Programming Event-Driven Simulation

IMPORTANT: This slide deck builds on top of what has already been developed in Introduction to SimObjects and Debugging gem5.

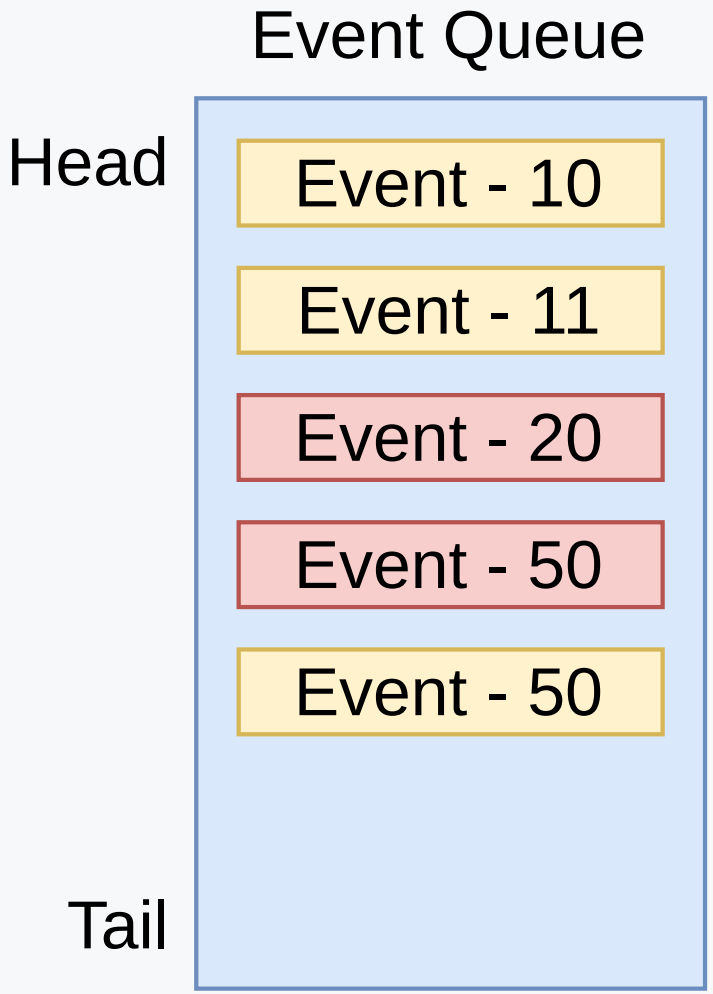
Refresher On Event Driven Simulation

gem5 architecture: Simulation

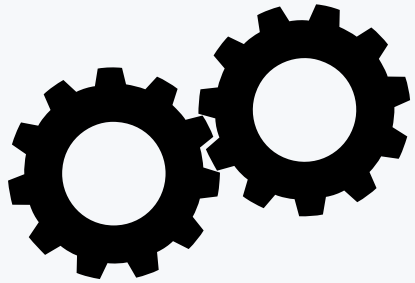
gem5 is a ***discrete event simulator***

At each timestep, gem5:

1. Event at the head is dequeued
2. The event is executed
3. New events are scheduled



Time: 10

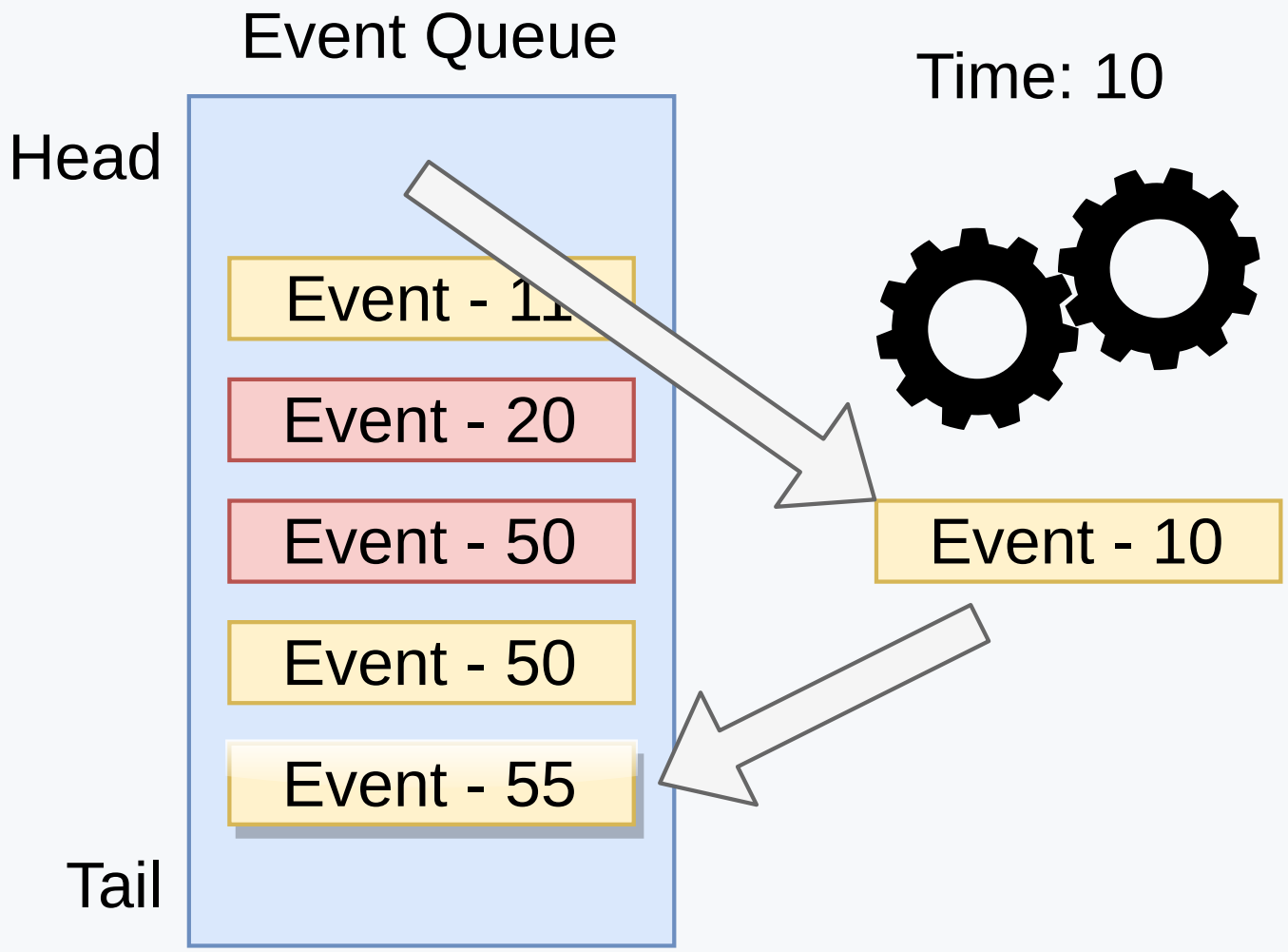


gem5 architecture: Simulation

gem5 is a ***discrete event simulator***

At each timestep, gem5:

1. Event at the head is dequeued
2. The event is executed
3. New events are scheduled



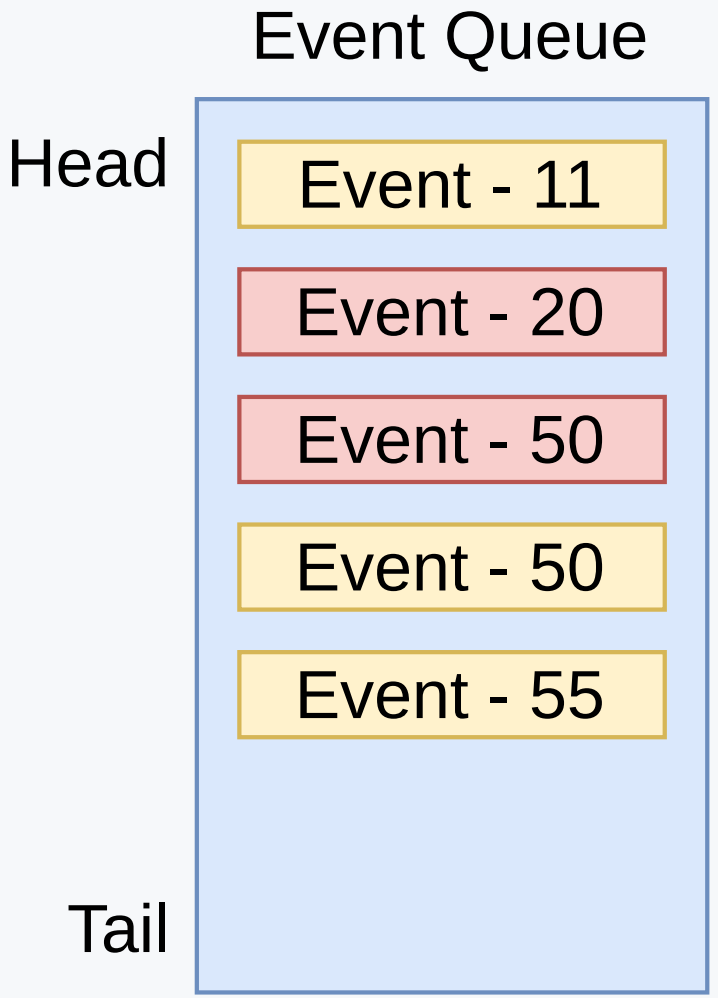
gem5 architecture: Simulation

gem5 is a ***discrete event simulator***

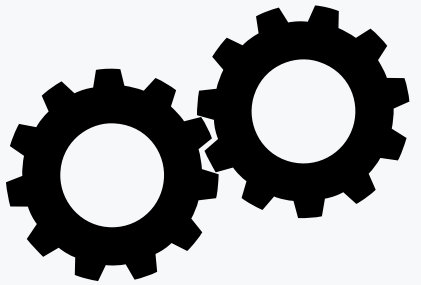
At each timestep, gem5:

1. Event at the head is dequeued
2. The event is executed
3. New events are scheduled

All SimObjects can enqueue events onto the event queue



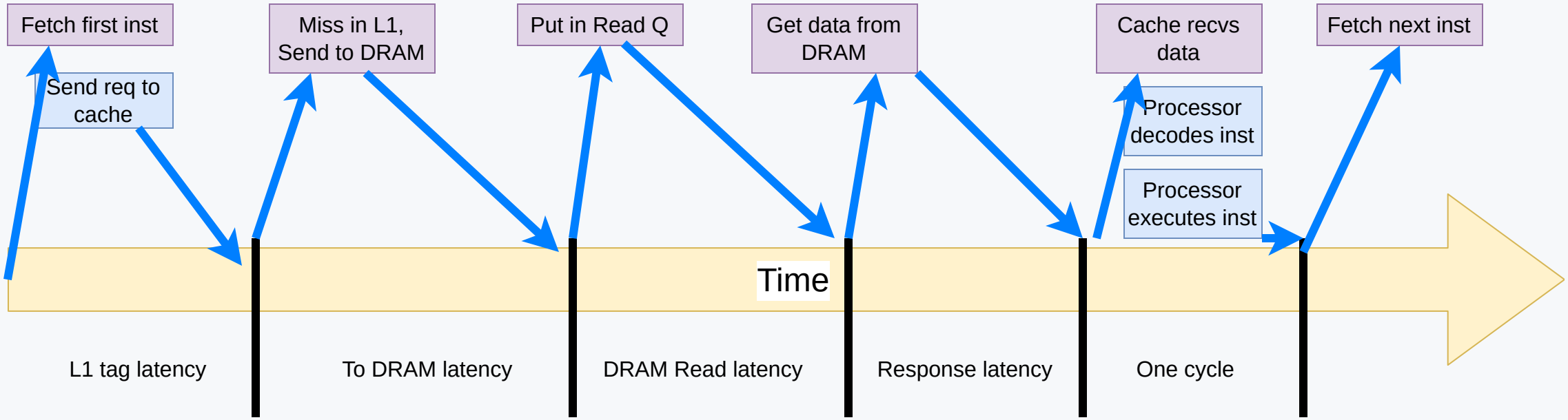
Time: 11



Discrete event simulation example

Discrete event simulation example

Discrete event simulation example



To model things that take time, schedule the *next* event in the future (latency of current event).
Can call functions instead of scheduling events, but they occur *in the same tick*.

Discrete event simulation

"Time" needs a unit.
In gem5, we use a unit called "Tick".

Need to convert a simulation "tick" to user-understandable time, e.g. seconds.

This is the global simulation tick rate.
Usually this is 1 ps per tick or 10^{12} ticks per second.

Event-Driven Simulation: Abstract Thoughts

`Event-Driven Simulation` is a method for simulation where the simulator reacts to the occurrence of `events`. Each type of `event` will have its own specific reaction.

The reaction to an `event` is defined by a call to a specific function that is referred to as the `callback` function.

The `callback` function might itself cause new `events` to occur. The new `events` can be of the same type or a different type as the `event` that caused the call to the `callback` function.

Event-Driven Simulation: Abstract Thoughts (cont.)

Let's look at an example to understand it better. Let's say that at time t_0 event A occurs. The simulator will react by calling $A.callback$. Let's say below is the definition for $A.callback$.

```
# This is python, but gem5's events are in C++
class A(Event):
    def callback():
        print("Reacting to Event A")
        delay = 1000
        curr_time = Simulator.get_current_time()
        schedule(B, current_time + delay)
```

This way every time event A occurs, event B will occur 1000 units of time later. Then, the simulator will react by calling $B.callback$.

Event-Driven Simulation: Practical View

An event-driven simulator needs to facilitate the following:

- Notion of time: The simulator needs to track the global time of the simulation and allow access to the current time. It also needs to move the time forward.
- Interface to `events`: The simulator needs to define the base interface for `events` in the simulator so that they can define and raise (i.e. make occur/schedule) new `events`.
  - The base interface of `event` should allow for `events` to be tied to `callback` functions.

Event-Driven Simulation: Practical View (cont.)

Let's see how this will look if you were to write your own hardware simulator.

1. In the beginning ($t = 0$), the simulator will schedule an event that makes the CPU cores fetch an instruction. Let's call that type of event `CPU::fetch`.
2. When simulator reaches ($t = 0$), the simulator will react to all the `events` that are scheduled at that time. If we have 2 cores, this means that simulator needs to call `cpu_0::fetch::callback` and `cpu_1::fetch::callback`. This happens in the same *logical* tick, but could happen in series in the simulator code.
3. `CPU::fetch::callback` will have to then find out what the next program counter is and send a request to the instruction cache to fetch the instruction. Therefore, it will schedule an event like `CPU::accessICache` in the future.

To impose the latency of the fetch we will schedule `CPU::accessICache` at `current_time + fetch_delay`, i.e. `schedule(CPU::accessICache, currentTime() + fetch_delay)`. This will raise two `CPU::accessICache` events (e.g. `cpu_0::accessICache` and `cpu_1::accessICache`) after `fetch_delay` units of time in the future.

Event-Driven Simulation: Practical View (cont.)

4. When the simulator has finished reacting to all events that occurred at $t = 0$, it will move time to the closest time that an event is scheduled to occur ($t = 0 + fetch_delay$ in this case).
5. At time $t = fetch_delay$ the simulator will call `cpu_0::accessICache::callback` and `cpu_1::accessICache::callback` to react to both events. These events will probably access the instruction caches and then might schedule events to handle misses in the cache like `Cache::handleMiss`.
6. This process will continue until the program we're simulating is finished.

Event-Driven Simulation in gem5

Let's look at src/sim/eventq.hh. In there you will see a declaration for class `Event` that has a function called `process` like below.

```
public:

  /*
   * This member function is invoked when the event is processed
   * (occurs).  There is no default implementation; each subclass
   * must provide its own implementation.  The event is not
   * automatically deleted after it is processed (to allow for
   * statically allocated event objects).
   *
   * If the AutoDestroy flag is set, the object is deleted once it
   * is processed.
   *
   * @ingroup api_eventq
   */
  virtual void process() = 0;
```

A Hypothetical Example for Event

Let's now see how class `Event` would be used in a `SimObject` that models a CPU. **CAUTION**: This is a hypothetical example and is not at all what is already implemented in gem5.

In this example, every time an instance of `FetchEvent` occurs (`cpu_0::nextFetch` and not `CPU::nextFetch`), the simulator will call `processFetch` from the `CPU` instance that owns the event.

```
class CPU: public ClockedObject
{
  public:
    void processFetch(); // Function to model fetch
  private:
    class FetchEvent: public Event
    {
      private:
        CPU* owner;
      public:
        FetchEvent(CPU* owner): Event(), owner(owner)
        {}
        virtual void process() override
        {
            // call processFetch from the CPU that owns this
            owner->processFetch();
        }
    };
    FetchEvent nextFetch;
};
```

EventFunctionWrapper

In addition to class `Event`, you can find the declaration for `EventFunctionWrapper` in src/sim/eventq.hh. This class wraps an `event` with a callable object that will be called when `Event::process` is called. The following lines from `src/sim/eventq.hh` are useful to look over.

For `EventFunctionWrapper` the function `process` is defined as a call to `callback` which is passed as an argument to the constructor of `EventFunctionWrapper`. Additionally, we will need to give each object a name through the constructor.

```
public:
    /**
     * This function wraps a function into an event, to be
     * executed later.
     * @ingroup api_eventq
     */
    EventFunctionWrapper(const std::function<void(void)> &callback,
                         const std::string &name,
                         bool del = false,
                         Priority p = Default_Pri)
        : Event(p), callback(callback), _name(name)
    {
        if (del)
            setFlags(AutoDelete);
    }
    void process() { callback(); }
```

Detour: m5.simulate: SimObject::startup

Below is a snippet of code from the definition of `m5.simulate` in src/python/m5/simulate.py:

By calling `m5.simulate`, gem5 will call the function `startup` from every `SimObject` in the system. Let's take a look at `startup` in header file for `SimObject` in src/sim/sim_object.hh.

`startup` is where we schedule the initial `events` that trigger a simulation (`CPU::nextFetch` in our hypothetical scenario).

```python
def simulate(*args, **kwargs):
    # ...
    if need_startup:
        root = objects.Root.getInstance()
        for obj in root.descendants():
            obj.startup()
        need_startup = False
```

```
/**
 * startup() is the final initialization call before simulation.
 * All state is initialized (including unserialized state, if any,
 * such as the curTick() value), so this is the appropriate place to
 * schedule initial event(s) for objects that need them.
*/
virtual void startup();
```

To complete the SimObject initialization

0. **Constructor**
   This is where you copy params and set up the object. This is where you should set up any initial state that is not dependent on other objects.
1. `init()`
   This is where you should set up any initial state that is dependent on other objects.
   At this point, you know all SimObjects have been *constructed*.
2. `regStats()`
   This is where you should register any statistics that you want to track.
3. `initState()` if starting afresh **OR** `loadState()` if restoring from a checkpoint.
   This is where you set up any state (e.g., registers, memory, etc.) which may be reset if running initially or have been saved in a checkpoint.
4. `resetStats()`
   Called automatically.
5. `startup()`
   This is where you should schedule any initial events that are needed to start the simulation.

Exercise: SimObject Events

In this exercise, we will add an `event` to our `HelloSimObject` to print `Hello ...` periodically for a certain number of times. We will also add a `GoodByeSimObject` that will print `GoodBye ...` after `Hello ...` is done.

Exercise 1: nextHelloEvent

nextHelloEvent

The completed files for exercise 1 are under the directory materials/03-Developing-gem5-models/03-event-driven-sim/step-1.

Now, let's add an `event` to our `HelloSimObject` to print `Hello ...` periodically for a certain number of times (i.e. `num_hellos`). Let's add it to the header file for `HelloSimObject` in src/bootcamp/hello-sim-object.hh.

First, we need to include `sim/eventq.hh` so we can add a member of type `EventFunctionWrapper`. Add the following line to do this. **REMEMBER**: Make sure to follow the right order of includes.

```
#include "sim/eventq.hh
```

Creating nextHelloEvent

Next, we need to declare a member of type `EventFunctionWrapper` which we will call `nextHelloEvent`.

We also need to define a `std::function<void>()` as the `callback` function for `nextHelloEvent`.

- `std::function<void>()` is a callable with return type `void` and no input arguments.

To do this, add the following lines to your declaration of the `HelloSimObject` class.

```
  private:
    EventFunctionWrapper nextHelloEvent;
    void processNextHelloEvent();
```

nextHelloEvent: Header File

This is how your `hello_sim_object.hh` should look after all the changes.

```
#ifndef __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__
#define __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__

#include "params/HelloSimObject.hh"
#include "sim/eventq.hh"
#include "sim/sim_object.hh"

namespace gem5
{

class HelloSimObject: public SimObject
{
  private:
    EventFunctionWrapper nextHelloEvent;
    void processNextHelloEvent();

  public:
    HelloSimObject(const HelloSimObjectParams& params);
};

} // namespace gem5

#endif // __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__
```

nextHelloEvent: HelloSimObject: Constructor

Now, let's change our definition of the constructor of `HelloSimObject` to initialize `nextHelloEvent`. Let's add the following line to the initialization list in `HelloSimObject::HelloSimObject` which you can find in `src/bootcamp/hello-sim-object/hello_sim_object.cc`.

```
    nextHelloEvent([this](){ processNextHelloEvent(); }, name() + ".nextHelloEvent")
```

Here, we're initializing `nextHelloEvent` with a `callback` function that calls `processNextHelloEvent` and a name for the event.

The syntax shows that we're "capturing" `this` in the lambda function. This is necessary because the lambda function needs to access the member function `processNextHelloEvent`.

We are calling this event `name()` (which is the python variable name used for the instance of this SimObject) `+ ".nextHelloEvent"`. This is a common pattern to name events.

nextHelloEvent: Cpp file

```
HelloSimObject::HelloSimObject(const HelloSimObjectParams& params):
    SimObject(params),
    nextHelloEvent([this](){ processNextHelloEvent(); }, name() + ".nextHelloEvent")
{
    for (int i = 0; i < params.num_hellos; i++) {
        std::cout << "i: " << i << ", Hello from HelloSimObject's constructor!" << std::endl;
    }
    DPRINTF(HelloExampleFlag, "%s: Hello from HelloSimObject's constructor!\n", __func__);
}
```

nextHelloEvent Callback: processNextHelloEvent

Now, let's define `processNextHelloEvent` to print `Hello ...` `num_hellos` times every `500 Ticks`. To track the number of `Hello ...` statements we have printed, let's declare a `private` member to count them. Add the following declaration to the `private` scope of class `HelloSimObject` in `src/bootcamp/hello-sim-object/hello_sim_object.hh`.

```
  private:
    int remainingHellosToPrintByEvent;
```

This is how the declaration for `HelloSimObject` should look after the changes.

```
class HelloSimObject: public SimObject
{
  private:
    int remainingHellosToPrintByEvent;

    EventFunctionWrapper nextHelloEvent;
    void processNextHelloEvent();

  public:
    HelloSimObject(const HelloSimObjectParams& params);
};
```

nextHelloEvent Callback: processNextHelloEvent cont.

Now, let's update the constructor of `HelloSimObject` to initialize `remainingHellosToPrintByEvent` to `params.num_hellos`. Do this by adding the following line above the initialization line for `nextHelloEvent`.

```
    remainingHellosToPrintByEvent(params.num_hellos)
```

Let's also make sure user passes a positive number for `num_hellos` by adding a `fatal_if` statement like below to the beginning of the body of `HelloSimObject::HelloSimObject`.

```
    fatal_if(params.num_hellos <= 0, "num_hellos should be positive!");
```

nextHelloEvent Callback: processNextHelloEvent: Almost There

This is how `HelloSimObject::HelloSimObject` should look after the changes.

```
HelloSimObject::HelloSimObject(const HelloSimObjectParams& params):
    SimObject(params),
    remainingHellosToPrintByEvent(params.num_hellos),
    nextHelloEvent([this](){ processNextHelloEvent(); }, name() + ".nextHelloEvent")
{
    fatal_if(params.num_hellos <= 0, "num_hellos should be positive!");
    for (int i = 0; i < params.num_hellos; i++) {
        std::cout << "i: " << i << ", Hello from HelloSimObject's constructor!" << std::endl;
    }
    DPRINTF(HelloExampleFlag, "%s: Hello from HelloSimObject's constructor!\n", __func__);
}
```

nextHelloEvent Callback: processNextHelloEvent: Finally!

Now we are ready to define `HelloSimObject::processNextHelloEvent`. Let's add the following code to `hello_sim_object.cc`.

```
void
HelloSimObject::processNextHelloEvent()
{
    std::cout << "tick: " << curTick() << ", Hello from ";
    std::cout << "HelloSimObject::processNextHelloEvent!";
    std::cout << std::endl;
    remainingHellosToPrintByEvent--;
    if (remainingHellosToPrintByEvent > 0) {
        schedule(nextHelloEvent, curTick() + 500);
    }
}
```

Looking at the code, we do the following every time `nextHelloEvent` occurs (i.e. `processNextHelloEvent` is called):

- Print `Hello ...`.
- Decrement `remainingHellosToPrintByEvent`.
- Check if we have remaining prints to do. If so, we will schedule `nextHelloEvent` 500 ticks into the future. **NOTE**: `curTick` is a function that returns the current simulator time in `Ticks`.

HelloSimObject::startup: Header File

Let's add a declaration for `startup` in `HelloSimObject`. We will use `startup` to schedule the first occurrence of `nextHelloEvent`. Since `startup` is a `public` and `virtual` function that `HelloSimObject` inherits from `SimObject`, we will add the following line to the `public` scope of `HelloSimObject`. We will add the `override` directive to tell the compiler that we intend to override the original definition from `SimObject`.

```
public:
  virtual void startup() override;
```

HelloSimObject::startup: Source File

Now, let's define `HelloSimObject::startup` to schedule `nextHelloEvent`. Since `startup` is called in the beginning of simulation (i.e. $t = 0\ Ticks$) and is **only called once**, let's put `panic_if` statements to assert them. Moreover, `nextHelloEvent` should not be scheduled at the time so let's assert that too.

Add the following code to `src/bootcamp/hello-sim-object/hello_sim_object.cc` to define `HelloSimObject::startup`.

```
void
HelloSimObject::startup()
{
    panic_if(curTick() != 0, "startup called at a tick other than 0");
    panic_if(nextHelloEvent.scheduled(), "nextHelloEvent is scheduled before HelloSimObject::startup is called!");
    schedule(nextHelloEvent, curTick() + 500);
}
```

Exercise: Answers (SConscript)

We are ready to compile gem5 to apply the changes. But before we compile, let's go over how every file should look.

- src/bootcamp/hello-sim-object/SConscript:

```
Import("*")

SimObject("HelloSimObject.py", sim_objects=["HelloSimObject"])

Source("hello_sim_object.cc")

DebugFlag("HelloExampleFlag")
```

Exercise: Answers (HelloSimObject.py)

- src/bootcamp/hello-sim-object/HelloSimObject.py:

```
from m5.objects.SimObject import SimObject
from m5.params import *

class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"

    num_hellos = Param.Int("Number of times to say Hello.")
```

Exercise: Answers (hello_sim_object.hh)

- This is how src/bootcamp/hello-sim-object/hello_sim_object.hh should look.

```
#ifndef __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__
#define __BOOTCAMP_HELLO_SIM_OBJECT_HELLO_SIM_OBJECT_HH__

#include "params/HelloSimObject.hh"
#include "sim/eventq.hh"
#include "sim/sim_object.hh"

namespace gem5
{

class HelloSimObject: public SimObject
{
  private:
    int remainingHellosToPrintByEvent;

    EventFunctionWrapper nextHelloEvent;
    void processNextHelloEvent();

  public:
    HelloSimObject(const HelloSimObjectParams& params);
    virtual void startup() override;
};

} // namespace gem5
```

Exercise: Answers (hello_sim_object.cc)

- This is how src/bootcamp/hello-sim-object/hello_sim_object.cc should look.

```cpp
#include "bootcamp/hello-sim-object/hello_sim_object.hh"

#include <iostream>

namespace gem5
{

HelloSimObject::HelloSimObject(const HelloSimObjectParams& params):
    SimObject(params),
    remainingHellosToPrintByEvent(params.num_hellos),
    nextHelloEvent(
        [this](){ processNextHelloEvent(); }, name() + ".nextHelloEvent"
    )
{
    fatal_if(params.num_hellos <= 0, "num_hellos should be positive!");
    for (int i = 0; i < params.num_hellos; i++) {
        std::cout << "i: " << i << ", Hello from HelloSimObject's "
        std::cout << "constructor!" << std::endl;
    }
    DPRINTF(HelloExampleFlag, "%s: Hello from HelloSimObject's constructor!\n",
                              __func__);
}
```

```cpp
void
HelloSimObject::startup()
{
    panic_if(curTick() != 0,
        "startup called at a tick other than 0");
    panic_if(nextHelloEvent.scheduled(),
        "nextHelloEvent is scheduled before "
        "HelloSimObject::startup is called!");
    schedule(nextHelloEvent, curTick() + 500);
}

void
HelloSimObject::processNextHelloEvent()
{
    std::cout << "tick: " << curTick() << ", Hello from ";
    std::cout << "HelloSimObject::processNextHelloEvent!" << std::endl;
    remainingHellosToPrintByEvent--;
    if (remainingHellosToPrintByEvent > 0) {
        schedule(nextHelloEvent, curTick() + 500);
    }
}

} // namespace gem5
```

Let's Compile and Simulate

Run the following command in the base gem5 directory to rebuild gem5.

```
scons build/NULL/gem5.opt -j$(nproc)
```

Now, simulate your configuration by running the same script as before.

```
./build/NULL/gem5.opt configs/bootcamp/hello-sim-object/second-hello-example.py
```

In the next slide, there is recording of what you should expect to see.

<script src="https://asciinema.org/a/UiLAZT0Ryi75nkLQSs0AC0OWI.js" id="asciicast-UiLAZT0Ryi75nkLQSs0AC0OWI" async="true"></script>

Investigating the event queue

You can also investigate the event queue by using the `Event` debug flag.

```
./build/NULL/gem5.opt --debug-flags=Event configs/bootcamp/hello-sim-object/second-hello-example.py
```

Step 2: SimObjects as Parameters

In this exercise, we will learn about adding a `SimObject` as a parameter. We will add a new SimObject that the `HelloSimObject` can communicate with, `GoodByeSimObject`, that will print `GoodBye ...` after `Hello ...` is done.

Exercise 2: GoodByeSimObject

In this step, we will learn about adding a `SimObject` as a parameter. To do this, let's first build our second `SimObject` called `GoodByeSimObject`. As you remember, we need to declare `GoodByeSimObject` in Python. Let's open `src/bootcamp/hello-sim-object/HelloSimObject.py` and add the following code to it.

```
class GoodByeSimObject(SimObject):
    type = "GoodByeSimObject"
    cxx_header = "bootcamp/hello-sim-object/goodbye_sim_object.hh"
    cxx_class = "gem5::GoodByeSimObject"
```

Also, let's register `GoodByeSimObject` by editing `SConscript`. Open `src/bootcamp/hello-sim-object/SConscript` and add `GoodByeSimObject` to the list of `SimObjects` in `HelloSimObject.py`. This is how the line show look after the changes.

```
SimObject("HelloSimObject.py", sim_objects=["HelloSimObject", "GoodByeSimObject"])
```

GoodByeExampleFlag

Let's add `goodbye_sim_object.cc` (which we will create later) as a source file. Do it by adding the following line to the `src/bootcamp/hello-sim-object/SConscript`.

```
Source("goodbye_sim_object.cc")
```

Let's also add `GoodByeExampleFlag` so that we can use to print debug in `GoodByeSimObject`. Do it by adding the following line to `src/bootcamp/hello-sim-object/SConscript`.

```
DebugFlag("GoodByeExampleFlag")
```

GoodByeExampleFlag (cont.)

CompoundFlag

In addition to `DebugFlags`, we can define `CompoundFlags` that enable a set of `DebugFlags` when they are enabled. Let's define a `CompoundFlag` called `GreetFlag` that will enable `HelloExampleFlag`, `GoodByeExampleFlag`. To do it, add the following line to `src/bootcamp/hello-sim-object/SConscript`.

```
CompoundFlag("GreetFlag", ["HelloExampleFlag", "GoodByeExampleFlag"])
```

Current Version: HelloSimObject.py

This is how HelloSimObject.py should look after the changes.

```
from m5.objects.SimObject import SimObject
from m5.params import *

class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"

    num_hellos = Param.Int("Number of times to say Hello.")

class GoodByeSimObject(SimObject):
    type = "GoodByeSimObject"
    cxx_header = "bootcamp/hello-sim-object/goodbye_sim_object.hh"
    cxx_class = "gem5::GoodByeSimObject"
```

Current Version: SConscript

This is how SConscript should look after the changes.

```
Import("*")

SimObject("HelloSimObject.py", sim_objects=["HelloSimObject", "GoodByeSimObject"])

Source("hello_sim_object.cc")
Source("goodbye_sim_object.cc")

DebugFlag("HelloExampleFlag")
DebugFlag("GoodByeExampleFlag")
CompoundFlag("GreetFlag", ["HelloExampleFlag", "GoodByeExampleFlag"])
```

Notice the change in the `SimObject` line. This says that the file `HelloSimObject.py` contains two SimObjects, `HelloSimObject` and `GoodByeSimObject`.

GoodByeSimObject: Specification

In our design, let's have `GoodByeSimObject` debug print a `GoodBye ...` statement. It will do it when the `sayGoodBye` function is called, which will schedule an `event` to say GoodBye.

In the next slides you can find the completed version for src/bootcamp/hello-sim-object/goodbye_sim_object.hh and src/bootcamp/hello-sim-object/goodbye_sim_object.cc.

IMPORTANT: I'm not going to go over the details of the files, look through this file thoroughly and make sure you understand what every line is supposed to do.

On your own

The goal with the `GoodByeSimObject` is to have it print `GoodBye ...` after `Hello ...` is done.

Specifically, it should have a `sayGoodBye` function that schedules an *event* to print `"GoodBye ..."` after 500 ticks.
The event should be named `nextGoodByeEvent` and you should create a function `processNextGoodByeEvent` which the event calls.

Looking forward

Eventually (in a couple of steps), the `HelloSimObject` will call `sayGoodBye` from `goodByeObject` when it is done printing `Hello ...`.

GoodByeSimObject: Header File

```
#ifndef __BOOTCAMP_HELLO_SIM_OBJECT_GOODBYE_SIM_OBJECT_HH__
#define __BOOTCAMP_HELLO_SIM_OBJECT_GOODBYE_SIM_OBJECT_HH__

#include "params/GoodByeSimObject.hh"
#include "sim/eventq.hh"
#include "sim/sim_object.hh"

namespace gem5
{

class GoodByeSimObject: public SimObject
{
  private:
    EventFunctionWrapper nextGoodByeEvent;
    void processNextGoodByeEvent();

  public:
    GoodByeSimObject(const GoodByeSimObjectParams& params);

    void sayGoodBye();
};

} // namespace gem5

#endif // __BOOTCAMP_HELLO_SIM_OBJECT_GOODBYE_SIM_OBJECT_HH__
```

GoodByeSimObject: Source File

```
#include "bootcamp/hello-sim-object/goodbye_sim_object.hh"

#include "base/trace.hh"
#include "debug/GoodByeExampleFlag.hh"

namespace gem5
{

GoodByeSimObject::GoodByeSimObject(const GoodByeSimObjectParams& params):
    SimObject(params),
    nextGoodByeEvent([this]() { processNextGoodByeEvent(); }, name() + ".nextGoodByeEvent" )
{}

void
GoodByeSimObject::sayGoodBye() {
    panic_if(nextGoodByeEvent.scheduled(), "GoodByeSimObject::sayGoodBye called while nextGoodByeEvent is scheduled!");
    schedule(nextGoodByeEvent, curTick() + 500);
}

void
GoodByeSimObject::processNextGoodByeEvent()
{
    DPRINTF(GoodByeExampleFlag, "%s: GoodBye from GoodByeSimObejct::processNextGoodByeEvent!\n", __func__);
}

} // namespace gem5
```

GoodByeSimObject as a Param

In this step we will add a parameter to `HelloSimObject` that is of type `GoodByeSimObject`. To do this we will simply add the following line to the declaration of `HelloSimObject` in `src/bootcamp/hello-sim-object/HelloSimObject.py`.

```
    goodbye_object = Param.GoodByeSimObject("GoodByeSimObject to say goodbye after done saying hello.")
```

This is how the declaration of `HelloSimObject` should look after the changes.

```
class HelloSimObject(SimObject):
    type = "HelloSimObject"
    cxx_header = "bootcamp/hello-sim-object/hello_sim_object.hh"
    cxx_class = "gem5::HelloSimObject"

    num_hellos = Param.Int("Number of times to say Hello.")

    goodbye_object = Param.GoodByeSimObject("GoodByeSimObject to say goodbye after done saying hello.")
```

HelloSimObject: Header File

Adding the `goodbye_object` parameter will add a new member to `HelloSimObjectParams` of `gem5::GoodByeSimObject*` type. We will see this in the future.

We can use that parameter to initialize a pointer to an object of `GoodByeSimObject` which we will use to call `sayGoodBye` when we run out of `Hello ...` statements to print.

First, let's include the header file for `GoodByeSimObject` in `src/bootcamp/hello-sim-object/hello_sim_object.hh` by adding the following line. **REMEMBER**: Follow gem5's convention for including order (alphabetical).

HelloSimObject: Updated header file

```
#include "bootcamp/hello-sim-object/goodbye_sim_object.hh"
```

Now, let's add a new member to `HelloSimObject` that is a pointer to `GoodByeSimObject`. Add the following line to `src/bootcamp/hello-sim-object/hello_sim_object.hh`.

```
  private:
    GoodByeSimObject* goodByeObject;
```

When passing SimObject's as parameters, in the SimObject C++ code you get a pointer to the object.
This is useful for when you have tightly coupled objects for which you want one object to directly be able to call functions on the other.

For more loosely-coupled objects (e.g., cores connected to caches), you should use other gem5 APIs for communication. These APIs include `Ports` and a few other APIs.

HelloSimObject: Source File constructor

Now let's initialize `goodByeObject` from the parameters by adding the following line to the initialization list in `HelloSimObject::HelloSimObject`.

```
    goodByeObject(params.goodbye_object)
```

This constructor initialization is the same as initializing any other parameter.
The `params` type (`HelloSimObjectParams`) will have the right type for the `goodbye_object` member.

HelloSimObject: Source File changes

Now, let's add an `else` body to `if (remainingHellosToPrintByEvent > 0)` in `processNextHelloEvent` to call `sayGoodBye` from `goodByeObject`. Below is how `processNextHelloEvent` in `src/bootcamp/hello-sim-object/hello_sim_object.cc` should look after the changes.

```
void
HelloSimObject::processNextHelloEvent()
{
    std::cout << "tick: " << curTick() << ", Hello from HelloSimObject::processNextHelloEvent!" << std::endl;
    remainingHellosToPrintByEvent--;
    if (remainingHellosToPrintByEvent > 0) {
        schedule(nextHelloEvent, curTick() + 500);
    } else {
        goodByeObject->sayGoodBye();
    }
}
```

Let's Build

```
scons build/NULL/gem5.opt -j$(nproc)
```

After the compilation is done, look at `build/NULL/params/HelloSimObject.hh`. Notice that `gem5::GoodByeSimObject * goodbye_object` is added. Below is the declaration for `HelloSimObjectParams`.

```
namespace gem5
{
struct HelloSimObjectParams
    : public SimObjectParams
{
    gem5::HelloSimObject * create() const;
    gem5::GoodByeSimObject * goodbye_object;
    int num_hellos;
};

} // namespace gem5
```

Configuration Script

Let's create a new configuration script (`third-hello-example.py`) by copying `configs/bootcamp/hello-sim-object/second-hello-example.py`. Do it by running the following command in the base gem5 directory.

Now, we need to give a value to `goodbye_object` parameter from `HelloSimObject`. We will create an object of `GoodByeSimObject` for this parameter.

Let's start by importing `GoodByeSimObject`. Do it by simply adding `GoodByeSimObject` to `from m5.objects.HelloSimObject import HelloSimObject`. This is how the import statement should look after the changes.

```
from m5.objects.HelloSimObject import HelloSimObject, GoodByeSimObject
```

Configuration Script (cont.)

Now, let's add the following line to give a value to `goodbye_object` from `root.hello`.

```
root.hello.goodbye_object = GoodByeSimObject()
```

This line instantiates a new `GoodByeSimObject` and assigns it to the `goodbye_object` parameter of the `HelloSimObject` instance `root.hello`.

The `GoodByeSimObject` will be a *child* of the `HelloSimObject` instance `root.hello`.

If you use the `--debug-flags=Event` flag, you can see the event queue and the order of events being scheduled. You'll also see that the `name()` of the `GoodByeSimObject` is `hello.goodbye_object`.

Let's Simulate

Now let's simulate `third-hello-example.py` once with `GoodByeExampleFlag` and once with `GreetFlag` enabled and compare the outputs.

Run the following command in the base gem5 directory to simulate `third-hello-example.py` with `GoodByeExampleFlag` enabled.

```
./build/NULL/gem5.opt --debug-flags=GoodByeExampleFlag configs/bootcamp/hello-sim-object/third-hello-example.py
```

In the next slide, there is a recording of my terminal when I run the command above.

<script src="https://asciinema.org/a/9vTP6wE1Yu0ihlKjA4j7TxEMm.js" id="asciicast-9vTP6wE1Yu0ihlKjA4j7TxEMm" async="true"></script>

Let's Simulate: Part 2

Run the following command in the base gem5 directory to simulate `third-hello-example.py` with `GreetFlag` enabled.

```
./build/NULL/gem5.opt --debug-flags=GreetFlag configs/bootcamp/hello-sim-object/third-hello-example.py
```

In the next slide, there is a recording of my terminal when I run the command above.

<script src="https://asciinema.org/a/2cz336gLt2ZZBysroLhVbqBHs.js" id="asciicast-2cz336gLt2ZZBysroLhVbqBHs" async="true"></script>

Conclusion

In this section, we learned about `events` and how to use them in gem5. We also learned about `EventFunctionWrapper` and how to use it to schedule events. We also learned about `SimObject` initialization and how to use `startup` to schedule initial events.

In the exercises, we learned how to add an `event` to a `SimObject` and how to use `SimObjects` as parameters. We also learned how to use `DebugFlags` and `CompoundFlags` to print debug messages.

In the next section, we will learn about `Ports` and how to use them to communicate between `SimObjects`.