Real-time Scheduling

COE718: Embedded System Design

http://www.ee.ryerson.ca/~courses/coe718/

Dr. Gul N. Khan

http://www.ee.ryerson.ca/~gnkhan

Electrical and Computer Engineering

Ryerson University

Overview

- RTX - Preemptive Scheduling
- Real-time Scheduling Techniques
  - Fixed-Priority and Earliest Deadline First Scheduling
- Utilization and Response-time Analysis
- Priority Inversion
- Sporadic and Aperiodic Process Scheduling

Chapters 9 and 10 of Text by D. W. Lewis, and Keil-RTX documents

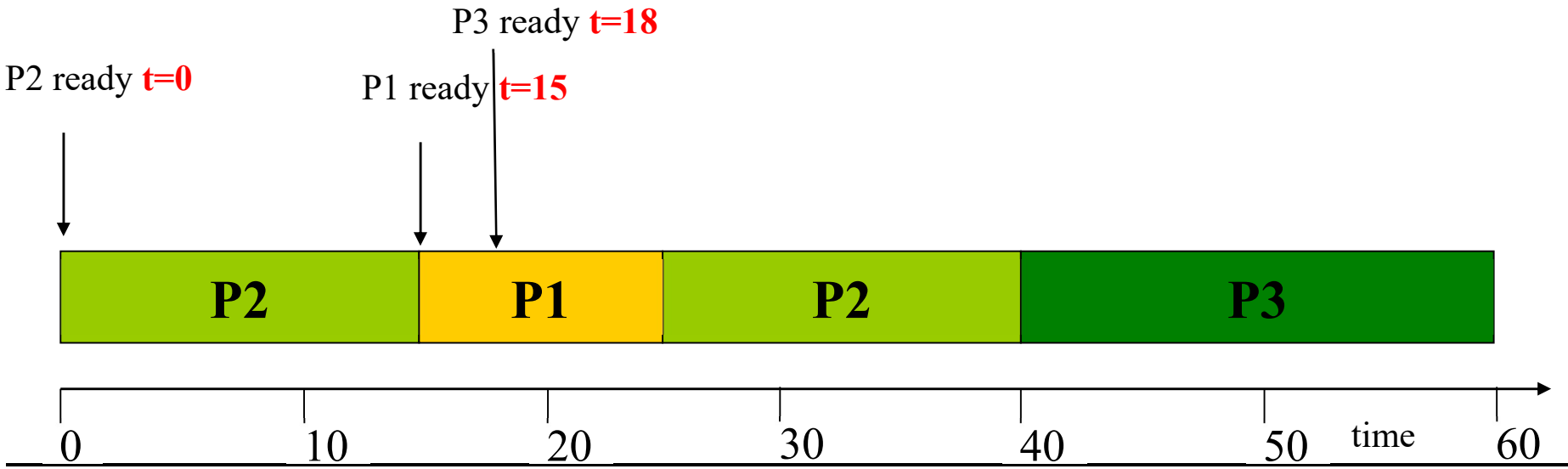
Priority-driven Scheduling

Rules:

- each process has a fixed priority (1 highest);
- highest-priority ready process gets CPU;
- process continues until done.

Processes

- P1: priority 1, execution time 10
- P2: priority 2, execution time 30
- P3: priority 3, execution time 20



RTX Scheduling Options

RTX allows us to build an application with three different kernel-scheduling options:

- **Pre-Emptive scheduling**
  Each task has a **different priority** and will run until it is pre-empted.
- **Round-Robin scheduling**
  Each task has the **same priority** and will run for a fixed period, or time slice.

- **Co-operative multi-tasking**
  Each task has the **same priority** and the Round-Robin is disabled. Each task will run until it reached a blocking OS call.

RTX: Preemptive Scheduling

When a task with a higher priority than the currently running task becomes ready to run, RTX **suspends** the currently running task. A preemptive task switch occurs when:

- the task scheduler is executed from the system **tick timer interrupt**. Task scheduler processes the **delays** of tasks. If the delay for a task with a higher priority has expired, then the higher priority task starts to execute instead of the current task.
- an **event** is set for a higher priority task by the currently running task or by an interrupt service routine. The currently running task is suspended, and the higher priority task starts to run.
- a token is returned to a **semaphore**, and a higher priority task is waiting for the semaphore token. The currently running task is suspended, and the higher priority task starts to run. The token can be returned by the currently running task or by an interrupt service routine.
- a **mutex** is released and a higher priority task is waiting for the mutex. The currently running task is suspended, and the higher priority task starts to run.

Pre-emptive Scheduling

A preemptive task switch occurs when:

- a message is posted to a **mailbox**, and a higher priority task is waiting for the mailbox message. The currently running task is suspended, and the higher priority task starts to run. The message can be posted by the currently running task or by an interrupt service routine.
- a **mailbox** is **full**, and a higher priority task is waiting to post a message to a mailbox. As soon as the currently running task or an interrupt service routine takes a message out from the mailbox, the higher priority task starts to run.
- the **priority** of the currently running task is reduced. If another task is ready to run and has a higher priority than the new priority of the currently running task, then the current task is suspended immediately, and the higher priority task resumes its execution.

os_tsk_prio ( ) function changes the execution priority of the task identified by the argument *task_id* `e.g.`

```
os_tsk_prio_self (5);
os_tsk_prio(tsk2, 10);
```

RTX - Preemptive Switching Example

- Task job1 has a higher priority than task job2.
- When job1 starts, it creates task job2 and then enters the **os_evt_wait_or** function.
- The RTX kernel suspends job1 at this point, and job2 starts executing.
- As soon as job2 sets an event flag for job1, the RTX kernel suspends job2 and then resumes job1.
- Task job1 then increments counter cnt1 and calls the **os_evt_wait_or** function, which suspends it again.
- The kernel resumes job2, which increments counter cnt2 and sets an event flag for job1. **os_evt_set**
- This process of task switching continues indefinitely.

```
#include <rtl.h>

OS_TID tsk1,tsk2;
int    cnt1,cnt2;

__task void job1 (void);
__task void job2 (void);
```

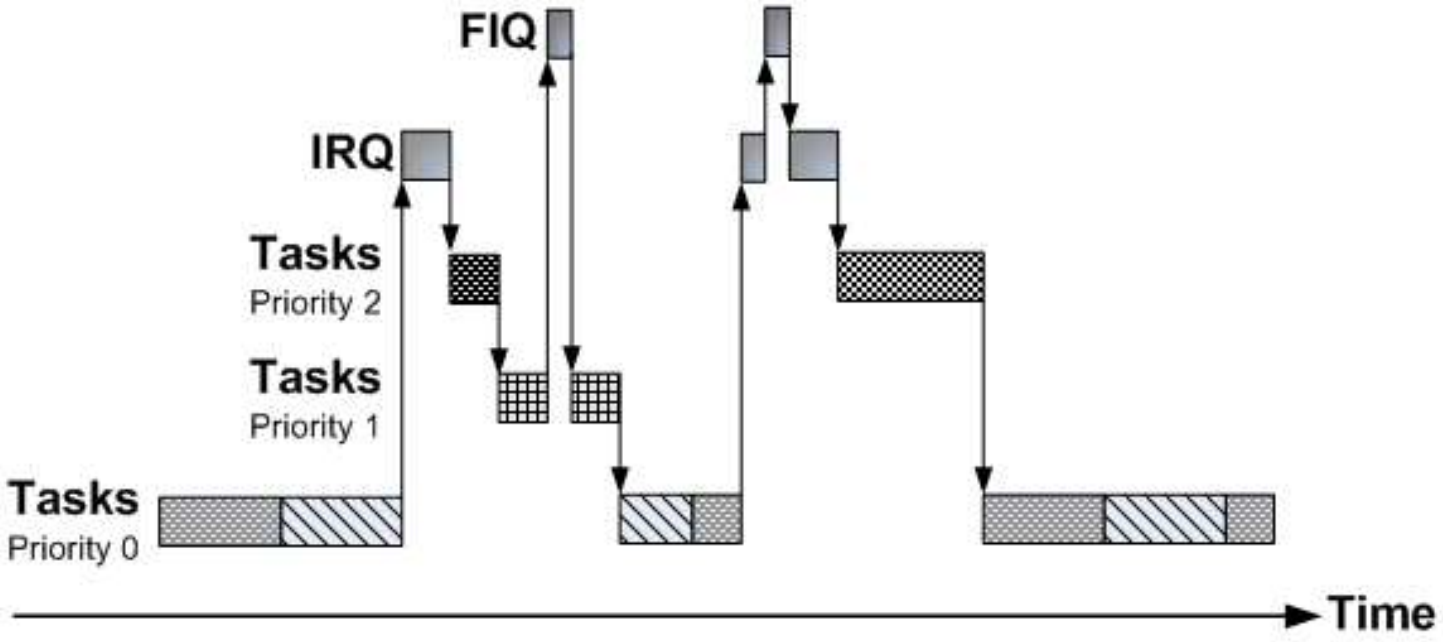
RTX - Task Switching Example (cont.)

```
__task void job1 (void) {
  os_tsk_prio (2);
  tsk1 = os_tsk_self ();
  os_tsk_create (job2, 1);
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    cnt1++;
  }
}

__task void job2 (void) {
  while (1) {
    os_evt_set (0x0001, tsk1);
    cnt2++;
  }
}

void main (void) {
  os_sys_init (job1);
  while (1);
}
```

RTX and Interrupt Functions

RTX can work with interrupt functions in **parallel**. However, it is better to avoid IRQ nesting.

An IRQ function can send a signal/ or message to start a high priority task.



Interrupt functions are added to an ARM application in the same way as in any other non-RTX projects.

FIQ interrupts are never disabled by the RTX kernel.

Threads - Signals/Waits

Threads support is available with for RTX with CMSIS-RTOS API
http://www.keil.com/pack/doc/cmsis_rtx/_using.html

```
void led_Thread1 (void const *argument);
osThreadDef(led_Thread1, osPriorityNormal, 1, 0);
osThreadId Thr_led_ID1;

int main(void){
...
  Thr_led_ID1 =
     osThreadCreate(osThread(led_Thread1), NULL);
...}
```

A wait flag may be set in the code as follows:

```
  osSignalWait (0x03,osWaitForever);
```

The thread is waiting for the signal flag 0x03 to be asserted. `osWaitForever` parameter indicates the maximum duration in msecs that the thread should wait to be signaled.

A signal may be send to a thread or cleared using:

```
  osSignalSet(Thr_led_ID2,0x01);    or
  osSignalClear(Thr_led_ID2, 0x01);
```

The Scheduling Problem

- Can we meet all deadlines?
- Must be able to meet deadlines in all cases.
- How much CPU time, we need to meet the deadlines?

Process Initiation

- Periodic process: executes on (almost) every period.
  Same as Round Robin?
- Aperiodic process: executes on demand.

Analyzing aperiodic process set is harder---must consider worst-case combinations of process activations.
In other words, worst-case scenarios for which the tasks may activate.

Process Timing Requirements

Period: interval between process activations.
Initiation interval: reciprocal of period.
Initiation time: time at which process becomes ready.
Deadline: time at which process must finish.
Timing violations.

What happens if a process doesn't finish by its deadline?

Hard deadline: system fails if missed.
Soft deadline: user may notice, but system doesn't necessarily fail.

Example: Space Shuttle software error A software timing error delayed shuttle's first launch:

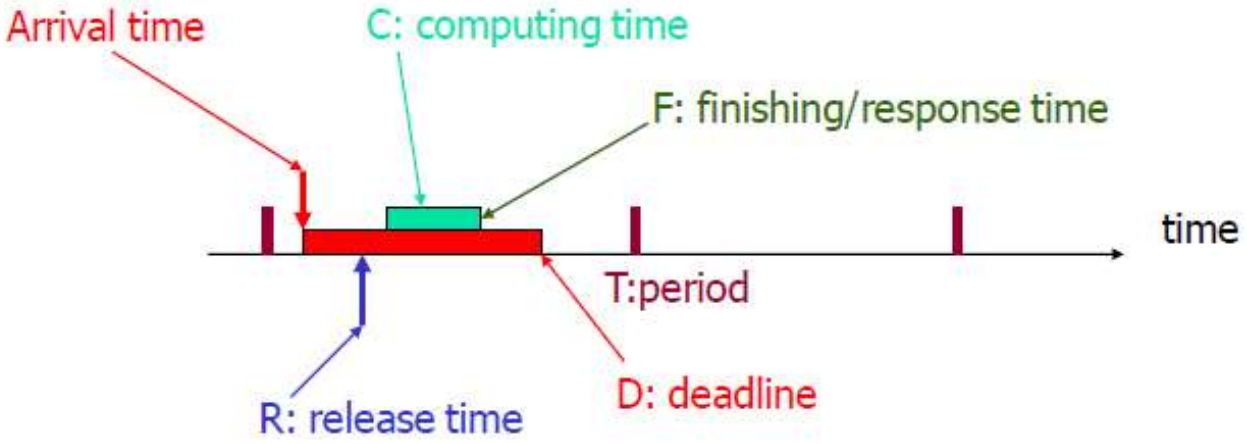
- Primary control system PASS and backup system BFS.
- BFS failed to synchronize with PASS.
- Change to one routine added delay that threw off start time calculation.
- 1 in 67 chance of timing problem.

Process Model

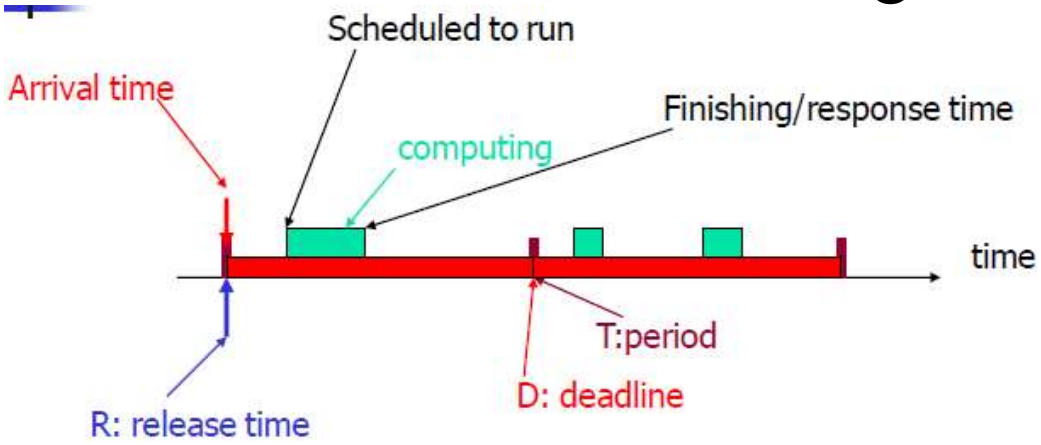
- The application is assumed to consist of a fixed set of processes.
- Processes are completely independent of each other.
- All system's overheads, context-switching times and so on are ignored (i.e. assumed to have zero cost)
- All processes are periodic, with known periods.
- All processes have a 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.

Scheduling Periodic & Aperiodic Tasks

Aperiodic Task Scheduling



Periodic Task Scheduling



Real-time Scheduling Techniques

- Fixed-Priority Scheduling (FPS)
- Earliest Deadline First (EDF)

FPS: Fixed-Priority Scheduling

- This is the most widely used approach.
- Each process has a fixed, (static) priority that is computed before execution.
- 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.

FPS: Fixed-Priority Scheduling

Rate Monotonic Priority Assignment

- Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
- For two processes i and j:

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

- An optimal priority assignment means: if any process set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme
- Priority 1 is the lowest (least) priority

Priority Assignment: An Example

Period T: Minimum time between process releases.
C: Worst-case computation time (WCET) of the process.
U: The utilization of each process (equal to C/T).
R: Worst-case response time of the process.
B: Worst-case blocking time for the process.
D: Deadline of the process.
N: Number of process.
The interference time of the process.
Release jitter of the process.

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

Utilization-Based Analysis

For D=T process sets, a sufficient but not necessary schedulability test exists.

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

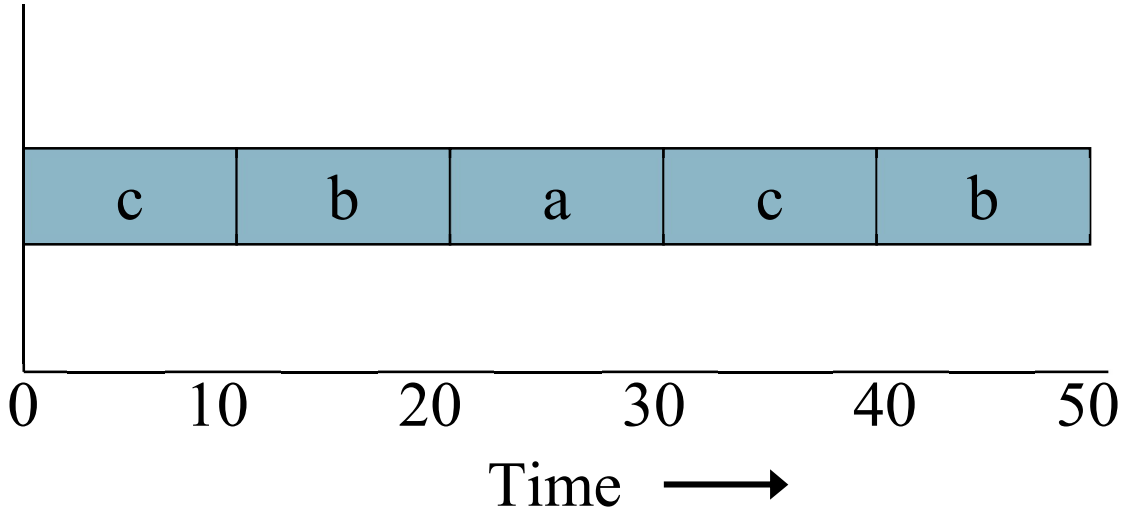
N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

Utilization-Based Analysis

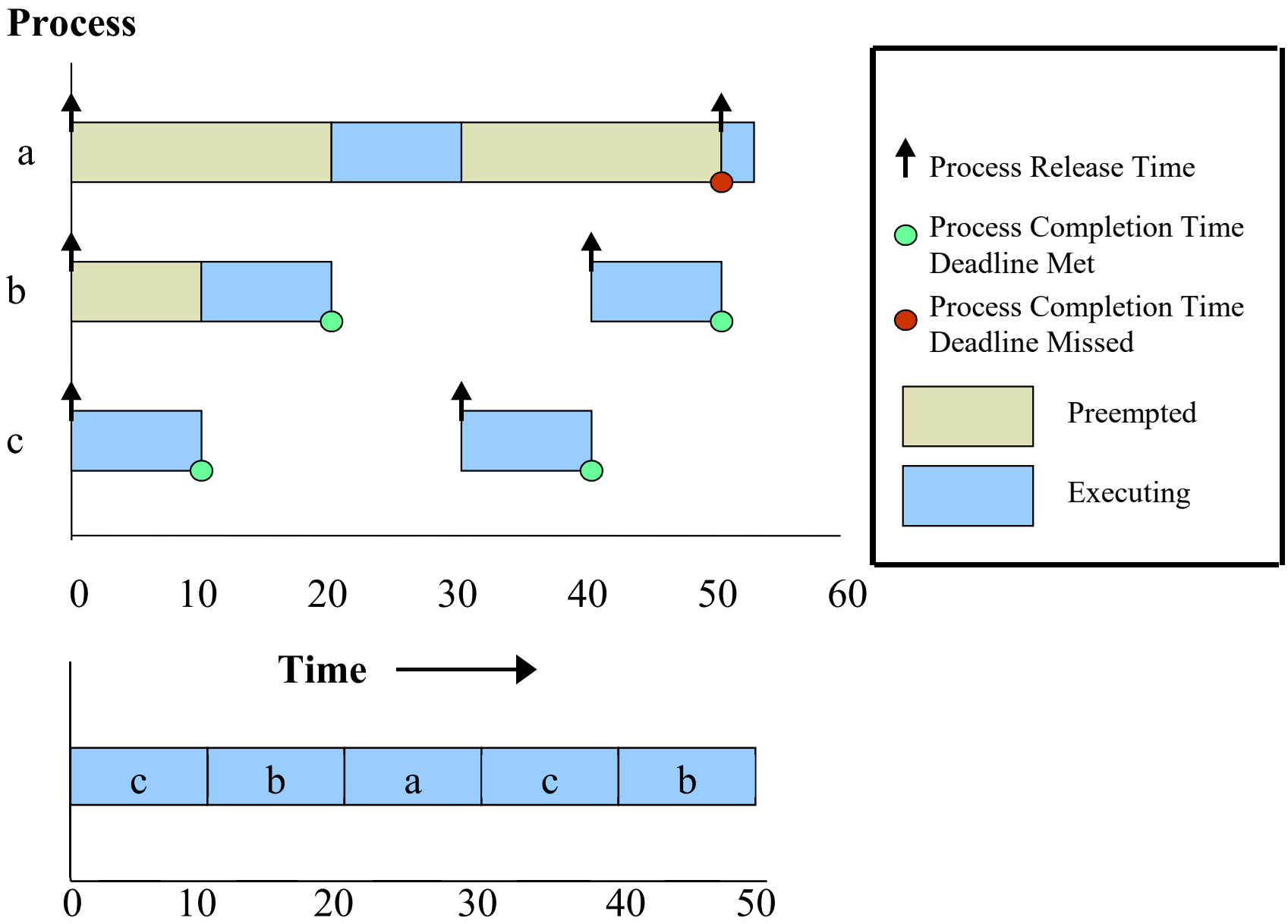
Process Set A

Process	Period, T	Computation Time, 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



Utilization-Based Analysis

Process Set B

Process	Period T	Computation Time 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 (or 77.5%)
- This is below the threshold for three processes (0.78) and, hence, this process set will meet all its deadlines.

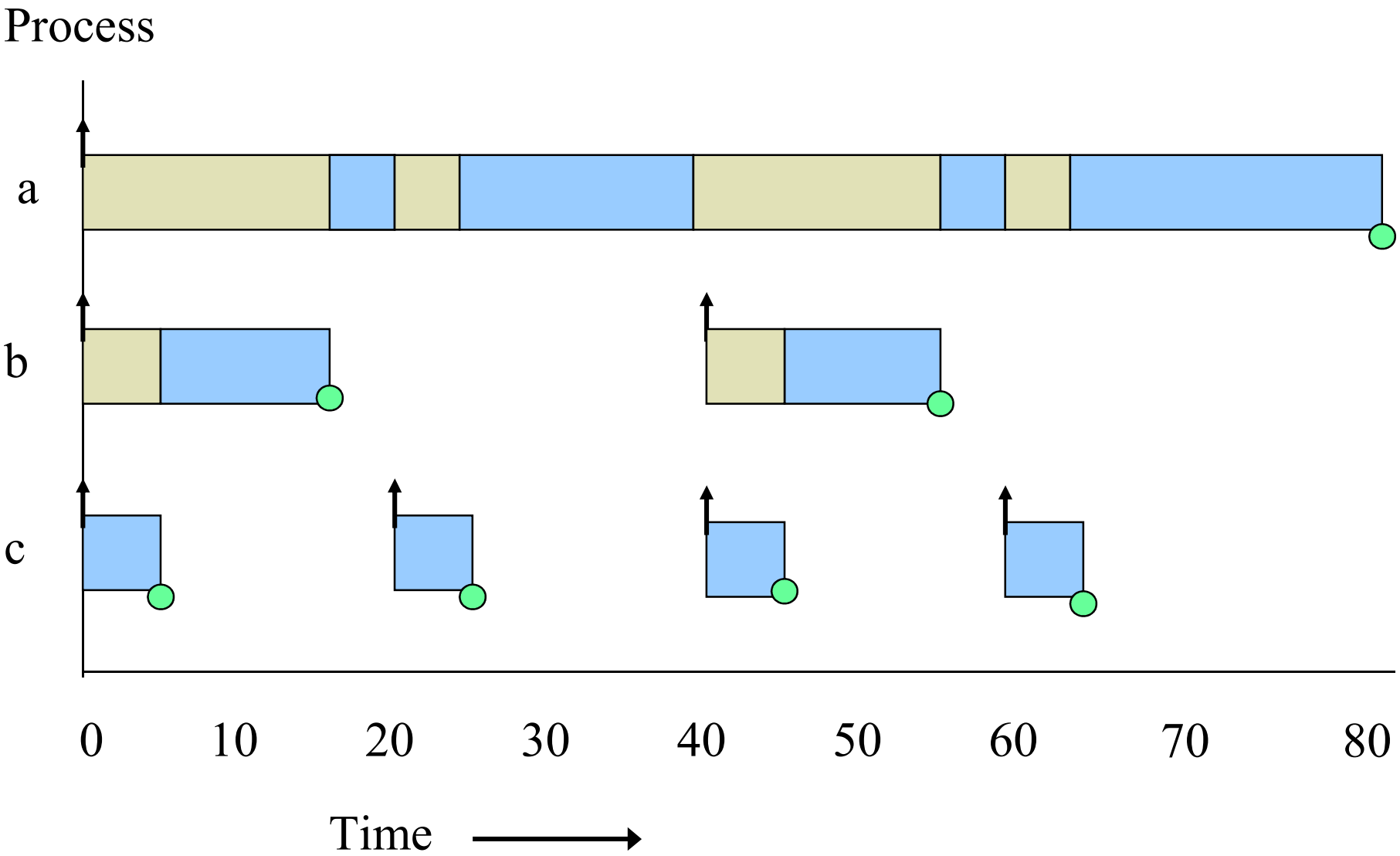
Utilization-Based Analysis

Process Set C

Process	Period T	Computation Time 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

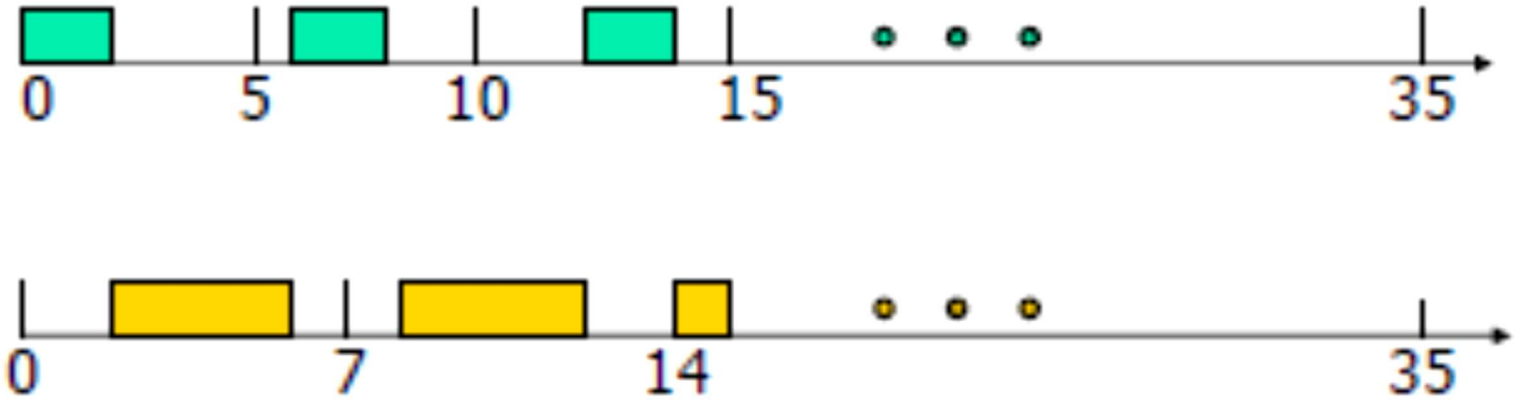


Utilization-based test is neither exact nor general but its O(N)

A Simple Example

Task(C,T)

- Task set: {(2,5),(4,7)}
- U = 2/5 + 4/7= 34/35 ~ 0.97 (schedulable!)



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.
- It is possible 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 (VBS) Scheduling

- If a system can become overloaded then simple static priorities or deadlines are 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 preemptive or non pre-emptive form.

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 that 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:

Response-Time Analysis

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

During *R*, each higher priority task *j* will execute a no. of times.

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

$$\text{Total interference} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Response Time

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

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 than R_i (e.g. 0 or C_i)

Response Time Calculation 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
```

Response Time Calculation Example

Process Set D

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

$$R_a = 3$$

$$w_b^0 = 3$$

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

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

$$R_b = 6$$

Response Time Calculation

Process c

$$w_c^0 = 5$$

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

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

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

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

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

$$R_c =$$

Process Set C

Process	Period, T	Computation Time, 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 utilization 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.

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 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.

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.

WCET Analysis

Need 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 x 100 (+overhead), say 1005.
- But if ***Cond*** only true on 3 occasions then cost is 375

Real-time Scheduling Exercises

Exercise-1:

Consider three processes P, Q and S. P has a period of 100msec in which it requires 30msecs of processing. The corresponding values for Q and S are (6, 1) and (25, 5) respectively. Assume that P is the most important process in the system, followed by Q and then S.

(1) What is the behavior of the scheduler if priority is based on importance?
(2) What is the process utilization of P, Q and S.
(3) How should the process be scheduled so that all deadlines are met.
(4) Illustrate one of the schemes that allows these processes to be scheduled.

Exercise-2:

Add a fourth process R, to the set of processes given in Exercise-1. Failure of this process will not lead to safety being undermined. R has a period of 50ms, but has a processing requirement that is data dependent and varies from 5 to 25 ms. Discuss how this process should be integrated with P, Q and S.

Hard and Soft Real-time Processes

Hard Real-time Process: The deadline must not be missed.
Soft Real-time Process: The application is tolerant of missed deadlines.

- In many situations the WCET (worst-case execution time) figures for sporadic processes are considerably higher than the averages.
- Measuring schedulability with worst-case figures may lead to very low processor utilizations.
- Interrupts often arrive in bursts e.g. an abnormal sensor reading may lead to significant additional computation.

Sporadic Processes

- A Sporadic process is that which has hard real-time applications.
- Sporadic 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/Soft Process Scheduling 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
-
- Rule 2 ensures that no hard 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

- Aperiodic processes have soft real-time jobs.
- They 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 (sporadic)* can be employed.
- Servers protect the processing resources needed by hard processes but otherwise allow soft processes to run as soon as possible.

Process Sets with D < T

- For D = T, Rate Monotonic priority ordering is optimal.
- For D < T, (DMPO) 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	Computation Time, 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 of "DMPO is Optimal" is given in the text

Deadline Scheduling Exercises

Exercise:
Consider a set of 5 aperiodic tasks whose execution profiles are given below. Develop the scheduling diagram of these processes employing EDF and FCFS.

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

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.
- The process is said to suffer ***priority inversion***.
- If a process is waiting for a lower-priority process, the process is said to be ***blocked***.
- Dynamic priorities can vary during execution.

One has to avoid Priority Inversion.

Bounded Priority Inversion

Duration is not longer than that of the critical section where the lower-priority process owns the resource.

Unbounded Priority Inversion

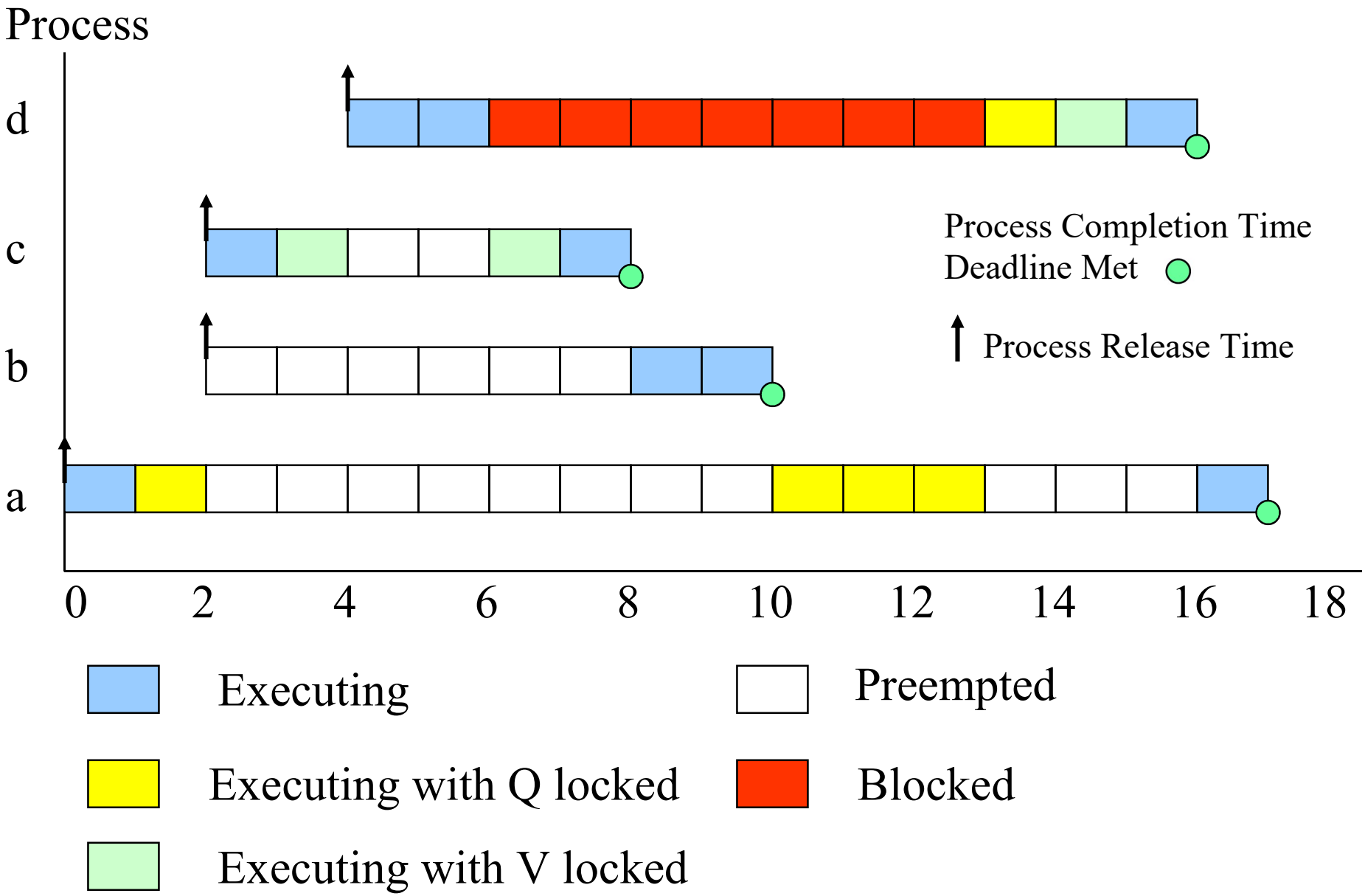
Priority Inversion

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

Example of Priority Inversion

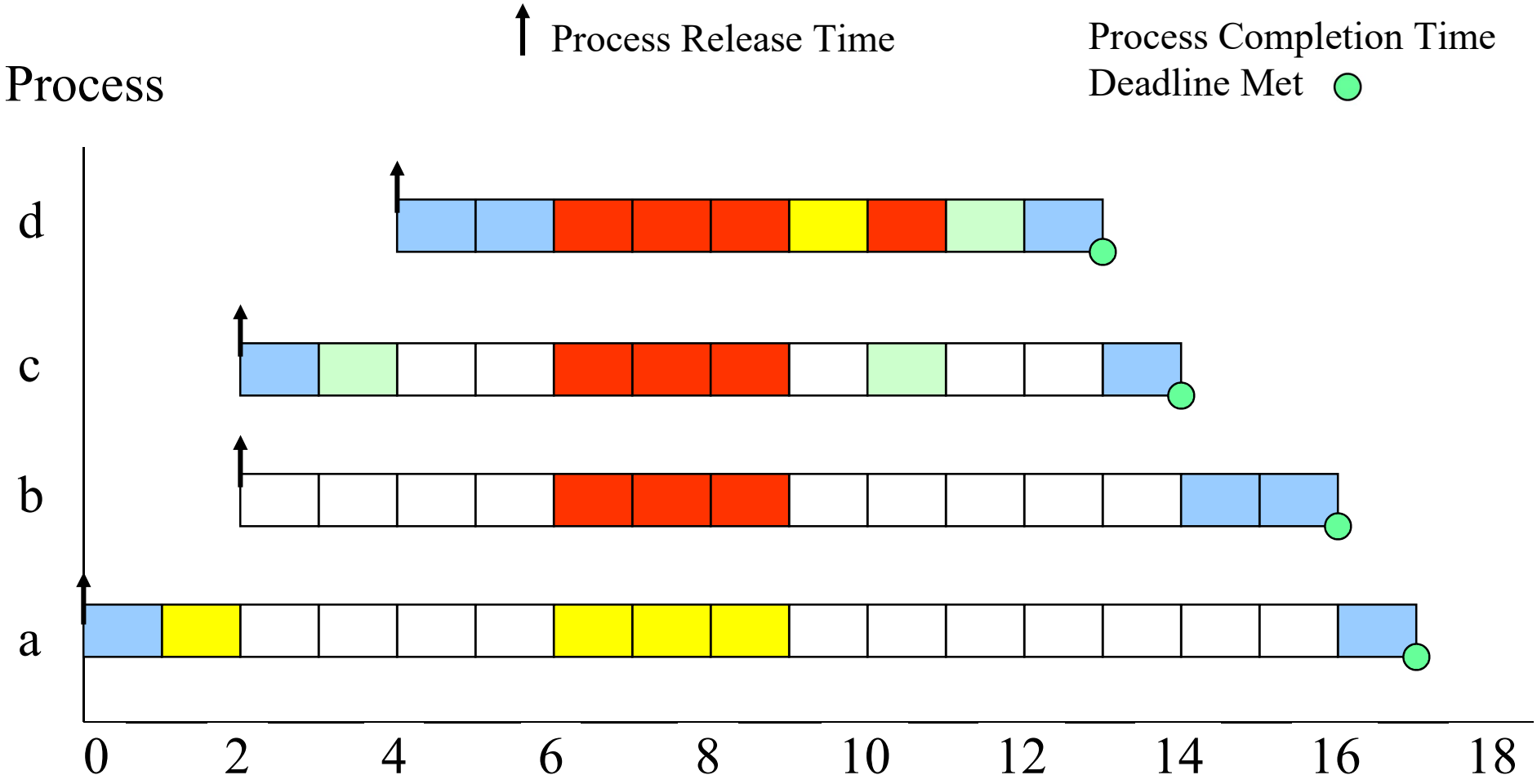
Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Example of Priority Inversion



Priority Inheritance

If process ***a*** is blocking the process ***d***, then it runs with the priority of ***d***.



Calculating Blocking

- If a process has **m** critical sections that can lead to blocking 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:

$$R_i = C_i + B_i + I_i$$

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

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

RTX - Priority Inversions

To prevent priority inversions, RTX employs the **Priority inheritance** technique. For a short time, the lower-priority task runs at a priority of a higher-priority pending task.

The RTX **mutex** objects (Mutual Exclusive Lock objects) employ the Priority inheritance.

Mutex Management Routines

os_mut_init:	Initializes a mutex object.
os_mut_release:	Releases a mutex object.
os_mut_wait:	Waits for a mutex object to become available

- Mutual exclusion locks (mutexes) are an alternative to avoid synchronization and memory access problems.
- Mutexes are software objects that a task can use to lock the common resource. Only the task that locks the mutex can access the common resource.
- The kernel blocks all other tasks that request the mutex until the task that locked the mutex unlocks it.

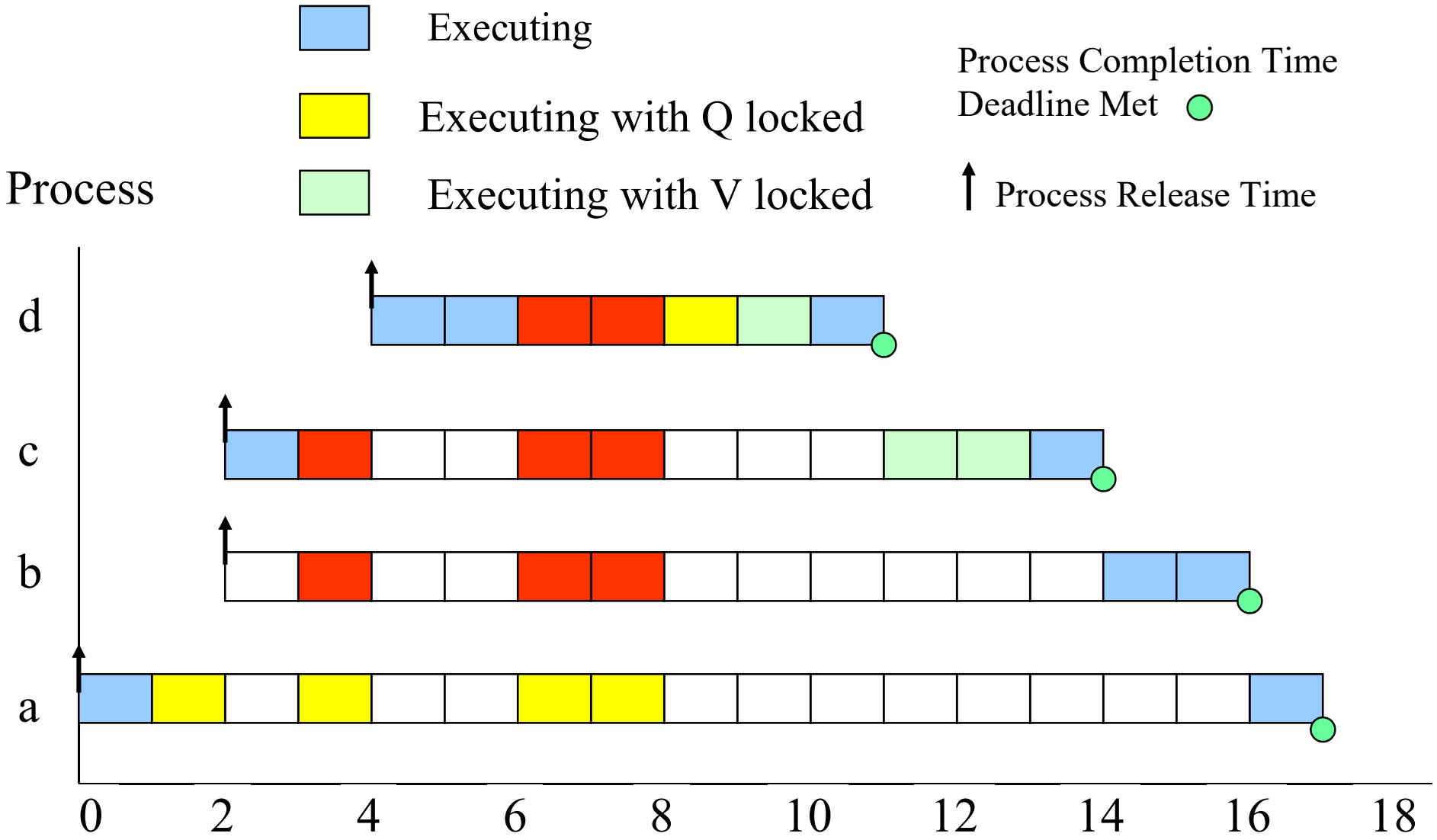
Priority Ceiling Protocols

OCPP: Original ceiling priority protocol
ICPP: Immediate ceiling priority protocol

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 the 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)

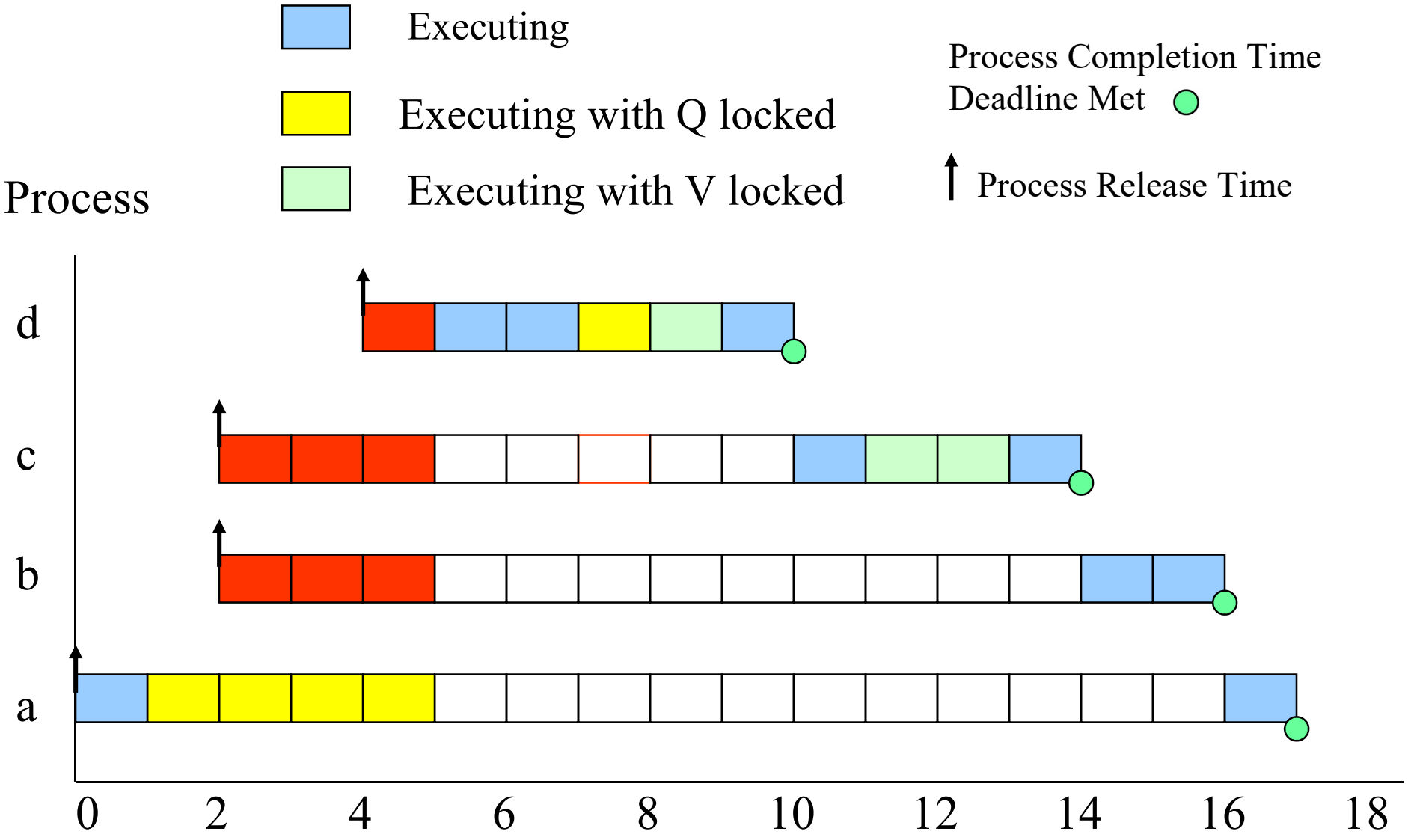
OCPP Inheritance



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

The worst-case behavior of the two ceiling schemes is identical (from a scheduling view point)

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

There are some points of difference:

- ICPP 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.

ICPP is called Priority Protect Protocol in POSIX

Mars Pathfinder Suffered Unbounded Priority Inversion

Low-priority Meteorological Process:

Acquired the (shared) bus.

Medium-priority, Long-running, Communications Process:

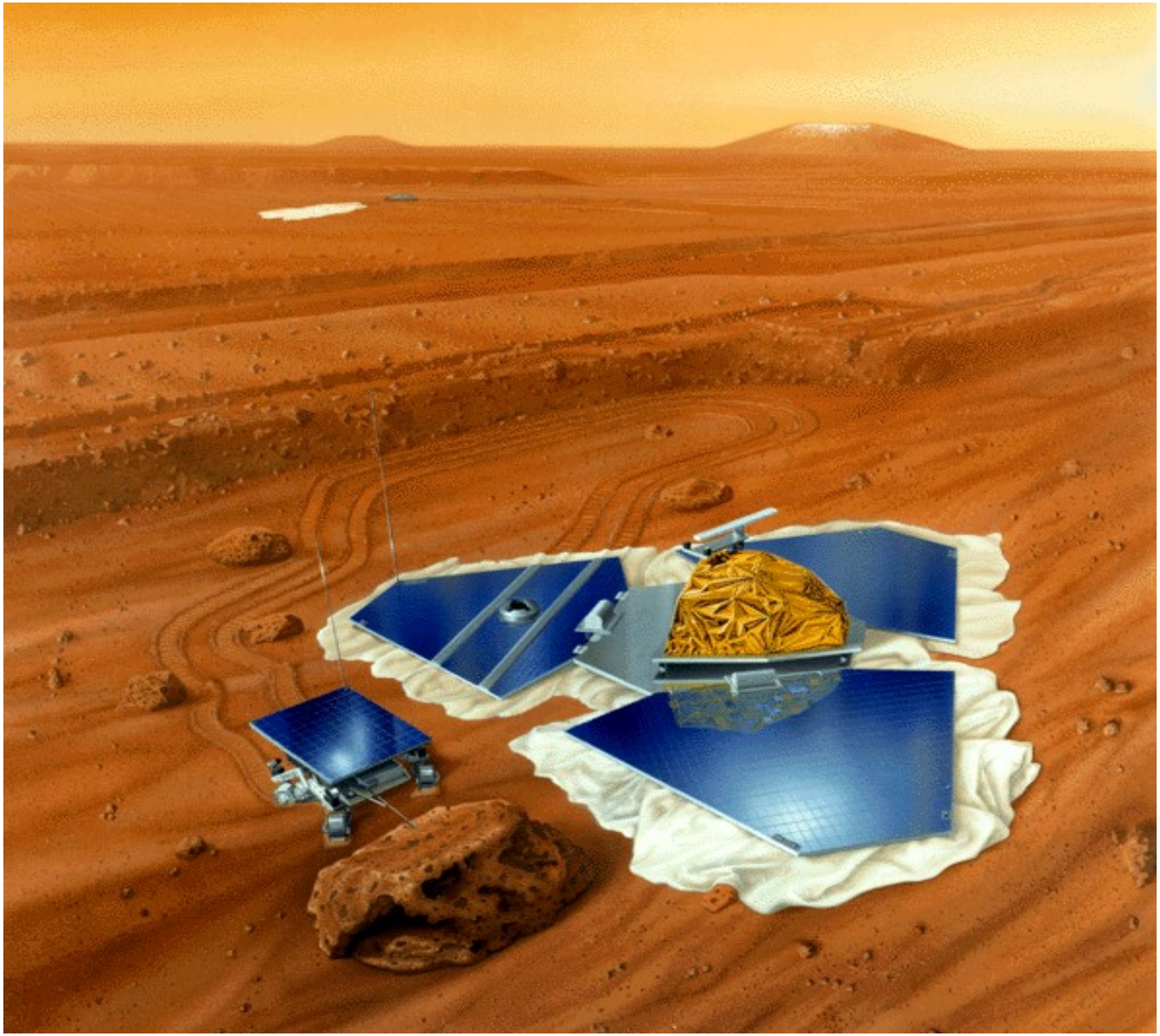
Woke up and preempted the meteorological thread.

High-priority Bus Management Process:

Woke up and was blocked because it couldn't acquire the bus;

When it couldn't meet its deadline it reinitialized the computer via a hardware reset.

Mars Pathfinder



Duration of Unbounded Priority Inversion

Limiting the duration of unbounded priority inversion prevents low-priority process from being preempted by the medium-priority processes during the priority inversion.

Technique: Manipulate process priorities at run-time.

Scheduling: Processes with higher priority are scheduled to run first.

Objective: Assign priorities in such a way that all outputs are computed before their *deadlines*.

- Deadline-Driven Assignment: Assign highest priorities to processes with shortest deadlines.
- Rate Monotonic Assignment: Assign highest priorities to processes that run most frequently without regard to deadlines.

Modified Process Model

Until Now:

- 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 to the Original Model

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

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
-
- If the extra computation time resulting 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

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

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.