Intel® Simics® Simulator Internals Training

Lab 01-04

Object Lifecycle and Memory Management

1 Introduction

In this lab you will create and delete Intel® Simics® simulator objects with and without port objects and observe how they lifecycle goes from allocation over initialization through existence to de-initialization and de-allocation. Afterwards you will learn how to track and monitor overall memory usage, non-image memory usage and image memory usage.

2 Object Lifecycle

In this lab, you will experiment with and witness the life cycle of objects in the Simics® simulator. You will control and look at how the simulator creates and deletes objects in a Simics simulation.

2.1 Start Simics

1. Start a new, empty simulation session in your project.

   On Linux:

   ```
   ./simics
   ```

   On Windows:

   ```
   simics.bat
   ```

2.2 Create and delete a single object

This lab makes use of the **`serial-mux-out`** device class, which is a class written in C in order to have full access to all the lifecycle functions of the simulator API at the lowest level. You can find the source code in the training package, but it is not necessary to look at the source code in order to understand the lab.

2. At the simulator command line, raise the log level to 3, to engage the logging in the setup functions of the **`serial-mux-out`** device class (you have to set the log level globally, since you cannot set the log level for an object that does not yet exist):

   ```
   simics> log-level 3
   ```

3. For reference, enable showing the log level on all logs.

   ```
   simics> log-setup -level
   ```

4. Use **`SIM_create_object()`** to create a single instance of the **`serial-mux-out`** class.

   ```
   simics> @SIM_create_object("serial-out-mux", "smux", [])
   ```

   With log level 3, this should print a series of info logs as the device set up functions are called:

   ```
   [smux info 3] init() called
   [smux info 3] finalize() called
   [smux info 3] objects_finalized() called
   <the serial-out-mux 'smux'>
   ```

   Note that the device allocation function cannot log, but once **`alloc()`** has succeeded, all other setup functions can use the Intel Simics simulator log system.

5. Delete the object, using **`SIM_delete_object()`**. You need to use a reference to the actual object as it is wrapped in the Python API, not just the name as a string. Retrieve the object reference from the name using **`SIM_get_object()`**:

   ```
   simics> @SIM_delete_object(SIM_get_object("smux"))
   ```

The functions used to delete an object should log (note that Simics itself comes in and logs the deletion too):

```
[smux info 3] deinit() called
[sim info 3] deleting object smux from the configuration
[smux info 3] dealloc() called
0
```

6. Check that there is no **smux** object in the configuration:

```
simics> list-objects substr=smux
```

The command result should not show any objects.

7. Another way to check for the existence of an object is the command **object-exists**:

```
simics> object-exists name = smux
```

The command should return **FALSE** since no such object exists

2.3 Create a set of objects using SIM_add_configuration

While it is reasonable to create a single object using **SIM_create_object()**, the preferred API for object creation in the Intel Simics simulator is **SIM_add_configuration()**.With this call, it is possible to create a set of objects simultaneously, including all references back and forth between the objects. The API is used from Python, since it relies on creating Python **pre_conf_object** objects that represent the objects to be created. We will look at an example of that and then see how this plays out in a simulation.

8. **Open** a second terminal in your project directory (the Intel Simics simulator is already running in the first terminal).
9. **Identify the absolute location** of the file targets/simics-internals-training/04-many-mux.py that is part of the training package.

   On Linux

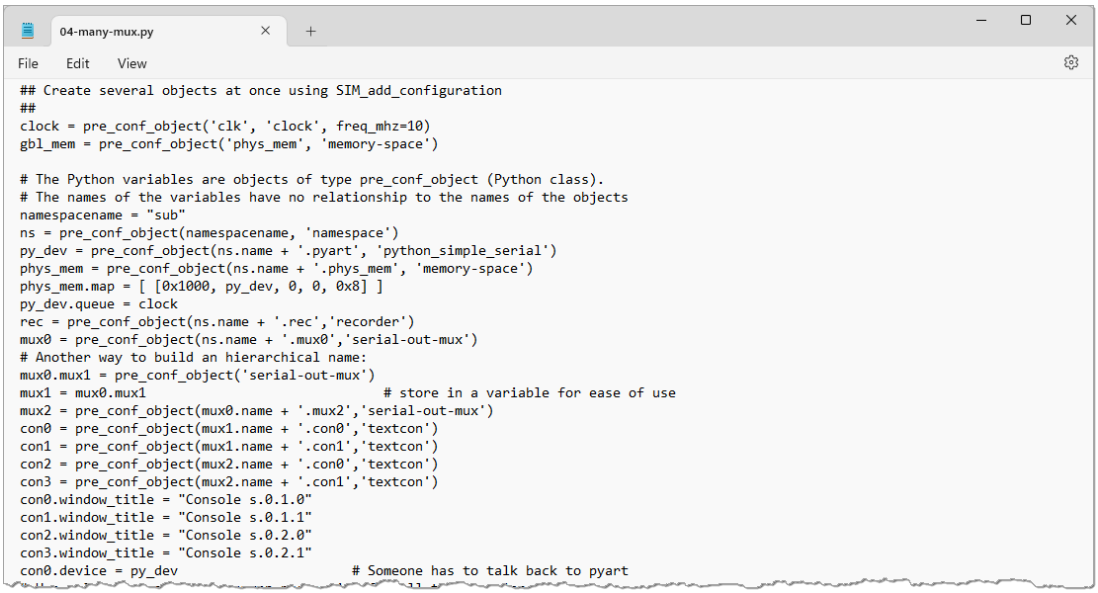
```
bin/lookup-file -f targets/simics-internals-training/04-many-mux.py
```

   On Windows

```
bin\lookup-file.bat -f targets/simics-internals-training/04-many-mux.py
```

   You will see the full path to the file, for use in the next step.

10. Using an editor of your choice, **open the file** that you identified in the previous step.
11. **Inspect** the file.



You see many calls to **pre_conf_object**, whose results are assigned to various Python variables. There is also some code establishing connections between the devices, but that is of no relevance for this lab.

Close to the end of the file you will see the call to **SIM_add_configuration** that takes the list of **pre_conf_objects** as a first argument. This call will perform some checks if all prerequisites for device instantiation are met, and if so, instantiate the set of **pre_conf_objects** and by that turns them into fully-fledged simulation objects.

12. Go to the **Intel Simics simulator CLI** (in case you closed the earlier session and start a new one, remember to set the log level back to 3).

13. Load the target **`simics-internals-training/04-many-mux`** which will run the file you just inspected.

```
simics> load-target "simics-internals-training/04-many-mux"
```

The log output shows the sequence in which the various setup functions and attribute setters in the serial mux objects are called (since only the serial mux has been written with logging for these calls):

```
[sub.mux0 info 3] init() called
[sub.mux0.mux1 info 3] init() called
[sub.mux0.mux2 info 3] init() called
[sub.mux0 info 3] set for attribute "original_target" called
[sub.mux0 info 3] set for attribute "mux_target" called
[sub.mux0.mux1 info 3] set for attribute "original_target" called
[sub.mux0.mux1 info 3] set for attribute "mux_target" called
[sub.mux0.mux2 info 3] set for attribute "original_target" called
[sub.mux0.mux2 info 3] set for attribute "mux_target" called
[sub.mux0 info 3] finalize() called
[sub.mux0.mux1 info 3] finalize() called
...
[sub.mux0.mux2 info 3] finalize() called
...
[sub.mux0 info 3] objects_finalized() called
[sub.mux0.mux1 info 3] objects_finalized() called
[sub.mux0.mux2 info 3] objects_finalized() called
...
```

All **`init()`** functions are called first, then all attribute setters, then all the **`finalize()`**, and finally **`objects_finalized()`**. The functions are called starting at the root of the object tree, with children always called after the parent.

14. Check the set of objects that were created.

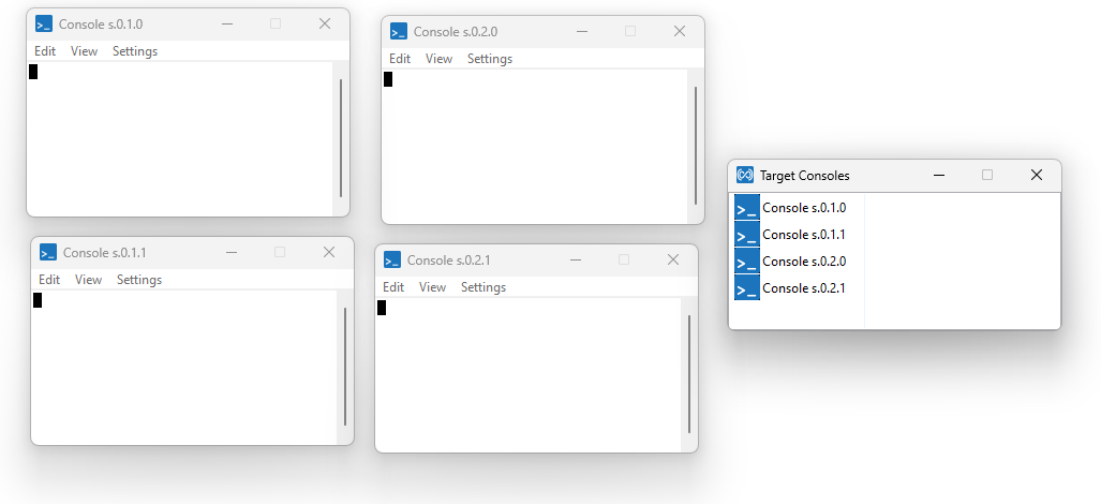
```
simics> list-objects -tree -show-port-objects
```

Look for the subtree rooted at "**`sub`**":

```
└ sub ┐
      ├ mux0 ┐
      │      ├ mux1 ┐
      │      │      ├ con0 ┐
      │      │      │      ├ frontend
      │      │      │      ├ tcp
      │      │      │      └ unix_socket
      │      │      └ con1 ┐
      │      │             ├ frontend
      │      │             ├ tcp
      │      │             └ unix_socket
      │      └ mux2 ┐
      │             ├ con0 ┐
      │             │      ├ frontend
      │             │      ├ tcp
      │             │      └ unix_socket
      │             └ con1 ┐
      │                    ├ frontend
      │                    ├ tcp
      │                    └ unix_socket
      ├ phys_mem
      ├ pyart
      └ rec
```

All these objects were created by the **`SIM_add_configuration()`** call. Note the "**`frontend`**", "**tcp**", and "**unix_socket**" objects underneath the consoles – these are port objects created as part of the console objects, and were not listed in the Python code.

15. Four console windows were created, alongside a console management window:



16. To send output to the consoles, write to a memory-mapped simple serial device found at **`0x200_1000`**. The send register is at offset **`0x00`** in the device's mapping. The

device itself is a very simple serial output device written in Python, not any real hardware UART.

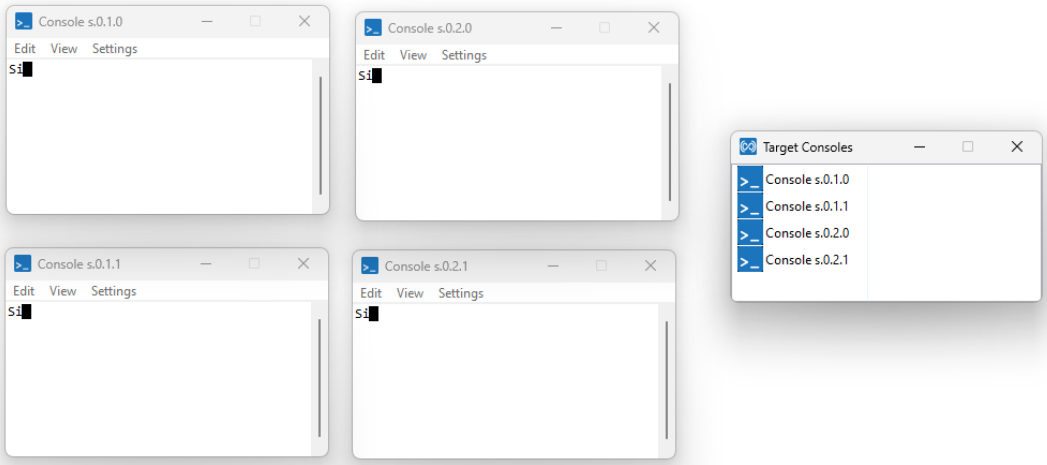
For example:

```
simics> phys_mem.write 0x200_1000 0x53 1
```

17. Send another character:

```
simics> phys_mem.write 0x200_1000 0x69 1
```

18. Check output in the **console** windows. The same output should appear in all windows.



2.4 Delete objects

19. Before deleting objects, inspect the current state of the event queue within the **clk** object.

```
simics> print-event-queue clk
```

You'll see that there are two events scheduled for cycle 1000. Both by the same object named **sub.pyart**. The device scheduled them when it received the characters through the two memory writes earlier.

20. Try to delete the "**sub**" namespace, which should bring along all its hierarchical sub-objects:

```
simics> @SIM_delete_object(conf.sub)
```

This will fail, as the Simics system sees that there are references to it from the outside.

```
[sim info 1] object 'sub.phys_mem' is still referred to in attribute 'map' of object
'phys_mem'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
simics.SimExc_General: There are still external references to the objects to delete
```

21. Remove the mapping that it complains about:

```
simics> phys_mem.del-map sub.phys_mem
```

22. Try to delete "**sub**" again.

```
simics> @SIM_delete_object(conf.sub)
```

Check the log output from the deletion functions in the mux objects, along with the log messages from Simics itself

```
[sub.mux0 info] deinit() called
[sub.mux0.mux1 info] deinit() called
[sub.mux0.mux2 info] deinit() called
[sim info] deleting object sub from the configuration
[sim info] deleting object sub.mux0 from the configuration
[sub.mux0 info] dealloc() called
[sim info] deleting object sub.mux0.mux1 from the configuration
[sub.mux0.mux1 info] dealloc() called
[sim info] deleting object sub.mux0.mux1.con0 from the configuration
[sim info] deleting object sub.mux0.mux1.con0.frontend from the configuration
[sim info] deleting object sub.mux0.mux1.con0.tcp from the configuration
[sim info] deleting object sub.mux0.mux1.con0.unix_socket from the configuration
[sim info] deleting object sub.mux0.mux1.con1 from the configuration
[sim info] deleting object sub.mux0.mux1.con1.frontend from the configuration
[sim info] deleting object sub.mux0.mux1.con1.tcp from the configuration
[sim info] deleting object sub.mux0.mux1.con1.unix_socket from the configuration
[sim info] deleting object sub.mux0.mux2 from the configuration
[sub.mux0.mux2 info] dealloc() called
[sim info] deleting object sub.mux0.mux2.con0 from the configuration
[sim info] deleting object sub.mux0.mux2.con0.frontend from the configuration
[sim info] deleting object sub.mux0.mux2.con0.tcp from the configuration
[sim info] deleting object sub.mux0.mux2.con0.unix_socket from the configuration
[sim info] deleting object sub.mux0.mux2.con1 from the configuration
[sim info] deleting object sub.mux0.mux2.con1.frontend from the configuration
[sim info] deleting object sub.mux0.mux2.con1.tcp from the configuration
[sim info] deleting object sub.mux0.mux2.con1.unix_socket from the configuration
[sim info] deleting object sub.phys_mem from the configuration
[sim info] deleting object sub.pyart from the configuration
[sim info] deleting object sub.rec from the configuration
0
```

Note how **deinit()** is called on all objects, and then **dealloc()** just when Simics logs that it is deleting the object in question. Also note that **sub.pyart** is part of the deleted subsystem, while the clock **clk** has not been deleted.

23. The device **sub.pyart** had events scheduled on the **clk** object. Check the state of the event queue again.

```
simics> print-event-queue clk
```

Note that the events are gone. The simulator ensures that when a device is deleted, all its pending events are cancelled.

24. Check that nothing is left of the object tree:

```
simics> list-objects substr = sub -tree
```

Which should return empty output.

25. A second way to check for its non-existence:

```
simics> object-exists sub
```

This should return FALSE.

26. **Quit** the simulation session.

```
simics> quit
```

3 Object Creation and Deletion with Port Objects

The previous lab only used objects with interfaces directly on the objects themselves. Typically, objects use port objects to provide their interfaces. When coding in DML, port object code is added by the DML compiler. In this case, a C device will be used that creates a port object class "by hand" to demonstrate what it looks like at the simulator API level.

3.1 Start Simics

1. Start a new, empty simulation session:

   On Linux:

   ```
   $ ./simics
   ```

   On Windows:

   ```
   C:\...> simics.bat
   ```

2. When Simics has started, go to the Simics command line.

3.2 List classes

3. Use **`list-classes`** to list the classes containing the string "**`mux`**" in their name. Also, use **`-m`** to list the module they come from:

   ```
   simics> list-classes substr = mux -m
   ```

   The result should be three classes:

   ```
   The following classes are available:
   ┌──────────────────────────┬──────────────┬──────────────────────────────┐
   │          Class           │    Module    │       Short description      │
   ├──────────────────────────┼──────────────┼──────────────────────────────┤
   │serial-out-mux            │serial-out-mux│N/A (module is not loaded yet)│
   │serial-out-mux-p          │serial-out-mux│N/A (module is not loaded yet)│
   │serial-out-mux-serial-in  │serial-out-mux│N/A (module is not loaded yet)│
   └──────────────────────────┴──────────────┴──────────────────────────────┘
   ```

   Note how all classes come from the same module. The class with suffix "-p" is the serial mux that uses port objects and the class serial-out-mux-serial-in is the class that the ports objects will be create from.

4. Check the help on the port class:

   ```
   simics> help serial-out-mux-serial-in
   ```

   Since the module has not been loaded (no object has been created and no explicit load-module command issued), the result is that no description is available:

```
Class serial-out-mux-serial-in

   Description
      This class has not been loaded into Simics yet. To access its documentation,
      load the serial-out-mux module (this can be done by running the 'load-module
      serial-out-mux' Simics CLI command).

   Provided By
      serial-out-mux
```

5. Load the module. This is only needed to access the help before creating any objects of the class.

```
simics> load-module serial-out-mux
```

6. Check the help again.

```
simics> help serial-out-mux-serial-in
```

   This should provide a bit more information, even though there is not all that much to say about the port class:

```
simics> help serial-out-mux-serial-in
Class serial-out-mux-serial-in

   Description
      Class demonstrating the use of port classes to create, port objects, for the
      serial out multiplexer main class.

   Interfaces Implemented
      conf_object, log_object, serial_device

   Provided By
      serial-out-mux (from Training)

Command List

   Commands defined by interface conf_object
      log-group, log-level

   Commands
      info                print information about the object
      status              print status of the object
```

7. Load the explore-memory-map module, to see some port classes that are created using **`SIM_register_simple_port()`**.

```
simics> load-module explore-mem-map-device
```

8. List the classes called "explore"-something:

```
simics> list-classes substr=explore -show-port-classes
```

   This finds several classes:

```
The following classes are available:
```

Class	Short description
explore-mem-map-device	memory map exploration
explore-mem-map-device.logger	bank logging all accesses without storing data
explore-mem-map-device.regs	register bank with example register set
explore-mem-map-device.regs_o_p	little Endian (LE) example register bank
explore-mem-map-device.regs_o_p_be	big Endian (LE) example register bank

Note that the port classes have names like "**main_class.port_name**" with a dot in the name. These names are automatically generated when Simics generates the classes, using the **SIM_register_simple_port()** API call.

9. Check the help on the **logger** port class:

```
simics> help explore-mem-map-device.logger
```

Note that the DML compiler has generated quite a bit of functionality for this class:

```
lass explore-mem-map-device.logger

   Description
      Register bank that logs all accesses but does not actually store any data

   Interfaces Implemented
      bank_instrumentation_subscribe, conf_object, instrumentation_order, io_memory,
log_object,
      register_view, register_view_catalog, register_view_read_only

   Port Interfaces
      core_dev_access_count (probe) : Port for generic device access count

   Provided By
      explore-mem-map-device (from Training)

Command List

   Commands defined by interface bank_instrumentation_subscribe
      bp-break-bank, bp-run-until-bank, bp-trace-bank, bp-wait-for-bank

   Commands defined by interface conf_object
      log-group, log-level

   Commands defined by interface instrumentation_order
      instrumentation-move, instrumentation-order
```

3.3 Create a set of objects, including port objects

Like in the previous lab, you will create a set of serial mux devices by loading the target **simics-internals-training/04-many-mux**. You will inform the target script that you want to use the model variant that uses ports by setting a script parameter and hence, when connecting the serial mux devices, instead of pointing directly at the other device model, the script will point at the port in each device.

10. Go back to the **simulator command line.** Raise the log level to 3, to engage the logging in the setup functions of the **`serial-mux-out`** device:

```
simics> log-level 3
```

11. For reference, show the log level on all log messages.

```
simics> log-setup -level
```

12. Load the target simics-internals-training/04-many-mux again.

```
simics> load-target "simics-internals-training/04-many-mux" use_ports =
TRUE
```

The log output shows the sequence of the setup. Note how the port objects are added the set of created objects, and how they are created as part of the set of objects underneath their parent object:

```
[sub.mux0 info 3] init() called
[sub.mux0.mux1 info 3] init() called
[sub.mux0.mux1.port.serial_in info 3] (port object) init() called
[sub.mux0.mux2 info 3] init() called
[sub.mux0.mux2.port.serial_in info 3] (port object) init() called
[sub.mux0.port.serial_in info 3] (port object) init() called
…
[sub.mux0 info 3] objects_finalized() called
[sub.mux0.mux1 info 3] objects_finalized() called
[sub.mux0.mux1.port.serial_in info 3] (port object) objects_finalized() called
[sub.mux0.mux2 info 3] objects_finalized() called
[sub.mux0.mux2.port.serial_in info 3] (port object) objects_finalized() called
[sub.mux0.port.serial_in info 3] (port object) objects_finalized() called
```

13. Check the set of created objects:

```
simics> list-objects -tree namespace=sub -show-port-objects
```

With the port objects, there are now "**`.port.serial_in`**" port objects underneath the three mux devices:

```
┐
├ mux0 ┐
│       ├ mux1 ┐
│       │       ├ con0 ┐
│       │       │       ├ frontend
│       │       │       ├ tcp
│       │       │       └ unix_socket
│       │       ├ con1 ┐
│       │       │       ├ frontend
│       │       │       ├ tcp
│       │       │       └ unix_socket
│       │       └ port ┐
│       │               └ serial_in
│       ├ mux2 ┐
│       │       ├ con0 ┐
│       │       │       ├ frontend
│       │       │       ├ tcp
│       │       │       └ unix_socket
│       │       ├ con1 ┐
│       │       │       ├ frontend
│       │       │       ├ tcp
│       │       │       └ unix_socket
│       │       └ port ┐
│       │               └ serial_in
│       └ port ┐
│               └ serial_in
├ phys_mem
├ pyart
└ rec
```

Note how the **port** namespace for **mux0** exists at the same level as the subordinate objects in the hierarchy.

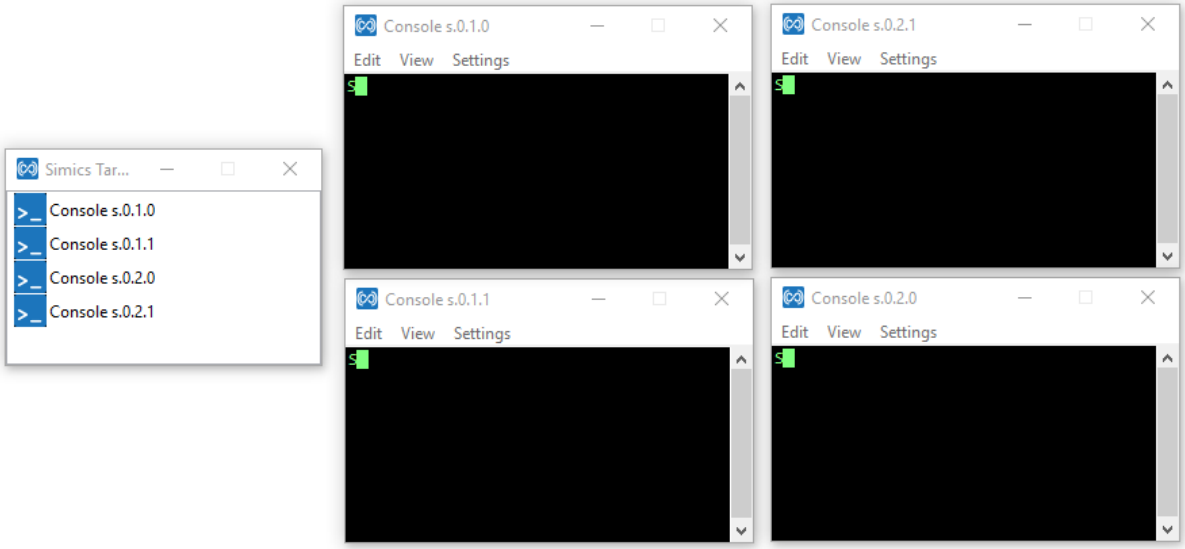
14. Test the flow of characters through the serial devices:

```
simics> phys_mem.write 0x200_1000 0x53 1
```

Check the log output for the flow from the **pyart** object through the input ports (**serial_in**) to the main object **write** function onwards.

```
[sub.pyart info 2] Write to transmit register (0x53)
[sub.pyart info 2] Sending character to console (0x53)
[sub.mux0.port.serial_in info 2] (port object) write(), incoming value: 0x53 ('S')
[sub.mux0 info 2] (main object) write(), incoming value: 0x53 ('S')
[sub.mux0 info 3] Passing on to mux target
[sub.mux0.mux2.port.serial_in info 2] (port object) write(), incoming value: 0x53 ('S')
[sub.mux0.mux2 info 2] (main object) write(), incoming value: 0x53 ('S')
[sub.mux0.mux2 info 3] Passing on to mux target
[sub.mux0.mux2 info 3] Passing on to original target
[sub.mux0 info 3] Passing on to original target
[sub.mux0.mux1.port.serial_in info 2] (port object) write(), incoming value: 0x53 ('S')
[sub.mux0.mux1 info 2] (main object) write(), incoming value: 0x53 ('S')
[sub.mux0.mux1 info 3] Passing on to mux target
[sub.mux0.mux1 info 3] Passing on to original target
```

15. Check the output in the **console** windows:



3.4 Delete objects

16. Remove the mapping that prevents deletion (just like in the previous lab):

```
simics> phys_mem.del-map sub.phys_mem
```

17. Delete the "**sub**" object:

```
simics> @SIM_delete_object(conf.sub)
```

The log messages indicate the order in which the deletion functions are called.

First, **deinit()** is called in tree order from the top:

```
[sub.mux0 info 3] deinit() called
[sub.mux0.mux1 info 3] deinit() called
[sub.mux0.mux1.port.serial_in info 3] (port object) deinit() called
[sub.mux0.mux2 info 3] deinit() called
[sub.mux0.mux2.port.serial_in info 3] (port object) deinit() called
[sub.mux0.port.serial_in info 3] (port object) deinit() called
```

Next, the actual deletion is performed by calling **dealloc()**. Only the objects from the serial-out-mux class have logging in place for this, but Simics itself logs each deleted object from the **sim** object at level 3:

```
[sim info 3] deleting object sub from the configuration
[sim info 3] deleting object sub.mux0 from the configuration
[sub.mux0 info 3] dealloc() called
[sim info 3] deleting object sub.mux0.mux1 from the configuration
[sub.mux0.mux1 info 3] dealloc() called
[sim info 3] deleting object sub.mux0.mux1.con0 from the configuration
[sim info 3] deleting object sub.mux0.mux1.con0.frontend from the configuration
[sim info 3] deleting object sub.mux0.mux1.con1 from the configuration
[sim info 3] deleting object sub.mux0.mux1.con1.frontend from the configuration
[sim info 3] deleting object sub.mux0.mux1.port from the configuration
[sim info 3] deleting object sub.mux0.mux1.port.serial_in from the configuration
[sub.mux0.mux1.port.serial_in info 3] (port object) dealloc() called
[sim info 3] deleting object sub.mux0.mux2 from the configuration
[sub.mux0.mux2 info 3] dealloc() called
[sim info 3] deleting object sub.mux0.mux2.con0 from the configuration
[sim info 3] deleting object sub.mux0.mux2.con0.frontend from the configuration
[sim info 3] deleting object sub.mux0.mux2.con1 from the configuration
[sim info 3] deleting object sub.mux0.mux2.con1.frontend from the configuration
[sim info 3] deleting object sub.mux0.mux2.port from the configuration
[sim info 3] deleting object sub.mux0.mux2.port.serial_in from the configuration
[sub.mux0.mux2.port.serial_in info 3] (port object) dealloc() called
[sim info 3] deleting object sub.mux0.port from the configuration
[sim info 3] deleting object sub.mux0.port.serial_in from the configuration
[sub.mux0.port.serial_in info 3] (port object) dealloc() called
[sim info 3] deleting object sub.phys_mem from the configuration
[sim info 3] deleting object sub.pyart from the configuration
[sim info 3] deleting object sub.rec from the configuration
```

The port objects are deleted and their deallocation functions called in the same way as other child objects.

18. Check that the objects are indeed deleted. In contrast to the previous section's check (in exercise 2.4) we will not filter the result to see that there are still other objects in the simulation.

```
simics> list-objects -tree
```

You will notice that the unfiltered list of objects no longer contains the object **sub** or any of its own subobjects.

19. **Quit** this Simics session.

```
simics> quit
```

4 Memory Tracking with Probes

The first thing to do when it comes to memory tracking is to see how the simulator memory usage develops over time. You only need to dig deeper if the trend look suspicious. We will use a simple test setup here that is designed to consume a lot of host memory, via simulator mechanisms and not by leaking memory in the simulator itself.

4.1 Understanding the test application

1. Start a new simulation session for the target **`simics-internals-training/04-memory-usage`**:

   On Linux:

   ```
   ./simics simics-internals-training/04-memory-usage
   ```

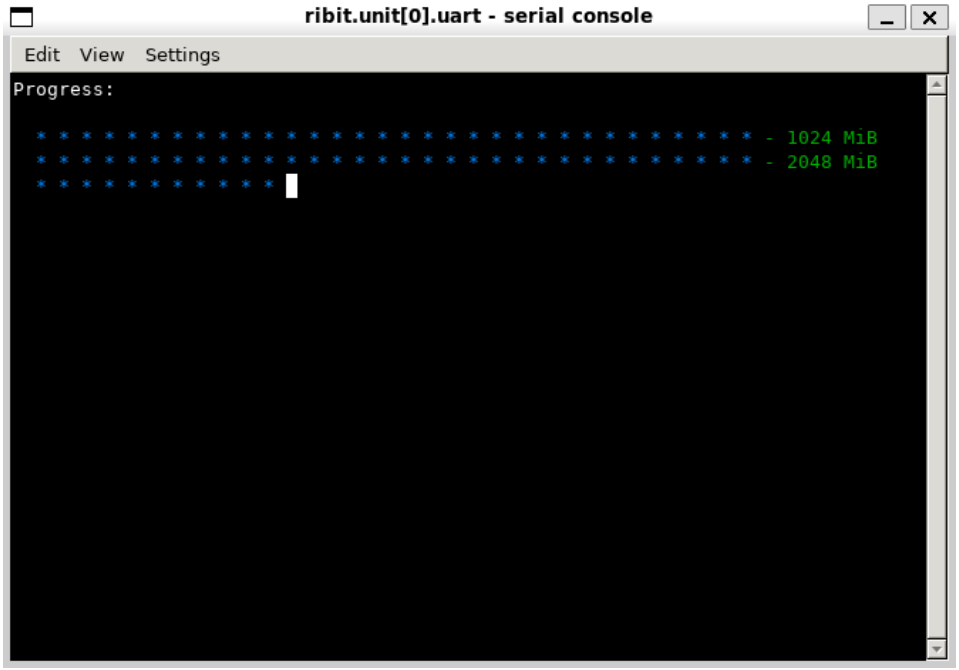
   On Windows:

   ```
   simics.bat simics-internals-training/04-memory-usage
   ```

2. Run the simulation for 0.5 seconds and observe the serial console.

   ```
   simics> run-seconds 0.5
   ```

   You see the serial filling with asterisks and at the end of a line you see a memory size number like “1024 MiB”.



   The application works within an area of physical memory, writing an 8-byte value at an offset that is an integer multiple of 1024 from the start of the memory area. Every asterisk indicates that 32768 writes have been performed. With 32 asterisks per row, each row represents that 32*32768*1024, or 1024 MiB, have been covered. The end of the line always shows the accumulated memory coverage.

Since the simulator image system allocates target memory in pages of 8kiB size and it only allocates such pages when they are accessed and changed (either compared to the underlying image object or compared to the initial empty state) that means that every eight access will allocate a new target memory page in the image system. Each asterisk represents the allocation of 4096 target memory pages.

As the program runs, the simulator will need a lot of host memory to hold the content of the target memory pages.

Not visible here, but important to understand is that the code executes a magic instruction after each allocation. We will leverage that later.

3. **Quit** the simulation

```
simics> quit
```

4.2 Enabling Probes and Check Available Probe Kinds

4. Start the simulation with the same target as before.

   On Linux:

```
./simics simics-internals-training/04-memory-usage
```

   On Windows:

```
simics.bat simics-internals-training/04-memory-usage
```

5. Enable probes framework:

```
simics> enable-probes
```

   Since enabled probes can introduce overheads and slow the simulation down they are disabled by default.

6. The exact set of available probes depend on the version of the Intel Simics simulator you are using and the model you are running.

   For this lab, let's search for probe kinds with "mem" in their name that should be related to memory:

```
simics> probes.list-kinds substr = mem
```

   You will see a few different probes. Read their descriptions to get a feel for what is possible. Note that some probes access the state of the host itself, not just the simulator process.

4.3 Create a Probe Monitor

Probes can be read and polled manually using the **probes.read** command, but when tracking memory, active regular monitoring is preferred. This is performed using probe monitors, which regularly read out one or more probes and report their value.

7. Create a monitor that reads values every half second in real time:

```
simics> new-probe-monitor clock = "ribit.unit[0].hart" sampling-mode =
realtime interval = 0.5
```

You can also do sampling based on virtual time or simulation notifications. Note that the sampling will only happen while the simulation is in running mode.

8. Add probes to the monitor. For memory, it makes sense to track both the simulator process memory and the overall host memory consumption (as the simulator process eats into the host memory). Since the test program uses the image system, it should also be monitored.

```
simics> pm0.add-probe mode=current probe=sim:sim.process.memory.resident
simics> pm0.add-probe mode=delta probe=sim:sim.process.memory.resident
simics> pm0.add-probe mode=current probe=sim:sim.process.memory.virtual
simics> pm0.add-probe mode=delta probe=sim:sim.process.memory.virtual
simics> pm0.add-probe mode=current probe=host:host.memory.used
simics> pm0.add-probe mode=delta probe=host:host.memory.used
simics> pm0.add-probe mode=current probe=sim:sim.image.memory_usage
simics> pm0.add-probe mode=delta probe=sim:sim.image.memory_usage
```

We are interested in total memory usage by the simulator process, the memory situation on the host, and the image memory usage. The mode "**current**" means we want to see the current usage and "**delta**" is the difference to the previous sample. Note that many probes can be added to the same monitor. In general, it is a good idea to track multiple probes to get a better understanding of what goes on.

Measuring the size of the simulator process is not entirely trivial. For that reason, we track both the resident and virtual memory size of the process. It is possible that the simulator allocates memory that it does not use, bumping the virtual size without increasing the resident size. The operating system can also swap out virtual memory pages used by the simulator, reducing the resident size compared to the virtual size. The virtual size should be seen as the maximum possible memory consumption of the simulator process.

4.4 Observe the Memory Usage

9. Run the simulation forward until the string "8192" appears on the console.

```
simics> bp.console_string.run-until object =
"ribit.unit[0].console.con" string = "8192"
```

You will see probe monitor prints showing how the memory consumption develops over time. The image memory usage dominates. It increases almost as much as total memory usage of the process.

The output looks something like this:

Current Process Mem-Res	Process Mem-Res	Current Process Mem-Virt	Process Mem-Virt	Current Host Mem	Host Mem	Current Image-Memory	Image-Memory
349.3 MiB	201.0 MiB	575.3 MiB	202.5 MiB	18.99 GiB	190.65 MiB	189.64 MiB	188.42 MiB
778.6 MiB	429.3 MiB	1009.4 MiB	434.2 MiB	19.40 GiB	424.04 MiB	595.19 MiB	405.55 MiB

…

The first two columns show the resident memory usage of the process. The next two the virtual usage, which in this case is some 230 MiB larger than the resident. The host is using 19 GiB already when the simulator starts to run, and this will go up as the simulator process increases in size. The last two columns show the size and change to the image system.

The change to the simulator process size is very similar to the change in the image memory system, indicating the size of the simulator process is going up due to the image system and not some misbehaving device.

Depending on your host, you may or may not encounter an image memory swap, meaning that the simulator writes out parts of the image system memory to disk. If not, memory usage of the process should be about 8GiB higher at the end, compared to the first sample. The target software has touched 8GiB of target memory in a way that forces the image system to allocate pages to hold the data.

As you can see, we had increasing image and non-image memory usage and we will inspect both in the next sections.

10. **Quit** the simulation.

```
simics> quit
```

5 Image Memory Inspection

The image system has its own dedicated inspection commands.

5.1 Inspecting Initial Image Memory Usage

1. Start a new simulation session with the **simics-internals-training/04-memory-usage** again.

```
[./]simics[.bat] simics-internals-training/04-memory-usage
```

2. Check the image usage right after the simulation has loaded the model.

```
simics> print-image-memory-stats
```

You'll see that two images have currently pages loaded into memory. This is the consequence of loading the initial code into the platform.

3. Add the **-all** option to see all images in the current setup.

```
simics> print-image-memory-stats -all
```

You'll see that there are two more images in the system, both currently in their initial (empty) state and hence without any pages allocated to memory.

4. The default is to show the memory usage as pages. To see it converted to human-readable numbers, use the **-h** option:

```
simics> print-image-memory-stats -h -all
```

5.2 Manually inspect image memory usage over time

5. Create a **breakpoint** on the magic instruction that is executed in the code after one iteration of the write loop. This will let you "step" the program forward.

```
simics> $bpid=(bp.magic.break number = 42 object =
"ribit.unit[0].hart")
```

6. **Run** the simulation, and wait for it to hit the breakpoint after the very first iteration:

```
simics> run
```

7. **Check** the image memory usage again.

```
simics> print-image-memory-stats
```

You see that the global RAM image (**ribit.global_ram.own_image**) has seen its first memory page allocated. This is due to the software writing pages, as discussed above.

8. We have executed one write so far. Now let's ignore the breakpoint for 30 times, so that we will hit the 32nd occurrence.

```
simics> bp.ignore-count $bpid 30
```

9. Check the breakpoint state:

```
simics> bp.list
```

10. **Run** until the next breakpoint hit.

```
simics> run
```

11. Check that the breakpoint consumed its ignore count, and indicates that it has been hit 32 times:

```
simics> bp.list
```

12. Once again, check the image memory usage.

```
simics> print-image-memory-stats
```

Since we just hit the breakpoint after the 32nd write, we expect that 4 pages are allocated (8 writes per page) and that is indeed the case.

13. Run forward again, which will hit the breakpoint after the next write.

```
simics> run
```

14. Check the image memory statistics one more time.

```
simics> print-image-memory-stats
```

You see that a fifth image page has now been allocated because the write loop just went past the fourth page.

5.3 Testing the Image Memory Limit

15. Inspect the current image memory limit.

```
simics> set-image-memory-limit
```

You will see what the current image memory limit is. This indicates the point at which the Intel Simics simulator will swap out allocated target memory pages to disk. For speed reasons, the simulator will always try to hold as much data in RAM as possible, but at some point, host RAM is exhausted, and swapping is inevitable.

The default image memory limit is calculated from the total amount of RAM in your host machine. This is likely above 1GiB, as modern machines tend to have more than that. You also see the location of the image system swap file. This should be kept on a local fast disk, to minimize the performance impact.

16. Set the image memory limit to 1GiB (argument is in MiB).

```
simics> set-image-memory-limit 1024
```

We lower the image memory radically in order to force the image system to swap as the target software is running.

17. Check the size of the memory used by the image system currently:

```
simics> print-image-memory-stats -h
```

Check the "**`total`**" line. It should be below the 1GiB limit. Note the final line, "**`Number of swapouts: 0`**", i.e., no image swapping has been done yet.

18. Delete the magic breakpoint.

```
simics> bp.delete $bpid
```

    We remove it so that it does not interfere with our next steps.

19. Create a probe monitor to monitor the image system memory usage and process size over time. Use virtual time sampling with a carefully calibrated interval that should make it sample approximately each time a star is printed on the serial console.

```
simics> enable-probes
simics> new-probe-monitor  clock="ribit.unit[0].hart" sampling-
mode=virtual interval=0.0065536
simics> pm0.add-probe mode=current probe=sim:sim.image.memory_usage
simics> pm0.add-probe mode=delta probe=sim:sim.image.memory_usage
simics> pm0.add-probe mode=current probe=sim:sim.process.memory.virtual
simics> pm0.add-probe mode=delta probe=sim:sim.process.memory.virtual
```

20. **Run** until we see 4096 in the serial console, which means after 4GiB of target memory has been touched. We expect that to run into swapping 4 times.

```
simics> bp.console_string.run-until object =
"ribit.unit[0].console.con" string = "4096"
```

    While running, you might notice that the simulator seems to "stop" at some times. Those pauses happen due to the swapping.

    You also see prints from the monitor, every time showing an image memory increase of about 32 MiB, which corresponds to the 32,786 1kiB writes that the application does per printed asterisk.

    You also see that the image memory drops by about 1 GiB at several points during the run. This happens whenever the image memory usage hits the memory limit, forcing a swap out. The simulator process virtual memory does not quite change by the same amount.

21. Inspect the image memory statistics again:

```
simics> print-image-memory-stats
```

    Note how it shows how many pages and bytes were swapped and how much time the swapping took.

22. Another way to see the same info as with the probes is with **system-perfmeter**. This is a good approach if you have performance issues, and you are not yet sure where they come from. Then **system-perfmeter** can monitor various performance related things together with the image memory usage.

```
simics> system-perfmeter -window -mem sample_time = 0.0065536
```

    Note that we use the same virtual time sample period as the probe monitor.

23. Now, run until 12GiB of target memory have been "used".

```
simics> bp.console_string.run-until object =
"ribit.unit[0].console.con" string = "12288"
```

Looking at the **system-perfmeter** output you see the "**Mem**" column (showing how much of the image memory limit has been used up) increasing. Every time it is about to hit 100% it drops down again because the image system swapped out pages.

24. This concludes the lab. Quit the simulation session.

```
simics> quit
```

6 Memory Tracking with malloc-debug

In the previous section we saw what looks like an ever-increasing memory usage. Even though we know that this results from the normal operation of the target software (it touches many target memory location forcing the simulator to allocate pages to store the memory content) let's use the malloc-debug feature of the Intel Simics simulator to get a better understanding of what is allocated during simulation.

NOTE: The malloc-debug feature intentionally excludes the image system, to avoids its likely huge allocations of memory hiding issues in other code .

6.1 Start the Simulator with Memory Tracking Enabled

1. To start the Intel Simics simulator with memory tracking you need to set an environment variable. It is best to do that in a temporary fashion such that the setting does not pollute your normal work environment.

   In your project, start the target **`simics-internals-training/04-memory-usage`** with memory tracking enabled.

   On Linux

   ```
   SIMICS_MEMORY_TRACKING=1 ./simics simics-internals-training/04-memory-usage
   ```

   On Windows

   ```
   cmd /C "set SIMICS_MEMORY_TRACKING=1&& simics.bat simics-internals-training/04-memory-usage"
   ```

   This will enable the tracking and data will be collected from the very start. No further need to enable the tracking.

2. Double-check that the variable was spotted by the simulator:

   ```
   simics> env substr = MEMORY
   ```

   This should show that the environment variable **`SIMICS_MEMORY_TRACKING`** is set **`1`**.

3. Run the simulation forward until the first line is complete (when the string "1024" is printed).

   ```
   simics> bp.console_string.run-until object = "ribit.unit[0].console.con" string = "1024"
   ```

   This should finish quickly.

6.2 Enable the malloc-debug Analysis Commands

4. Enable the malloc-debug commands.

   ```
   simics> enable-unsupported-feature malloc-debug
   ```

   The memory tracking was already active. This simply enables the commands used to access the information. Being unsupported means that the feature is maintained at a best-effort basis and subject to change. Thus, it is not available per default.

5. Change the default output radix to decimal for easier readability.

```
simics> output-radix 10 3
```

6.3 Inspect Memory Allocations

6. Now check the current list of allocation sites.

```
simics> mm-list-sites
```

This shows you allocation sites in the code that currently have the most bytes allocated.

You should see an allocation site of type **image_spage_t** amongst the top ten. These are internal control structures for allocated image memory pages, but they are NOT the actual target memory. You should notice that they add up to roughly 8MiB while we have more than 1GiB of target memory allocated right now. You can also see that the allocation count is above 131,072 (which we need for the 1GiB of target memory that was accessed). The additional pages are image memory pages that hold the software code and variables, etc.

The other top allocations are related to the processor model.

7. Another view of the allocated memory is to sort by allocation count:

```
simics> mm-list-allocations
```

8. They can also be listed by type:

```
simics> mm-list-types
```

9. The output can be made more readable by using -human-readable to convert sizes to MiB etc. It can also be limited to the top items. For example:

```
simics> mm-list-sites -human-readable max = 10
```

10. The number of **image_spage_t** allocations should correlate with the number of pages reported by the print-image-memory-stats command:

```
simics> print-image-memory-stats
```

11. The target code execution has right now written until the end of an 8kiB target memory page (since we just finished a line) and the next write will have to allocate a new page to hold the content of the simulated target memory.

Place a breakpoint on the magic instruction number 42, which happens just after we write the next data to memory.

```
simics> $bpid=(bp.magic.break number = 42 object =
"ribit.unit[0].hart")
```

12. **Run** the simulation forward to hit the breakpoint.

```
simics> run
```

13. Now list the sites again.

```
simics> mm-list-sites -human-readable max = 10
```

The counter for **image_spage_t** allocations went up by one, because the image write that just happened allocated a new image system memory page and hence also a control structure. If you look carefully, you can also see that the CPU page control and some other control structures have increased by one.

So apparently, we just saw how a write to simulated memory forced the simulator to allocate a new page of host memory to store simulated memory content.

14. Now, let's observe how these pages are used. To this end, set the ignore count of the magic breakpoint.

```
simics> bp.ignore-count num = 6 id = $bpid
```

Since we just observed the first write to a page and we know that 8 writes fit into a page, we ignore to stop at the 2nd to 7th write and will stop after the 8th again.

15. Double-check the breakpoint information.

```
simics> bp.show $bpid
```

You see it was hit once so far.

16. **Run** the simulation until the breakpoint hits:

```
simics> run
```

17. Check the breakpoint information again.

```
simics> bp.show $bpid
```

You see we hit the breakpoint 8 times now, so we saw 8 write since the last page was allocated, so if pages are properly used, the allocation count of the control structure **image_spage_t** should be still the same compared to what you saw the last time you checked the allocation sites.

18. Check the memory allocations again.

```
simics> mm-list-sites -human-readable max = 10
```

You see that the count is the same as before.

19. **Run** the simulation until the breakpoint hits again:

```
simics> run
```

20. Check the memory allocations again.

```
simics> mm-list-sites -human-readable max = 10
```

As expected, another page data structure has been allocated. Knowing that the write stride is 1kiB this confirms that the page size is 8kiB.

In this lab, you learned how malloc-debug helps you track the memory usage. The general idea is to regularly check the allocations and see if there is a site or type whose allocations just keep increasing over time. If the allocations are the result of how the target software operates, this might indicate a memory leak in the model code and you know where to start looking.

21. Quit the simulation.

```
simics> quit
```