(Un)Suspend(able)

Dr Mark Burton
Engineer, Principal/Manager
Qualcomm France S.A.R.L.

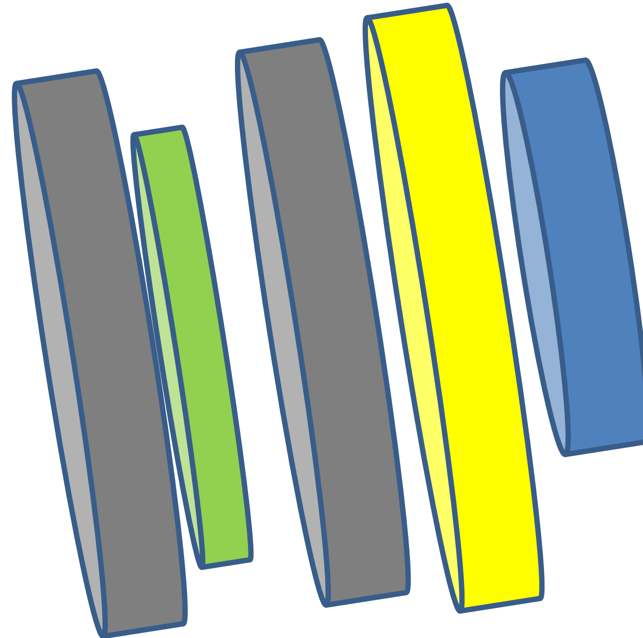
Copyright Permission

- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **Qualcomm Incorporated** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on the standard.

Remember 2019?

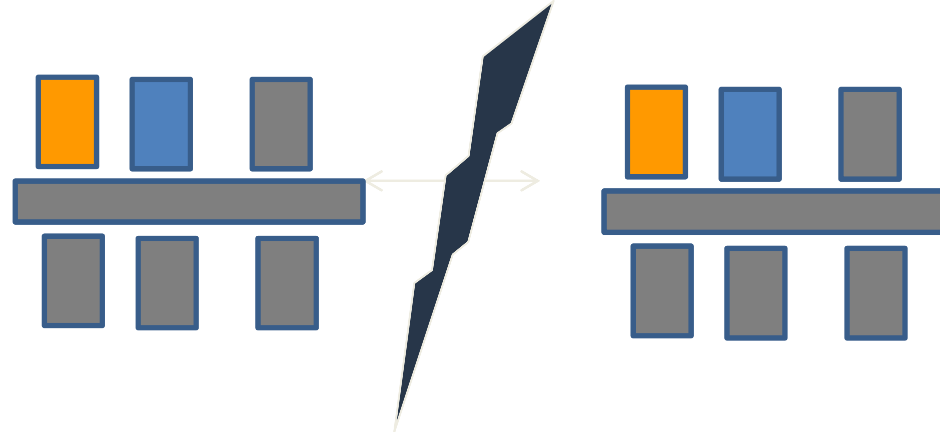
Plus ça change, plus c'est la même chose

- Syncromesh – it’s all about connections between things moving at different speeds!



Some sort of synchromesh thing!

Motivation 1: Multiple Threads (and processes)

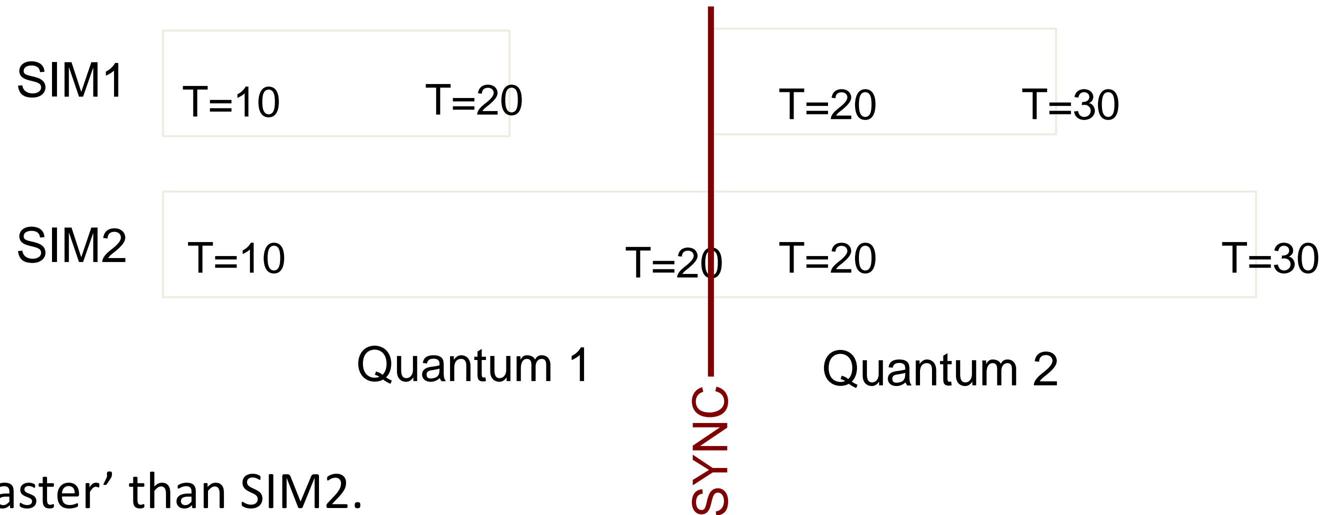


We are talking about divide simulations based on architectural features (like CPU's or Ethernet)

(not on a SystemC thread/method level)

Time (Stops for nobody?)

- We have some choices:
  - Don’t worry about time – let every simulation run at the speed it wants!
  - Try to keep within a ‘Quantum’ (Introduced in TLM 2.0)



  - SIM1 runs ‘faster’ than SIM2.
  - At least once per quantum, the simulations are synchronized.
  - SIM1 needs to ‘wait’ for SIM2.

SUSPEND!

- We need a mechanism
  to **SUSPEND** a simulation

  while it waits for the other simulations catch up (and send an event to continue).

- **NB 'suspend' exists (!) for individual threads.**
- We need to suspend all threads, so that SystemC has nothing more to do.

SystemC 2.3.1…

SOME BACKGROUND…

© Accellera Systems Initiative

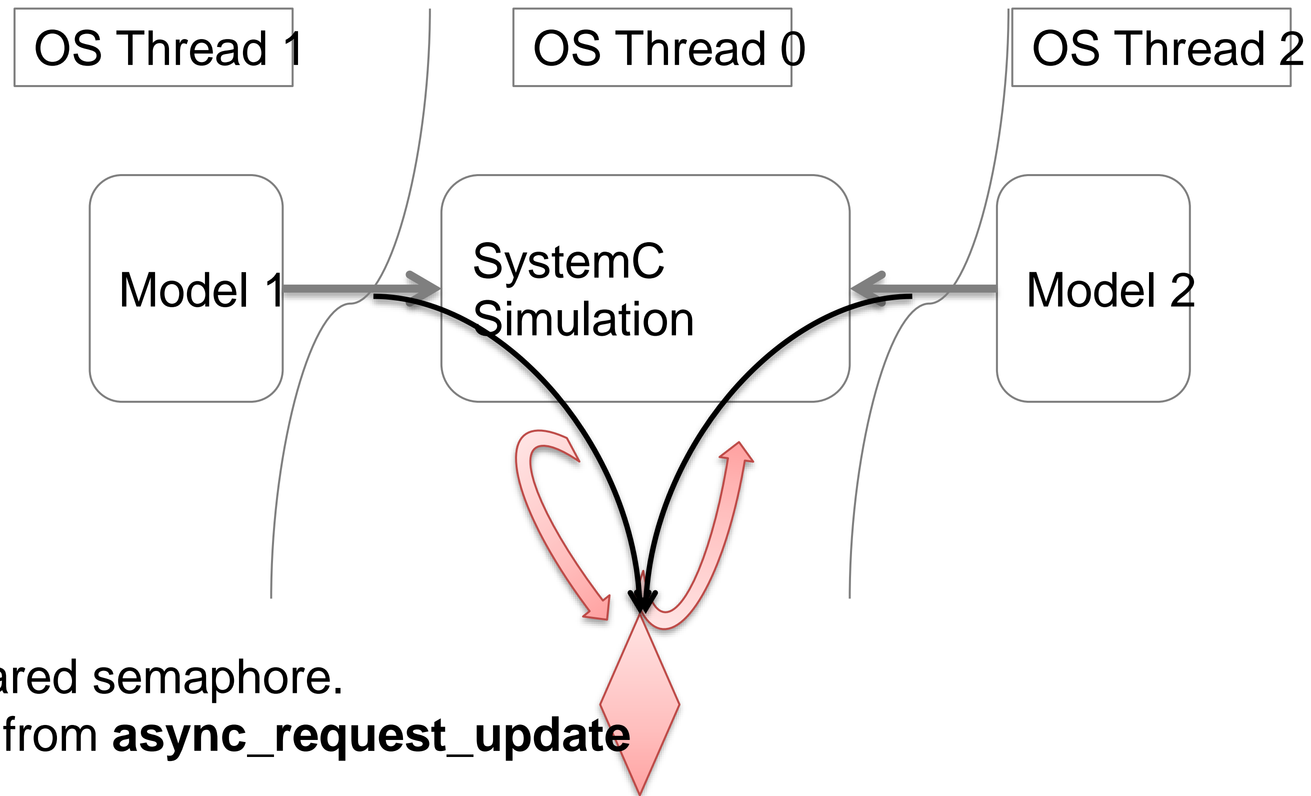
async_request_update

- **async_request_update** – allows an external 'event' to be inserted into a running SystemC kernel in a thread safe way.
- Basis of any (all) communication between two simulations

- Problem : If a simulation runs out of events, then …
  we better not stop!

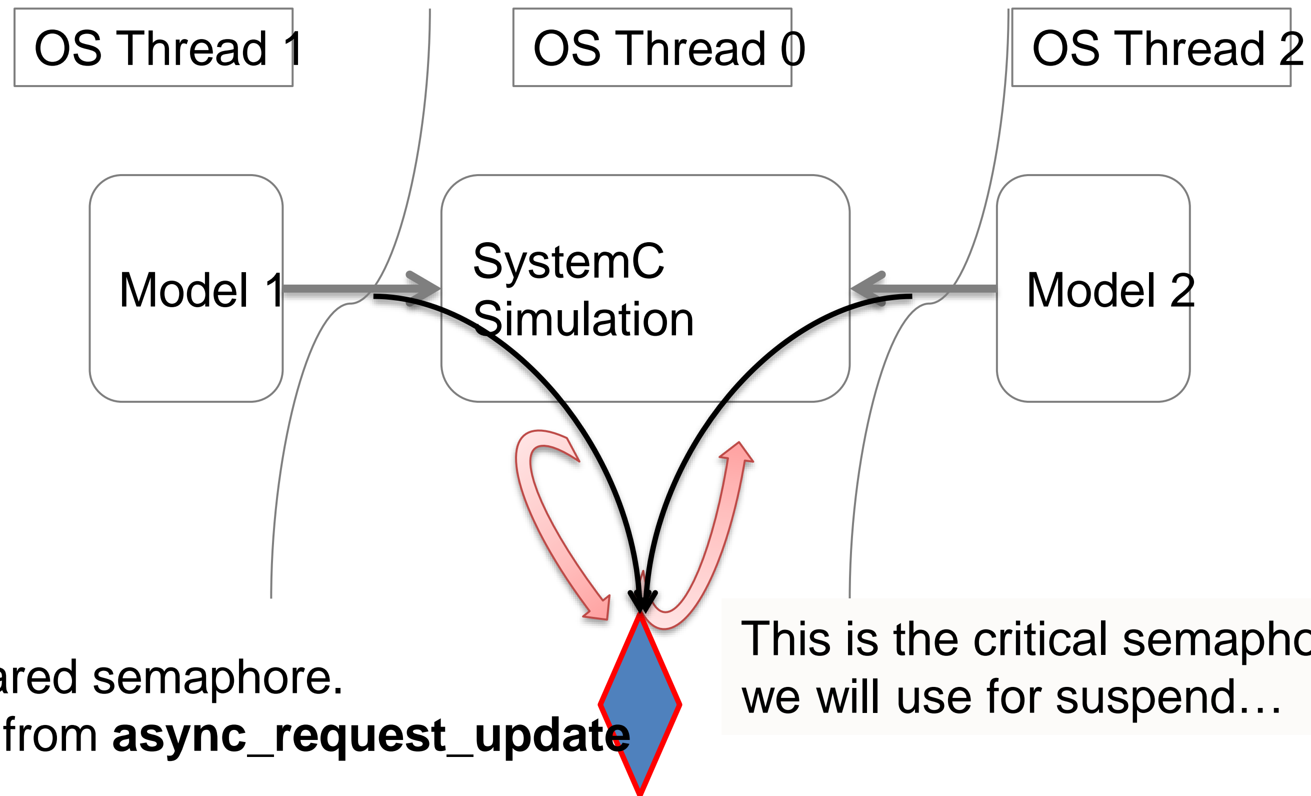
  (And if we have suspended all the threads, that can happen a lot!)
- And we need a single common semaphore

Compose-able solution for **2.3.2**



Single shared semaphore.
Triggered from **async_request_update**

Compose-able solution for **2.3.2**



Single shared semaphore.
Triggered from **async_request_update**

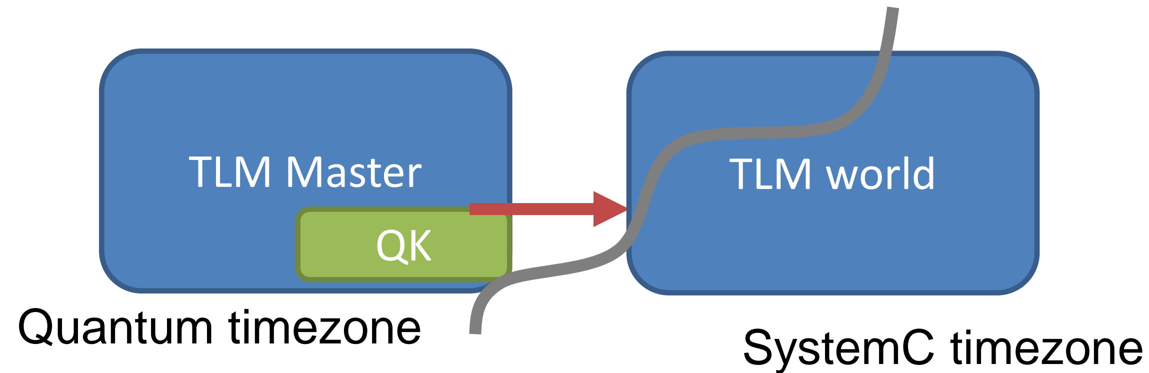
Details (Available in SystemC 2.3.2+)

- By default the semaphore is not used, SystemC exits normally.
- **bool async_attach_suspending** // proposed for IEEE 1666-202x
  - Prim_channels can elect to attach to an external source of events (and therefore request the presence of the semaphore)
- **bool async_detach_suspending**
  - A prim_channel can elect to detach from an external source of events (and therefore remove the request for the presence of the semaphore). If no prim_channels are attached to external events, the semaphore plays no role in simulation.
- The semaphore is only checked (and potentially waited for) if no further events are available
  - starvation, which is checked inside the simulation kernel anyway (using **next_time** returns 0), so there is no additional cost here.
- The semaphore is released when **async_request_update** is called.

MULTI-THREADED QUANTUM KEEPER

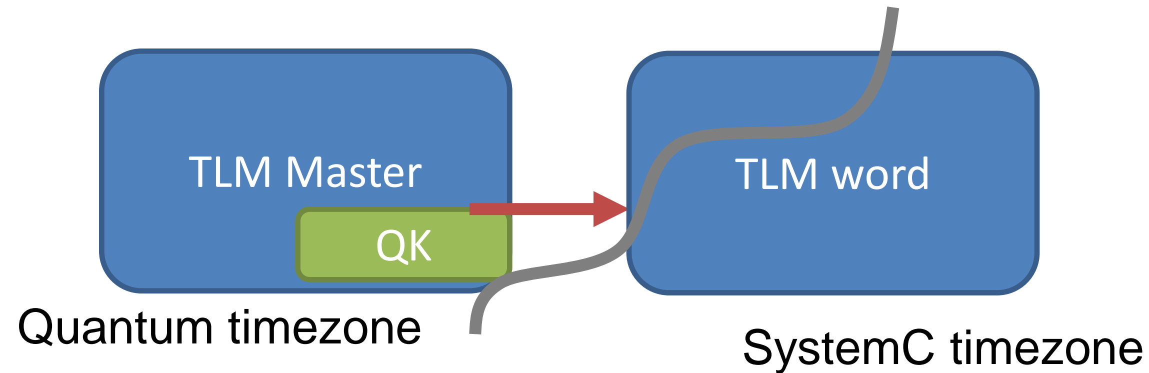
Building on TLM Quantums

TLM Quantum Keeper API



- The notion of "TLM Quantum" allows multiple 'timezones'.
- The Quantum Keeper is responsible for maintaining synchronisation between 'timezones'.

TLM Quantum Keeper



- Different ‘timezones’ could be in different threads !
- The Quantum Keeper is STILL responsible for maintaining synchronisation between ‘timezones’.

Underpinning services for QK

- Within SystemC the only ‘service’ the QK needs is to be able to ‘stop’ the timezone that has gone ahead, and allow the other timezone to catch up.
- TLM defines that the ‘quantum timezone’ is ahead of ‘SystemC timezone’
- The only ‘service’ needed is to call ‘wait’ in SystemC.

Multithread services for QK

- Between 2 threads, the SystemC timezone may run independently of the Quantum timezone.
- Hence the QK needs a way to stop SystemC time, while the Quantum time 'catches up'.

- Reverse of normal SystemC QK.

Building on central semaphore

SUSPEND

accellera SYSTEMS INITIATIVE™ © Accellera Systems Initiative

Suspend

- Add a kernel function
- suspend_all()
  - Request to suspend all threads (and pending events)
  - All threads become un-schedulable
  - (The simulation will run out of schedulable tasks instantly, and fall into the 'semaphore').
- Unsuspend()
  - Request to re-instate all threads (and pending events).
  - *You can call unsuspend from a async_update method as you will not be in the semaphore at that point.*

© Accellera Systems Initiative

Unsuspendable/Suspendable

- A thread may mark itself as ‘unsuspendable’.
- This only effects the ‘global’ suspend_all mechanism.
- For save/restore, all b_transports that are non-re-entrant will have to be non-suspendable.
- For thread sync, all b_transports being processed on behalf of an external simulation should be marked as non-suspendable.

new API

- The New API includes:

```
void sc_suspend_all()
void sc_unsuspend_all()
void sc_unsuspendable()
void sc_suspendable()
```

- It also includes changes to the state and ways to discover if the simulator is suspended using

```
void sc_(un)register_stage_callback( const sc_stage_callback_if&, int );
void sc_unregister_stage_callback( const sc_stage_callback_if&, int );
```

accellera SYSTEMS INITIATIVE

SYSTEMC EVOLUTION DAY DEC 8, 2022 | MUNICH | GERMANY

suspend_all/unsuspend_all :

- Requests the kernel to '**atomically suspend**' all processes (using the same semantics as the thread suspend() call). This is atomic in that the kernel will only suspend all the processes together, such that they can be suspended and **unsuspended without any side effects**.
  - Calling suspend_all(), and subsiquently calling unsuspend_all() will have no effect on the suspended status of an individual thread.
- A process may call suspend_all() followed by unsuspend_all, the calls should be 'paired', (multiple calls to either suspend_all() or unsuspend_all() will be ignored).
- Outside of the context of a process, it is the programmers responsibility to ensure that the calls are paired.
- As a consequence, multiple calls to suspend_all() may be made (within separate processes, or from within sc_main). So long as there have been more calls to suspend_all() than to unsuspend_all(), the kernel will suspend all processes.

unsuspendable()/suspendable()

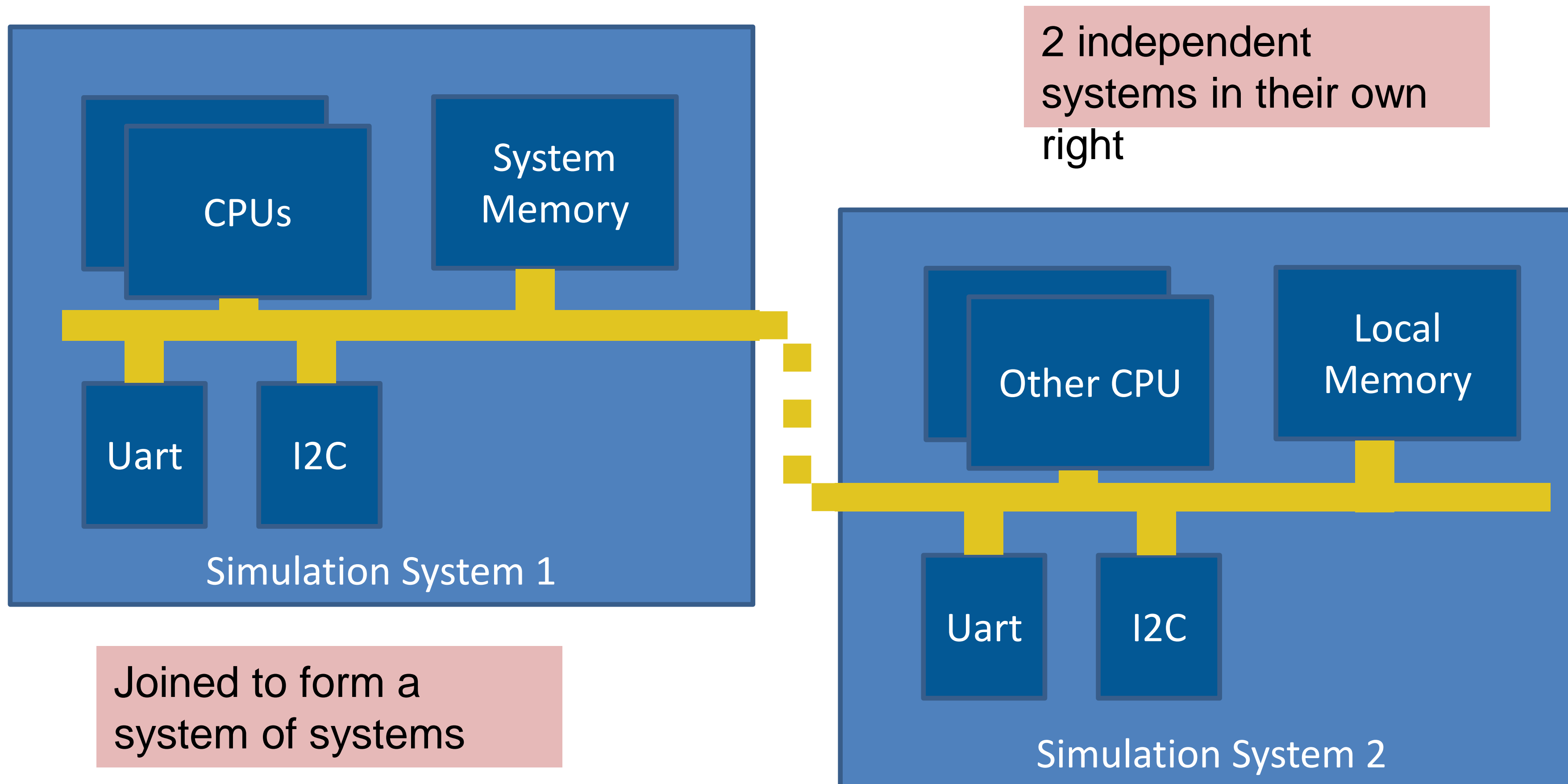
- This pair of functions provides an ‘opt-out’ for specific process to the suspend_all. The consequence is that if there is a process that has opted out, the kernel will not be able to suspend_all (as it would no longer be atomic).
- These functions can only be called from within a process.
- A process should only call suspendable/unsuspendable in pairs (multiple calls to either will be ignored).

© Accellera Systems Initiative

Connecting it all together.

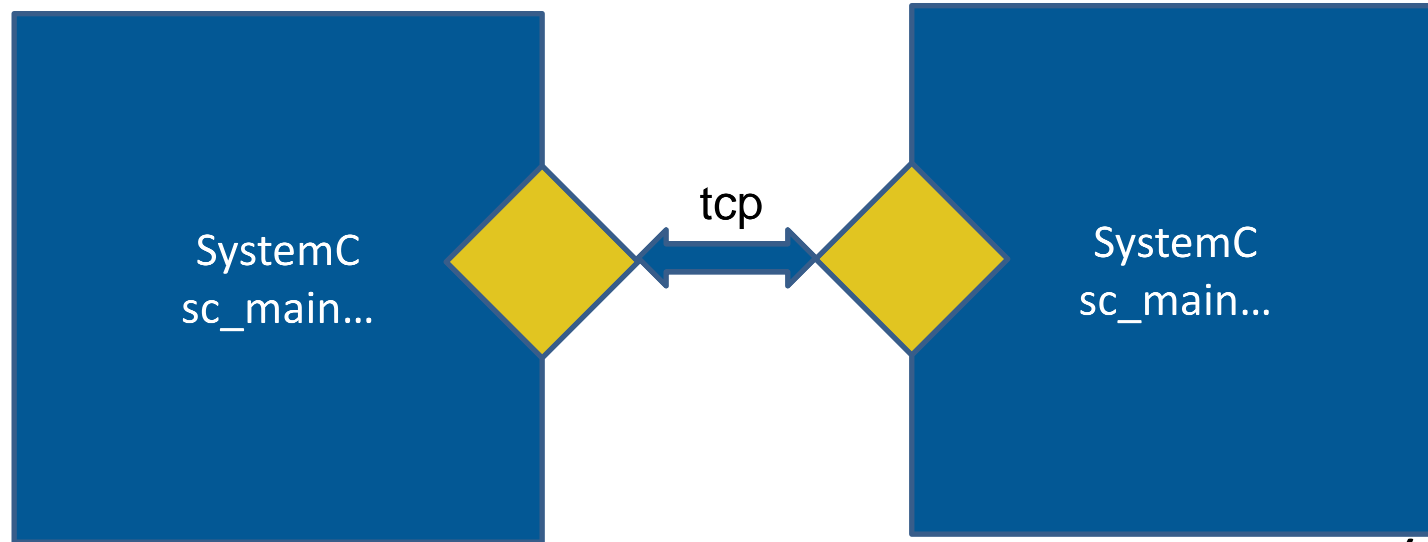
SYSTEMC'S OF SYSTEMC'S !!!

SystemCs of SystemCs



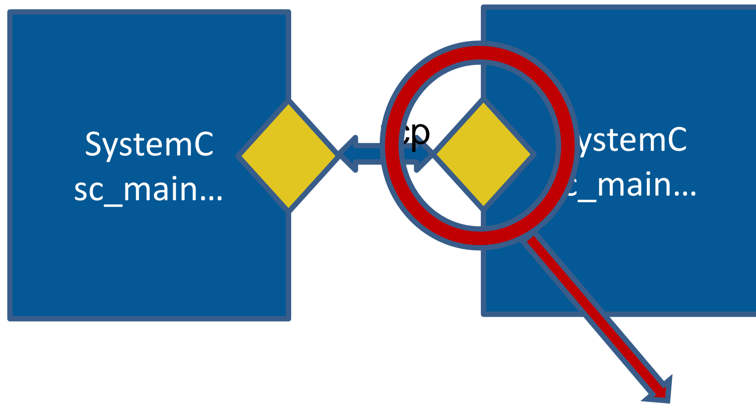
SystemC RPC

- Wouldn't it be nice …



(Nothing new…)

But now with “(un)suspend”

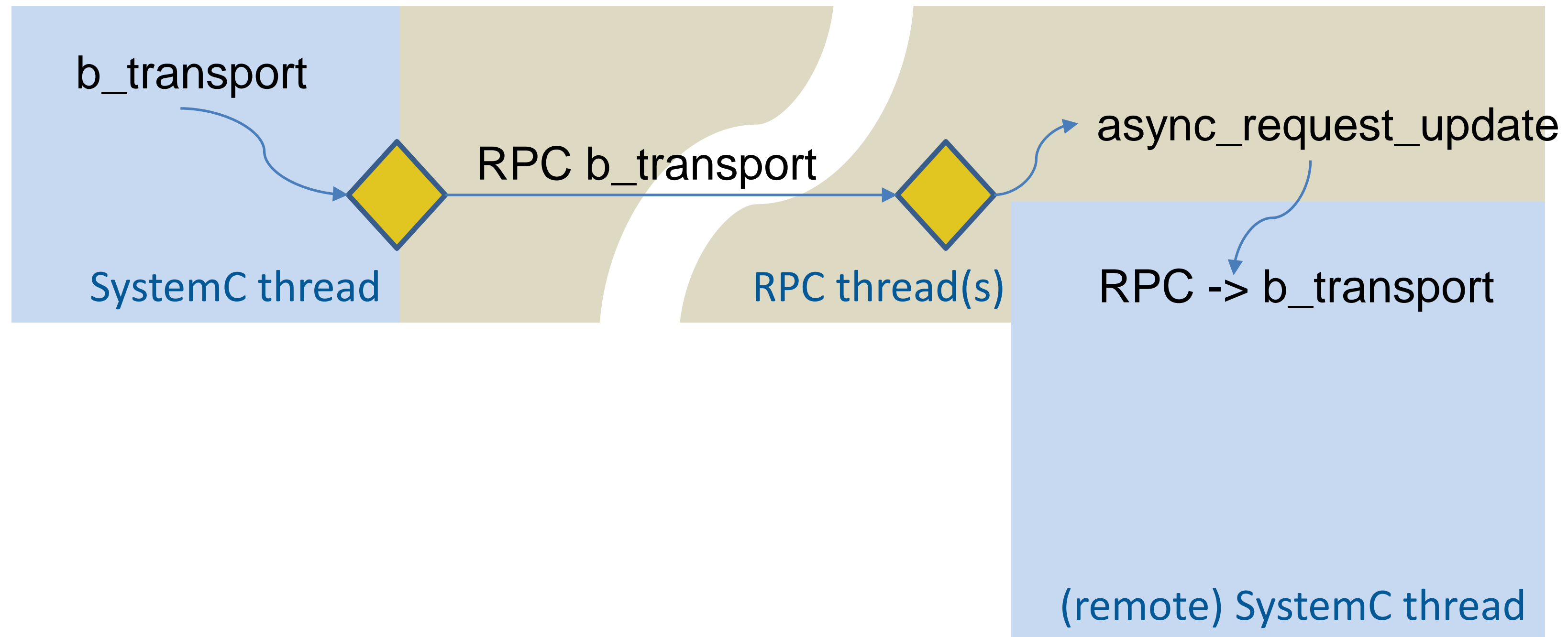


1. Carry TLM and sc_signal interface over TCP RPC calls
2. Use attach/detach and async events based on async_notify
3. (Optionally) use shared memory for DMI calls
4. Provide ‘synchronization policies’ between SystemC’s using (un)suspend
5. N-N mapping of TLM sockets and Signal’s to facilitate wiring.
6. CCI database is ‘shared’
7. SystemC code is executed on the SystemC thread(*)

* Mostly – see later

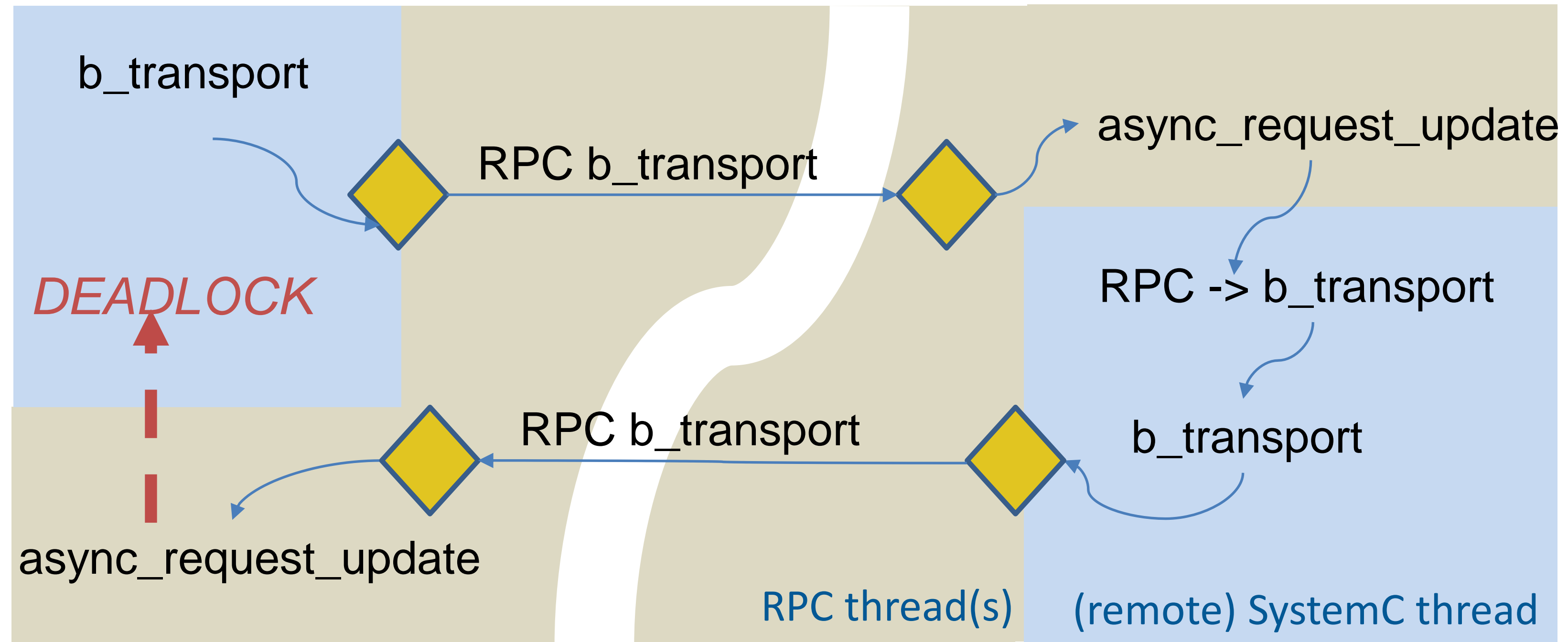
accellera SYSTEMS INITIATIVE
© Accellera Systems Initiative

Running on SystemC



© Accellera Systems Initiative

LOOPS !!!



Loop Rules

- b_transport loops wont work – find a better place to divide your system!

- But : What about a b_transport that causes a DMI invalidate !!!

- **Conclusion : DMI should be ‘thread safe’ !**
- **Allow get_dmi_ptr and invalidate_dmi to run on any thread!**

accellera SYSTEMS INITIATIVE™

© Accellera Systems Initiative

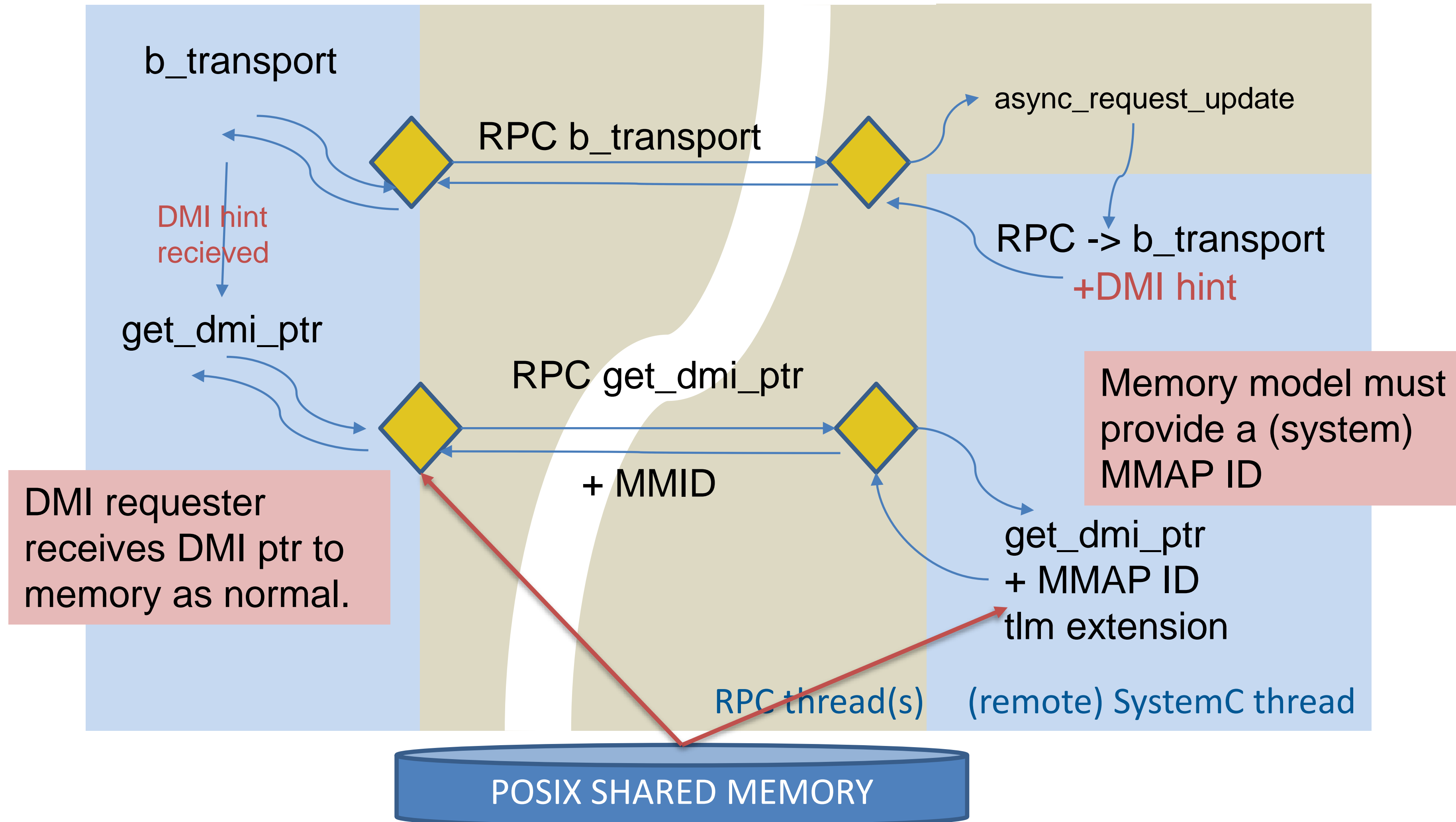
SYSTEMC™ EVOLUTION DAY
DEC 8, 2022 | MUNICH | GERMANY

Thread safe DMI ?

- The intention of the DMI interface is that it does not pass via notifications etc – so it should be possible to make thread safe.
- This would require ‘work’ on peoples DMI mechanisms.
- (at the very minimum, if a b_transport causes a DMI invalidate, then the invalidate must be executed on a separate thread otherwise there is a deadlock).
- Of course, the DMI interface could use async_events to trigger behavior in SystemC if required.

© Accellera Systems Initiative

DMI Identical call sequence to TLM-2.0



© Accellera Systems Initiative

Synchronisation

- **TLM 2.0 Rule : Delta time can’t be ‘negative’**

- The ‘Initiator’ simulation time has to be behind of the ‘target’. (so that the ‘delta’ time is positive)
- While one simulation is the initiator, time will be synchronized (with various ‘quantum’ policies).
- If the initiator swaps, then a synchronization would be forced.

(WIP)

CCI

- CCI parameter values may be set by one simulator and need to be used by the other.
- The RPC link simply needs to provide a way to set values (call-backs, aliases, etc are all tricky over RPC!)

- Special care is needed to translate the parameter names between the simulations.
- Parameters passed to the other simulation should be marked as consumed.

FAQ

- SystemC + ‘Something else’ – OF COURSE
  - The RPC protocol is simple
- Between hosts (No shared memory) - OF COURSE
  - Just don’t provide the DMI hint
- Handle other socket types – Fill your boots
  - This could be easily done, (come and help!)
- Handle nb_transport – Should be easy
  - Would be a natural fit ! (come and help!)

Show Me The Code !

- Current implementation:
- PassRPC class has N TLM ports and M Signal ports (bi directional and zero-or-more-bound)
- ShmemIDExtension holds the MMAP ID
- Memory handles MMAP (with block based late allocation).
- Makes use of a ‘multithread quantum keeper’
  – Which has a number of policies.
- Donation will be made to Accellera SCP.

© Accellera Systems Initiative