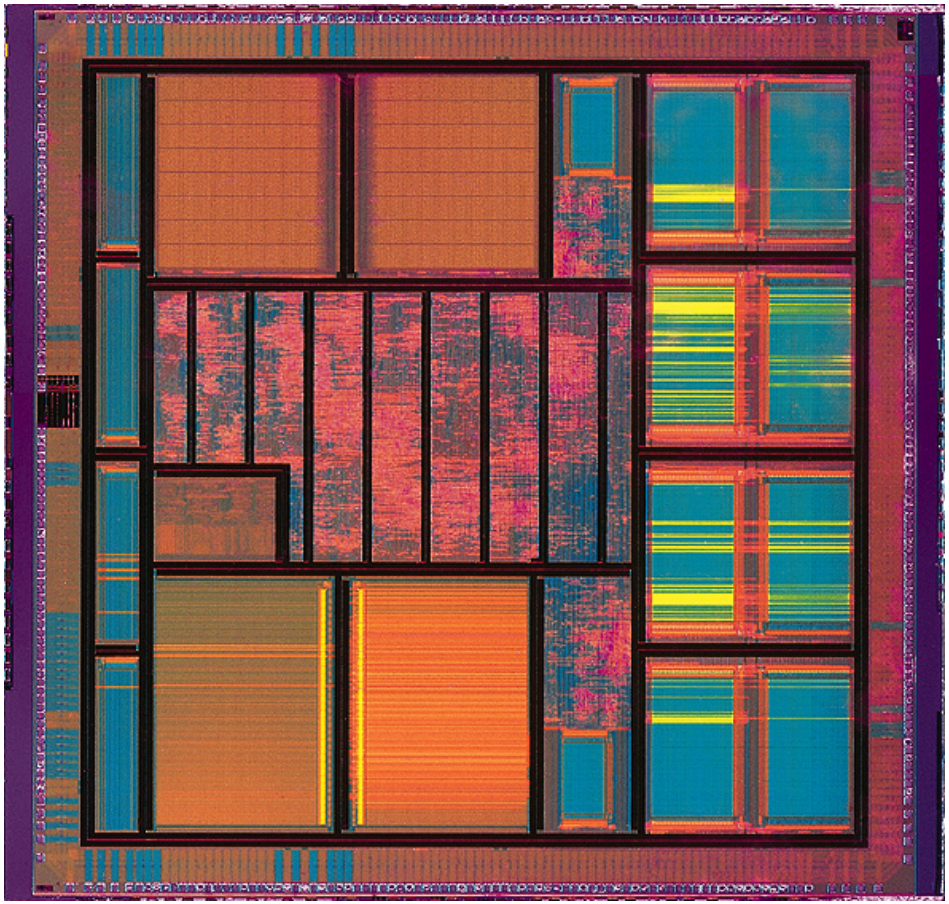
Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach

Matthieu Moy

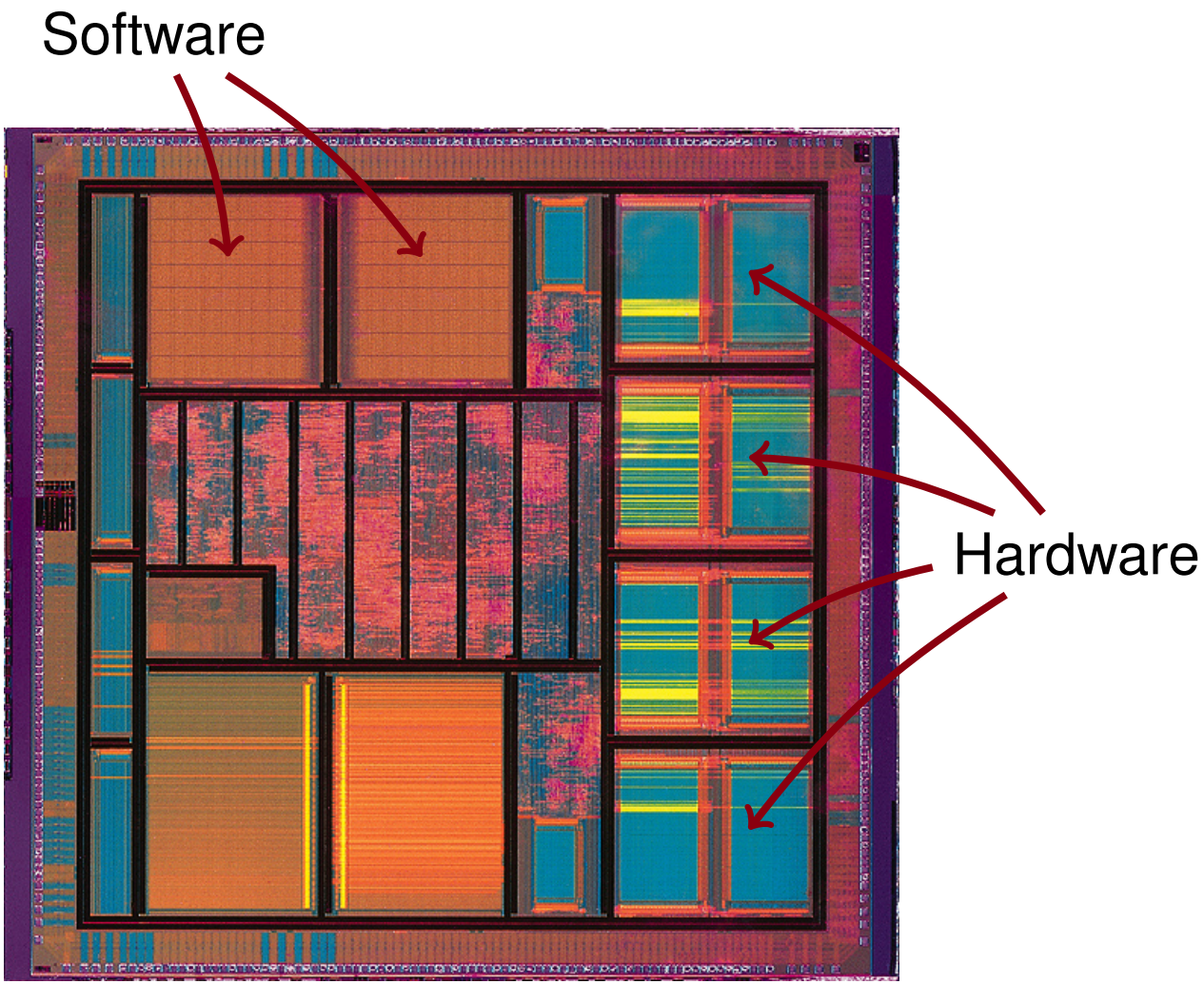
Grenoble University / Verimag
France

DATE, March 19th 2013

Modern Systems-on-a-Chip



Modern Systems-on-a-Chip



Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification

Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification
- Transaction-Level Modeling (TLM):
  - High level of abstraction
  - Suitable for

Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification
- Transaction-Level Modeling (TLM):
  - High level of abstraction
  - Suitable for

Industry Standard = SystemC/TLM

Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification
- Transaction-Level Modeling (TLM):
  - High level of abstraction
  - Suitable for

Industry Standard = SystemC/TLM

Issue: SystemC has co-routine semantics
$\Rightarrow$ hard to exploit host parallelism.

Outline

Issues and Solutions for Parallelization

1. Preserve SystemC scheduling semantics
2. Avoid introducing data-races (e.g. `i++` on shared variable)

Issues and Solutions for Parallelization

1. Preserve SystemC scheduling semantics
   (a) Parallelization within instant/δ-cycle
   (b) Optimistic approaches: require specific coding style
2. Avoid introducing data-races (e.g. `i++` on shared variable)

Issues and Solutions for Parallelization

1. Preserve SystemC scheduling semantics
   (a) Parallelization within instant/δ-cycle
   (b) Optimistic approaches: require specific coding style
2. Avoid introducing data-races (e.g. `i++` on shared variable)
   (c) Assume no shared variables
   (d) Semantics-preserving: don't run two processes accessing the same variables

Issues and Solutions for Parallelization

1. Preserve SystemC scheduling semantics
   (a) Parallelization within instant/δ-cycle
   (b) Optimistic approaches: require specific coding style
2. Avoid introducing data-races (e.g. `i++` on shared variable)
   (c) Assume no shared variables
   (d) Semantics-preserving: don't run two processes accessing the same variables

- Most approaches work for RTL/cycle-accurate:
  - Clocks $\Rightarrow$ many processes executing at the same time (a)
  - `sc_signal` $\Rightarrow$ avoids shared variables (c)

Issues and Solutions for Parallelization

1. Preserve SystemC scheduling semantics
   (a) Parallelization within instant/δ-cycle
   (b) Optimistic approaches: require specific coding style
2. Avoid introducing data-races (e.g. `i++` on shared variable)
   (c) Assume no shared variables
   (d) Semantics-preserving: don't run two processes accessing the same variables

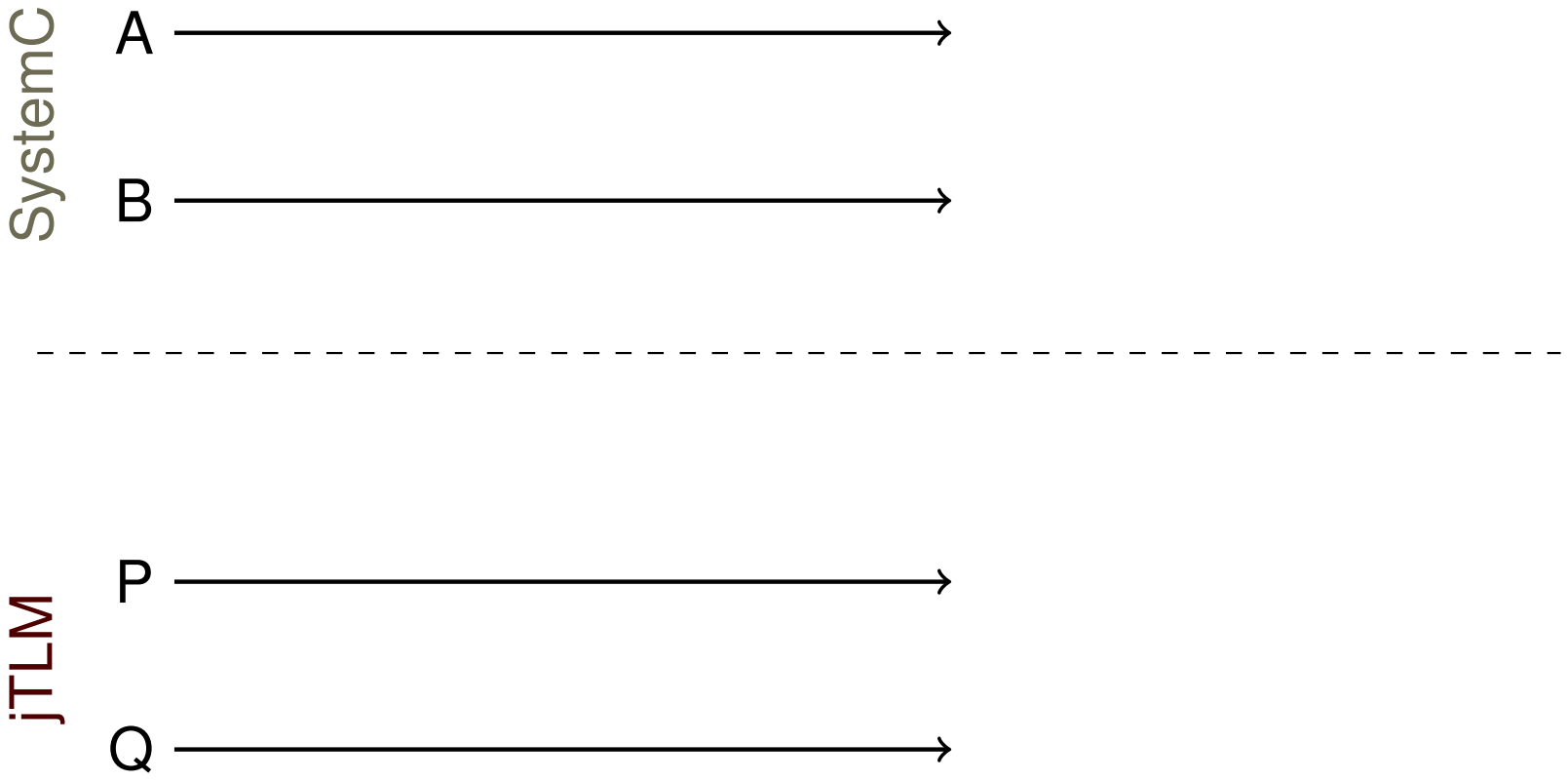
- Most approaches work for RTL/cycle-accurate:
  - Clocks $\Rightarrow$ many processes executing at the same time (a)
  - `sc_signal` $\Rightarrow$ avoids shared variables (c)
- Problems with loosely timed TLM:
  - Loose timing $\Rightarrow$ few processes runnable at the same time (a)
  - Communication using function calls $\Rightarrow$ many shared variables (c)
  - Few `wait` statements $\Rightarrow$ 1 SystemC transition touches many variables (d)

Our Approach: SC-DURING

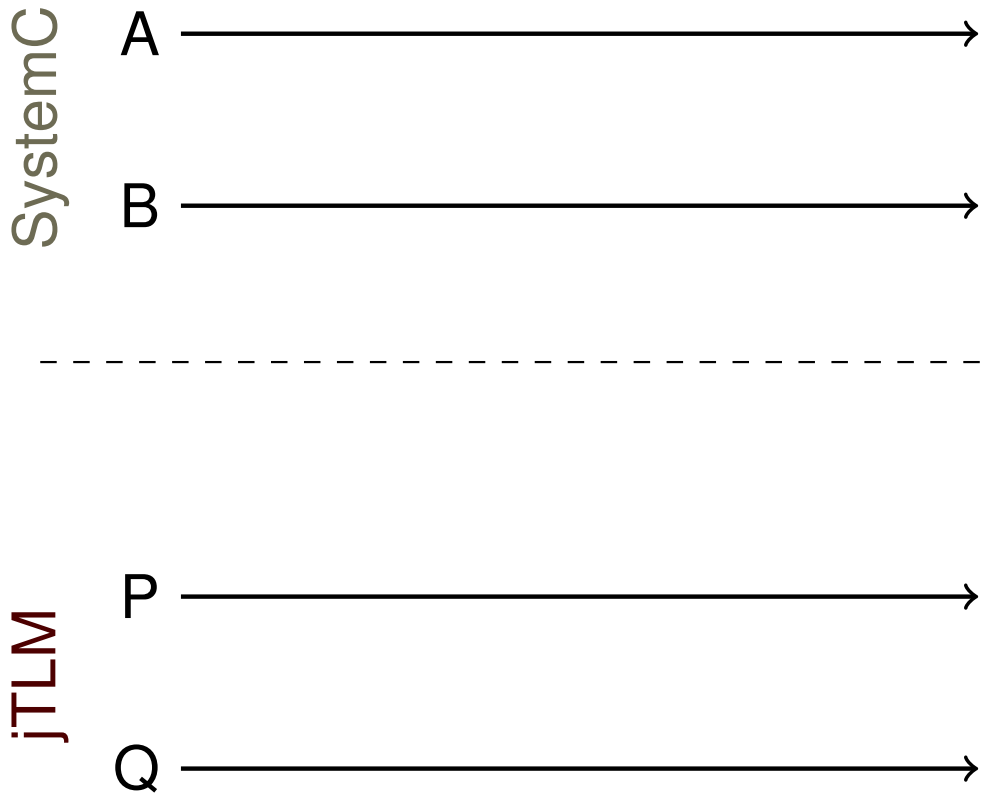
- Goals:
  - Efficient for loosely timed SystemC/TLM
  - Existing code should continue working
  - Work with any SystemC implementation
- Principle:
  - Library (adds constructs, feel free not to use them)
  - Notion of *duration*
- Source of inspiration: jTLM (Java simulator)

Outline

(Simulated) Time in SystemC and jTLM



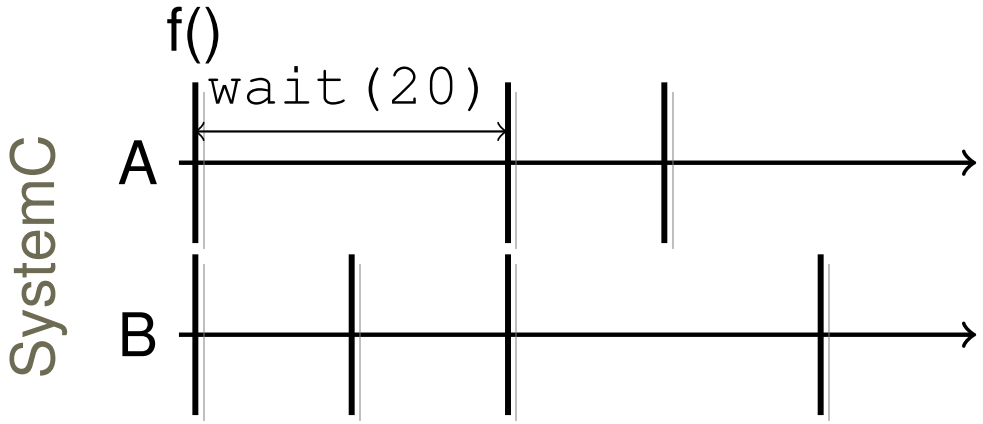
(Simulated) Time in SystemC and jTLM



Process A:

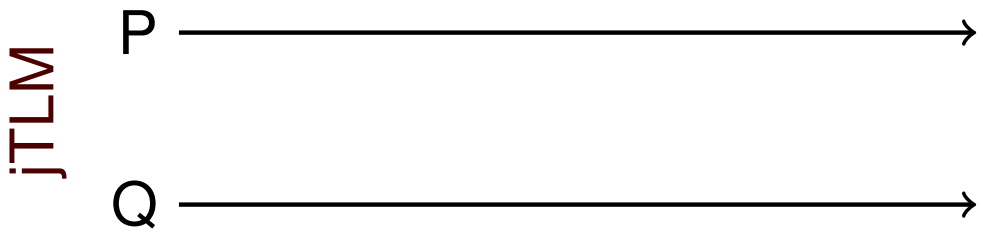
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

(Simulated) Time in SystemC and jTLM

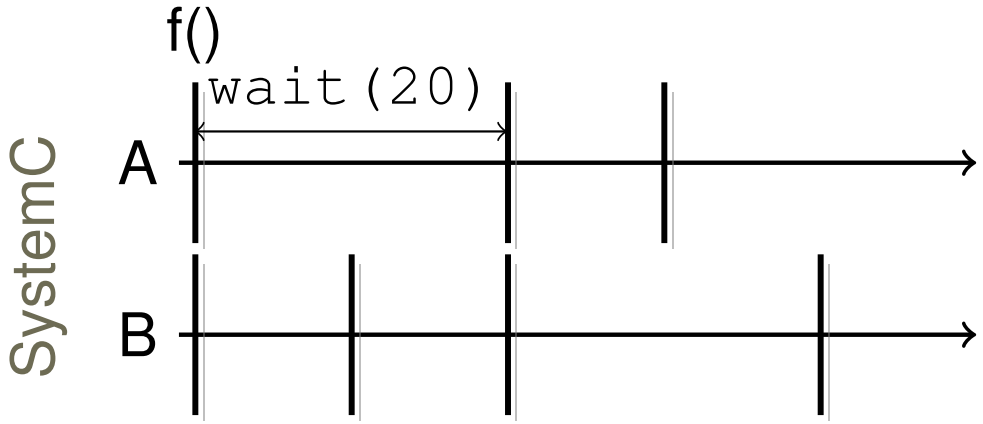


Process A:

```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

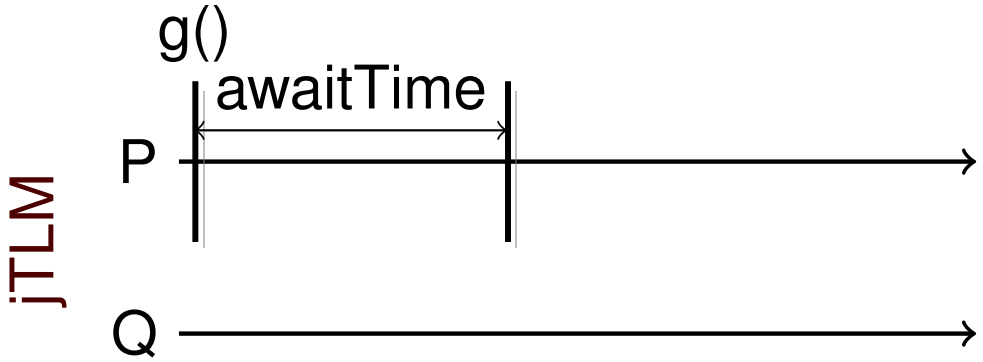


(Simulated) Time in SystemC and jTLM



Process A:

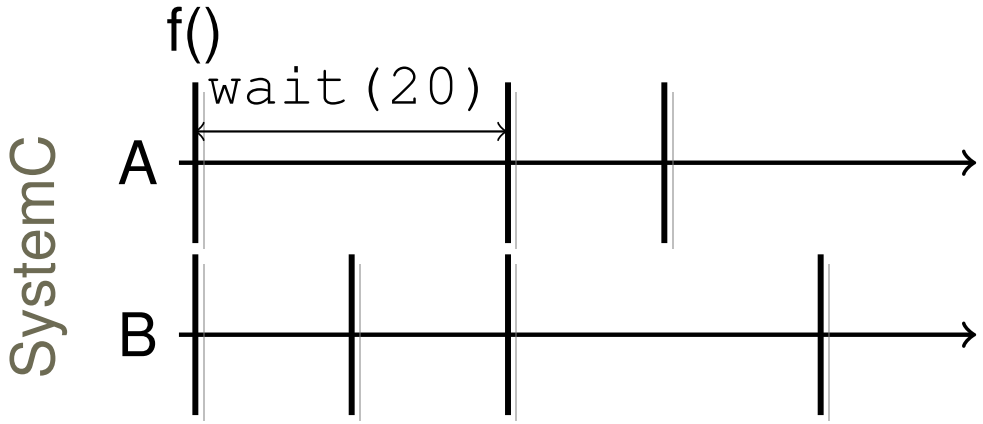
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```



Process P:

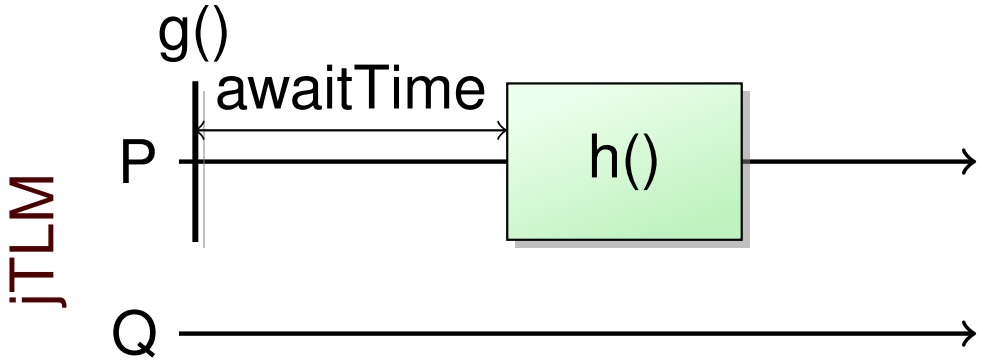
```
g();
awaitTime(20);
```

(Simulated) Time in SystemC and jTLM



Process A:

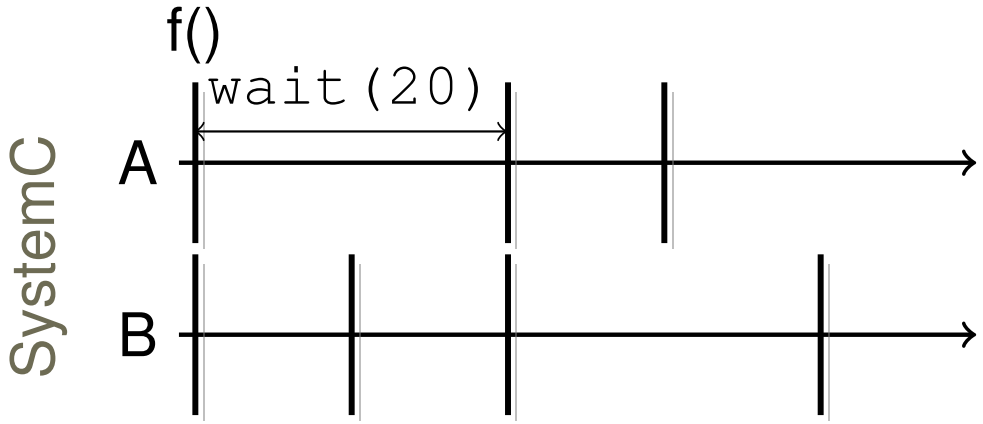
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```



Process P:

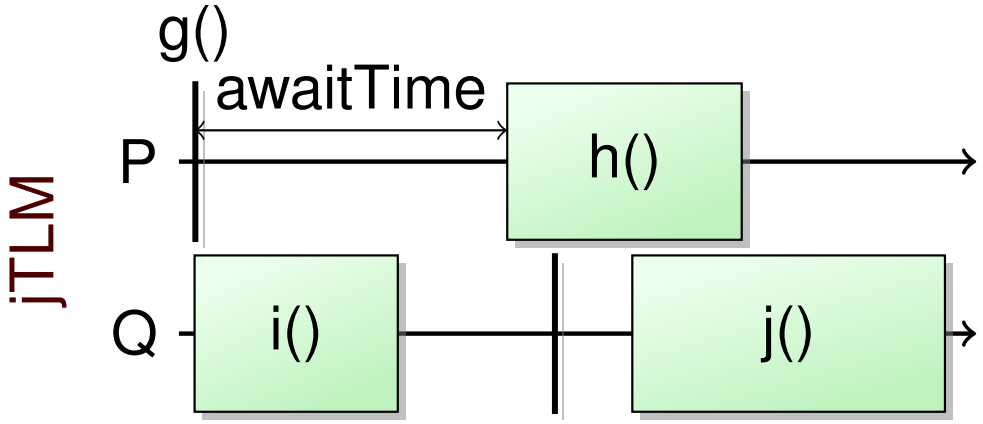
```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```

(Simulated) Time in SystemC and jTLM



Process A:

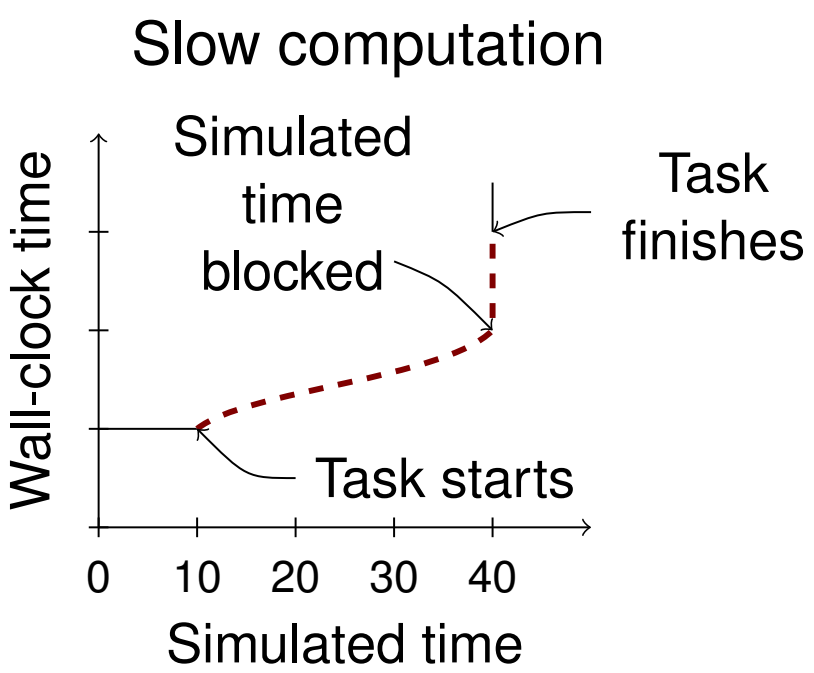
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

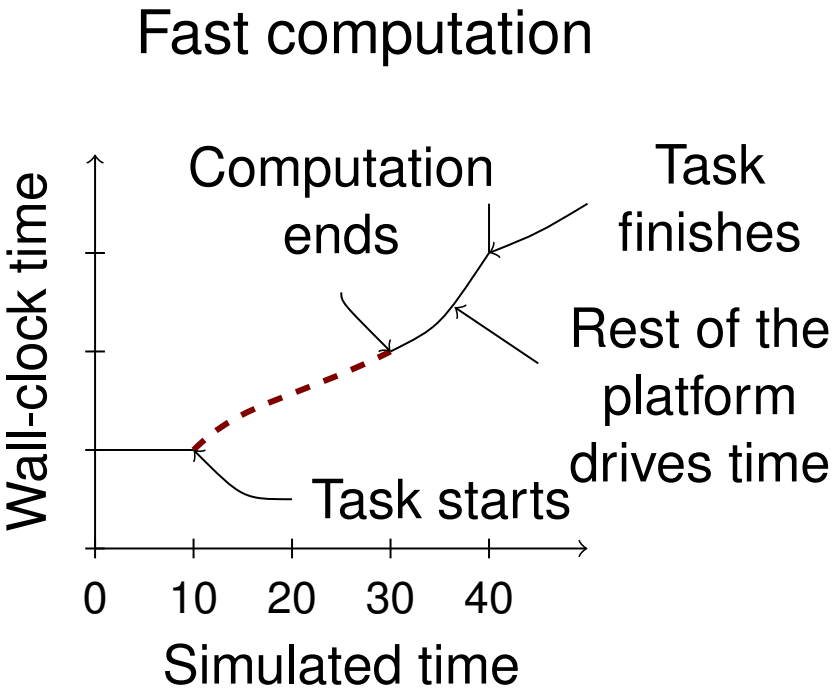


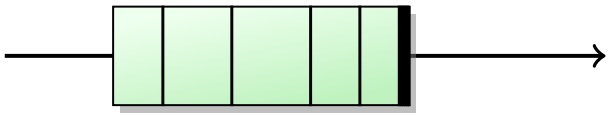
Process P:

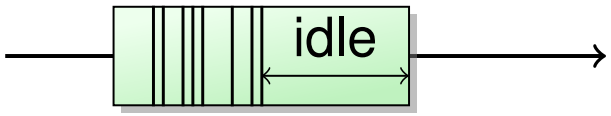
```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```

Execution of `consumesTime(T)`

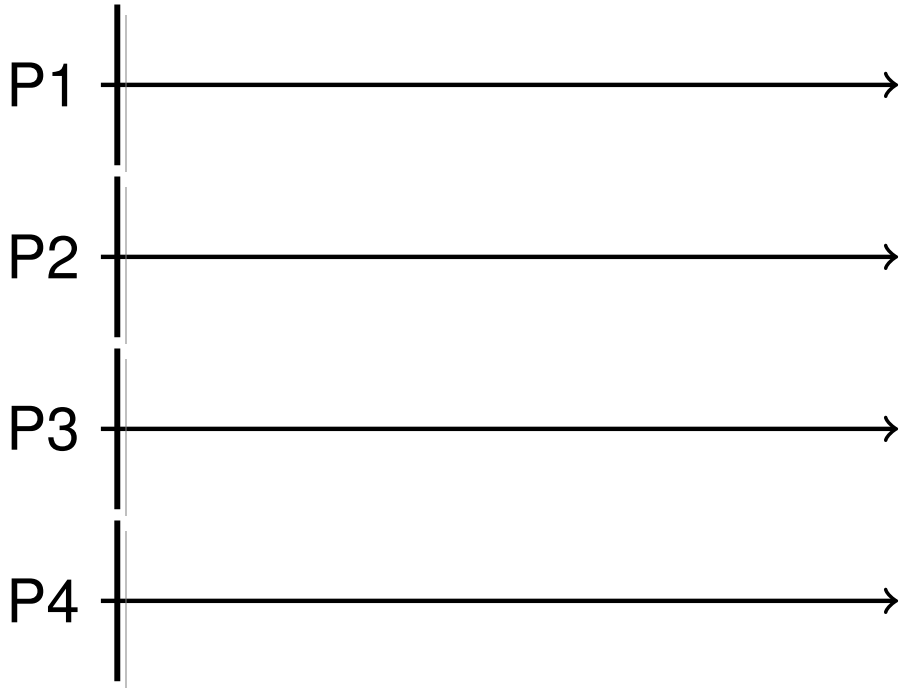








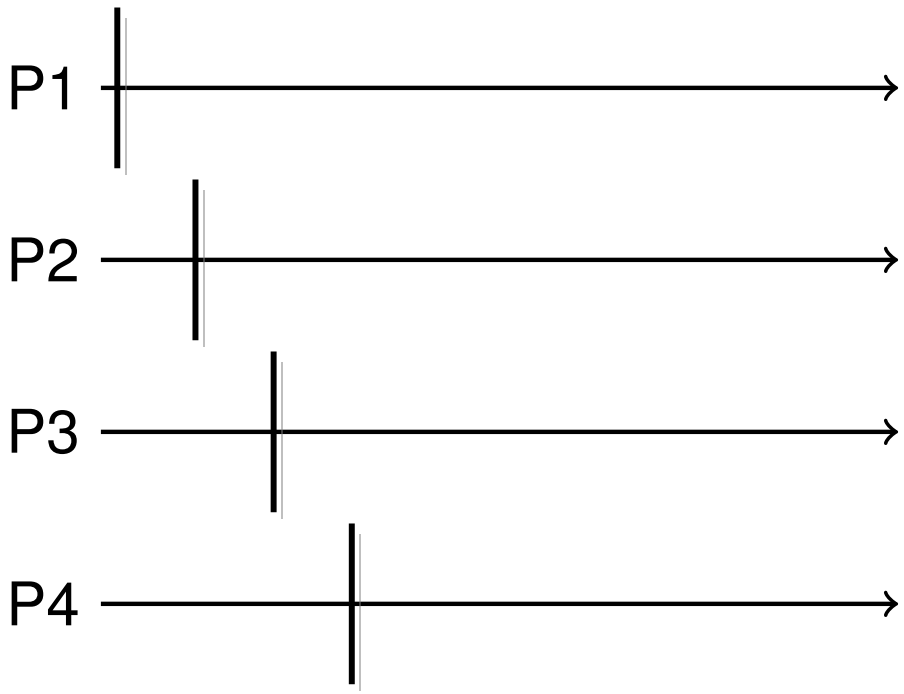
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel

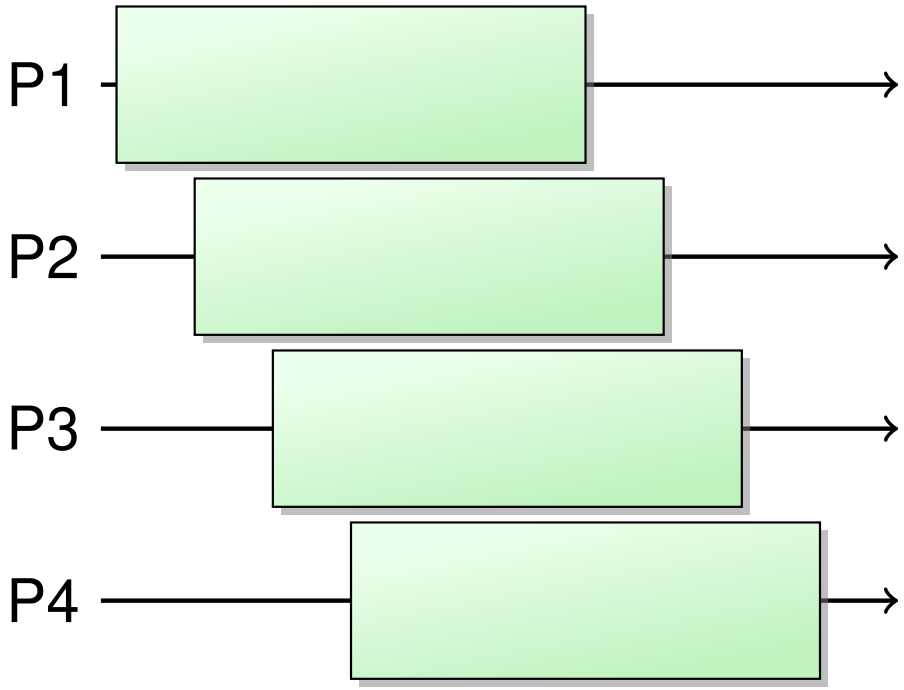
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't

Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't
- Overlapping tasks do

Outline

jTLM is cool ...

jTLM is cool ...

... but nobody will use it.

jTLM is cool ...

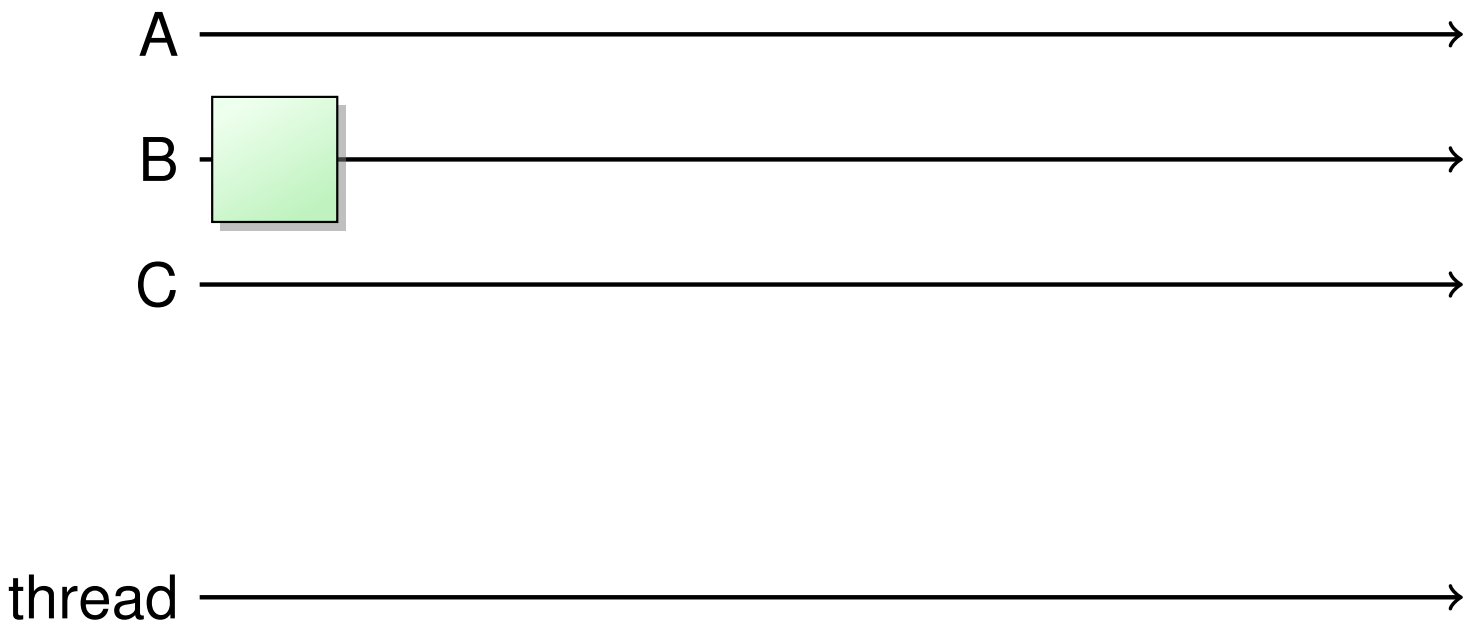
... but nobody will use it.

⇒ Implement the duration idea in SystemC:

- Keep the SystemC scheduler
- let SystemC processes delegate computation to a separate thread

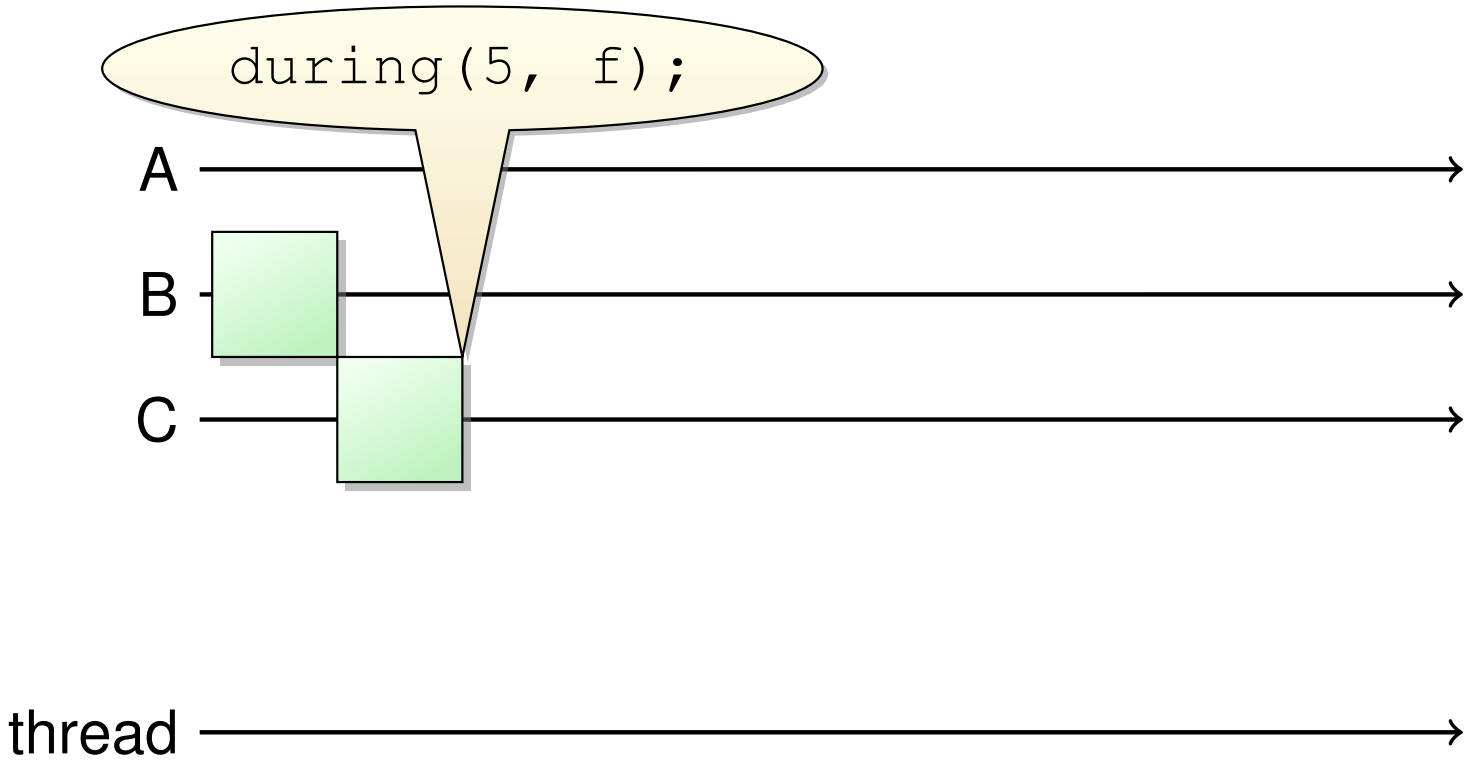
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



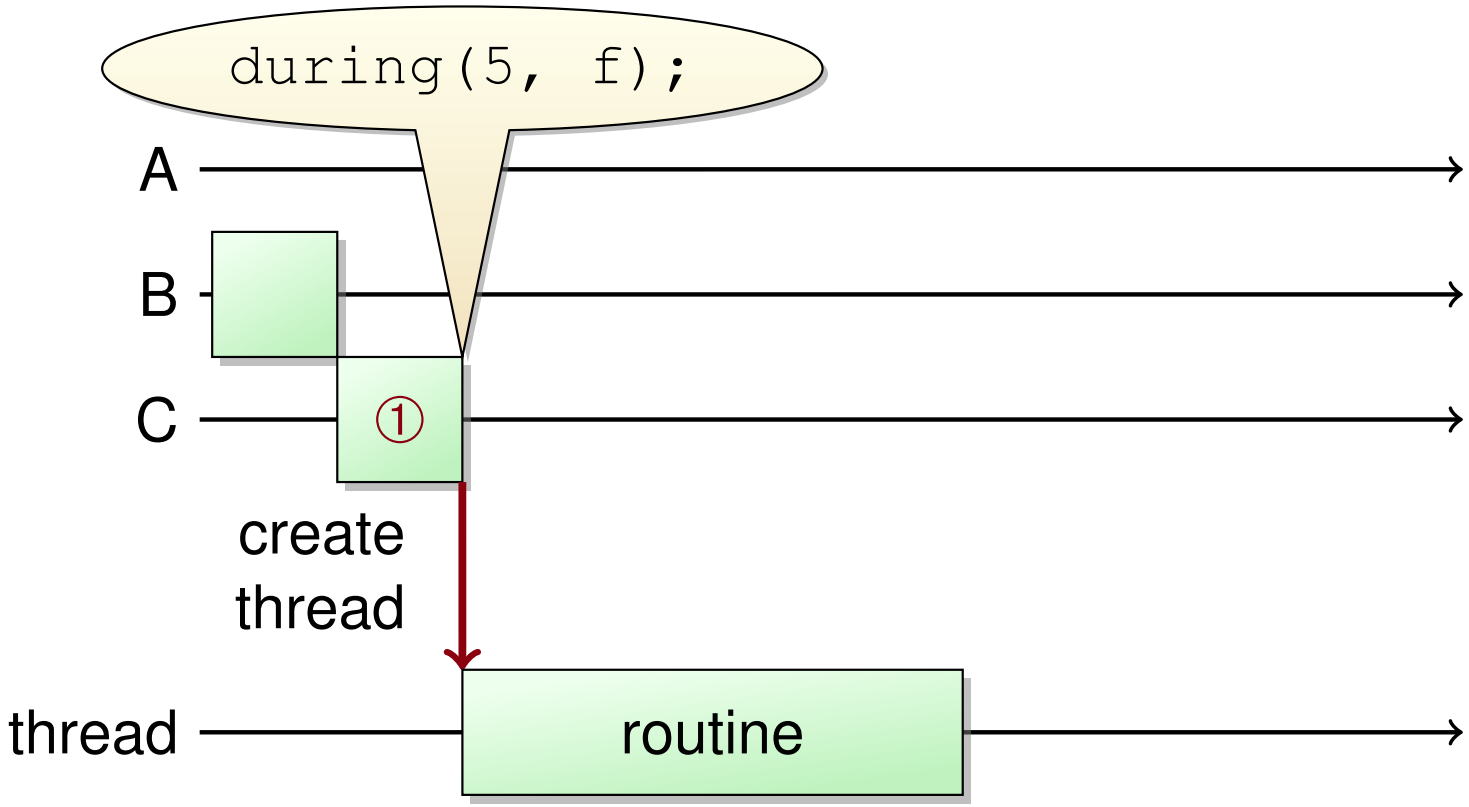
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



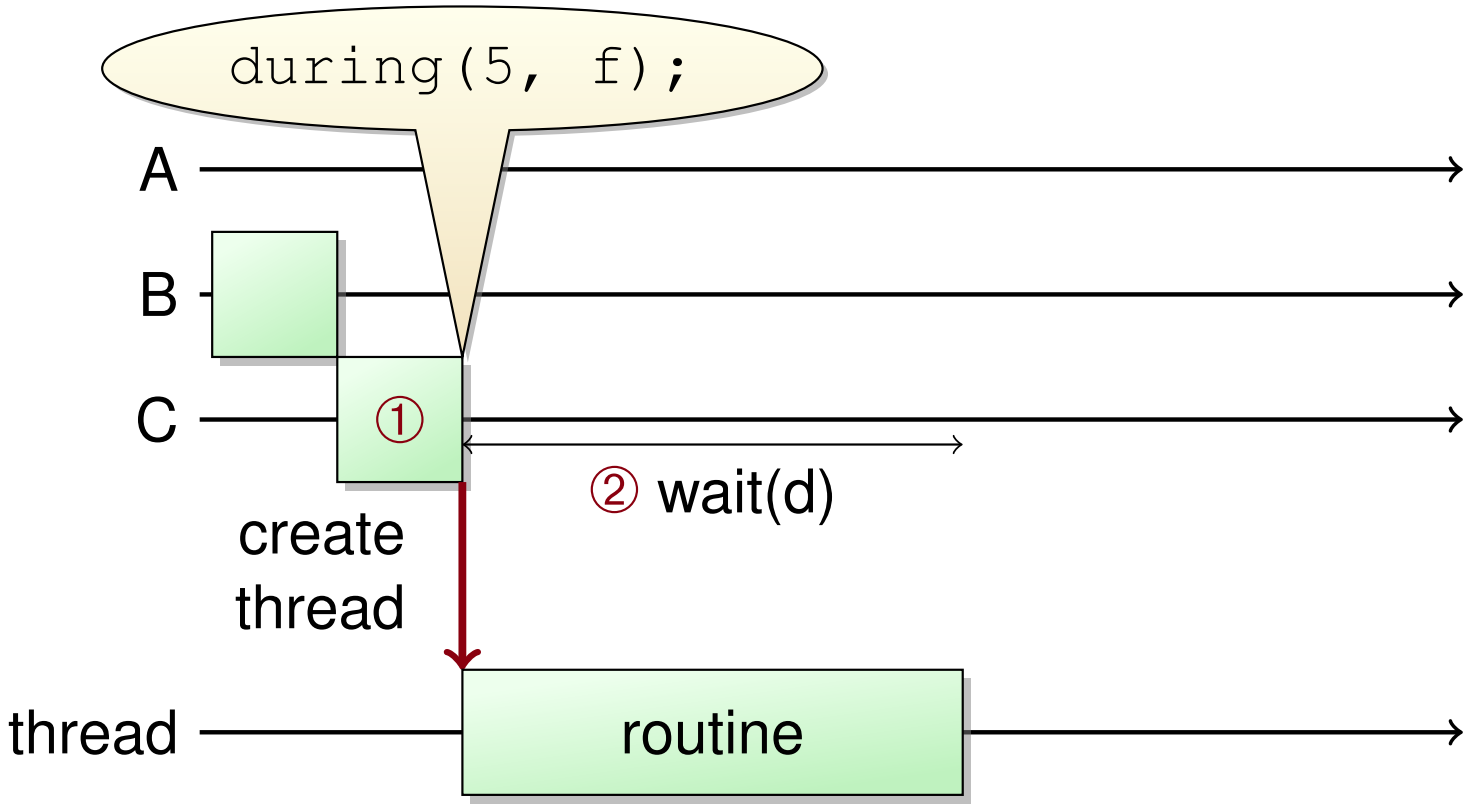
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



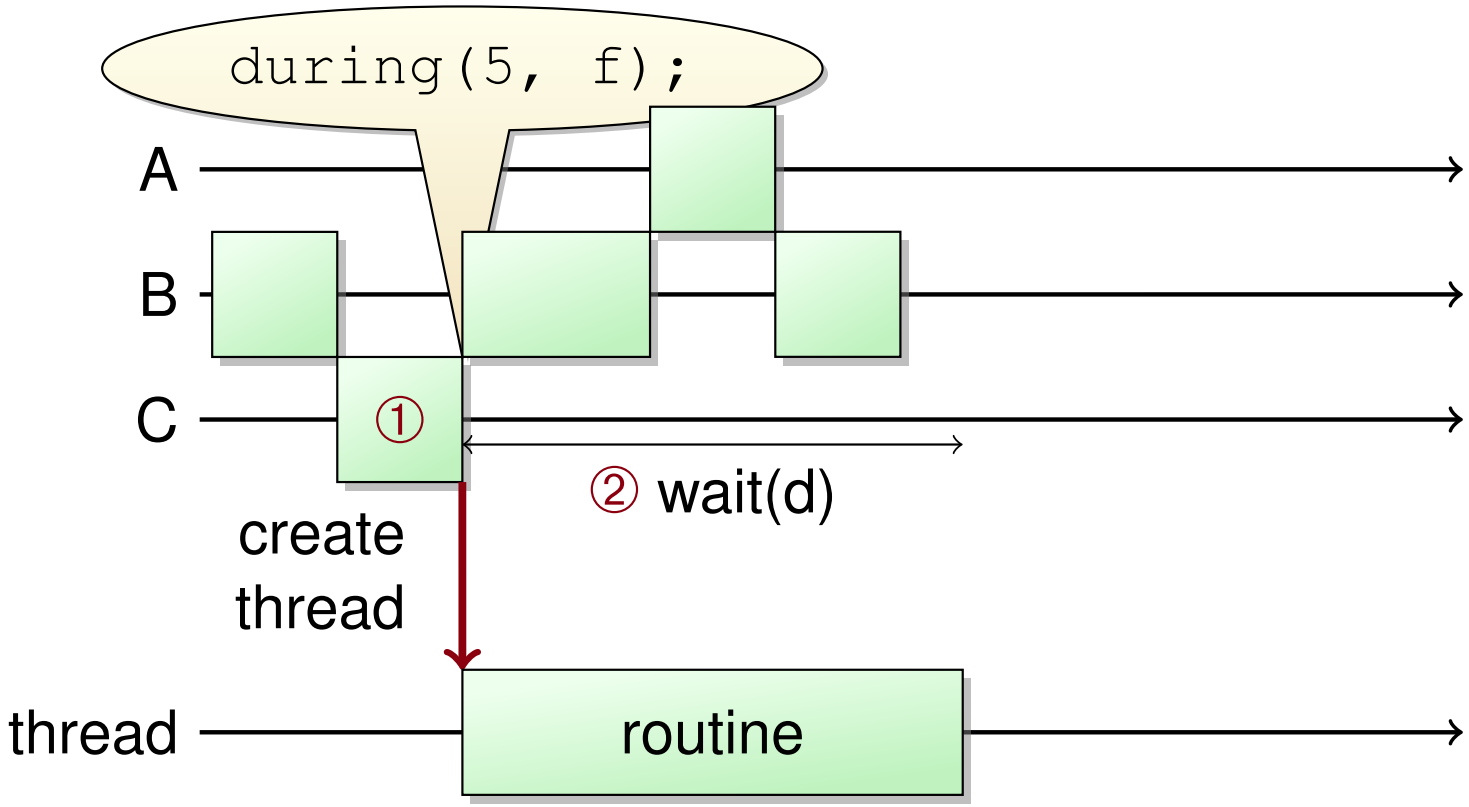
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



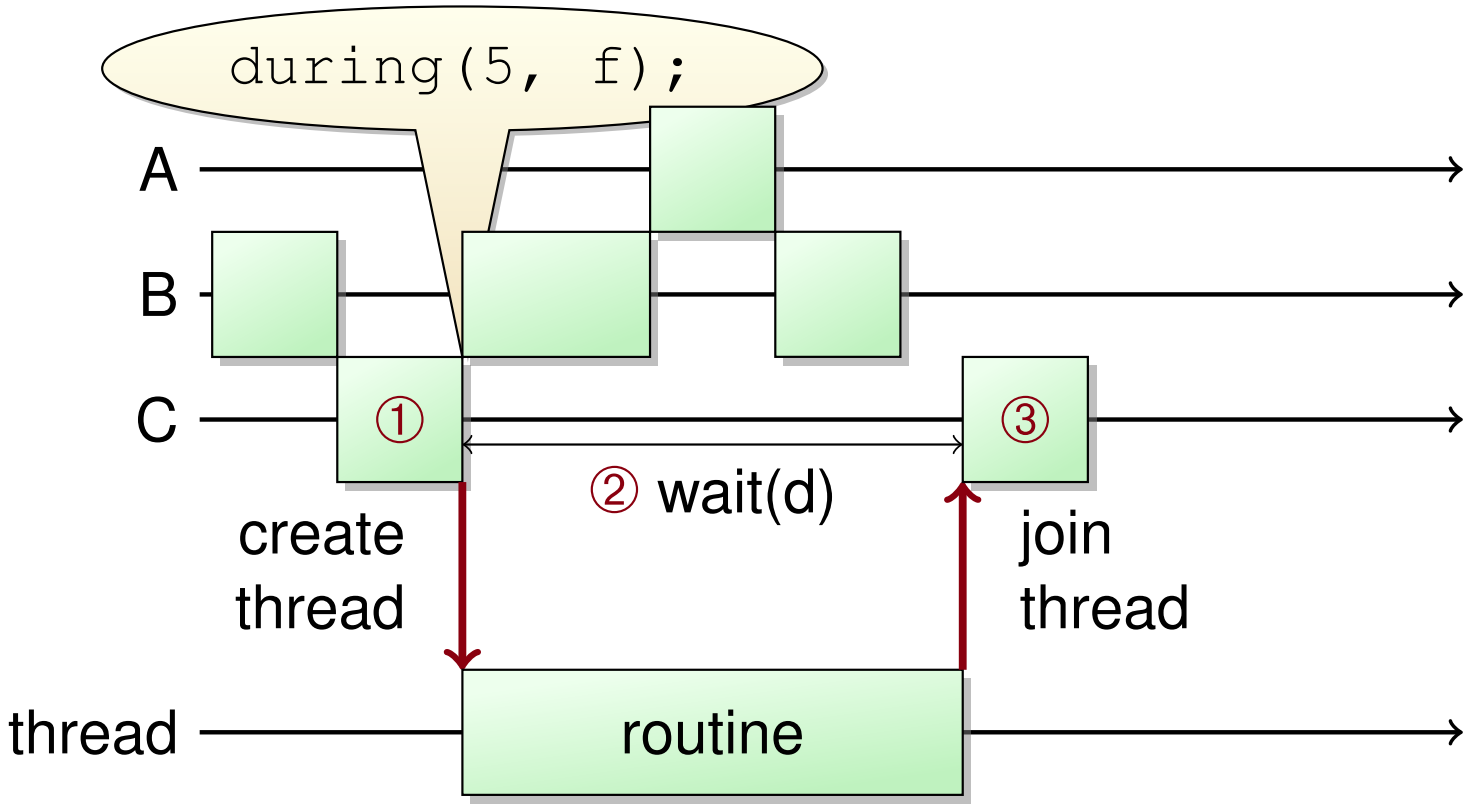
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



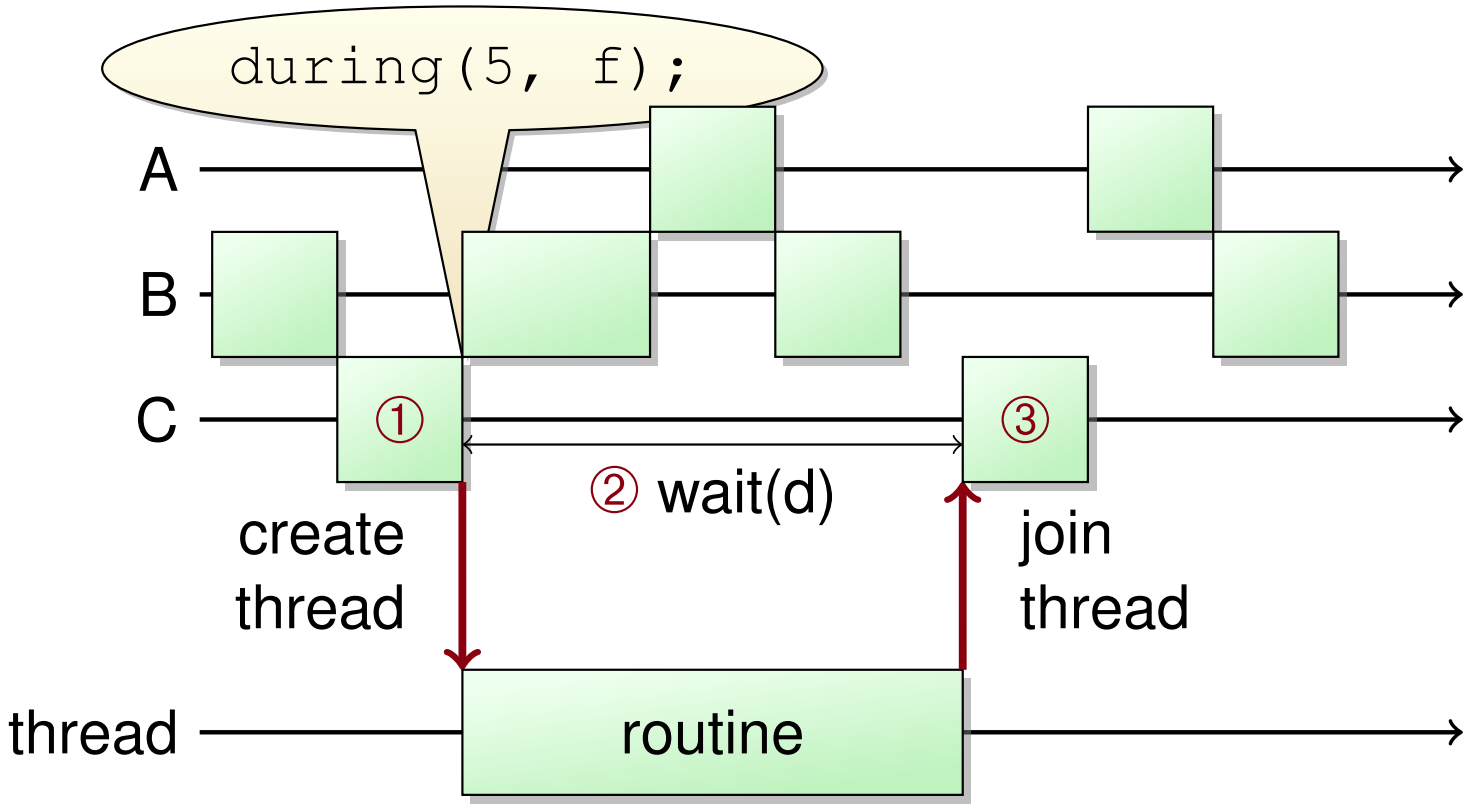
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(duration); // let SystemC execute
③     t.join(); // wait for thread completion
}
```



Wait ... are you saying that parallelization is just about fork/join?

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

When Things are Easy: Pure Function

Before

```
compute_in_systemc();

// my profiler says it's
// performance critical.
// does not communicate
// with other processes.
big_computation();
wait(10, SC_MS);

next_computation();
```

After

```
compute_in_systemc();

// Won't be a performance
// bottleneck anymore
during(10, SC_MS,
       big_computation);

next_computation();
```

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

... and sometimes it isn't

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

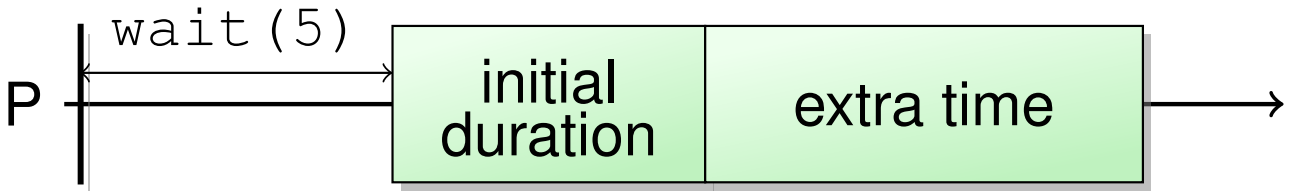
... and sometimes it isn't:

Time synchronization: make sure things are executed at the right simulated time

Data/scheduler synchronization: avoid data-race between tasks, processes and the SystemC scheduler.

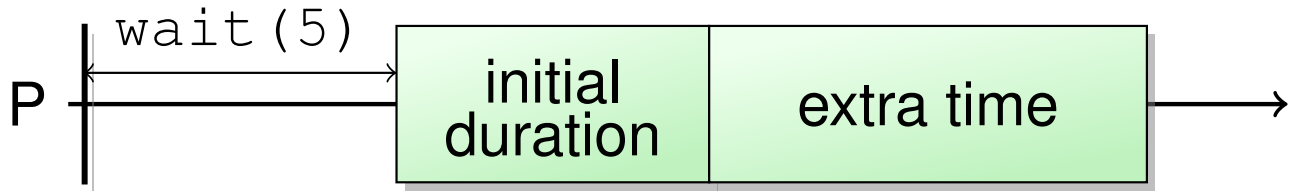
SC-DURING: Synchronization

extra_time(t): increase current task duration



SC-DURING: Synchronization

extra_time(t): increase current task duration



catch_up(t): block task until SystemC's time reaches the end of the current task

```
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```

extra_time(): Sketch of Implementation

- SystemC side:

```
void during(duration, routine) {
    end = now() + duration;
    boost::thread t(routine);
    // used to be just sc_core::wait(duration)
    while (now() != end)
        sc_core::wait(end - now());
    t.join();
}
```

- SC-DURING task side:

```
void extra_time(duration) {
    end += duration;
}
```

```
void catch_up() {
    while (now() != end)
        // avoid busy-waiting
        condition.wait();
}
```

Temporal decoupling and SC-DURING

Plain SystemC

```
f();
// instead of wait(42)
t_local += 42;
g();
t_local += 12;

// Re-synchronize with
// SystemC time
wait(t_local);
t_local = 0;

i();
```

Inside SC-DURING tasks

```
f();
// instead of wait(42)
extra_time(42);
g();
extra_time(12);

// Re-synchronize with
// SystemC time
catch_up();

i();
```

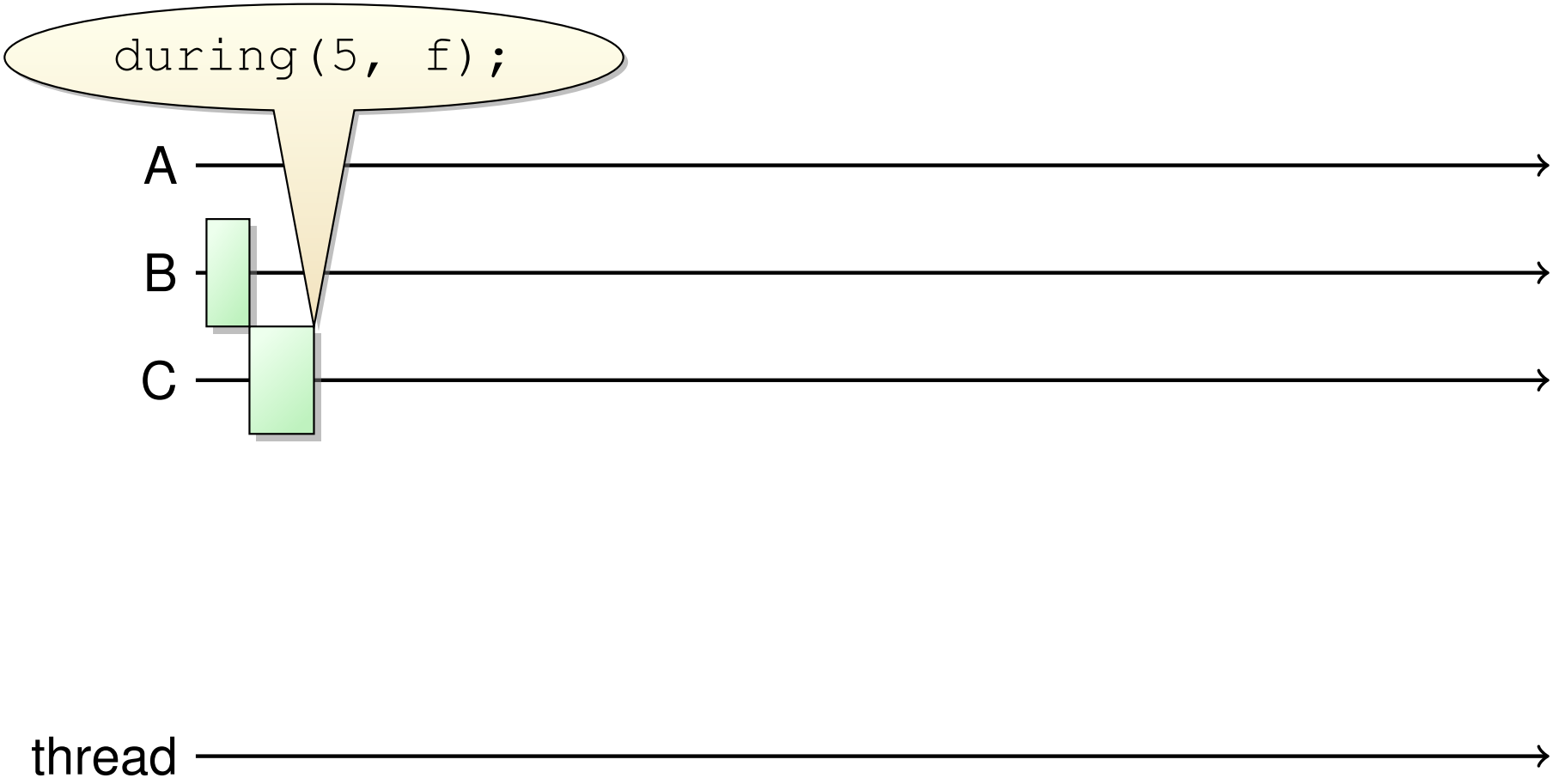
`sc_call()`: be cooperative for a while

sc_call(f): call function `f` in the context of SystemC

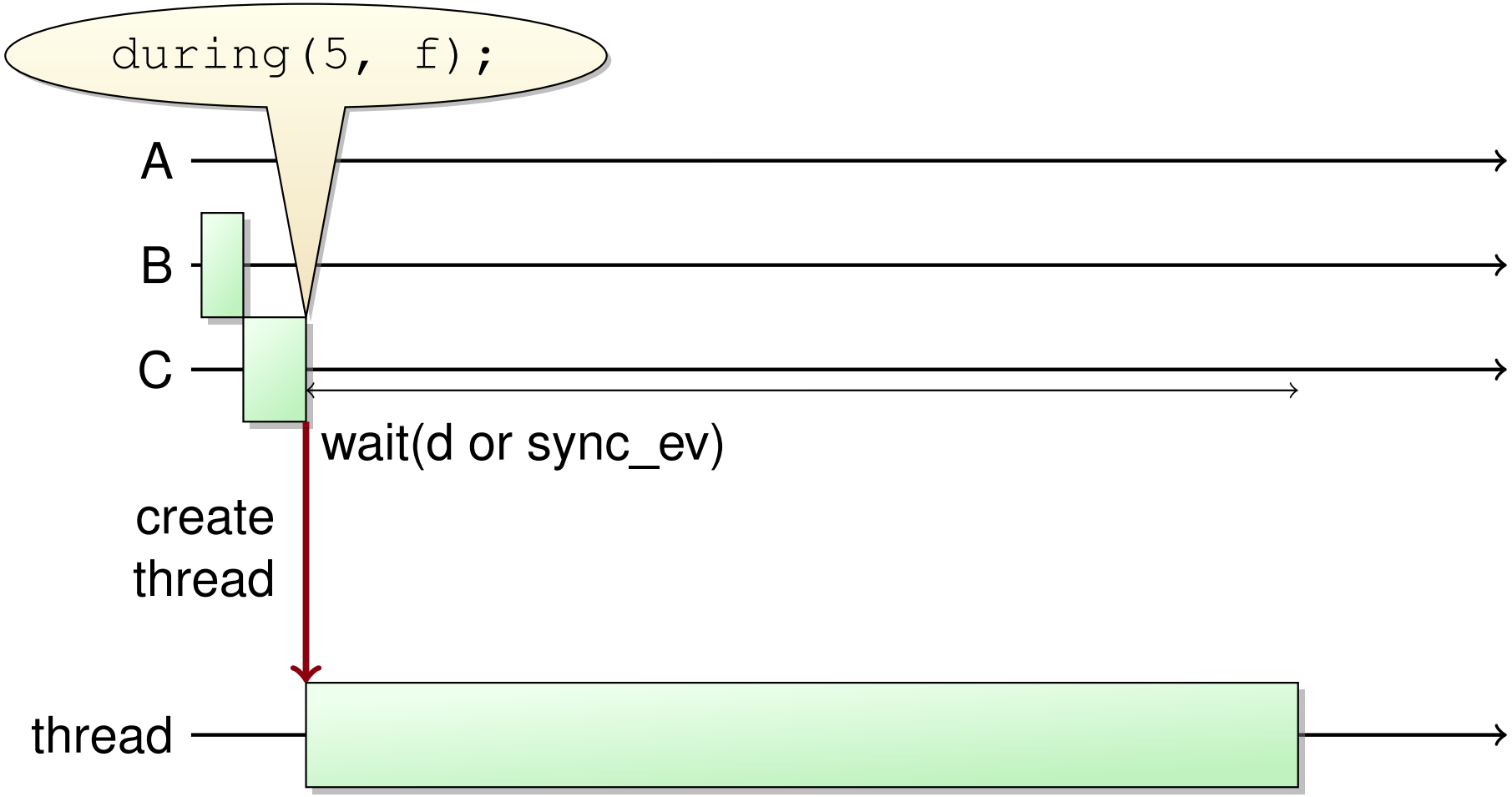
```
e.notify(); // Forbidden in during tasks

sc_call("e.notify()"); // OK (modulo syntax)
sc_call("i++"); // implicit big lock,
                // no data-race
```

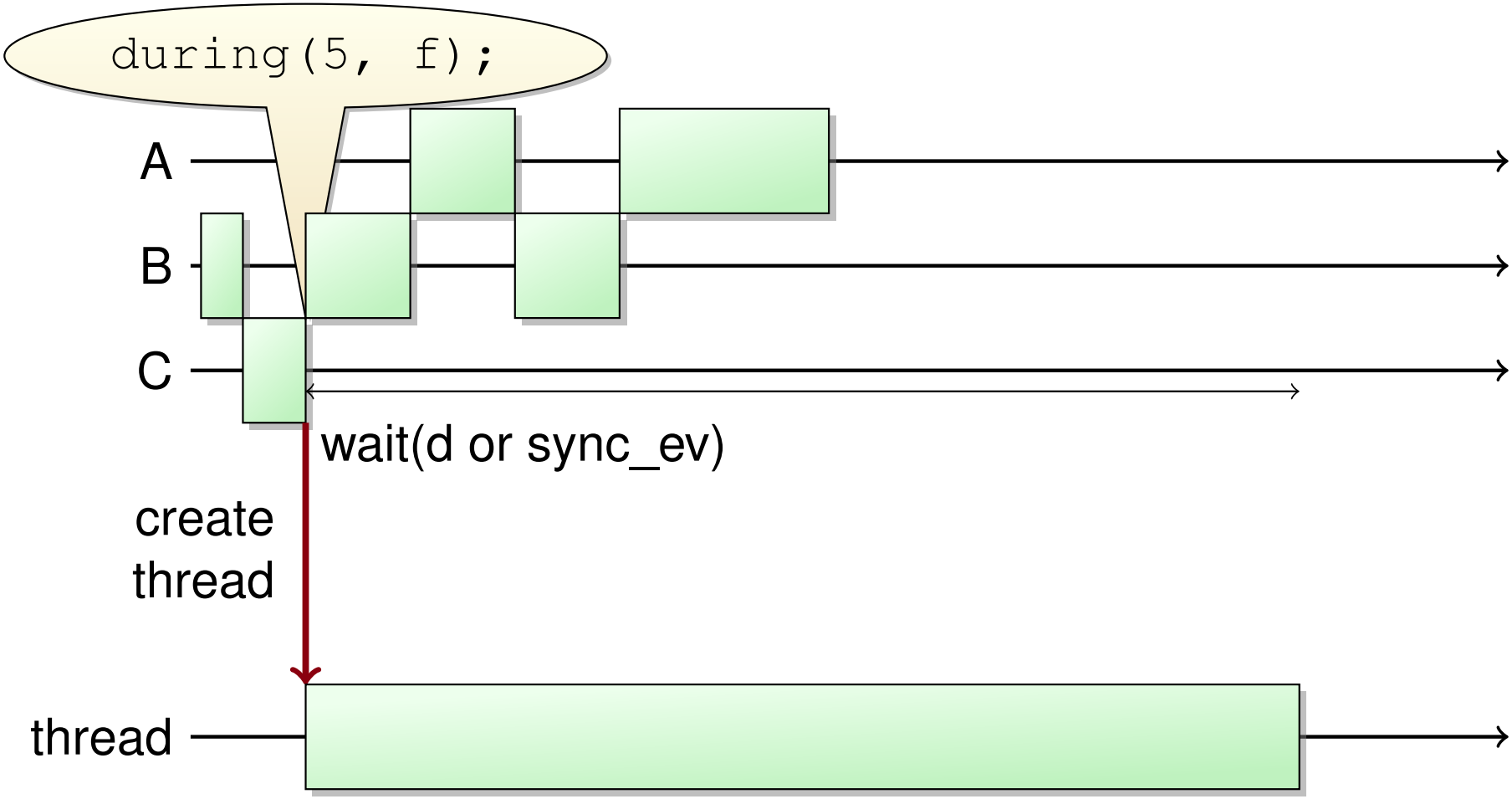
`sc_call()`: Sketch of Implementation



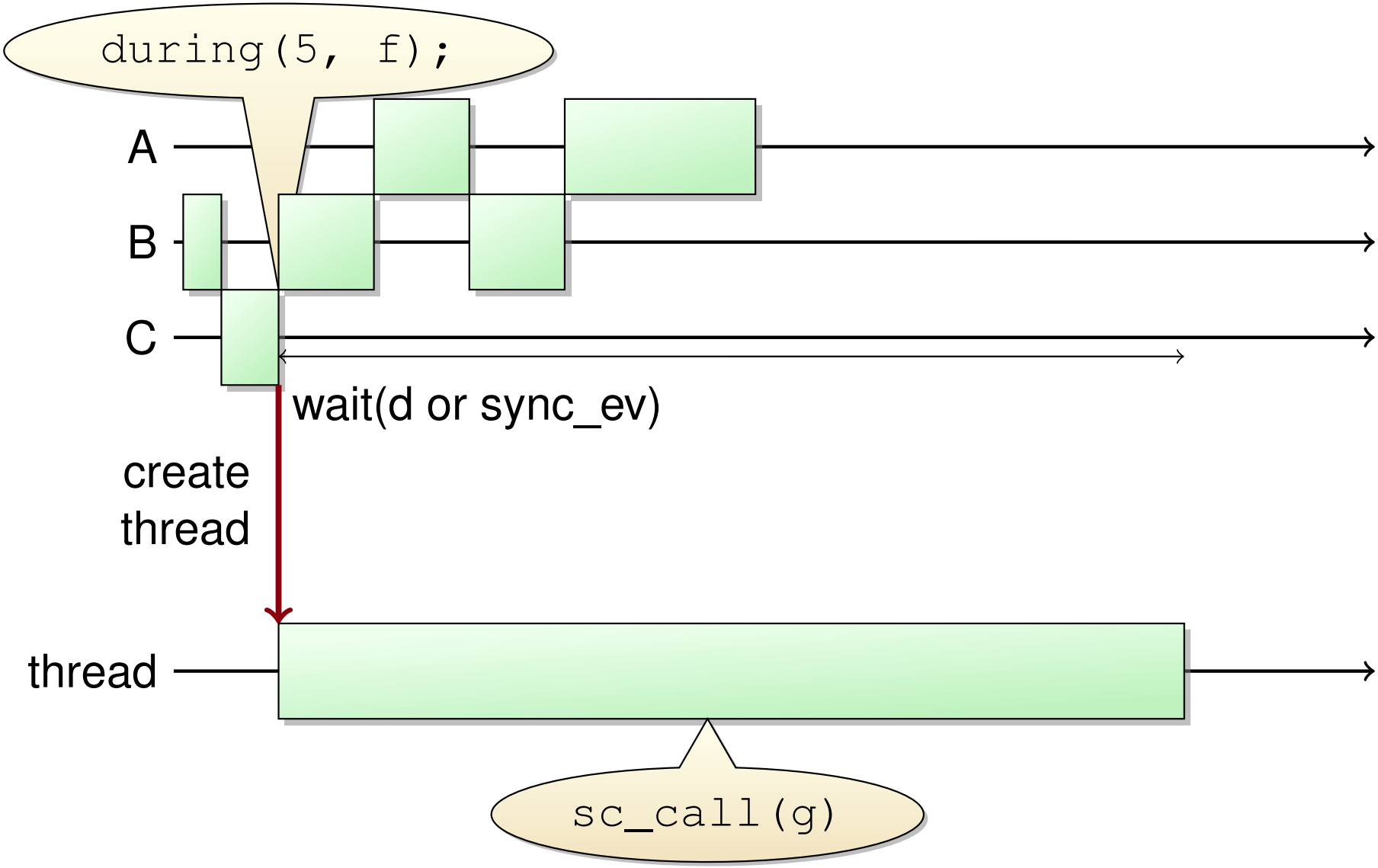
`sc_call()`: Sketch of Implementation



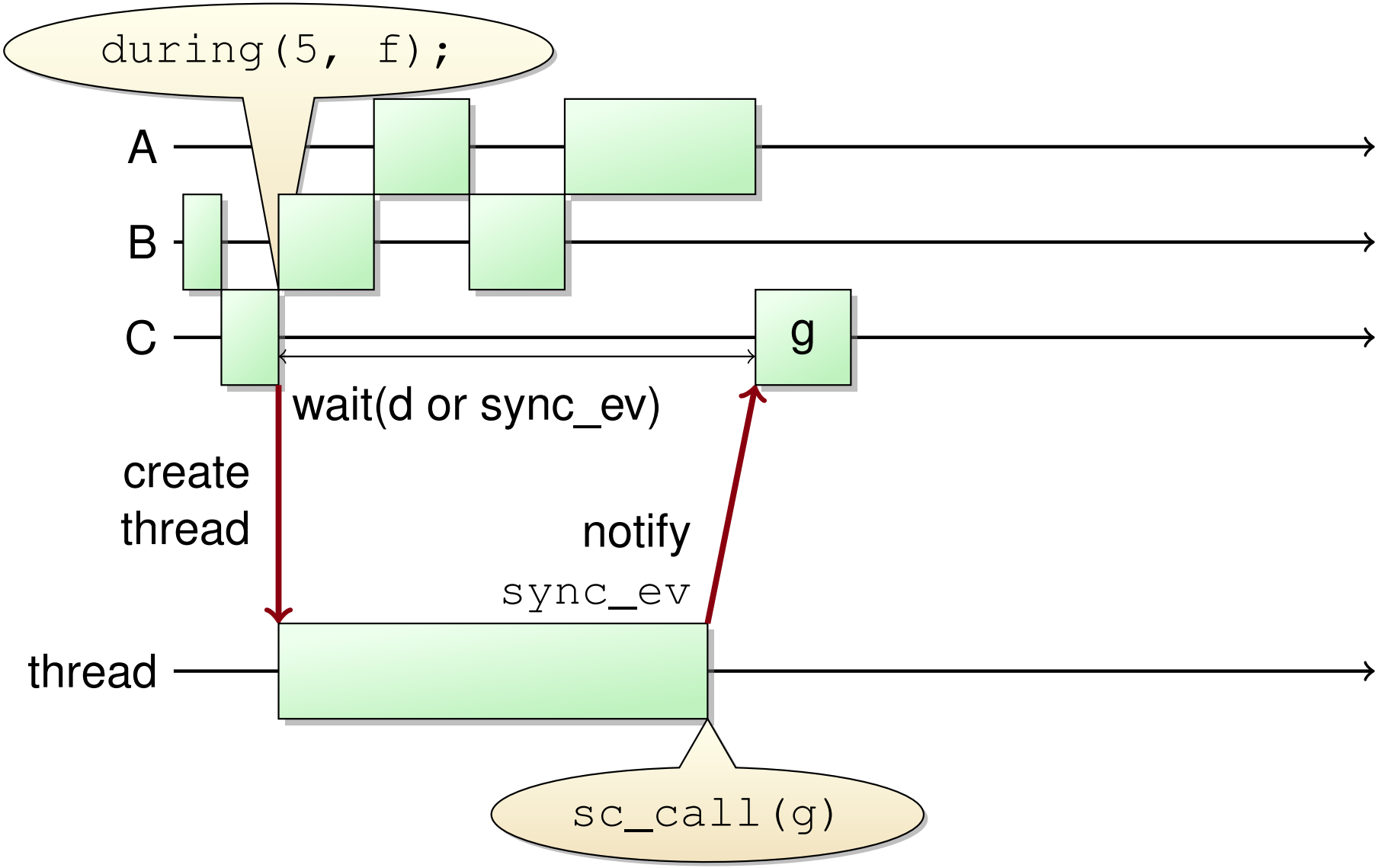
`sc_call()`: Sketch of Implementation



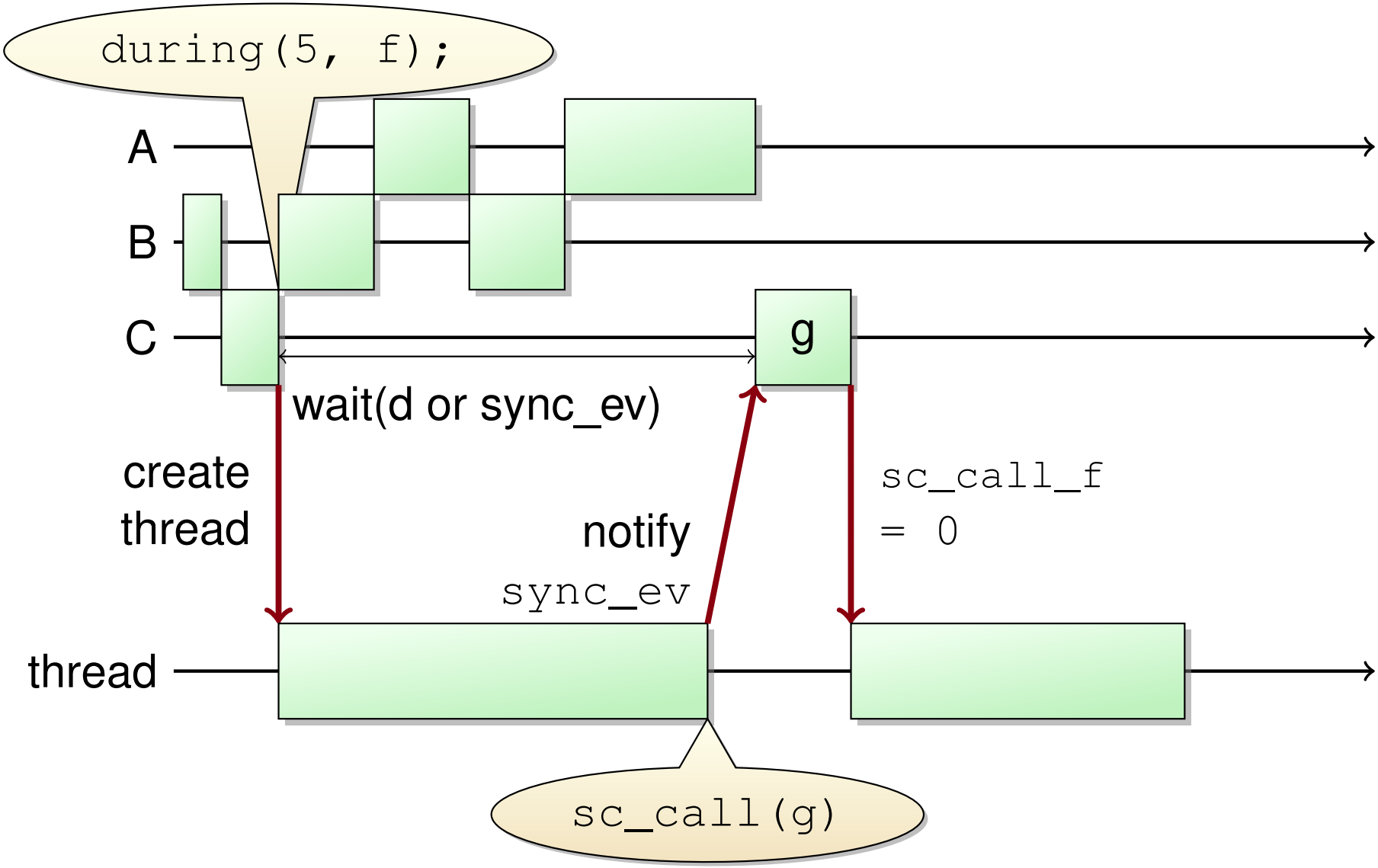
sc_call(): Sketch of Implementation



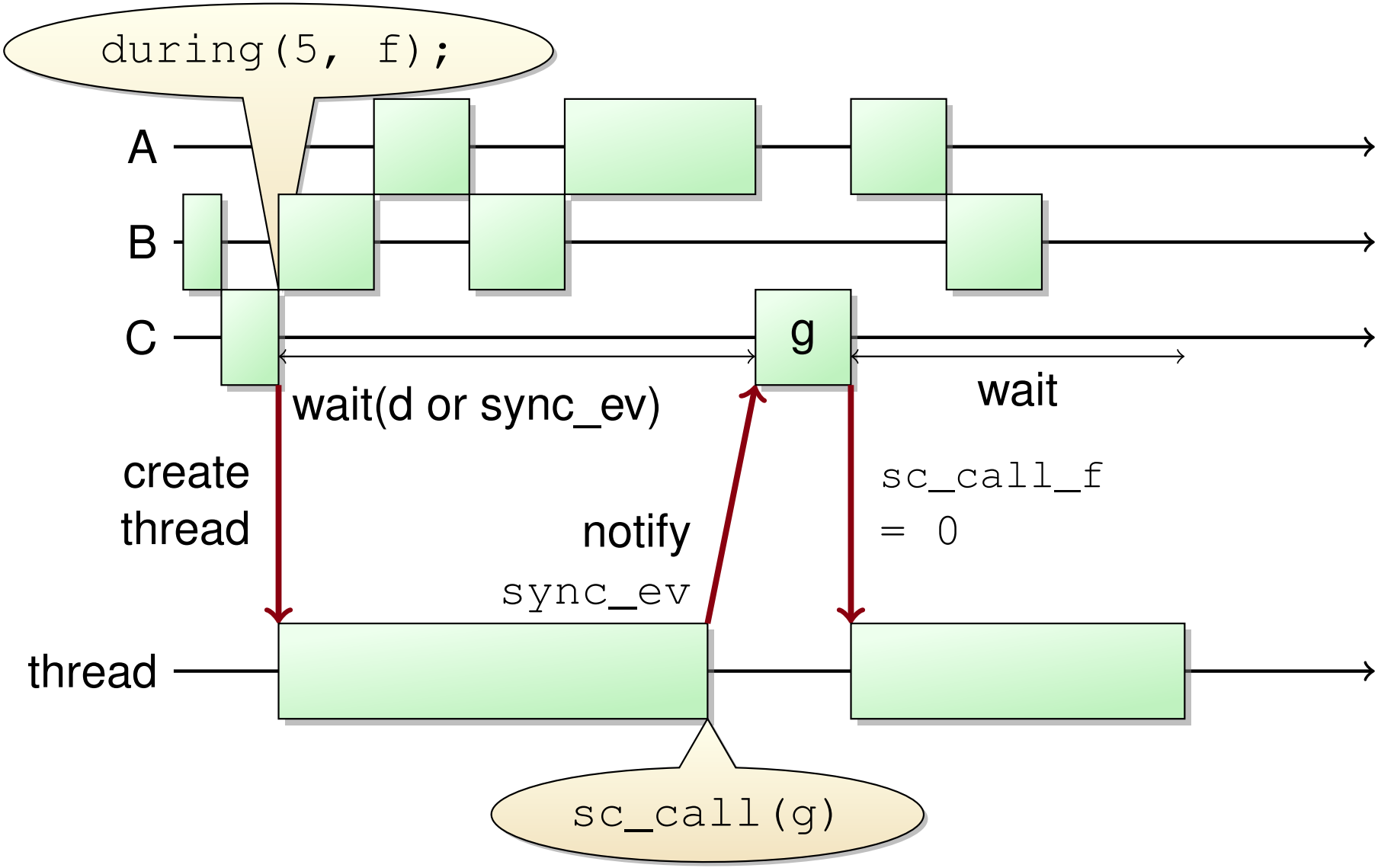
sc_call(): Sketch of Implementation



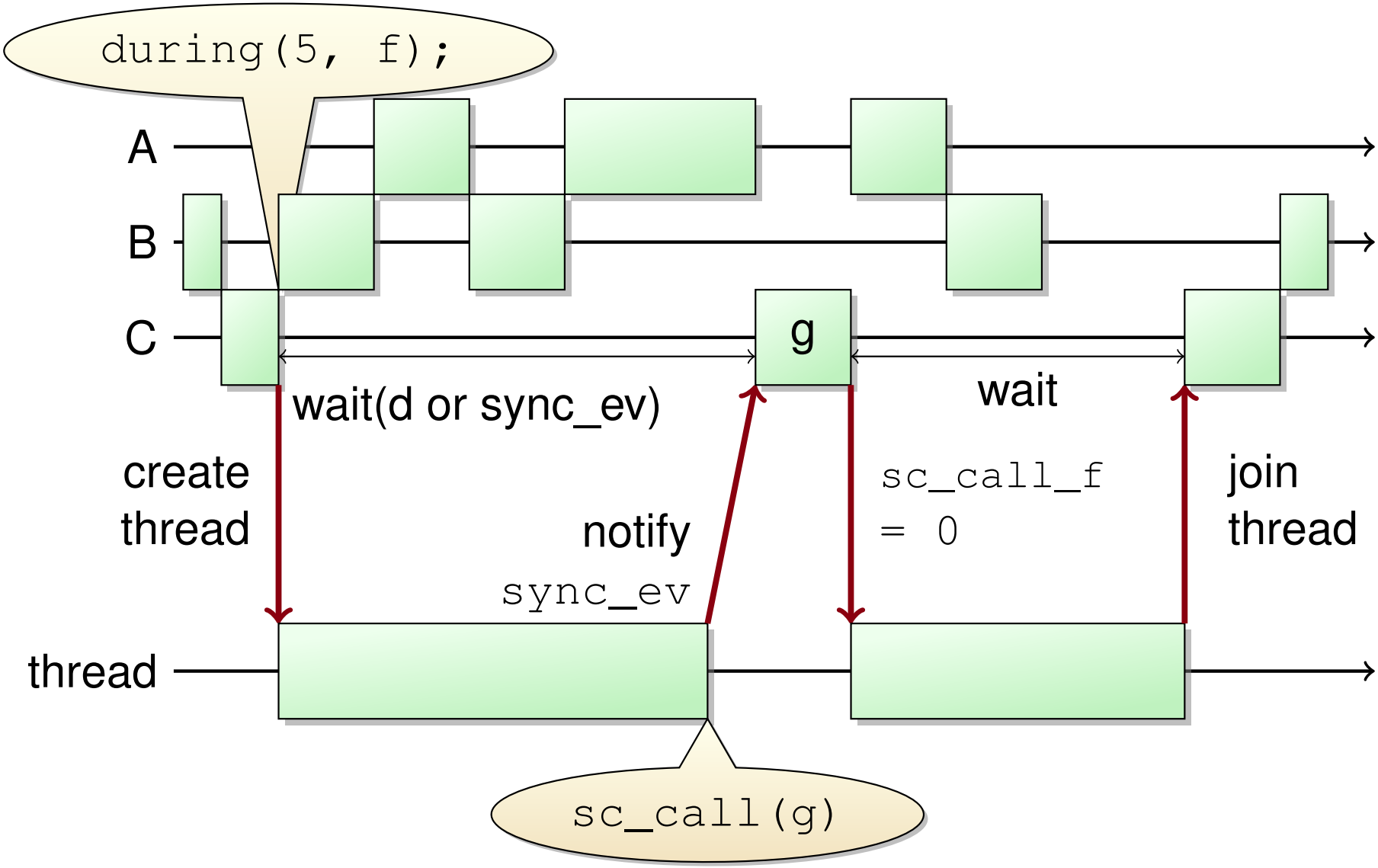
sc_call(): Sketch of Implementation



sc_call(): Sketch of Implementation



sc_call(): Sketch of Implementation

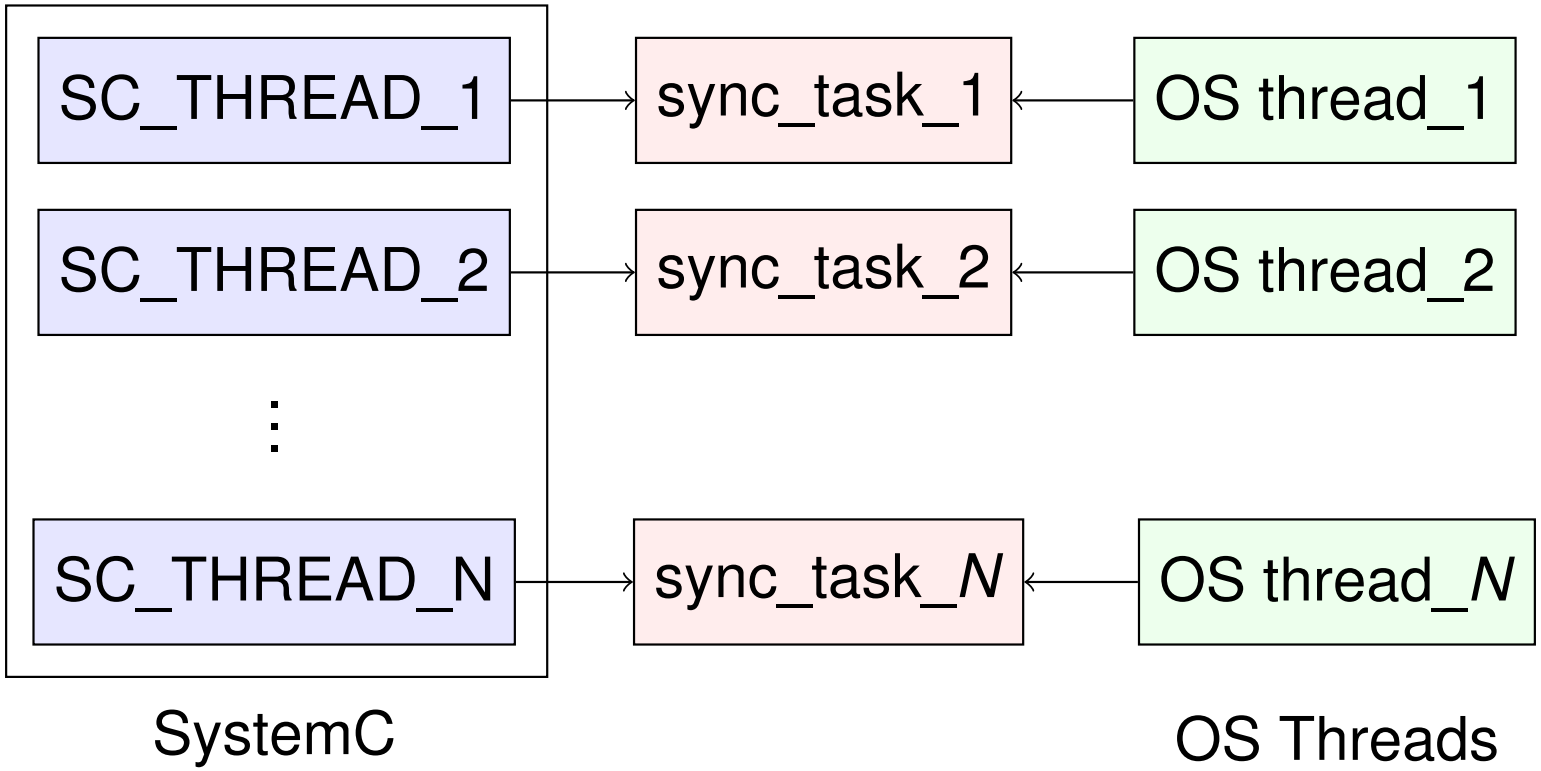


sc_call: Sketch of Implementation

```
void during(duration, f) {
   end = now() + duration;
   boost::thread t(f);
   while (now() != end) {
      // wait sync_ev
      // with timeout:
      sc_core::wait
         (sync_ev,// <--
          end - now());
      if (sc_call_f) {
         sc_call_f();// <--
         sc_call_f = 0;
         condition.notify();
      }
   }
   t.join();
}
```

```
void sc_call(f) {
   sc_call_f = f;
   // Implemented
   // with SystemC 2.3's
   // async_request_update()
   async_notify_event
       (sync_ev);
   while(sc_call_f != 0) {
      condition.wait();
   }
}
```

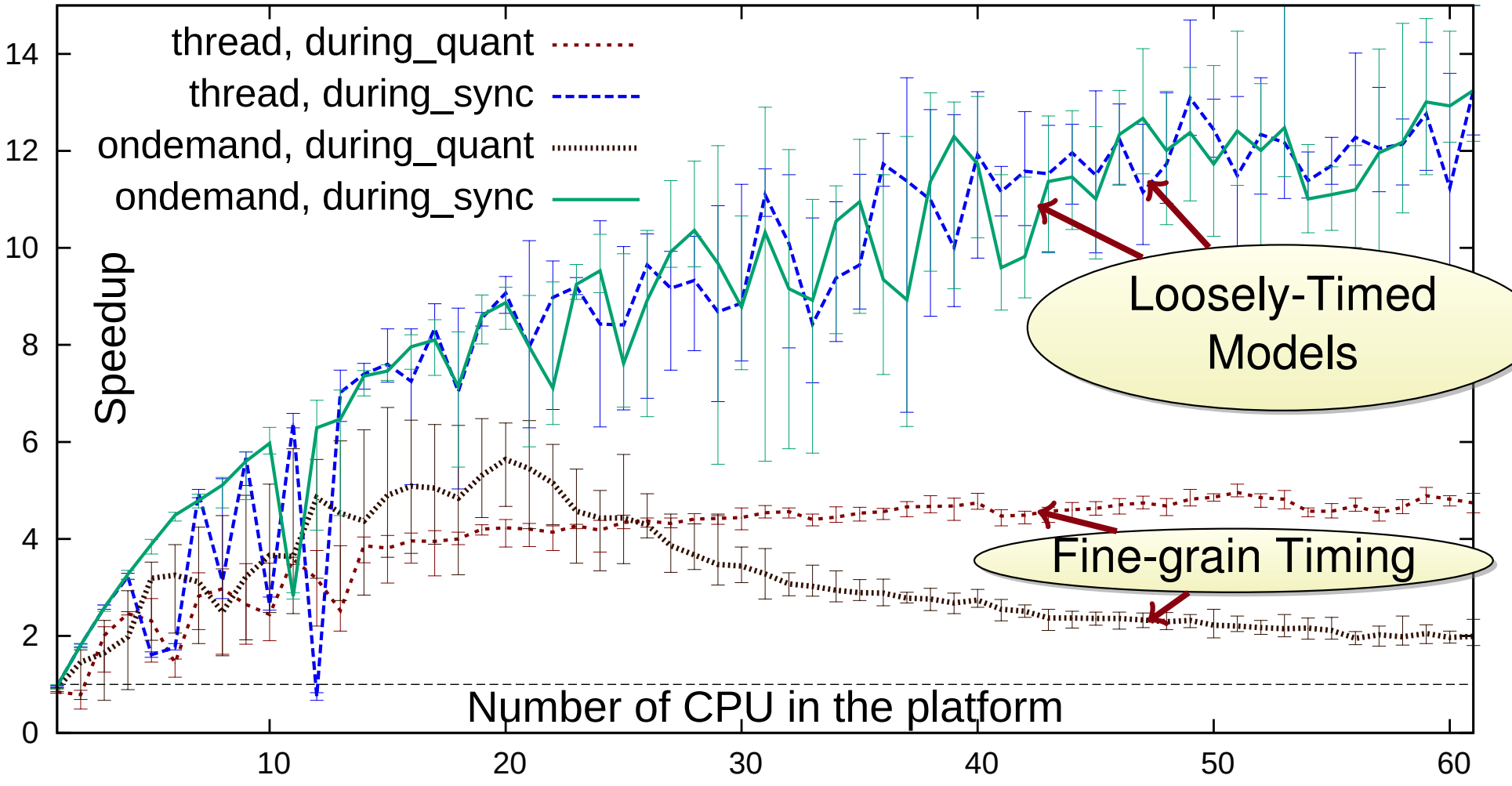
SC-DURING: Actual Implementation



Possible strategies:

- **SEQ** Sequential (= reference)
- **THREAD** Thread created/destroyed each time
- **POOL** Pre-allocated worker threads pool
- **ONDEMAND** Thread created on demand and reused later

SC-DURING: Results



Test machine has $4 \times 12 = 48$ cores

Outline

SC-DURING: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- Future work: performance optimizations (e.g. atomic operations + polling instead of system calls)

Try it:

`https://forge.imag.fr/projects/sc-during/`

SC-DURING: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- Future work: performance optimizations (e.g. atomic operations + polling instead of system calls)

Try it:
`https://forge.imag.fr/projects/sc-during/`

Questions?

SC-DURING: Conclusion

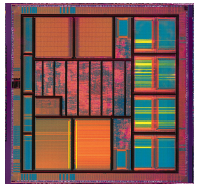
- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- Future work: performance optimizations (e.g. atomic operations + polling instead of system calls)

Try it:
`https://forge.imag.fr/projects/sc-during/`

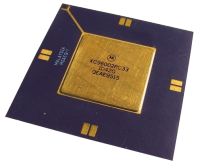
Questions?

Thank You!

Sources



http://en.wikipedia.org/wiki/File:Diopsis.jpg
(Peter John Bishop, CC Attribution-Share Alike 3.0 Unported)



http://www.fotopedia.com/items/flickr-367843750
(oskay@fotopedia, CC Attribution 2.0 Generic)