3

Processes

Uwe R. Zimmer – International University Bremen

References for this chapter

[Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
Prentice Hall, 1990

[Bollella01]
Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, Mark Turnbull & Rudy Belliardi
The Real-Time Specification for Java
http://www.rtj.org

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Silberschatz01] – Chapter 4,5
Abraham Silberschatz, Peter Bear Galvin, Greg Gagne
Operating System Concepts
John Wiley & Sons, Inc., 2001

[Stallings2001] – Chapter 3,4
William Stallings
Operating Systems
Prentice Hall, 2001

all references and some links are available on the course page

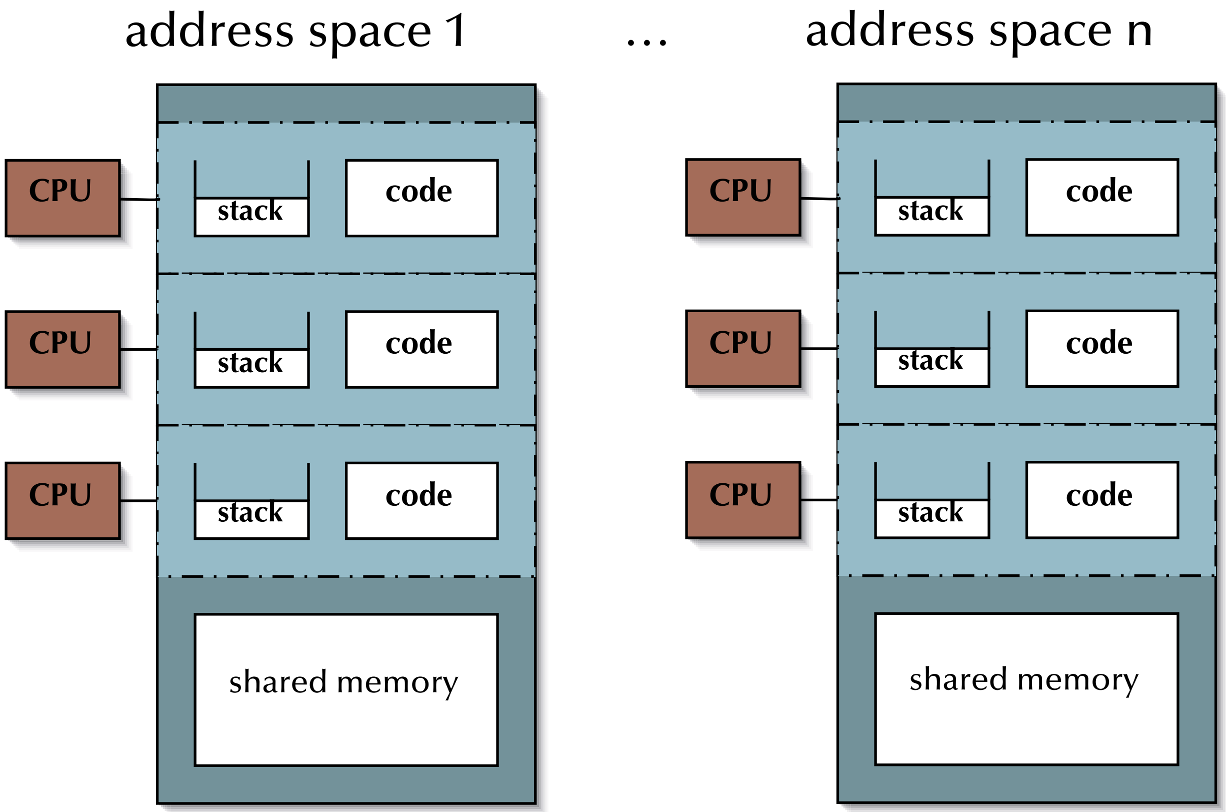
Introduction to processes and threads

1 CPU per control-flow

for specific configurations only:

- distributed μcontrollers
- physical process control systems:
  1 cpu per task,
  connected via a typ. fast bus-system (VME, PCI)

☞ no need for process management



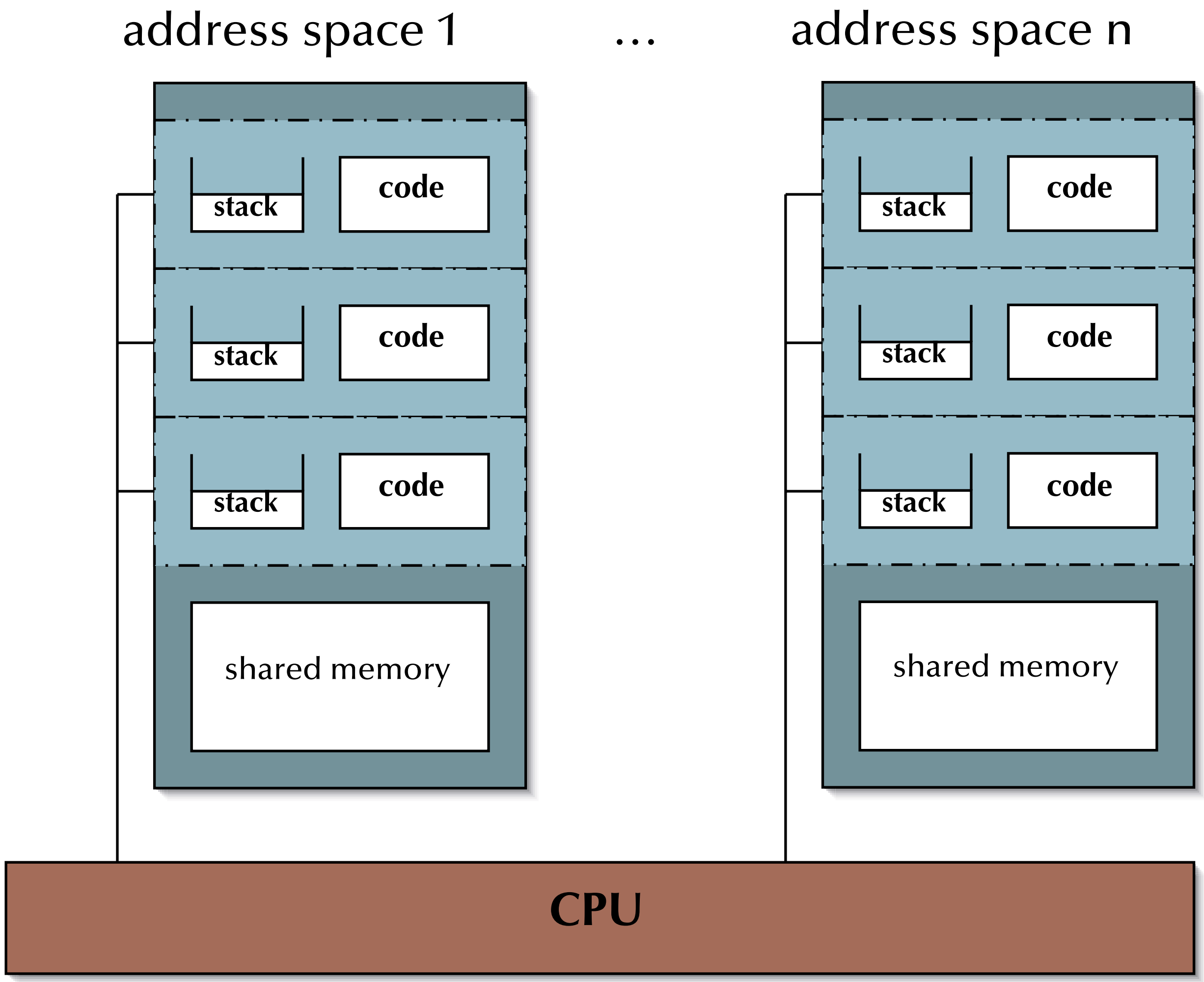
Introduction to processes and threads

1 CPU for all control-flows

- OS: emulate one CPU for every control-flow

☞ **multi-tasking** operating system

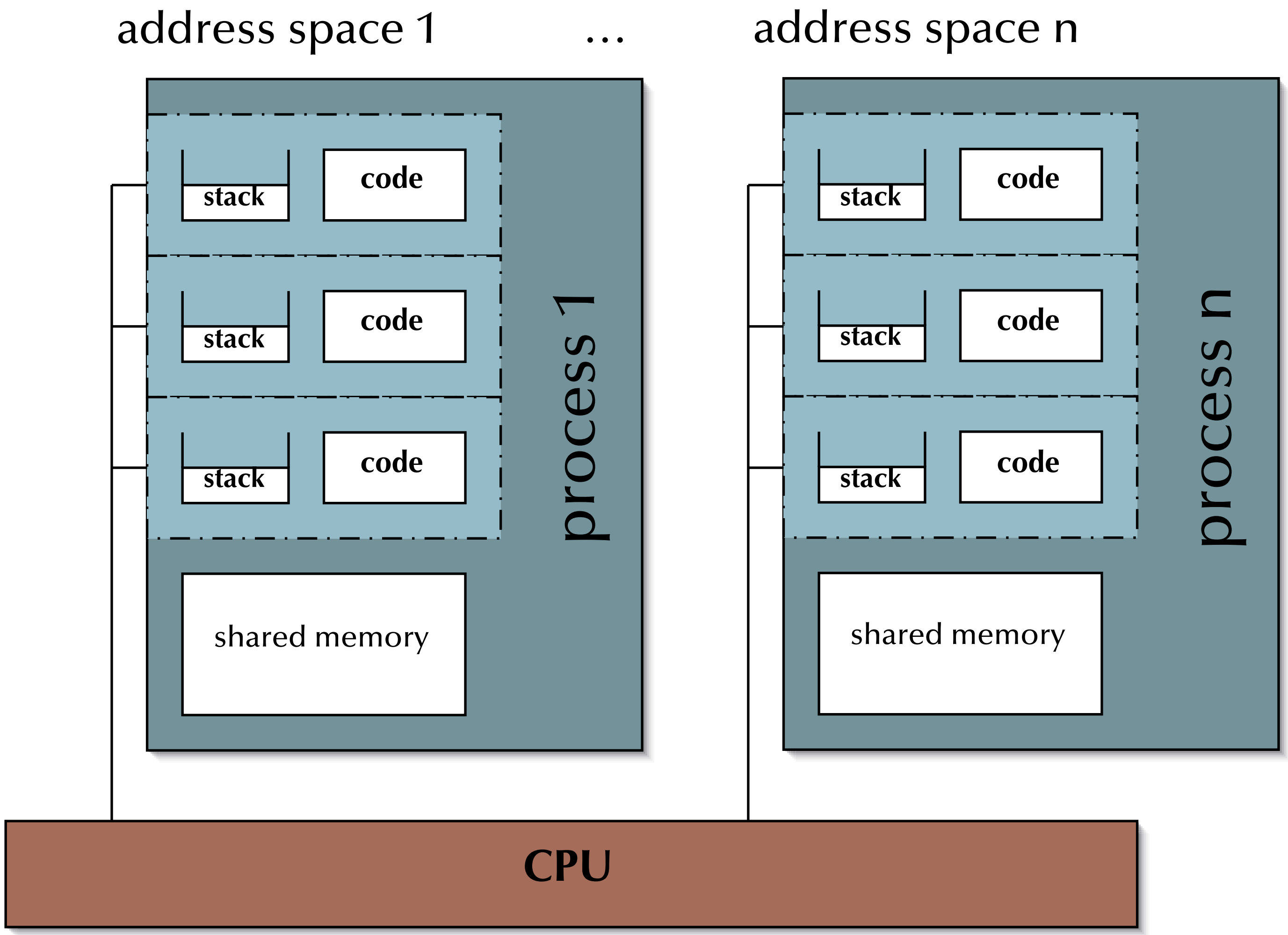
- support for memory protection becomes essential



Introduction to processes and threads

Processes

- **Process** ::=
  address space
  + control flow(s)
- Kernel has full knowledge about all *processes* as well as their *requirements* and current *resources* (see below)

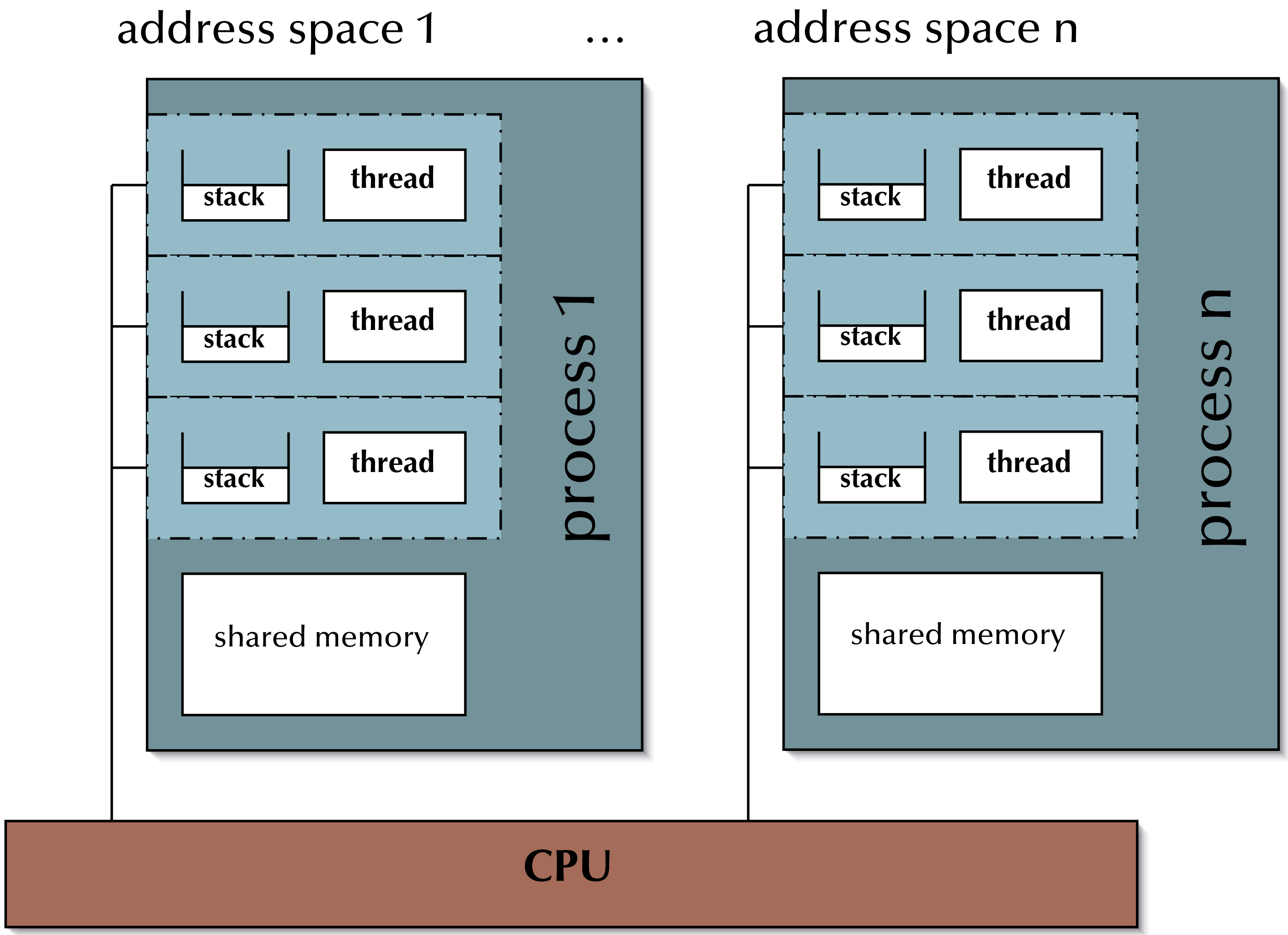


Introduction to processes and threads

Threads

Threads (individual control-flows) can be handled:

- *inside the kernel*:
    - kernel scheduling
    - I/O block-releases according to external signal
- *outside the kernel*:
    - user-level scheduling
    - no signals to threads



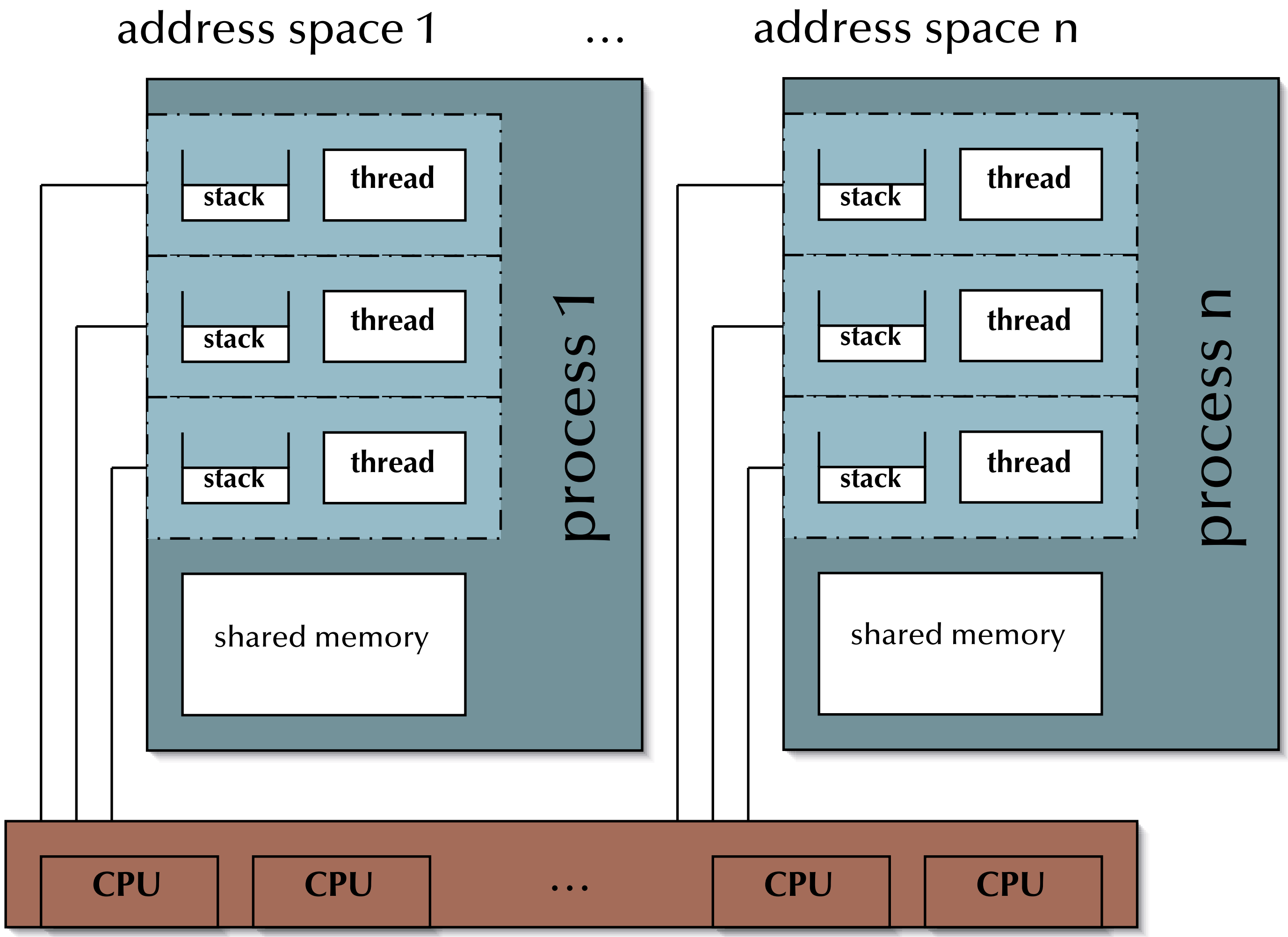
Introduction to processes and threads

Multi-processor-systems

- The kernel may execute multiple processes at a time.

☞ Address space and resource restrictions of individual CPUs and processes/threads need to be considered.

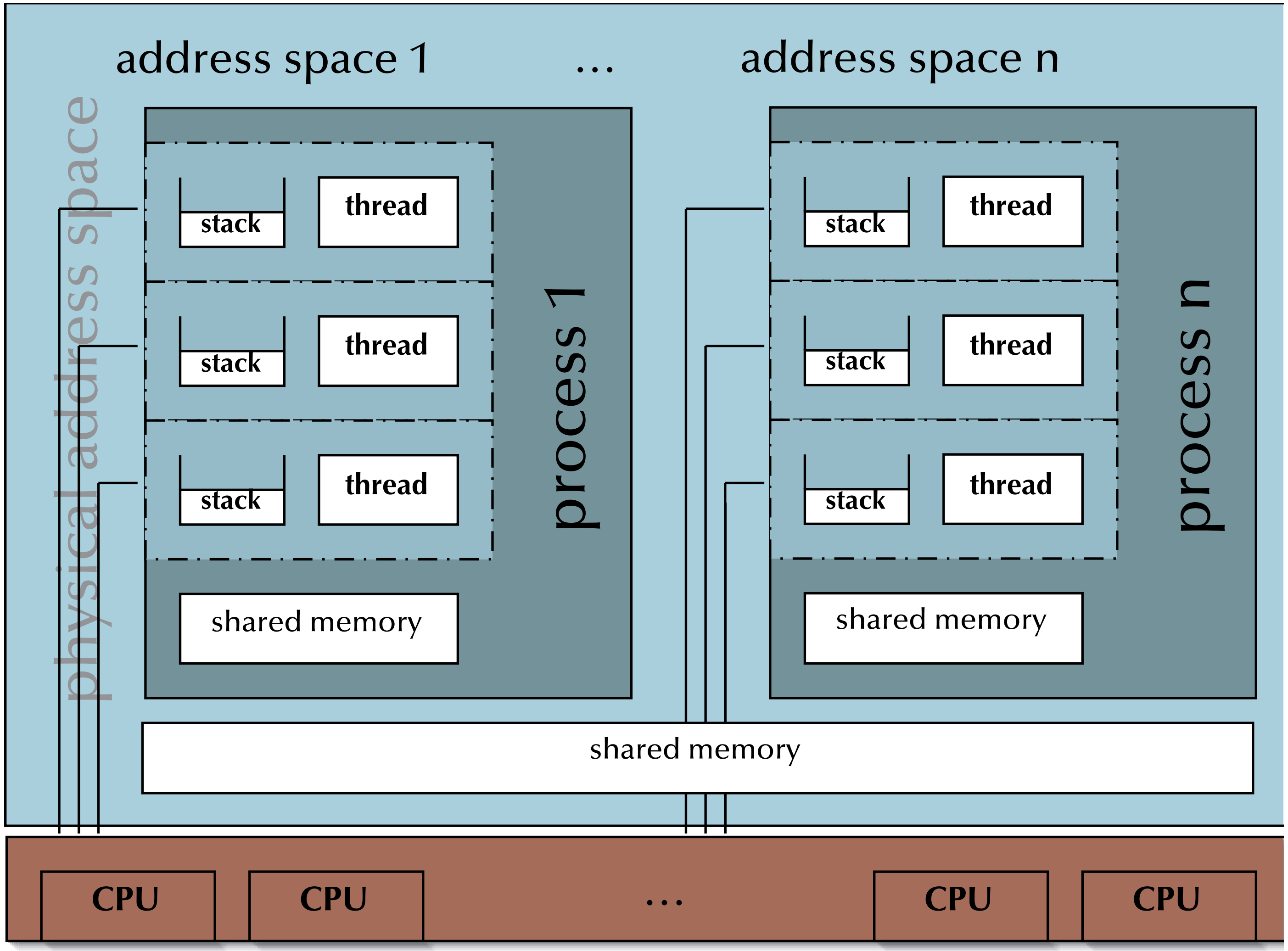
☞ Caching, synchronization, and memory protection need to be adapted.



Symmetric Multiprocessing (SMP)

- all CPUs share the same physical address space (and access to resources)

☞ processes/threads can be executed on any available CPU



Processes ↔ Threads

Also processes can share memory
and the exact interpretation of threads is different in different operating systems:

- ☞ Threads can be regarded as a group of processes, which share some resources (☞ process-hierarchy)
- ☞ Due to the overlap in resources, the attributes attached to threads are less than for ‘first-class-citizen-processes’

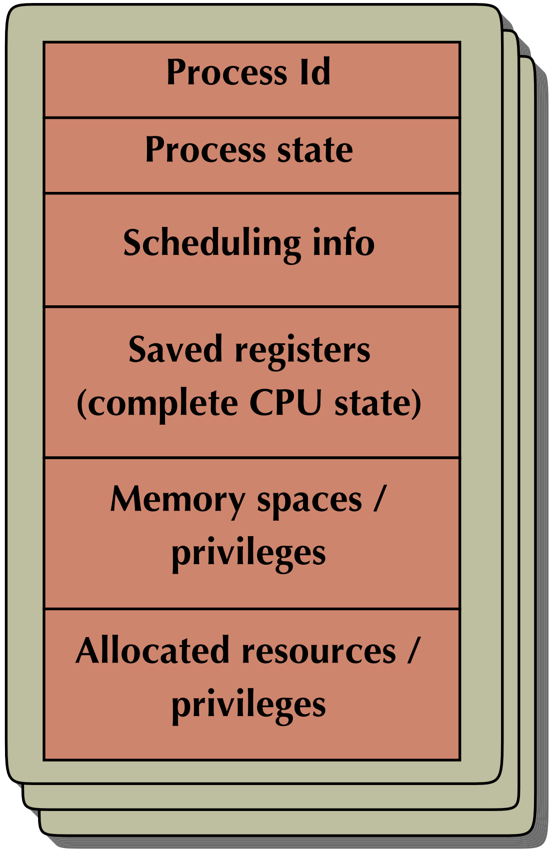
- ☞ Thread switching and inter-thread communications can be more efficient than on full-process-level
- ☞ Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions

Process Control Blocks

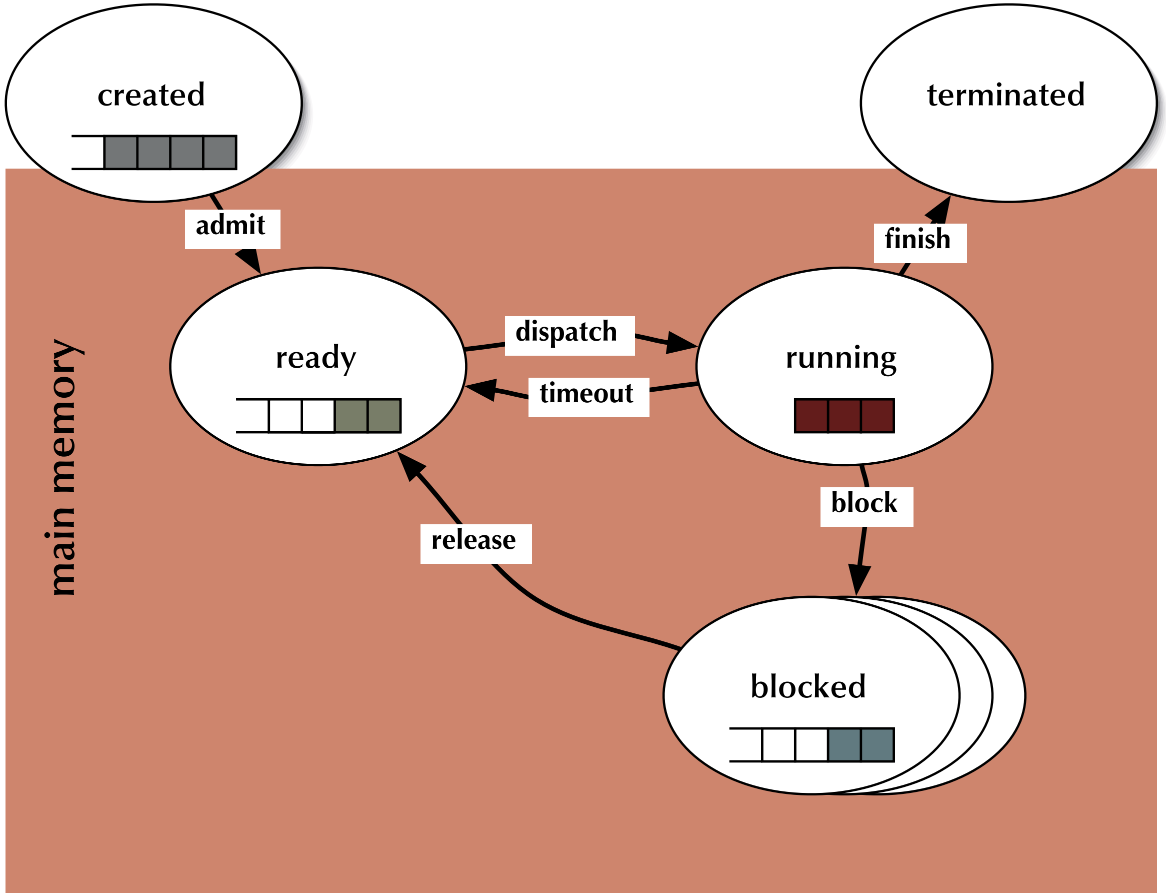
- **Process Id**
- **Process state**:
  {created, ready, executing, blocked, suspended, …}
- **Scheduling info**:
  priorities, deadlines, consumed CPU-time, …
- **CPU state**:
  saved/restored information while context switches
  (incl. the program counter, stack pointer, …)
- **Memory spaces / privileges**:
  memory base, limits, shared areas, …
- **Allocated resources / privileges**:
  open and requested devices and files

… PCBs are usually enqueued at a certain state or condition

Process Control Blocks (PCBs)



Process states



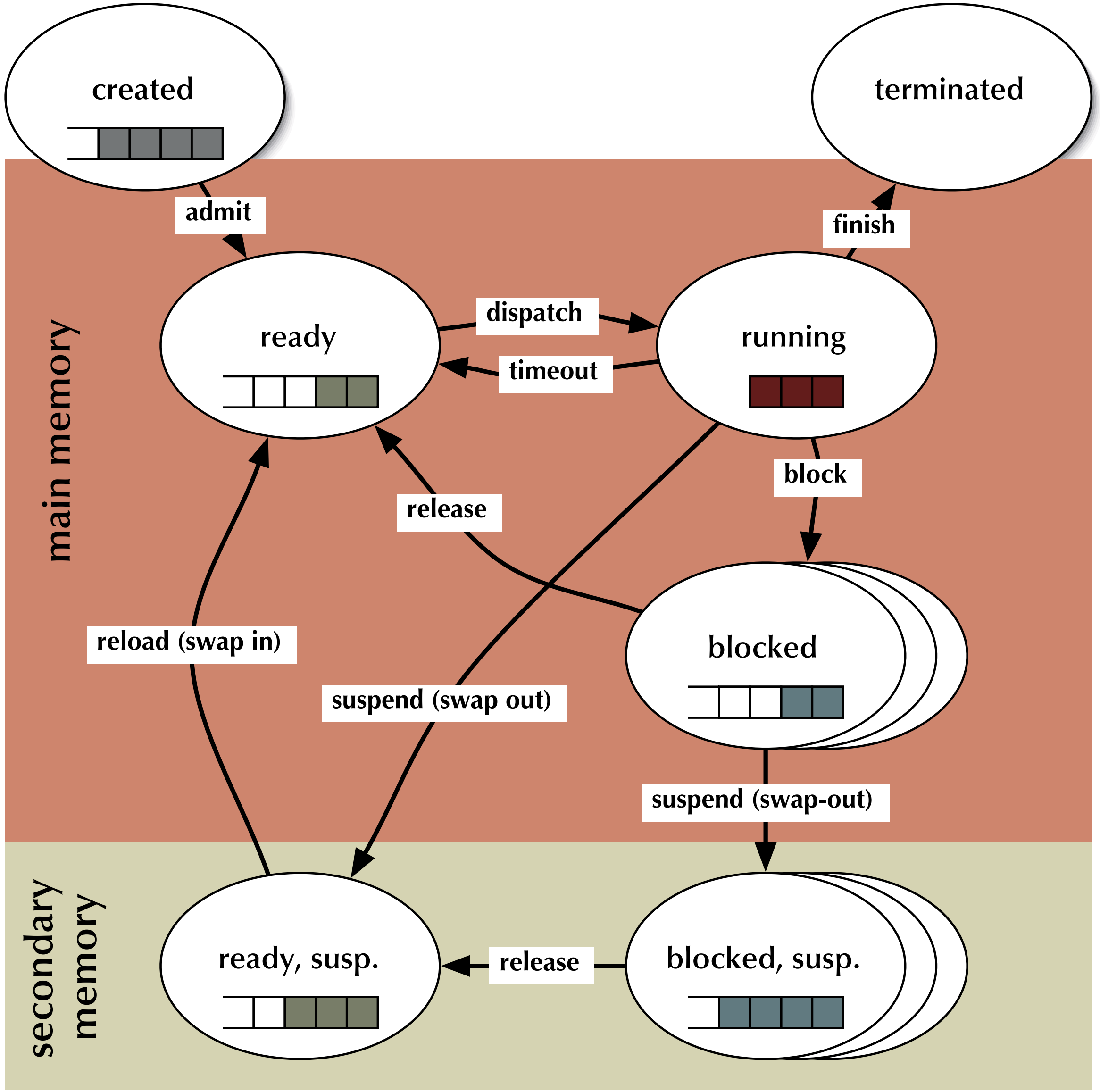
- **created**: the task is ready to run,
but not yet considered by any dispatcher
– waiting for admission

- **ready**: ready to run
– waiting for a free CPU

- **running**: holds a CPU and executes

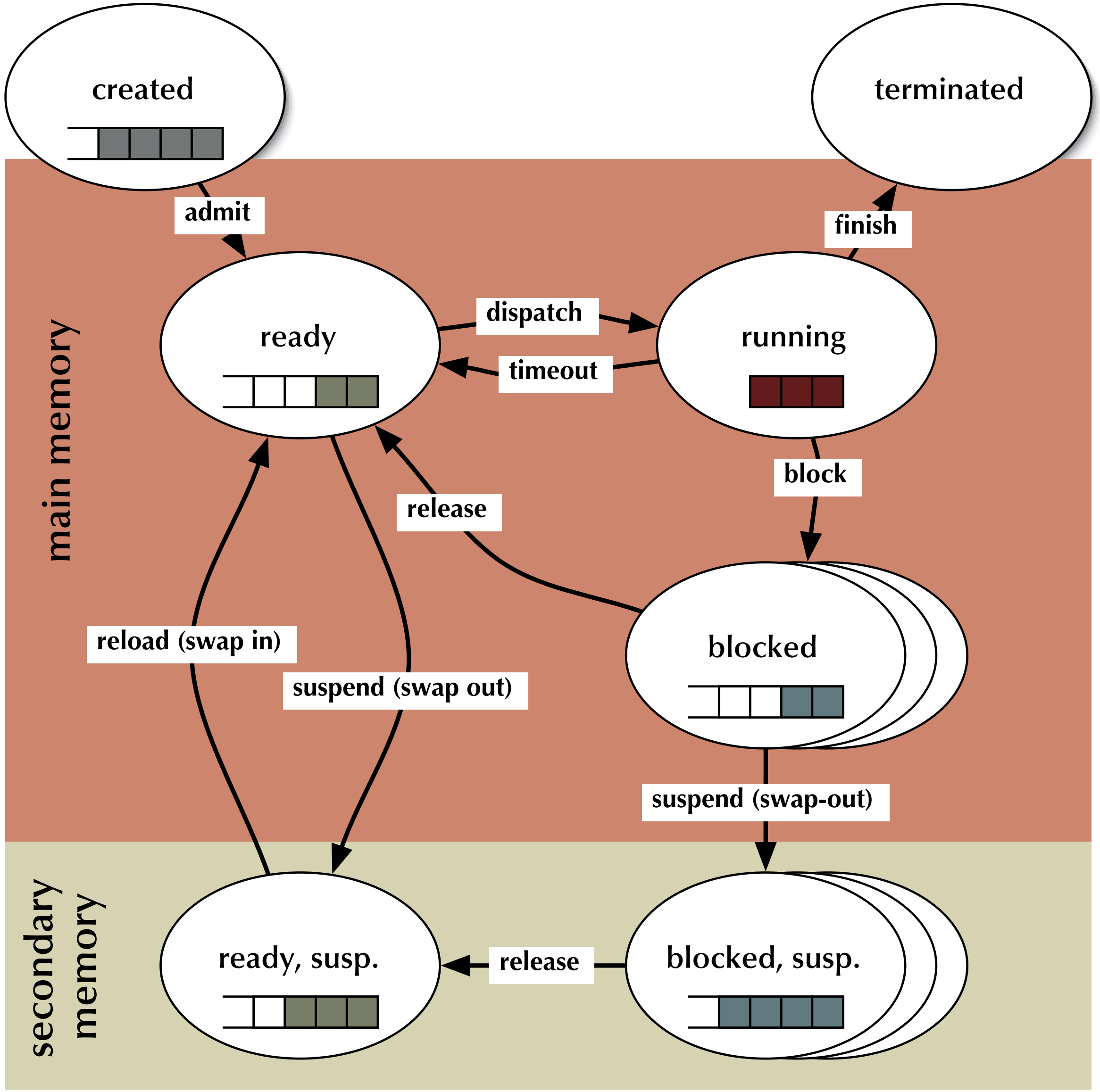
- **blocked**: not ready to run
– waiting for a a resource to become available

Process states



- **created**: the task is ready to run, but not yet considered by any dispatcher
  – waiting for admission
- **ready**: ready to run
  – waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  – waiting for a resource
- **suspended states**: swapped out of main memory (not time critical processes)
  – waiting for main memory space (and other resources)

Process states



- **created**: the task is ready to run, but not yet considered by any dispatcher – waiting for admission

- **ready**: ready to run – waiting for a free CPU

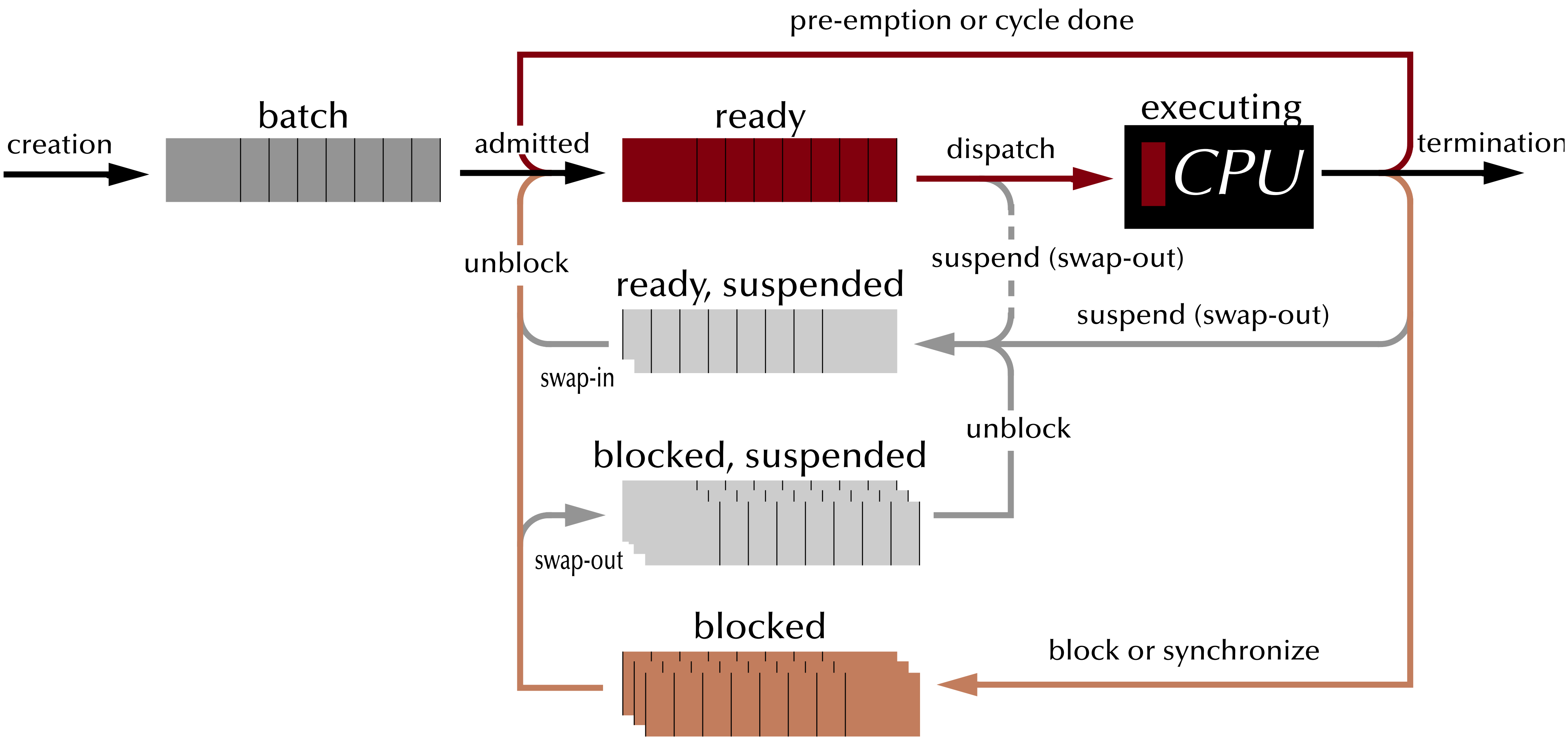
- **running**: holds a CPU and executes

- **blocked**: not ready to run – waiting for a resource

- **suspended states**: swapped out of main memory (not time critical processes) – waiting for main memory space (and other resources)

☞ dispatching and suspending can be independent modules here

Process states



Synchronization methods

• Shared memory based synchronization

• Semaphores	☞ ‘C’, POSIX — Dijkstra
• Conditional critical regions	☞ Edison (experimental)
• Monitors	☞ Modula-1, Mesa — Dijkstra, Hoare, …
• Mutexes & conditional variables	☞ POSIX
• Synchronized methods	☞ Real-time Java
• Protected objects	☞ Ada95

• Message based synchronization

• Asynchronous messages	☞ e.g. POSIX, …
• Synchronous messages	☞ e.g. Ada95, CHILL, Occam2
• Remote invocation, remote procedure call	☞ e.g. Ada95, …
• Synchronization in distributed systems	☞ e.g. CORBA, …

Synchronization in operating systems

☞ There are many concurrent entities in operating systems:

- Interrupt handlers
- Processes
- Dispatchers
- Timers
- …

… and … operating systems need to be expandible or very robust …

Thus *all* data is declared …

☞ … ***either*** local (and protected by language-, or hardware-mechanisms)

☞ … ***or*** it is ‘out in the open’ and all access need to be synchronized!

The need for synchronization

Synchronization: the run-time overhead?

☞ Is the potential overhead justified for simple data-structures:

```
                    int i;
                     ......
i++;  {in one thread}   |   i=0; {in another thread}
```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?

The need for synchronization

```
                         int i;
                         ......
i++;  {in one thread}       |      i=0; {in another thread}
```

- Depending on the hardware and the compiler, it might be atomic, it might be not:

☞ Handling a 64-bit integer on a 8- or 16-bit controller *will not be atomic*
… but perhaps it is an 8-bit integer.

☞ Any manipulations on the main memory *will not be atomic*
… but perhaps it is a register.

☞ Broken down to a load-operate-store cycle, the operations *will not be atomic*
… but perhaps the processor supplies atomic operations for the actual case.

☞ Assuming that all ‘perhapses’ are applying: how to expand this code?

The need for synchronization

```
int i;
......
i++;  {in one thread}     |     i=0; {in another thread}
```

☞ Unfortunately: the chances that such programming errors turn out are usually small and some implicit by chance synchronization in the rest of the system might prevent them at all.

- Many effects stemming from asynchronous memory accesses are interpreted as (hardware) ‘glitches’, since they are rare and effect usually only some parts of the data.
- On assembler level: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.

In anything higher than assembler level on small, predictable μcontrollers:

☞ Measures for synchronization are required!

Some synchronization terms:

- ***Condition synchronization:***
  synchronize a task with an event given by another task.
- ***Critical sections:***
  code fragments which contain access to shared resources and need to be executed without interference with other critical sections, sharing the same resources.
- ***Mutual exclusion:***
  protection against asynchronous access to critical sections.
- ***Atomic operations:***
  the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware).

☞ there must be a set of atomic operations to start with!

Synchronization by flags

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:

Task 1: `x := 0;` | Task 2: `x := 5;`

will result in **either** `x = 0` **xor** `x = 5` — and no other value is ever observable.

Synchronization by flags

Assuming further that there is a shared memory area between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.

Condition synchronization by flags

```
var Flag : boolean := false;
```

```
process P1;
   statement X;

   repeat until Flag;

   statement Y;
end P1;
```

```
process P2;
   statement A;

   Flag := true;

   statement B;
end P2;
```

Sequence of operations: [A | X] ➠ [B | Y]

Synchronization by flags

Assuming further that there is a shared memory between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions:

Memory flag method is ok for simple condition synchronization, but …

☞ … is not sufficient for general mutual exclusion in critical sections!

☞ … busy-waiting is required to poll the synchronization condition!

☞ More powerful synchronization operations are required for critical sections

Synchronization by semaphores

(Dijkstra 1968)

Assuming further that there is a shared memory between two processes:

- a set of processes agree on a variable `S` operating
  as a flag to indicate synchronization conditions … *and* …
- an atomic operation `P` on `S` — `P` stands for 'passeren' (Dutch for 'pass'):
    - `P: [if S > 0 then S := S - 1]` also: '`Wait`', '`Suspend_Until_True`'
- an atomic operation `V` on `S` — `V` stands for 'vrygeven' (Dutch for 'to release'):
    - `V: [S := S + 1]` also: '`Signal`', '`Set_True`'

☞ the variable `S` is then called a **semaphore**.

OS-level: `P` is usually also suspending the current task until `S > 0`.
CPU-level: `P` indicates whether it was successful, but the operation is not blocking.

Condition synchronization by semaphores

```
                    var sync : semaphore := 0;

process P1;                          process P2;
   statement X;                         statement A;

   wait (sync);                         signal (sync);

   statement Y;                         statement B;
end P1;                              end P2;
```

Sequence of operations: [A | X] ➠ [B | Y]

Mutual exclusion by semaphores

```
var mutex : semaphore := 1;
```

```
process P1;                         process P2;
   statement X;                        statement A;

   wait (mutex);                       wait (mutex);
      statement Y;                        statement B;
   signal (mutex);                     signal (mutex);

   statement Z;                        statement C;
end P1;                             end P2;
```

Sequence of operations: [A | X] ➠ [B ➠ Y **xor** Y ➠ B] ➠ [C | Z]

Semaphores

Types of semaphores:

- **General semaphores (counting semaphores)**: non-negative number; (range limited by the system) `P` and `V` increment and decrement the semaphore by one.
- **Binary semaphore**s: restricted to [0, 1]; Multiple `V` (`Signal`) calls have the same effect than 1 call.
    - binary semaphores are sufficient to create all other semaphore forms.
    - atomic 'test-and-set' operations at hardware level are usually binary semaphores.
- **Quantity semaphores**: The increment (and decrement) value for the semaphore is specified as a parameter with `P` and `V`.

Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;

   procedure Set_True  (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);

   function Current_State (S : Suspension_Object) return Boolean;

   procedure Suspend_Until_True (S : in out Suspension_Object);

private
   … -- not specified by the language
end Ada.Synchronous_Task_Control;
```

Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;

   procedure Set_True  (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);

   function Current_State (S : Suspension_Object) return Boolean;

   procedure Suspend_Until_True (S : in out Suspension_Object);

private
   … -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! ('strict version of a binary semaphore') (`Program_Error` will be raised with the second task trying to suspend itself)

☞ no queues! ☞ minimal run-time overhead

Semaphores in Ada95

for very special cases only, in general:

medieval

```
package Ada.Synchronous_Task_Control is
    type Suspension_Object is limited private;

    procedure Set_True  (S : in out Suspension_Object);
    procedure Set_False (S : in out Suspension_Object);

    function Current_State (S : Suspension_Object) return Boolean;

    procedure Suspend_Until_True (S : in out Suspension_Object);

private
    … -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! (strict version of a binary semaphore) (`Program_Error` will be raised with the second task trying to suspend itself)

☞ no queues ☞ minimal run-time overhead

Semaphores in POSIX

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy   (sem_t *sem_location);

int sem_wait      (sem_t *sem_location);
int sem_trywait   (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);

int sem_post      (sem_t *sem_location);

int sem_getvalue  (sem_t *sem_location, int *value);
```

Semaphores in POSIX

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy   (sem_t *sem_location);

int sem_wait      (sem_t *sem_location);
int sem_trywait   (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);

int sem_post      (sem_t *sem_location);

int sem_getvalue  (sem_t *sem_location, int *value);
```

generate semaphore for usage between processes
(otherwise for threads of the same process only)

Semaphores in POSIX

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy   (sem_t *sem_location);

int sem_wait      (sem_t *sem_location);
int sem_trywait   (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);

int sem_post      (sem_t *sem_location);

int sem_getvalue  (sem_t *sem_location, int *value);
```

delivers the number of waiting processes as a negative integer,
if there are processes waiting on this semaphore

Semaphores in POSIX

```
void allocate (priority_t P)
{
   sem_wait (&mutex);
   if (busy) {
      sem_post (&mutex);
      sem_wait (&cond[P]);
   }
   busy = 1;
   sem_post (&mutex);
}
```

```
sem_t mutex, cond[2];
typedef emun {low, high} priority_t;
int waiting
int busy
```

```
void deallocate (priority_t P)
{
   sem_wait (&mutex);
   busy = 0;
   sem_getvalue (&cond[high],
                 &waiting);
   if (waiting < 0) {
      sem_post (&cond[high]);
   }
   else {
      sem_getvalue (&cond[low],
                    &waiting);
      if (waiting < 0) {
         sem_post (&cond[low]);
      }
      else {
         sem_post (&mutex);
}  }  }
```

Synchronization

Deadlock by semaphores

```
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension_Object;
```

```
task B;
task body B is
begin
   …
   Suspend_Until_True (Y);
   Suspend_Until_True (X);
   …
end B;
```

```
task A;
task body A is
begin
   …
   Suspend_Until_True (X);
   Suspend_Until_True (Y);
   …
end A;
```

- ☞ could raise a `Program_Error` in Ada95.
- ☞ produces a potential **deadlock** when implemented with general semaphores.

☞ Deadlocks can be generated by all kinds of synchronization methods.

Criticism of semaphores

- Semaphores are not bound to any resource or method or region
  - ☞ Adding or deleting a single semaphore operation some place might stall the whole system
- Semaphores are scattered all over the code
  - ☞ hard to read, error-prone

☞ Semaphores are considered not adequate for complex systems.

(all concurrent and real-time languages offer more abstract and safer synchronization methods).

Conditional critical regions

Basic idea:

- Critical regions are *a set of code sections in different processes*, which are guaranteed to be ***executed in mutual exclusion***:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.
- ***Condition synchronisation*** is provided by *guards*:
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion). If the guard evaluates false, the process is suspended / delayed.
- As with semaphores, no access order can be assumed.

Conditional critical regions

```
buffer : buffer_t;
resource critial_buffer_region : buffer;
```

```
process producer;

   loop

      region critial_buffer_region
         when buffer.size < N do

            -- place in buffer etc.

      end region

   end loop;
end producer
```

```
process consumer;

   loop

      region critial_buffer_region
         when buffer.size > 0 do

            -- take from buffer etc.

      end region

   end loop;
end consumer
```

Criticism of conditional critical regions

- All guards need to be re-evaluated, when any conditional critical region is left:

  ☞ all involved processes are activated to test their guards
  ☞ there is no order in the re-evaluation phase ☞ potential livelocks

- As with semaphores the conditional critical regions are scattered all over the code.

  ☞ on a larger scale: same problems as with semaphores.

The language Edison uses conditional critical regions
for synchronization in a multiprocessor environment
(each process is associated with exactly one processor).

Monitors

(Modula-1, Mesa — Dijkstra, Hoare)

Basic idea:

- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- Formulate all operations as procedures or functions.
- Prohibit access to data-structures, other than by the monitor-procedures.
- Assure mutual exclusion of the monitor-procedures.

Monitors

```
monitor buffer;
   export append, take;
   var (* declare protected vars *)
   procedure append (I : integer);
      …
   procedure take (var I : integer);
      …
begin
   (* initialisation *)
end;
```

How to realize conditional synchronization?

Monitors with condition synchronization

(Hoare)

Hoare-monitors:

- Condition variables are implemented by semaphores (`Wait` and `Signal`).
- Queues for tasks suspended on condition variables are realized.
- A suspended task releases its lock on the monitor, enabling another task to enter.

☞ More efficient evaluation of the guards:
the task leaving the monitor can evaluate all guards and the right tasks can be activated.

☞ Blocked tasks may be ordered and livelocks prevented.

Monitors with condition synchronization

```
monitor buffer;
   export append, take;
   var BUF                              : array [ … ] of integer;
   top, base                            : 0..size-1;
   NumberInBuffer                       : integer;
   spaceavailable, itemavailable : condition;

   procedure append (I : integer);
      begin
         if NumberInBuffer = size then

            wait (spaceavailable);

         end if;
         BUF[top] := I; NumberInBuffer := NumberInBuffer+1;
         top := (top+1) mod size;

         signal (itemavailable)

      end append;   …
```

Monitors with condition synchronization

```
…
   procedure take (var I : integer);
      begin
         if NumberInBuffer = 0 then
            wait (itemavailable);
         end if;
         I := BUF[base];
         base := (base+1) mod size;
         NumberInBuffer := NumberInBuffer-1;
         signal (spaceavailable);
      end take;
begin (* initialisation *)
   NumberInBuffer := 0;
   top := 0; base := 0
end;
```

The signalling and the waiting process are both active in the monitor!

Monitors with condition synchronization

Suggestions to overcome the multiple-tasks-in-monitor-problem:

- A `signal` is allowed **only as the last action** of a process before it leaves the monitor.
- A `signal` operation has the side-effect of **executing a `return`** statement.
- Hoare, Modula-1, POSIX: a `signal` operation which unblocks another process has the side-effect of **blocking the current process**; this process will only execute again once the monitor is unlocked again.
- A `signal` operation which unblocks a process does not block the caller, but the unblocked process must **gain access to the monitor again**.

Monitors in Modula-1

- `wait (s, r)`:
  delays the caller until condition variable `s` is true (`r` is the rank (or 'priority') of the caller).
- `send (s)`:
  If a process is waiting for the condition variable `s`,
  then the process at the top of the queue of the highest filled rank is activated
  (and the caller suspended).
- `awaited (s)`:
  check for waiting processes on `s`.

Monitors in Modula-1

```
INTERFACE MODULE resource_control;
   DEFINE allocate, deallocate;
   VAR busy : BOOLEAN; free : SIGNAL;
   PROCEDURE allocate;
   BEGIN
      IF busy THEN WAIT (free) END;
      busy := TRUE;
   END;
   PROCEDURE deallocate;
   BEGIN
      busy := FALSE;
      SEND (free); -- or: IF AWAITED (free) THEN SEND (free);
   END;
BEGIN
   busy := false;
END.
```

Monitors in ‘C’ / POSIX

(types and creation)

Synchronization between POSIX-threads:

```
typedef … pthread_mutex_t;
typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;
typedef … pthread_condattr_t;

int pthread_mutex_init       (      pthread_mutex_t     *mutex,
                              const pthread_mutexattr_t *attr);
int pthread_mutex_destroy    (      pthread_mutex_t     *mutex);

int pthread_cond_init        (      pthread_cond_t      *cond,
                              const pthread_condattr_t  *attr);
int pthread_cond_destroy     (      pthread_cond_t      *cond);

…
```

Monitors in ‘C’ / POSIX

(types and creation)

Synchronization between POSIX-threads:

```
typedef … pthread_mutex_t;
typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;
typedef … pthread_condattr_t;

int pthread_mutex_init      (
                              const

int pthread_mutex_destroy   (

int pthread_cond_init       (
                              const

int pthread_cond_destroy    (

…
```

Attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- sharing of mutexes and condition variables between processes
- priority ceiling
- clock used for timeouts
- … … …

Monitors in ‘C’ / POSIX

(types and creation)

Synchronization between POSIX-threads:

```
typedef … pthread_mutex_t;
typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;
typedef … pthread_condattr_t;

int pthread_mutex_init      (       pthread_mutex_t       *mutex,

int pthread_mutex_destroy   (                                     );

int pthread_cond_init       (

int pthread_cond_destroy    (
…
```

Undefined, if locked

Undefined, if threads are waiting

Monitors in ‘C’ / POSIX

(operators)

```
…
int pthread_mutex_lock      (       pthread_mutex_t      *mutex);
int pthread_mutex_trylock   (       pthread_mutex_t      *mutex);
int pthread_mutex_timedlock (       pthread_mutex_t      *mutex,
                             const  struct timespec      *abstime);
int pthread_mutex_unlock    (       pthread_mutex_t      *mutex);

int pthread_cond_wait       (       pthread_cond_t       *cond,
                                    pthread_mutex_t      *mutex);
int pthread_cond_timedwait  (       pthread_cond_t       *cond,
                                    pthread_mutex_t      *mutex,
                             const  struct timespec      *abstime);

int pthread_cond_signal     (       pthread_cond_t       *cond);
int pthread_cond_broadcast  (       pthread_cond_t       *cond);
```

Monitors in ‘C’ / POSIX

(operators)

```
...

int pthread_mutex_lock      (       pthread_mutex_t      *mutex);
int pthread_mutex_trylock   (       pthread_mutex_t      *mutex);
int pthread_mutex_timedlock (       pthread_mutex_t      *mutex,
                             const struct timespec       *abstime);
int pthread_mutex_unlock    (       pthread_mutex_t      *mutex);

int pthread_cond_wait       (       p
                                    p
int pthread_cond_timedwait  (       pthread_cond_t       *cond,
                                    p
                             const s

int pthread_cond_signal     (       pthread_cond_t       *cond);
int pthread_cond_broadcast  (       pthread_cond_t       *cond);
```

unblocking ‘at least one’ thread

unblocking all threads

Monitors in ‘C’ / POSIX

(operators)

…

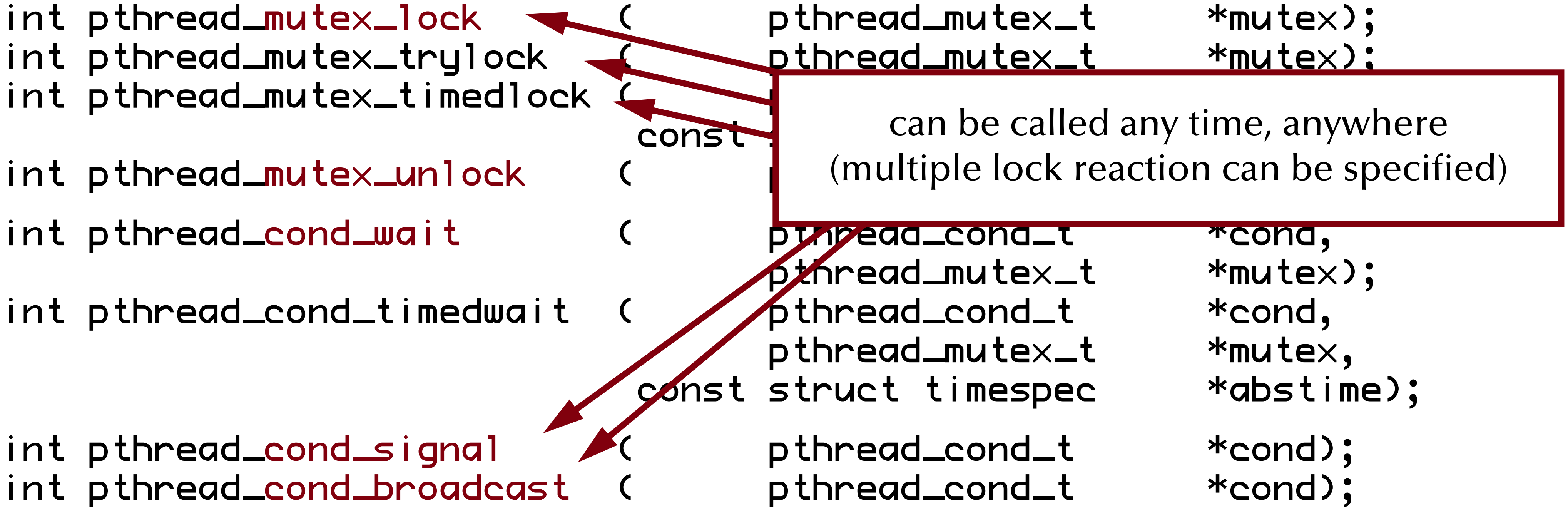
```
int pthread_mutex_lock      (      pthread_mutex_t      *mutex);
int pthread_mutex_trylock   (      pthread_mutex_t      *mutex);
int pthread_mutex_timedlock (      pthr
                             const stru
int pthread_mutex_unlock    (      pthr
int pthread_cond_wait       (      pthr
                                   pthr
int pthread_cond_timedwait  (      pthr
                                   pthread_mutex_t      *mutex,
                             const struct timespec      *abstime);

int pthread_cond_signal     (      pthread_cond_t       *cond);
int pthread_cond_broadcast  (      pthread_cond_t       *cond);
```

undefined,

if called out of order!

Monitors in ‘C’ / POSIX

(operators)



...

```
int pthread_mutex_lock        (      pthread_mutex_t      *mutex);
int pthread_mutex_trylock     (      pthread_mutex_t      *mutex);
int pthread_mutex_timedlock   (      p
                               const
int pthread_mutex_unlock      (      p
int pthread_cond_wait         (      pthread_cond_t       *cond,
                                     pthread_mutex_t      *mutex);
int pthread_cond_timedwait    (      pthread_cond_t       *cond,
                                     pthread_mutex_t      *mutex,
                               const struct timespec      *abstime);
int pthread_cond_signal       (      pthread_cond_t       *cond);
int pthread_cond_broadcast    (      pthread_cond_t       *cond);
```

can be called any time, anywhere
(multiple lock reaction can be specified)

Monitors in ‘C’ / POSIX

(example, definitions)

```
#define BUFF_SIZE 10

typedef struct {
   pthread_mutex_t mutex;
   pthread_cond_t  buffer_not_full;
   pthread_cond_t  buffer_not_empty;
   int             count, first, last;
   int             buf[BUFF_SIZE];
} buffer;
```

Monitors in ‘C’ / POSIX

(example, operations)

```
int append (int item, buffer *B) {

   PTHREAD_MUTEX_LOCK (&B->mutex);
   while (B->count == BUFF_SIZE) {
      PTHREAD_COND_WAIT (
         &B->buffer_not_full,
         &B->mutex);
   }
   PTHREAD_MUTEX_UNLOCK (&B->mutex);
   PTHREAD_COND_SIGNAL (
      &B->buffer_not_empty);
   return 0;
}
```

```
int take (int *item, buffer *B) {

   PTHREAD_MUTEX_LOCK (&B->mutex);
   while (B->count == 0) {
      PTHREAD_COND_WAIT (
         &B->buffer_not_empty,
         &B->mutex);
   }
   PTHREAD_MUTEX_UNLOCK (&B->mutex);
   PTHREAD_COND_SIGNAL (
      &B->buffer_not_full);
   return 0;
}
```

Monitors in Java

Java provides two mechanisms to construct monitors:

- Synchronized methods and code blocks
  all methods and code blocks which are using the `synchronized` tag
  are mutually exclusive with respect to the addressed class.
- Notification methods: `wait`, `notify`, and `notifyAll`
  can be used only in synchronized regions and are waking any or all threads,
  which are waiting in the same synchronized object.

Monitors in Java

Considerations:

1. Synchronized methods and code blocks:

- In order to implement a monitor *all* methods in an object need to be synchronized.

  ☞ any other standard method can break the monitor and enter at any time.

- Methods outside the monitor-object can synchronize at this object.

  ☞ it is impossible to analyse a monitor locally, since lock accesses can exist all over the system.

- Static data is shared between all objects of a class.

  ☞ access to static data need to be synchronized over the whole class.

  Either in static synchronized blocks: `synchronized (this.getClass())` **{…}**
  or in static methods: `public synchronized static` *<method>* **{…}**

Monitors in Java

Considerations:

2. Notification methods: `wait`, `notify`, and `notifyAll`

- `wait` suspends the thread and releases the local lock only

  ☞ nested `wait`-calls will keep all enclosing locks.

- `notify` and `notifyAll` does not release the lock.

  ☞ methods, which are activated via notification need to wait for lock-access.

- `wait`-suspended threads are hold in a queue (Real-time Java only!), thus `notify{All}` is waking the threads in order ☞ livelocks are prevented at this level .

- There are no explicit conditional variables.

  ☞ every notified thread needs
  to wait for the lock to be released **and** to re-evaluate its entry condition

Monitors in Java

(multiple-readers-one-writer-example)

each of the **readers** uses these monitor.calls:

```
startRead ();
   // read the shared data only
stopRead ();
```

each of the **writers** uses these monitor.calls:

```
startWrite ();
   // manipulate the shared data
stopWrite ();
```

☞ construct a monitor, which allows
multiple readers
or
one writer
at a time inside the critical regions

Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
public class ReadersWriters
{
   private int     readers        = 0;
   private int     waitingWriters = 0;
   private boolean writing        = false;

…
```

Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
…   public synchronized void StartWrite () throws InterruptedException
    {
       while (readers > 0 || writing)
       {
          waitingWriters++;
          wait();
          waitingWriters--;
       }
       writing = true;
    }

    public synchronized void StopWrite()
    {
       writing = false;
       notifyAll ();
    } …
```

Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
…   public synchronized void StartRead () throws InterruptedException
    {
       while (writing || waitingWriters > 0)
       {
          wait();
       }
       readers++;
    }

    public synchronized void StopRead()
    {
       readers--;
       if (readers == 0) notifyAll();
    }
}
```

whenever a synchronized region is left:

- **all** thread are notified
- **all** threads are
  re-evaluating their guards

Monitors in Java

Standard monitor solution:

- declare the monitored data-structures private to the monitor object (non-static).
- introduce a class `ConditionVariable`:

```
public class ConditionVariable {
   public boolean wantToSleep = false;
}
```

- introduce synchronization-scopes in monitor-methods:
  ☞ synchronize on the *adequate conditional variables* ***first*** and
  ☞ synchronize on the *monitor-object* ***second.***
- make sure that ***all*** methods in the monitor are implementing the correct synchronizations.
- make sure that ***no other method*** in the whole system is synchronizing on this monitor-object.

Monitors in Java

(multiple-readers-one-writer-example: usage of external conditional variables)

```
public class ReadersWriters
{
   private int     readers        = 0;
   private int     waitingReaders = 0;
   private int     waitingWriters = 0;
   private boolean writing        = false;

   ConditionVariable OkToRead  = new ConditionVariable ();
   ConditionVariable OkToWrite = new ConditionVariable ();

...
```

Monitors in Java

```
…  public void StartWrite () throws InterruptedException
   {
      synchronized (OkToWrite)
      {
         synchronized (this)
         {
            if (writing | readers > 0) {
               waitingWriters++;
               OkToWrite.wantToSleep = true;
            } else {
               writing = true;
               OkToWrite.wantToSleep = false;
            }
         }
         if (OkToWrite.wantToSleep) OkToWrite.wait ();
   }  } …
```

Monitors in Java

```
…  public void StopWrite ()
   {
      synchronized (OkToRead)
      {
         synchronized (OkToWrite)
         {
            synchronized (this)
            {
               if (waitingWriters > 0) {
                  waitingWriters--;
                  OkToWrite.notify (); // wakeup one writer
               } else {
                  writing = false;
                  OkToRead.notifyAll (); // wakeup all readers
                  readers = waitingReaders;
                  waitingReaders = 0;
               }
   }  }  }  } …
```

Monitors in Java

```
…  public void StartRead () throws InterruptedException
   {
      synchronized (OkToRead)
      {
         synchronized (this)
         {
            if (writing | waitingWriters > 0) {
               waitingReaders++;
               OkToRead.wantToSleep = true;
            } else {
               readers++;
               OkToRead.wantToSleep = false;
            }
         }
         if (OkToRead.wantToSleep) OkToRead.wait ();
   }  } …
```

Monitors in Java

```
…   public void StopRead ()
    {
        synchronized (OkToWrite)
        {
            synchronized (this)
            {
                readers--;
                if (readers == 0 & waitingWriters > 0) {
                    waitingWriters--;
                    OkToWrite.notify ();
                }
            }
        }
    }
}
```

Object-orientation and synchronization

Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analysed considering the implementation of all involved methods and guards:

☞ new methods cannot be added without re-evaluating the whole class!

In opposition to the general re-usage idea of object-oriented programming,
the re-usage of synchronized classes (e.g. monitors) need to be considered carefully.

☞ The parent class might need to be adapted in order to suit the global synchronization scheme.

☞ Inheritance anomaly (Matsuoka & Yonezawa '93)

Methods to design and analyse expandible synchronized systems exist,
but are fairly complex and are not provided in any current object-oriented language.

Monitors in POSIX & Java

☞ flexible and universal,
but relies on conventions rather than compilers

POSIX offers conditional variables

Java is more supportive than POSIX
in terms of data-encapsulation

Extreme care must be taken when employing
object-oriented programming and monitors

Nested monitor calls

Assuming a thread in a monitor is calling an operation in another monitor
and is suspended at a conditional variable there:

- ☞ the called monitor is aware of the suspension and allows other threads to enter.
- ☞ the calling monitor is possibly *not aware* of the suspension and **keeps its lock!**
- ☞ the unjustified locked calling monitor
  reduces the system performance and leads to potential deadlocks.

Suggestions to solve this situation:

- Maintain the lock anyway: e.g. POSIX, Real-time Java
- Prohibit nested procedure calls: e.g. Modula-1
- Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada95

Criticism of monitors

- Mutual exclusion is solved elegantly and safely.
- Conditional synchronization is on the level of semaphores still
  - ☞ all criticism on semaphores apply

☞ mixture of low-level and high-level synchronization constructs.

Synchronization by protected objects

Combine

- the **encapsulation** feature of monitors

with

- the **coordinated entries** of conditional critical regions

to

☞ Protected objects

- *all* controlled data and operations are encapsulated
- *all* operations are mutual exclusive
- entry guards are *attached* to operations
- the protected interface allows for operations on data
- no protected data is accessible (other than by defined operations)
- tasks are queued (according to their priorities)

Synchronization by protected objects in Ada95

(simultaneous read-access)

Some read-only operations *do not need to be mutual exclusive*:

```
protected type Shared_Data (Initial : Data_Item) is

   function  Read return Data_Item;
   procedure Write (New_Value : in Data_Item);

private
   The_Data : Data_Item := Initial;
end Shared_Data_Item;
```

- protected *functions* can have '`in`' parameters only and are not allowed to alter the private data (enforced by the compiler).

☞ protected functions allow **simultaneous access** (but mutual exclusive with other operations).

- there is no defined priority between functions and other protected operations in Ada95.

Synchronization by protected objects in Ada95

Condition synchronization is realized in the form of protected procedures combined with boolean conditional variables (**barriers**): ☞ **entries** in Ada95:

```
Buffer_Size : constant Integer := 10;

type    Index    is mod Buffer_Size;
subtype Count    is Natural range 0 .. Buffer_Size;
type    Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is

   entry Get (Item : out Data_Item);
   entry Put (Item : in Data_Item);
private
   First  : Index := Index'First;
   Last   : Index := Index'Last;
   Num    : Count := 0;
   Buffer : Buffer_T;

end Bounded_Buffer;
```

Synchronization by protected objects in Ada95

(barriers)

```
protected body Bounded_Buffer is
   entry Get (Item : out Data_Item) when Num > 0 is
      begin
         Item  := Buffer (First);
         First := First + 1;
         Num   := Num - 1;
      end Get;
   entry Put (Item : in Data_Item) when Num < Buffer_Size is
      begin
         Last          := Last + 1;
         Buffer (Last) := Item;
         Num           := Num + 1;
      end Put;
end Bounded_Buffer;
```

Synchronization by protected objects in Ada95

Protected entries are used like task entries:

```
Buffer : Bounded_Buffer;
```

```
select
   Buffer.Put (Some_Data);
or
   delay 10.0;
      -- do something after 10 s.
end select;

select
   Buffer.Get (Some_Data);
else
   -- do something else
end select;
```

```
select
   delay 10.0;
then abort
   Buffer.Put (Some_Data);
      -- try to enter for 10 s.
end select;

select
   Buffer.Get (Some_Data);
then abort
   -- meanwhile try something else
end select;
```

Synchronization by protected objects in Ada95

(barrier evaluation)

Barrier evaluations and task activations:

- on ***calling a protected entry***, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).
- on ***leaving a protected procedure or entry***, related barriers with tasks queued are evaluated (only those parts of the barriers which might have been altered by this procedure / entry or which might have changed since the last evaluation).

Barriers are not evaluated *while inside* a protected object or *on leaving a protected function*.

Synchronization by protected objects in Ada95

(operations on entry queues)

The `count` attribute indicate the number of tasks waiting at a specific queue:

```
protected Blocker is

   entry Proceed;

private
   Release : Boolean := False;
end Blocker;
```

```
protected body Blocker is

   entry Proceed
      when Proceed'count = 5
        or Release is
   begin
      Release := Proceed'count > 0;
   end Proceed;

end Blocker;
```

Synchronization by protected objects in Ada95

(operations on entry queues)

The `count` attribute indicate the number of tasks waiting at a specific queue:

```
protected type Broadcast is

   entry Receive  (M: out Message);
   procedure Send (M: in  Message);

private

   New_Message : Message;
   Arrived     : Boolean := False;

end Blocker;
```

```
protected body Broadcast is

   entry Receive (M: out Message)
      when Arrived is
   begin
      M := New_Message
      Arrived := Receive'count > 0;
   end Proceed;

   procedure Send (M: in  Message) is
   begin
      New_Message := M;
      Arrived := Receive'count > 0;
   end Send;

end Blocker;
```

Synchronization by protected objects in Ada95

(entry families, requeue & private entries)

Further refinements on task control by:

- ***Entry families***:
  a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.
- ***Requeue facility***:
  protected operations can use ‘`requeue`’ to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.

  ‘***Internal progress first*’*-rule***: internally requeued tasks are placed at the **head** of the waiting queue!
- ***Private entries***:
  protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.

Synchronization by protected objects in Ada95

(requeue & private entries)

How to implement a queue, at which every task can be released only once per triggering event?

```
package Single_Release is

   entry     Wait;
   procedure Trigger;

end Single_Release;
```

Synchronization by protected objects in Ada95

(requeue & private entries)

How to implement a queue, at which every task can be released only once per triggering event?

☞ e.g. by employing a second (private) entry:

```
package Single_Release is

   entry     Wait;
   procedure Trigger;

private
   Front_Door,
   Main_Door   : Boolean := False;

   entry Queue;

end Single_Release;
```

Synchronization by protected objects in Ada95

(requeue & private entries)

```
package body Single_Release is

   entry Wait
      when Front_Door is

      begin
         if Wait'Count = 0 then
            Front_Door := False;
            Main_Door  := True;
         end if;

         requeue Queue;

      end Wait;

   entry Queue
      when Main_Door is

      begin
         if Queue’count = 0 then
            Main_Door := False;
         end if;;
      end Queue;

   procedure Trigger is
      begin
         Front_Door := True;
      end Trigger;

end Single_Release;
```

opening the main door before requeuing?

Synchronization by protected objects in Ada95

(restrictions applying to protected operations)

Code inside a protected procedure, function or entry is bound to non-blocking operations (which would keep the whole protected object locked).

Thus the following operations are prohibited:

- entry call statements
- delay statements
- task creations or activations
- calls to sub-programs which contains a potentially blocking operation
- select statements
- accept statements

☞ The **`requeue`** facility allows for a potentially blocking operation, but releases the current lock!

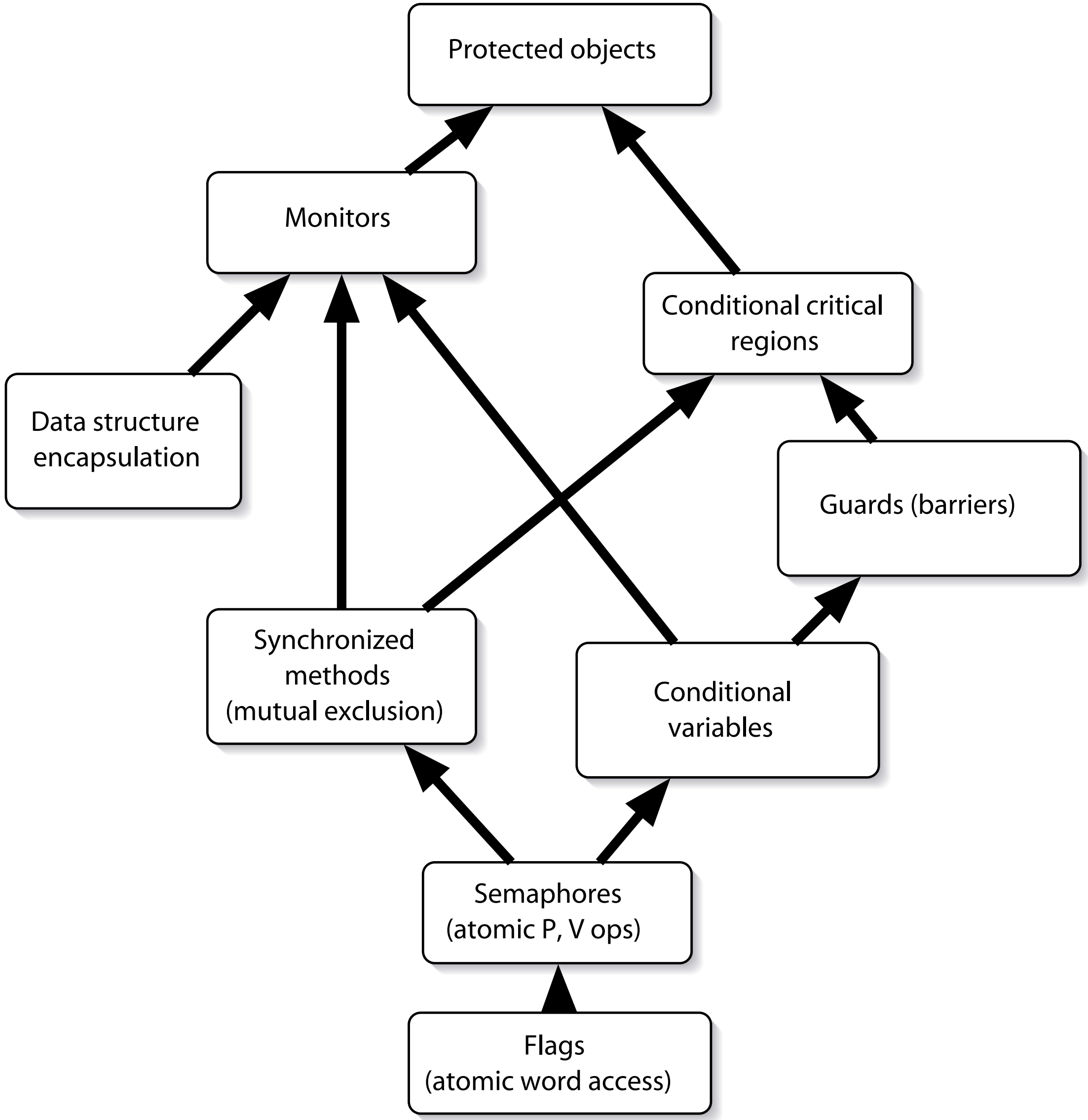
Summary

Shared memory based synchronization

General

Criteria:

- level of abstraction
- centralized vs. distributed concepts
- support for consistency and correctness validations
- error sensitivity
- predictability
- efficiency

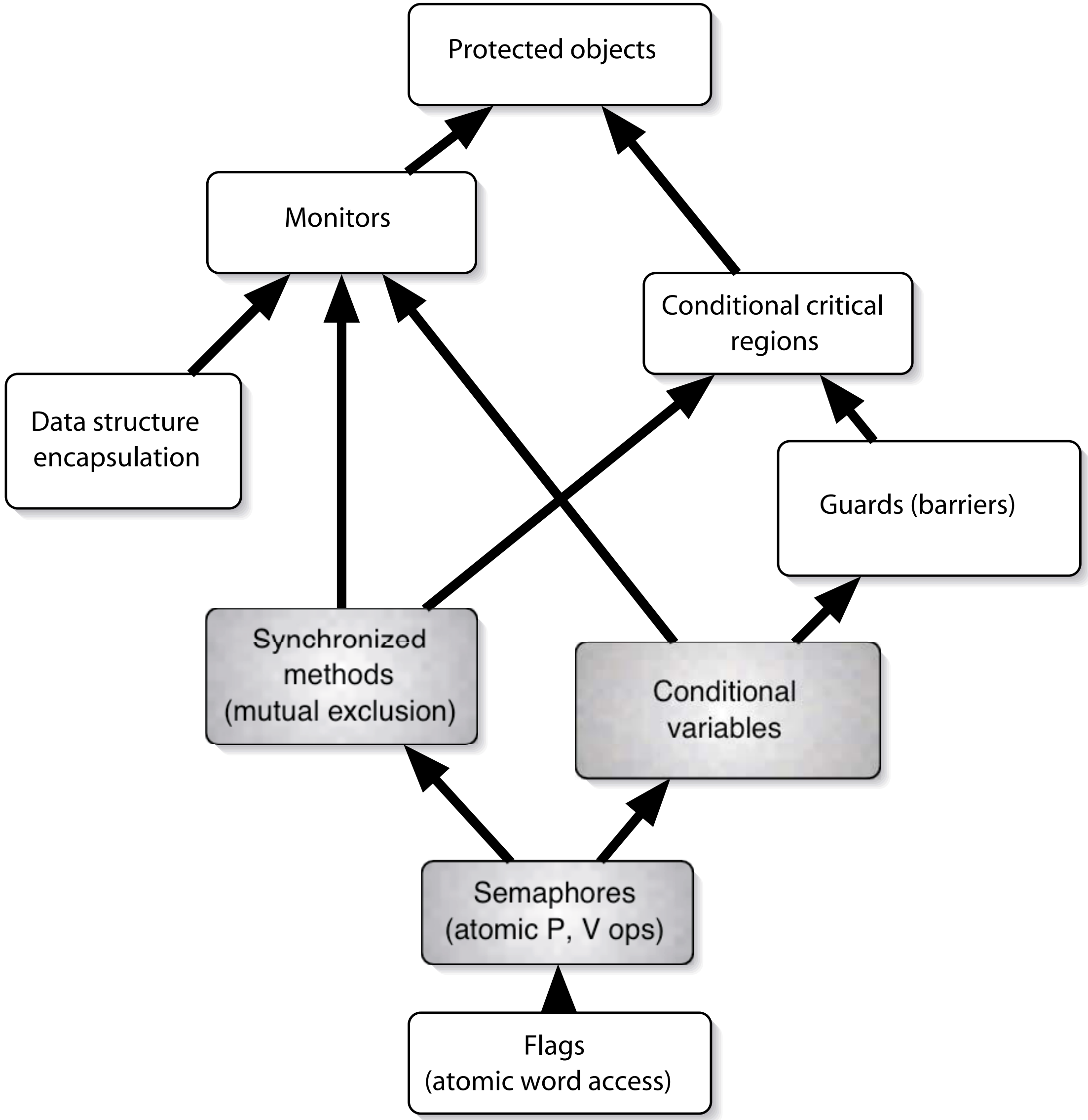


Summary

Shared memory based synchronization

POSIX

- all low level constructs available.
- no connection with the actual data-structures.
- error-prone.
- non-determinism introduced by 'release some' semantics of conditional variables (cond_signal).

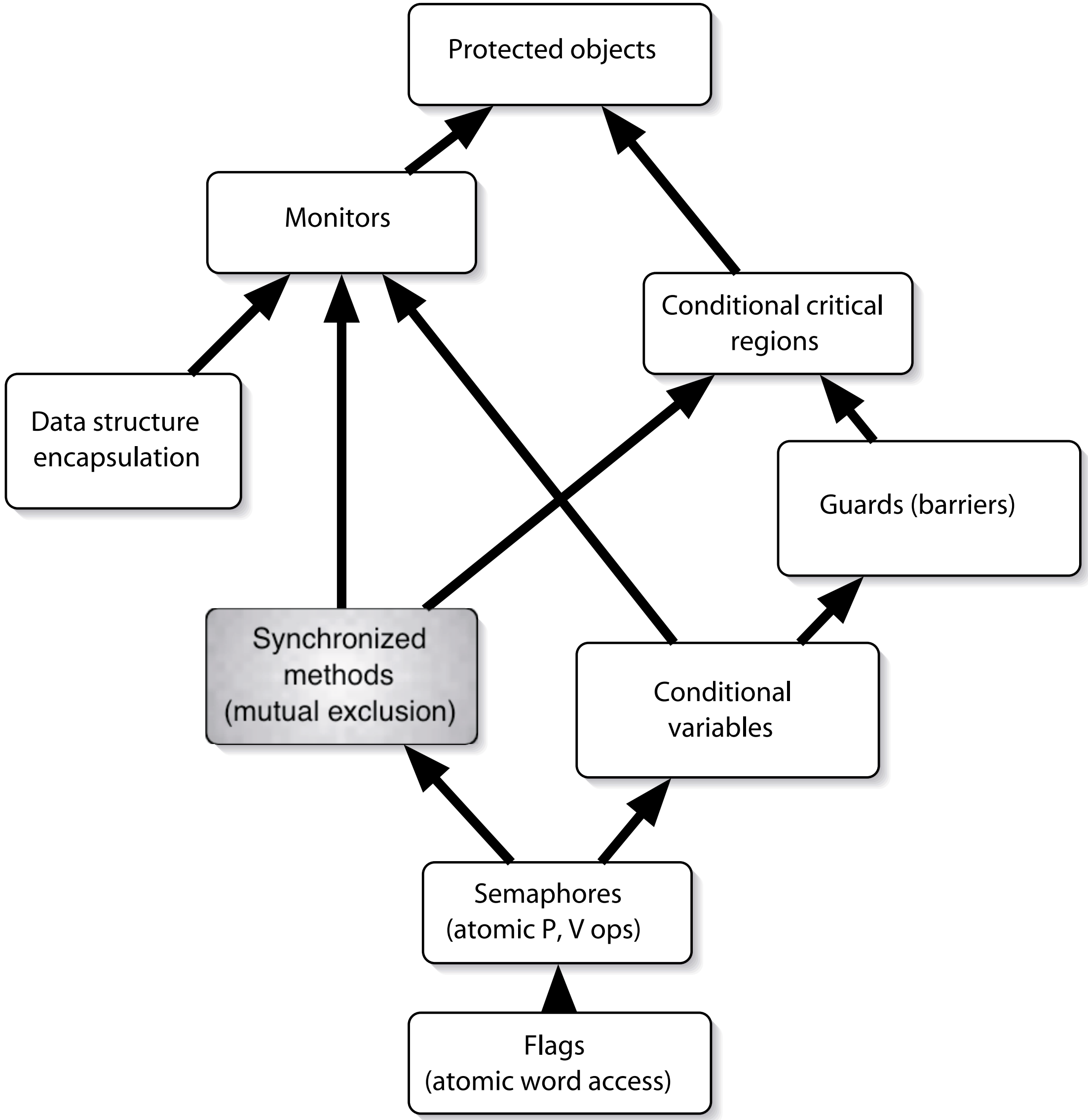


Summary

Shared memory based synchronization

Java

- mutual exclusion (synchronized methods) as the only support.
- general notification feature (no conditional variables)
- non-restricted object oriented extension introduces hard to predict timing behaviours.



Summary

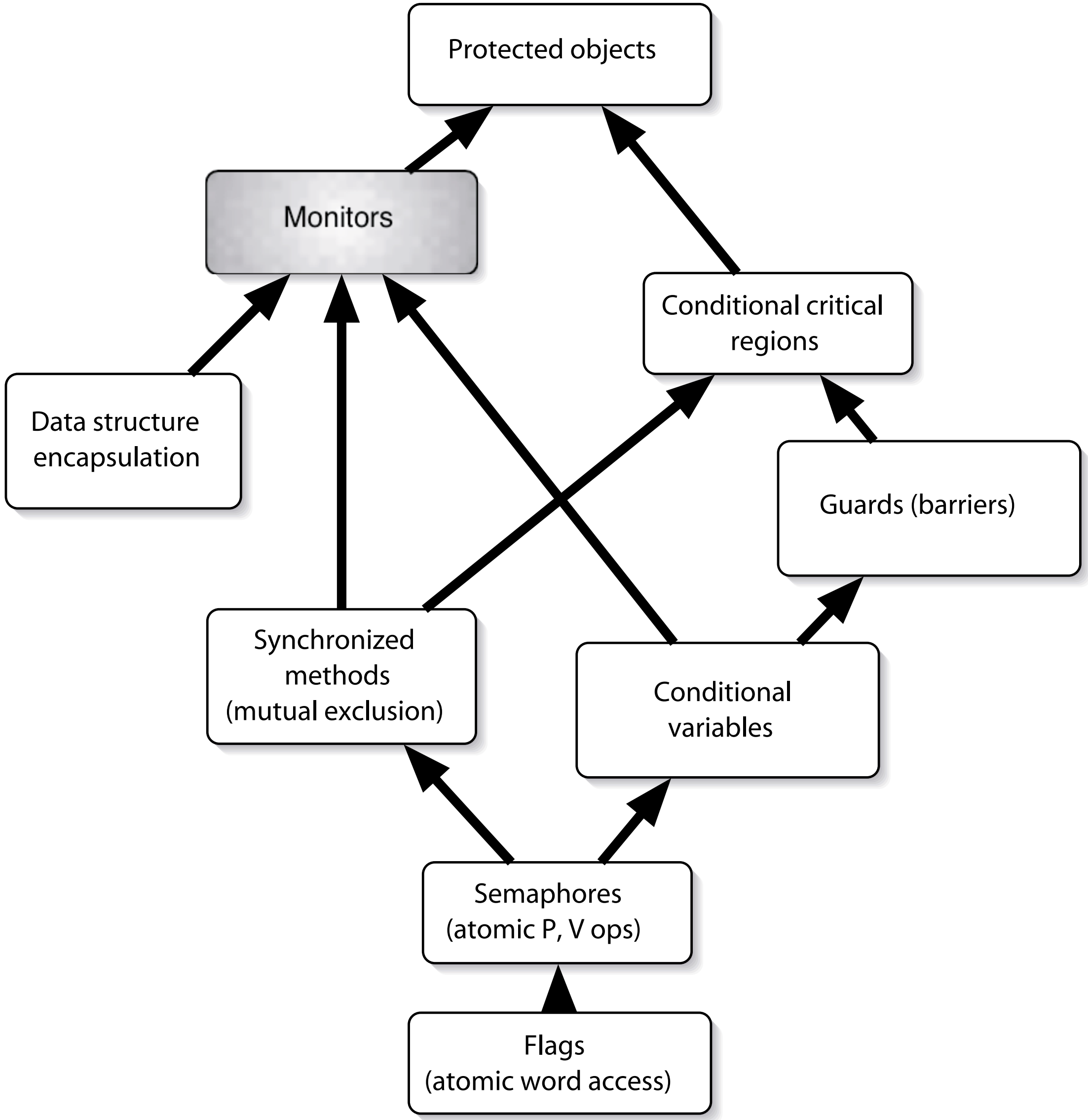
Shared memory based synchronization

Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).

… no more, no less, …

☞ all features of and criticism about monitors apply.



Summary

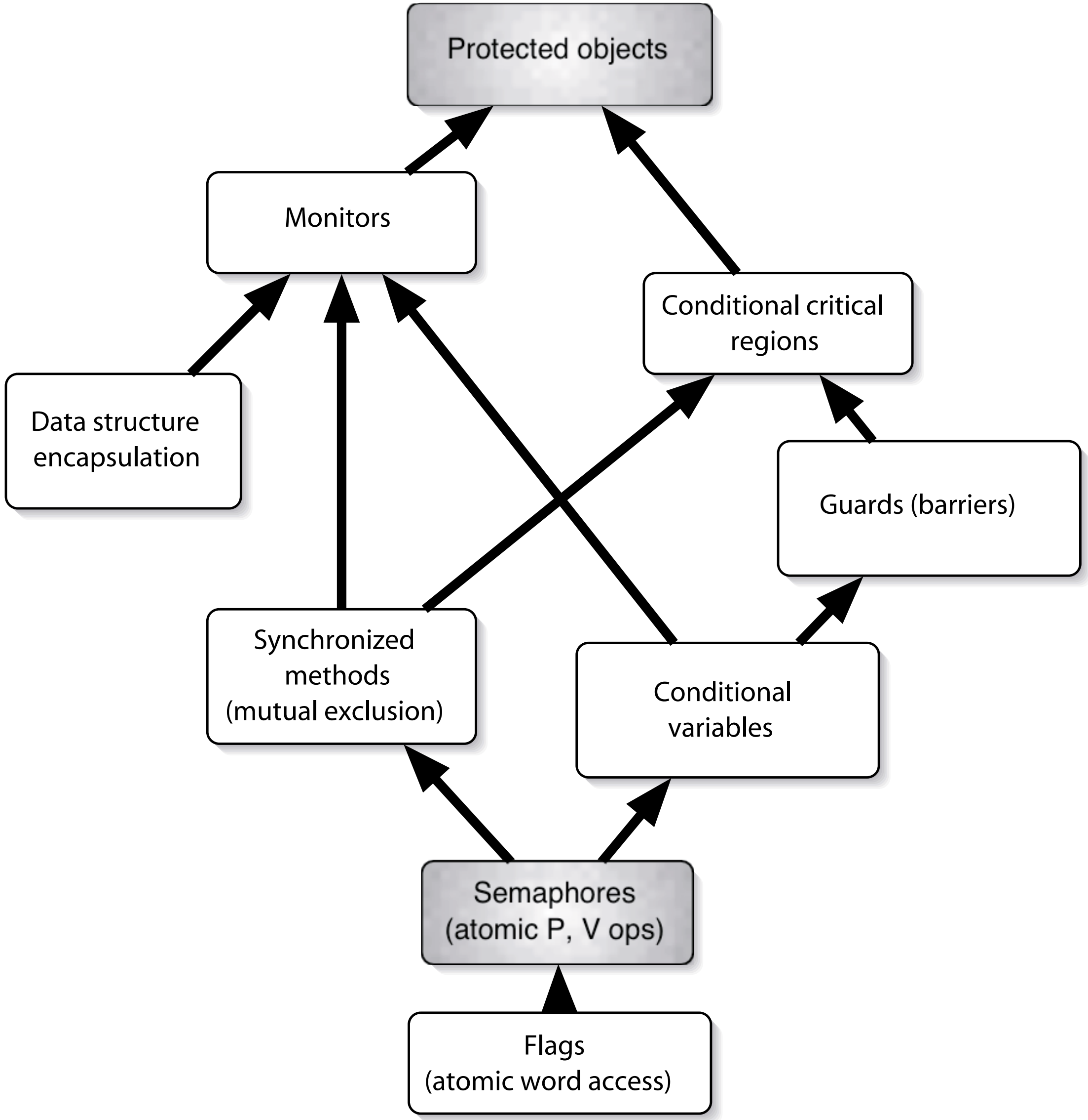
Shared memory based synchronization

Ada95

- complete synchronization support
- low-level semaphores
  for very special cases.
- predictable timing (☞ scheduler).

☞ most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather then the programmer.

(Ada95 is currently without any mainstream competitor in this field)

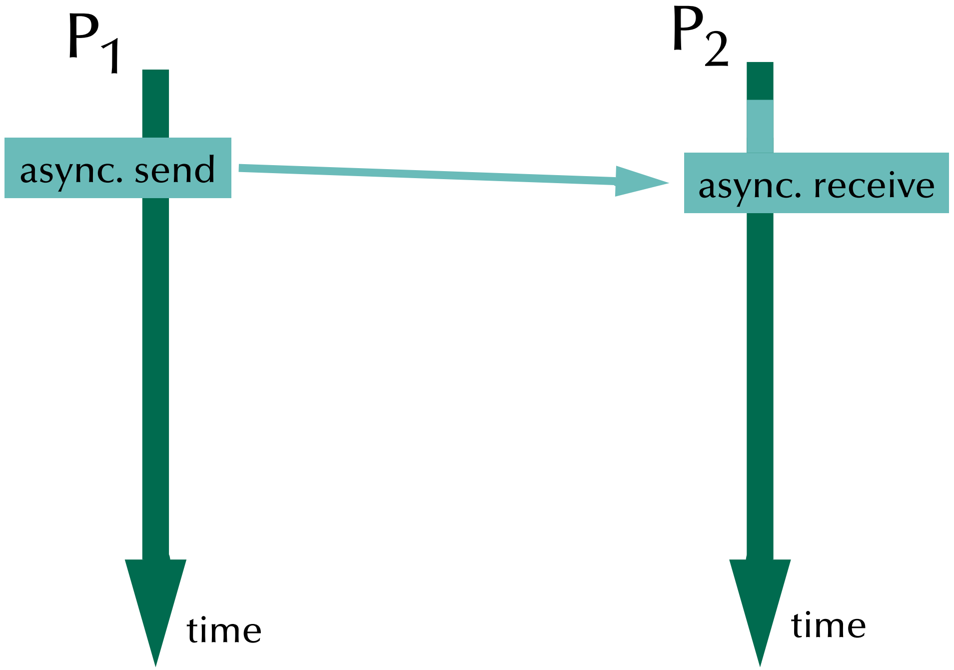


Message-based synchronization

- Synchronization model
    - Asynchronous
    - Synchronous
    - Remote invocation
- Addressing (name space)
    - direct communication
    - mail-box communication
- Message structure
    - arbitrary
    - restricted to ‘basic’ types
    - restricted to un-typed communications

Message-based synchronization

Asynchronous messages

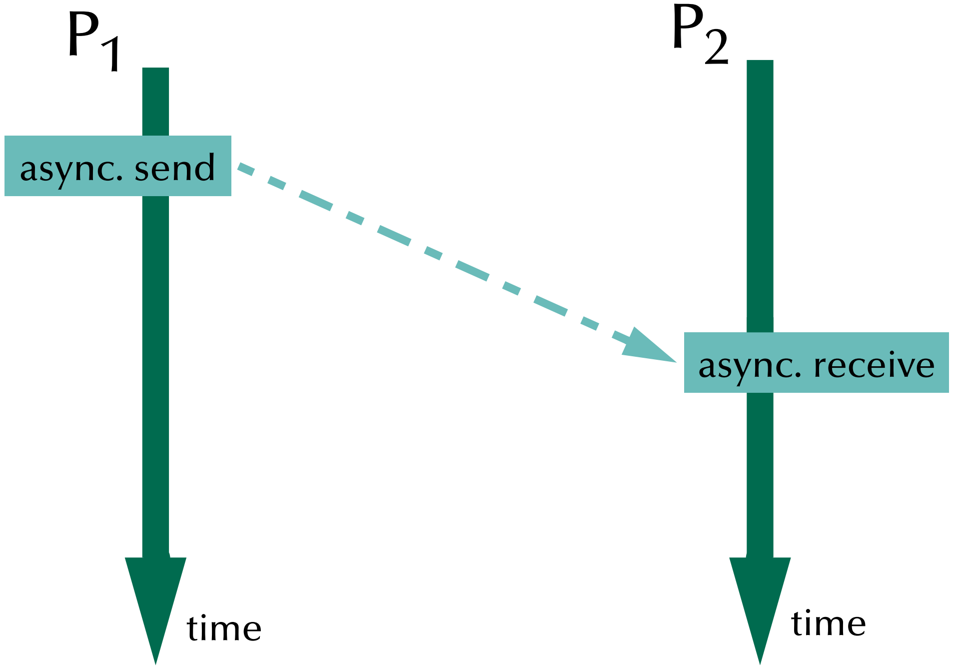


If there is a listener:

☞ send the message directly

Message-based synchronization

Asynchronous messages

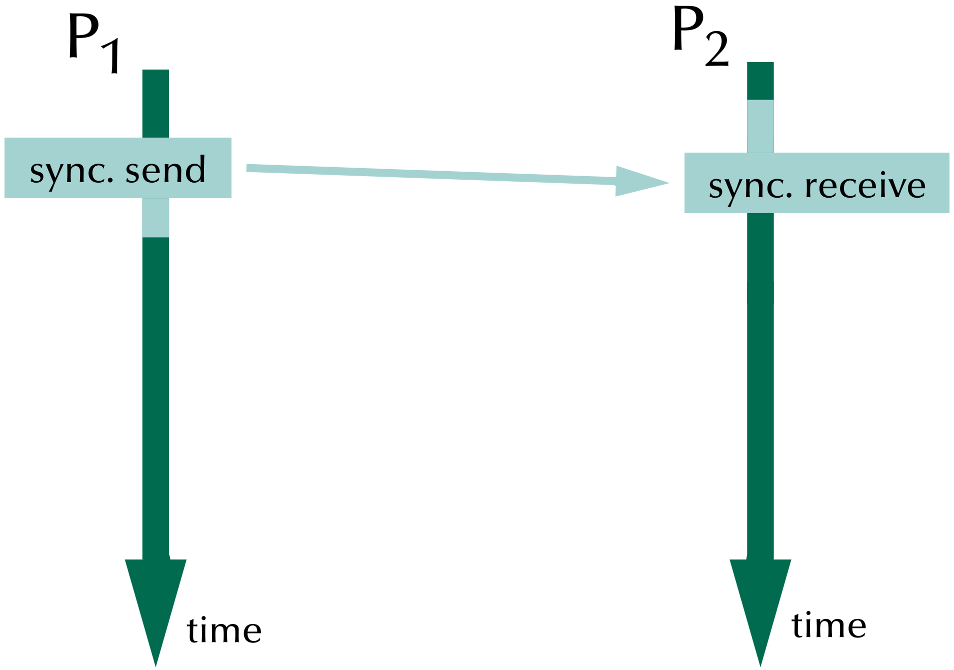


If the receiver becomes available at a later stage:

☞ the message need to be buffered

Message-based synchronization

Synchronous messages



Delay the sender:

- until the receiver got the message

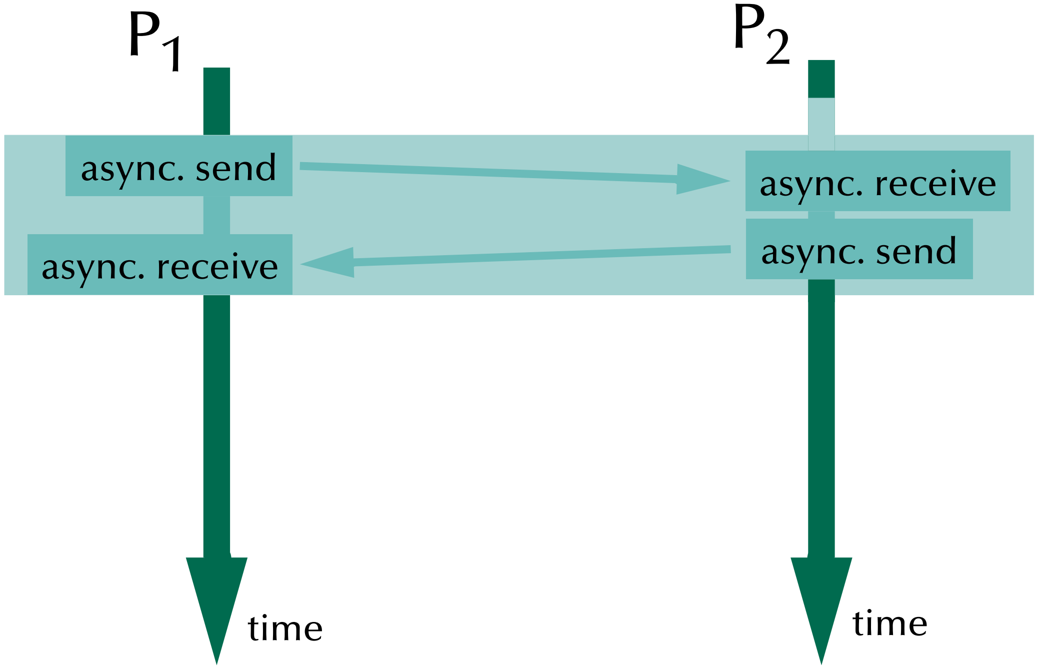
Message-based synchronization

Synchronous messages

Delay the sender:

- until the receiver got the message

☞ two asynchronous messages required

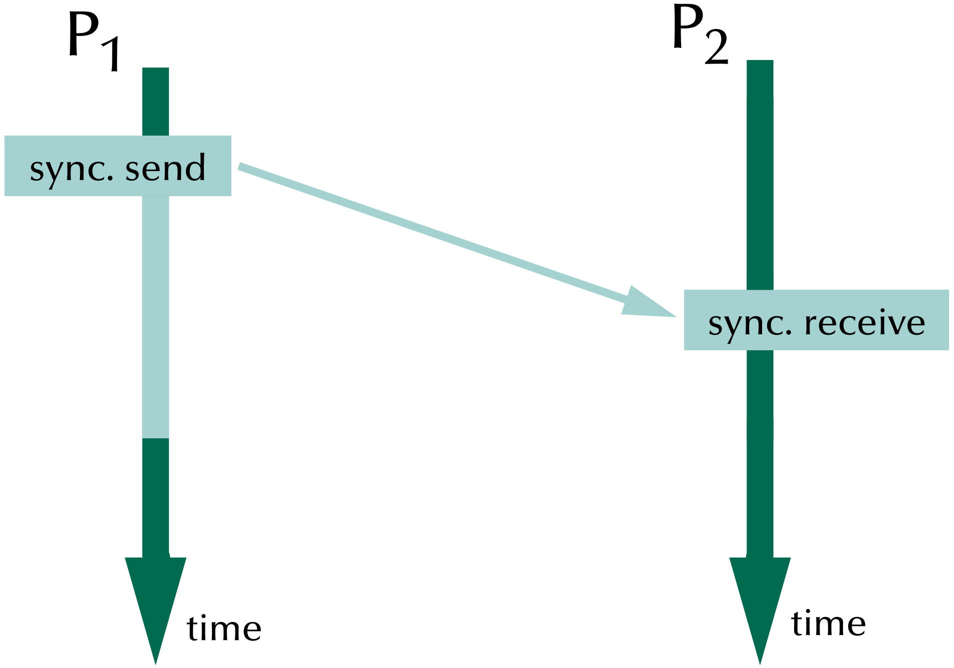


Message-based synchronization

Synchronous messages

Delay the sender until:

- a receiver is available
- a receiver got the message

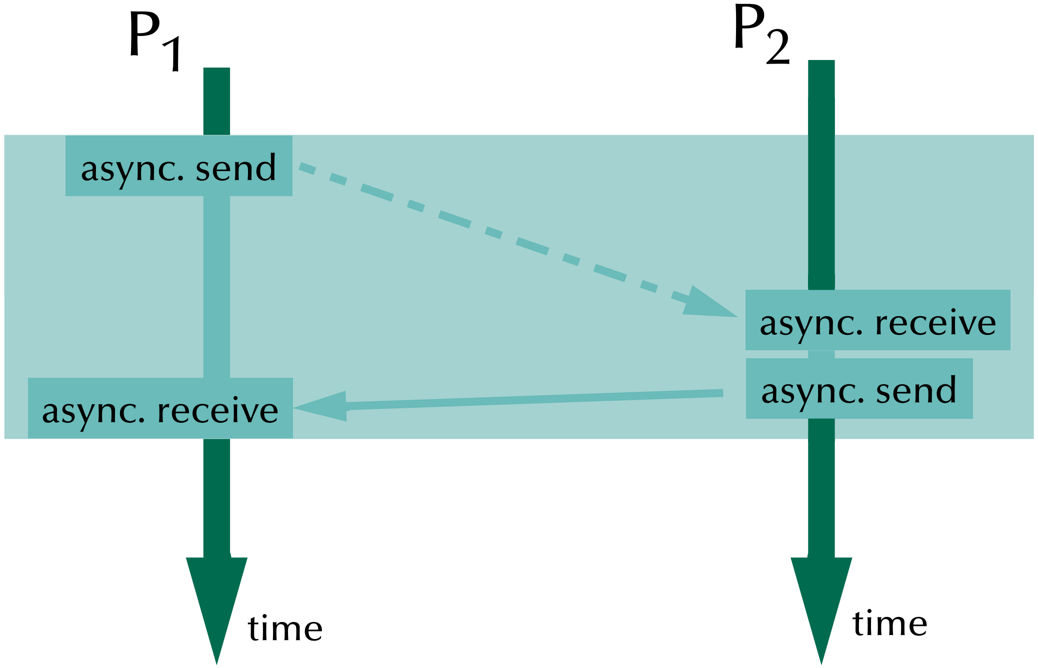


Message-based synchronization

Synchronous messages

If the receiver becomes available at a later stage:

☞ messages need to be buffered

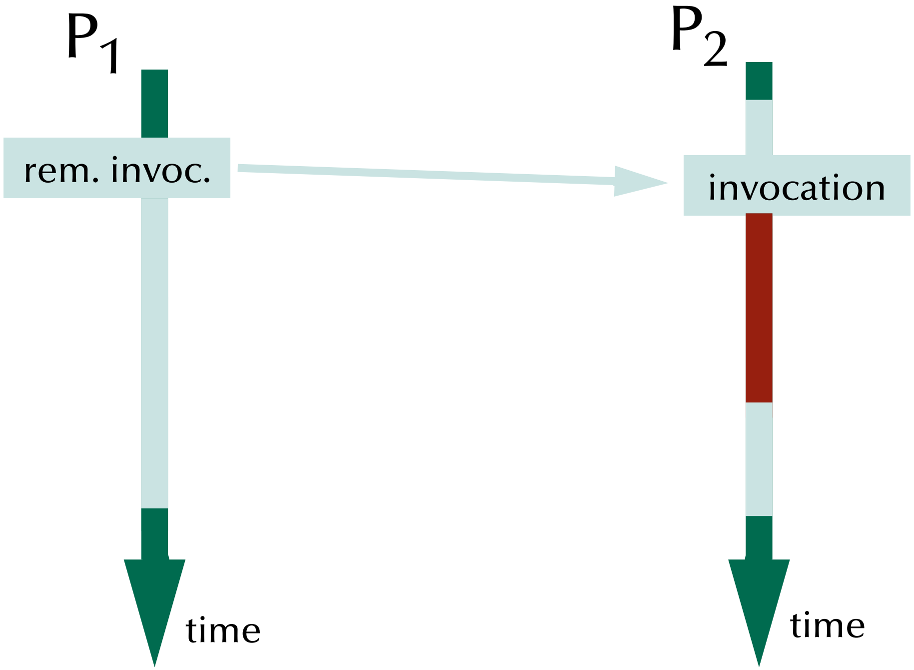


Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

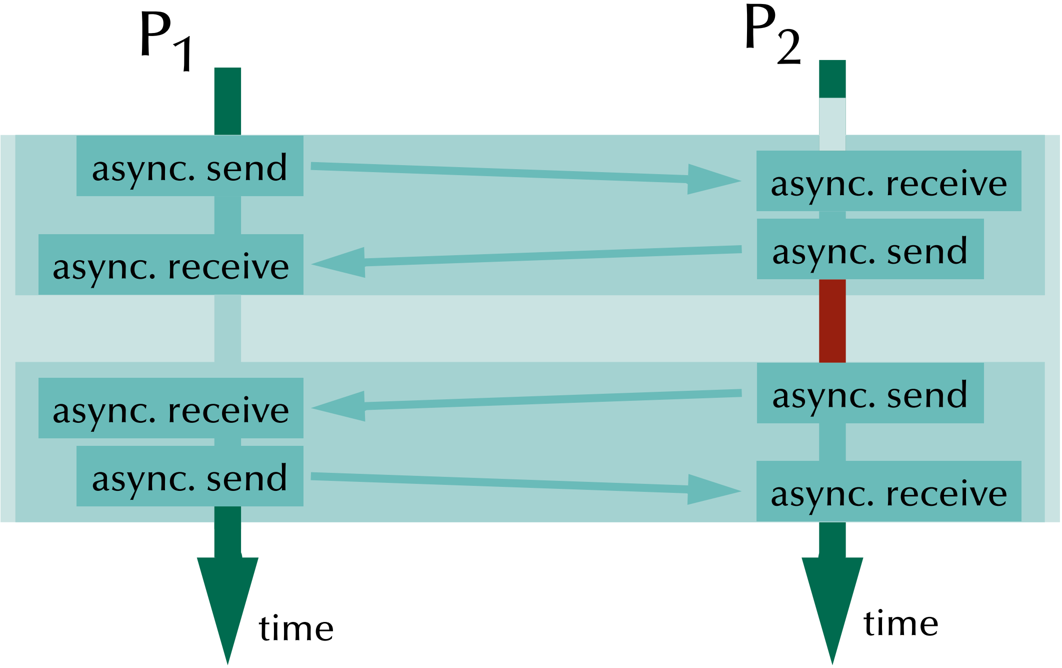


Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

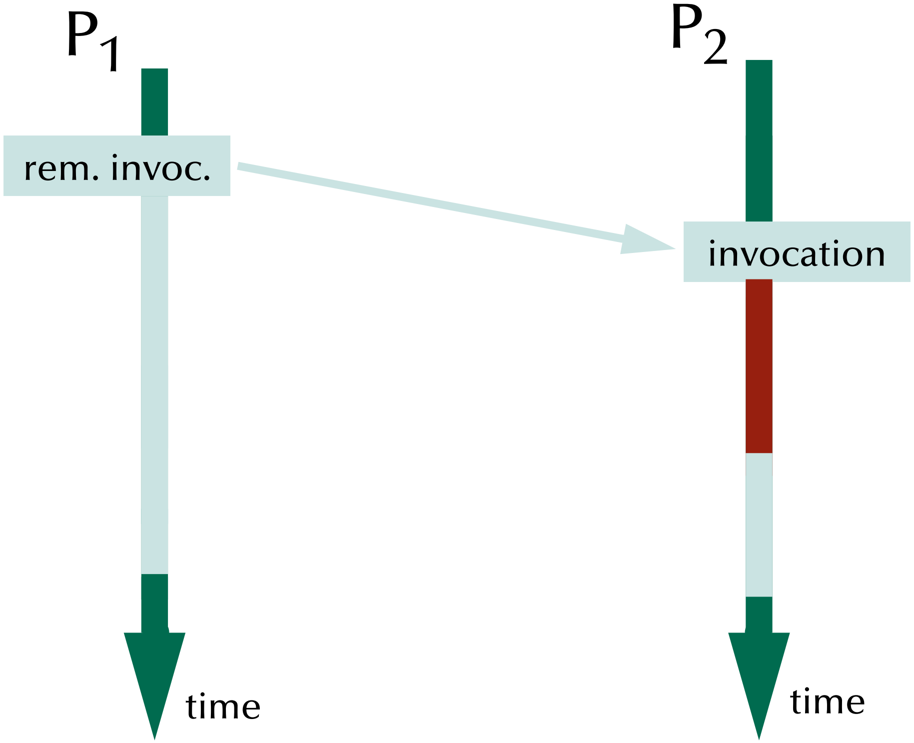


Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine

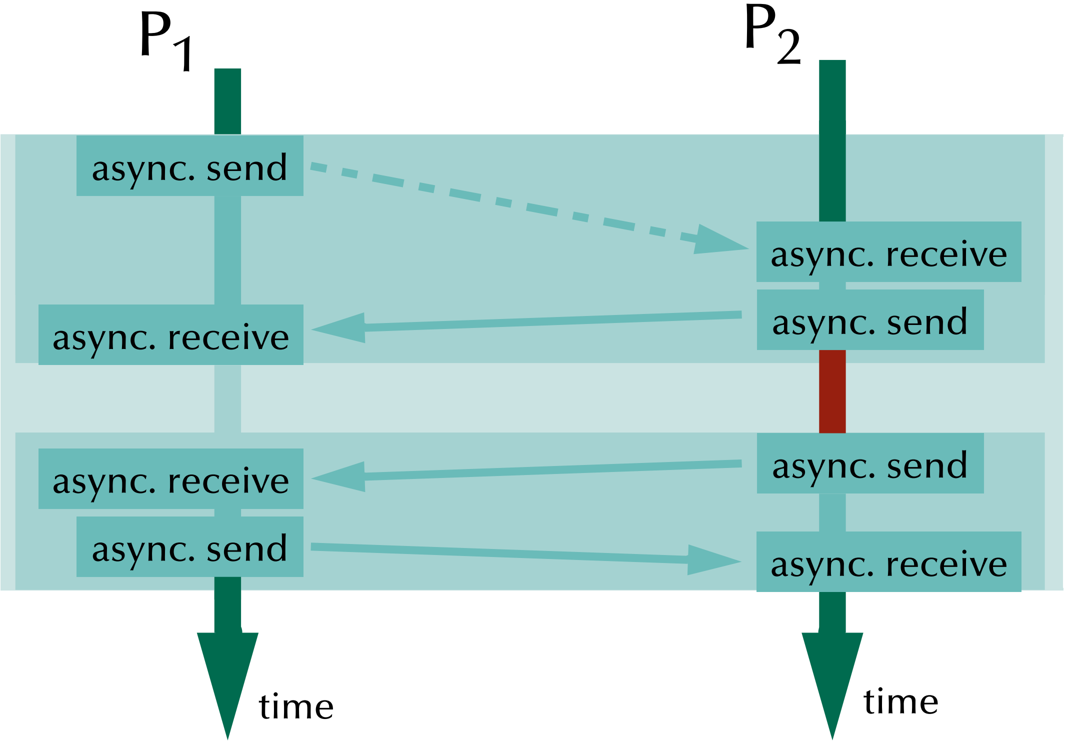


Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine

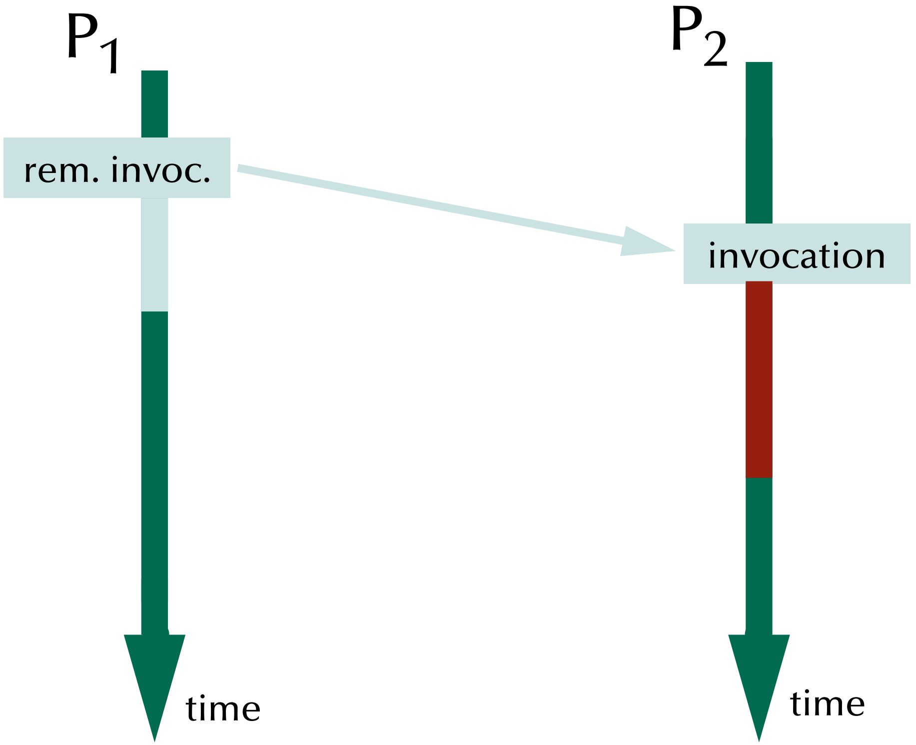


Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message

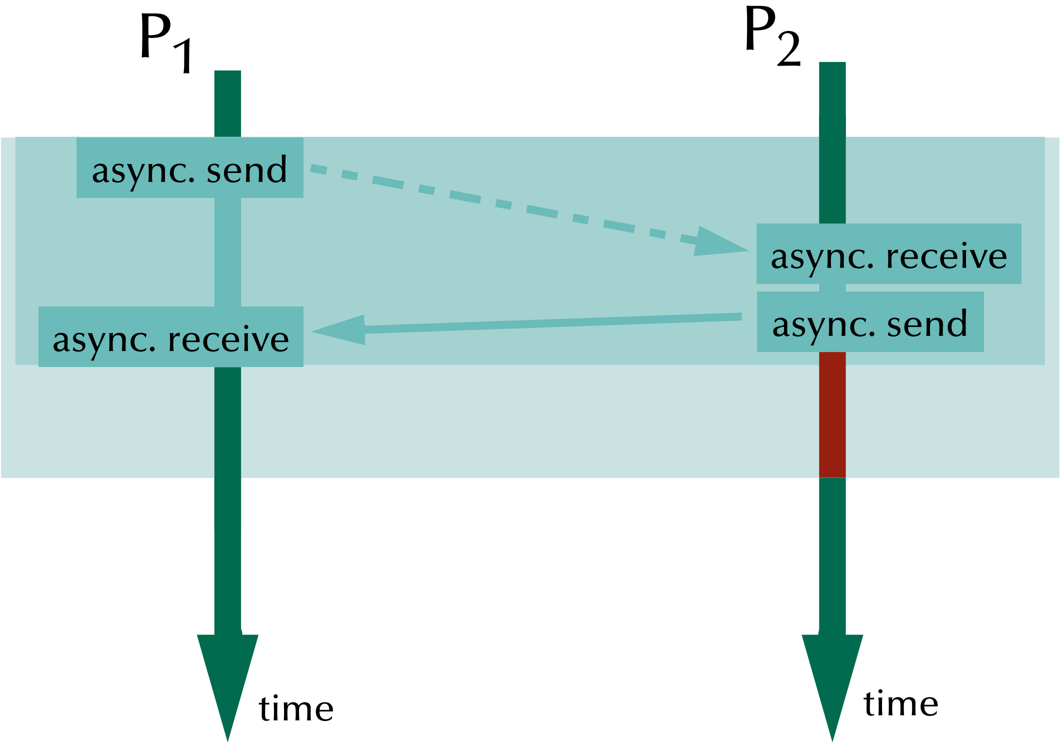


Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message



Synchronous vs. asynchronous communications

Purpose ‘**synchronization**’: ☞ synchronous messages / remote invocations
Purpose ‘**in-time delivery**’: ☞ asynchronous messages / asynchronous remote invocations

☞ ‘Real’ synchronous message passing in distributed systems requires hardware support.

☞ Asynchronous message passing requires the usage of (infinite?) buffers.

- Synchronous communications are emulated by a combination of asynchronous messages in some systems.
- Asynchronous communications can be emulated in synchronized message passing systems by introducing ‘buffer-tasks’ (de-coupling sender and receiver as well as allowing for broadcasts).

Addressing (name space)

Direct vs. indirect:

```
send     <message> to   <process-name>
wait for <message> from <process-name>
send     <message> to   <mailbox>
wait for <message> from <mailbox>
```

Asymmetrical addressing:

```
send     <message> to …
wait for <message>
```

☞ Client-server paradigm

Addressing (name space)

Communication medium:

Connections	*Functionality*
one-to-one	buffer, queue, synchronization
one-to-many	multicast
one-to-all	broadcast
many-to-one	local server, synchronization
all-to-one	general server, synchronization
many-to-many	general network- or bus-system

Message structure

- Machine dependent representations need to be taken care of in a distributed environment.
- Communication system is often outside the typed language environment.

Most communication systems are handling streams (packets) of a basic element type only.

☞ Conversion routines for data-structures other then the basic element type are supplied …

… manually (POSIX)
… semi-automatic (Real-time CORBA)
… automatic and are typed-persistent (Ada95)

Message structure (Ada95)

```
package Ada.Streams is
   pragma Pure (Streams);

   type Root_Stream_Type is abstract tagged limited private;

   type Stream_Element is mod implementation-defined;

   type Stream_Element_Offset is range implementation-defined;

   subtype Stream_Element_Count is
      Stream_Element_Offset range 0..Stream_Element_Offset'Last;

   type Stream_Element_Array is
      array (Stream_Element_Offset range <>) of Stream_Element;

   procedure Read  (…) is abstract;
   procedure Write (…) is abstract;

private
   … -- not specified by the language
end Ada.Streams;
```

Message structure (Ada95)

Reading and writing values of any type to a stream:

```
procedure S'Write(
   Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T);
procedure S'Class'Write(
   Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T'Class);

procedure S'Read(
   Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class'Read(
   Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)
```

Reading and writing values, bounds and discriminants of any type to a stream:

```
procedure S'Output(
   Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T);

function  S'Input(
   Stream : access Ada.Streams.Root_Stream_Type'Class) return T;
```

Message-based synchronization

Practical message-passing systems:

POSIX:	“message queues”: ☞ **ordered indirect [asymmetrical | symmetrical] asynchronous byte-level many-to-many message passing**
CHILL:	“buffers”, ”signals”: ☞ **ordered indirect [asymmetrical | symmetrical] [synchronous | asynchronous] typed [many-to-many | many-to-one] message passing**
Occam2:	“channels”: ☞ **indirect symmetrical synchronous fully-typed one-to-one message passing**
Ada95:	“(extended) rendezvous”: ☞ **ordered direct asymmetrical [synchronous | asynchronous] fully-typed many-to-one remote invocation**
Java:	no communication via messages available

Message-based synchronization

Practical message-passing systems:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX:	*	*	*		*		*	**bytes**			*	**message passing**
CHILL:	*	*	*	*	*		*	**typed**		*	*	**message passing**
Occam2:		*		*			*	**fully typed**	*			**message passing**
Ada95:	*		*	*	*	*		**fully typed**		*		**remote invocation**
Java:	no communication via messages available											

Message-based synchronization in Occam2

Communication is ensured by means of a ‘channel’, which:

- can be used by one writer and one reader process only
- and is synchronous:

```
CHAN OF INT SensorChannel:
PAR
   INT reading:
   SEQ i = 0 FOR 1000
      SEQ
         -- generate reading
         SensorChannel ! reading
   INT data:
   SEQ i = 0 FOR 1000
      SEQ
         SensorChannel ? data
         -- employ data
```

tasks are synchronized at these points

Message-based synchronization in CHILL

CHILL is the ‘CCITT High Level Language’,
where **CCITT** is the Comité Consultatif International Télégraphique et Téléphonique.
The CHILL language development was started in 1973 and standardized in 1979.

☞ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```
dcl SensorBuffer buffer (32) int;
…
send SensorBuffer (reading);      |        receive case
                                  |           (SensorBuffer in data) : …
                                  |        esac;

signal SensorChannel = (int) to consumertype;
…
send SensorChannel (reading)      |        receive case
   to consumer                    |           (SensorChannel in data): …
                                  |        esac;
```

Message-based synchronization in CHILL

CHILL is the ‘CCITT High Level Language’,
where **CCITT** is the Comité Consultatif International Télégraphique et Téléphonique.
The CHILL language development was started in 1973 and standardized in 1979.

☞ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```
dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading);
```

asynchronous

```
receive case
  (SensorBuffer in data) : ...
esac;
```

```
signal SensorChannel = (int) to consumertype;
...
send SensorChannel (reading)
    to consumer
```

synchronous

```
receive case
  (SensorChannel in data): ...
esac;
```

Message-based synchronization in Ada95

Ada95 supports remote invocations ((extended) rendezvous) in form of:

- **entry points** in tasks
- **full set of parameter profiles** supported

If the local and the remote task are on different architectures,
or if an intermediate communication system is employed:

☞ parameters incl. bounds and discriminants are 'tunnelled' through byte-stream-formats.

Synchronization:

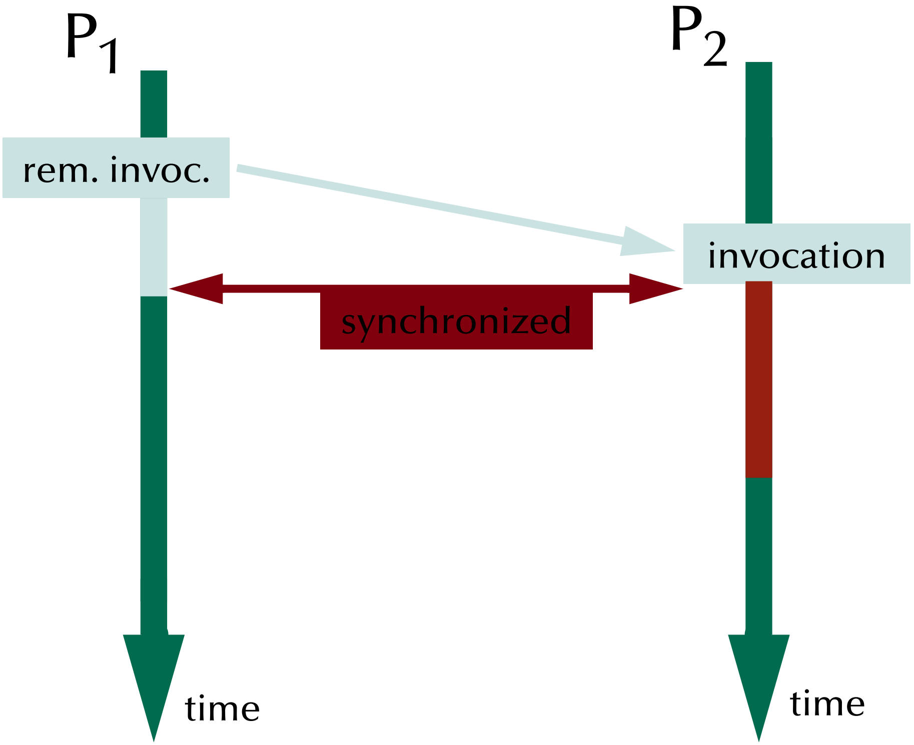
- both tasks are synchronized at the beginning of the remote invocation (☞ '**rendezvous**')
- the calling task if blocked until the remote routine is completed (☞ '**extended rendezvous**')

Message-based synchronization in Ada95

Remote invocation
(Rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver started an addressed routine

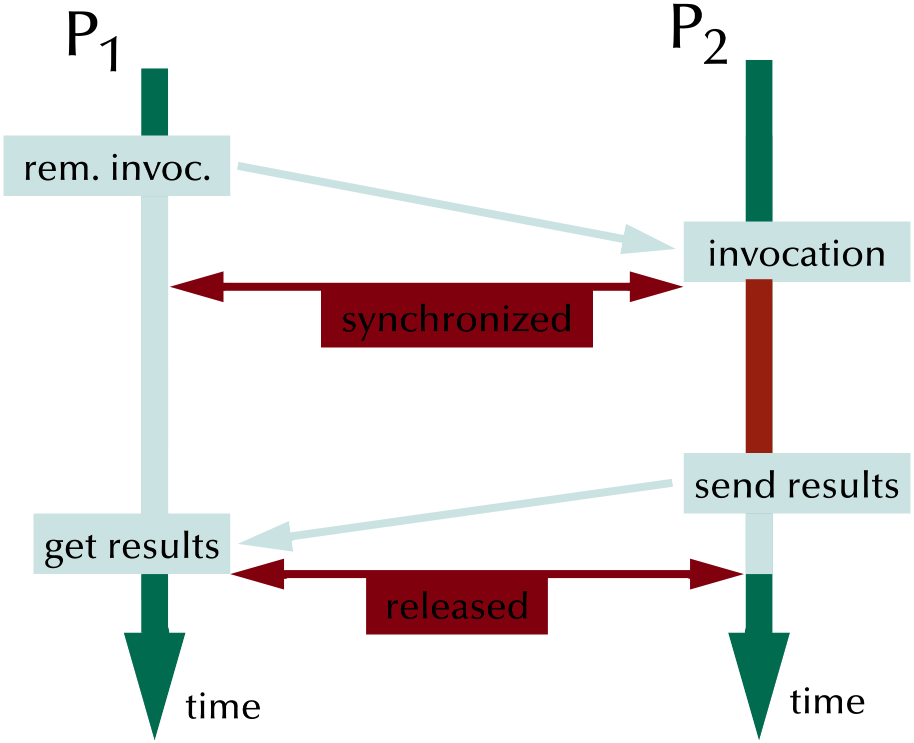


Message-based synchronization in Ada95

Remote invocation
(Extended rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results



Message-based synchronization in Ada95

(Rendezvous)

```
…
<entry_name> [(index)] <parameters>
… -- waiting for synchronization
… --
… --
… --
…
…
…
…
…
```

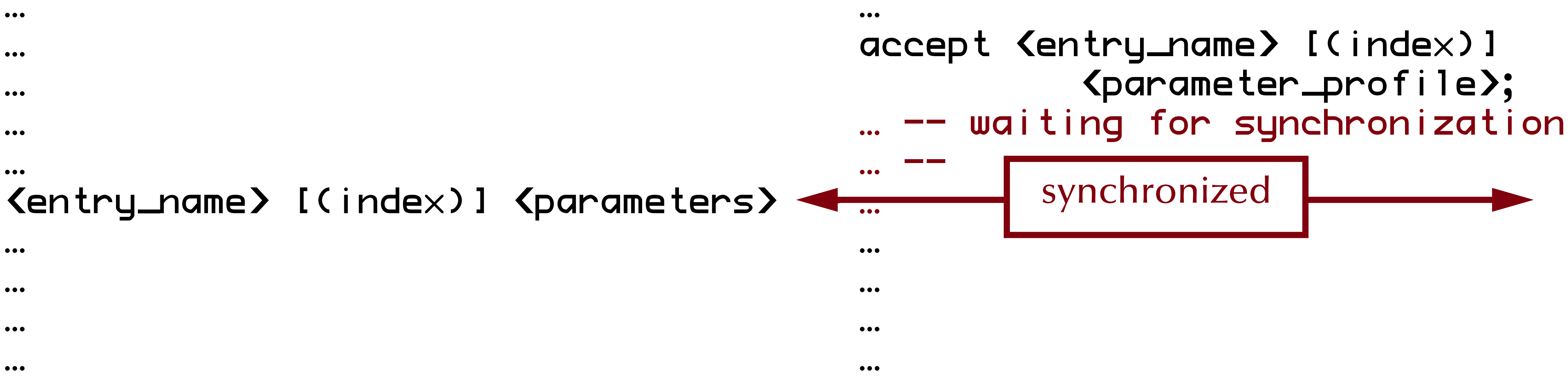
synchronized

```
…
…
…
…
…
accept <entry_name> [(index)]
          <parameter_profile>;
…
…
…
…
…
```

Synchronization

Message-based synchronization in Ada95

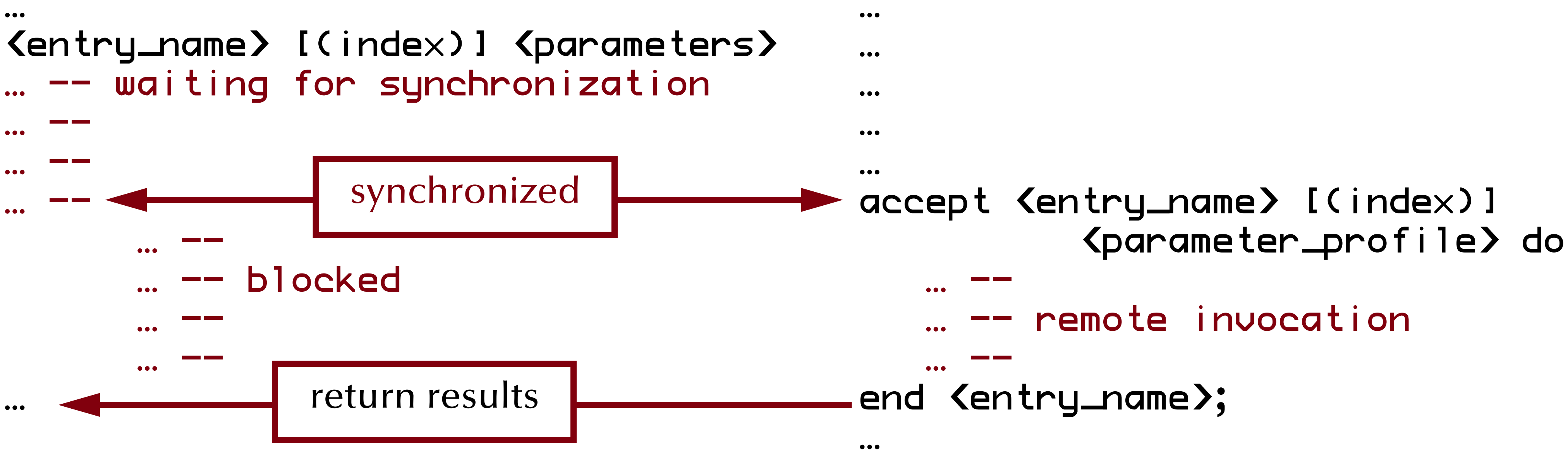
(Rendezvous)



```
...                                        ...
...                                        accept <entry_name> [(index)]
...                                                   <parameter_profile>;
...                                        ... -- waiting for synchronization
...                                        ... --
<entry_name> [(index)] <parameters>        ...
...                                        ...
...                                        ...
...                                        ...
...                                        ...
```

Message-based synchronization in Ada95

(Extended rendezvous)

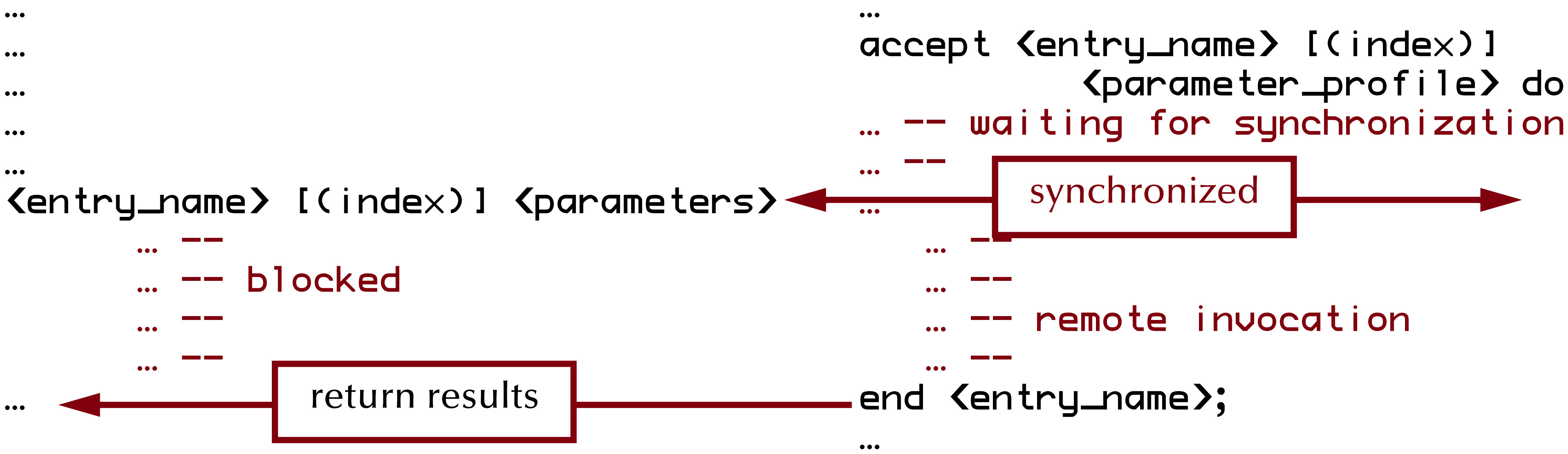


```
…
<entry_name> [(index)] <parameters>
… -- waiting for synchronization
… --
… --
… --
      … --
      … -- blocked
      … --
      … --
…
```

```
…
…
…
…
…
accept <entry_name> [(index)]
           <parameter_profile> do
   … --
   … -- remote invocation
   … --
end <entry_name>;
…
```

Message-based synchronization in Ada95

(Extended rendezvous)



```
…
…
…
…
…
<entry_name> [(index)] <parameters>
          … --
          … -- blocked
          … --
          … --
…
```

```
…
accept <entry_name> [(index)]
                <parameter_profile> do
… -- waiting for synchronization
… --
…
     … --
     … --
     … -- remote invocation
     … --
end <entry_name>;
…
```

Message-based synchronization in Ada95

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entries *can* call other blocking operations.
- Accept statements can be nested (but need to be different).

  ☞ helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).

  Exceptions, which are not handled during the rendezvous phase
  are propagated to *all* involved tasks.
- Parameters cannot be direct '`access`' parameters, but can be access-types.

Message-based synchronization in Ada95

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entries *can* call other blocking operations.
- Accept statements can be nested (but need to be different).

  ☞ helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).

  Exceptions, which are not handled during the rendezvous phase
  are propagated to *all* involved tasks.
- Parameters cannot be direct '`access`' parameters, but can be access-types.
- `'count` on task-entries is defined, but is only accessible from inside the tasks owning the entry.
- **Entry families** (arrays of entries) are supported.
- **Private entries** (accessible for internal tasks) are supported.

Selective waiting

Dijkstra's guarded commands:

```
if x <= y -> m := x
□  x >= y -> m := y
fi
```

selection is non-deterministic!

- ☞ the programmer needs to design the alternatives as 'parallel' options: all cases need to be covered and overlapping conditions need to lead to the same result

Extremely different philosophy: 'C'-switch:

```
switch (x) {
   case 1: r := 3;
   case 2: r := 2; break;
   case 3: r := 1;
}
```

- ☞ the sequence of alternatives has a crucial role.

Message-based selective synchronization in Ada95

Forms of selective waiting:

```
select_statement ::= selective_accept         |
                     conditional_entry_call |
                     timed_entry_call         |
                     asynchronous_select
```

... underlying concept: Dijkstra's guarded commands

`selective_accept` implements ...

- ... wait for more than a single rendezvous at any one time
- ... time-out if no rendezvous is forthcoming within a specified time
- ... withdraw its offer to communicate if no rendezvous is available immediately
- ... terminate if no clients can possibly call its entries

Message-based selective synchronization in Ada95

`selective_accept` in its full syntactical form in Ada95:

```
selective_accept ::= select
                           [guard] selective_accept_alternative
                      { or [guard] selective_accept_alternative
                      [ else sequence_of_statements ]
                     end select;

guard ::= when <condition> =>

selective_accept_alternative ::= accept_alternative    |
                                 delay_alternative     |
                                 terminate_alternative

accept_alternative    ::= accept_statement [ sequence_of_statements ]
delay_alternative     ::= delay_statement [ sequence_of_statements ]
terminate_alternative ::= terminate;
```

Basic forms of selective synchronization

(select-or)

```
select
   accept … do …
   end …
or
   accept … do …
   end …
or
   accept … do …
   end …
or
   accept … do …
   end …
…
end select;
```

- If none of the named entries have been called, the task is suspended until one of the entries is addressed by another task.
- The selection of an accept is non-deterministic, in case that multiple entries are called.

☞ The selection can be controlled by means of the real-time systems annex.

- The select statement is completed, when at least one of the entries has been called and those accept-block has been executed.

Basic forms of selective synchronization

(guarded select-or)

```
select
   when <condition> =>
      accept … do …
      end …
or
   when <condition> =>
      accept … do …
      end …
or
   when <condition> =>
      accept … do …
      end …
…
end select;
```

- Analogue to Dijkstra's guarded commands
- all accepts closed will raise a Program_Error

☞ set of conditions need to be complete

Basic forms of selective synchronization

(guarded select-or-else)

```
select
   [ when <condition> => ]
      accept … do …
      end …
or
   [ when <condition> => ]
      accept … do …
      end …
or
   [ when <condition> => ]
      accept … do …
      end …
else
   <statements>
…
end select;
```

- If none of the open entries can be accepted immediately, the else alternative is selected.
- There can be only one else alternative and it cannot be guarded.

Basic forms of selective synchronization

(guarded select-or-delay)

```
select
    [ when <condition> => ]
        accept … do …
        end …
or
    [ when <condition> => ]
        delay …
        <statements>
or
    [ when <condition> => ]
        delay …
        <statements>
…
end select;
```

- If none of the open entries has been called before the amount of time specified in the earliest open delay alternative, this delay alternative is selected.
- There can be multiple delay alternatives if more than one delay alternative expires simultaneously, either one may be chosen.
- `delay` and `delay until` can be employed.

Basic forms of selective synchronization

(guarded select-or-terminate)

```
select
    [ when <condition> => ]
        accept … do …
        end …
or
    [ when <condition> => ]
        accept … do …
        end …
or
    [ when <condition> => ]
        terminate;
…
end select;
```

The terminate alternative is chosen if none of the entries can ever be called again, i.e.:

- all tasks which can possibly call any of the named entries are terminated.

or

- all remaining active tasks which can possibly call any of the named entries are waiting on selective terminate statements and none of their open entries can be called any longer.

☞ This task and all its dependent waiting-for-termination tasks are terminated together.

Basic forms of selective synchronization

(guarded select-or-else select-or-delay select-or-terminate)

```
select

or

else
   <statements>
...
end select;
```

`else-delay-terminate`
alternatives
cannot be mixed!

```
select
   [ when <condition> => ]
      accept ... do ...
      end ...
or
   [ when <condition> => ]
      delay ...
      <statements>
...
end select;

select
   [ when <condition> => ]
      accept ... do ...
      end ...
or
   [ when <condition> => ]
      terminate;
...
end select;
```

Non-determinism in selective synchronizations

☞ If equal alternatives are given, then the program correctness (incl. the timing specifications) must not be affected by the actual selection.

- If alternatives have different priorities,
  this can be expressed e.g. by means of the Ada real-time annex.

- Non-determinism in concurrent systems is or can be also introduced by:
    - non-ordered monitor or other queues
    - buffering / routing message passing systems
    - non-deterministic schedulers
    - timer quantization
    - … any form of asynchronism

Conditional & timed entry-calls

```
conditional_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   else
      sequence_of_statements
   end select;
```

```
select
   Light_Monitor.Wait_for_Light;
   Lux := True;
else
   Lux := False;
end;
```

```
timed_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   or
      delay_alternative
   end select;
```

```
select
   Controller.Request (Medium)
      (Some_Item);
   -- process data
or
   delay 45.0;
   -- try something else
end select;
```

Conditional & timed entry-calls

```
conditional_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   else
      sequence_of_statements
   end select;
```

```
timed_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   or
      delay_alternative
   end select;
```

```
select
   Light_Monitor.Wait_for
   Lux := True;
else
   Lux := False;
end;
```

```
ler.Request (Medium)
e_Item);
ess data

5.0;
something else
end select;
```

There is only

one entry call

and either

one `‘else‘`

or

one `‘or delay’`

Conditional & timed entry-calls

```
conditional_entry_call ::=
   select
      entry_call_statement
      [sequ
   else
      seque
   end sele
```

```
timed_entry_call ::=
   select
      entry_call_statement
      [sequence of statements]
```

The idea in both cases is to ***withdraw a synchronization request*** and *not* to implement polling or busy-waiting.

```
select
   Light_Monitor.Wait_for_Light;
   Lux := True;
else
   Lux := False;
end;
```

```
select
   Controller.Request (Medium)
      (Some_Item);
   -- process data
or
   delay 45.0;
   -- try something else
end select;
```

Synchronization

- **Shared memory based synchronization**
    - Flags, condition variables, semaphores, …
      … conditional critical regions, monitors, protected objects.
    - Guard evaluation times, nested monitor calls, deadlocks, …
      … simultaneous reading, queue management.
    - Synchronization and object orientation, blocking operations and re-queuing.
- **Message based synchronization**
    - Synchronization models, addressing modes, message structures
    - Selective accepts, selective calls
    - Indeterminism in message based synchronization

Deadlocks

Synchronization may lead to

☞ DEADLOCKS

… a closer look on deadlocks
and what can be done about them …

Reserving resources in reverse order

```
var reserve_1, reserve_2: semaphore := 1;
```

```
process P1;                              process P2;
   statement X;                             statement A;

   wait (reserve_1);                        wait (reserve_2);
   wait (reserve_2);                        wait (reserve_1);
      statement Y; - employ resources          statement B; - employ resources
   signal (reserve_2);                      signal (reserve_1);
   signal (reserve_1);                      signal (reserve_2);

   statement Z;                             statement C;
end P1;                                  end P2;
```

Sequence of operations : $[A \mid X] \Rightarrow \{[B \Rightarrow Y] \text{ xor } [Y \Rightarrow B]\} \Rightarrow [C \mid Z]$
or : $[A \mid X] \Rightarrow$ `deadlocked!`

Circular dependencies

```
var reserve_1, reserve_2, reserve_3: semaphore := 1;

process P1;
   statement X;

   wait (reserve_1);
   wait (reserve_2);
      statement Y;
   signal (reserve_2);
   signal (reserve_1);

   statement Z;
end P1;

process P2;
   statement A;

   wait (reserve_2);
   wait (reserve_3);
      statement B;
   signal (reserve_3);
   signal (reserve_2);

   statement C;
end P2;

process P3;
   statement K;

   wait (reserve_3);
   wait (reserve_1);
      statement L;
   signal (reserve_1);
   signal (reserve_3);

   statement M;
end P3;
```

Sequence of operations : [A | X | K] ➠ {[B ➠ Y ➠ L] xor …} ➠ [C | Z | M]
or : [A | X | K] ➠ `deadlocked!`

Necessary deadlock conditions:

1. **Mutual exclusion**:
   resources cannot be used simultaneously
2. **Hold and wait**:
   a process applies for a resource, while it is holding another resource (sequential requests)
3. **No pre-emption**:
   resources cannot be pre-empted; only the process itself can release resources
4. **Circular wait**:
   a ring list of processes exists, where every process waits for release of a resource by the next one

☞ system may be deadlocked, when ***all*** these conditions apply!

Deadlock strategies:

1. Ignorance
   - ☞ Kill unresponsive processes

2. Deadlock detection & recovery
   - ☞ find deadlocked processes and recover the system in a coordinated way

3. Deadlock avoidance
   - ☞ the resulting system state is checked before any resources are actually assigned

4. Deadlock prevention
   - ☞ the system prevents deadlocks by its structure

Deadlock prevention

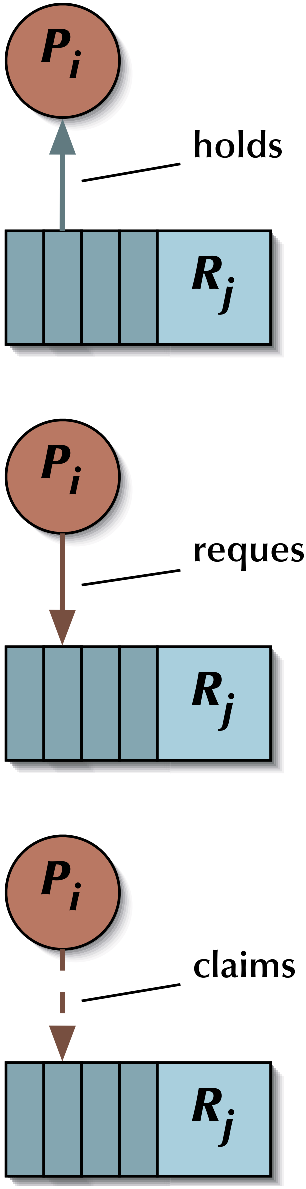
(remove one of the four deadlock conditions)

1. **Mutual exclusion**:
   Applicable to specific cases only; usually this can only be removed by replication of resources.
2. **Hold and wait**:
   Processes are forced to allocate all their required resources at once,
   often at the time of admittance to the main dispatcher – done in many static realtime-systems.
3. **No pre-emption**:
   If the current state of a resource can be stored and restored easily, then they can be pre-empted.
   Usually resources are pre-empted from processes, which are currently not ready to run.
4. **Circular wait**:
   A circular wait can be avoided by a global ordering of all resources, e.g. resources can only be requested in a specific order – hard to maintain in a dynamic system configuration.

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



$RAG = \{V, E\}$; vertices and edges

$V = P \cup R$; vertices are processes or resource types:

$P = \{P_1, P_2, \ldots, P_n\}$; processes

$R = \{R_1, R_2, \ldots R_k\}$; resource types

$E = E_r \cup E_a \cup E_c$; claims, requests and assignments

$E_c = \{P_i \rightarrow R_j, \ldots\}$; claims

$E_r = \{P_i \rightarrow R_j, \ldots\}$; requests

$E_a = \{R_i \rightarrow P_j, \ldots\}$; assignments

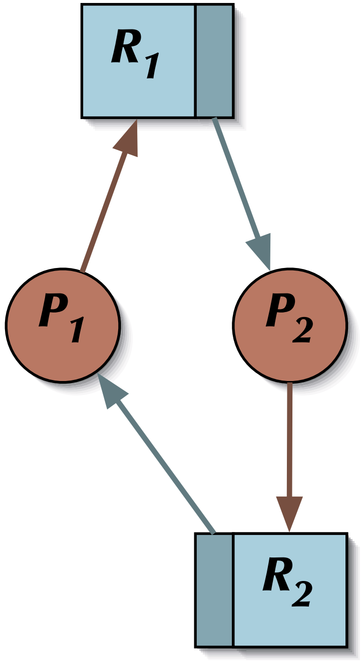
Note: a resourcefully may have more than one instance

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

the two process, reverse allocation deadlock:

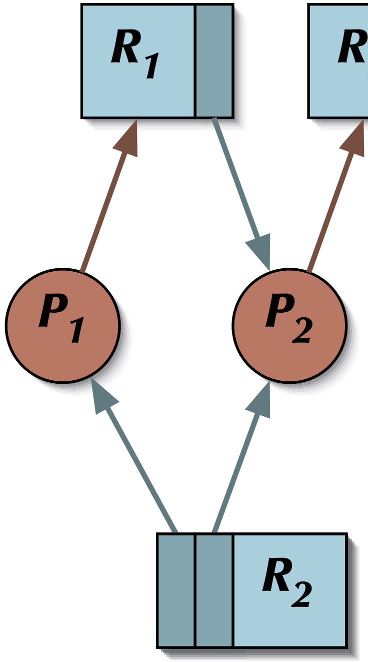


Deadlocks

Resource Allocation Graphs

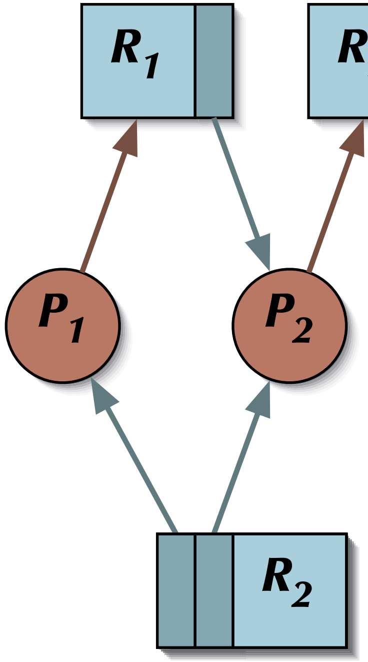
(Silberschatz, Galvin & Gagne)

Is this a deadlock situation? ☞



Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



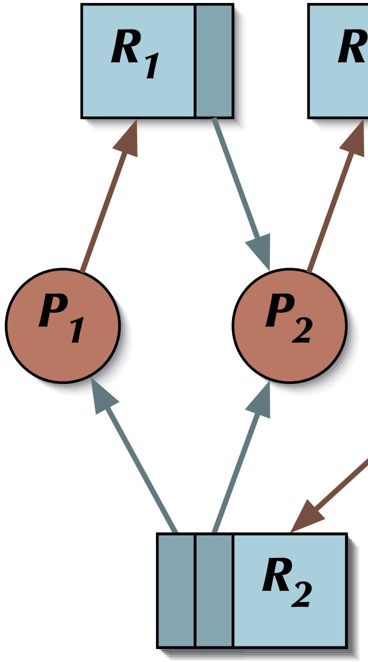
no, there is no circular dependency

Deadlocks

Resource Allocation Graphs

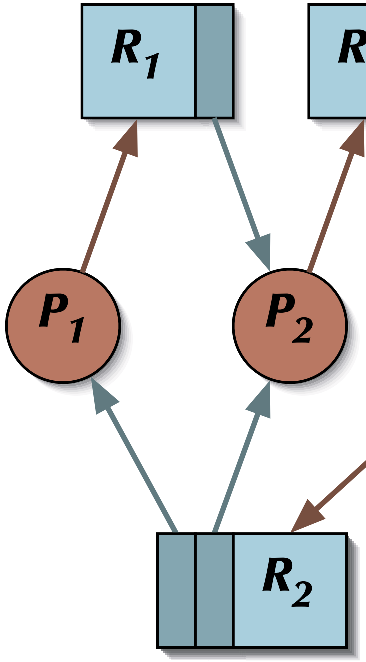
(Silberschatz, Galvin & Gagne)

Is this a deadlock situation? ☞



Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



yes, there are circular dependencies:

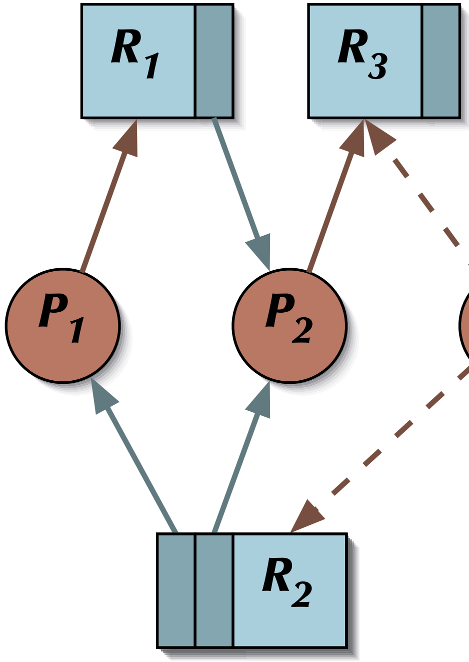
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

as well as: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

☞ ***IF some processes are deadlocked, THEN there are cycles in the resource allocation graph***

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

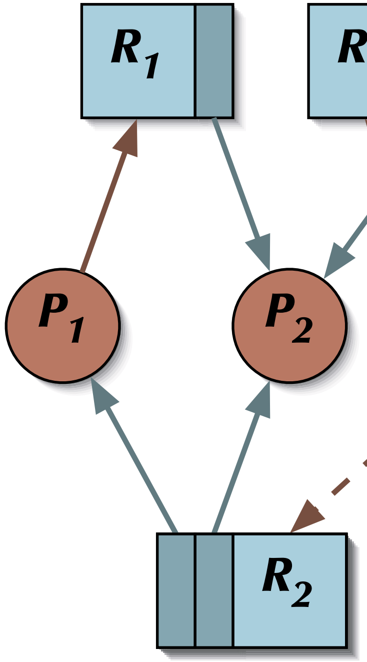


Assuming all claims of P_3 are known in advance,

☞ Could the deadlock situation be avoided?

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



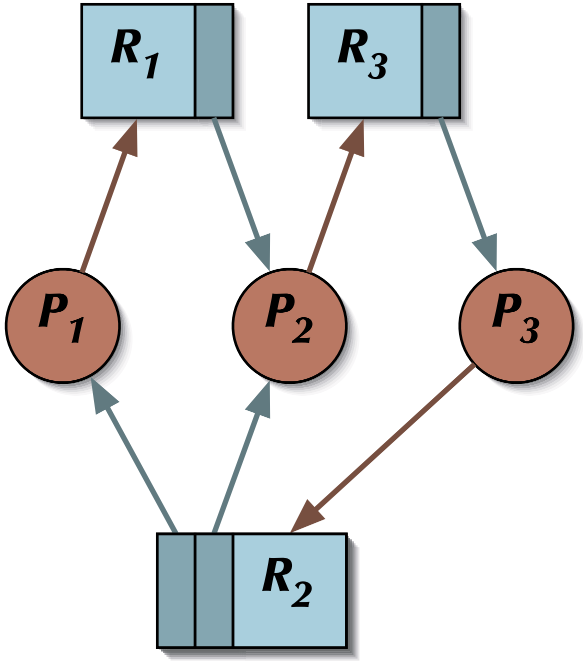
yes, when resources are assigned so that there are no resulting circular dependencies:

☞ in this case: assign R_3 to P_2 (instead of P_3)

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



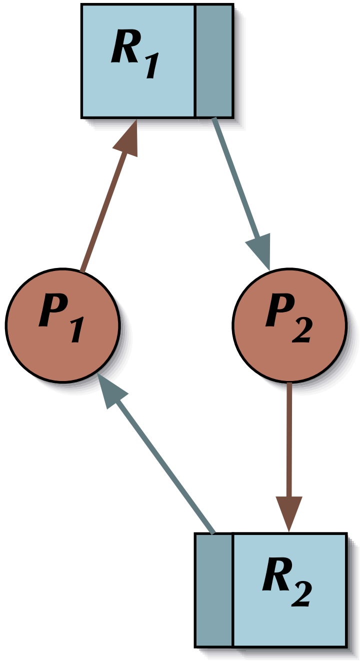
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

as well as: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

☞ ***ARE some processes deadlocked, IF there are cycles in the resource allocation graph?***

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

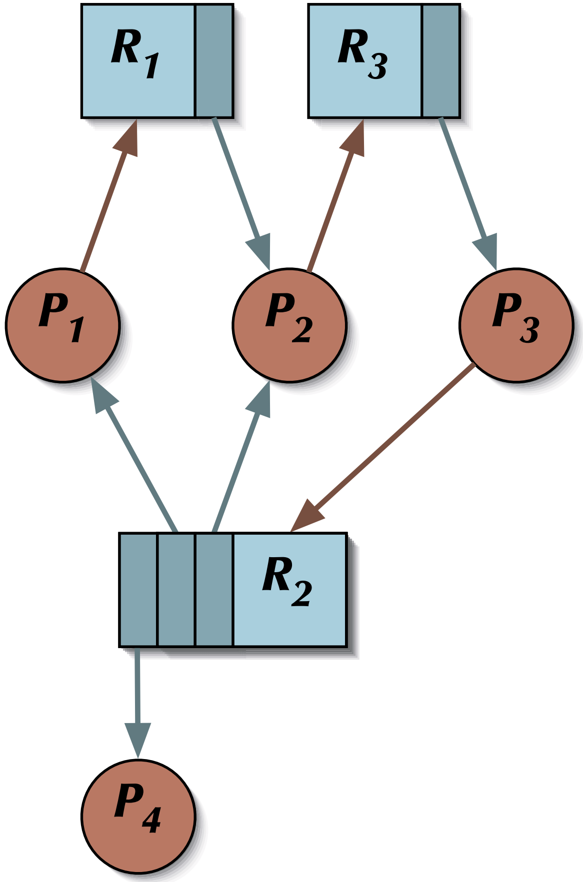


yes,
if there is only one instance per resource type:

☞ ***IF there are cycles in the resource allocation graph AND there is only one instance per resource type, THEN some processes are deadlocked!***

Resource Allocation Graphs

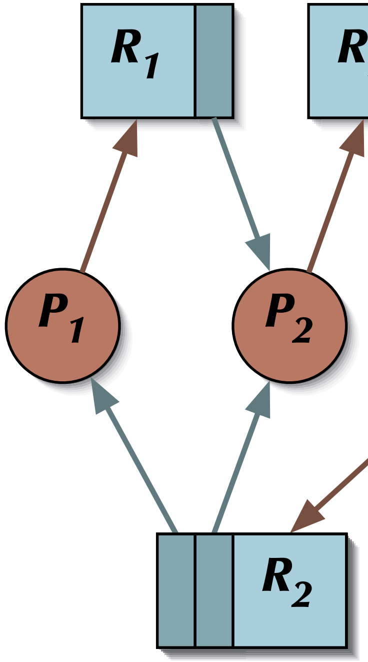
(Silberschatz, Galvin & Gagne)



no,
if there is more than one instance
per resource type:

☞ ***IF there are cycles in the resource allocation graph AND there is more than one instance per resource type, THEN some processes may be deadlocked!***

Deadlocks



How to detect deadlocks in the general case?

(of multiple instances per resource)

Banker's algorithm

There are n processes and m resource types in the system. Let $i \in 1 \ldots n$ and $j \in 1 \ldots m$:

- $Allocated[i, j]$
  ☞ the number of resources of type j allocated by process i.
- $Free[j]$
  ☞ the number of available resources of type j.
- $Claimed[i, j]$
  ☞ the number of resources of type j required by process i to complete *eventually*.
- $Request[i, j]$
  ☞ the number of *currently* requested resources of type j by process i.

Temporary variables:

- $Completed[i]$: boolean vector indicating processes, which may complete right now.
- $Simulated\_Free[j]$: available resources, if some processes complete and de-allocate.

Banker's algorithm

Checking for a deadlock situation

```
1. $Simulated\_Free \Leftarrow Free;\ \forall i: Completed[i] \Leftarrow False$

2. While $\exists i: \neg Completed[i]$
     and $\forall j: Requested[i, j] < Simulated\_Free[j]$ do: {request i can be granted}

         $\forall j: Simulated\_Free[j] \Leftarrow Simulated\_Free[j] + Allocated[i, j]$
         $Completed[i] \Leftarrow True$

3. If $\forall i: Completed[i]$ then the system is deadlock-free!
   (otherwise all processes i with $Completed[i] = False$ are deadlocked)
```

Banker's algorithm

Checking the current system state

```
1. $Simulated\_Free \Leftarrow Free;\ \forall i: Completed[i] \Leftarrow False$

2. While $\exists i: \neg Completed[i]$
       and $\forall j: Claimed[i, j] < Simulated\_Free[j]$ do: {meaning process i can complete}

           $\forall j: Simulated\_Free[j] \Leftarrow Simulated\_Free[j] + Allocated[i, j]$
           $Completed[i] \Leftarrow True$

3. If $\forall i: Completed[i]$ then the system is safe!
   (e.g. no process is currently deadlocked and no process can be deadlocked in any future state)
```

Banker's algorithm

Checking the validity of a resource request

```
If (Request < Claimed) and (Request < Free) then
   Free      := Free      - Request;
   Claimed   := Claimed   - Request;
   Allocated := Allocated + Request;
   ☞ Apply system state check (as above)
   If System_is_safe then
      ☞ Actually grant request
   else
      -- restore former system state (Free, Claimed, Allocated)
   end if;
end if;
```

Deadlock recovery

☞ Stop or restart one or multiple of the deadlocked processes and reclaim its resources

☞ Pre-empt one of the involved resources (and restore an earlier state of the victim process)

Deadlock recovery does not deal with the source of the problem!

(the system may deadlock again right away)

☞ use deadlock prevention or deadlock avoidance instead

Deadlocks

- **Ignorance & recovery**
  - ☞ ‘kill some seemingly persistently blocked processes from time to time’ (exasperation)
- **Deadlock detection & recovery**
  - ☞ multiple methods for detection, e.g. resource allocation graphs, Banker’s algorithm
  - ☞ recovery is mostly ‘ugly’
- **Deadlock avoidance**
  - ☞ check system safety before allocating resources, e.g. Banker’s algorithm
- **Deadlock prevention**
  - ☞ eliminate one of the pre-conditions for deadlocks

Purpose of scheduling

A scheduling scheme provides two features:

- Ordering the use of resources (e.g. CPUs, networks)
- Predicting the worst-case behaviour of the system when the scheduling algorithm is applied

  … in case that some or all information about the expected resource requests are known

A prediction can then be used

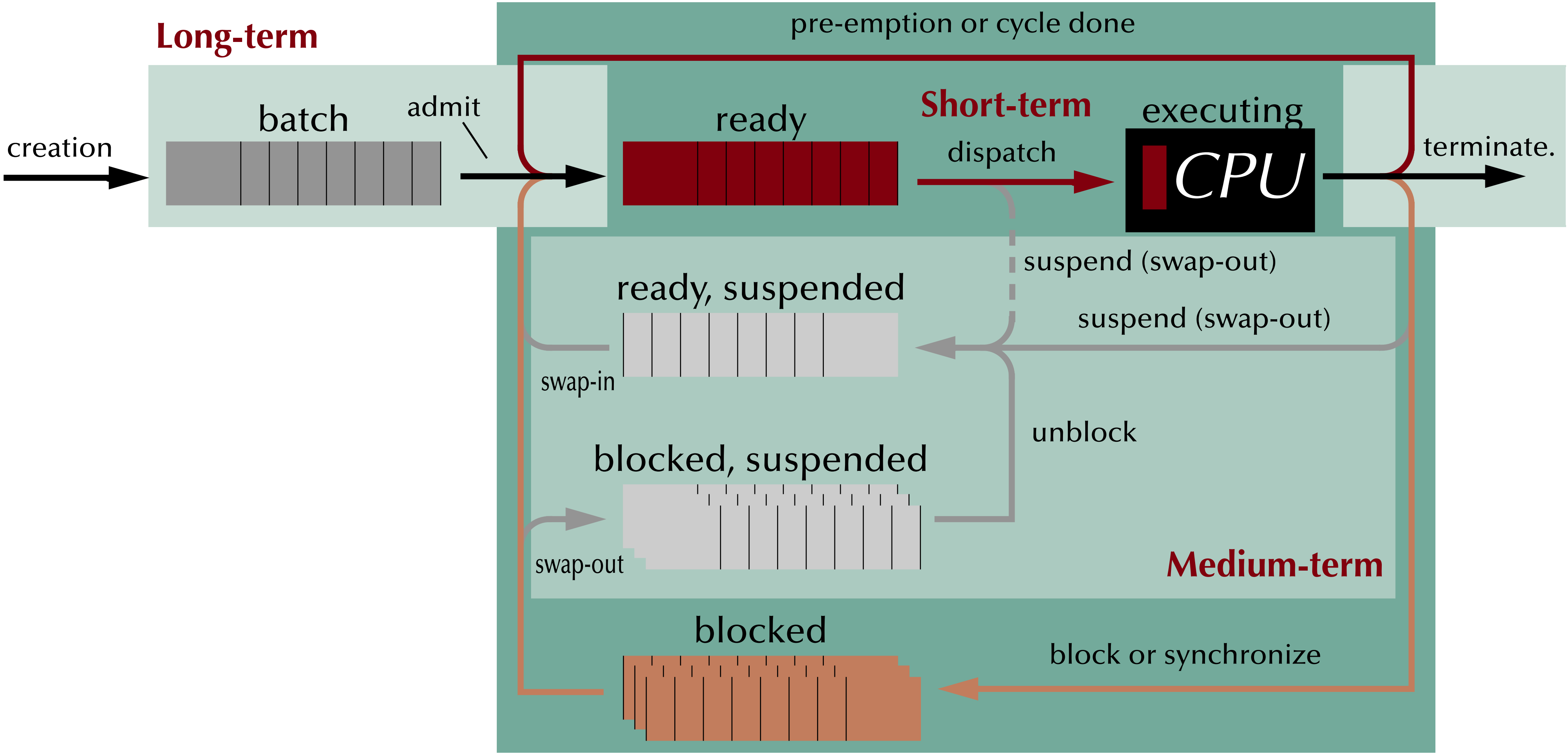
☞ at compile-run: to confirm the overall resource requirements of the application, or

☞ at run-time: to permit acceptance of additional usage/reservation requests.

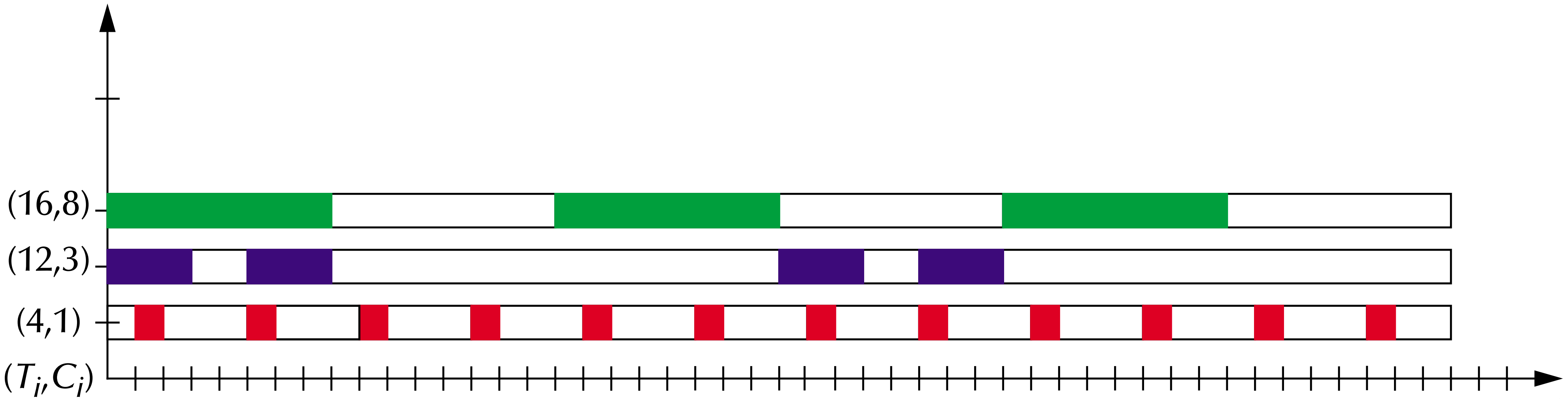
Criteria for scheduling methods

	Performance criteria: minimize the …	**Predictability criteria:** minimize the diversion from given
Process / user perspective:		
Waiting time	maximum / average / variance	minimal and maximal waiting times
Response time	maximum / average / variance	minimal and maximal response times
Turnaround time	maximum / average / variance	deadlines
System perspective:		
Throughput	maximum / average / variance of CPU time per process	–
Utilization	CPU idle time	–

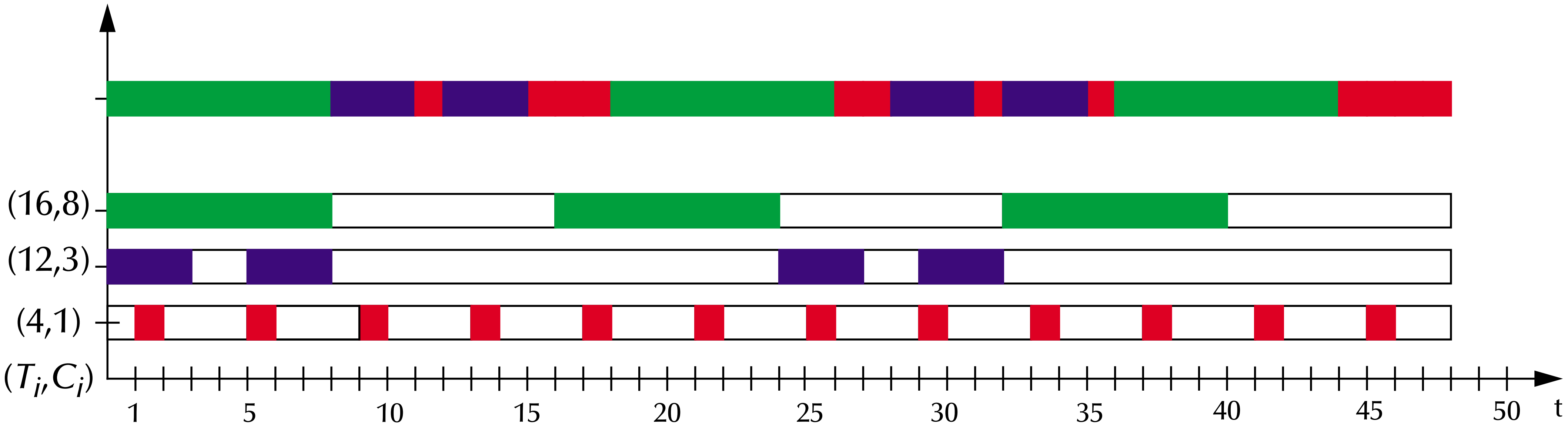
Time scales of scheduling



Example: Requested times

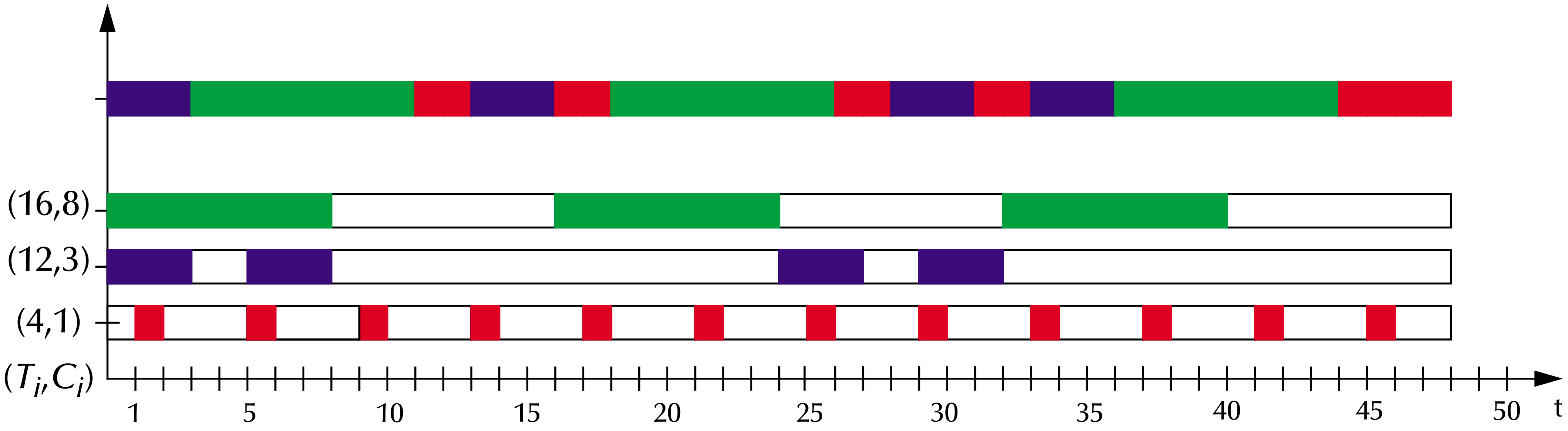


First come, first served (FCFS) *– bad case: (arrival order: 🟩, 🟪, 🟥)*



Waiting time: 0…11; average: **5.95** – **Turnaround time: 3…12**; average: **8.47**

First come, first served (FCFS) – nice case: (arrival order: 🟥 , 🟪 , 🟩)

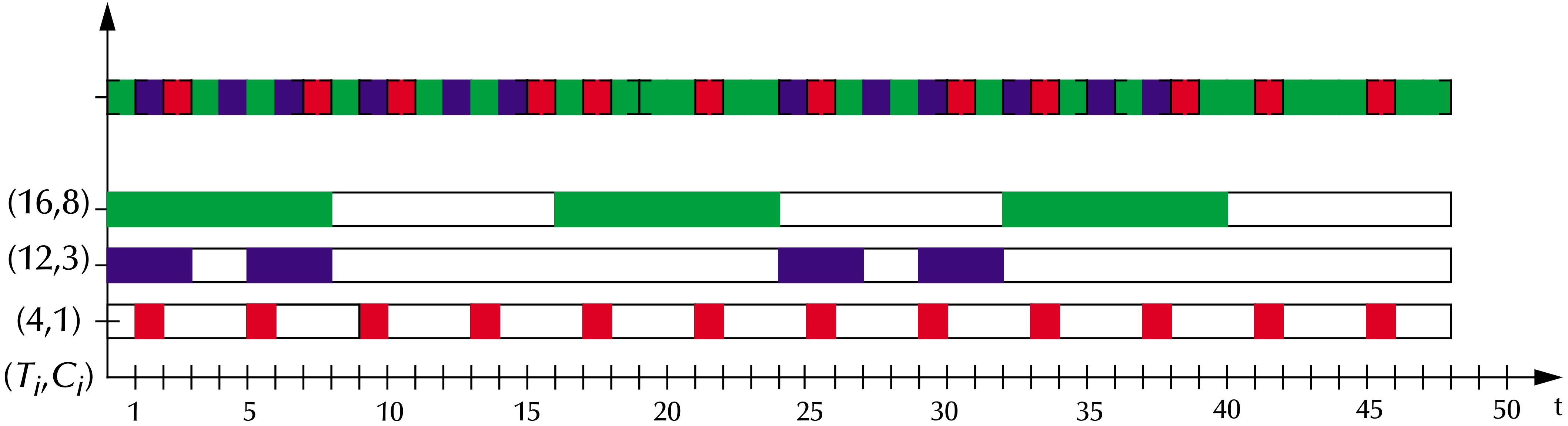


Waiting time: 0…11; average: **5.47** – **Turnaround time: 3…12**; average: **8.00**

☞ The actual average waiting time for FCFS may vary here between: **5.47** and **5.95**

Scheduling

Round robin (RR) – *pre-emption*



Waiting time: 0…**4**; average: **1.21** – **Turnaround time: 1**…**19**; average: **5.63**

☞ *Waiting* and *average turnaround time* is going down, but *maximal turnaround* time is going up

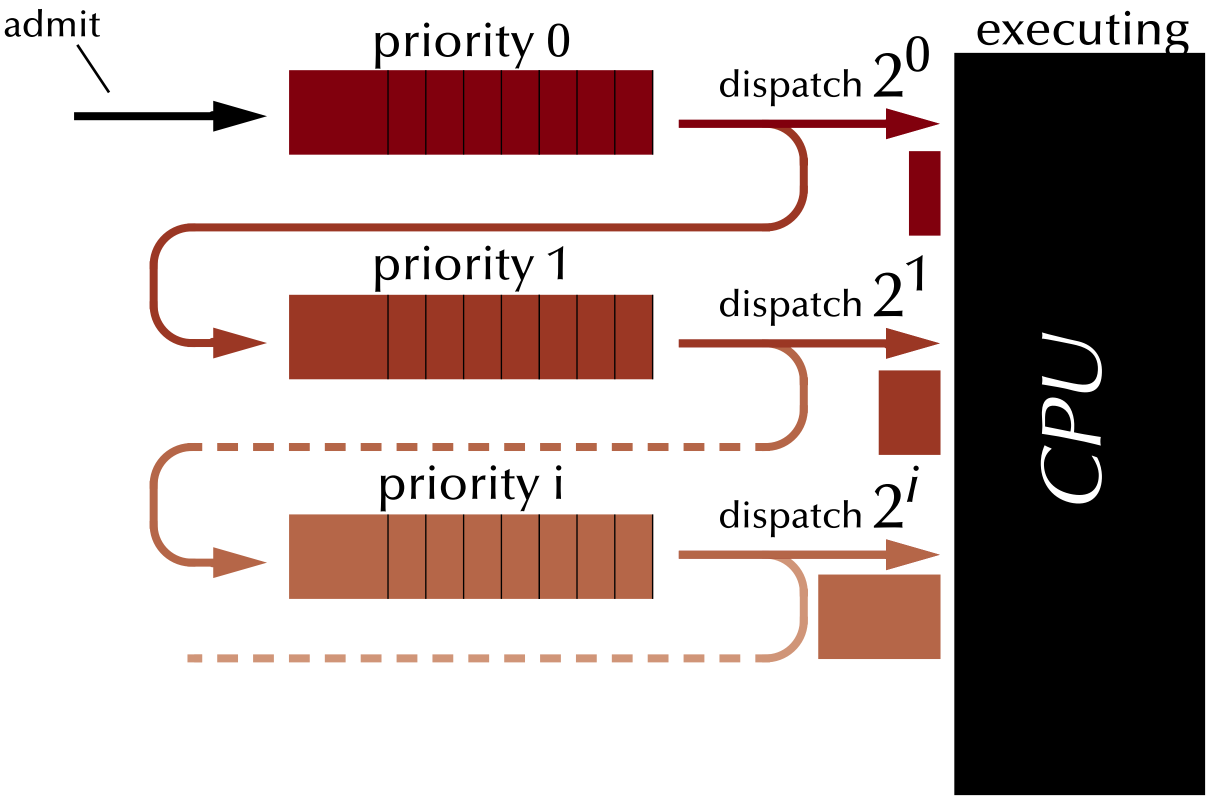
… assuming that task-switching is free and always possible

Feedback with 2^i pre-emption intervals – *pre-emption*

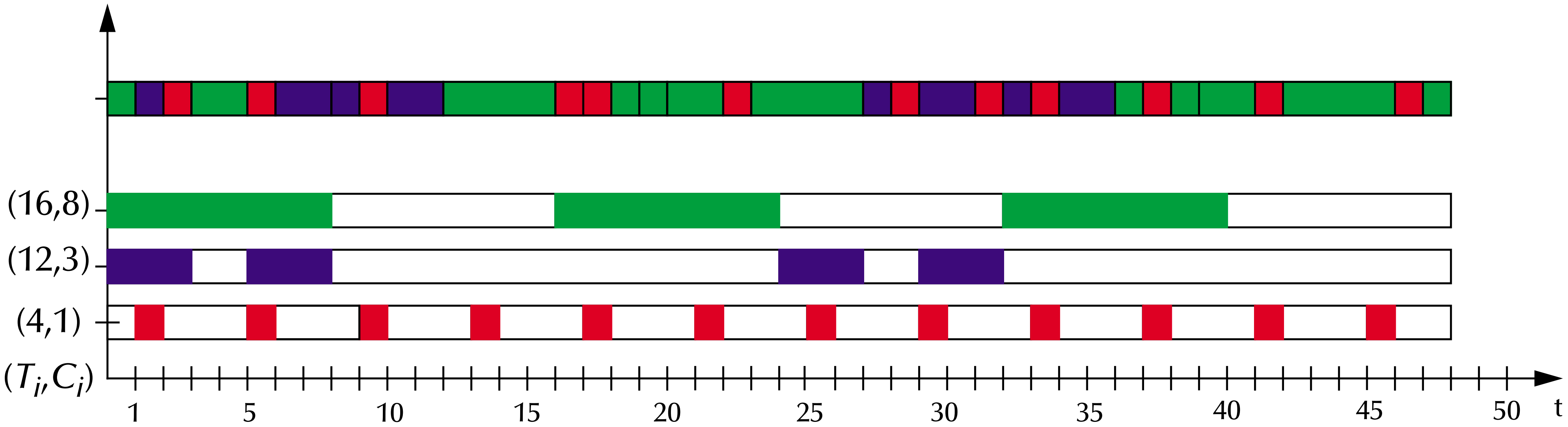
- implement multiple hierarchical ready-queues
- fetch processes from the highest filled ready queue
- dispatch more CPU time for lower priorities (2^i units)

☞ processes on lower ranks may suffer starvation

☞ new and short tasks will be preferred



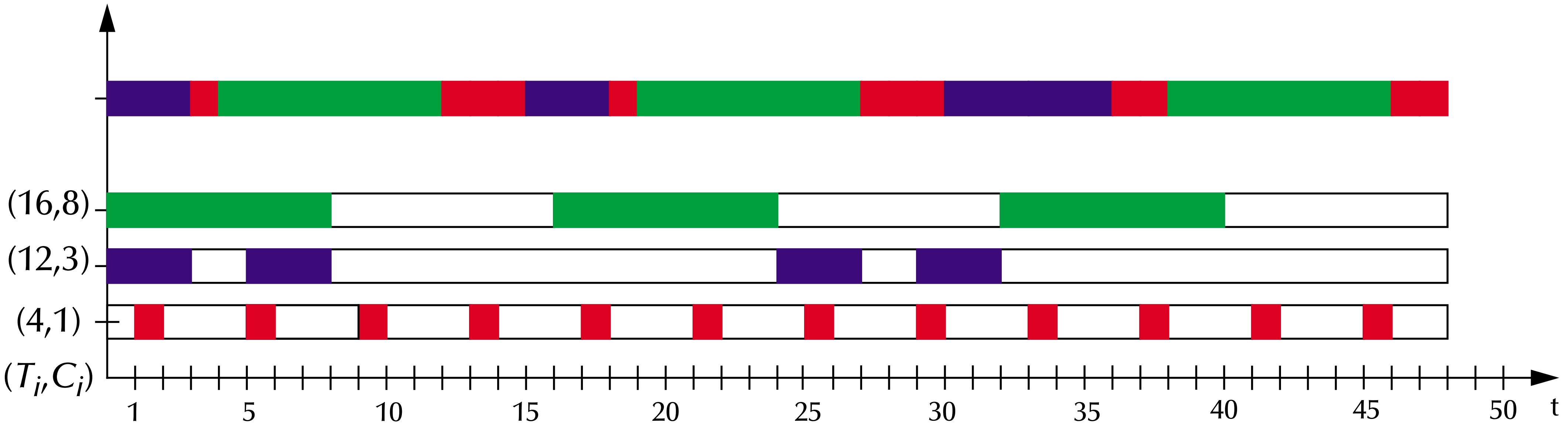
Feedback with 2^i pre-emption intervals – *pre-emption*



Waiting time: 0…**6**; average: **1.79** – **Turnaround time:** 1…**21**; average **5.63**

☞ *less task switches* than RR,
but long processes can suffer starvation!

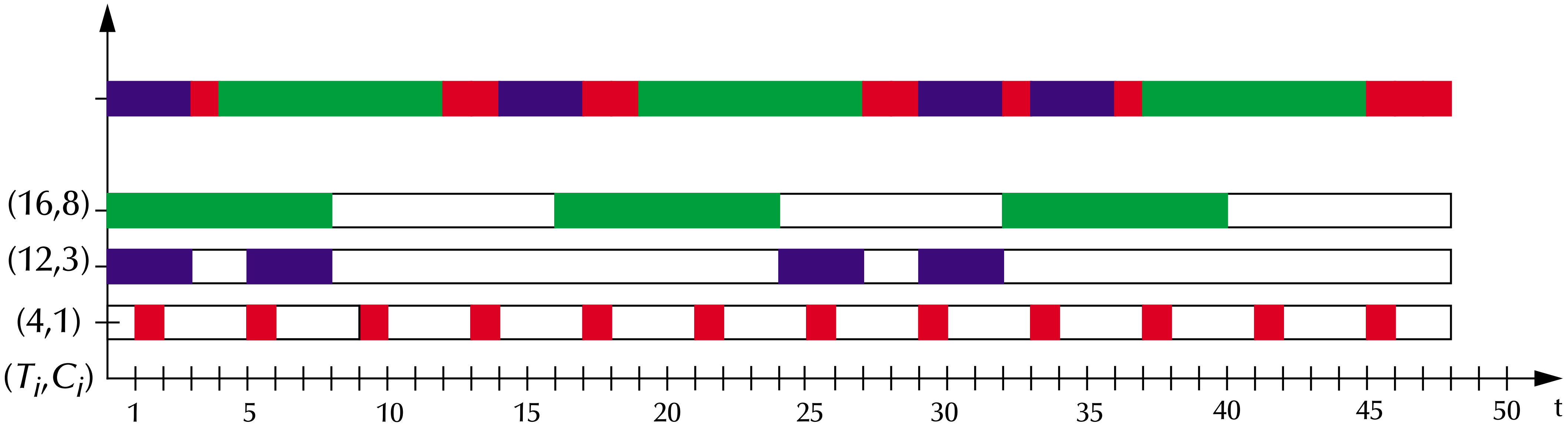
Shortest job first (SJF) – C_i is known



Waiting time: 0…10; average: **3.47** – **Turnaround time: 1…14**; average: **6.00**

☞ *on average* this is doing better than FCFS

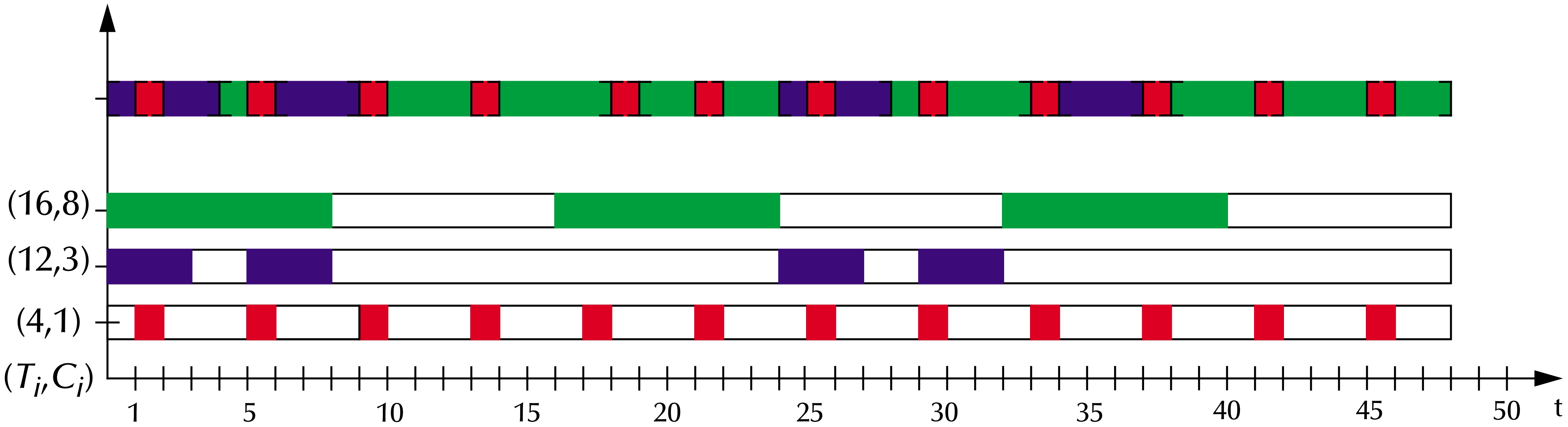
Highest response ratio first (HRRF) – C_i is known



Response ratio: $(W_i + C_i)/C_i$ – **Waiting time**: 0…**9**; average: **4.11** – **Turnaround time**: 1…**13**; average **6.63**

☞ *on average* this is doing worse than SJF,
but the *maximal* waiting and turnaround times and variance might be reduced!

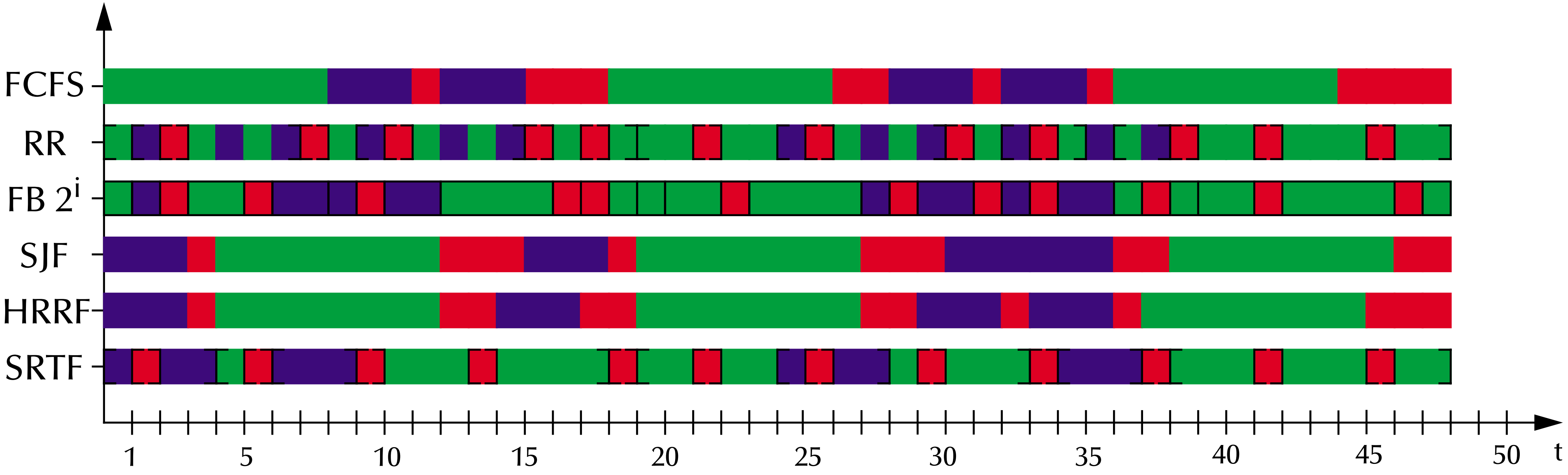
Shortest remaining time first (SRTF) – C_i is known + pre-emption



Waiting time: 0…**6**; average: **1.05** – **Turnaround time:** 1…**18**; average **4.42**

☞ *on average* this is doing better than FCFS, SJF or HRRF,
but the *maximal turnaround* time is going up and there are many task-switches!

Non-realtime scheduling methods



- ☞ CPU utilization: 100% in all cases.
- ☞ Pre-emptive methods perform better, assuming that the overhead is negligible.
- ☞ Knowledge of C_i (computation times) has a significant impact on scheduler performance.

Scheduling

	Selection	Pre-emption	Waiting	Turnaround	Preferred processes	Starvation possible?
			in high load situations			
FCFS	$max(W_i)$	no	possibly long	possibly long	long	no
RR	equal share	yes	bound	possibly long	none	no
Feedback	priority queues	yes	short on average	very short on average, large maximum	short	yes
SJF	$min(C_i)$	no	short on average	short on average	short	yes
HRRF	$max\left(\frac{W_i + C_i}{C_i}\right)$	no	short on average, lower variance	short on average, lower variance	balanced, towards short	no
SRTF	$min(C_i - E_i)$	yes	very short on average	very short on average, large maximum	short	yes

Towards predictable scheduling …

☞ Task behaviours are more specified (restricted).

☞ Task requirements from the operating systems are more specific.

☞ Task sets are often fully or mostly static.

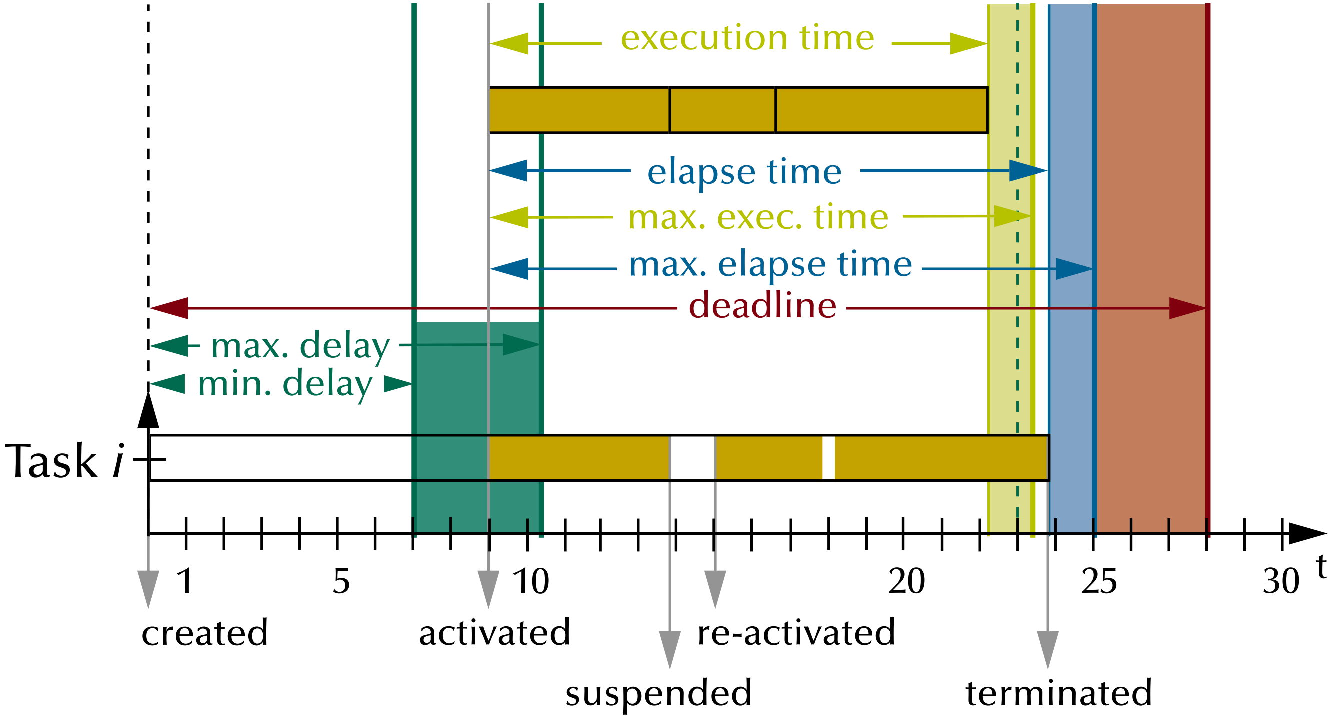
☞ Sporadic and urgent requests (e.g. user interaction, alarms) need to be addressed.

¬ CPU-utilization and throughput (system oriented performance measures) are not important!

Temporal scopes

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline



Some common scope attributes

Temporal Scopes can be:

Periodic	– e.g. controllers, samplers, monitors
Aperiodic	– e.g. ‘periodic on average’ tasks, burst requests
Sporadic / Transient	– e.g. mode changes, occasional services

Deadlines (absolute, elapse, or execution time) can be:

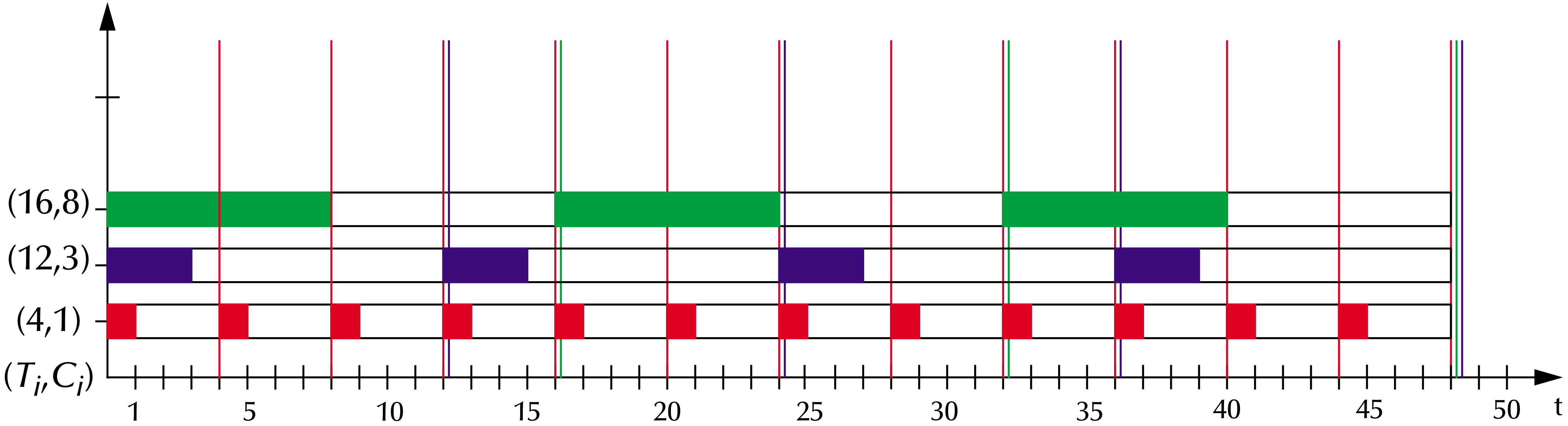
Hard	– single failure leads to severe malfunction
Firm	– results are meaningless after the deadline
	– only multiple or permanent failures threaten the whole system
Soft	– results may still by useful after the deadline

A simple process model

- The number of processes in the system is fixed.
- All processes are periodic and all periods are known.
- All deadlines are identical with the process cycle times (periods).
- The worst case execution time is known for all processes.
- All processes are independent.
- All processes are released at once.
- The task-switching overhead is negligible.

☞ this model can only be applied to a specific group of hard real-time systems.
(extensions to this model will be discussed later in this chapter).

Introducing deadlines



Dynamic scheduling

Earliest deadline first (EDF)

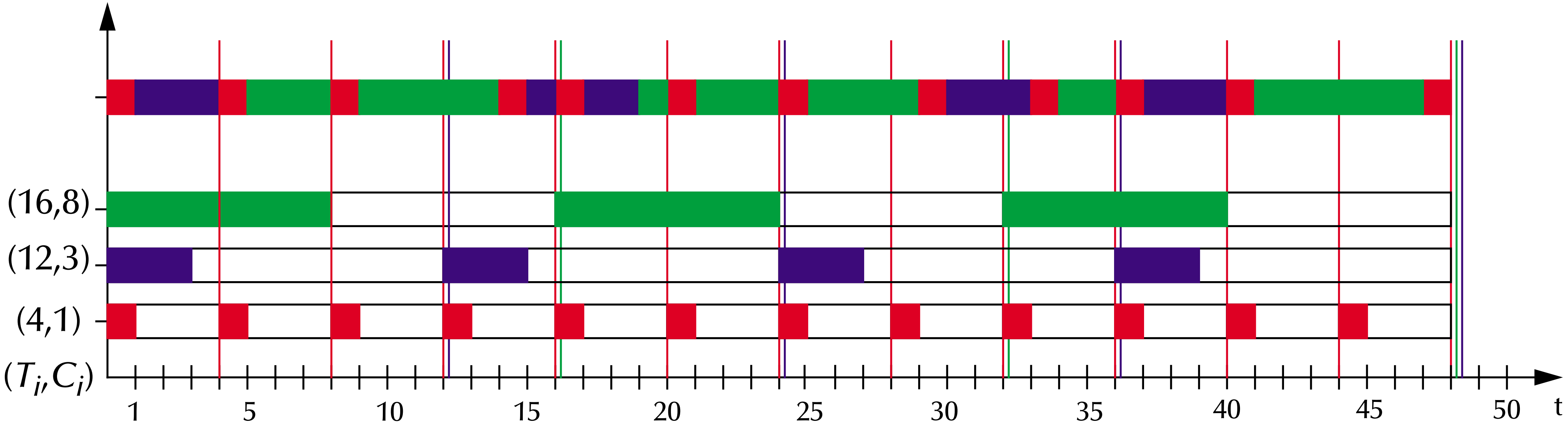
1. Determine (one of) the processe(s) with the closest deadline.
2. Execute this process
   - 2-a until it finishes
   - 2-b or until another process' deadline is found closer then the current one.

☞ Pre-emptive scheme

☞ Dynamic scheme,
since the dispatched process is selected at run-time, due to the current deadlines.

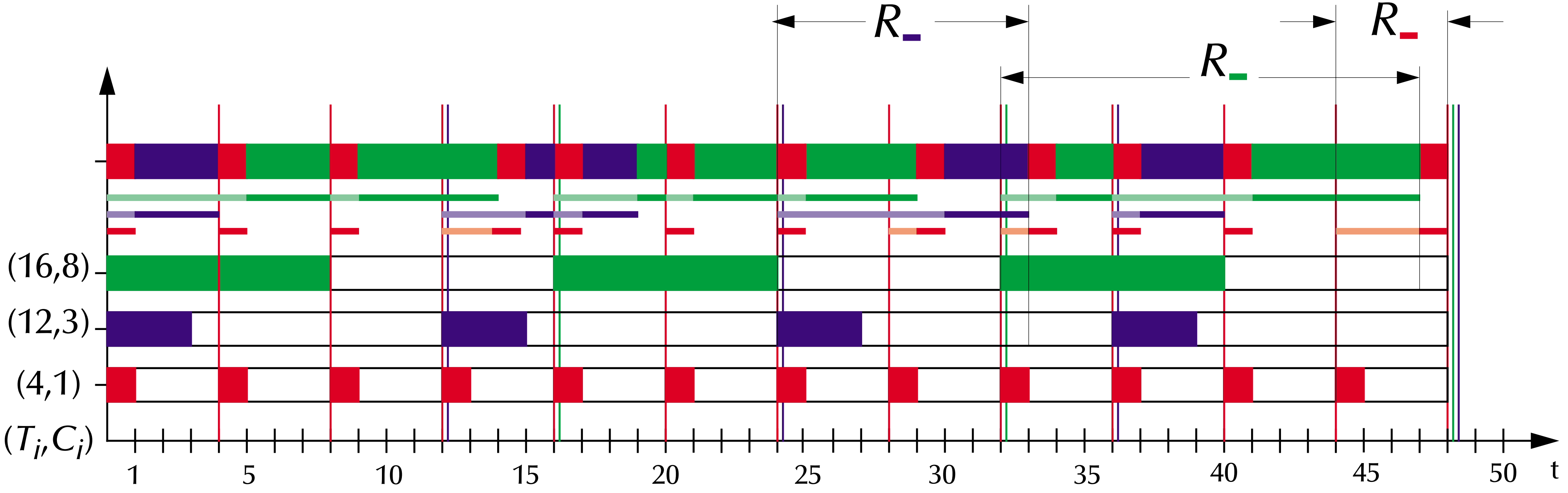
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first



1. Schedule the earliest deadline first
2. Avoid task switches (in case of equal deadlines)

Earliest deadline first: Response times

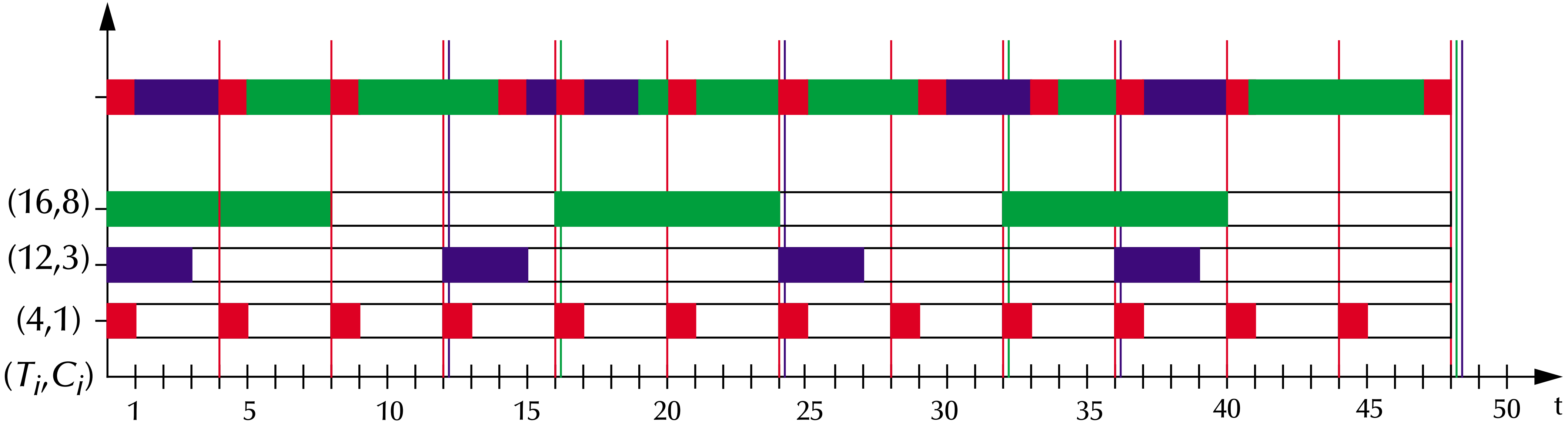


worst case response times R_i (maximal time in which the request from task T_i is served):

- ☞ can be close or identical to deadlines.
- ☞ small or none spare capacity, if any task misses its expected computation time.

Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Maximal utilization



☞ maximal possible utilization: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ ☞ sufficient & necessary test!

with C_i, T_i the computation and cycle times of task *i*

(the deadlines D_i are assumed to be identical with the cycles times T_i here)

Fixed Priority Scheduling (FPS), rate monotonic

1. Each process is assigned a fixed priority according to its cycle time T_i:

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

2. At any point in time: dispatch the process with the highest priority

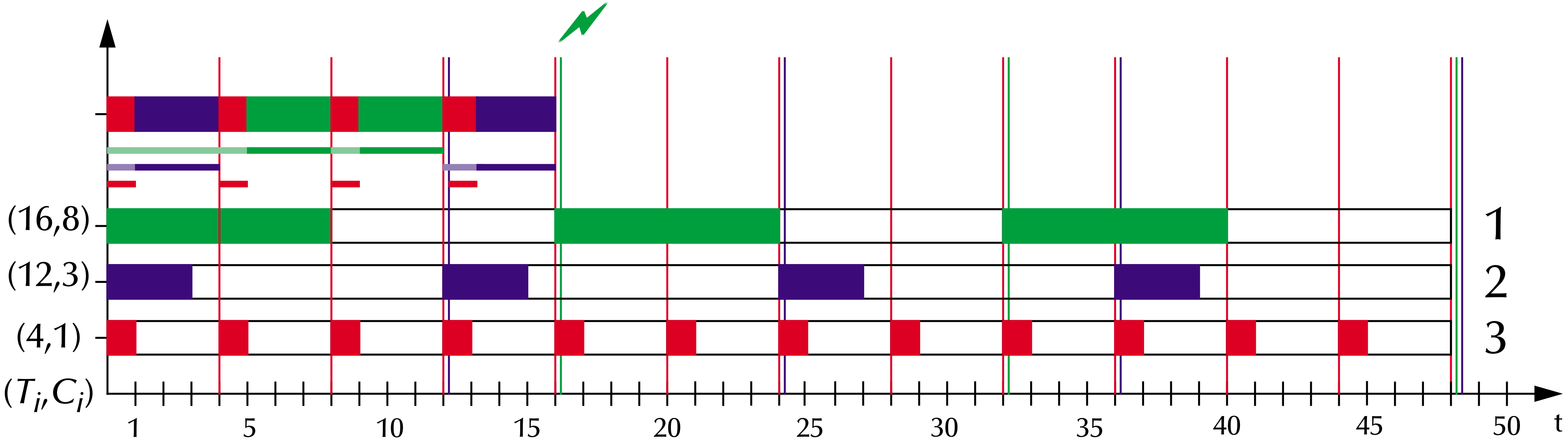
☞ Pre-emptive scheme

☞ Static scheme,
since the dispatch order of processes is fixed and calculated off-line.

- Rate monotonic ordering is **optimal** (in the framework of fixed priority schedulers), i.e. ***if*** a process set is schedulable under a FPS-scheme, ***then*** it is also schedulable by applying rate monotonic priorities.

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

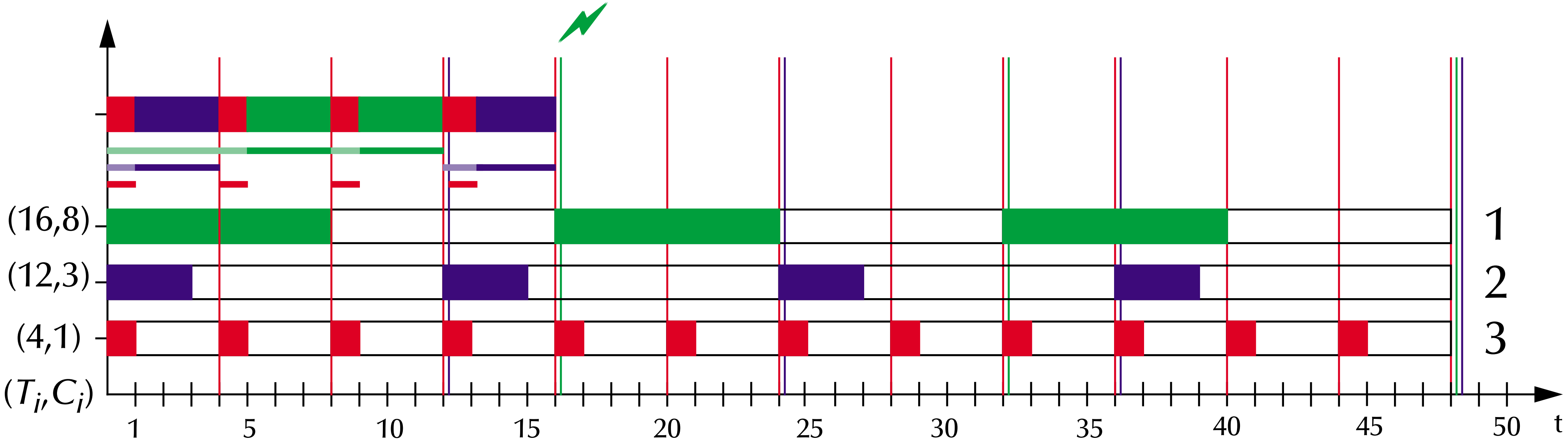
Rate monotonic priorities



☞ assign task priorities according to the cycle times T_i (identical to deadline D_i).

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

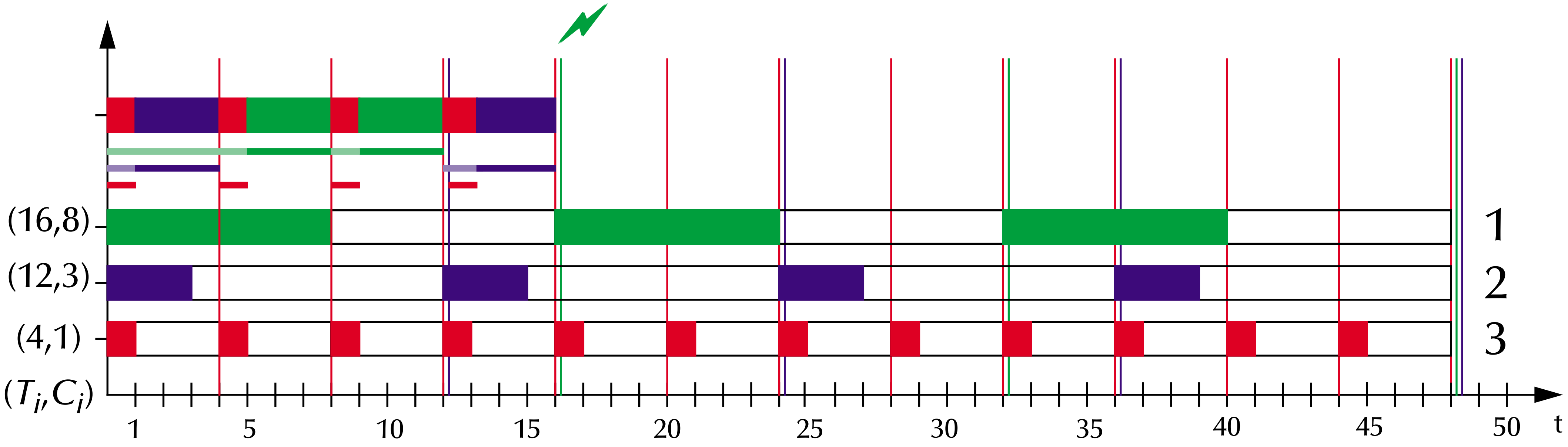
Rate monotonic priorities



max. utilization test: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N\left(2^{\frac{1}{N}} - 1\right)$ ☞ sufficient, but not necessary test!

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

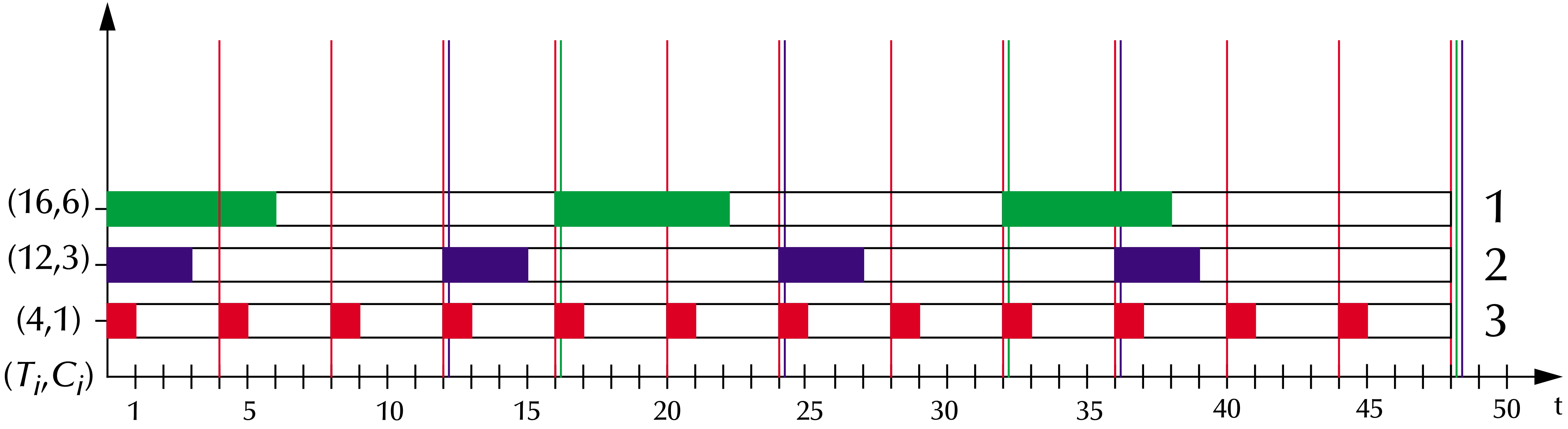
Rate monotonic priorities



utilization test: $\sum_{i=1}^{n} \frac{C_i}{T_i} = 1 > 0.779 \approx N\left(2^{\frac{1}{N}} - 1\right)$ ☞ not guaranteed!

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

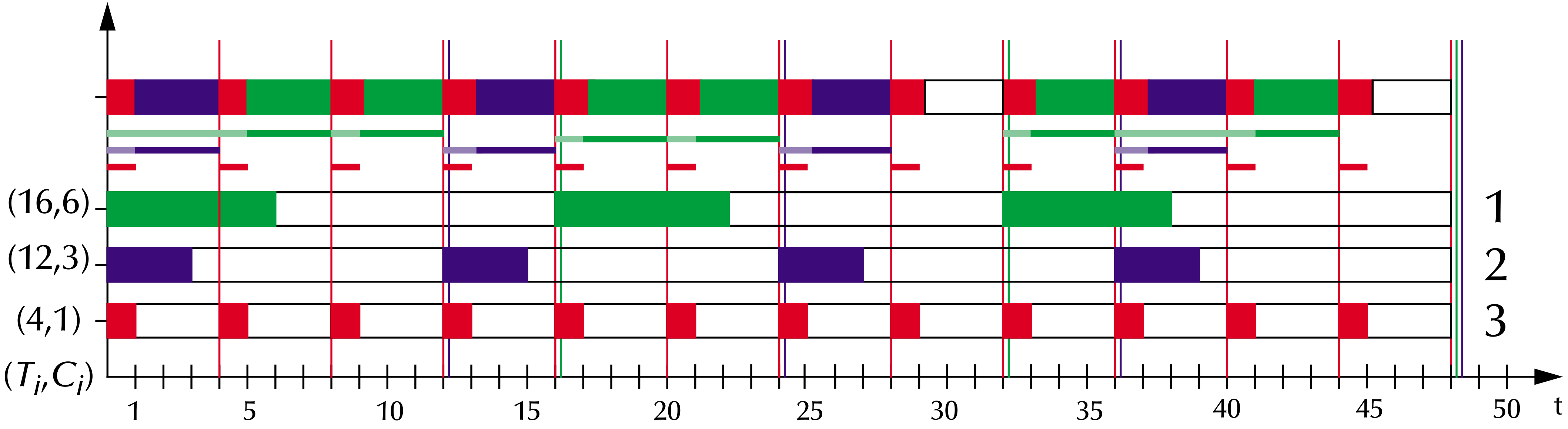
Rate monotonic priorities (reduced requests)



max. utilization test: $\sum_{i=1}^{n} \frac{C_i}{T_i} \le N\left(2^{\frac{1}{N}} - 1\right)$

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

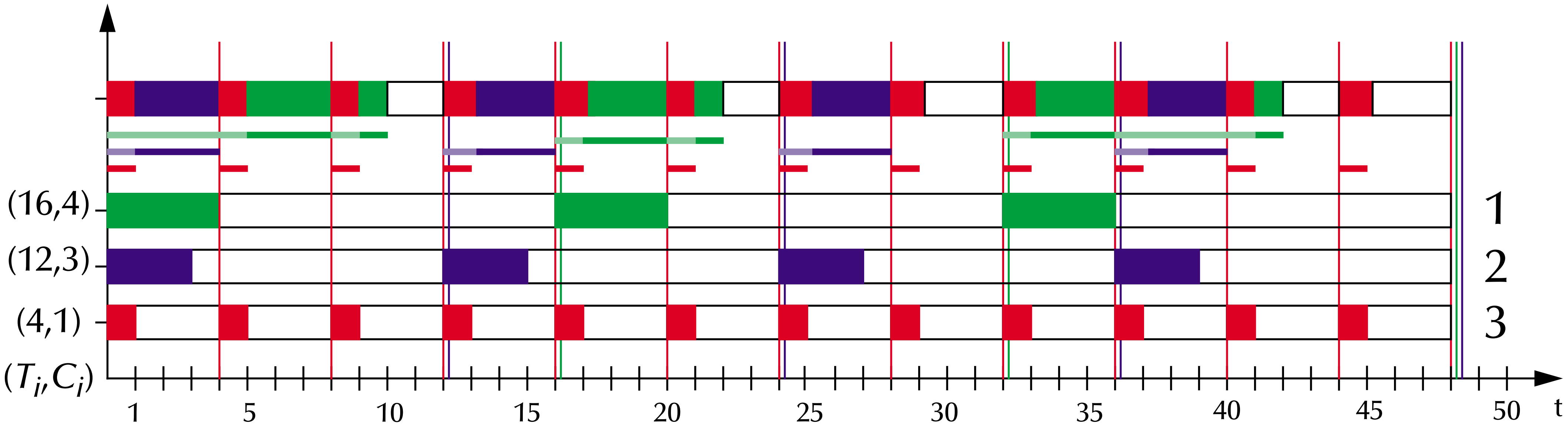
Rate monotonic priorities (reduced requests)



☞ utilization: $\frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx 3\left(2^{\frac{1}{3}} - 1\right)$; $\sum_{i=1}^{n} \frac{C_i}{T_i} \le N\left(2^{\frac{1}{N}} - 1\right)$ ☞ not guaranteed!

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

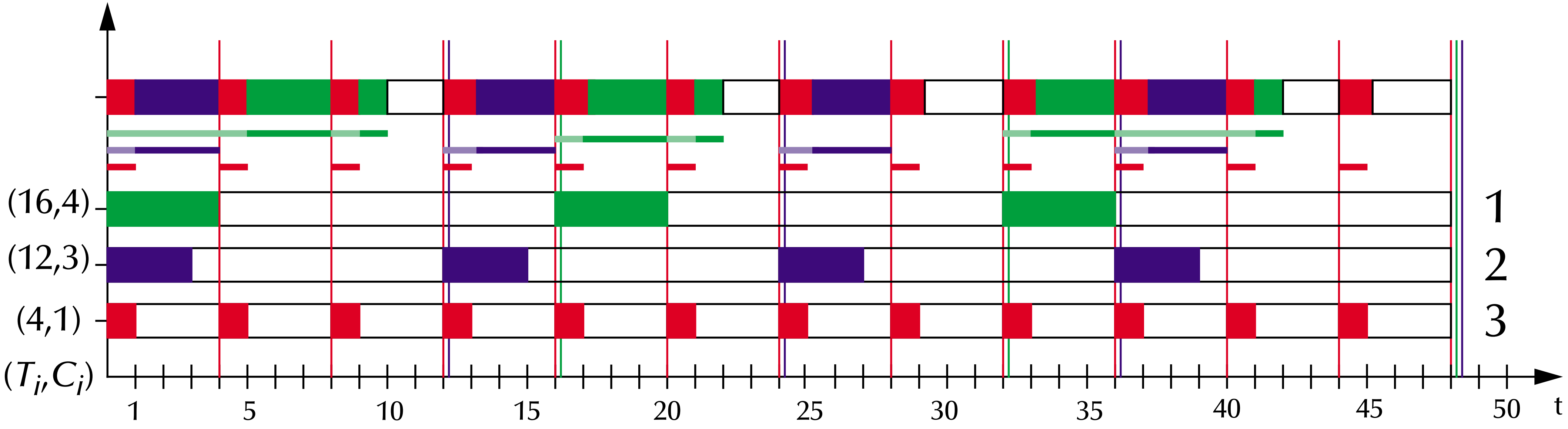
Rate monotonic priorities (further reduced requests)



☞ utilization: $\frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 \le 0.779 \approx 3\left(2^{\frac{1}{3}} - 1\right)$; $\sum_{i=1}^{n} \frac{C_i}{T_i} \le N\left(2^{\frac{1}{N}} - 1\right)$ ☞ guaranteed!

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

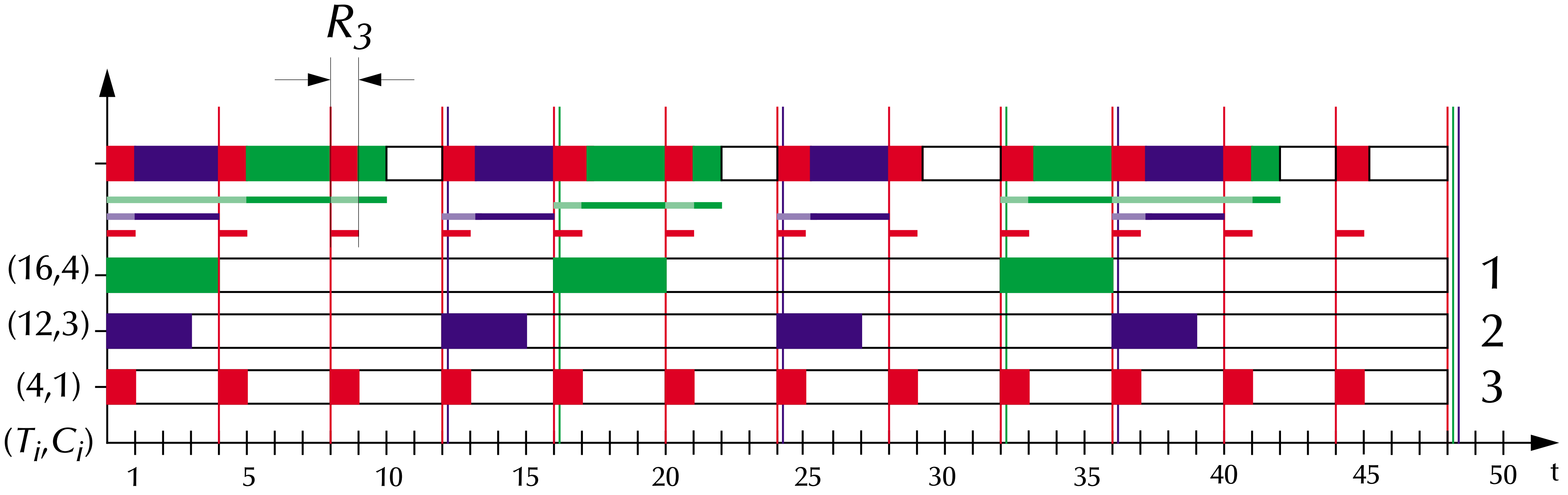
Response time analysis *(further reduced requests)*



☞ calculate the worst case response times for each task individually.

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

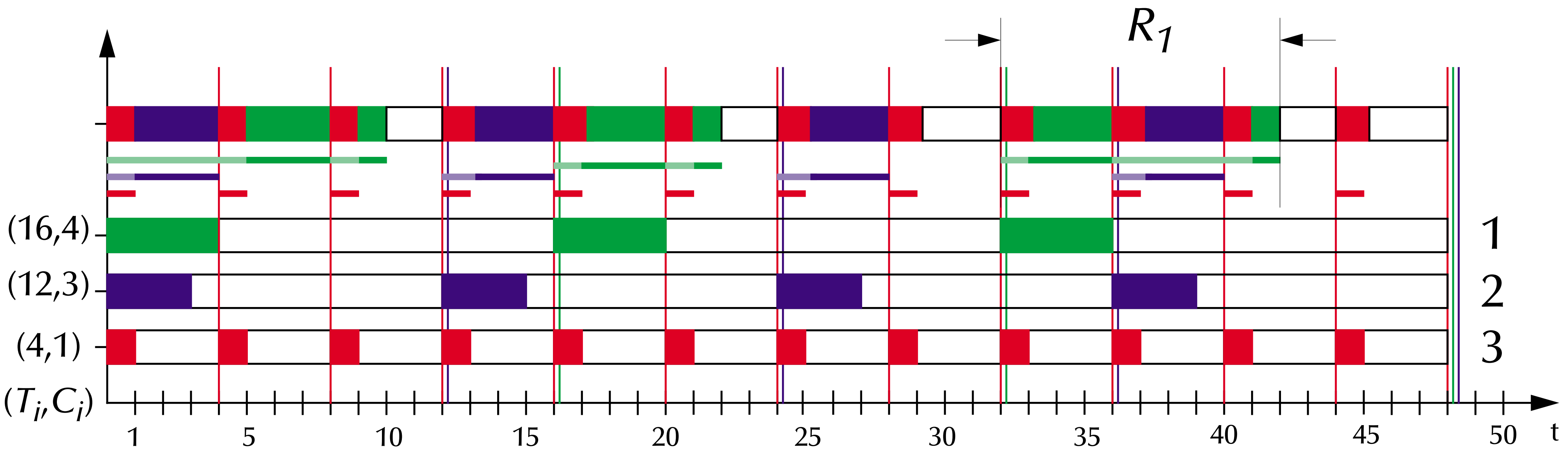
Response time analysis *(further reduced requests)*



☞ for the highest priority task: $R_3 = C_3$

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

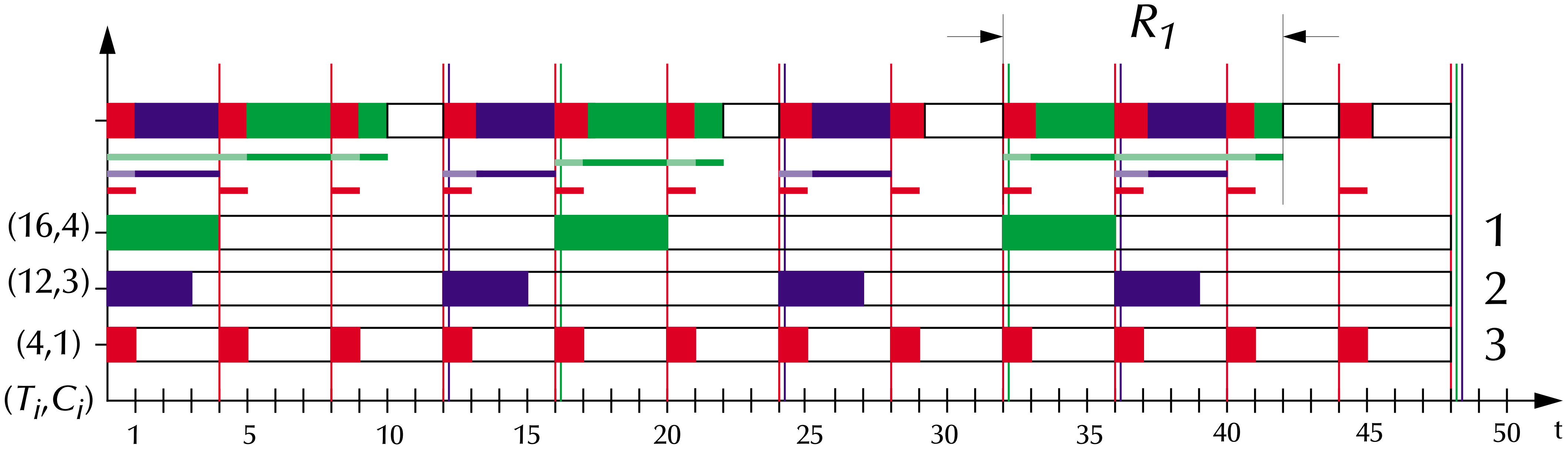
Response time analysis *(further reduced requests)*



☞ for other tasks: $R_i = C_i + I_i$ = computation C_i + interference I_i

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)



for other tasks: $R_i = C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

$$R_i = C_i + \sum_{j > i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

☞ fixed-point equation!

☞ form recurrent equation: $$R_i^{k+1} = C_i + \sum_{j > i} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \quad (1)$$

☞ starting with $R_i^0 = C_i$

☞ Iterate (1) until $R_i^{k+1} = R_i^k$ or $R_i^{k+1} > T_i$

Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

The worst case for EDF is *not* necessarily when all tasks are released at once!

☞ all possible combinations in a full hyper -cycle need to be considered!

- The response times are bounded by the cycle times as long as the maximal utilization is ≤ 1.
- Other tasks need to be considered only, if their deadline is closer or equal to the current task.

Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

$$R_i(a) = \left\lfloor \frac{a}{T_i} + 1 \right\rfloor C_i + \sum_{j \neq i} {}_{min}\left\{ \left\lceil \frac{R_i(a)}{T_j} \right\rceil, {}_{max}\left\{ 0, \left\lfloor \frac{a + T_i - T_j}{T_j} \right\rfloor + 1 \right\} \right\} C_j$$

☞ $$R_i^{k+1}(a) = \left\lfloor \frac{a}{T_i} + 1 \right\rfloor C_i + \sum_{j \neq i} {}_{min}\left\{ \left\lceil \frac{R_i^k(a)}{T_j} \right\rceil, {}_{max}\left\{ 0, \left\lfloor \frac{a + T_i - T_j}{T_j} \right\rfloor + 1 \right\} \right\} C_j \quad (2)$$

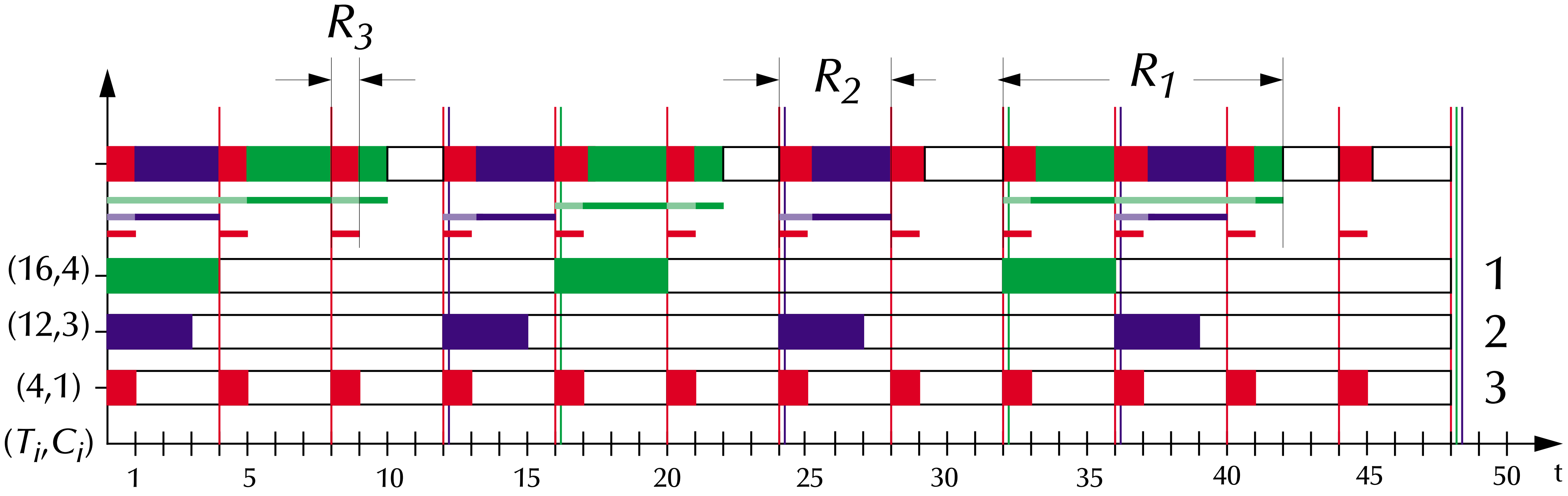
☞ starting with $R_i^0(a) = a + C_i$

☞ Iterate (2) until $R_i^{k+1}(a) = R_i^k(a)$

☞ $R_i = {}_{max}\{R_i(a) - a\}_{a \in A}$; where $A = scm\{T_i\}$

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

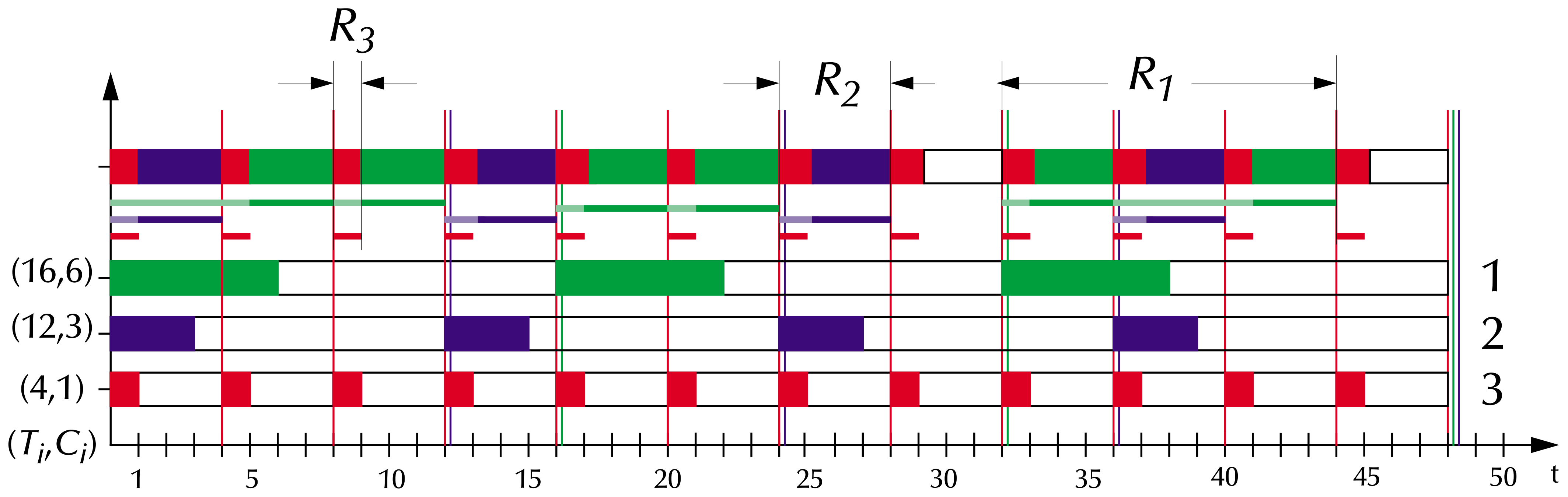
Response time analysis *(further reduced requests)*



☞ $R_i = C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$; $R_3 = 1$✔; $R_2 = 4$✔; $R_1 = 10$✔ and $\sum_{i=1}^{n} \frac{C_i}{T_i} \le N\left(2^{\frac{1}{N}} - 1\right)$✔

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

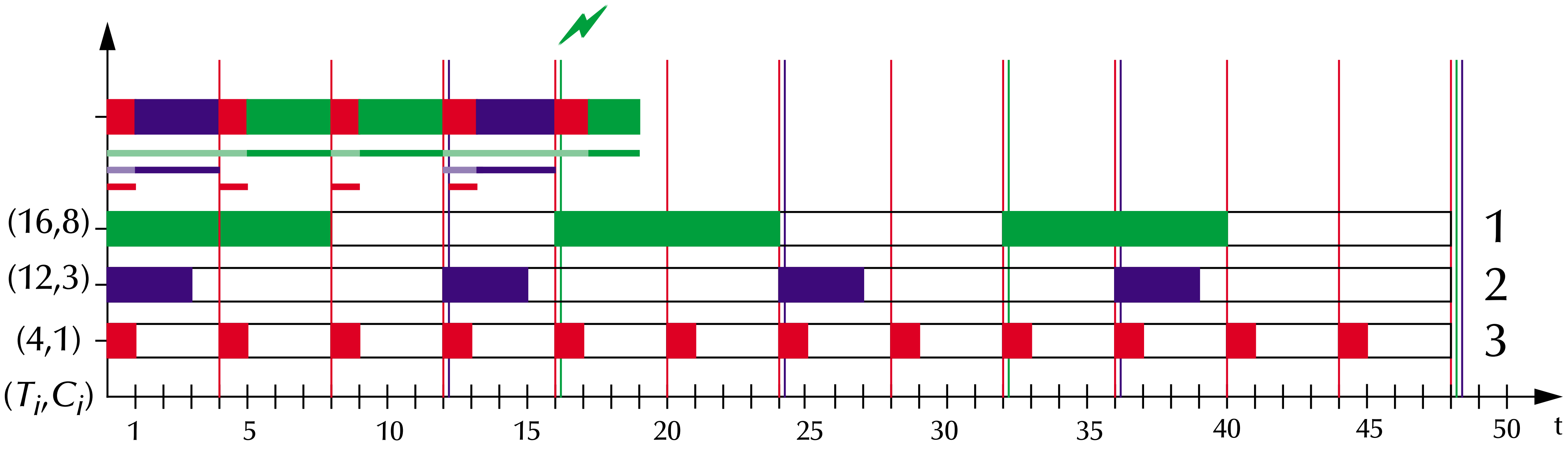
Response time analysis (reduced requests)



☞ $R_i = C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$; $R_3 = 1$✓; $R_2 = 4$✓; $R_1 = 12$✓ but $\sum_{i=1}^{n} \frac{C_i}{T_i} > N\left(2^{\frac{1}{N}} - 1\right)$ ✗

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

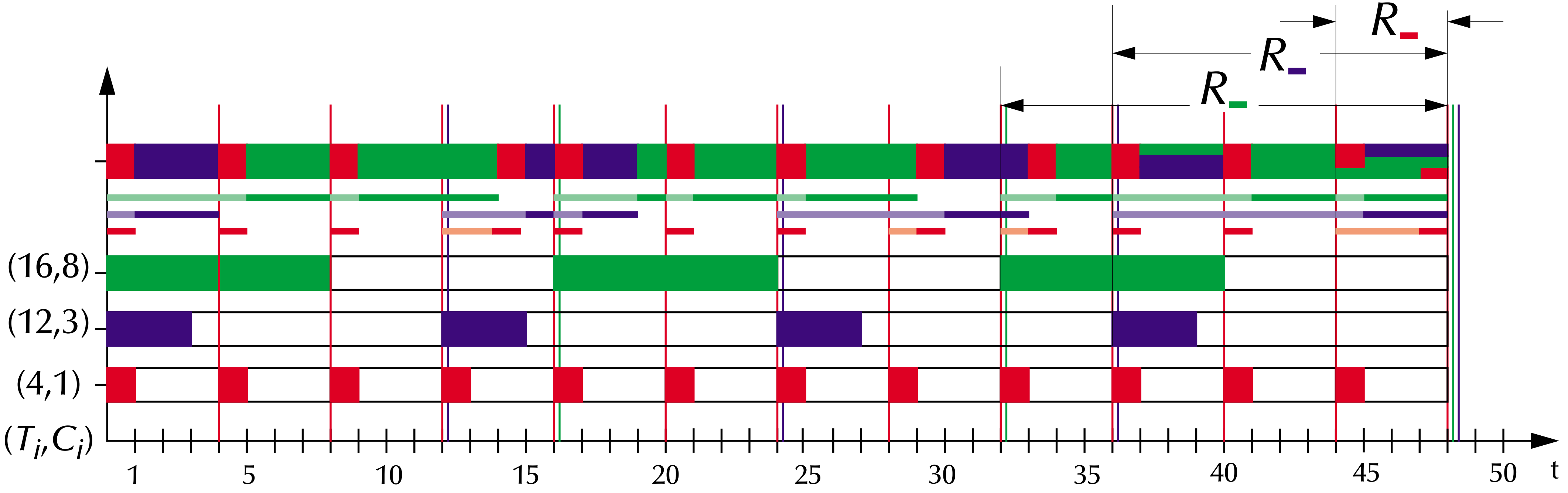
Response time analysis *(full requests)*



☞ $R_i = C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j;\ R_3 = 1$ ✓; $R_2 = 4$ ✓; $R_1 = 19$ ✗ and $\sum_{i=1}^{n} \frac{C_i}{T_i} > N\left(2^{\frac{1}{N}} - 1\right)$ ✗

Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis *(full requests)*

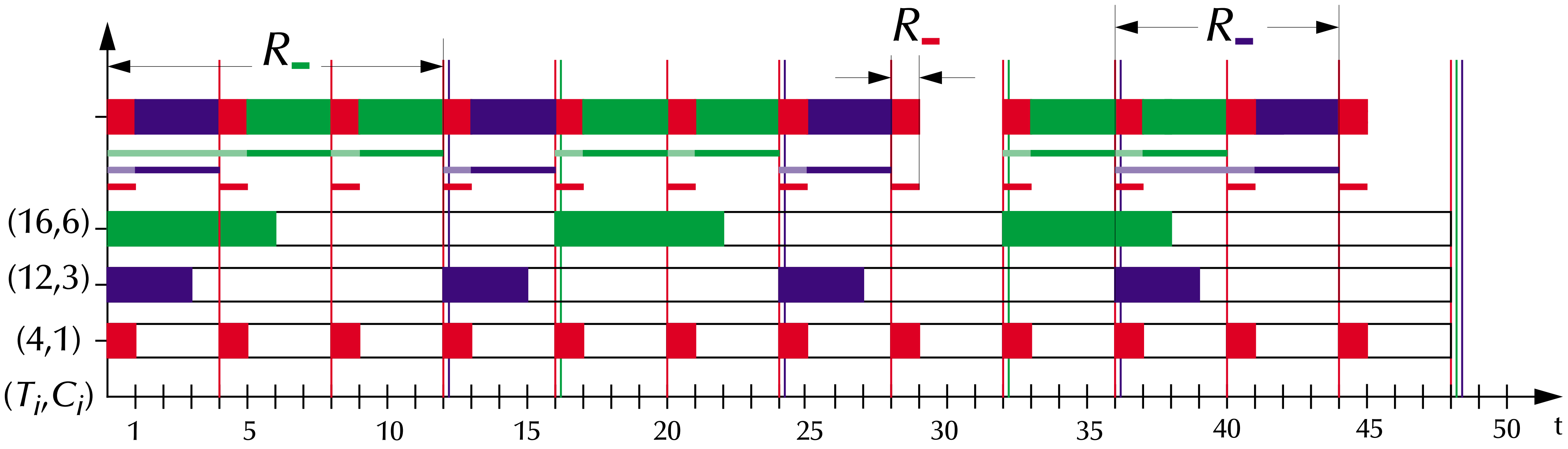


☞ testing all combinations in a hyper-period: LCM of $\{T_i\}$ — here: 48

$$R_{🟩} : 16 \leq 16 ✓ = T_{🟩} ; \qquad R_{🟪} : 12 \leq 12 ✓ = T_{🟪} ; \qquad R_{🟥} : 4 \leq 4 ✓ = T_{🟥}$$

Dynamic scheduling: Earliest Deadline First (EDF)

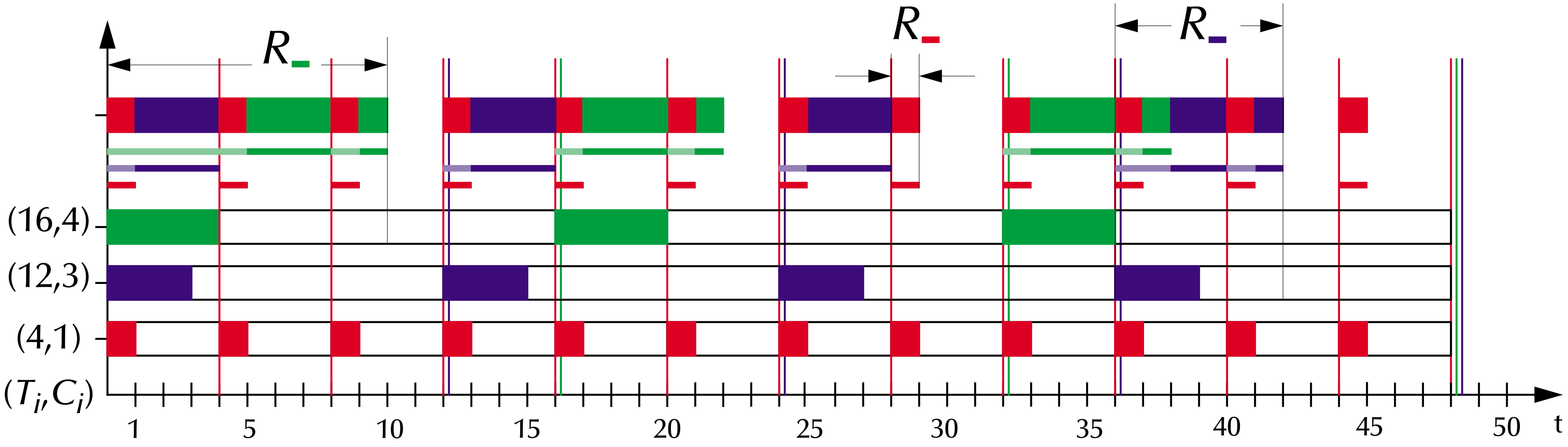
Response time analysis (reduced requests)



☞ relaxed task-set changes:

R🟩 : 16 → 12 ≤ 16✔ = *T*🟩 ;   *R*🟪 : 12 → 8 ≤ 12✔ = *T*🟪 ;   *R*🟥 : 4 → 1 ≤ 4✔ = *T*🟥

Response time analysis *(further reduced requests)*



☞ further relaxed task-set changes:

R🟩 $: 12 \rightarrow 10 \leq 16$✔ $= T$🟩 ; R🟪 $: 8 \rightarrow 6 \leq 12$✔ $= T$🟪 ; R🟥 $: 1 \rightarrow 1 \leq 4$✔ $= T$🟥

Response time analysis (comparison)

	Fixed Priority Scheduling		Earliest Deadline First	
	utilization test	response times $\{R_i\}$	utilization test	response times $\{R_i\}$
$\{(T_i, C_i)\} = \{(16, 8);(12, 3);(4, 1)\}$	✗ (1.000)	{✗, 4, 1}	✔ (1.000)	{16, 12, 4}
$\{(T_i, C_i)\} = \{(16, 6);(12, 3);(4, 1)\}$	✗ (0.875)	{12, 4, 1}	✔ (0.875)	{12, 8, 1}
$\{(T_i, C_i)\} = \{(16, 4);(12, 3);(4, 1)\}$	✔ (0.750)	{10, 4, 1}	✔ (0.750)	{10, 6, 1}
	$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N\left(2^{\frac{1}{N}} - 1\right)$	$C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$	$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$	check full hyper-cycle

Fixed Priority Scheduling ↔ Earliest Deadline First

- EDF can handle higher (full) utilization than FPS.
- FPS is easier to implement and implies less run-time overhead

- Graceful degradation features (resource is over-booked):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline ☞ and can trigger a cascade of failed deadlines.

- Response time analysis and utilization tests:
  - FPS: O(n) utilization test — response time analysis: fixed point equation
  - EDS: O(n) utilization test — response time analysis: fixed point equation in hyper-cycle

Extensions which we will introduce:

- tasks are periodic
  - ☞ we will introduce sporadic and aperiodic processes
- tasks are independent
  - ☞ we will introduce schedules for interacting tasks
- deadlines are identical with task's period time ($D = T$)
  - ☞ Real-time course
- pre-emptive scheduling
  - ☞ Real-time course
- worst case execution times are known
  - ☞ Real-time course

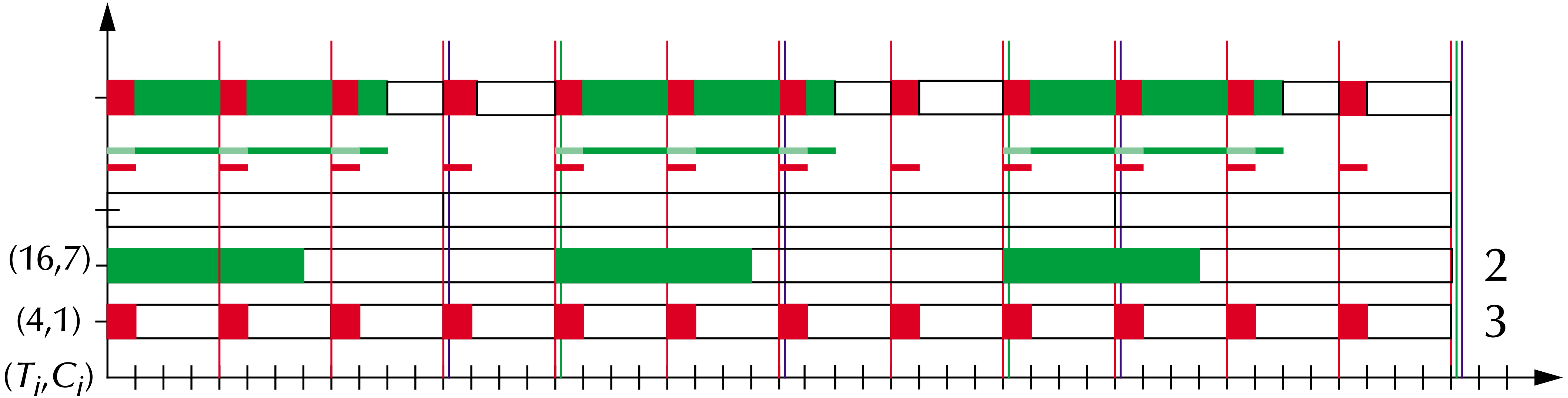
Scheduling — real-world considerations

… including

aperiodic, sporadic & ‘soft’ real-time tasks

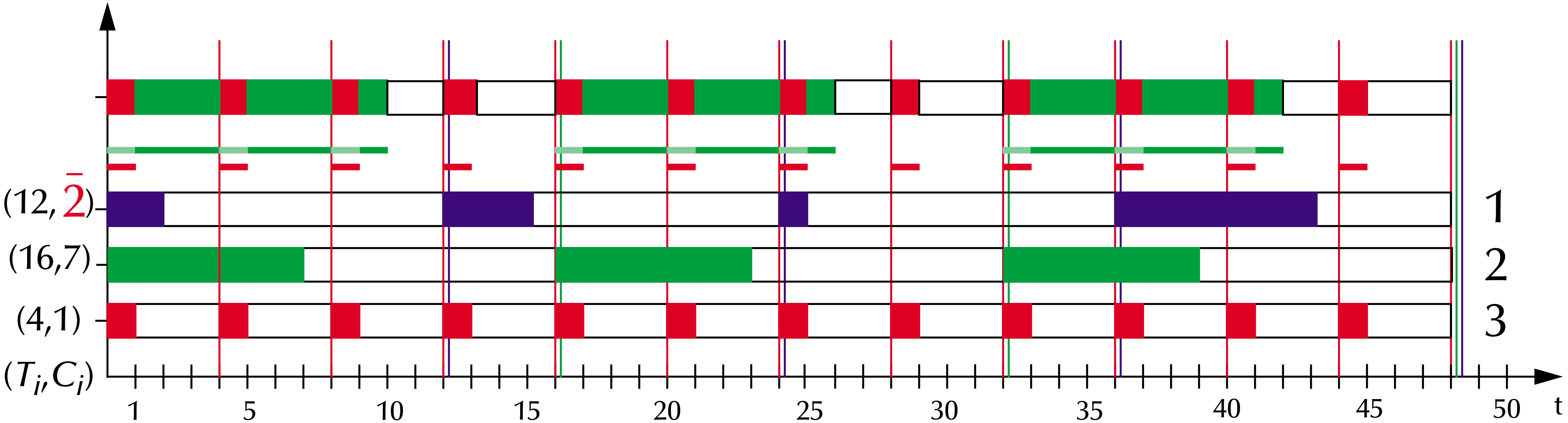
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Hard real-time tasks



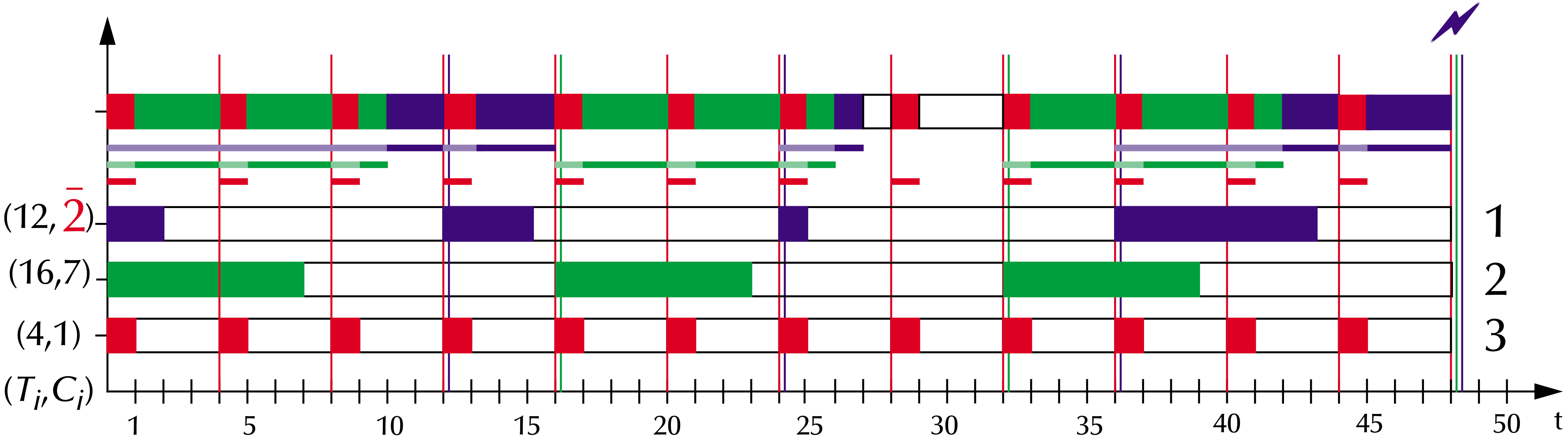
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Introducing soft real-time tasks



Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

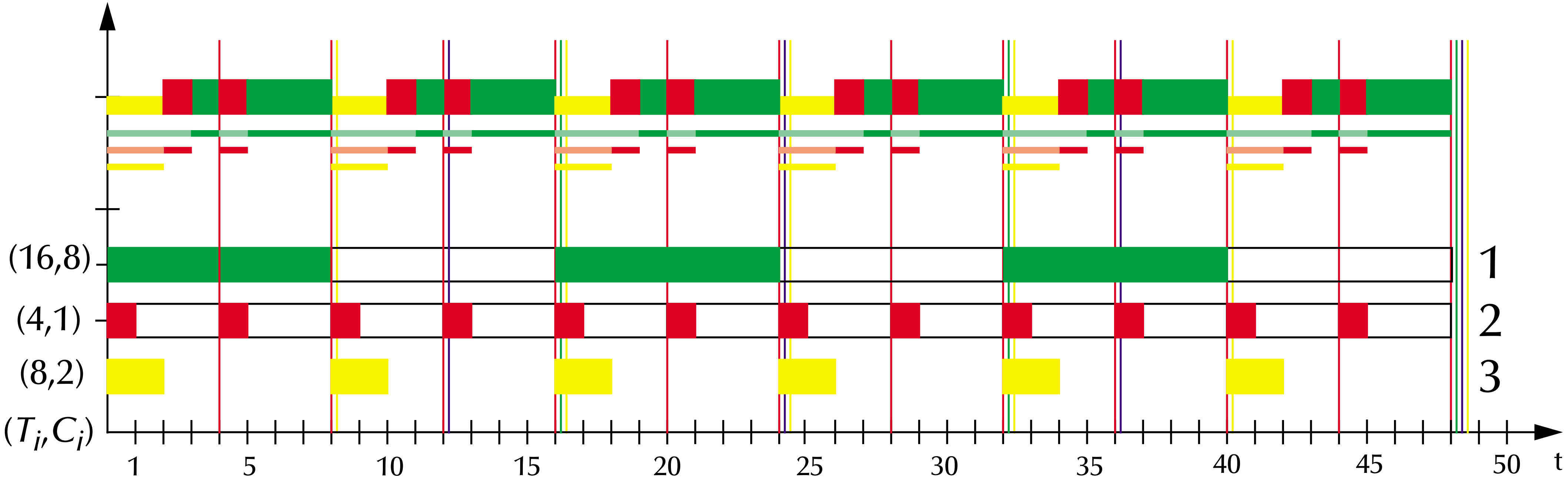
Introducing soft real-time tasks



- ☞ set can be scheduled using average computation and period times
- ☞ hard real-time tasks can be scheduled under worst case conditions (including worst case behaviours of soft real-time tasks)

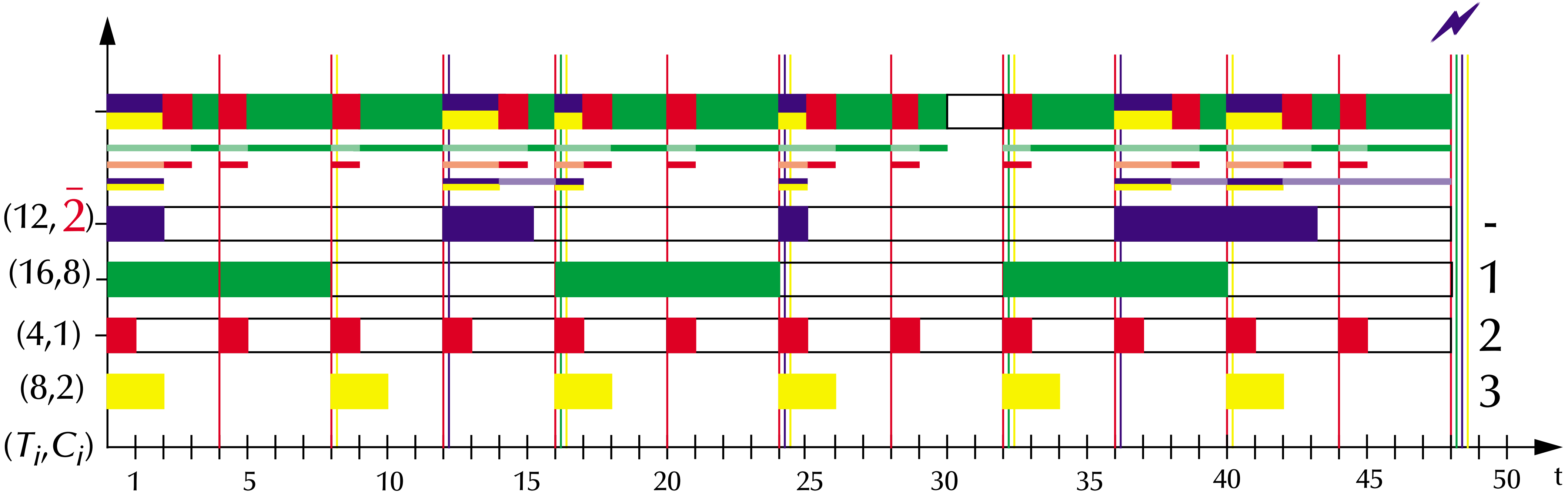
Static scheduling: FPS, rate monotonic + server

Introducing a server task



Server is established at a high priority

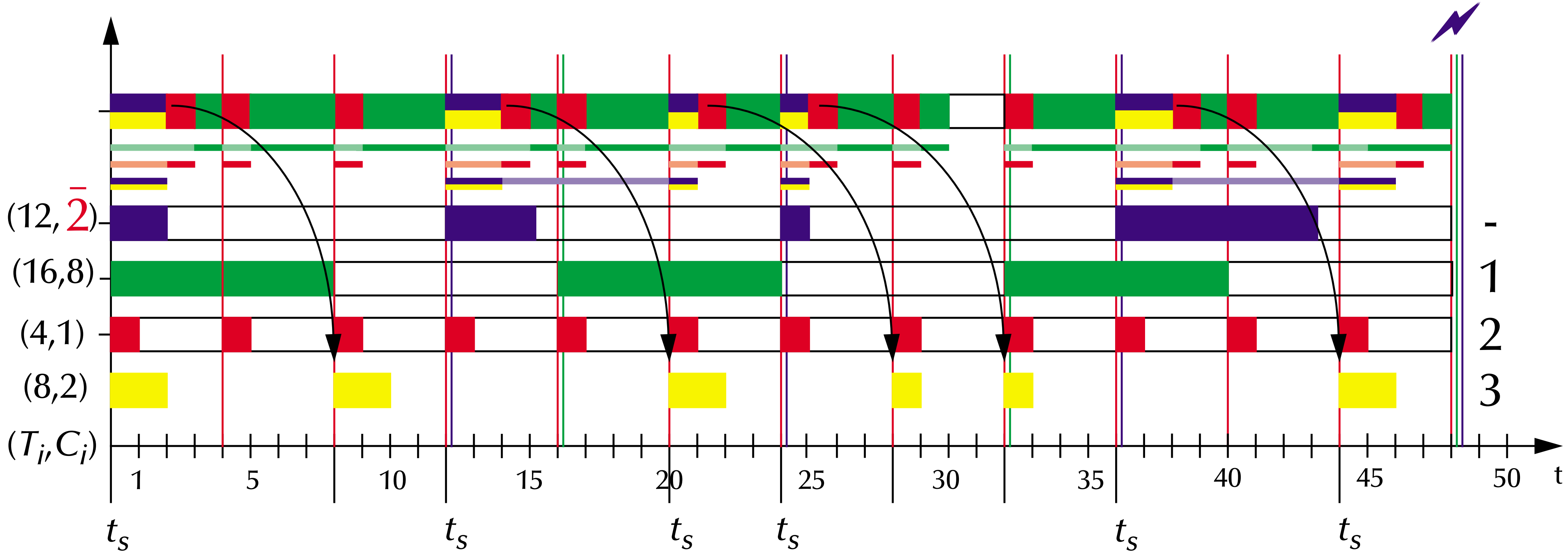
Introducing a server task: Deferrable Server



☞ Deferrable Server (DS): Capacity replenished every T_s (here: 8)

Static scheduling: FPS, rate monotonic + server

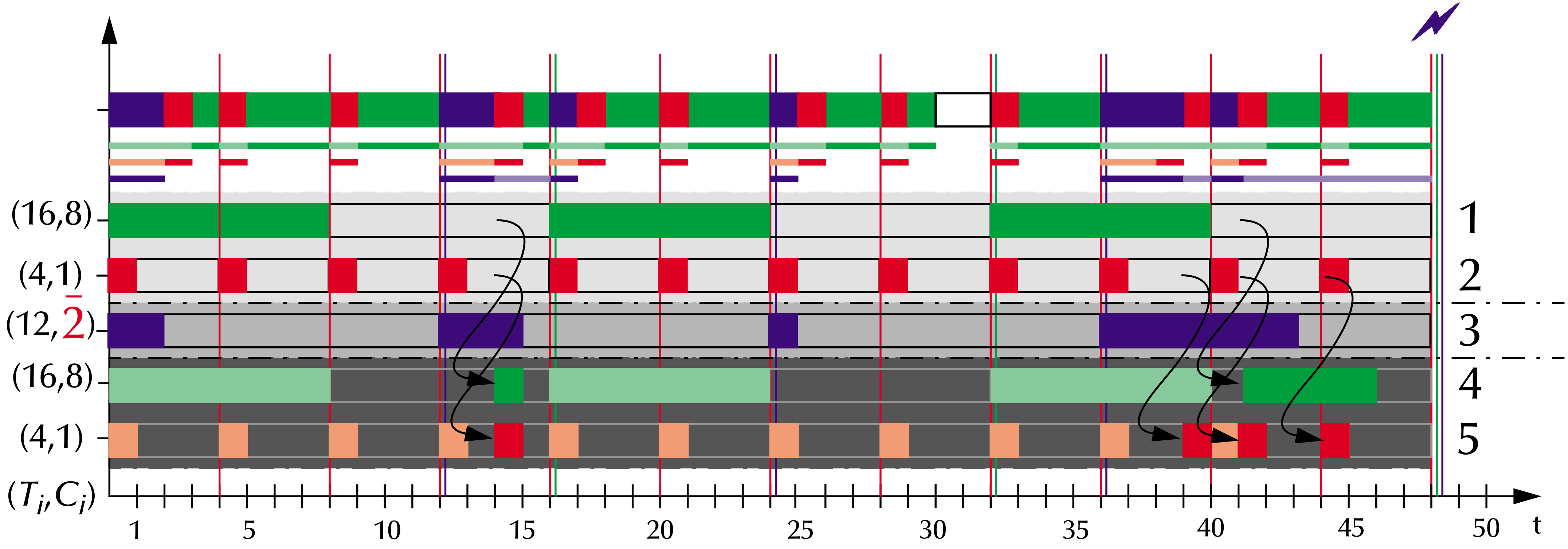
Introducing a server task: Sporadic Server



☞ Sporadic Server (SS): Capacity replenished T_s units after t_s ☞ POSIX

Static scheduling: Fixed Priority Scheduling (FPS), dual-priorities

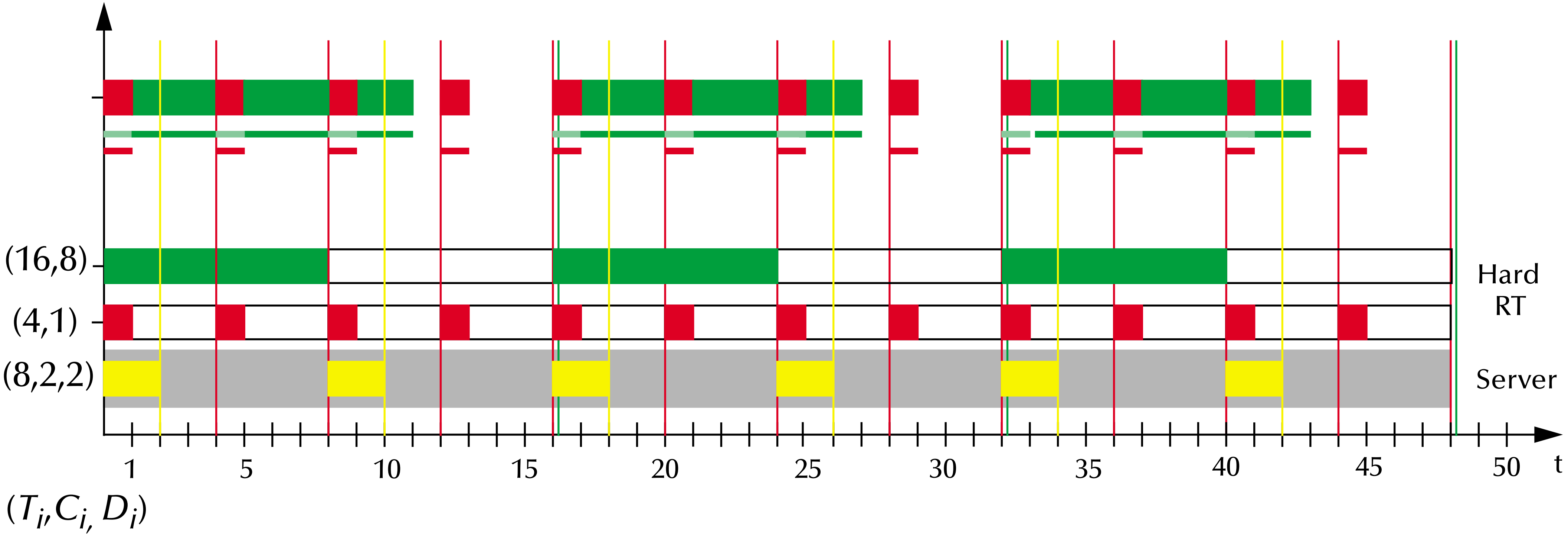
Introducing dual priorities



☞ start hard rt-tasks in low priorities; promote them in time to higher ones

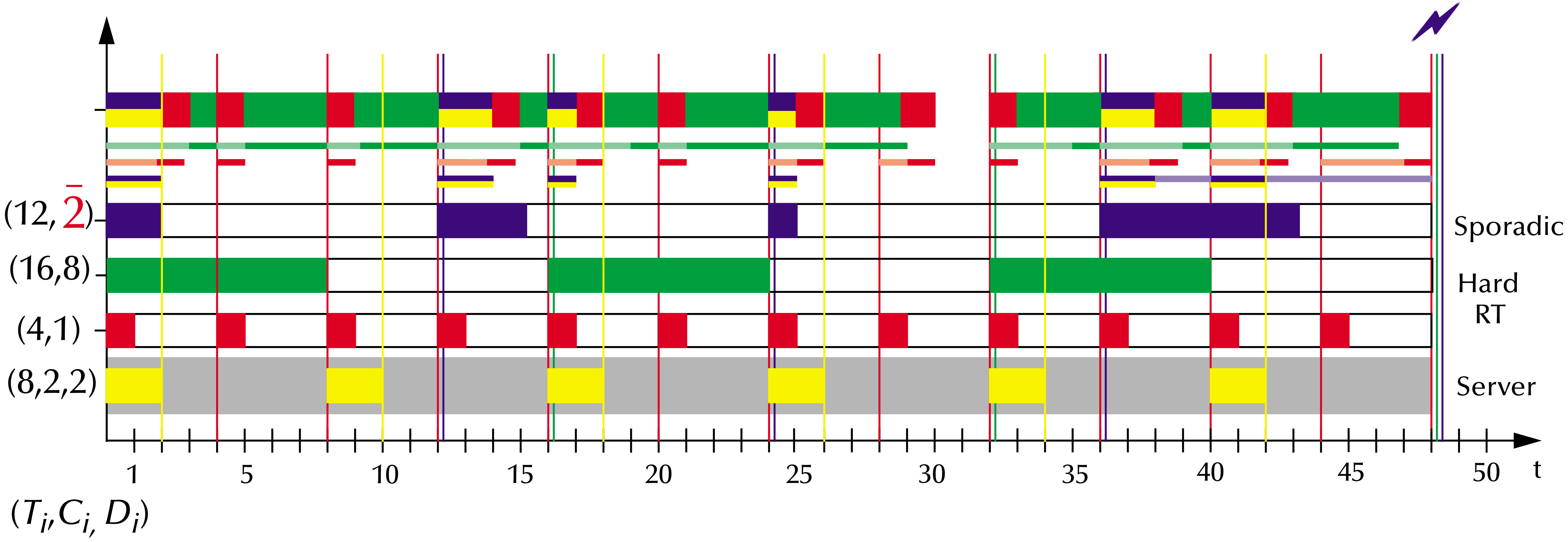
Dynamic scheduling: Earliest Deadline First+ aperiodic server

Introducing a server task to EDF



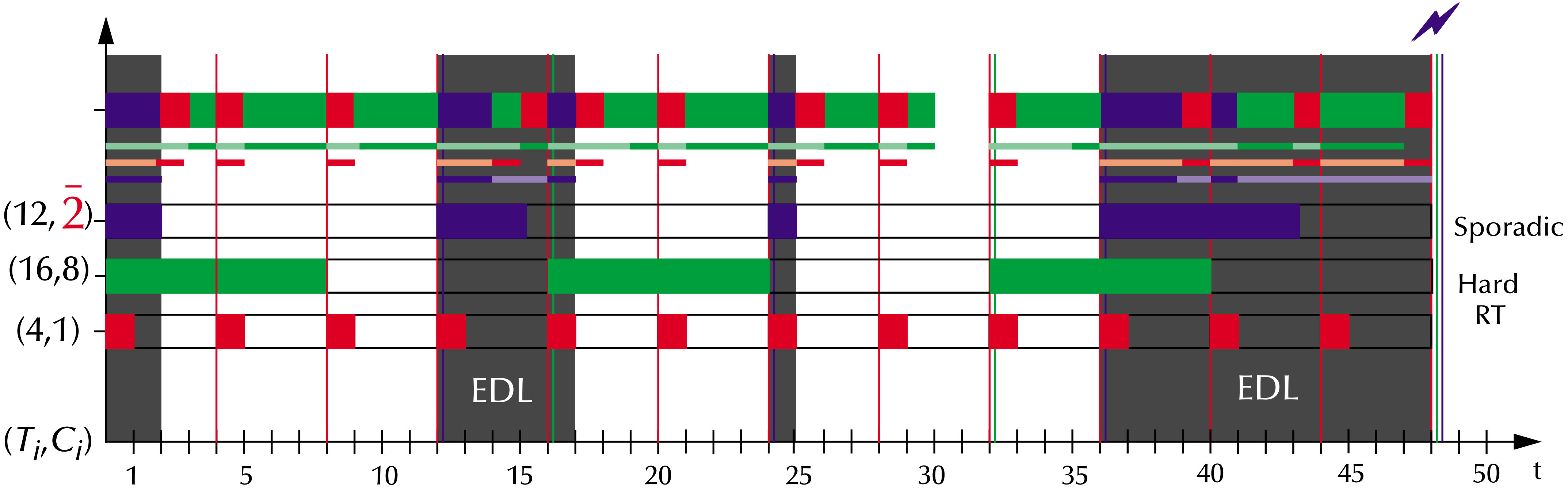
Dynamic scheduling: Earliest Deadline First + aperiodic server

Introducing a server task to EDF



Dynamic scheduling: Earliest Deadline First + aperiodic tasks

Switching between EDF & Earliest Deadline Last (EDL)



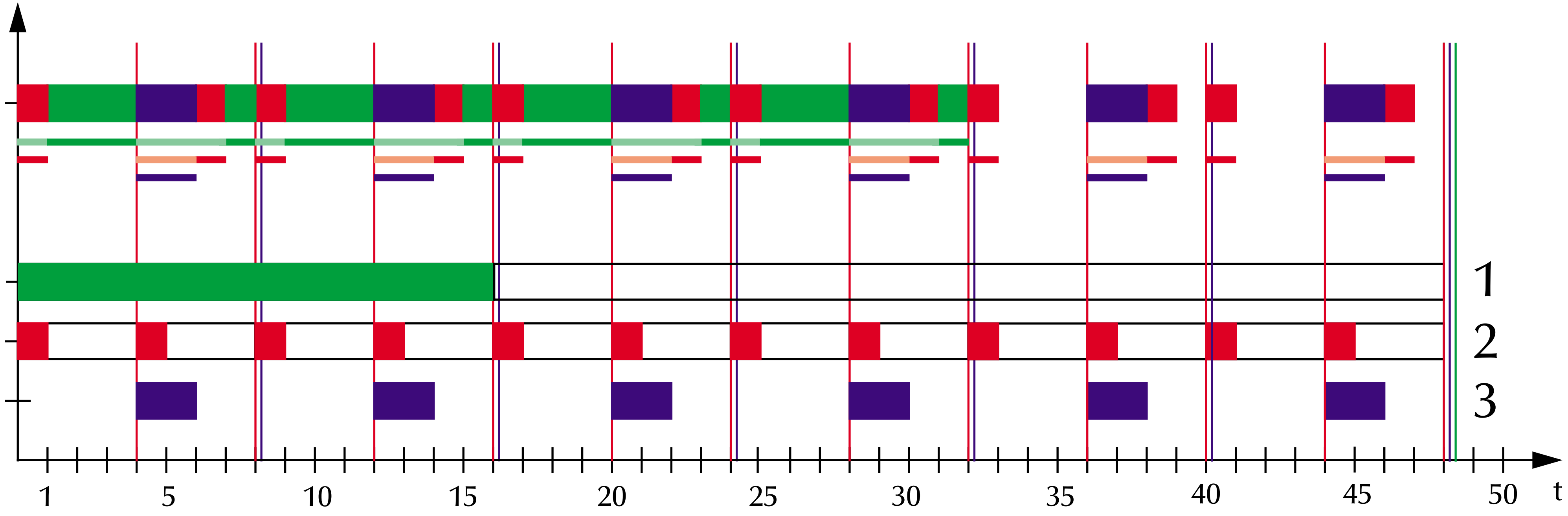
Scheduling — real-world considerations

… including

task interdependencies

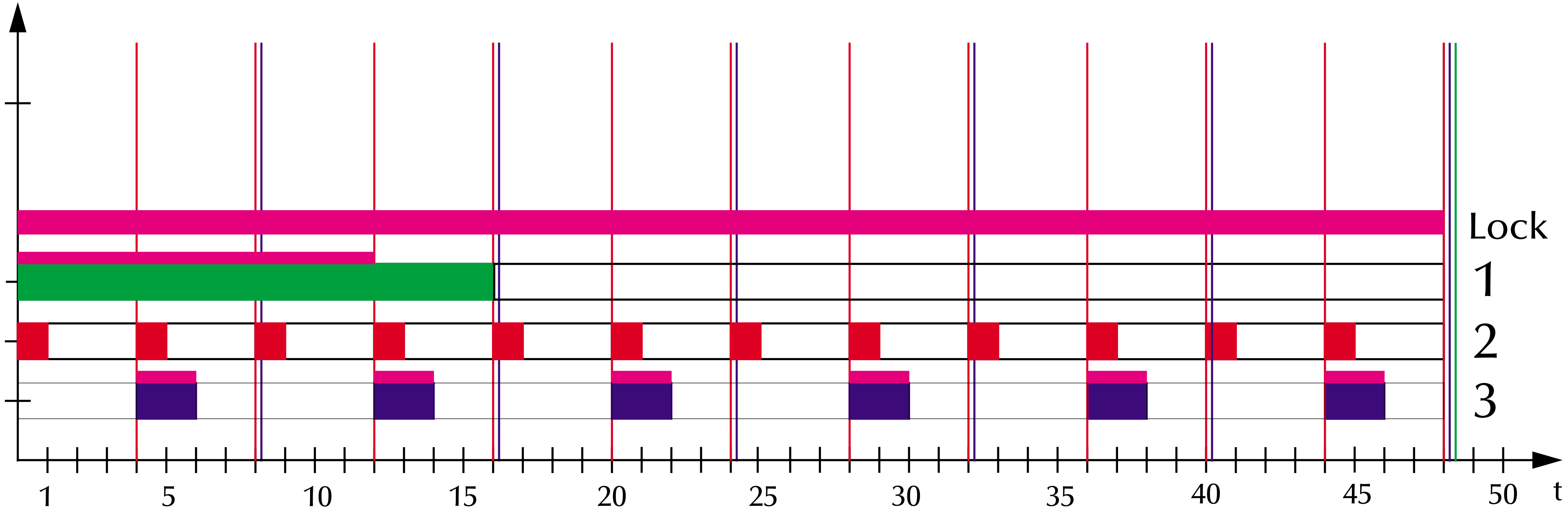
Scheduling: Interdependencies

Schedule for independent tasks



(independent task set)

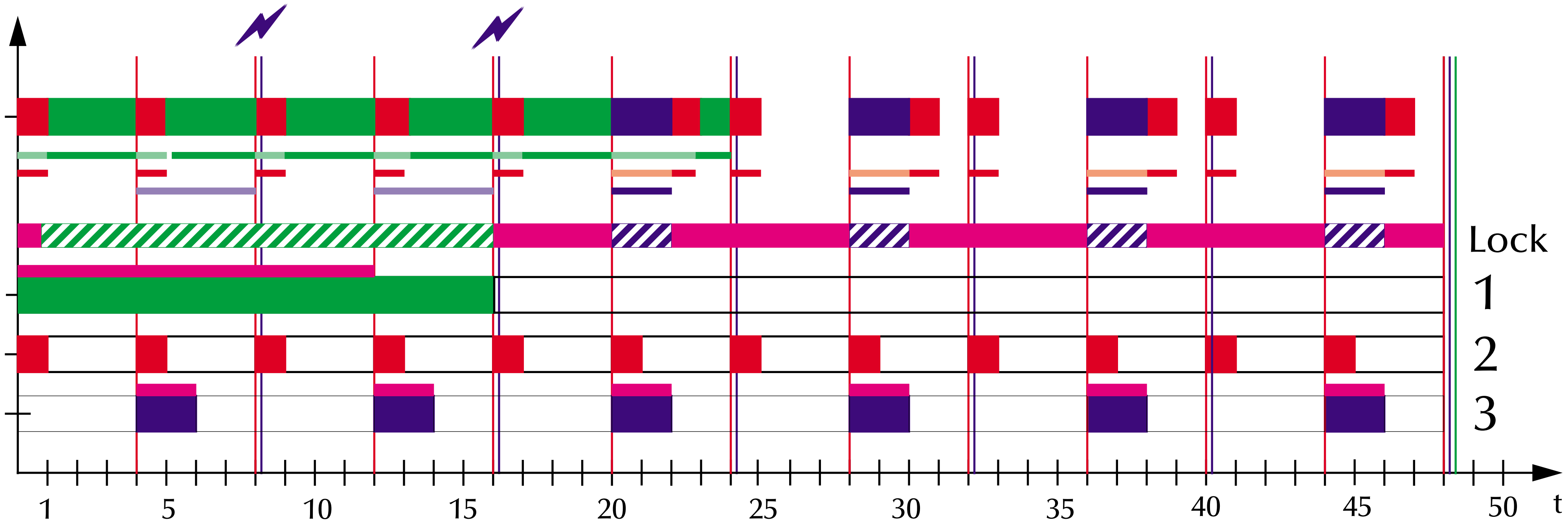
Synchronized via lock



(interdependent task set ☞ lock 🟥 shared between 🟩 and 🟪)

Scheduling: Interdependencies

Synchronized via lock



☞ *Priority inversion*

(interdependent task set ☞ lock ■ shared between 🟩 and 🟪)

Priority inheritance

Task t_i inherits the priority of t_j, if:

1. $P_i < P_j$
2. task t_i has locked a resource Q
3. task t_j is blocked waiting for resource Q to be released

Scheduling: Interdependencies

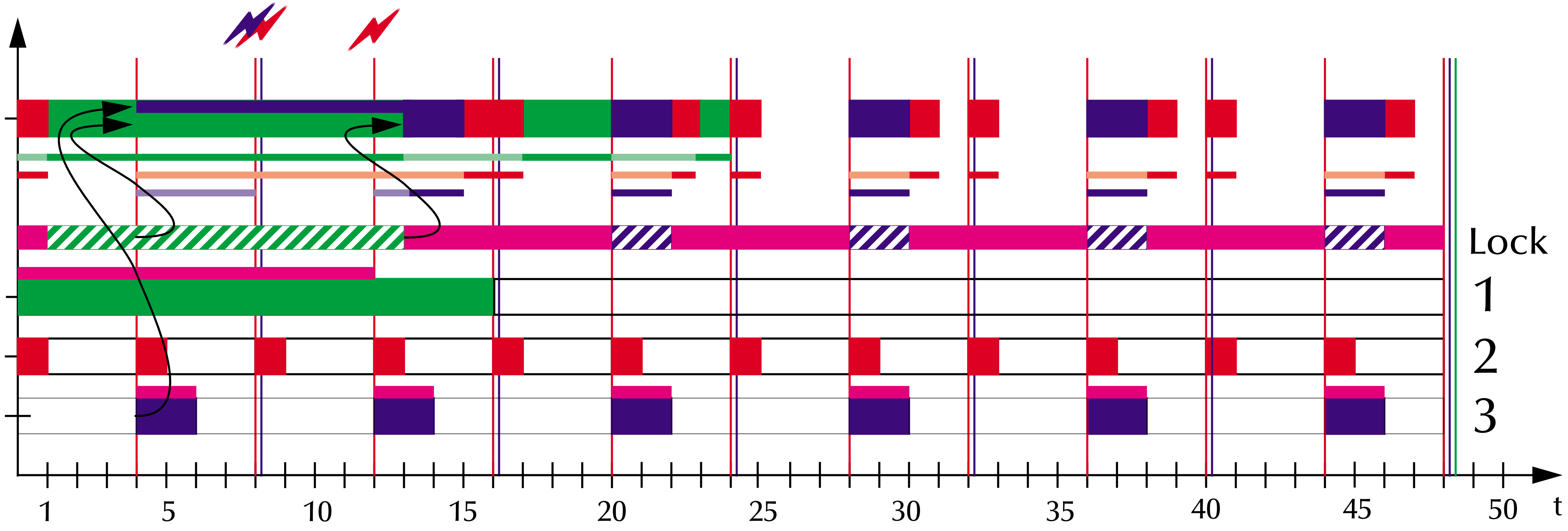
Priority inheritance

Maximal blocking time for task t_i: $B_i = \sum_{r=1}^{R} usage(r, i)C(r)$

- with R the number of critical sections
- $usage(r, i)$ a boolean (0/1) function indicating that r is used by at least one t_j with $P_j < P_i$ and at least one t_k with $P_k \geq P_i$
- $C(r)$ is the worst case computation time in critical section r

a task can only be blocked once for each employed resource!

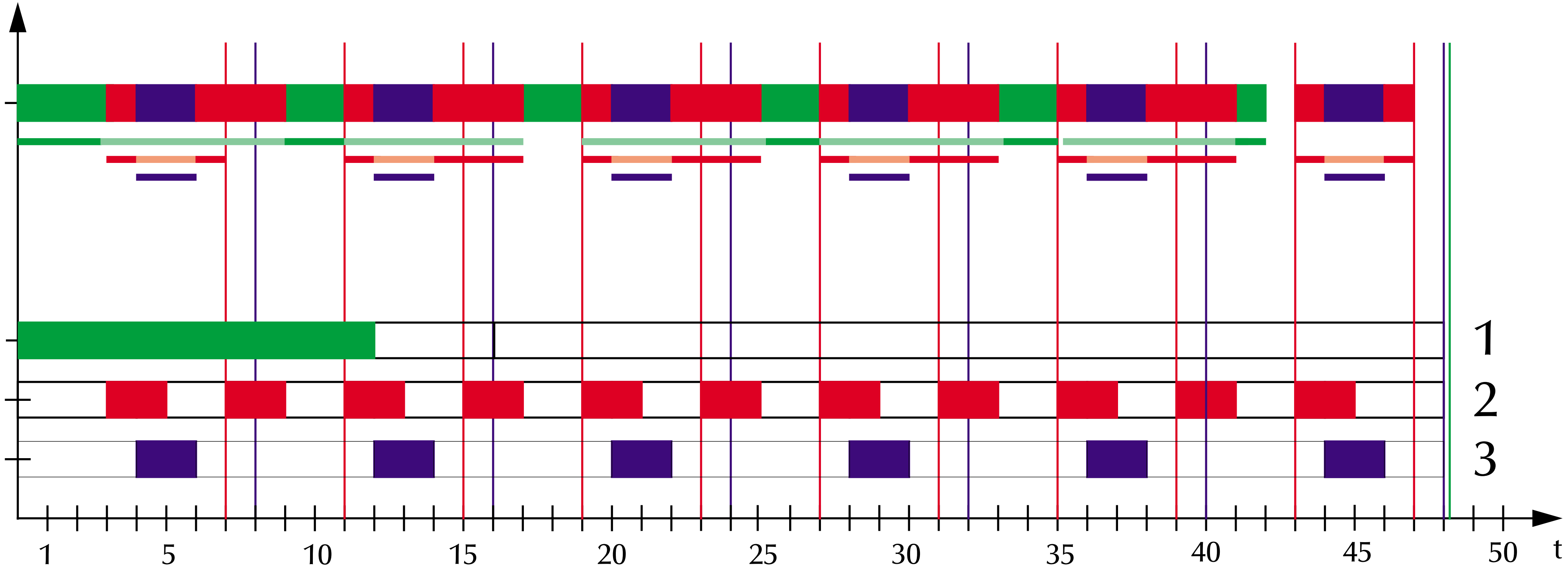
Priority inheritance



(🟩 inherits priority of 🟪 , when 🟩 is in lock and 🟪 is dispatched)

Scheduling: Interdependencies

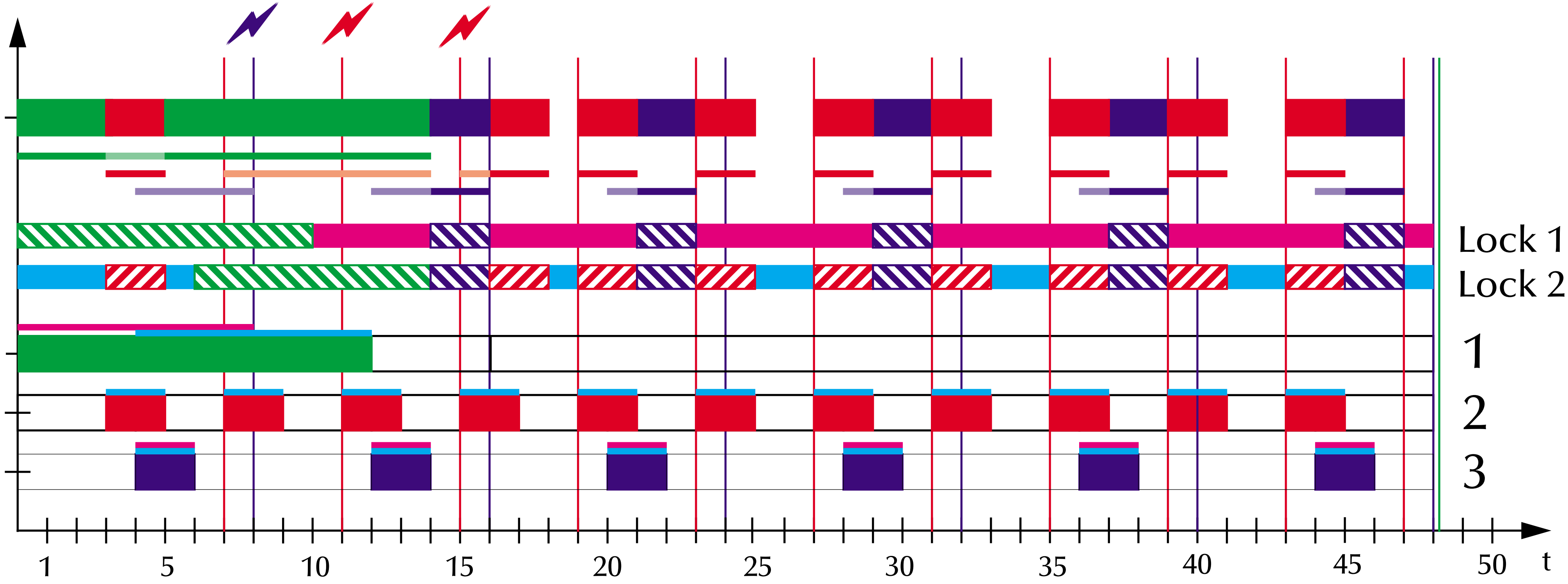
A more complex example



(independent task set)

Scheduling: Interdependencies

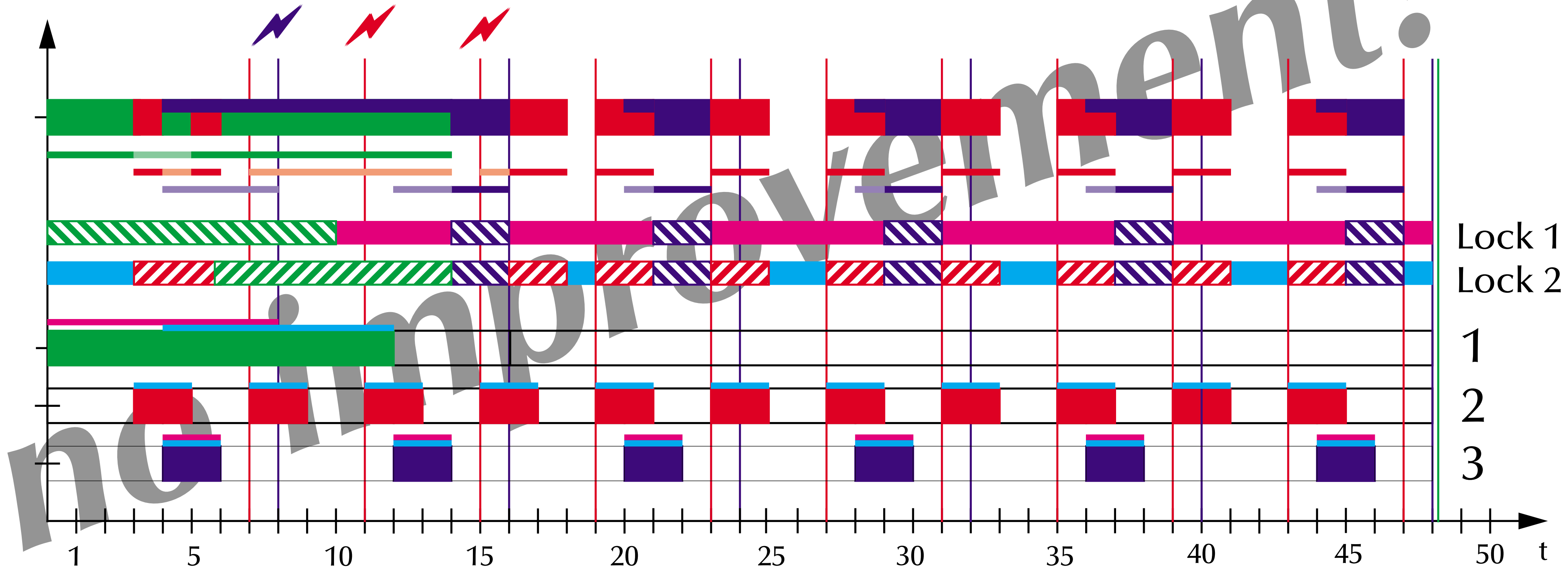
Interdependencies



☞ Priority inversion

Scheduling: Interdependencies

Priority inheritance

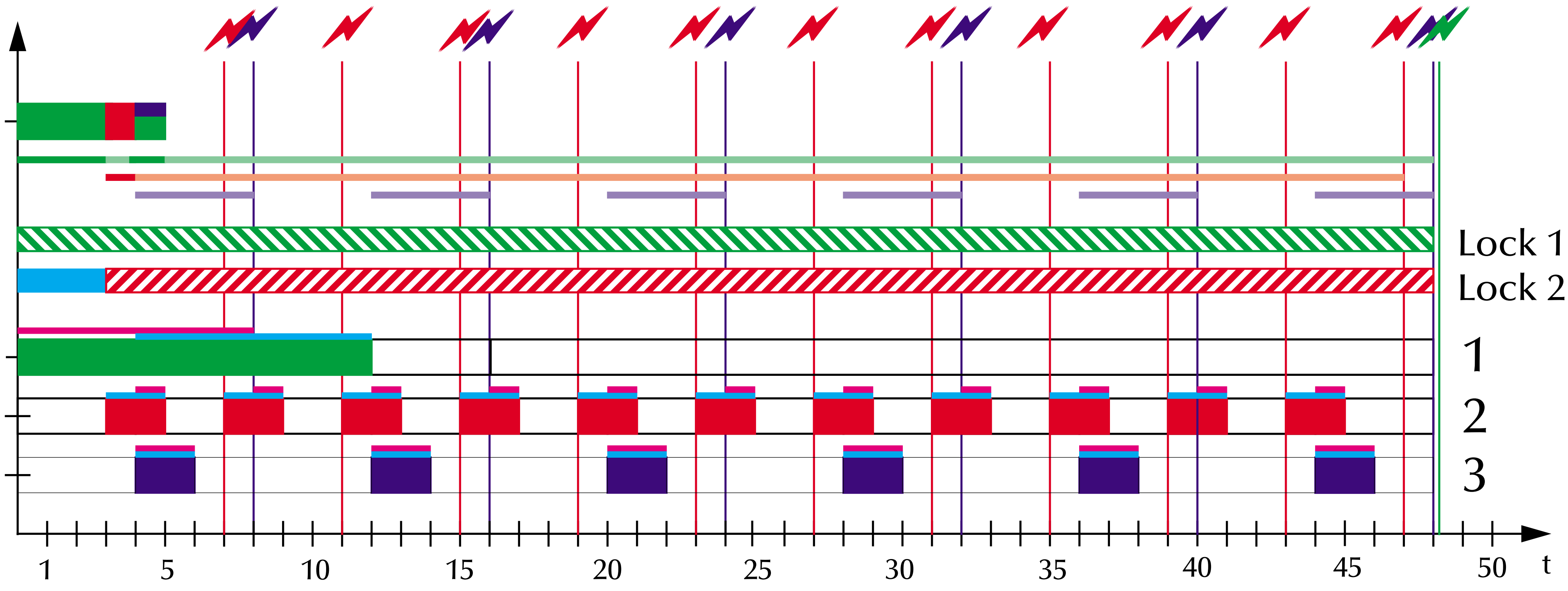


no improvement!

(🟩 and 🟥 inherit priority of 🟪, when in lock and 🟪 is dispatched)

Scheduling: Interdependencies

One additional lock request



☞ *Deadlock*

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- Each task t_i has static default priority P_i.
- Each resource (lock, monitor) R_k has a static ceiling priority C_k, which is the maximum of priorities of the tasks t_i which employ this resource.

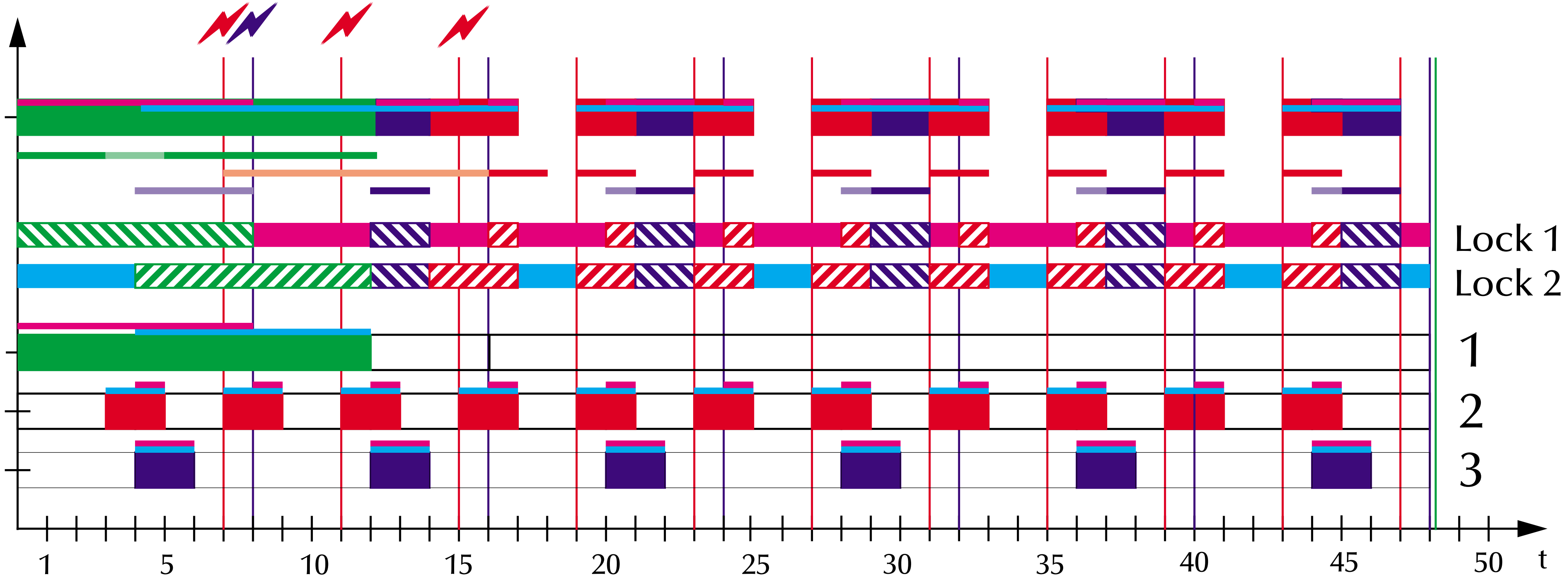
$$C_k = max_i\{employ(i, k) \cdot P_i\}$$

- Each task t_i has a dynamic priority P_i^D, which is the maximum of its own static priority and the ceiling priorities of any resource it has locked.

$$P_i^D = max\{P_i, max_k\{locked(i, k) \cdot C_k\}\}$$

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol *(POSIX, Ada, RT-Java)*



(🟩 , 🟥 and 🟪 inherit the ceiling priority of ■ or 🟦 when entering the lock)

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- ☞ Tasks are dispatched only if all employed resources are available.
- ☞ Deadlocks are prevented
- ☞ Number of context switches is reduced

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol *(POSIX, Ada, RT-Java)*

$$\text{Maximal blocking time: } B_i = max_{r=1}^{R}\{usage(r, i) \cdot C(r)\}$$

- with R the number of critical sections
- $usage(r, i)$ a boolean (0/1) function indicating that r is used by at least one t_j with $P_j < P_i$ and at least one t_k with $P_k \geq P_i$
- $C(r)$ is the worst case computation time in critical section r

a task can only be blocked once by any lower priority task!

Scheduling

- **Basic performance based scheduling**
  - *C_i is not known*: first-come-first-served (FCFS), round robin (RR), and feedback-scheduling
  - *C_i is known*: shortest job first (SJF), highest response ration first (HRRF), shortest remaining time first (SRTF)-scheduling
- **Basic predictable scheduling**
  - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO)
  - Earliest Deadline First (EDF)
- **Real-world extensions**
  - Aperiodic, sporadic, soft real-time tasks
  - Synchronized talks (priority inheritance, priority ceiling protocols)

Processes

- **Processes and threads**
    - Architectures, definitions, process states
- **Synchronization**
    - Shared memory based synchronization
    - Message based synchronization
- **Deadlocks**
    - Detection, avoidance, and prevention (& recovery)
- **Scheduling**
    - Basic performance based scheduling
    - Basic predictable scheduling
    - Aperiodic, sporadic, and synchronized tasks