Towards Run-Time Verification for Real-Time Safety-Critical Systems

Geoffrey Nelissen

Systems Do not Get Simpler

- Every day, computing platforms become more and more complex
  - Unicore → Multicore → Manycore
  - Apparition of buses, then NoCs
  - Shared memory and shared caches
  - Complex cache replacement policies
  - …
- Software becomes more complex too
  - Parallel and/or distributed tasks
  - Execution affinities (➔ migrations)
  - …

➔Difficult to model, analyse and verify

Systems Do not Get Simpler

- Yet, safety critical systems would like to adopt those new architectures
  - For their performances
  - To reduce costs
  - For availability reasons
  - …

➔ One must prove that all the system requirements are respected

Two Types of Properties to Verify

- Functional properties
- Extra-functional properties

Two Types of Properties to Verify

- Functional properties
  - Everything that relate to the result produced or the order of execution
  - Examples:
    - A must execute before B
    - If A executes than B must eventually execute
    - C cannot execute between A and B
    - The result returned by A must be positive
    - The sensor readings cannot be smaller than 3
- Extra-functional properties

Two Types of Properties to Verify

- Functional properties
- Extra-functional properties

Two Types of Properties to Verify

- Functional properties
- Extra-functional properties
  - Everything that does not relate to the result produced or the order of execution
  - Examples:
    - A must complete within 10ms
    - A cannot execute for more than 5ms
    - B must execute at last 10ms after A
    - Core temperature must remain under 60°C
    - Power consumption must remain under 5W
  - In this work, we limit ourselves to timing properties

Static Verification is Usually Impractical

- For functional properties
  - Time and complexity to actually prove something
  - Explosion of possible states
  - Theoretical limitations of the models
- For extra-functional properties
  - most data are available only at run-time

Modeling and Analysis are Based on Assumptions

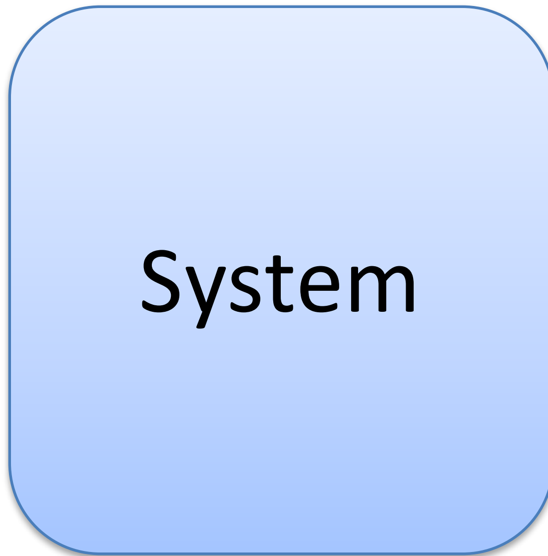
- For instance, in real-time scheduling theory:
  - Execution time never exceeds the WCET
  - The minimum inter-arrival time is lower bounded by T_i
  - The jitter is upper bounded by J_i
  - …

➔ Nothing proves that they are actually respected at run-time

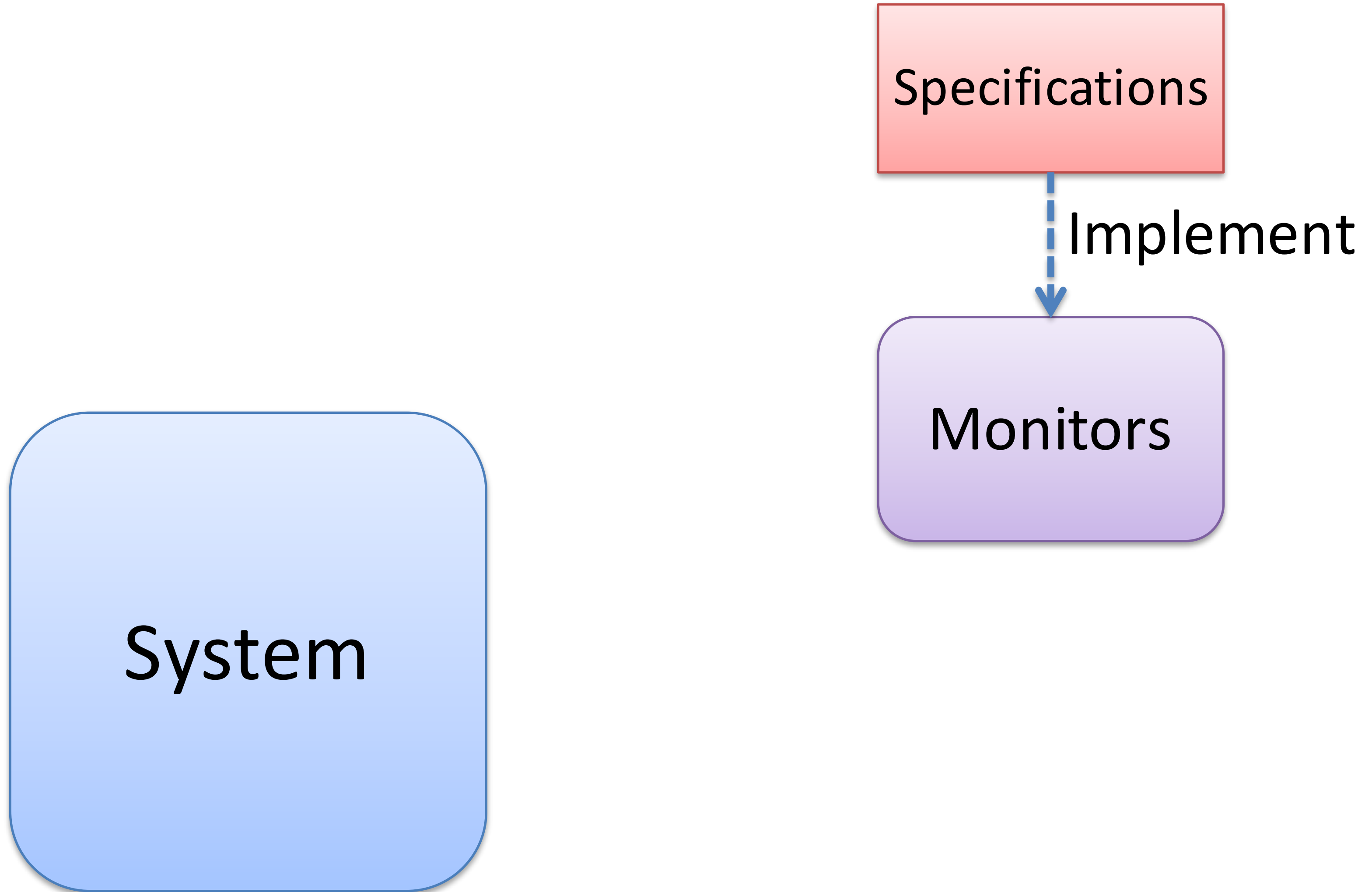
➔ Testing is used to increase the confidence but does not cover all cases

RUN-TIME VERIFICATION: BASIC CONCEPTS

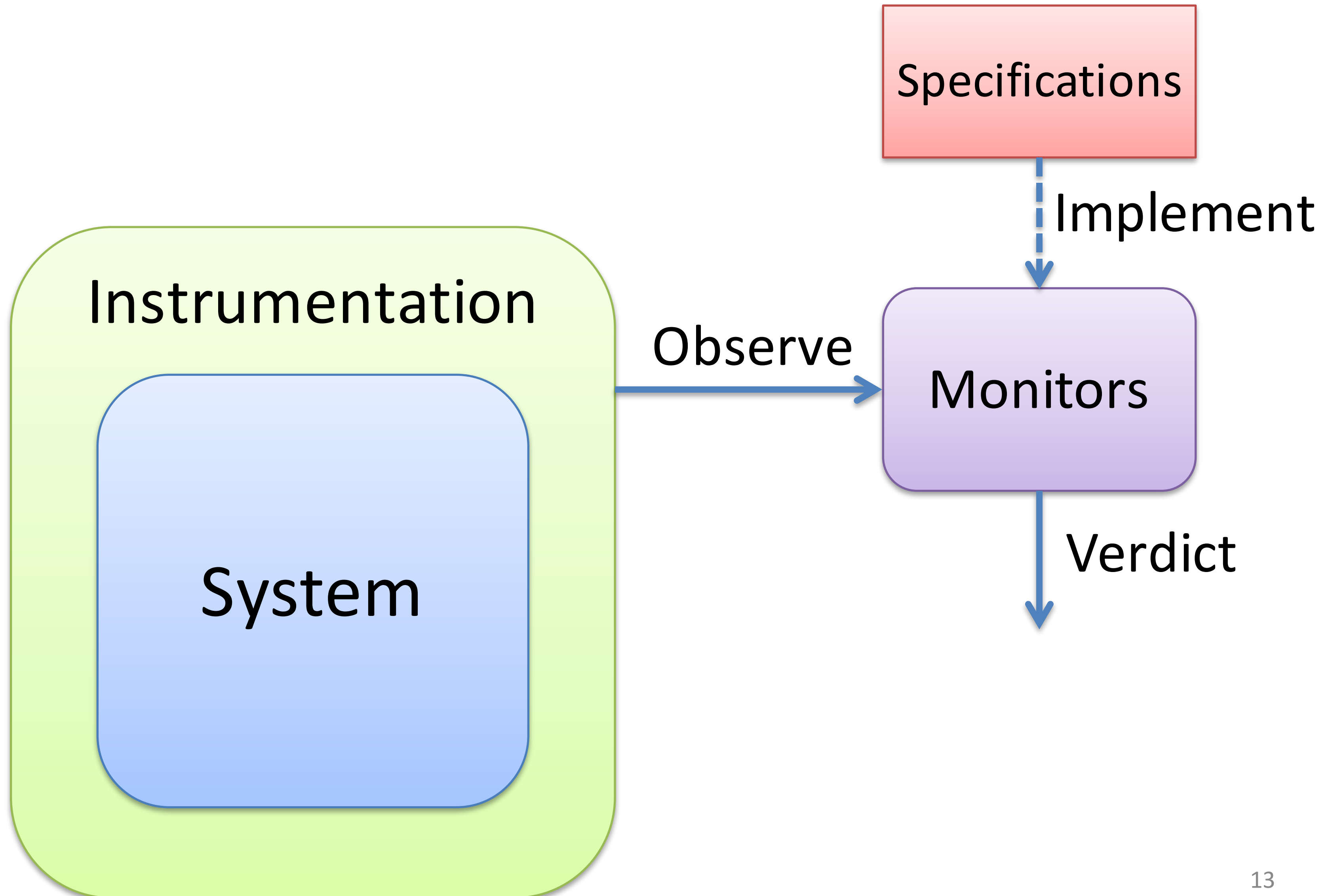
Run-time verification



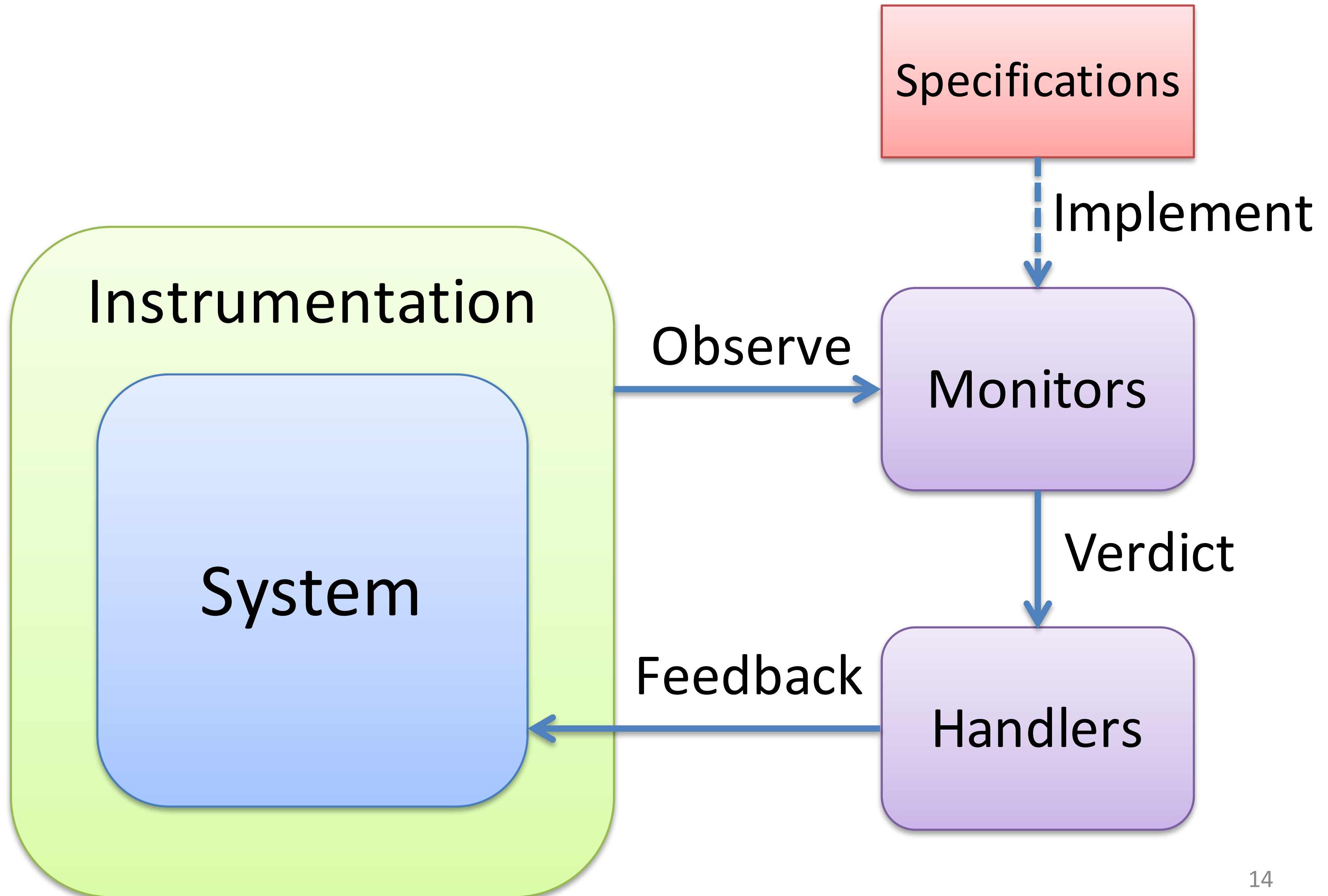
Run-time verification



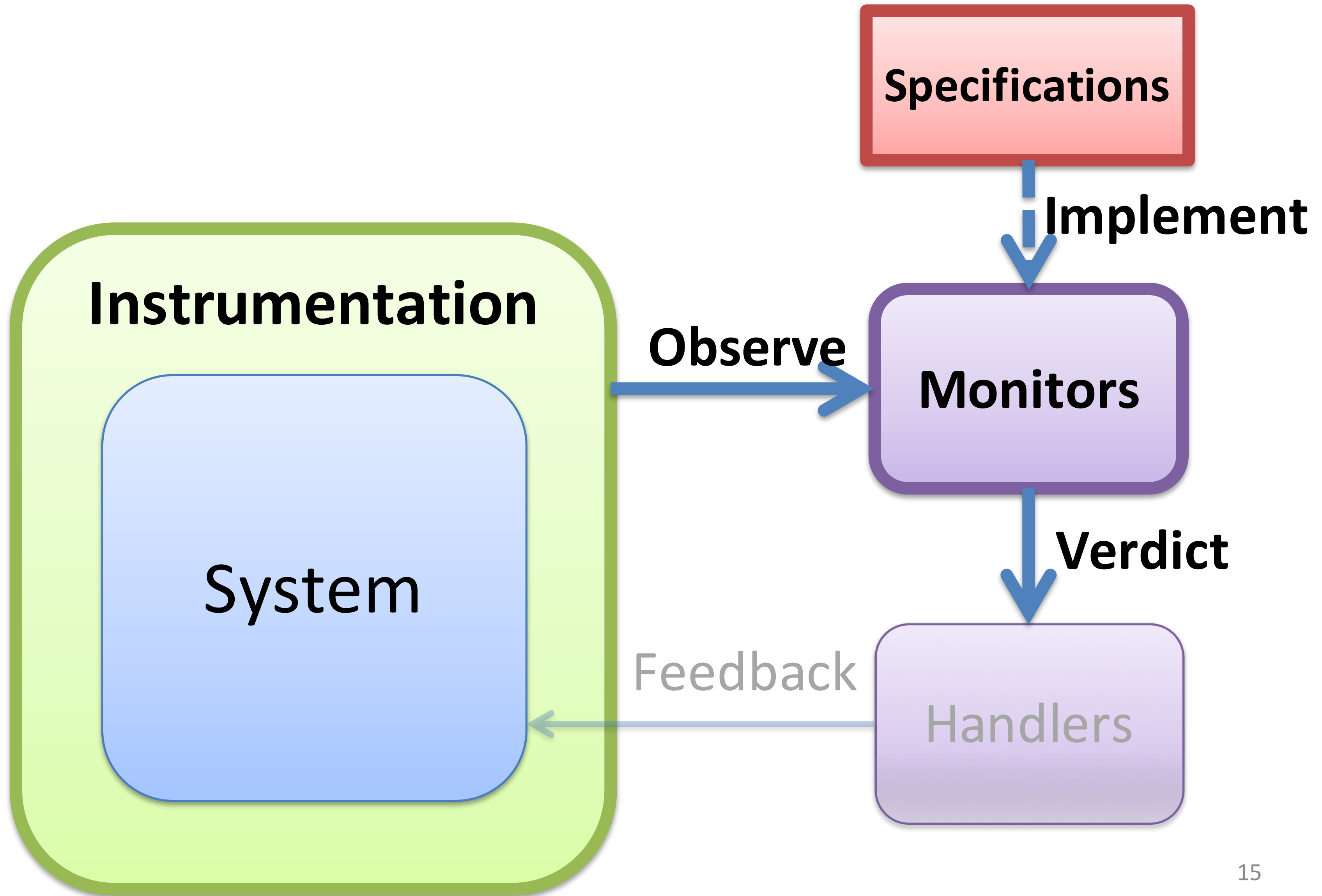
Run-time verification



Run-time verification



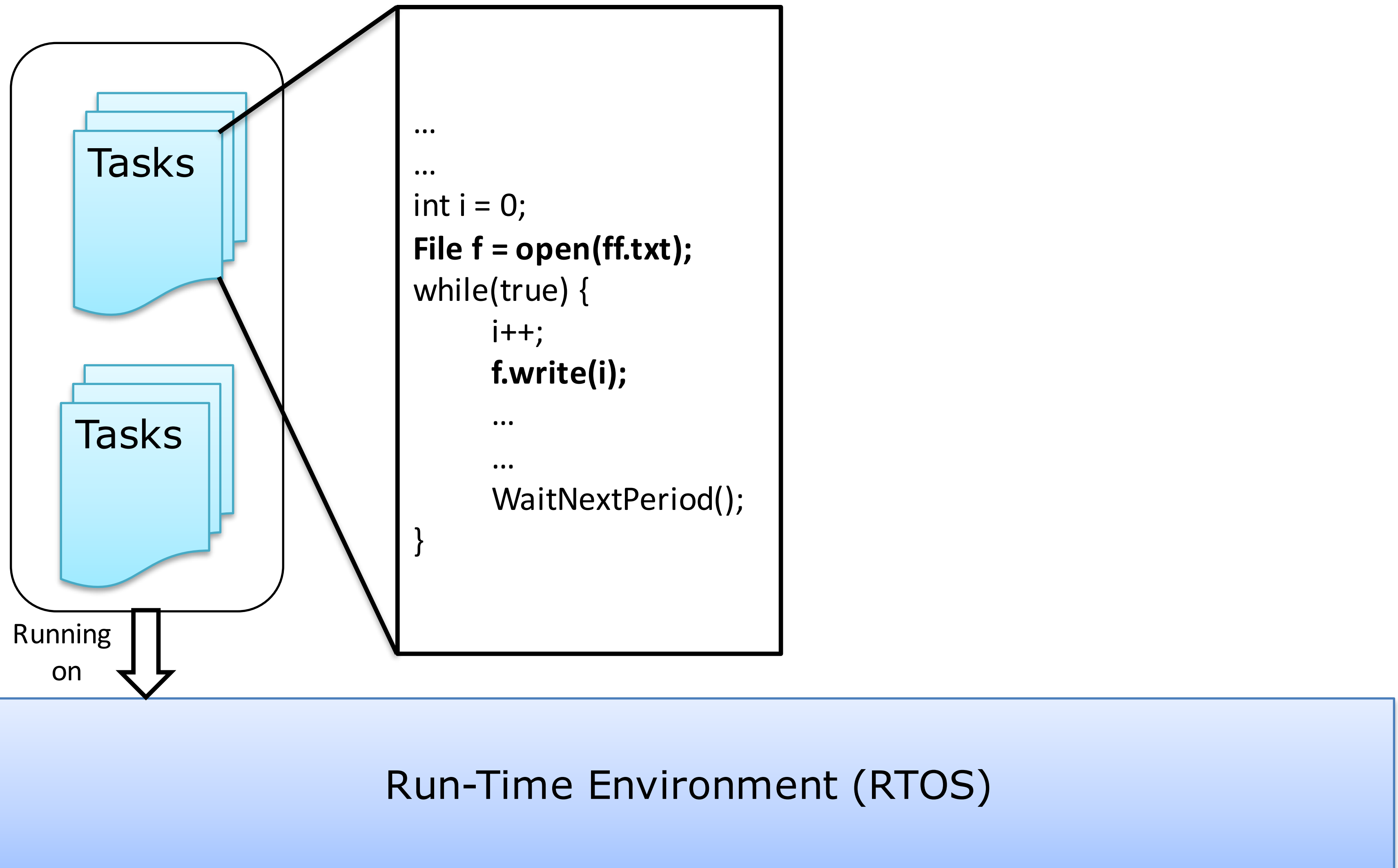
Run-time verification



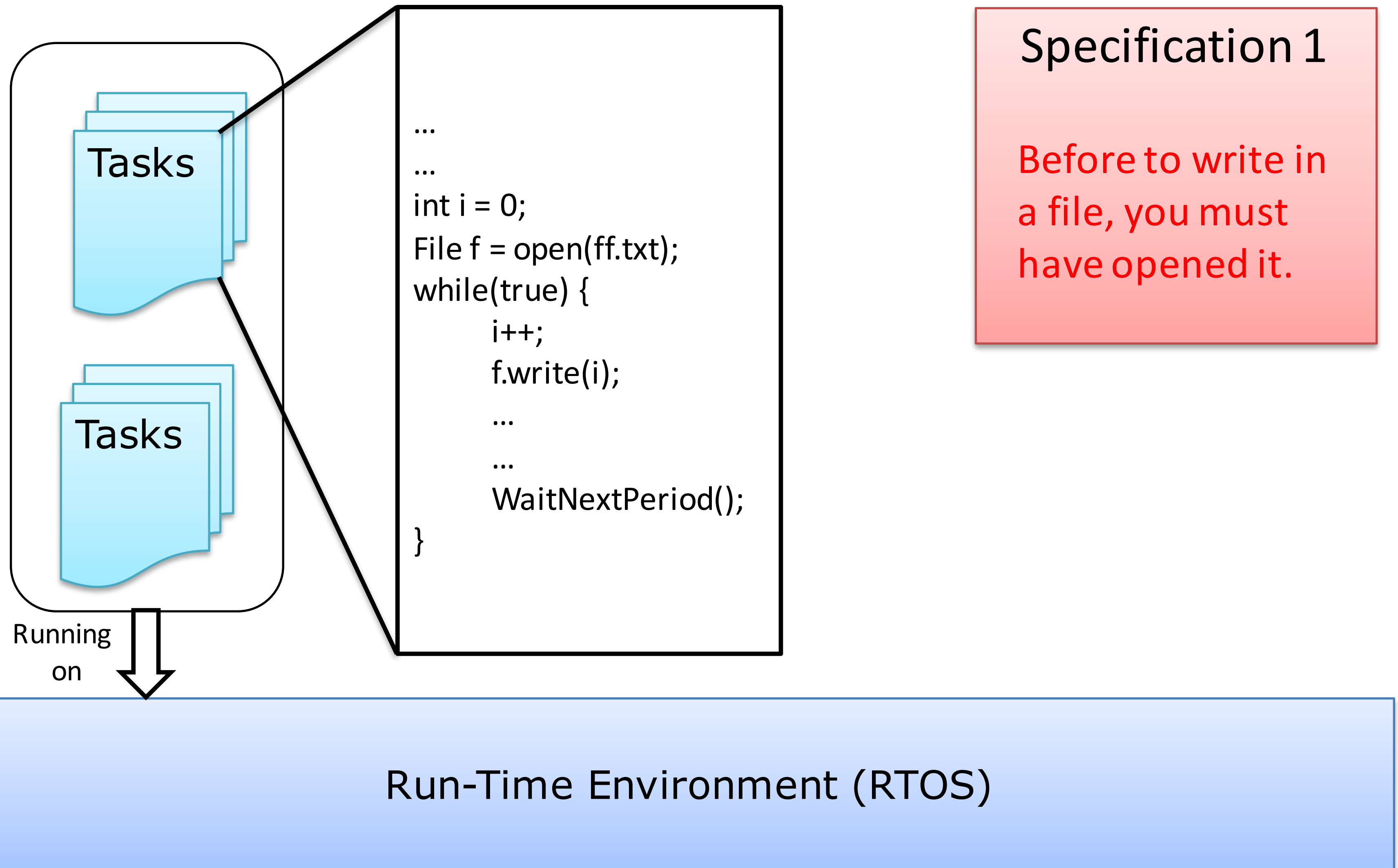
EXAMPLES

Verification of a Functional Property

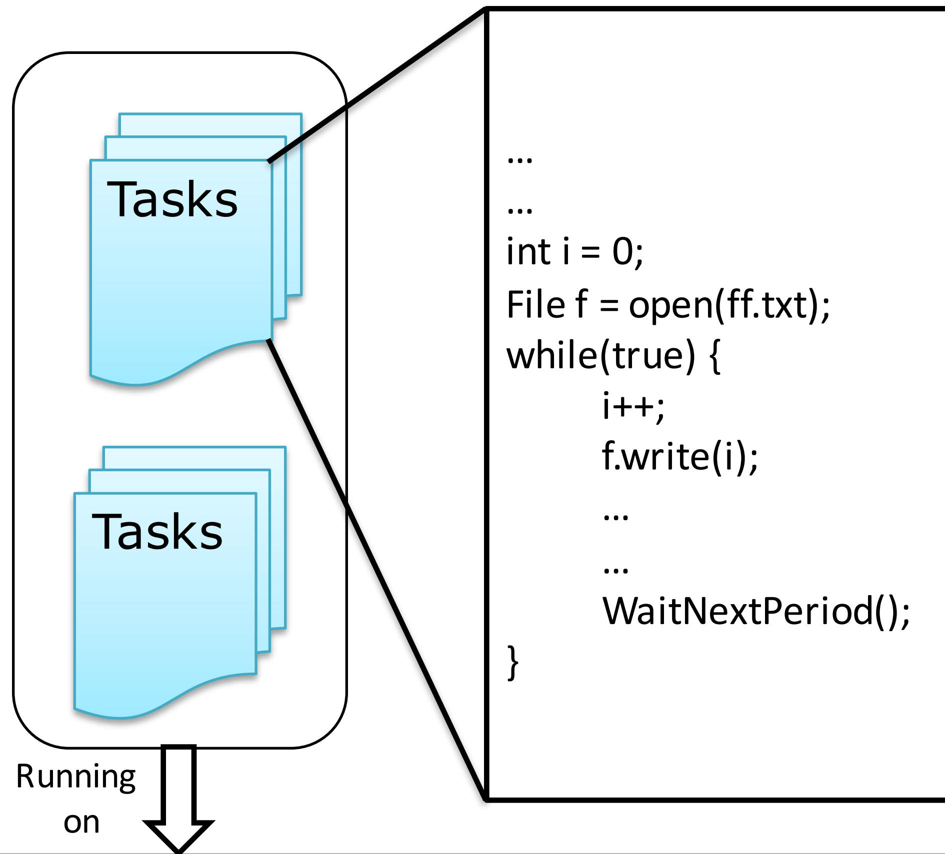
Verification of a Functional Property

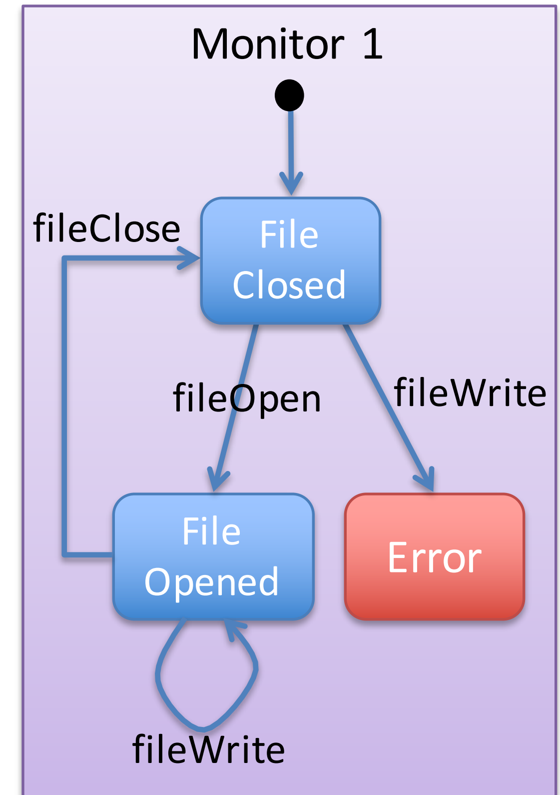


Verification of a Functional Property



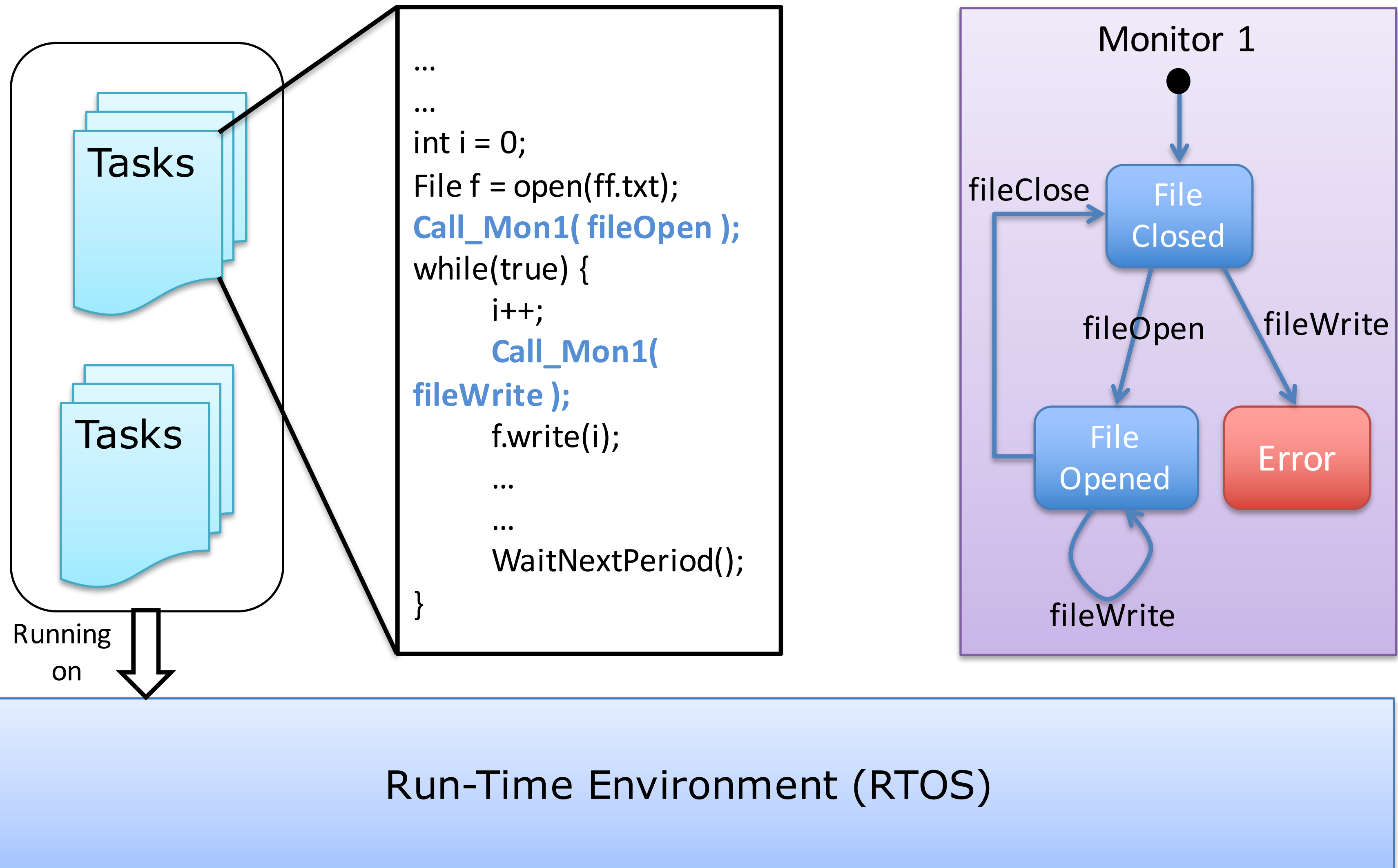
Verification of a Functional Property



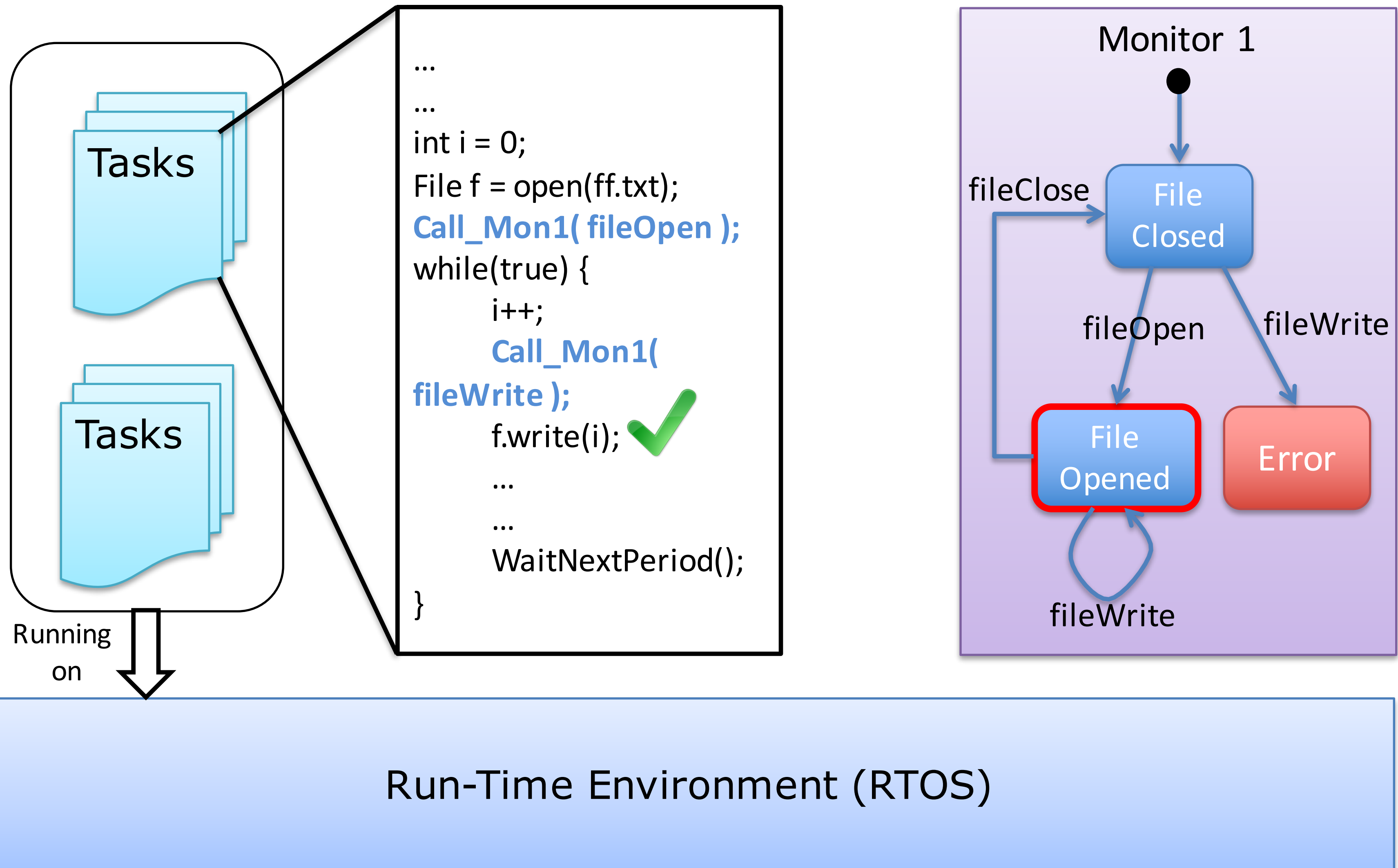


Run-Time Environment (RTOS)

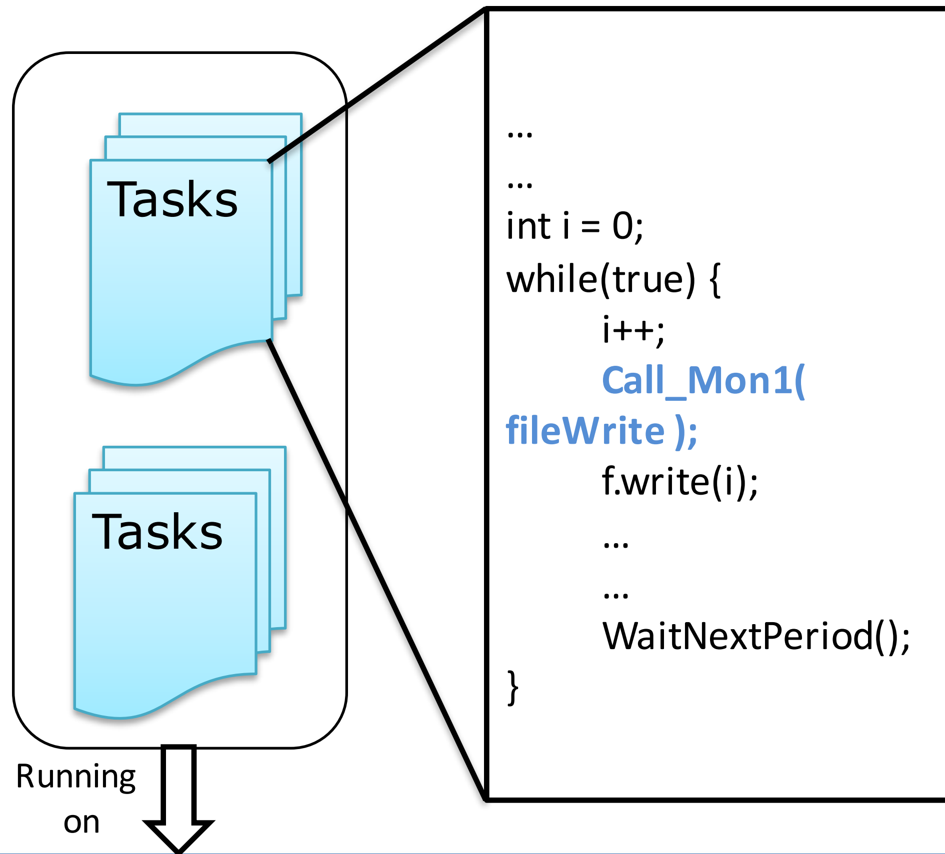
Verification of a Functional Property

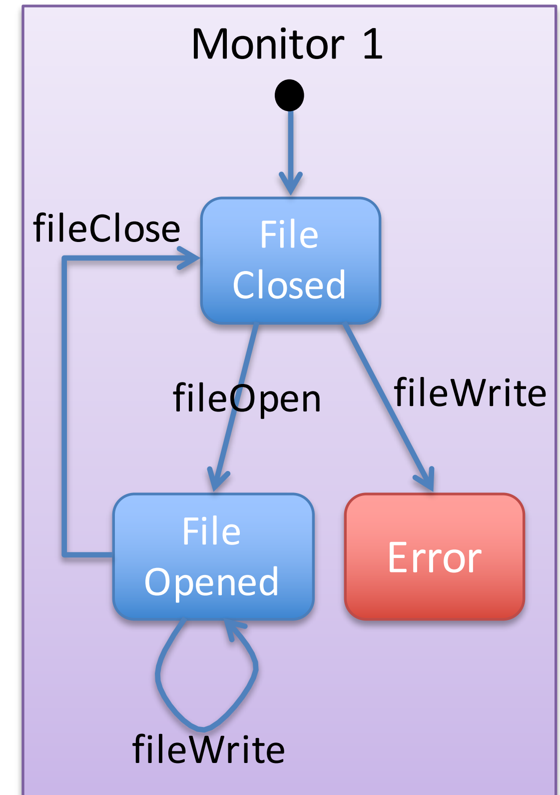


Verification of a Functional Property



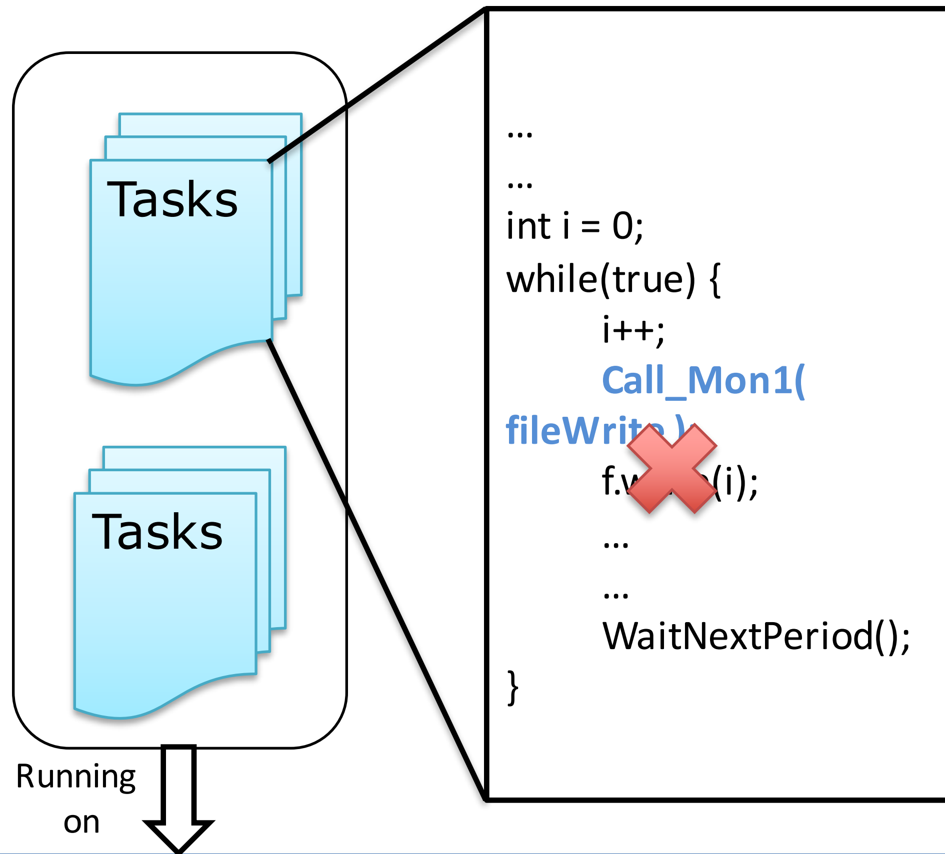
Verification of a Functional Property

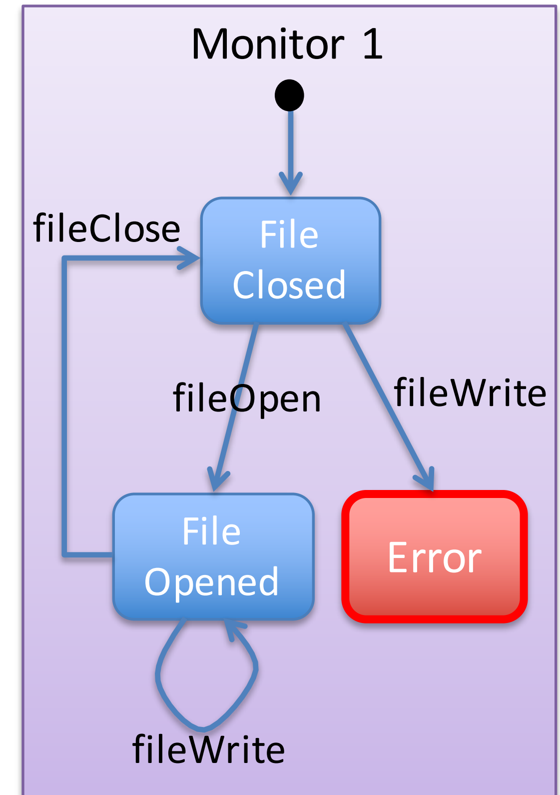




Run-Time Environment (RTOS)

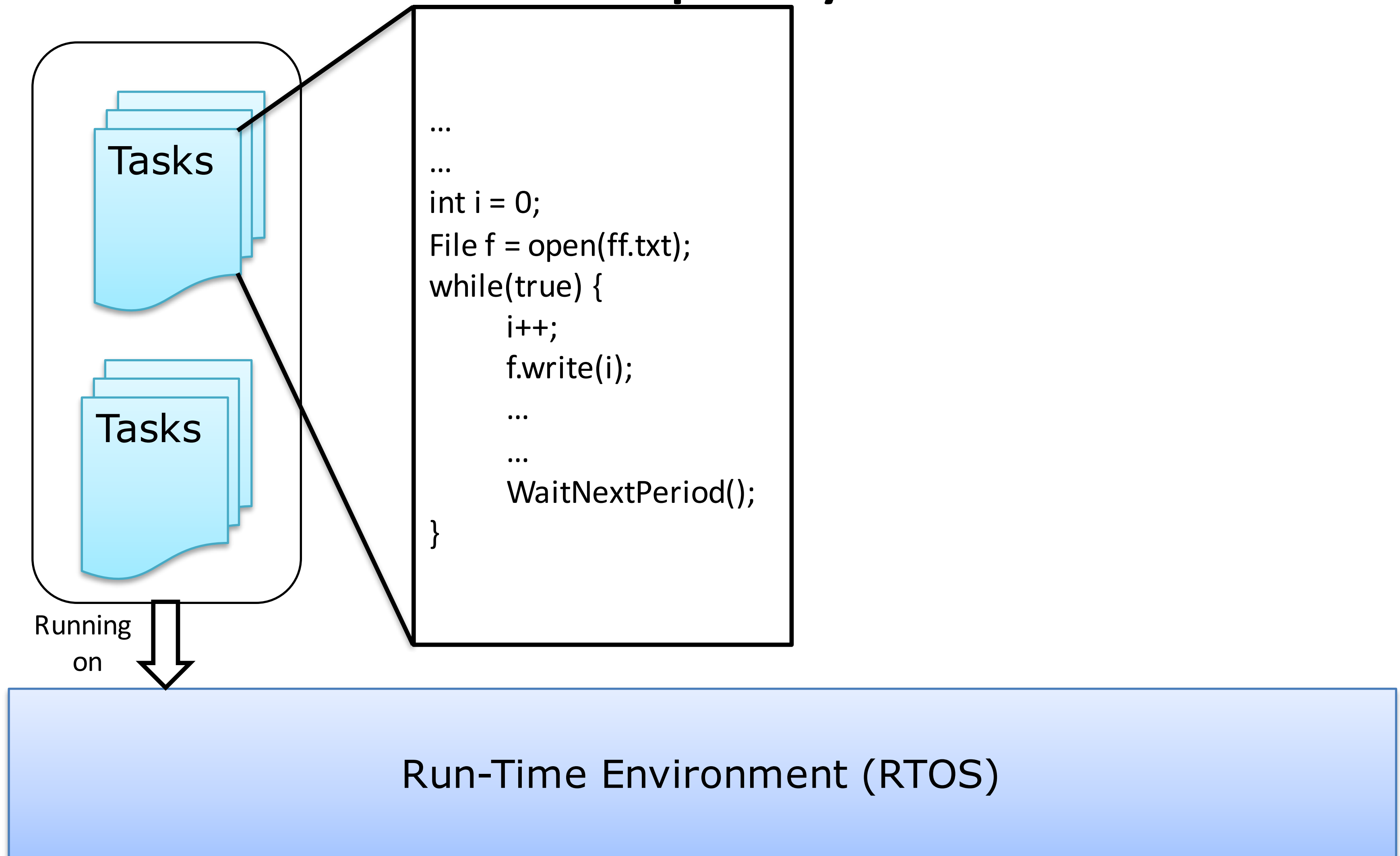
Verification of a Functional Property



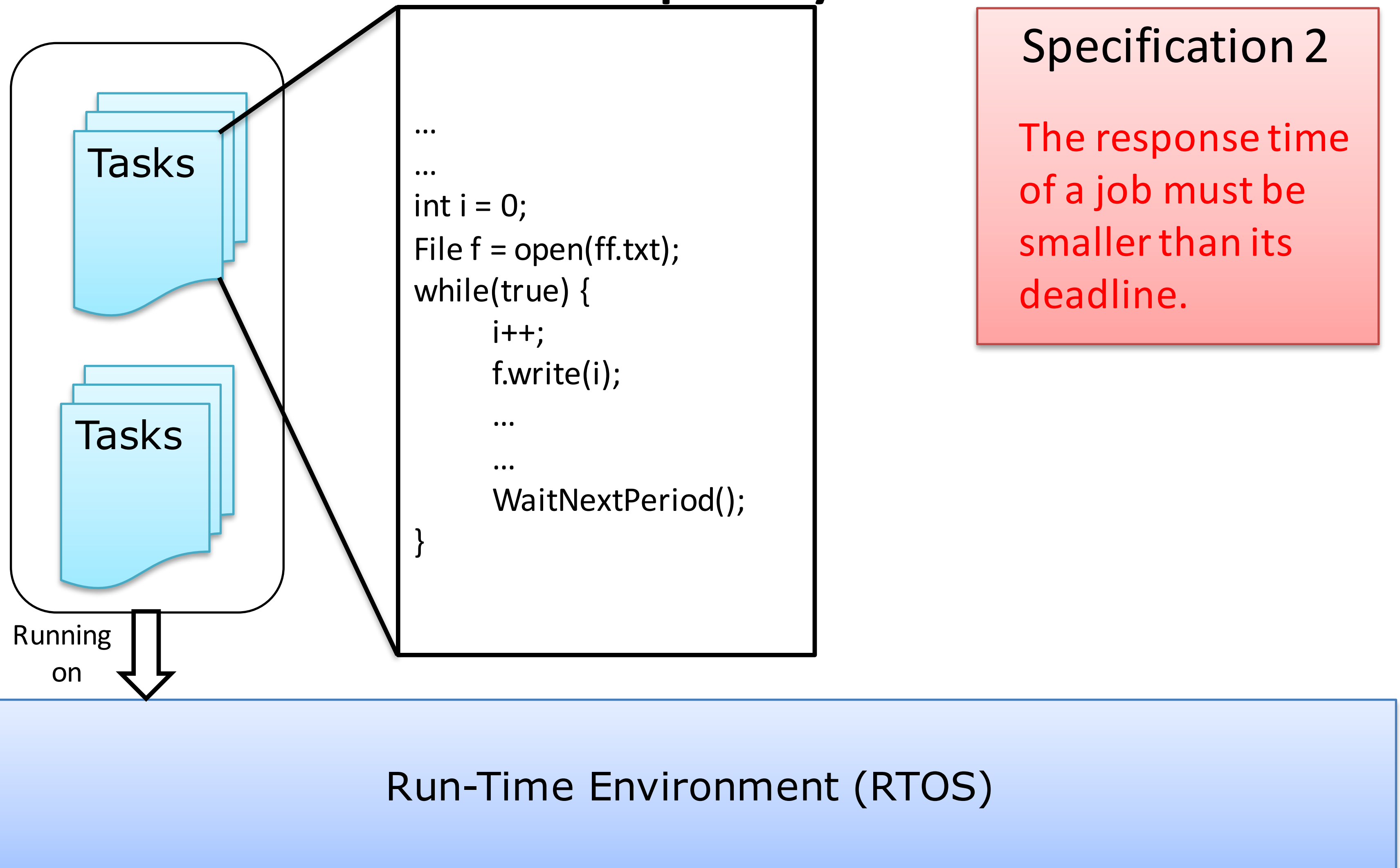


Run-Time Environment (RTOS)

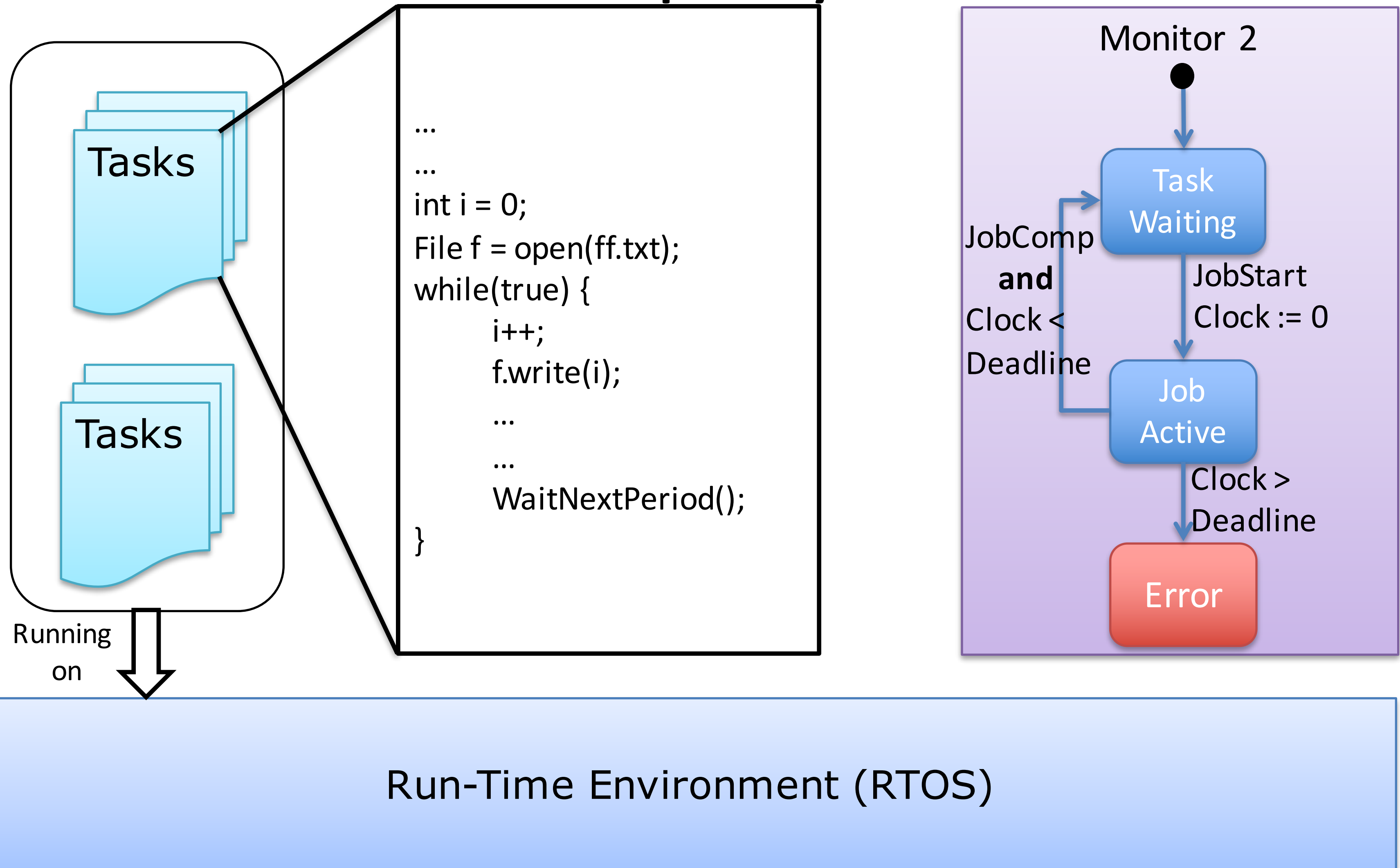
Verification of an Extra-Functional Property



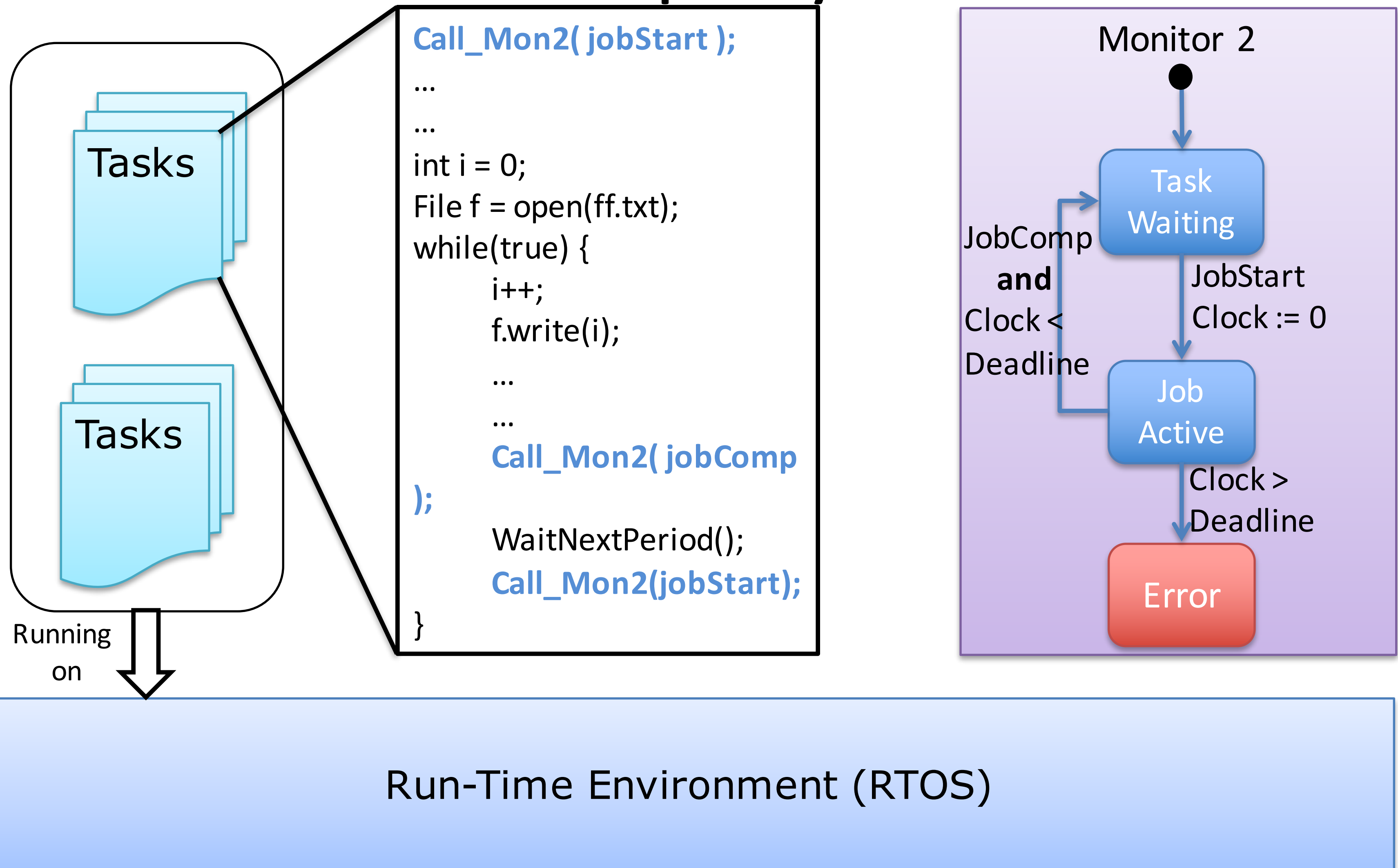
Verification of an Extra-Functional Property



Verification of an Extra-Functional Property

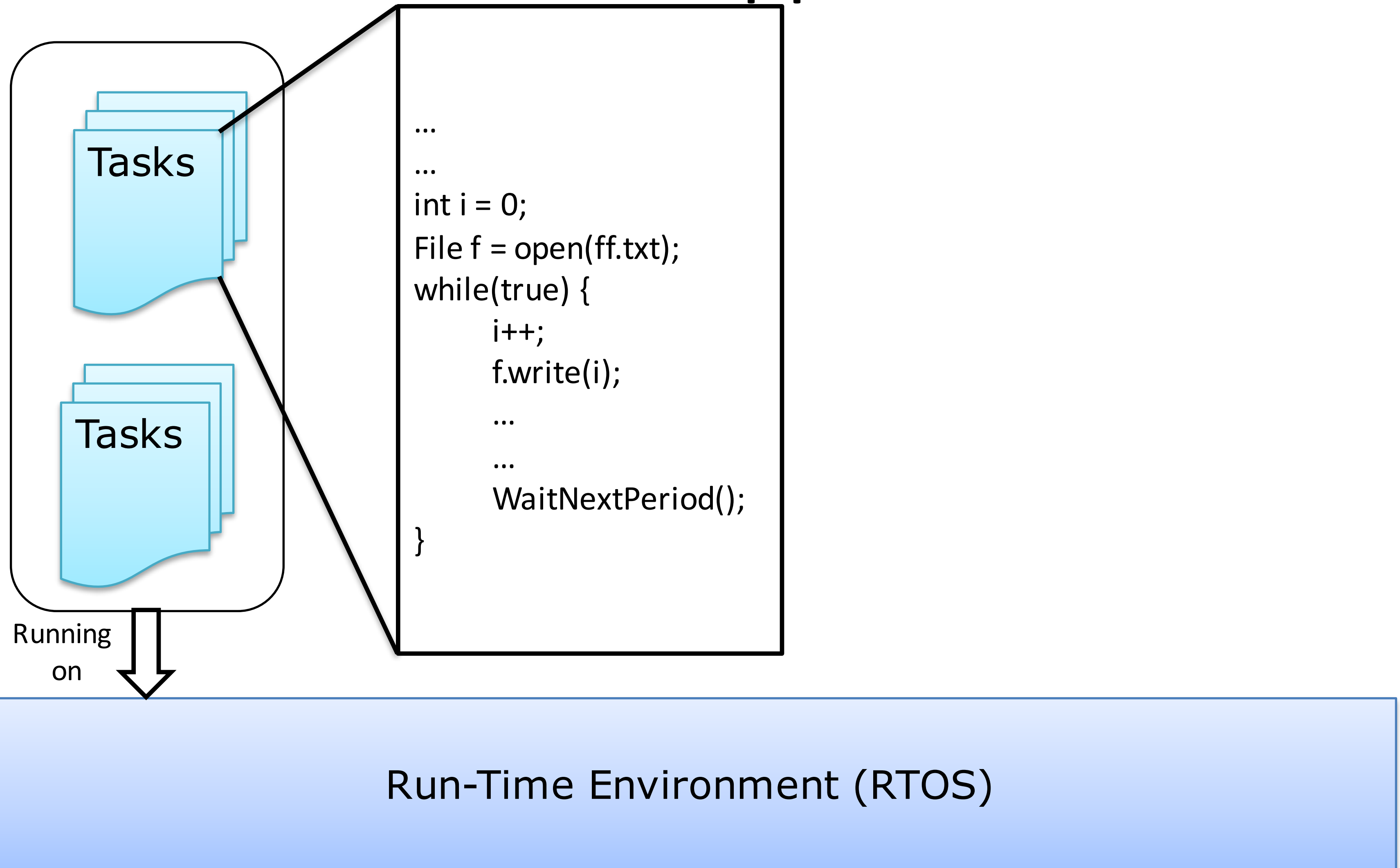


Verification of an Extra-Functional Property

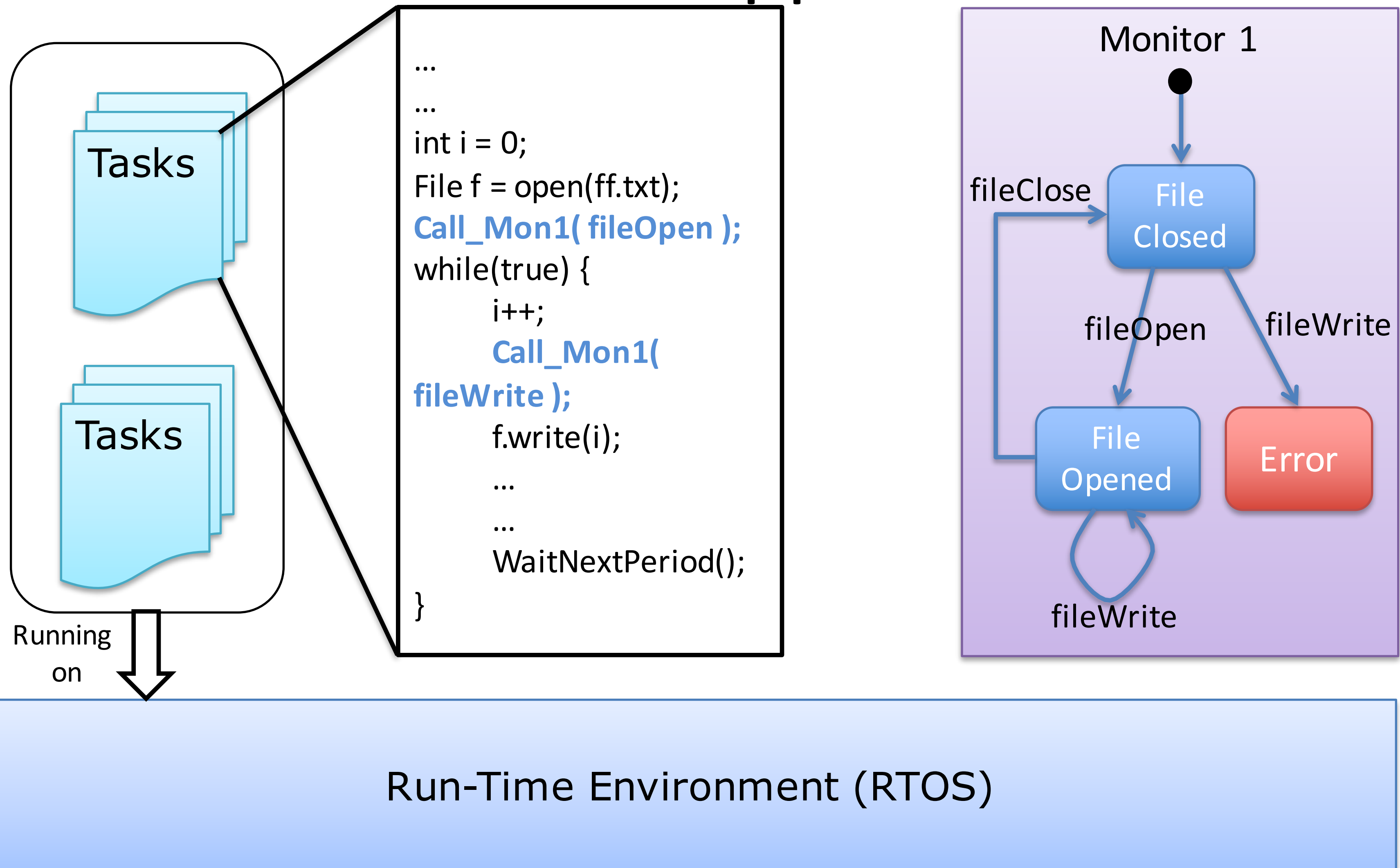


LIMITATIONS OF CURRENT ARCHITECTURES

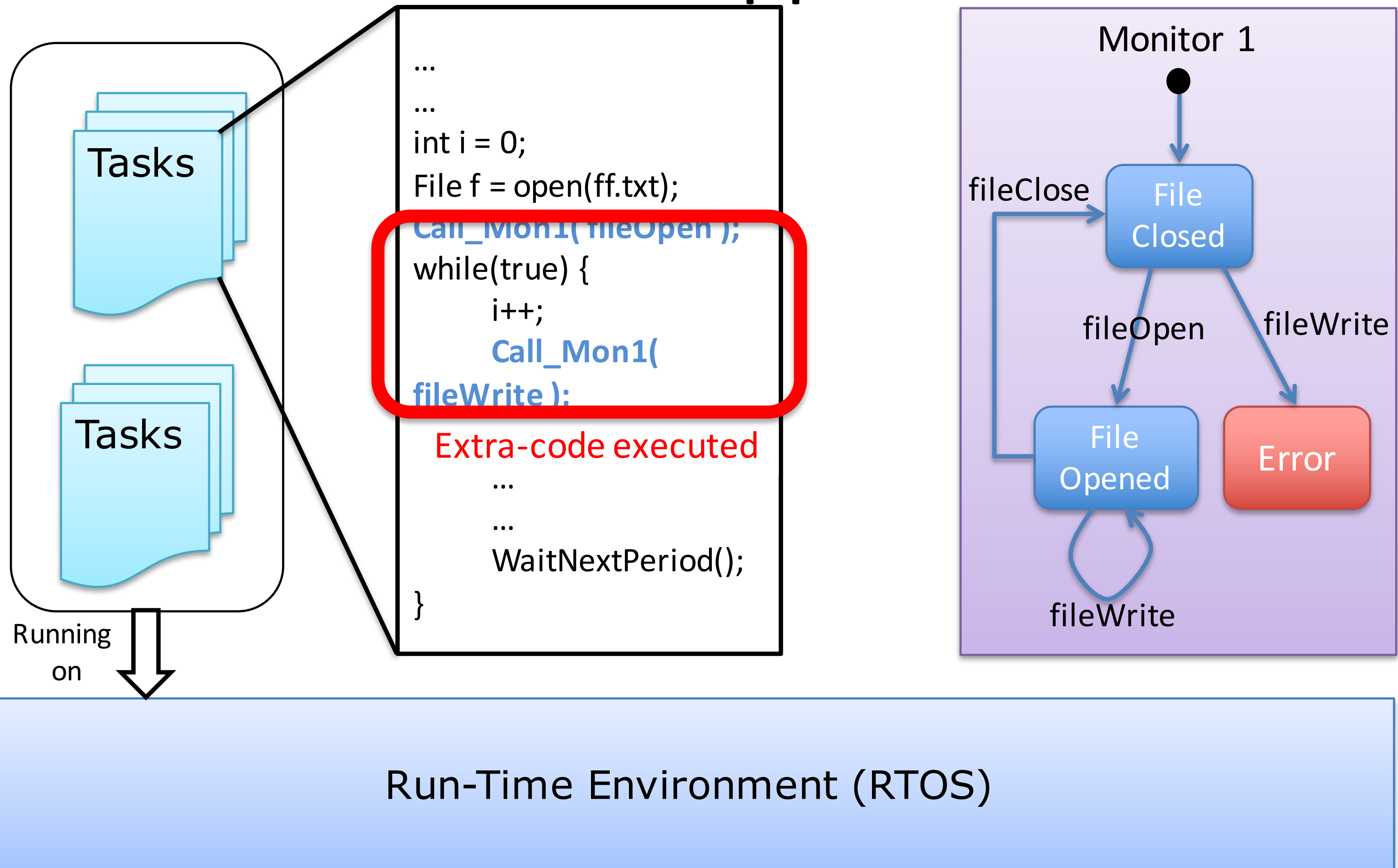
Change Timing Properties of the Monitored Application



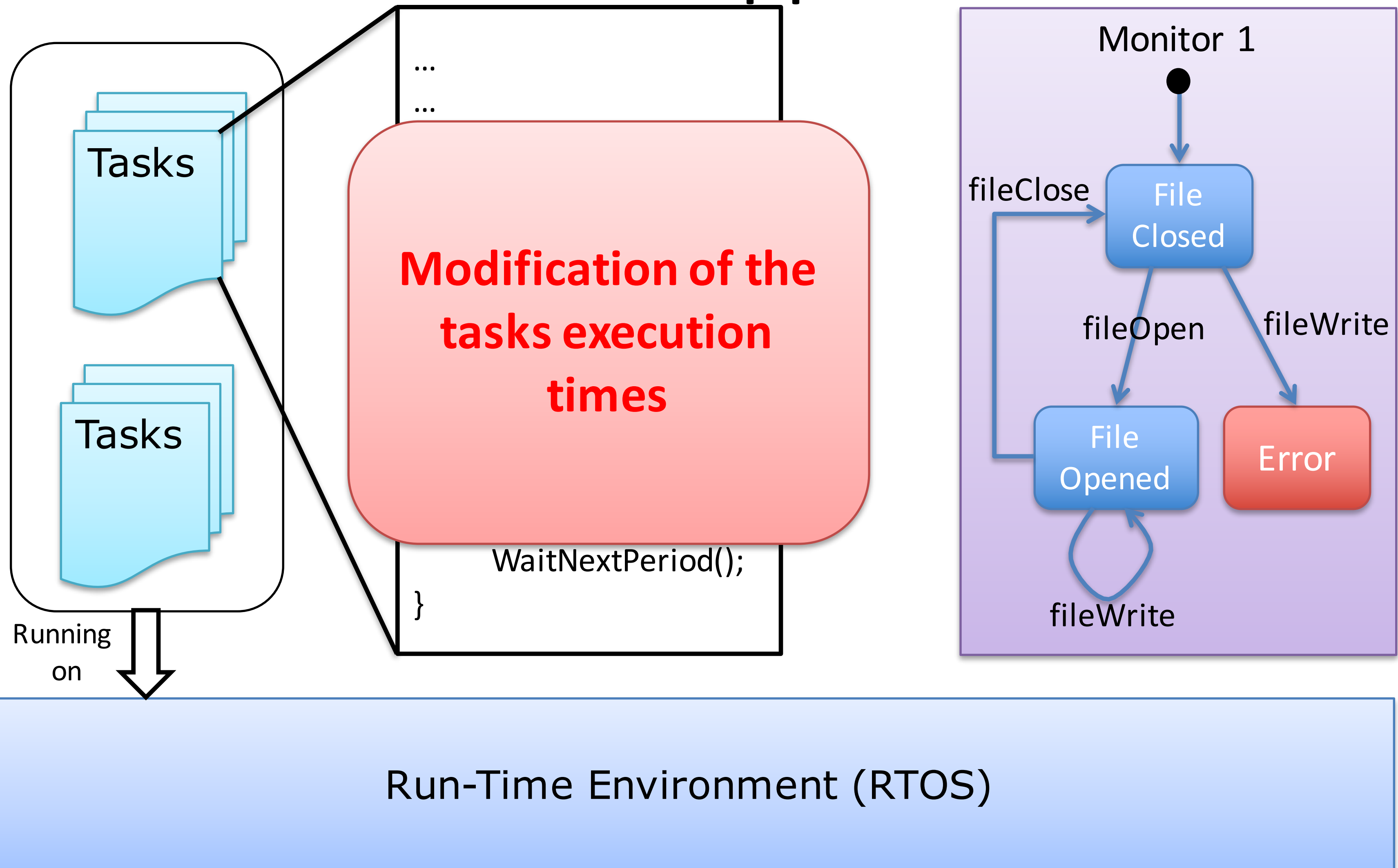
Change Timing Properties of the Monitored Application



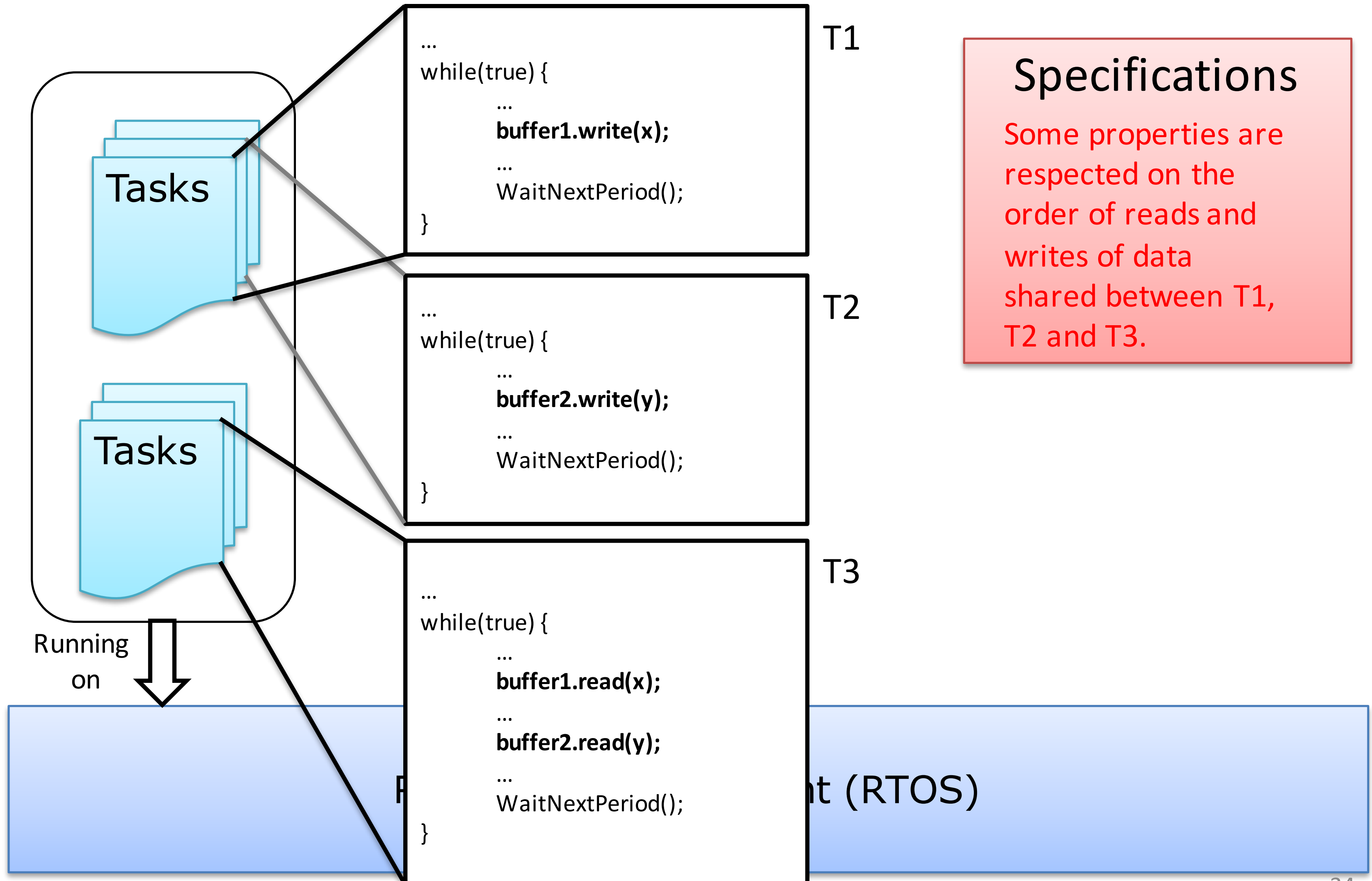
Change Timing Properties of the Monitored Application



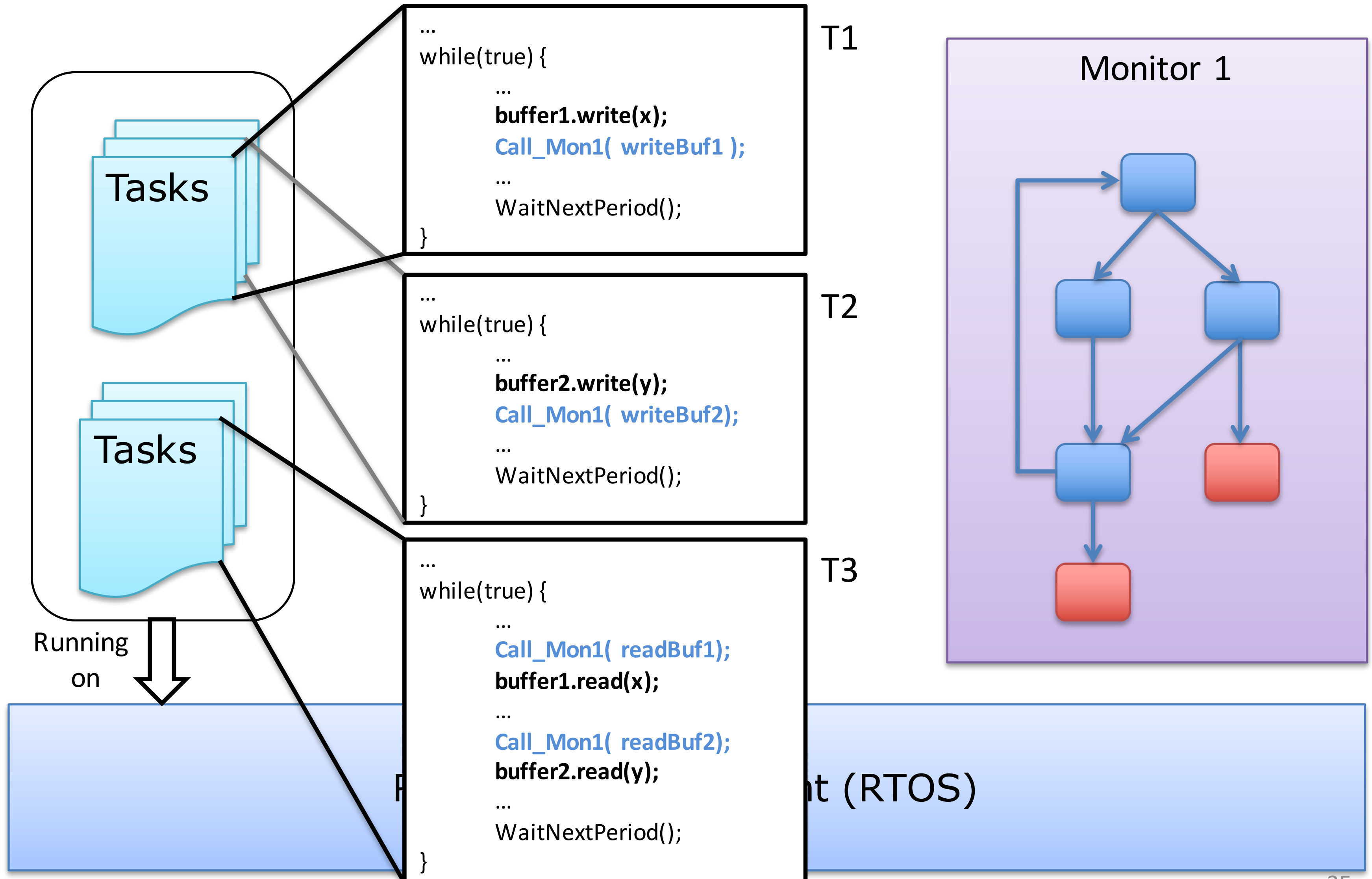
Change Timing Properties of the Monitored Application



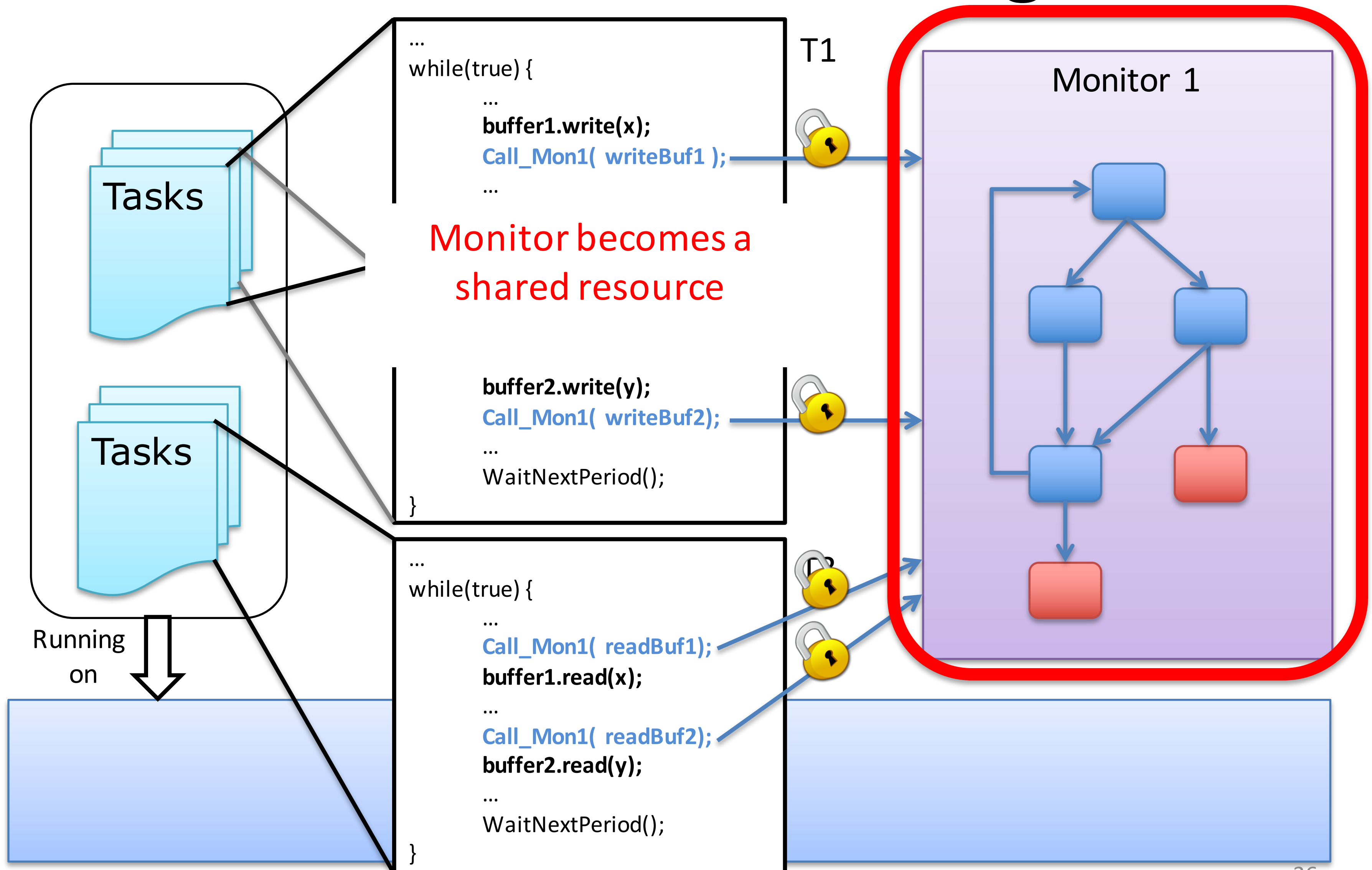
No Time Partitioning



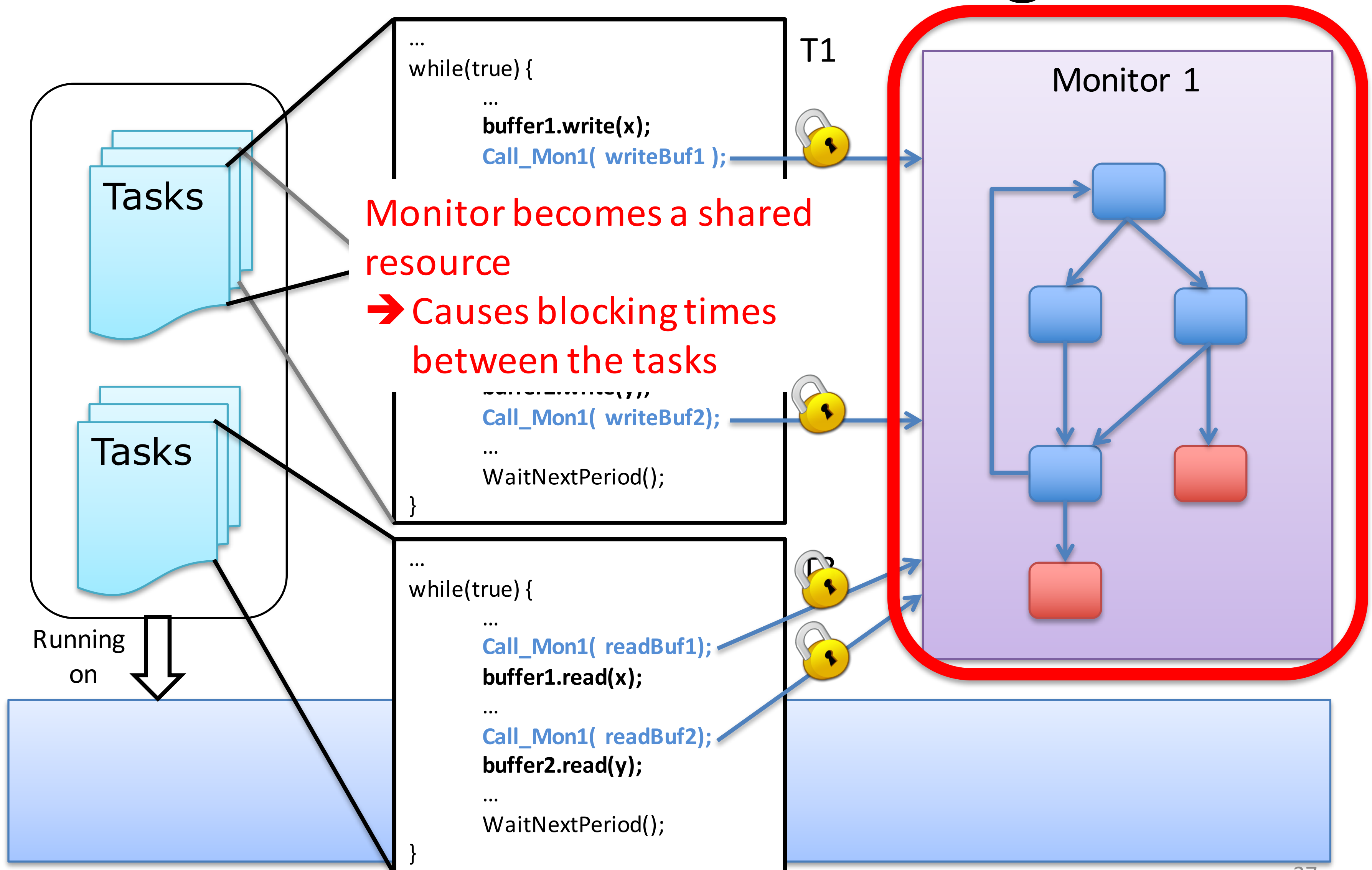
No Time Partitioning



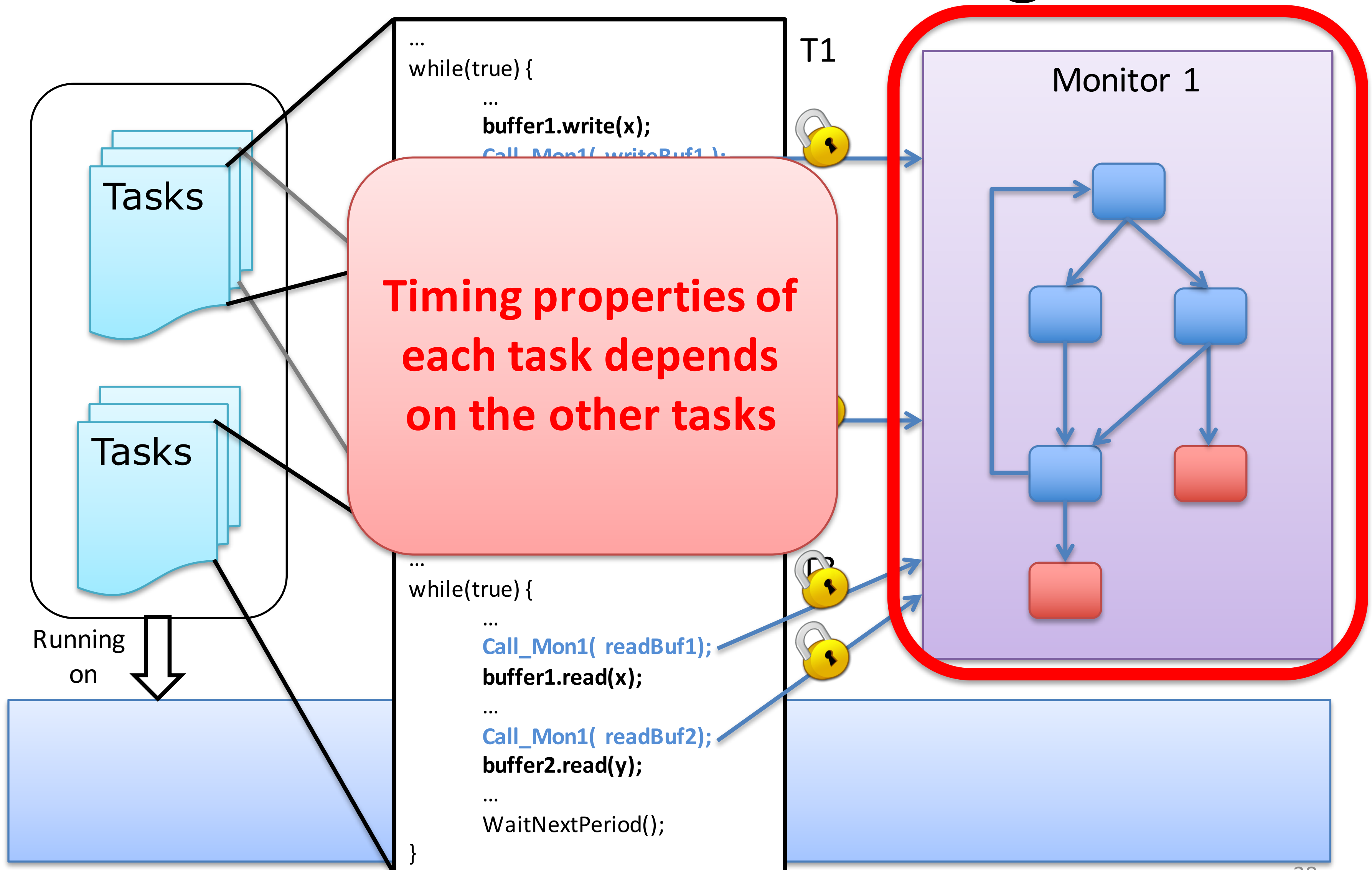
No Time Partitioning



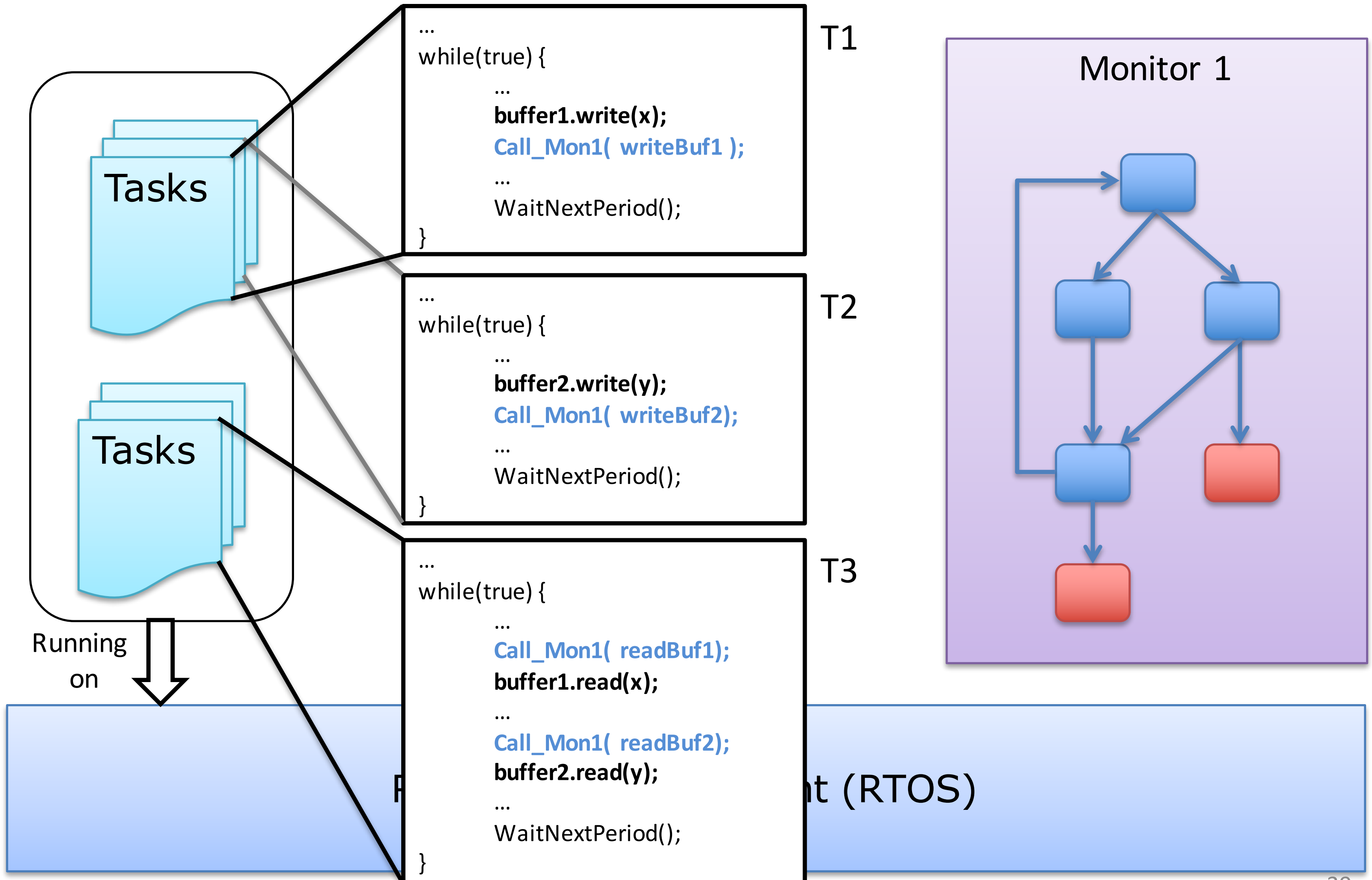
No Time Partitioning



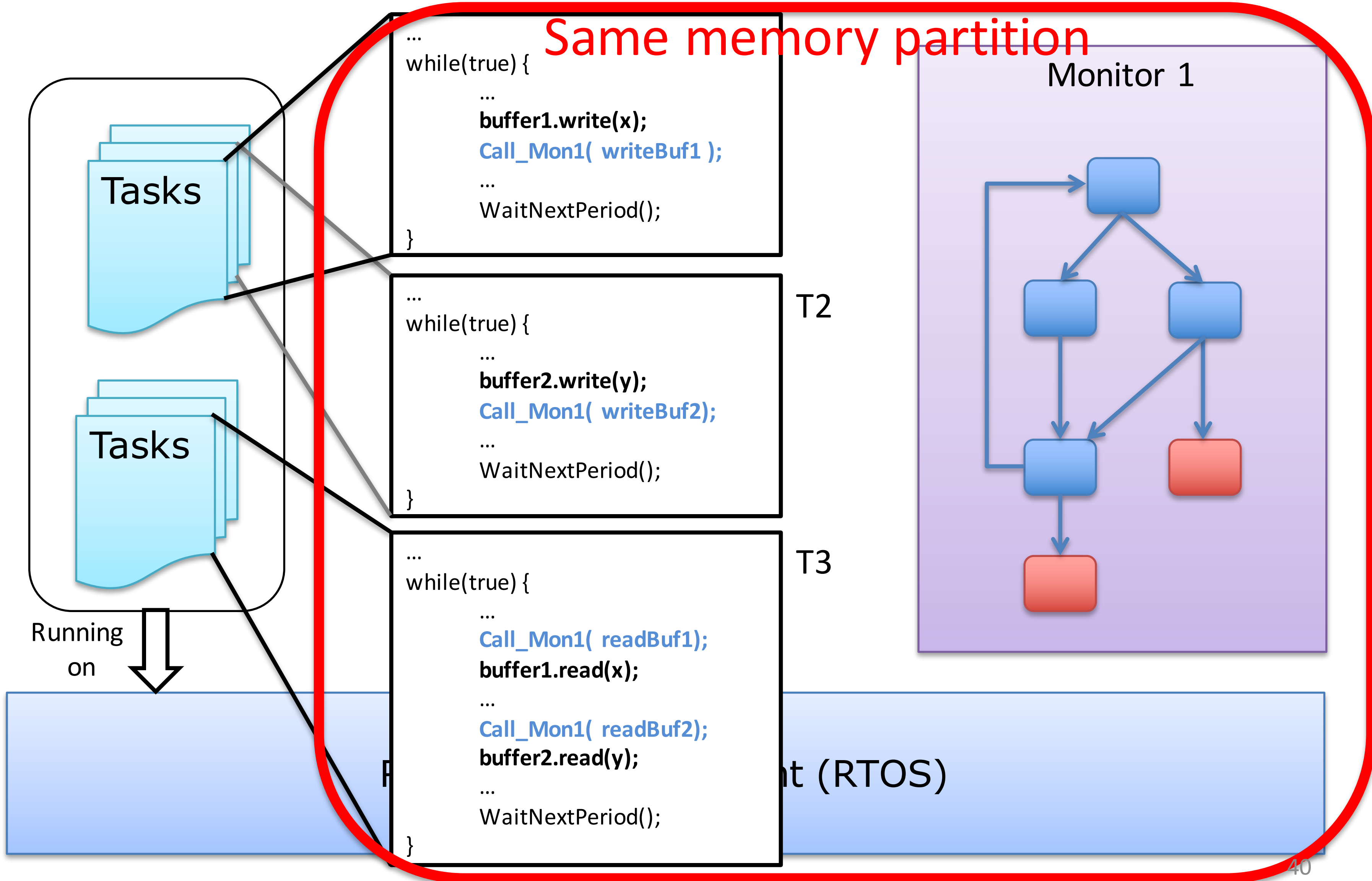
No Time Partitioning



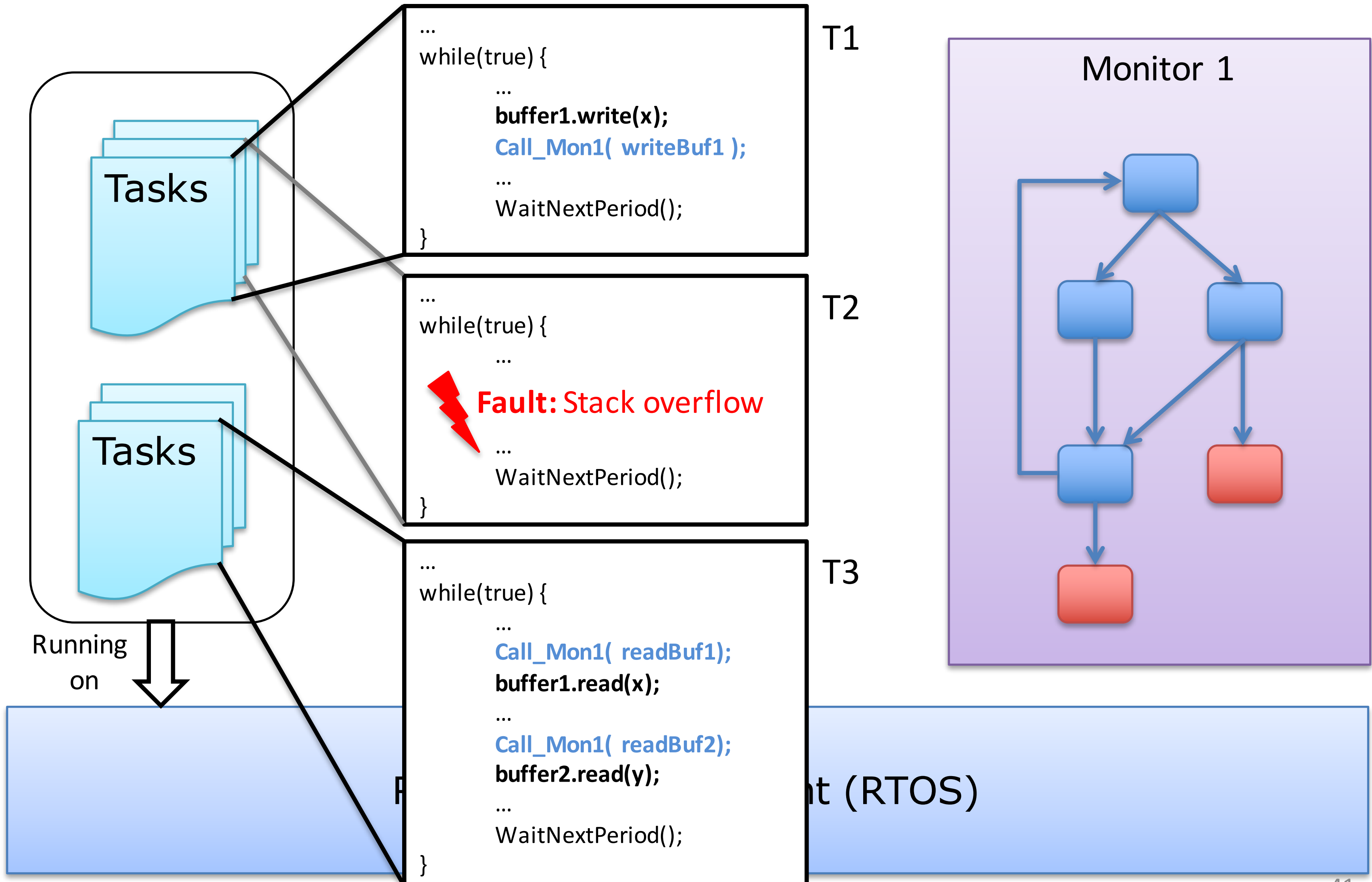
No Space Partitioning



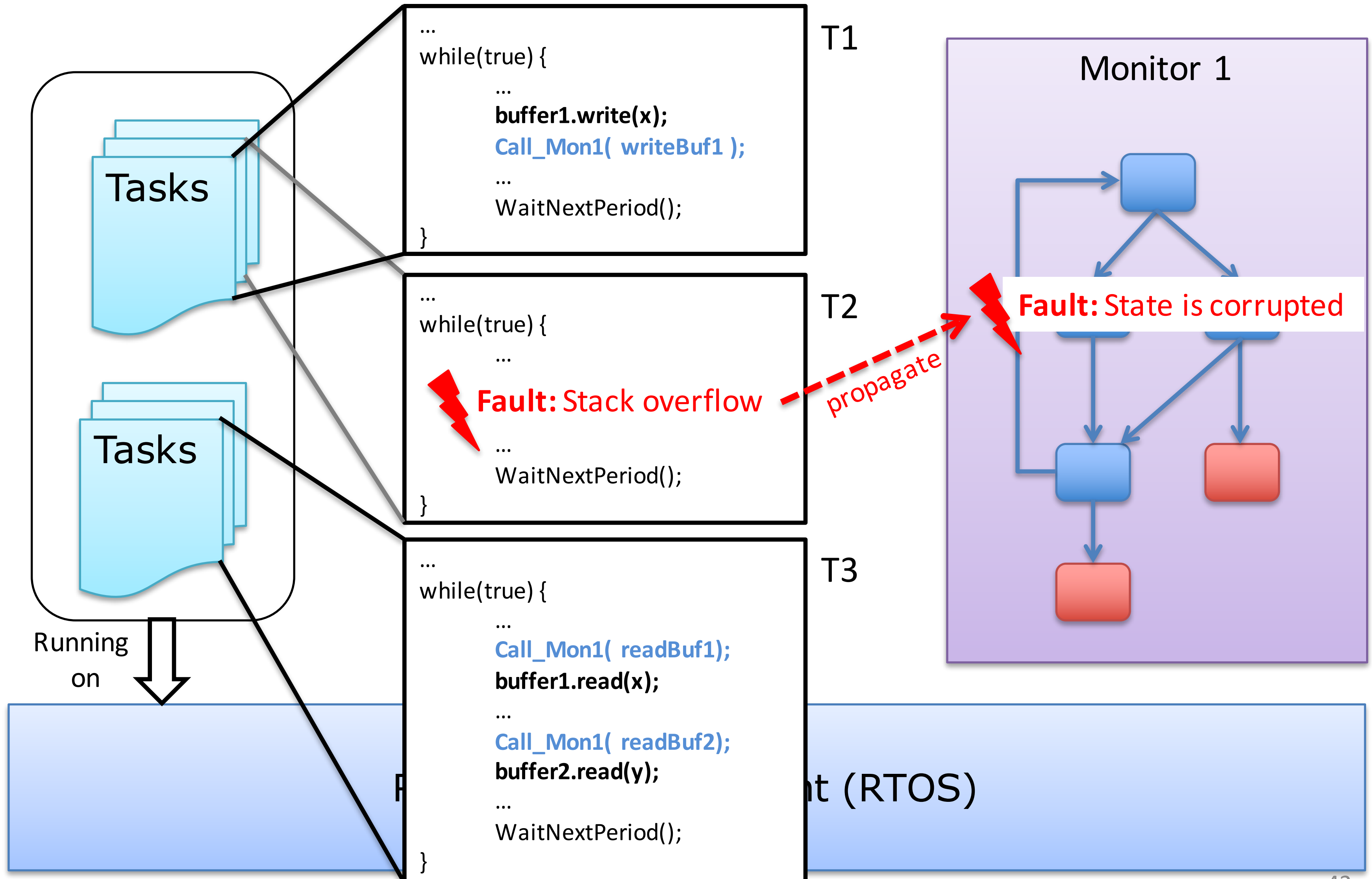
No Space Partitioning



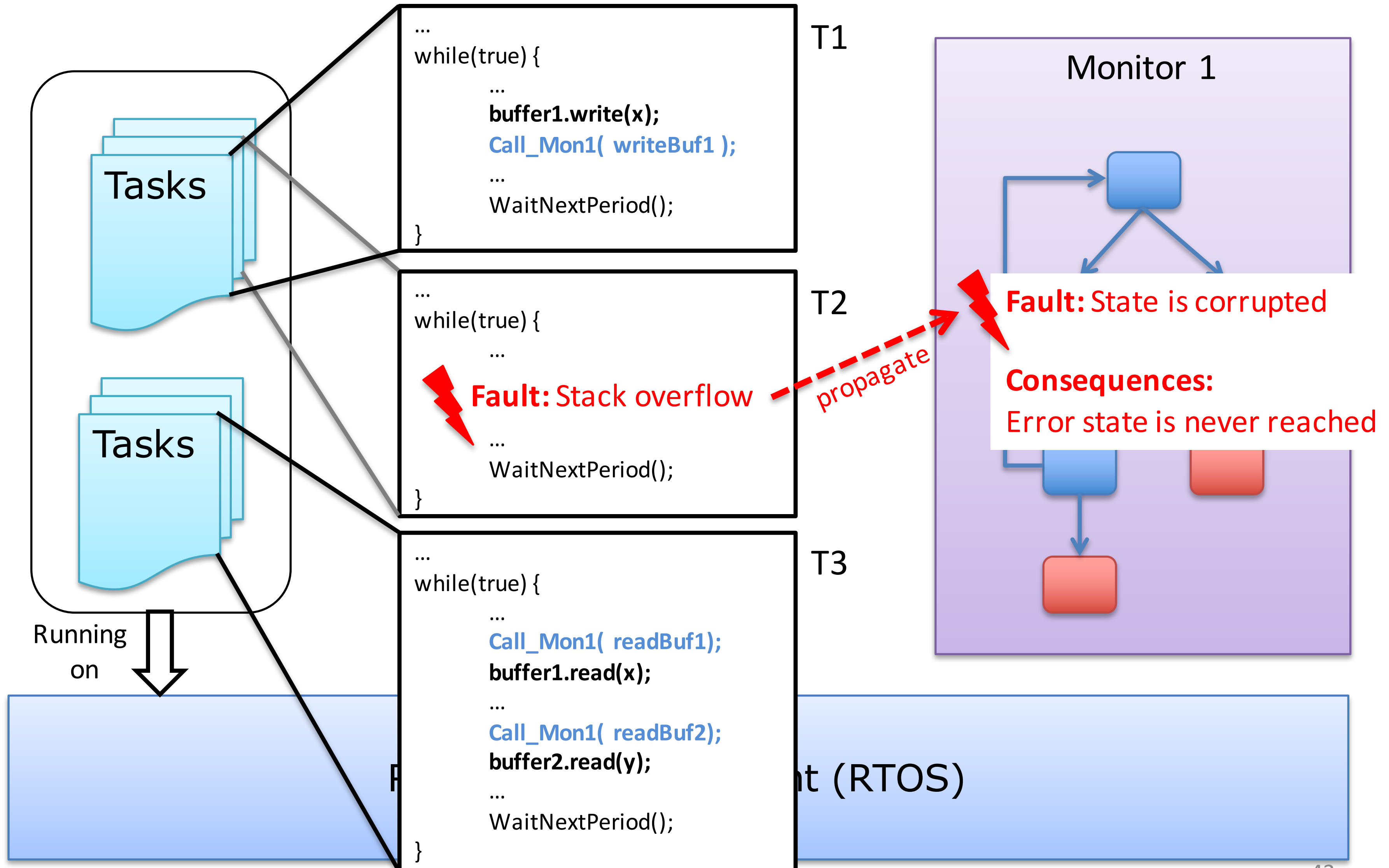
No Space Partitioning



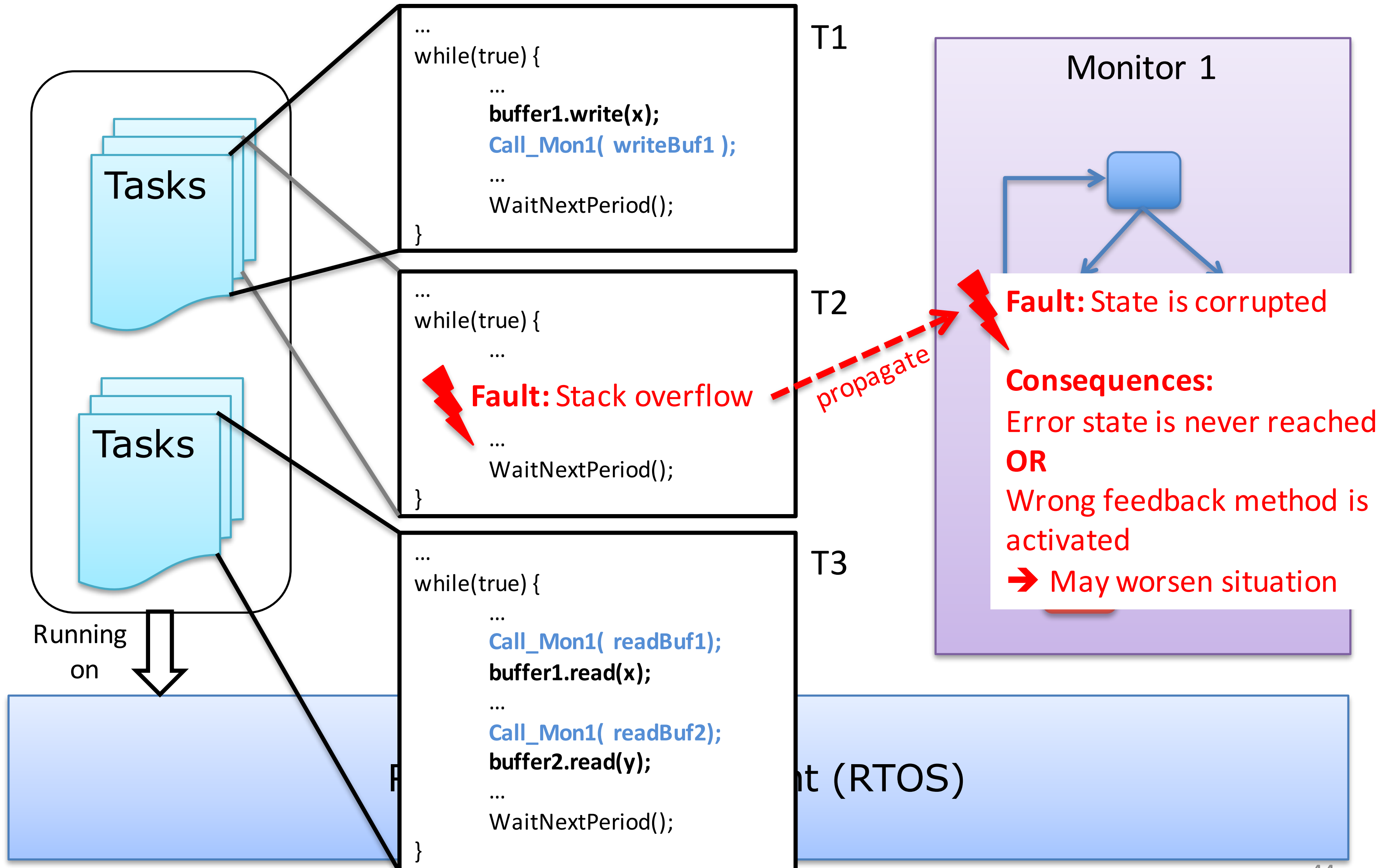
No Space Partitioning



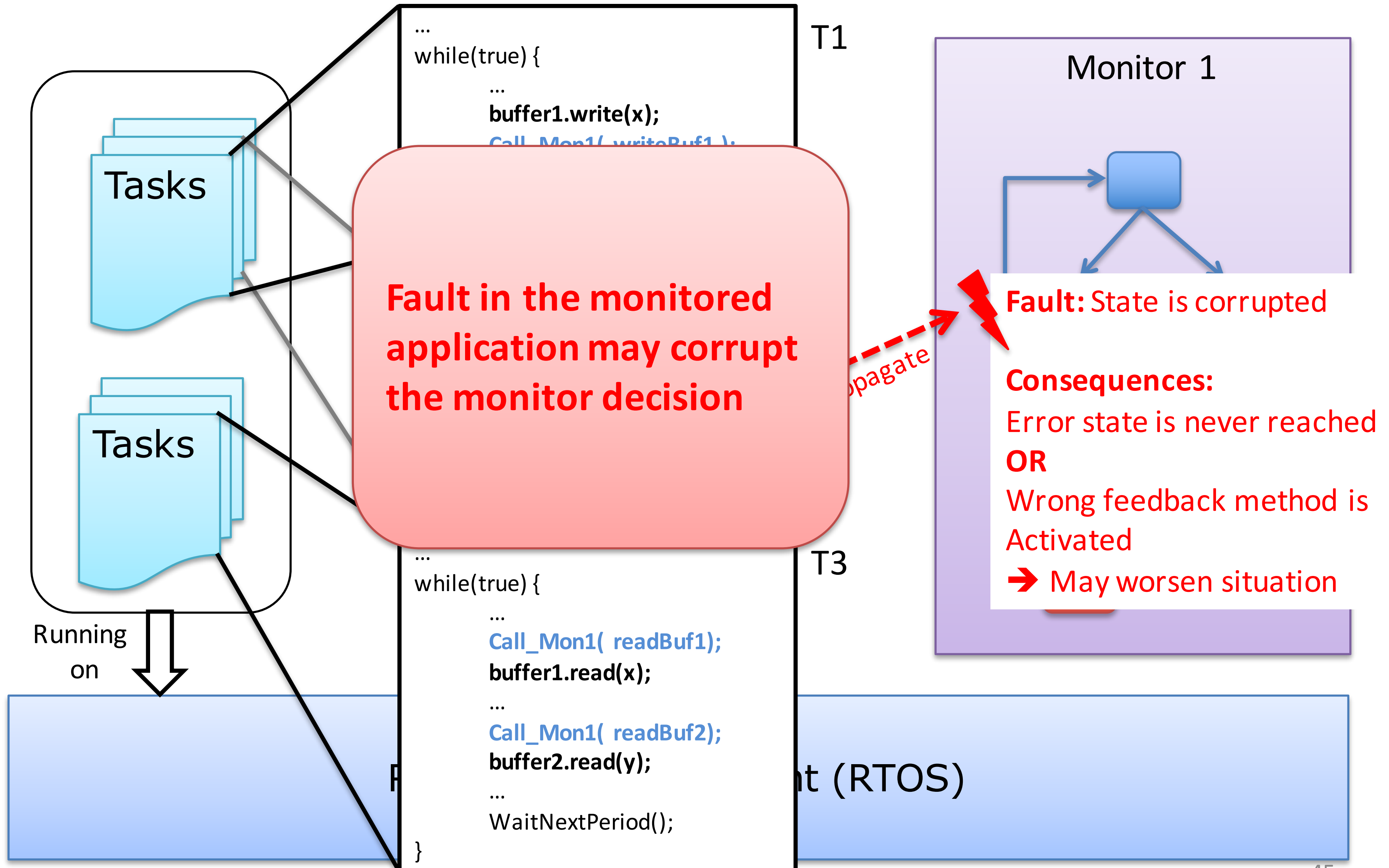
No Space Partitioning



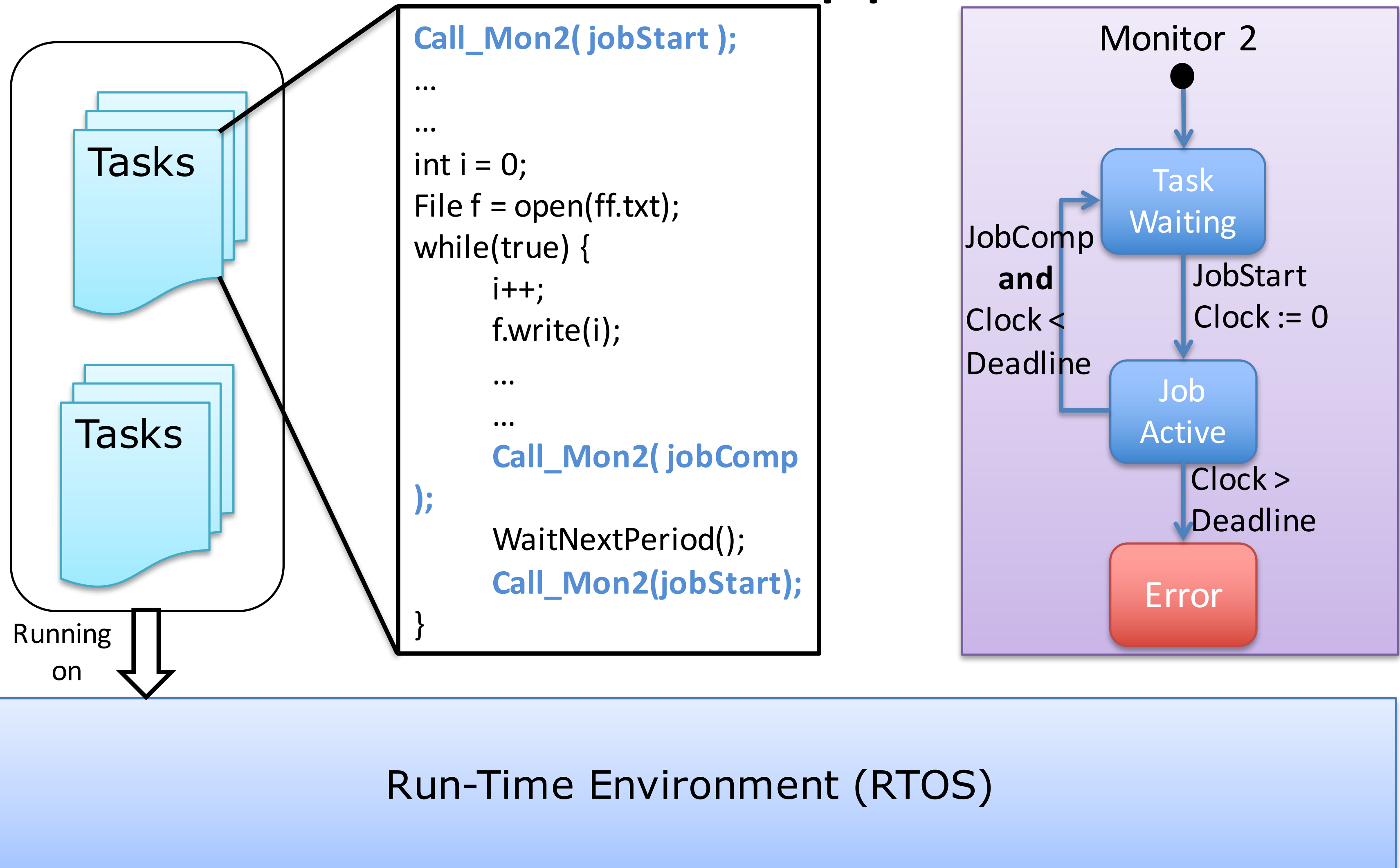
No Space Partitioning



No Space Partitioning



No Independence between Monitors and Monitored Application



No Independence between Monitors and Monitored Application

Tasks

Tasks

Running on

```
Call_Mon2( jobStart );
…
…
	…
	Call_Mon2( jobComp );
	WaitNextPeriod();
	Call_Mon2(jobStart);
}
```

State of the monitor updated only when called by the task

Monitor 2

Task Waiting

JobStart
Clock := 0

Job Active

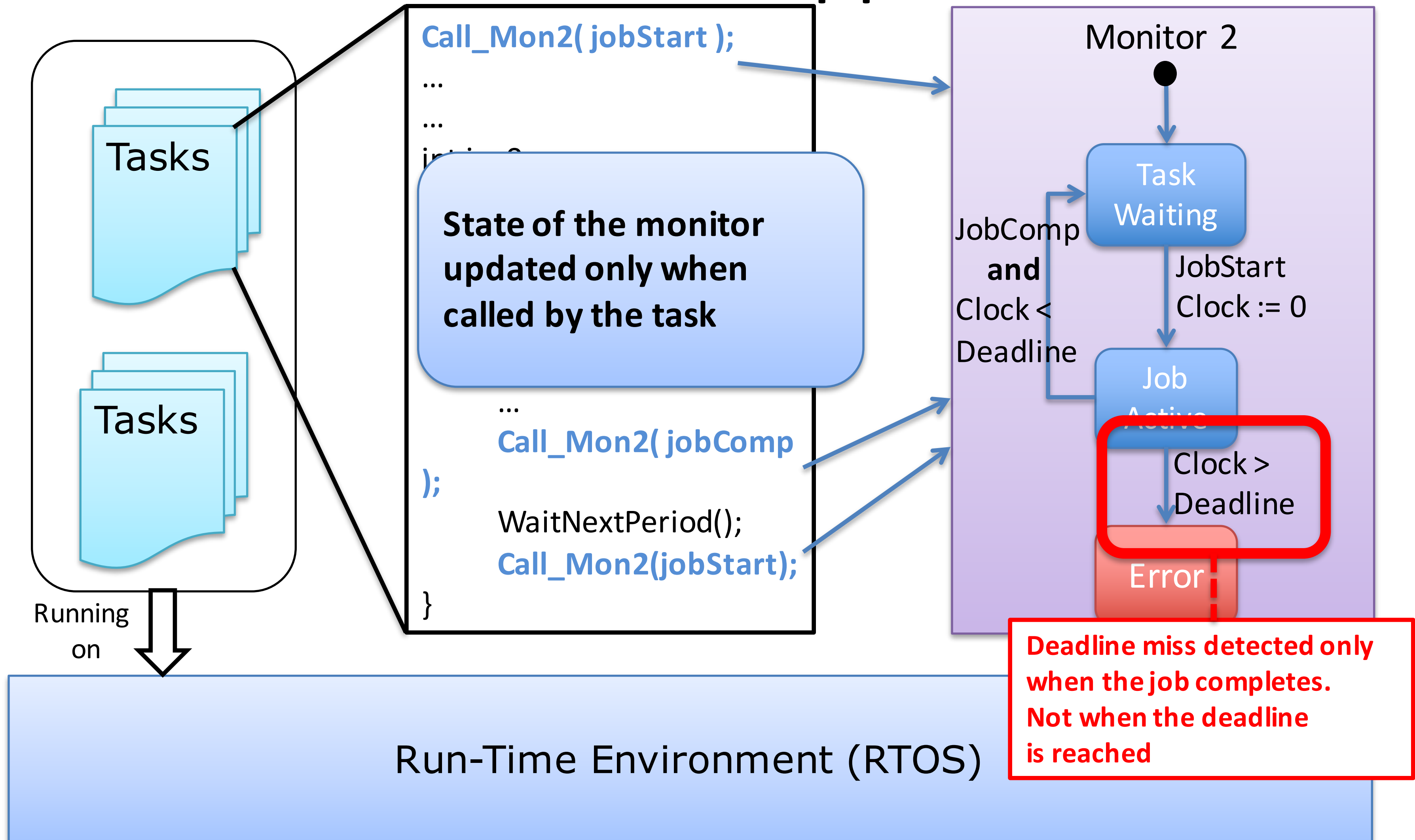
JobComp **and** Clock < Deadline

Clock > Deadline

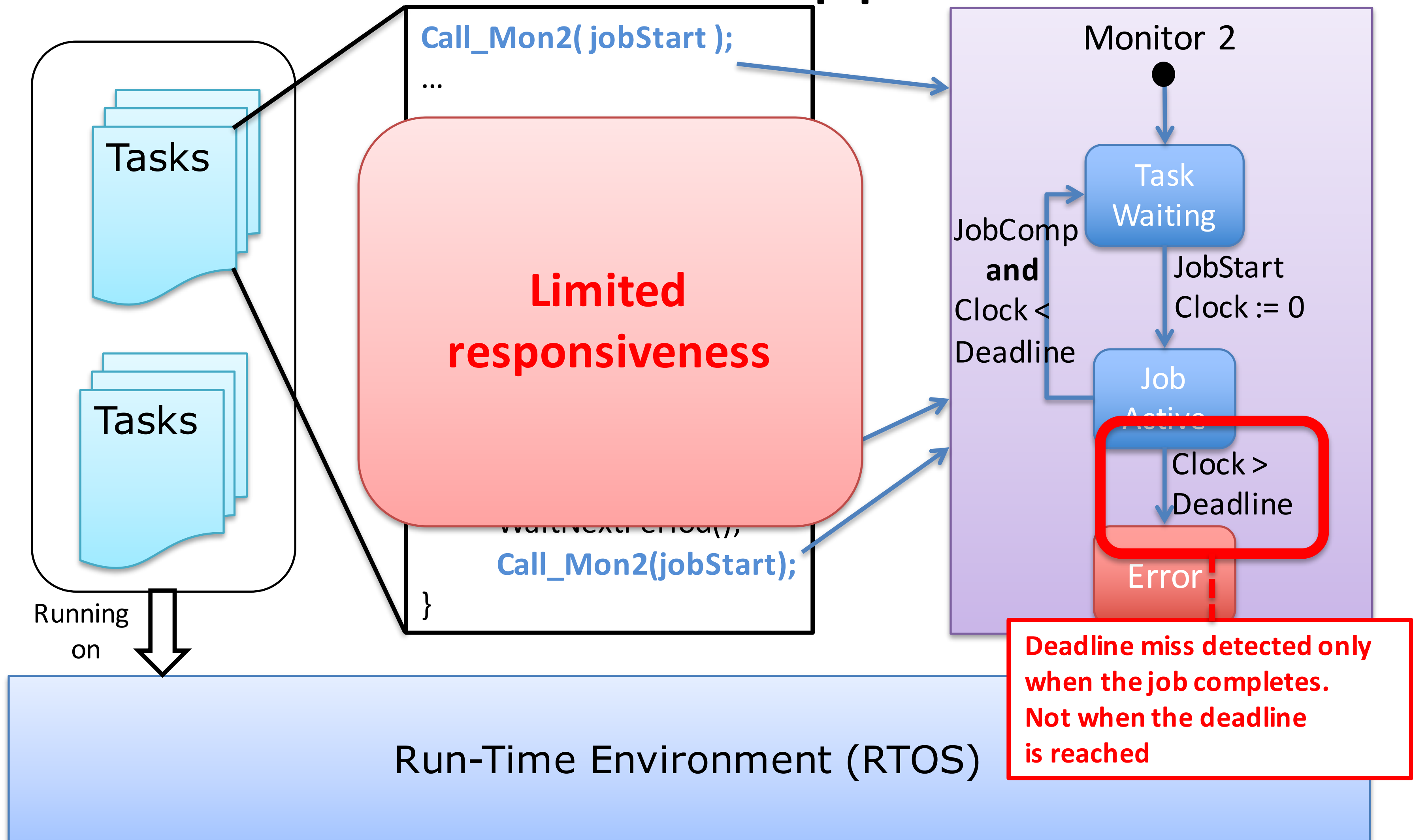
Error

Run-Time Environment (RTOS)

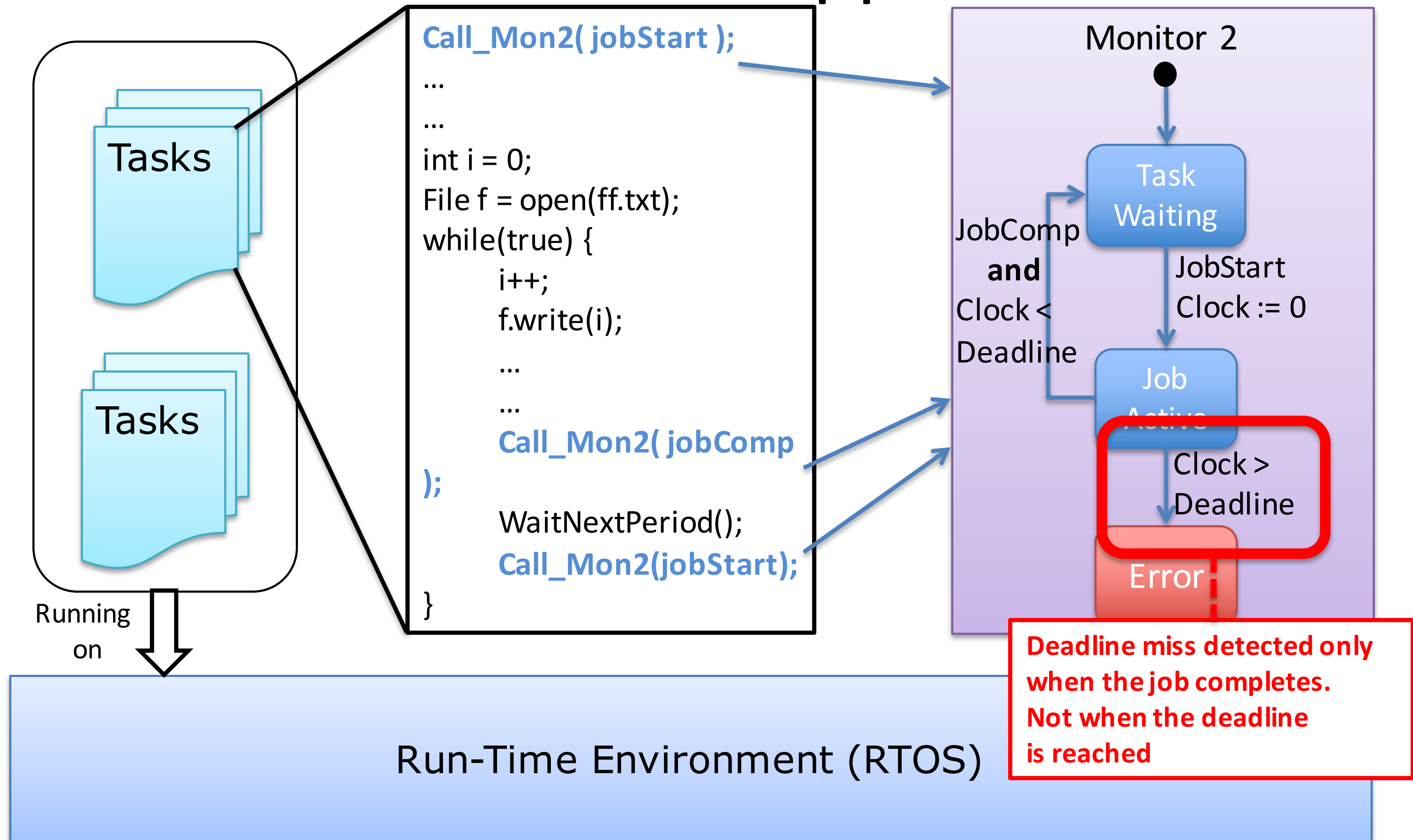
No Independence between Monitors and Monitored Application



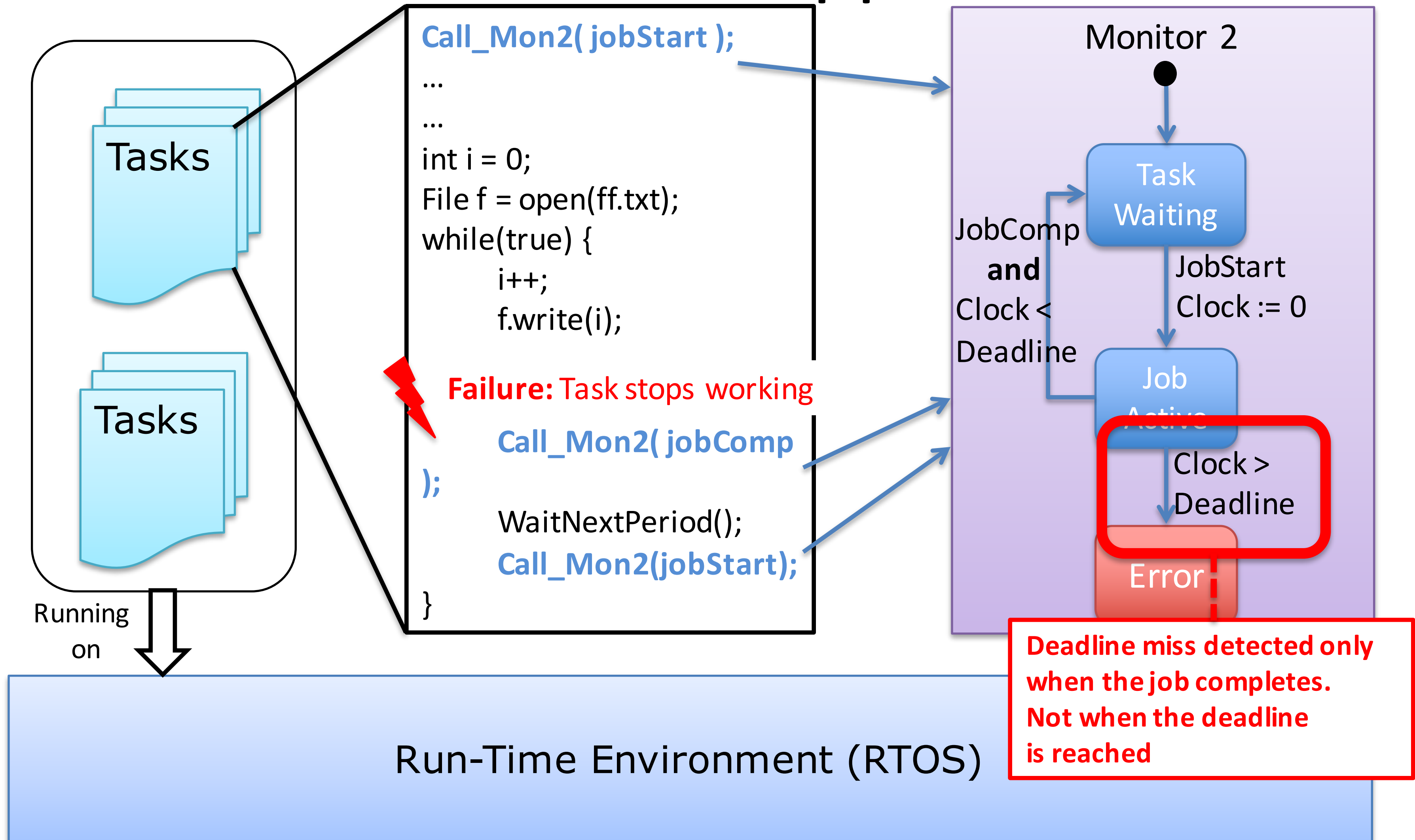
No Independence between Monitors and Monitored Application



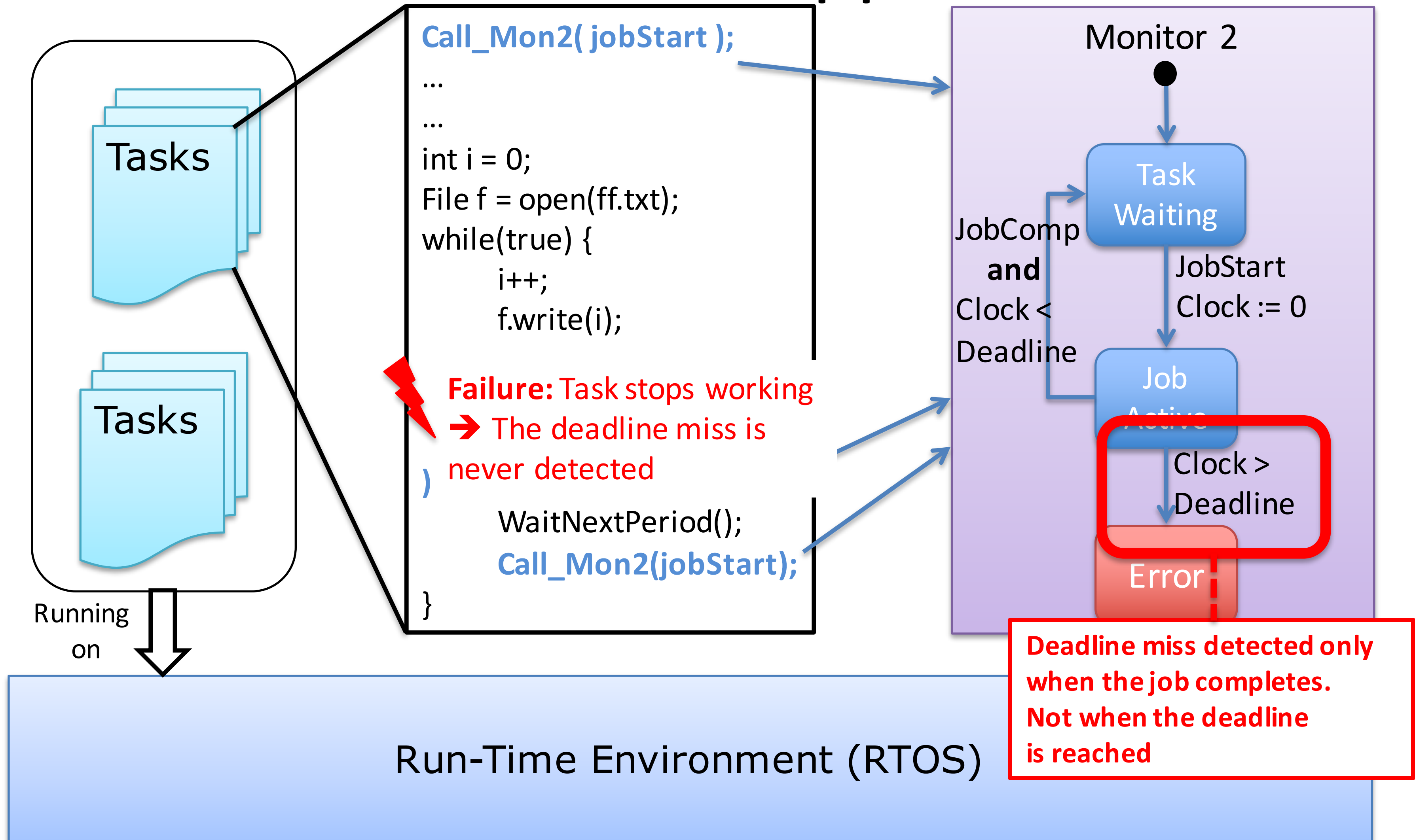
No Independence between Monitors and Monitored Application



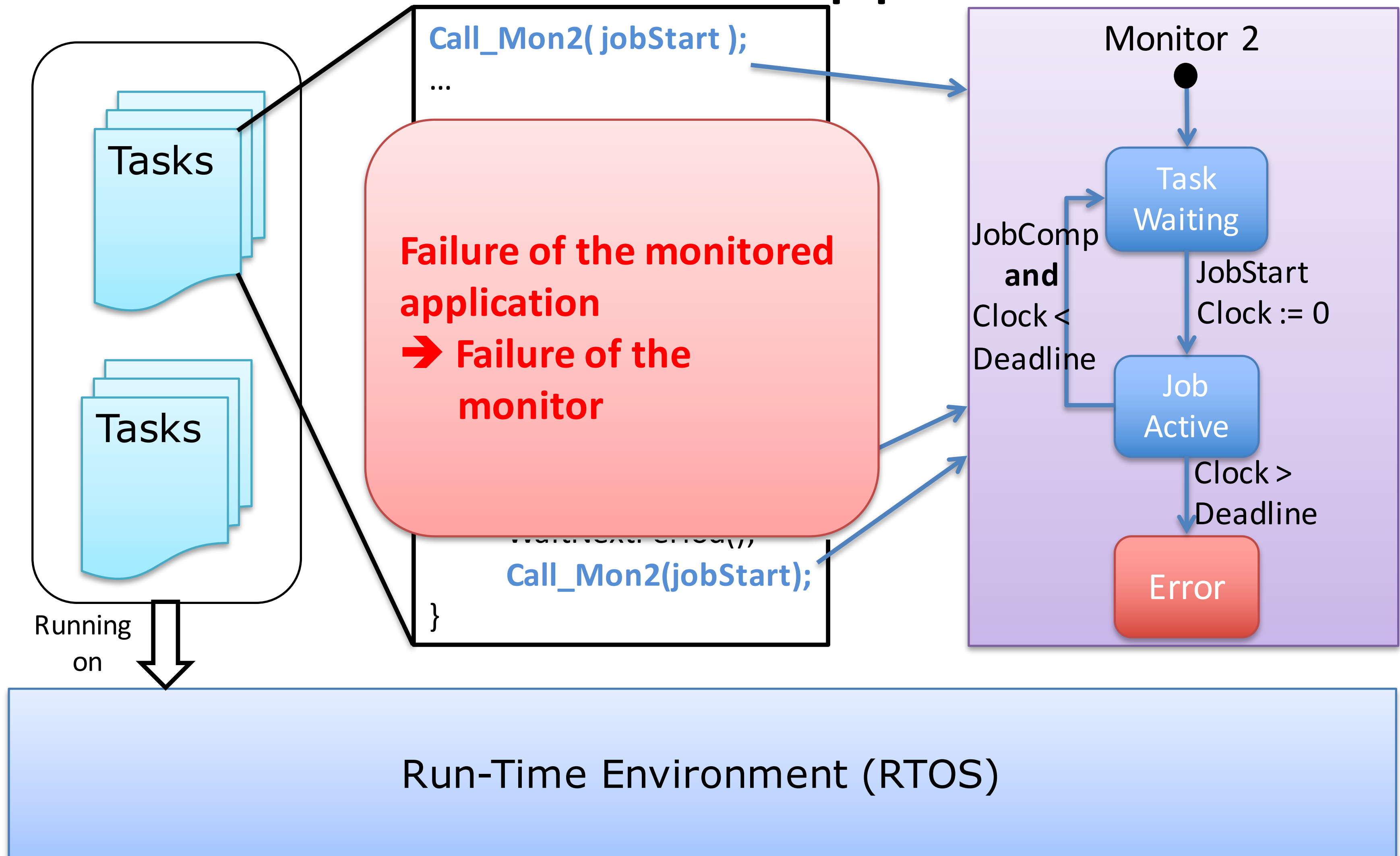
No Independence between Monitors and Monitored Application



No Independence between Monitors and Monitored Application



No Independence between Monitors and Monitored Application

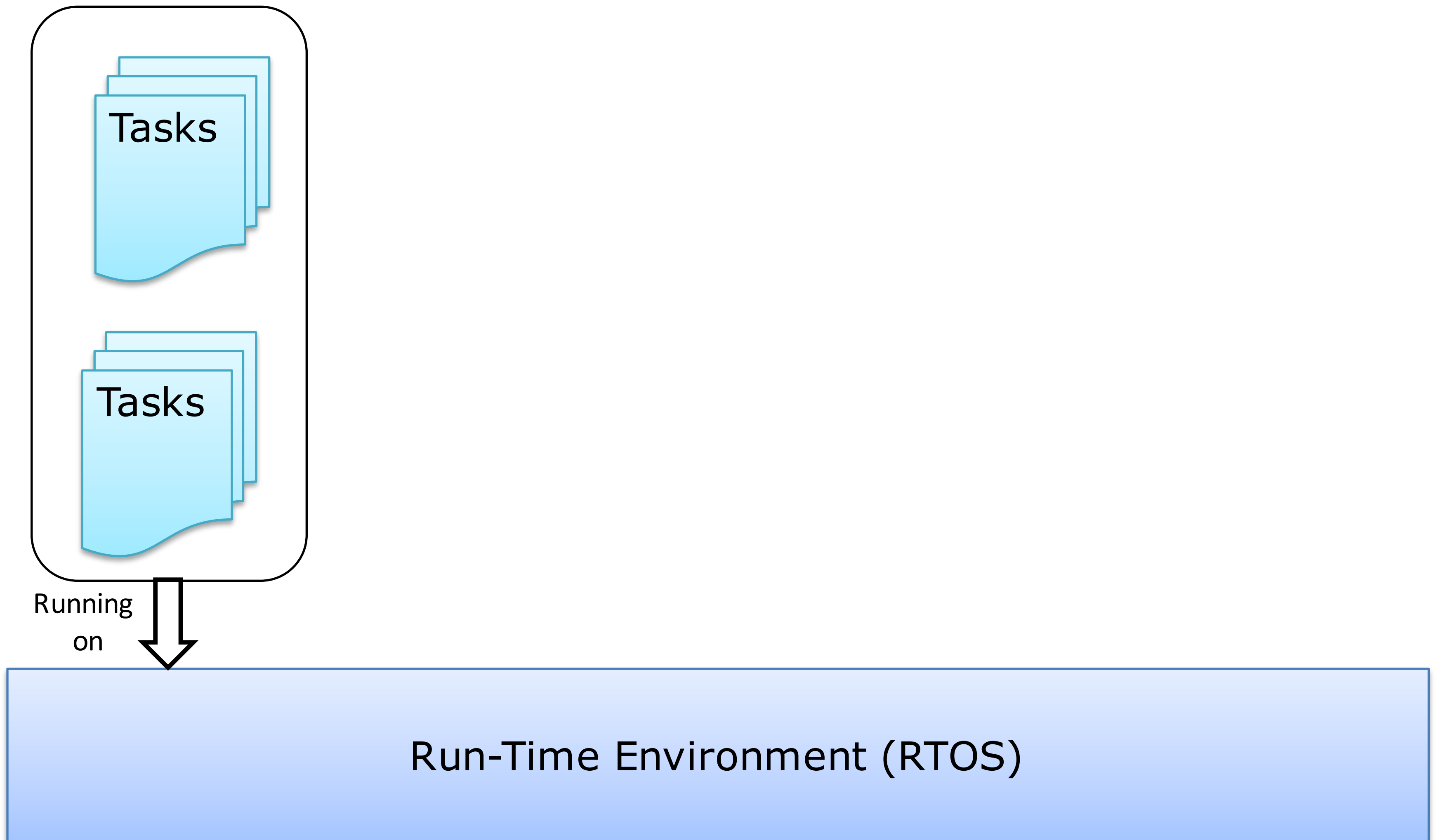


Limitations Summary

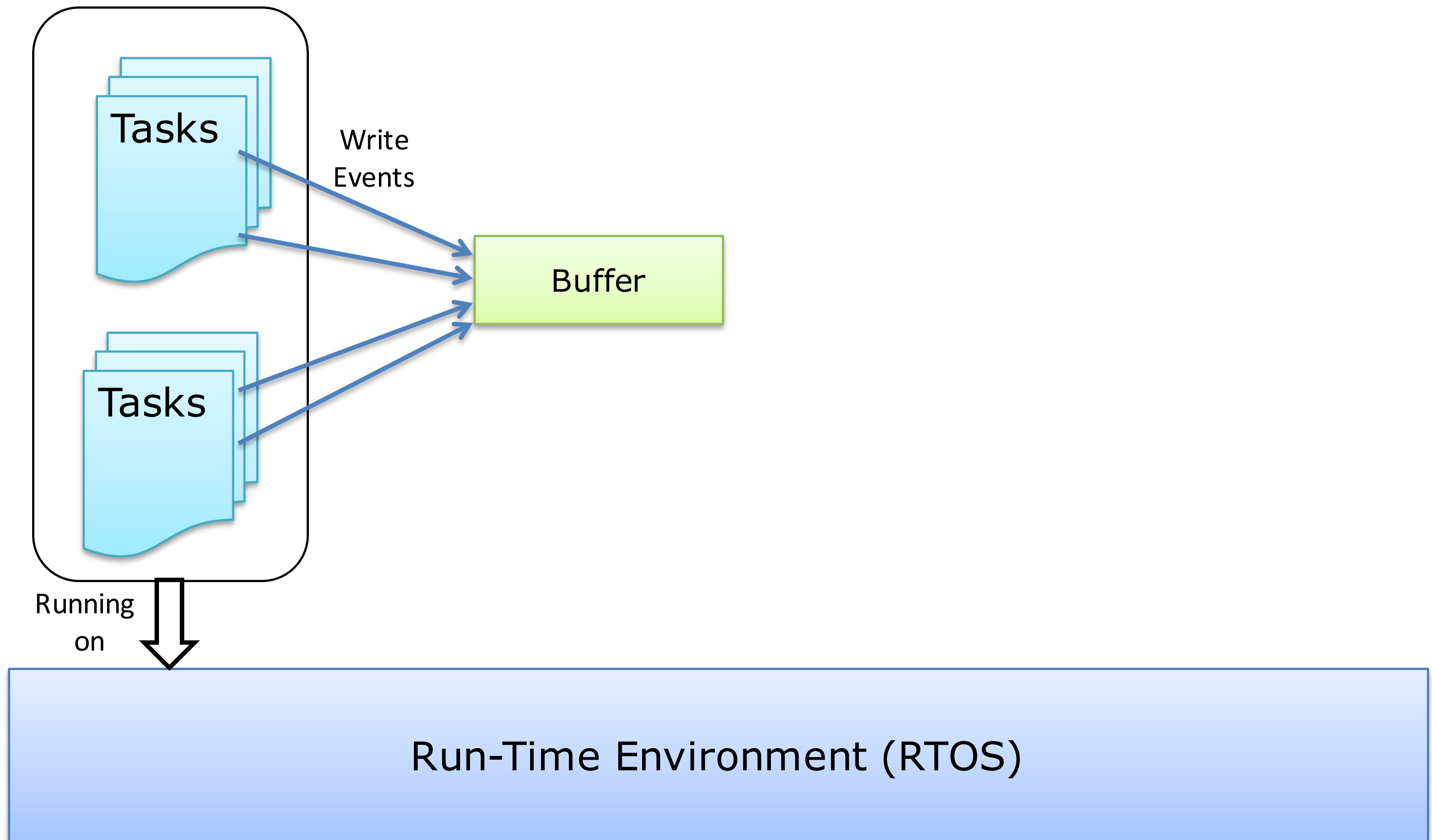
- Impact task execution times
- No time partitioning
  - Response time of one task is influenced by other tasks
- No space partitioning
  - Possible corruption of the monitor by a task and/or other monitors
- No independence between monitors and monitored application
  - Failure of the monitored task ➔ failure of the monitor
- Limited responsiveness

ALTERNATIVES IN THE STATE-OF-THE-ART

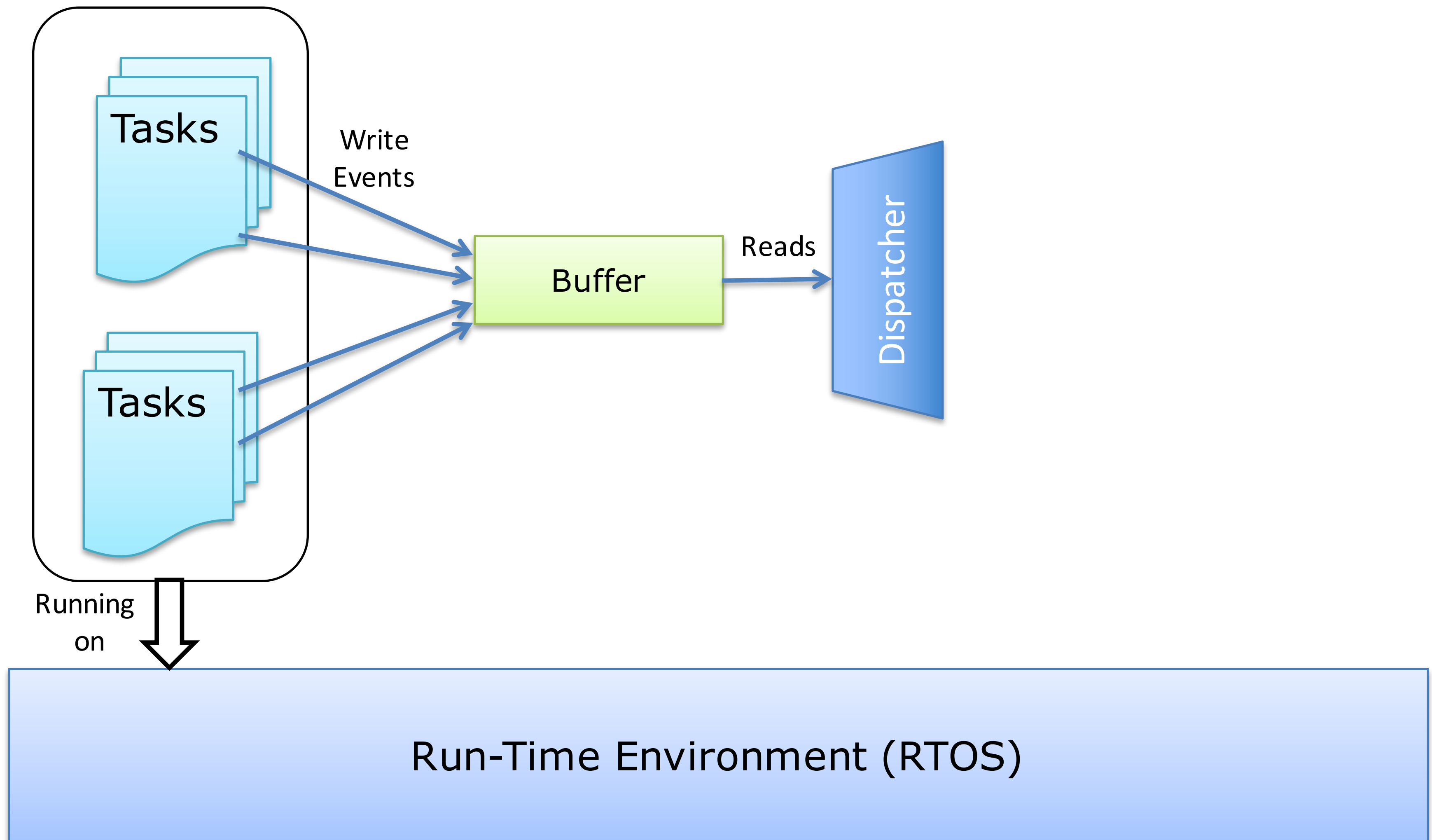
Framework Architecture



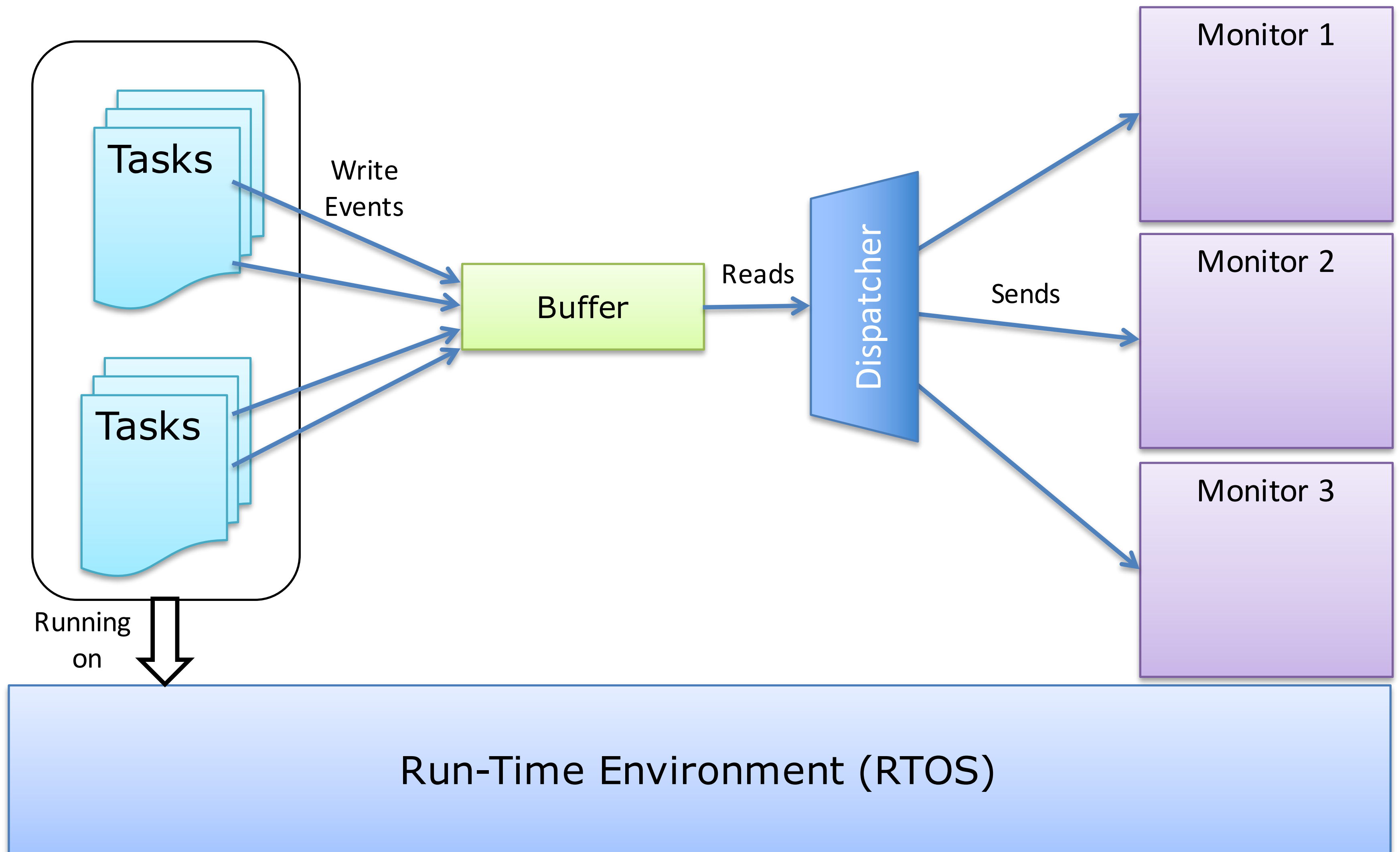
One Shared Buffer



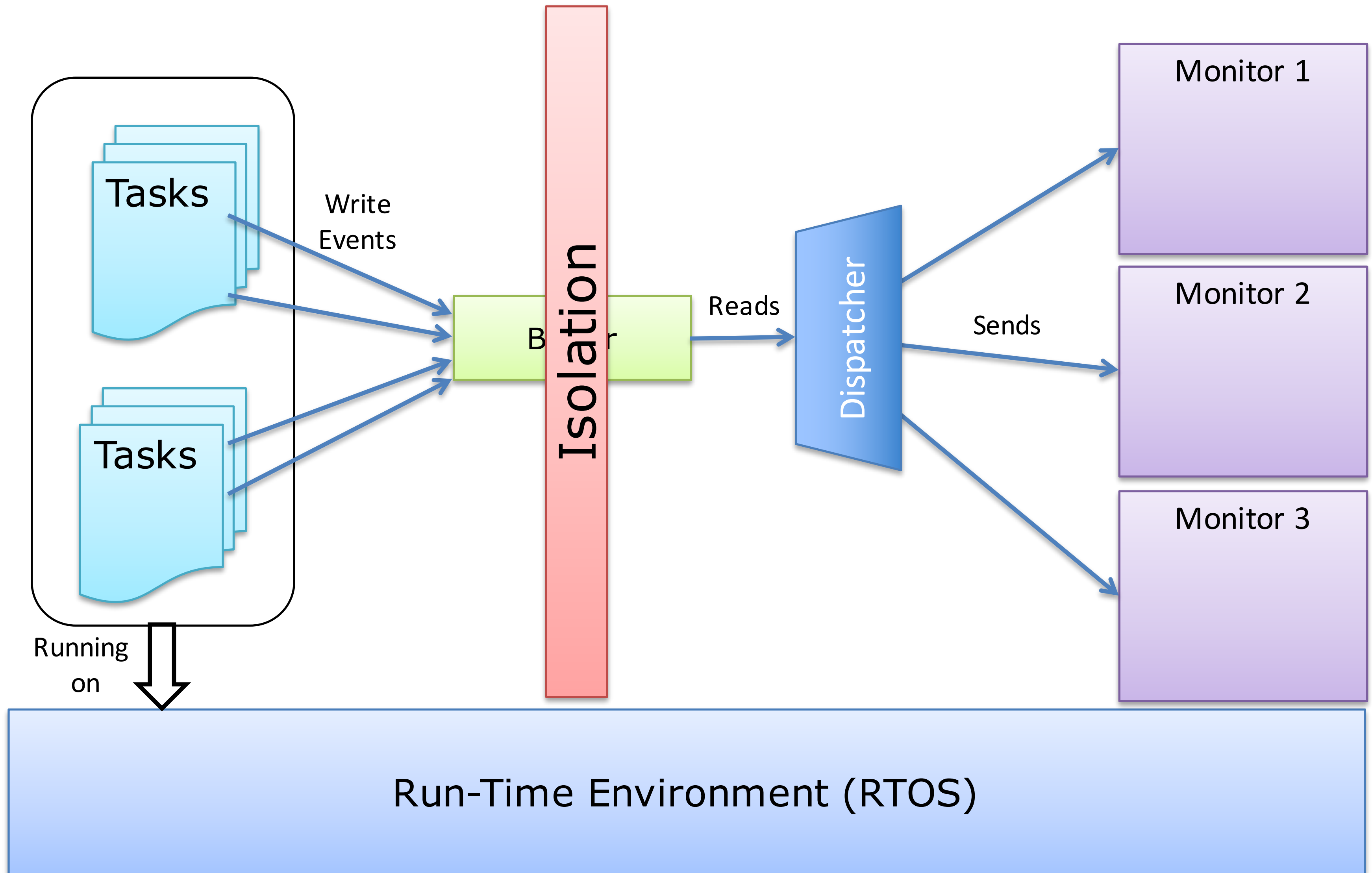
One Shared Buffer



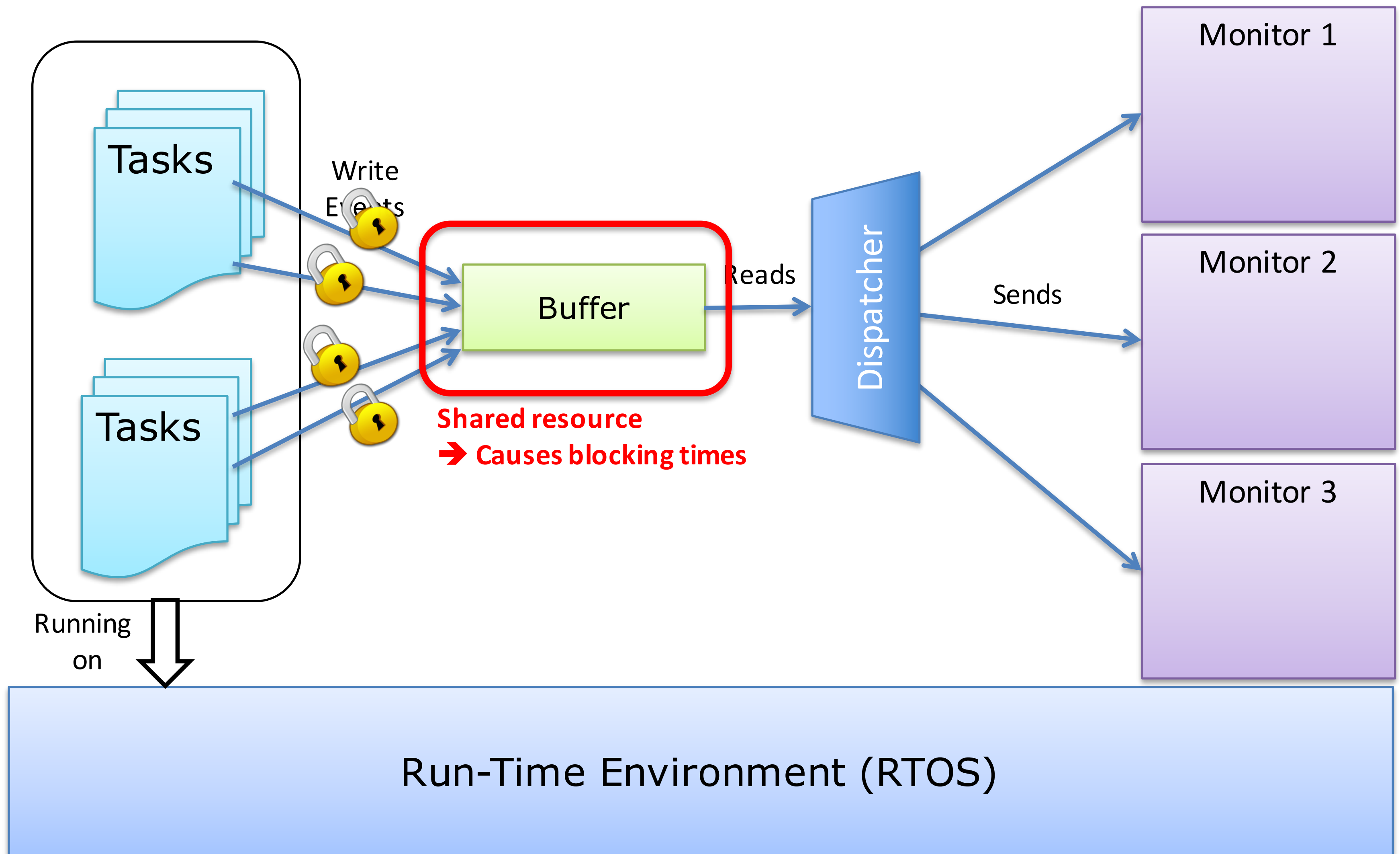
One Shared Buffer



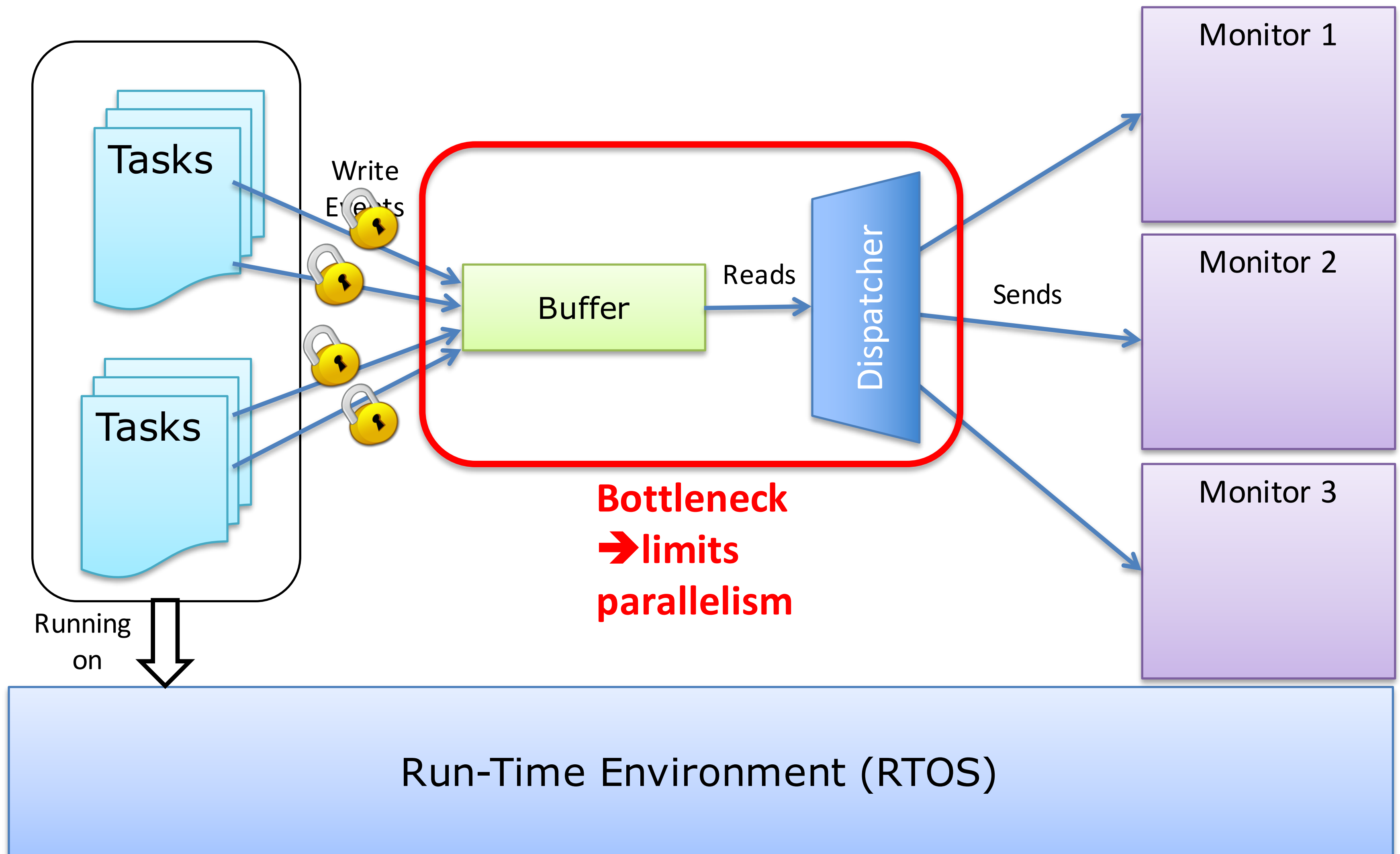
One Shared Buffer



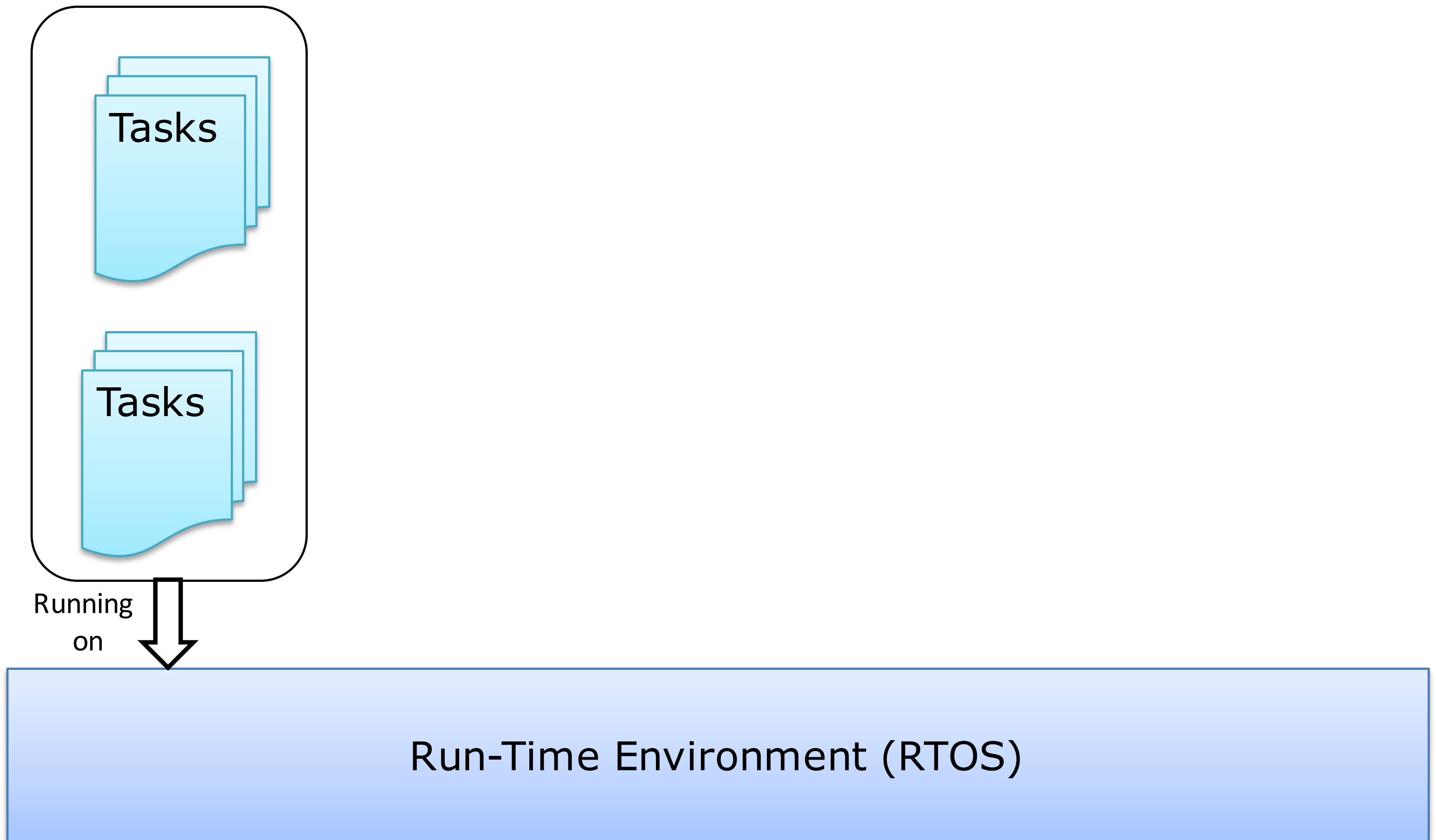
One Shared Buffer



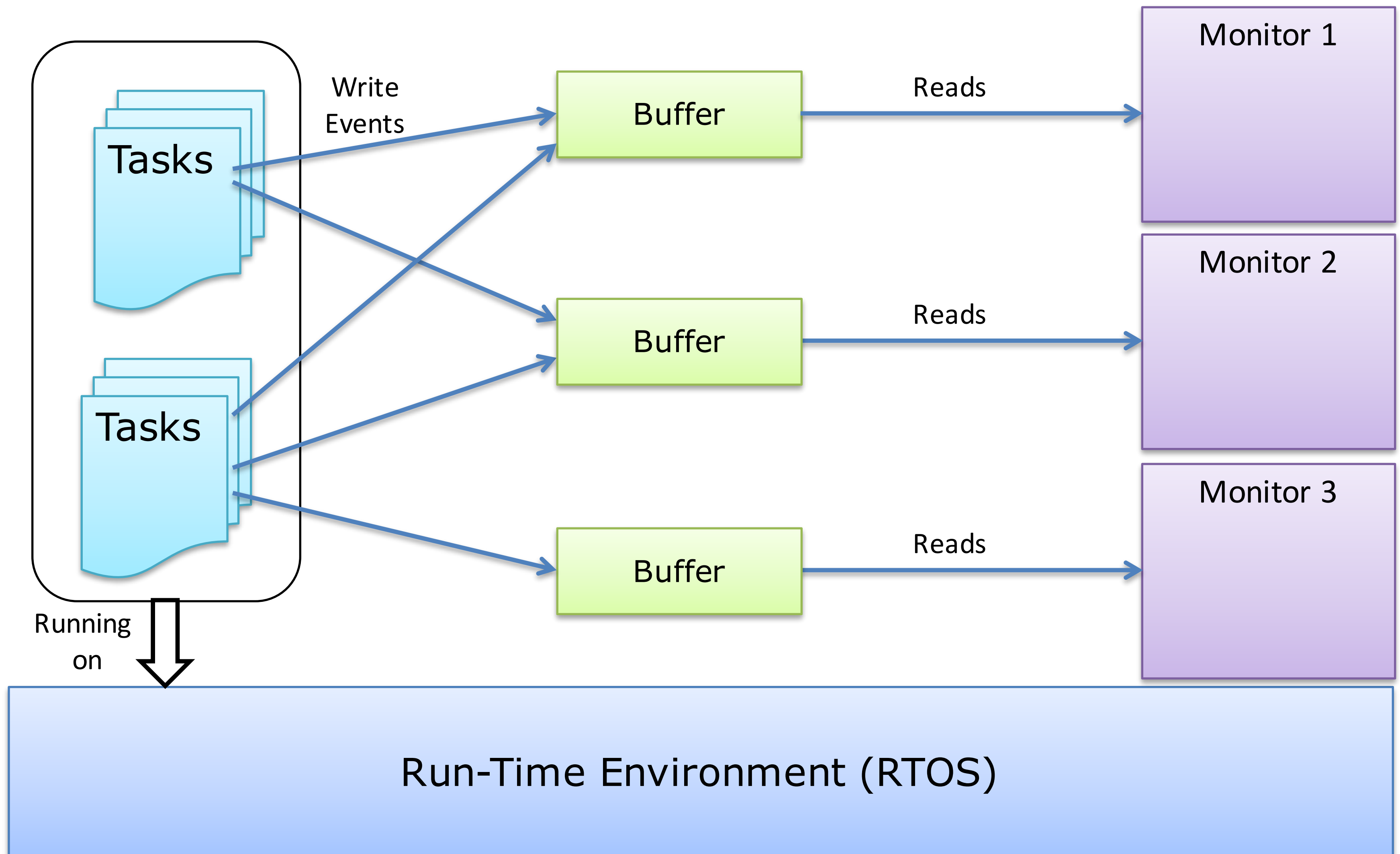
One Shared Buffer



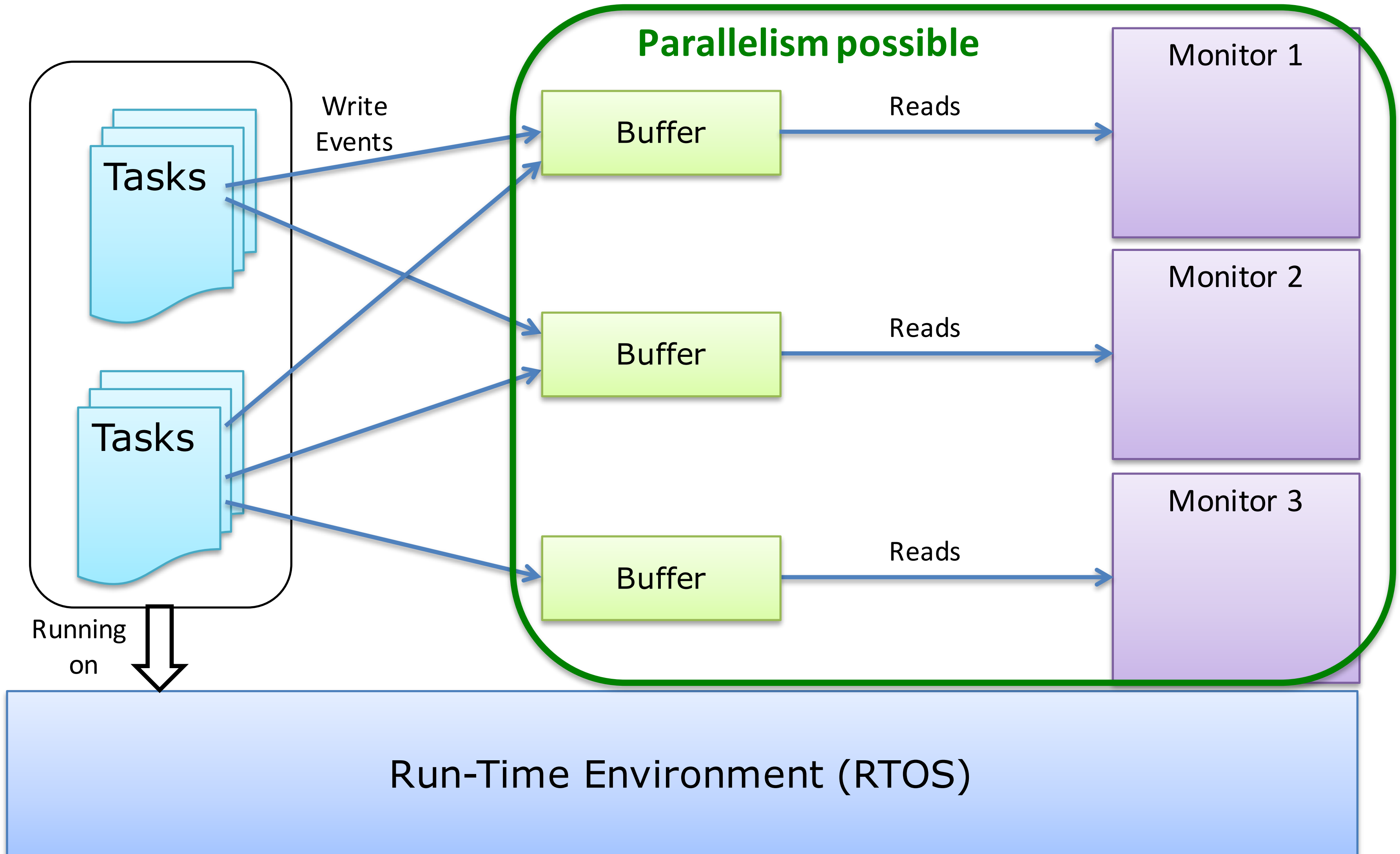
One Buffer per Monitor



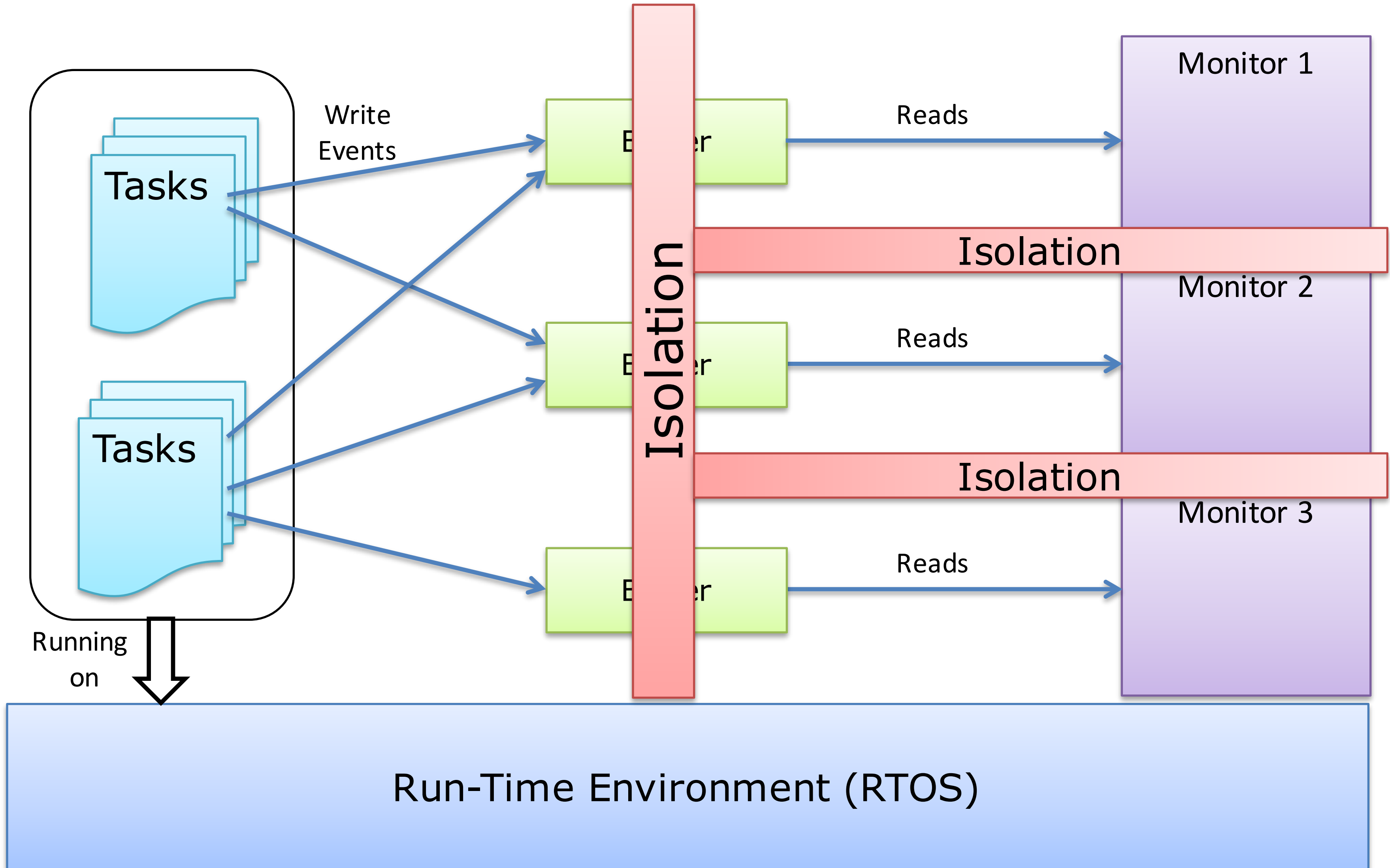
One Buffer per Monitor



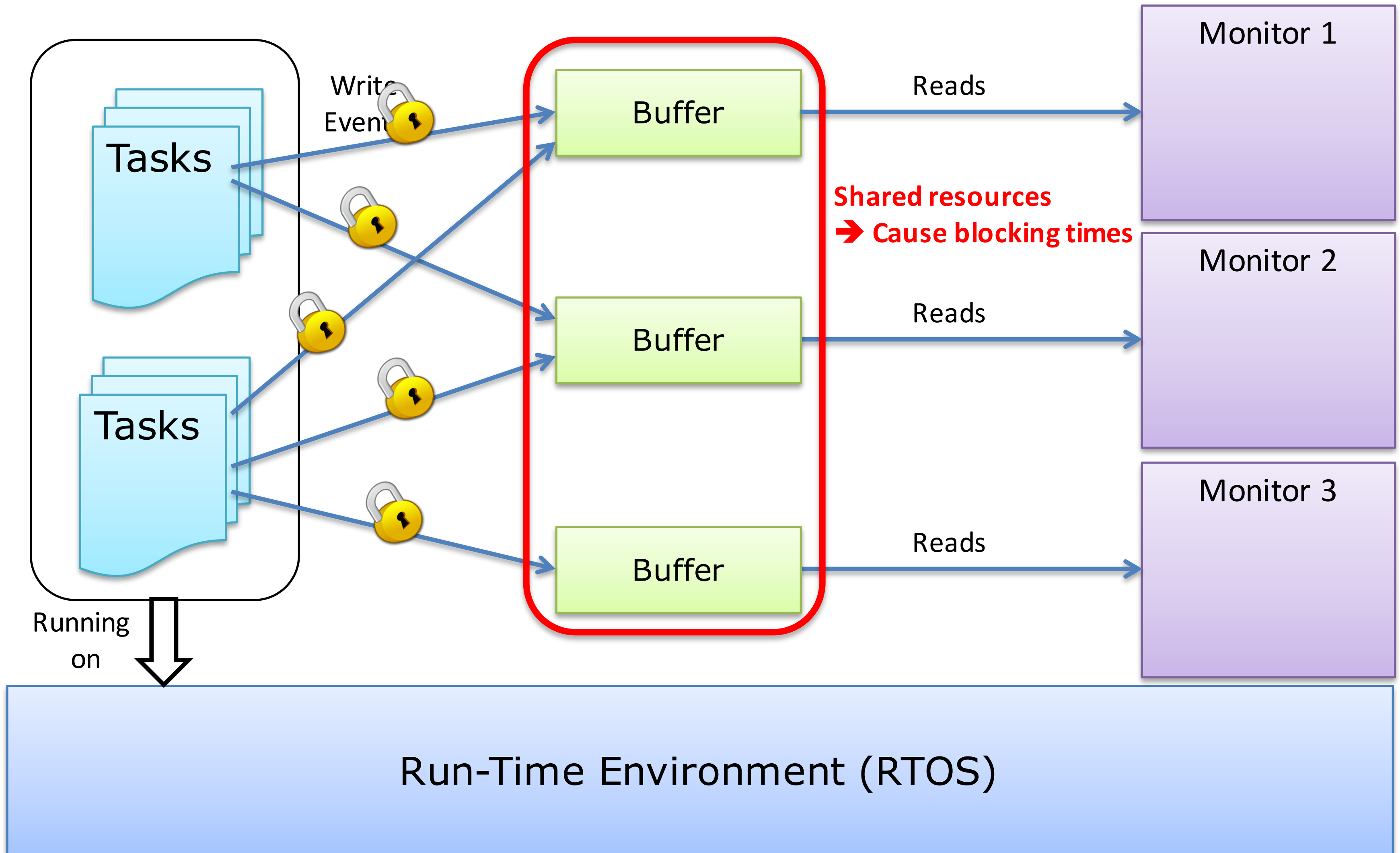
One Buffer per Monitor



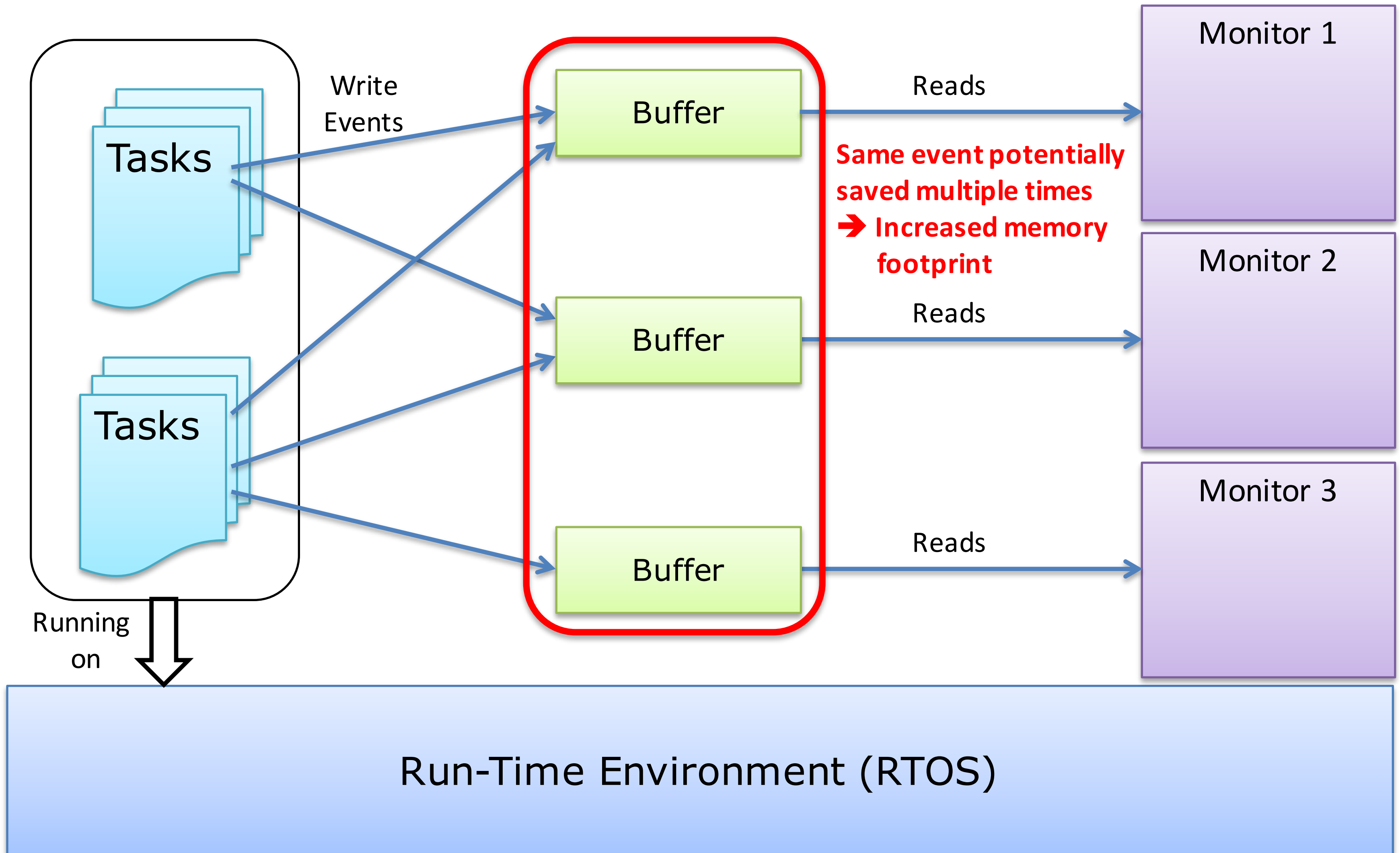
One Buffer per Monitor



One Buffer per Monitor



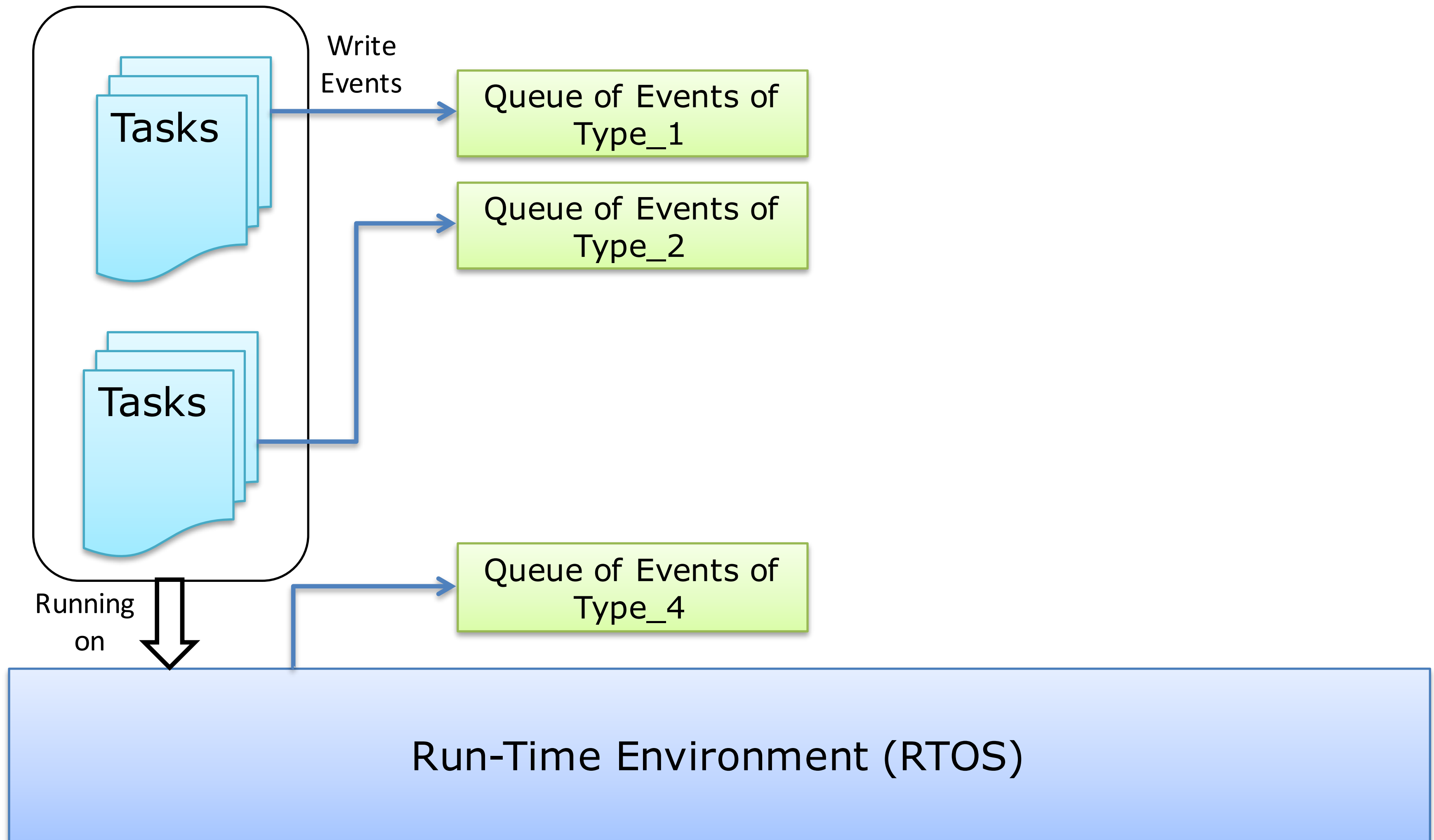
One Buffer per Monitor



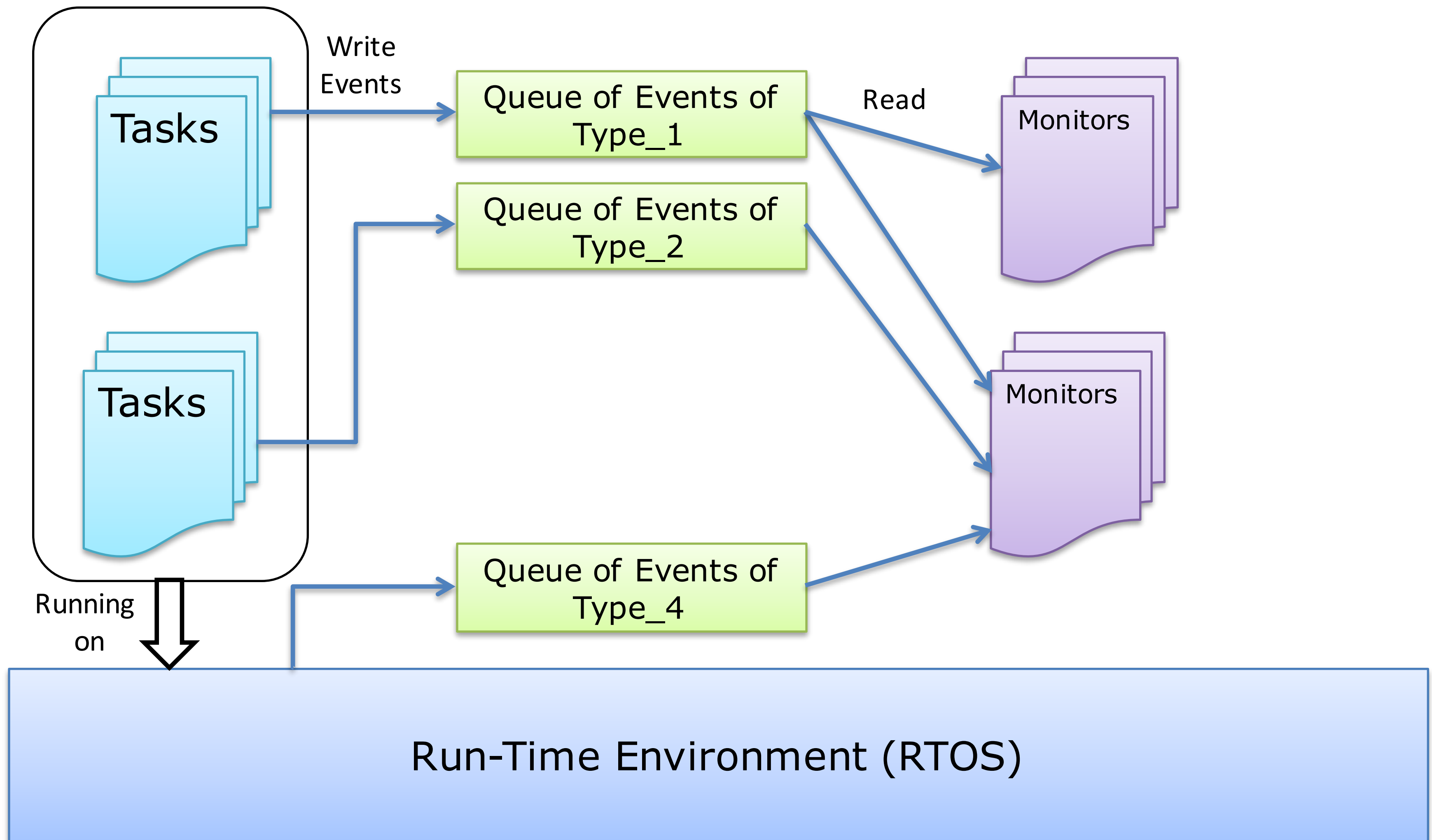
A NOVEL RUN-TIME MONITORING ARCHITECTURE

One Buffer per Event Type

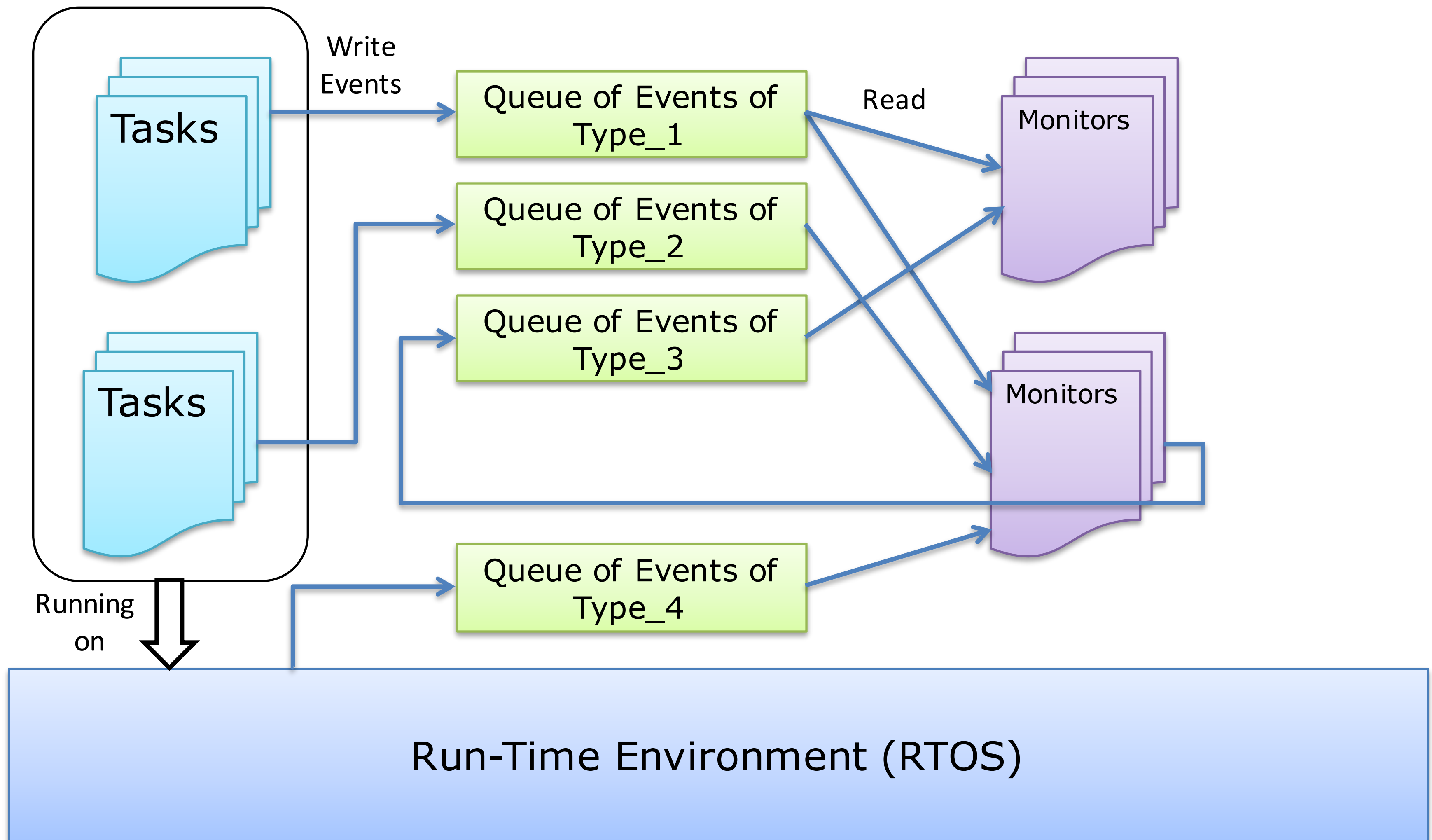
One Buffer per Event Type



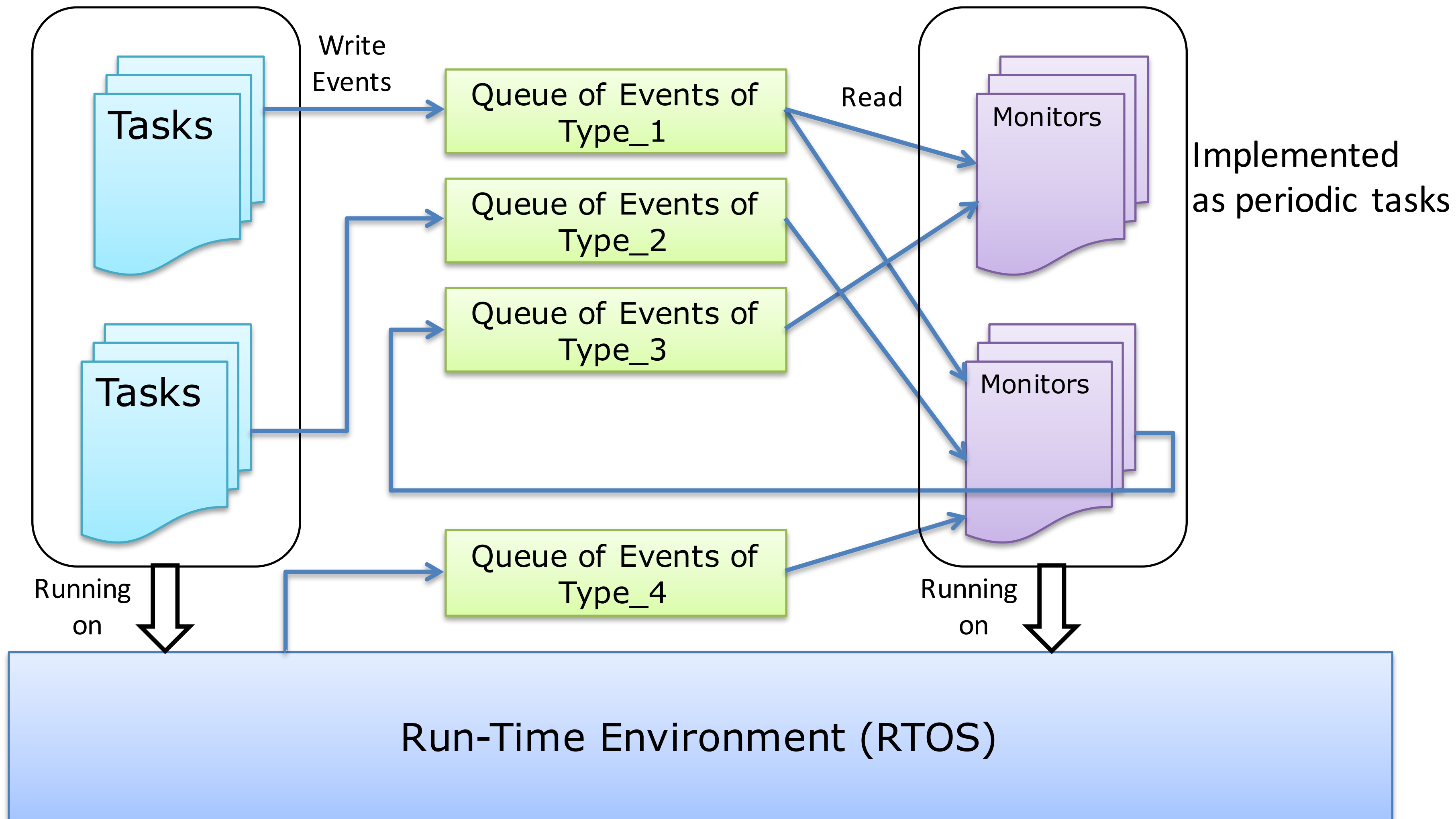
One Buffer per Event Type



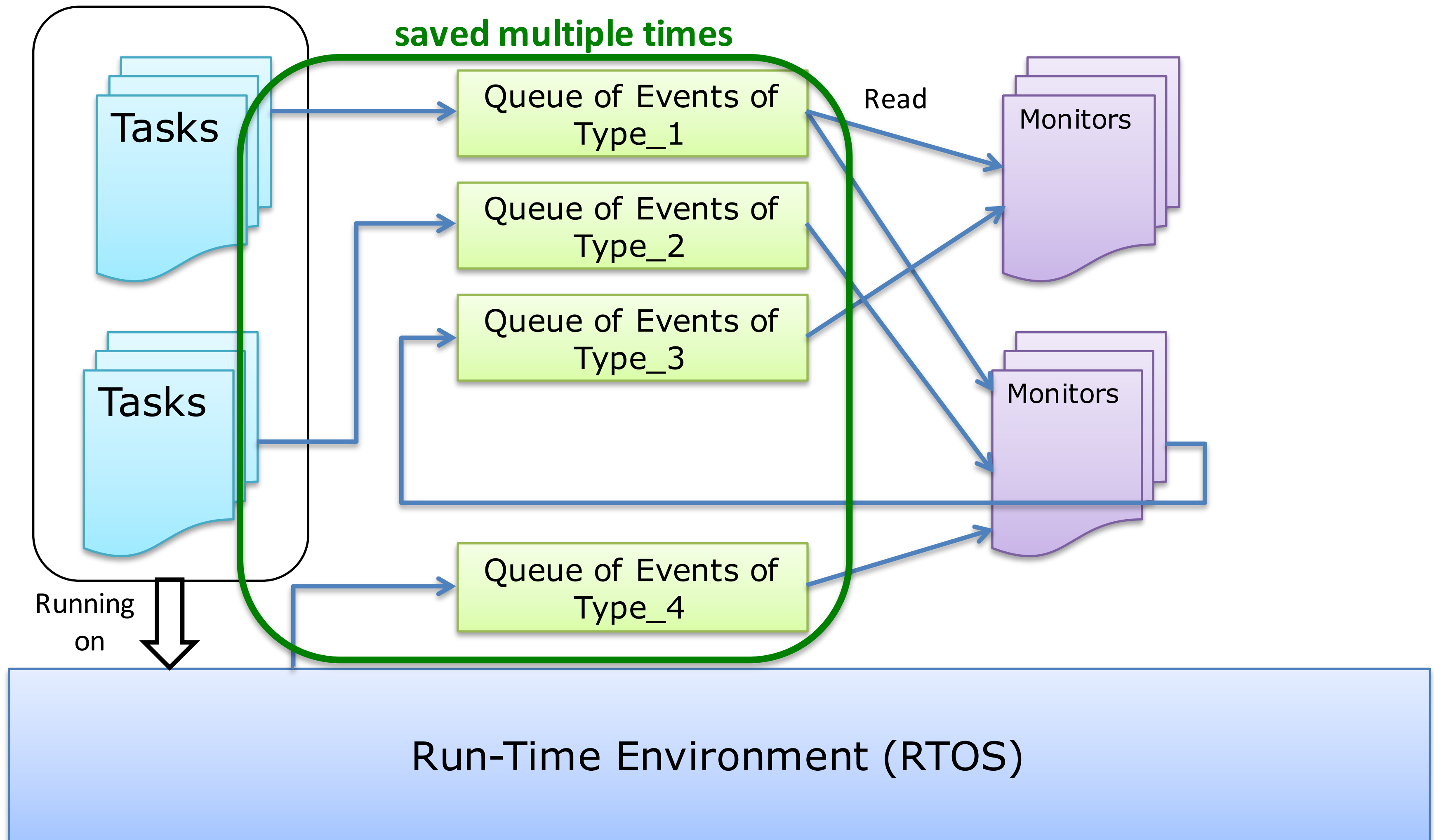
One Buffer per Event Type



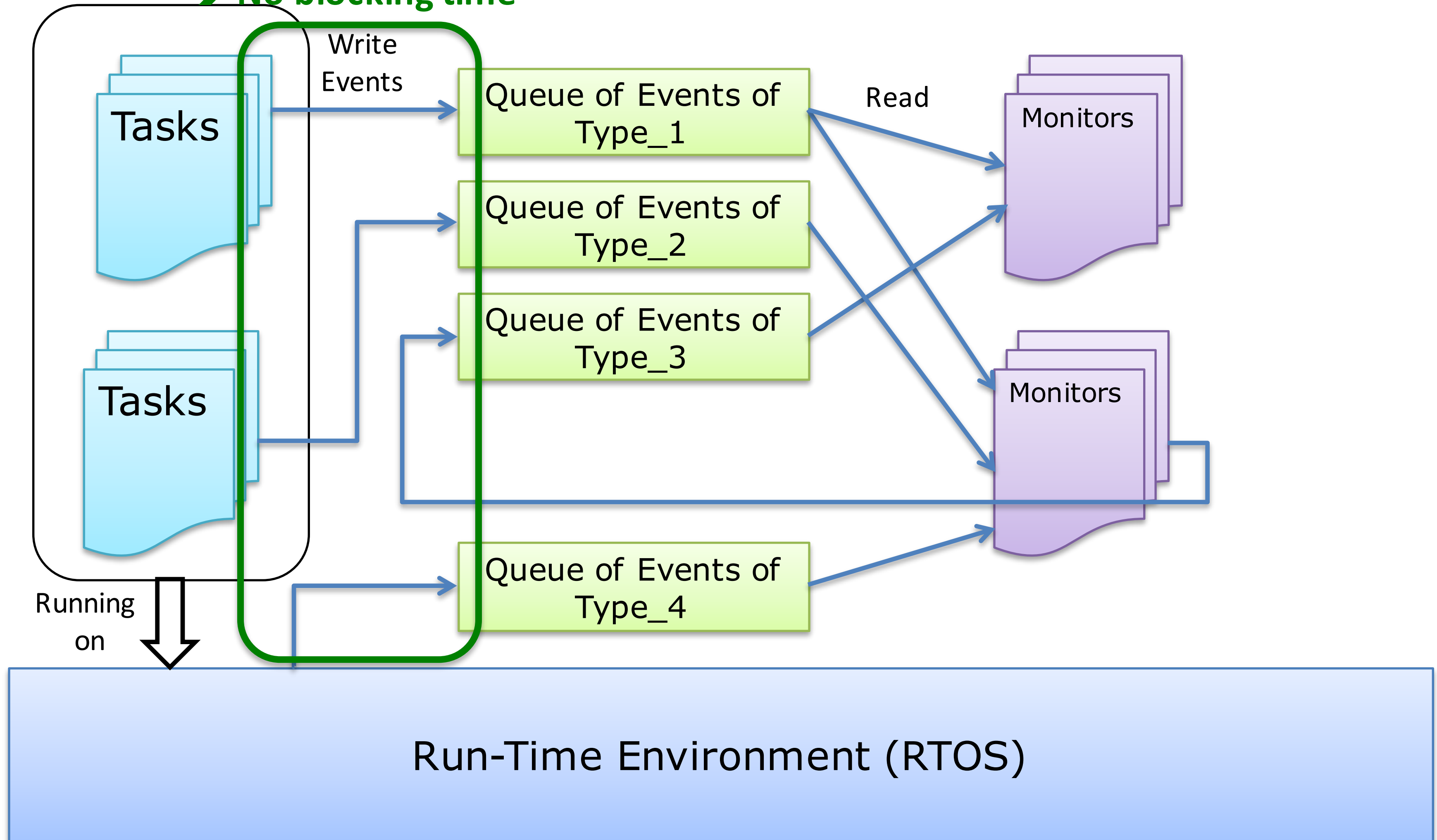
One Buffer per Event Type



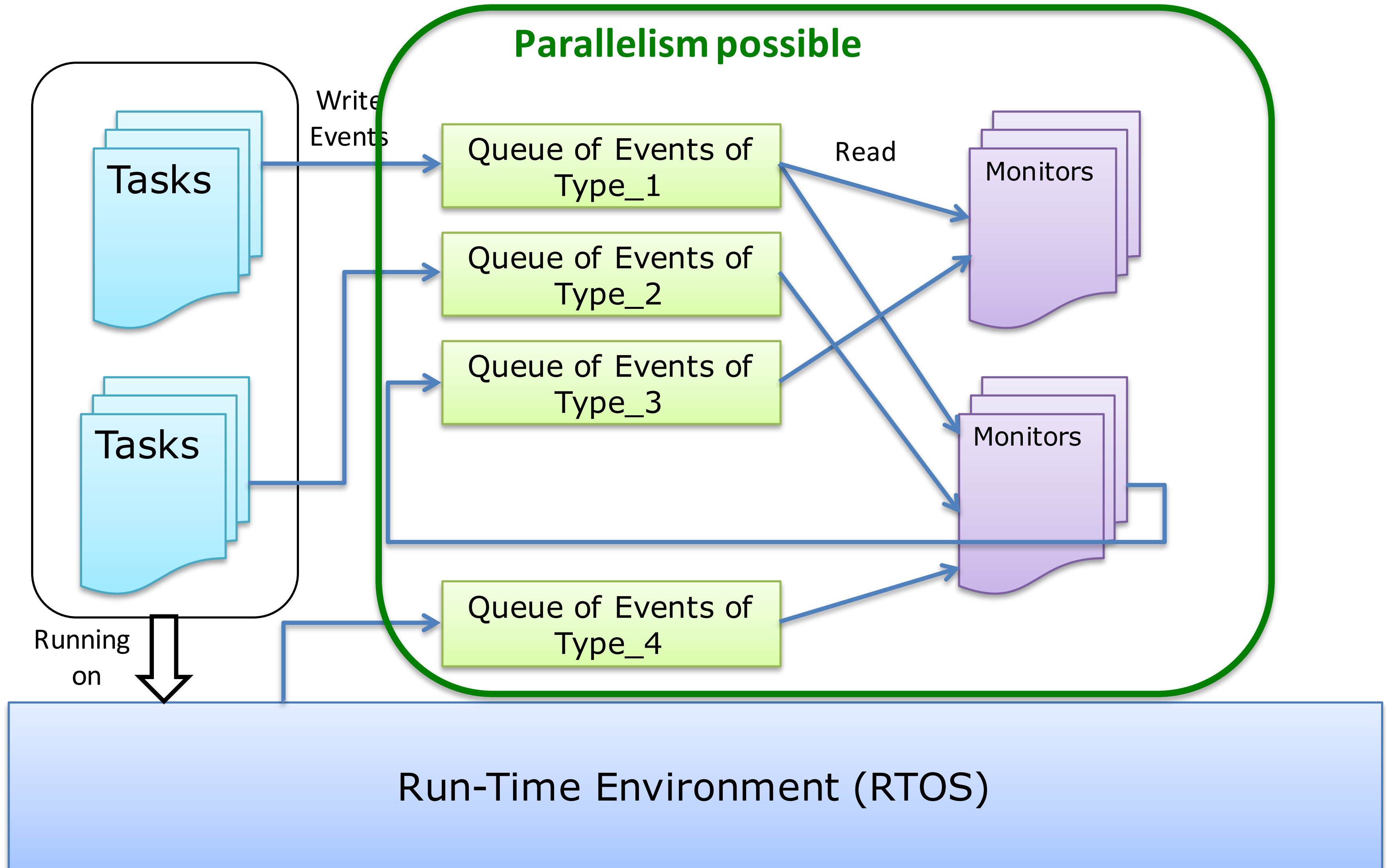
One Buffer per Event Type



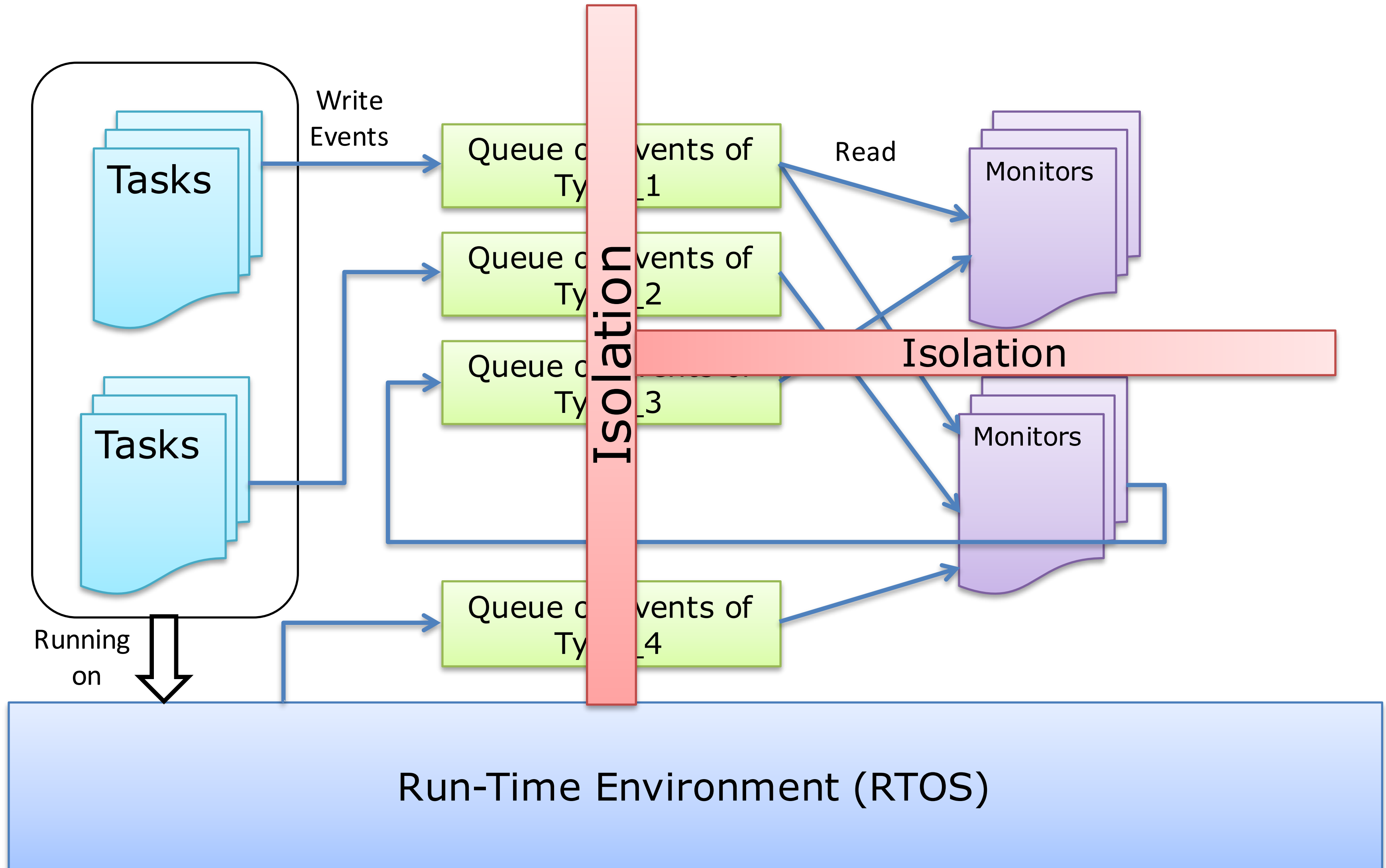
One Buffer per Event Type



One Buffer per Event Type



One Buffer per Event Type



One Buffer per Event Type

- Events of different types used by a same monitor are not ordered
  - ➔The monitor must reorder them
  - ➔Does not require more reads than when there is only one buffer per monitor

One Buffer per Event Type

- If T_i is the period of the monitor, any error is detected in strictly less than

$$R_i = 2 \times T_i$$

➔The responsiveness can be configured

- Assuming the system is schedulable

A RUN-TIME VERIFICATION FRAMEWORK FOR REAL-TIME SYSTEMS

How to Generate Monitors?

- Programming them **by hand**?

How to Generate Monitors?

- Programming them **by hand**?
  - ➔It may be complex to capture all possible cases
  - ➔Possibility to introduce bugs in the monitor
  - ➔Difficult to prove their correctness
  - ➔Hardly ease the certification process

How to Generate Monitors?

- Programming them **by hand**?
  - ➔ It may be complex to capture all possible cases
  - ➔ Possibility to introduce bugs in the monitor
  - ➔ Difficult to prove their correctness
  - ➔ Hardly ease the certification process

- Solution: Rely on
  - high level formal specification languages
  - Correct-by-construction monitor generation

High-Level Formal Specification Language

- Several existing tools
  - Mac: extended version of regular expressions
  - Eagle, Hawk: temporal logic
  - RuleR: formal rule based system
  - Java-MOP: multi-language → regular expressions, temporal logic, rule based, finite state machines
  - …

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
        static int counter = 0;
        int writes = 0;

        event open after() returning(FileWriter f) :
                call(FileWriter.new(..)) {
                        this.writes = 0;
                }

        event write before(FileWriter f) :
                call(* write(..)) && target(f) {
                        this.writes ++;
                }

        event close after(FileWriter f) :
                 call(* close(..)) && target(f) {}
```

```
        ere : (open write* close)*

        @fail {
                System.out.println("write  after close");
                __RESET;
        }
        @match {
                System.out.println((++(counter))
                        + ":" + writes);
        }
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
        static int counter = 0;
        int writes = 0;

        event open after() returning(FileWriter f) :
                call(FileWriter.new(..)) {
                        this.writes = 0;
                }

        event write before(FileWriter f) :
                call(* write(..)) && target(f) {
                        this.writes ++;
                }

        event close after(FileWriter f) :
                 call(* close(..)) && target(f) {}
```

Name of the monitor

```
        ere : (open write* close)*

        @fail {
                System.out.println("write  after close");
                __RESET;
        }
        @match {
                System.out.println((++(counter))
                        + ":" + writes);
        }
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
    static int counter = 0;
    int writes = 0;

    event open after() returning(FileWriter f) :
        call(FileWriter.new(..)) {
            this.writes = 0;
        }

    event write before(FileWriter f) :
        call(* write(..)) && target(f) {
            this.writes ++;
        }

    event close after(FileWriter f) :
        call(* close(..)) && target(f) {}
```

Declaration of internal variables

```
    ere : (open write* close)*

    @fail {
        System.out.println("write  after close");
        __RESET;
    }
    @match {
        System.out.println((++(counter))
            + ":" + writes);
    }
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
	static int counter = 0;
	int writes = 0;

	event open after() returning(FileWriter f) :
		call(FileWriter.new(..)) {
			this.writes = 0;
		}

	event write before(FileWriter f) :
		call(* write(..)) && target(f) {
			this.writes ++;
		}

	event close after(FileWriter f) :
		 call(* close(..)) && target(f) {}
```

Declaration of the events used by the monitor

```
	ere : (open write* close)*

	@fail {
		System.out.println("write  after close");
		__RESET;
	}
	@match {
		System.out.println((++(counter))
			+ ":" + writes);
	}
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
        static int counter = 0;
        int writes = 0;

        event open after() returning(FileWriter f) :
                call(FileWriter.new(..)) {
                        this.writes = 0;
                }

        event write before(FileWriter f) :
                call(* write(..)) && target(f) {
                        this.writes ++;
                }

        event close after(FileWriter f) :
                 call(* close(..)) && target(f) {}
```

event **open** generated after a call to FileWriter.new(..)

```
        ere : (open write* close)*

        @fail {
                System.out.println("write after close");
                __RESET;
        }
        @match {
                System.out.println((++(counter))
                        + ":" + writes);
        }
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
	static int counter = 0;
	int writes = 0;

	event open after() returning(FileWriter f) :
		call(FileWriter.new(..)) {
			this.writes = 0;
		}

	event write before(FileWriter f) :
		call(* write(..)) && target(f) {
			this.writes ++;
		}

	event close after(FileWriter f) :
		 call(* close(..)) && target(f) {}
```

Specification

```
	ere : (open write* close)*

	@fail {
		System.out.println("write  after close");
		__RESET;
	}
	@match {
		System.out.println((++(counter))
			+ ":" + writes);
	}
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
	static int counter = 0;
	int writes = 0;

	event open after() returning(FileWriter f) :
		call(FileWriter.new(..)) {
			this.writes = 0;
		}

	event write before(FileWriter f) :
		call(* write(..)) && target(f) {
			this.writes ++;
		}

	event close after(FileWriter f) :
		 call(* close(..)) && target(f) {}
```

What must be done in case of respect or failure of the specification

```
	ere : (open write* close)*

	@fail {
		System.out.println("write after close");
		__RESET;
	}
	@match {
		System.out.println((++(counter))
			+ ":" + writes);
	}
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
	static int counter = 0;
	int writes = 0;

	event open after() returning(FileWriter f) :
		call(FileWriter.new(..)) {
			this.writes = 0;
		}

	event write before(FileWriter f) :
		call(* write(..)) && target(f) {
			this.writes ++;
		}

	event close after(FileWriter f) :
		 call(* close(..)) && target(f) {}
```

Can be used by existing tools (based on Aspect Oriented Programming) to automatically instrument the application code

```
	ere : (open write* close)*

	@fail {
		System.out.println("write  after close");
		__RESET;
	}
	@match {
		System.out.println((++(counter))
			+ ":" + writes);
	}
}
```

An Example with Java-MOP using Regular Expressions

```
package mop;
import java.io.*;
import java.util.*;

SafeFileWriter(FileWriter f) {
        static int counter = 0;
        int writes = 0;

        event open after() returning(FileWriter f) :
                call(FileWriter.new(..)) {
                        this.writes = 0;
                }

        event write before(FileWriter f) :
                call(* write(..)) && target(f) {
                        this.writes ++;
                }

        event close after(FileWriter f) :
                 call(* close(..)) && target(f) {}
```

Can automatically be translated in a finite state machine.

And than in code implementing the monitor

```
        ere : (open write* close)*

        @fail {
                System.out.println("write after close");
                __RESET;
        }
        @match {
                System.out.println((++(counter))
                        + ":" + writes);
        }
}
```

Limitations of Existing Tools

- Limited notion of time
  - E.g., impossible for MOP to check the execution time of a job, or the jitter on a release period
- Most expressive tools are extremely complex
  - E.g., possible to express exec. time and jitter with RuleR at the cost of multiple recursive rules
- Do not generate code for a monitoring architecture suited for safety-critical systems

➔ Unsuited to real-time safety critical systems

WORK IN PROGRESS

Work in Progress

- Design of a new specification language (Sangeeth)
  - Suited to real-time safety critical systems
  - Easy to use for engineers
- Automatic generation of complex automata that describe the monitor behaviour based on the specifications (Sangeeth)
- Automatic code-generation for monitors from the generated automata (Sonia and Vedanshi in IIIT-D)
- Integration of the monitoring architecture as a service in an RTOS which is ARINC compliant (Humberto)