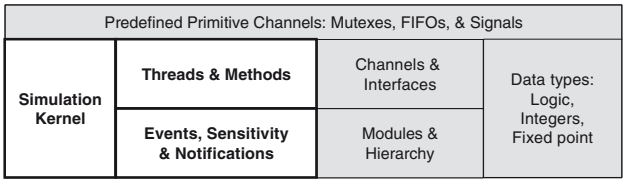
Chapter 6
Concurrency

Processes and Events

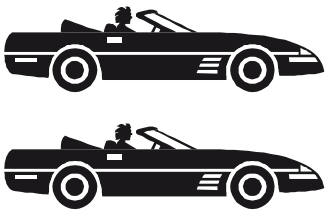


This chapter examines the topic of concurrency, which is fundamental to simulating with SystemC. We will first take a look at some types of concurrency, and then we will visit the simulation kernel used by SystemC. This examination will let us discuss SystemC thread processes, events, and sensitivity.

6.1 Understanding Concurrency

Many activities in a real system occur at the same time or concurrently. For example, when simulating something like a traffic pattern with multiple cars, the goal is to model the vehicles independently. In other words, the cars operate in parallel.

Software typically executes using a single thread of activity because there is usually only one processor on which to run, and partly because a thread is much easier to manage. On the other hand, in real systems many things occur simultaneously. For example, when an automobile executes a left turn, it is likely that the left turn indicator is flashing, the brakes are engaged to slow down the vehicle, engine power is decreased as the driver lets off the accelerator, and the transmission is shifting to a lower gear. All of these activities can occur at the same instant.



SystemC uses **simulation processes** to model concurrency. As with most event-driven simulators, concurrency is not true concurrent execution. In fact, simulated concurrency works like cooperative multitasking. When a simulation process runs, it is expected to execute a small segment of code and then return control to the simulation kernel. The SystemC simulator depends on a cooperative multitasking (non-preemptive) kernel that cannot force a running process to return control. This feature is unlike many operating systems that preemptively interrupt running processes to switch to a different process. A poorly behaved process that hogs control and

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_6, © Springer Science+Business Media, LLC 2010

doesn't yield control to the simulation kernel will cause SystemC simulation to hang. We'll look closer at this subject of control after we explain how SystemC uses C++ to model the process.

SystemC simulation processes are simply C++ functions designated by the programmer to be used as processes. You simply tell the simulation kernel which functions are to be used as simulation processes. This action is known as process registration. The simulation kernel then schedules and calls each of these functions as needed.

We've already seen how to register one type of process back in the chapter on SystemC modules using the **SC_THREAD**. Its syntax was simple as shown below (Fig. 6.1).

```
SC_THREAD(MEMBER_FUNCTION);
```

Fig. 6.1 Syntax of **SC_THREAD** macro

SC_THREAD has a few restrictions. First, it can be used only within a SystemC module; hence, the function must be a member function of the module class. Second, it must be used only during the elaboration stage, which we will talk about shortly. Suffice it to say that placing **SC_THREAD** in the module's constructor meets this requirement. Lastly, the member function must exist and the function can take no arguments and return no values. In other words, the function argument list is **void**, and the return value is **void**.

Let go back to the issue of control. We stated that processes must voluntarily yield control. Yielding control may take one of two forms. For one form, simulation processes yield control by executing a **return**. For the processes registered with **SC_THREAD**, executing **return** is uninteresting because it means the process has ended permanently.

The more interesting form is calling SystemC's **wait()** function. The **wait()** function suspends the process temporarily while SystemC proceeds to execute other processes, and then the function resumes by returning.

We've already seen one syntax for **wait(***delay_sc_time***)** in the last chapter. There are several more syntaxes, but we will only introduce one more for the moment (Fig. 6.2), which relates to time delays.

```
wait(TIME_DELAY,TIME_UNITS); // Convenience
```

Fig. 6.2 Another syntax of **wait()**

While we're on the subject of waiting for a time delay, note that waiting on a negative time does not make sense and is not defined in the standard. In any case, you should avoid using negative time delays. Simulators should probably issue a fatal error for this situation.

Let's see how normal time-out waits affect simulation. Here (Fig. 6.3) is a simple example with two processes. Both processes uses **wait**(*TIME_DELAY*) to simulate the passage of time. One process models a windshield wiper. The other process models the emergency blinkers in a similar manner.

```
//FILE: two_processes.h
SC_MODULE(two_processes) {
  void wiper_thread(void); // process
  void blinker_thread(void); // process
  SC_CTOR(two_processes) {
    SC_THREAD(wiper_thread); // register process
    SC_THREAD(blinker_thread); // register process
  }
};
```

```
//FILE: two_processes.cpp
void two_processes::wiper_thread(void) {
  while (true) {
    wipe_left();
    wait(500,SC_MS);
    wipe_right();
    wait(500,SC_MS);
  }//endwhile
}

void two_processes::blinker_thread(void) {
  while (true) {
    blinker = true;
    cout << "Blink ON" << endl;
    wait(300,SC_MS);
    cout << "Blink OFF" << endl;
    blinker = false;
    wait(300,SC_MS);
  }//endwhile
}
```

Fig. 6.3 Two processes using **wait()**

It is instructive to compile and run this code. You should download the examples that accompany this book and run the *two_processes* example. Notice how both processes use infinite loops. The loops are useful because if the processes ever return, they will never run again.

SystemC thread simulation processes typically begin execution at the start of simulation and continue in an endless loop until the simulation ends. Thread processes are started once; if they terminate, they cannot be restarted. Thread processes are required to periodically return control to the simulation kernel, allowing other processes to run. A thread process returns control to the simulation kernel by executing a **wait()** method call specifying an event or a time-out. Each time a thread returns control to the simulation kernel, its execution state is saved, which lets the process be resumed when the **wait()** returns.

6.2 Simplified Simulation Engine

Before continuing into the details of the SystemC syntax, we should examine the operation of the SystemC simulator. In this section, we explain thread processes using a simplified version (Fig. 6.4) of the simulation kernel. The remaining details will be covered later in this and other chapters. SystemC simulations consist of four major stages: elaboration, initialization, simulation, and post-processing.

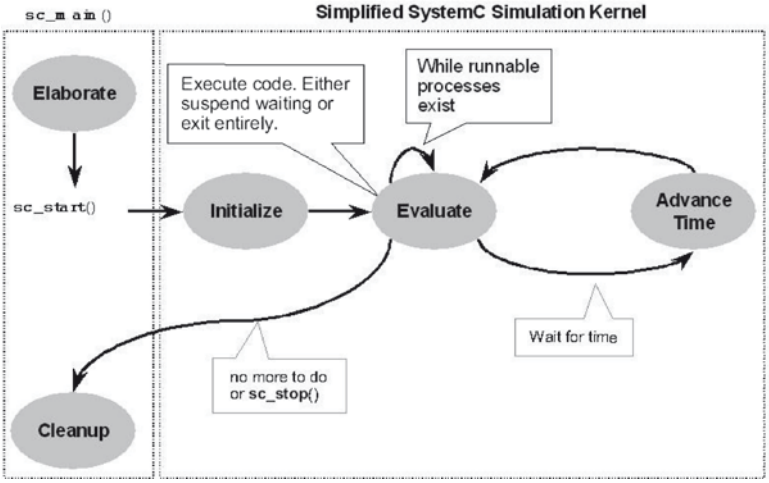


Fig. 6.4 Simplified SystemC simulation engine

In the previously described elaboration stage, SystemC components are instantiated and connected to create a model ready for simulation. This stage is also where process registration occurs. The elaboration stage ends with a call to `sc_start()`, which invokes the simulation kernel and begins the initialization stage.

In the initialization stage, the simulation kernel identifies all simulation processes and places them in either the runnable or waiting process set as illustrated in the next figure. By default, most simulation processes are placed into the set of runnable processes. Those processes explicitly requesting no initialization are placed into the set of waiting processes.

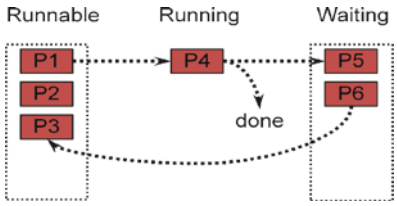


Fig. 6.5 Process sets

The simulation stage is commonly described as a state machine that schedules processes to run and advances simulation time. In this simplified simulation stage there are two internal phases: **`evaluate`** and **`advance-time`**. Simulation begins with evaluate.

During **`evaluate`**, all runnable processes are run one at a time. Each process runs until either it executes a **`wait`**(*`TIME_DELAY`*) or a return. If a **`wait`**(*`TIME_DELAY`*) is executed, then the time information is stored as an upcoming time event in the scheduling priority queue shown in the next (figure 6.6). The evaluate continues until there are no runnable processes left.

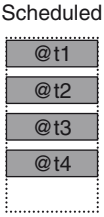


Fig. 6.6 Scheduling priority queue

It is important to note that the ordering of processes to run is unspecified in the standard. This means that when selecting which process to run from the set of runnable processes, any process may be chosen. Remember, we are simulating concurrency, which means that ideally we would run all of the processes simultaneously; however, we have only one processor to actually execute the simulation and so one process at a time is chosen. Different implementations of the simulator may run processes in different orderings; however, a given implementation must run processes in the same order to allow repeatability.

Once the set of all runnable processes has been emptied, then control passes to the advance-time phase to advance simulation time. Simulated time is moved forward to the closest time with a scheduled event. Time advancement moves processes waiting for that particular time into the runnable set, allowing the evaluation phase to continue.

This progression from evaluate to advance-time continues until one of three things occurs:

- All processes have yielded (i.e., there's nothing in the runnable set)
- A process has executed the function **`sc_stop()`**
- Internal 64-bit time variable runs out of values (i.e., maximum time is reached)

At this point, simulation stops and we proceed into the post-processing stage (AKA cleanup).

We'll now turn our attention to some details of the processes.

6.3 Another Look at Concurrency and Time

Let's take a look at a hypothetical example to better understand how time and execution interact. Consider a design with four processes as illustrated in Fig. 6.7. For this discussion, assume the times, t_1, t_2, and t_3 are non-zero. Each process contains lines of code or statements ($stmt_{A1}$, $stmt_{A2}$, …) and executions of wait methods (**wait(**t_1**)**, **wait(**t_2**)**,…)

```
Process_A() {
  //@ t0
  stmtA1;
  stmtA2;
  wait(t1);
  stmtA3;
  stmtA4;
  wait(t2);a
  stmtA5;
  stmtA6;
  wait(t3);
}
```

```
Process_B() {
  //@ t0
  stmtB1;
  stmtB2;
  wait(t1);
  stmtB3;
  stmtB4;
  wait(t2);
  stmtB5;
  stmtB6;
  wait(t3);
}
```

```
Process_C() {
  //@ t0
  stmtC1;
  stmtC2;
  wait(t1);
  stmtC3;
  stmtC4;
  wait(t2);
  stmtC5;
  stmtC6;
  wait(t3);
}
```

```
Process_D() {
  //@ t0
  stmtD1;
  stmtD2;
  wait(t1);
  stmtD3;
  wait(
    SC_ZERO_TIME);
  stmtD4;
  wait(t3);
}
```

Fig. 6.7 Four processes

Notice that `Process_D` skips t_2. At first glance, it might be perceived that time passes as shown below in Fig. 6.8. The uninterrupted solid and gray line portions indicate program activity. Vertical discontinuities indicate a wait.

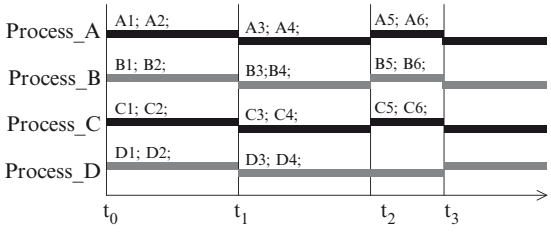


Fig. 6.8 Simulated activity—perceived

Each process' statements take some time and are evenly distributed along simulation time. Perhaps surprisingly that is *not* how it works at all. In fact, actual simulated activity is shown in Fig. 6.9.

Each set of statements executes in zero time! Let's expand the time scale to expose the simulator's internal activity as well. This expansion exposes the operation of the scheduler at work (Fig. 6.10).

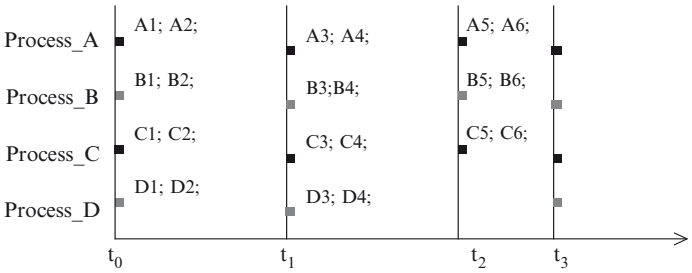


Fig. 6.9 Simulated activity—actual

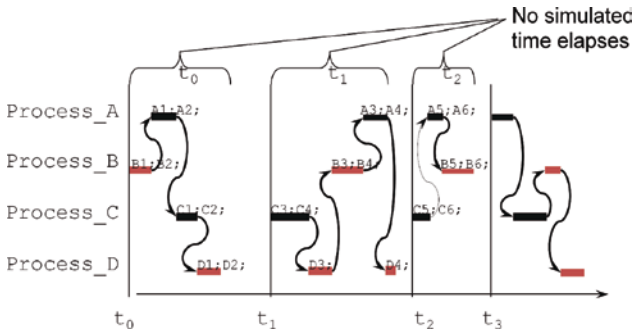


Fig. 6.10 Simulated activity with simulator time expanded

Notice that process execution order appears random in this example. This apparent random sequencing is allowed by the SystemC standard. Now for any given simulator and set of code, there is also a requirement that the simulation be deterministic in the sense that one may rerun the simulation and obtain the same results.

All of the executed statements in this example execute during the same **`evaluate`** phase.

As a final consideration, the previous diagrams would be equally valid with any or all of the indicated times t_1, t_2, or t_3 as zero (i.e., `SC_ZERO_TIME`). Once you grasp these fundamental concepts, understanding SystemC behaviors will become much easier.

6.4 The SystemC Thread Process

SystemC has two basic types of simulation processes: thread processes and method processes. Thread processes are the easiest to code and are the most popular for SystemC applications. They are named thread processes because their behavior most closely models the usual software connotation of a thread of execution. We look at the SystemC method processes later in this chapter.

SystemC thread processes begin execution at the start of simulation and typically continue in an endless loop until the simulation ends. SystemC thread processes are started once and only once by the simulator.

As we have noted, a running thread process has complete control of the simulation until it decides to yield control to the simulator kernel. A thread process can return control in two ways. First, by waiting, which suspends the process to be resumed later, and second, by simply exiting (returning). A thread that exits cannot be resumed and will not be restarted for the duration of the simulation.

Thread processes that implement endless loops must have at least one explicit or implicit call to the **`wait()`** function. One or more calls to **`wait()`** may be included in the endless loop; these calls are explicit waits. A wait call may be contained in a function called from the threads; this is called an implicit wait. For instance, a blocking **`read()`** call to an instance of the **`sc_fifo`**<*T*> invokes **`wait()`** when the FIFO is empty.

6.5 SystemC Events

Before discussing threads more extensively, it is necessary to discuss events. SystemC is an event-driven simulator. In fact, we have already discussed one type of event: the time-out event that occurs implicitly with the **`wait`***(TIME_DELAY)* syntax.

An event is something that happens at a specific instant in time. An event has no value and no duration. SystemC uses the **`sc_event`** class to model events. This

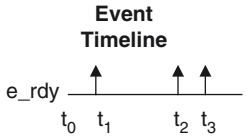


Fig. 6.11 Graphical representation of an event

class allows explicit triggering (launching, firing, causing) of events by means of a notification method.

DEFINITION: A SystemC event is the occurrence of an **`sc_event`** notification and happens at a single instant in time. An event has no duration or value.

Once an event occurs, there is no trace of its occurrence other than the side effects that may be observed as a result of processes that were waiting for the event. We will explain how processes wait a few sections ahead. The next diagram (Fig. 6.11) illustrates an event **`e_rdy`** triggering at three different instants. Note that unlike a waveform, events have no time width.

Because events have no duration, you must be watching to catch them. Quite a few coding errors are due to not understanding this simple rule. Let's restate it.

RULE: To observe an event, the observer must be watching for the event prior to its notification.

If an event occurs and no processes are waiting to catch it, the event goes unnoticed. The syntax to declare a named event is simple and shown in Fig. 6.12.

Remember that an **`sc_event`** has no value, and you can perform only two actions with a **`sc_event`**: wait for it or cause it to occur.

```
sc_event event_name1[,event_namei]…;
```

Fig. 6.12 Syntax of **sc_event**

```
event_name.notify(void);          // Immediate
event_name.notify(SC_ZERO_TIME);// Delayed
event_name.notify(sc_time);       // Timed (time>0)
event_name.notify(double,units);// Convenience
```

Fig. 6.13 Syntax of **sc_event::notify()**

6.5.1 Causing Events

Events are explicitly caused using the **notify()** method of an **sc_event** object. Here (Fig. 6.13) is the syntax:

Invoking an immediate **notify(void)** causes any processes waiting for the event to be immediately moved from the waiting set into the runnable set for execution. This topic will be examined in more detail in an upcoming section.

Delayed notification occurs when a time of zero is specified. The predefined constant **SC_ZERO_TIME** is simply **sc_time(0,SC_SEC)** Processes waiting for a delayed notification will execute only after all runnable processes in the current evaluation state have executed. For now, it is sufficient to consider that delayed notification is treated the same as a timed notification with a time of zero. This feature is quite useful as we will see later.

Timed notification would appear to be the easiest to understand. Timed events are scheduled to occur at some time in the future.

One confounding aspect of timed events, which includes delayed events, concerns multiple notifications. An **sc_event** may have no more than a single outstanding scheduled event, and only the nearest time notification is allowed. If a nearer notification is scheduled, the previous outstanding scheduled event is canceled. For example, consider the following (Fig. 6.14):

```
sc_event A_event;
A_event.notify(10,SC_NS);
A_event.notify( 5,SC_NS); // only this one stays
A_event.notify(15,SC_NS);
```

Fig. 6.14 Syntax of **notify()** method

```
event_name.cancel();
```

Fig. 6.15 Syntax of **cancel()** method

Outstanding scheduled events may be canceled with the **cancel()** method (Fig. 6.15). Note that immediate events cannot be canceled because they happen at the precise instant they are notified (i.e., immediately).

6.6 Catching Events for Thread Processes

Thread processes rely on the **wait**() method to suspend their execution. The **wait**() method supplied by SystemC has several syntaxes shown in the next figure, Fig 6.16.

When the wait function is called, control is returned to the simulator kernel, the state of the current thread process is saved, and eventually a new process is allowed to run.

When a suspended thread process is selected to run, the simulation kernel restores the calling context, and the process resumes execution at the statement following the call to **wait**().

```
wait(time);                  // timeout is the event
wait(double,time_unit);      // convenience
wait(event);                 // single event
wait(event1 | eventn…);      // any of these
wait(event1 & eventn…);      // all of these
wait(time,event);            // event or timeout
wait(time,event1 | eventn…);// any event or timeout
wait(time,event1 & eventn…);// all events or timeout
wait(); // static sensitivity - discussed later
```

Fig. 6.16 Syntax of **SC_THREAD wait()**

The first two syntaxes for **wait** (*time*) provide a delay for a period of simulation time as described in the Chapter 5.

The next several forms specify events and suspend execution until one or all the events have occurred. The operator | is defined to mean any of these events; whichever one happens first will cause a return to wait. The operator **&** is defined to mean all of these events in any order must occur before wait returns. The last syntax, **wait**(), will be deferred to a joint discussion with static sensitivity later in this chapter.

The three forms that have time with a second argument constitute a time-out. This result is really just the logical **or** of a time event with other events. Use of a

```
…
sc_event ack_event, bus_error_event;
…
sc_time start_time(sc_time_stamp());
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (sc_time_stamp()-start_time == t_MAX_DELAY) {
  break; // path for a time out
  …
```

Fig. 6.17 Example using time-out variation of **wait**()

time-out is handy when testing protocols and various error conditions and an example is given in Fig. 6.17.

Notice when multiple events are or'ed, it is not possible to know which event occurred in a multiple event wait situation as events have no value. Thus (Fig. 6.18), it is illegal to test an event for **`true`** or **`false`**.

```
if (ack_event) do_something; // syntax error!
```

Fig. 6.18 Example of illegal Boolean compare of **`sc_event`**

It is legal to test a flag that is set by the process that caused an event; however, it is problematic to clear this flag properly.

6.7 Zero-Time and Immediate Notifications

Two concepts, zero-time delays and immediate notification, bear special treatment. Consider again the simulation engine diagram below Fig. 6.19 (reproduced for convenience).

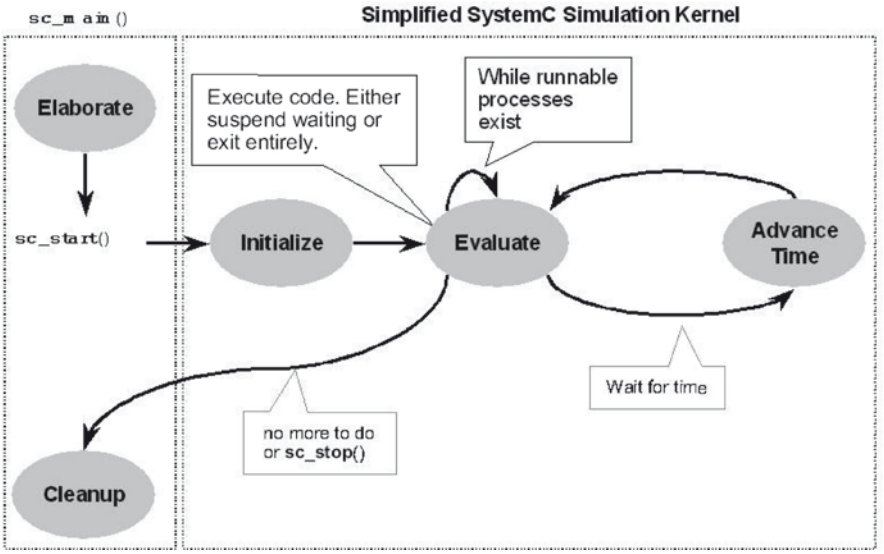


Fig. 6.19 Simplified SystemC simulation engine

Consider what it means to execute **`wait(SC_ZERO_TIME)`**. What does it mean to advance time by zero? The primary effect is that a process waiting for zero-time will resume after all the runnable processes have yielded. Since zero is

always closer than any other time, then all processes waiting for zero-time will be next in line to become runnable.

```
SC_MODULE(missing_event) {
  SC_CTOR(missing_event) {
    SC_THREAD(B_thread); // ordered to cause
    SC_THREAD(A_thread); // problems
    SC_THREAD(C_thread);
  }
  void A_thread() {
    a_event.notify(); // immediate!
    cout << "A sent a_event!" << endl;
  }
  void B_thread() {
    wait(a_event);
    cout << "B got a_event!" << endl;
  }
  void C_thread() {
    wait(a_event);
    cout << "C got a_event!" << endl;
  }
  sc_event a_event;
};
```

Fig. 6.20 Example of zero-time

This feature can be very useful. Recall the rule that to observe an event, the observer must be watching for the event prior to its notification. Now imagine we have three processes, `A_thread, B_thread,` and `C_thread`. The implementation code is shown in Fig. 6.20.

Suppose they execute in the order `A_thread, B_thread, C_thread`. Furthermore, notice that `A_thread` does an immediate notification of an event, `a_event`, which `B_thread and C_thread` are going to wait for.

If either `B_thread` or `C_thread` have not issued the **`wait`**`(a_event)` call prior to `A_thread` notifying the event, then they will miss the event. If the event never happens again, then when `B_thread` or `C_thread` issue the **`wait`**`(a_event)` call, they will wait forever.

If the event happened a second time, then `B_thread` or `C_thread` would continue, but they would have missed one of the events. Missing an event can be devastating to a simulation. This situation can be avoided by use of the zero-time delayed notification, **`notify(SC_ZERO_TIME)`**. The reason is that delayed notifications are issued only after completing all runnable processes.

Lest you think that you can simply fix the problem by ordering the processes appropriately, recall that SystemC implementations are free to choose processes from

the runnable set in any order. This unpredictability of selection is because we are simulating concurrency. Also, consider that in a real-world simulation, there may be hundreds of processes. The bottom line is that you must write your code so that it does not depend on the order of process execution except by design (e.g., using an event to force ordering). Consider the example in Fig. 6.21 that correctly handles this case.

```
SC_MODULE(ordered_events) {
  SC_CTOR(ordered_events) {
    SC_THREAD(B_thread); // ordered to cause
    SC_THREAD(A_thread); // problems
    SC_THREAD(C_thread);
  }
  void A_thread() {
    while (true) {
      a_event.notify(SC_ZERO_TIME);
      cout << "A sent a_event!" << endl;
      wait(c_event);
      cout << "A got c_event!" << endl;
    }//endwhile
  }
  void B_thread() {
    while (true) {
      b_event.notify(SC_ZERO_TIME);
      cout << "B sent b_event!" << endl;
      wait(a_event);
      cout << "B got a_event!" << endl;
    }//endwhile
  }
  void C_thread() {
    while (true) {
      c_event.notify(SC_ZERO_TIME);
      cout << "C sent c_event!" << endl;
      wait(b_event);
      cout << "C got b_event!" << endl;
    }//endwhile
  }
  sc_event a_event, b_event, c_event;
};
```

Fig. 6.21 Example of properly ordered events

Here is another difficulty that can arise when using immediate notification. When a process executes **`notify(void)`**, it may cause one or more processes in the waiting set to be moved into the runnable set. Consider the example in Fig. 6.22.

```
SC_MODULE(event_hogs) {
  SC_CTOR(event_hogs) {
    SC_THREAD(A_thread);
    SC_THREAD(B_thread);
    SC_THREAD(C_thread);
    // Following ensures sc_stop() works
    sc_set_stop_mode(SC_STOP_IMMEDIATE);
  }
  sc_event a_event, b_event;
  void A_thread() {
    while(true) {
      cout << "A@" << sc_time_stamp() << endl;
      a_event.notify(); // immediate!
      wait(b_event);
    }
  }
  void B_thread() {
    int count(8); // limit execution
    while(true) {
      cout << "B@" << sc_time_stamp() << endl;
      b_event.notify(); // immediate!
      wait(a_event);
      if (count-- == 0) sc_stop();
    }
  }
  void C_thread() {
    while(true) {
      cout << "C@" << sc_time_stamp() << endl;
      wait(1,SC_NS);
    }
  }
};
```

Fig. 6.22 Example of improper use of events

6.8 Understanding Events by Way of Example

The best way to understand events is by way of example. Notice in the code of Fig. 6.23 that all notifications execute at the same instant.

```
…
sc_event action;
sc_time now(sc_time_stamp()); //observe current time
//immediately cause action to fire
action.notify();
//schedule new action for 20 ms from now
action.notify(20,SC_MS);
//reschedule action for 1.5 ns from now
action.notify(1.5,SC_NS);
//useless, redundant
action.notify(1.5,SC_NS);
//useless preempted by event at 1.5 ns
action.notify(3.0,SC_NS);
//reschedule action for evaluate cycle
action.notify(SC_ZERO_TIME);
//useless, preempted by action event at SC_ZERO_TIME
action.notify(1,SC_SEC);
//cancel action entirely
action.cancel();
//schedule new action for 1 femto sec from now
action.notify(20,SC_FS);
…
```

Fig. 6.23 Example of **`sc_event notify()`** and **`cancel()`** methods

To illustrate the use of events, let's consider how one might model the interaction between switches on a steering wheel column and the remotely located signal indicators (lamps). The example illustrated in Fig. 6.24 models a mechanism that detects switching activity and notifies the appropriate indicator. For simplicity, only the stop light interaction is modeled here.

In this model, the process `turn_knob_thread` provides a stimulus and interacts with the process `stop_signal_thread`. The idea is to have several threads representing different signal indicators. The `turn_knob_thread` process directs each indicator to turn on or off via the `signal_stop` and `signals_off` events. The indicators provide their status via the `stop_indicator_on` and `stop_indicator_off` events.

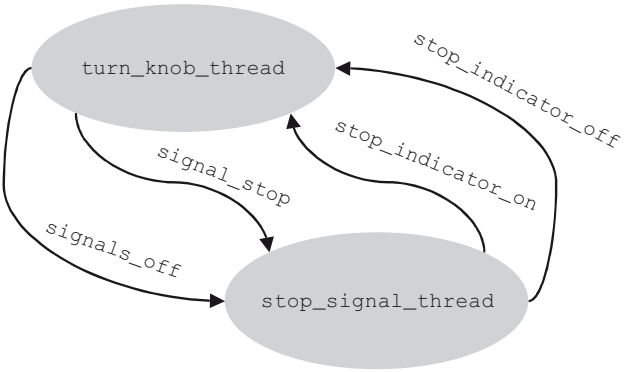


Fig. 6.24 Turn of events illustration

An interesting aspect of the example shown in Fig. 6.25 through Fig. 6.27 is consideration of process ordering effects. Recall the rule that "To see an event, a process must be waiting for it." It is because of this requirement that the `turn_knob_thread` implementation starts out with **`wait(SC_ZERO_TIME)`**. Without that pause, if `turn_knob_thread` runs first, then the stop_signal_thread will never see any events because it will not have executed the first **`wait`**`()`. As a result, the simulation would starve and exit.

```
//FILE: turn_of_events.h
SC_MODULE(turn_of_events) {
  // Constructor
  SC_CTOR(turn_of_events) {
    SC_THREAD(turn_knob_thread);
    SC_THREAD(stop_signal_thread);
  }
  sc_event signal_stop, signals_off;
  sc_event stop_indicator_on, stop_indicator_off;
  void turn_knob_thread();   // stimulus process
  void stop_signal_thread(); // indicator process
};//endclass turn_of_events
```

Fig. 6.25 Example of **`turn_of_events`** header

```
//FILE: turn_of_events.cpp
void turn_of_events::turn_knob_thread() {
  // This process provides stimulus to the design
  // by way of standard I/O.
  enum directions {STOP='S', OFF='F'};
  char direction; // Selects appropriate indicator
  bool did_stop = false;
  // allow other threads to get into waiting state
  wait(SC_ZERO_TIME);
  while(true) {
    // Sit in an infinite loop awaiting keyboard
    // or STDIN input to drive the stimulus…
    cout << "Signal command: ";
    cin >> direction;
    switch (direction) {
      case STOP:
        // Make sure the other signals are off
        signals_off.notify();
        signal_stop.notify(); // Turn stop light on
        // Wait for acknowledgement of indicator
        wait(stop_indicator_on);
        did_stop = true;
        break;
      case OFF:
        // Make the other signals are off
        signals_off.notify();
        if (did_stop) wait(stop_indicator_off);
        did_stop = false;
        break;
    }//endswitch
  }//endforever
}//end turn_knob_thread()
```

Fig. 6.26 Example of **`turn_of_events`** stimulus

```
void turn_of_events::stop_signal_thread() {
  while(true) {
    wait(signal_stop);
    cout << "STOPPING     !!!!!!" << endl;
    stop_indicator_on.notify();
    wait(signals_off);
    cout << "Stop off     ------" << endl;
    stop_indicator_off.notify();
  }//endforever
}//end stop_signal_thread()
```

Fig. 6.27 Example of **turn_of_events** indicator

```
% ./turn_of_events.x
Signal command: S
STOPPING  !!!!!!
Signal command: F
Stop off  ------…
```

Fig. 6.28 Example of **turn_of_events** output

Similarly, consider what would happen if the `signals_off` event were issued before `signal_stop`. If an unconditional wait for acknowledgement occurred, the simulation would exit. It would exit because the `turn_knob_thread` would be waiting on an event that never occurs because the `stop_signal_thread` was not in a position to issue that event.

The preceding example, `turn_of_events`, models two processes with SystemC threads. The `turn_knob_thread` takes input from the keyboard and notifies the `stop_signal_thread`.

6.9 The SystemC Method Process

As mentioned earlier, SystemC has more than one type of process. The **SC_METHOD** process is in some ways simpler than the **SC_THREAD**; however, this simplicity makes it more difficult to use for some modeling styles. To its credit, **SC_METHOD** may be slightly more efficient in some situations than **SC_THREAD**.

What is different about an **SC_METHOD**? One major difference is invocation. **SC_METHOD** processes never suspend internally (i.e., they can never invoke **wait**()). Instead, **SC_METHOD** processes run completely and return.

In some sense, **SC_METHOD** processes are similar to the Verilog `always@` block or the VHDL process. By contrast, if an **SC_THREAD** terminates, it never runs again in the current simulation.

Because **SC_METHOD** processes are prohibited from suspending internally, they may not call the **wait()** method. Attempting to call **wait()** either directly or indirectly from an **SC_METHOD** will result in a run-time error.

Implicit waits result from calling functions that are defined such that they may issue a **wait()**. These are known as blocking methods. As discussed later in this book, the **read()** and **write()** methods of the **sc_fifo**<*T*> data type are examples of blocking methods. Thus, **SC_METHOD** processes must avoid using calls to blocking methods.

The syntax for **SC_METHOD** processes follows (Fig. 6.29) and is identical to **SC_THREAD** except for the keyword **SC_METHOD**:

```
SC_METHOD(process_name);//Located INSIDE constructor
```

Fig. 6.29 Syntax of **SC_METHOD**

A note on the choice of these keywords might be useful. The similarity of names between an **SC_METHOD** process and a regular object-oriented method betrays its name. An **SC_METHOD** executes without interruption and returns to the caller (the simulation kernel). By contrast, an **SC_THREAD** process is more akin to an operating system thread, which suspends (waits) returning control to the simulation kernel and later the kernel can resume that thread process.

Variables allocated in **SC_THREAD** processes are persistent. **SC_METHOD** processes must declare and initialize variables each time the method is invoked. For this reason, **SC_METHOD** processes typically rely on module local data members declared within the **SC_MODULE**. **SC_THREAD** processes tend to use locally declared variables.

GUIDELINE: To differentiate threads from methods, we strongly recommend adopting a naming style. One naming style appends *_thread* or *_method* as appropriate. Being able to differentiate processes based on names becomes useful during debug.

```
next_trigger(time);
next_trigger(timeout,time_unit); //convenience
next_trigger(event);
next_trigger(event1 | event_i…); //any of these
next_trigger(event1 & event_i…); //all of these
                                 //required
next_trigger(timeout,event); //event with timeout
next_trigger(timeout,event1 | event_i…);//any + timeout
next_trigger(timeout,event1 & event_i…);//all + timeout
next_trigger(void); //re-establish static sensitivity
```

Fig. 6.30 Syntax of **SC_METHOD next_trigger()**

6.10 Catching Events for Method Processes

SC_METHOD processes dynamically specify their sensitivity by means of the **next_trigger**() method as shown in Fig 6.30. This method has the same syntax as the **wait**() method but with a slightly different behavior.

As with **wait**(), the multiple event syntaxes do not specify order. Thus, with **next_trigger**(evt1 & evt2), it is not possible to know which occurred first. It is only possible to assert that both evt1 and evt2 happened.

The **wait**() method suspends **SC_THREAD** processes; however, **SC_METHOD** processes are not allowed to suspend. The **next_trigger**() method has the effect of temporarily setting a sensitivity list that affects the **SC_METHOD**. The **next_trigger**() method may be called repeatedly, and each invocation encountered overrides the previous. The last **next_trigger**() executed before a return from the process determines the sensitivity for a recall or of the process. The initialization call is vital to making this work. See the **next_trigger()** code in the downloads section of the web site for an example.

You should note that it is critical for every path through an **SC_METHOD** to specify at least one **next_trigger**() for the process to be called by the scheduler. Without a **next_trigger**() or static sensitivity (discussed in the next section), an **SC_METHOD** will never be executed again.

You might be tempted to place a default **next_trigger**() as the first statement of the **SC_METHOD**, since subsequent calls to **next_trigger**() will overwrite any previous calls. For fairly simple designs, this approach may work; however, there is a potential problem since it could mask problem where you intended a different trigger. It might be better to both allow the possibility of hanging and insert some code to determine if the process is active. An even better solution is to use a tool that can statically analyze your code for correctness. Having said that, the next section discusses a technique that guarantees a default **next_trigger**().

6.11 Static Sensitivity for Processes

Thus far, we've discussed techniques of dynamically (i.e., during simulation) specifying how a process will resume (either by **SC_THREAD** using **wait**() or by **SC_METHOD** using **next_trigger**(). The concept of actively determining what will cause a process to resume is often called **dynamic sensitivity**.

SystemC provides another type of sensitivity for processes called **static sensitivity**. Static sensitivity establishes the parameters for resumption during elaboration (i.e., before simulation begins). Once established, static sensitivity parameters cannot be changed (i.e., they're static).

The purpose of specifying static sensitivity is simply a convenience. Some processes have a single list of items to which they are sensitive. To preserve code and make this clearer to the reader, use a static list rather than dynamically specifying the same thing over and over again. It is possible to override static sensitivity with

```
// IMPORTANT: Must follow process registration
sensitive << event [<< event]…; // streaming style
```

Fig. 6.31 Syntax of sensitive

dynamic sensitivity, but this is rarely used. Static sensitivity is most commonly used with the RTL level of coding.

Static sensitivity is established for each process immediately after its process registration. Events may be added to a processes static sensitivity list, which is initially empty, using the insertion operator (<<) on a special **sc_module** data member called **sensitive**. The syntax is shown in Fig. 6.31.

The event object on the right has several variations that we need to defer until we have a more concrete notion of the concepts of channels, ports, and event finders. Suffice it to say that some objects can return events to be added to the sensitivity list.

For method processes, simply not specifying **next_trigger**() at all implies that static sensitivity, if any exists, will be used. Thus static sensitivity is most commonly used with method processes; however, use of the **wait(void)** syntax lets thread processes also use static sensitivity. In fact, **next_trigger**() was originally invented as a technique to alter the method process sensitivity.

```
SC_MODULE(gas_station) {
  sc_event e_request1, e_request2;
  sc_event e_tank_filled;
  SC_CTOR(gas_station) {
    SC_THREAD(customer1_thread);
      sensitive(e_tank_filled); // functional
                                // notation
    SC_METHOD(attendant_method);
      sensitive << e_request1
                << e_request2; // streaming notation
    SC_THREAD(customer2_thread);
  }
  void attendant_method();
  void customer1_thread();
  void customer2_thread();
};
```

Fig. 6.32 Example of gas station declarations

For the next few sections, we will examine the problem of modeling access to a gas station to illustrate the use of sensitivity coupled with events. Initially, we model a single pump station with an attendant and only two customers as illustrated in Fig. 6.33. The declarations for this example in Fig. 6.32 illustrate the use of the **sensitive** method.

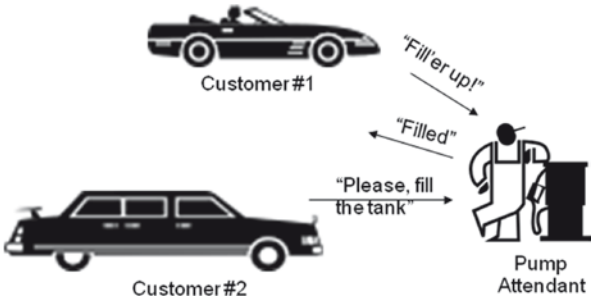


Fig. 6.33 Initial Gas Station Illustration

The `gas_station` module has two processes with different sensitivity lists and one, `customer2_thread`, which has none. The `attendant_method` implicitly executes every time an `e_request1` or `e_request2` event occurs (unless dynamic sensitivity is invoked by the simulation process).

Notice the indentation used in Fig. 6.32. This format helps draw attention to the sensitivity being associated with only the most recent process registration.

Figure 6.34 has some fragments of the implementation code focused on the elements of this chapter. You can find the full code in the downloads section of the web site.

```
…
void gas_station::customer1_thread() {
  while (true) {
    wait(EMPTY_TIME);
    cout << "Customer1 needs gas" << endl;
    m_tank1 = 0;
    do {
      e_request1.notify();
      wait(); // use static sensitivity
    } while (m_tank1 == 0);
  }//endforever
}//end customer1_thread()

// omitting customer2_thread (almost identical
// except using wait(e_request2);)

void gas_station::attendant_method() {
  if (!m_filling) {
      …
      cout << "Filling tank" << endl;
      m_filling = true;
      next_trigger(FILL_TIME);
      …
  } else {
      …
      e_filled.notify(SC_ZERO_TIME);
      cout << "Filled tank" << endl;
    …
    m_filling = false;
    …
  }//endif
}//end attendant_method()
```

Fig. 6.34 Example of gas station implementation

```
…
Customer1 needs gas
Filling tank
Filled tank
…
```

Fig. 6.35 Example of gas station sample output

6.12 Altering Initialization

The simulation engine description specifies that processes are executed at least once initially by placing processes in the runnable set during the initialization stage. This approach makes no sense in the preceding `gas_station` model as the `attendant_method` would fill the tank before being requested.

Thus, it may be necessary to specify that some processes should not be made runnable at initialization. For this situation, SystemC provides the **`dont_initialize`**`()` method. The syntax follows:

Note that the use of **`dont_initialize`**`()` requires a static sensitivity list; otherwise, there would be nothing to start the process. Now our `gas_station` module looks like Fig. 6.37.

```
// IMPORTANT: Must follow process registration
dont_initialize();
```

Fig. 6.36 Syntax of **`dont_initialize`**`()`

```
…
SC_METHOD(attendant_method);
  sensitive(fillup_request);
  dont_initialize();
…
```

Fig. 6.37 Example of **`dont_initialize`**`()`

```
  sc_event_queue event_name1("event_name1")…;
```

Fig. 6.38 Syntax of **`sc_event_queue`**

6.13 The SystemC Event Queue

In light of the limitation that **sc_event** may only have a single outstanding event scheduled, the **sc_event_queue** was added with the syntax shown in Fig. 6.38. This addition allows a single event to be scheduled multiple times even for the same instant in time! When events are scheduled for the same instant in time, each happens in a separate evaluation.

There are certain situations in which one may wish to catch multiple events; however, often it may indicate a misunderstanding of the model. Most modeling only requires the **sc_event**. Furthermore, **sc_event_queue** has a performance impact. Since events impact a huge portion of simulation time, performance in this area needs to be carefully considered.

sc_event_queue is slightly different from **sc_event**. First, **sc_event_queue** objects do not support immediate notification since there is obviously no need to queue these. Second, the **cancel**() method is replaced with **cancel_all**() to emphasize that it cancels all outstanding **sc_event_queue** notifications.

The following diagram may clarify what the preceding code does:

```
sc_event_queue action;
wait(10,SC_NS)//assert time=10ns
sc_time now1(sc_time_stamp());//observe current time
action.notify(20,SC_NS);//schedule for 20ns from now
action.notify(10,SC_NS);//schedule for 20ns from now
action.cancel_all();//cancel all actions entirely
action.notify(8,SC_NS);//schedule for 8 ns from now
action.notify(1.5,SC_NS);// 1.5 ns from now
action.notify(1.5,SC_NS);// another identical action
action.notify(3.0,SC_NS);// 3.0 ns from now
action.notify(SC_ZERO_TIME);//after all runnable
action.notify(SC_ZERO_TIME);//and yet another
action.notify(12,SC_NS);// 12 ns from now
sc_time now2(sc_time_stamp());//observe current time
```

Fig. 6.39 Example of **sc_event_queue**

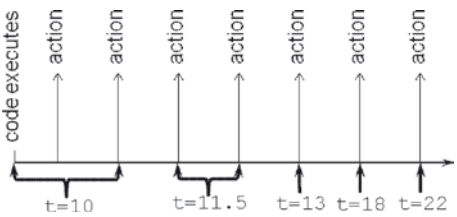


Fig. 6.40 Example of **sc_event_queue**

Notice that the first two events were completely cancelled as was explicitly stated and that all seven remaining events occurred. In fact, the evaluation phase may be entered seven distinct times, if there are processes waiting on the event. Finally, observe that time never passes between now1 and now2 because the code never yielded to the simulation kernel in that section of code. Seven events then remain to be waited upon.

6.14 Exercises

For the following exercises, use the samples provided in www.scftgu.com

Exercise 6.1: Examine and run the `turn_of_events` example. Remove the **`wait(SC_ZERO_TIME)`** in the `turn_knob_thread` process. Is this guaranteed to give different behavior? If your implementation of SystemC does not create different behavior, insert a **`wait`** (**`SC_ZERO_TIME`**) at the top of **`stop_signal_thread`**. What is the behavior that you observe? Why?

Exercise 6.2: Learn how to delay time using an **`SC_METHOD`**. Examine, predict the behavior, compile, and run the `method_delay` example from the downloaded examples.

Exercise 6.3: This exercise illustrates some of the complexities of **`SC_METHOD`** processes. Examine, predict the behavior, compile, and run the **`next_trigger`** example.

Exercise 6.4: This exercise will clarify the differences between **`sc_event`** and **`sc_event_queue`** constructs. Examine, predict the behavior, compile, and run the `event_filled` example.