Scheduling

- Goal
  - To understand the role that scheduling and schedulability analysis plays in predicting that real-time applications meet their deadlines
- Topics
  - Simple process model
  - The cyclic executive approach
  - Process-based scheduling
  - Utilization-based schedulability tests
  - Response time analysis for FPS and EDF
  - Worst-case execution time
  - Sporadic and aperiodic processes
  - Process systems with D < T
  - Process interactions, blocking and priority ceiling protocols
  - An extendible process model
  - Dynamic systems and on-line analysis
  - Programming priority-based systems

Scheduling

- In general, a scheduling scheme provides two features:
  - An algorithm for ordering the use of system resources (in particular the CPUs)
  - A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied
- The prediction can then be used to confirm the temporal requirements of the application

Simple Process Model

- The application is assumed to consist of a fixed set of processes
- All processes are periodic, with known periods
- The processes are completely independent of each other
- All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
- All processes have a deadline equal to their period (that is, each process must complete before it is next released)
- All processes have a fixed worst-case execution time

Standard Notation

`B`	Worst-case blocking time for the process (if applicable)
`C`	Worst-case computation time (WCET) of the process
`D`	Deadline of the process
`I`	The interference time of the process
`J`	Release jitter of the process
`N`	Number of processes in the system
`P`	Priority assigned to the process (if applicable)
`R`	Worst-case response time of the process
`T`	Minimum time between process releases (process period)
`U`	The utilization of each process (equal to C/T)
`a-z`	The name of a process

Cyclic Executives

- One common way of implementing hard real-time systems is to use a cyclic executive
- Here the design is concurrent but the code is produced as a collection of procedures
- Procedures are mapped onto a set of minor cycles that constitute the complete schedule (or major cycle)
- Minor cycle dictates the minimum cycle time
- Major cycle dictates the maximum cycle time

Has the advantage of being fully deterministic

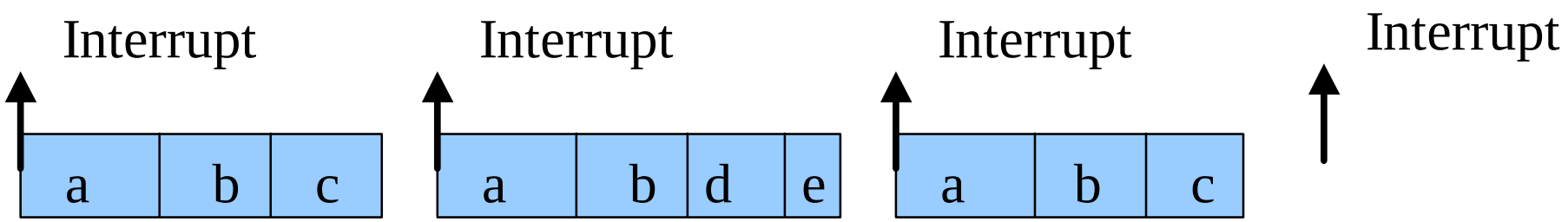
Consider Process Set

Process	Period, T	Computation Time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Cyclic Executive

```
loop
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
  procedure_for_e;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
end loop;
```

Time-line for Process Set



Properties

- No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls
- The procedures share a common address space and can thus pass data between themselves. This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible
- All "*process*" periods must be a multiple of the minor cycle time

Problems with Cycle Executives

- The difficulty of incorporating processes with long periods; the major cycle time is the maximum period that can be accommodated without secondary schedules
- Sporadic activities are difficult (impossible!) to incorporate
- The cyclic executive is difficult to construct and difficult to maintain — it is a NP-hard problem
- Any “process” with a sizable computation time will need to be split into a fixed number of fixed sized procedures (this may cut across the structure of the code from a software engineering perspective, and hence may be error-prone)
- More flexible scheduling methods are difficult to support
- Determinism is not required, but predictability is

Process-Based Scheduling

- Scheduling approaches
  - Fixed-Priority Scheduling (FPS)
  - Earliest Deadline First (EDF)
  - Value-Based Scheduling (VBS)

Fixed-Priority Scheduling (FPS)

- This is the most widely used approach and is the main focus of this course
- Each process has a fixed, static, priority which is computer pre-run-time
- The runnable processes are executed in the order determined by their priority
- In real-time systems, the “priority” of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity

Earliest Deadline First (EDF) Scheduling

- The runnable processes are executed in the order determined by the absolute deadlines of the processes
- The next process to run being the one with the shortest (nearest) deadline
- Although it is usual to know the relative deadlines of each process (e.g. 25ms after release), the absolute deadlines are computed at run time and hence the scheme is described as dynamic

Value-Based Scheduling (VBS)

- If a system can become overloaded then the use of simple static priorities or deadlines is not sufficient; a more adaptive scheme is needed
- This often takes the form of assigning a value to each process and employing an on-line value-based scheduling algorithm to decide which process to run next

Preemption and Non-preemption

- With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one
- In a preemptive scheme, there will be an immediate switch to the higher-priority process
- With non-preemption, the lower-priority process will be allowed to complete before the other executes
- Preemptive schemes enable higher-priority processes to be more reactive, and hence they are preferred
- Alternative strategies allow a lower priority process to continue to execute for a bounded time
- These schemes are known as deferred preemption or cooperative dispatching
- Schemes such as EDF and VBS can also take on a pre-emptive or non pre-emptive form

FPS and Rate Monotonic Priority Assignment

- Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
- I.e, for two processes `i` and `j`,

$$T_i < T_j \Rightarrow P_i > P_j$$

- This assignment is optimal in the sense that if any process set can be scheduled (using pre-emptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme
- Note, priority 1 is the lowest (least) priority

Example Priority Assignment

Process	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Utilisation-Based Analysis

- For D=T task sets only
- A simple sufficient but not necessary schedulability test exists

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\,(2^{1/N} - 1)$$

$$U \leq 0.69 \text{ as } N \to \infty$$

Utilization Bounds

N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

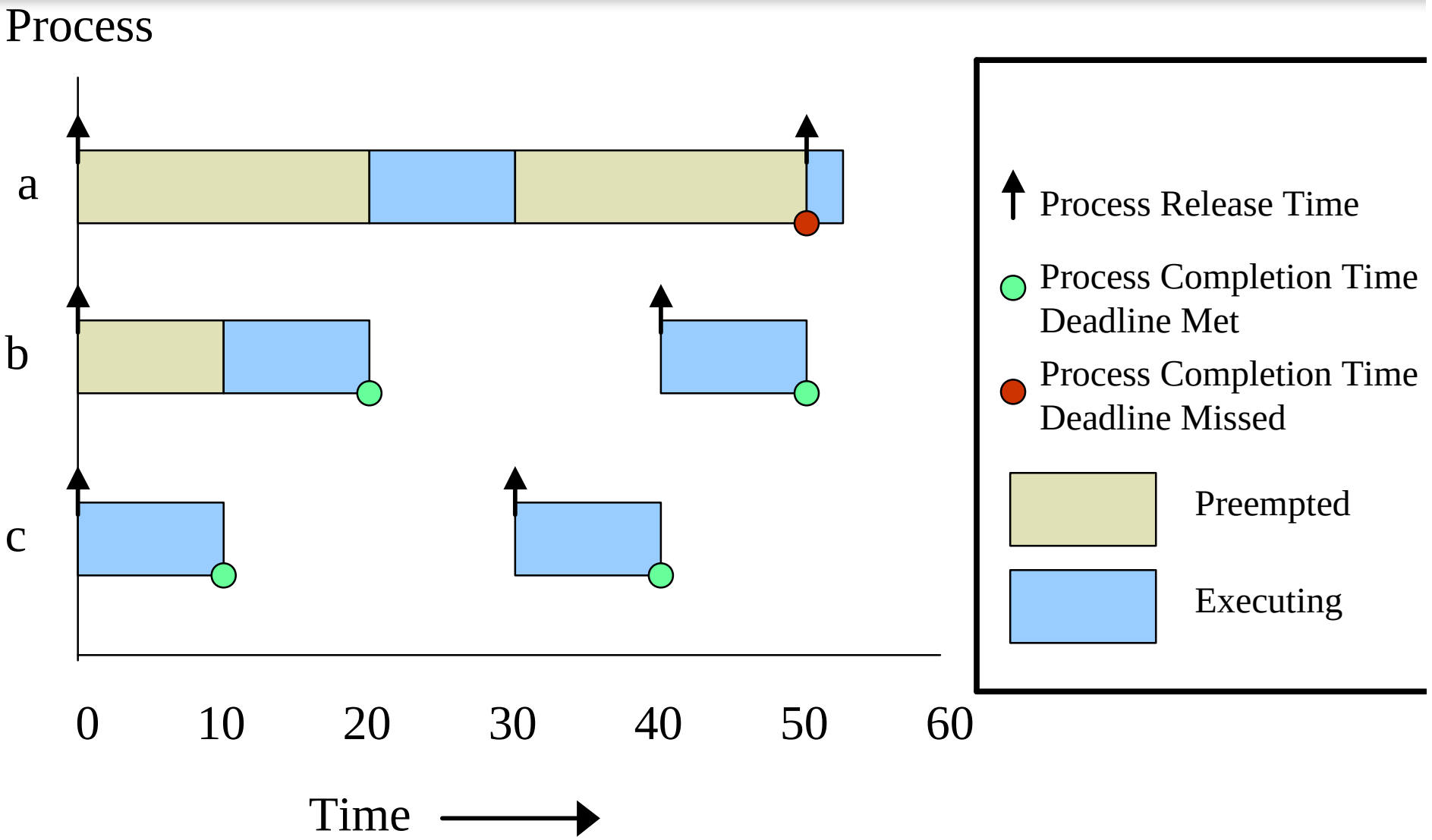
Approaches 69.3% asymptotically

Process Set A

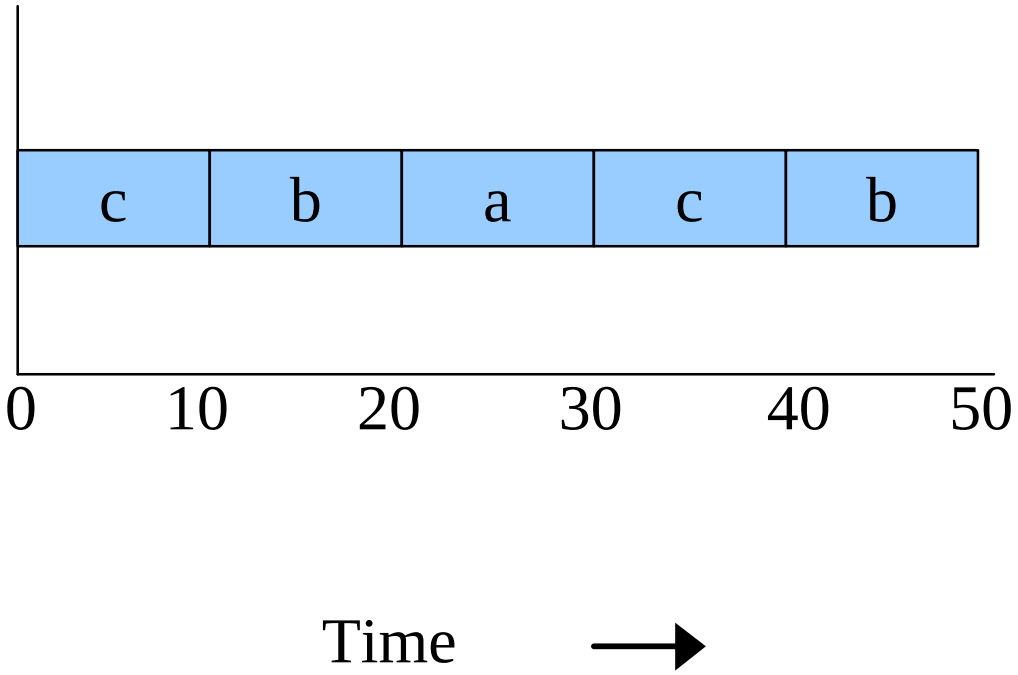
Process	Period T	ComputationTime C	Priority P	Utilization U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three processes (0.78) and, hence, this process set fails the utilization test

Time-line for Process Set A



Gantt Chart for Process Set A



Process Set B

Process	Period T	ComputationTime C	Priority P	Utilization U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

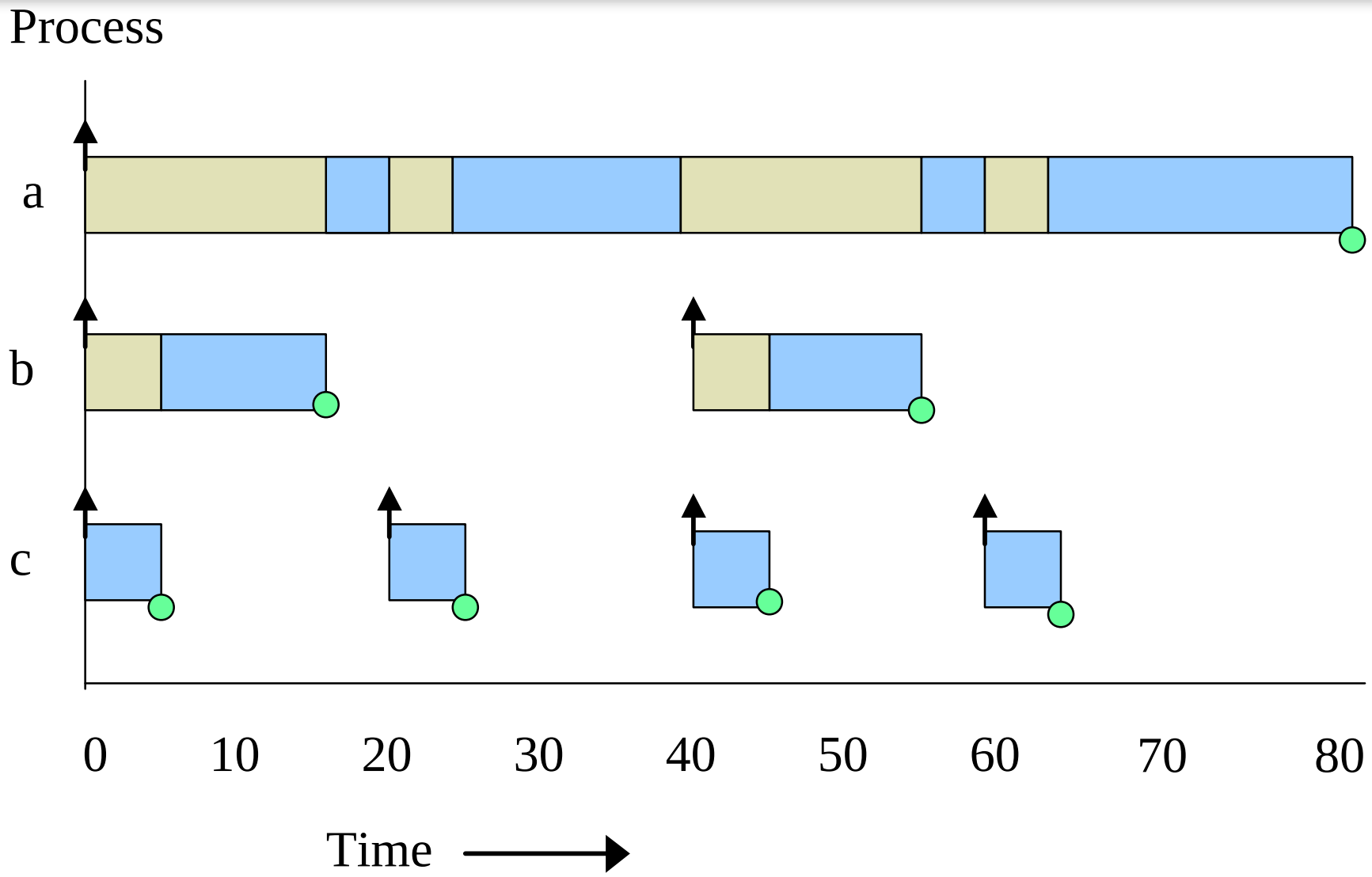
- The combined utilization is 0.775
- This is below the threshold for three processes (0.78) and, hence, this process set will meet all its deadlines

Process Set C

Process	Period T	ComputationTime C	Priority P	Utilization U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

- The combined utilization is 1.0
- This is above the threshold for three processes (0.78) but the process set will meet all its deadlines

Time-line for Process Set C



Criticism of Utilisation-based Tests

- Not exact
- Not general
- BUT it is O(N)

The test is said to be sufficient but not necessary

Utilization-based Test for EDF

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

A much simpler test

- Superior to FPS; it can support high utilizations. However
- FPS is easier to implement as priorities are static
- EDF is dynamic and requires a more complex run-time system which will have higher overhead
- It is easier to incorporate processes without deadlines into FPS; giving a process an arbitrary deadline is more artificial
- It is easier to incorporate other factors into the notion of priority than it is into the notion of deadline
- During overload situations
  - FPS is more predictable; Low priority process miss their deadlines first
  - EDF is unpredictable; a domino effect can occur in which a large number of processes miss deadlines

Response-Time Analysis

- Here task *i*'s worst-case response time, *R*, is calculated first and then checked (trivially) with its deadline

$$R_i \leq D_i$$

$$R_i = C_i + I_i$$

Where *I* is the interference from higher priority tasks

Calculating R

During R, each higher priority task j will execute a number of times:

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

The ceiling function $\lceil \ \rceil$ gives the smallest integer greater than the fractional number on which it acts. So the ceiling of 1/3 is 1, of 6/5 is 2, and of 6/3 is 2.

Total interference is given by:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Response Time Equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Where $hp(i)$ is the set of tasks with priority higher than task i

Solve by forming a recurrence relationship:

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

The set of values $w_i^0, w_i^1, w_i^2, ..., w_i^n, ..$ is monotonically non decreasing
When $w_i^n = w_i^{n+1}$ the solution to the equation has been found, w_i^0 must not be greater that R_i (e.g. 0 or C_i)

Response Time Algorithm

```
for i in 1..N loop -- for each process in turn
  n := 0
  w_i^n := C_i
loop
    calculate new w_i^(n+1)
    if  w_i^(n+1) = w_i^n  then
      R_i = w_i^n
      exit value found
    end if
    if  w_i^(n+1) > T_i  then
      exit value not found
    end if
    n := n + 1
  end loop
end loop
```

Process Set D

Process	Period T	ComputationTime C	Priority P
a	7	3	3
b	12	3	2
c	20	5	1

$$R_a = 3$$

$$w_b^0 = 3$$

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$w_b^2 = 6 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$R_b = 6$$

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$$R_c = 20$$

Revisit: Process Set C

Process	Period T	ComputationTime C	Priority P	Response time R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

- The combined utilization is 1.0
- This was above the ulilization threshold for three processes (0.78), therefore it failed the test
- The response time analysis shows that the process set will meet all its deadlines
- RTA is necessary and sufficient

Response Time Analysis

- Is sufficient and necessary
- If the process set passes the test they will meet all their deadlines; if they fail the test then, at run-time, a process will miss its deadline (unless the computation time estimations themselves turn out to be pessimistic)

Worst-Case Execution Time - WCET

- Obtained by either measurement or analysis
- The problem with measurement is that it is difficult to be sure when the worst case has been observed
- The drawback of analysis is that an effective model of the processor (including caches, pipelines, memory wait states and so on) must be available

WCET— Finding C

Most analysis techniques involve two distinct activities.

- The first takes the process and decomposes its code into a directed graph of basic blocks
- These basic blocks represent straight-line code.
- The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time
- Once the times for all the basic blocks are known, the directed graph can be collapsed

Need for Semantic Information

```
for I in 1.. 10 loop
   if Cond then
      -- basic block of cost 100
   else
      -- basic block of cost 10
   end if;
end loop;
```

- Simple cost 10*100 (+overhead), say 1005.

- But if `Cond` only true on 3 occasions then cost is 375

Sporadic Processes

- Sporadics processes have a minimum inter-arrival time
- They also require D<T

- The response time algorithm for fixed priority scheduling works perfectly for values of D less than T as long as the stopping criteria becomes $W_i^{n+1} > D_i$
- It also works perfectly well with any priority ordering — hp(i) always gives the set of higher-priority processes

Hard and Soft Processes

- In many situations the worst-case figures for sporadic processes are considerably higher than the averages
- Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation
- Measuring schedulability with worst-case figures may lead to very low processor utilizations being observed in the actual running system

General Guidelines

Rule 1 — all processes should be schedulable using average execution times and average arrival rates

Rule 2 — all hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

- A consequent of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines
- This condition is known as a transient overload
- Rule 2 ensures that no hard real-time process will miss its deadline
- If Rule 2 gives rise to unacceptably low utilizations for “normal execution” then action must be taken to reduce the worst-case execution times (or arrival rates)

Aperiodic Processes

- These do not have minimum inter-arrival times
- Can run aperiodic processes at a priority below the priorities assigned to hard processes, therefore, they cannot steal, in a pre-emptive system, resources from the hard processes
- This does not provide adequate support to soft processes which will often miss their deadlines
- To improve the situation for soft processes, a server can be employed.
- Servers protect the processing resources needed by hard processes but otherwise allow soft processes to run as soon as possible.
- POSIX supports Sporadic Servers

Process Sets with D < T

- For D = T, Rate Monotonic priority ordering is optimal
- For D < T, Deadline Monotonic priority ordering is optimal

$$D_i < D_j \Rightarrow P_i > P_j$$

D < T Example Process Set

Process	Period T	Deadline D	ComputationTime C	Priority P	Response time R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

Proof that DMPO is Optimal

- Deadline monotonic priority ordering (DMPO) is optimal if any process set, Q, that is schedulable by priority scheme, W, is also schedulable by DMPO

- The proof of optimality of DMPO involves transforming the priorities of Q (as assigned by W) until the ordering is DMPO
- Each step of the transformation will preserve schedulability

DMPO Proof Continued

- Let `i` and `j` be two processes (with adjacent priorities) in `Q` such that under `W`

$$P_i > P_j \wedge D_i > D_j$$

- Define scheme `W'` to be identical to `W` except that processes `i` and `j` are swapped

Consider the schedulability of `Q` under `W'`

- All processes with priorities greater than P_i will be unaffected by this change to lower-priority processes
- All processes with priorities lower than P_j will be unaffected; they will all experience the same interference from `i` and `j`
- Process `j`, which was schedulable under `W`, now has a higher priority, suffers less interference, and hence must be schedulable under `W'`

DMPO Proof Continued

- All that is left is the need to show that process i, which has had its priority lowered, is still schedulable
- Under W

$$R_j < D_j, D_j < D_i \; and \; D_i \leq T_i$$

- Hence process j only interferes once during the execution of i
- It follows that:

$$R'_i = R_j \leq D_j < D_i$$

- It can be concluded that process i is schedulable after the switch
- Priority scheme W′ can now be transformed to W" by choosing two more processes that are in the wrong order for DMP and switching them

Process Interactions and Blocking

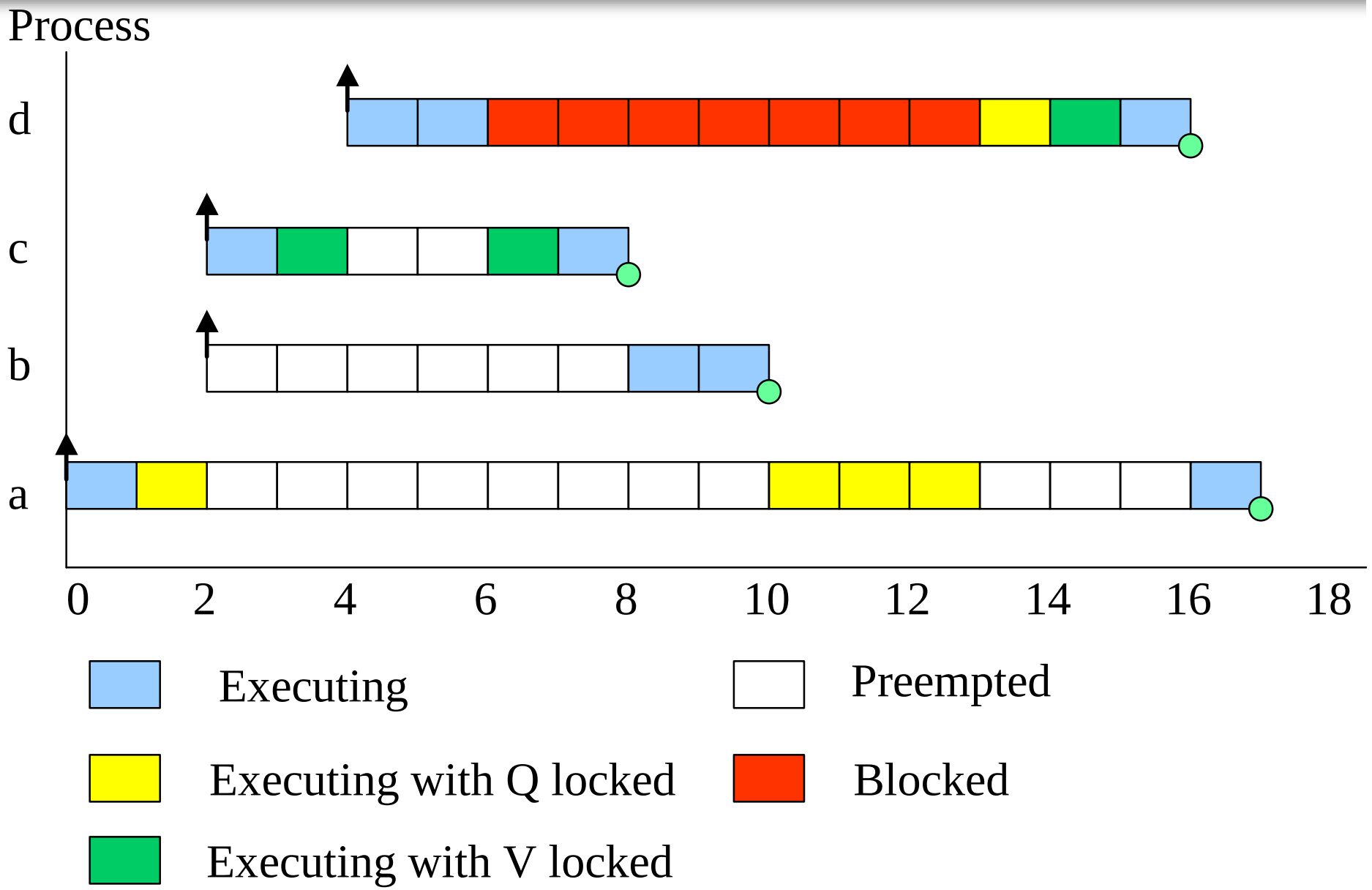
- If a process is suspended waiting for a lower-priority process to complete some required computation then the priority model is, in some sense, being undermined
- It is said to suffer priority inversion
- If a process is waiting for a lower-priority process, it is said to be blocked

Priority Inversion

- To illustrate an extreme example of priority inversion, consider the executions of four periodic processes: a, b, c and d; and two resources: Q and V

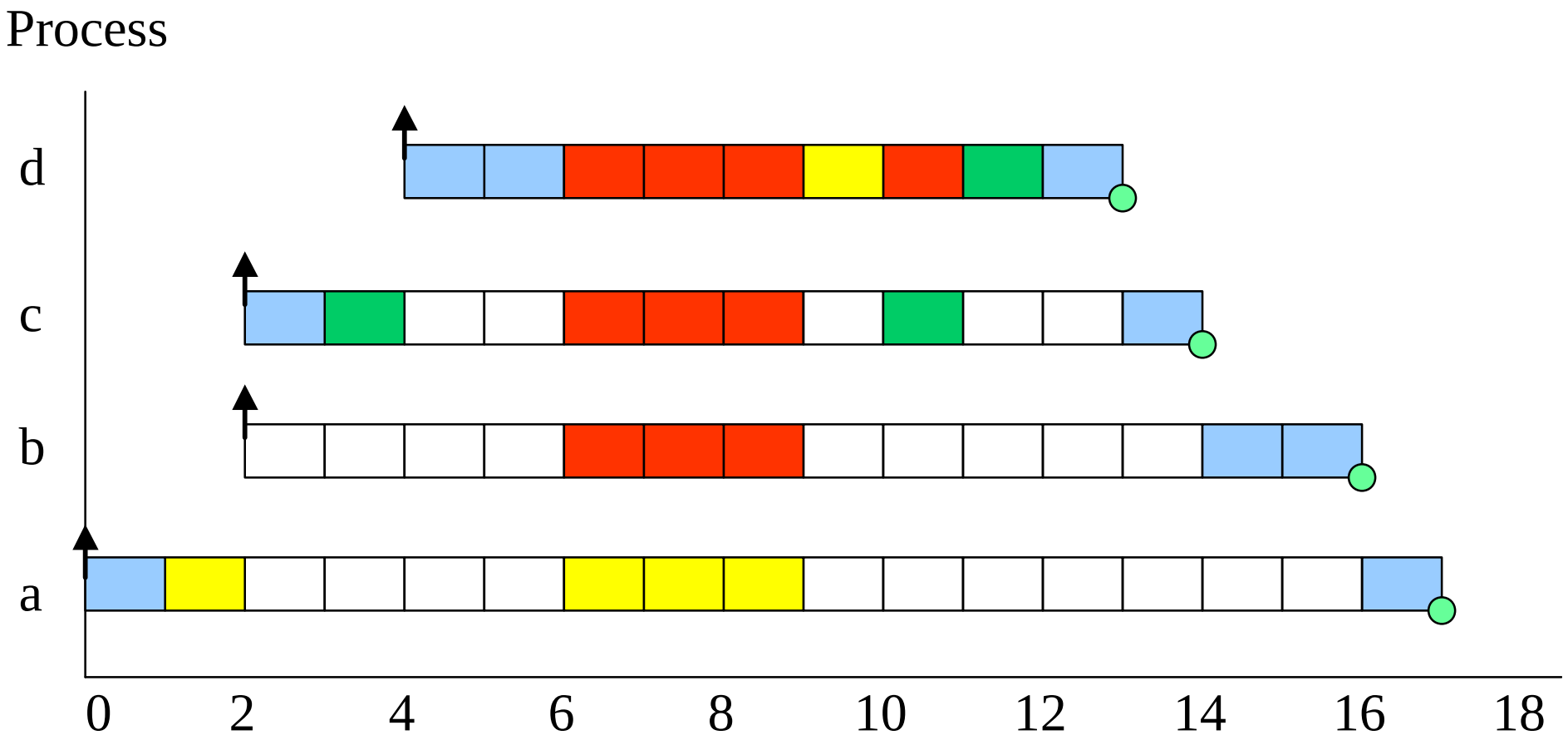
Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Example of Priority Inversion



Priority Inheritance

- If process p is blocking process q, then q runs with p's priority



Calculating Blocking

- If a process has `m` critical sections that can lead to it being blocked then the maximum number of times it can be blocked is `m`
- If `B` is the maximum blocking time and `K` is the number of critical sections, the process `i` has an upper bound on its blocking given by:

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

Response Time and Blocking

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Priority Ceiling Protocols

Two forms

- Original ceiling priority protocol
- Immediate ceiling priority protocol

On a Single Processor

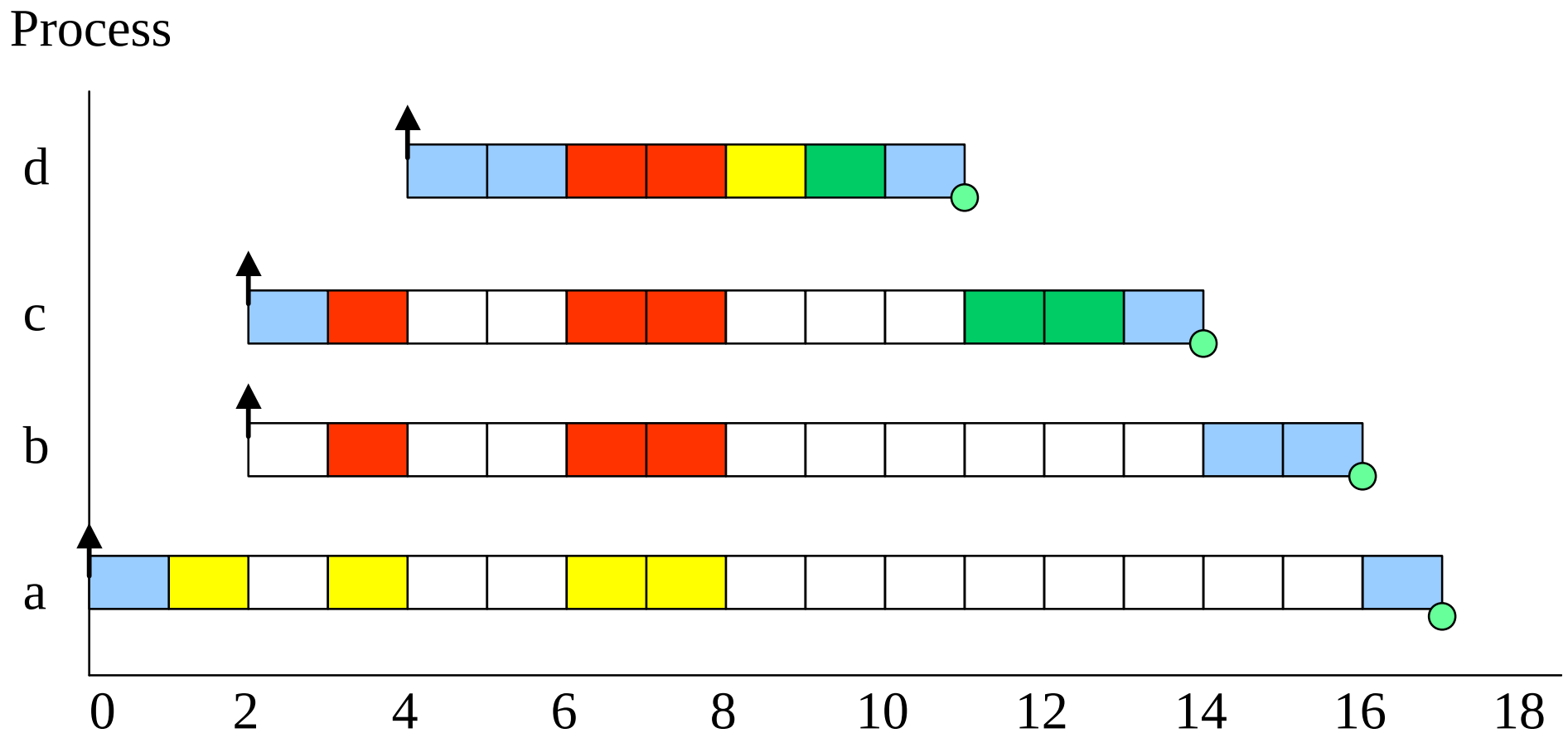
- A high-priority process can be blocked at most once during its execution by lower-priority processes
- Deadlocks are prevented
- Transitive blocking is prevented
- Mutual exclusive access to resources is ensured (by the protocol itself

OCPP

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme)
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it
- A process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself)

$$B_i = \max_{k=1}^{k} usage(k,i)C(k)$$

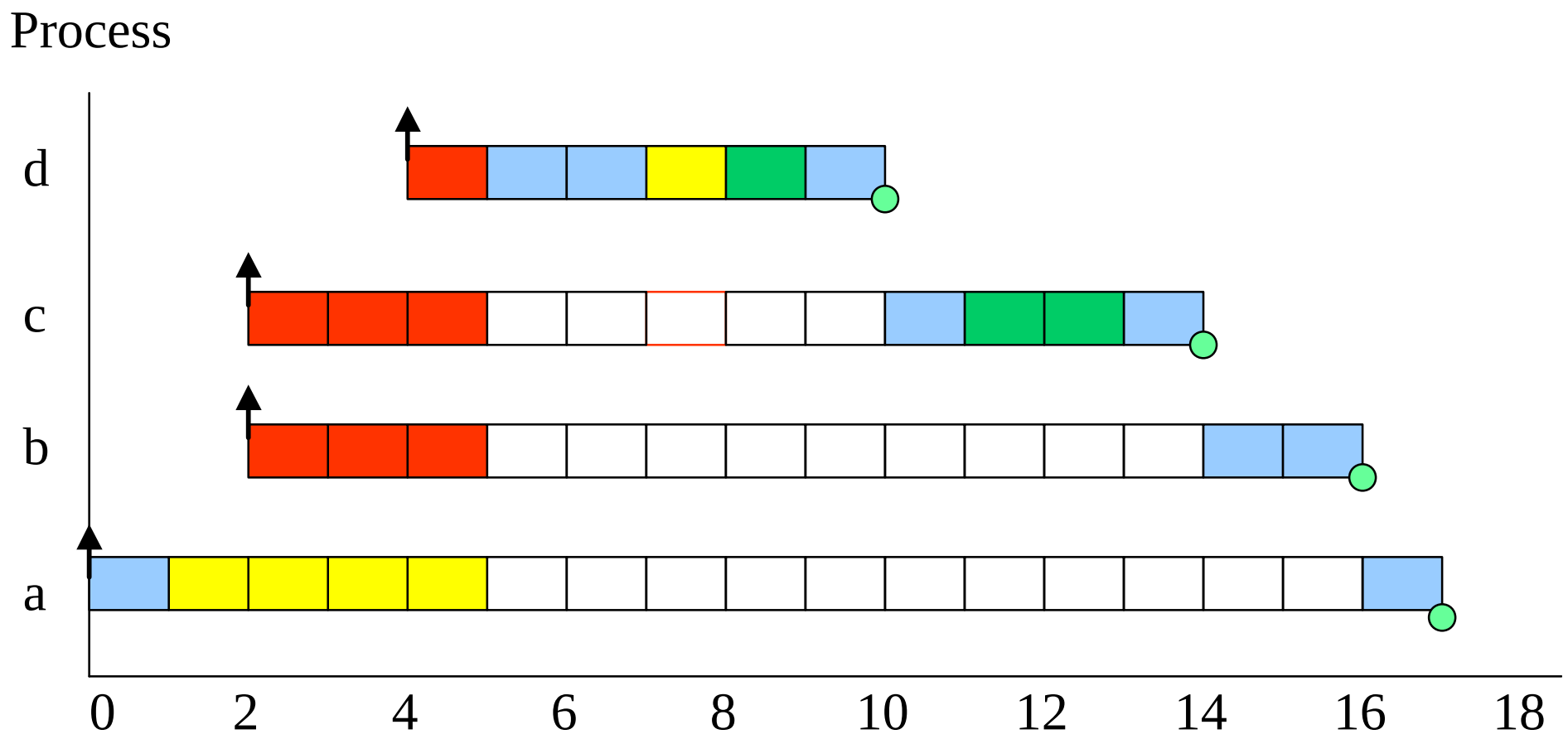
OCPP Inheritance



ICPP

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
- A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked
- As a consequence, a process will only suffer a block at the very beginning of its execution
- Once the process starts actually executing, all the resources it needs must be free; if they were not, then some process would have an equal or higher priority and the process's execution would be postponed

ICPP Inheritance



OCPP versus ICPP

- Although the worst-case behaviour of the two ceiling schemes is identical (from a scheduling view point), there are some points of difference:
  - ICCP is easier to implement than the original (OCPP) as blocking relationships need not be monitored
  - ICPP leads to less context switches as blocking is prior to first execution
  - ICPP requires more priority movements as this happens with all resource usage
  - OCPP changes priority only if an actual block has occurred
- Note that ICPP is called Priority Protect Protocol in POSIX and Priority Ceiling Emulation in Real-Time Java

An Extendible Process Model

So far:

- Deadlines can be less than period (D<T)
- Sporadic and aperiodic processes, as well as periodic processes, can be supported
- Process interactions are possible, with the resulting blocking being factored into the response time equations

Extensions

- Cooperative Scheduling
- Release Jitter
- Arbitrary Deadlines
- Fault Tolerance
- Offsets
- Optimal Priority Assignment

Cooperative Scheduling

- True preemptive behaviour is not always acceptable for safety-critical systems
- Cooperative or deferred preemption splits processes into slots
- Mutual exclusion is via non-preemption
- The use of deferred preemption has two important advantages
  - It increases the schedulability of the system, and it can lead to lower values of C
  - With deferred preemption, no interference can occur during the last slot of execution.

Cooperative Scheduling

- Let the execution time of the final block be F_i

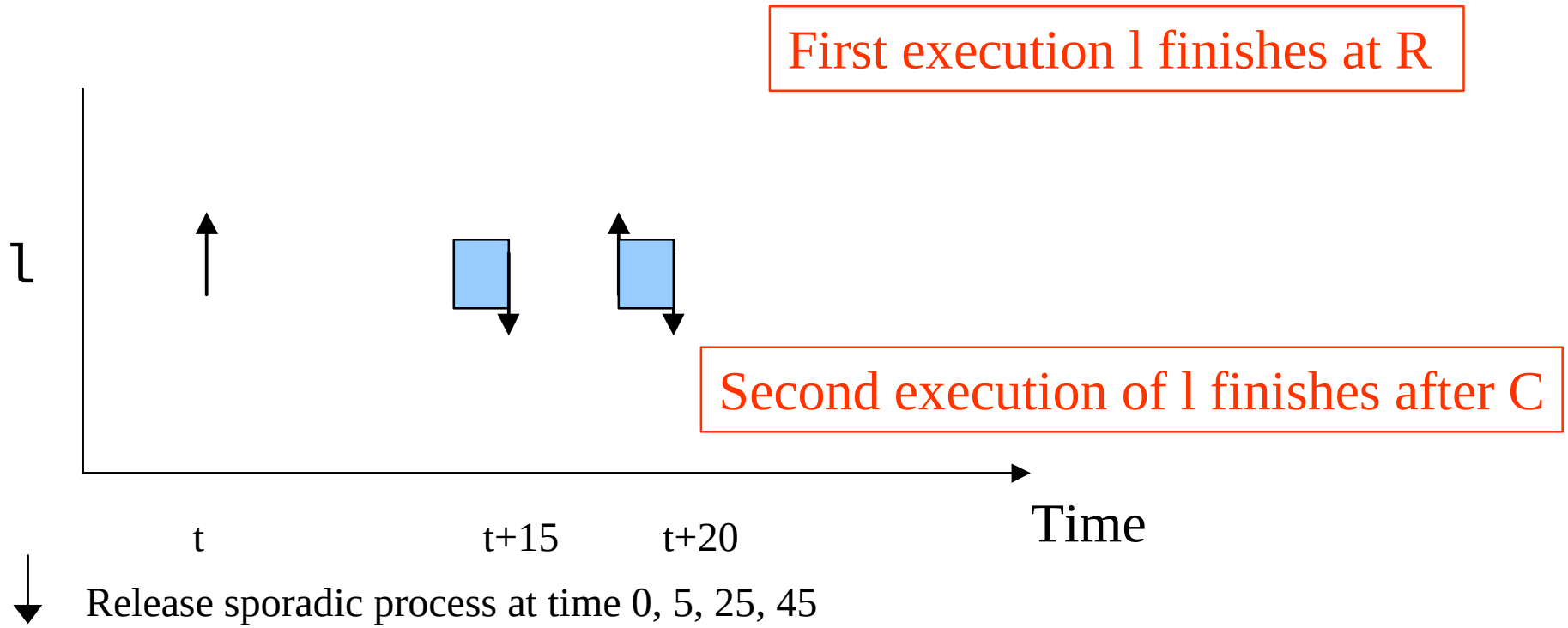
$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- When this converges that is, $w_i^n = w_i^{n+1}$, the response time is given by:

$$R_i = w_i^n + F_i$$

Release Jitter

- A key issue for distributed systems
- Consider the release of a sporadic process on a different processor by a periodic process, `l`, with a period of 20



Release Jitter

- Sporadic is released at `0, T-J, 2T-J, 3T-J`
- Examination of the derivation of the schedulability equation implies that process `i` will suffer
  - one interference from process s if $R_i \in [0, T - J)$
  - two interfernces if $R_i \in [T - J, 2T - J)$
  - three interference if $R_i \in [2T - J, 3T - J)$
- This can be represented in the response time equations

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$

- If response time is to be measured relative to the real release time then the jitter value must be added

$$R_i^{periodic} = R_i + J_i$$

Arbitrary Deadlines

- To cater for situations where D (and hence potentially R) > T

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

$$R_i(q) = w_i^n(q) - qT_i$$

- The number of releases is bounded by the lowest value of q for which the following relation is true: $R_i(q) \le T_i$
- The worst-case response time is then the maximum value found for each q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

Arbitrary Deadlines

- When formulation is combined with the effect of release jitter, two alterations to the above analysis must be made
- First, the interference factor must be increased if any higher priority processes suffers release jitter:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j$$

- The other change involves the process itself. If it can suffer release jitter then two consecutive windows could overlap if response time plus jitter is greater than period.

$$R_i(q) = w_i^n(q) - qT_i + J_i$$

Fault Tolerance

- Fault tolerance via either forward or backward error recovery always results in extra computation
- This could be an exception handler or a recovery block.
- In a real-time fault tolerant system, deadlines should still be met even when a certain level of faults occur
- This level of fault tolerance is know as the fault model
- If the extra computation time that results from an error in process `i` is C_i^f

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f$$

- where `hep(i)` is set of processes with priority equal to or higher than `i`

Fault Tolerance

- If F is the number of faults allows

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f$$

- If there is a minimum arrival interval T_f

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

Offsets

- So far assumed all processes share a common release time (critical instant)

Process	T	D	C	R
a	8	5	4	4
b	20	10	4	8
c	20	12	4	16

- With offsets

Process	T	D	C	O	R
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	8

Arbitrary offsets are not amenable to analysis

Non-Optimal Analysis

- In most realistic systems, process periods are not arbitrary but are likely to be related to one another
- As in the example just illustrated, two processes have a common period. In these situations it is ease to give one an offset (of $T/2$) and to analyse the resulting system using a transformation technique that removes the offset — and, hence, critical instant analysis applies.
- In the example, processes b and c (having the offset of 10) are replaced by a single notional process with period 10, computation time 4, deadline 10 but no offset

Non-Optimal Analysis

- This notional process has two important properties.
  - If it is schedulable (when sharing a critical instant with all other processes) then the two real process will meet their deadlines when one is given the half period offset
  - If all lower priority processes are schedulable when suffering interference from the notional process (and all other high-priority processes) then they will remain schedulable when the notional process is replaced by the two real process (one with the offset).
- These properties follow from the observation that the notional process always uses more (or equal) CPU time than the two real process

Process	T	D	C	O	R
a	8	5	4	0	4
n	10	10	4	0	8

Notional Process Parameters

$$T_n = \frac{T_a}{2} = \frac{T_b}{2}$$

$$C_n = Max(C_a, C_b)$$

$$D_n = Min(D_a, D_b)$$

$$P_n = Max(P_a, P_b)$$

Can be extended to more than two processes

Priority Assignment

Theorem

- If process p is assigned the lowest priority and is feasible then, if a feasible priority ordering exists for the complete process set, an ordering exists with process p assigned the lowest priority

```
procedure Assign_Pri (Set : in out Process_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok;  -- failed to find a schedulable process
  end loop;
end Assign_Pri;
```

Dynamic Systems and Online Analysis

- There are dynamic soft real-time applications in which arrival patterns and computation times are not known *a priori*
- Although some level of off-line analysis may still be applicable, this can no longer be complete and hence some form of on-line analysis is required
- The main task of an on-line scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment
- EDF is a dynamic scheduling scheme that is an optimal
- During transient overloads EDF performs very badly. It is possible to get a cascade effect in which each process misses its deadline but uses sufficient resources to result in the next process also missing its deadline.

Admission Schemes

- To counter this detrimental domino effect many on-line schemes have two mechanisms:
  - an admissions control module that limits the number of processes that are allowed to compete for the processors, and
  - an EDF dispatching routine for those processes that are admitted
- An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively

Values

- If some processes are to be admitted, whilst others rejected, the relative importance of each process must be known
- This is usually achieved by assigning value
- Values can be classified
  - Static: the process always has the same value whenever it is released.
  - Dynamic: the process's value can only be computed at the time the process is released (because it is dependent on either environmental factors or the current state of the system)
  - Adaptive: here the dynamic nature of the system is such that the value of the process will change during its execution
- To assign static values requires the domain specialists to articulate their understanding of the desirable behaviour of the system

Programming Priority-Based Systems

- Ada
- POSIX
- Real-Time Java

Ada: Real-Time Annex

- Ada 95 has a flexible model:
  - base and active priorities
  - priority ceiling locking
  - various dispatching policies using active priority
  - dynamic priorities

```
subtype Any_Priority is Integer
        range Implementation-Defined;
subtype Priority is Any_Priority range
        Any_Priority'First .. Implementation-Defined;
subtype Interrupt_Priority is Any_Priority range
        Priority'Last + 1 .. Any_Priority'Last;
Default_Priority : constant Priority :=
        (Priority'First + Priority'Last)/2;
```

An implementation must support a range of `Priority` of at least 30 and at least one distinct `Interrupt_Priority`

Assigning Base Priorities

- Using a pragma

```
task Controller is
  pragma Priority(10);
end Controller;


task type Servers(Pri : System.Priority) is
  -- each instance of the task can have a
  -- different priority
  entry Service1(...);
  entry Service2(...);
  pragma Priority(Pri);
end Servers;
```

Priority Ceiling Locking

- Protected objects need to maintain the consistency of their data
- Mutual exclusion can be guaranteed by use of the priority model
- Each protected object is assigned a ceiling priority which is greater than or equal to the highest priority of any of its calling tasks
- When a task calls a protected operation, its priority is immediately raised to that of the protected object
- If a task wishing to enter a protected operation is running then the protected object cannot be already occupied

Ceiling Locking

- Each protected object is assigned a priority using a pragma
- If the pragma is missing, `Priority'Last` is assumed
- `Program_Error` is raised if the calling task's active priority is greater than the ceiling
- If an interrupt handler is attached to a protected operation and the wrong ceiling priority has been set, then the program becomes erroneous
- With ceiling locking, an effective implementation will use the thread of the calling task to execute not only the protected operation but also to execute the code of any other tasks that are released as a result of the call

Example of Ceiling Priority

```
protected Gate_Control is
   pragma Priority(28);
   entry Stop_And_Close;
   procedure Open;
private
   Gate : Boolean := False;
end Gate_Control;
```

```
protected body Gate_Control is
   entry Stop_And_Close
            when Gate is
   begin
      Gate := False;
   end;
   procedure Open is
   begin
      Gate := True;
   end;
end Gate_Control;
```

Example

- Assume task `T`, priority 20, calls `Stop_And_Close` and is blocked. Later task `S`, priority 27, calls `Open`. The thread executing `S` will undertake the following operations:
  - the code of `Open` for `S`
  - evaluate the barrier on the entry and note that `T` can now proceed
  - the code `Stop_And_Close` for `T`
  - evaluate the barrier again
  - continue with the execution of `S` after its call on the protected object
- There is no context switch

Active Priorities

- A task entering a protected operation has its priority raised

- A task's active priority might also change during:
  - task activation — a task inherits the active priority of the parent task which created it (to avoid priority inversion)
  - during a rendezvous — the task executing a rendezvous will inherit the active priority of the caller if it is greater than its current active priority
  - Note: no inheritance when waiting for task termination

Dispatching

- The order of dispatching is determined by the tasks' active priorities
- Default is preemptive priority based
- Not defined exactly what this means on a multi-processor system
- One policy defined by annex:
  `FIFO_Within_Priority`
- When a task becomes runnable it is placed at the back on the run queue for its priority; when it is preempted, it is placed at the front

Entry Queue Policies

- A programmer may choose the queuing policy for a task's entry queue and the select statement
- Two predefined policies: `FIFO_Queuing` (default) and `Priority_Queuing`
- With `Priority_Queuing` and the select statement, an alternative that is open and has the highest priority task queued (of all open alternatives) is chosen
- If there are two open with equal priority tasks, the one which appears textually first in the program is chosen
- Tasks are queued in active priority order, if active priority changes then no requeuing takes place; if the base priority changes, the task is removed and requeued

Dynamic Priorities

- Some applications require the base priority of a task to change dynamically: e.g., mode changes, or to implement dynamic scheduling schemes such as earliest deadline scheduling

Package Specification

```
with Ada.Task_Identification; use Ada;
package Ada.Dynamic_Priorities is
   procedure Set_Priority(Priority : System.Any_Priority;
     T : Task_Identification.Task_Id :=
     Task_Identification.Current_Task);
   function Get_Priority(T : T_Identification.Task_Id
      := Task_Identification.Current_Task)
      return System.Any_Priority;
      -- raise Tasking_Error if task has terminated
   -- Both raise Program_Error if a Null_Task_Id is passed
private
   -- not specified by the language
end Ada.Dynamic_Priorities;
```

Dynamic Priorities

- The effect of a change of base priorities should be as soon as practical but not during an abort deferred operation and no later than the next abort completion point

- Changing a task's base priority can affect its active priority and have an impact on dispatching and queuing

POSIX

- POSIX supports priority-based scheduling, and has options to support priority inheritance and ceiling protocols
- Priorities may be set dynamically
- Within the priority-based facilities, there are four policies:
  - FIFO: a process/thread runs until it completes or it is blocked
  - Round-Robin: a process/thread runs until it completes or it is blocked or its time quantum has expired
  - Sporadic Server: a process/thread runs as a sporadic server
  - OTHER: an implementation-defined
- For each policy, there is a minimum range of priorities that must be supported; 32 for FIFO and round-robin
- The scheduling policy can be set on a per process and a per thread basis

POSIX

- Threads may be created with a system contention option, in which case they compete with other system threads according to their policy and priority
- Alternatively, threads can be created with a process contention option where they must compete with other threads (created with a process contention) in the parent process
  - It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention
- A specific implementation must decide which to support

Sporadic Server

- A sporadic server assigns a limited amount of CPU capacity to handle events, has a replenishment period, a budget, and two priorities
- The server runs at a high priority when it has some budget left and a low one when its budget is exhausted
- When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget
- The amount of budget consumed is replenished at the time the server was activated plus the replenishment period
- When its budget reaches zero, the server's priority is set to the low value

Other Facilities

POSIX allows:

- priority inheritance to be associated with mutexes (priority protected protocol= ICPP)
- message queues to be priority ordered
- functions for dynamically getting and setting a thread's priority
- threads to indicate whether their attributes should be inherited by any child thread they create

RT Java Threads and Scheduling

- There are two entities in Real-Time Java which can be scheduled:
  - `RealtimeThreads` (and `NoHeapRealtimeThread`)
  - `AsynEventHandler` (and `BoundAyncEventHandler`)

- Objects which are to be scheduled must
  - implement the `Schedulable` interface
  - specify their
    - `SchedulingParameters`
    - `ReleaseParameters`
    - `MemoryParameters`

Real-Time Java

- Real-Time Java implementations are required to support at least 28 real-time priority levels
- As with Ada and POSIX, the larger the integer value, the higher the priority
- Non real-time threads are given priority levels below the minimum real-time priority
- Note, scheduling parameters are bound to threads at thread creation time; if the parameter objects are changed, they have an immediate impact on the associated thread
- Like Ada and Real-Time POSIX, Real-Time Java supports a pre-emptive priority-based dispatching policy
- Unlike Ada and RT POSIX, RT Java does not require a preempted thread to be placed at the head of the run queue associated with its priority level

The Schedulable Interface

```
public Interface Schedulable extends java.lang.Runnable
{
  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);


  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters release);

  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(
          SchedulingParameters scheduling);


  public Scheduler getScheduler();
  public void setScheduler(Scheduler scheduler);
}
```

Scheduling Parameters

```
public abstract class SchedulingParameters
{  public SchedulingParameters(); }

public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters(int priority);

  public int getPriority(); // at least 28 priority levels
  public void setPriority(int priority) throws
                          IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters
{
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```

RT Java: Scheduler

- Real-Time Java supports a high-level scheduler whose goals are:
  - to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm, and
  - to set the priority of the schedulable objects according to the priority assignment algorithm associated with the feasibility algorithm
- Hence, whilst Ada and Real-Time POSIX focus on static off-line schedulability analysis, Real-Time Java addresses more dynamic systems with the potential for on-line analysis

The Scheduler

```
public abstract class Scheduler
{
  public Scheduler();
  protected abstract void addToFeasibility(Schedulable s);
  protected abstract void removeFromFeasibility(Schedulable s);

  public abstract boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
         ReleaseParameters release, MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

The Scheduler

- The `Scheduler` abstract class
- The `isFeasible` method considers only the set of schedulable objects that have been added to its feasibility list (via the `addToFeasibility` and `removeFromFeasibility` methods)
- The method `changeIfFeasible` checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed
- If it is, the parameters are changed
- Static methods allow the default scheduler to be queried or set
- RT Java does not require an implementation to provide an on-line feasibility algorithm

The Priority Scheduler

```
class PriorityScheduler extends Scheduler
{
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  ...

  public void fireSchedulable(Schedulable schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();

  public static PriorityScheduler instance();
  ...
}
```

Standard preemptive priority-based scheduling

Other Facilities

- Priority inheritance and ICCP (called priority ceiling emulation)
- Support for aperiodic threads in the form of processing groups; a group of aperiodic threads can be linked together and assigned characteristics which aid the feasibility analysis

Summary

- A scheduling scheme defines an algorithm for resource sharing and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.
- With a cyclic executive, the application code must be packed into a fixed number of minor cycles such that the cyclic execution of the sequence of minor cycles (the major cycle) will enable all system deadlines to be met
- The cyclic executive approach has major drawbacks many of which are solved by priority-based systems
- Simple utilization-based schedulability tests are not exact

Summary

- Response time analysis is flexible and caters for:
  - Periodic and sporadic processes
  - Blocking caused by IPC
  - Cooperative scheduling
  - Arbitrary deadlines
  - Release jitter
  - Fault tolerance
  - Offsets
- Ada, RT POSIX and RT Java support preemptive priority-based scheduling
- Ada and RT POSIX focus on static off-line schedulability analysis, RT Java addresses more dynamic systems with the potential for on-line analysis