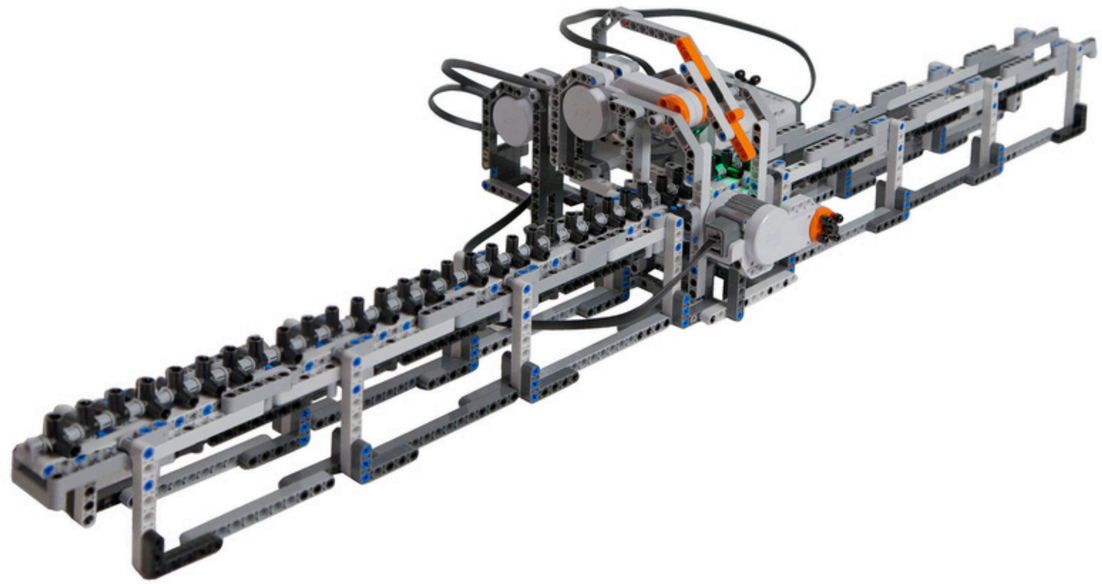
Compositional Sequentialization of Periodic Programs*

Soonho Kong
soonhok@cs.cmu.edu
Carnegie Mellon University

*Work with Sagar Chaki(SEI/CMU), Arie Gurfinkel(SEI/CMU), and Ofer Strichman(Technion - Israel Institute of Technology)

Target of Verification:

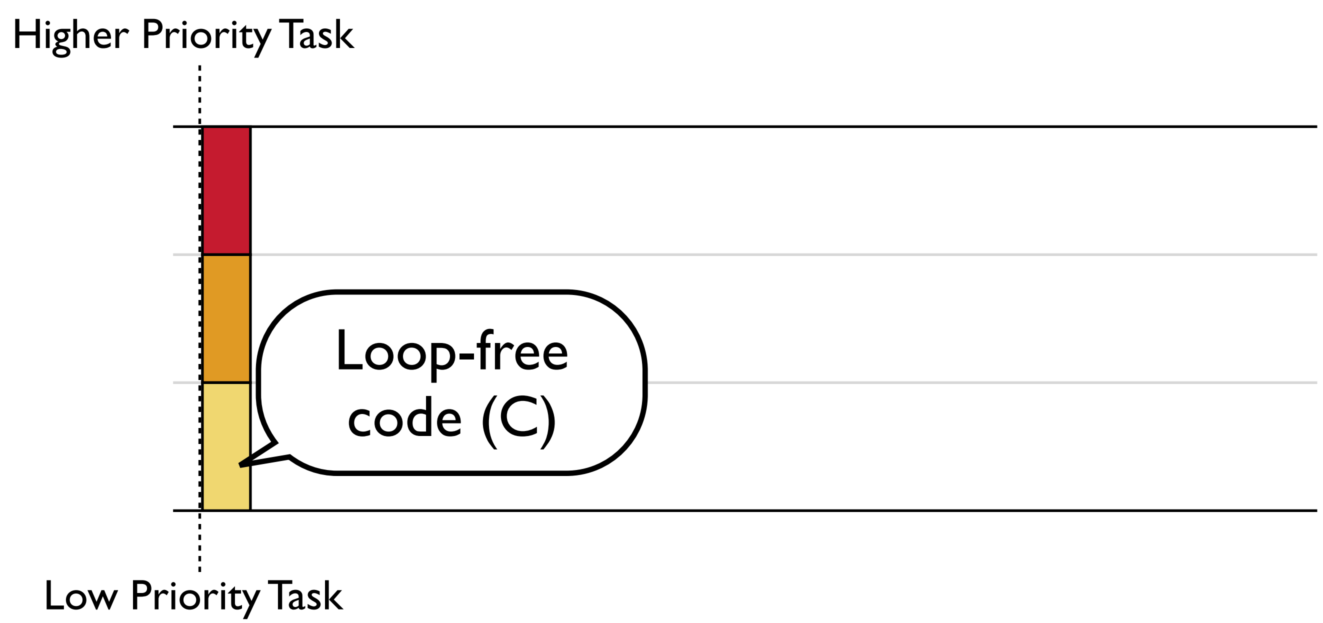
Periodic Programs



Task $\tau = (I, T, P, C, A)$

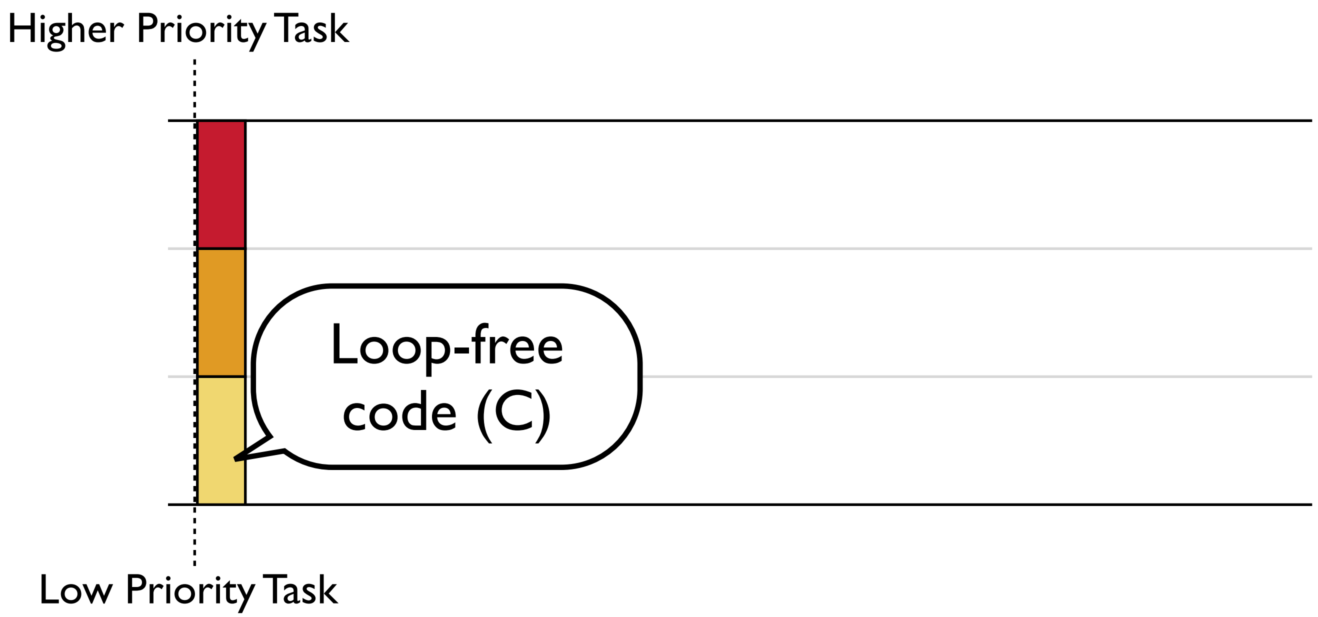
Priority

Task $\tau = (I, T, P, C, A)$



TaskBody

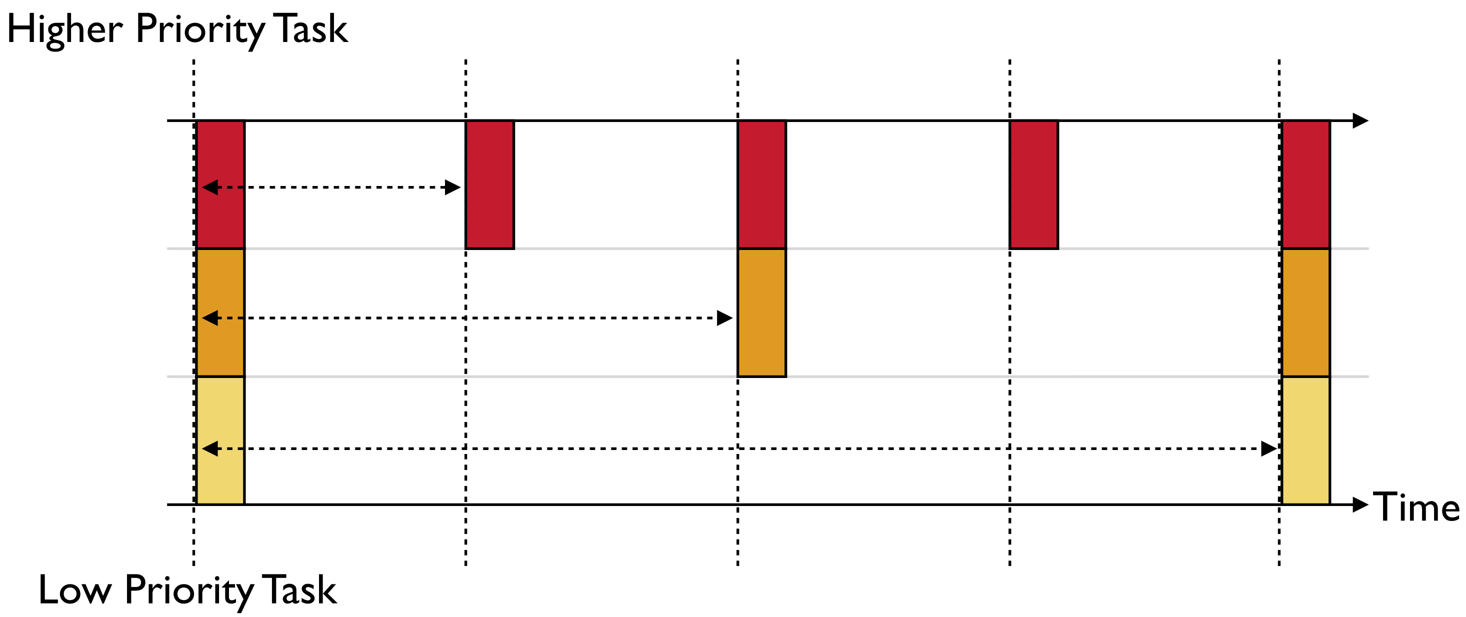
Task $\tau = (I, T, P, C, A)$

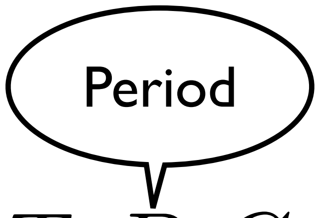


TaskBody

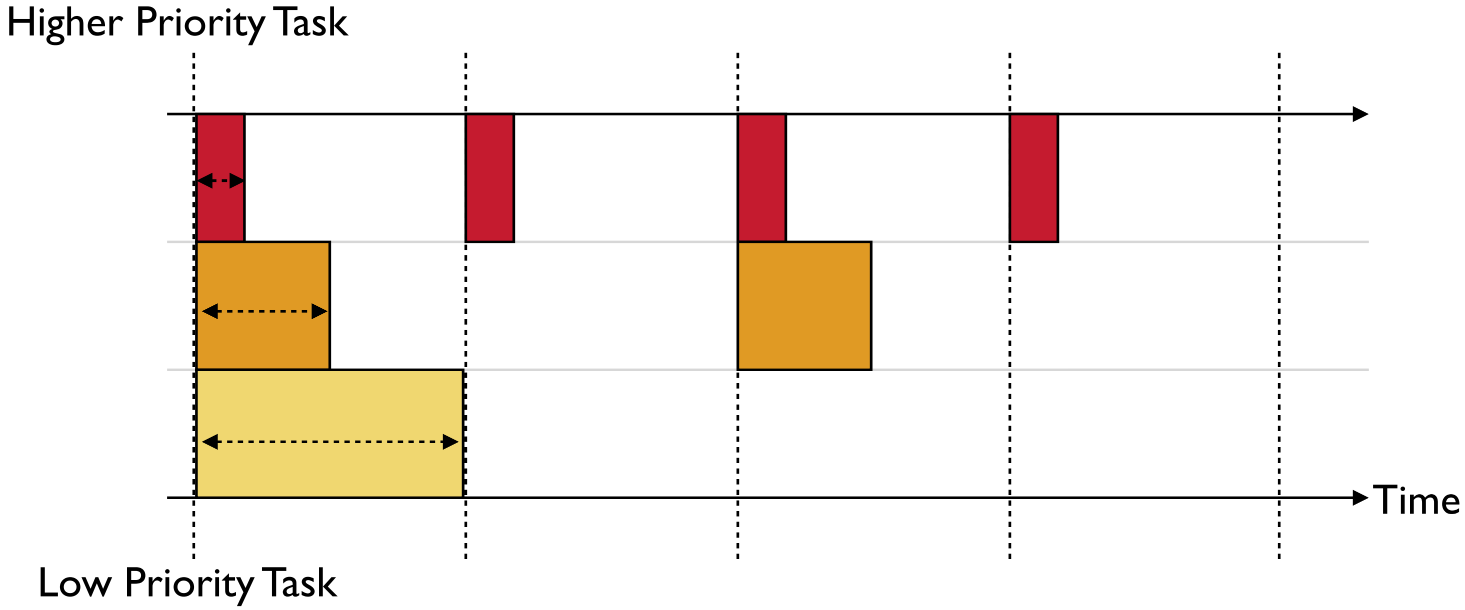
$$\text{Task } \tau = (I, T, P, C, A)$$

```
TASK(Controller)
{
  int old_state = state;
  if(R(need_to_run_nxtbg)) {
    bg_nxtcolorsensor(true);
    W(need_to_run_nxtbg, false);
  }

  switch (TM_mode) {
  case TM_CALIBRATE:
    W(threshold, calibrate());
    if(R(threshold) > 0) {
      TM_mode = TM_INIT;
    }
    break;

  case TM_INIT:
    /* initialize internal variable */
    init();
    TM_mode = TM_HALT;
    break;

  case TM_OPERATION:
    switch(C_state) {
    case C_READ:
      if(R(need_to_read)) {
        if(nxt_motor_get_count(READ_MOTOR) < READ_REV && R(R_state) == RE
          W(R_state, READ_MOVE_HEADER_FORWARD);
        } else if(nxt_motor_get_count(READ_MOTOR) >= READ_REV && R(R_state
          W(R_state, READ_SENSOR);
        }
      } else {
        W(R_state, READ_IDLE);
        C_state = C_TRANS;
      }
      break;

    case C_TRANS:
      old_state = state;
      if(transition(state, R(input))) {
        TM_mode = TM_HALT;
      } else {
        C_state = C_WRITE;
      }
      break;

    case C_WRITE:

      /* Check if we need to chagne the bit */
      if(R(input) != R(output)) {
        /* Check the header and move it back if necessary */
```

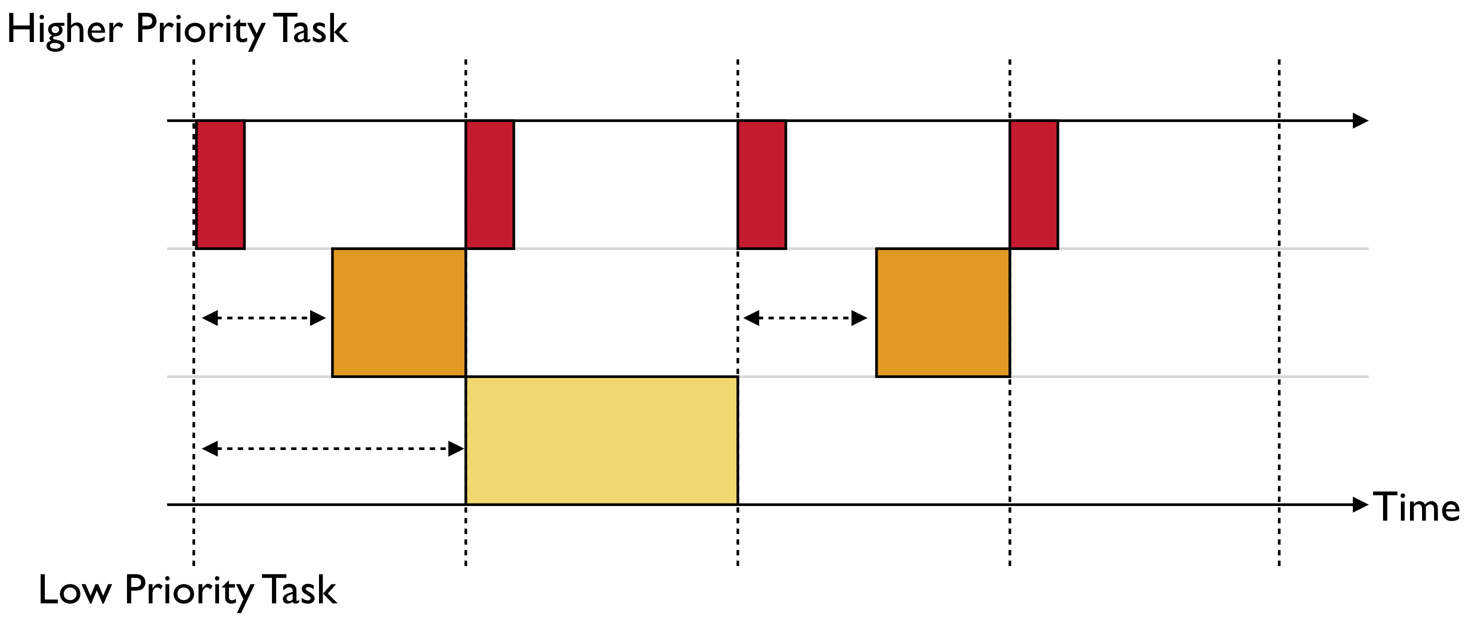




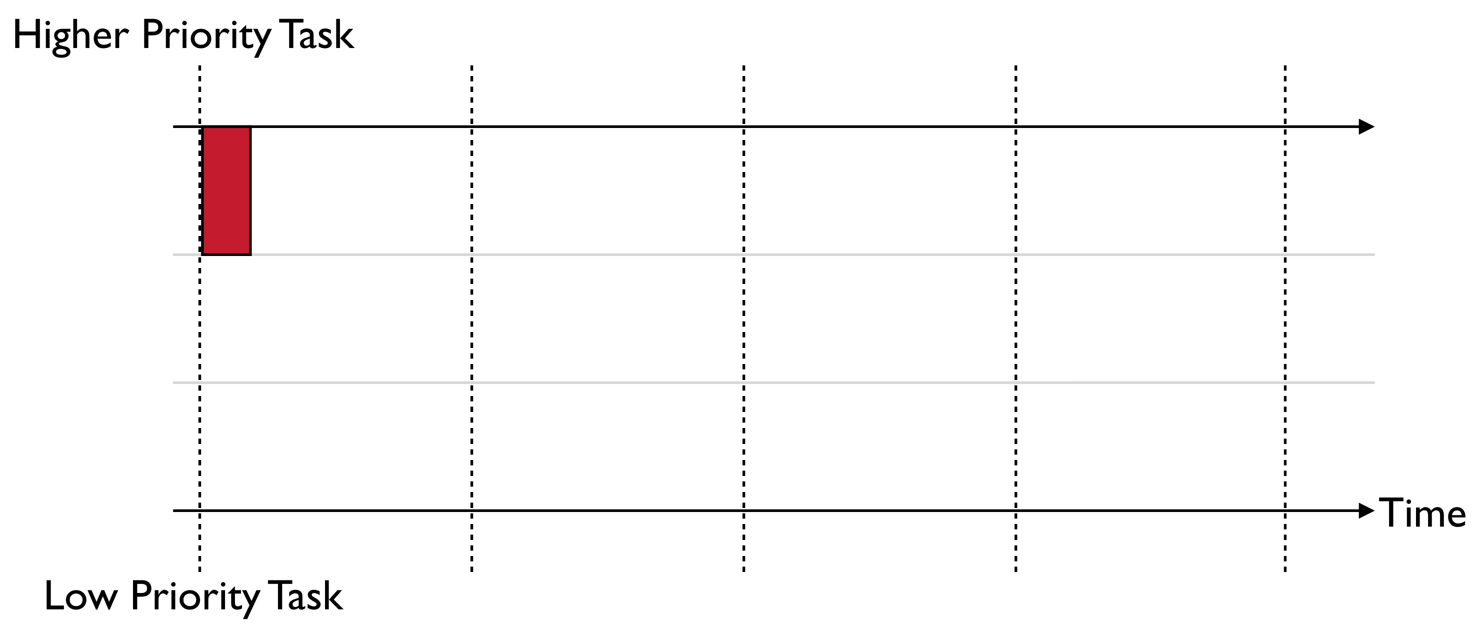
Task $\tau = (I, T, P, C, A)$



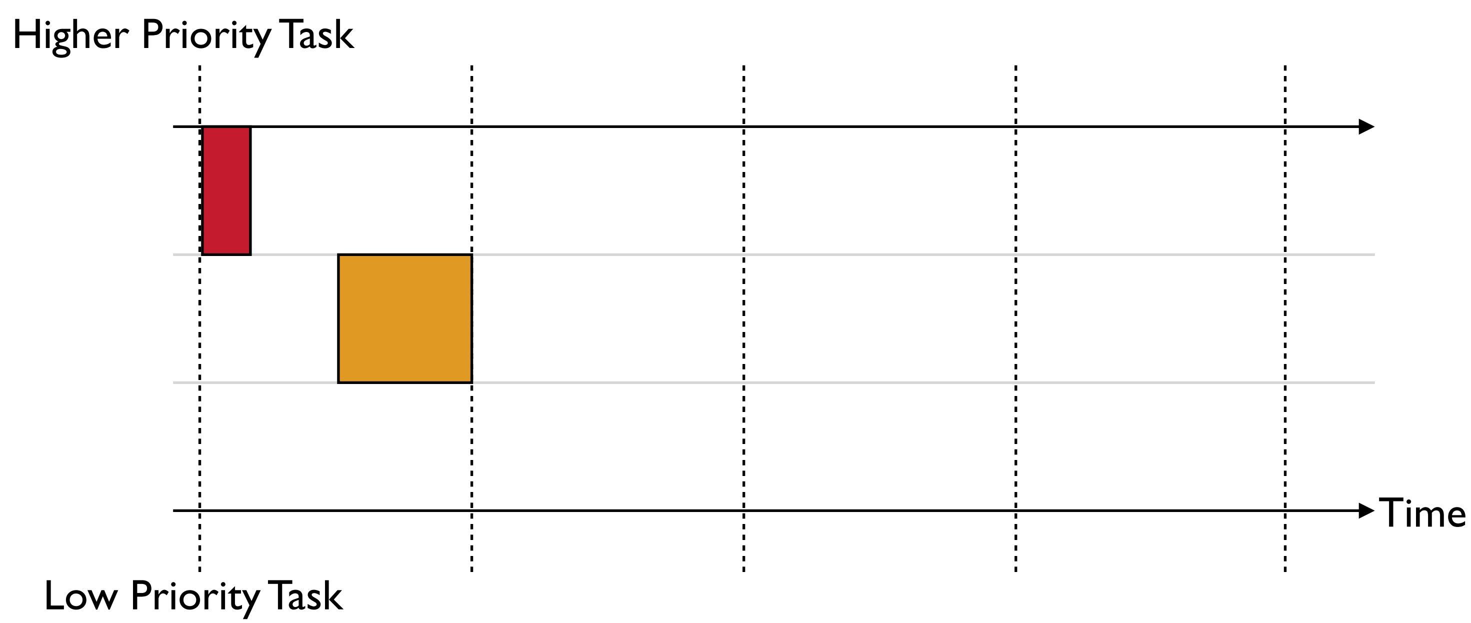
Task $\tau = (I, T, P, C, A)$



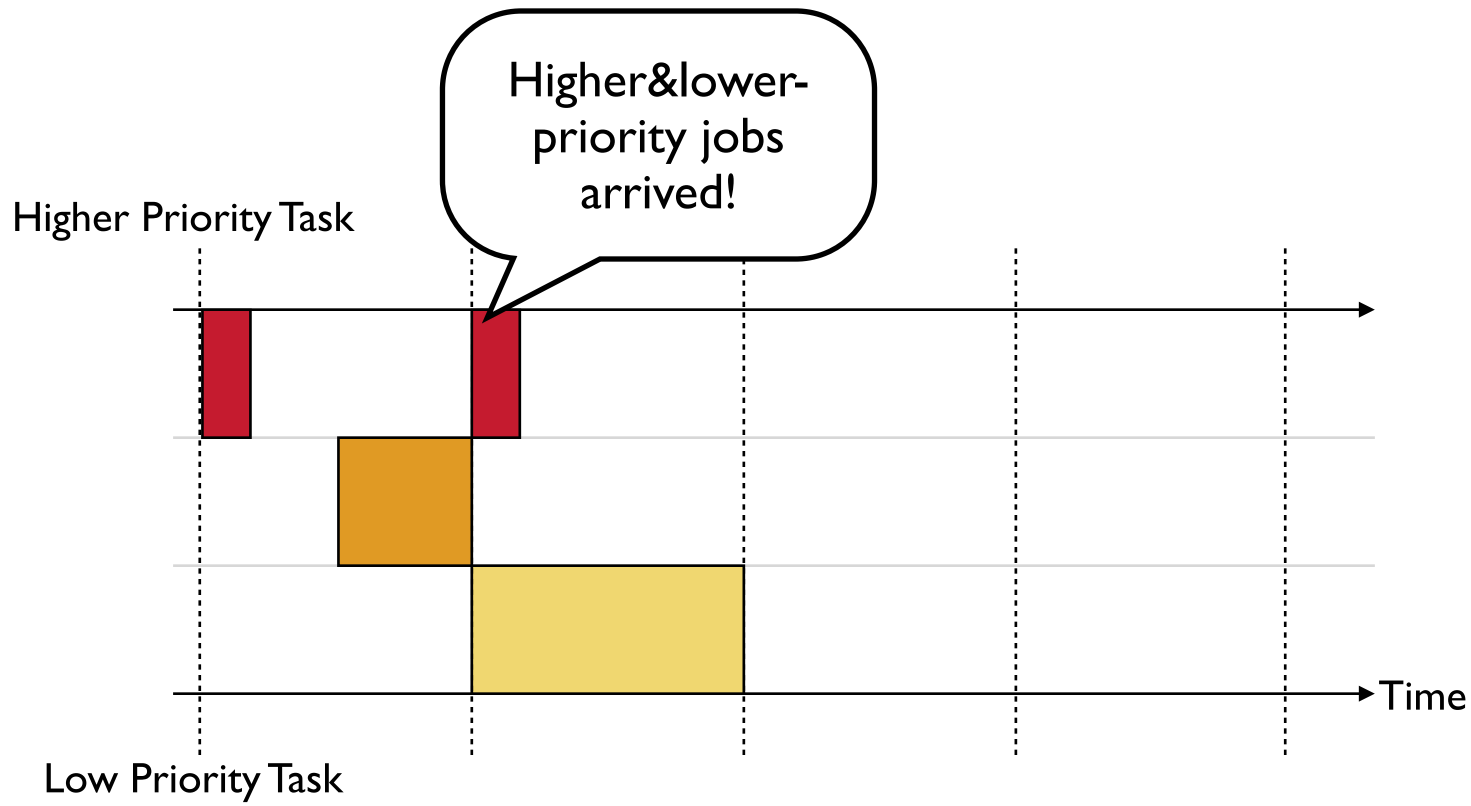
Task $\tau = (I, T, P, C, A)$



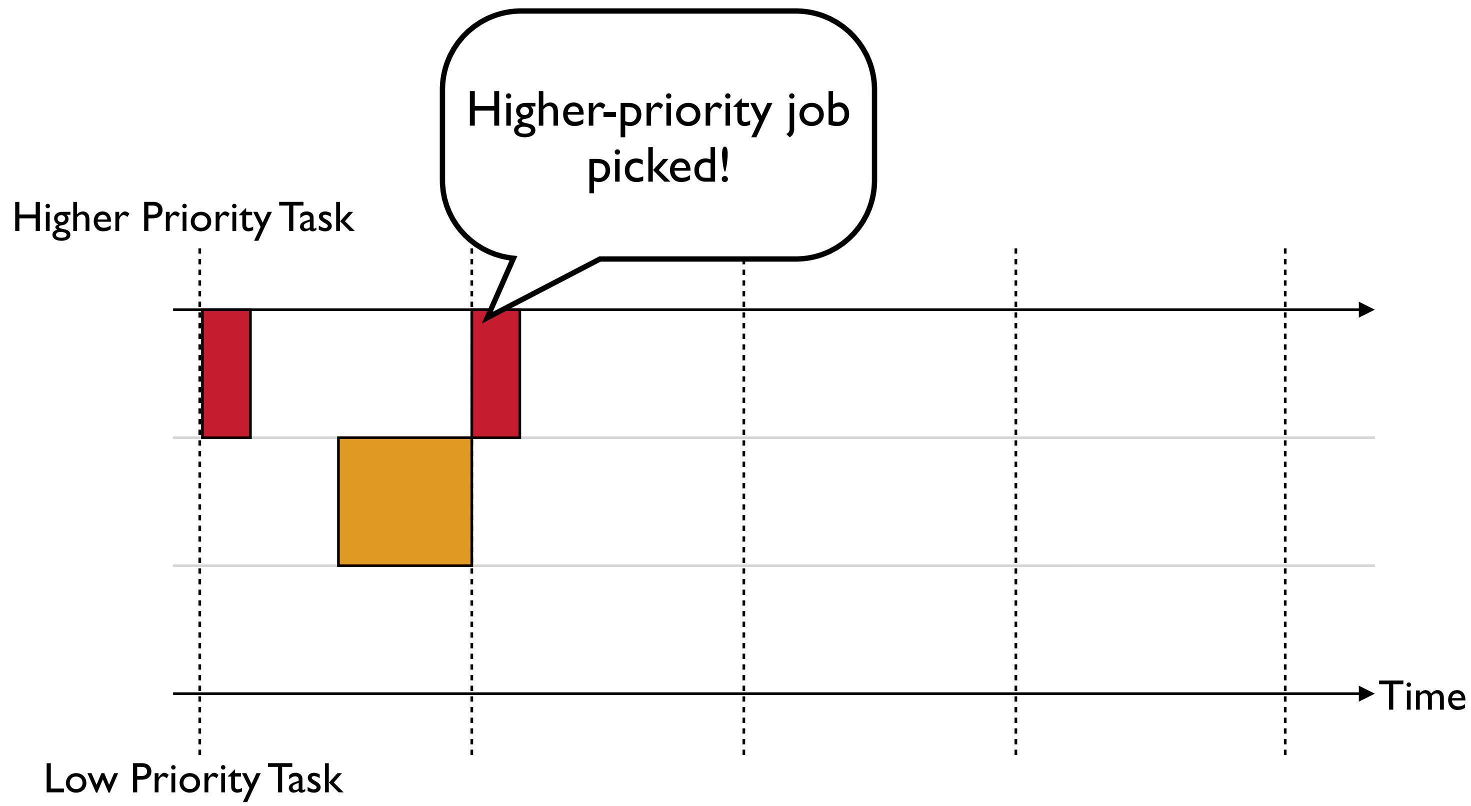
Preemptive Fixed Priority-based Scheduling



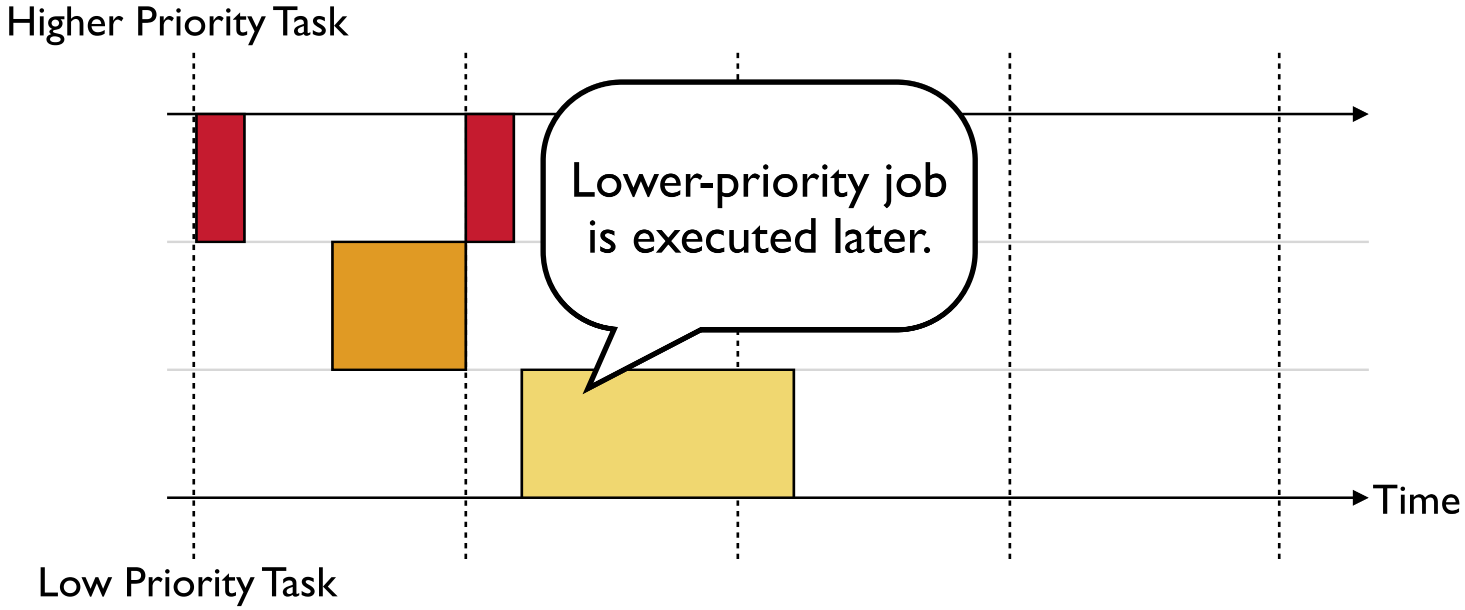
Preemptive Fixed Priority-based Scheduling



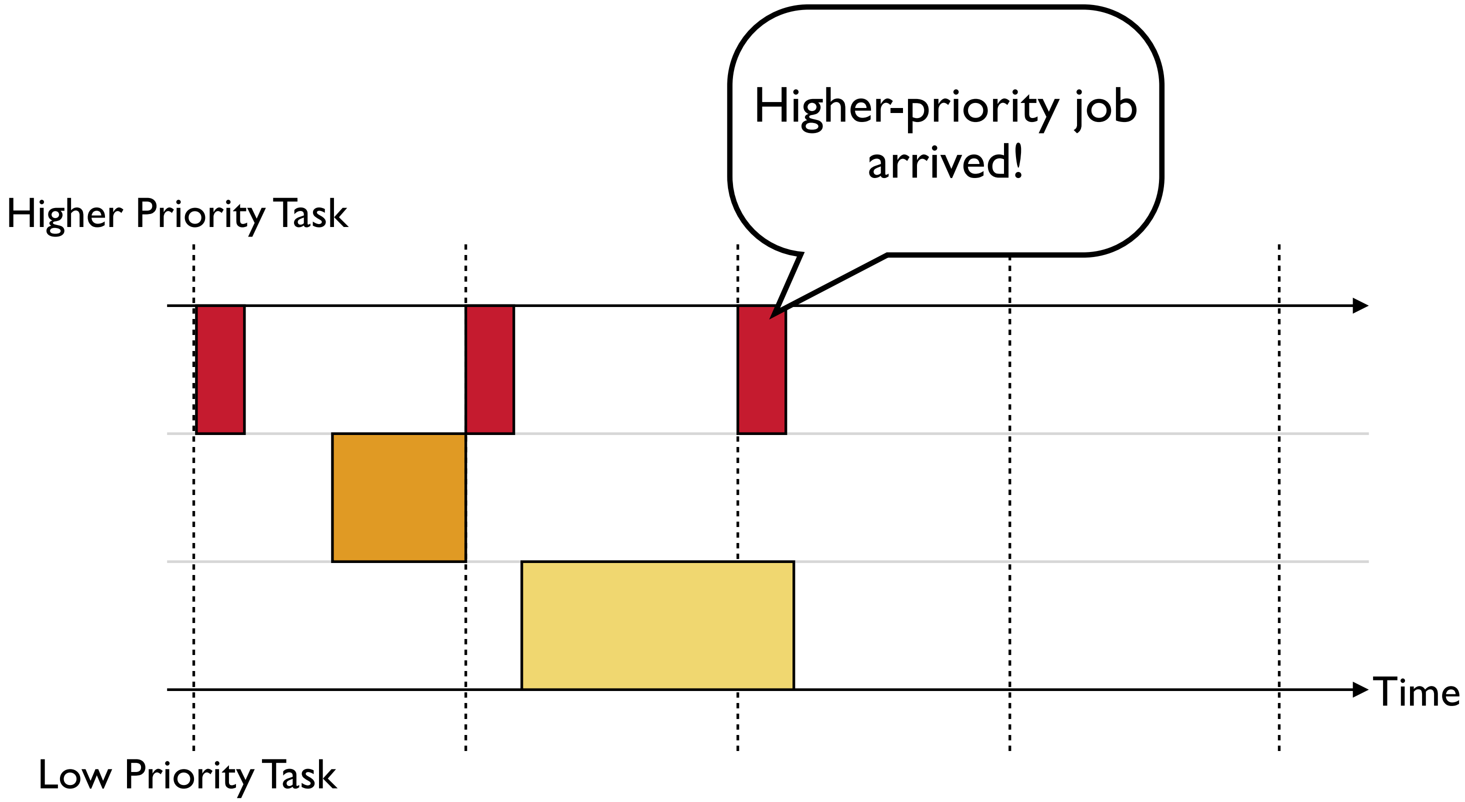
Preemptive Fixed Priority-based Scheduling



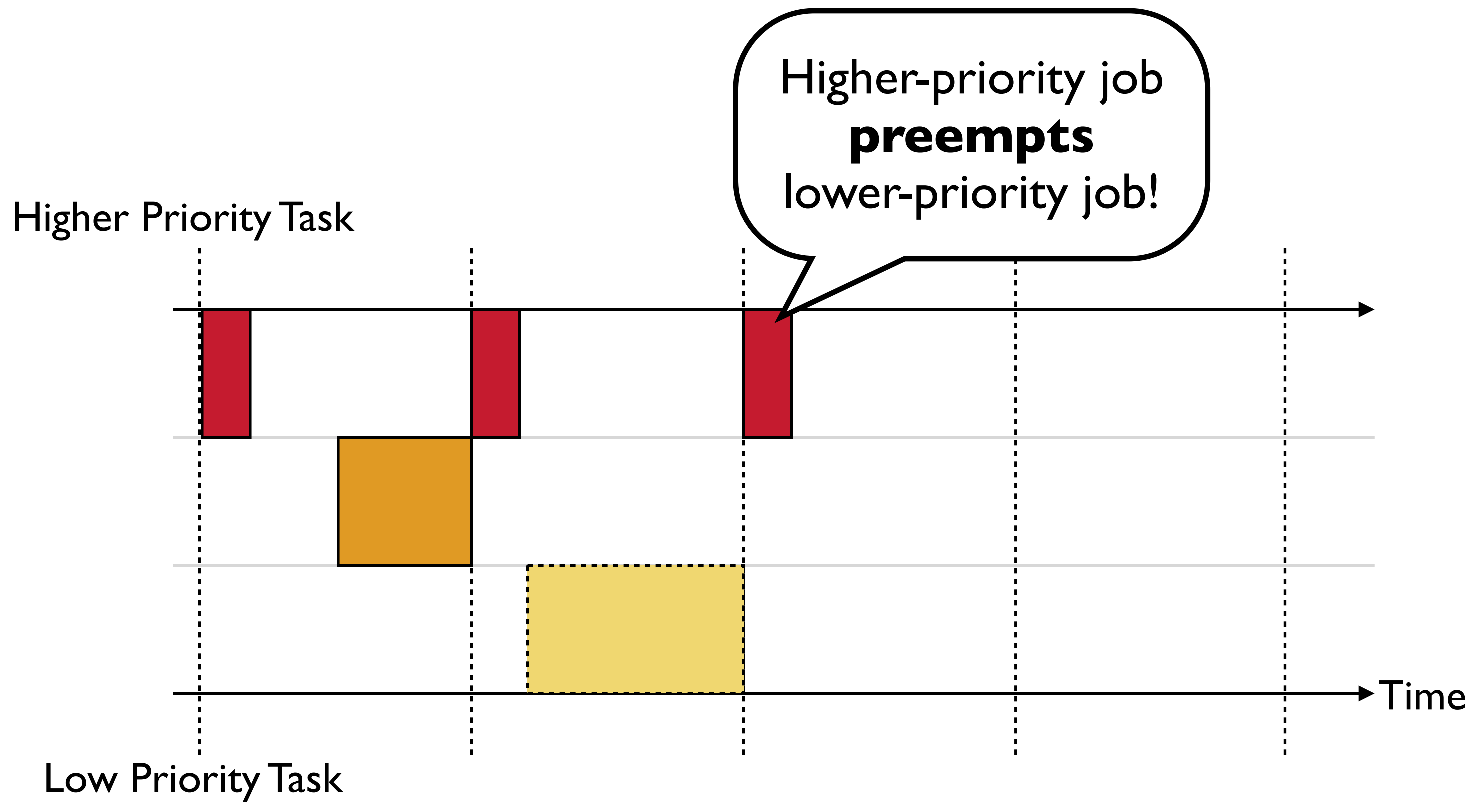
Preemptive Fixed <u>Priority-based</u> Scheduling



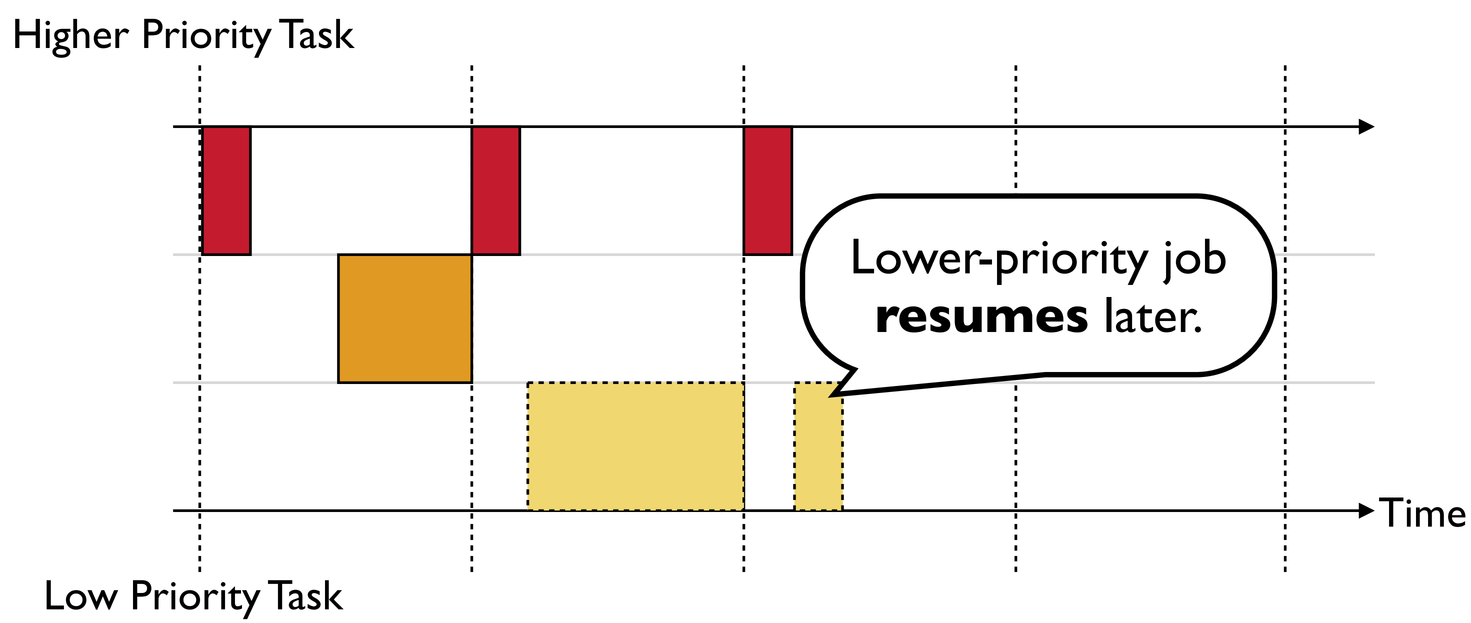
Preemptive Fixed Priority-based Scheduling



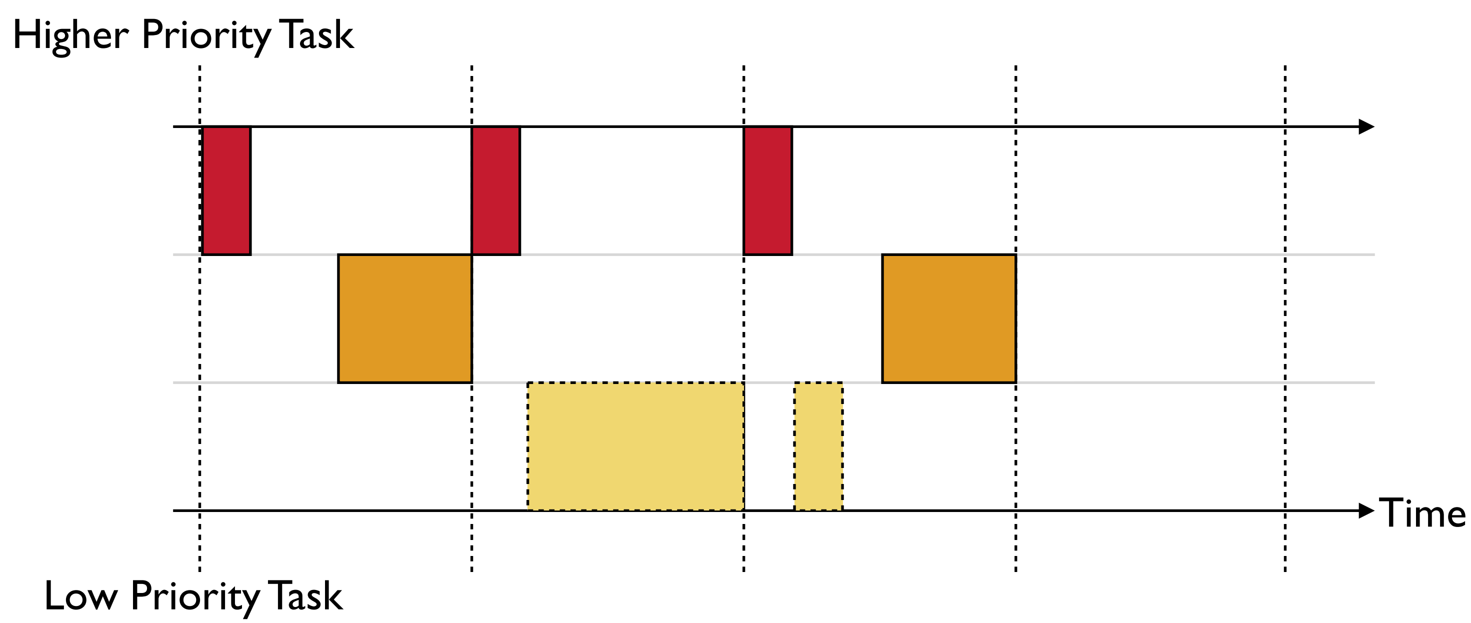
Preemptive Fixed Priority-based Scheduling



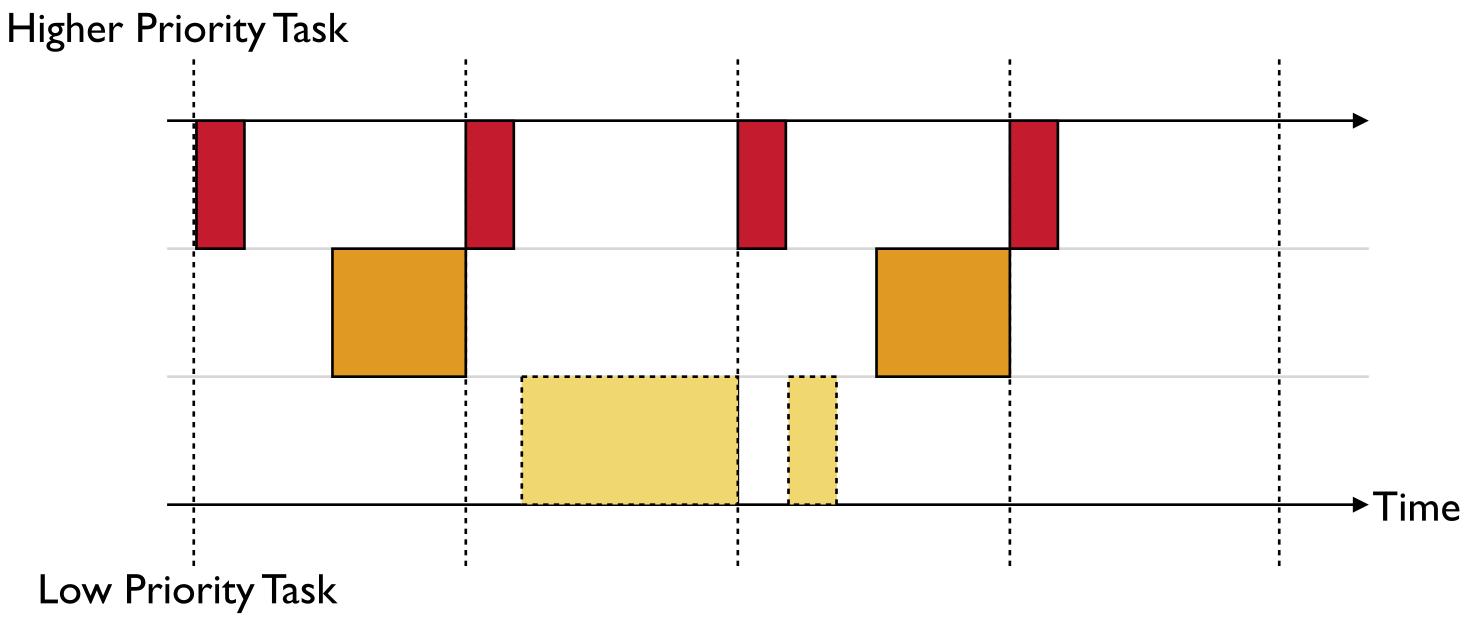
<u>Preemptive</u> Fixed Priority-based Scheduling



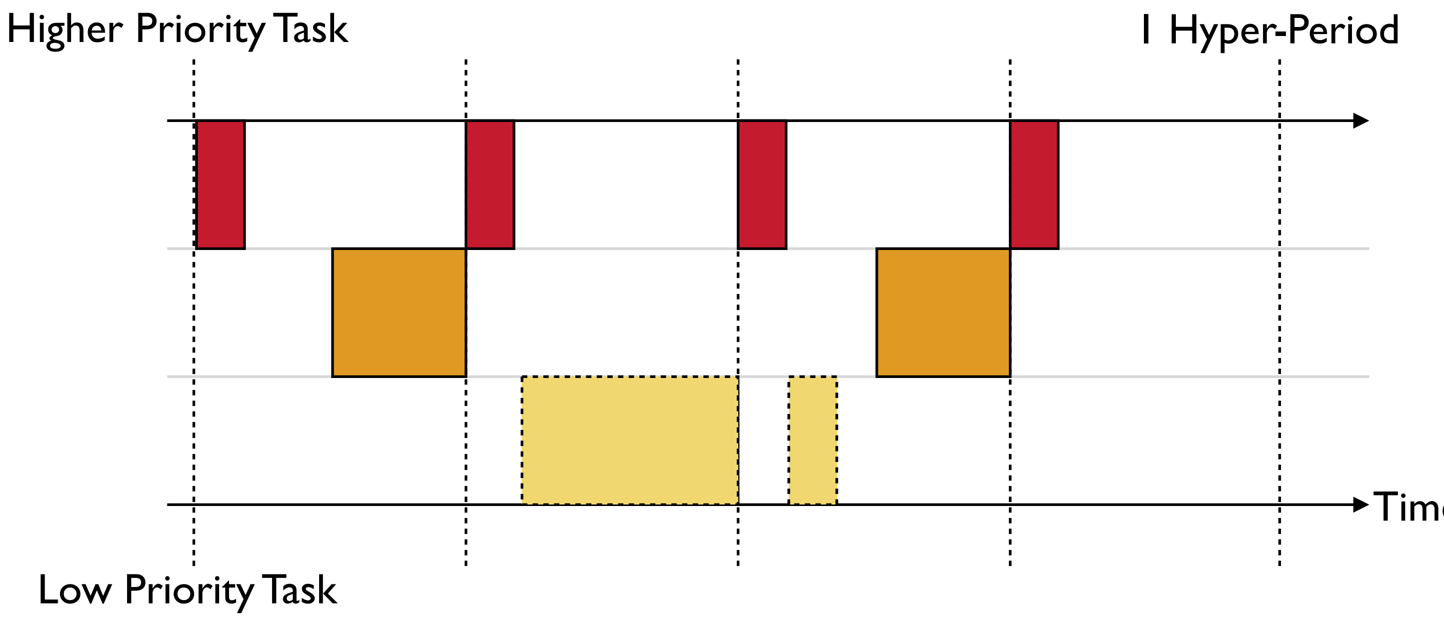
<u>Preemptive</u> Fixed Priority-based Scheduling



Preemptive Fixed Priority-based Scheduling



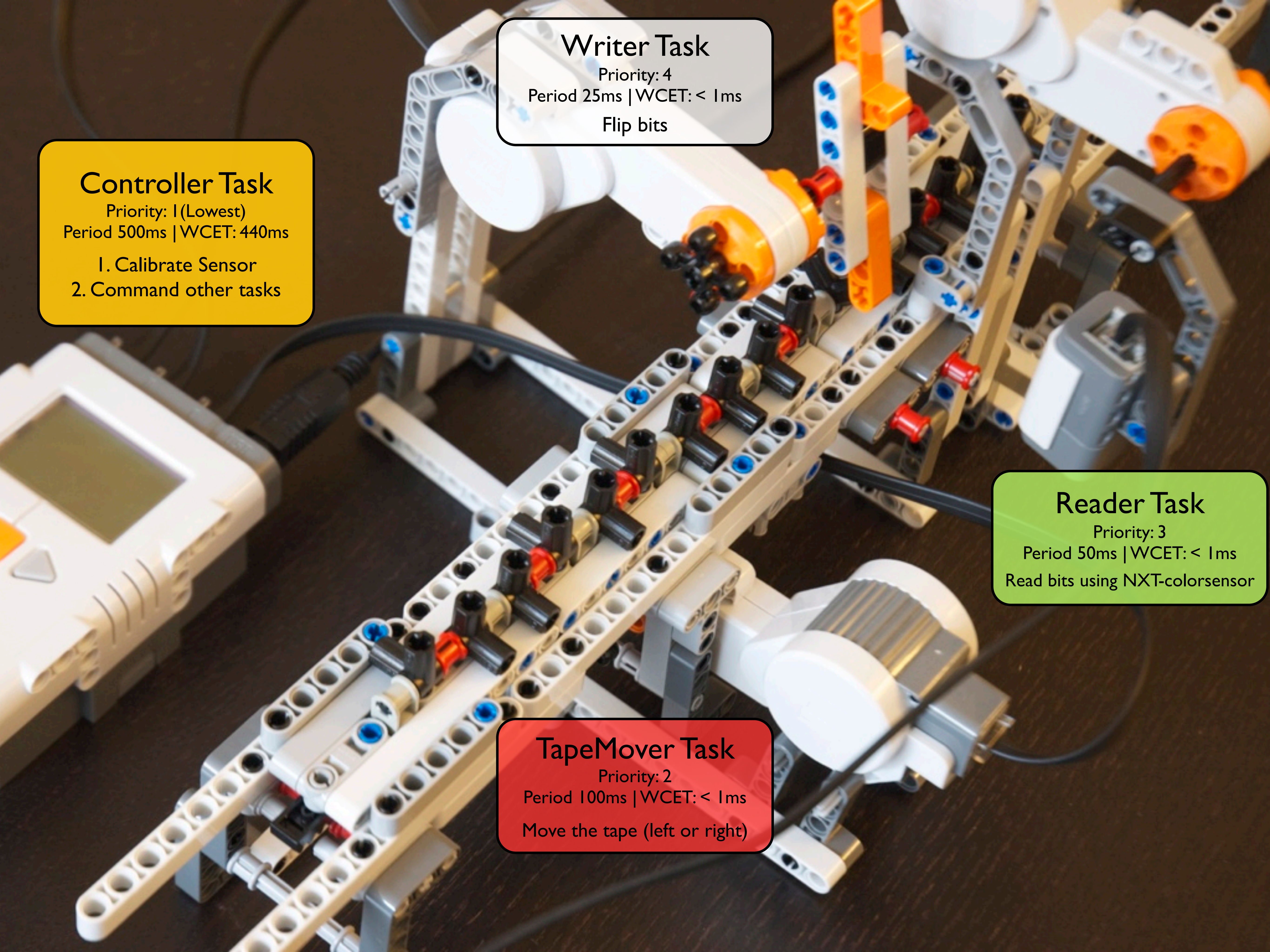
Preemptive Fixed Priority-based Scheduling



Preemptive Fixed Priority-based Scheduling

Case Study:

Concurrent Turing Machine



Writer Task
Priority: 4
Period 25ms | WCET: < 1ms
Flip bits
Controller Task
Priority: 1 (Lowest)
Period 500ms | WCET: 440ms
1. Calibrate Sensor
2. Command other tasks
Reader Task
Priority: 3
Period 50ms | WCET: < 1ms
Read bits using NXT-colorsensor
TapeMover Task
Priority: 2
Period 100ms | WCET: < 1ms
Move the tape (left or right)

Isn’t LEGO Mindstorms just a TOY?

SIEMENS

RENAULT

PSA PEUGEOT CITROËN

BMW

No, it runs OSEK/VDX-compatible RTOS.

Open Systems and their Interfaces for the Electronics in Motor Vehicles

a standard software architecture for the various electronic control units (ECUs) throughout a car

OPEL

BOSCH

DAIMLERCHRYSLER

DEMO

Unary Addition

2 + 3 = ?

http://www.youtube.com/watch?v=teDyd0d5M4o

Properties

Property 1: When a bit is being read, all the motors should **stop**.

Properties

Property 1: When a bit is being read, all the motors should **stop**.

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Properties

Property 1: When a bit is being read, all the motors should **stop**.

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Property 3: When tape moves, the writer motor and read motor should **stop**.

Properties

Property 1: When a bit is being read, all the motors should **stop**.

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Property 3: When tape moves, the writer motor and read motor should **stop**.

Property 4: When a bit is being read, the sensor should be on **Green** mode

Properties

Property 1: When a bit is being read, all the motors should **stop**.

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Property 3: When tape moves, the writer motor and read motor should **stop**.

Property 4: When a bit is being read, the sensor should be on **Green** mode

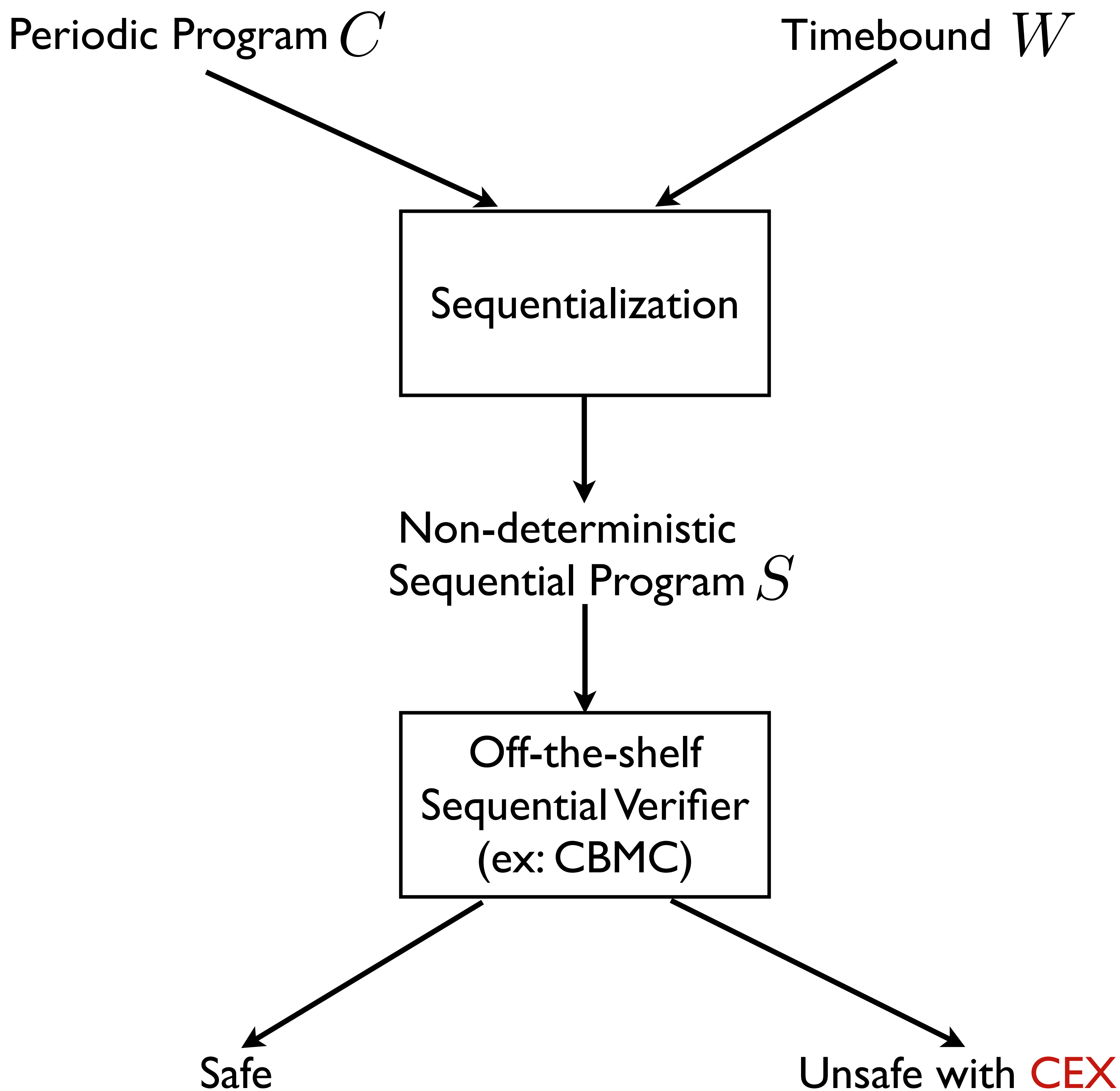
Property 5: The sensor mode must be switched in **Controller Task**, not in Reader Task

Properties

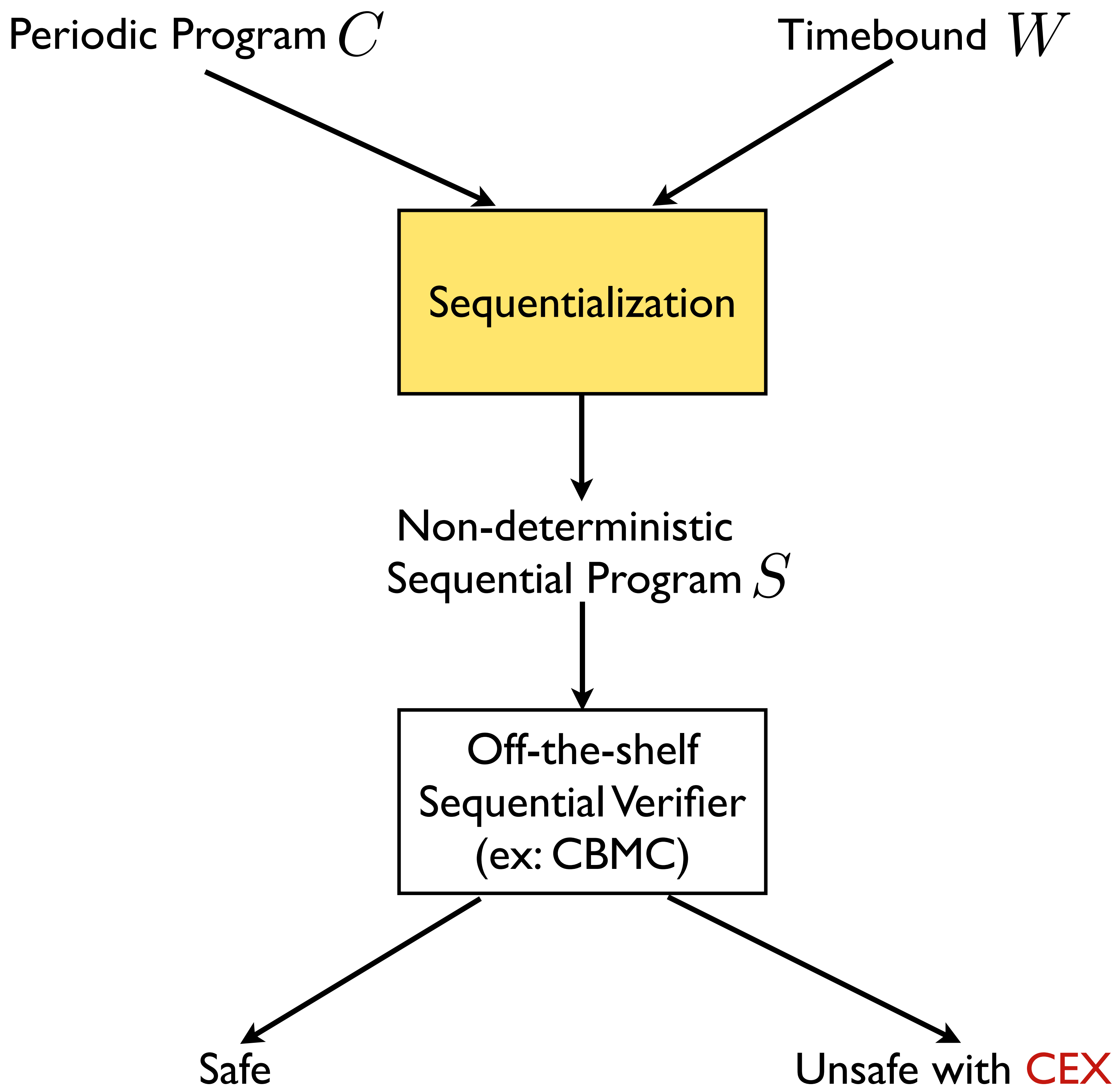
```
  case READ_SENSOR:
    if(ecrobot_get_nxtcolorsensor_mode(COLOR_SENSOR) != NXT_LIGHTSENSOR_GREEN) {
      ecrobot_set_nxtcolorsensor(COLOR_SENSOR, NXT_LIGHTSENSOR_GREEN);
      W(need_to_run_nxtbg, true);
    }

    if(!R(need_to_run_nxtbg)) {
      /* Turn the sensor on */
#ifdef VERIFICATION
      /* Property 1: When a bit is being read,
         all the motors should be stopped.  */
      /* PASSED with 80*/
      assert(R(R_speed) == 0 && R(W_speed) == 0 && R(T_speed) == 0);

      /* Property 4: When a bit is being read,
         the sensor should be on Green mode */
      assert(ecrobot_get_nxtcolorsensor_mode(COLOR_SENSOR) == NXT_LIGHTSENSOR_GREEN);
#endif
      ecrobot_set_nxtcolorsensor(COLOR_SENSOR, NXT_LIGHTSENSOR_GREEN);

      /* Read Sensor Value */
      bg_nxtcolorsensor(false);
      color = ecrobot_get_nxtcolorsensor_light(COLOR_SENSOR);
```

Key Idea:

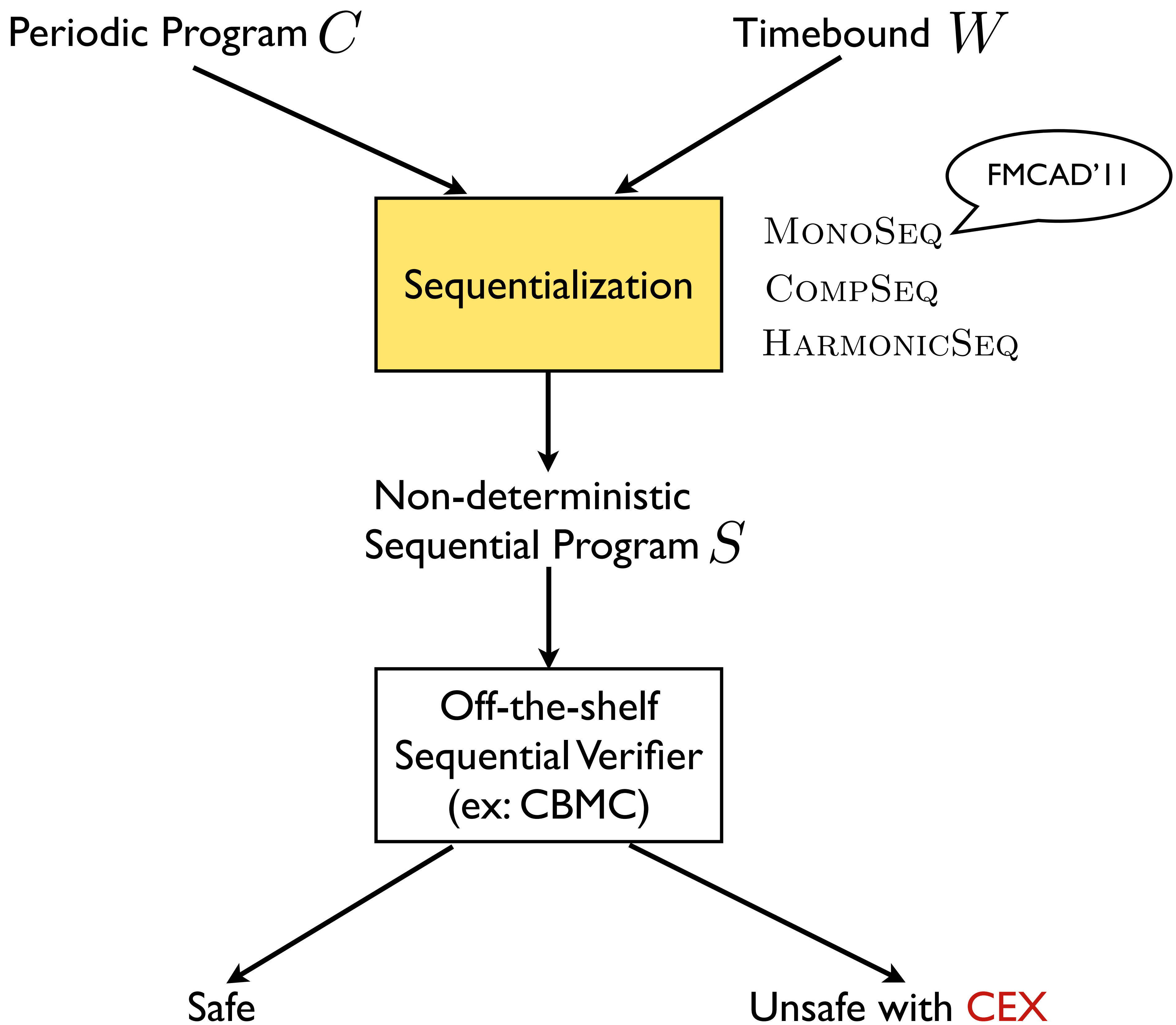
Sequentialization



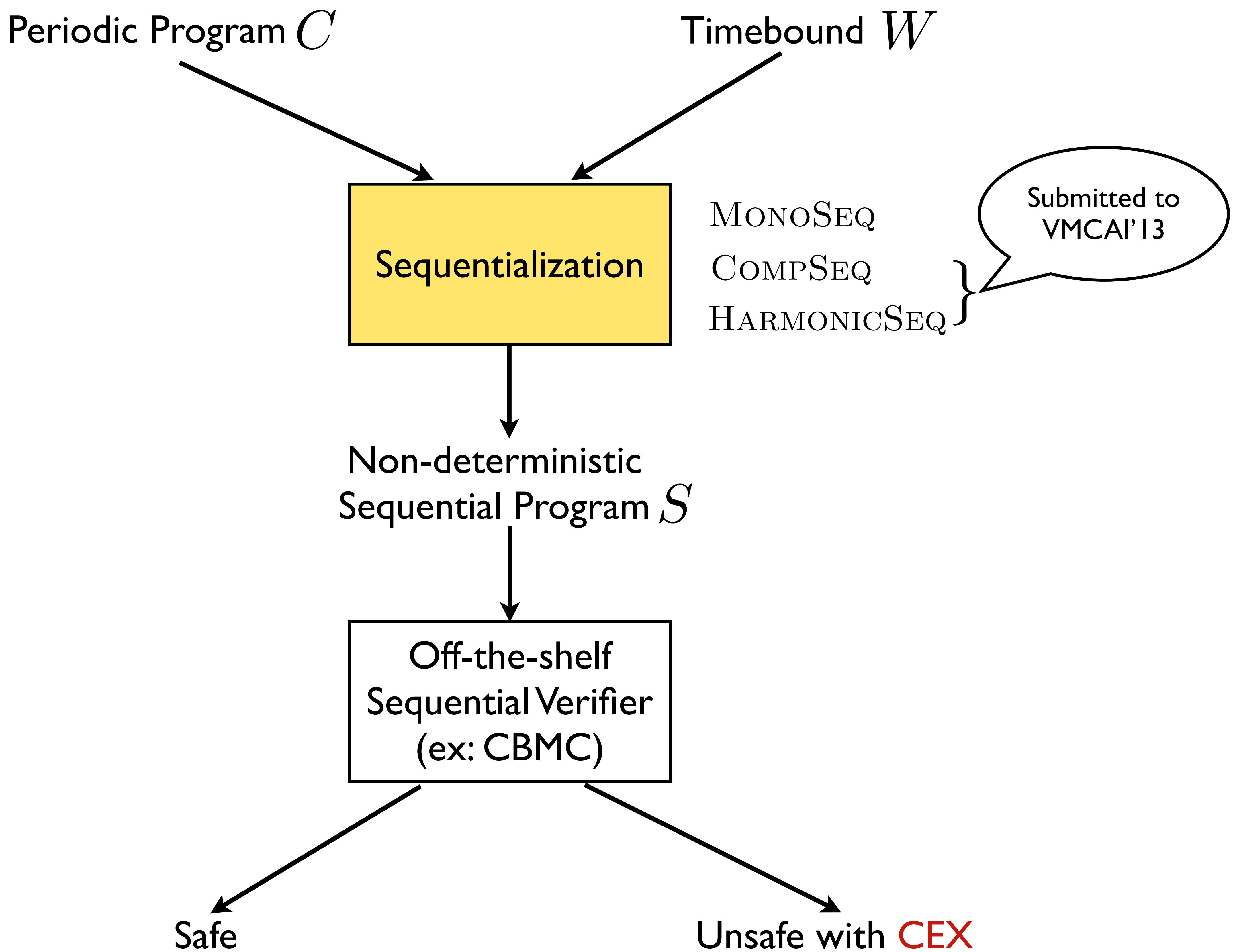
Time-bounded Verification of Periodic Programs via Sequentialization



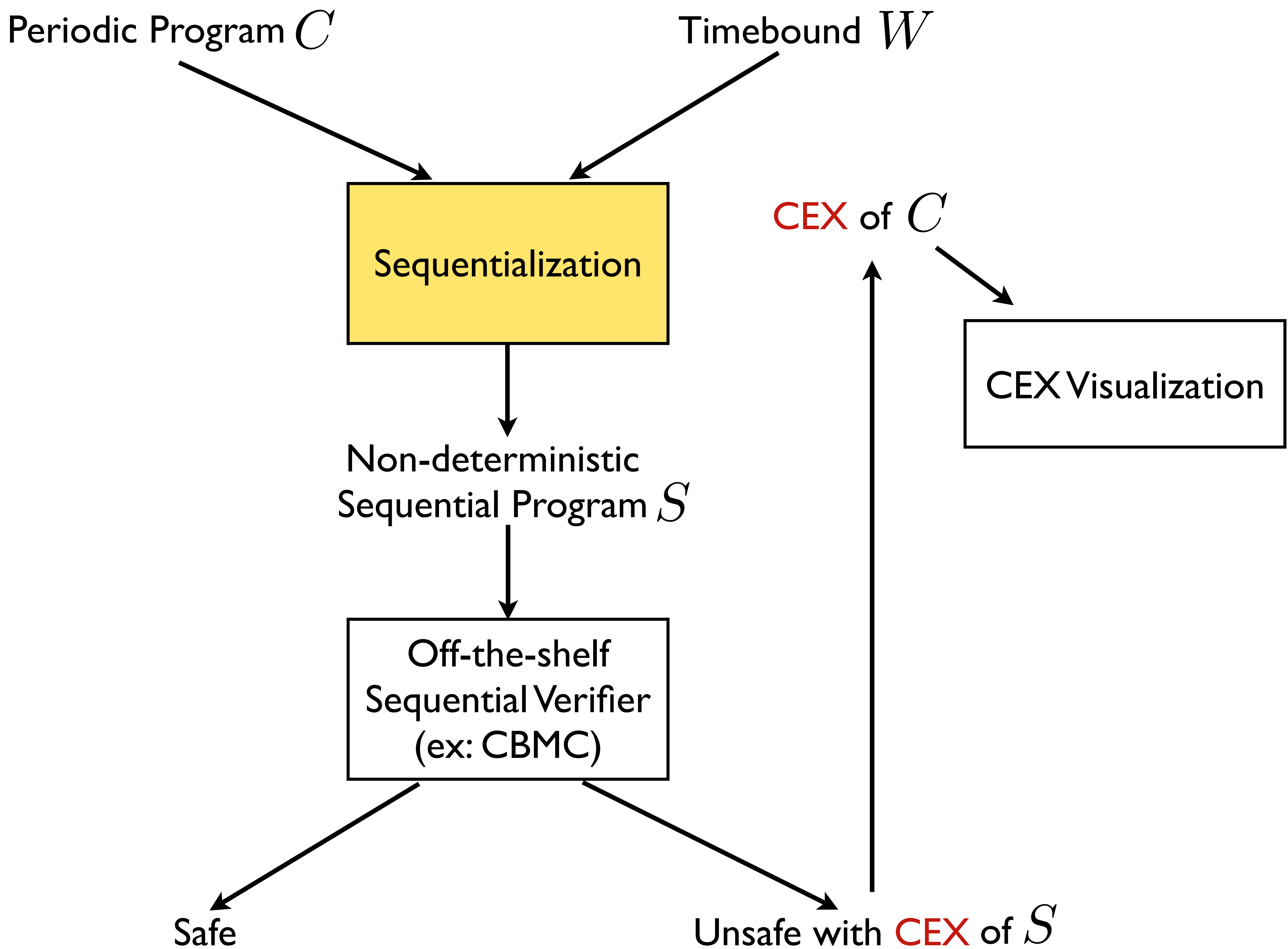
Time-bounded Verification of Periodic Programs via Sequentialization



Time-bounded Verification of Periodic Programs via Sequentialization



Time-bounded Verification of Periodic Programs via Sequentialization



Time-bounded Verification of Periodic Programs via Sequentialization

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

MonoSeq Sequentialization

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MONOSEQ Sequentialization

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

Our Approach (MonoSeq):

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

MonoSeq Sequentialization

Exponential Blow-up!

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Program + Constraints

Our Approach (MonoSeq):

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

MonoSeq Sequentialization

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Ex:

```
x:= nondet();
y:= 10;
assume(x > 10);
```

Our Approach (MonoSeq):

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

MonoSeq Sequentialization

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

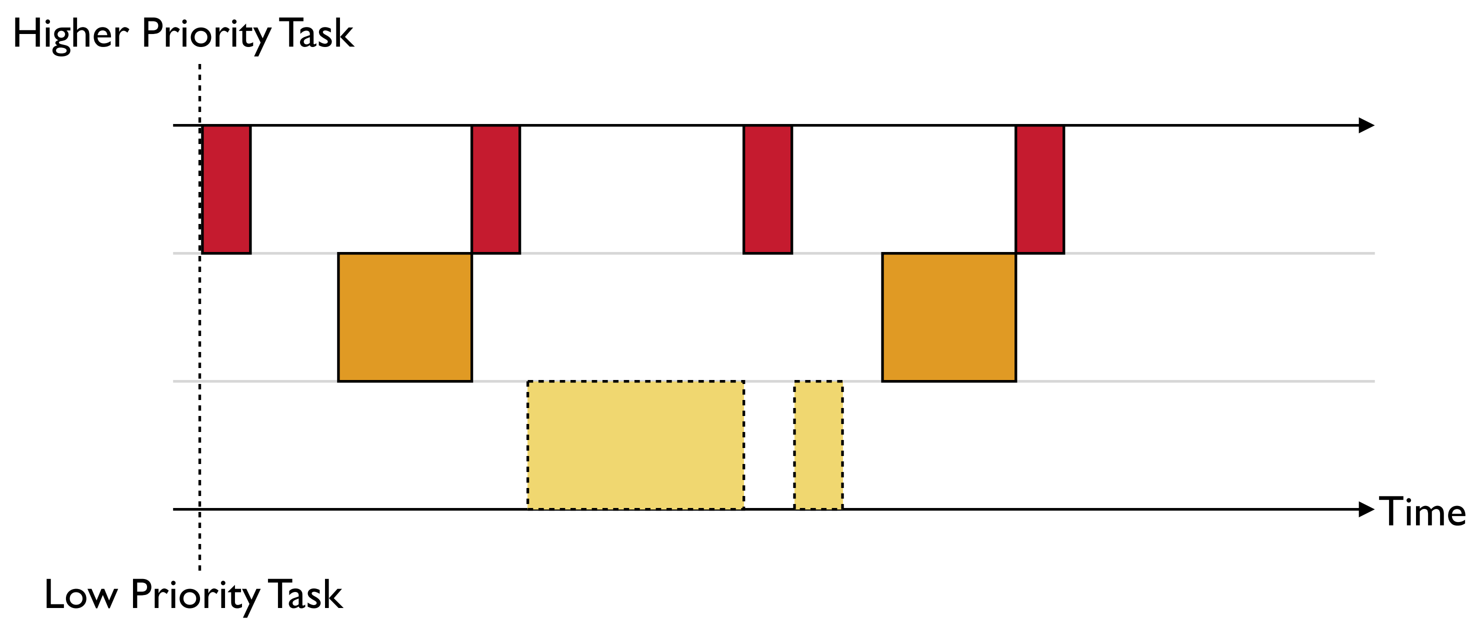
Our Approach (MonoSeq):

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

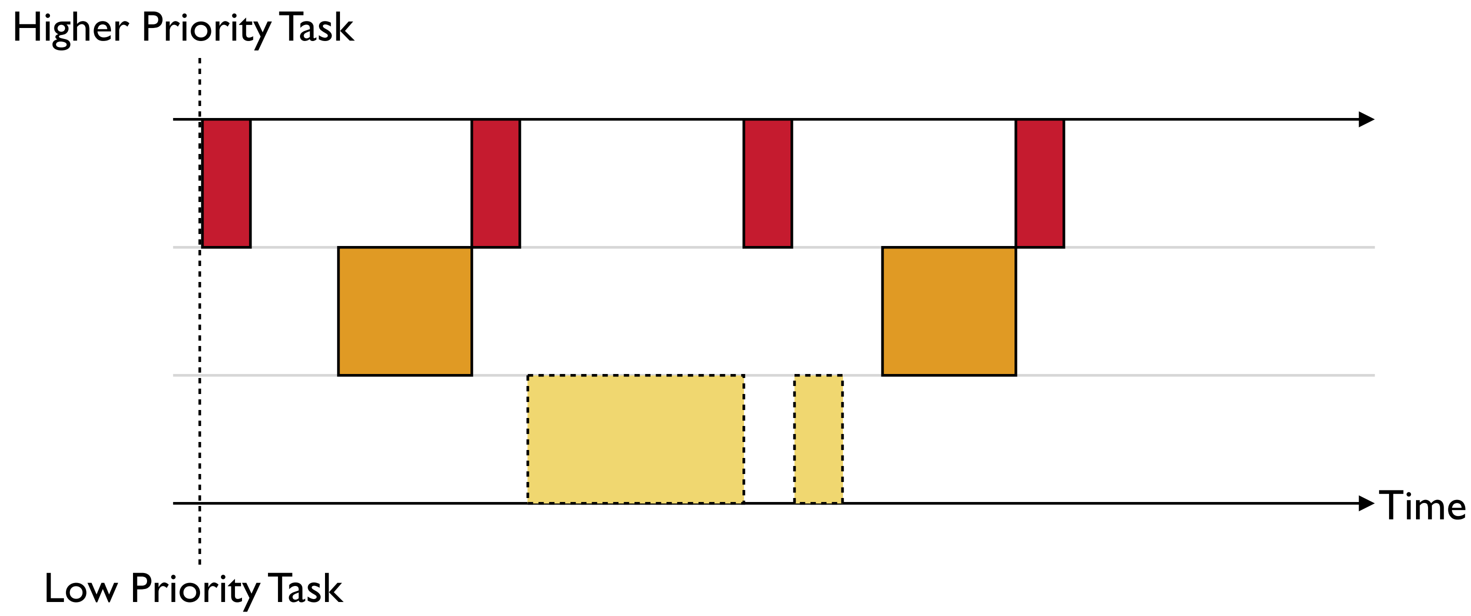
$O(R)$
where R= # of Jobs

$O(R^2)$
where R= # of Jobs

MonoSeq Sequentialization

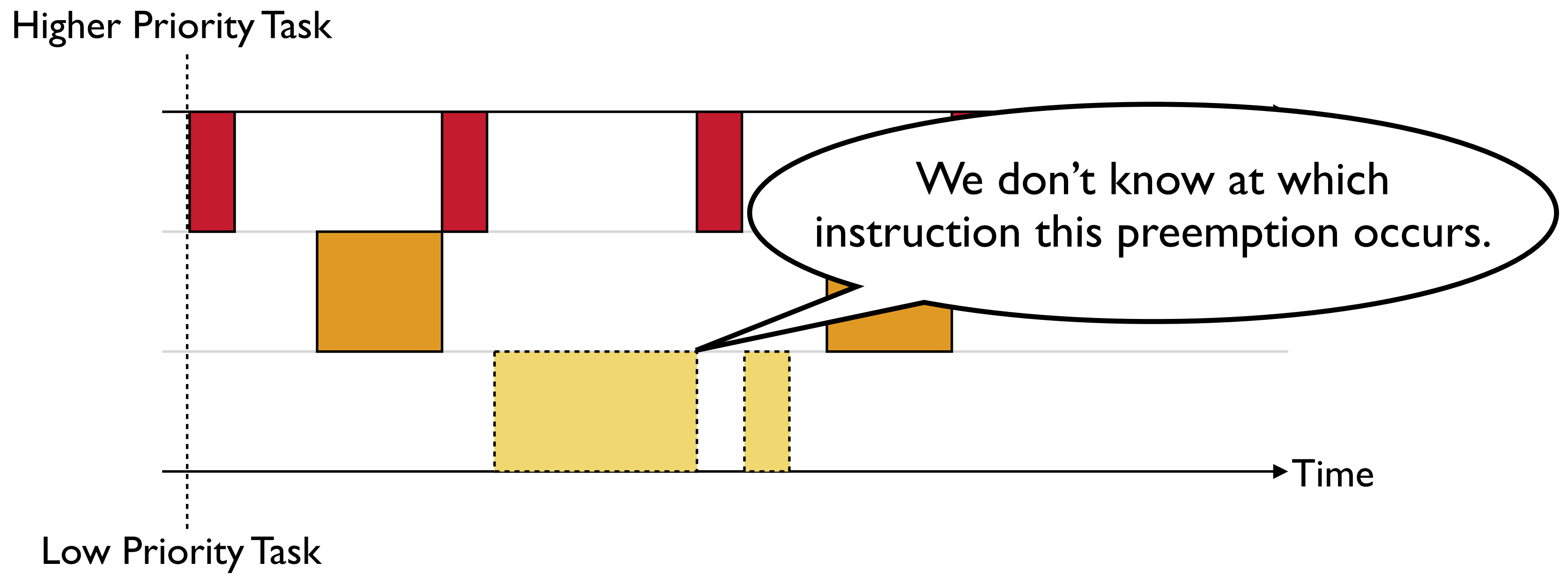


Job-bounded Abstraction



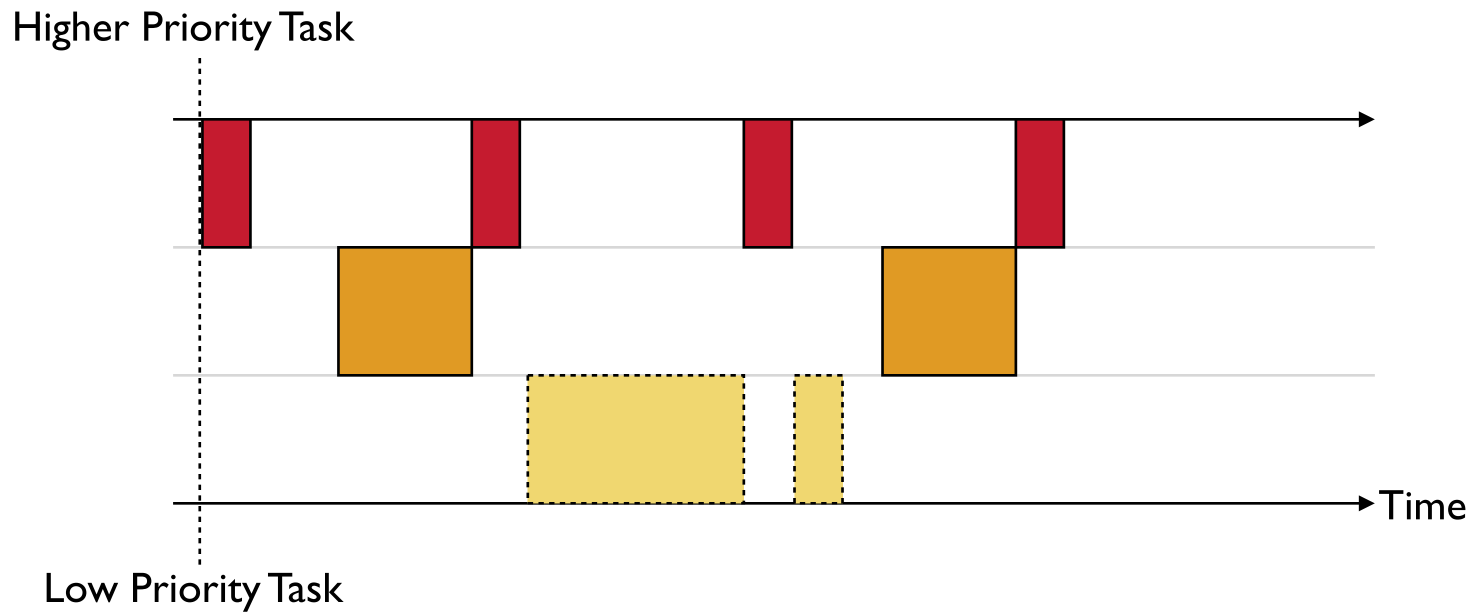
Because we are interested in logical properties,
We abstract the absolute time with relative order of execution

Job-bounded Abstraction

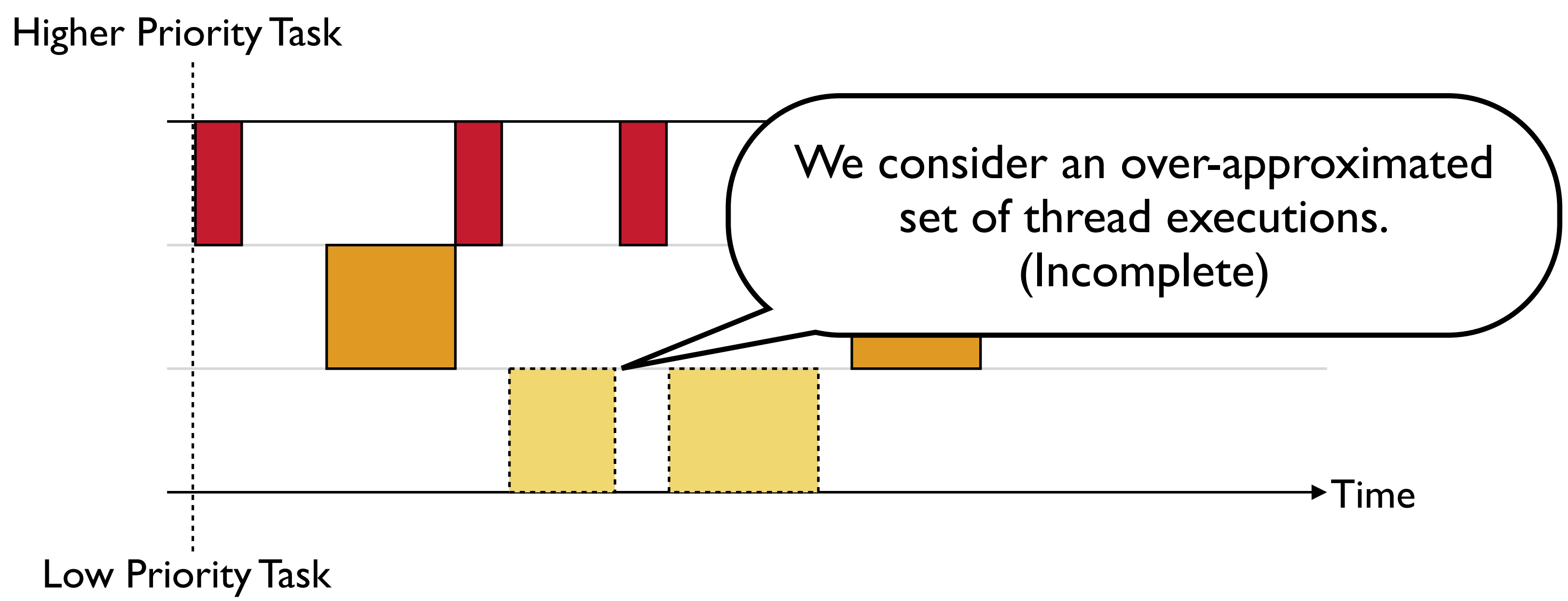


Because we are interested in logical properties,
We abstract the absolute time with relative order of execution

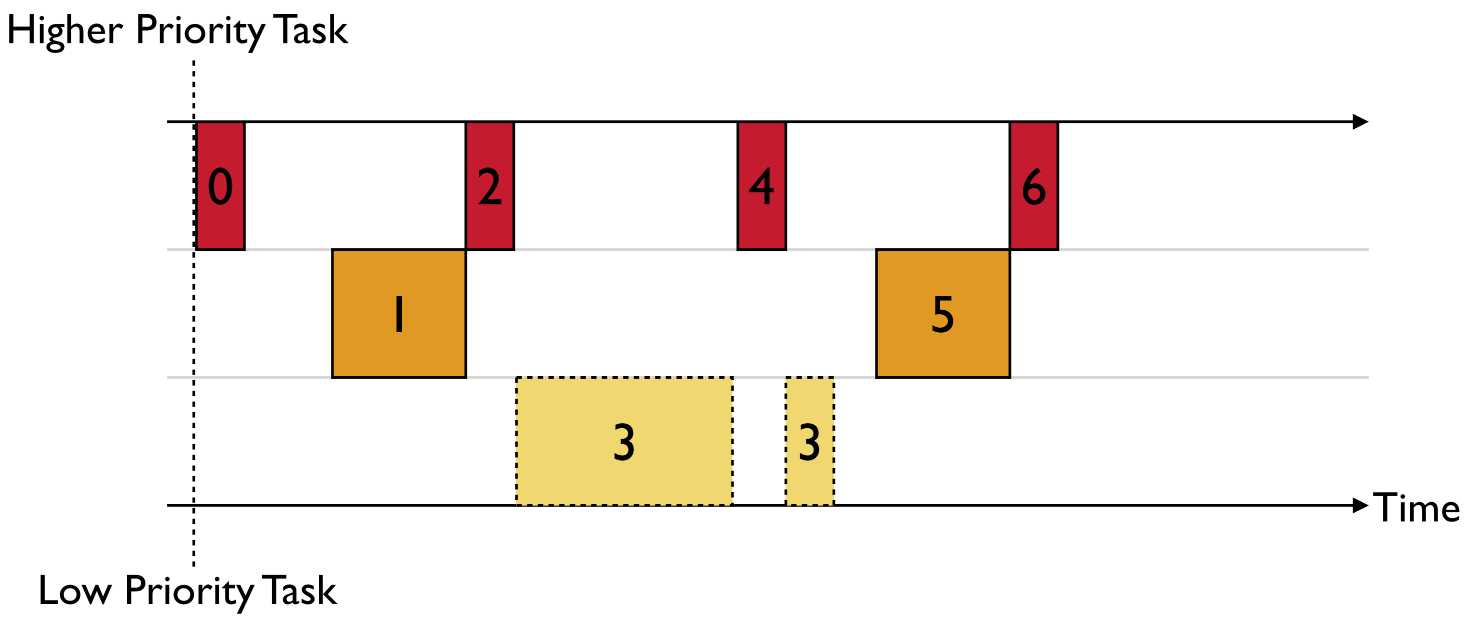
Job-bounded Abstraction



Because we are interested in logical properties,
We abstract the absolute time with relative order of execution

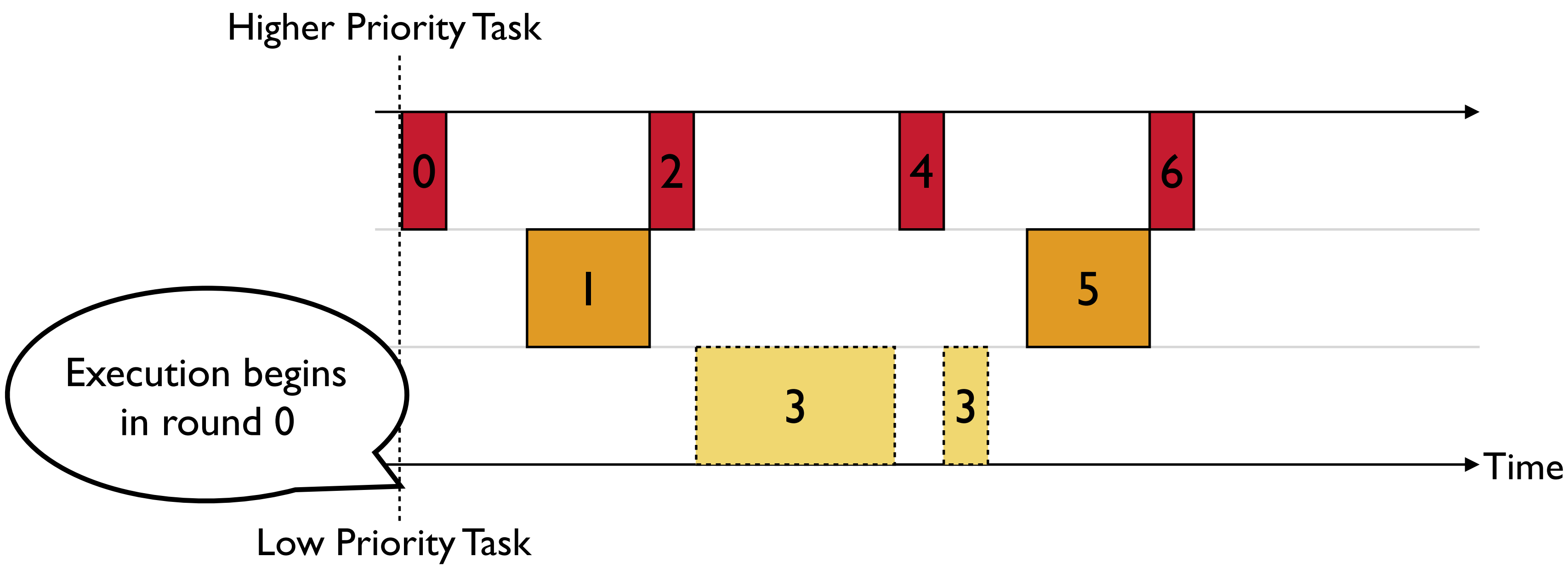


Job-bounded Abstraction



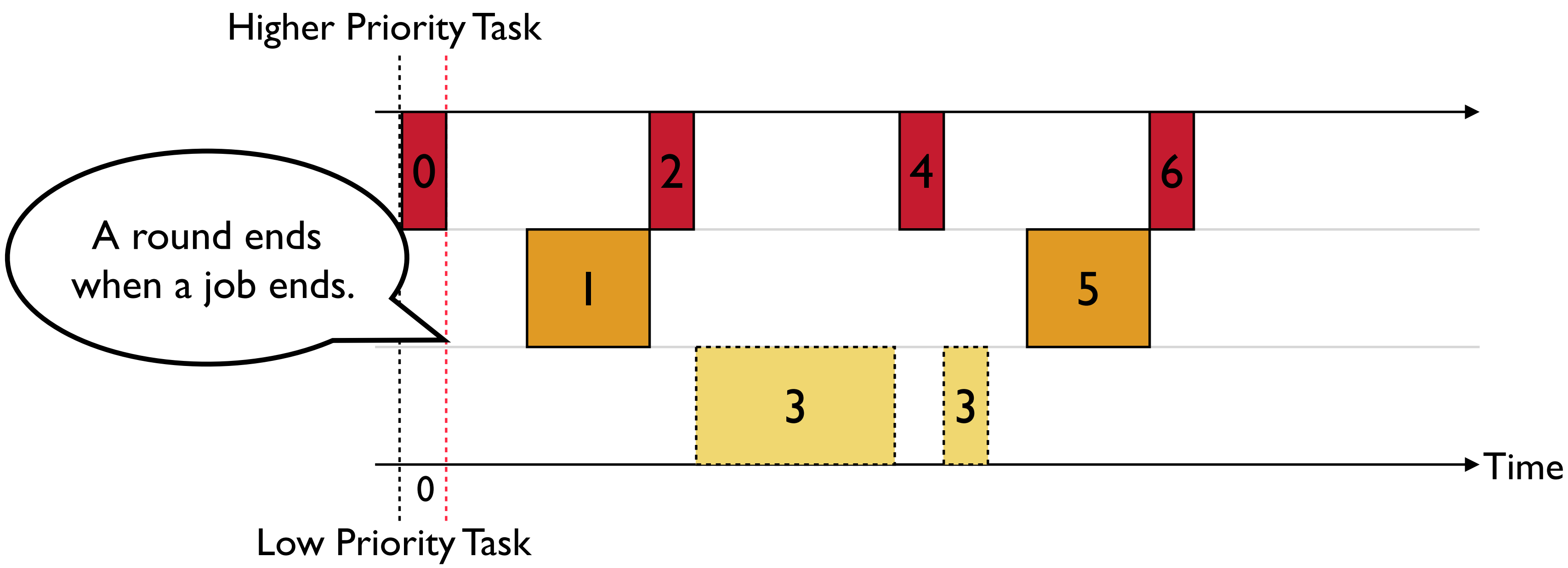
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



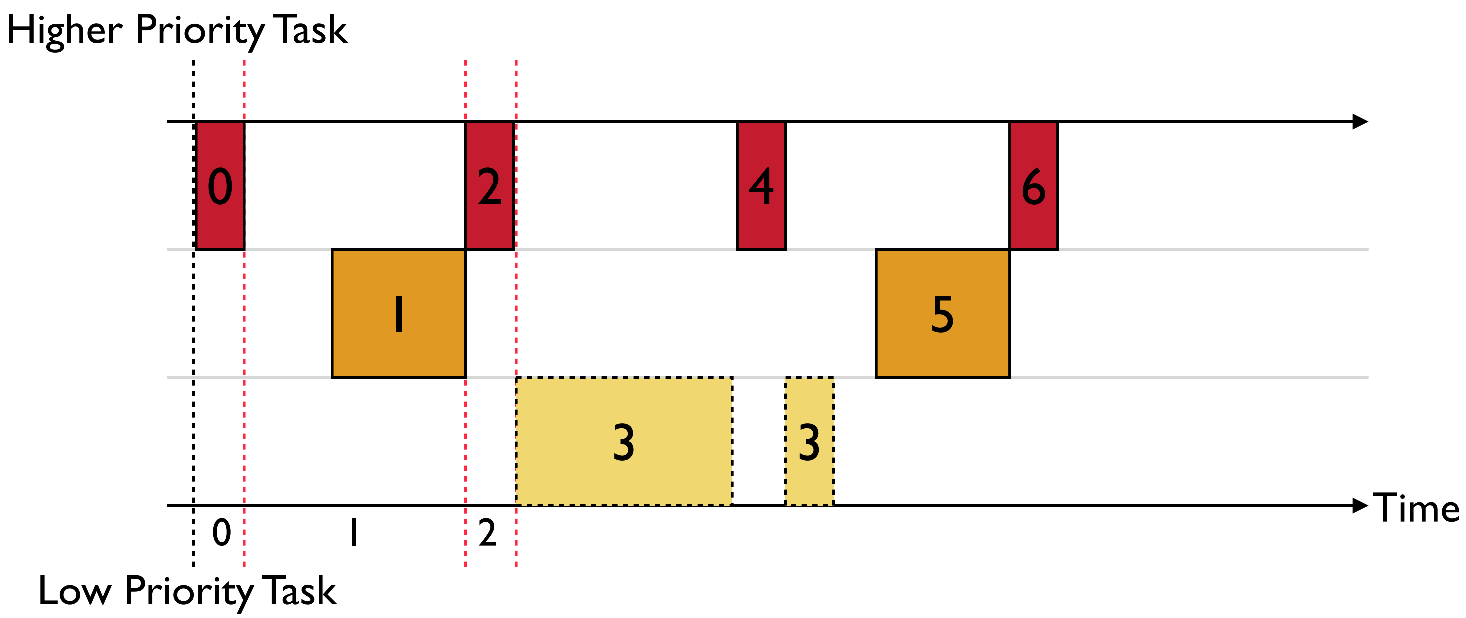
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



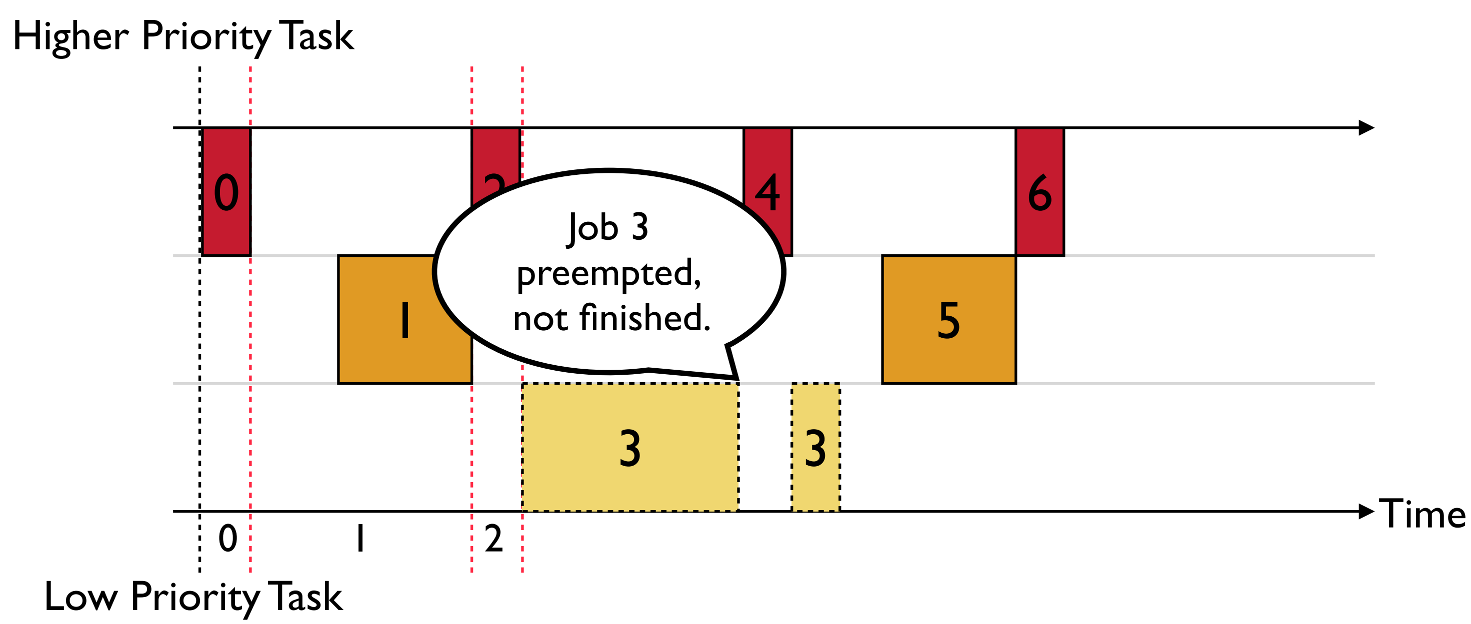
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



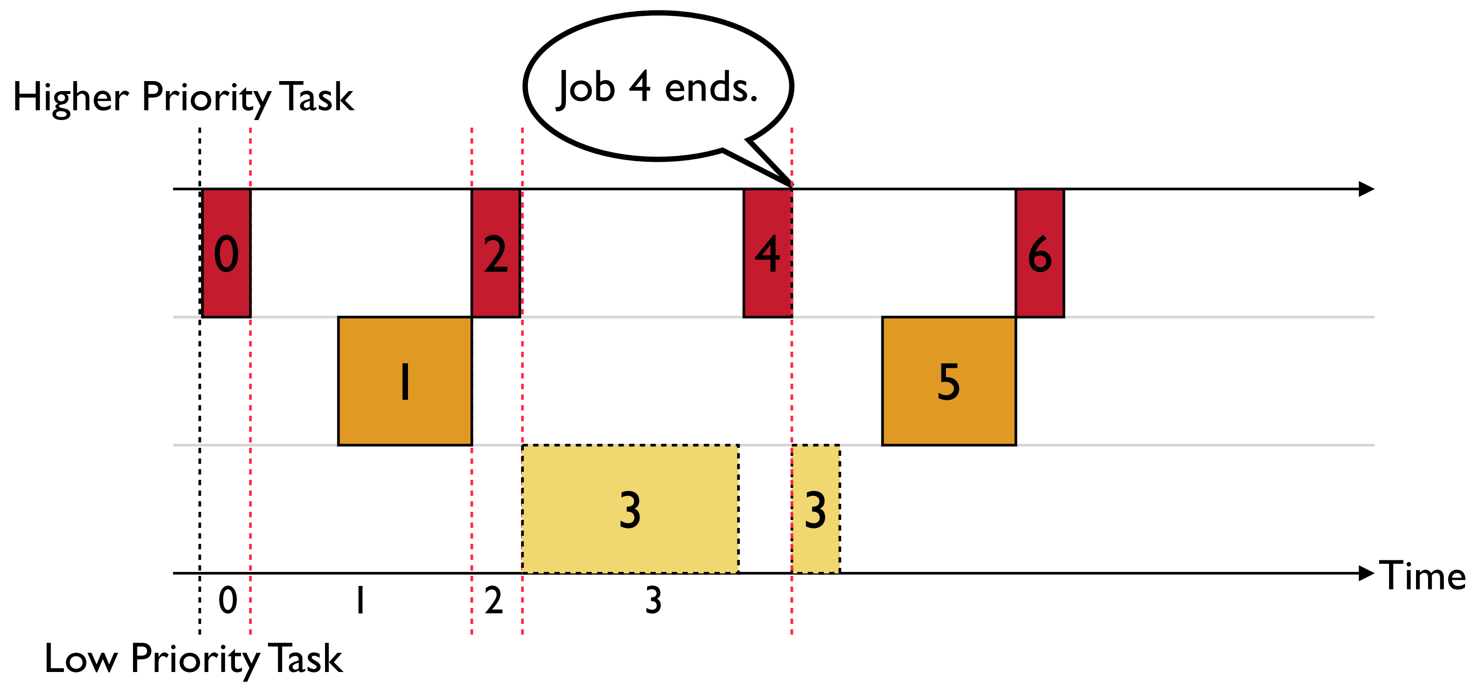
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



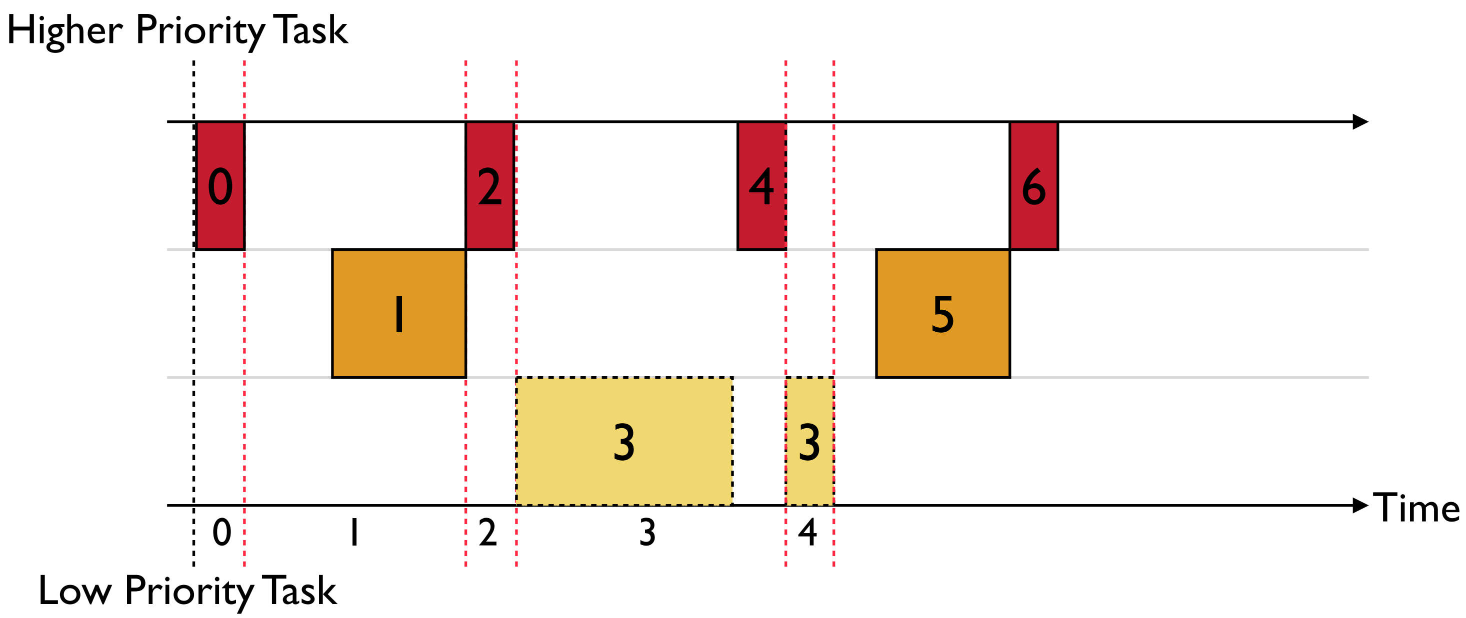
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



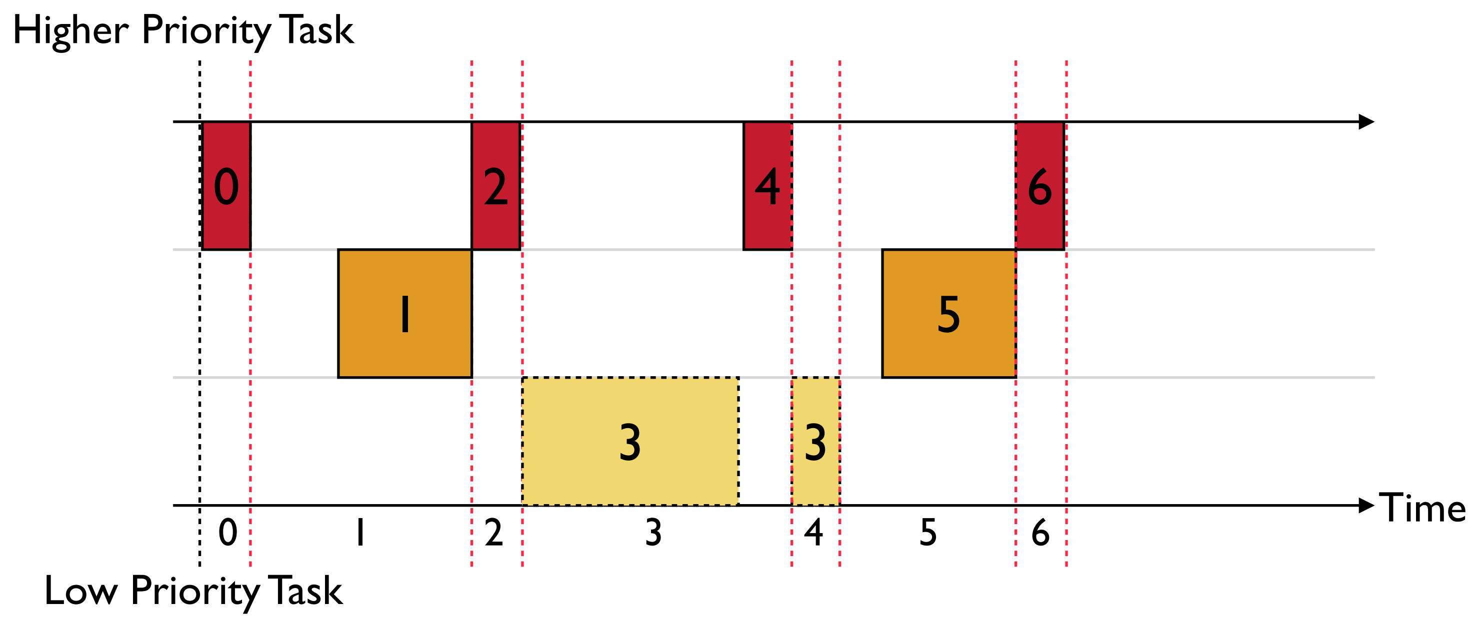
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



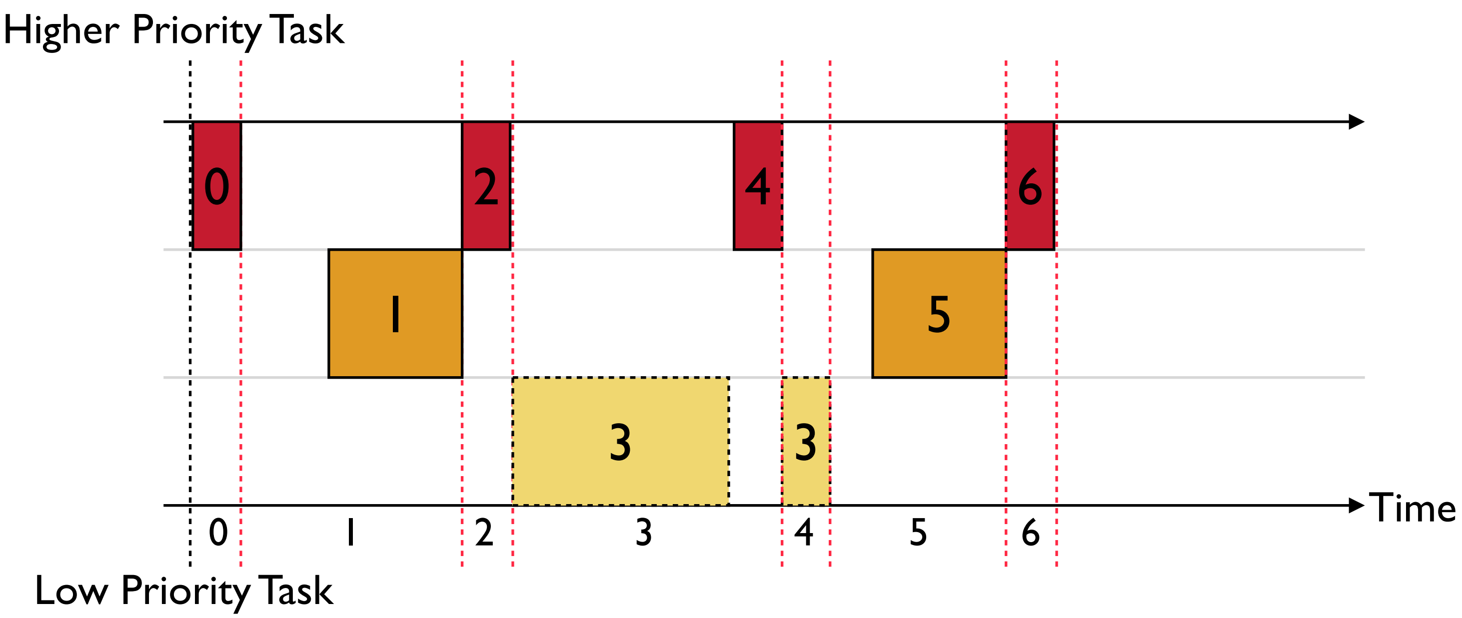
Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



Observation:
Any execution can be partitioned into *scheduling rounds*

Partition Execution into Rounds



Observation:
Any execution can be partitioned into *scheduling rounds*

$$\# \text{ of Jobs} = \# \text{ of Rounds}$$

Partition Execution into Rounds

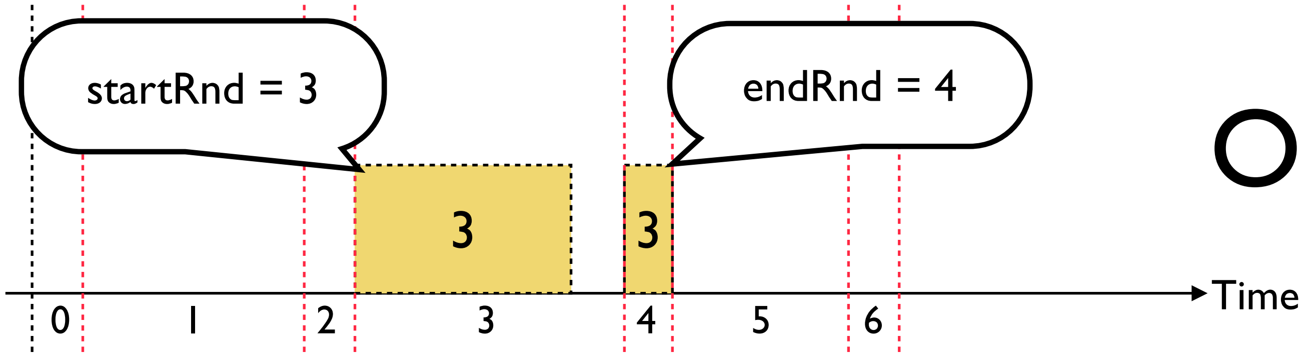
Add constraints to enforce **legal** job scheduling

1. Jobs are sequential:

MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

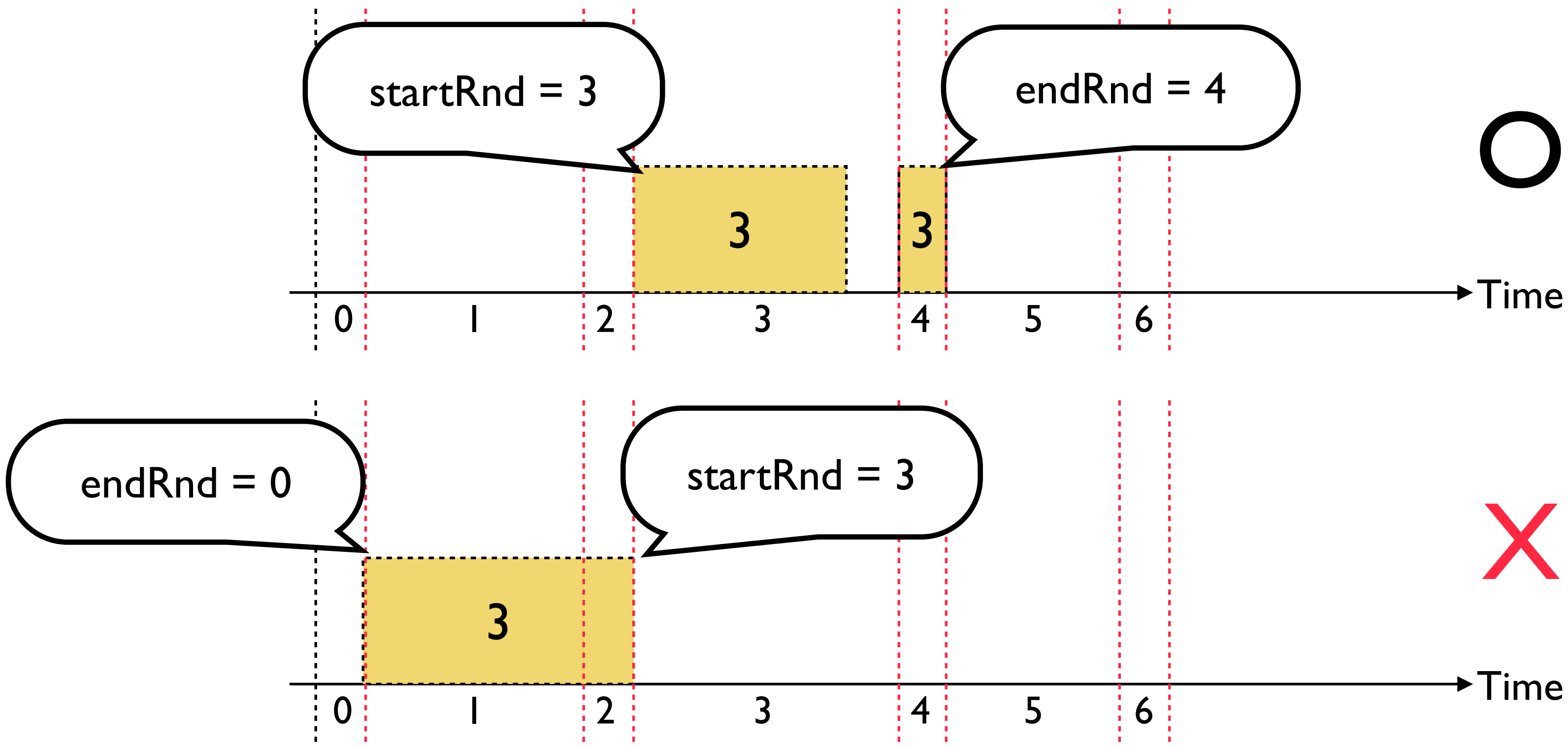
1. Jobs are sequential:



MONOSEQ Sequentialization

Add constraints to enforce **legal** job scheduling

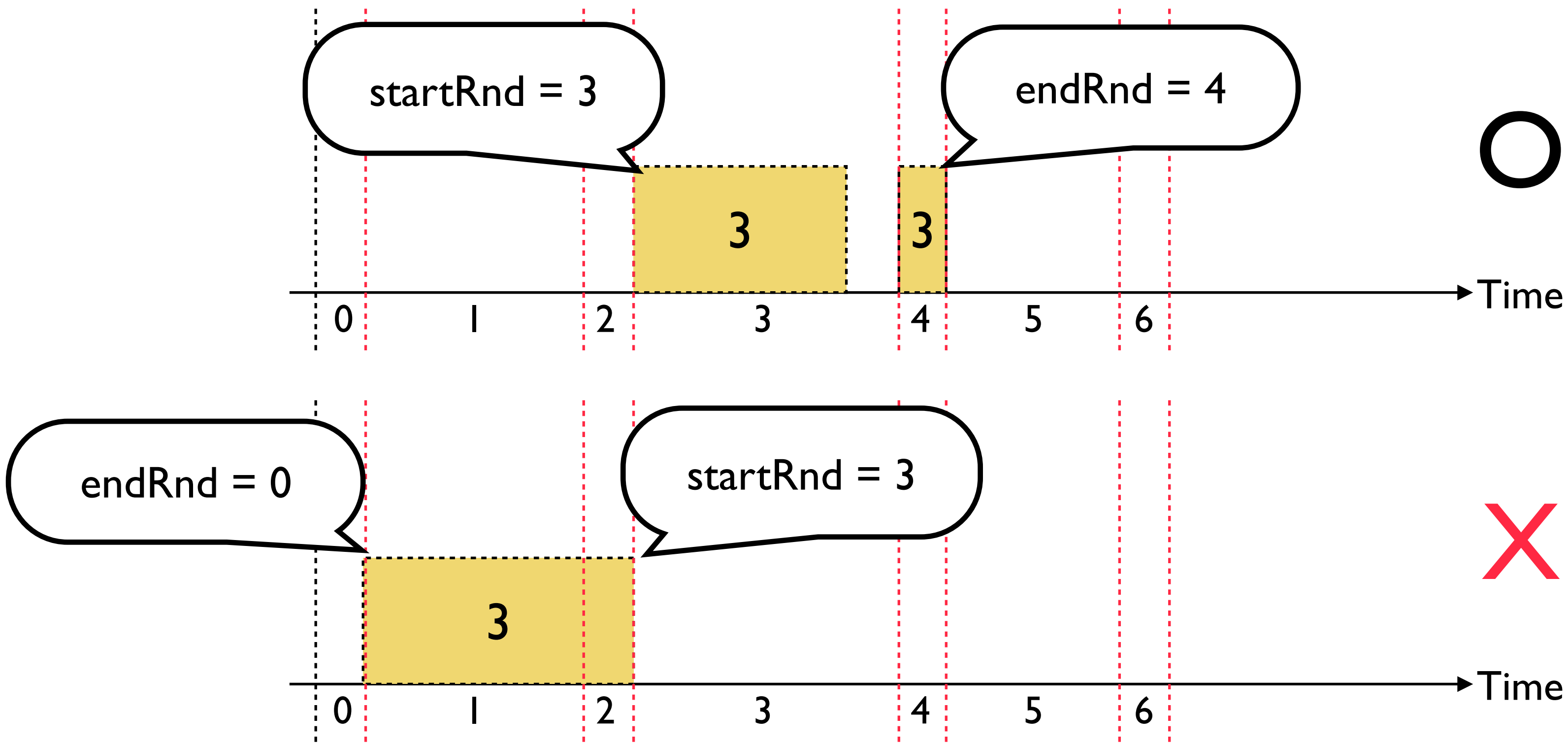
1. Jobs are sequential:



MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

1. Jobs are sequential:



// Jobs are sequential

$$\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$$
$$\text{assume}($$
$$0 \leq start[t][j] \leq end[t][j] \leq R \wedge$$
$$(\neg \text{last}(t, j) \Rightarrow end[t][j] \leq start[t][j+1]))$$

MonoSeq **Sequentialization**

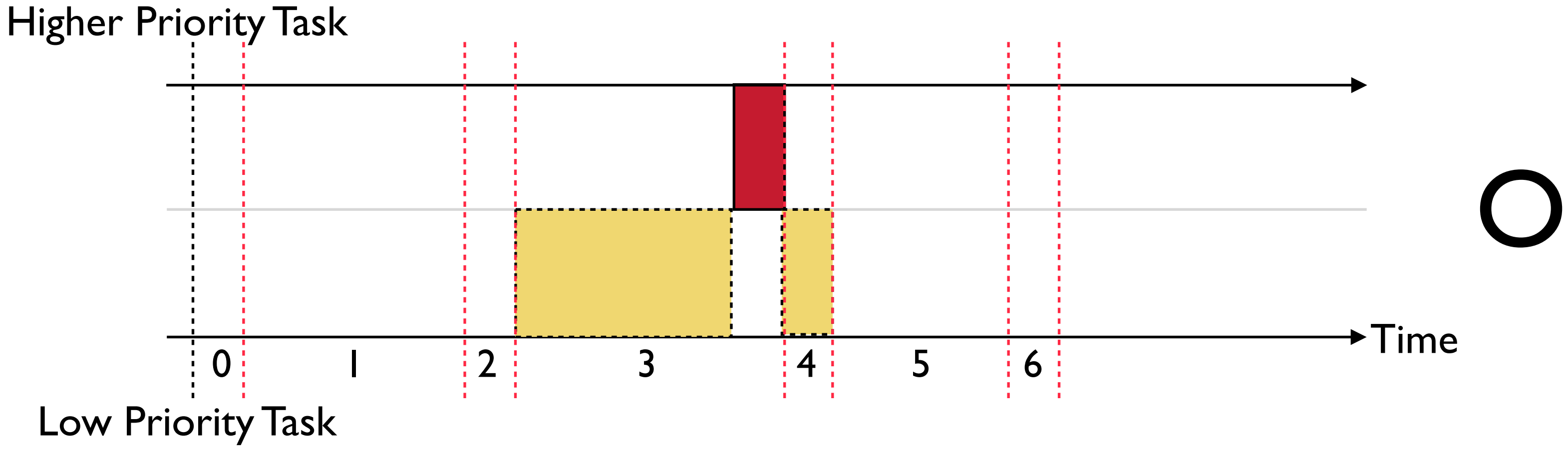
Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:

MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

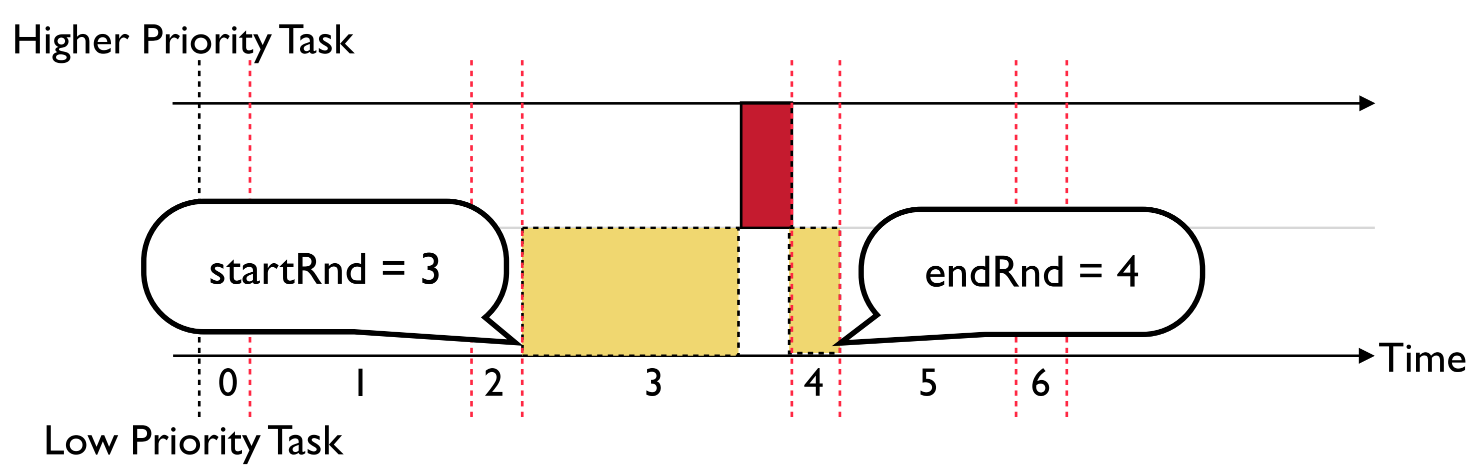
2. Jobs are well-nested:



MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

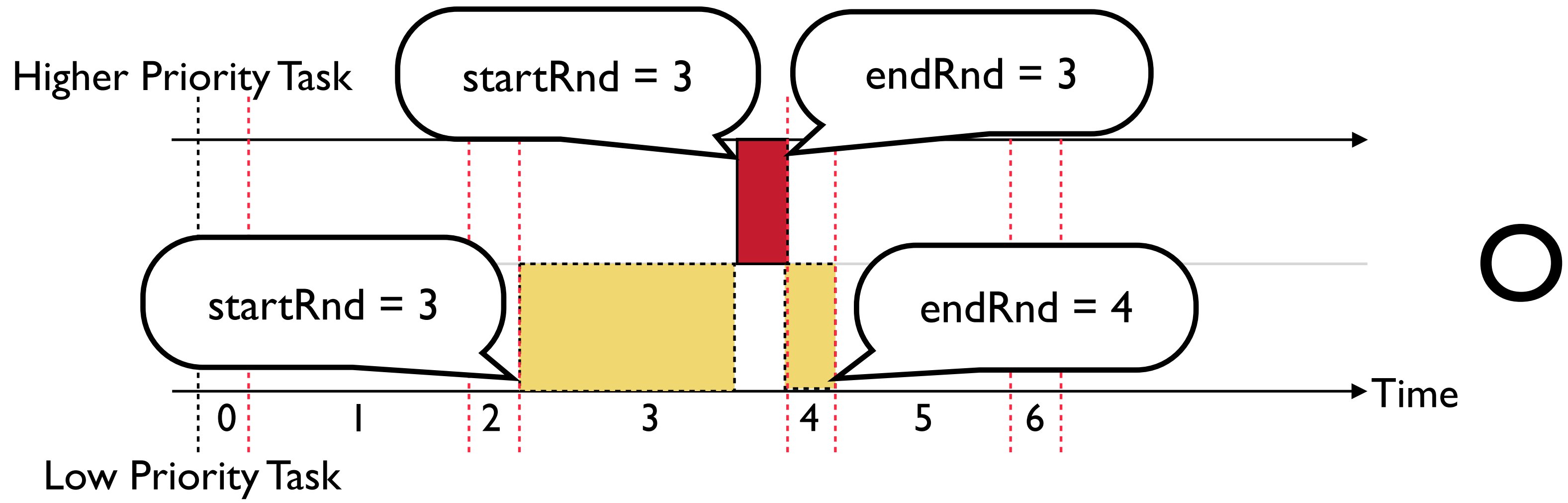
2. Jobs are well-nested:



MONOSEQ Sequentialization

Add constraints to enforce **legal** job scheduling

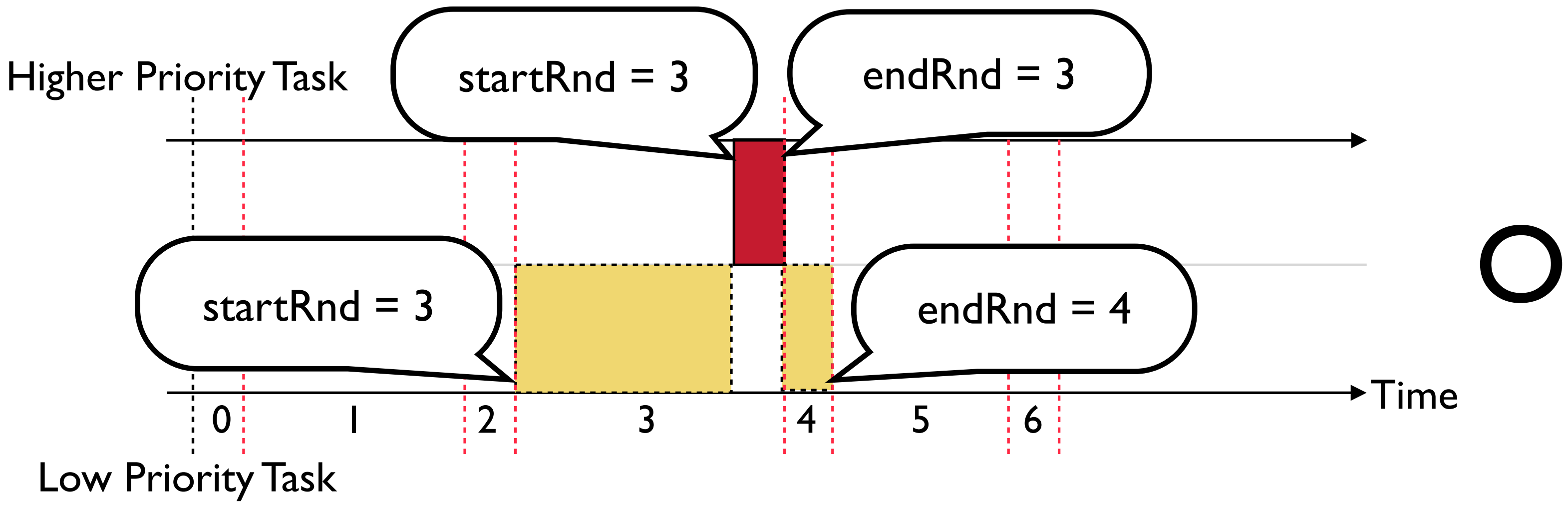
2. Jobs are well-nested:

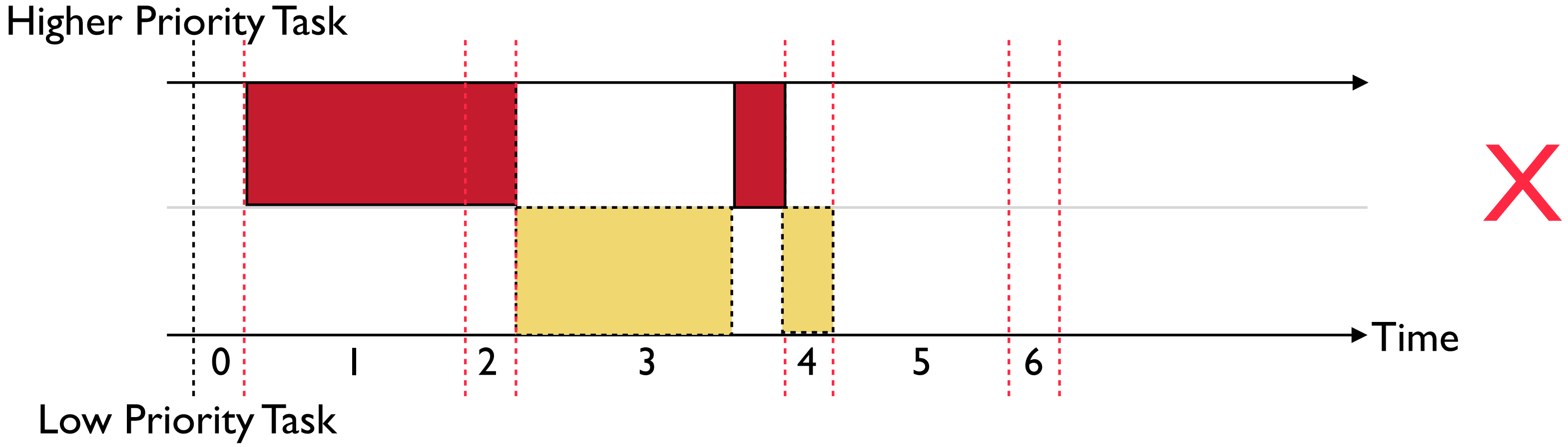


MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:

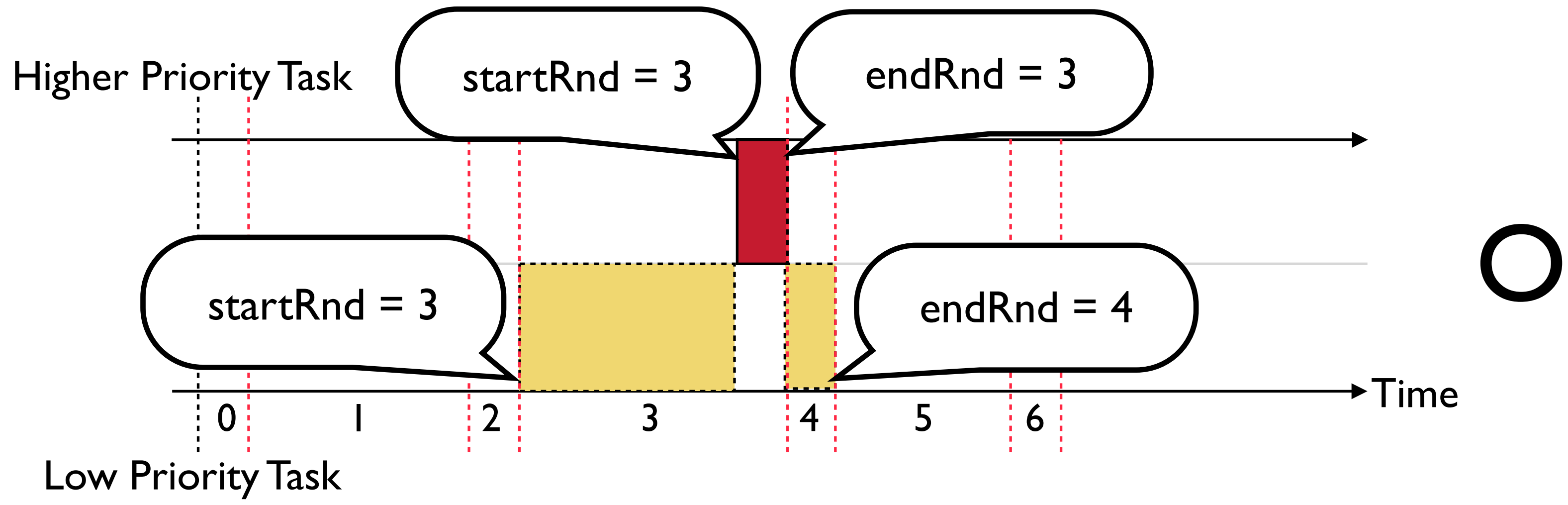


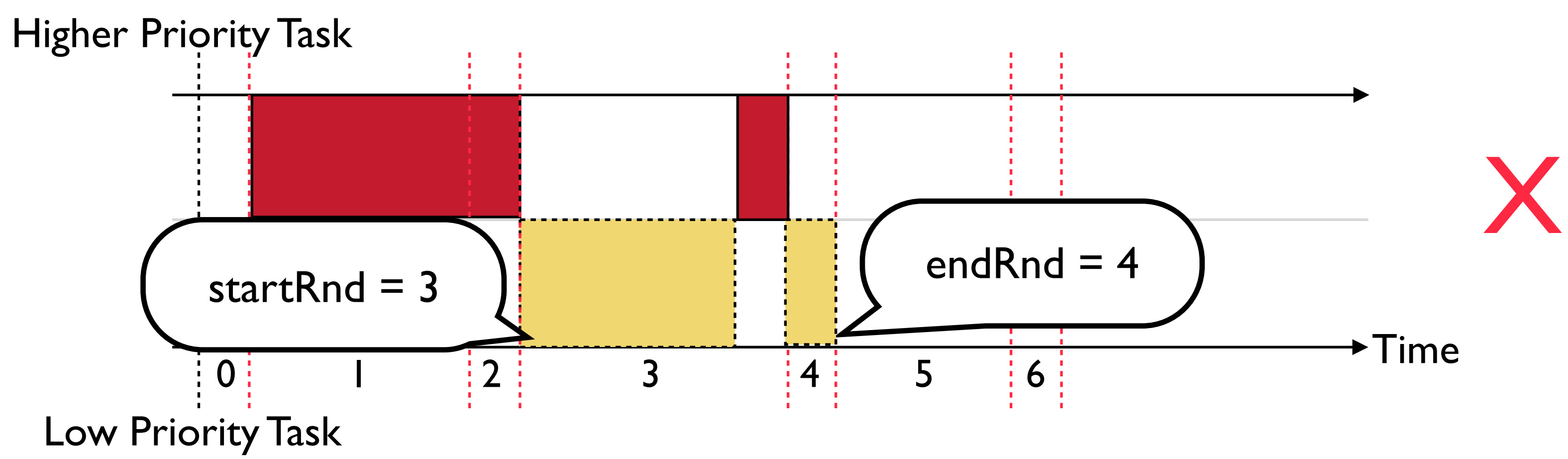


MONOSEQ Sequentialization

Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:

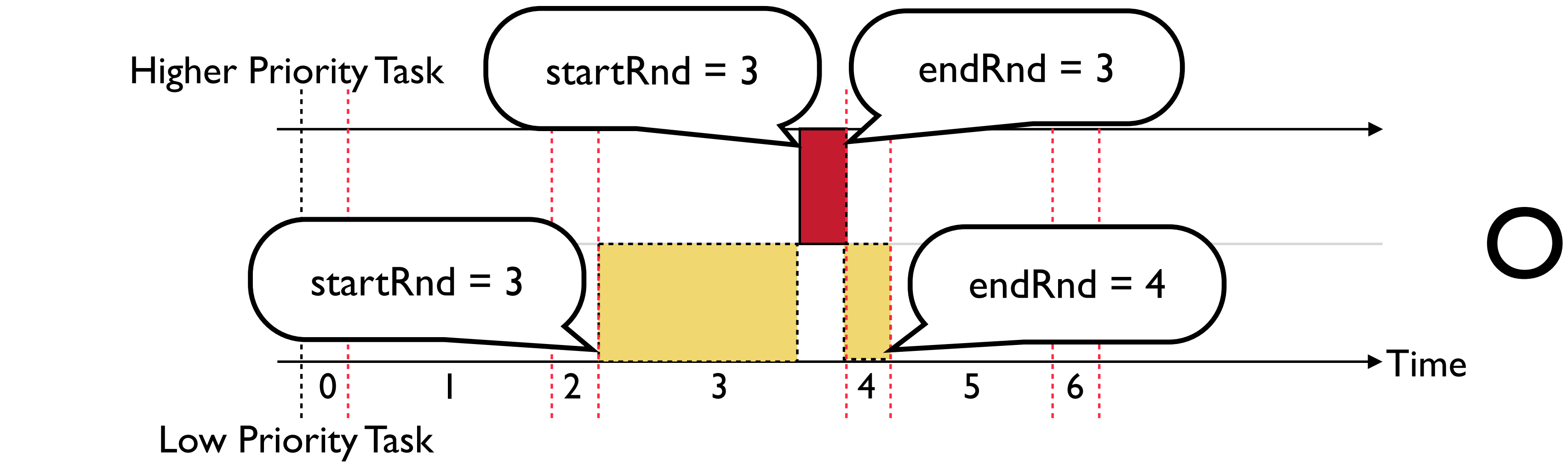


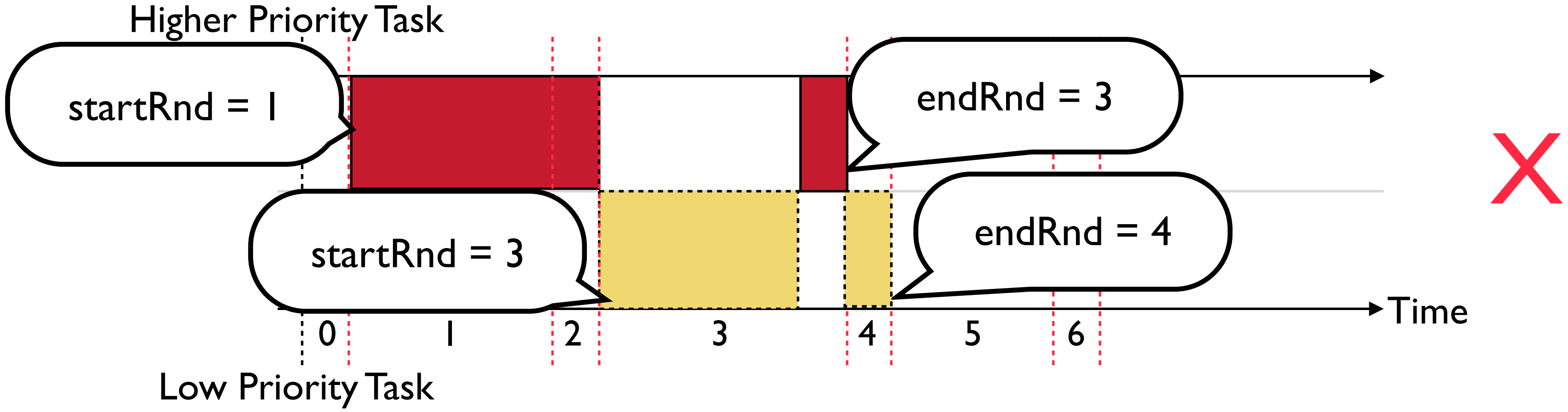


MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:

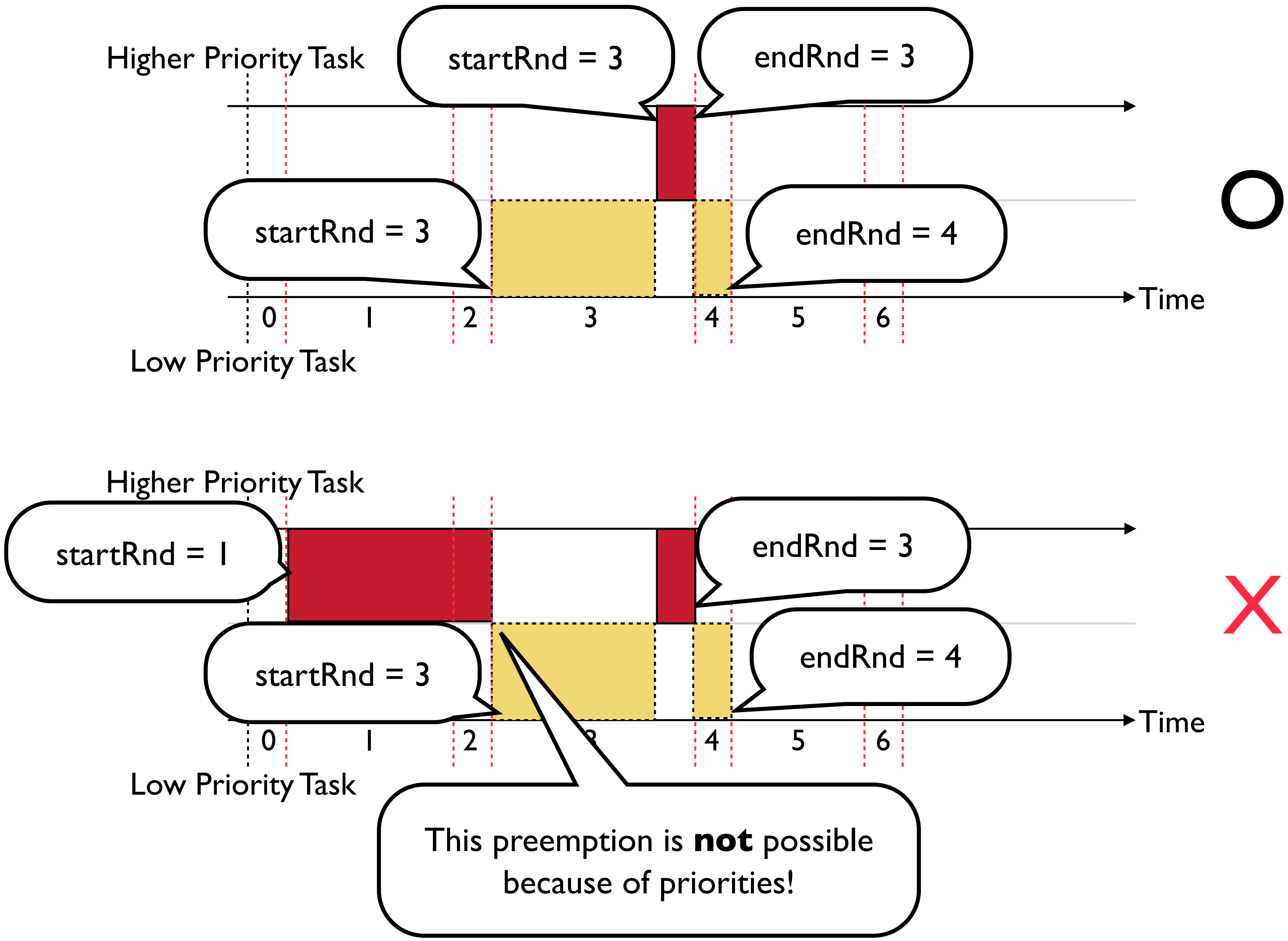




MONOSEQ Sequentialization

Add constraints to enforce **legal** job scheduling

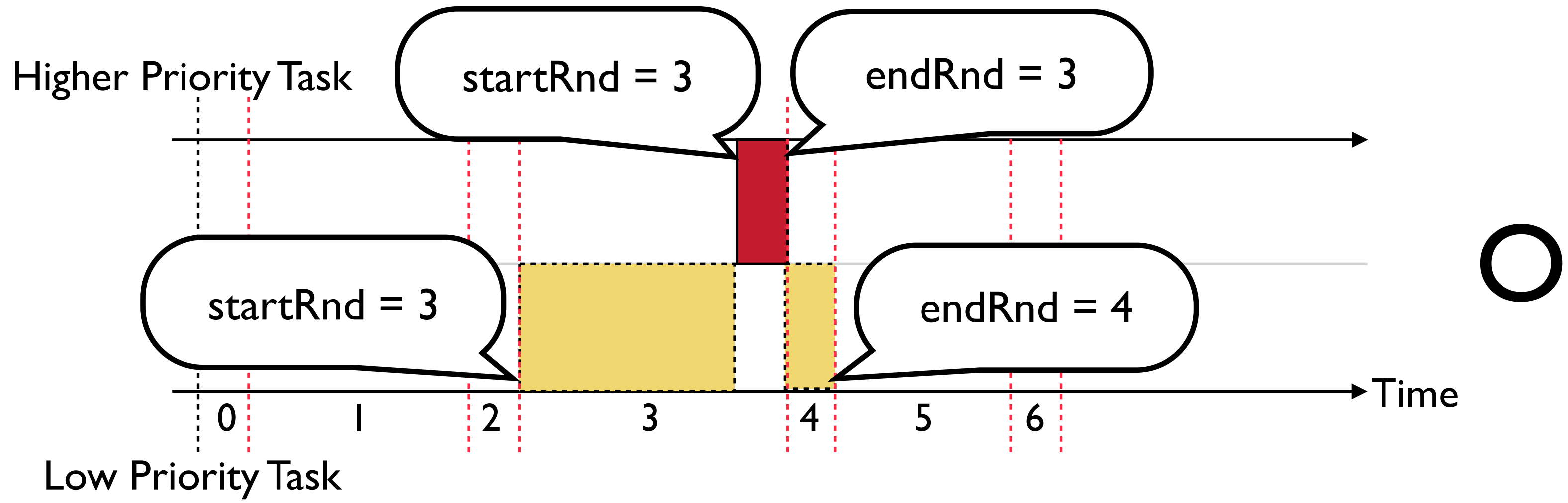
2. Jobs are well-nested:

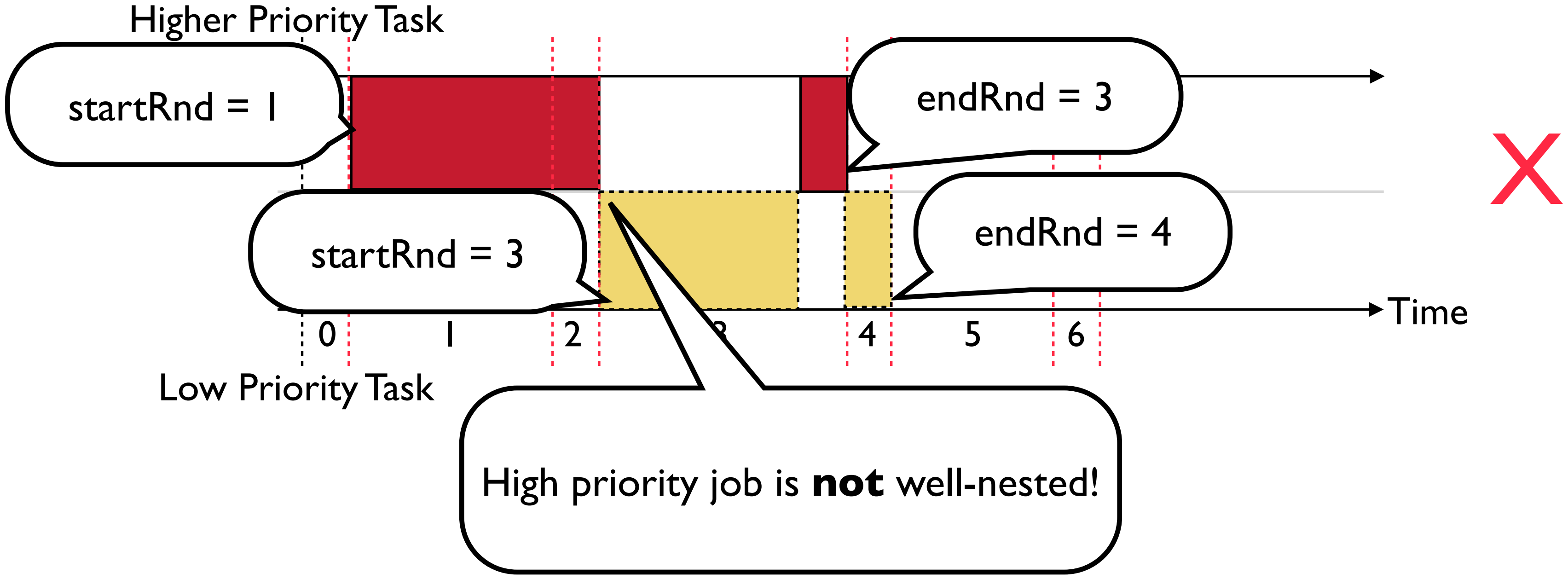


MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:

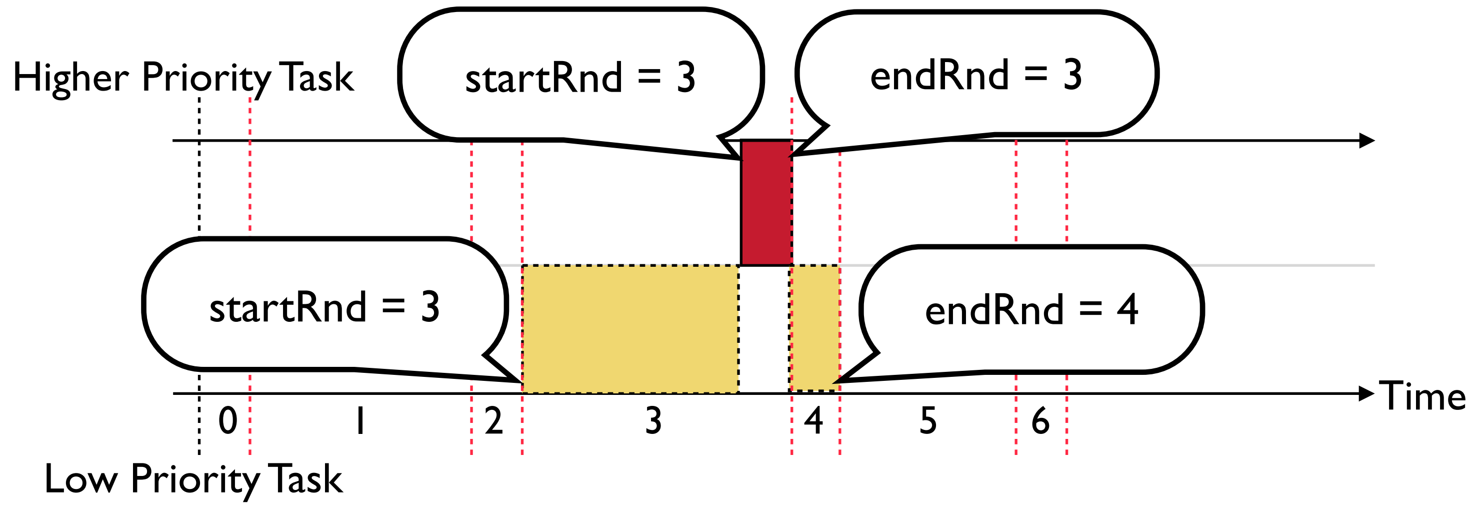




MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

2. Jobs are well-nested:



// Jobs are well-nested

$$\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$$
$$\text{assume}($$
$$\big(t_1 < t_2 \wedge$$
$$start[t_1][j_1] \leq end[t_2][j_2] \wedge$$
$$start[t_2][j_2] \leq end[t_1][j_1]\big) \Rightarrow$$
$$(start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]))$$

MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

3. Jobs respect preemption bounds:

$PB_{t_1}^{t_2}$ = Upper bound on the number of times t_1 can preempt t_2 .

$\textsc{MonoSeq}$ Sequentialization

Add constraints to enforce **legal** job scheduling

3. Jobs respect preemption bounds:

RMA(Rate Monotonic Analysis) defines it

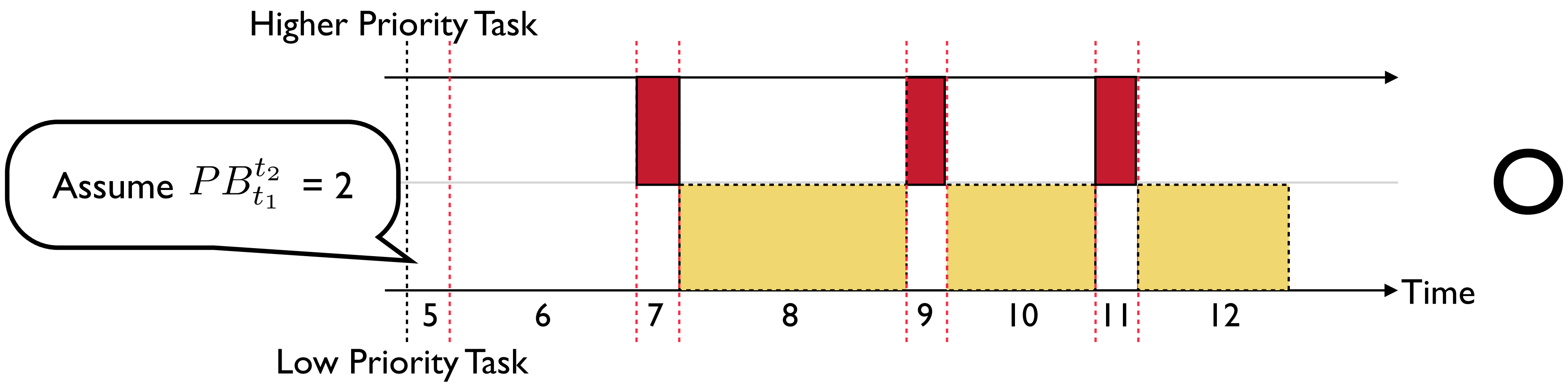
$PB_{t_1}^{t_2}$ = Upper bound on the number of times t_1 can preempt t_2 .

$\textsc{MonoSeq}$ Sequentialization

Add constraints to enforce **legal** job scheduling

3. Jobs respect preemption bounds:

$PB_{t_1}^{t_2}$ = Upper bound on the number of times t_1 can preempt t_2 .

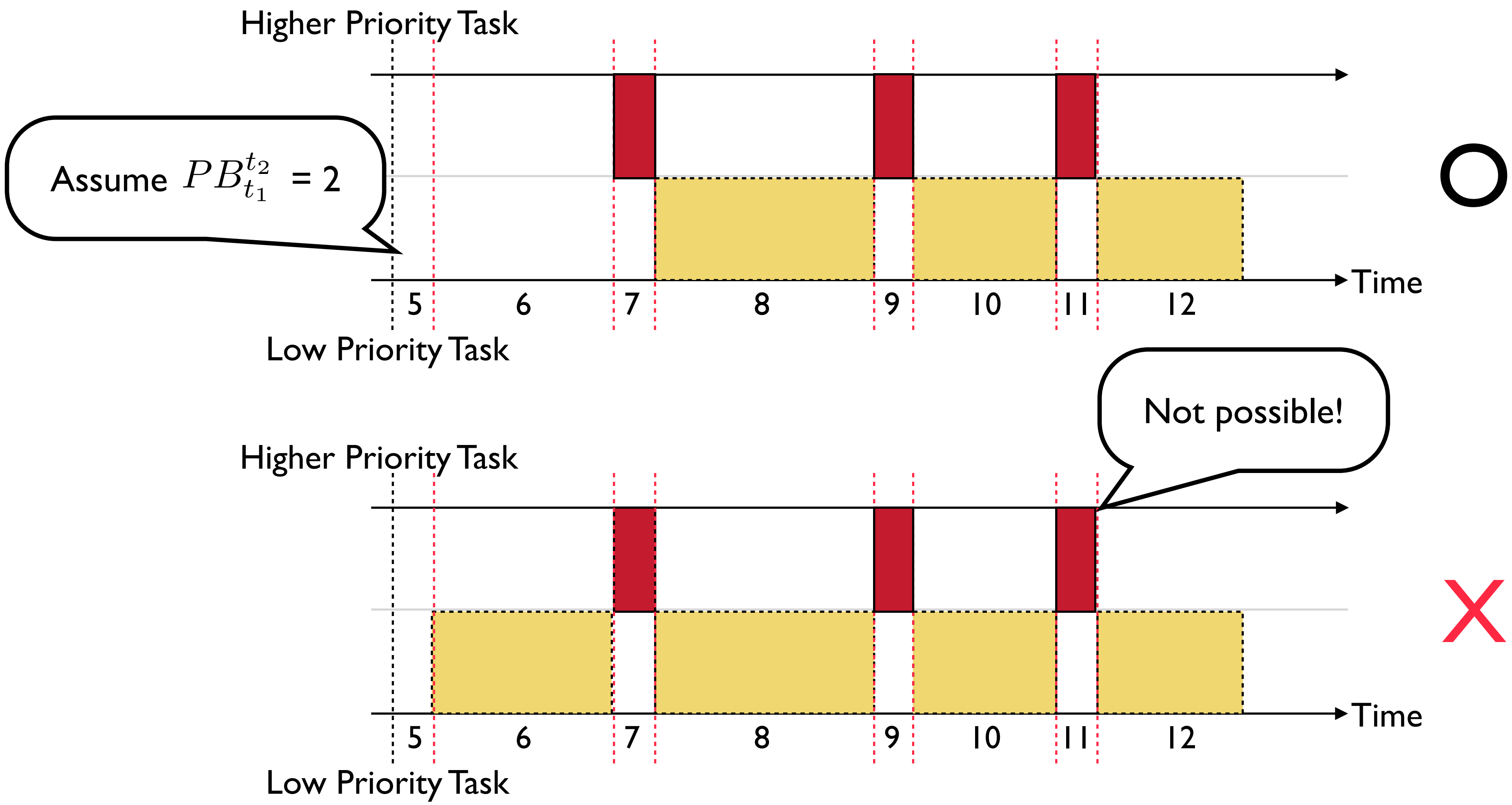


MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

3. Jobs respect preemption bounds:

$PB_{t_1}^{t_2}$ = Upper bound on the number of times t_1 can preempt t_2 .

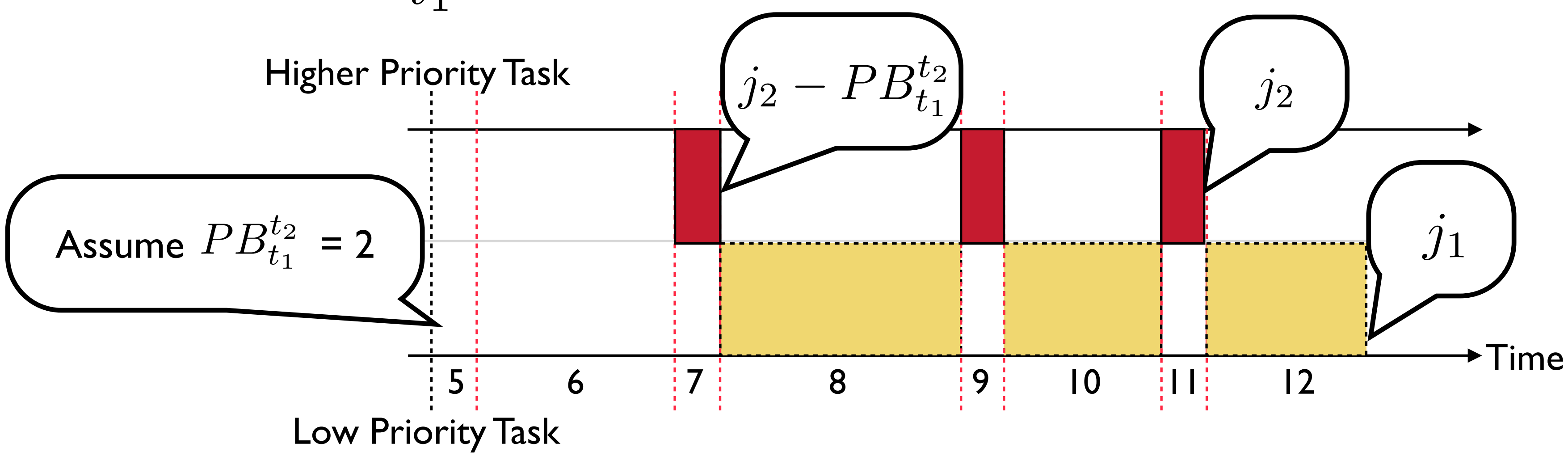


MonoSeq Sequentialization

Add constraints to enforce **legal** job scheduling

3. Jobs respect preemption bounds:

$PB_{t_1}^{t_2}$ = Upper bound on the number of times t_1 can preempt t_2 .

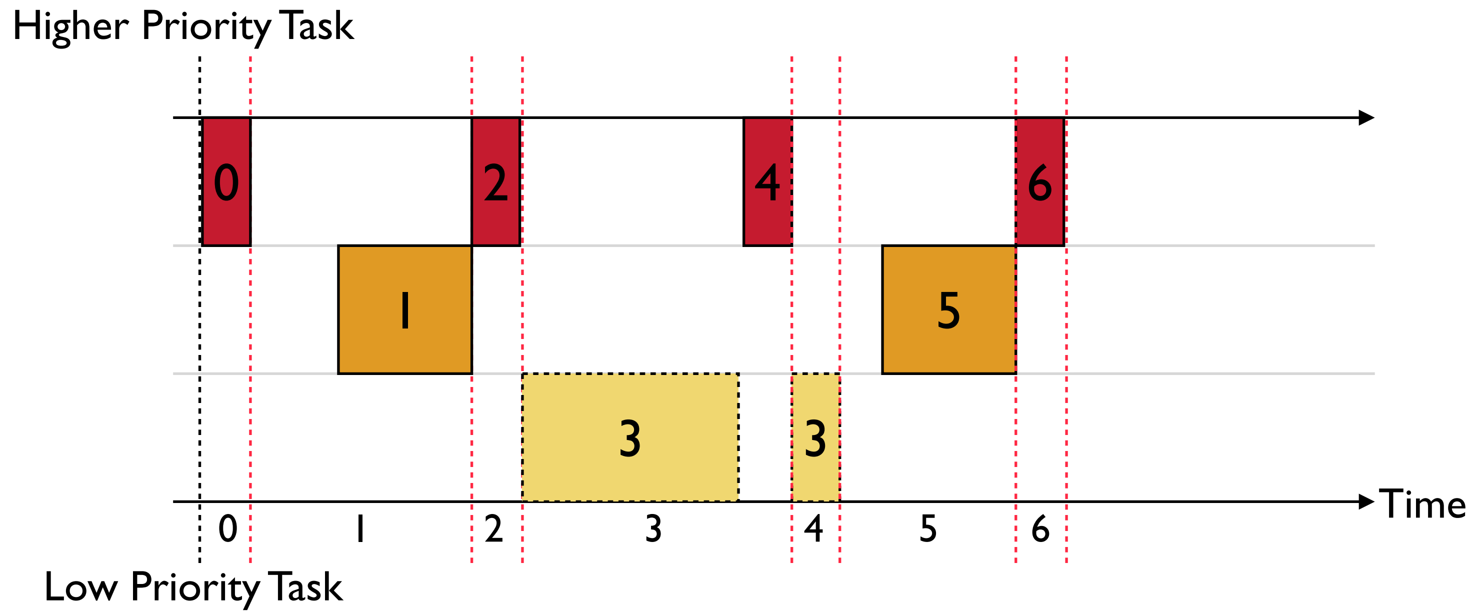


// Jobs respect preemption bounds

$$\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$$
$$\text{assume}($$
$$\big(t_1 < t_2 \wedge j_2 \geq PB_{t_1}^{t_2} \wedge$$
$$start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]\big) \Rightarrow$$
$$end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])$$

MonoSeq Sequentialization

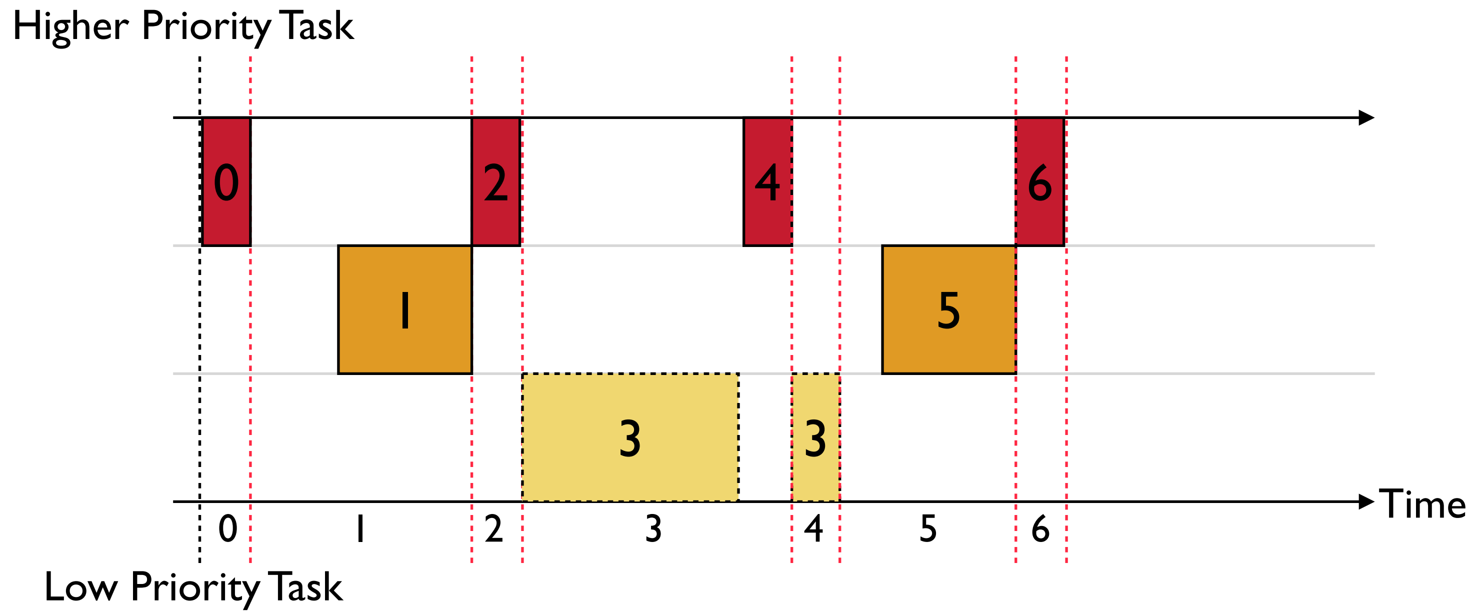
Sequential Execution



Replace accesses to global variable g with g[rnd]

MonoSeq Sequentialization

Sequential Execution

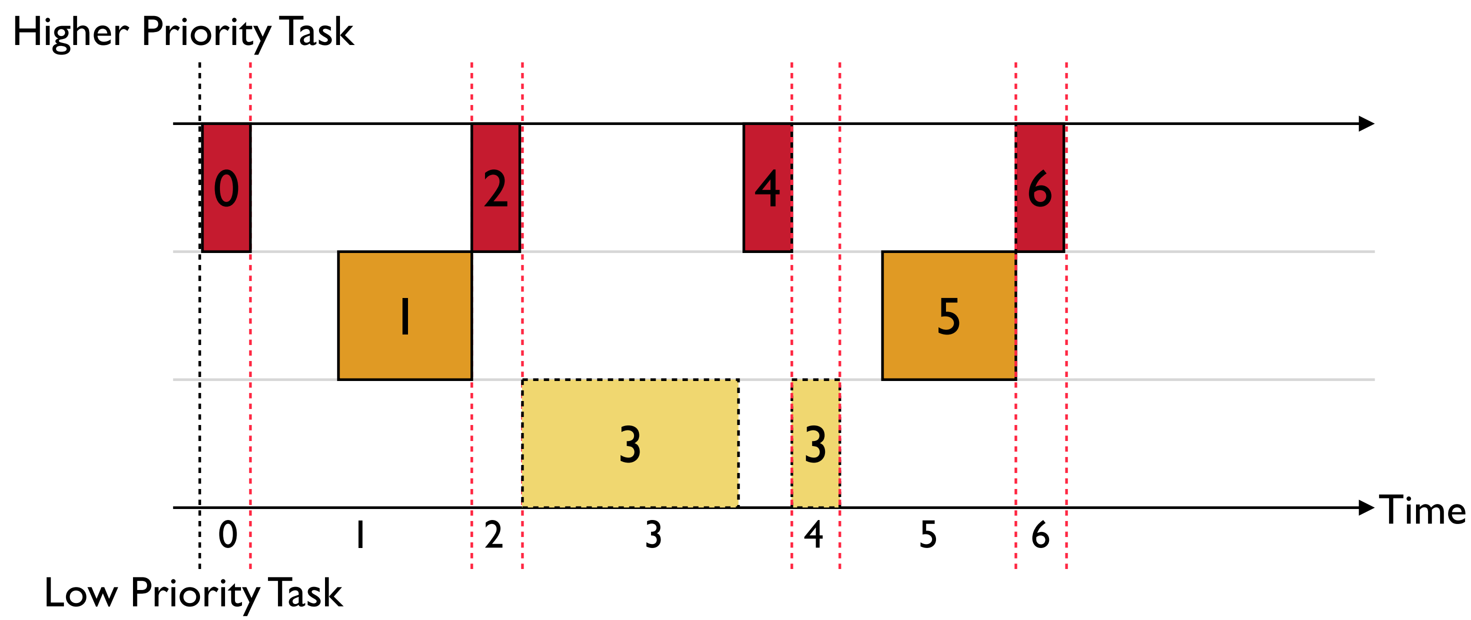


Replace accesses to global variable g with g[rnd]

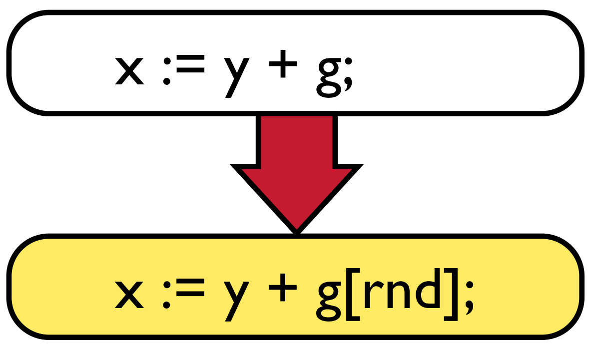
```
x := y + g;
```

MonoSeq Sequentialization

Sequential Execution

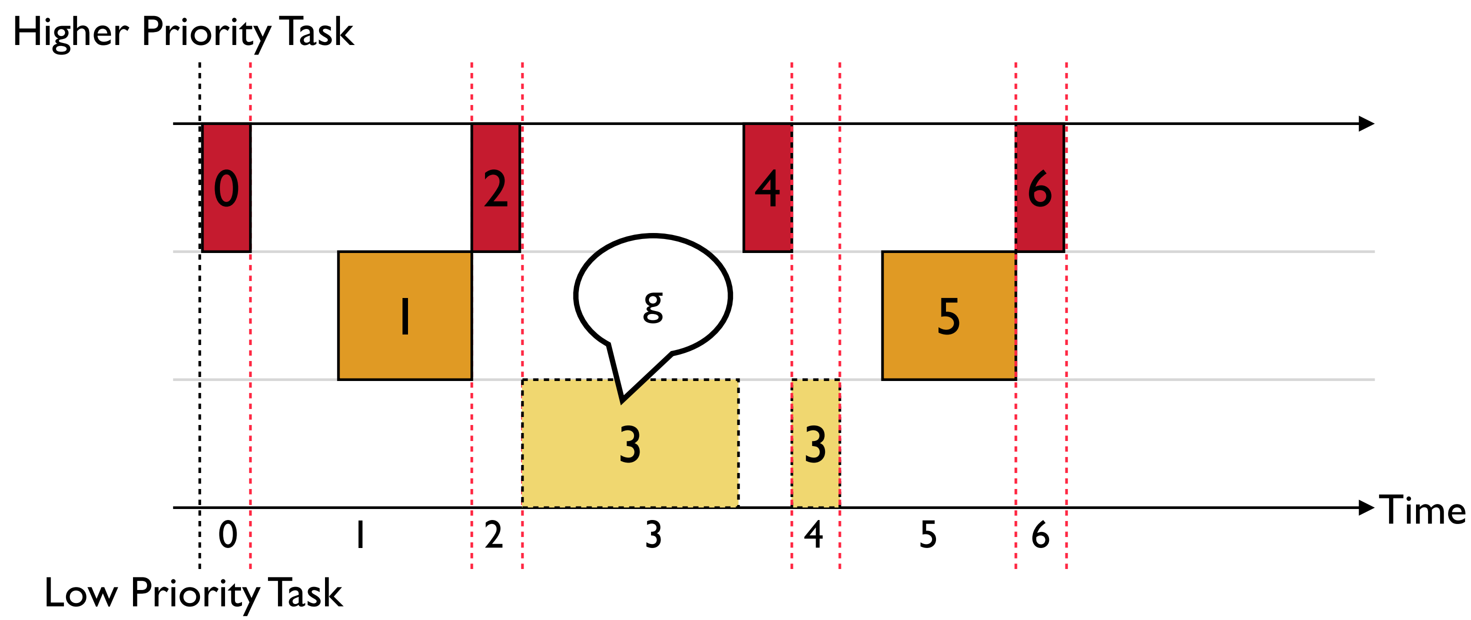


Replace accesses to global variable g with g[rnd]

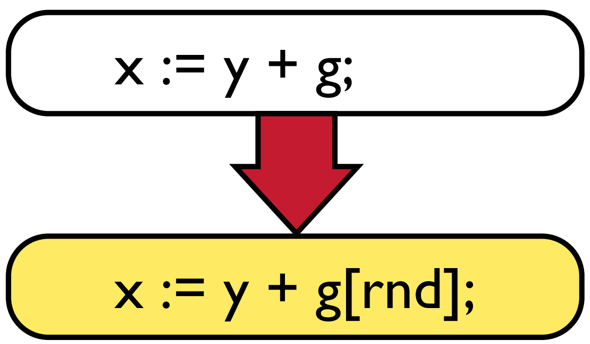


MONOSEQ Sequentialization

Sequential Execution

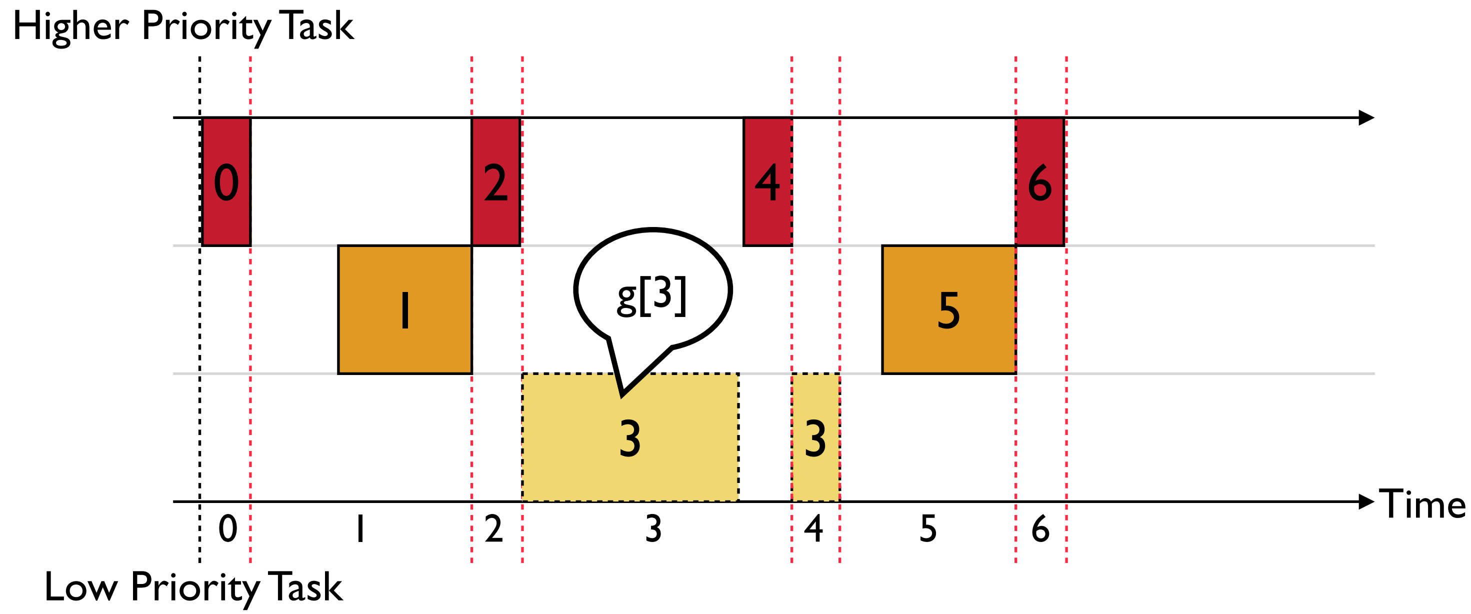


Replace accesses to global variable g with g[rnd]

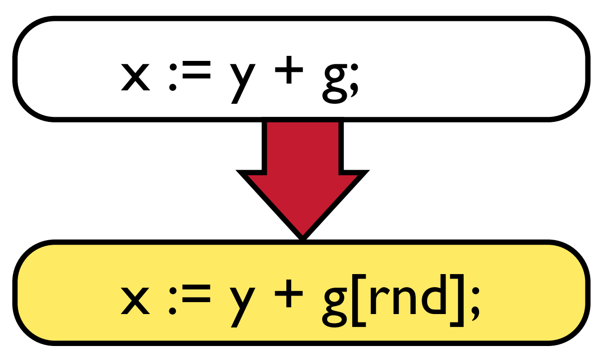


MONOSEQ Sequentialization

Sequential Execution

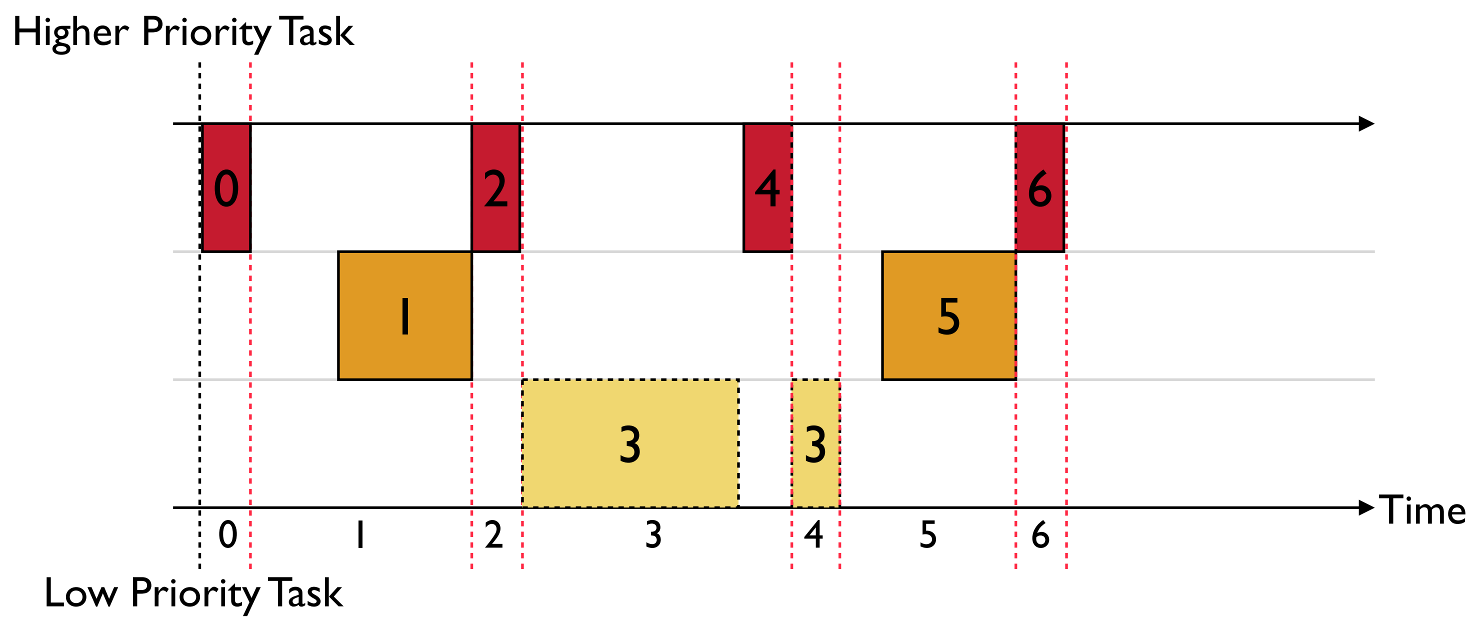


Replace accesses to global variable g with g[rnd]

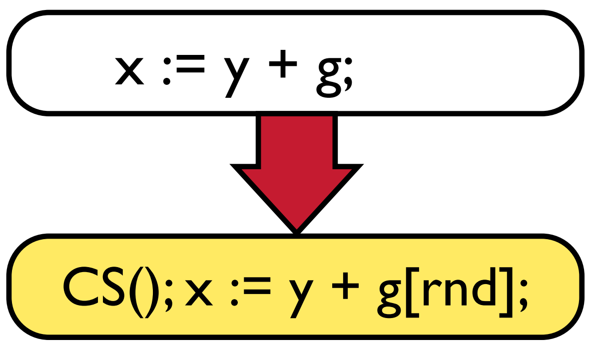


MONOSEQ Sequentialization

Sequential Execution

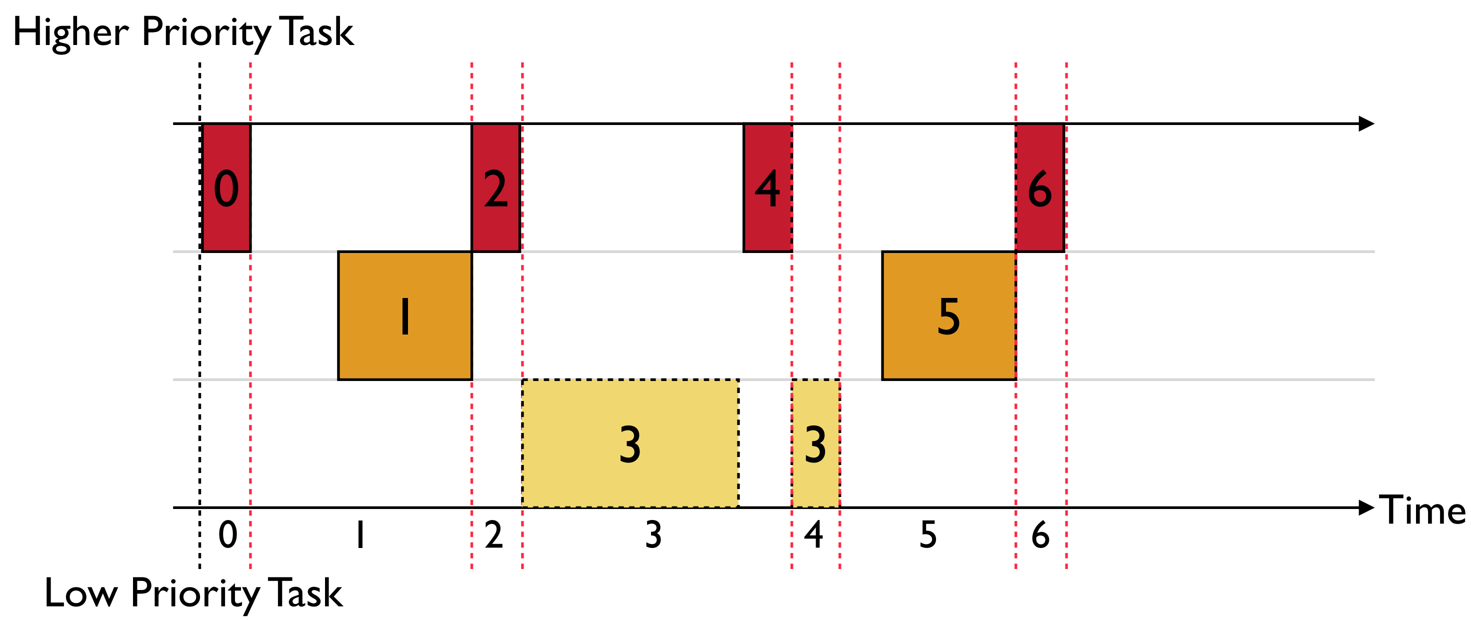


Add non-deterministic **context-switching** to statements

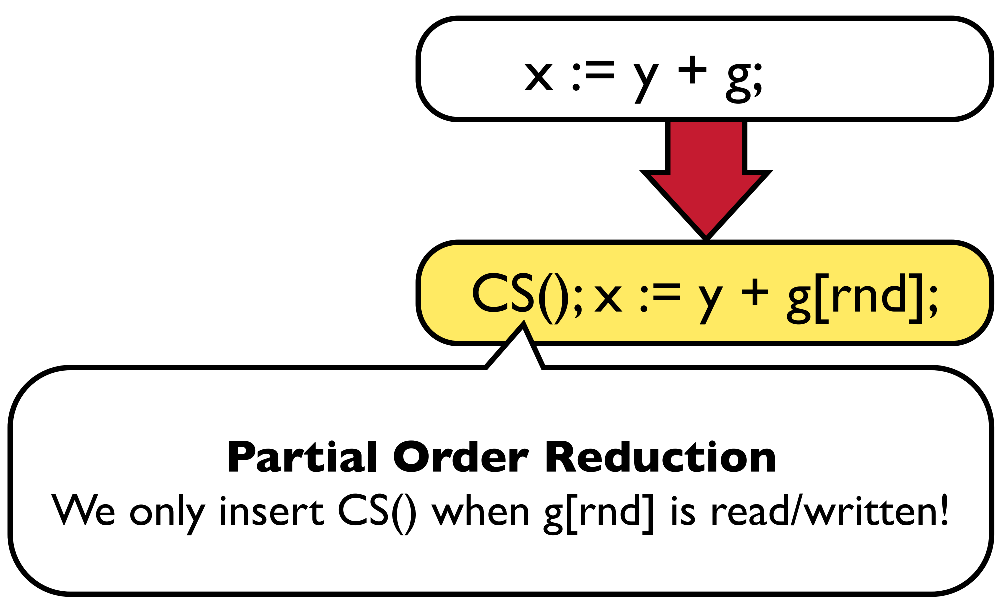


MonoSeq Sequentialization

Sequential Execution

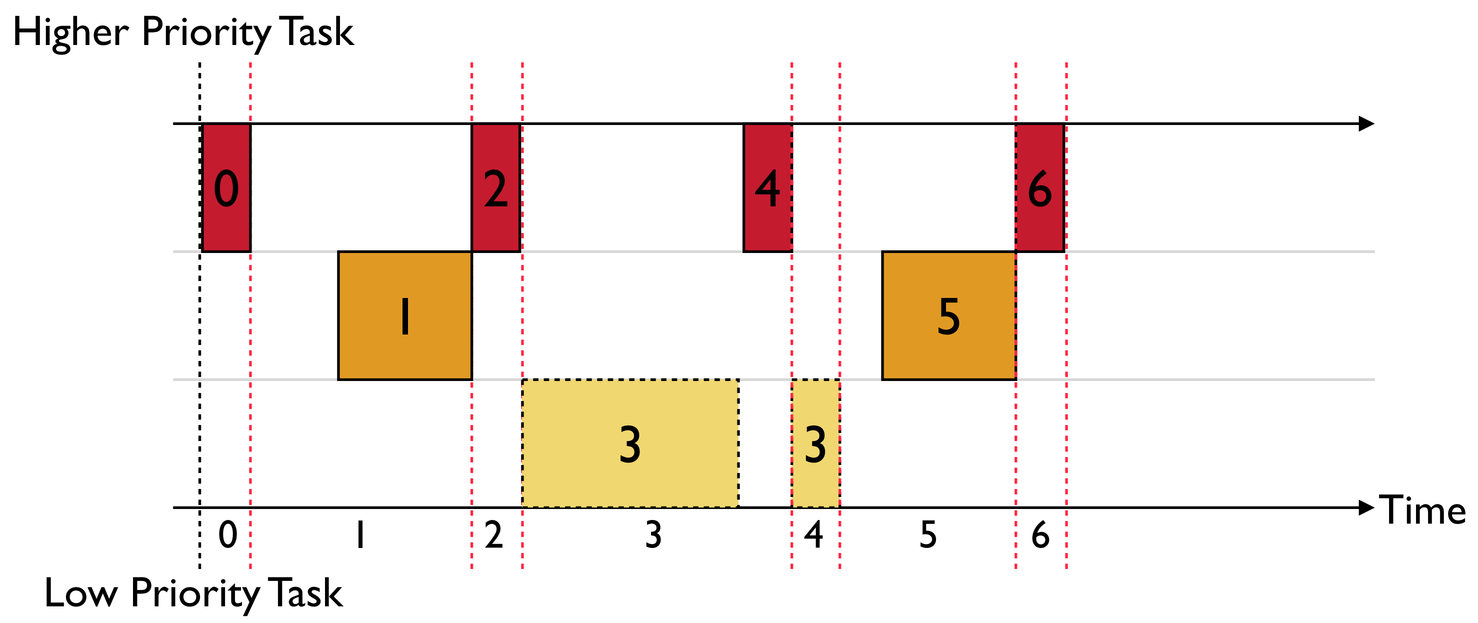


Add non-deterministic **context-switching** to statements



MONOSEQ Sequentialization

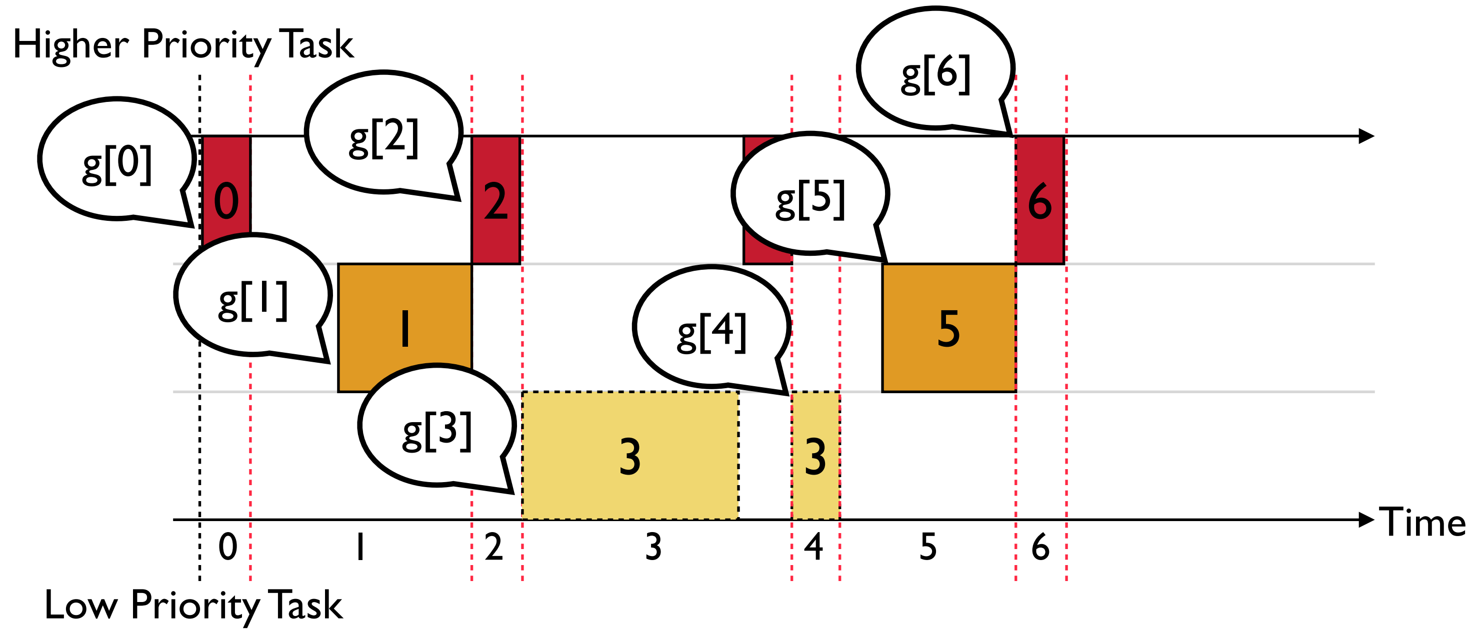
Sequential Execution



Guess non-deterministic initial value of each global in each round.

MonoSeq Sequentialization

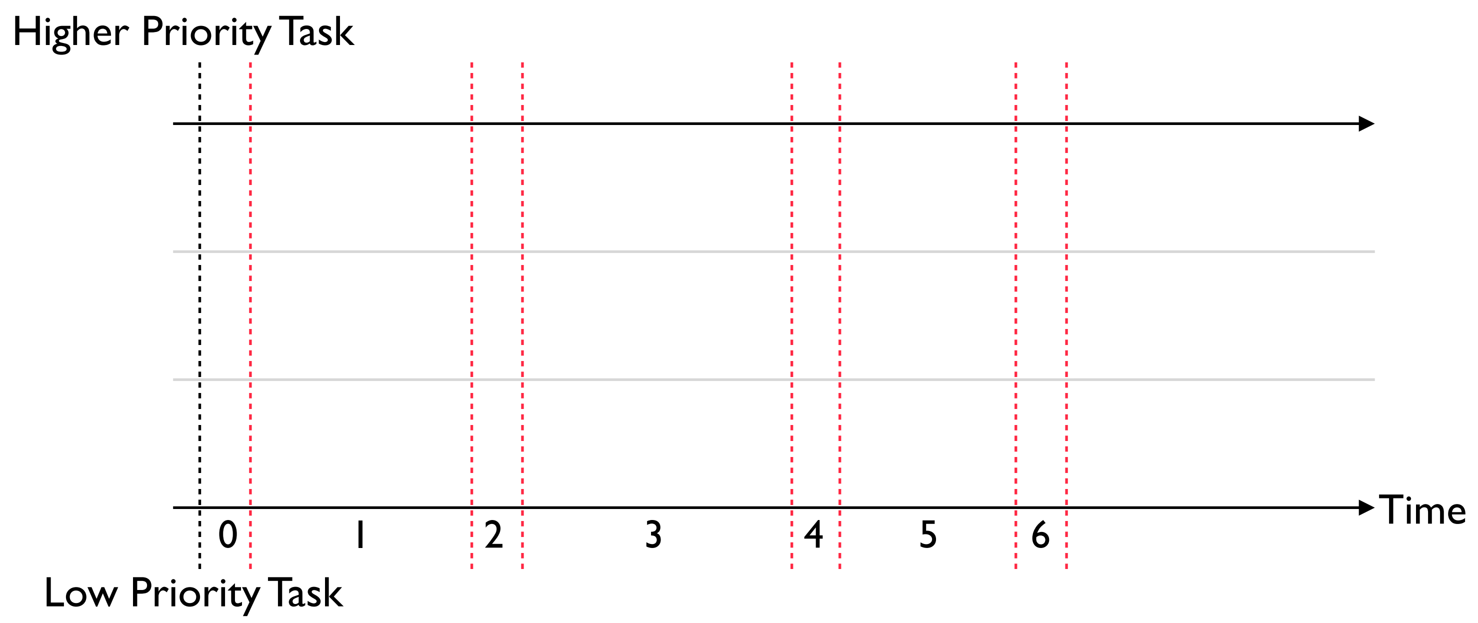
Sequential Execution



Guess non-deterministic initial value of each global in each round.

MonoSeq Sequentialization

Sequential Execution

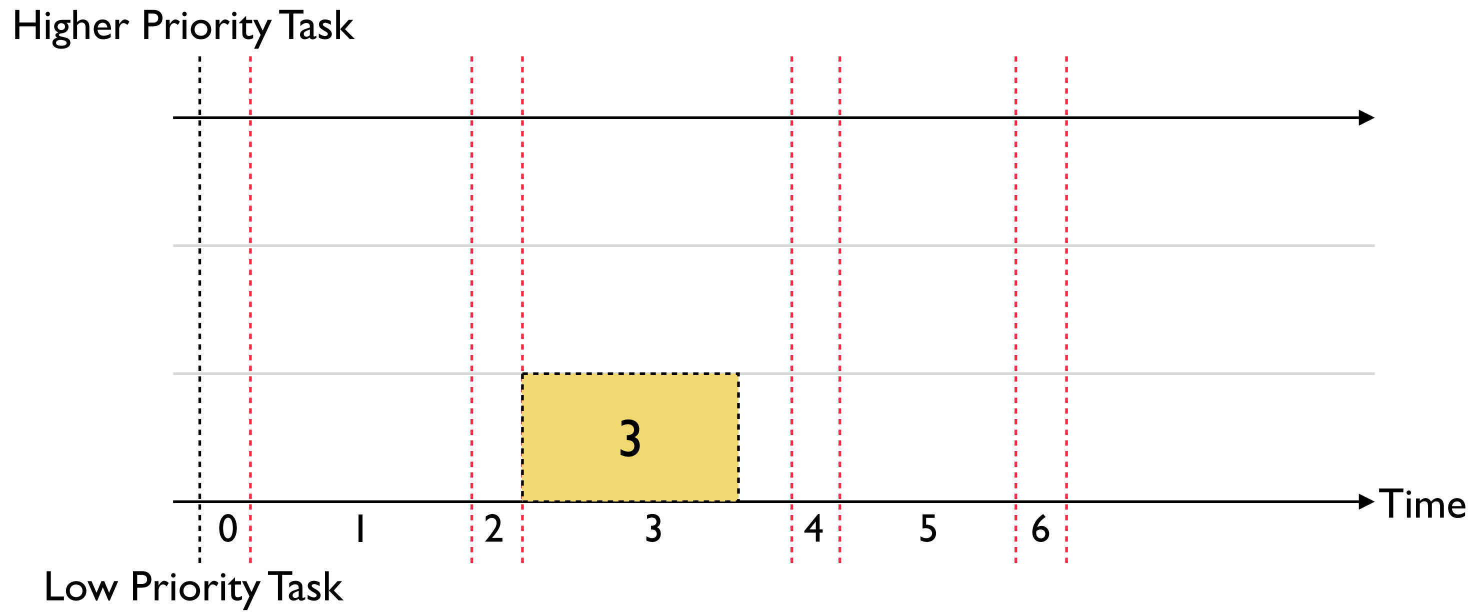


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

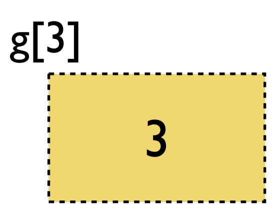
MonoSeq Sequentialization

Sequential Execution



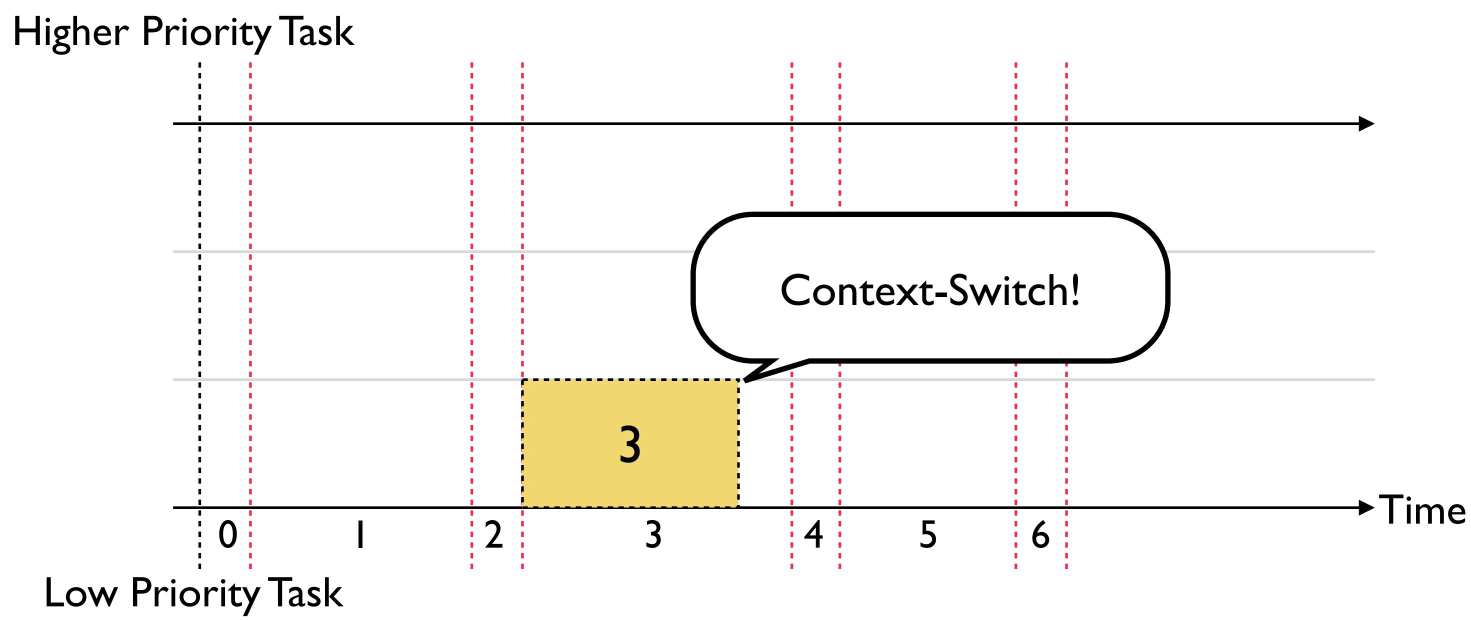
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



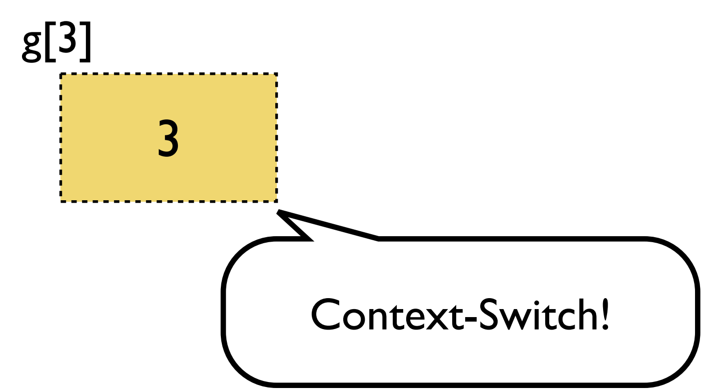
MonoSeq Sequentialization

Sequential Execution



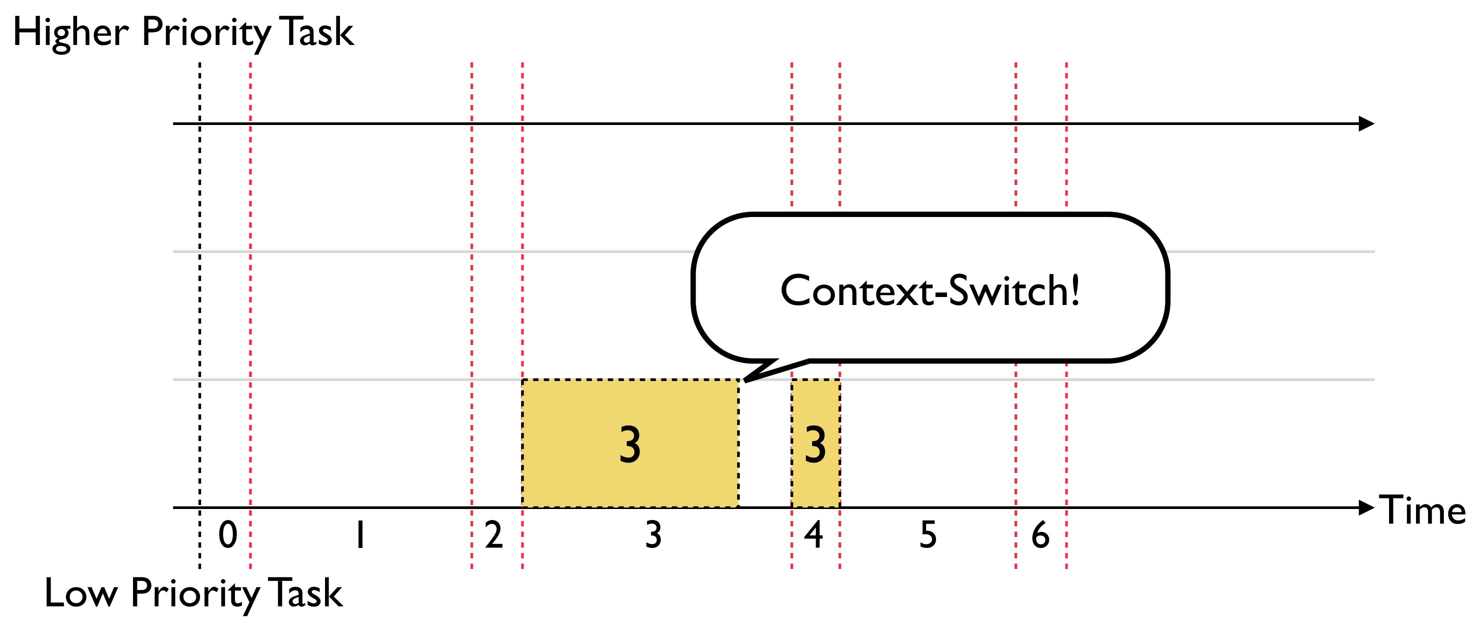
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



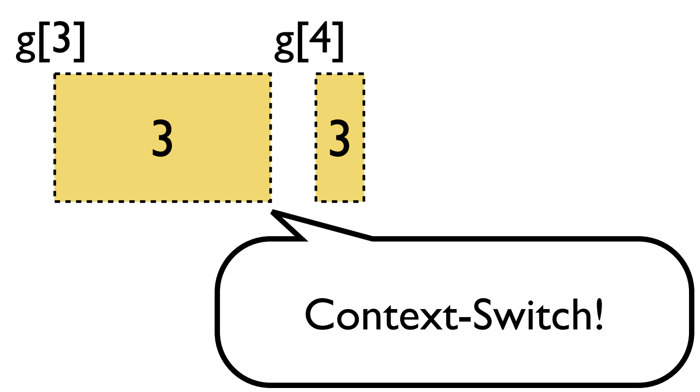
MonoSeq Sequentialization

Sequential Execution



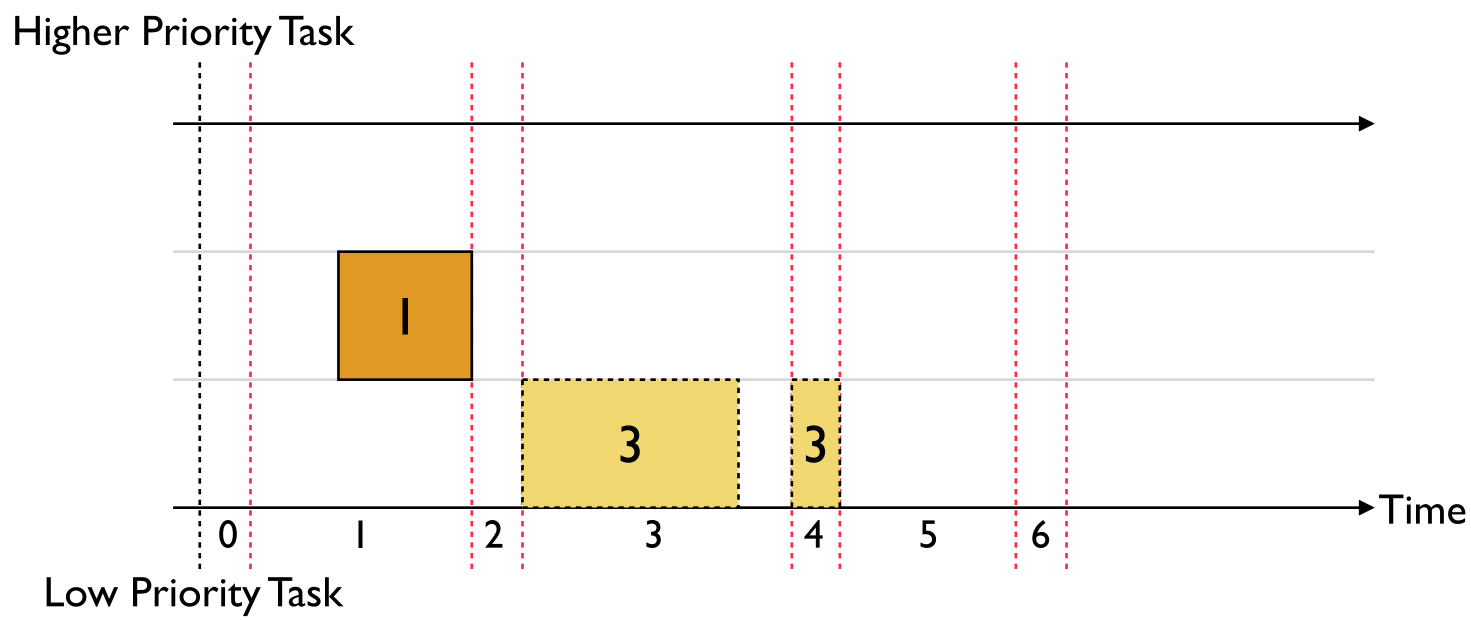
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



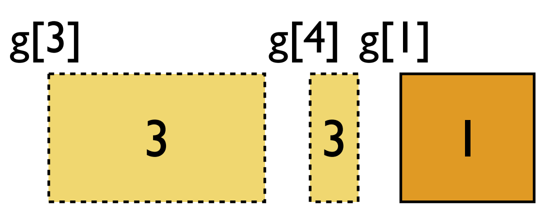
MonoSeq Sequentialization

Sequential Execution



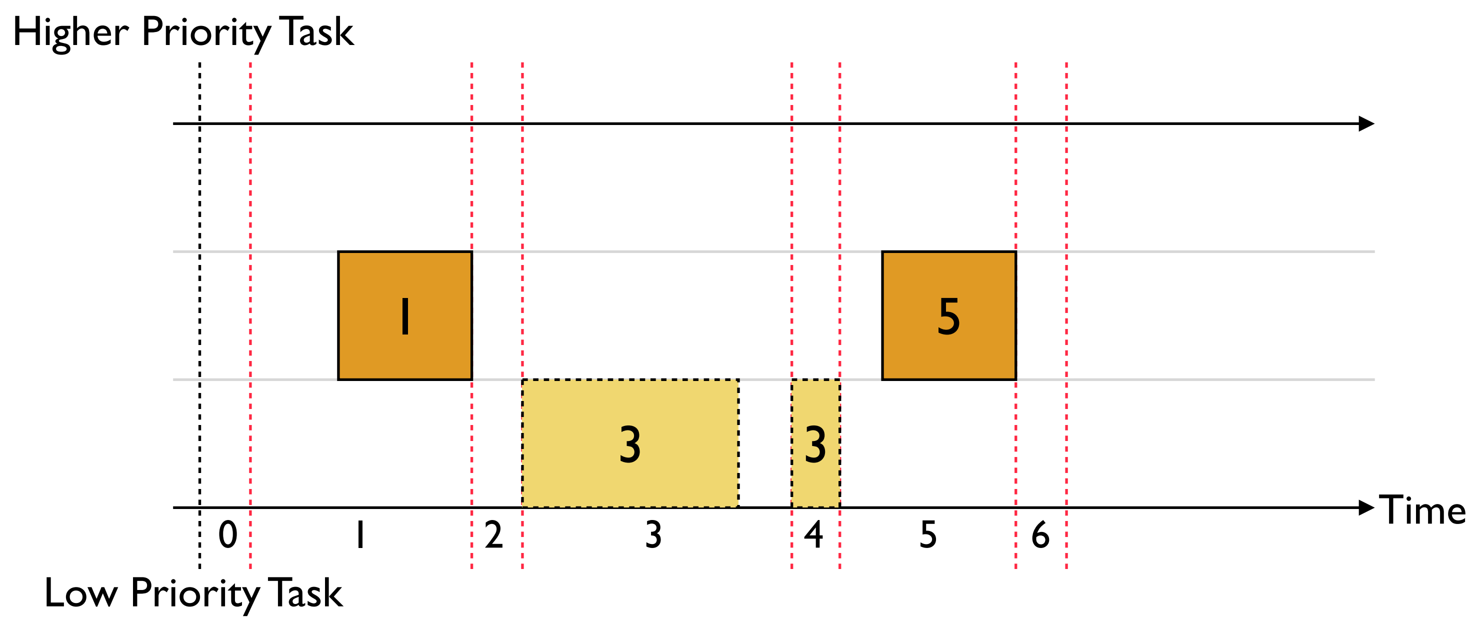
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



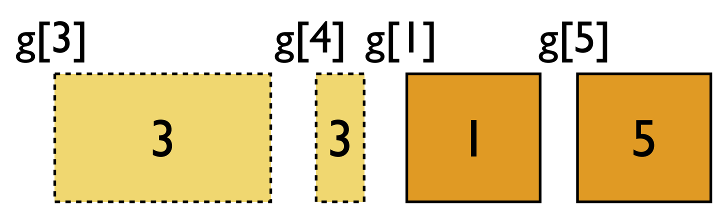
MonoSeq Sequentialization

Sequential Execution



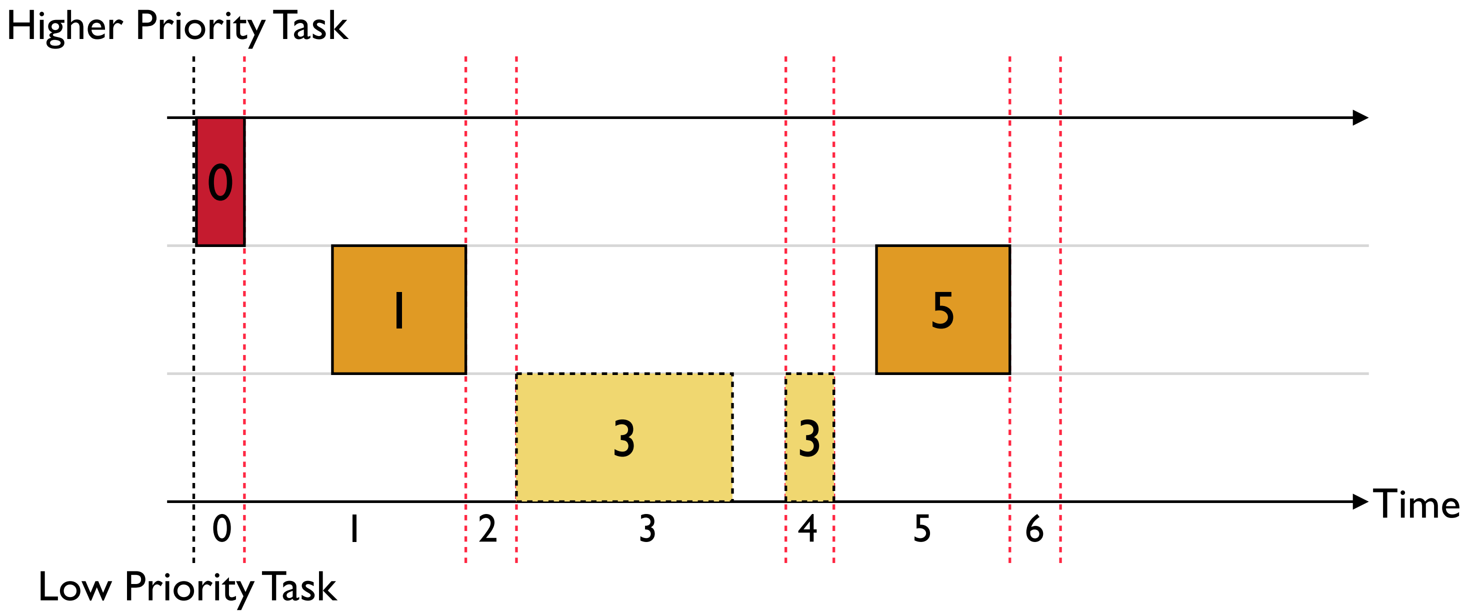
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



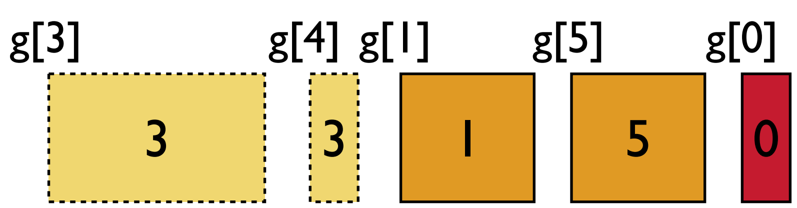
MONOSEQ Sequentialization

Sequential Execution



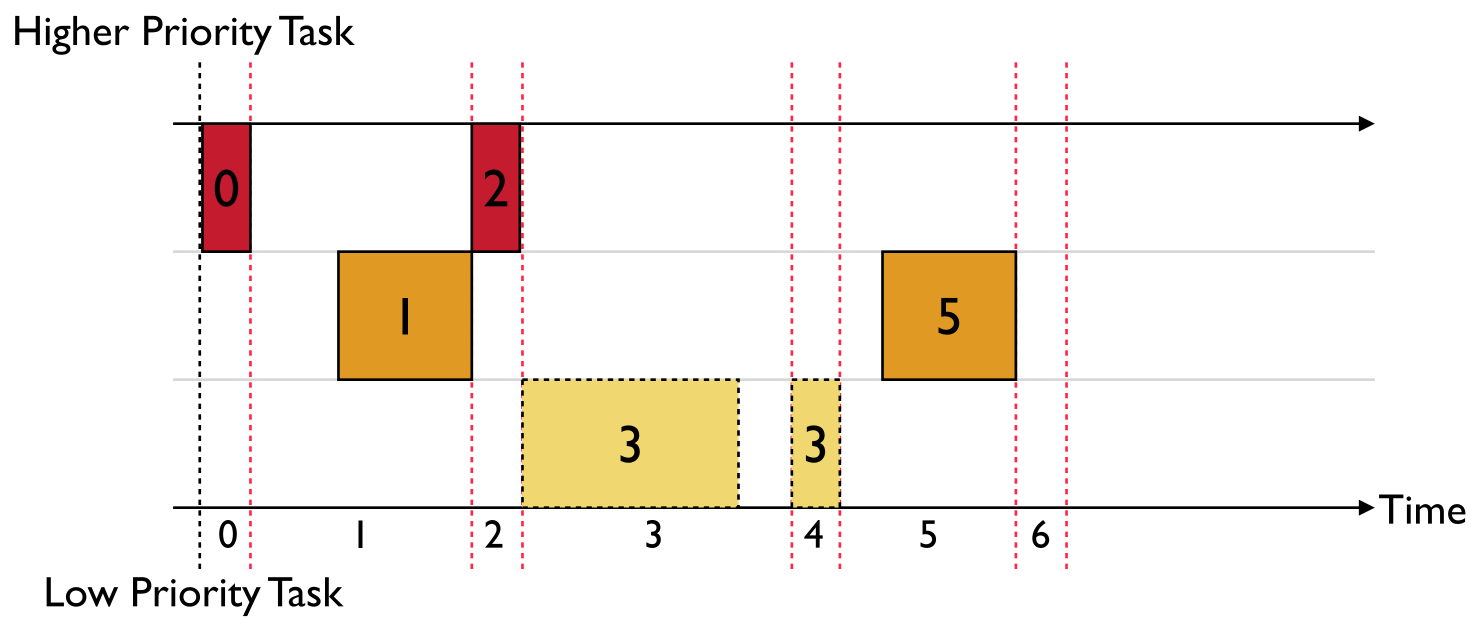
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



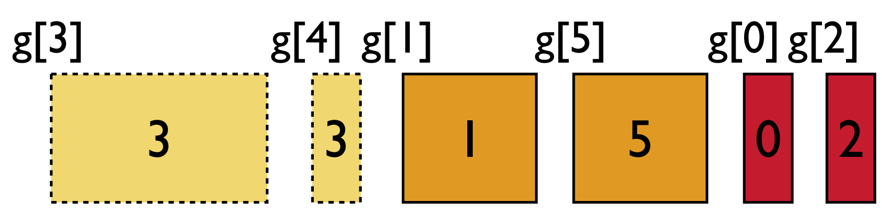
MONOSEQ Sequentialization

Sequential Execution



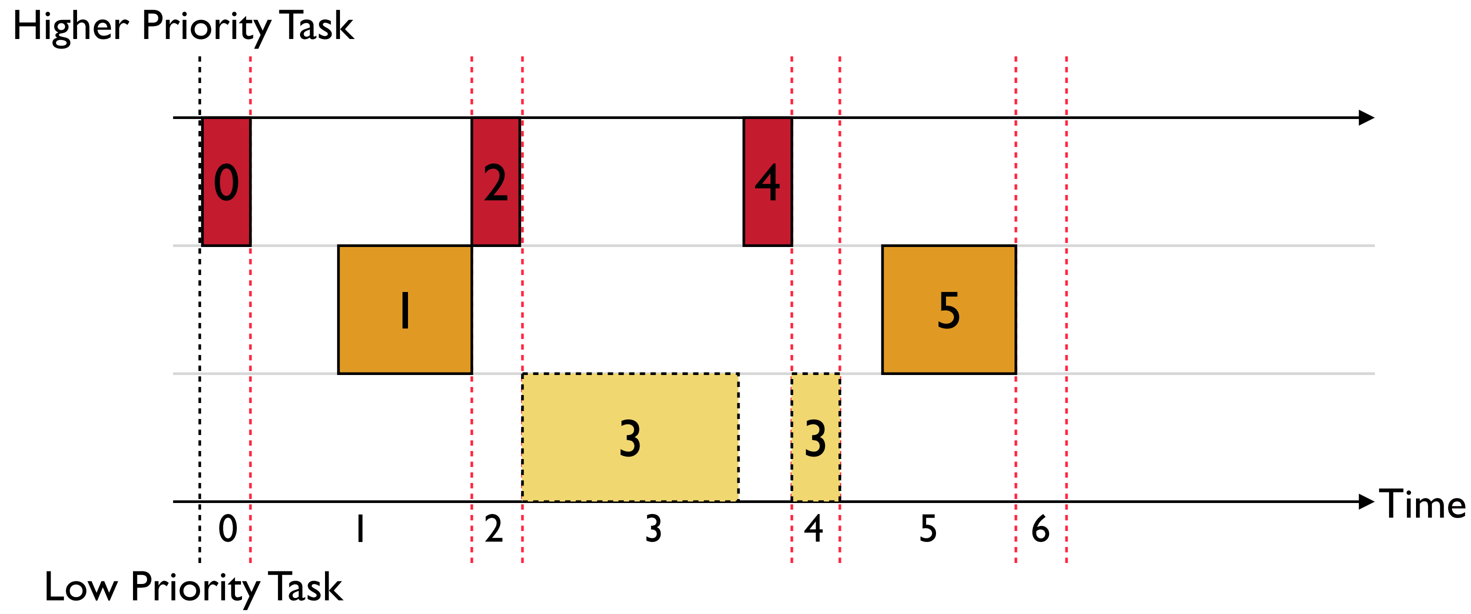
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



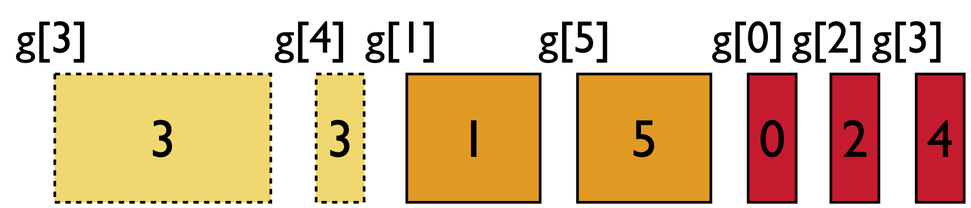
MONOSEQ Sequentialization

Sequential Execution



Guess non-deterministic initial value of each global in each round.

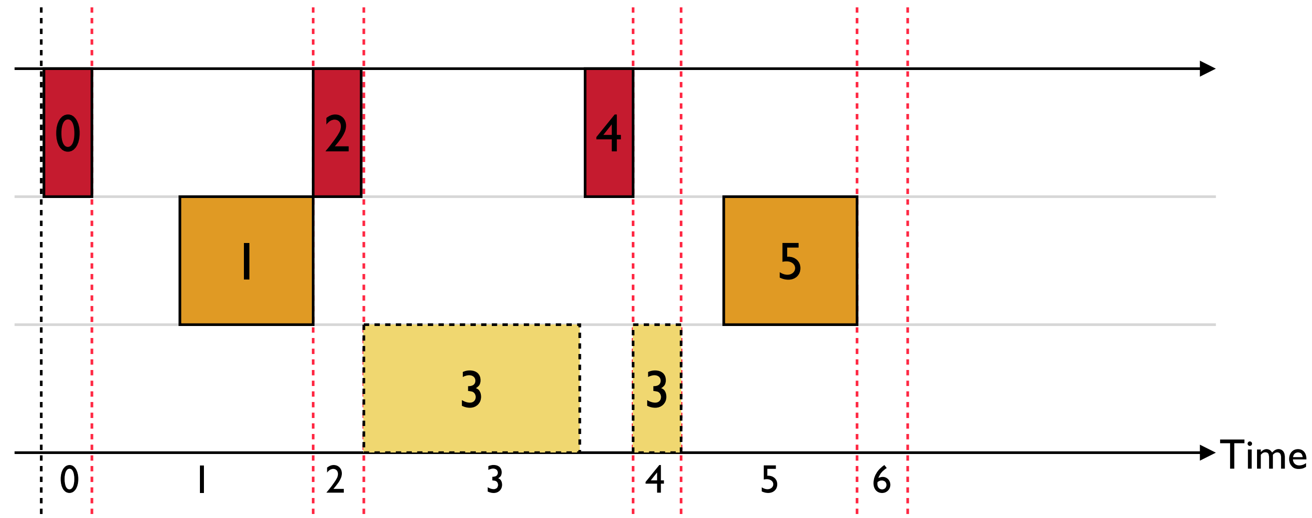
Execute Task Body, from lower priority task to higher priority task



MonoSeq Sequentialization

Sequential Execution

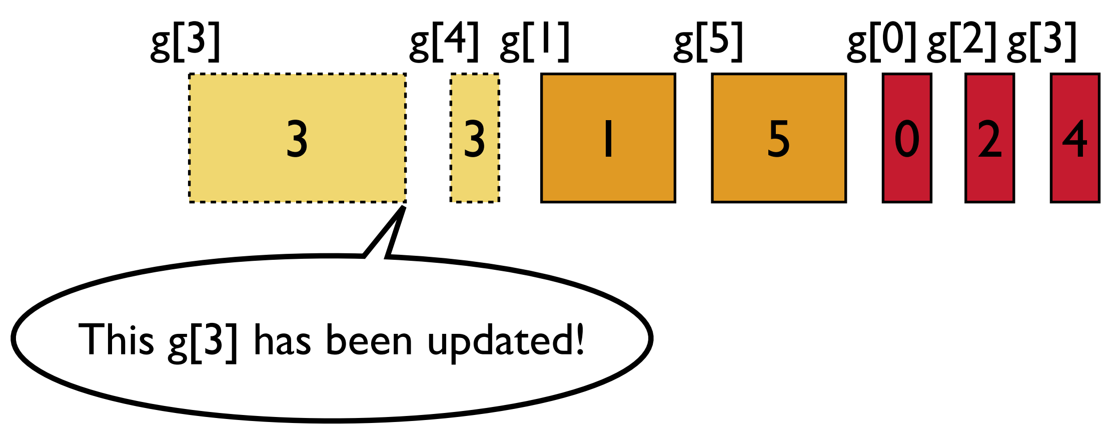
Higher Priority Task



Low Priority Task

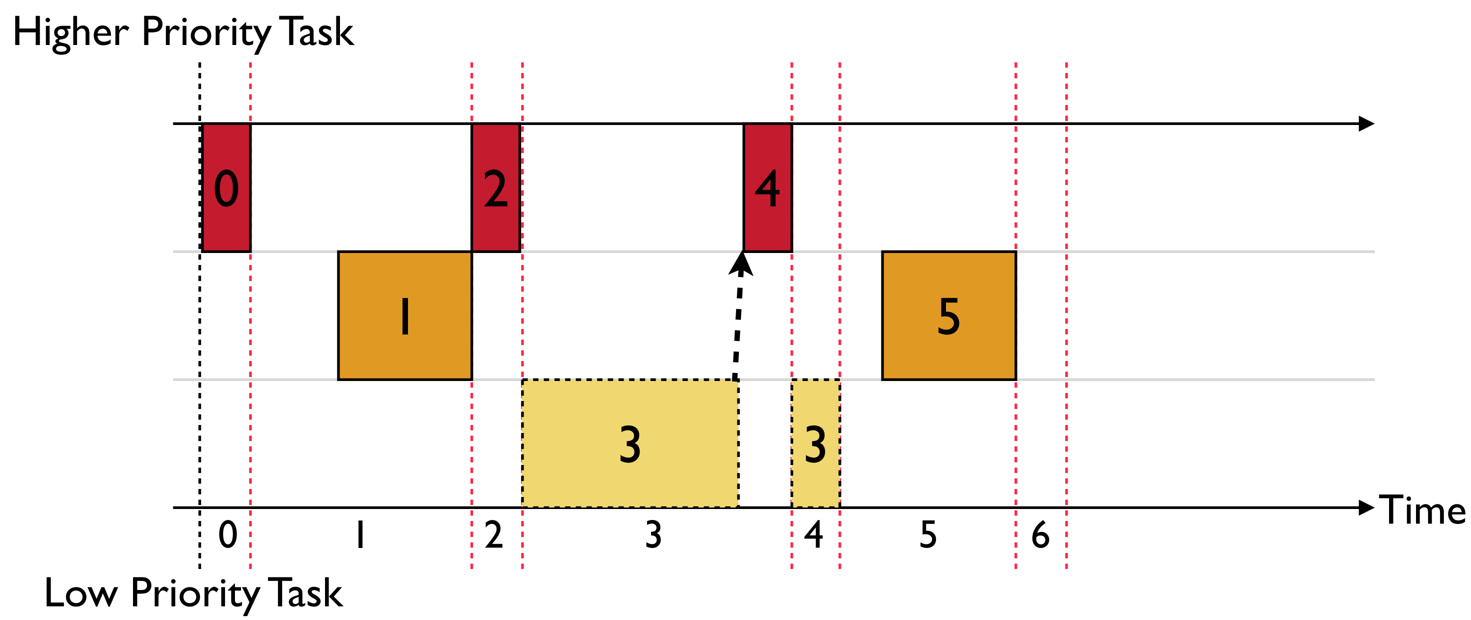
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



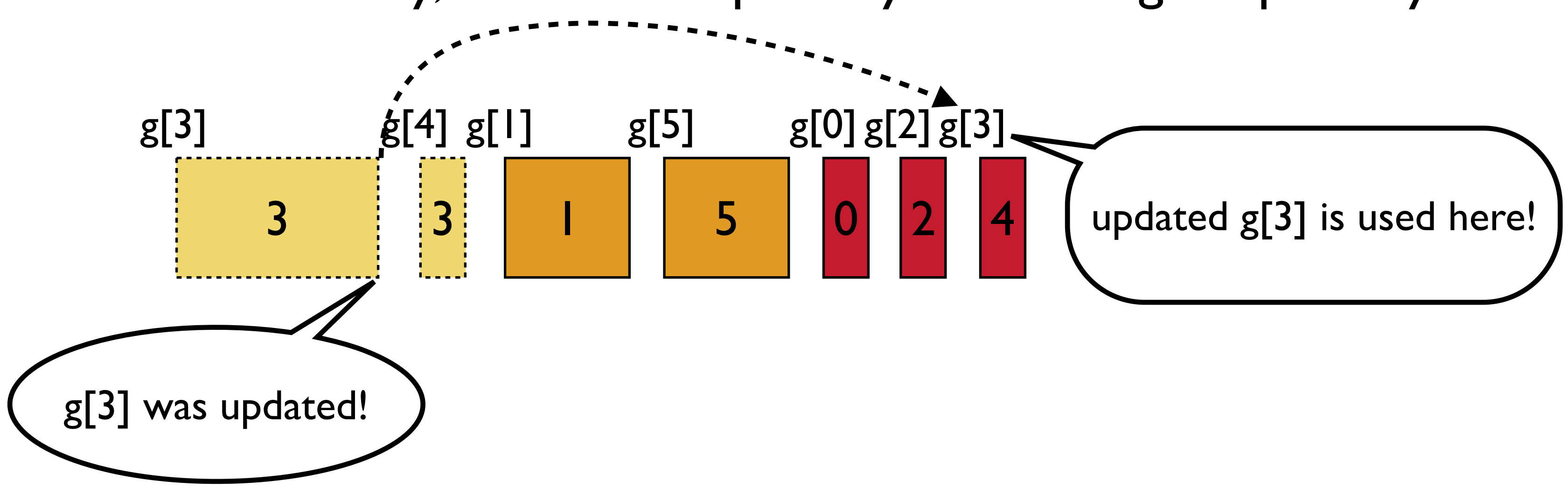
MONOSEQ Sequentialization

Sequential Execution



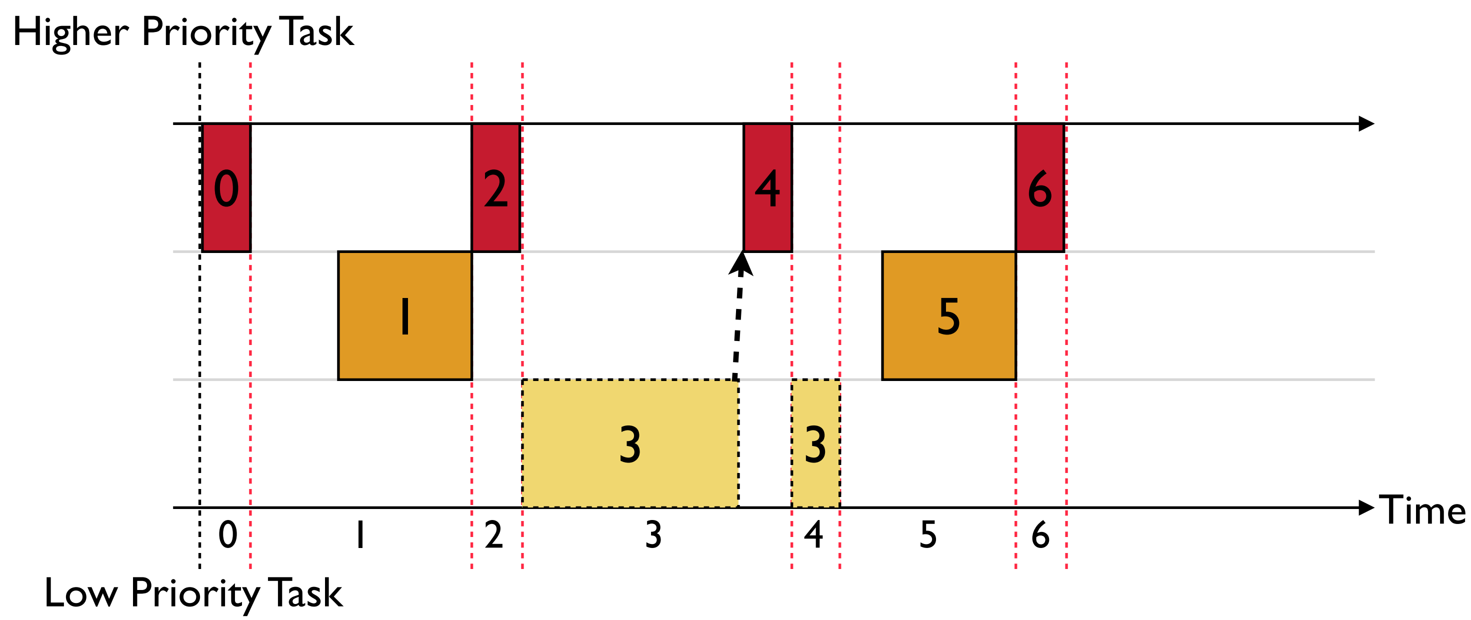
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



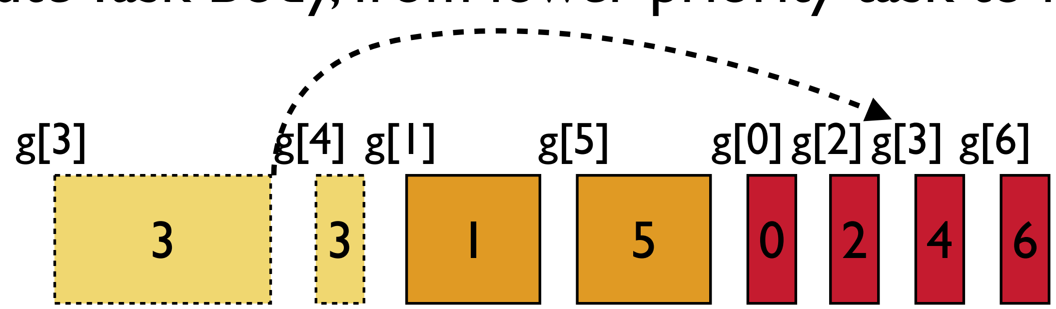
MonoSeq Sequentialization

Sequential Execution

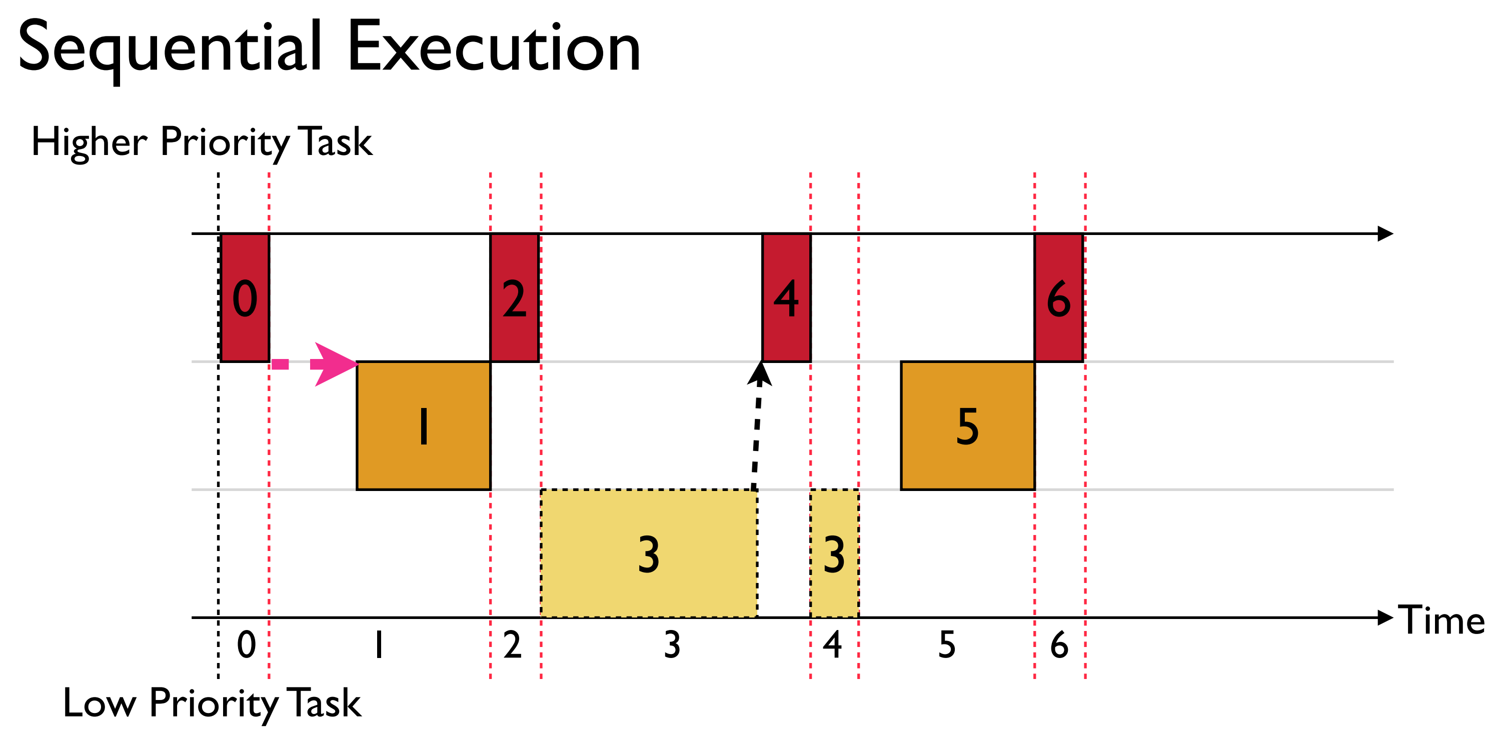


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task



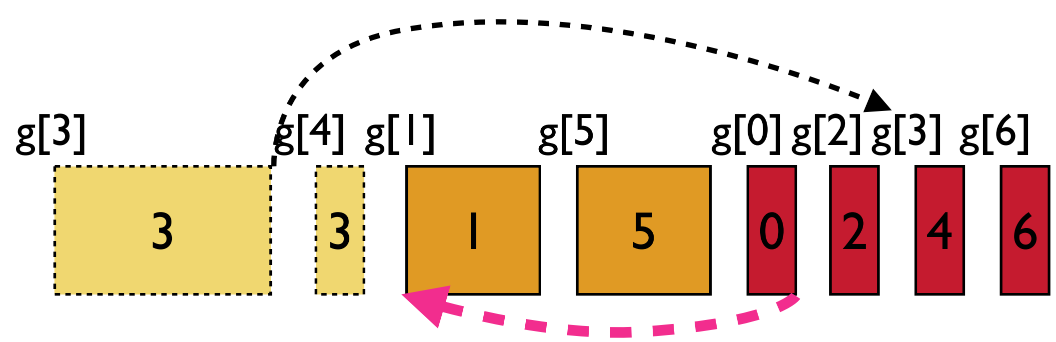
MONOSEQ Sequentialization



Guess non-deterministic initial value of each global in each round.

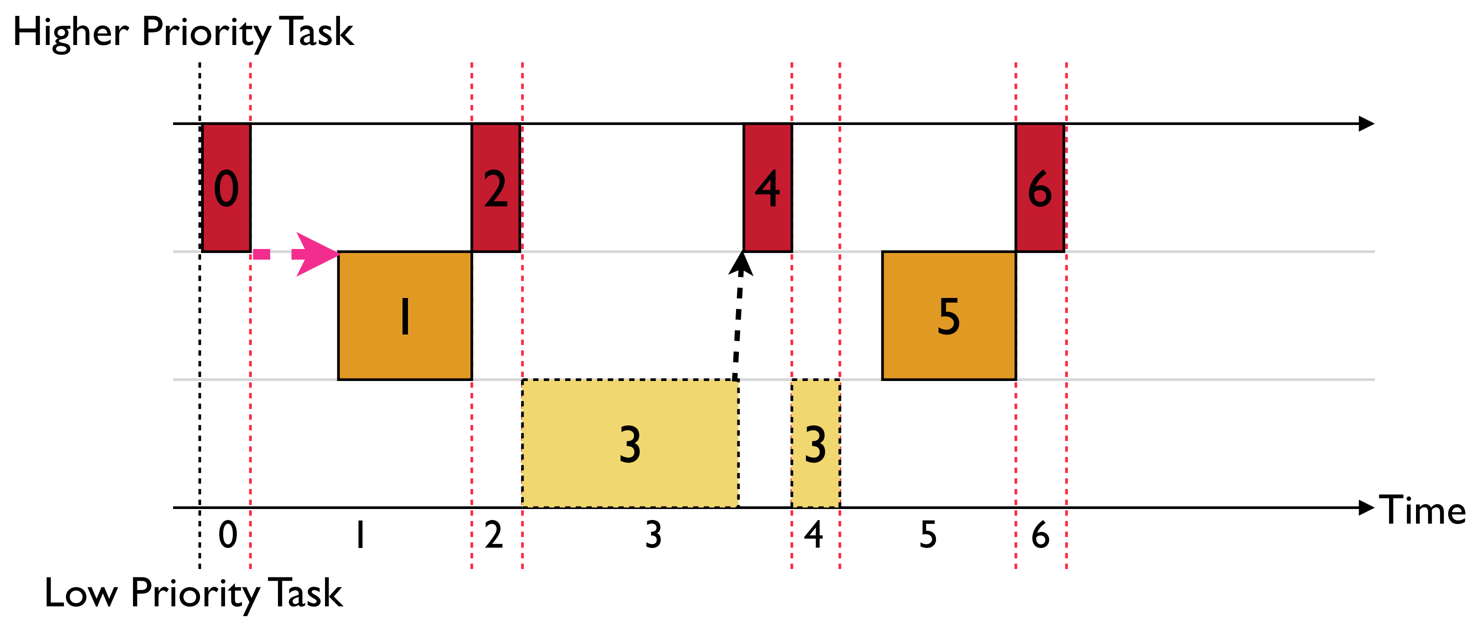
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MonoSeq Sequentialization

Sequential Execution



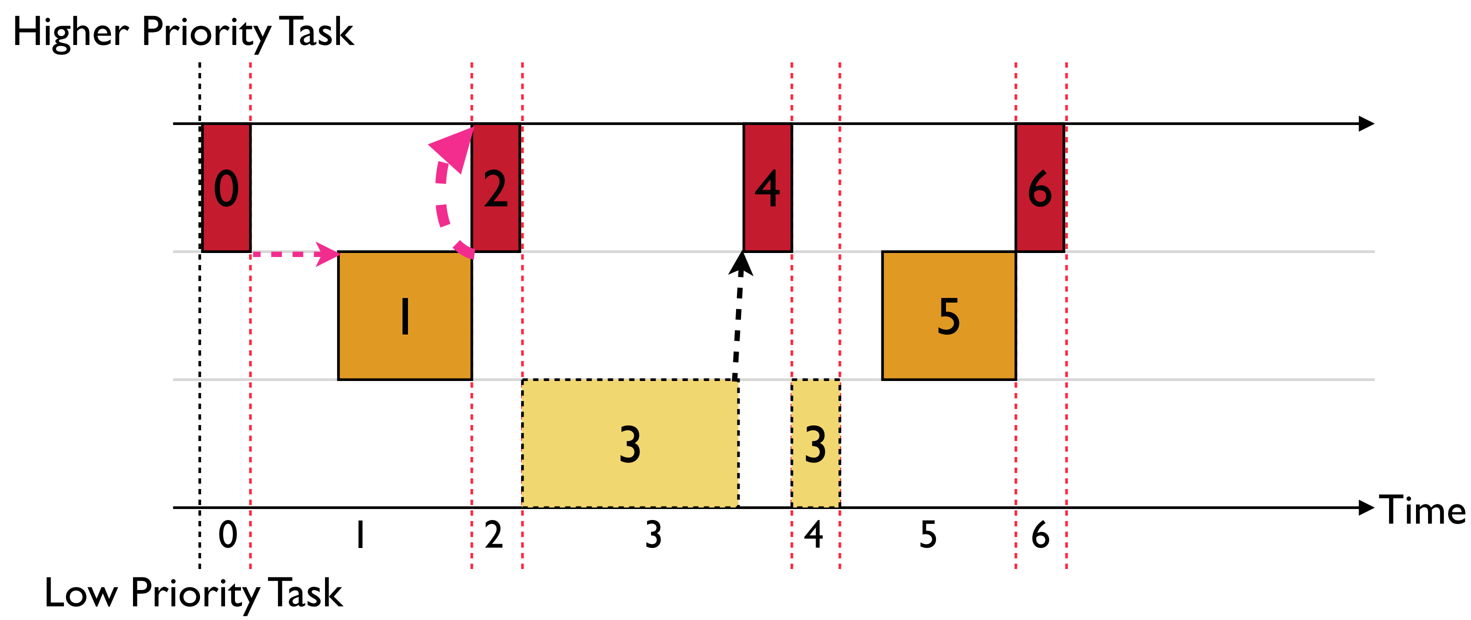
Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution

MONOSEQ Sequentialization

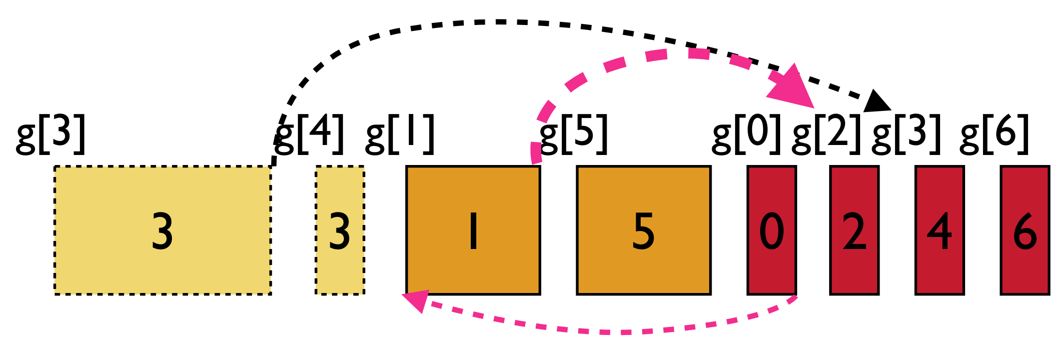
Sequential Execution



Guess non-deterministic initial value of each global in each round.

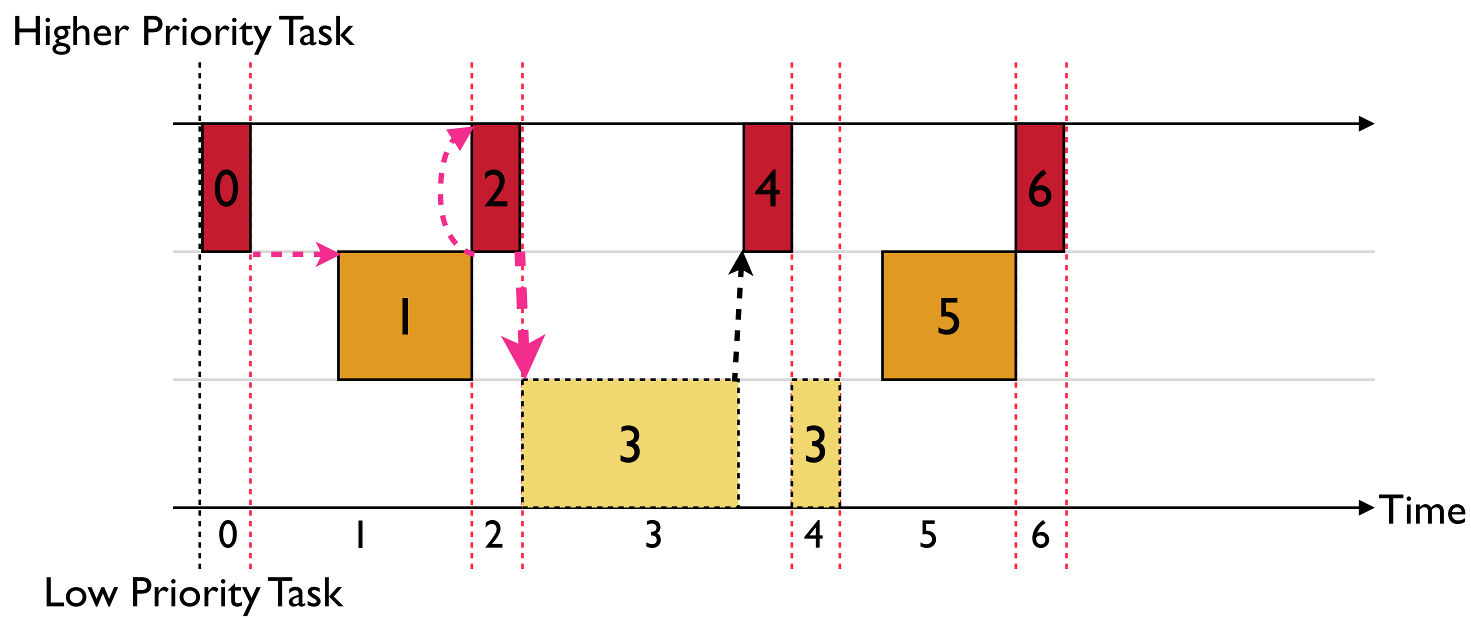
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MONOSEQ Sequentialization

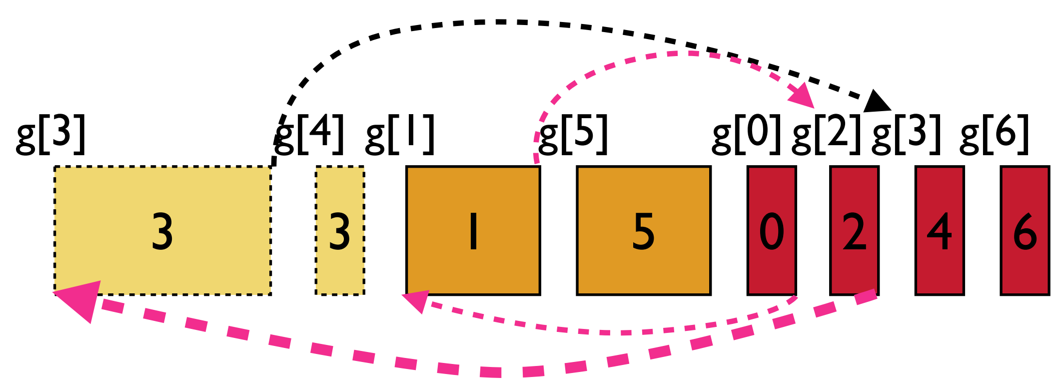
Sequential Execution



Guess non-deterministic initial value of each global in each round.

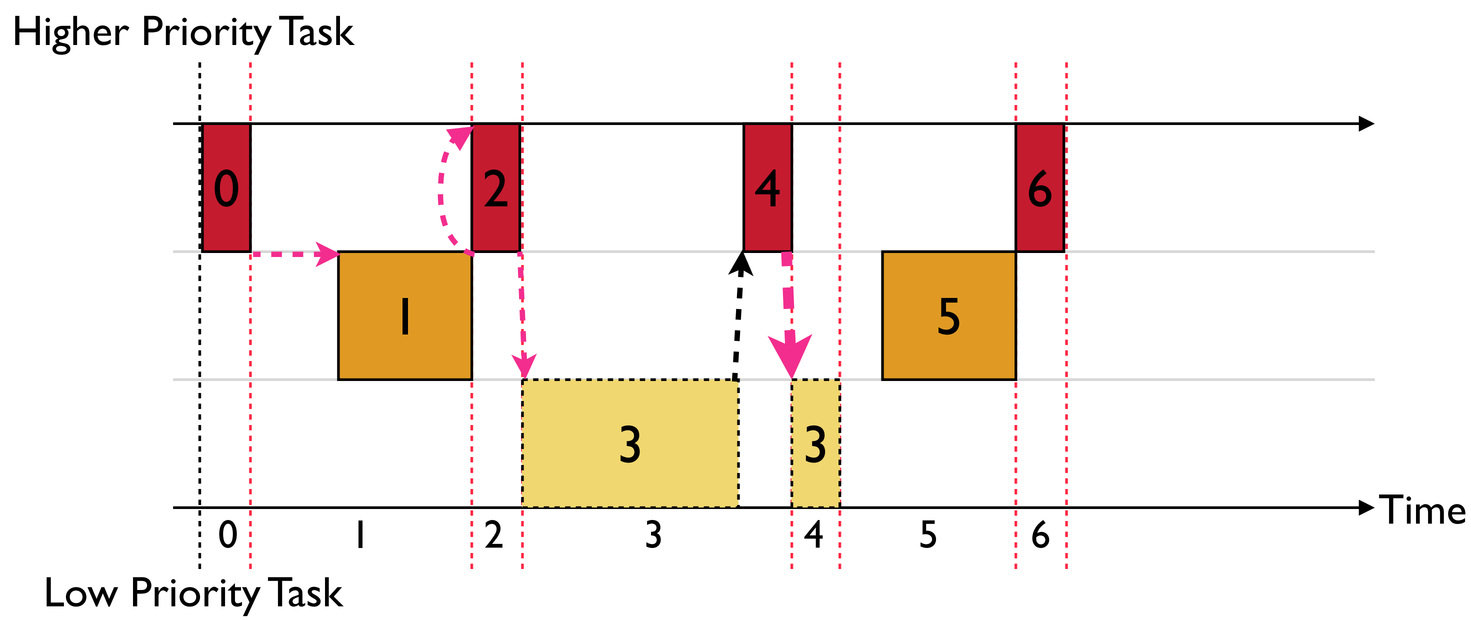
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MonoSeq Sequentialization

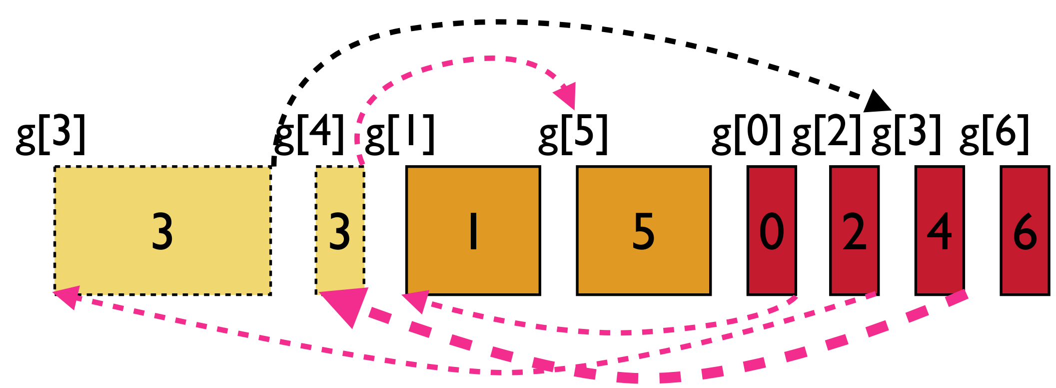
Sequential Execution



Guess non-deterministic initial value of each global in each round.

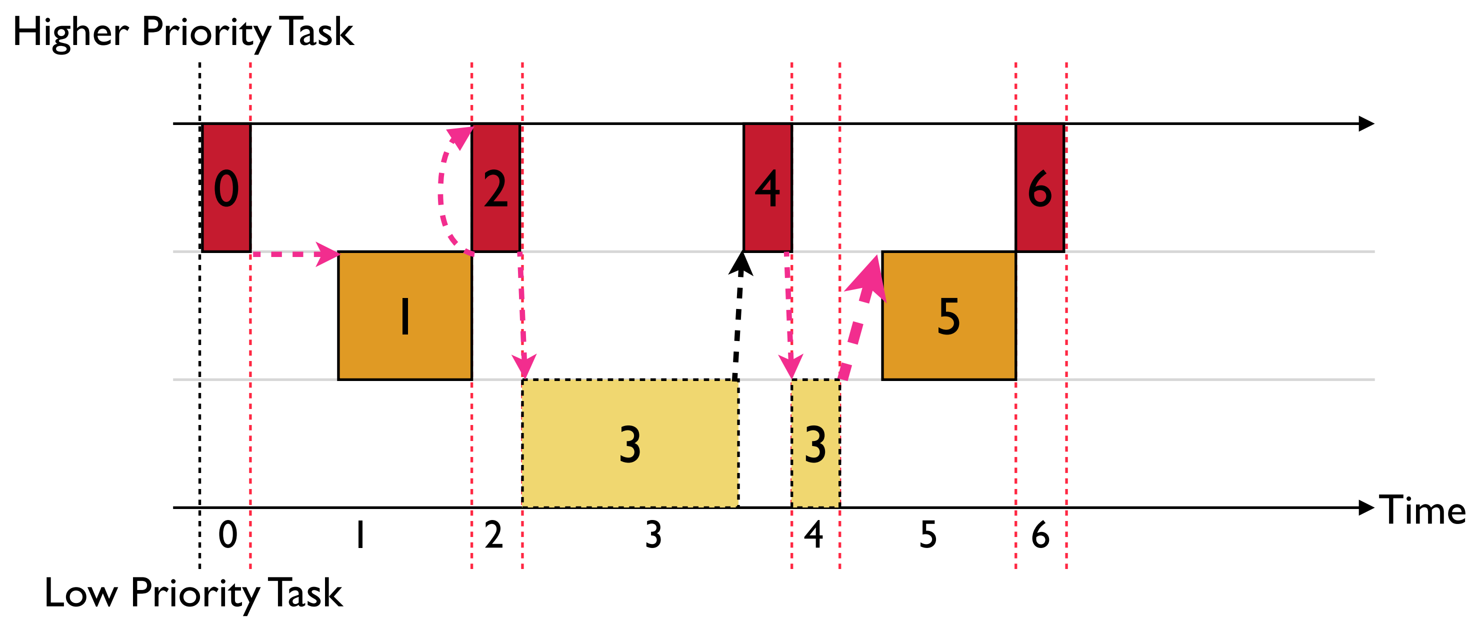
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MonoSeq Sequentialization

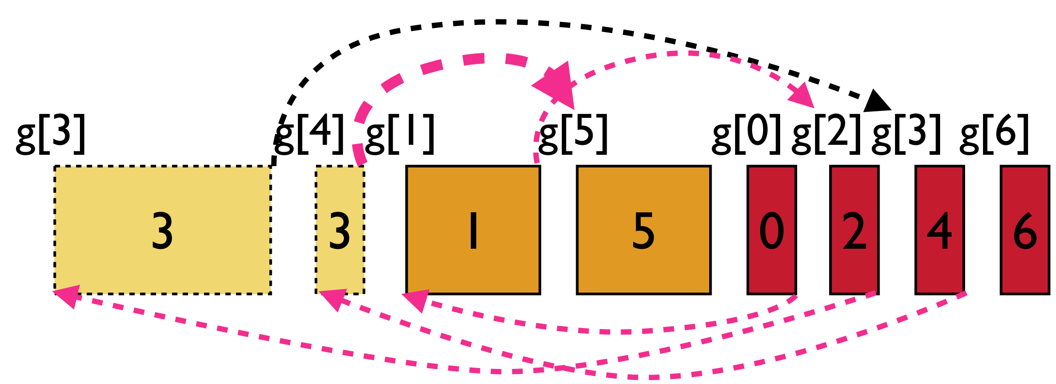
Sequential Execution



Guess non-deterministic initial value of each global in each round.

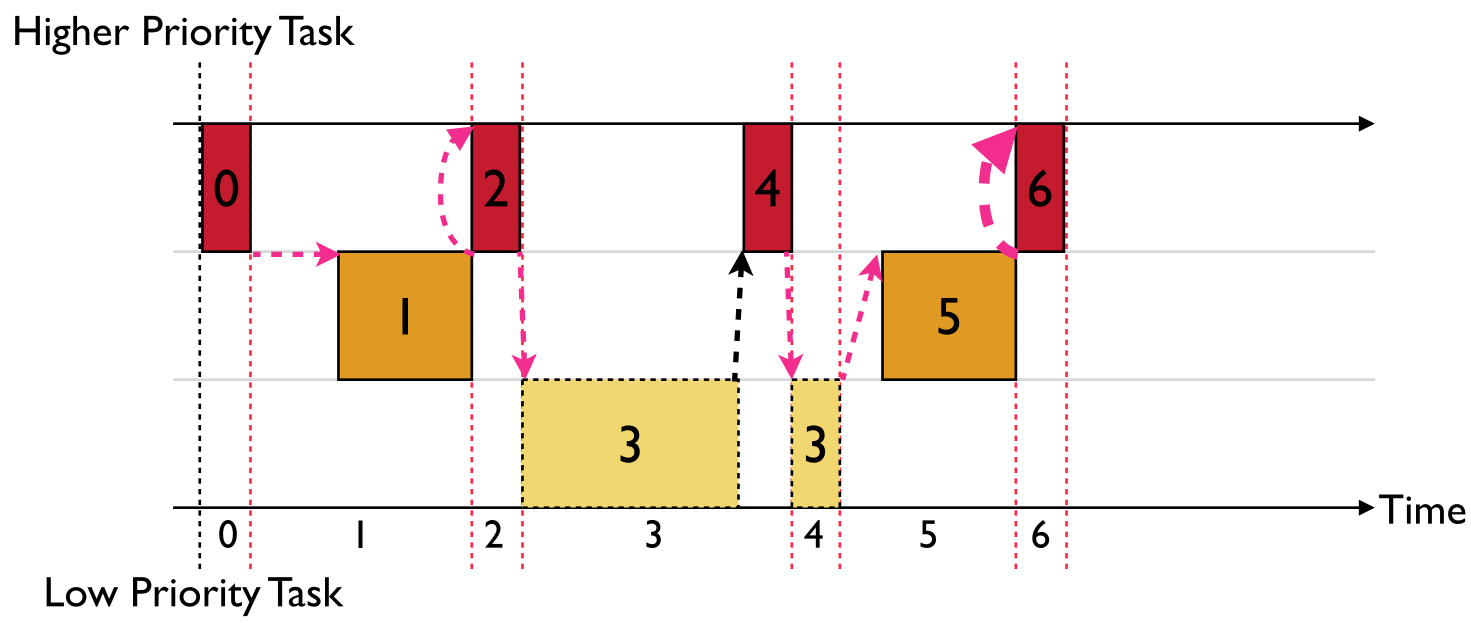
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MonoSeq Sequentialization

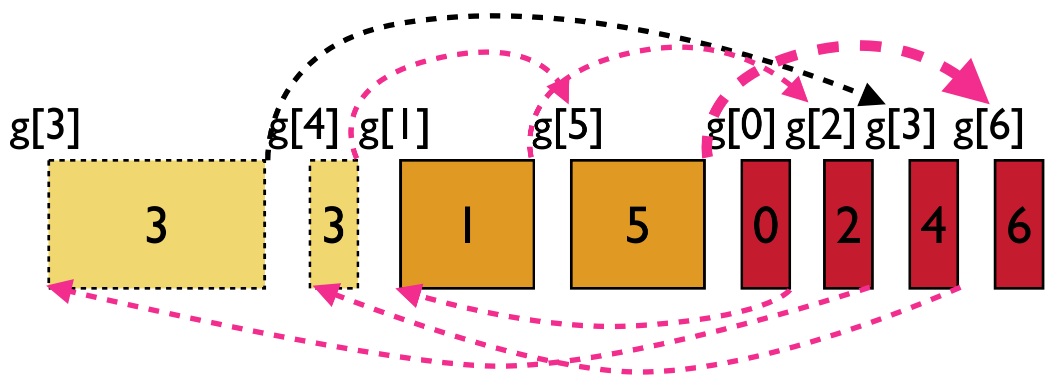
Sequential Execution



Guess non-deterministic initial value of each global in each round.

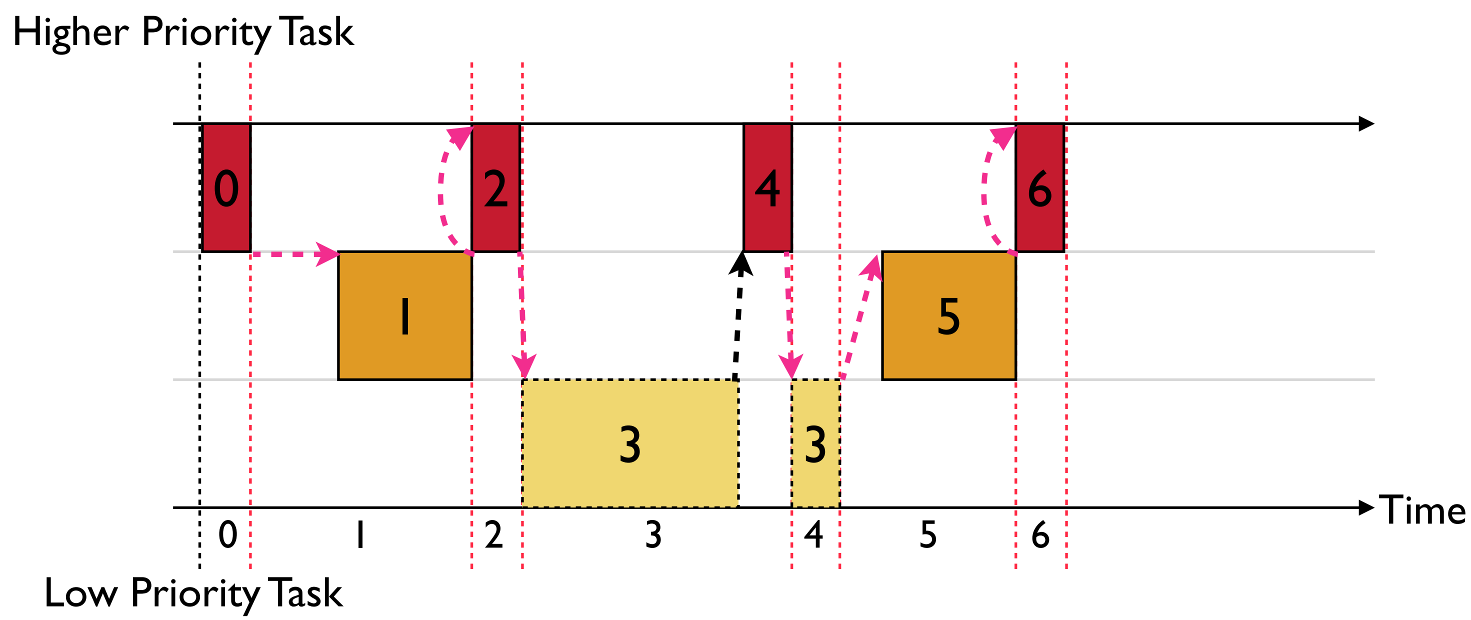
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MonoSeq Sequentialization

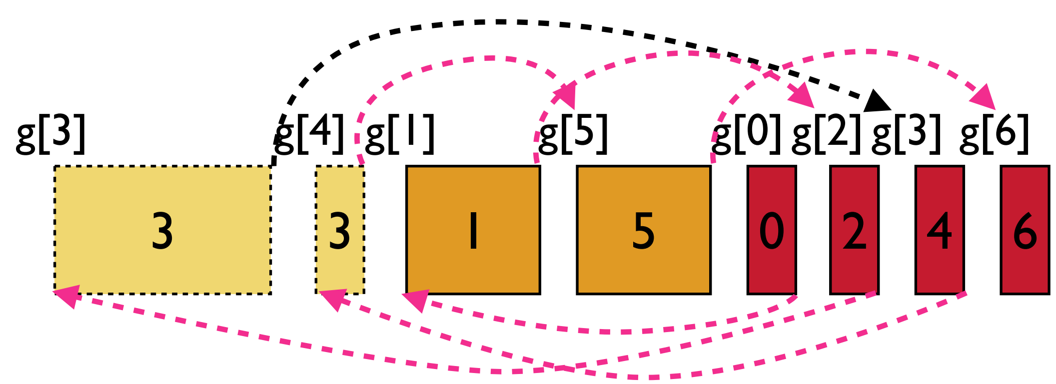
Sequential Execution



Guess non-deterministic initial value of each global in each round.

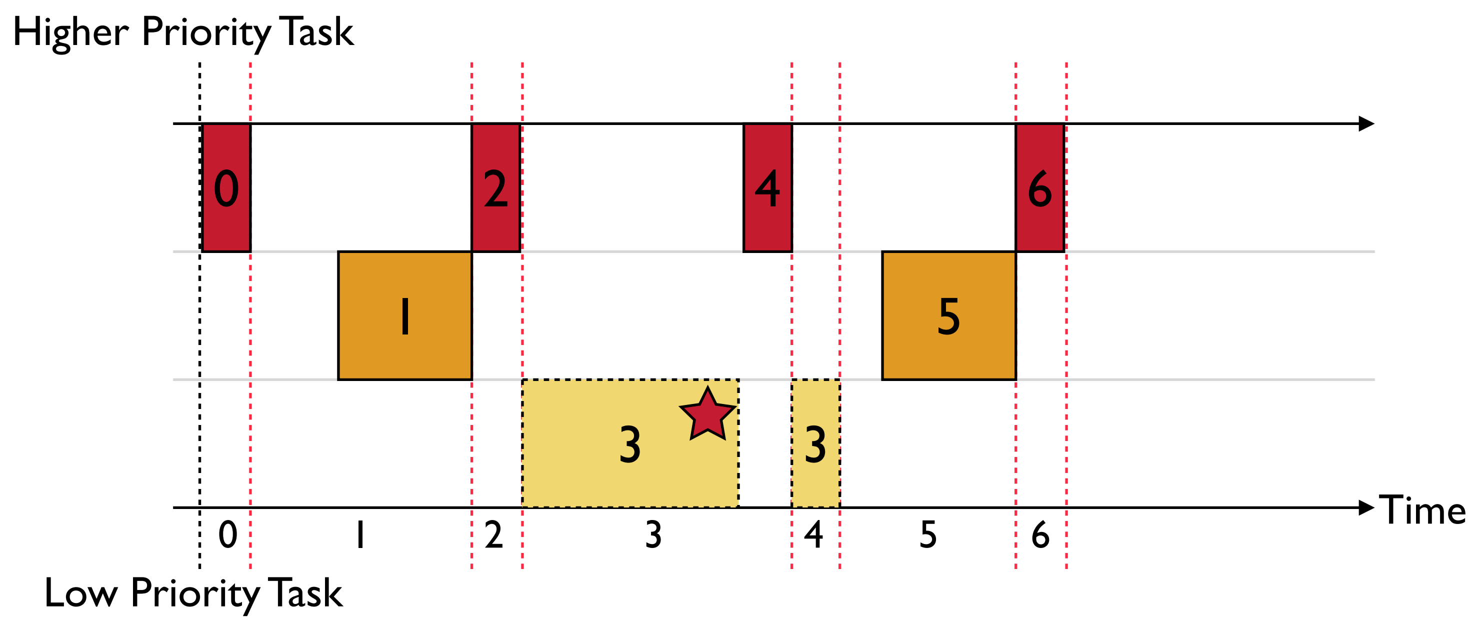
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution



MONOSEQ Sequentialization

Sequential Execution



Guess non-deterministic initial value of each global in each round.

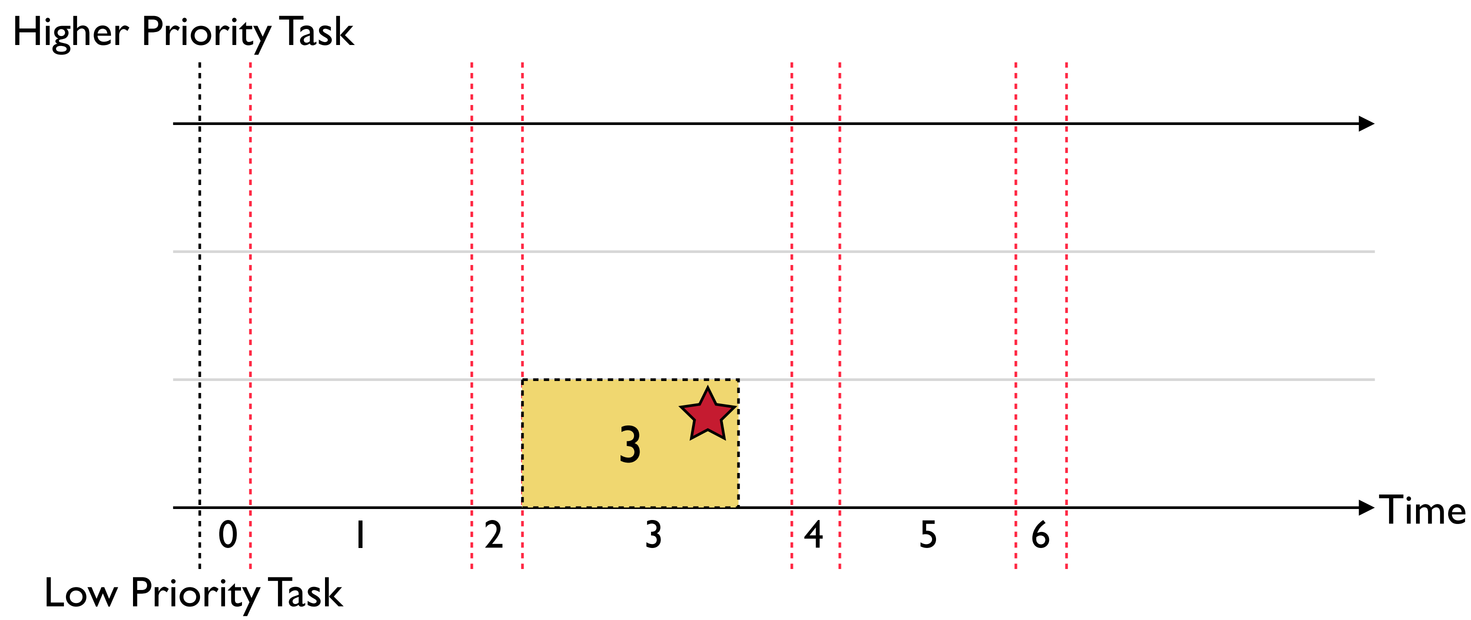
Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to respect the order of execution

Save assertions and check them at the end of execution.

MonoSeq Sequentialization

Sequential Execution

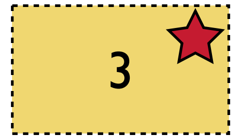


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

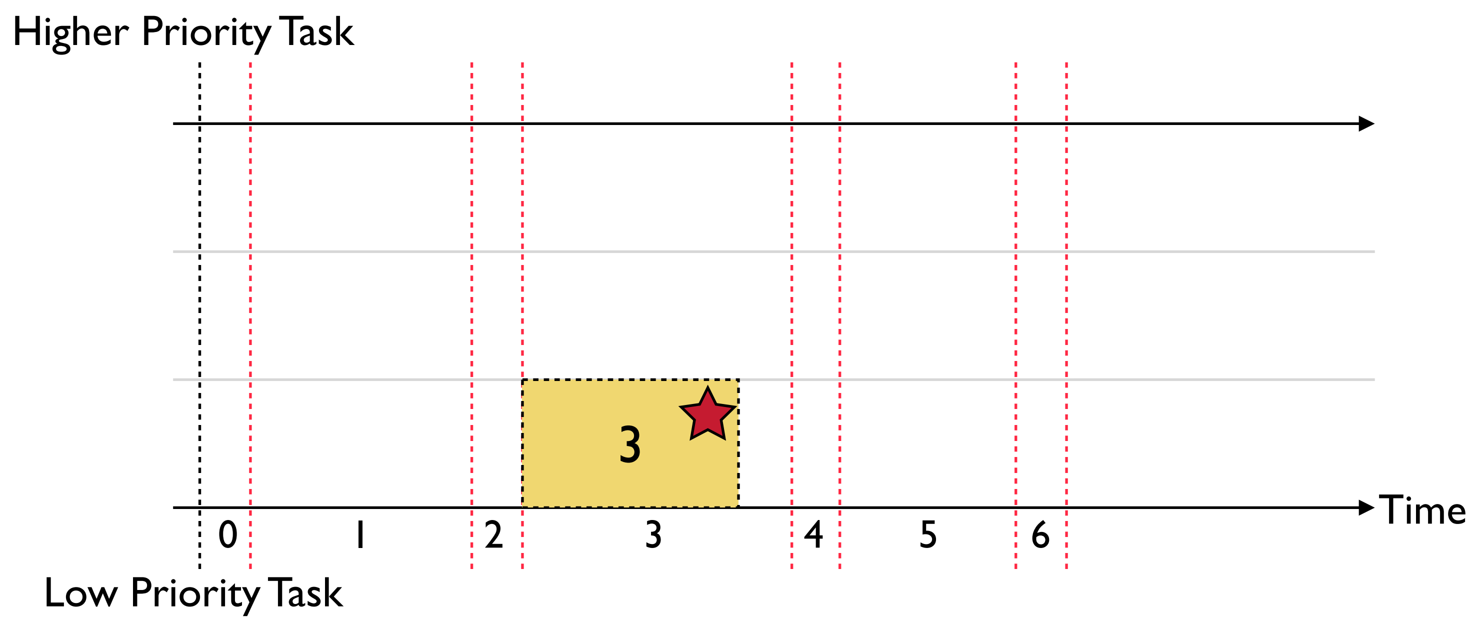
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

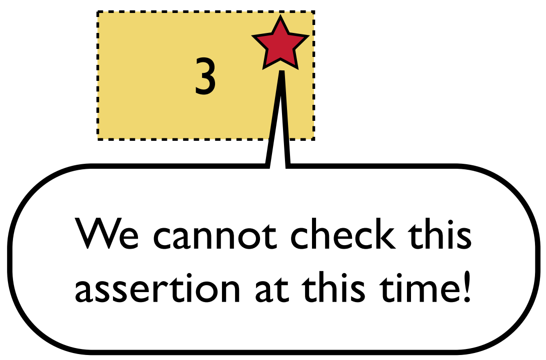


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

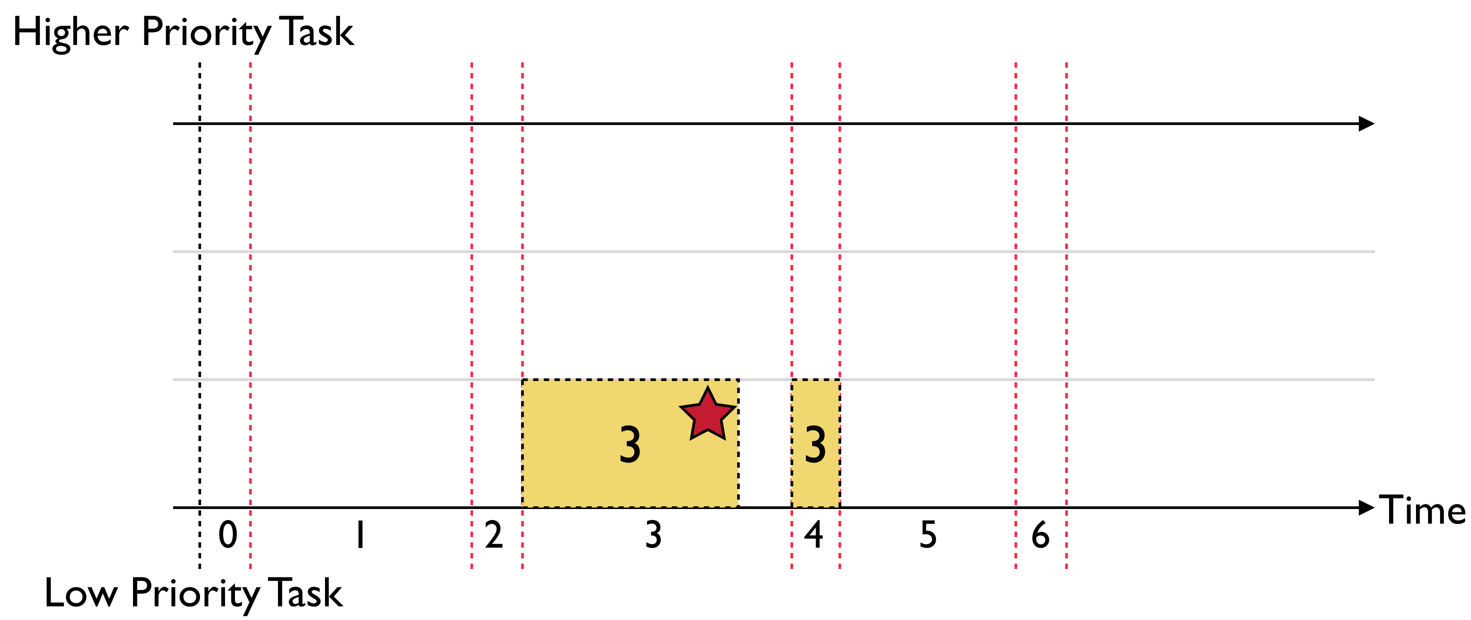
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

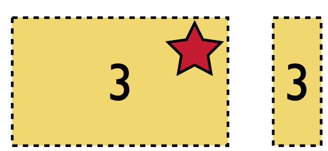


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

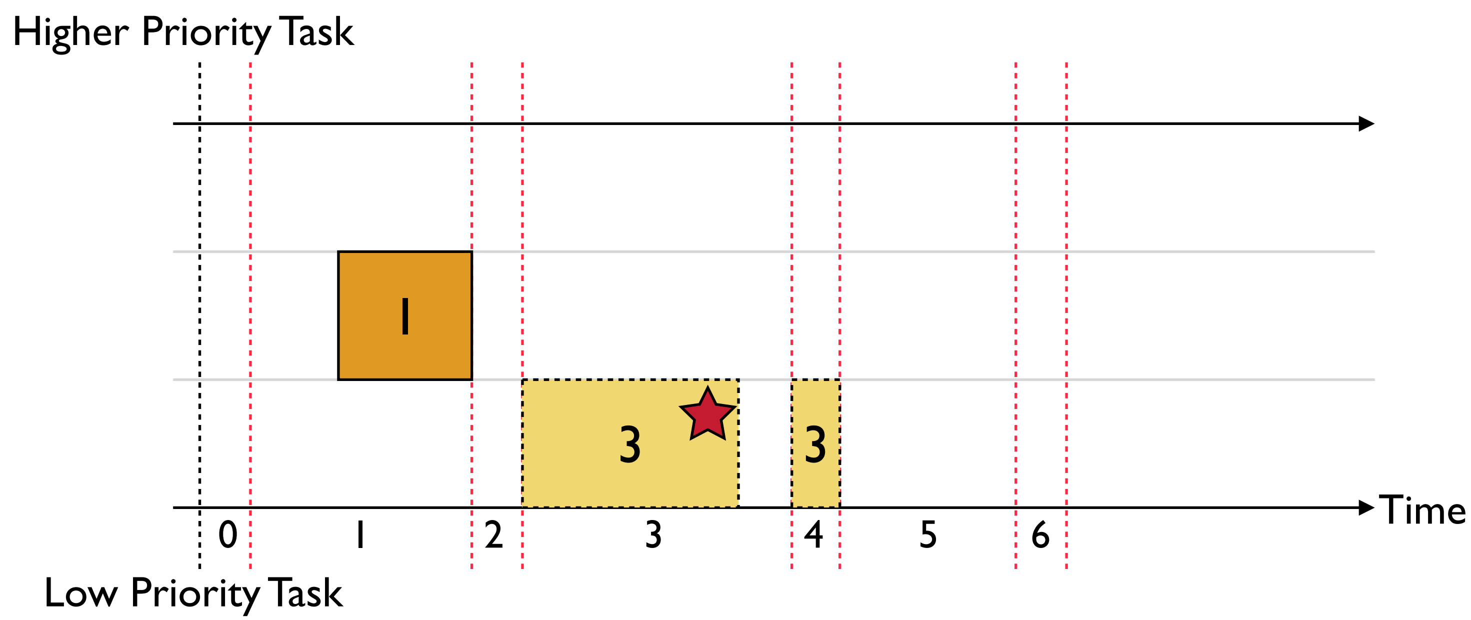
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MONOSEQ Sequentialization

Sequential Execution

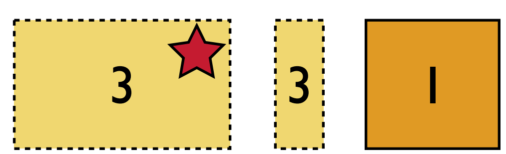


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

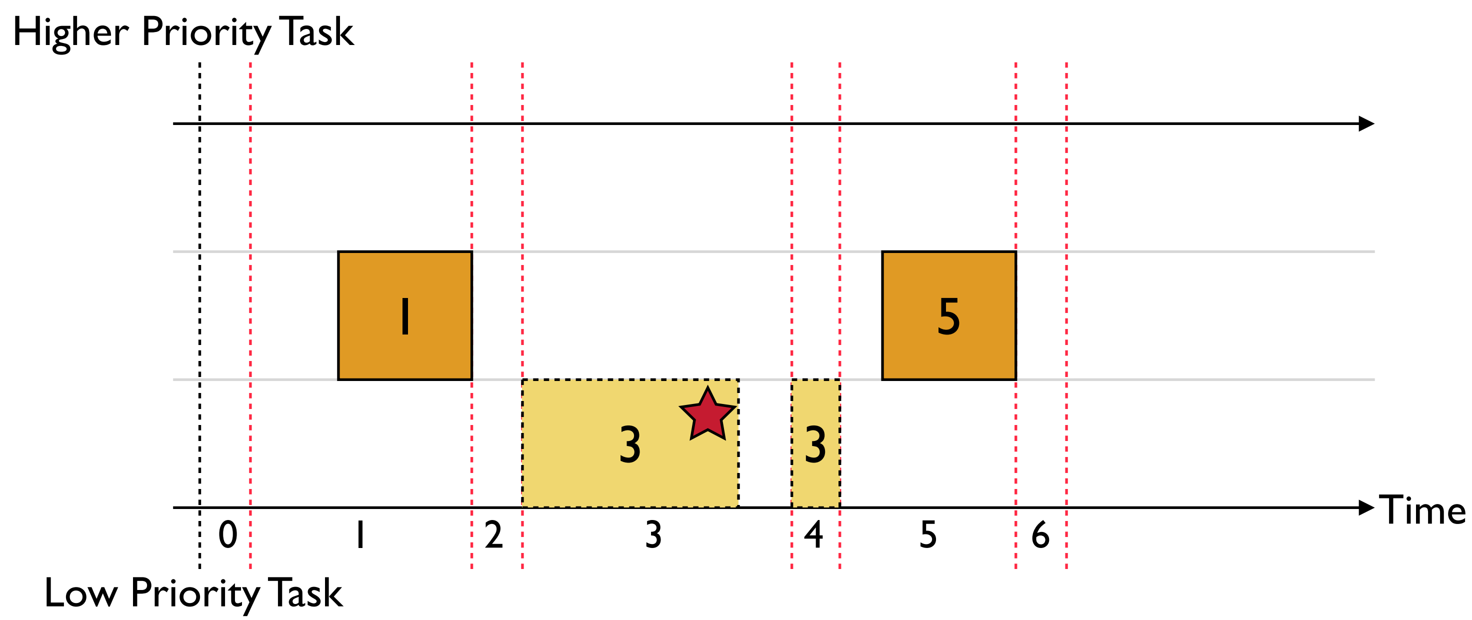
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

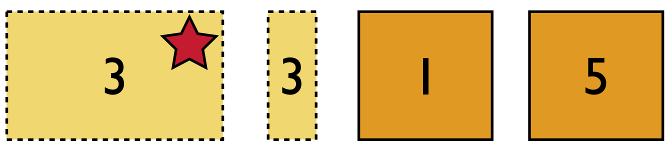


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

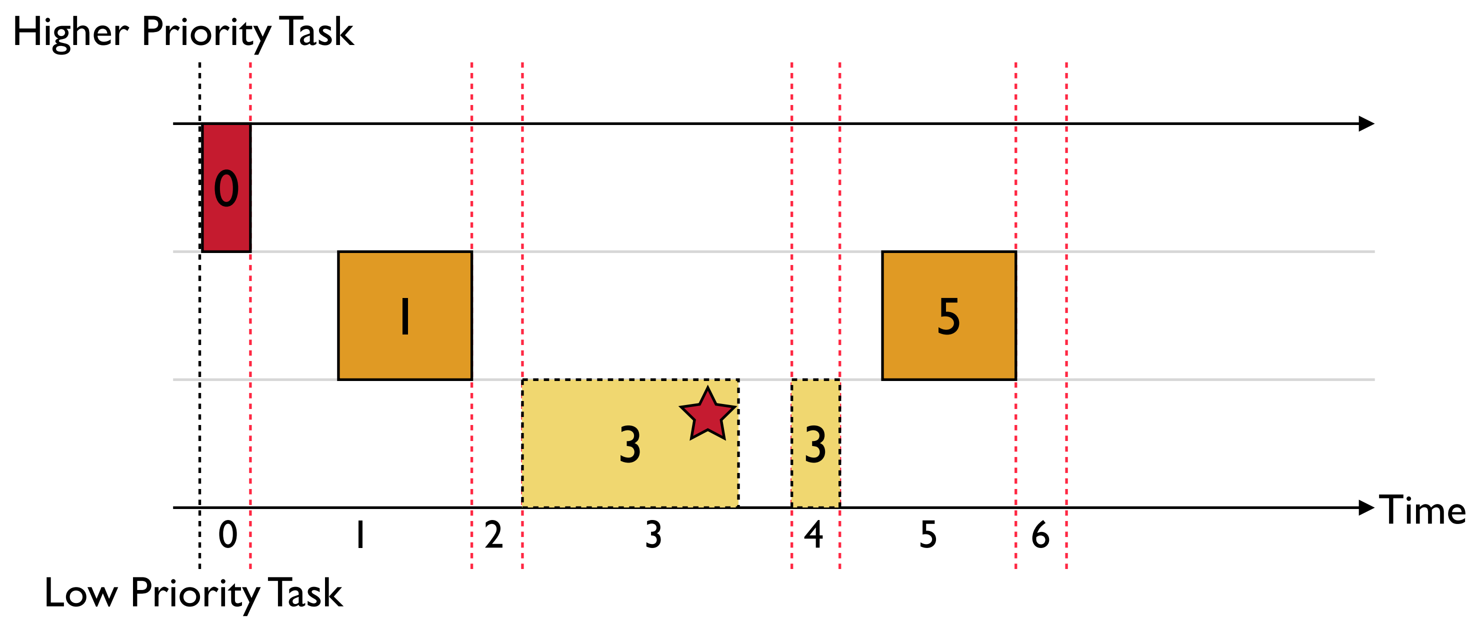
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

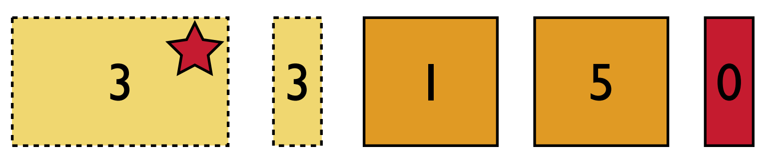


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

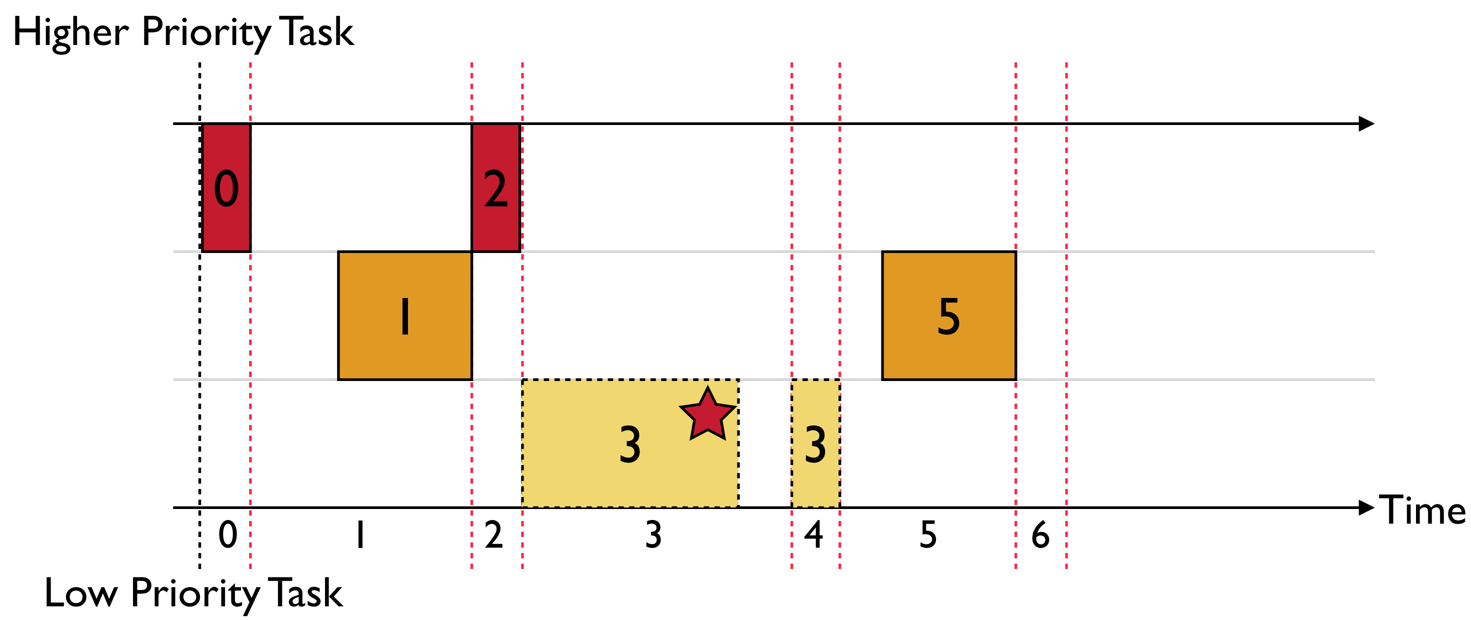
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MONOSEQ Sequentialization

Sequential Execution

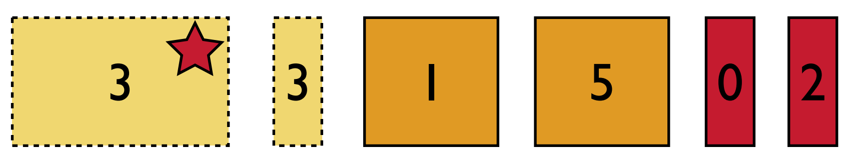


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

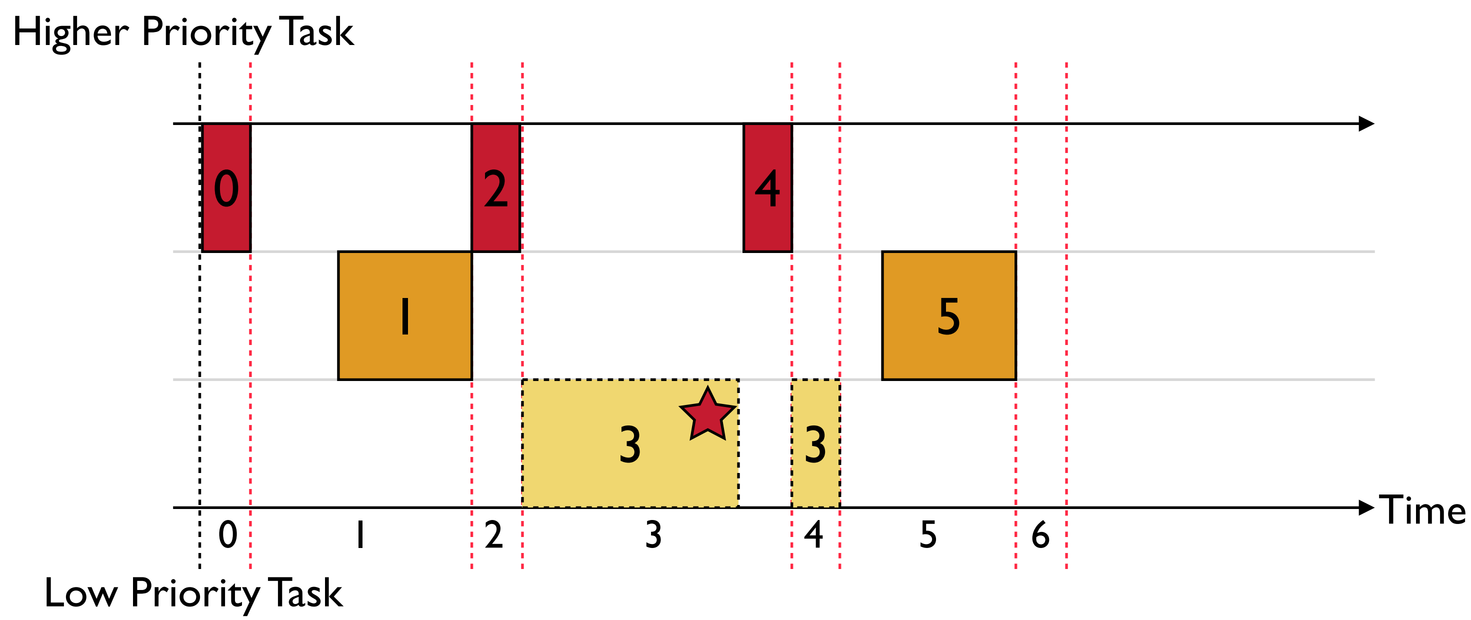
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

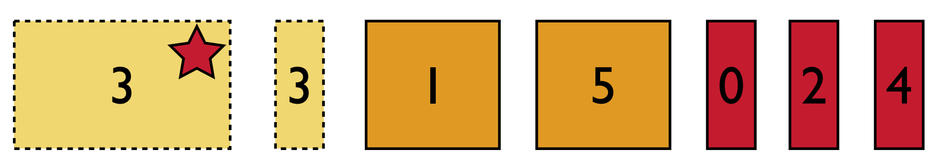


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

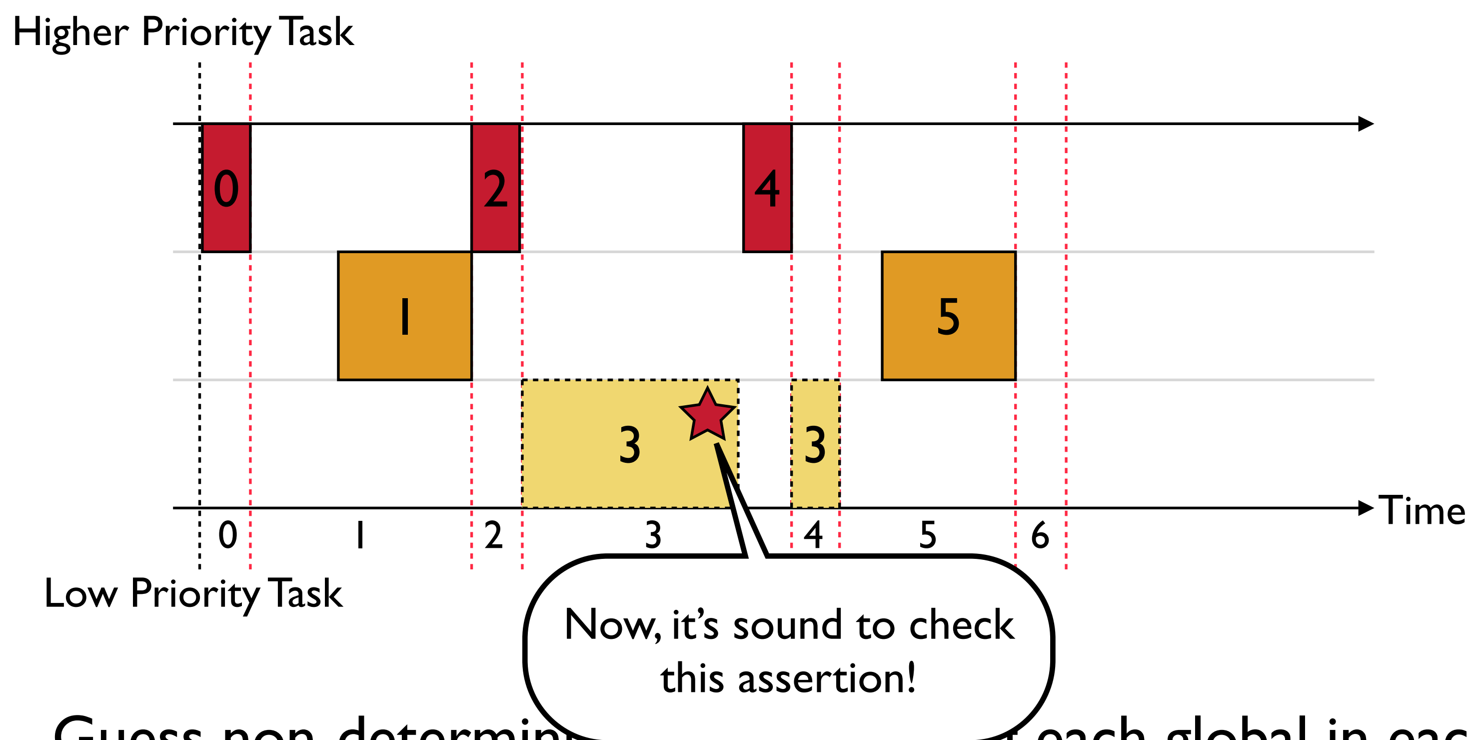
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

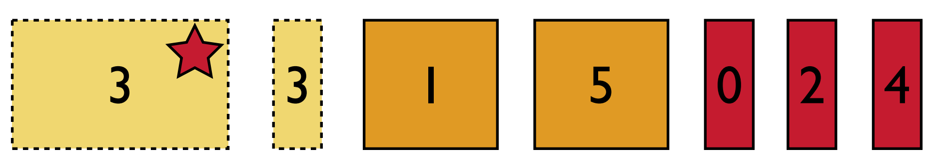


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

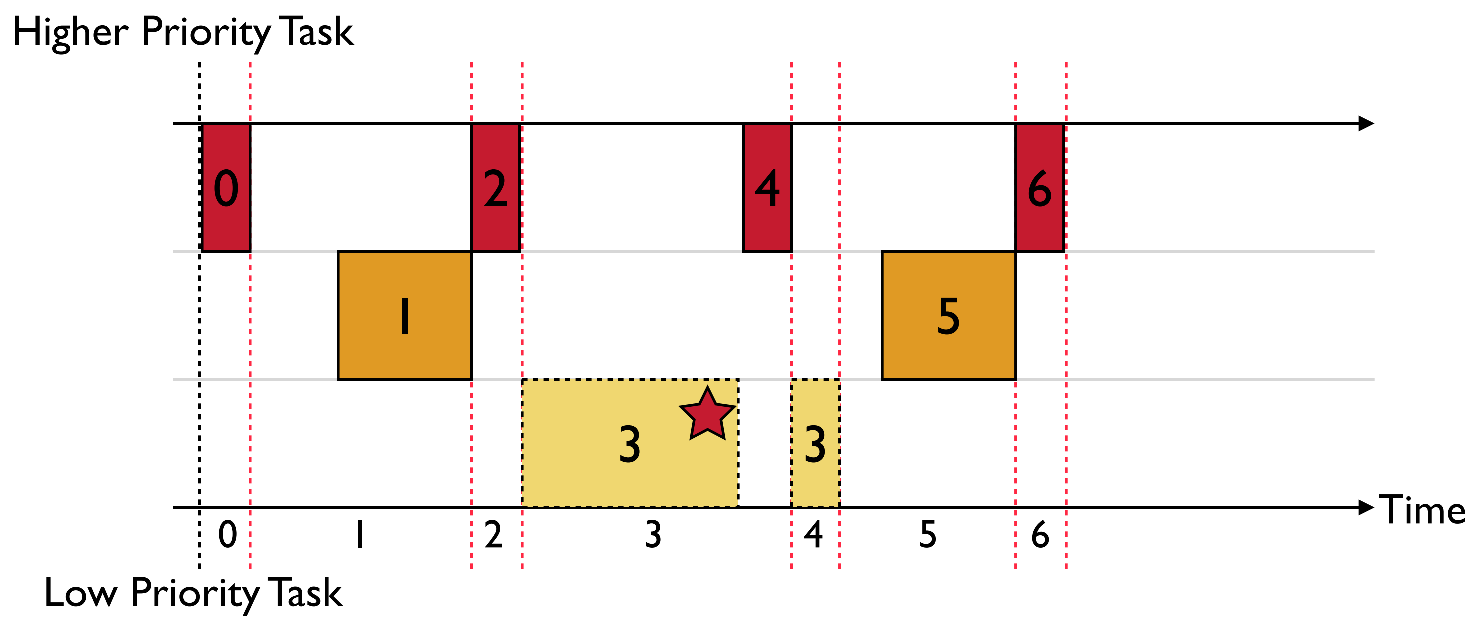
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

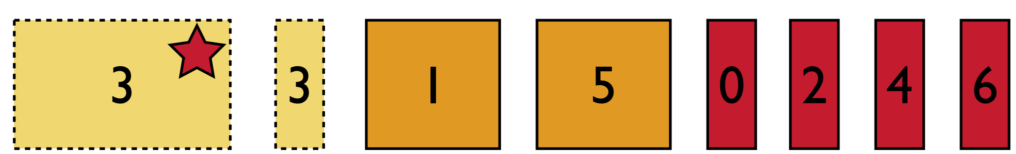


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

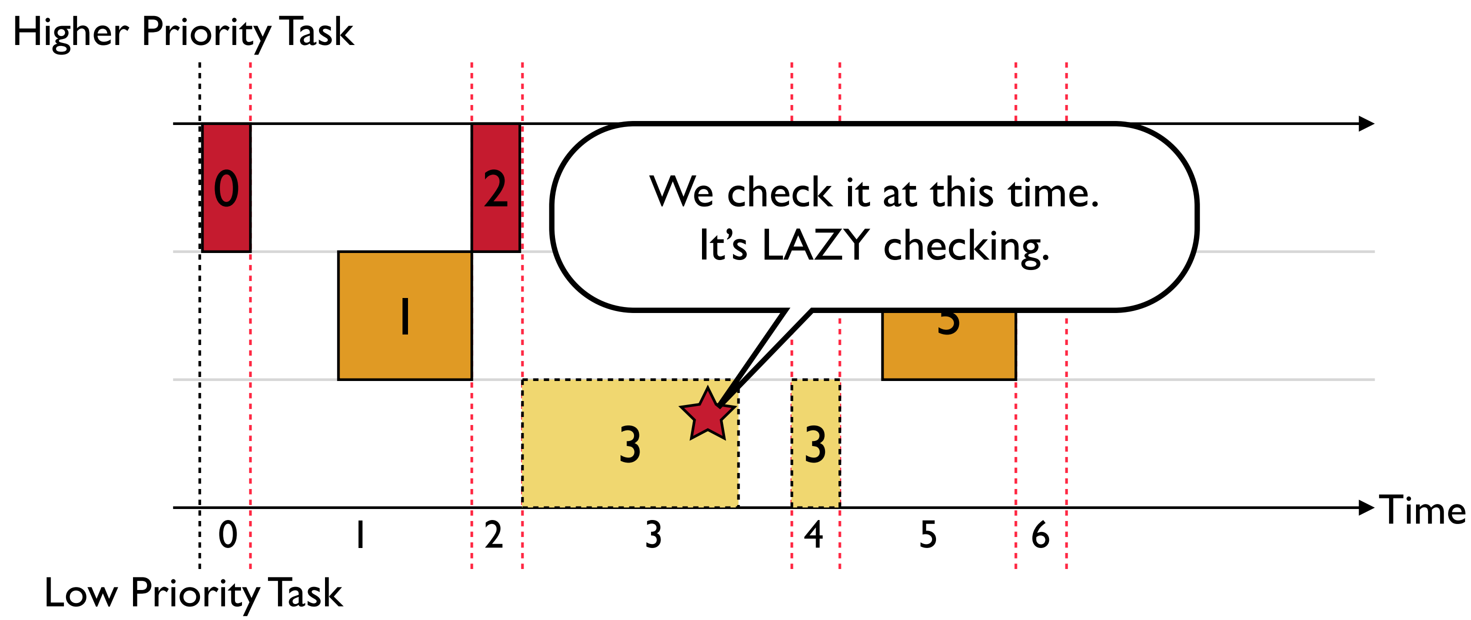
Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Sequential Execution

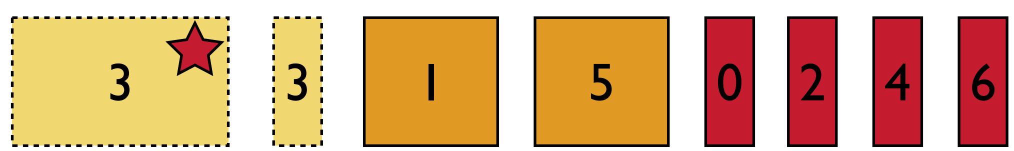


Guess non-deterministic initial value of each global in each round.

Execute Task Body, from lower priority task to higher priority task

Constraint the value of global variables g[ ]s to exclude infeasible execution

Save assertions and check them at the end of execution.



MonoSeq Sequentialization

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, $\text{last}(t, j)$ is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

var $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
$\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
var $localAssert[\,][\,]$

function MAIN( )
  SCHEDULEJOBS()
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot localAssert[t][j] :=$ TRUE
  $\forall g \in \mathsf{G} \cdot g[0] := i_g$
  $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
  for $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
    $rnd := start[t][job]$
    $endRnd := end[t][job]$
    $\hat{T}_t()$
    assume$(rnd = endRnd)$
  $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$
   assume$(g[r] = v_g[r+1])$
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

function CS(Task t)
  if (*) **then return** FALSE
  $o := rnd$
  $rnd := *$
  assume$(o < rnd \le endRnd)$
  $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$
   assume(
    $t < t' \Rightarrow$
    $(rnd \le start[t'][j'] \vee rnd > end[t'][j']))$
  return TRUE

function SCHEDULEJOBS( )

  // Jobs are sequential
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$
   assume(
    $0 \le start[t][j] \le end[t][j] \le R \wedge$
    $(\neg \text{last}(t, j) \Rightarrow end[t][j] \le start[t][j+1]))$

  // Jobs are well-nested
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $\big(t_1 < t_2 \wedge$
    $start[t_1][j_1] \le end[t_2][j_2] \wedge$
    $start[t_2][j_2] \le end[t_1][j_1]\big) \Rightarrow$
     $(start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]))$

  // Jobs respect preemption bounds
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $\big(t_1 < t_2 \wedge j_2 \ge PB_{t_1}^{t_2} \wedge$
    $start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]\big) \Rightarrow$
     $end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])$

function $\hat{T}_t$( )
  Same as T_t, but
   each statement 'st' is replaced with:
    CS(t) ; $st[g \leftarrow g[rnd]]$,
   and each 'assert(e)' is replaced with:
    $localAssert[t][job] := e$

MonoSeq

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, $\mathrm{last}(t, j)$ is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

var $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
$\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
var $localAssert[\,][\,]$

function MAIN( )
    SCHEDULEJOBS()
    $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot localAssert[t][j] :=$ TRUE
    $\forall g \in \mathsf{G} \cdot g[0] := i_g$
    $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
    for $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
        $rnd := start[t][job]$
        $endRnd := end[t][job]$
        $\hat{T}_t()$
        assume$(rnd = endRnd)$
    $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$
     assume$(g[r] = v_g[r+1])$
    $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

function CS(Task t)
    if (*) **then return** FALSE
    $o := rnd$
    $rnd := *$
    assume$(o < rnd \leq endRnd)$
    $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$
     assume(
       $t < t' \Rightarrow$
       $(rnd \leq start[t'][j'] \vee rnd > end[t'][j']))$
    return TRUE

function SCHEDULEJOBS( )

    // Jobs are sequential
        $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$
         assume(
           $0 \leq start[t][j] \leq end[t][j] \leq R \wedge$
           $(\neg \mathrm{last}(t, j) \Rightarrow end[t][j] \leq start[t][j+1]))$

    // Jobs are well-nested
        $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
         assume(
           $(t_1 < t_2 \wedge$
           $start[t_1][j_1] \leq end[t_2][j_2] \wedge$
           $start[t_2][j_2] \leq end[t_1][j_1]) \Rightarrow$
            $(start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]))$

    // Jobs respect preemption bounds
        $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
         assume(
           $(t_1 < t_2 \wedge j_2 \geq PB_{t_1}^{t_2} \wedge$
           $start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]) \Rightarrow$
            $end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])$

function $\hat{T}_t$( )
    Same as T_t, but
      each statement 'st' is replaced with:
        CS(t) ; $st[g \leftarrow g[rnd]]$,
      and each 'assert(e)' is replaced with:
        $localAssert[t][job] := e$

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, $\mathrm{last}(t, j)$ is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

var $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
$\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
var $localAssert[\,][\,]$

function MAIN( )
  SCHEDULEJOBS()
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot localAssert[t][j] :=$ TRUE
  $\forall g \in \mathsf{G} \cdot g[0] := i_g$
  $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
  for $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
    $rnd := start[t][job]$
    $endRnd := end[t][job]$
    $\hat{T}_t()$
    assume$(rnd = endRnd)$
  $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$
   assume$(g[r] = v_g[r+1])$
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

function CS(Task t)
  if (*) **then return** FALSE
  $o := rnd$
  $rnd := *$
  assume$(o < rnd \leq endRnd)$
  $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$
   assume(
    $t < t' \Rightarrow$
    $(rnd \leq start[t'][j'] \vee rnd > end[t'][j']))$
  return TRUE

function SCHEDULEJOBS( )

// Jobs are sequential
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$
   assume(
    $0 \leq start[t][j] \leq end[t][j] \leq R \wedge$
    $(\neg \mathrm{last}(t, j) \Rightarrow end[t][j] \leq start[t][j+1]))$

// Jobs are well-nested
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $(t_1 < t_2 \wedge$
    $start[t_1][j_1] \leq end[t_2][j_2] \wedge$
    $start[t_2][j_2] \leq end[t_1][j_1]) \Rightarrow$
     $(start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]))$

// Jobs respect preemption bounds
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $(t_1 < t_2 \wedge j_2 \geq PB^{t_2}_{t_1} \wedge$
    $start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]) \Rightarrow$
     $end[t_2][j_2 - PB^{t_2}_{t_1}] < start[t_1][j_1])$

function $\hat{T}_t$( )
  Same as T_t, but
   each statement 'st' is replaced with:
    CS(t) ; $st[g \leftarrow g[rnd]]$,
   and each 'assert(e)' is replaced with:
    $localAssert[t][job] := e$

MonoSeq

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, $\text{last}(t, j)$ is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

var $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
$\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
var $localAssert[\,][\,]$

function MAIN( )
  SCHEDULEJOBS()
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot localAssert[t][j] :=$ TRUE
  $\forall g \in \mathsf{G} \cdot g[0] := i_g$
  $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
  for $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
    $rnd := start[t][job]$
    $endRnd := end[t][job]$
    $\hat{T}_t()$
    assume$(rnd = endRnd)$
  $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$
   assume$(g[r] = v_g[r+1])$
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

function CS(Task t)
  if (*) **then return** FALSE
  $o := rnd$
  $rnd := *$
  assume$(o < rnd \le endRnd)$
  $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$
   assume(
    $t < t' \Rightarrow$
    $(rnd \le start[t'][j'] \vee rnd > end[t'][j']))$
  return TRUE

function SCHEDULEJOBS( )

// Jobs are sequential
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$
   assume(
    $0 \le start[t][j] \le end[t][j] \le R \wedge$
    $(\neg\text{last}(t, j) \Rightarrow end[t][j] \le start[t][j+1]))$

// Jobs are well-nested
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $(t_1 < t_2 \wedge$
    $start[t_1][j_1] \le end[t_2][j_2] \wedge$
    $start[t_2][j_2] \le end[t_1][j_1]) \Rightarrow$
    $(start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]))$

// Jobs respect preemption bounds
  $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$
   assume(
    $(t_1 < t_2 \wedge j_2 \ge PB_{t_1}^{t_2} \wedge$
    $start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]) \Rightarrow$
    $end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])$

function $\hat{T}_t$( )
Same as T_t, but
 each statement 'st' is replaced with:
  CS(t) ; $st[g \leftarrow g[rnd]]$,
 and each 'assert(e)' is replaced with:
  $localAssert[t][job] := e$

MonoSeq

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bound... by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of ... is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, ... value.

It also supports two types of common locking mechanisms by encoding them as constraints:

1. Preemption Lock
2. Priority Ceiling Lock

var $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
$\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
var *localAssert*$[\,][\,]$

function MAIN( )
  SCHEDULEJOBS()
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ *localAssert*$[t][j] :=$ TRUE
  $\forall g \in \mathsf{G} \cdot g[0] := i_g$
  $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
  for $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
    $rnd := start[t][job]$
    $endRnd := end[t][job]$
    $\hat{T}_t()$
    assume$(rnd = endRnd)$
  $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$ assume$(g[r] = v_g[r+1])$
  $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

function CS(Task t)
  if (*) **then return** FALSE
  $o := rnd$
  $rnd := *$
  assume$(o < rnd \le endRnd)$
  $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$ assume($t < t' \Rightarrow (rnd \le start[t'][j'] \vee rnd > end[t'][j']))$
  return TRUE

$(\neg \text{last}(t, j) \Rightarrow end[t][j] \le start[t][j+1]))$

// Jobs are well-nested

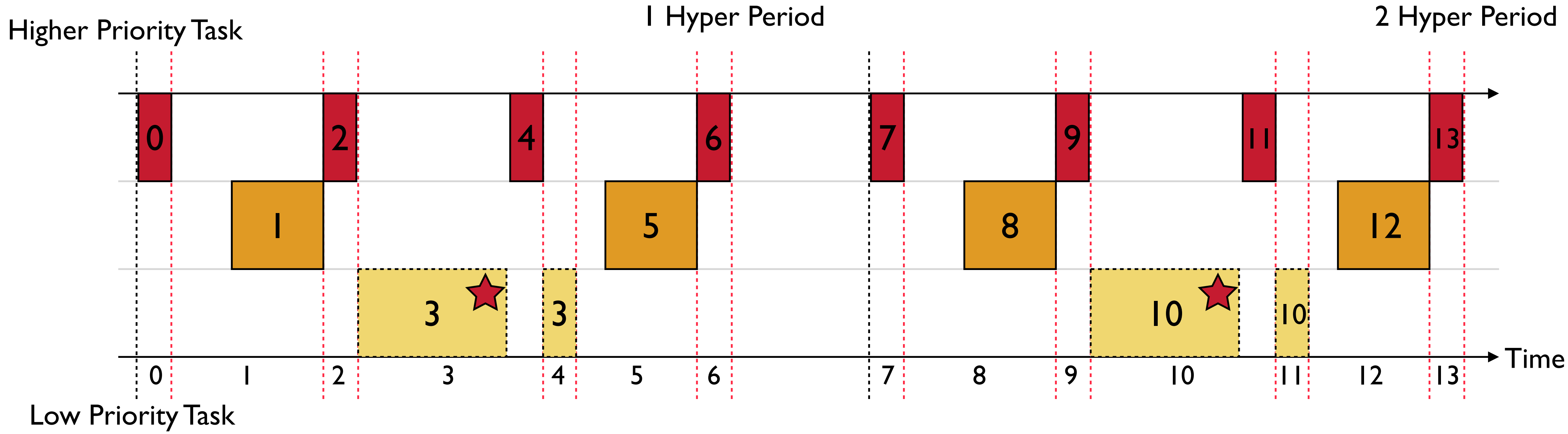
$\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume($\big(t_1 < t_2 \wedge start[t_1][j_1] \le end[t_2][j_2] \wedge start[t_2][j_2] \le end[t_1][j_1]\big) \Rightarrow (start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]))$

// Jobs respect preemption bounds

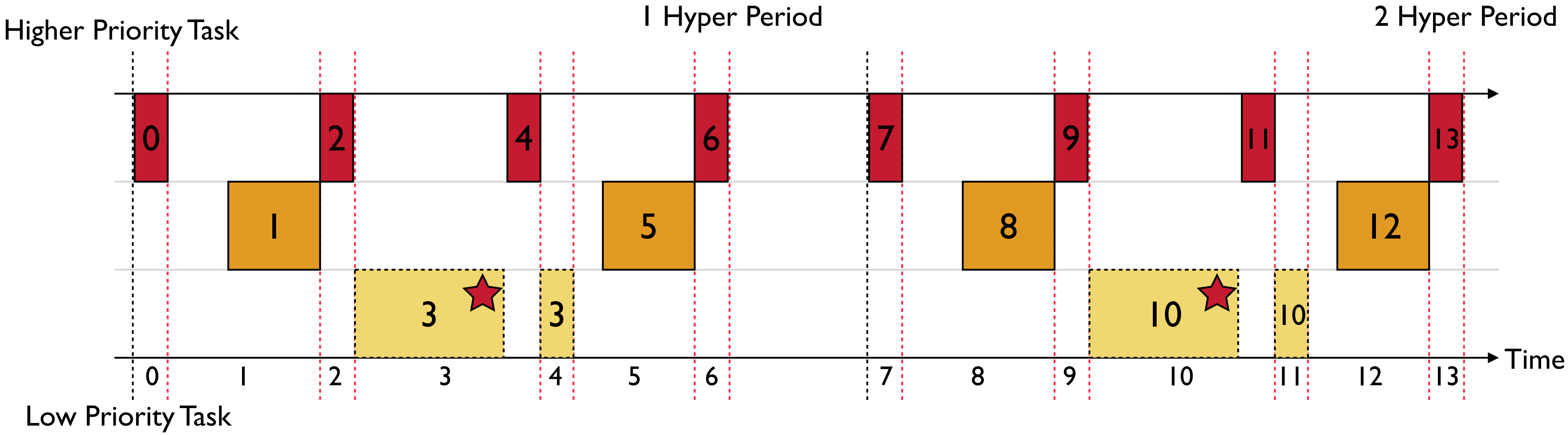
$\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume($\big(t_1 < t_2 \wedge j_2 \ge PB_{t_1}^{t_2} \wedge start[t_1][j_1] \le start[t_2][j_2] \le end[t_2][j_2] < end[t_1][j_1]\big) \Rightarrow end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])$

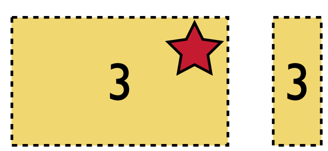
function $\hat{T}_t$( )
  Same as T_t, but each statement 'st' is replaced with:
    CS(t) ; $st[g \leftarrow g[rnd]]$,
  and each 'assert(e)' is replaced with:
    localAssert$[t][job] := e$

Observations

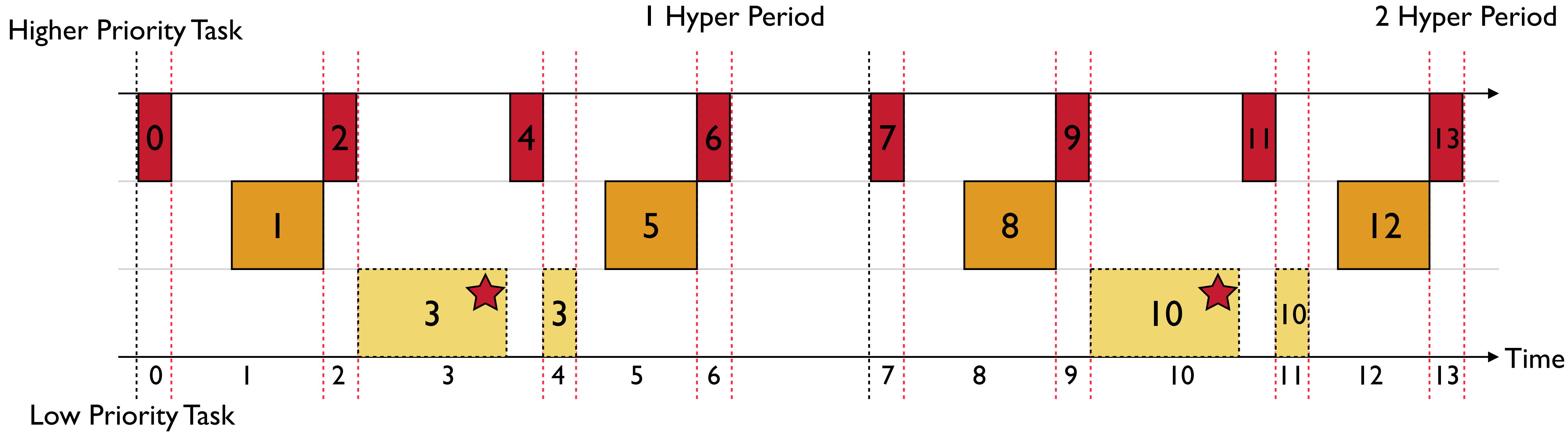


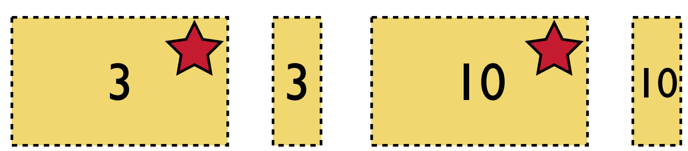
Observations



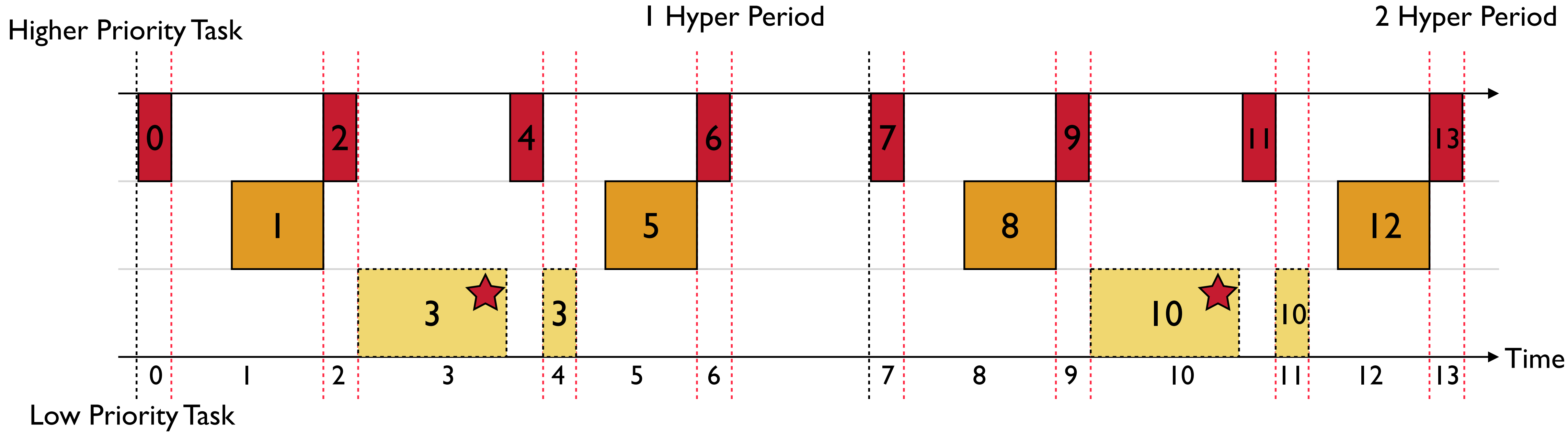


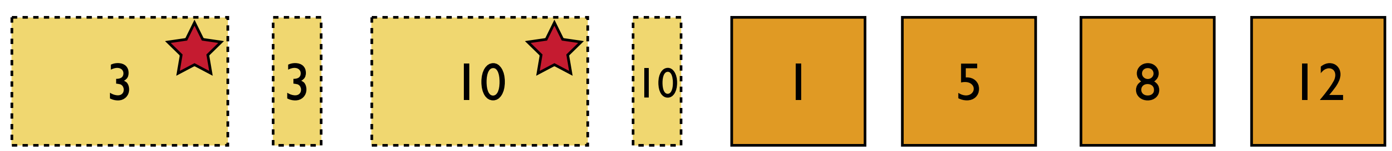
Observations



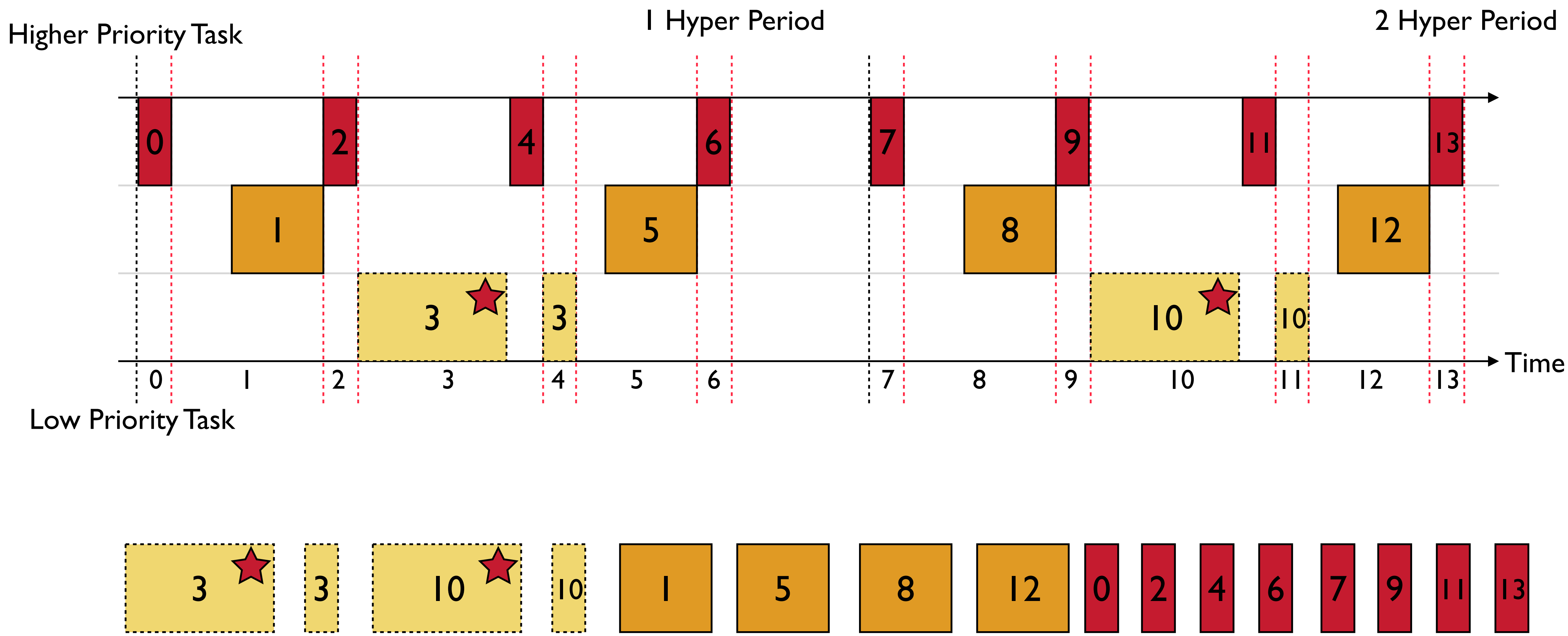


Observations

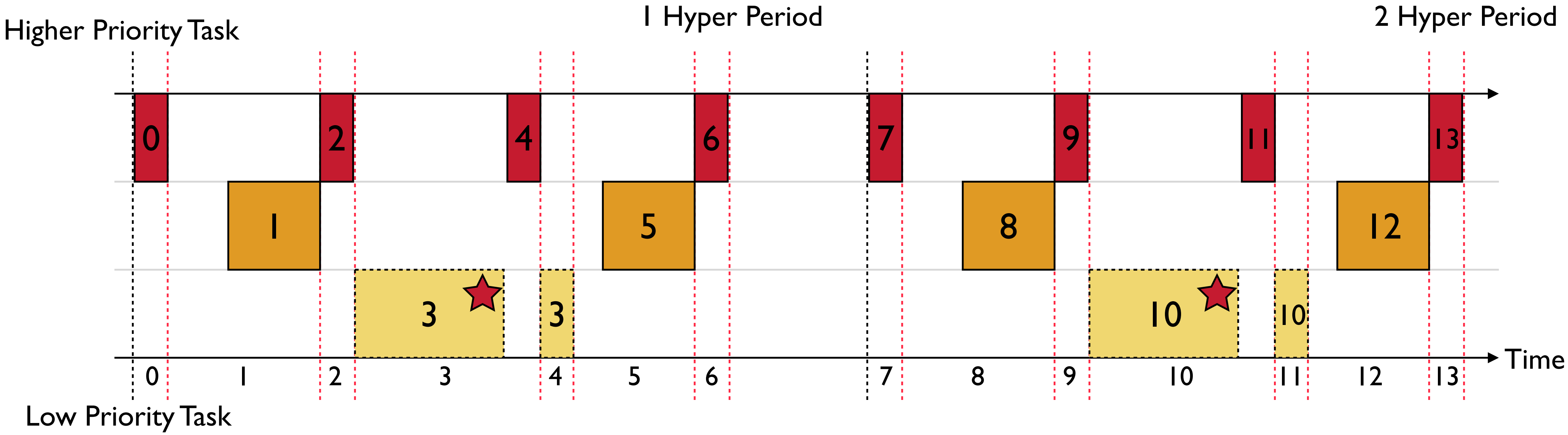


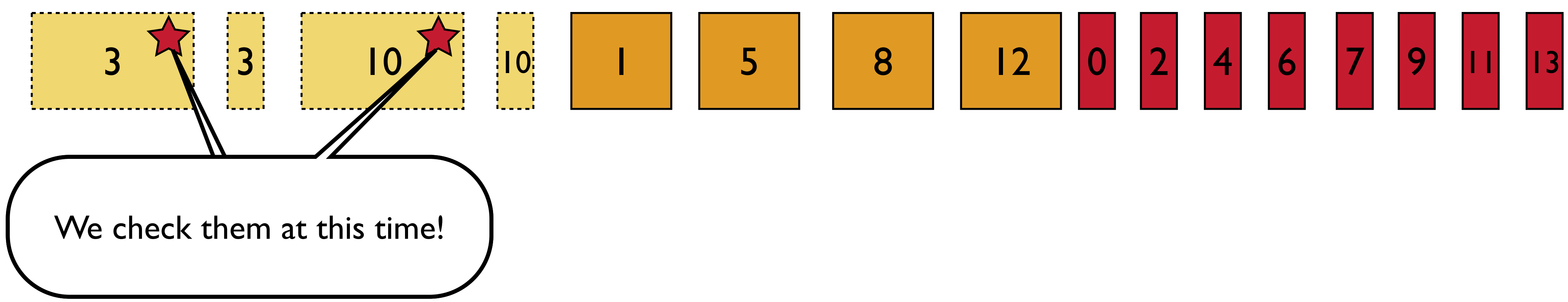


Observations

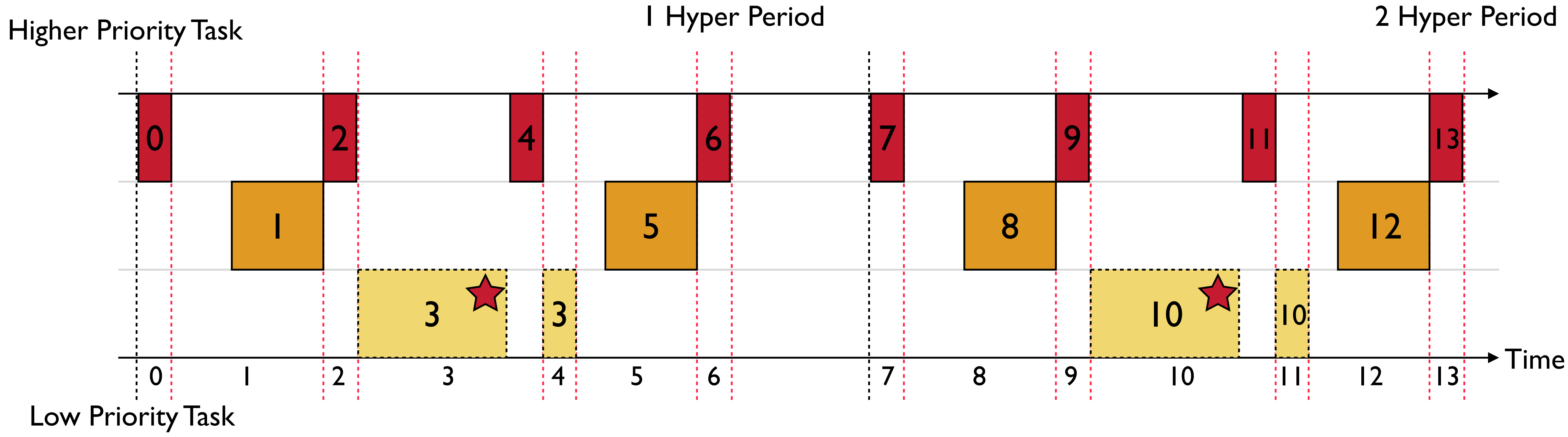


Observations



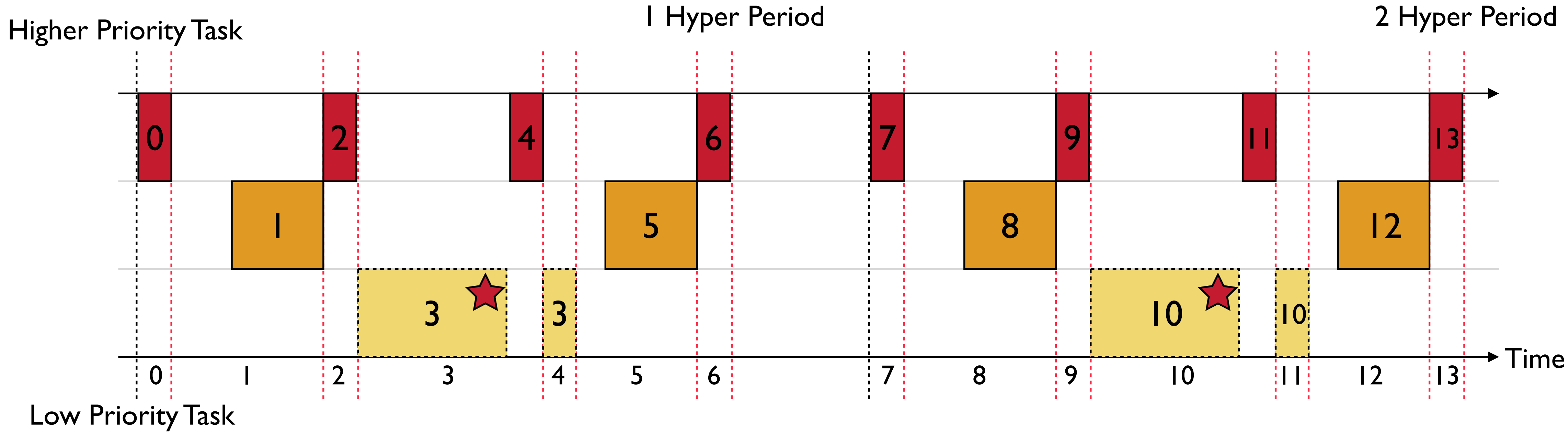


Observations

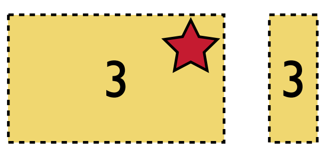


We should consider "Hyper-period"!

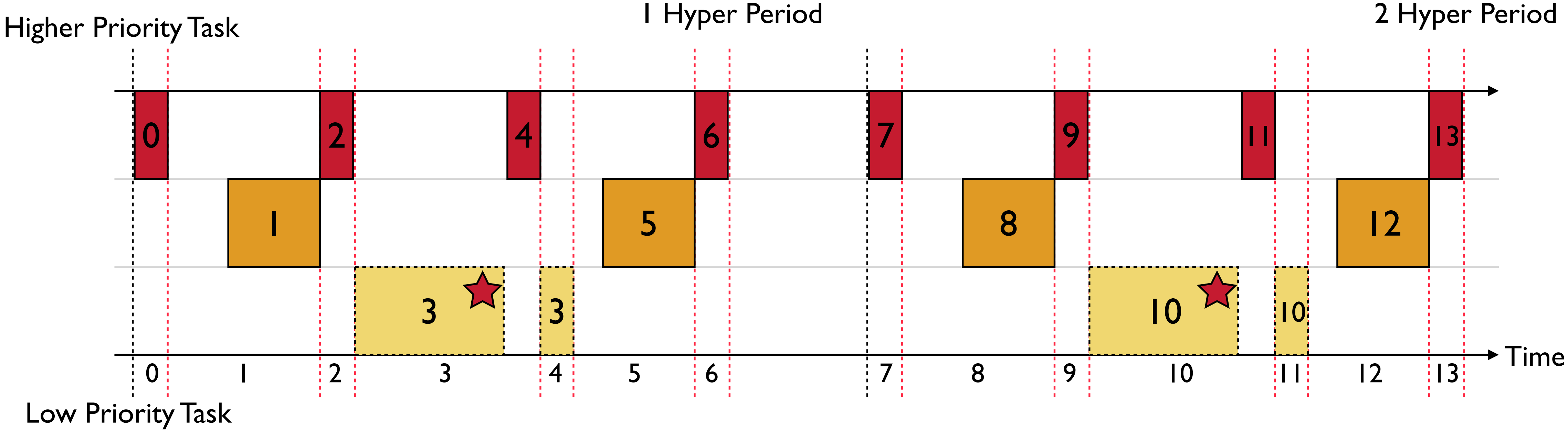
Observations



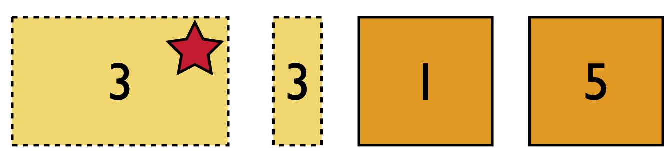
We should consider "Hyper-period"!



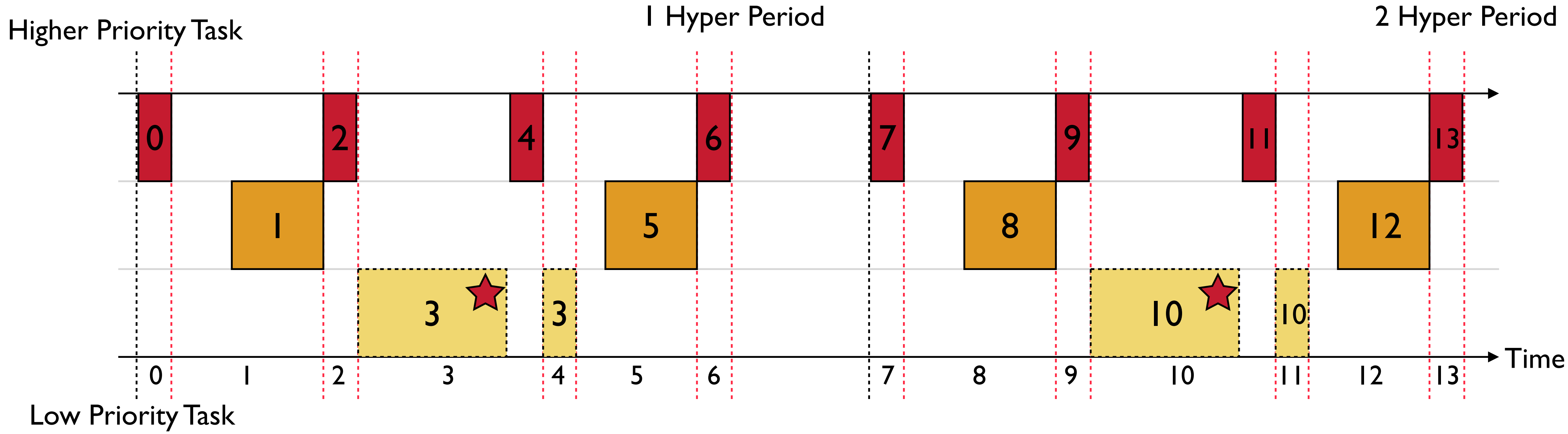
Observations



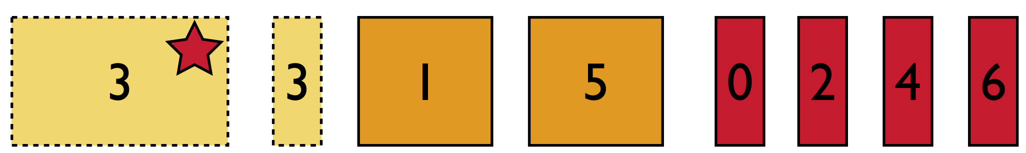
We should consider "Hyper-period"!



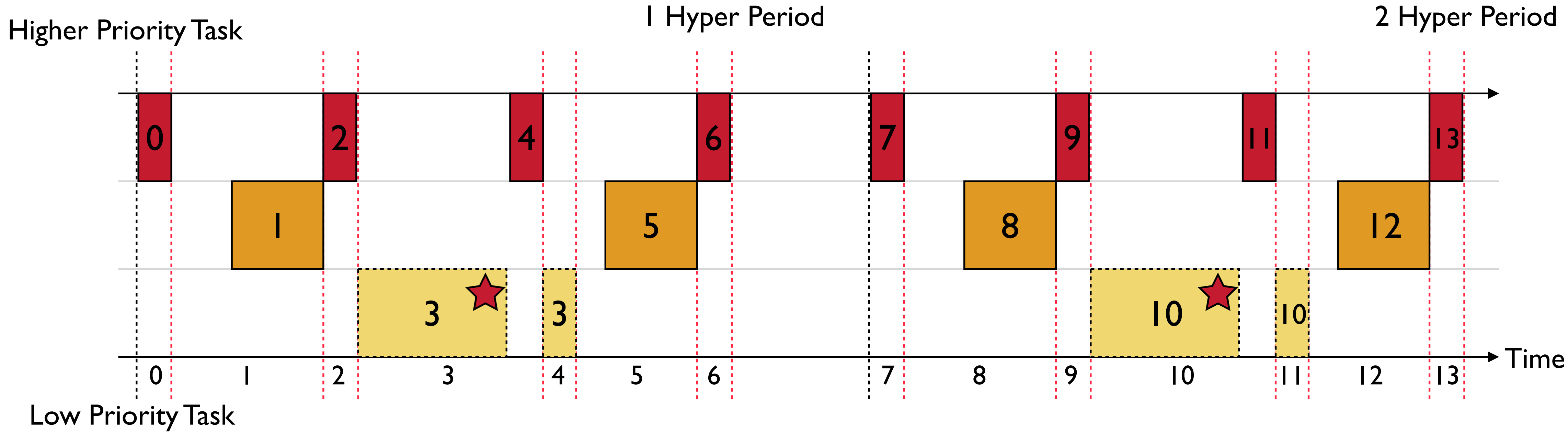
Observations



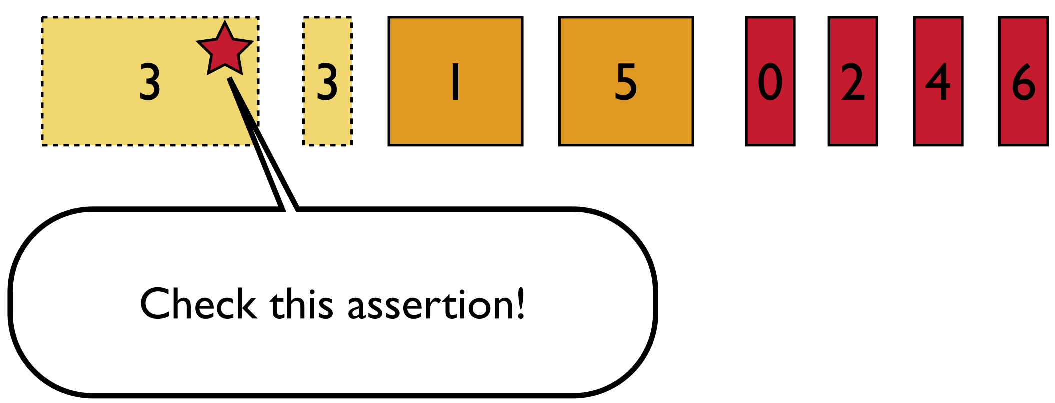
We should consider "Hyper-period"!



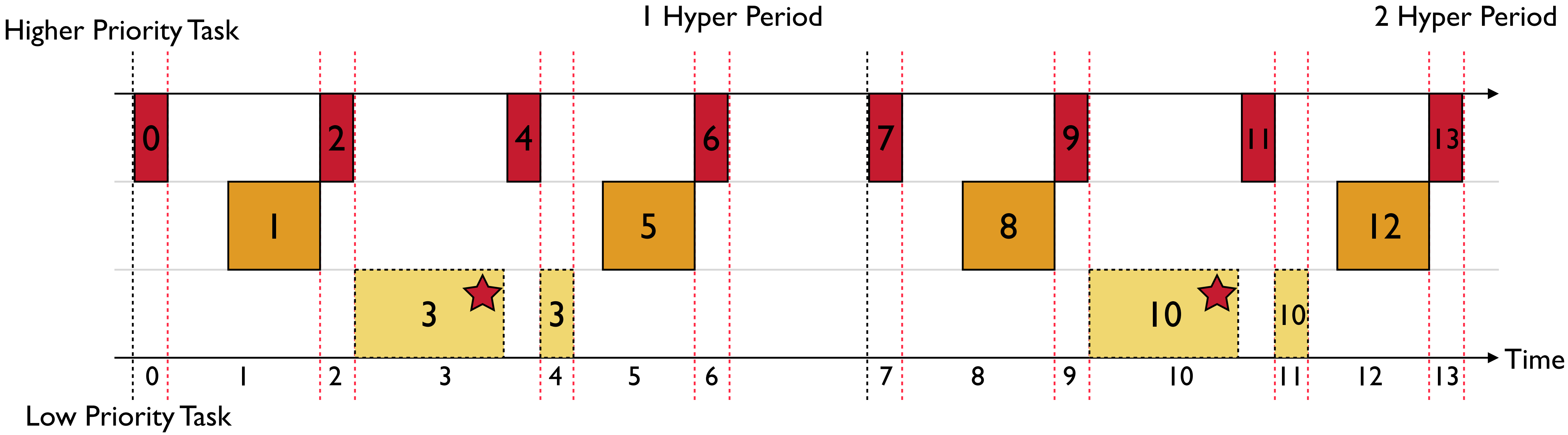
Observations



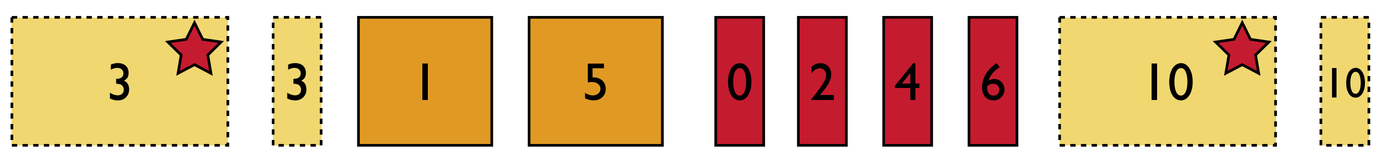
We should consider "Hyper-period"!



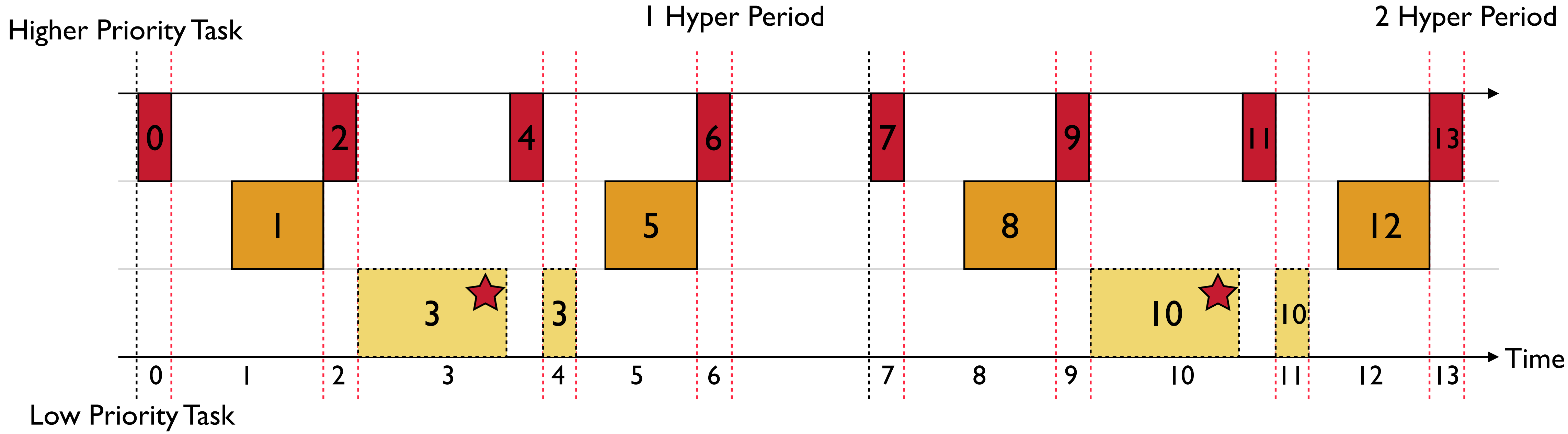
Observations



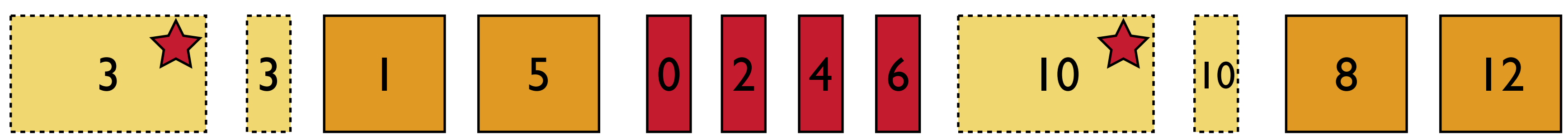
We should consider "Hyper-period"!



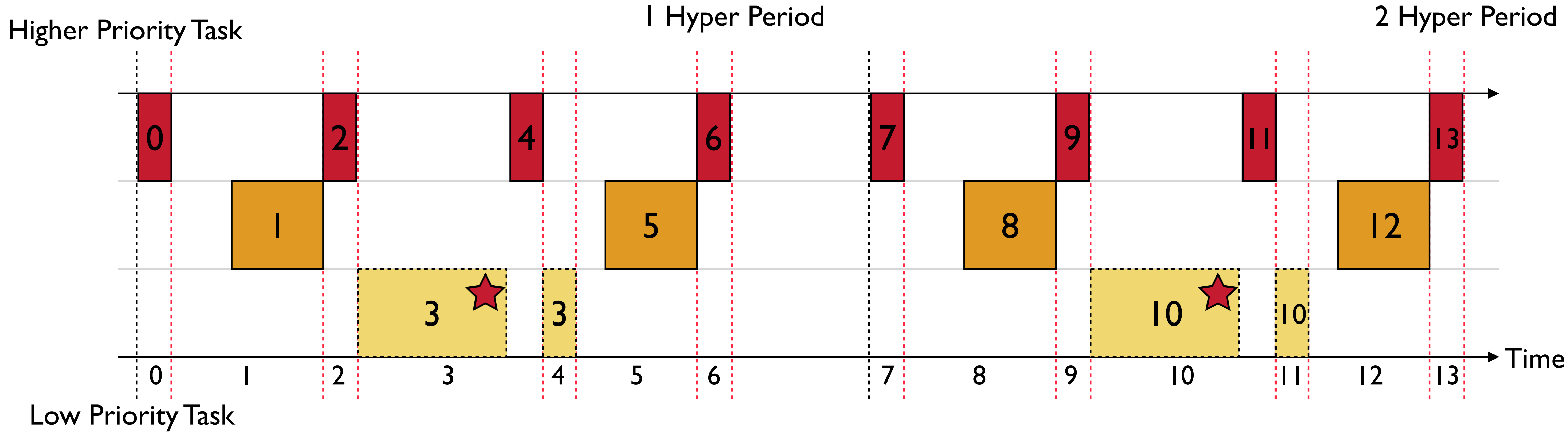
Observations



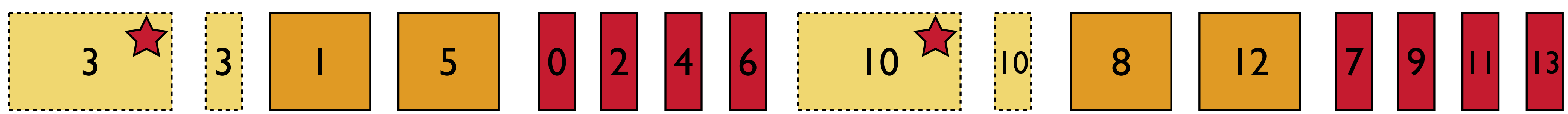
We should consider "Hyper-period"!



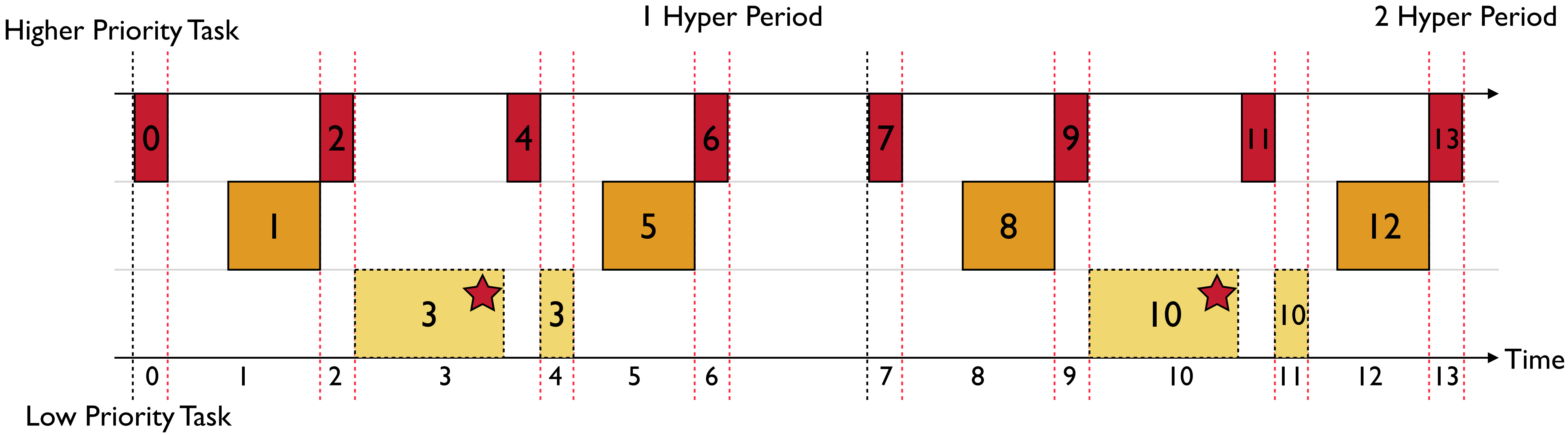
Observations



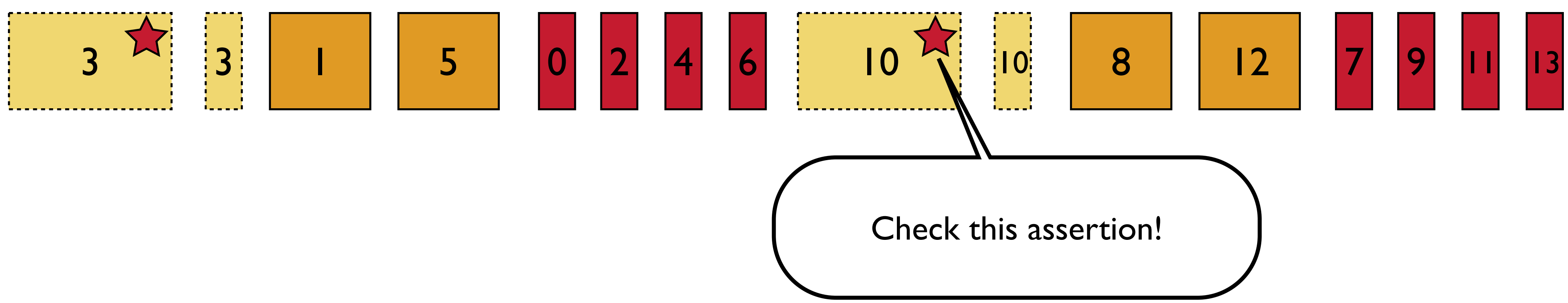
We should consider "Hyper-period"!



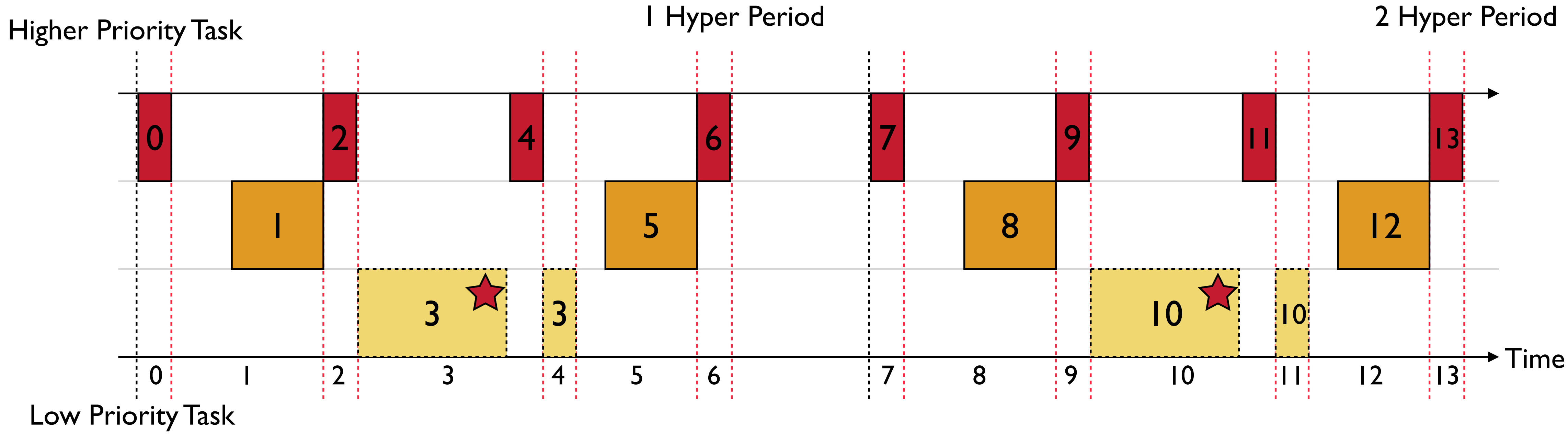
Observations



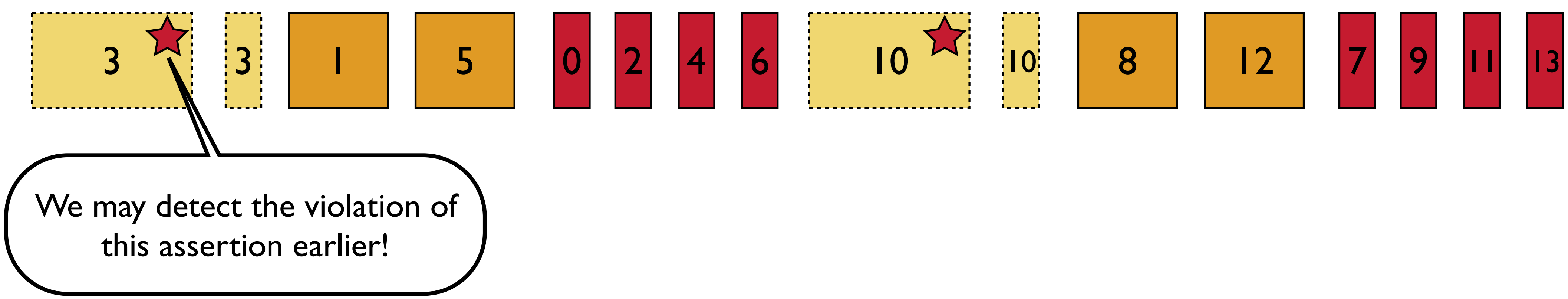
We should consider "Hyper-period"!



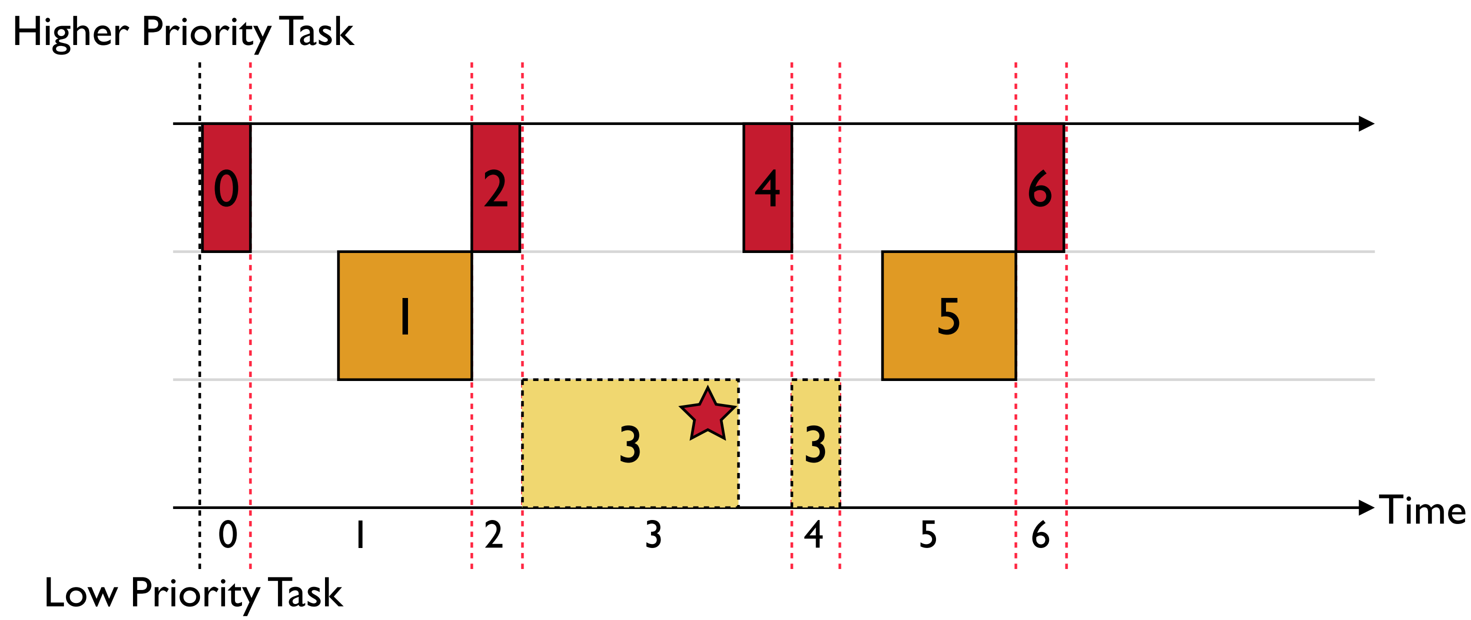
Observations



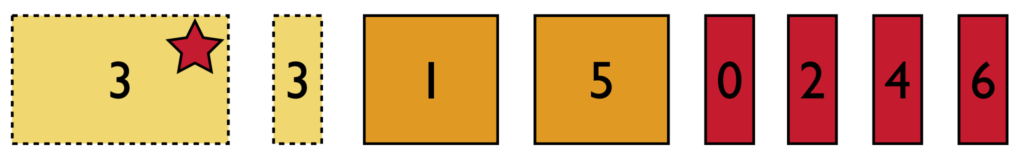
We should consider “Hyper-period”!



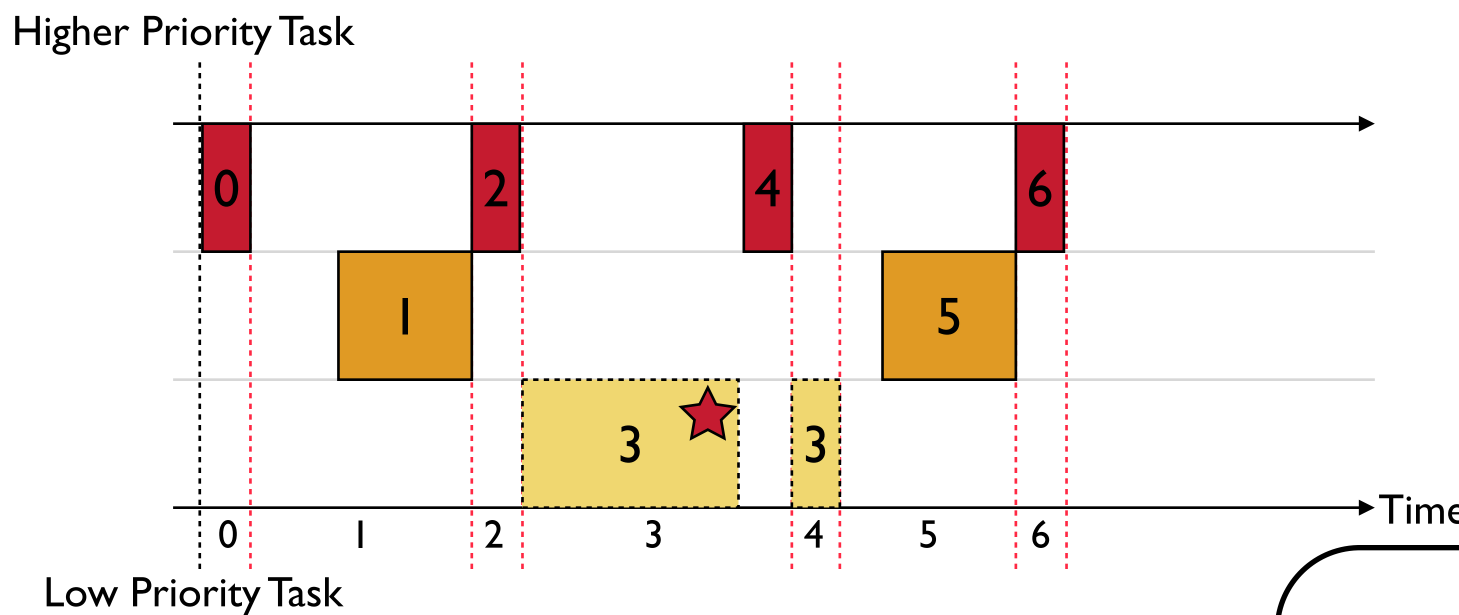
Observations



We executed jobs in the order of their priorities.

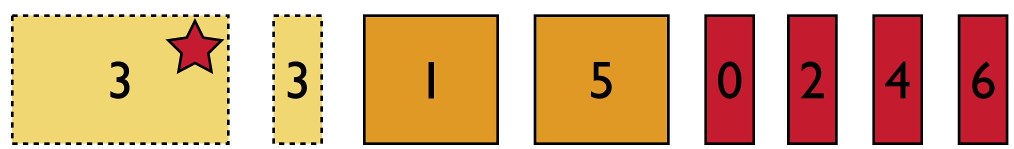


Observations

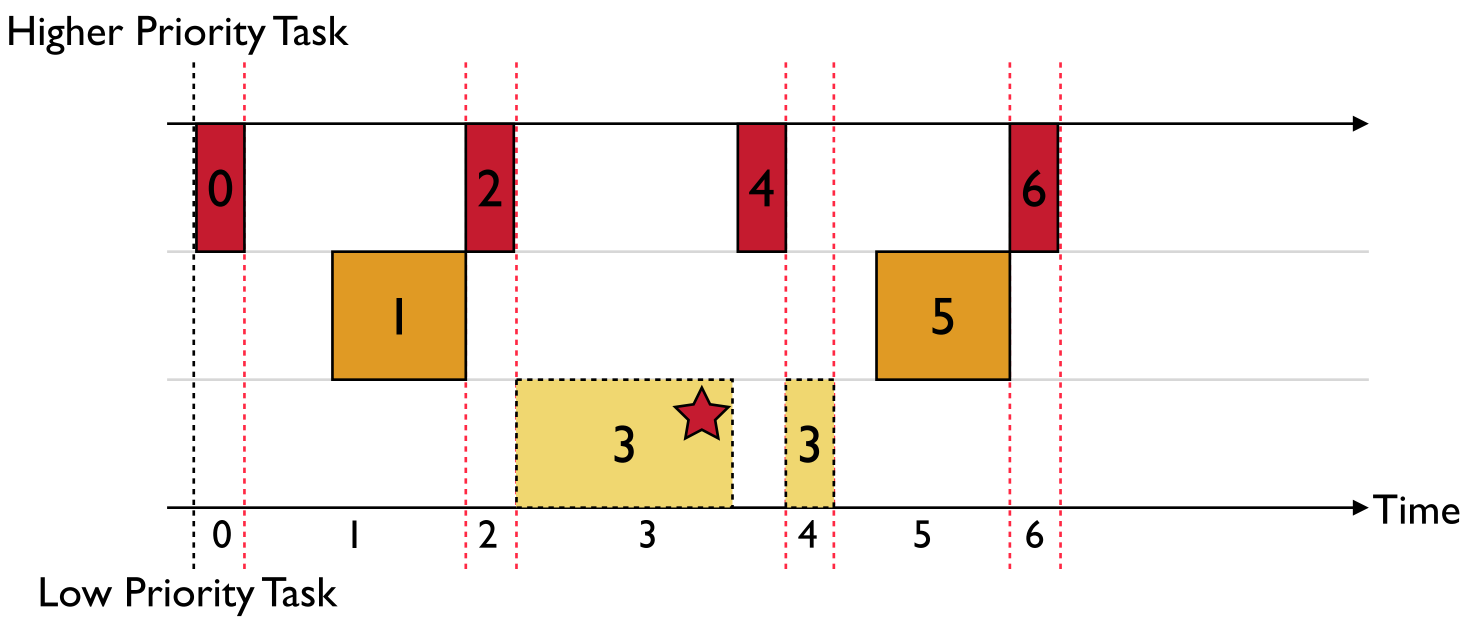


Can we do better?

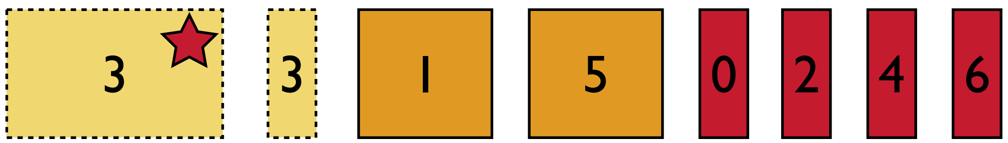
We executed jobs in the order of their priorities.



Observations



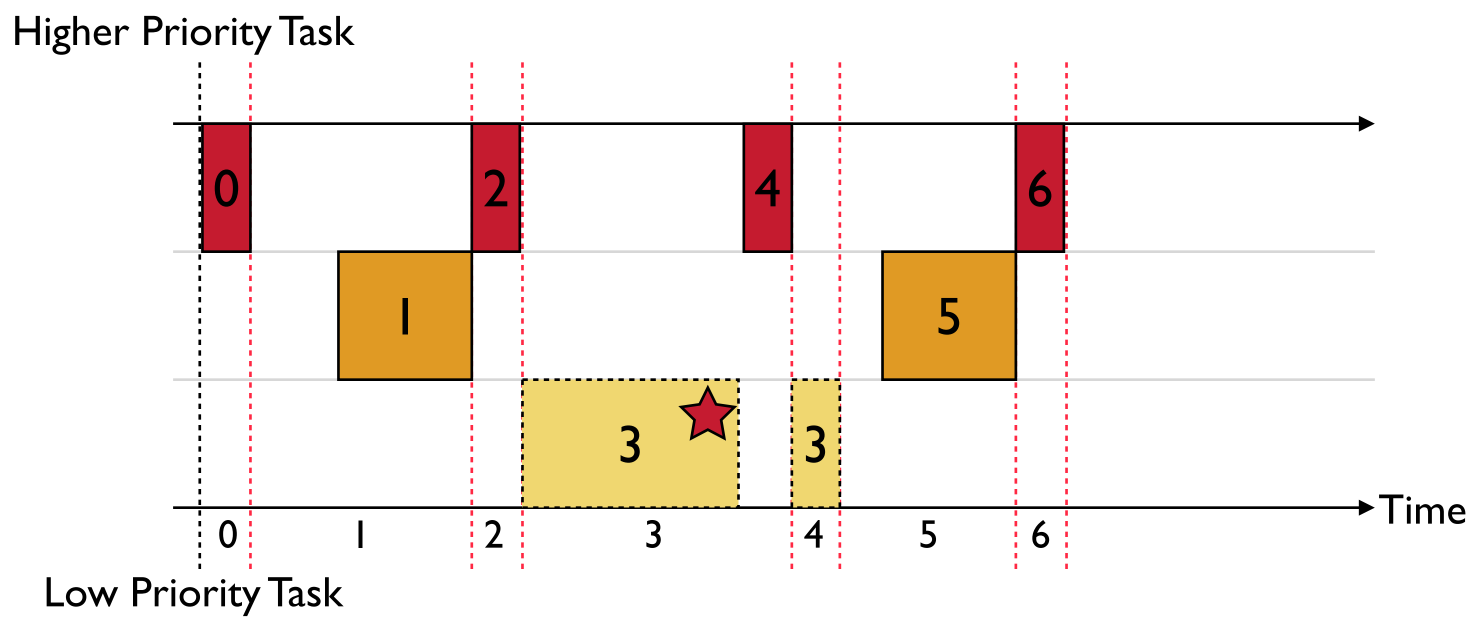
We executed jobs in the order of their priorities.



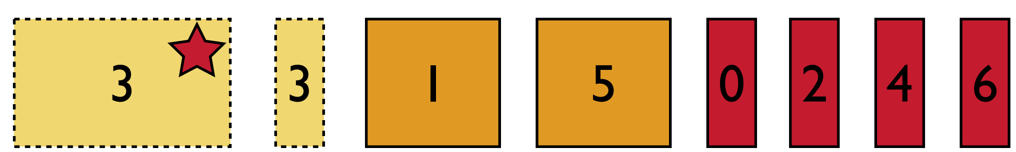
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2$$

Observations



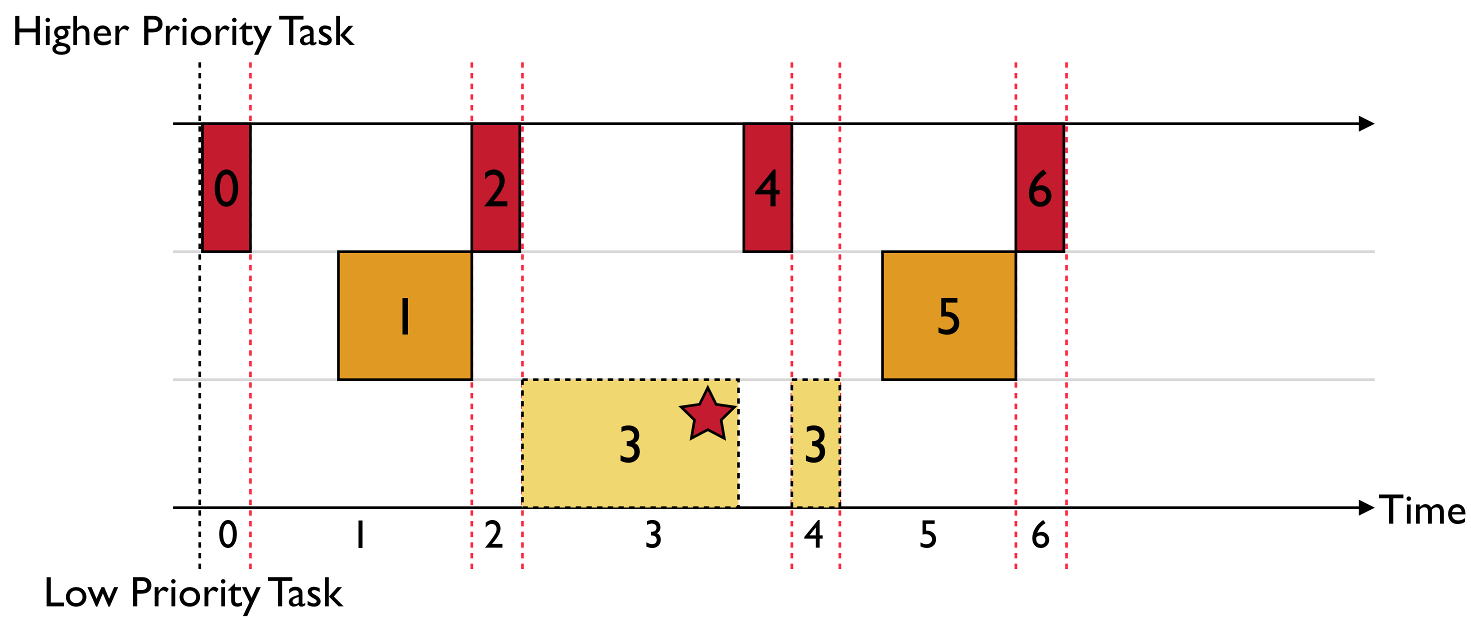
We executed jobs in the order of their priorities.



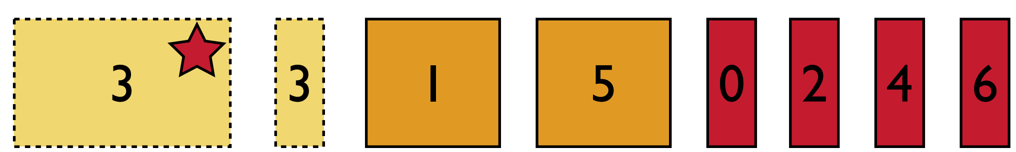
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \\ \end{cases}$$

Observations

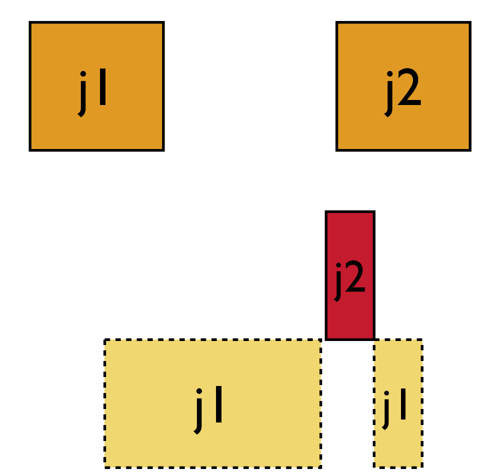


We executed jobs in the order of their priorities.

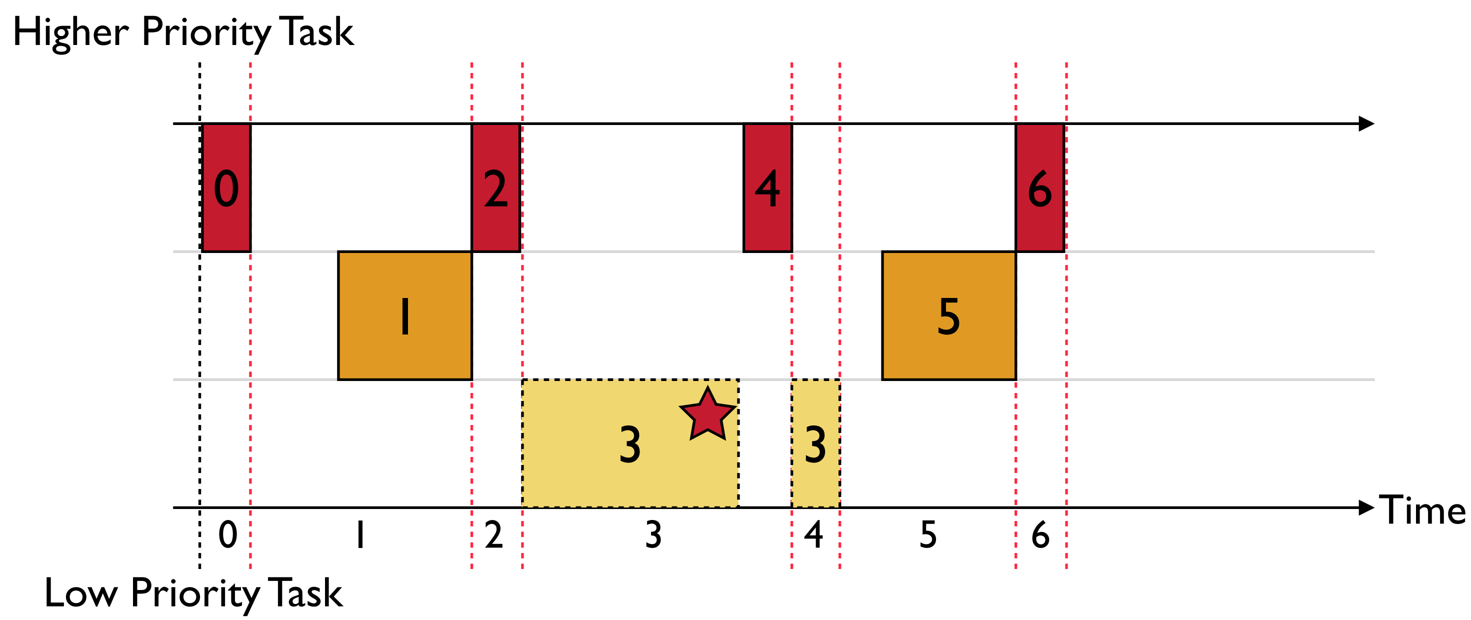


Order jobs by $\sqsubset$ relation

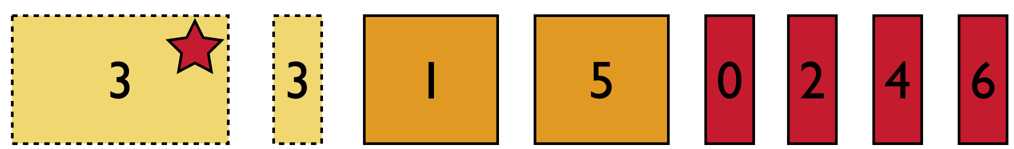
$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$



Observations

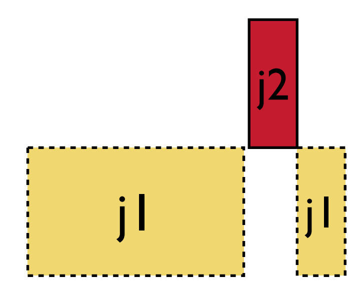


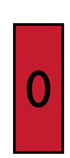
We executed jobs in the order of their priorities.



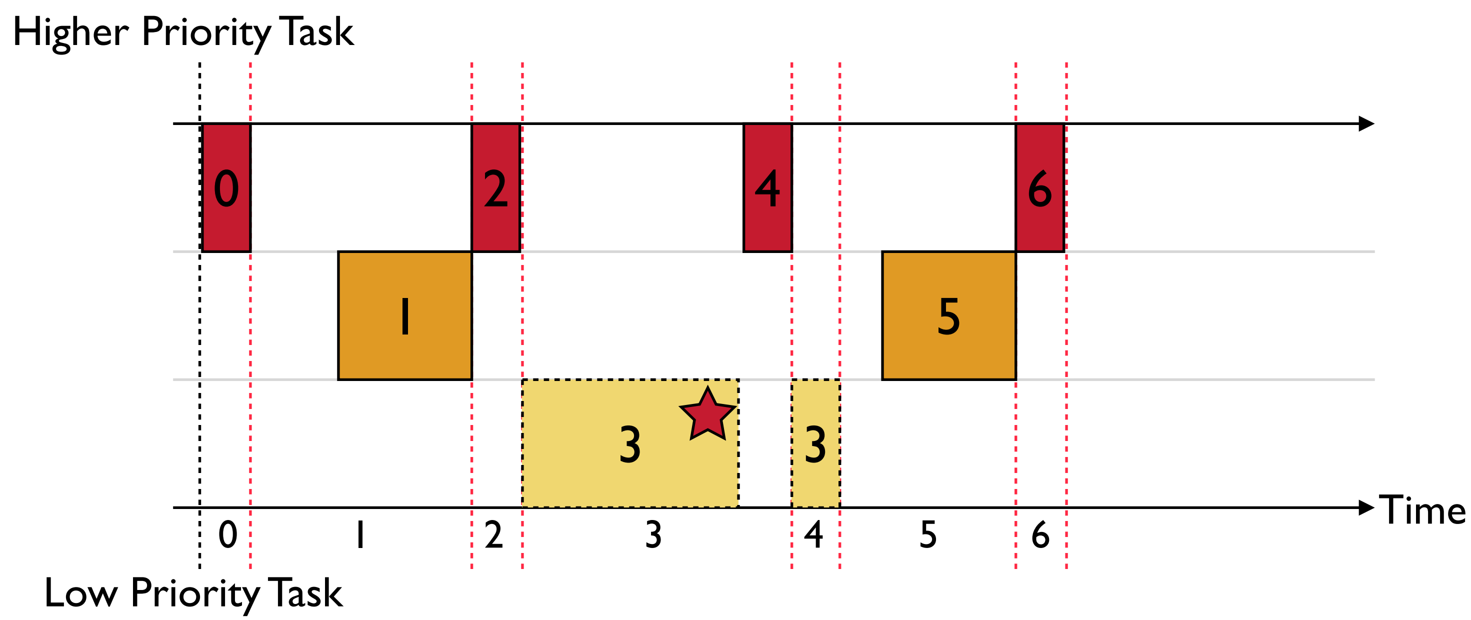
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

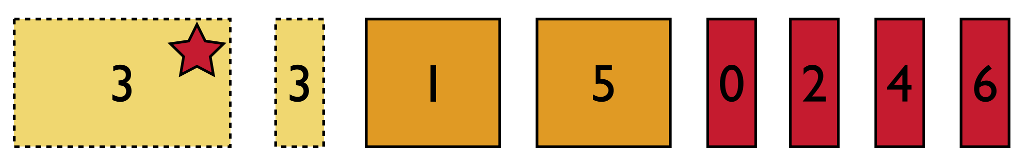




Observations

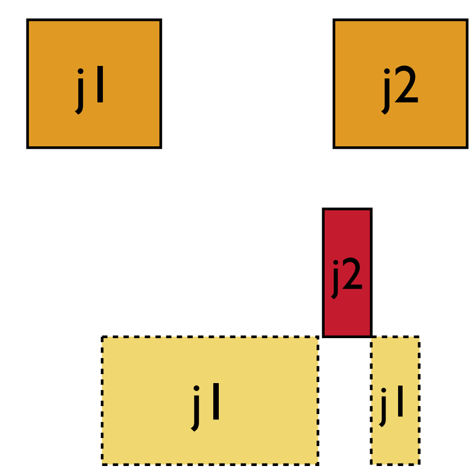


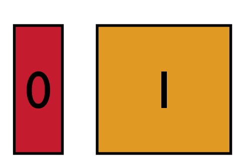
We executed jobs in the order of their priorities.



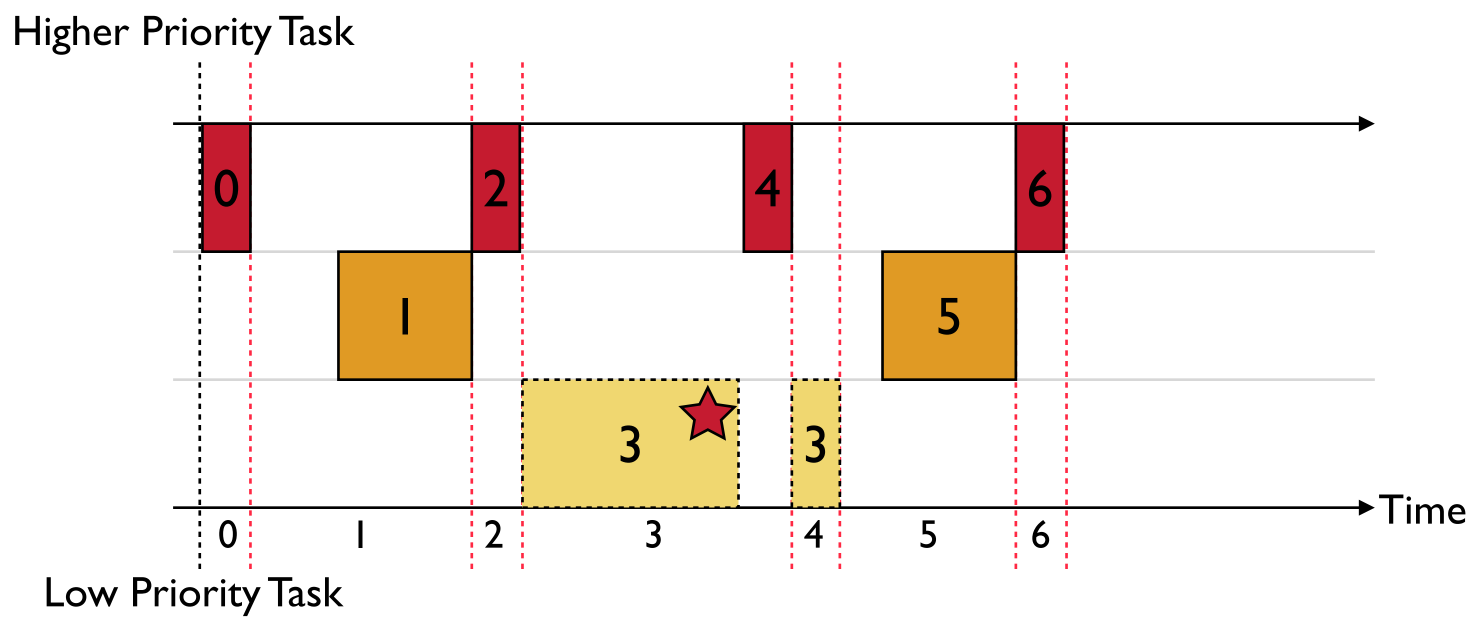
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

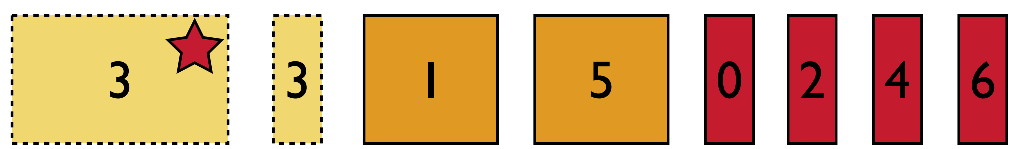




Observations

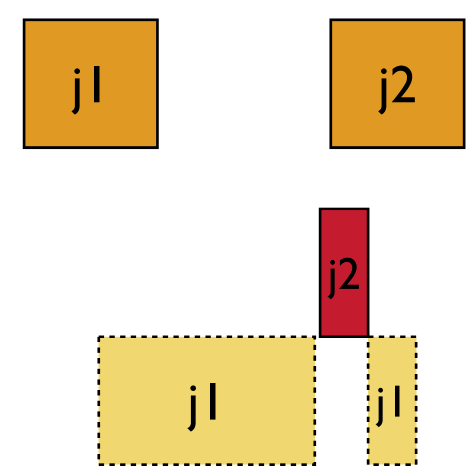


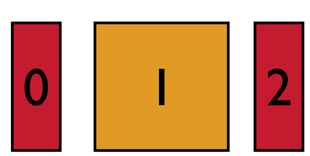
We executed jobs in the order of their priorities.



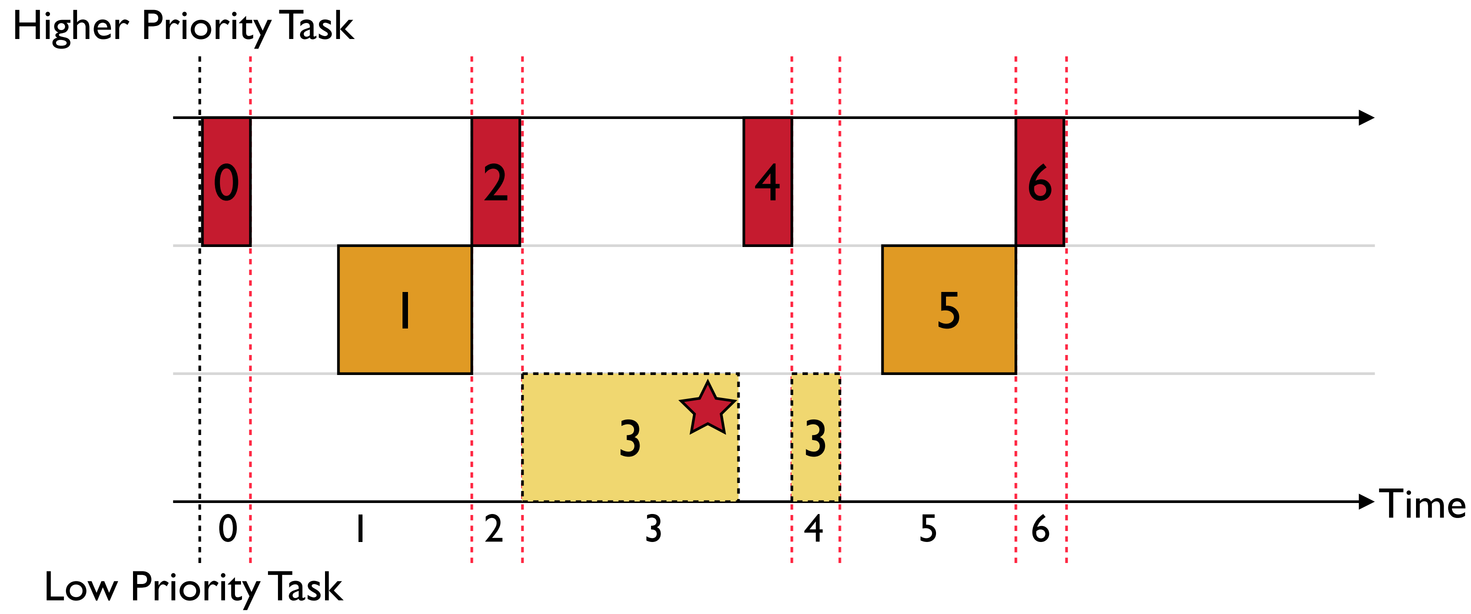
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

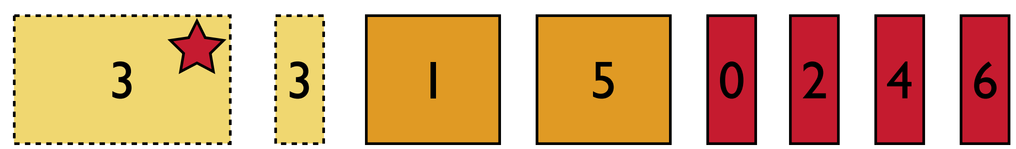




Observations

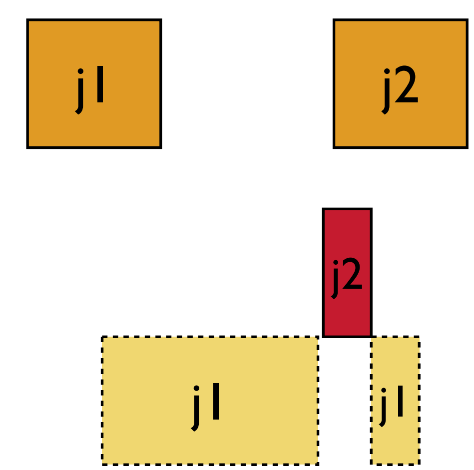


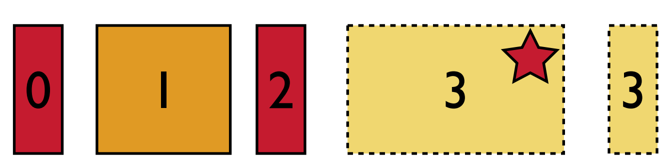
We executed jobs in the order of their priorities.



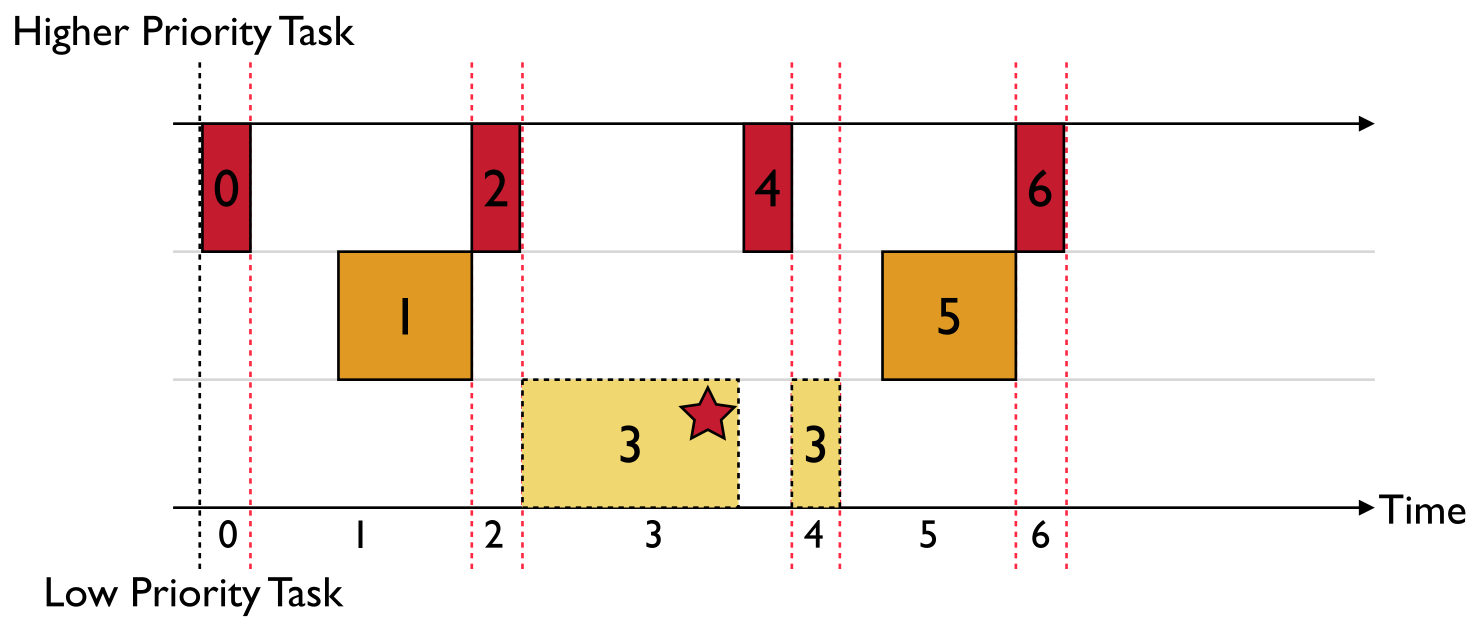
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

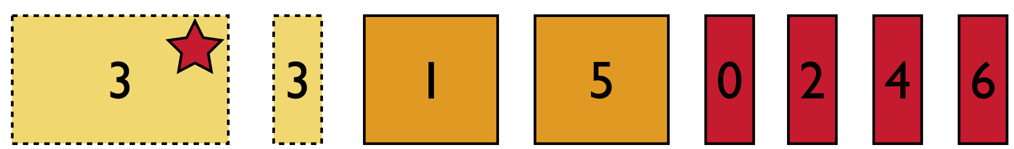




Observations

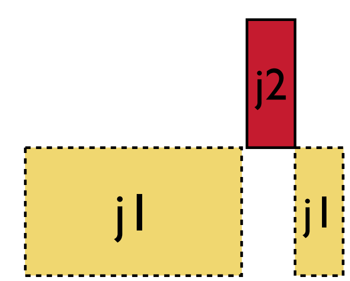


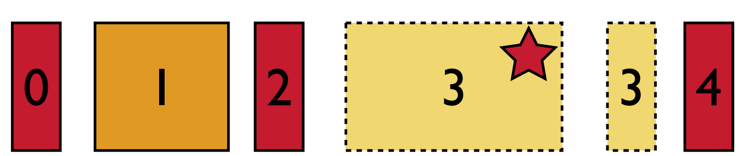
We executed jobs in the order of their priorities.



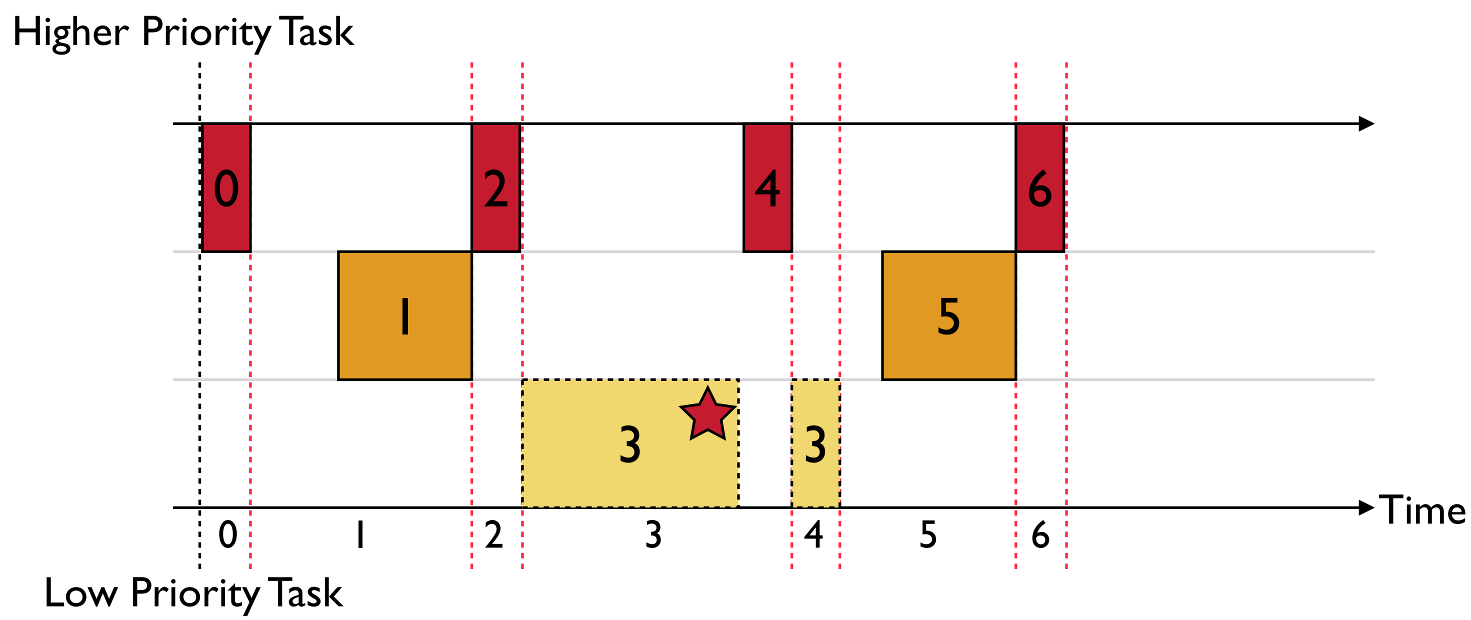
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

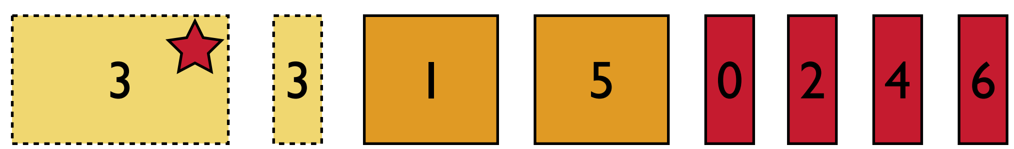




Observations

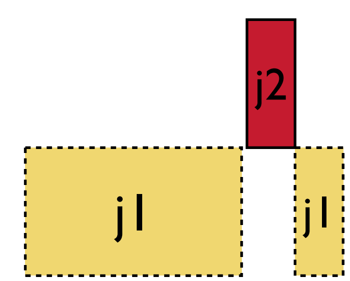


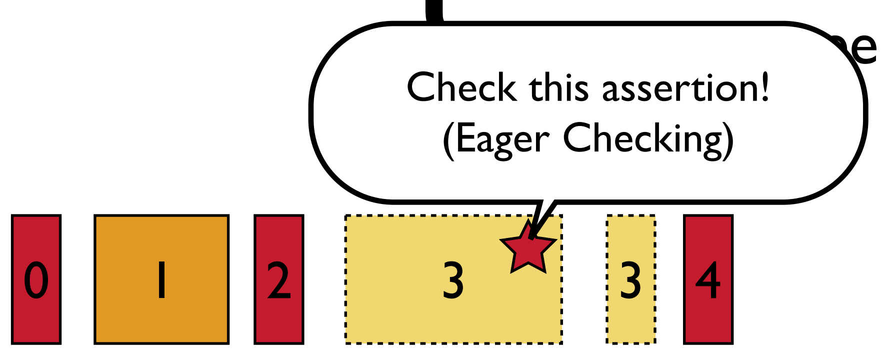
We executed jobs in the order of their priorities.



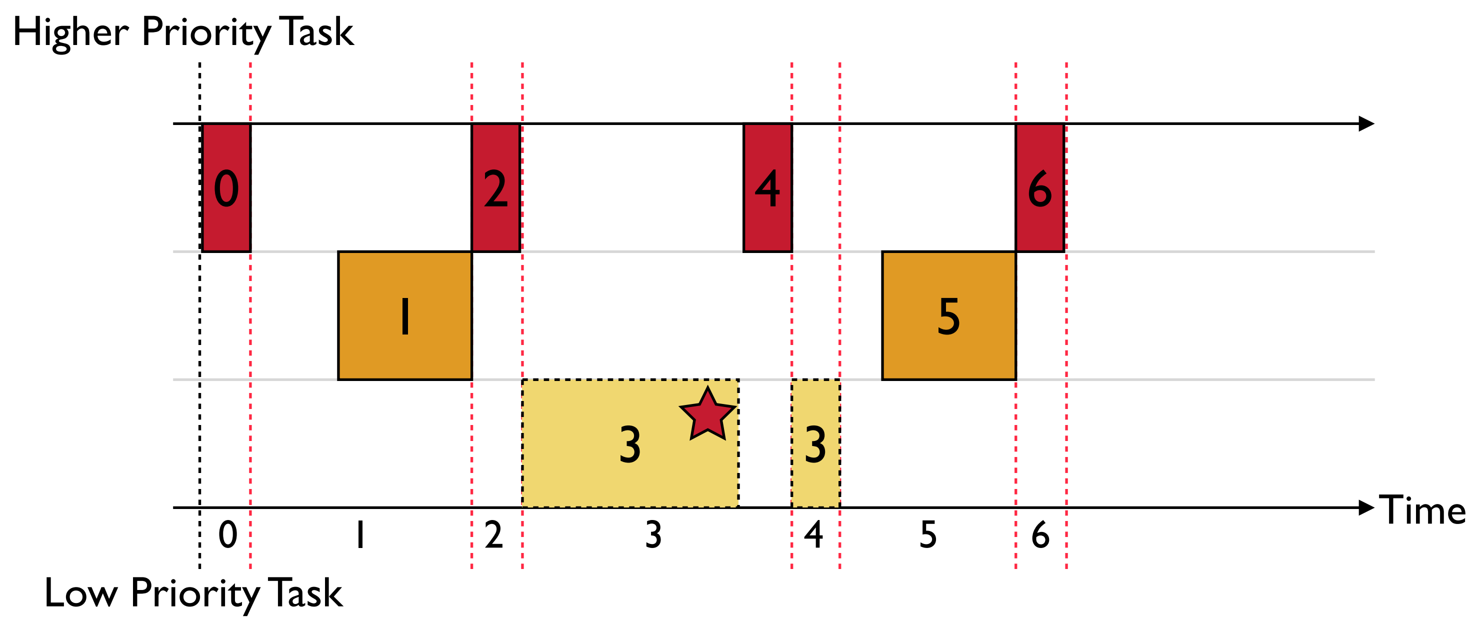
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{...empted by j2} \end{cases}$$

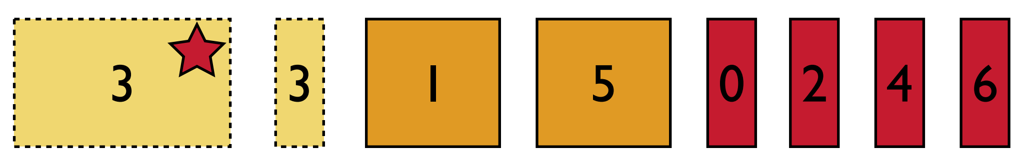




Observations

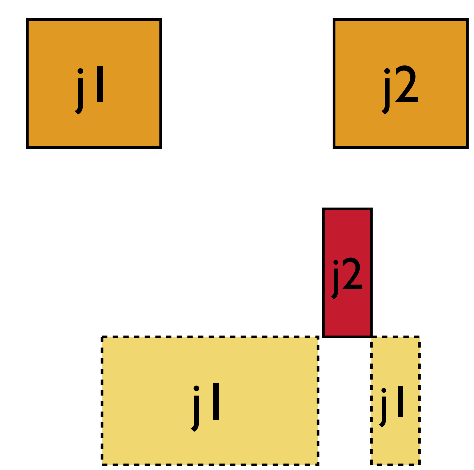


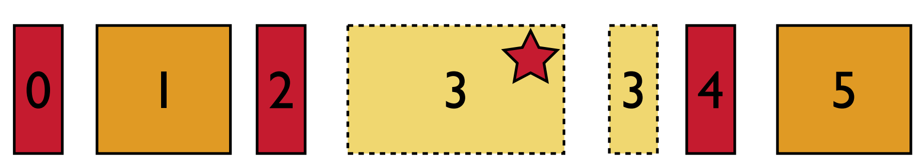
We executed jobs in the order of their priorities.



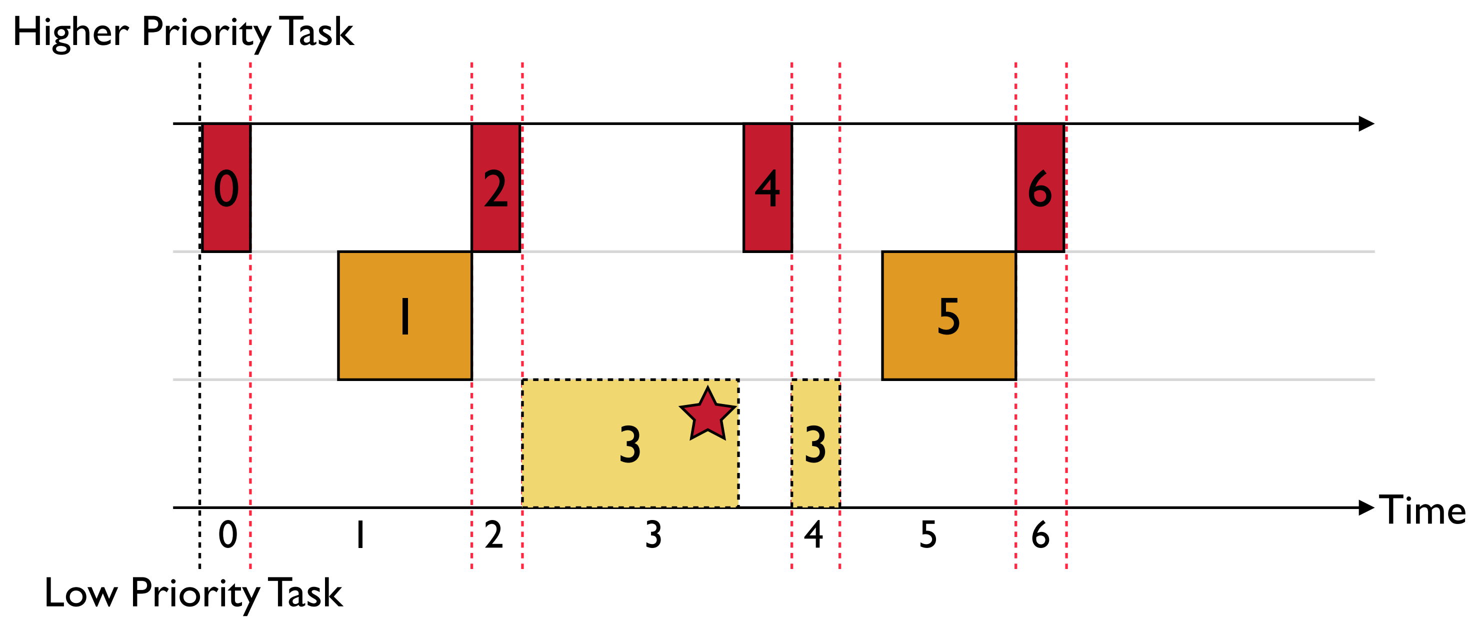
Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$

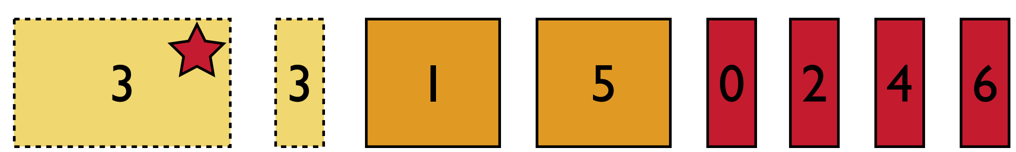




Observations

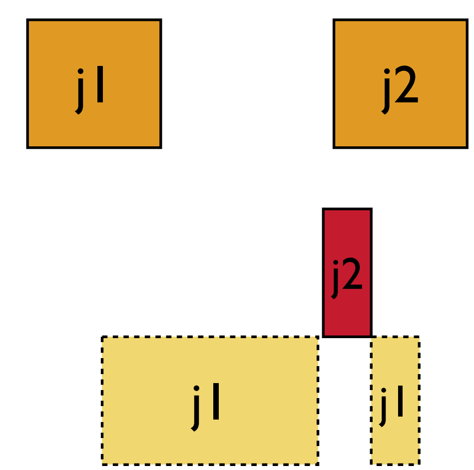


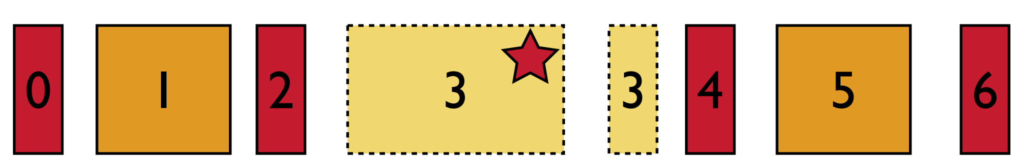
We executed jobs in the order of their priorities.



Order jobs by $\sqsubset$ relation

$$j_1 \sqsubset j_2 \begin{cases} \text{j1 ends before j2 starts} \\ \text{or} \\ \text{j1 can be preempted by j2} \end{cases}$$





Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

var $rnd, start[\,], end[\,], localAssert[\,]$
$\forall g \in \mathsf{G} \centerdot$ **var** $g[\,], v_g[\,]$

function MAIN( )
    $\forall g \in \mathsf{G} \centerdot\ g[0] := i_g$
    HYPERPERIOD()

function HYPERPERIOD( )
    SCHEDULEJOBS()
    $\forall g \in \mathsf{G} \centerdot\ \forall r \in [1, R) \centerdot$
        $v_g[r] := *; g[r] := v_g[r]$
    let the ordering of jobs by $\sqsubset$ *be*
    $j_0 \sqsubset j_1 \sqsubset \ldots j_{R-1}$
    RUNJOB(j_0); ...; RUNJOB(j_{R-1})

function SCHEDULEJOBS( )
    $\forall j \in \mathsf{J} \centerdot\ start[j] = *; end[j] = *$
    // *Jobs are sequential*
    $\forall i \in [0, N) \centerdot\ \forall k \in [0, J_i) \centerdot$ assume
        $(0 \le start[\mathsf{J}(i,k)] \le end[\mathsf{J}(i,k)] < R)$
    // *Jobs are well-separated*
    $\forall j_1 \lhd j_2 \centerdot$ assume$(end[j_1] < start[j_2])$
    $\forall j_1 \uparrow j_2 \centerdot$ assume$(start[j_1] \le start[j_2])$
    // *Jobs are well-nested*
    $\forall j_1 \uparrow j_2 \centerdot$ assume$(start[j_2] \le end[j_1]$
        $\implies (start[j_2] \le end[j_2] < end[j_1]))$

function RUNJOB(Job j)
    $localAssert[j] := 1$
    $rnd := start[j]$
    $\hat{T}(j)$
    assume$(rnd = end[j])$
    if $rnd < R - 1$ **then**
        $\forall g \in \mathsf{G} \centerdot$ assume
            $(g[rnd] = v_g[rnd + 1])$
    $X := \{j' \mid (j' = j \lor j' \uparrow j) \land$
        $(\forall j'' \neq j \centerdot\ j' \uparrow j'' \Rightarrow j'' \sqsubset j)\}$
    $\forall j' \in X \centerdot$ assert$(localAssert[j'])$

function $\hat{T}$(Job j)
    Obtained from T_t *by replacing*
    each statement 'st' with:
    CS(j) ; $st[g \leftarrow g[rnd]]$
    and each 'assert(e)' with:
    $localAssert[j] := e$

function CS(Job j)
    if $(*)$ **then return** FALSE
    $o := rnd\,; rnd := *$
    assume$(o < rnd \le end[j])$
    $\forall j' \in \mathsf{J} \centerdot\ j \uparrow j' \implies$
        assume$(rnd \le start[j'] \lor$
            $rnd > end[j'])$
    return TRUE

CompSeq

Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

```
var rnd, start[], end[], localAssert[]
∀g ∈ G . var g[], v_g[]

function MAIN( )
    ∀g ∈ G . g[0] := i_g
    HYPERPERIOD()

function HYPERPERIOD( )
    SCHEDULEJOBS()
    ∀g ∈ G . ∀r ∈ [1, R).
        v_g[r] := ∗; g[r] := v_g[r]
    let the ordering of jobs by ⊏ be
    j_0 ⊏ j_1 ⊏ ... j_{R−1}
    RUNJOB(j_0); ... ; RUNJOB(j_{R−1})

function SCHEDULEJOBS( )
    ∀j ∈ J . start[j] = ∗; end[j] = ∗
  // Jobs are sequential
    ∀i ∈ [0, N) . ∀k ∈ [0, J_i) . assume
      (0 ≤ start[J(i, k)] ≤ end[J(i, k)] < R)
  // Jobs are well-separated
    ∀j_1 ◁ j_2 . assume(end[j_1] < start[j_2])
    ∀j_1 ↑ j_2 . assume(start[j_1] ≤ start[j_2])
  // Jobs are well-nested
    ∀j_1 ↑ j_2 . assume(start[j_2] ≤ end[j_1]
      ⟹ (start[j_2] ≤ end[j_2] < end[j_1]))

function RUNJOB(Job j)
    localAssert[j] := 1
    rnd := start[j]
    T̂(j)
    assume(rnd = end[j])
    if rnd < R − 1 then
        ∀g ∈ G . assume
            (g[rnd] = v_g[rnd + 1])
    X := {j′ | (j′ = j ∨ j′ ↑ j) ∧
        (∀j″ ≠ j . j′ ↑ j″ ⇒ j″ ⊏ j)}
    ∀j′ ∈ X . assert(localAssert[j′])

function T̂(Job j)
Obtained from T_t by replacing
  each statement 'st' with:
    CS(j) ; st[g ← g[rnd]]
  and each 'assert(e)' with:
    localAssert[j] := e

function CS(Job j)
    if (∗) then return FALSE
    o := rnd ; rnd := ∗
    assume(o < rnd ≤ end[j])
    ∀j′ ∈ J . j ↑ j′ ⟹
        assume(rnd ≤ start[j′]∨
            rnd > end[j′])
    return TRUE
```

CompSeq

Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

var $rnd, start[\,], end[\,], localAssert[\,]$
$\forall g \in \mathsf{G}\,\centerdot$ **var** $g[\,], v_g[\,]$

function MAIN( )
    $\forall g \in \mathsf{G}\,\centerdot\; g[0] := i_g$
    HYPERPERIOD()

function HYPERPERIOD( )
    SCHEDULEJOBS()
    $\forall g \in \mathsf{G}\,\centerdot\; \forall r \in [1, R)\centerdot$
        $v_g[r] := *; g[r] := v_g[r]$
    let the ordering of jobs by $\sqsubset$ *be*
    $j_0 \sqsubset j_1 \sqsubset \ldots j_{R-1}$
    RUNJOB(j_0); ...; RUNJOB(j_{R-1})

function SCHEDULEJOBS( )
    $\forall j \in \mathsf{J}\,\centerdot\; start[j] = *; end[j] = *$
    // Jobs are sequential
    $\forall i \in [0, N)\,\centerdot\; \forall k \in [0, J_i)\,\centerdot$ assume
        $(0 \le start[\mathsf{J}(i,k)] \le end[\mathsf{J}(i,k)] < R)$
    // Jobs are well-separated
    $\forall j_1 \lhd j_2\,\centerdot$ assume$(end[j_1] < start[j_2])$
    $\forall j_1 \uparrow j_2\,\centerdot$ assume$(start[j_1] \le start[j_2])$
    // Jobs are well-nested
    $\forall j_1 \uparrow j_2\,\centerdot$ assume$(start[j_2] \le end[j_1]$
        $\Longrightarrow (start[j_2] \le end[j_2] < end[j_1]))$

function RUNJOB(Job j)
    $localAssert[j] := 1$
    $rnd := start[j]$
    $\hat{T}(j)$
    assume$(rnd = end[j])$
    if $rnd < R - 1$ **then**
        $\forall g \in \mathsf{G}\,\centerdot$ assume
            $(g[rnd] = v_g[rnd + 1])$
    $X := \{j' \mid (j' = j \vee j' \uparrow j) \wedge$
        $(\forall j'' \neq j\,\centerdot\; j' \uparrow j'' \Rightarrow j'' \sqsubset j)\}$
    $\forall j' \in X\,\centerdot$ assert$(localAssert[j'])$

function $\hat{T}$(Job j)
    Obtained from T_t *by replacing*
    each statement 'st' with:
    CS(j) ; $st[g \leftarrow g[rnd]]$
    and each 'assert(e)' with:
    $localAssert[j] := e$

function CS(Job j)
    if $(*)$ **then return** FALSE
    $o := rnd\,;\, rnd := *$
    assume$(o < rnd \le end[j])$
    $\forall j' \in \mathsf{J}\,\centerdot\; j \uparrow j' \implies$
        assume$(rnd \le start[j'] \vee$
            $rnd > end[j'])$
    return TRUE

CompSeq

Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

var $rnd, start[\,], end[\,], localAssert[\,]$
$\forall g \in \mathsf{G}\,.\,$ **var** $g[\,], v_g[\,]$

function MAIN( )
  $\forall g \in \mathsf{G}\,.\; g[0] := i_g$
  HYPERPERIOD()

function HYPERPERIOD( )
  SCHEDULEJOBS()
  $\forall g \in \mathsf{G}\,.\; \forall r \in [1, R)\,.$
    $v_g[r] := *; g[r] := v_g[r]$
  let the ordering of jobs by $\sqsubset$ *be*
  $j_0 \sqsubset j_1 \sqsubset \ldots j_{R-1}$
  RUNJOB(j_0); …; RUNJOB(j_{R-1})

function SCHEDULEJOBS( )
  $\forall j \in \mathsf{J}\,.\; start[j] = *; end[j] = *$
  // *Jobs are sequential*
  $\forall i \in [0, N)\,.\; \forall k \in [0, J_i)\,.$ assume
    $(0 \le start[\mathsf{J}(i,k)] \le end[\mathsf{J}(i,k)] < R)$
  // *Jobs are well-separated*
  $\forall j_1 \lhd j_2\,.$ assume$(end[j_1] < start[j_2])$
  $\forall j_1 \uparrow j_2\,.$ assume$(start[j_1] \le start[j_2])$
  // *Jobs are well-nested*
  $\forall j_1 \uparrow j_2\,.$ assume$(start[j_2] \le end[j_1]$
    $\implies (start[j_2] \le end[j_2] < end[j_1]))$

function RUNJOB(Job j)
  $localAssert[j] := 1$
  $rnd := start[j]$
  $\hat{T}(j)$
  assume$(rnd = end[j])$
  if $rnd < R - 1$ **then**
    $\forall g \in \mathsf{G}\,.$ assume
      $(g[rnd] = v_g[rnd + 1])$
  $X := \{j' \mid (j' = j \vee j' \uparrow j) \wedge$
    $(\forall j'' \ne j\,.\; j' \uparrow j'' \Rightarrow j'' \sqsubset j)\}$
  $\forall j' \in X\,.$ assert$(localAssert[j'])$

function $\hat{T}$(Job j)
  Obtained from T_t *by replacing*
  each statement 'st' with:
  CS(j) ; $st[g \leftarrow g[rnd]]$
  and each 'assert(e)' with:
  $localAssert[j] := e$

function CS(Job j)
  if $(*)$ **then return** FALSE
  $o := rnd\,;\, rnd := *$
  assume$(o < rnd \le end[j])$
  $\forall j' \in \mathsf{J}\,.\; j \uparrow j' \implies$
    assume$(rnd \le start[j'] \vee$
      $rnd > end[j'])$
  return TRUE

CompSeq

Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

var $rnd, start[\,], end[\,], localAssert[\,]$
$\forall g \in \mathsf{G}\,.$ **var** $g[\,], v_g[\,]$

function MAIN( )
  $\forall g \in \mathsf{G}\,.\ g[0] := i_g$
  HYPERPERIOD()

function HYPERPERIOD( )
  SCHEDULEJOBS()
  $\forall g \in \mathsf{G}\,.\ \forall r \in [1, R)\,.$
    $v_g[r] := *; g[r] := v_g[r]$
  let the ordering of jobs by $\sqsubset$ *be*
  $j_0 \sqsubset j_1 \sqsubset \ldots j_{R-1}$
  RUNJOB(j_0); $\ldots$; RUNJOB(j_{R-1})

function SCHEDULEJOBS( )
  $\forall j \in \mathsf{J}\,.\ start[j] = *; end[j] = *$
  // Jobs are sequential
  $\forall i \in [0, N)\,.\ \forall k \in [0, J_i)\,.$ assume
    $(0 \le start[\mathsf{J}(i,k)] \le end[\mathsf{J}(i,k)] < R)$
  // Jobs are well-separated
  $\forall j_1 \lhd j_2\,.$ assume$(end[j_1] < start[j_2])$
  $\forall j_1 \uparrow j_2\,.$ assume$(start[j_1] \le start[j_2])$
  // Jobs are well-nested
  $\forall j_1 \uparrow j_2\,.$ assume$(start[j_2] \le end[j_1]$
    $\implies (start[j_2] \le end[j_2] < end[j_1]))$

function RUNJOB(Job j)
  $localAssert[j] := 1$
  $rnd := start[j]$
  $\hat{T}(j)$
  assume$(rnd = end[j])$
  if $rnd < R - 1$ **then**
    $\forall g \in \mathsf{G}\,.$ assume
      $(g[rnd] = v_g[rnd + 1])$
  $X := \{j' \mid (j' = j \vee j' \uparrow j) \wedge$
    $(\forall j'' \neq j\,.\ j' \uparrow j'' \Rightarrow j'' \sqsubset j)\}$
  $\forall j' \in X\,.$ assert$(localAssert[j'])$

function $\hat{T}$(Job j)
Obtained from T_t *by replacing*
 each statement 'st' with:
  CS(j) ; $st[g \leftarrow g[rnd]]$
 and each 'assert(e)' with:
  $localAssert[j] := e$

function CS(Job j)
  if $(*)$ **then return** FALSE
  $o := rnd\,; rnd := *$
  assume$(o < rnd \le end[j])$
  $\forall j' \in \mathsf{J}\,.\ j \uparrow j' \implies$
    assume$(rnd \le start[j'] \vee$
      $rnd > end[j'])$
  return TRUE

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MonoSeq/CompSeq:

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R^2)$
where R= # of Jobs

Sequentialization Algorithms

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MonoSeq/CompSeq:

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R^2)$
where R= # of Jobs

HarmonicSeq: Only for **Harmonic** Periodic Program

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

Sequentialization Algorithms

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MonoSeq/CompSeq:

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R^2)$
where R= # of Jobs

HarmonicSeq: Only for **Harmonic** Periodic Program

1. Construct a **non**[...]tialized program
2. Enforce legal job s[...]
infeasible thread e[...]**straints**

$P_i \le P_j \implies P_i | P_j$

Sequentialization Algorithms

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MonoSeq/CompSeq:

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R^2)$
where R= # of Jobs

HarmonicSeq: Only for **Harmonic** Periodic Program

1. Construct a **non**...
2. Enforce legal job s...
infeasible thread e...

Common in Real-time Embedded Systems:

1. 100% CPU Utilization
2. More predictable battery usage

Sequentialization Algorithms

Naive Approach:

1. Enumerate all possible (sequentialized) executions
2. Verify each of them

Exponential Blow-up!

MonoSeq/CompSeq:

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R^2)$
where R= # of Jobs

HarmonicSeq: Only for **Harmonic** Periodic Program

1. Construct a **non-deterministic** sequentialized program
2. Enforce legal job scheduling and prune out infeasible thread executions by adding **constraints**

$O(R \cdot N)$
where N= # of tasks, usually exponentially smaller than R

Sequentialization Algorithms

Algorithm 2 Procedure to assign legal starting and ending rounds to jobs in a harmonic program.

var $min[\,], max[\,]$ *//extra variables*

function scheduleHarmonic( )

$\quad \forall j \in \mathsf{J} \,.\; start[j] = *; end[j] = *; min[j] = *; max[j] = *$

// Correctness of min and max

$\quad \forall n \in \mathcal{T} \,.\; isleaf(n) \implies \text{assume}(min[n] = start[n] \wedge max[n] = end[n])$

$\quad \forall n \in \mathcal{T} \,.\; \neg isleaf(n) \implies \text{assume}(min[n] = \text{MIN}(start[n], min[first(n)]))$

$\quad \forall n \in \mathcal{T} \,.\; \neg isleaf(n) \implies \text{assume}(max[n] = \text{MAX}(end[n], max[last(n)]))$

// Jobs are sequential

$\quad \forall n \in \mathcal{T} \,.\; \text{assume}(low(n) \leq start[n] \leq end[n] \leq high(n))$

// Jobs are well-separated

$\quad \forall n \in \mathcal{T} \,.\; hasNext(n) \implies \text{assume}(max[n] < min[next(n)])$

$\quad \forall j_1 \uparrow j_2 \,.\; \text{assume}(start[j_1] \leq start[j_2])$

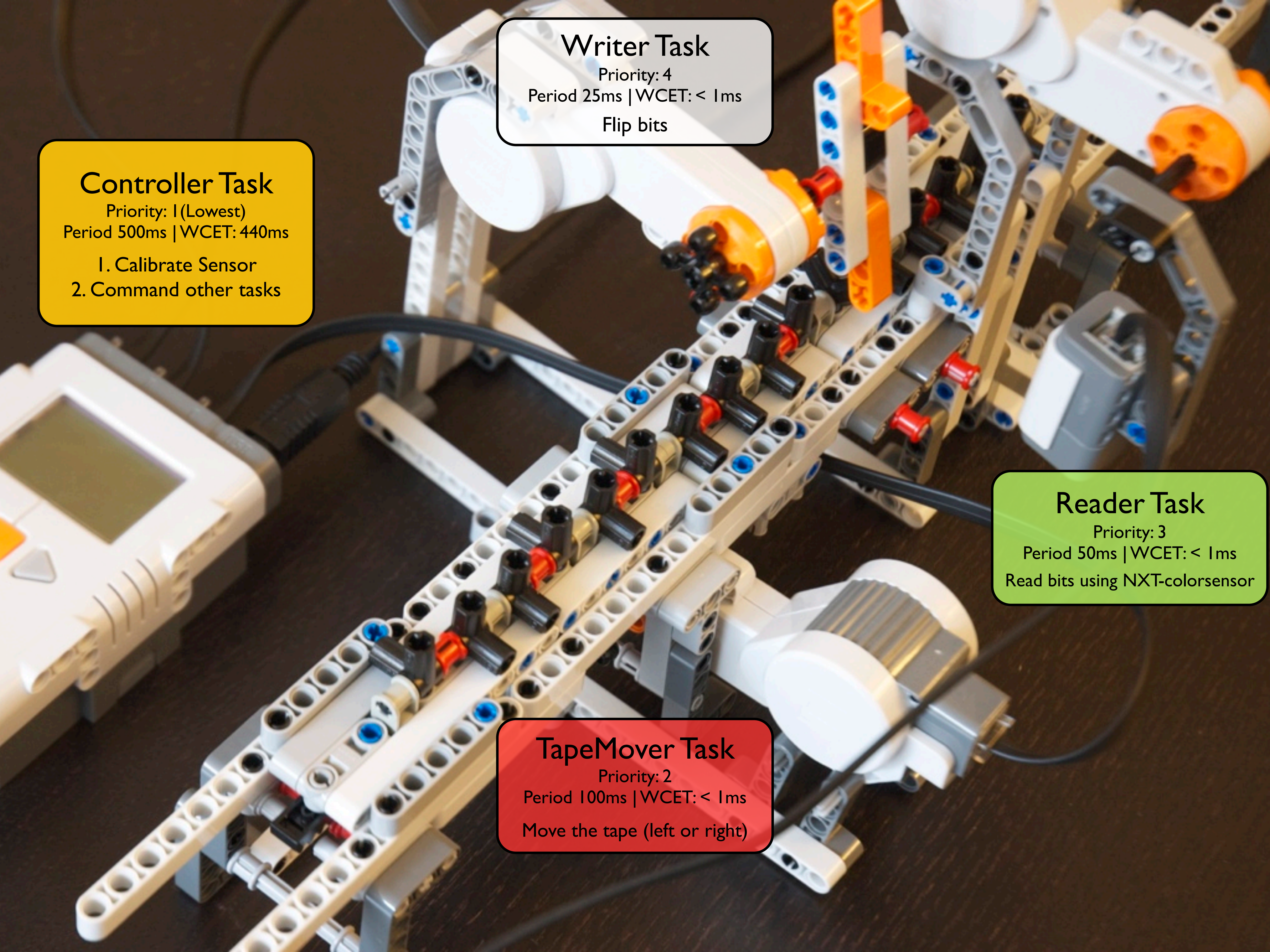
// Jobs are well-nested

$\quad \forall j_1 \uparrow j_2 \,.\; \text{assume}(start[j_2] \leq end[j_1] \implies (start[j_2] \leq end[j_2] < end[j_1]))$

$\mathcal{T}(n)$	= sub-tree of $\mathcal{T}$ rooted at n	$isleaf(n)$	= true iff n is a leaf node
$level(n)$	= level of node n	$size(n)$	= number of nodes in $\mathcal{T}(n)$
$id(n)$	= position of n in the DFS order of $\mathcal{T}$	$hasNext(n)$	= true iff n is *not* the last node at level $level(n)$
$next(n)$	= node after n at level $level(n)$	$first(n)$	= first child of n
$last(n)$	= last child of n	$maxid(n)$	$= id(n) + size(n) - 1$
$low(n)$	$= id(n) - level(n)$	$high(n)$	$= maxid(n)$

Case Study:

Concurrent Turing Machine



Writer Task
Priority: 4
Period 25ms | WCET: < 1ms
Flip bits
Controller Task
Priority: 1 (Lowest)
Period 500ms | WCET: 440ms
1. Calibrate Sensor
2. Command other tasks
Reader Task
Priority: 3
Period 50ms | WCET: < 1ms
Read bits using NXT-colorsensor
TapeMover Task
Priority: 2
Period 100ms | WCET: < 1ms
Move the tape (left or right)

Properties

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bit */
  if(R(input) != R(output)) {
    /* Check the header and move it back if necessary */
    if(nxt_motor_get_count(READ_MOTOR) > 0 && R(R_state) == READ_IDLE) {
      W(R_state, READ_MOVE_HEADER_BACKWARD);
    }

    /* Check the header and flip the bit if it is safe to do */
    if(nxt_motor_get_count(READ_MOTOR) <= 0 && R(W_state) == WRITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for writer */
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

Writer Task

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer flips a bit, the tape motor and read
       motor should be stopped. */

    /* FAILED!! with BOUND 120 */
    assert(R(T_speed) == 0 && R(R_speed) == 0);
#endif
```

Properties

Property 2: When writer flips a bit, the tap … hould **stop**.

If the READ header is up,
Move it back
to avoid collision!

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bi
  if(R(input) != R(output)) {
```

Properties

Property 2: When writer flips a bit, the tap … hould **stop**.

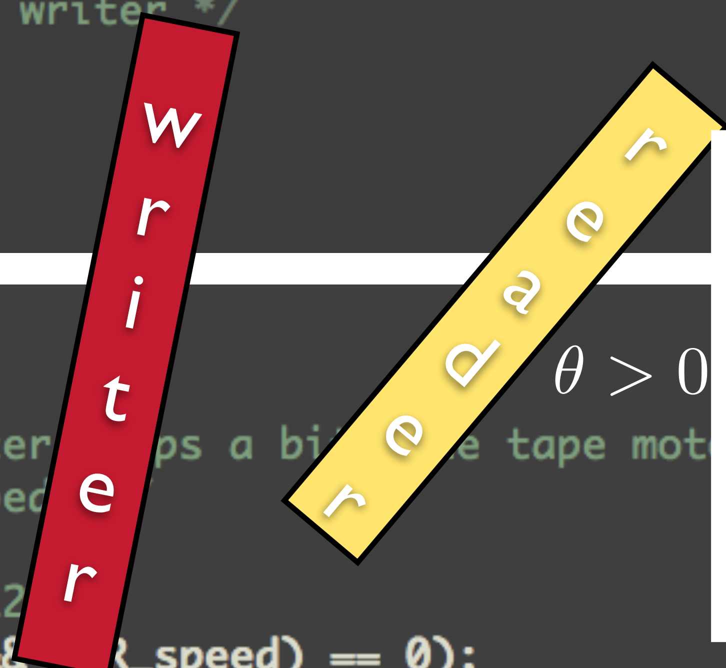
If the READ header is up,
Move it back
to avoid collision!

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bi
  if(R(input) != R(output)) {
    /* Check the header and move it back if necessary */
    if(nxt_motor_get_count(READ_MOTOR) > 0 && R(R_state) == READ_IDLE) {
      W(R_state, READ_MOVE_HEADER_BACKWARD);
    }

    /* Check the header and flip the bit if it is safe to do */
    if(nxt_motor_get_count(READ_MOTOR) <= 0 && R(W_state) == WRITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for write */
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

Writer Task

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer    ps a bi    e tape mot r and read
       motor should be stopped

    /* FAILED!! with BOUND 12
    assert(R(T_speed) == 0 &&     _speed) == 0);
#endif
```



Proper...

Property 2: When writer flips a bit, the tap... ...hould **stop**.

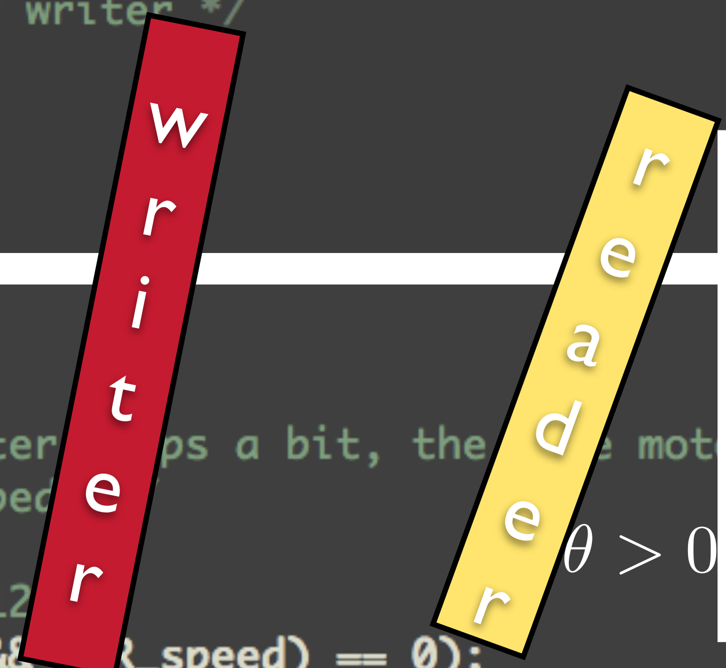
If the READ header is up,
Move it back
to avoid collision!

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bi
  if(R(input) != R(output)) {
    /* Check the header and move it back if necessary */
    if(nxt_motor_get_count(READ_MOTOR) > 0 && R(R_state) == READ_IDLE) {
      W(R_state, READ_MOVE_HEADER_BACKWARD);
    }

    /* Check the header and flip the bit if it is safe to do */
    if(nxt_motor_get_count(READ_MOTOR) <= 0 && R(W_state) == WRITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for write
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

Writer Task

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer   ps a bit, the   e mot r and read
       motor should be stopped

    /* FAILED!! with BOUND 12
    assert(R(T_speed) == 0 &&   _speed) == 0);
#endif
```



Properties

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bit
  if(R(input) != R(output)) {
    /* Check the header and move it bac
    if(nxt_motor_get_count(READ_MOTOR)                    ) {
      W(R_state, READ_MOVE_HEADER_BACKW
    }

    /* Check the header and flip the bit if it is safe to do */
    if(nxt_motor_get_count(READ_MOTOR) <= 0 && R(W_state) == WRITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for writer */
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

OK, it's safe to write!

Writer Task

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer   ps a bit, the tape mot   r   read
       motor should be stopped

    /* FAILED!! with BOUND 12
    assert(R(T_speed) == 0 &&      _speed) == 0);
#endif
```

writer

reader

$\theta <= 0$

Properties

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bit */
  if(R(input) != R(output)) {
    /* Check the header and move i      ck if necessary */
    if(nxt_motor_get_count(READ_M     ) > 0 && R(R_state) == READ_IDLE) {
      W(R_state, READ_MOVE_HEADER     KWARD);
    }

    /* Check the header and fli     e bit if it is safe to     /
    if(nxt_motor_get_count(READ     OR) <= 0 && R(W_state) =     ITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for w     er */
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

$\theta <= 0$

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer flips a
       motor should be stopped. */

    /* FAILED!! with BOUND 120 */
    assert(R(T_speed) == 0 && R(R_speed) == 0);
#endif
```

NO!
The position of READ header is in safe area (<=0), however it's possible that it is **still moving**!

Properties

Property 2: When writer flips a bit, the tape motor and read motor should **stop**.

Controller Task

```
case C_WRITE:

  /* Check if we need to chagne the bit */
  if(R(input) != R(output)) {
    /* Check the header and move i   ck if necessary */
    if(nxt_motor_get_count(READ_M   ) > 0 && R(R_state) == READ_IDLE) {
      W(R_state, READ_MOVE_HEADER   KWARD);
    }

    /* Check the header and fli   e bit if it is safe to   /
    if(nxt_motor_get_count(READ   OR) <= 0 && R(W_state) =   ITE_IDLE) {
      W(W_state, WRITE_FLIP);
    }
  } else {
    /* Nothing to change for w   er */
    W(W_state, WRITE_IDLE);
    C_state = C_MOVE;
  }
  break;
```

writer

reader

$\theta <= 0$

```
  case WRITE_FLIP:
#ifdef VERIFICATION
    /* Property 3: When writer flips a b
       motor should be stopped. */

    /* FAILED!! with BOUND 120 */
    assert(R(T_speed) == 0 && R(R_speed) == 0);
#endif
```

REKH(out tool) can find a counterexample within 2mins.

Experimental Results

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

Name	OL	monoSeq SL	monoSeq SAT Size GL	monoSeq SAT Size Var	monoSeq SAT Size Clause	monoSeq S	monoSeq Time (sec)	harmonicSeq SL	harmonicSeq SAT Size GL	harmonicSeq SAT Size Var	harmonicSeq SAT Size Clause	harmonicSeq S	harmonicSeq Time (sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		MONOSEQ						HARMONICSEQ					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – ‘Y’ for SAFE, ‘N’ for UNSAFE, and ‘U’ for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

Name	OL	monoSeq SL	monoSeq SAT Size GL	monoSeq SAT Size Var	monoSeq SAT Size Clause	monoSeq S	monoSeq Time (sec)	harmonicSeq SL	harmonicSeq SAT Size GL	harmonicSeq SAT Size Var	harmonicSeq SAT Size Clause	harmonicSeq S	harmonicSeq Time (sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		MONOSEQ						HARMONICSEQ					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name	OL	SL	SAT Size GL	SAT Size Var	SAT Size Clause	S	Time (sec)	SL	SAT Size GL	SAT Size Var	SAT Size Clause	S	Time (sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	354K	N	4.36
nxt.ok2	388	2,215	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	356K	Y	4.69
nxt.bug2	405	2,389	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	372K	N	5.81
nxt.ok3	405	2,389	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	358K	Y	5.71
aso.bug1	421	2,557	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	568K	N	6.67
aso.bug2	421	2,627	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

MonoSeq considers **infeasible** thread executions and declares the program unsafe (False Alarm)

Table 2. Experimental results of concurrent Turing Machine. H = # of hyper-periods, OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (85,000s); Time = verification time in sec.

Name	H	OL	monoSeq SL	monoSeq SAT Size GL	monoSeq SAT Size Var	monoSeq SAT Size Clause	monoSeq S	monoSeq Time (sec)	harmonicSeq SL	harmonicSeq SAT Size GL	harmonicSeq SAT Size Var	harmonicSeq SAT Size Clause	harmonicSeq S	harmonicSeq Time (sec)
ctm.ok1	4	613	13K	121K	2,737K	8,774K	Y	44,781	7K	111K	1,063K	3,497K	Y	93.39
ctm.ok2	4	610	13K	119K	2,728K	8,738K	Y	21,804	7K	109K	1,055K	3,467K	Y	87.60
ctm.bug2	4	611	13K	118K	2,707K	8,674K	N	2,281	7K	108K	1,047K	3,441K	N	86.18
ctm.ok3	6	612	20K	222K	6,276K	20,163K	U	—	7K	171K	1,667K	5,566K	Y	243.76
ctm.bug3	6	612	20K	214K	5,914K	19,044K	N	84,625	7K	165K	1,609K	5,383K	N	248.65
ctm.ok4	8	613	29K	333K	10,390K	33,550K	U	—	7K	222K	2,178K	7,417K	Y	534.38

Table 2. Experimental results of concurrent Turing Machine. H = # of hyper-periods, OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (85,000s); Time = verification time in sec.

			monoSeq						harmonicSeq					
Name	H	OL	SL	SAT Size GL	Var	Clause	S	Time (sec)	SL	SAT Size GL	Var	Clause	S	Time (sec)
ctm.ok1	4	613	13K	121K	2,737K	8,774K	Y	44,781	7K	111K	1,063K	3,497K	Y	93.39
ctm.ok2	4	610	13K	119K	2,728K	8,738K	Y	21,804	7K	109K	1,055K	3,467K	Y	87.60
ctm.bug2	4	611	13K	118K	2,707K	8,674K	N	2,281	7K	108K	1,047K	3,441K	N	86.18
ctm.ok3	6	612	20K	222K	6,276K	20,163K	U	—	7K	171K	1,667K	5,566K	Y	243.76
ctm.bug3	6	612	20K	214K	5,914K	19,044K	N	84,625	7K	165K	1,609K	5,383K	N	248.65
ctm.ok4	8	613	29K	333K	10,390K	33,550K	U	—	7K	222K	2,178K	7,417K	Y	534.38

Table 2. Experimental results of concurrent Turing Machine. H = # of hyper-periods, OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (85,000s); Time = verification time in sec.

			monoSeq						harmonicSeq					
Name				SAT Size			S	Time		SAT Size			S	Time
	H	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
ctm.ok1	4	613	13K	121K	2,737K	8,774K	Y	44,781	7K	111K	1,063K	3,497K	Y	93.39
ctm.ok2	4	610	13K	119K	2,728K	8,738K	Y	21,804	7K	109K	1,055K	3,467K	Y	87.60
ctm.bug2	4	611	13K	118K	2,707K	8,674K	N	2,281	7K	108K	1,047K	3,441K	N	86.18
ctm.ok3	6	612	20K	222K	6,276K	20,163K	U	—	7K	171K	1,667K	5,566K	Y	243.76
ctm.bug3	6	612	20K	214K	5,914K	19,044K	N	84,625	7K	165K	1,609K	5,383K	N	248.65
ctm.ok4	8	613	29K	333K	10,390K	33,550K	U	—	7K	222K	2,178K	7,417K	Y	534.38

480x Faster!

Thank you