten synchronization on a wait causes a run-time exception.
- Deadlock avoidance is difficult.

Does Java „synchronized“ really provide a Monitor ?

Consider:

```
public final class SynchNode {
   private int incoming = 0;
   private int outgoing = 0;
   private SynchNode[] links = new SynchNode[20];

   public SynchNode() { }

   public synchronized void link(SynchNode X) throws aSeriousFit {
      if (outgoing >= 20) {throw new aSeriousFit();};
      links[outgoing] = X;
      outgoing++;
      X.incoming++;
   }
}
```

…. SynchNode A,B,C;

A.link(B); and C.link(B); executed by two concurrent threads

What went wrong here ?

- The Java „synchronized“ semantics guarantee mutual exclusion for method calls on A and for method calls on C in the example, but not “against” each other. The accesses to *B.incoming* are therefore completely unsynchronized! Race conditions are possible, rendering the *incoming* count seriously wrong.

- While the receiving object itself is protected, the parameters are not.

- Although the components of SynchNode were made private, the visibility encapsulation is type-related and hence does not protect parameters against unsynchronized access by a method call for an object of the same type.

- EP opinion: to have a language with **type-related visibility encapsulation** and **object-related synchronization encapsulation** is a FUNDAMENTAL and MAJOR design bug in a language.

- **The practical advice:** in Java, even private components need to be read and written by synchronized methods. (this is a style guideline)

7.4.3 Ada 9X Protected Types

- data encapsulation as in monitors
- control abstraction as in conditional critical regions

Like MESA, Ada distinguishes several kinds of access operations for the monitored data:

- functions: can only read the monitored data; mutual exclusion of function calls is unnecessary
- procedures: can read or write the monitored data, but cannot be suspended on a condition
- entries: can read or write the monitored data and can be suspended on a condition

Exclusivity of writing operations is guaranteed.

Example 1: Implementing a counting semaphore

```
protected type Counting_Semaphore(Initial: Integer := 0) is
       entry Acquire;          -- P Operation
       procedure Release;      -- V Operation
       function Current_Count return Integer;
private  -- the protected data
       Count: Integer := Initial;
end Counting_Semaphore;
```

N.B. This example serves only to demonstrate the various forms of operations. With protected types, counting semaphores are superfluous, as the counting is more efficiently done directly in the protected operations that need resource counting.

```
protected body Counting_Semaphore is

    entry Acquire when Count > 0 is -- Suspend until Count > 0, then decrement it
        begin
                Count := Count - 1;
        end Acquire;

    procedure Release is
        begin
                Count := Count + 1;
        end Release;

    function Current_Count return Integer is
        begin
                return Count;
        end Current_Count;

end Counting_Semaphore;
```

Waiting until a condition is satisfied is done very much as in a conditional critical region but protected types exploit the integration into the language to address the efficient re-evaluation of the conditional guards and avoid all race conditions:

- before exiting a monitored routine that can modify the protected data (procedure, entry), the conditions of suspended entries are re-evaluated (in order of the queueing discipline; this ends when an entry with satisfied condition is found).
- This is done still under protection of the implicit semaphore, so that no race condition can arise ("evaluation by proxy" model; with some cleverness in the implementation, this is possible without a context switch).
- If one of the conditions is true, "control" over the monitored data can be handed directly to the resumed process if it is of higher priority. No V and P recompetition is necessary.

Example 2: “Mailbox”

```
protected type Mailbox(Box_Size: Positive) is

        entry Put(Item: in Item_Type);
                -- Adds item to mailbox; waits if mailbox is full
        entry Get(Item: out Item_Type);
                -- Removes item from mailbox; waits if mailbox is empty
private
        Data : Item_Array(1..Box_Size);
        Count: Natural := 0;
        In_Index, Out_Index : Positive := 1;
end Mailbox;
```

```
protected body Mailbox is
        entry Put(Item: in Item_Type)
                when Count < Box_Size is
        begin
                Date(In_Index) := Item;
                In_Index := In_Index mod Box_Size + 1;
                Count := Count + 1;
        end Put;

        entry Get(Item: out Item_Type)
                when Count > 0 is
        begin
                Item := Data(Out_Index);
                Out_Index := Out_Index mod Box_Size + 1;
                Count := Count - 1;
        end Get;
end Mailbox;
```

With the above declarations of a mailbox type, our producer/consumer problem now reads:

```
Item_Buffer: Mailbox(Buffer_Size);
task type Producer;
task type Consumer;

task body Producer is
    Item: Item_Type;
begin
    loop
        ... -- produce an item here
        Item_Buffer.Put(Item);
    end loop;
end Producer;

task body Consumer is
    Item: Item_Type;
begin
    loop
        Item_Buffer.Get(Item);
        ... -- consume the item
    end loop;
end Consumer;

Cooks: array(1..5) of Producer;
Gang: array(1..10) of Consumer;
begin -- activate the tasks
```

Note how readable and conceptually clean our program has become compared to earlier versions using semaphores, condition variables and alike.

Example 3: “Persistent Signal”

```
protected type Persistent_Signal is
        entry Wait; -- Wait for signal to arrive, if not yet occurred
        procedure Signal; -- Send the signal
private
        Signaled: Boolean := false;
end Persistent_Signal;

protected body Persistent_Signal is
        entry Wait when Signaled is
        begin
                Signaled := false; -- Clear "consumed" signal
        end Wait;
        procedure Signal is
        begin
                Signaled := True; -- Set signal for current or a future WAITer
        end Signal;
end Persistent_Signal;
```

N.B.: This is, of course, the same as a mutex initialized to true and expressed in the terminology of signaling.

Example 4: “Transient Broadcast Signal, Signaler waits for completion”

```
protected type Transient_Signal is
        entry Wait;        -- Wait for signal to arrive in the future
        entry Broadcast;   -- Send the signal to all waiting tasks
private
        -- need no data; attribute 'Count is enough
end Transient_Signal;

protected body Transient_Signal is
        entry Wait when Broadcast'Count > 0;
                -- wait until a task is queued at Broadcast
        entry Broadcast when Wait'Count = 0;
                -- wait until all "Wait"ing tasks are released
end Transient_Signal;
```

The 'Count attribute applied to an entry yields the number of tasks currently queued on the entry. (The value is subject to race conditions which will not matter for the above use.)
Note the ease with which this rather difficult signaling paradigm is handled with protected types.

7.5 Monitors and Deadlocks

If a failing condition check on entry is defined to release the mutex on the monitor, it cannot be the cause for a deadlock caused by circular dependencies. Only the mutex itself can contribute to a deadlock.

As a result, there is a simple rule that avoids deadlocks if exclusively monitors (and monitor-like structures,e.g. protected objects) are used for synchronization:

If there are no nested calls on monitors from within monitor code, then there cannot be a deadlock.

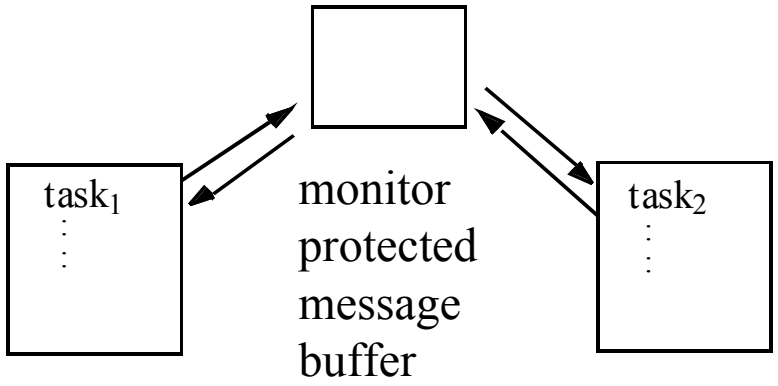
This is a good rule to remember, if only to know where a deadlock cannot possibly arise. (It negates condition 2 of deadlocks, i.e. the gradual acquisition of resources.)

In reality, such nested calls are sometimes unavoidable. Moreover, the check is not easy, since the nested call can be issued by a transitively called subprogram. Again, the global call graph is useful to conservatively verify an absence of a nested monitor call.

Note that the conditions enumerated earlier for a deadlock were necessary but not sufficient conditions. Nested monitor calls are not a problem per se. A guaranteed deadlock arises only if the same monitor is accessed recursively from within another monitor (again, the call graph helps to conservatively decide this property).

7.6 Communication Paradigms

7.6.1 Mailbox Paradigm



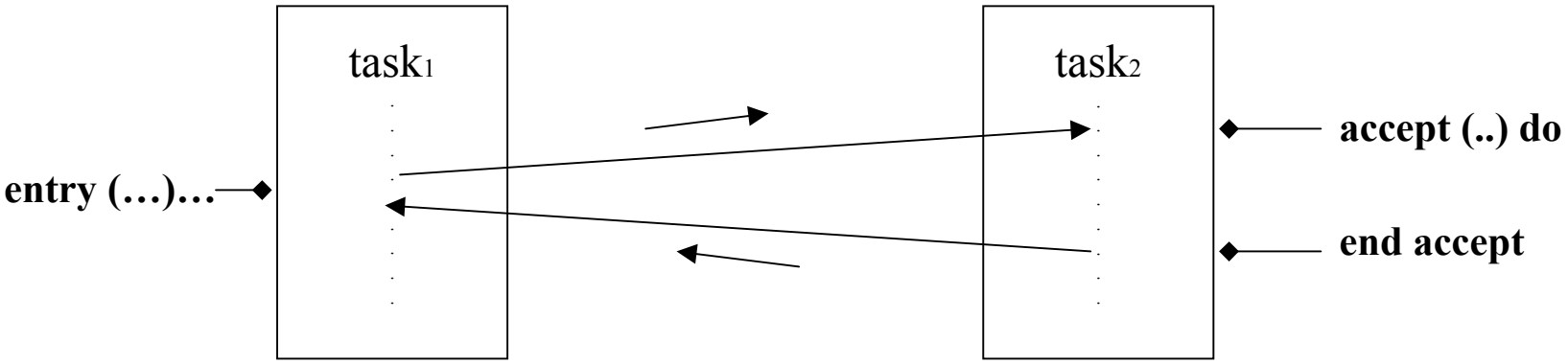
Advantages:

- tasks remain anonymous
  - adding additional tasks (producer or consumer) is without problems
- variations in speed can be balanced out, so that high through-put can be achieved
- producing task does not wait until receiving task gets the “mail”

Disadvantages:

- tasks remain anonymous
  - communication between two specific tasks is very difficult to achieve, inefficient, and almost impossible to verify if more than two tasks access the mailbox.
  - synchronization of the tasks with each other (so that they "know" how far the other has progressed) is equally difficult.
- producing task cannot wait until receiving task gets the "mail"; each information exchange activity is uni-directional.
- one central mailbox may be a significant bottleneck in larger systems; multiple mailboxes may create a reachability problem for exchanges among certain tasks.

7.6.2 Rendezvous Paradigm



Communication can also be done directly between interested tasks, synchronizing them for this purpose. In the **rendezvous** paradigm, the interaction is requested by $task_1$ (via an entry call to the other task) and accepted by $task_2$ (via an accept statement).

In distributed systems, the rendezvous paradigm is often realized by synchronous **remote procedure calls.**

Either task waits for the other to reach the respective point in their execution. When both tasks have reached these points, parameters are passed, $task_2$ executes the rendezvous code specified in the accept statement, if any, and potentially returns results in the parameters, while $task_1$ waits for the rendezvous to end. After the rendezvous, both tasks proceed concurrently.

Advantages:

- communication is possible in both directions
- rendezvous without parameter exchange and rendezvous code is a pure synchronization
- the communication between specific tasks is verifiable more easily
- rendezvous is reasonably straightforward to implement on distributed systems without shared memory

Disadvantages:

- always a synchronization
- requesting task must know accepting task (but not vice-versa)

7.7 Rendezvous in Ada

The specification of a task (type) includes declarations of the entries that tasks of this type are willing to accept.

- specification of the task (type) contains the entry declarations:

```
task T is
    entry E (...Parameters...);
    entry B;   -- parameterless, usually for synchronization
    entry C(1..10) (...Parameters...);   -- a family of 10 entries
end T;
```

- the body of the task (type) contains the accept statements in ist code:

```
task body T is
        ...
begin
                              ...
        accept E (...Parameters...) do
                    -- the rendezvous code
        end;
        accept B; -- only a synchronization
        for i in 1..10 loop
            accept C(i)(...Parameters..) do
                  ... -- accepting C entries in round-robin fashion
                    end;
        ...
end T;
```

- calling entries of a task (entry call)

```
T.E (...Arguments...);
T.B;
T.C(5)(...Arguments...);
```

The semantics of the entry call T.E(...):

Case 1 :

T has not yet reached the **accept** statement for E

→ the calling process is suspended and queued on the entry T.E

Case 2a :

T had already reached the **accept** statement earlier

→ T was suspended waiting for a call on T.E; it will now be resumed for the rendezvous as described for case 2 b

Case 2 b:

T and the calling process arrive concurrently at the accept statement and the entry call, respectively

- T and the calling process are being “coupled together” (there will be scheduling, priority and error handling implications), parameters are passed, the rendezvous code between **do** and **end** is executed, [**in**] **out** Parameters are passed back, and then the two tasks decouple and proceed independently.

As there is a lot of waiting for rendezvous and because the sequence of accepts in a task body implies a matching sequence of entry calls, the simple, unconditional form of entry calls and accept statements is extended by a more elaborate mechanism, the **select** statement ...

7.7.1 Select Statement for Accepts

```
select
        when ... => accept E... do ... end; stmts;
or
        when ... => accept B... do ... end; stmts;
or
        accept C(1)... do ... end; stmts;
else
       ...  -- if none of the above applies, do something else here
end select;
```

In lieu of **accept** statements, **delay** statements or at most one **terminate** statement is allowed. These alternatives and the optional **else** part are mutually exclusive.

Semantics: if the **when** condition **("guard")** evaluates to true or is not present, the alternative is called "open". Among the open **accept** alternatives, an already waiting entry call will be selected (as determined by the queuing policy) for execution of the alternative starting with the respective rendezvous.

If no rendezvous is possible and there is an open **delay** alternative, the task will wait at least for the duration of the delay for an entry call to arrive. If none arrives, the **delay** alternative is taken.

If no rendezvous is possible immediately and there is an **else** part, the **else** part is executed.

If no rendezvous is possible and there is a **terminate** alternative, the task will wait for an entry call to arrive or it will cooperatively terminate (see section 7.10).

Example:

```
task Mail_Box_Mgr(Size: Integer := 100) is
        entry Put (Item : in Item_Type);
        entry Get (Item : out Item_Type);
end Mail_Box_Mgr;


task body Mail_Box_Mgr is
        Buf: array(1..Size) of Item_Type;
        Filled: Integer range 0..Size := 0;
        Next_In, Next_Out: Integer range 1..Size := 1;
begin
....
```

```
    ...
    loop
        select
            when Filled < Size =>
               accept Put(Item : in Item_Type) do
                      Buf(Next_In) := Item;
               end Put;
               Next_In := (Next_In mod Size) + 1;
               Filled := Filled + 1;
        or
            when Filled > 0 =>
                accept Get(Item : out Item_Type) do
                   Item := Buf(Next_Out);
                end Get;
                Next_Out := (Next_Out mod Size) + 1;
                Filled := Filled - 1;
        end select;
    end loop;
end Mail_Box_Mgr;
```

What makes this different from a protected type?

In the example, very little. Several general observations, though:

- Rendezvous code is kept to a minimum. The task control structure guarantees that no interfering operation will be accepted until the data updates are done concurrently to the callers.
- Looping over a select statement is a very common paradigm to accept different entry calls in arbitrary order.
- Other control structures can be used to enforce any needed sequencing of different entry calls (e.g., an “init” preceding any other access). Thus, interface protocols can be enforced.
- Naturally, elaborate code can be executed between accept statements, which creates the real concurrency of the task.

This task can now be used by other tasks, using rendezvous as the communication paradigm:

```
task type Producer;
task type Consumer;

task body Producer is
    Stuff : Item_Type;
begin
    loop
        -- produce Stuff --
        Mail_Box_Mgr.Put(Stuff);
    end loop;
end Producer;
```

```
task body Consumer is
    Value : Item_Type;
begin
    loop
        Mail_Box_Mgr.Get(Value);
        -- consume Value --
    end loop;
end Consumer;
```

```
Cooks: array(1..5) of Producer;
Gang: array(1..10) of Consumer;
Clerk: Mail_box_Mgr;

begin -- activate the tasks
```

7.7.2 Select Statement for Entry Calls

```
select
        entry_call; stmts;
   or
        delay ... ; stmts;
end select;
```

This construct specifies a **timed entry call**. It allows the caller to time-out on the call. I.e., the calling process will wait for the rendezvous at least as long as the specified delay. When the delay expires, the delay alternative is taken.

```
select
        entry_call; stmts;
   else
        stmts;
end select;
```

This construct specifies a **conditional entry call**. The else alternative is taken if the entry cannot be accepted immediately (because the accepting task is not waiting at the matching accept statement).

N.B. On the caller side, the control over the rendezvous is not as elaborate as on the callee side. The main reason is that with the model so far, any task can be only on two queues at any given time waiting for events to happen (one of which is the delay queue). This simplifies the run-time implementation significantly.

7.7.3 Requeuing

Sometimes an accepting task (or protected object) would like to pass responsibility for satisfying the request expressed by an entry call to another task (or protected object) without acting as a continuing intermediary in the synchronization and communication.

This facility is supported by the **requeue** construct. It redirects the entry call to the other entry and then “exits the picture” to be ready to serve other entry calls. The original caller now waits for a rendezvous with the third task (or executes/waits on the entry of the protected object, respectively).

Example:

```
task A is
   entry E(X: in out integer);
end A;

task B is
   entry XYZ(X: in out integer);
end B;

task body B is
begin
   ...
   accept XYZ(X: in out integer) do
        requeue A.E;
   end XYZ;
end B;
```

This allows tasks acting as managers of other tasks. The requeuing decision can be based on the parameters passed. A requeuing to another entry of the same task is also possible.

7.8 Error Situations in Ada Tasking

As in any code, errors might occur in the execution of a task resulting in an exception. The “normal” exception propagation and handling mechanism applies, except in three special circumstances:

- if the exception is propagated to the outermost level of the task body and remains unhandled there, the task will be completed and, after some transparent clean-up actions, terminated. The exception will not be propagated to other tasks, except when raised during task activation.
- if the exception is propagated out of an accept statement, it propagates **both** to the caller of the entry and to the frame enclosing the accept statement.
- if an exception is propagated out of a protected operation, the implicit lock on the protected object is released.

The tasking primitives give rise to additional exceptional situations:

- if an entry call is made to an already completed task, Tasking_Error is raised. This applies equally to all queued entry calls, if the accepting task becomes completed.
- if a task propagates an exception during its activation, Tasking_Error is raised at the point of activation after any jointly activated tasks have been activated.

It is generally a good idea to have an exception handler in the task body to prevent the “silent” termination of a task due to an unhandled exception.

7.9 Asynchronous Control of Execution

Sometimes it is necessary to terminate a thread of control “from the outside”, i.e. asynchronously. This may be necessary as part of fault detection and recovery or even as part of intentional system design:
a task that no longer responds to service requests should be terminated; a computation that takes too long or is no longer needed because the application environment has changed should be stopped.

Ada provides two mechanisms for this purpose:

- an abort statement that aborts a specified task (and all tasks of which the specified task is a master), i.e.

```
abort T;  -- will abort the task T and its children
```

- the asynchronous transfer of control statement to abort a specified sequence of statements

7.9.1 Task Abortion

Obviously, the abortion of a task is an extreme measure. It should be used with extreme care to avoid subsequent omission faults. Race conditions need to be considered very carefully, so that other tasks will not attempt synchronization or communication with the task about to be aborted.

Abortion is also a “problem” for the aborted thread of control. Since the abortion occurs asynchronously, it can occur at an arbitrary point in execution. To prevent subsequent damage to the operation of other tasks as a consequence of state left inconsistent by the aborted task, certain actions are defined to be **abort-deferred**, i.e. the aborting will be delayed until the action is completed.

Among the abort-deferred actions are:

- any protected operation (so that its atomicity is preserved and the lock released)
- any operation involving the initialization, copying or finalization of a controlled object (a subject not covered as yet in this course)
- waiting for termination of dependent tasks (so that these tasks will indeed be awaited)

The semantics of abort are such that, from the abstract point of view presented to other tasks, the tasks are terminated after any abort-deferred operation has completed. The reality of execution is that the real abortion may not happen until the aborted tasks gain control of the processor again to perform their implicit clean-up operations.

The delayed abortion matters in three respects:

- some amount of CPU resource will be needed to complete the abortion;
- any existing race conditions may be affected by this delay in abortion;
- in languages that provide for controlled objects, i.e. objects for which a user-provided routine is implicitly invoked at the end of their lifetime ("**finalization**", "**destructors**"), abortion may include a significant amount of finalization processing.

7.9.2 Asynchronous Transfer of Control Statement

Ada also provides for a mechanism to abort the execution of a sequence of statements within a thread of control:

```
select
   triggering_statement; stmts;
then abort
   sequence_of_statements;
end select;
```

where the triggering_statement is either a delay statement or an entry call.

The semantics of this form of select statement is that the sequence_of_statements is executed. However, if the delay of the triggering statement expires or the entry call is accepted during this execution, this execution is aborted and the triggering alternative is taken.

In the delay form, this select statement is particularly useful to create a “watchdog timer” functionality in which the sequence of statements is given a hard deadline to complete or else be aborted.

In the entry form, it can be useful to express “mode change” functionality, that is the overall system is no longer interested in the computation which, hence, should not consume more resources.

All the semantics noted and remarks made about the abort statement apply here, as well. In addition, since the abortion may interrupt non-atomic assignments (and, hence, leave completely invalid value representations in memory), the code, and in particular the code executed after an aborted sequence of statements, needs to be crafted with special care.

Although we expressed the issues with the asynchronous abortion of control threads in terms of the Ada model, the very same issues apply to models offered by other languages or kernels as well.

In particular, POSIX offers asynchronous signals that can be delivered to threads where they need to be handled or else will terminate the thread or process without further cleanup. That is, all responsibility for leaving the computation state in a consistent form in the case of an abort rests with the programmer.

A deferral can be achieved by explicitly blocking signals at the beginning of critical sections and unblocking them at the end. Any signals arrived in the interim will be handled when the signal is unblocked. However, not all signals can be blocked.

For a more detailed discussion of POSIX signals, see Burns, Section 10.6.

By casting tasking primitives into higher-level language constructs (rather than requiring kernel calls by the user), the implementation actions needed to keep the execution correct and reasonably safe against bad effects of race conditions will be incorporated automatically into the generated code. There is less opportunity for programming errors, more opportunity for race condition avoidance, and a more comprehensible software.

The downside is that there is more reliance on the correctness of the compiler and less opportunity to fine-tune the synchronization aspects of the generated code, using deep human knowledge about the overall system behavior.

7.10 Task/Process/Thread „Begin” and „End”

Creation: (allocating memory, etc.)

- as part of system initialization (global tasks)
- dynamically during system execution or because of allocation

Activation:

- usually implicit upon creation (threads, allocated Ada tasks, processes) or soon thereafter (declared Ada tasks)
- some languages require explicit activation after creation (e.g. „activate” or „init” call)

Termination:

- reaching the end of code:

  rather problematic (for the user) in the case of communicating tasks - an obvious race condition with entry calls exists

- asynchronous „abort”, e.g., by other task:

  rather problematic because of the interruption at an arbitrary point in execution

  - data consistency, resources held

- asynchronous failure exception by other task, allowing for “last wishes” processing (e.g. via POSIX signals):

  - worse problems than for asynchronous abort

- cooperative termination:

  tasks „agree” to jointly terminate; this avoids race conditions for entry calls

  - high cost of synchronizing implementation

Example of cooperative termination:

```
task type T is ...;

task body T is
begin
    ...
    select
       when ... => accept ... do ...
    or
       terminate;
    end select;
    ...
end T;
```

```
declare
  T1, T2: T;
begin
  T1.E;
end; --(1)
```

At point (1), this block awaits the termination of the two tasks T1 and T2. Only now will the terminate alternatives be considered. If indeed T1 and T2 are waiting at a terminate alternative, T1 and T2 will terminate and the block will end its execution.

Note that these semantics avoid all race conditions: At the time of the termination, it is guaranteed that no task in the system will issue an entry call to T1 or T2, while termination is being considered and then done.

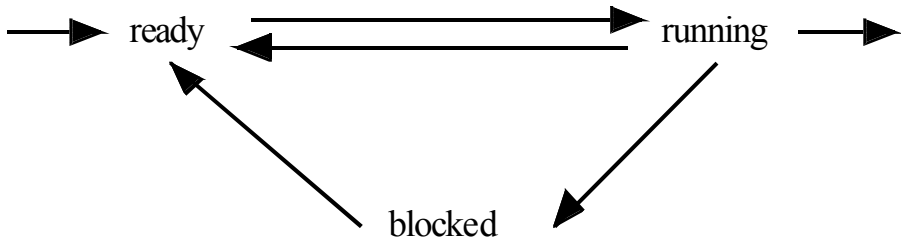
7.11 Implementation of Task Administration

During execution, the run-time system of the programming language (possibly assisted or directly provided by an operating system) is responsible for the administration of the tasks and, in particular, their scheduling.

A task is typically represented by a **task control block (TCB)** in which attributes of the task, e.g. its priority, and relevant state information is stored. Much of the administration is concerned with keeping the task (control block) on the various queues that match the state of the task. After the activation and before the termination phase, a task will be in one of the following three states:

- ready (to execute)
- running
- blocked (due to waiting on a signal, lock, and so on)

The state transitions are:



The most important queues are:

- READY_QUEUE: the queue(s) of tasks ready to execute but waiting to be assigned the processor. There might be multiple such queues (different priorities, different CPUs).
- RUNNING: a variable identifying the task currently executing. On a multiprocessor, this might be an array. On a single processor, the top of the READY_QUEUE can be (ab)used to represent this variable.

- EVENT_QUEUEs: queues for tasks that are blocked, waiting for some event (in the broadest sense) to occur. Unlike the READY_QUEUE and RUNNING, there are usually many of these queues, often incorporated into other data structures (e.g. in semaphores or TCBs).

The synchronization primitives have the effect of moving tasks from RUNNING to the respective EVENT_QUEUE when the task is to be blocked, and from this EVENT_QUEUE to the READY_QUEUE when the task is to be released (or, in some models, directly to RUNNING, preempting the current task).

Prior to evicting a task from RUNNING, its execution status needs to be saved in the TCB. Another task is selected from the READY_QUEUE (by the scheduling code in the run-time system), assigned to RUNNING, and its execution state is restored from its TCB.

Then its execution is initiated. The actions concerned with the replacement of the task in RUNNING are called a **context switch**. The maximum duration of a context switch adds to the latency of preemption.

Manipulations of the various queues need to be indivisible and generally non-interruptible operations.

An important question is the enumeration of execution points at which the scheduler gets involved. Task blockage is a necessary scheduling point. To what extent task release is a scheduling point depends on the scheduling regime. In preemptive priority-based scheduling it is a mandatory scheduling point.

7.12 An Example System (see Burns, Chapter 11)

Transmission requests are to be routed via three alternative communication lines. Lines may be open or overloaded, as established by the line controller. The router shall select a line that is not overloaded, or, if all lines are overloaded, the requestor must wait for an open line.

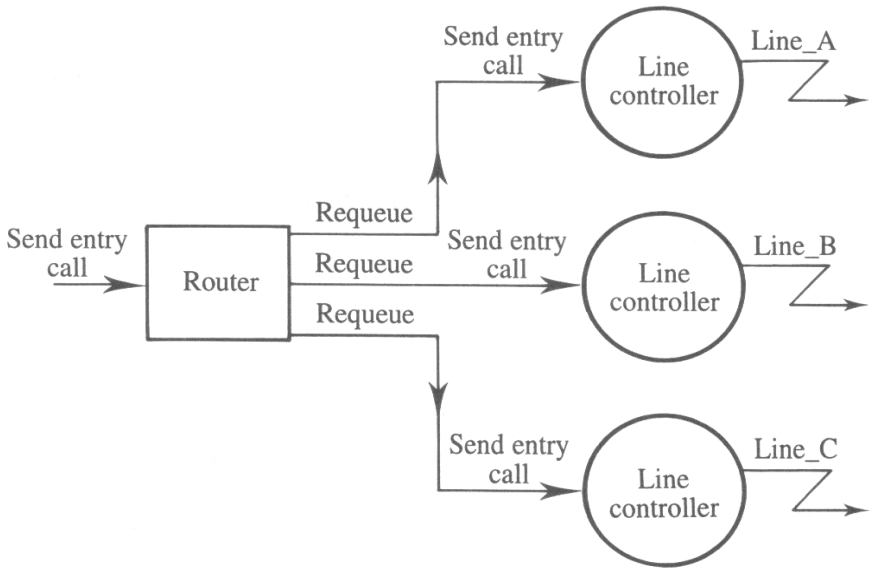


Figure 11.1 A network router.

```
type Line_Id is (Line_A, Line_B, Line_C);
type Line_Status is array (Line_Id) of Boolean;

protected Router is
        entry Send( ... );  -- will wait for an available line
        procedure Overloaded (Line : Line_Id);
        procedure Clear (Line : Line_Id);
private
        Ok : Line_Status := (others => True);
end Router;

--  requests can be issued to Router.Send, e.g., ...
Router.Send(...);  -- will wait for a free line
select
      Router.Send(...);  -- will wait for 1 sec. for a line
or    delay 1.0;
end select;
```

```
task type Line_Controller(Line : Line_Id) is
        entry Request( ... ); -- same parameter profile as Router.Send
      end Line_Controller;

 task body Line_Controller is
 begin
       loop
             select
                   accept Request( ... ) do
                         -- service request
                   end Request;
             or
                   terminate;
             end select;
             -- billing, diagnostics and housekeeping including possibly
             -- calls like Router.Overloaded(Line);  or Router.Clear(Line);
       end loop;
 end Line_Controller;
```

```
La:  Line_Controller(Line_A);
Lb:  Line_Controller (Line_B);
Lc:  Line_Controller(Line_C);

protected body Router is

      entry Send(...) when Ok(Line_A) or else Ok(Line_B) or else Ok(Line_C) is
      begin
            if Ok(Line_A) then
                  requeue La.Request with abort;
            elsif Ok(Line_B) then
                  requeue Lb.Request with abort;
            else
                  requeue Lc.Request with abort;
            end if;
      end Send;

      procedure Overloaded(Line : Line_Id) is
            begin Ok(Line) := False; end Overloaded;

      procedure Clear(Line : Line_Id) is
            begin Ok(Line) := True; end Clear;
end Router;
```

Real Time Programming

Chapter 8

Device Communication, Interrupts

8 Device Communication, Interrupts

Two different strategies can be identified to interface with external devices:

- in **status-driven control** the executing program periodically queries the status of the device explicitly and, if the device has delivered input, reacts to this input. This **polling** of devices need not imply busy waiting.
- in **interrupt-driven control,** the device raises the interrupt to get the attention of the executing program. The executing program is asychronously interrupted for the execution of an interrupt handler (to be provided by the programmer) and resumed after the handler completes.

Status-driven control is cumbersome and expensive in the sense that the polling needs to be included explicitly in the executing code and may often merely find that the data from the device has not yet arrived. However, it is completely synchronous and, thus, easier to include in various program analyses (WCET, schedulability, deadlock avoidance, correctness, etc.).

Interrupt-driven control is the predominant mechanism today. It generally is more efficient, as the executing program needs not execute polling code. However, the asynchronous nature of interrupts makes schedulability analysis more difficult. It also implies careful software design to avoid the possibility of undesired race conditions. It may also impose requirements on the nature of the code of the executing program (depending on the mechanism used for executing interrupt handlers). For example, all stack manipulations may need to be safe and (physically) atomic.

The general mechanism of interrupt handling involves the following elements, whose details are heavily dependent on the target system:

- The programmer needs to associate an **interrupt handler**, in the form of a subprogram or similar, with the interrupt.

- Each device is associated with a **device status register,** accessible by special instructions or, more often, tied to a specific memory address, providing details about the device status and the nature of an interrupt to its handler.

- When the device raises the interrupt, the hard/firmware will suspend the currently executing process, invoke the interrupt handler, and upon its completion resume the suspended process. (The specific support provided is a crucial information for the writer of the interrupt handler whose contents may have to obey significant restrictions.)

- Data communication with a device is usually through **data registers or I/O memory areas** at specific memory addresses known to both the device and the interrupt handler.

- Interrupt handling may in turn be interrupted. Some mechanism is available to "**mask**", "**block**" or "**disable**" interrupts, i.e. to prevent interruption of the executing process or interrupt handler by an incoming interrupt. A blocked interrupt may be "**pending**", i.e. will be queued and handled once unblocked, or may simply be lost.

The "target system" is either the hardware system or the kernel or the interrupt model supported by a programming language. In the latter two cases, capabilities may be implemented by software on top of the hardware facilities, providing additional functionality and some degree of hardware-independence.

8.1 Execution Model for Interrupts

A predominant model for interrupts utilizes an **interrupt vector** that associates handler code with each interrupt in the vector. When the interrupt occurs, the hardware vectors execution to the respective handler code. (This handler may have to vector in turn if multiple devices use the same interrupt.) The nature of this vectored execution is a key characteristic of the interrupt model. Various alternatives apply:

- the handler code is executed **as a procedure**. Execution will occur either on the stack of the interrupted process (in which case all generated code or hand-written assembly code must be safe and atomic w.r.t. stack operations) or on a separate interrupt stack.

In this case, the handler **MUST NOT** execute a blocking operation (since there is no blockable entity). This restriction makes access to global data without race conditions difficult. (Without special priority protocols (see later), only data whose updates are physically atomic can be safely accessed. In addition, any updates must not require coordination with updates of other shared data.)

- the handler code is executed **as a process.** This model (and the following model) requires a full context switch. In this case, the handler can employ all operations available to processes. In particular, it may call on blocking operations, e.g., to access shared global data.
- the interrupt is viewed **as an asynchronous event** for a specific receiving task, e.g. (in Ada) an entry call on a task or protected object within a task or (in Modula-1) the signaling of a condition variable.
- others... (see Burns, chapter 15.2.2)

8.2 Disabling/Enabling Interrupts

At times, it will be necessary to prevent interruptions, in particular during the handling of an important interrupt. The mechanisms vary and include:

- **status-related disabling**: the device or interrupt status information includes a modifiable bit, enabling or disabling any interrupts by this device, respectively this interrupt by any device.
- **interrupt masking** keeps a modifiable bitmap at a preestablished memory address, one bit per interrupt and typically directly mappable onto the interrupt vector. A bit value of 1 in the mask indicates that the respective interrupt is blocked.
- **interrupt levels (or priorities)** associate each interrupt with a certain level. During the handling of an interrupt, all interrupts of equal or lower level are blocked (excluding those at highest level that cannot be blocked at all). When a handler completes, blocking automatically reverts to the state before the interrupt handler was invoked.

The Ada model of interrupts supports the priority model. The Ada run-time system will map this model to whatever underlying model is supported by the hardware. (Restrictions on priorities of hardware-defined interrupts may apply.)

8.3 Accessing Status Registers

Access to status registers needs to be programmed very carefully:

- since status and data transfer registers change their value without explicit program action, compilers need to be made aware of the nature of the accessed entity, so that they do not generate code that caches an already retrieved value and reuses it when there are no intermediate updates in the code. Also, all updates in the source code must really be written to the respective memory location. Such memory is termed **“volatile”**. Compiler directives usually use this term to indicate this property.

- since updating a status register may cause the hardware to initiate actions, such updates must be physically atomic to ensure a consistent status register value. Often, a “volatile” directive implies an “atomic” directive to the compiler.

- since the physical layout of the registers is determined by hardware, the application code must respect the layout. (Not all languages provide mechanisms to model such layout for user-defined types.)

Read the documentation carefully and follow it to ensure these properties.

The layout issue can be dealt with in two radically different ways by the application code:

- model the register as a bitstring; manipulate it with bitstring operations. (A few source code lines usually suffice; however, they are quite unreadable and VERY error-prone.)

- model the register as a logical record (which it is) and supply directives for the layout. (This tends to be quite verbose, but it provides good documentation of the code and prevents many errors at compile-time.)

8.4 An Ada Example of Interrupt Handling

(taken and adapted from Burns, section 15.4.3)

This example is a simple driver for an Analog-to-Digital Converter (ADC) which receives and converts analog data such as temperature or pressure measurements from sensors and translates them into integer values. The device status register has 16 bits and a structure documented as follows:

Bit	Name	Meaning
0	A/D Start	Set to 1 to start a conversion.
6	Interrupt Enable/Disable	Set to 1 to enable interrupts.
7	Done	Set to 1 when conversion is complete.
8-13	Channel	The converter has 64 analog inputs; the particular one required is indicated by the value of Channel.
15	Error	Set to 1 by the converter if device malfunctions.

The device status register is located at octal address 8#150002#; the 16-bit result register at address 8#150000#.The system is a 16-bit, low-endian architecture.

To hide the “dirty details”, we encapsulate the ADC driver and its interrupt handling within a package “Adc”:

```
package Adc is

        Max_Measure : constant := (2**16) - 1;

        type Channel is range 0..63;

        subtype Measurement is Integer range 0 .. Max_Measure;


        procedure Read(Ch: Channel; M : out Measurement);
        -- returns a measurement from the sensor on the channel;
        -- potentially blocking, if ADC busy;
        -- may raise Conversion_Error

        Conversion_Error : exception;
        -- raised if ADC indicates failure to obtain in-range value

private

        for Channel'Size use 6;
                -- indicates that six bits only must be used

end Adc;
```

We factor the implementation of this package in three parts: the description of the registers in Adc.Registers; the interrupt handling in Adc.Driver, and the implementation of package Adc.

For the description of the registers, we need a variety of facilities of the language, specified in Chapter 13 of the Ada Reference Manual, that allow the description of physical layouts and locations.

```
with System;
        private package Adc.Registers is
            -- a child package, visible only to the body of Adc and other descendants of Adc.
                Bits_In_Word : constant := 16;
                Word : constant := 2; -- bytes in machine word
                type Flag is (Down, Set);
```

```
type Control_Register is
  record
        Ad_Start:                       Flag;
        Ienable:                        Flag;
        Done:                           Flag;
        Ch:                             Channel;
        Error:                          Flag;
  end record;

for Control_Register use
  record        -- specifies the layout of the control register
        Ad_Start        at 0*Word range         0..0;
        Ienable         at 0*Word range         6..6;
        Done            at 0*Word range         7..7;
        Ch              at 0*Word range         8..13;
        Error           at 0*Word range         15..15;
  end record;
for Control_Register'Size use Bits_In_Word;
        -- the register is 16 bits long ...
for Control_Register'Alignment use Word;
        -- on a word boundary ...
for Control_Register'Bit_Order use System.Low_Order_First;
        -- and the bit numbering starts at the low end
```

```
      pragma Atomic(Control_Register); -- implies "Volatile"

      type Data_Register is range 0 .. Max_Measure;

      for Data_Register'Size use Bits_In_Word;
            -- the register is 16 bits long

end Adc.Registers;
```

To implement the interrupt handling, we use a protected operation of a protected object as the interrupt handler. This interrupt handler must be installed as the interrupt handler for the interrupt that the ADC device is using.

The mechanism for informing the Ada compiler to do everything needed to install the operation as the handler is the

```
pragma Attach_Handler (Handler_Name, Expression);
```

This pragma can appear in the specification or body of a library-level protected object and allows the static association of a named handler with the interrupt identified by the expression; the handler becomes attached when the protected object is created. Raises Program_Error

a) when the protected object is created and the interrupt is reserved,

b) if the interrupt already has a user-defined handler, or

c) if a ceiling priority defined for the protected object is not in the range of Ada.System.Interrupt_Priority.

There is also a way to associate handlers dynamically with interrupts. To let the compiler know that an operation will be used as an interrupt handler without permanently attaching it, the following pragma is to be used:

```
pragma Interrupt_Handler (Handler_Name);
```

To inform the compiler about the interrupt priority, the

```
pragma Interrupt_Priority(Expression);
```

is used, where the Expression is an Integer value in the range of Ada.System.Interrupt_Priority (which is higher than any priority that can be set by pragma Priority).

For further details on the Ada interrupt handling capabilities, refer to the Ada Reference Manual, Section C.3.

We are now ready to write the private Driver package:

```
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
with Ada.Interrupts.Names; use Ada.Interrupts;
with Adc.Registers; use Adc.Registers;
private package Adc.Driver is

        Control_Reg: aliased Control_Register;
                -- aliased indicates that pointers are used to access it
        for Control_Reg'Address use To_Address (8#150002#);
                -- specifies the address of the control register

        Data_Reg: aliased Data_Register;
        for Data_Reg'Address use To_Address (8#150000#);

        Adc_Priority: constant Interrupt_Priority := 63;

...
```

```
        protected type Interrupt_Interface
                (Int_Id : Interrupt_Id;
                 Cr: access Control_Register;
                 Dr: access Data_Register ) is

                entry Read(Chan: Channel; M : out Measurement);

        private
                entry Done(Chan : Channel; M : out Measurement);
                procedure Handler;
                pragma Attach_Handler (Handler, Int_Id);
                pragma Interrupt_Priority(Adc_Priority);
                Interrupt_Occurred: Boolean := False;
                Next_Request: Boolean := True;
        end Interrupt_Interface;

        Adc_Interface: Interrupt_Interface(Names.Adc,
                                           Control_Reg'Access,
                                           Data_Reg'Access);
        -- this assumes that 'Adc' is registered as an Interrupt_Id in Ada.Interrupts.Names
        -- 'Access gives the address of the object
end Adc.Driver;
```

```
package body Adc.Driver is

    protected body Interrupt_Interface is

        entry Read(Chan : Channel; M : out Measurement)
                                        when Next_Request is
            Shadow_Register : Control_Register;
        begin
            Shadow_Register := (Ad_Start => Set, Ienable => Set,
                                        Done => Down, Ch => Chan, Error => Down);
            -- use a temporary to build the value to be assigned atomically
            Cr.all := Shadow_Register;
            -- at this point the ADC will start obtaining the sensor value
            Interrupt_Occurred := False;
            Next_Request := False;
            requeue Done;  -- we let the caller wait to get the measurement
                           -- but free the object to be able to receive the interrupt
        end Read;

        procedure Handler is
        begin
            Interrupt_Occurred := True; -- causes 'Done' to proceed
        end Handler;
```

```
            entry Done(Chan : Channel; M : out Measurement)
                        when Interrupt_Occurred is
            begin
                        Next_Request :=True;
                        if Cr.Done =Set and Cr.Error = Down then
                                    M := Measurement(Dr.all);
                        else
                                    raise Conversion_Error;
                        end if;
            end Done;
      end Interrupt_Interface;

end Adc.Driver;
```

Finally, we can implement the ADC interface. It includes some fault tolerance by trying three times in case of ADC-failures to overcome transient faults of the sensor or ADC device.

```
with Adc.Driver; use Adc.Driver;
package body Adc is

        procedure Read(Ch : Channel; M : out Measurement) is
        begin
            for I in 1..3 loop
                begin
                    Adc_Interface.Read(Ch, M);
                    return;  -- if no exception was raised
                exception
                    when Conversion_Error => null; -- try again
                end;
            end loop;
            raise Conversion_Error;
        end Read;

end Adc;
```

N.B. The prevailing usage of use-clauses in the example is motivated by the size of the slides. It really is bad programming style to use more than one use-clause to obtain direct visibility into other packages.

Real Time Programming

Chapter 9
Scheduling - “Take Two”

9 Scheduling - “Take Two”

The models presented in Chapter 5 assume process independence in calculating response times. Synchronization and communication among processes introduce another factor that can lengthen the response time of processes because processes may be blocked waiting for access to a global resource or for some signal to be sent.

Such times during which a process is blocked need to be added to the response time formula. That is,

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times C_j$$

becomes

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times C_j$$

where B_i stands for the total time during which process i is blocked, waiting for a resource to become available.

Some basis for calculating or conservatively approximating the blocking time is needed (and will be addressed later on).

One might assume that, because critical sections are usually kept as short as possible, the blocking time a process experiences upon waiting for a shared resource is equally short.

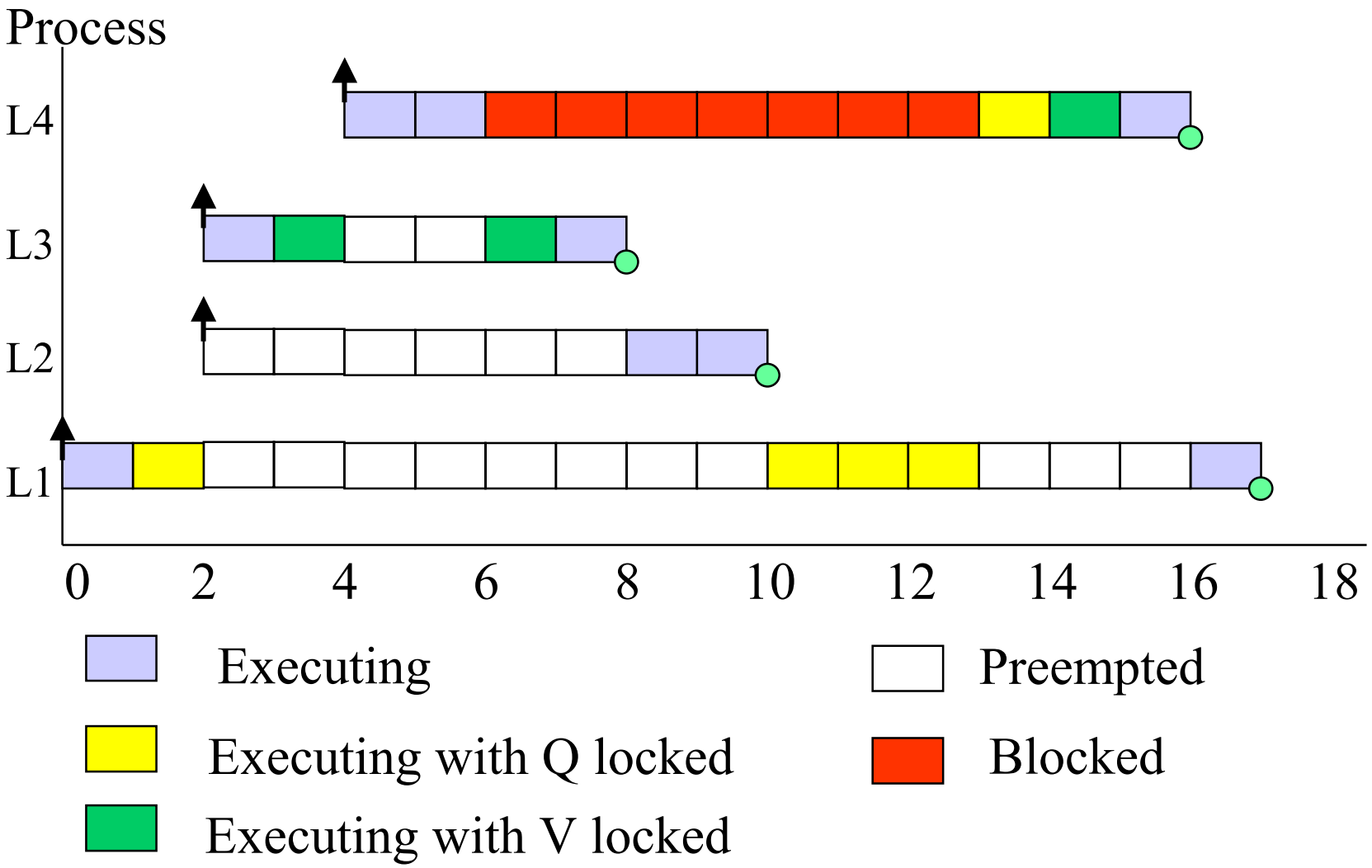
This assumption is fundamentally flawed for fixed priority assignment schemes. A phenomenon termed ***priority inversion*** may (and usually will) occur, as the following example demonstrates.

9.1 Example of Priority Inversion

Consider the following process set and execution patterns:

Process	Priority	Exec.sequence	Release time
L4	4	EEQVE	4
L3	3	EVVE	2
L2	2	EE	2
L1	1	EQQQQE	0

where E, Q, and V stand for one unit of time of execution with Q and V representing mutually exclusive access to resources Q and V. The release time indicates when the process happens to become runnable on the example time line.


Process
L4
L3
L2
L1
0
2
4
6
8
10
12
14
16
18
Executing
Preempted
Executing with Q locked
Blocked
Executing with V locked

The previous figure depicts the system behavior under a fixed priority scheme.

Note the behavior of L4. As it starts executing, it soon finds resource Q locked by process L1 which was preempted during its critical section accessing Q. Since L1 is of the lowest priority, it will not continue in its critical section until L2 and L3 are finished (or are blocked in turn). Thus, the behavior of L4 is as if, upon accessing Q, its priority were lowered to the priority of L1. This effect is called **priority inversion.**

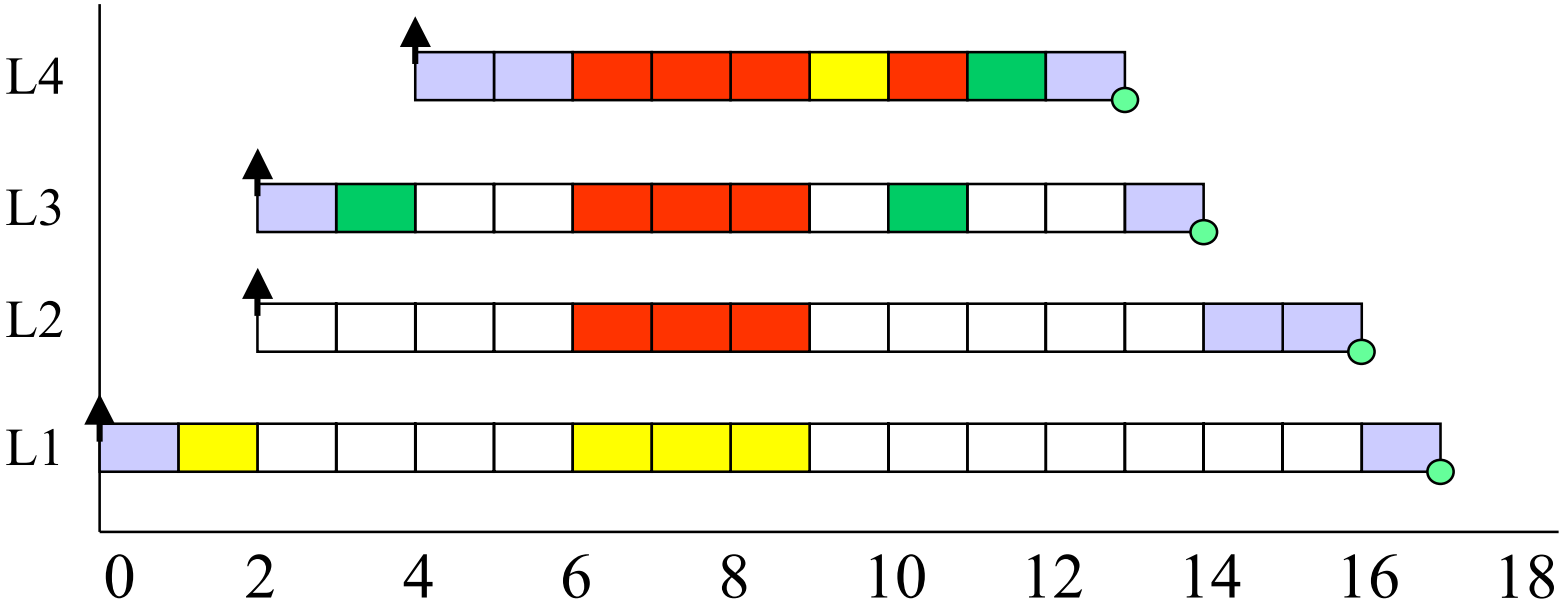
Under a fixed priority scheme it occurs every time when a high-priority process accesses an exclusive resource currently held by a lower-priority process.

9.2 Priority inheritance model

Historically (1987), the first attempt to fight the priority inversion effect was the introduction of priority inheritance. It attempts to “speed up” the process that will free the resource by temporarily assigning it the priority of the waiting process until it frees the resource. Priorities now change dynamically during execution in response to blocking.

In our example, L1 would inherit priority 4 from L4 and free the resource Q by time 9 (rather than 13), allowing L4 to proceed much earlier.

Priority Inheritance



Now, however, processes L2 and L3, although totally uninvolved in the contention over Q, suffer additional “blockage” by L1. (L2 does not even access any shared resource!)

It is a property of this priority inheritance model that each process can be blocked at most once for every exclusive resource it accesses.

In addition, it can be “blocked” by any lower-priority process that temporarily inherits a higher priority (technically: not really blocked but prevented from executing despite a higher static priority).

In the following, we use the term ***“true blocking”*** for the first case and ***“priority blocking”*** for the second.

For the simple priority inheritance model, the maximum blocking time of process i can be conservatively approximated as

$$B_i = \sum_{k=1}^{K} usage\ (k,i) \times CS(k)$$

where K is the number of resources,

CS(k) is the WCET of accessing resource k, and

usage(k,i) is defined as

= 1 if resource k is used by at least one process with a priority less than process i **and** at least one process with a priority greater than or equal to the priority of process i (including process i itself).

= 0 otherwise.

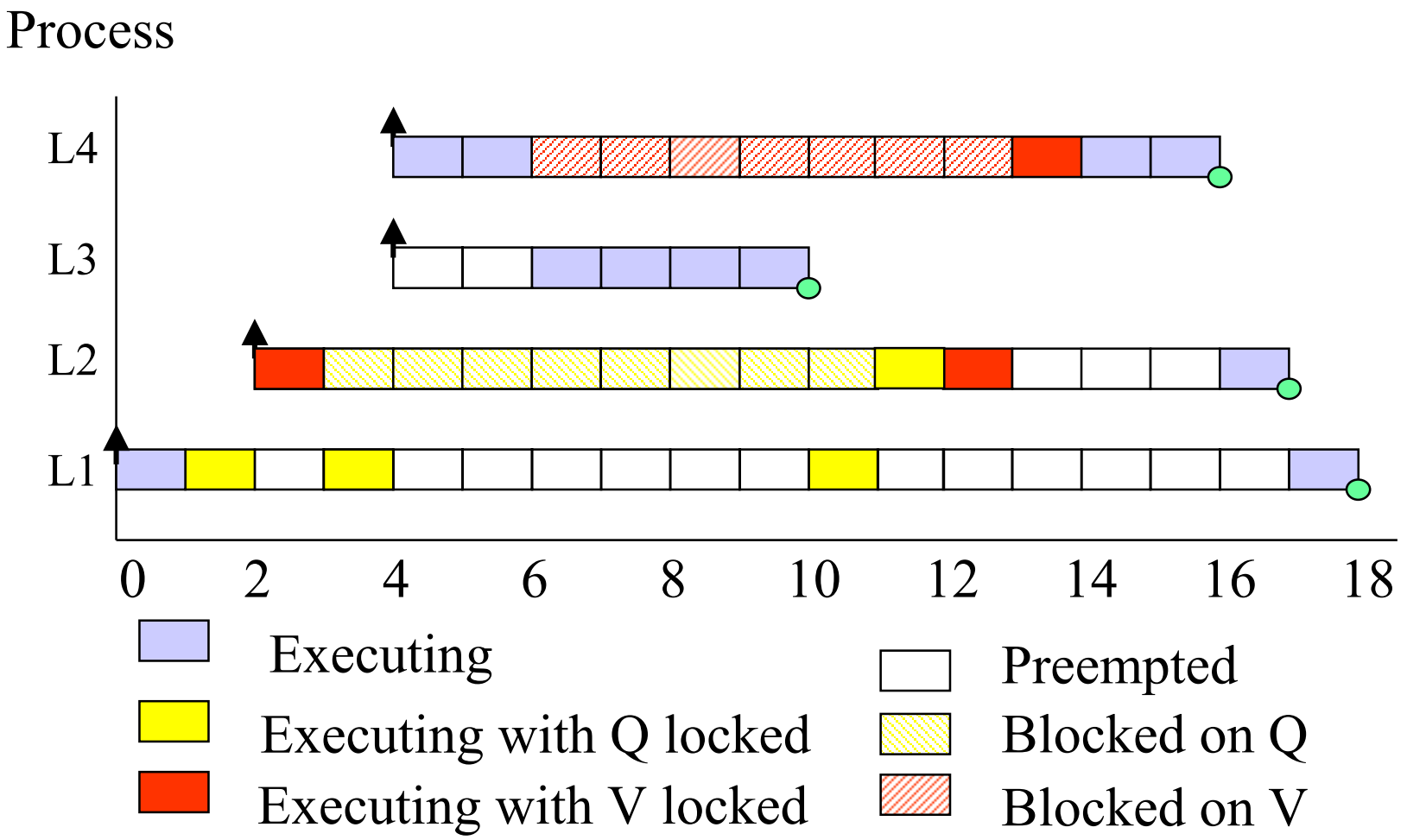
Note: the “equal” case covers the L1/L4 true blocking. The “greater” case covers the priority blocking on L2 and L3.

This is a (very) conservative estimate. There is no guarantee that the worst case ever arises, since the blocking will depend on the specific execution sequences and the release times of processes on any actual time line. Consequently, the Response Time Analysis will yield very pessimistic values.

The above presentation describes a **simple priority inheritance** scheme which ignores the effects of nested monitor calls and blocking operations other than mutually exclusive accesses to shared resources.

In general, blocking situations can be transitive: A high-priority process L4 may be blocked on a resource held by a medium-priority process L2 which in turn is already blocked on a resource held by a low-priority process L1.

Priority Inversion despite Simple Inheritance



Under the simple inheritance scheme, L1 has inherited the priority of L2. L2 subsequently inherits the priority of L4 but to no avail, since it is blocked on L1. Meanwhile, an independent process L3 is a runnable process taking priority over L1. Again, a priority inversion results for L4.

To cover this situation as well, **transitive priority inheritance** needs to be applied. In our example, when L2 inherits the priority of L4, L1 must also inherit the priority of L4 to avoid the priority inversion. In this scheme, inherited priorities need to be propagated along all chains of blocking relationships.

Priority inheritance is not easy to implement: Transitive priority inheritance implies frequent changes to the priority order in queues, potentially affecting multiple processes. The overhead cost may noticeably affect response times and is difficult to estimate.

For explicit semaphores, condition variables and alike, it is also very difficult, if not impossible, to determine which process(es) to “speed up” in order to end the blockage as quickly as possible, i.e. which process(es) should inherit the priority of the blocked process. Only for a monitor concept, it is very easy to identify the process that will free a resource.

This realization led to different strategies used in today’s systems based on monitors: **Priority Ceiling Protocols**

We start again with a model focused entirely on non-nested monitors and expand it later to account for other process interactions. **The described properties rely on this assumption.**

9.3 Priority Ceiling Protocols

The notion in priority ceiling protocols is to associate a priority value with each resource and to restrict access to the resource such that only processes with a current priority less than or equal to this priority value can lock the resource. This value is referred to as the **priority ceiling** of the resource.

(The explanation in Burns that the priority ceiling is the maximum priority of the processes that use the resource is a different way of expressing the above restriction.)

This restriction puts a constraint on the designer of the real-time system which is not always easy to satisfy. However, a number of significant benefits accrue (as we shall see).

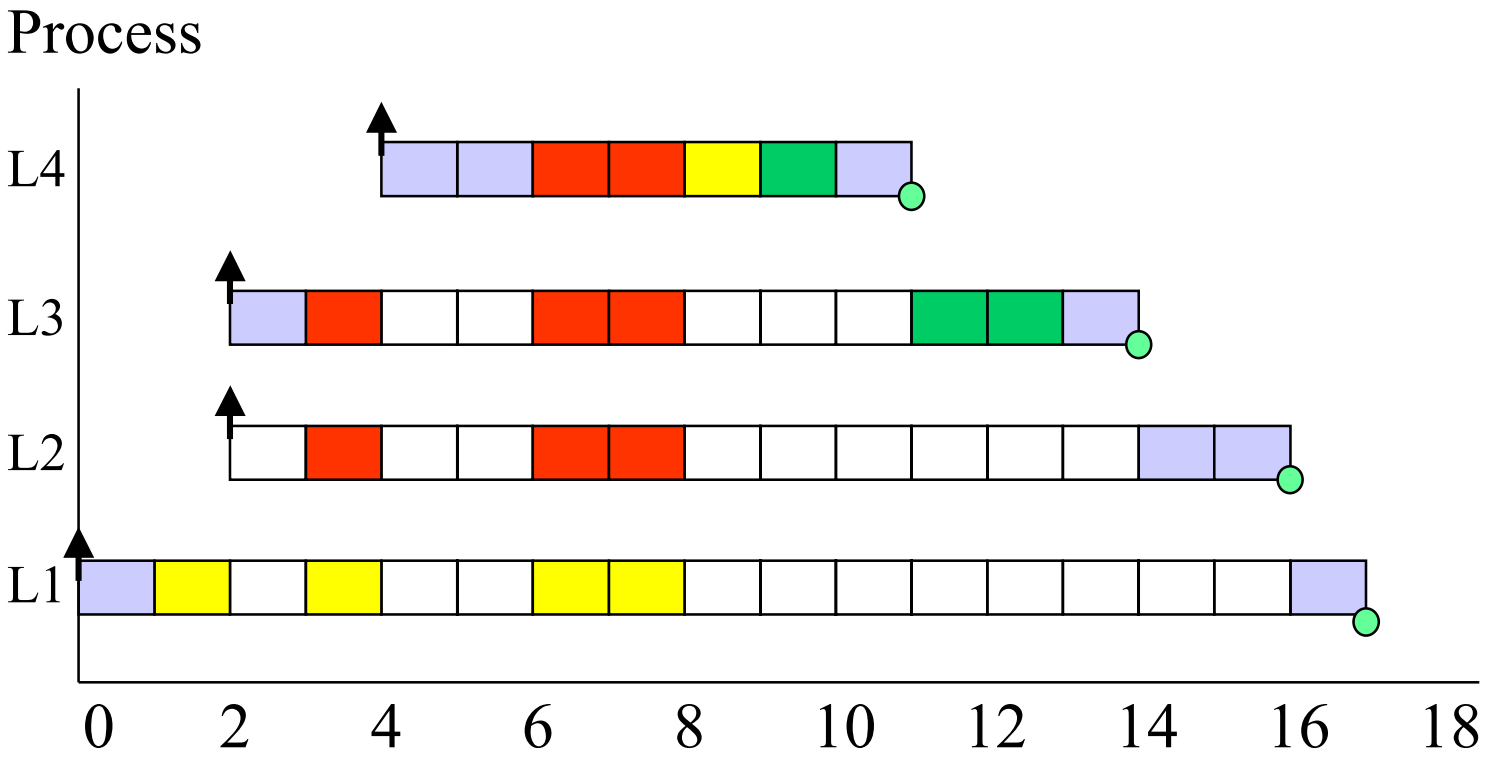
9.3.1 The Original Ceiling Priority Protocol (OCPP)

In this protocol, we have

- a static priority for each process
- a priority ceiling for each resource
- a dynamic process priority, which is the maximum of its static priority and any resource ceiling priority it inherits because it blocks a higher-priority process on that resource
- an added restriction that a process can lock a resource only if its dynamic priority is higher than the priority ceiling of ANY resource currently locked by other processes.

Note that this protocol has the same issue of needing to identify blocking processes as the priority inheritance model.

OCPP Schedule for the Sample Taskset



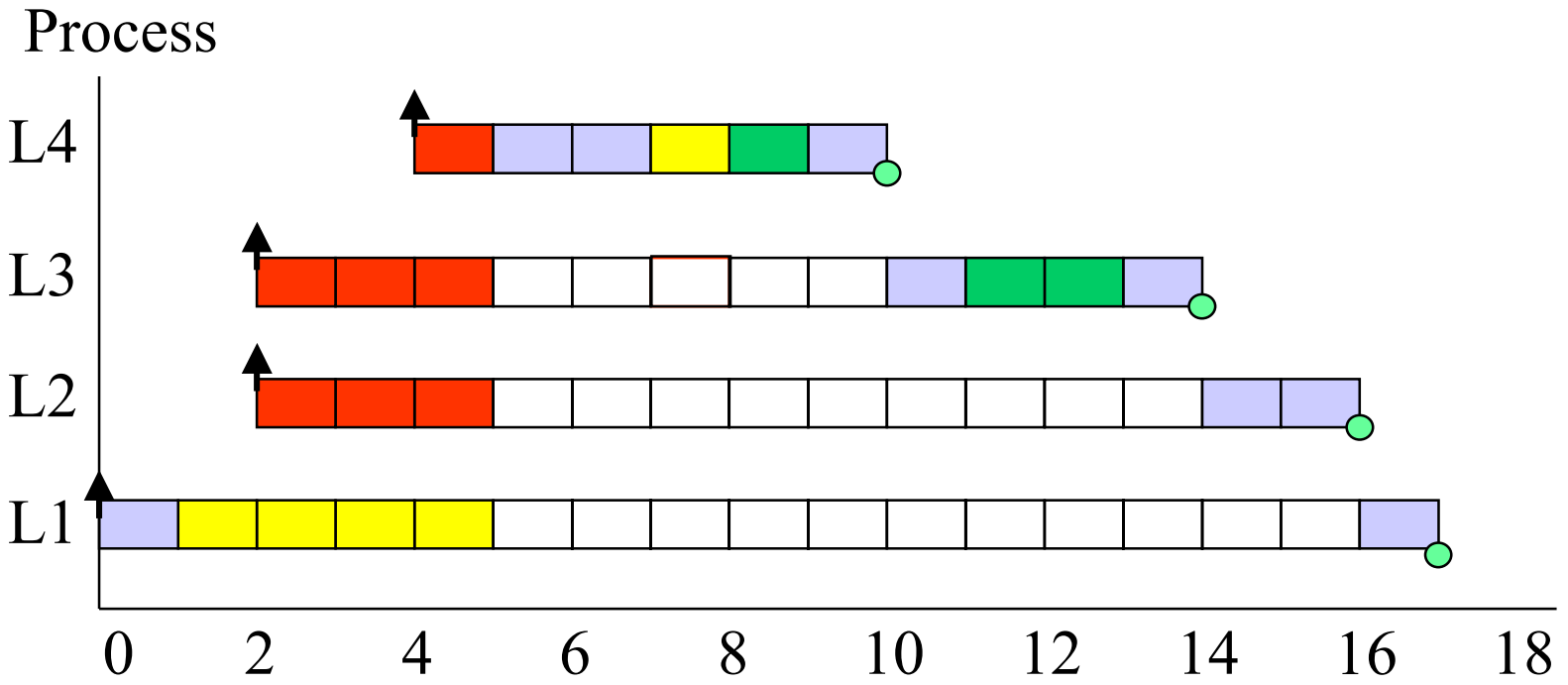
9.3.2 The Immediate Ceiling Priority Protocol (ICPP)

In this protocol, we have

- a static priority for each process
- a priority ceiling for each resource
- a dynamic process priority, which is the maximum of its static priority and the priority ceilings of any resources the process currently holds
- (no additional restrictions)

Note that in this model the process priority may increase upon access to a resource regardless of whether another process attempts a conflicting access. The issue of identifying blocking processes as in OCPP does not exist for ICPP.

ICPP Schedule for the Sample Task Set



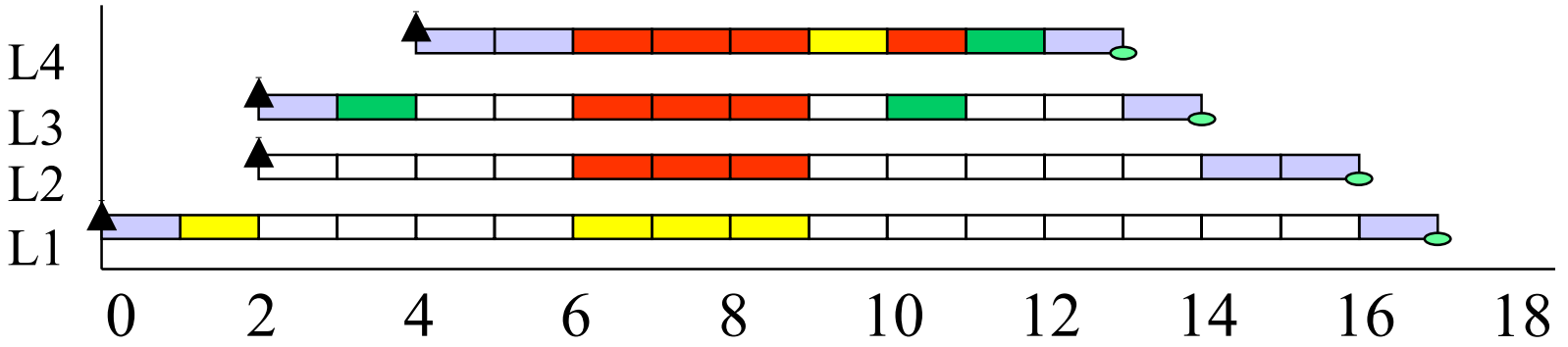
9.3.3 Commonalities and Differences of OCPP and ICPP

Both ICPP and OCPP guarantee that any process will be blocked at most once during its execution. For OCPP this may be a true or a priority blocking. For ICPP on a uniprocessor it can only be a priority blocking; moreover, it can only occur at the point when the process becomes runnable.

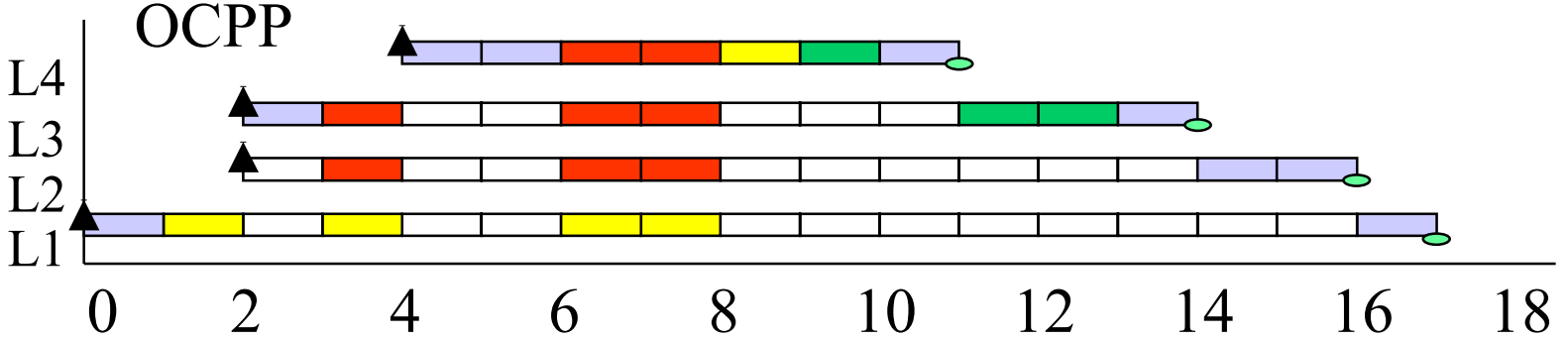
For ICPP this is easy to see. It cannot be avoided that, upon a process becoming runnable, another process holds a resource with a higher or equal priority ceiling and, thus, a priority blocking results.

However, once the process starts executing, no other process with lower static priority can hold a resource that is accessible to the process (or else it would be executing instead). Unless preempted, the process will therefore run to completion.

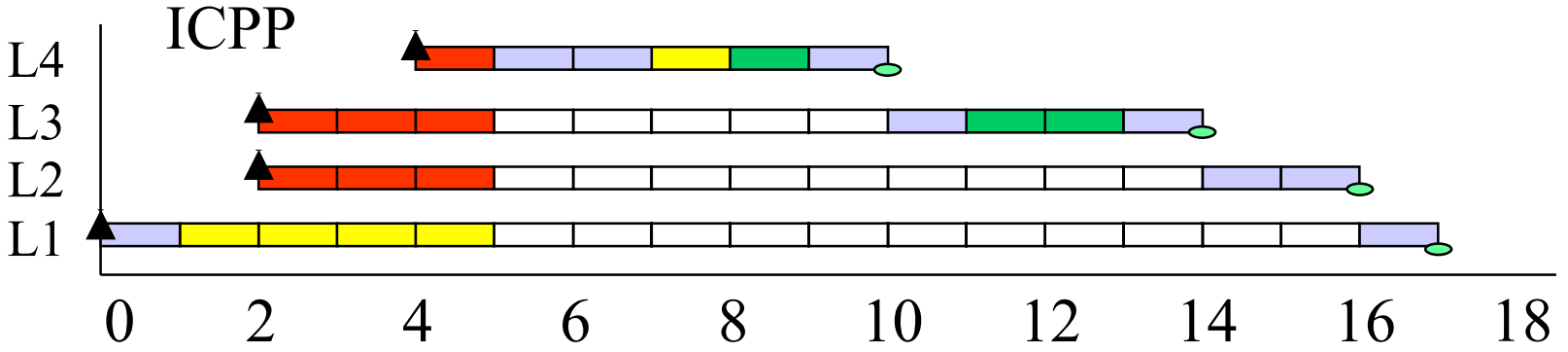
Priority Inheritance



OCPP



ICPP



Priorities will guarantee that, on a uniprocessor, a preempted process will not be resumed before the preempting process has released all resources.

For OCPP the situation is quite similar except that a true blockage is possible, since the lower-priority process gains its inherited priority only when the blocking occurs. Once the resource is freed, no further lower-priority process can gain the CPU and, thus, a conflicting lock.

ICPP is clearly easier to implement and easier to understand.

ICPP minimizes context switches since, after starting execution, the process cannot be blocked. ICPP implies more priority manipulations, as dynamic priorities are adjusted upon each access to a resource, while under OCPP priority adjustments occur only when higher-priority processes are blocked.

ICPP has the built-in assumption that the blocking and the freeing process are the same, i.e. the monitor concept. OCPP might be adaptable to other models as well.

9.3.4 Impact on Response Time Analysis

Since only a single blocking is possible for a process under either protocol, a considerably better formula applies to the blocking time:

$$B_i = max\ (\ usage\ (k,i)\ \times\ CS\ (k)\ )$$

i.e. the worst WCET of any critical section of any resource k, for which usage(k,i)=1, determines the maximum blocking time for process i.

9.3.5 Mutual Exclusion on Uniprocessors without Locking

Under ICPP, the rule that no process may access a resource with a priority ceiling less than its dynamic priority ensures that, once a process acquires a resource, its manipulations cannot be interfered with by any other process, provided that no physical parallelism is possible (i.e. in the uniprocessor case). Any physical locking of the resource by mutexes becomes superfluous. The protocol guarantees mutual exclusion by definition.

An interference could only arise if the process is preempted. However, it can be preempted only by a process of a priority higher than its current dynamic priority, which in the case of ICPP is the highest priority ceiling of any resource held. Consequently, the preempting process cannot access the resources held by the preempted process, since its priority is higher than their priority ceiling.

Under OCPP this guarantee exists only if priority inheritance is combined with a preemption in favor of the blocking process despite equal priority of other runnable processes (usually FIFO applies within a priority). This is yet another advantage of ICPP.

9.3.6 Deadlock Prevention

Since ICPP guarantees unblocked execution of any process after its initial priority blockage, there is no opportunity for a deadlock.

Under OCPP, the fourth rule assures that any acquired second resource is freed before it can be requested by another process. Hence, no deadlock opportunity exists.

9.4 Complications, Caveats, and Extensions

The preceding models are well suited for analyzing blockage by mutual exclusion mechanisms, in particular by monitors. As presented so far, they assume the absence of blocking for any other reason.

- If a potentially blocking operation (other than another monitor) is called within a monitor, the guarantees of ICPP and OCPP regarding deadlocks and mutual exclusion without physical locking are partially voided.

  They remain in effect only if the critical operations on the shared data occur after this blocking operation and any blocking releases the lock on the resource.

  Thus, language-supported entry barriers at the beginning of protected operations still provide the guarantees, while general condition variables do not.

- Of course, **calling a delay within a protected operation is utter stupidity:** Not only is it counter to the notion that protected operations should be completed as quickly as possible. It will also void the above guarantees.

- All the models assume **priority-ordered queues**. If FIFO queueing applies on monitor calls, the blockage time must be multiplied by the maximum number of tasks that can be queued up.

  In a slight modification of our earlier example, L2 could queue on Q held by L1, followed by L4 queuing on Q. Under FIFO queueing, L4 now needs to wait for both L1 and L2 to execute their critical sections.

  (As FIFO queueing is practically never used in hard real-time systems, we do not expand on this issue.)

- Priority inheritance can be partially adapted to rendezvous or, more generally, to any model in which the event ending the blockage can be uniquely attributed to a process by a-priori analysis.

  The models provide little or no basis for analyzing or influencing blocking on condition variables, barrier conditions, and similar mechanisms, i.e., general waiting situations.

  Here we are dealing with a generally undecidable problem and, thus, no tool can exist that will always be able to identify the releasing event. Partial solutions may be developed in the future. (I am not aware of any approaches that could be used in practice today.)

Some other complications have already been solved. The field of scheduling theory and WCET analysis is rapidly evolving. ***Watch the literature and the marketplace for the latest developments!***

9.4.1 Delays vs. Ceiling Protocols

The previous formulae assumed that a process will not release the CPU for any reason but preemption or blocking for mutual exclusion. A timed suspension (a delay) does precisely that, though, as does any waiting on a condition.

In this case the once-only blocking of processes guaranteed by ICPP or OCPP no longer holds, since the blocking scenario may repeat every time the suspension ends and the process contends again for CPU and resources.

Thus, our formula for the maximum blocking time becomes

$$R_i = C_i + (N_i + 1) \times B_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times C_j$$

where N_i is the number of delays (or, generally, waiting for any condition) of process i. That is, for every such wait, the blockage term is added again.

The effect of waiting needs to be added to C_i. This is very easy for relative delay waiting and extremely difficult/impossible for waiting on a condition.

Engineering trade-offs will need to be applied, depending on the cause for the delay.

For example, a device reading process may need to wait for some known stabilization time until a sensor value can be read, or for some known data transfer time.

If such **latency** is short (and the response time has a tight deadline), the waiting may better be realized by a busy wait on the device status register. (See also the impact of interrupts later on.)

Yet another alternative is the method of **period displacement** in which values are read on one cycle and processed on the next. In essence, the periodic delay is used to subsume the delay needed to accommodate the latency of the device.

Example of a period displacement:

```
loop
        delay until next_release;
        ...  -- set up device
        delay 0.03;  -- release CPU for other processing during warm-up
        ...  -- read (by polling) and process data
end loop;
```

turns into the simpler loop ...

```
-- set up device for next reading
loop
        delay until next_release;
        ...  -- read and process data
        ... -- set up device for next reading
end loop;
```

Of course, period displacement needs to obey certain constraints:

- First, the requirements of the application and the device characteristics must allow for acting on values already outdated by about one period T. The maximum staleness is bounded by T+R.
- Second, under all circumstances D <= T - S must hold, where S is the latency of the device. (T must be large enough so that, under worst-case assumptions, the process will not be restarted before the previous cycle has ended and the latency has passed.)

9.4.2 The Cost of Context Switches

So far, the formulae assumed either a zero context switch time or an inclusion of the context switch time in the WCET of the processes. We can be more precise, however, and may need to for demanding applications on architectures with slow context switches.

To accommodate context switch times, our response time formula (for ICPP or OCPP, no delays, periodic processes only) gets extended to

$$R_i = CS1 + C_i + B_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times (CS1 + CS2 + C_j)$$

where CS1 is the WCET cost of a context switch to a process and CS2 the WCET cost of a context switch away from a process at the end of its execution.

Maximum context switch time is an **essential documentation item** for the kernel or run-time system.

9.4.3 Interrupts and Sporadic Processes vs. RTA

In Chapter 5 we noted that sporadic processes are modeled in the response time analysis as periodic processes with worst-case or average inter-arrival interval as their period T.

Sporadic processes are usually initiated by interrupts. Regardless of the priority of the process, the initiating interrupt will be handled at a higher priority than all software processes and thus will preempt any executing process. The response time of the latter will be affected. To account for this effect, the response time formula needs to be adjusted to be ...

$$R_i = CS1 + C_i + B_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil \times (CS1 + CS2 + C_j)$$

$$+ \sum_{k \in sporadics} \lceil R_i / T_k \rceil \times (CSH1 + CSH2 + C_k)$$

where HC_k is the WCET of the interrupt handler that causes sporadic process k to be released. The context switch times to and from the interrupt handler are also added to the formula as CSH1 and CSH2. Depending on the interrupt model, they may be considerably smaller than CS1 and CS2.

If interrupts can arise that are not tied to the release of a sporadic process (i.e. act as asynchronous signals), the above formula can be applied regardless by including these interrupts in the second summation with a T_k equal to their worst-case, resp. average inter-arrival interval.

9.4.4 Impact by the Clock Handler and the Delay Queue

Special consideration ought to be given to the interrupts caused by the real-time clock. These interrupts are needed primarily for the release management of periodic tasks, respectively the management of the delay queue on which delayed processes await the expiry of their delay.

Usually the clock is set to interrupt at regular intervals. If a high-resolution clock is required, this interval will be correspondingly short, causing many interrupts that preempt executing processes and thus lengthen their response times.

Moreover, as part of interrupt handling, processes with expired delays will need to be moved to the ready queue. This effort is not of constant complexity.

Specific formulae for incorporating the cost of the clock interrupts into the response time analysis are found in Burns, section 16.3.3.

The example in the referenced section shows that, in complicated systems with many processes and some tight deadlines, the worst-case impact of the clock interrupt handling can be significant and bears watching.

9.5 Other scheduling regimes

There are two notable scheduling regimes that are based entirely on a dynamic determination of priorities:

- **earliest deadline first (EDF):** in this scheme the scheduler will always pick the runnable process with the deadline closest to the present.
- **least slack time:** in this scheme the scheduler always picks the runnable process whose deadline minus the still needed execution time is lowest.

Both models imply that all processes have deadlines and that these deadlines are explicitly known to the scheduler. The least-slack-time strategy requires in addition that the WCET (or ACET) of each process is known to the scheduler and that the run-time system keeps track of the time already spent in executing the process.

For independent periodic processes and T = D, these regimes are optimal in the sense that 100% utilization bound ensures schedulability, and that they will meet deadlines D < T, if at all possible. Consequently, they achieve much higher CPU utilization than the presented priority-based schemes.

On the downside, failed deadlines during transient overloads cannot be pre-determined.

Moreover, when blocking comes into play, the simple paradigm fails utterly to account for the equivalent of priority inversion (a process with a deadline far in the future blocking a more urgent process).

Baker‘s Protocol (1990) addresses this issue ...

9.5.1 EDF Scheduling and Shared Resources

Under Baker's Protocol (1990, a.k.a. "Stack Resource Policy protocol"):

- Dispatching is controlled by absolute deadline, i.e., EDF scheduling
- Preemption levels are used for shared resources
- A newly released task can preempt the currently executing task iff:
  - its absolute deadline is earlier, and
  - its preemption-level is greater than that of the highest locked resource.

If preemption levels are assigned according to relative deadline, Baker proved (under the same premises as ICPP) that the protocol then guarantees:

- Deadlock free execution
- Maximum of one block per task invocation

9.5.2. Preemption Levels vs. Priority Scheduling

Ada 2005 implements Baker's protocol in the context of priority scheduling by applying the following rules:

- For an EDF task, use its base priority to represent its preemption level; the base priority plays no direct role in determining the active priority
- Assign ceiling priorities (a.k.a. preemption levels) to protected objects in the usual way; execution within a protected object is at ceiling level
- Order ready queues by absolute deadline
- Scheduling selects from the highest priority queue that is not empty (same as in FP scheduling)

When a task S to be scheduled under the EDF regime becomes ready, the ready queue to which it is added is determined as follows:

- Define a ready queue at priority level p as being busy if a task has locked a PO with ceiling p – denote this task as T(p)
- S is added to highest priority busy ready queue p such that deadline of S is earlier than T(p) and base priority of S is greater than p
- If no such p exists, S is put on the Priority'First queue (resp. the lowest priority queue of the applicable EDF range in mixed dispatching)

These rules implement Baker's protocol and hence guarantee a single blocking for each task (under the same premises as ICPP for FP scheduling).

9.5.3 Mixed Dispatching in Ada 2005

Ada 2005 allows different dispatching policies to be used together in a controlled and predictable way

- Fixed Priority Preemptive Scheduling (FIFO, FP)
- Fixed Priority Non-Preemptive Scheduling
- Round-Robin Time-Sliced Scheduling (RR)
- Earliest Deadline First Scheduling (EDF)

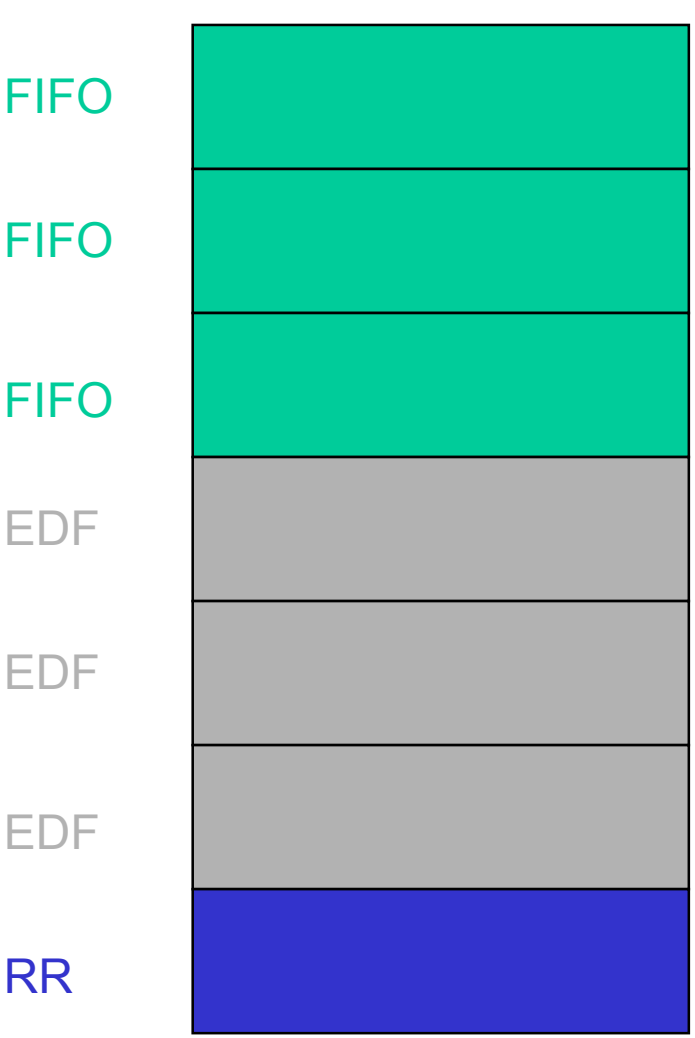
Protected objects can be used to communicate even across policies. As a consequence of the model, tasks can migrate between policies for the duration of their access to protected objects, while their active priority is in the range of the respective priority. The base priority of a task determines its initial placement.

Example: Splitting the Priority Range

The following pragmas specify the cooexistence of RR, EDF, and FIFO-FP scheduling for tasks with their base priority in the given ranges.

```
pragma Priority_Specific_Dispatching
         (Round_Robin_Within_Priorities,1,1);
pragma Priority_Specific_Dispatching
         (EDF_Across_Priorities,2,10);
pragma Priority_Specific_Dispatching
         (FIFO_Within_Priorities,11,24);
```

Example: Splitting the Priority Range



Hard-deadline tasks; RTA guarantees the deadlines

Soft-deadline tasks; EDF will meet as many deadlines as possible

Background tasks without deadlines

9.6 Summary on Scheduling

There is a wealth of knowledge available about the prediction of process response times. While it does not cover all mechanisms available today for process synchronization and communication, many applications are written in a style that is partially or fully amenable to such analyses.

By necessity, this course has only introduced the basic elements of the analyses. Specialized literature will give much more precise formulae for these analyses and cover more cases.

Computer support for WCET estimation is practically a necessity; for evaluating the response time formulae it is highly advisable. Tools are beginning to appear on the market that support the user in schedul-ability and response time analyses.

Kernels and/or languages support and exploit some of the protocols explained in this course. For example, ICCP must be supported by real-time conformant Ada compilers; Real-Time POSIX supports it under the name of Priority Protect Protocol.

The Grand Challenges today seem to be:

- obtain reliable WCET and ACET values needed for these analyses: research results look very promising
- include condition variables and alike mechanisms in the response time analyses: here the prospects are rather dim today

Real-Time Programming

Chapter 10

Patterns and Paradigms

10 Patterns and Paradigms

In this chapter we try to establish some “rules of thumb” about the structure of real-time systems and patterns for applying language concepts to achieve desired functionality. As such, they need to be moderated and adapted to the specific circumstances of the system design.

10.1 The Use of Tasks/Threads

The main purpose of using tasks or threads is to formulate processing activities that can, to a large extent, proceed independently of each other. The main advantage of their use is that the timing of the assignment of CPUs to their execution is delegated (and concentrated) to the task/thread administration component of the run-time system/kernel.

Rule of Thumb: Do not use a task/thread when most of its processing needs to be fully synchronized with any task that requests its services, i.e. the other task needs to wait anyhow for completion of the processing. In that case, use subprograms or monitor operations, respectively, if exclusive access to resources shared with other tasks is involved.

Particularly important candidates for tasks are:

- code that needs to be executed upon events that occur asynchronously to executing code. Examples are:
  - tasks to be executed periodically
  - tasks to be executed upon an external signal, e.g. an interrupt
- code that may have extended waiting periods

10.2 Periodic Execution

The usual implementation pattern for periodic execution of “tasks” is a single task that loops and delays.

```
task body T is
begin
        Next_Time := ART.Clock;
        loop
            delay until Next_Time;
            Next_Time := Next_Time + Cycle_Length;
            do_it; -- the actions to be performed in each period
        end loop;
end T;
```

This pattern is only applicable if R < T, i.e. it is guaranteed that the work is done before the next period begins.

A different pattern not subject to this restriction would be:

```
task body T is
begin
        do_it; -- the actions to be performed in each period
end T;
```

and somewhere, in another ("dispatcher") task or in a hand-written task scheduler, code with the following semantics:

```
loop
        delay until Next_Time;
        Next_Time := Next_Time + Cycle_Length;
        Task_Handle := new T;
end loop;
```

The disadvantage of this alternative is that the number of tasks is now dynamic and that the execution cost of task initialization and termination is borne for every invocation. The advantage is that (within schedulability) R > T is not an issue.

10.3 Processing Triggered by Interrupt

Two cases arise:

- the processing is of high urgency
- the processing is of low urgency (although the interrupt has a priority higher than any software thread)

In the first case, it is a matter of taste or language rules whether the entire processing is done in the interrupt handler or mainly by a task of appropriate priority and triggered by the interrupt (see below). Rules to consider are:

- the interrupt rules (automatic blocking of other interrupts during interrupt handling and task processing)
- the task priority rules
- interactions with protocols, e.g. the ICPP protocol, since the processing is likely to involve shared resources

In the second case, it becomes mandatory to keep the interrupt handler as short as possible and do the processing in a task that waits for notification about the interrupt. E.g.

```
task body T is
begin
    loop
        Interrupt_Mgr.Interrupt_Happened; -- waiting for the interrupt
        process_it; -- the actions to be performed upon interrupt
    end loop;
end T;
```

where the blocking call on Interrupt_Happened causes the task to wait for the interrupt. A version of the interrupt handling reduced to the essentials looks as follows:

```
protected Interrupt_Mgr is
                entry Interrupt_Happened;
        private
                procedure Handler;
                pragma Attach_Handler (Handler, Int_Id);
                pragma Interrupt_Priority(...);
                Interrupt_Pending: Boolean := False;
        end Interrupt_Interface;

protected body Interrupt_Mgr is
        procedure Handler is
        begin
                Interrupt_Pending := True;
        end Handler;

        entry Interrupt_Happened
                when Interrupt_Pending is
        begin
                Interrupt_Pending := False;
        end Interrupt_Happened;

end Interrupt_Mgr;
```

Variations of this theme have been presented in previous chapters, e.g. interrupts treated as persistent signals (as above), as transient signals, as broadcasts, or interrupts tied to reading values from a device.

10.4 Trade-offs for Concurrency and Context Switching

Consider the two following examples which functionally achieve precisely the same effects and synchronization:

```
protected type Mailbox(Box_Size: Positive) is
        entry Put(Item: in Item_Type);   -- Add item to mailbox
        entry Get(Item: out Item_Type); -- Remove item from mailbox
private
        Data : Item_Array(1..Box_Size);
        Count: Natural := 0;
        In_Index, Out_Index : Positive := 1;
end Mailbox;
```

```
task type Mailbox(Box_Size: Positive) is
        entry Put(Item: in Item_Type);   -- Add item to mailbox
        entry Get(Item: out Item_Type); -- Remove item from mailbox
end Mailbox;
```

```
protected body Mailbox is
        entry Put(Item: in Item_Type)
                when Count < Box_Size is
        begin
                Data(In_Index) := Item;
                In_Index := In_Index mod Box_Size + 1;
                Count := Count + 1;
        end Put;
        entry Get(Item: out Item_Type)
                when Count > 0 is
        begin
                Item := Data(Out_Index);
                Out_Index := Out_Index mod Box_Size + 1;
                Count := Count - 1;
        end Get;
end Mailbox;
```

```
task body Mailbox is
        Data : Item_Array(1..Box_Size);
        Count: Natural := 0;
        In_Index, Out_Index : Positive := 1;
begin
        loop
            select
                when Count < Box_Size =>
                        accept Put(Item: in Item_Type)do
                        Data(In_Index) := Item;
                        end Put;
                In_Index := In_Index mod Box_Size + 1;
                Count := Count + 1;
            or
                when Count > 0 =>
                        accept Get(Item: out Item_Type)do
                        Item := Data(Out_Index);
                        end Get;
                Out_Index := Out_Index mod Box_Size + 1;
                Count := Count - 1;
            end select;
        end loop;
end Mailbox;
```

Both examples do the same thing: maintain a shared mailbox and ensure mutual exclusion of calls. From the viewpoint of their users, the calling interface looks exactly the same and behaves in the same way, e.g. is blocking when the mailbox is full or empty, respectively. E.g.

```
My_Box : Mailbox;
...
My_Box.Get(X);
```

is legal in both cases and does the same thing.

Yet, there is a subtle difference in concurrency behavior:

While calls on the entries of objects of the protected type will not complete until all the buffer data has been modified, calls on the entries of a task of the task type will return as soon as the item is deposited or retrieved; the processing of the other buffer-related data in a Mailbox task occurs concurrently with further processing by the calling task.

The mutual exclusion is still guaranteed as the Mailbox task will not accept another entry before such processing is done.

There are three consequences:

- (In theory..) a high-priority caller gets held up for a shorter time; the mailbox completes its manipulations concurrently.

- (In theory…) throughput should be increased due to higher concurrency which utilizes wait states more efficiently.

- In practice, for this example, the task solution is liable to be much slower. This is due to the context switch necessary for the rendezvous (which can be avoided for a protected operation).

Conclusions and Rule of Thumb:

To turn theoretical advantages into practical gains, the amount of concurrent processing in a task versus context switching time is a decisive factor. If such processing is expensive, a task is likely to be the much more efficient solution (provided that other tasks are occasionally blocked or delayed).

If (almost) all processing by a task is within accept statements, it is a prime candidate to be rewritten as a protected object. The rewriting is easy and even automatable, as the example shows. The entry calls remain unaffected by such rewrites.

Note: had we moved the “end Put” and “end Get” two lines down, the concurrency behavior of both examples would be the same. In this case, the protected object is unquestionably preferable. The task then is called a “passive task”, since it executes no code on its own. (Some Ada compilers recognize such patterns and implement passive tasks akin to protected objects.)

10.5 Some Typical Task Patterns

10.5.1 The Server Task

```
task body Server is
begin
        loop
            select
                when ... => accept E(...) do ... end E;
                            do_the_E_job;
            or
                        ...
            or
                when ... => accept X(...) do ... end X;
                            do_the_X_job;
            or
                terminate;
            end select;
        end loop;
end Server;
```

Characteristics of a Server Task:

- runs forever accepting requests and processing them concurrently to the requestors

- no code outside the individual accept branches

- often little or no internal state, i.e. no information is retained from one call to the next (otherwise, we might term it a “monitor task”, If its main job is related to the maintenance of local information).

  Nota bene: if the state is global, it, in turn, needs to be protected against concurrent access.

10.5.2 The Proxy Task

```
task body Proxy is
begin
    loop
            select
                    when ... =>
                      accept E(...) do
                       do_some_checking_and_preprocessing;
                       requeue Real_Server.E;
                      end E;
                      local_accounting_and_post_processing;
                    or
                      ...
            end select;
    end loop;
end Proxy;
```

Characteristics of a Proxy task:

- does some preprocessing and checking on parameters before requeueing the request to the real server (usually a very busy server which gets off-loaded by the proxy)
- often better done by a protected object, depending on the amount of postprocessing
- particularly useful when the real server is a bottleneck task monitoring a shared resource. The proxies can be replicated as often as needed
- very useful (with some modifications) when proxy and real server are on different nodes of a distributed system

10.5.3 The Pipeline Task

```
task body Pipeline is
begin
        loop
                select
                  when ... =>
                        accept E(...) do
                                catch the parameters;
                        end E;
                        results := does_what_is_it_good_at;
                        next_in_line_task.X(results);
                  or
                     ...
                end select;
        end loop;
end Pipeline;
```

Characteristics of a Pipeline task:

- used for phased, concurrent processing of data
- each pipeline task accepts the results from the previous phase, processes them and hands its results off to the next pipeline task for processing.
- e.g. to read sensor data, pipe them through a filter, and then pipe them to a task controlling some activators
- watch the priorities to keep the pipe flowing smoothly !
- can be nicely generalized to a hierarchical network

10.5.4 The Immortal Task

```
task body Resistent_Server is
begin
    Outer: loop -- ensures immortality
            begin
              loop -- the real task loop
                    select
                      when ... => accept Done; exit Outer;
                      or
                      ...
                    end select;
              end loop;
            exception
              when others => Log_The_Error;
            end;
    end loop Outer;
end Resistent_Server;
```

Characteristics of an Immortal task:

- the real task cycle gets enclosed by an exception frame; the handler logs any error that occurred; the outer loop is only there to restart the task loop in case of an exception

- this task cannot be stopped other than by external abort, power failure, or the explicit entry call 'Done' which causes exit of the outer loop and, hence, task completion

- it will continue to serve requests no matter what happened in the past (hopefully only transient faults), e.g. because services called upon were unreliable (they almost always are!)

- good for pure servers; otherwise, repair of state information requires much more cleverness in the exception handler

10.5.5 The Protocol Enforcing Task

```
task body Protocol is
  Status: boolean;
begin
    accept A(X: Boolean) do Status := X; end A;
    accept B;
    if Status then   accept C;
              else   accept D;
    end if;
    for i in 1..10 loop accept F; end loop;
    loop
            select
                    accept F;
            or
                    accept B;
            end select;
    end loop;
end Protocol;
```

Characteristics of a Protocol-Enforcing task:

- Protocol tasks describe, by means of their control flow, the precise sequence in which entry calls can and will be processed. Very useful to “sychronize” independent tasks within a workflow whose results need to be “assembled” in a certain order. E.g. movement of lathes and robot drills.

For simple sequencing control, protected objects might also suffice. See an example in Burns, 10.2.3. More complicated protocols, however, are very difficult to describe by the data-oriented representation in protected objects and quite natural to express in the control flow form of tasks.

10.5.6 The “Deadbeat” Task

```
task Server ...; -- another task
task body Dead_Beat is
begin
    accept A;
    if global.comp = 7 then accept B; end if;
    accept A;
    Server.X;
    loop
            select
                    when ... => accept X do Dead_Beat.A; end X;
            or      when ... => accept B do
                                    Server.Y;
                                    accept C;
                                end B;
            or      when ... => accept A do
                                    ... delay 3.0; ...
                                end A;
            end select;
            accept C;
    end loop;
end Dead_Beat;
```

Characteristics of this „dead-beat“ task:

- This code incorporates stupid (1), risky (3), highly suspicious (1-2), and fatal (1) code. Identify the respective code portions.

And, please, do not write such code !

(As four of these situations are “bounded errors” in Ada, you might get stern compiler warnings or run-time exceptions, anyway.)

Real-Time Programming

Chapter 11
Assorted Topics

11 Assorted Topics

11.1 The “Mode Change”

In many real-time applications, the requirements on the executing software changes more or less drastically as the application environment changes. Typical examples of mode changes are:

- in airborne vehicles, going from take-off mode to ascent mode and later to level flight mode
- switching to emergency mode upon equipment failures or sudden surges in demands
- start-up of assembly line versus control during production

Some of the necessary changes and preparatory programming measures are:

- the task periods may change – make the period increment a variable rather than a constant
- the task deadline and thus priority may change – most languages/kernels provide the capability to change task priorities dynamically
- new tasks may be needed – create them or (as is frequently done) have them ready, but blocked until the mode change occurs.
- old tasks are no longer needed – abort them (dangerous) or plan for their orderly termination (usually expensive and difficult if time is of the essence), or block them by external intervention (this capability may not be easily available).
- additional devices may need to be enabled – preplan their interrupt hookup

- the processing of data may need to change, e.g. from expensive but precise results to cheap but fast, approximate results and vice versa - have the algorithms decide depending on a global mode variable, or dispatch to different implementations based on the current mode.

At first glance, this may seem not altogether difficult, but it is in practice. Among others, the reasons are:

- when tasks change their behavior, there is a significantly increased risk of omission or commission errors and of value errors: the “new” tasks might try communicating or synchronizing with still “old” tasks and vice versa. There is simply no way, short of very expensive global synchronization, that on a particular clock tick the “system” switches in its entirety from old to new mode.

- critical tasks cannot simply cease operation for some period of time needed to achieve the transition to the new mode; they need to continue operating as best as they can, meeting their hard deadlines, etc.
- runnable but now obsolete tasks cannot simply be aborted without severest risk of state damage that would poison the operation in the new mode. An orderly shut-down is needed, yet there may not be enough time to do so NOW.
- with very limited exceptions (ATC in Ada), no safe way is available to abort a computation at a granularity less than a task/thread.
- while the “steady state” before and after the mode change may be amenable to formal response-time analysis, very little is known today on how to perform estimations of system behavior during the mode change, when the system is so much in flux, and yet some guarantees must be given.

11.2 A Little Terminology for Distributed Systems

(Distributed Systems could fill half the course, so do not expect many answers in this short overview.)

A distributed system is a system in which the software executes on multiple autonomous processors (ranging from a multi-processor architecture to a networked group of computer systems) working towards a common purpose (e.g. control of an aircraft).

One categorization of distributed systems is into **tightly and loosely coupled systems.**

In a tightly coupled system, the individual computers or CPUs have access to shared memory and, thus, a fairly simple mechanism for communication. Typically, the CPUs are of identical architecture. The system is then termed **"homogeneous"**. The most important aspect of homogeneity is that binary representations of basic data types are identical for all the systems (and, thus, can be easily exchanged in binary form).

In a loosely coupled system, there is no shared memory. Communication (and synchronization) needs to be achieved by inter-communication between the systems, e.g. in a local area network (LAN) or on a system bus. Quite often, the CPUs in a loosely coupled system are not of the same architecture. The system is then termed **“heterogeneous”**. In heterogeneous systems, data exchange between systems needs to take care of different representations of basic data types. Transforming the data into an architecture-neutral format for transfer is often called **“marshalling”** and **“unmarshalling”** of the data.

11.2.1 Tightly Coupled Systems

For tightly coupled systems, there are numerous ways of mapping concurrent systems to the actual hardware resources:

1. The CPUs might be used to execute threads, where the association of threads with CPUs is decided dynamically by he run-time system/kernel or statically prior to execution. There is a single run-time system controlling all threads.

2. The CPUs might be used to execute processes, possibly internally composed of threads. Process control is by system-wide administration, while the threads are administrated by a local run-time system within each process.

...others

=> Read the documentation of your kernels or compilers, respectively.

Model 1 has important implications for implementation details and for response-time analysis. For example, the mutual exclusion guarantee of ICPP no longer holds. All locks must be physically acquired. The blocking time formula for ICPP/OCPP no longer holds and essentially degrades to the formula for Priority Inheritance (since lower-priority processes are no longer prevented by preemption from gaining a lock on a resource needed later by a higher-priority process).

Counter-intuitively, a multiprocessor implementation of a task set may exhibit slower response times than on a uniprocessor. For the causes and for good advice on how to avoid this, see Burns, 14.7.

Model 2 basically mimics a loosely coupled system, albeit with much faster means of communication and synchronization between processes via the shared memory.

11.2.2 Loosely Coupled Systems

Many hard real-time systems fall into this category. Since there is no shared memory, all communication and synchronization will have to be mapped to available communication protocols supported by the communication links between the individual computers. The absence of memory makes the usual thread model inapplicable for concurrency across systems.

A meta-standard for the communication protocols is the OSI Reference Model that assigns responsibility of various aspects of communication to layers of protocols:

- physical layer: transport of raw data over comm. channel
- data link layer: error control/retries over the physical layer
- network layer: network specific routing of communication through a network; packet creation, connection build-up

- transport layer: network indeterminate layer over network layer
- session layer: application-oriented mutual authorization of communication, control of exchanges
- presentation layer: compression, encryption, marshalling
- application layer: the real application logic involving communication

The last three layers are the application programmer's job; the earlier four are the communication provider's responsibility.

Aside: The OSI model is "a bit heavy" (too many layers), so that implementations may not always follow this layering faithfully.

Some or most of an application programmer's job is actually taken over by the programming language or kernel.

Probably the most famous and prevalent mechanism provided for communication between processes running on different compute nodes of a distributed system is that of a ***"remote procedure call"***. Its semantics are that, although called on one system, the body of the procedure is actually located on another system and executed there. The ***synchronous RPC*** waits for notification of completion and possibly return values of the execution of the body and, thus, provides two-way communication. The ***asynchronous RPC*** issues the call but does not wait for completion of the body (and, hence, no return values are possible).

Nota bene: Passing pointers to an RPC as parameters is quite meaningless. Equally, *call-by-reference* is not an option for passing parameters to RPCs. *call-by-value-result* and plain *call-by-value* are the only technically feasible alternatives.

The asynchronous RPC is essentially equivalent to a message passing scheme, sending off the identification of the requested body and the parameters as a message (after any necessary marshalling).

In some kernels, the message passing mechanism (which, by necessity, also underlies RPC) is directly available to the user.

11.2.3 The Ada Model for Distributed Systems

Ada, as one of very few languages, provides some support for distribution. Among other features, it is quite unique in providing static checks for the type safety of the communication among “processes” executing on different systems.

Ada has the notion of an ***active partition*** (comprised of Ada packages including tasks for concurrency within the partition). An active partition is essentially an executable for a single process, containing its own run-time system responsible for the administration of tasks within the partition.

Asynchronous and synchronous RPCs between active partitions are directly supported by the language (with suitable restrictions on their parameter types).

Ada also offers the concept of a **passive partition,** i.e. a partition without its own thread of control, which may be made accessible to multiple active partitions. It serves to model shared memory, in case the distributed system has shared memory for some or all of its computing nodes.

Ada provides rather rudimentary support for user-defined automated marshalling and unmarshalling of data. (It may not be enough for heterogeneous systems, i.e. there may be a need for significant programming to be done in this case.)

The details of the model are found in Annex E of the Ada Reference Manual. It needs to be supported only by compilers that claim suitability for distributed system development.

11.2.4 Some Particular Difficulties in Distributed Systems

Foremost, there are tough issues of fault tolerance:

- how to deal with a failure of a computing node? (it usually is a fail-silent)
- how to deal with unreliable communication lines? (they almost always are!);
- how to deal with a commission or omission error of a process on another node?

There are scheduling issues if there are more processes than CPUs. (Or, in the partition model, there will be the question of which tasks to include in which partition.)

With synchronous RPC and hard real-time systems, it is vitally important to establish a bound on the amount of waiting for the completion of the call.

A significant issue is the absence of a fully synchronized notion of time. With high-resolution timers, it is beyond the state of the art in hardware to produce timers that, after a relatively short time span, would not differ in their measured time by noticeable amounts. (Put simply: "*The clocks are always out of synchronization*.") This becomes important when processes are to coordinate their actions in reaction to externally generated events, where the time of the event (either absolute or relative to other events) is of importance. For more on the subject, see Burns, Chapter 12.1 and 14.6.

THE END