Intel® Simics® Simulator Internals Training

Lab 01-10

Threading

1 Execution Scheduling: Multicore Threading

In this lab, you will look at how the execution scheduling works when running multithreaded across the cores in a single target system. It uses the setup already used in the labs for chapter 05 of the internals training.

1.1 Set up a Multithreaded Simulation

1. **Start** a new simulation session using the same target as previously:

```
$ ./simics simics-internals-training/05-time-quanta
```

2. **Set** the simulator to run in multicore mode, i.e., trying to run the processors in parallel on separate host threads.

```
simics> set-threading-mode multicore
```

3. **Check** that this had the desired result:

```
simics> set-threading-mode
```

Which should show that the cell is in multicore mode, and that the **max-time-span** is now guiding the scheduler:

```
| cell       | mode      | #td    | time-quantum | max-time-span | min-latency |
|------------|-----------|--------|--------------|---------------|-------------|
| ribit.cell | multicore | 0x0005 |    (10.0 µs) |       10.0 µs |   (10.0 ms) |
```

4. **Check** the resulting thread domains. I.e., potential parallelism.

```
simics> list-thread-domains
```

This shows that all the processors and the clock get their own thread domain and can thus potentially run in parallel to each other:

```
| Cell       | Domain | Objects          |
|------------|--------|------------------|
| ribit.cell |   #0*  | ribit.cell       |
|            |    #1  | ribit.clock      |
|            |    #2  | ribit.unit[0].hart |
|            |    #3  | ribit.unit[1].hart |
|            |    #4  | ribit.unit[2].hart |
|            |    #5  | ribit.unit[3].hart |

* thread domains marked with '*' are cell thread domains. To keep output shorter, only
the cell, CPU, and clock objects are reported for them.
```

There is also a "cell thread domain" that holds all the devices, for locking purposes.

5. **Check the simulation status** for the number of simulation threads:

```
simics> sim.status
```

This should show that the number of threads to use is unlimited, which really means that the simulator is only limited by the hardware. It also shows a worker thread limit, which is a limit on the number of just-in-time compiler threads. Finally, the

"simulation threads limit" shows the actual number of simulation threads that have been used. For example:

```
                Thread limit : Unlimited
        Worker threads limit : 5
    Simulation threads limit : 0
```

The actual number of threads will depend on the host the simulator is running on.

6. **Run** until the notifier:

```
simics> bp.notifier.run-until name = i-synchronizer-release
```

7. **Check the time** across the processors, including both steps and actual time:

```
simics> list-processors -all -cycles -steps -time
```

The processors should be similar, but it is possible that some are slightly ahead. Unlike the serial case, the precise number of cycles will depend on how the simulation happened to run. It is likely to end up similar to the serial case, but there are no guarantees.

For example, this is a state that could happen:

```
┌──────────────────┬──┬─────────────┬──────────┬─────────┬─────────┬──────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │  Steps  │ Cycles  │ Time (s) │
├──────────────────┼──┼─────────────┼──────────┼─────────┼─────────┼──────────┤
│ ribit.clock      │  │ clock       │ 100.00 MHz │ n/a     │ 286_000 │     0.00 │
│ ribit.unit[0].hart │ * │ riscv-rv64  │ 100.00 MHz │ 284_042 │ 287_000 │     0.00 │
│ ribit.unit[1].hart │  │ riscv-rv64  │ 100.00 MHz │ 284_042 │ 287_000 │     0.00 │
│ ribit.unit[2].hart │  │ riscv-rv64  │ 100.00 MHz │ 284_042 │ 287_000 │     0.00 │
│ ribit.unit[3].hart │  │ riscv-rv64  │ 100.00 MHz │ 284_043 │ 286_501 │     0.00 │
└──────────────────┴──┴─────────────┴──────────┴─────────┴─────────┴──────────┘
* = selected CPU
```

8. **Check the simulation status** for the number of simulation threads used so far. The limit should be greater than zero, and less than or equal to five since there the max available parallelism in this setup is five.

```
simics> sim.status
```

9. Make sure that **`ribit.unit[0].hart`** is the current processor:

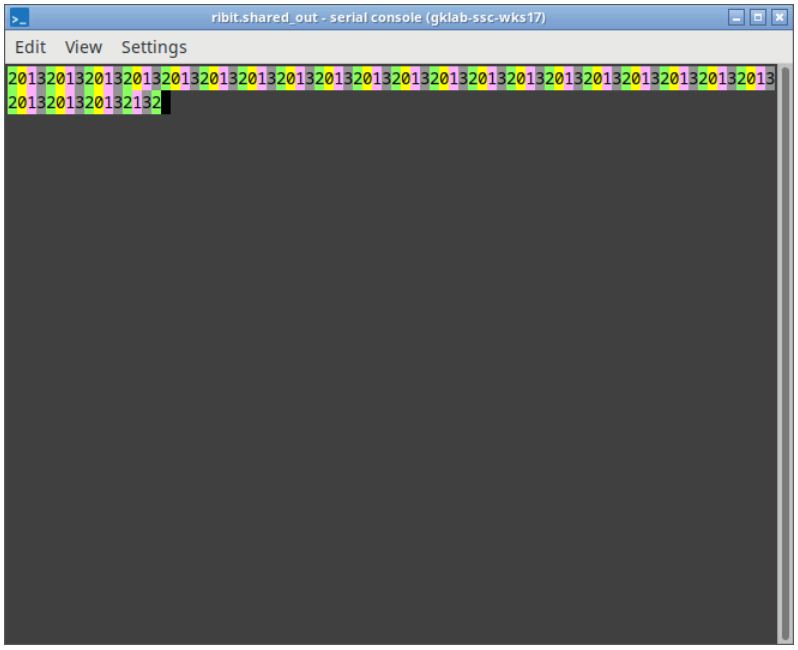
```
simics> pselect ribit.unit[0].hart
```

10. **Run 1000 cycles:**

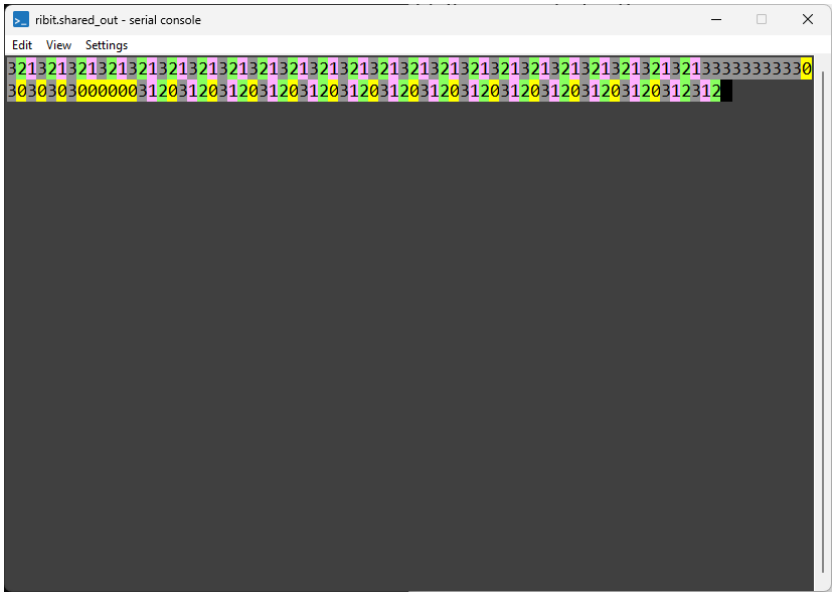
```
simics> run 1000 cycles
```

The output on the shared serial console will reflect how the different processors got scheduled onto host threads. The pattern might be regular or irregular, and the actual number of characters printed might vary.

For example, you could get a balanced execution like this:



Another example on a different host sees a more uneven execution:



There is a wrinkle to the parallelism. The shared serial console is driven by a device in the shared or global system. The same device is being accessed from each processor, and this requires synchronization that will introduce a certain amount of serialization on the execution. The way the locking for the shared cell thread domain works, the processors will queue up and wait for their turn, which will tend to create an interleaving pattern. If the processors were running completely on their own without interacting via a shared resource, they would likely drift apart more in time.

11. **Check the current time** on all processors:

```
simics> list-processors -all -cycles -steps -time
```

The processor that is the furthest ahead (highest cycle count) should be no more than one **max-time-span** apart from the processor that is lagging (has the lowest cycle count). All the other processors could be anywhere. It really depends on how far they have executed when told to stop.

The more even example above looks like this:

```
┌──────────────────┬──┬─────────────┬──────────┬─────────┬──────┬──────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │  Steps  │Cycles│Time (s)  │
├──────────────────┼──┼─────────────┼──────────┼─────────┼──────┼──────────┤
│ribit.clock       │  │clock        │100.00 MHz│n/a      │289000│    0.00  │
│ribit.unit[0].hart│* │riscv-rv64   │100.00 MHz│285042   │288000│    0.00  │
│ribit.unit[1].hart│  │riscv-rv64   │100.00 MHz│285049   │288007│    0.00  │
│ribit.unit[2].hart│  │riscv-rv64   │100.00 MHz│285085   │288043│    0.00  │
│ribit.unit[3].hart│  │riscv-rv64   │100.00 MHz│285049   │288007│    0.00  │
└──────────────────┴──┴─────────────┴──────────┴─────────┴──────┴──────────┘
```

The second example above looks like this. Processors one to three have executed almost half the time span before stopping, which is why more characters have been printed.

```
┌──────────────────┬──┬─────────────┬──────────┬─────────┬──────┬──────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │  Steps  │Cycles│Time (s)  │
├──────────────────┼──┼─────────────┼──────────┼─────────┼──────┼──────────┤
│ribit.clock       │  │clock        │100.00 MHz│n/a      │289_000│    0.00  │
│ribit.unit[0].hart│* │riscv-rv64   │100.00 MHz│285_042  │288_000│    0.00  │
│ribit.unit[1].hart│  │riscv-rv64   │100.00 MHz│285_553  │288_511│    0.00  │
│ribit.unit[2].hart│  │riscv-rv64   │100.00 MHz│285_553  │288_511│    0.00  │
│ribit.unit[3].hart│  │riscv-rv64   │100.00 MHz│286_058  │288_516│    0.00  │
└──────────────────┴──┴─────────────┴──────────┴─────────┴──────┴──────────┘
```

12. **Run for 2000 cycles** to see how the execution pattern evolves. There is no guarantee that it will look like the first 1000 cycles. Maybe repeat this a few times to get a larger sample set.

```
simics> run 2000 cycles
```

1.2 Combine with Cycles per Instruction

Multithreaded simulation still respects the clock frequencies and step rates of the processors, ensuring the same relative progress of different processors over time as when running in serial mode. Just not with the same interleaving.

Continue in the existing session.

13. Set processor three to run three instructions (steps) per cycle:

```
simics> ribit.unit[3].hart.set-step-rate 3
```

This has two effects that will affect the scheduling. It will make the processor run more instructions per unit of virtual time, so you would expect it to print more characters than the other processors over time. It will also make its progress in terms of virtual time per real time slower, as it has to do more work to move forward the same amount of virtual time.

14. **Run 1000 cycles:**

```
simics> run 1000 cycles
```

The output on the console should look pretty even still.

15. **Check the current time** on all processors:

```
simics> list-processors -all -cycles -steps -time
```

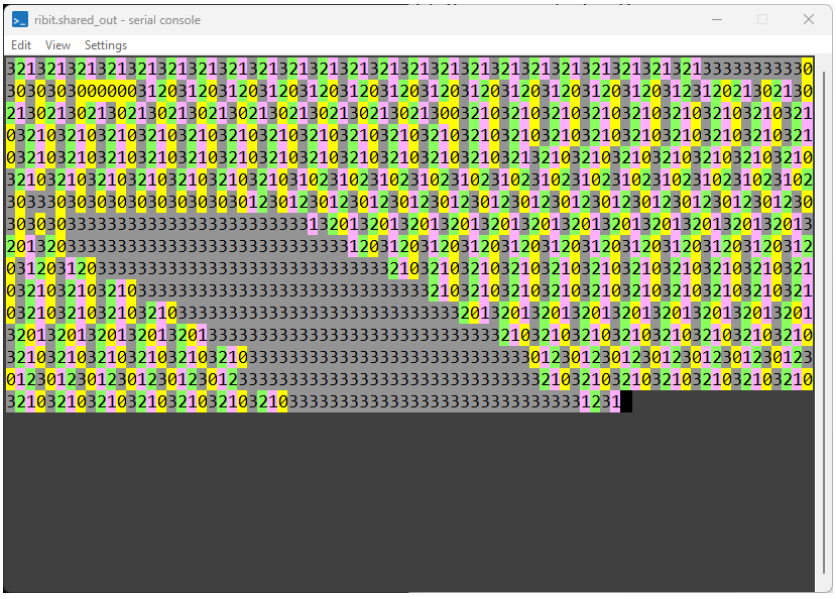
Processor 3 should now be lagging the other processors, as it has the most work to do.

16. **Run 5000 cycles:**

```
simics> run 5000 cycles
```

A longer run provides more opportunities for the processors to drift apart in virtual time and to see the effect of the scheduler.

For example, this pattern shows how processor three "catches up" with the other processors from time to time as it has fallen so far behind that all the processors stop and wait for it:



17. **Quit** this simulator session:

```
simics> quit
```

1.3 Limit the number of Threads

The simulator can artificially reduce the number of threads used to fewer than those available on the host.

18. **Start** a new simulation session using the same target as previously:

```
$ ./simics simics-internals-training/05-time-quanta
```

19. **Set** the simulator to run in multicore mode, i.e., trying to run the processors in parallel on separate host threads.

```
simics> set-threading-mode multicore
```

20. **Check** that this had the desired result:

```
simics> set-threading-mode
```

21. **Set the thread limit** to two threads. This restricts the simulator to never use more than two simulation threads to execute the work of the four instruction-set processors and the clock present in the simulation.

```
simics> set-thread-limit 2
```

22. **Run** until the notifier:

```
simics> bp.notifier.run-until name = i-synchronizer-release
```

23. **Run 5000 cycles:**

```
simics> run 5000 cycles
```

With just two host threads available, the execution will look different. It will tend to run two processors at a time, which is visible in the pattern printed to the shared console.

For example:

Another example, with a different pairing of the processors:

The execution that you observe is likely to be different.

24. **Quit** this simulator session:

```
simics> quit
```

The take-away is that multithreaded simulations are non-deterministic and the interleaving of operations from different processors when they access shared resources can be very random. There is no guarantee of balance, other than as dictated by the **max-time-span**.

2 Execution Scheduling: Multiple Cells

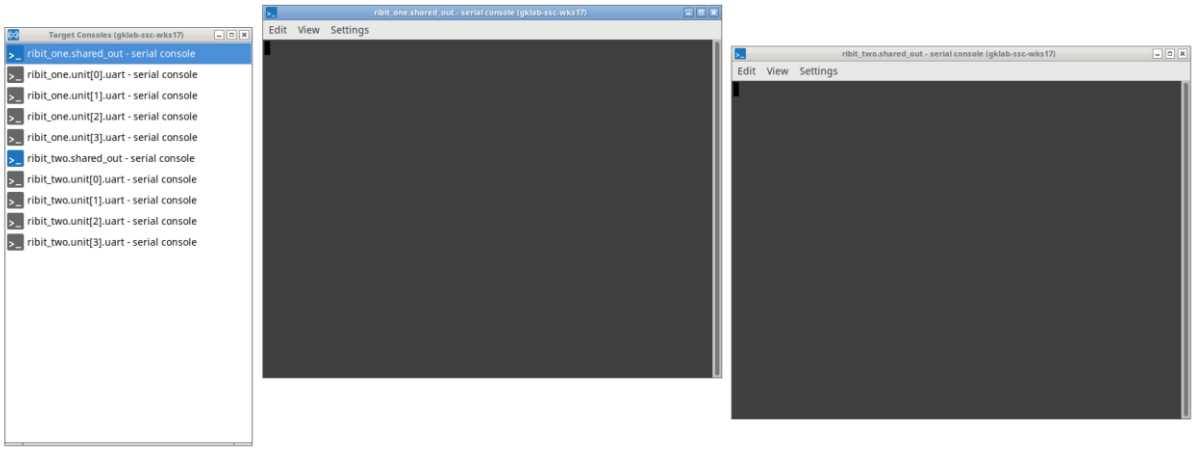
In this lab, you will see the effects of parallelizing the simulator execution across cells.

2.1 Start a Multiple-Cell Simulation

1. **Start** a new simulation session using the "multi" target:

```
$ ./simics simics-internals-training/05-time-quanta-multi
```

The target script hides all the per-processor serial consoles automatically since they really do not print anything particularly interesting. There is one shared console for each machine. The titles indicate the machines that they belong to:



2.2 Investigate the Cell Setup

The multiple-machine setup is split into cells, where each cell is like a mini-simulator in its own right.

2. **Investigate the cell setup** by looking at the thread domains:

```
simics> list-thread-domains
```

This shows that there are two cells in the system, each of them the same as the system used in the previous lab sections. The cells can run in parallel to each other.

```
┌──────────────┬──────┬──────────────────────────┐
│     Cell     │Domain│         Objects          │
├──────────────┼──────┼──────────────────────────┤
│ribit_one.cell│   #0*│ribit_one.cell            │
│              │      │ribit_one.clock           │
│              │      │ribit_one.unit[0].hart    │
│              │      │ribit_one.unit[1].hart    │
│              │      │ribit_one.unit[2].hart    │
│              │      │ribit_one.unit[3].hart    │
└──────────────┴──────┴──────────────────────────┘

┌──────────────┬──────┬──────────────────────────┐
│     Cell     │Domain│         Objects          │
├──────────────┼──────┼──────────────────────────┤
│ribit_two.cell│   #0*│ribit_two.cell            │
│              │      │ribit_two.clock           │
│              │      │ribit_two.unit[0].hart    │
│              │      │ribit_two.unit[1].hart    │
│              │      │ribit_two.unit[2].hart    │
│              │      │ribit_two.unit[3].hart    │
└──────────────┴──────┴──────────────────────────┘
* thread domains marked with '*' are cell thread domains. To keep output shorter, only
the cell, CPU, and clock objects are reported for them.
```

3. Another way to look at the cell setup is to use the **`list-processors`** command.

```
simics> list-processors -all
```

The **`Cell`** column shows the cell for each processor:

```
┌──────────────────┬──┬─────────┬──────────┬──────────────┐
│     CPU Name     │  │CPU Class│   Freq   │     Cell     │
├──────────────────┼──┼─────────┼──────────┼──────────────┤
│ribit_one.clock   │  │clock    │100.00 MHz│ribit_one.cell│
│ribit_one.unit[0].│* │riscv-rv6│100.00 MHz│ribit_one.cell│
│ribit_one.unit[1].│  │riscv-rv6│100.00 MHz│ribit_one.cell│
│ribit_one.unit[2].│  │riscv-rv6│100.00 MHz│ribit_one.cell│
│ribit_one.unit[3].│  │riscv-rv6│100.00 MHz│ribit_one.cell│
│ribit_two.clock   │  │clock    │100.00 MHz│ribit_two.cell│
…
```

4. Check the association from processors to cells, by reading the **`cell`** attribute of a processor object:

```
simics> list-attributes ribit_one.unit[0].hart cell
```

5. To find the cell of a non-processor object, **read** its **`queue`** attribute and then check the **`cell`** attribute of that object:

```
simics> ribit_two.shared_out->queue
```

This should return **`ribit_two.clock`**.

6. Follow the trail to the cell:

```
simics> ribit_two.clock->cell
```

7. The same operation can be done in Python, following the attribute chain:

```
simics> @conf.ribit_two.shared_out.attr.queue.attr.cell
```

8. Each cell has its own execution mode, which can be determined with the **`set-threading-mode`** command:

```
simics> set-threading-mode
```

The output shows that each cell is set to execute serially:

cell	mode	#td	time-quantum	max-time-span	min-latency
ribit_one.cell	serialized	0x1	10.0 µs	(10.0 µs)	10.0 ms
ribit_two.cell	serialized	0x1	10.0 µs	(10.0 µs)	10.0 ms

The **`min-latency`** determines how tightly synchronized the virtual time is across the simulation. The synchronization between cells is managed using sync domains.

9. **Find** the sync domain objects in the current simulation:

```
simics> list-objects class = sync_domain -class-desc
```

This finds a single sync domain:

Object	Class	Class description
default_sync_domain	<sync_domain>	controls synchronized nodes

10. **Check** the sync domain object setup:

```
simics> default_sync_domain.info
```

It shows that this sync domain contains the two cells, and that the sync time between them is 10 ms.

```
Information about default_sync_domain [class sync_domain]
=========================================================

    Min-latency : 10.0 ms
          Nodes : ribit_one.cell
                  ribit_two.cell
         Parent : none
```

11. Use the **`sync-info`** command to see the same information in a different form:

```
simics> sync-info
```

The **`sync-info`** command provides an overview of all sync domains in the simulator. In general, there can be a tree of domains with different levels of sync at different levels of the tree.

2.3 Run with Cells

As stated above, the synchronization between cells is different from the synchronization and temporal decoupling within a cell.

12. **Run** the currently selected processor for 50k cycles:

```
simics> run 50_000 cycles
```

This asks the current processor to run for 50k cycles, which works just like previously inside its cell. The other cell will likely (depends on available host threads) run concurrently with the cell that is being controlled by the command.

When the 50k cycles are up, the controlled cell will stop. The other cell will also stop as soon as it notices that the simulation is stopping, and the time difference between the two must be less than the min-latency.

13. **Check the current time** across the whole simulation:

```
simics> list-processors -all -cycles -steps
```

The second cell time is likely to be different from the first cell. Here is an example, your results will most likely be different. Note how in this case, the time in the second cell is up to 5000 cycles behind:

```
┌──────────────────────┬──┬─────────────┬──────────┬──────────────┬──────┬──────┐
│       CPU Name       │  │  CPU Class  │   Freq   │     Cell     │Steps │Cycles│
├──────────────────────┼──┼─────────────┼──────────┼──────────────┼──────┼──────┤
│ribit_one.clock       │  │clock        │100.00 MHz│ribit_one.cell│n/a   │ 51000│
│ribit_one.unit[0].hart│* │riscv-rv64   │100.00 MHz│ribit_one.cell│50000 │ 50000│
│ribit_one.unit[1].hart│  │riscv-rv64   │100.00 MHz│ribit_one.cell│50000 │ 50000│
│ribit_one.unit[2].hart│  │riscv-rv64   │100.00 MHz│ribit_one.cell│50000 │ 50000│
│ribit_one.unit[3].hart│  │riscv-rv64   │100.00 MHz│ribit_one.cell│50000 │ 50000│
│ribit_two.clock       │  │clock        │100.00 MHz│ribit_two.cell│n/a   │ 46000│
│ribit_two.unit[0].hart│  │riscv-rv64   │100.00 MHz│ribit_two.cell│46000 │ 46000│
│ribit_two.unit[1].hart│  │riscv-rv64   │100.00 MHz│ribit_two.cell│46000 │ 46000│
│ribit_two.unit[2].hart│  │riscv-rv64   │100.00 MHz│ribit_two.cell│45000 │ 45000│
│ribit_two.unit[3].hart│  │riscv-rv64   │100.00 MHz│ribit_two.cell│45000 │ 45000│
└──────────────────────┴──┴─────────────┴──────────┴──────────────┴──────┴──────┘
```

14. **Run** until the notifier hits – which means it hits on either cell.

```
simics> bp.notifier.run-until name = i-synchronizer-release
```

15. **Check the current time** across the whole simulation.

```
simics> list-processors -all -cycles -steps
```

16. For variety, **select processor core zero in cell two** as the current processor:

```
simics> pselect ribit_two.unit[0].hart
```

17. **Run 2000 cycles:**

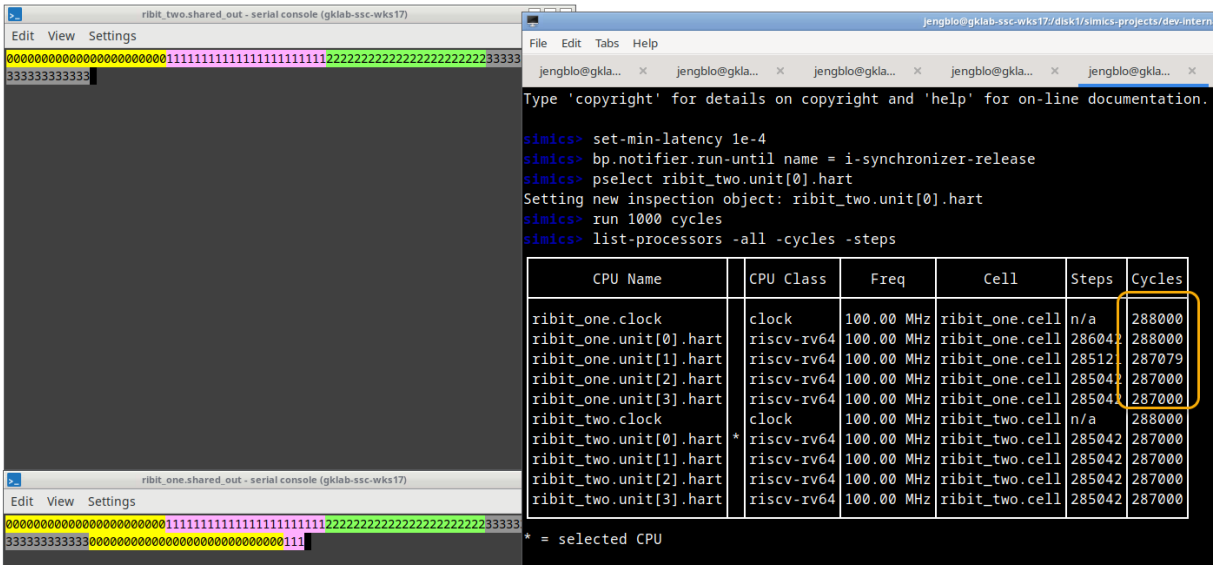
```
simics> run 2000 cycles
```

The simulation will stop precisely in cell two, but the stop in cell one might be earlier or later in time.

18. **Check the current time** across the whole simulation.

```
simics> list-processors -all -cycles -steps
```

19. **Look at the shared consoles.** For example, in this run cell one was ahead of cell two when the simulation stopped:



The serial console for cell one is the lower one, and it is consistent with the cycle count in cell one being ahead of cell two.

20. **Run 10000 cycles:**

```
simics> run 10_000 cycles
```

Note that the output to the console for each cell is totally identical and follows the time quantum setting. Unlike threading within a cell, multiple-cell threading is deterministic.

21. **Quit** this simulator session:

```
simics> quit
```

2.4 Visualizing the Effect of the Min-Latency

The min-latency setting determines the maximum time difference between the cells. When running each cell on its own thread and the same software on all cores, this latency is not all that visible since both cells progress at a similar rate. To truly see the effect, we run the simulator on a single thread, forcing the scheduler to run the cells round-robin on a single execution thread.

22. **Start** a new simulation session using the "multi" target, to get a clean start:

```
$ ./simics simics-internals-training/05-time-quanta-multi
```

23. **Reduce** the min-latency to 1 millisecond. The **`set-min-latency`** command uses an argument in seconds.

```
simics> set-min-latency 1e-3
```

24. **Check** that the change took effect:

```
simics> sync-info
```

The **Min-latency** shown should be 1 millisecond. Also note the "stop time" that indicates the furthest the time is allowed to be in any cell, given the current time of zero in all the cells.

25. Reduce the simulator to **using a single thread**:

```
simics> set-thread-limit 1
```

26. This should be visible in the simulator status:

```
simics> sim.status
```

Look for the thread limit line (the worker thread limit depends on your host):

```
…
Multithreading enabled : Enabled
          Thread limit : 1
  Worker threads limit : 4
…
```

27. **Set a breakpoint** on the notifier:

```
simics> bp.notifier.break name = i-synchronizer-release -once
```

By using a breakpoint it is possible to see which cell hits the notifier first.

28. **Run** until the notifier hits – which means it hits on either cell.

```
simics> r
```

When the simulation stops, check the output to see which cell hit the notifier:

```
simics> bp.notifier.break name = i-synchronizer-release -once
Breakpoint 1: Break on notifier 'i-synchronizer-release' on ribit_one.sync,
ribit_two.sync
simics> r
[ribit_one.sync] Breakpoint 1: ribit_one.sync triggered notifier 'i-synchronizer-
release'
```

29. **Check the current time** across the whole simulation.

```
simics> list-processors -all -cycles -steps
```

Note that the cell that hit the notifier is ahead of the other cell.

This should be cell one – the simulation setup is deterministic and should end up with the same results each time. If it is cell two, the rest of the results in this lab will be quite different.

30. **Check the synchronization state**:

```
simics> sync-info
```

It should show that cell one is slightly ahead of cell two and should be able to run until the Stop time before potentially switching to the other cell. Expected output:

```
simics> sync-info
Top sync domain: default_sync_domain
  Min-latency: 1.0 ms
  Report time: 0.00250 s
  Stop time  : 0.00300 s
  Node ribit_one.cell: 0.00285 s
  Node ribit_two.cell: 0.00250 s BLOCKING
```

31. **Make sure** the current processor is processor zero in cell one, i.e., in the cell that reached the breakpoint.

```
simics> pselect "ribit_one.unit[0].hart"
```

32. Run for 30 microseconds:

```
simics> run 100 us
```

This should result in only cell one running, and a few lines of output on its shared console. Cell two should still have a blank shared console.

33. Run another 50 microseconds:

```
simics> run 50 us
```

Now cell two should activate and run for a while, overtaking the output of cell one.

34. **Check the current time** across the whole simulation:

```
simics> list-processors -all -cycles -steps
```

The expectation is that cell two is ahead:

CPU Name		CPU Class	Freq	Cell	Steps	Cycles
ribit_one.clock		clock	100.00 MHz	ribit_one.cell	n/a	301_000
ribit_one.unit[0].hart	*	riscv-rv64	100.00 MHz	ribit_one.cell	298_993	300_000
ribit_one.unit[1].hart		riscv-rv64	100.00 MHz	ribit_one.cell	298_993	300_000
ribit_one.unit[2].hart		riscv-rv64	100.00 MHz	ribit_one.cell	298_993	300_000
ribit_one.unit[3].hart		riscv-rv64	100.00 MHz	ribit_one.cell	298_993	300_000
ribit_two.clock		clock	100.00 MHz	ribit_two.cell	n/a	350_000
ribit_two.unit[0].hart		riscv-rv64	100.00 MHz	ribit_two.cell	348_993	350_000
ribit_two.unit[1].hart		riscv-rv64	100.00 MHz	ribit_two.cell	348_993	350_000
ribit_two.unit[2].hart		riscv-rv64	100.00 MHz	ribit_two.cell	348_993	350_000
ribit_two.unit[3].hart		riscv-rv64	100.00 MHz	ribit_two.cell	348_993	350_000

35. This lab concludes, having shown that the synchronization in time between cells is typically looser than between individual cores inside a cell.

Quit this simulator session:

```
simics> quit
```

2.5 Time Quanta and Cells

Each cell can have its own local time quantum setting.

36. **Start** a new simulation session using the "multi" target:

```
$ ./simics simics-internals-training/05-time-quanta-multi
```

37. **Set** the time quantum length for cell one to 5 microseconds:

```
simics> set-time-quantum cell = ribit_one.cell 5e-6
```

38. **Set** the time quantum length for cell two to 20 microseconds:

```
simics> set-time-quantum cell = ribit_two.cell 2e-5
```

39. **Check** the results:

```
simics> set-time-quantum
```

The output shows the quantum per cell in microseconds, as well as the result in terms of cycles for all processors:

```
Current time quantum for ribit_one.cell: 5.0 µs
Current time quantum for ribit_two.cell: 20.0 µs
```

Cycles/quantum	Clock
500.00	ribit_one.clock
500.00	ribit_one.unit[0].hart
500.00	ribit_one.unit[1].hart
500.00	ribit_one.unit[2].hart
500.00	ribit_one.unit[3].hart
2000.00	ribit_two.clock
2000.00	ribit_two.unit[0].hart
2000.00	ribit_two.unit[1].hart
2000.00	ribit_two.unit[2].hart
2000.00	ribit_two.unit[3].hart

```
Default time quantum: 1000 cycles
```

40. **Run** until the notifier hits – which means it hits on either cell.

```
simics> bp.notifier.run-until name = i-synchronizer-release
```

41. **Run for 100 microseconds** (on the default processor):

```
simics> run 100 us
```

42. **Look** at the two serial consoles and note how the interleaving between the processors is smaller on cell one than on cell two due to the shorter time quantum. The precise stop state in terms of how much printing has happened on each console will vary as this is a multithreaded simulation.

The result should look something like this:

This demonstrates that each cell can have its own time quantum setting that is isolated from the settings in other cells, and that each cell internally runs a round-robin temporally decoupled scheduler.

43. **Quit** this simulator session:

```
simics> quit
```

3 Event Posting and Threading

Here we revisit the event posting lab from chapter 05, looking at how threading interact with event posting and event visibility across multiple separate processors in different thread domains.

3.1 Running a Multithreaded Simulation

When the simulation is running multithreaded, each clock will trigger their events at the same point in their own timelines. However, how those timelines align with other clocks will vary depending on the host scheduling of the worker threads in the simulator.

1. Start a new simulation session using the target **`simics-internals-training/05-events`**:

   ```
   $ ./simics simics-internals-training/05-events
   ```

2. Make the simulation run multithreaded:

   ```
   simics> set-threading-mode mode = multicore
   ```

3. Get the software stack started, by **running** until the synchronizer notifier hits:

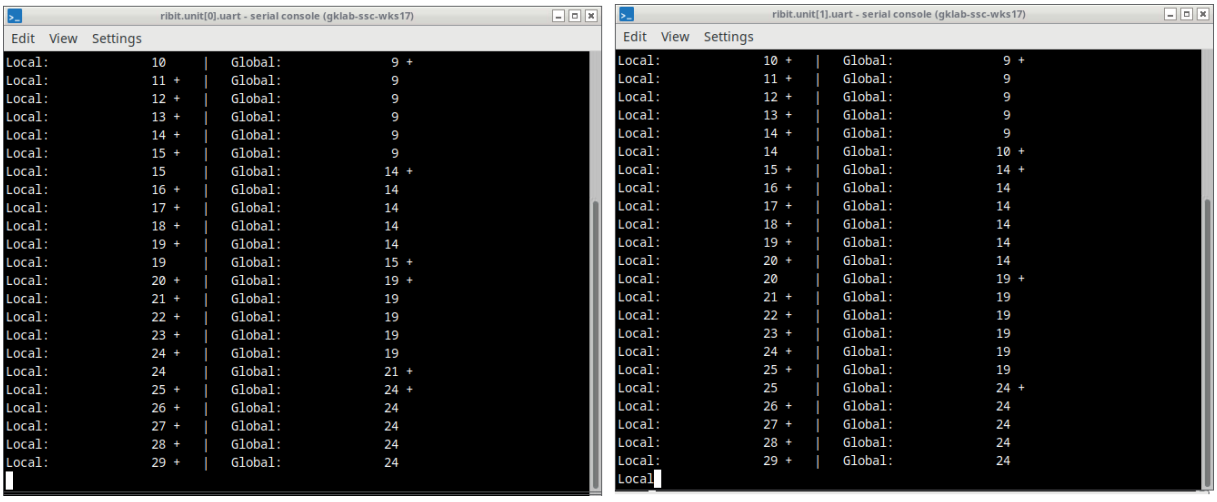
   ```
   simics> bp.notifier.run-until name = i-synchronizer-release
   ```

4. **Run** 300000 cycles:

   ```
   simics> run 300000 cycles
   ```

5. Look at the output on the serial consoles. You should see the global counter values increment alongside the local values. The precise jumps in the global value will depend on the local host environment and most likely differ from run to run. No matter the precise values, the key observation is that it is no longer tied to the time quantum boundaries in the way it was above.

   It might look like this:



6. The value of the global counter is likely looking a bit out of sync. Check its current value:

   ```
   simics> ribit.counter.status
   ```

Or:

```
simics> output-radix 10 3
simics> print-device-regs ribit.counter.bank.regs
```

7. Check the overall time in the simulator:

```
simics> list-processors -cycles -all
```

The global clock is most likely ahead of the processors (since it does less work for each cycle), but still its counter is likely behind.

The reason for this is that the global counter is started by code running on the processor in unit 0. This means that when the counter is started, the global clock is already ahead, and thus it starts counting blocks of 10000 cycles from a later point in its virtual time.

8. **Run** another 300000 cycles:

```
simics> run 300000 cycles
```

You should see the global counter lag behind the other counters. The precise offset will vary from line to line, since the execution is indeed non-deterministic.

9. Note the current time and counter value from the unit 0 counter.

```
simics> ribit.unit[0].counter.status
```

This will be used in the next step.

10. **Check** the current counter values and time on the other counters:

```
simics> ribit.unit[1].counter.status
simics> ribit.counter.status
```

They likely have different local times and different counter values.

11. **Quit** this simulation session.

```
simics> quit
```

3.2 Compare to a Serial Execution

12. Start a new simulation session using the target **`simics-internals-training/05-events`**:

```
$ ./simics simics-internals-training/05-events
```

Do not turn on threading. The point is to get a comparison point to the threaded simulation in the previous section.

13. Get the software stack to the same endpoint as in the previous section:

```
simics> bp.notifier.run-until name = i-synchronizer-release
simics> run 600000 cycles
```

14. Check the current time and counter value from the unit 0 counter.

```
simics> ribit.unit[0].counter.status
```

If you compare this to the values from the previous lab, you should see the same counter value. That depends only on the local behavior in unit 0: after all processors synchronize, we run for 600000 cycles, and that means the same number of increments to the local counter since the counting starts only after the synchronization point.

However, the actual virtual time is likely different compared to the threaded case. This is because the initial synchronization point also depends on when unit 1 gets its software all started. Which is not synchronized with unit 0 in a threaded simulation.

The semantics of devices attached to a certain clock or processor is the same in threaded and serial execution... as long as there is no interaction with other clocks that cause the simulation to become non-deterministic.

15. **Quit** this simulation session.

```
simics> quit
```