Multi-Thread Code Generation

Paul Caspi

Verimag-CNRS

(joint work with Norman Scaife, Stavros Tripakis, Christos Sofronis)

- Why and when ?

- How ?

Single Thread Code Generation

Allows generating code for any discrete-time model that can be simulated

Allows many optimisations

The need for Real-Time Operating System is minimised

Provides in general robust and efficient code

But in some cases it is very inefficient and even not possible:

need for multi-thread code generation

Multi-Periodic Systems

Models are based on null execution times

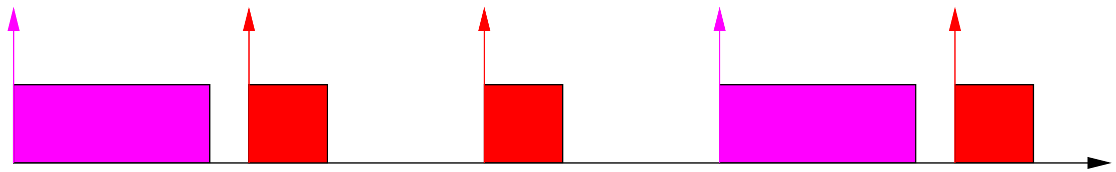
But implementations take time !!

Example:

- period (3,0)
- period(1,0)

single-thread code generation:

can yield:



Multi-Periodic Systems

Models are based on null execution times

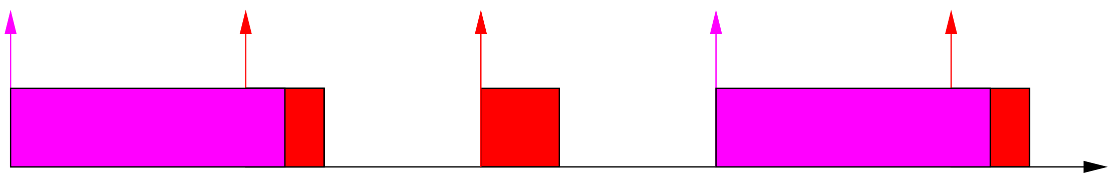
But implementations take time !!

Example:

- period (3,0)
- period(1,0)

single-thread code generation:

can yield even worse



Multi-Periodic Systems

Models are based on null execution times

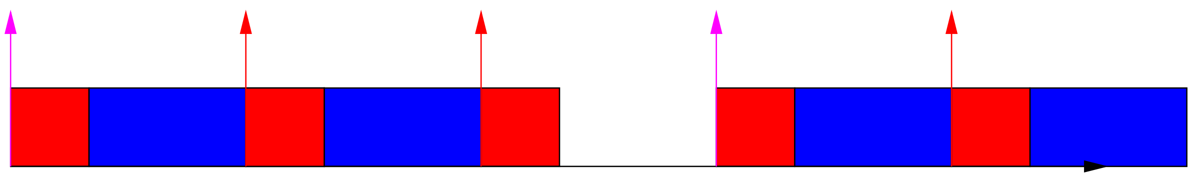
But implementations take time !!

Example:

- period (3,0)
- period(1,0)

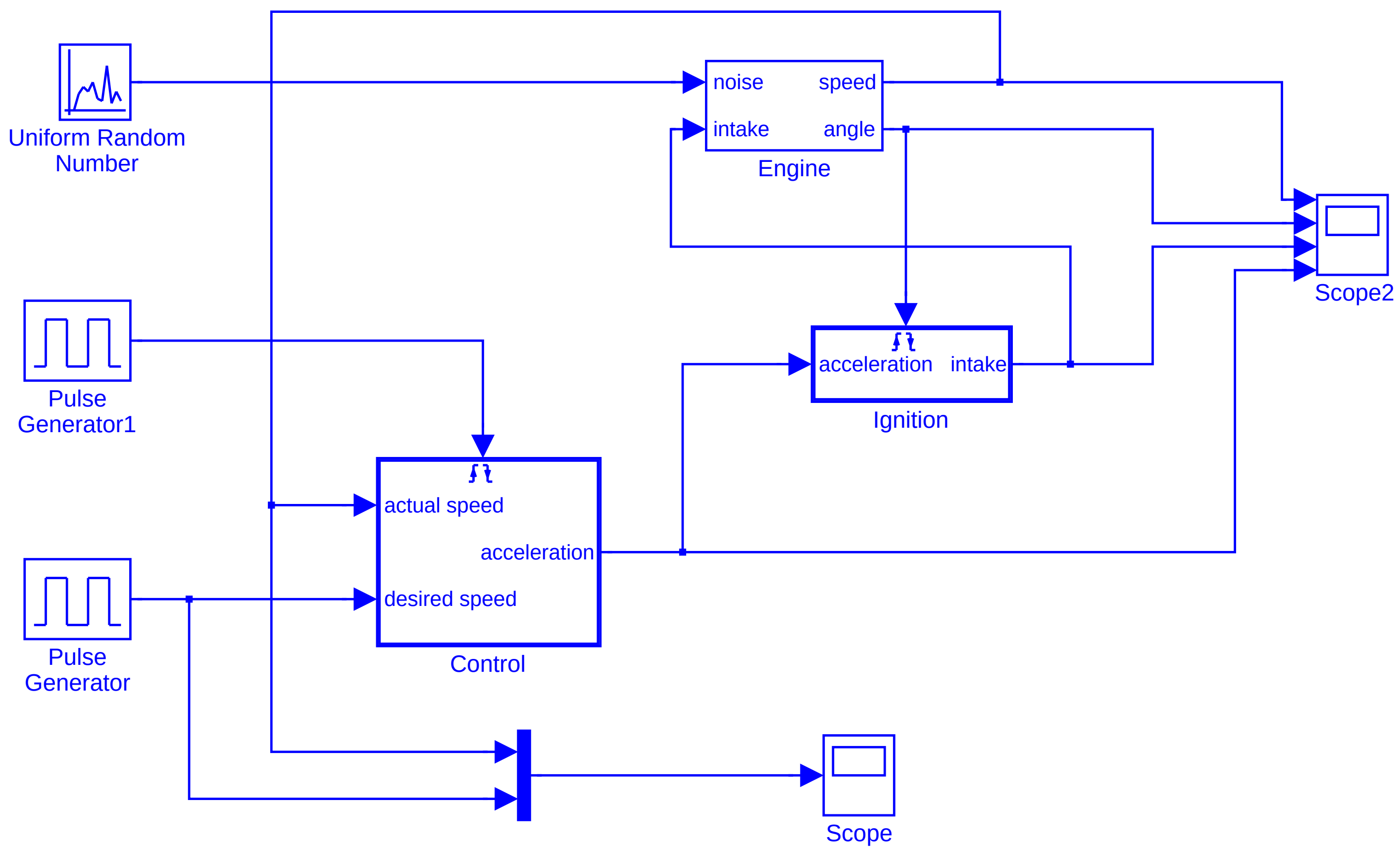
multi-thread code generation:

and preemptive scheduling can yield



Event and time-triggered systems

An engine control example:



Characteristics of the model

Based on several idealisations:

- The engine model is more or less accurate
- Computations are exact
- Computations take no time (synchronous abstraction)

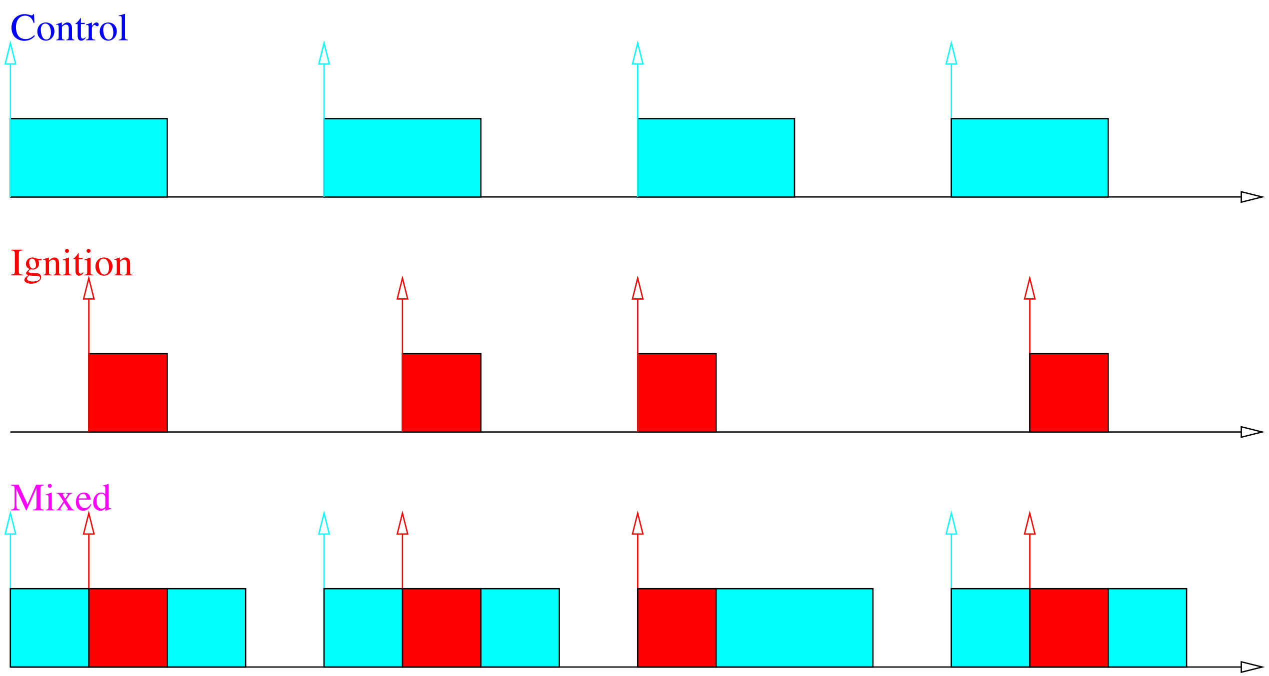
Implementation approximations

- Bounds on computation errors.
- Deadlines on executions

Domain dependent

Preemptive scheduling

If the deadline associated with event-triggered computations is smaller than the execution time of time-triggered tasks, preemptive scheduling is mandatory:



A Solution : Deadline Monotonic Scheduling

Schedulability test: formula of response times $$R_j = \sum_{i=1,j-1} \left\lceil \frac{R_j}{T_i} \right\rceil C_i + C_j$$

- thread priorities in decreasing order
- T_i minimum inter-arrival time of thread i
- C_i: worst case execution times of thread i
- $\left\lceil \frac{R_j}{T_i} \right\rceil$:number of times j can be preempted by i while executing
- $\left\lceil \frac{R_j}{T_i} \right\rceil C_i$: maximum time during which j can be preempted by i while executing
- The sum is taken on every thread with higher priority

A Solution : Deadline Monotonic Scheduling

Schedulability test: formula of response times $R_j = \sum_{i=1,j-1} \left\lceil \frac{R_j}{T_i} \right\rceil C_i + C_j$

R_j can be computed iteratively by

$$R_{j,0} = 0$$
$$R_{j,n+1} = \sum_{i=1,j-1} \left\lceil \frac{R_{j,n}}{T_i} \right\rceil C_i + C_j$$

until convergence

If D_j is the dead-line of thread j, ($D_j \leq T_j$), it suffices to verify for every j :

$$R_j < D_j$$

This schedulability test generalises Rate Monotonic Scheduling

Inter-task communication

Communication integrity, several approaches:

- Blocking approaches based on semaphores

  Priority inversion (pathfinder !!)

  priority inheritance, priority ceiling protocols

- Lock-free methods

- Loop-free, wait-free methods

  Burns et Chen (triple buffer)

  provide easier schedulability analysis ?

Bug of the Mars Pathfinder

semaphores

\+ RTOS

---

priority inversion

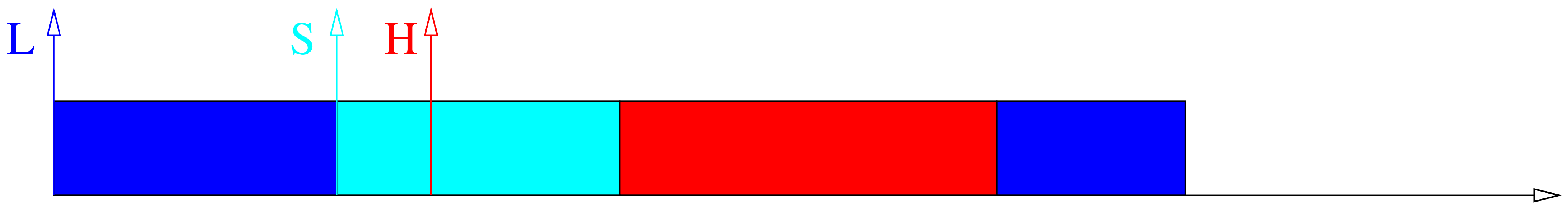
Semaphores

High and Low share a critical section

High wants to execute when Low is in critical section

High is stalled until Low gets out of the critical section

No Problem: the schedulability test can account for that



Priority Inversion

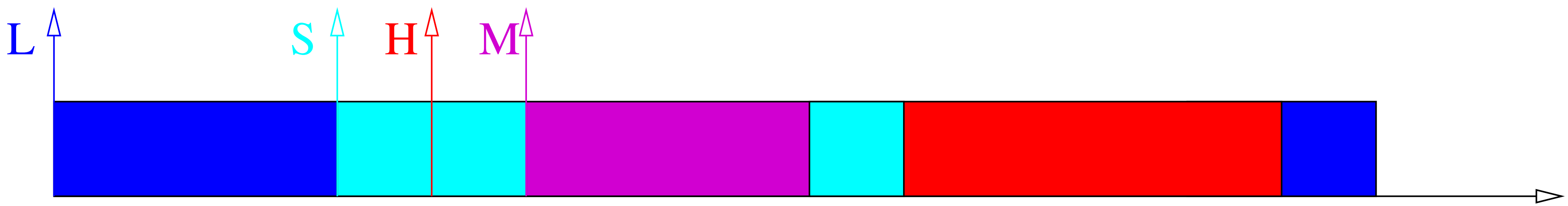
Medium doesn't share this critical section

Medium occurs when Low is in critical section

Medium preempts Low

High is stalled

Priority Inversion



What about semantics?

. . . and model-based development?

Preemption alters the ordering of computations

- In many cases it does not matter (robustness, continuity, faithfulness. . .)
- In some cases it can (discontinuities, critical races, . . .)

Can we propose executions that be functionally equivalent to the model?

Proposed solution

Ensures communication integrity and provides executions that are functionally equivalent to the model:

Based on:

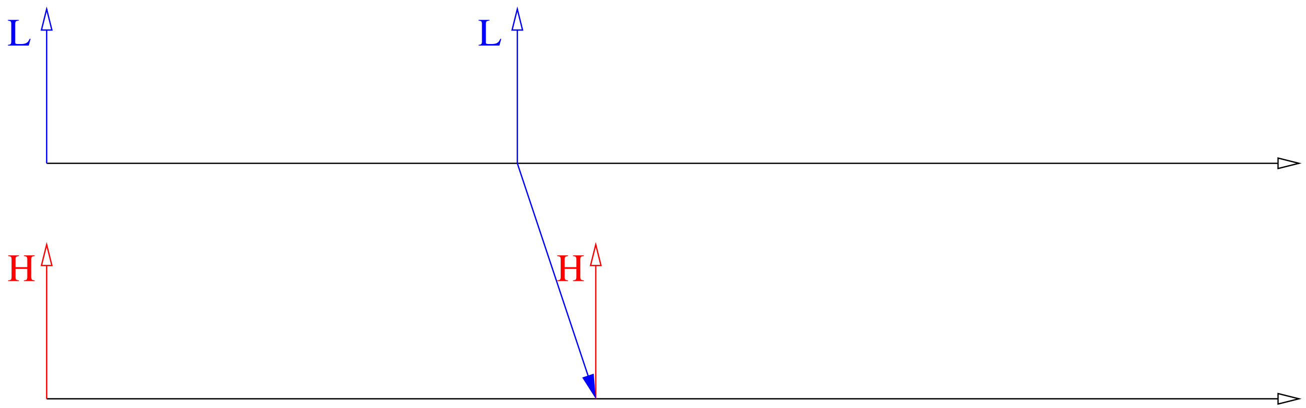
1. Syntactic checks: communications from low to high priority tasks should go through a unit delay on the low task trigger

2. Double buffer protocols where distinction is made between the occurrence of triggering events and the task executions

Why Is a Unit Delay Needed?

from Low to High:

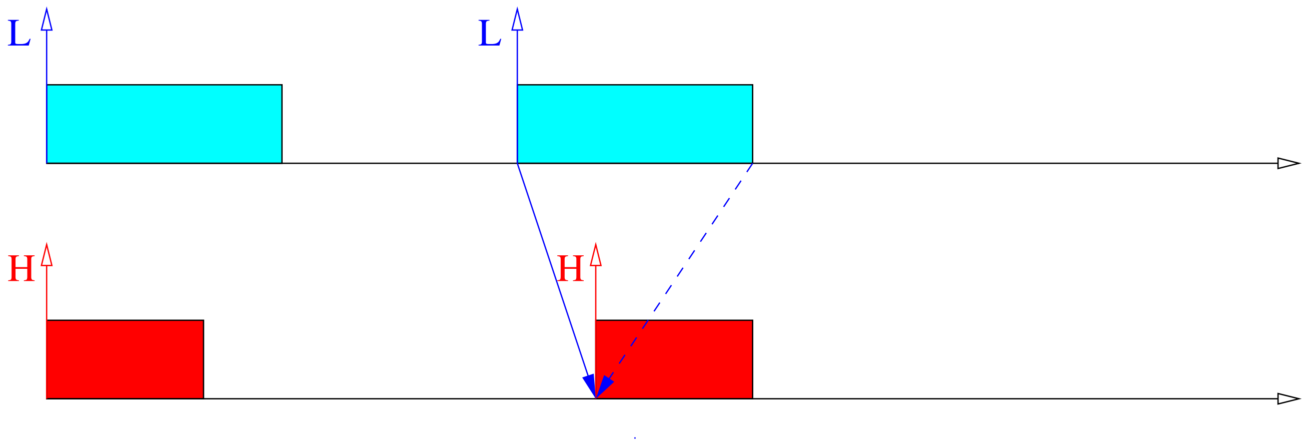
Ideal model communication without unit delay:



Why Is a Unit Delay Needed?

from Low to High:

Implemented communication without unit delay:

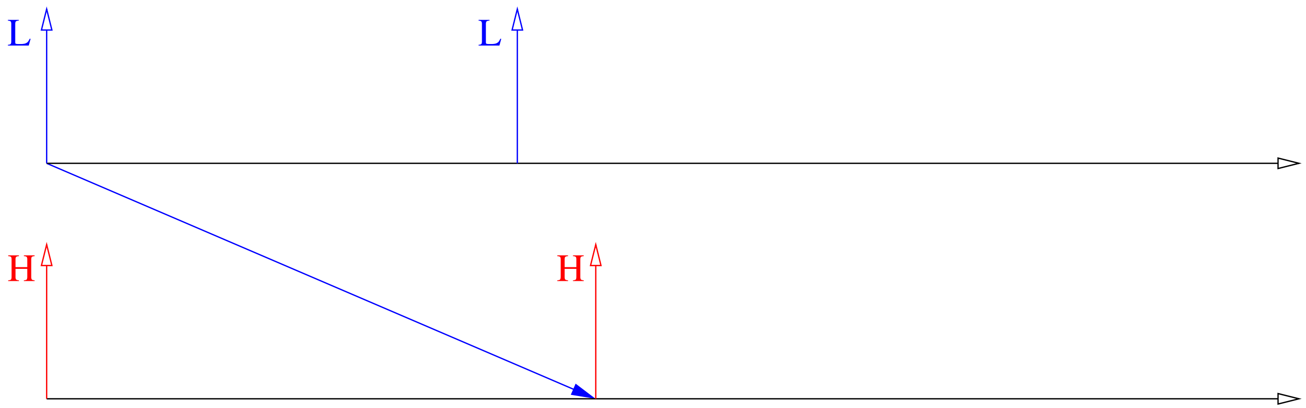


sometimes impossible

Why Is a Unit Delay Needed?

from Low to High:

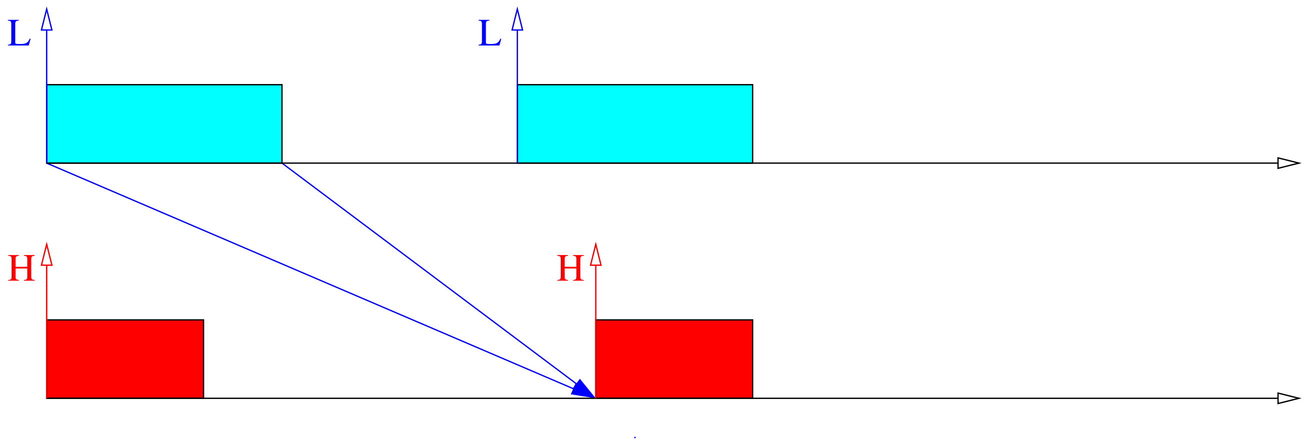
Ideal model with unit delay:



Why Is a Unit Delay Needed?

from Low to High:

Implemented communication with unit delay:



always possible

Double buffer protocol

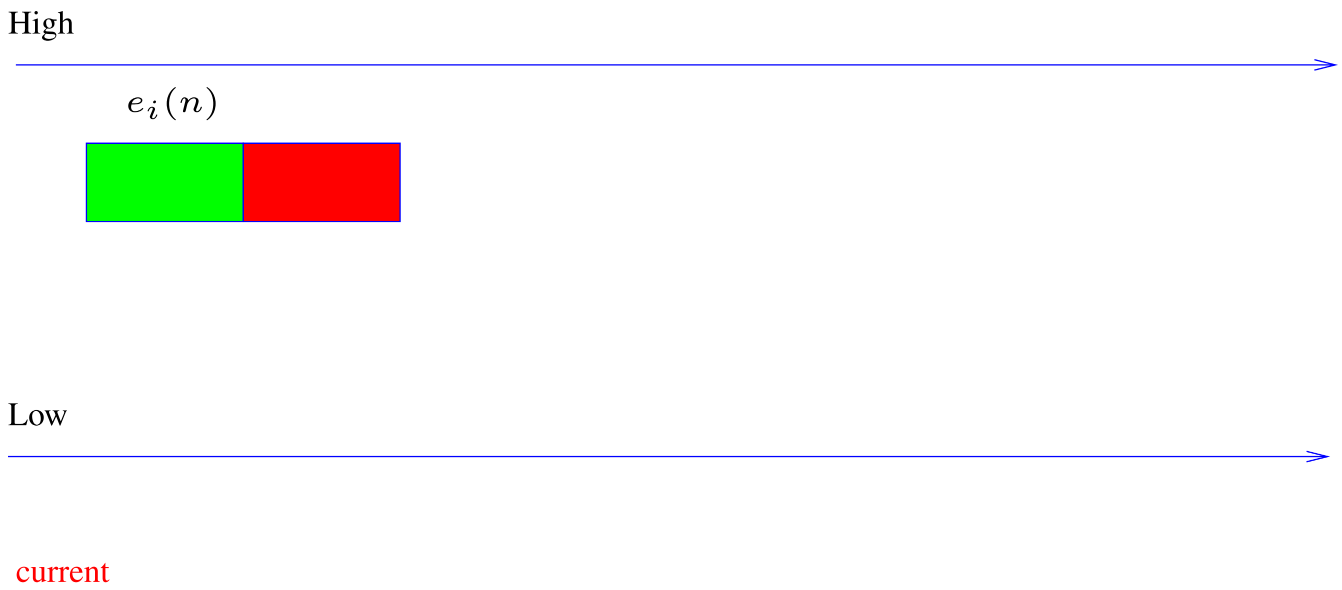
- From low to high
  - two buffers ("current" et "previous") managed by P_l, toggled when e_l takes place
  - when e_h occurs, P_h stores the address of "previous"
  - P_l writes to "current" et P_h reads into "previous"

- Bit toggling is assumed to take no time

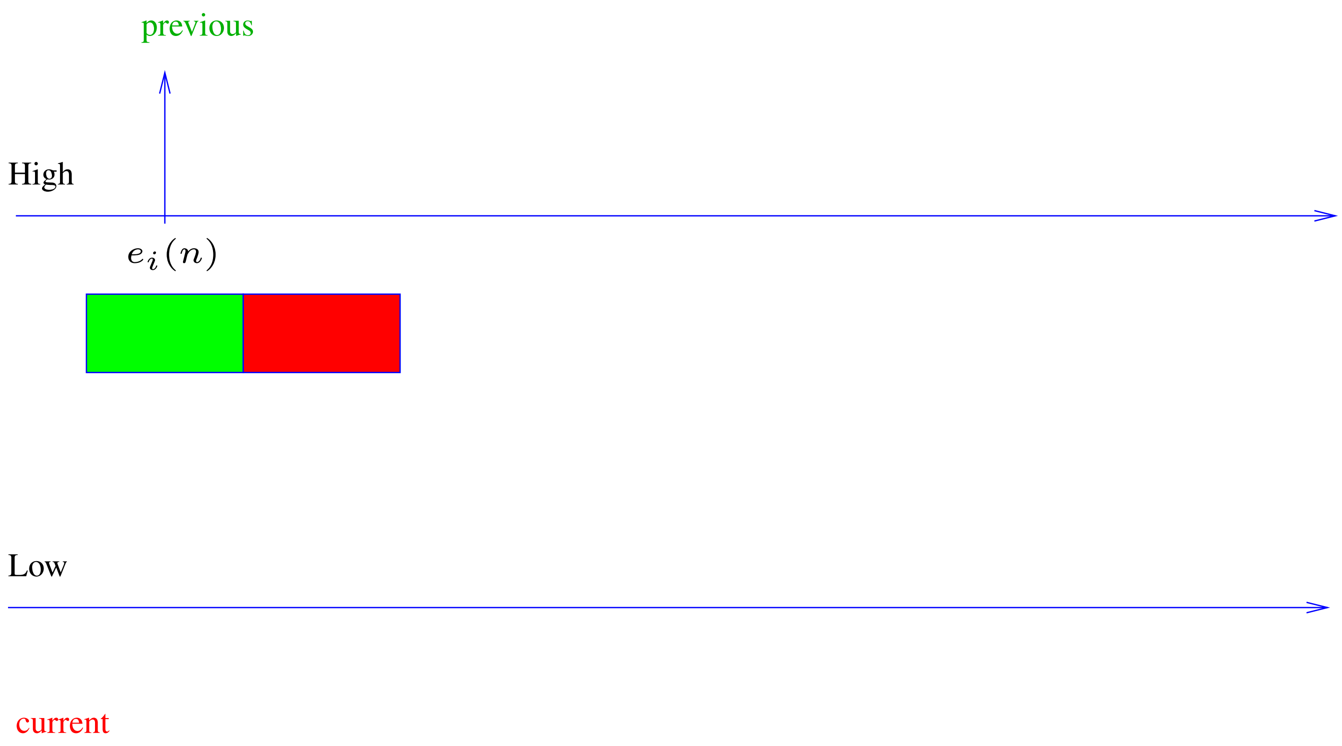
JAVA Implementation

```
public class LowToHigh extends Buffer{
  public LowToHigh(int ori, int dest,
                   Data odd1, Data even1){
    super(ori, dest, odd1, even1);
  }
  public void togglewrite(){
    current = !current;
  }
  public void toggleread(){
    previous = !current;
  }
}
```

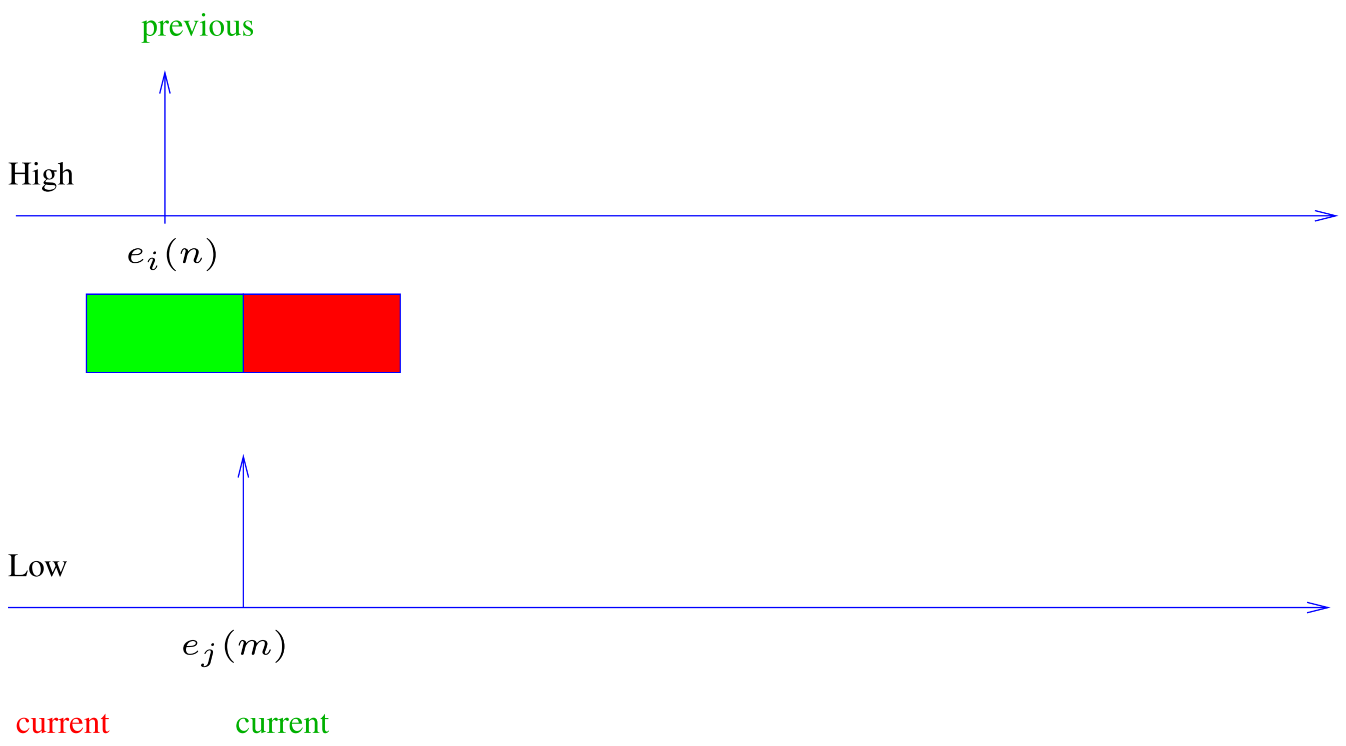
Low Priority to High Priority



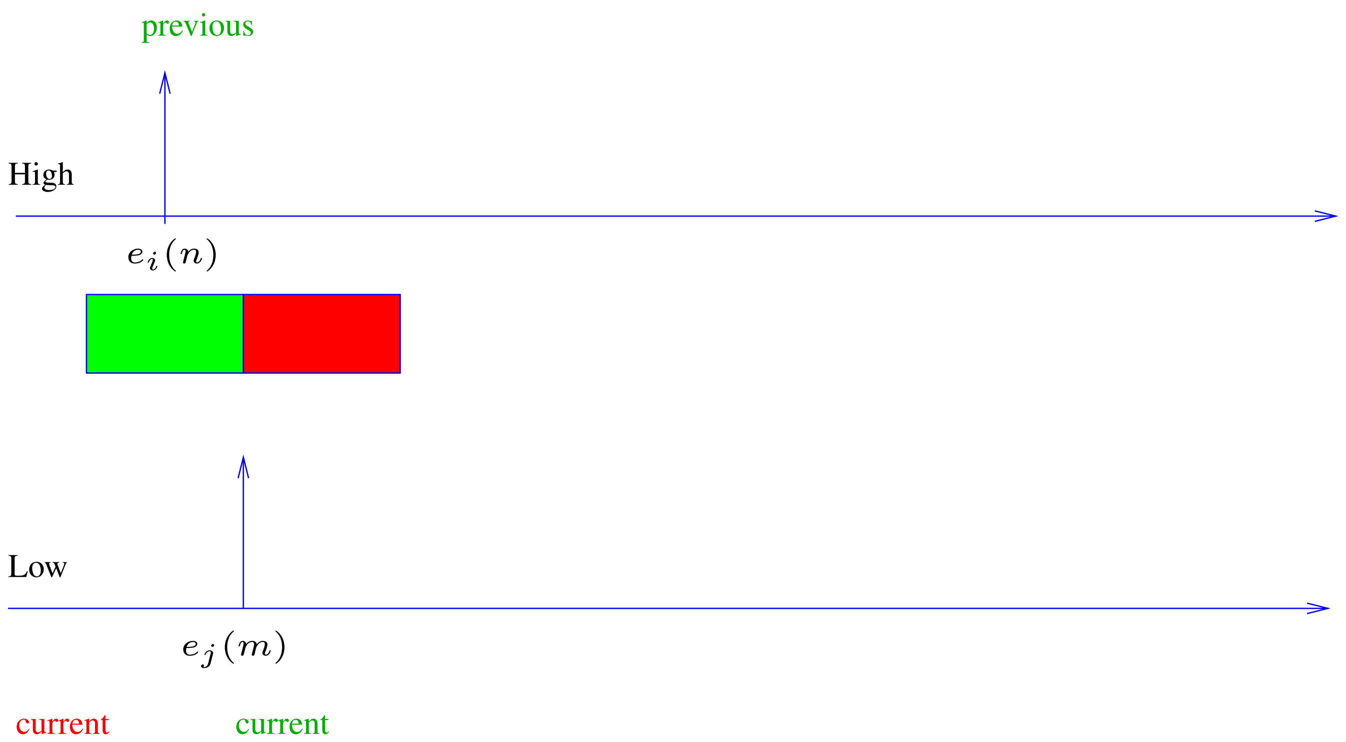
Low Priority to High Priority



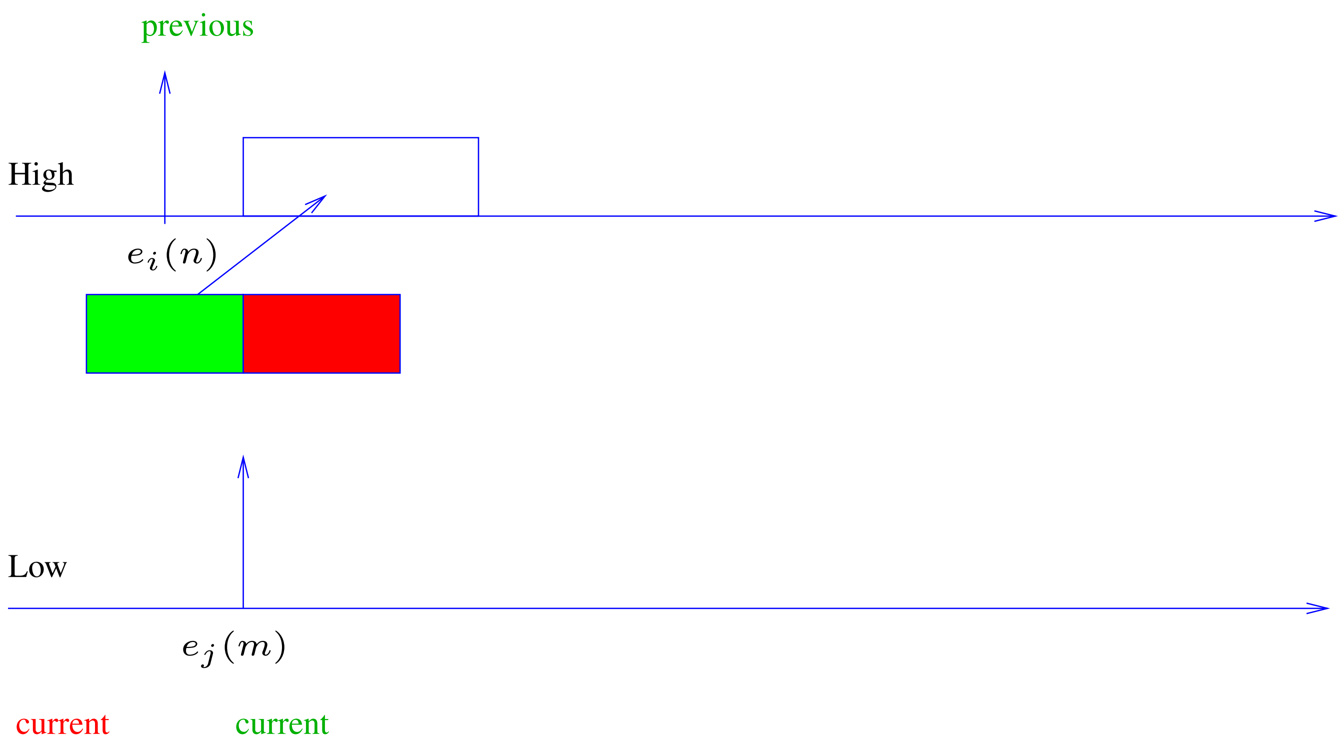
Low Priority to High Priority



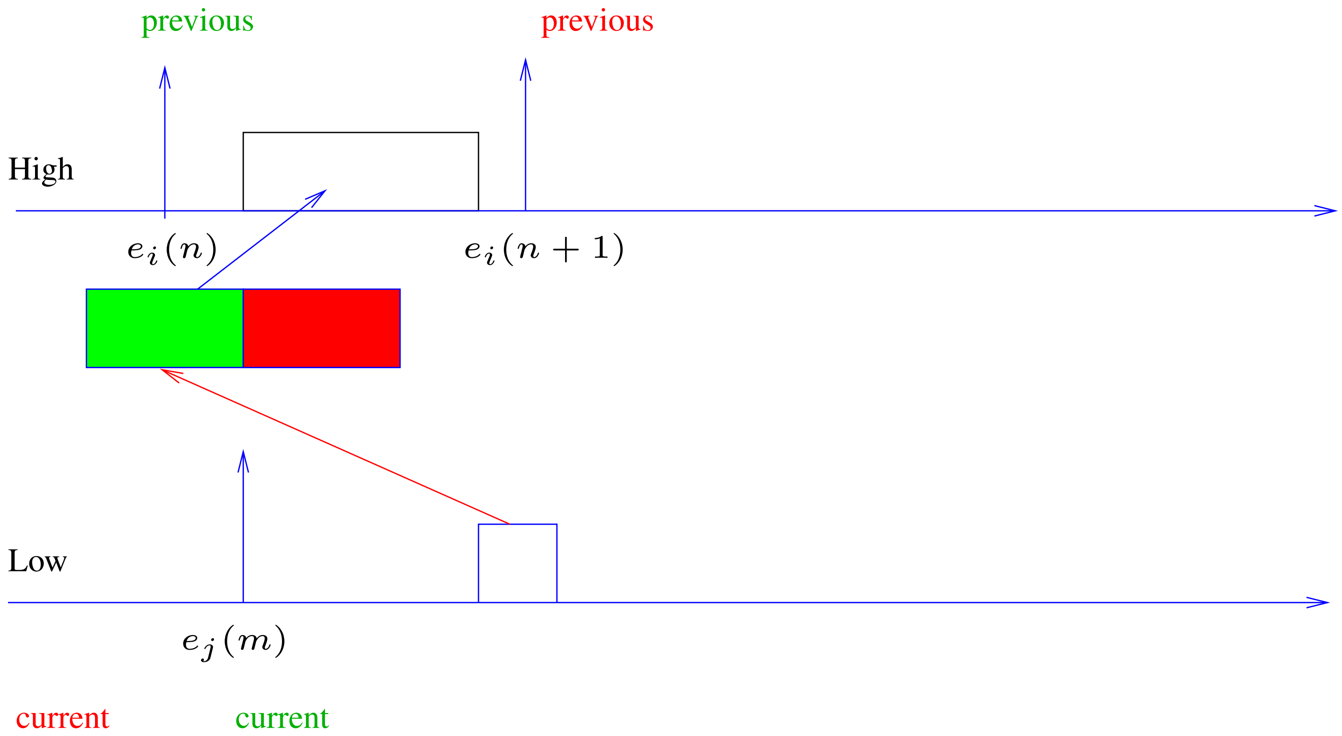
Low Priority to High Priority



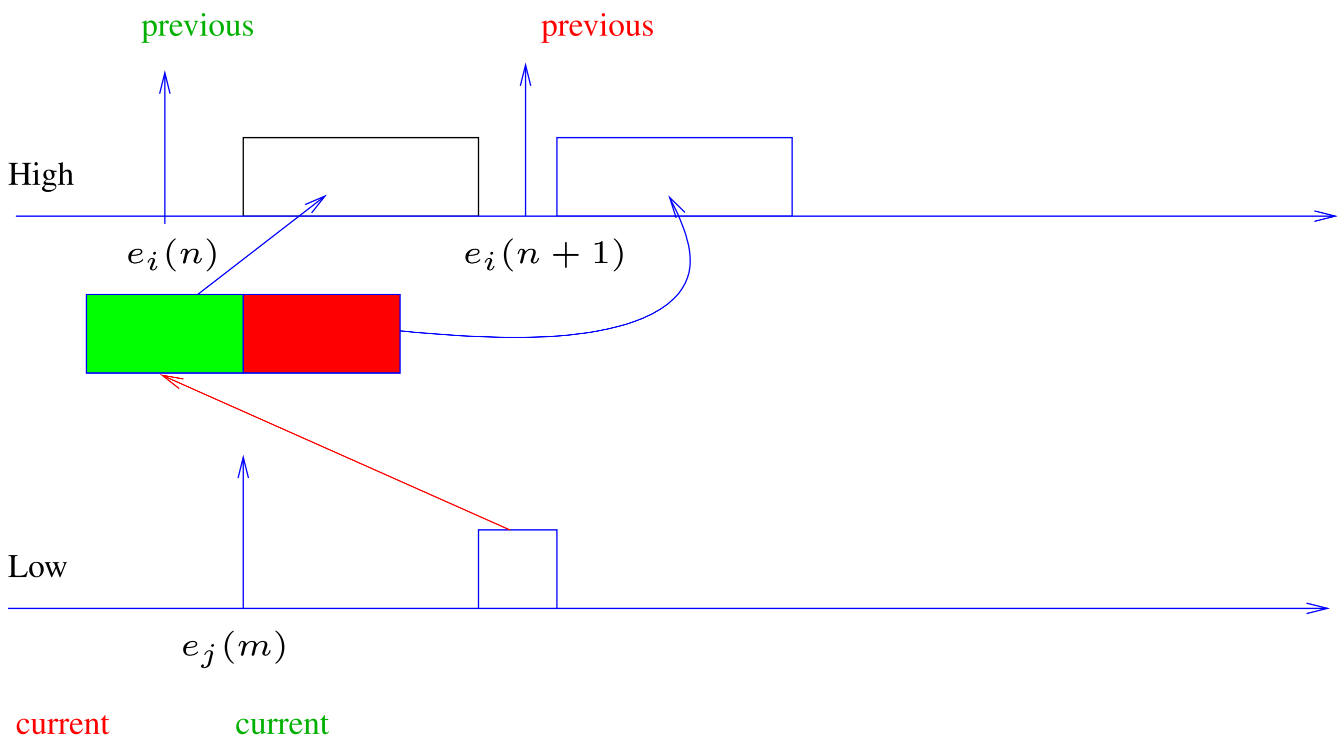
Low Priority to High Priority



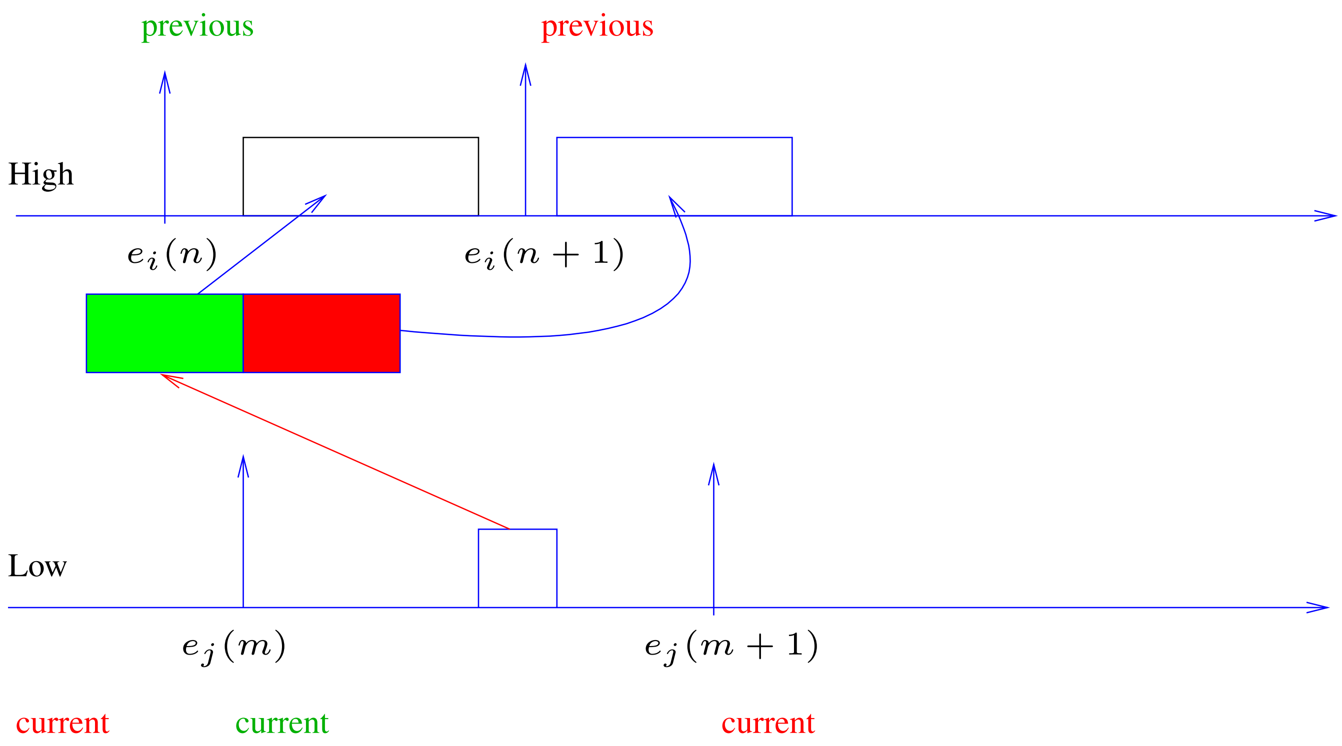
Low Priority to High Priority



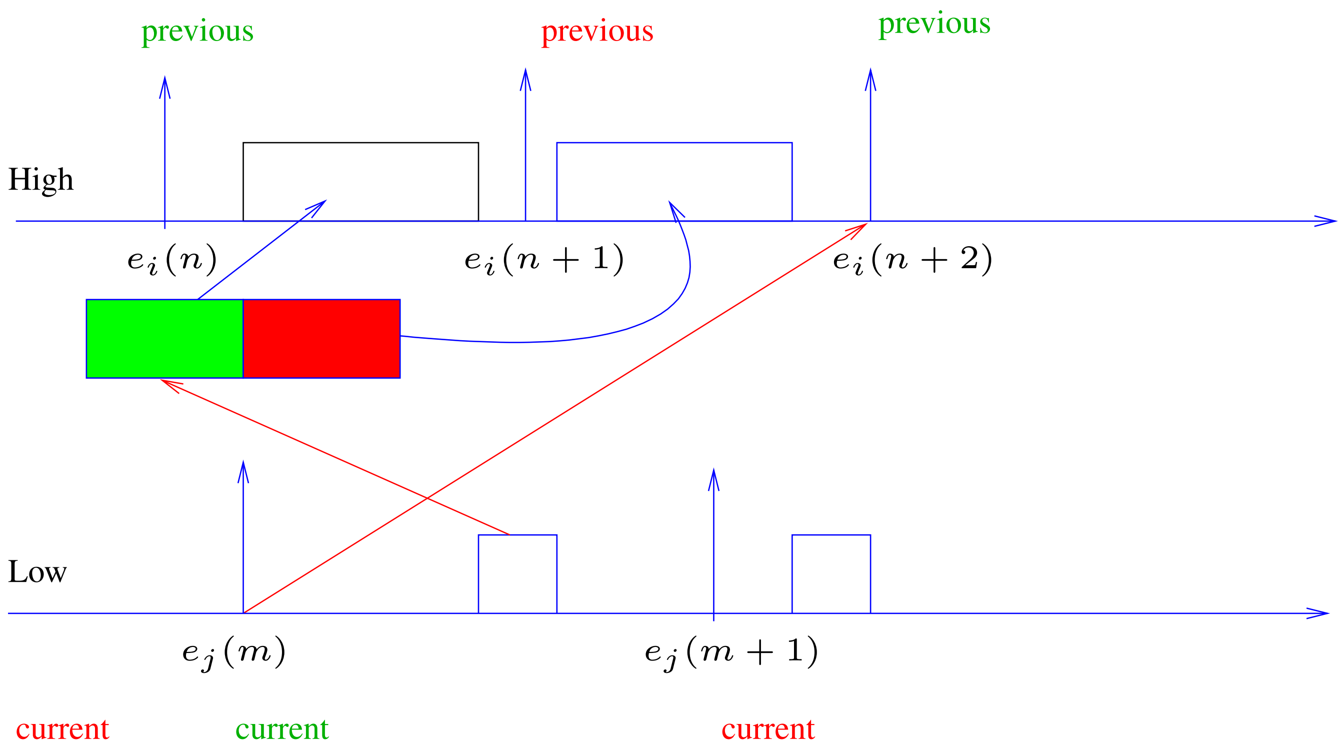
Low Priority to High Priority



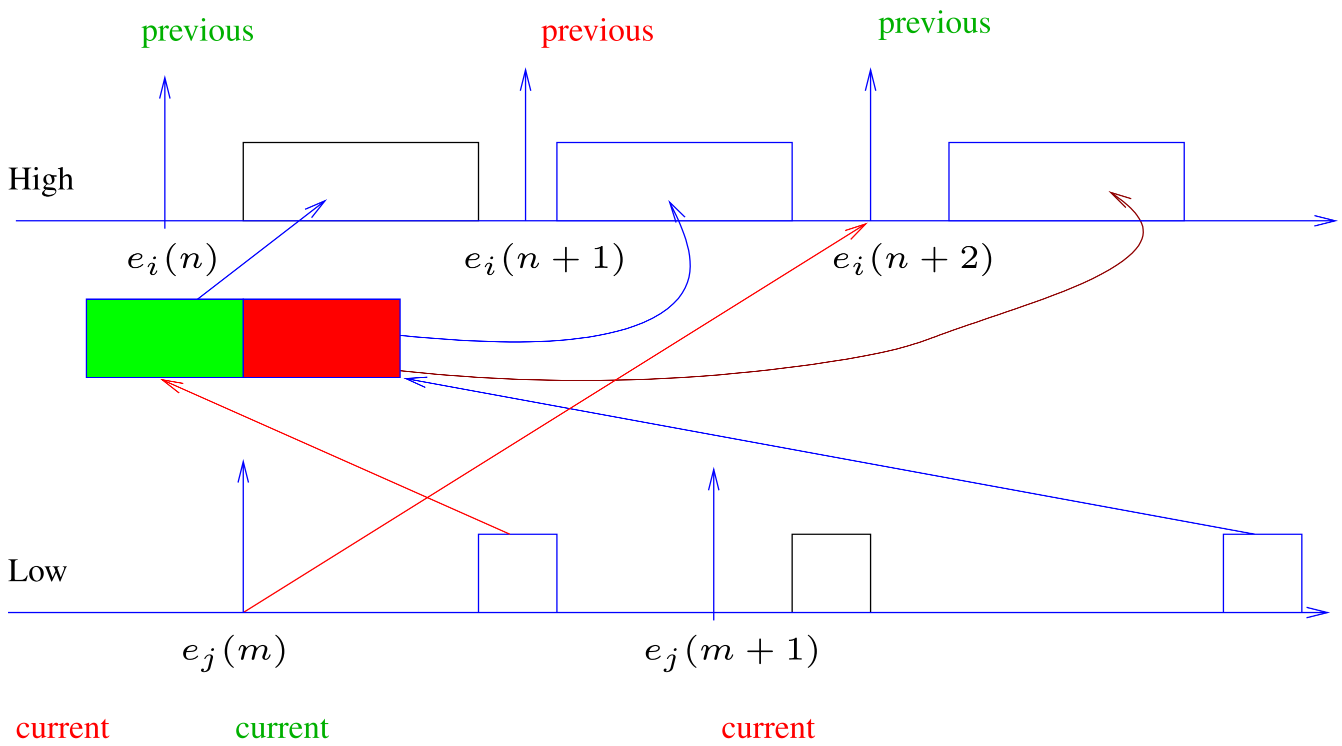
Low Priority to High Priority



Low Priority to High Priority



Low Priority to High Priority



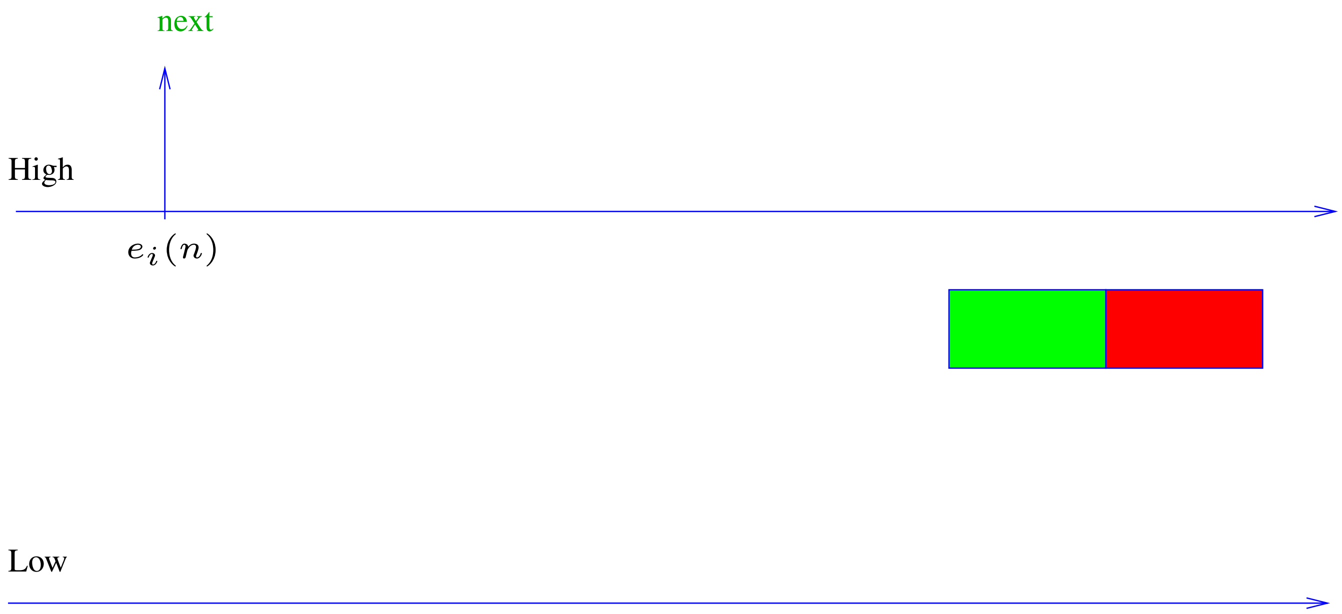
Double buffer protocol

- From high to low
  - double buffer (”current” et ”next”) managed by P_l
  - on e_l ”current” is set to ”next”
  - on e_h ”next” is toggled if ”current” equals ”next”
  - P_h writes to ”next” and P_l reads into ”current”
- Bit toggling is assumed to take no time

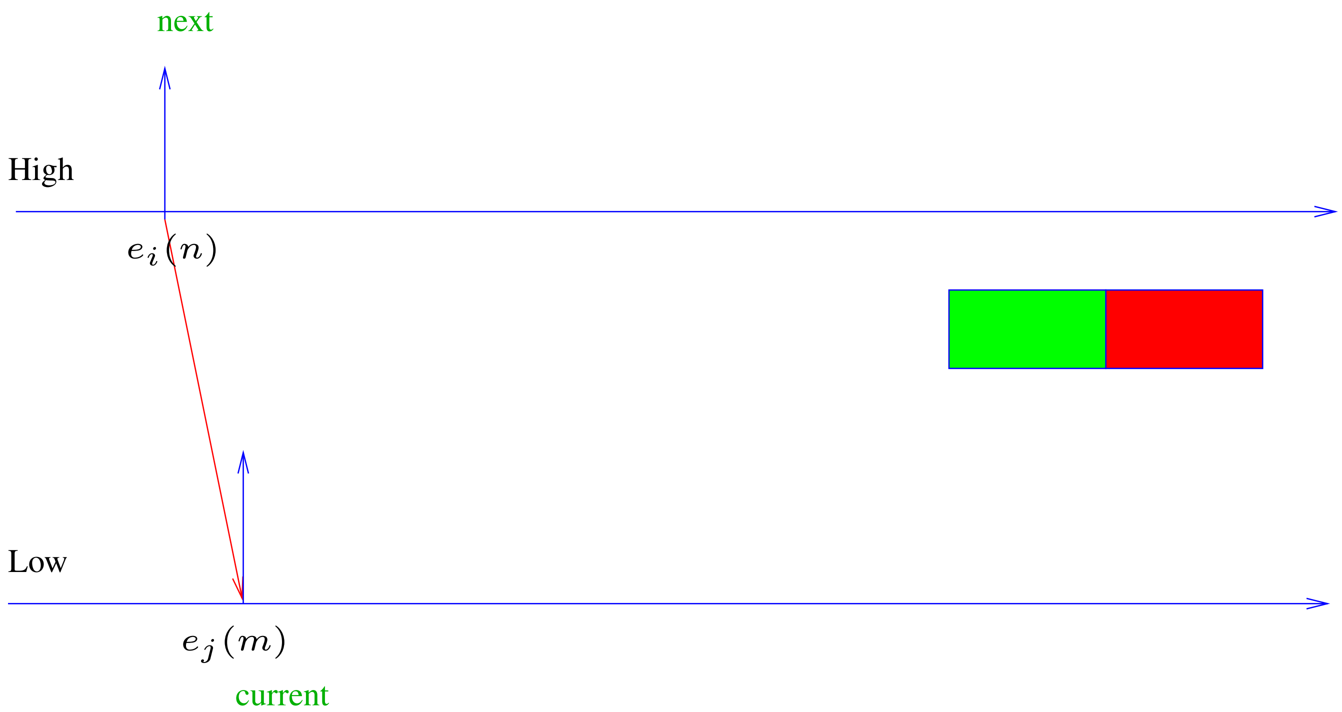
JAVA Implementation

```
public class HighToLow extends Buffer{
  public HighToLow(int ori, int dest,
                   Data odd1, Data even1){
    super(ori, dest, odd1, even1);
  }
  public void togglewrite(){
    if(current == next) next = !next;
  }
  public void toggleread(){
    current = next;
  }
}
```

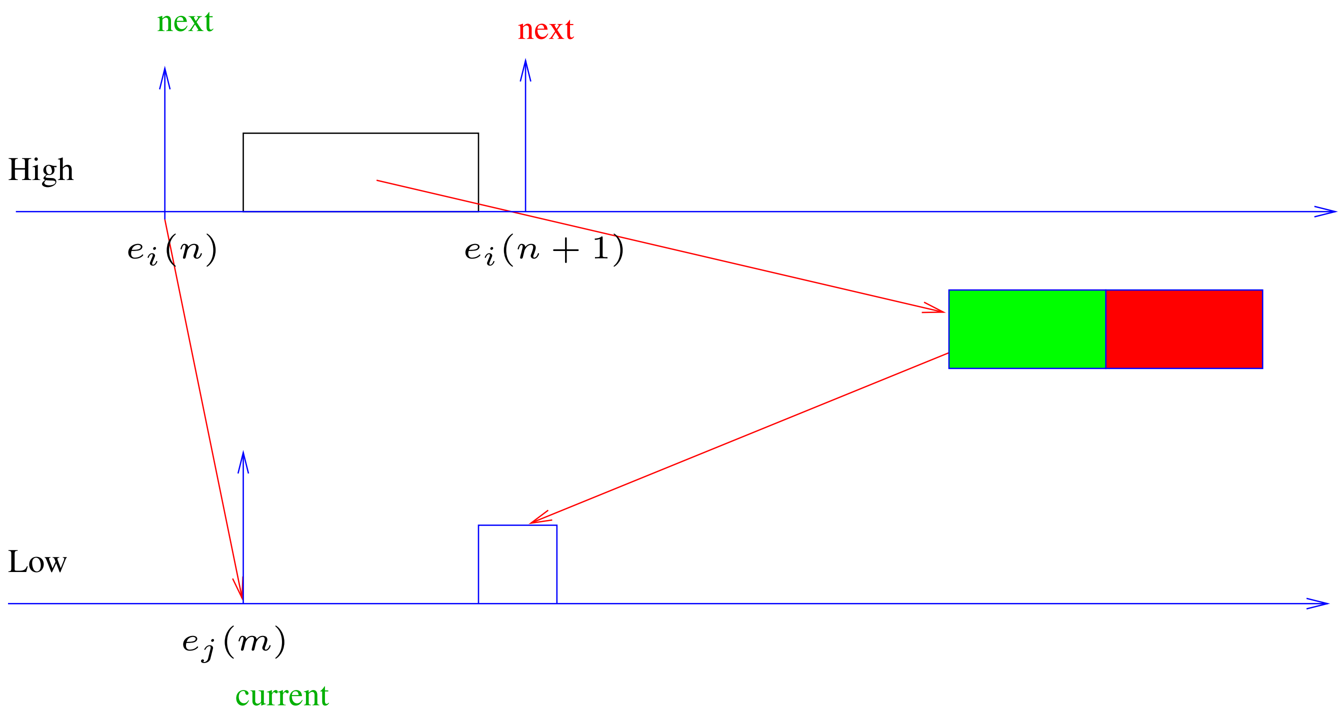
High Priority to Low Priority



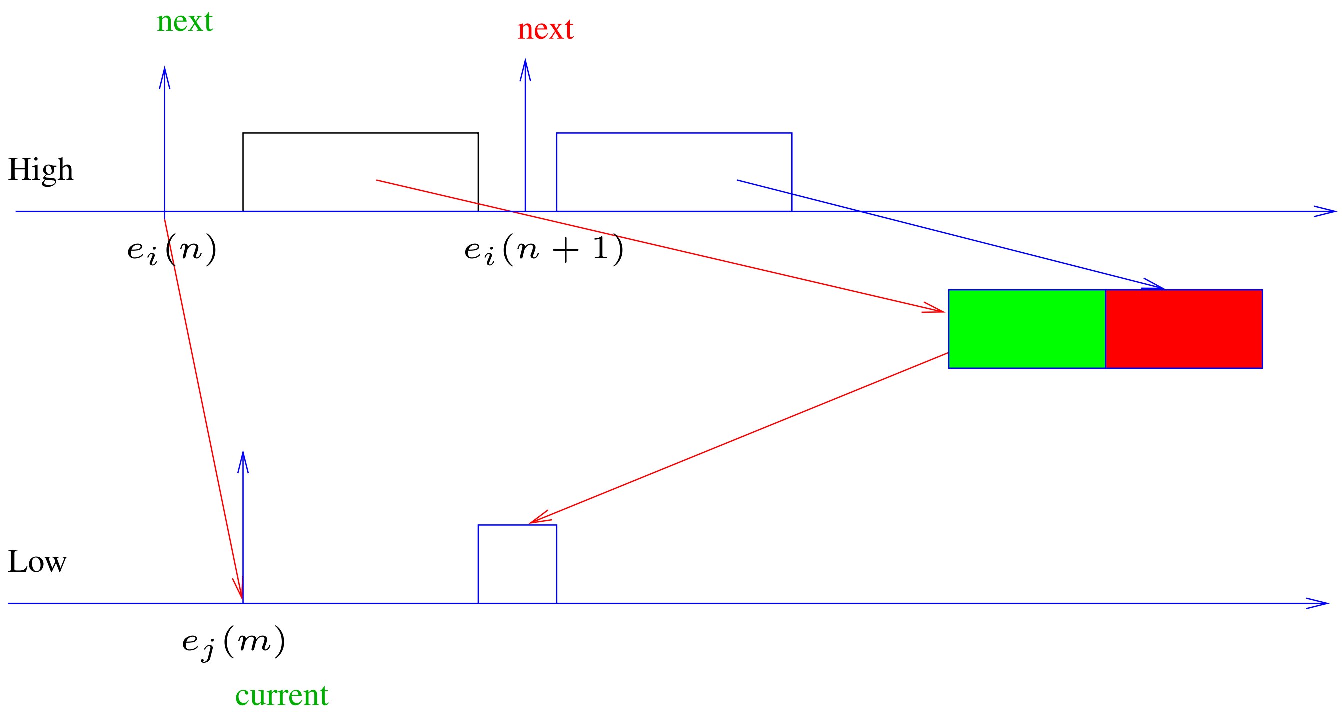
High Priority to Low Priority



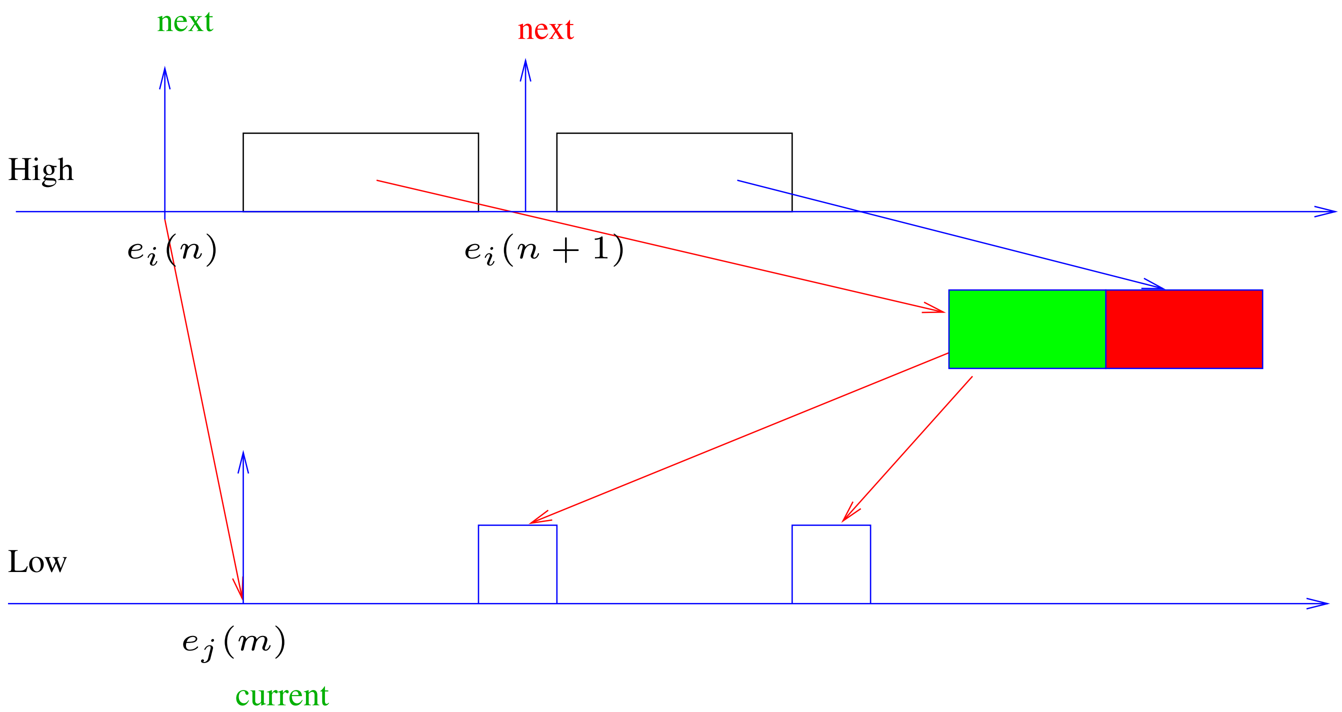
High Priority to Low Priority



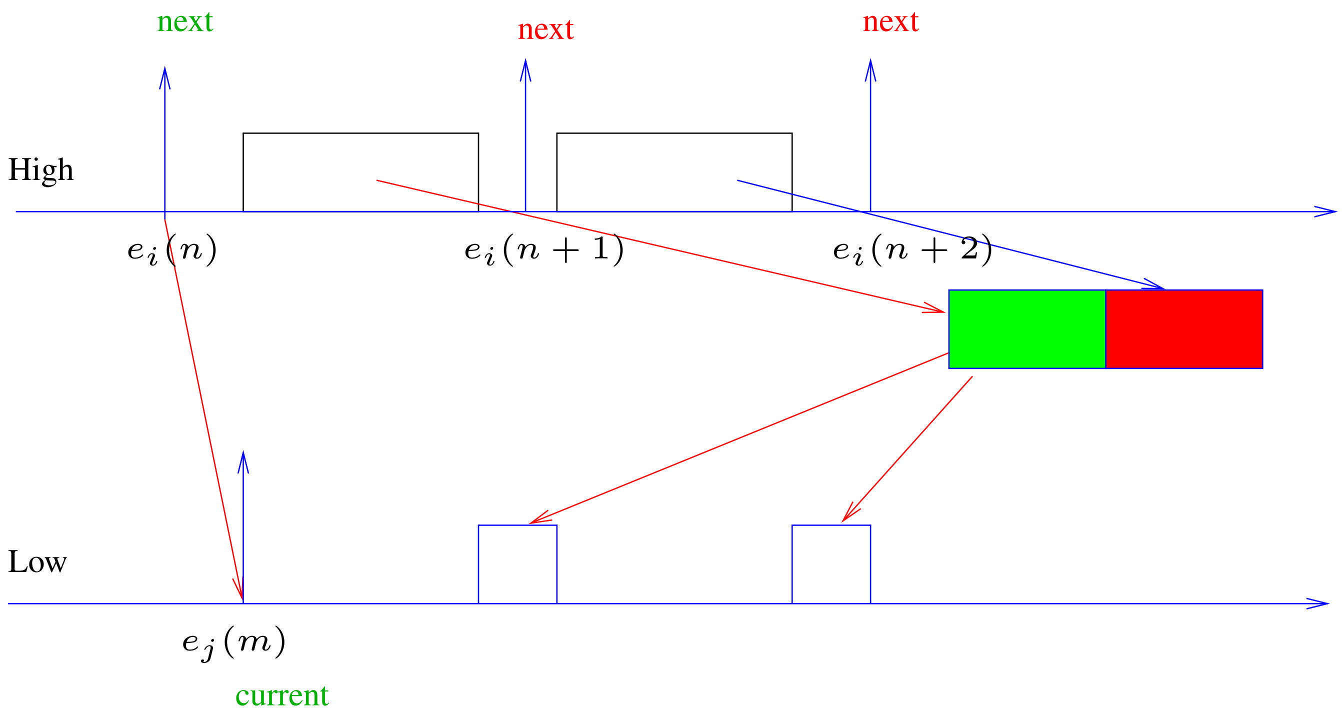
High Priority to Low Priority



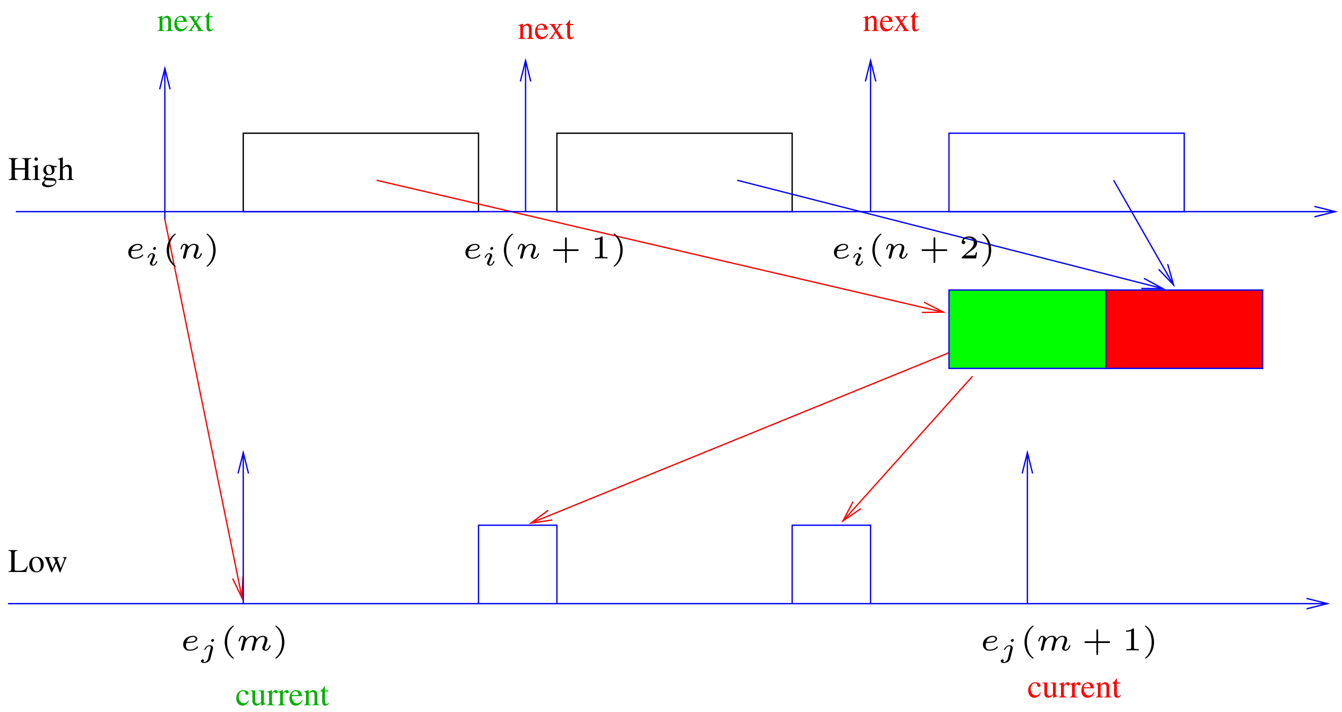
High Priority to Low Priority



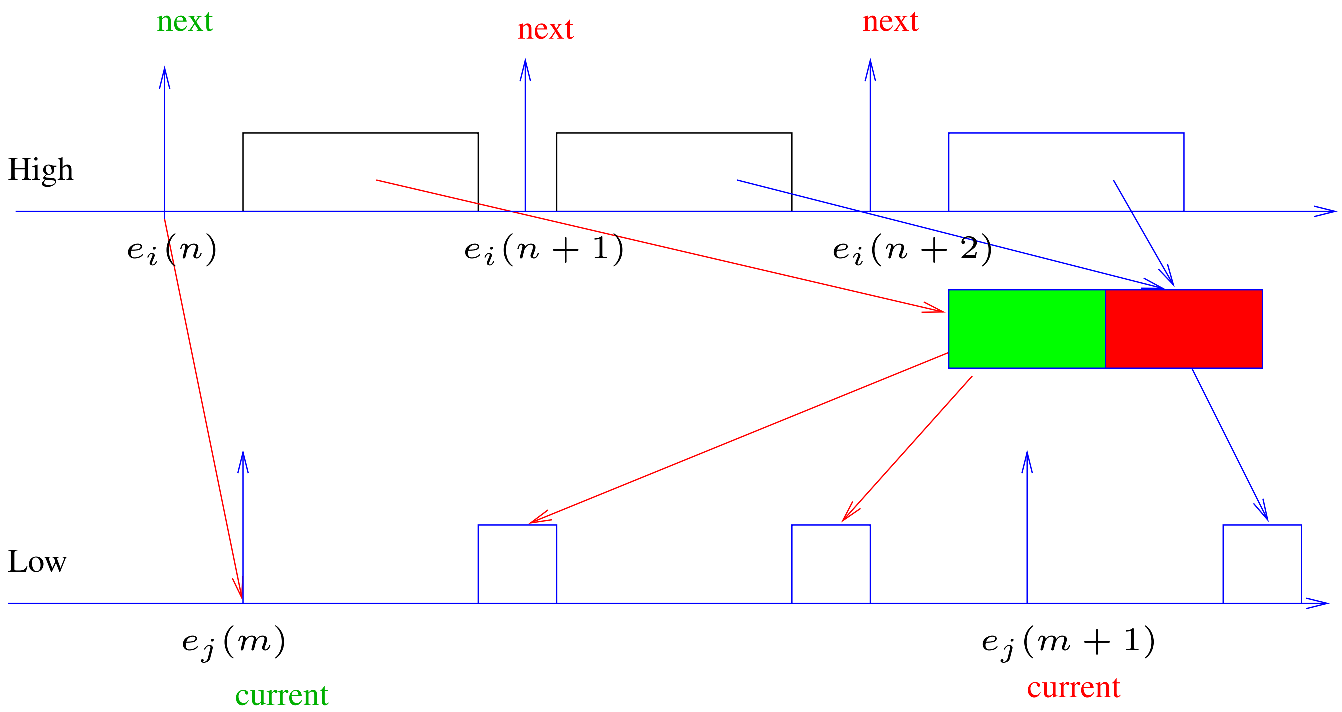
High Priority to Low Priority



High Priority to Low Priority



High Priority to Low Priority



Other Results

- Proof by Model-Checking

- Generalisation to EDF

  Works the same.

- Optimisation in the multi-periodic case

  n+1 is the number of buffers needed for a high priority task to communicate with n lower priority readers.

  (instead of 2n)

Proof by Model-Cheking

Model-checking with Lustre and Lesar

Principles:

- uninterpreted values and functions :
  boolean n-vectors
  $2^n > max\{$ number of values present in the system at a given time $\}$
- synchronous modelling of asynchronous systems
  events are input boolean flows constrained by assertions.

High to Low

```
node htlverif(val: bool^n; s1, sb1, se1, s2, sb2, se2: bool)
returns(prop: bool);
var ideal1, ideal2: bool^n;
let
  assert priority(s1, sb1, se1, s2, sb2, se2);
  ideal1 = if s1 then val
           else (init -> pre ideal1);
  ideal2 = if s2 then ideal1
           else (init -> pre ideal2);
  prop = if sb2
         then vecteq(ideal2, hightolowbuf(s1, s2, se1, ideal1))
         else true;
tel

# lesar verif.lus htlverif -v -diag -states 100000
DONE => 22489 states 88105 transitions
TRUE PROPERTY
```

Low to High Buffer

```
node lowtohighbuf(fromev, toev,  fromact: bool; fromval: bool^n)
returns (toval: bool^n);
var even,  odd: bool^n;
    bitfrom, bitto: bool;
let
  bitfrom = false -> if fromev then not pre bitfrom
                     else pre bitfrom;
  bitto = false -> if toev then not bitfrom
                     else pre bitto;
  even = if fromact and bitfrom then fromval
         else (init -> pre even);
  odd = if fromact and not bitfrom then fromval
        else (init -> pre odd);
  toval = if bitto then even
          else odd;
tel
```

Priority

```
-- s event occurrence
-- sb begin execution
-- se end of execution

node cyclic(s, sb, se: bool) returns (prop: bool);
let
  prop = after(s,  sb) and
    after(sb,  se) and
      after(se,  forgetfirst(s));
tel

-- s1 has higher priority than s2

node priority (s1, sb1, se1, s2, sb2, se2: bool)
returns (prop: bool);
let
  prop = cyclic(s1, sb1, se1) and
    cyclic(s2, sb2, se2) and
      neverbetween(s1, se1, sb2) and
        neverbetween(s1, se1, se2);
tel
```

Conclusion

- A simple protocol that gets preemptive implementations closer to (synchronous) models.

  Based on:

  - syntactic restrictions (unit delayed communications)
  - use of triggering events in buffer selection

- Several optimisations have been provided

Industrial Perspectives

There seems to be a clear industrial interest :

- Esterel-Technologies is currently prototyping the approach in the “Scade Drive” tool-box.

- Real-Time Workshop (Matlab) announces the same results (but unpublished)

- Parades (Roma) is currently exploring the same ways