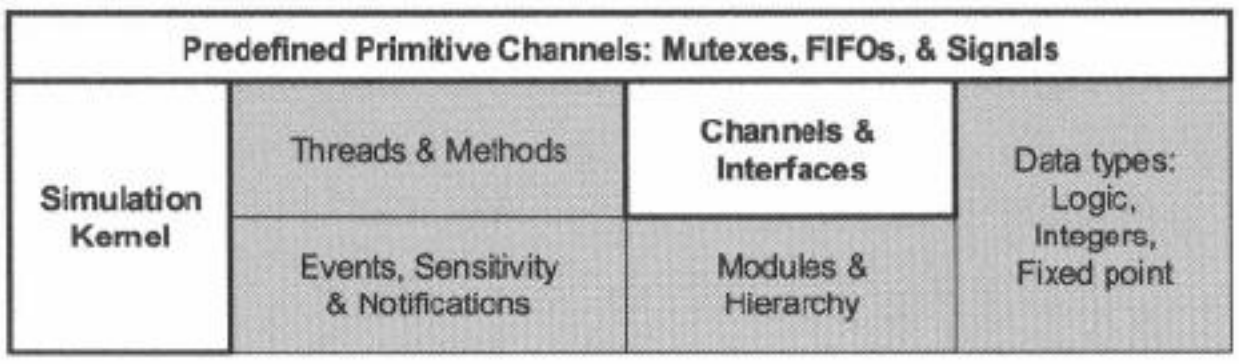

Chapter 8

BASIC CHANNELS

Thus far, we have communicated information between concurrent processes using events and using ordinary module member data. Because there are no guarantees about which processes execute next from the ready state, we must be extremely careful when sharing data.

Events let us manage this aspect of SystemC, but they require careful coding. Because events may be missed, it is important to update a handshake variable indicating when a request is made, and clear it when the request is acknowledged.

Data Communication

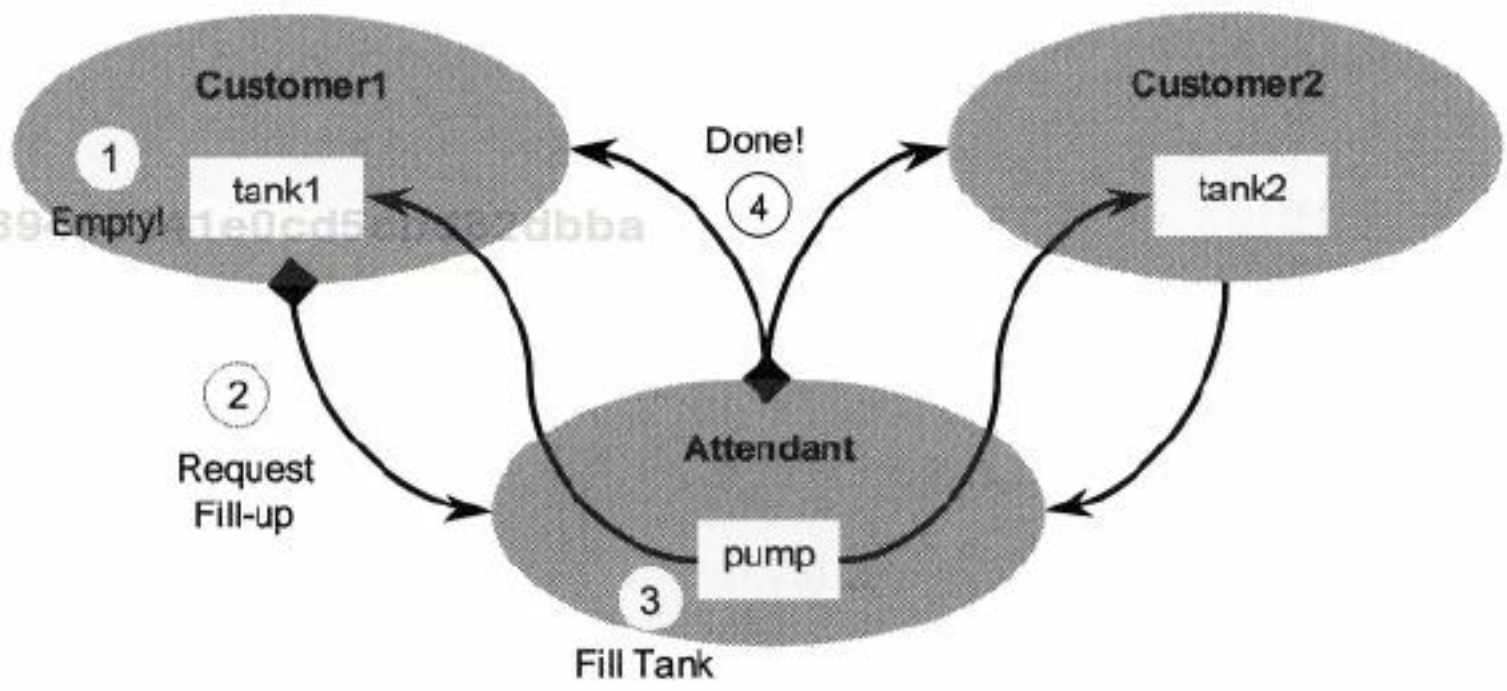

Figure 8-1. Gas Station Processes and Events

Let's consider the gas station example again as illustrated in *Figure 8-1*. The customer notices an empty tank ①. The attendant has to be watching when the customer requests a fill-up ②, and make note of it if in the middle of filling up another customer ③. More to the point, if the attendant waits on either customer's request (i.e., `wait(e_request1|e_request2)`), the semantics of `sc_event` doesn't allow the attendant to know which customer

made the request after the event happens. That is why the `gas_station` model uses the status of the gas tank as an indicator to choose whether to fill the tank. Similarly, the customer must watch to see if the tank actually was filled when the attendant yells done ④.

SystemC has built-in mechanisms, known as channels, to reduce the tedium of these chores, aid communications, and encapsulate complex communications. SystemC has two types of channels: primitive and hierarchical. This chapter concerns itself with primitive channels. Hierarchical channels are the subject matter of Chapter 13, Custom Channels.

8.1 Primitive Channels

SystemC's primitive channels are known as primitive because they contain no hierarchy, no processes, and are designed to be very fast due to their simplicity. All primitive channels inherit from the base class **`sc_prim_channel`**.

SystemC contains several built-in primitive channels. We will discuss channels exhibiting evaluate-update behavior in Chapter 9, Evaluate-Update Channels. In Chapter 13, Custom Channels, we will discuss custom primitive channels as an advanced topic. This chapter focuses on the simplest channels, **`sc_mutex`**, **`sc_semaphore`**, and **`sc_fifo`**.

8.2 sc_mutex

Mutex is short for mutual exclusion object. In computer programming, a mutex is a program object that lets multiple program threads share a common resource, such as file access, without colliding.

During elaboration, a mutex is created with a unique name; subsequently, any process that needs the resource must lock the mutex to prevent other processes from using the shared resource. The process should unlock the mutex when the resource is no longer needed. If another process attempts to access a locked mutex, that process is prevented until the mutex becomes available (unlocked).

SystemC provides mutex via the **`sc_mutex`** channel. This class contains several access methods including both blocked and unblocked styles. Remember blocking methods can only be used in **`SC_THREAD`** processes.

```
sc_mutex  NAME;

// To lock the mutex NAME  (wait until
// unlocked if in use)
NAME.lock();

// Non-blocking, returns true if success, else false
NAME.trylock()

// To free a previously locked mutex
NAME.unlock();
```

Figure 8-2. Syntax of sc_mutex

The example of the gas station attendant is a good example of a resource that needs to be shared. Only one car at a time is filled, assuming there is only a single gas pump.

Another example of a resource requiring a mutex would be the controls of an automobile. Only one driver at a time can sit in the driver's seat. In a simulation modeling the interaction of drivers across town with a variety of vehicles, this might be interesting to model.

```
sc_mutex  drivers_seat;
...
car->drivers_seat.lock();  // sim driver acquiring
                           // driver's seat
car->start();
... // operate vehicle
car->stop();
car->drivers_seat.unlock();  // sim driver leaving
                             // vehicle
```

Figure 8-3. Example of sc_mutex

An electronic design application of a **sc_mutex** might be arbitration for a shared bus. Here the ability of multiple masters to access the bus must be controlled. In lieu of an arbiter design, the **sc_mutex** might be used to manage the bus resource quickly until an arbiter can be arranged or designed.

In fact, the mutex might even he part of the class implementing the bus model as illustrated in the following example:

```
class bus {
  sc_mutex bus_access;
  ...
  void write(int addr,  int data) {
    bus_access.lock();
    // perform write
    bus_access.unlock();
  }
  ...
};//endclass
```

Figure 8-4. Example of sc_mutex Used in Bus Class

Used with an **SC_METHOD** process, access might look like this:

```
void grab_bus_method()  {
  if (bus_access.trylock()) {
    /* access bus */
  }//endif
}
```

Figure 8-5. Example of sc_mutex with an sc_method

One downside to the **sc_mutex** is the lack of an event that signals when an **sc_mutex** is freed, which requires using **trylock** repeatedly based on some other event or time based delay. Remember, unless your process waits (via a **wait** or **return**), you will not allow the process that currently owns the resource to free the resource, and the simulation will hang.

8.3 sc_semaphore

For some resources, you can have more than one copy or owner. A good example of this would be parking spaces in a parking lot.

To manage this type of resource, SystemC provides the **sc_semaphore**. When creating an **sc_semphore** object, it is necessary to specify how many are available. In a sense, a mutex is merely a semaphore with a count of one.

An **sc_semaphore** access consists of waiting for an available resource and then posting notice when finished with the resource.

```
sc_semaphore   NAME(COUNT);

//To lock a mutex,  NAME  (wait until
// unlocked if in use)
NAME.wait();

// Non-blocking, returns true if success else false
NAME.trywait()

//Returns number of available semaphores
NAME.get_value()

//To free a previously locked mutex
NAME.post();
```

Figure 8-6. Syntax of sc_semaphore

It is important to realize that the **sc_semaphore::wait()** is a distinctly different method from the **wait()** method previously discussed in conjunction with **SC_THREAD**s. In fact, under the hood, the **sc_semaphore::wait()** is implemented with the **wait**(event).

A modern gas station with self-service would be a good example for using semaphores. A semaphore could represent the number of available gas pumps.

```
SC_MODULE(gas_station) {
  sc_semaphore pump(12);
  void customer1_thread {
    for(;;) {
      // wait till tank empty
      ...
      // find an available gas pump
      pump.wait();
      // fill tank & pay
    }
};
```

Figure 8-7. Example of sc_semaphore—gas_station

A multi-port memory model is a good example of an electronic system-level design application for **sc_semaphore**. You might use the semaphore to indicate the number of read or write accesses allowed.

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  ...
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, int data) (
    write_ports.lock();
    // perform write
    write_ports.unlock();
  }
  ...
}; //endclass
```

Figure 8-8. Example of sc_semaphore—multiport_RAM

Other examples might include allocation of timeslots in a TDM (time division multiplex) scheme used in telephony, controlling tokens in a token ring, or perhaps even switching information to obtain better power management.

8.4 sc_fifo

Probably the most popular channel for modeling at the architectural level is the **sc_fifo**. First-in first-out queues (i.e., FIFOs) are a common data structure used to manage data flow. FIFOs are some of the simplest structures to manage.

In the very early stages of architectural design, the unbounded[25] STL **deque<>** (double ended queue) provides an easy implementation of a FIFO. Later, when bounds are determined or reasonable guesses at FIFO depths and SystemC elements come into stronger play, the **sc_fifo<>** may be used.

[25] Limited only by the resources of the simulation machine itself.

By default, an **sc_fifo<>** has a depth of 16. The data type (i.e., **typename**) of the elements also needs to be specified. An **sc_fifo** may contain any data type including large and complex structures (e.g., a TCP/IP packet or a disk block).

```
sc_fifo<ELEMENT_TYPENAME>  NAME(SIZE);

NAME.write  (VALUE);
NAME.read(REFERENCE);
... = NAME.read ()  /* function style */
if (NAME.nb_read (REFERENCE)) { // Non-blocking
                                // true if success
  ...
}
if (NAME.num_available() == 0)
  wait(NAME.data_written_event());
if (NAME.num_free() == 0)
  next_trigger(NAME.data_read_event());
```

Figure 8-9. Syntax of sc_fifo—Abbreviated

For example, FIFOs may be used to buffer data between an image processor and a bus, or a communications system might use FIFOs to buffer information packets as they traverse a network.

Some architectural models are based on Kahn process networks[26] for which unbounded FIFOs provide the interconnect fabric. Given an appropriate depth, you can use **sc_fifo** for this purpose as illustrated in the next very simple example.

[26] Kahn, G. 1974. The semantics of a simple language for parallel programming, in Proc. IFIP74, J.L. Rosenfeld (ed.), North-Holland, pp.471-475.

```
SC_MODULE(kahn_ex) {
  ...
  sc_fifo<double> a, b, y;
  ...
};
// Constructor
kahn_ex::kahn_ex() : a(10), b(10), y(20)
{
  ...
}
void kahn_ex::addsub_thread() {
  for(;;) {
    y.write(kA*a.read() + kB*b.read());
    y.write(kA*a.read() - kB*b.read());
  }//endforever
}
```

Figure 8-10. Example of sc_fifo kahn_ex

Software uses for FIFOs are numerous and include such concepts as mailboxes and other types of queues.

Note that when considering efficiency, passing pointers to large objects is most efficient. Be sure to consider using a safe pointer object if using a pointer. The **shared_ptr<>** of the GNU publicly licensed Boost library, http://www.boost.org, makes implementation of smart pointers very straight forward.

Generally speaking, the STL may be more suited to software FIFOs. The use of the STL **deque<>** might be used to manage an unknown number of stimulus data from a test bench.

In theory, an **sc_fifo** could be synthesized at a behavioral level. It currently remains for a synthesis tool vendor to provide the functionality.

8.5 Exercises

For the following exercises, use the samples provided in www.EklecticAlly.com/Book/.

Exercise 8.1: Examine, predict the output, compile, and run `mutex_ex`.

Examine 8.2: Examine, compile, and run `semaphore_ex`. Add another family member. Explain discrepancies in behavior.

Exercise 8.3: Examine, compile, and run `fifo_fir`. Add a second filter stage to the network.

Exercise 8.4: Examine, compile, and run `fifo_of_ptr`. Discuss how one might compensate for the simulated transfer of a large packet.

Exercise 8.5: Examine, compile, and run `fifo_of_smart_ptr`. Notice the absence of **`delete`**.

This page intentionally left blank