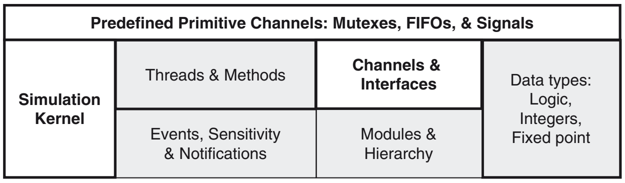
# Chapter 7 Dynamic Processes


Predefined Primitive Channels: Mutexes, FIFOs, & Signals
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy

In this chapter, we will focus on a discussion of dynamic threads and **`SC_FORK`**/**`SC_JOIN`** constructs. Dynamic processes can be particularly useful in testbench scenarios to track transaction completion or to spawn traffic generators dynamically. In the following sections, we will cover the syntax required to generate dynamic processes as well as some application examples.

## 7.1 Introduction

Thus far, all the process types discussed have been static. In other words, once the elaboration phase completes, all **`SC_THREAD`** and **`SC_METHOD`** processes have been established. SystemC 2.1 introduced the concept of dynamically spawned processes.

Dynamic processes are important for several reasons. At the top of the list is the ability to perform temporal checks such as those supported by PSL Sugar, Vera, and other verification languages. For instance, consider a bus protocol with split transactions and timing requirements. Once a request is issued, it is important to track the completion of that transaction for verification of that transaction. Since transactions may be split, each transaction will require a separate thread to monitor. Without dynamic process support, it would be necessary to pre-allocate a number of statically defined processes to accommodate the maximum number of possible outstanding requests.

## 7.2 sc_spawn

Let's look at the syntax and requirements that enable dynamic processes. First, to enable dynamic processes, it is necessary to use a pre-processor macro prior to the invocation of the SystemC header file. Figure 7.1 is one way to do this:

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
```

**Fig. 7.1** Syntax to enable dynamic threads

Other mechanisms involve the C++ compilation tools. For example, GNU gcc has a –D option (e.g., -DSC_INCLUDE_DYNAMIC_PROCESSES), which predefines the macro.

Next, declare the functions to be spawned as processes. Functions may be either normal functions or methods (i.e., member functions of a class). The dynamic facilities of SystemC allow for either **SC_THREAD** or **SC_METHOD** style processes. Unlike static processes, dynamic processes may have up to eight arguments and a return value. The return value will be provided via a reference variable in the actual spawn function. For example, consider the declarations in Fig. 7.2.

```
// Ordinary function declarations
void inject(void); // no args or return
int count_changes(sc_signal<int>& sig);

// Method function declarations
class TestChan : public sc_module {
  …
  bool Track(sc_signal<packet>& pkt);
  void Errors(int maxwarn, int maxerr);
  void Speed(void);
  …
};
```

**Fig. 7.2** Example functions used as dynamic processes

Having declared a function (possibly a member function) to be used as spawned process, you need to define the implementation and register the function with the kernel. You can register the dynamic processes within an **SC_THREAD** or with restrictions within an **SC_METHOD**. The basic syntax is shown in Fig. 7.3 and Fig. 7.4.

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    sc_bind(&funcName, ARGS…)//no return value
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    sc_bind(&methName, object, ARGS…)//no return
   ,processName
   ,spawnOptions
  );
```

**Fig. 7.3** Syntax to register dynamic processes with void return

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    &returnVar
   ,sc_bind(&funcName, ARGS…)
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    &returnVar
   ,sc_bind(&methodName, object, ARGS …)
   ,processName
   ,spawnOptions
  );
```

**Fig. 7.4** Syntax to register dynamic processes with return values

Note in the preceding that `object` is a reference to the calling module, and normally we just use the C++ keyword **`this`**, which refers to the calling object itself.

By default, arguments are passed by value. To pass by reference or by constant reference, a special syntax is required. This syntax is required to make **`sc_bind()`** practical.

```
sc_ref(var)  // reference
sc_cref(var) // constant reference
```

**Fig. 7.5** Syntax to pass process arguments by reference

The `processName` and `spawnOptions` are optional; however, if `spawnOptions` are used, then a `processName` is mandatory. All processes should have unique names. Fortunately, uniqueness of a process name includes the hierarchical instance as a prefix (i.e., **`name()`**). If a process spawns a process, then its name is used to prefix the spawned process name.

## 7.3 Spawn Options

Spawn options are determined by creating an **`sc_spawn_option`** object and then invoking one of several methods that set the options. Figure 7.6 is the syntax:

```
sc_spawn_option objname;
objname.spawn_method();// register as SC_METHOD
objname.dont_initialize();
objname.set_sensitivity(event_ptr);
objname.set_sensitivity(port_ptr);
objname.set_sensitivity(interface_ptr);
objname.set_sensitivity(event_finder_ptr);
objname.set_stack_size(value); // experts only!
```

**Fig. 7.6** Syntax to set spawn options

By default, spawned processes are thread processes. To specify a method process, you must call **`spawn_method()`** as shown above.

One last comment before we look at an example. The method **`sc_get_current_process_handle()`** may be used by the spawned process to reference the calling object. In particular, it may be useful to access **`name()`**.

## 7.4 A Spawned Process Example

That's a lot of syntax. Fortunately, you don't need to use all of it. Let's take a look at an example of the simplest case in Fig. 7.7. This example is an **`SC_THREAD`** that contains no parameters and returns no result. In other words, it looks like an **`SC_THREAD`** that just happens to be dynamically spawned. We highlight the important points.

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
…
void spawned_thread() {// This will be spawned
  cout << "INFO: spawned_thread "
       << sc_get_current_process_handle().name()
       << " @ " << sc_time_stamp() << endl;
  wait(10,SC_NS);
  cout << "INFO: Exiting" << endl;
}

void simple_spawn::main_thread() {
  wait(15,SC_NS);
  // Unused handle discarded
  sc_spawn(sc_bind(&spawned_thread));
  cout << "INFO: main_thread " << name()
       << " @ " << sc_time_stamp() << endl;
  wait(15,SC_NS);
  cout << "INFO: main_thread stopping "
       << " @ " << sc_time_stamp() << endl;
}
```

**Fig. 7.7** Example of a simple thread spawn

If you keep a handle on the spawned process, then it is also possible to await the termination of the process via the **sc_process_handle::terminated_event()** method. For example:

```
sc_process_handle h =
  sc_spawn(sc_bind(&spawned_thread));
// Do some work
…
// Wait for spawned thread to return
wait(h.terminated_event());
```

**Fig. 7.8** Example of waiting on a spawned process

Be careful not to **wait()** on an **SC_METHOD** process; currently, there is no way to terminate an **SC_METHOD**.[1]

An interesting observation about **sc_spawn** is that it may also be used within the constructor, and it may be used with the same member function multiple times as long as the process name is unique. This capability also means there is now a way to pass arguments to **SC_THREAD** and **SC_METHOD** as long as you are willing to use the longer syntax.

A dangerous aspect of spawned threads relates to the return value. If you pass a function or method that returns a value, it is critical that the referenced return location remains valid throughout the lifetime of the process. The result will be written without respect to whether the location is valid upon exit, possibly resulting in a really nasty bug.

The creation and management of dynamic processes is not for the faint of heart. On the other hand, learning to manage dynamic processes has great rewards. One of the simpler ways to manage dynamic processes is discussed in the next section on fork/join.

## 7.5 SC_FORK/SC_JOIN

Another use of dynamic threads is dynamic test configuration. This feature is exemplified with a verification strategy sometimes used by Verilog suites using fork/join. Although this technique does not let you create new modules or channels dynamically (because processes may choose to stimulate ports differently on the fly), you can reconfigure tests. Let's see how this might be done.

[1]However, see the last section of this chapter for a preview of upcoming features.

Consider the DUT in the following figure:

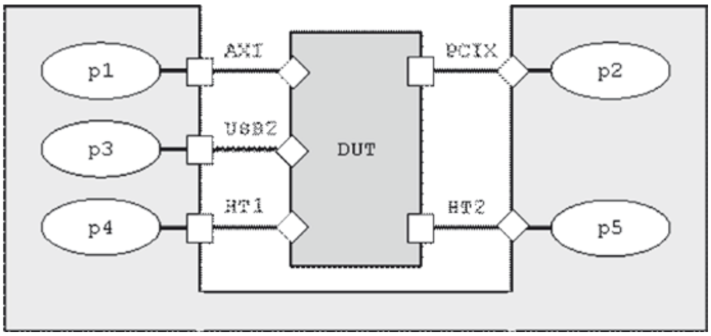


**Fig. 7.9** High-level testbench

For each interface (AXI, USB2, etc.), an independent process can be created either to send or receive information likely to be generated in a real system.

Using these processes and fork/join, a high-level test might look like Fig. 7.10 (Note: Syntax will be explained shortly.):

```
DataStream d1, d2;
SC_FORK
   sc_spawn(
     sc_bind(&dut::AXI_xmt,this,sc_ref(d1)), "p1")
  ,sc_spawn(
     sc_bind(&dut::PCIX_rcv,this,sc_ref(d1)),"p2")
  ,sc_spawn(
     sc_bind(&dut::USB2,this,sc_ref(d1)),     "p3")
  ,sc_spawn(
     sc_bind(&dut::HT1_xtm,this,sc_ref(d2)), "p4")
  ,sc_spawn(
     sc_bind(&dut::HT2_rcv,this,sc_ref(d2)), "p5")
SC_JOIN
```

**Fig. 7.10** Example of fork/join application

The syntax for the SystemC fork/join is shown in Fig. 7.11.

```
SC_FORK
  COMMA_SEPARATED_LIST_OF_SPAWNS
SC_JOIN
```

**Fig. 7.11** Syntax for fork/join

Let's look at an example that involves a number of syntax elements discussed thus far. First, let's inspect the header for this module in Fig. 7.12.

```
//FILE: Fork.h
SC_MODULE(Fork) {
  …
  sc_fifo<double> wheel_lf, wheel_rt;
  SC_CTOR(Fork);// Constructor
  // Declare processes to be used with fork/join
  void fork_thread();
  bool road_thread(sc_fifo<double>& which);
};
```

**Fig. 7.12** Example header for fork/join example

Notice that we pass a FIFO channel by reference so that `road_thread` can possibly access the FIFO channel. Passing ports or signals by reference would be a natural extension of this idea.

Now, let's inspect the rest of the code in Fig. 7.13.

```
//FILE: Fork.cpp
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
#include "Fork.h"
…
Fork::Fork(sc_module_name nm) //{{{
: sc_module(nm)
{
  SC_THREAD(fork_thread);
  …
}
void Fork::fork_thread() { //{{{
  bool lf_up, rt_up; // use for return values
  SC_FORK
    sc_spawn(
       &lf_up
      ,sc_bind(
          &Fork::road_thread
         ,this
         ,sc_ref(wheel_lf)
       )
      ,"lf" // process name
    )
    ,sc_spawn(
       &rt_up
      ,sc_bind(
          &Fork::road_thread
         ,this
         ,sc_ref(wheel_rt)
       )
      ,"rt" // process name
    )
  SC_JOIN
}
bool Fork::road_thread(sc_fifo<double>& which) { //
  // Do some work
  return (road > 0.0);
}
```

**Fig. 7.13** Example of fork/join

The full example may be found in the downloaded examples as `Fork`. This downloaded example also illustrates the use of **`sc_spawn`** instead of **`SC_THREAD`**. Using a capitalized word *fork* avoids collision with the Unix system call fork, which has nothing to do with SystemC's **`SC_FORK`**. Recall that SystemC is a cooperative multitasking system. You should not confuse Unix's fork facilities with these concepts.

## 7.6 Process Control Methods

The following information is being considered for addition to the SystemC standard. We include it here because we are fairly certain these extensions will be part of the standard within the lifetime of this book's publication. You may not be able to use these until you get access to OSCI version 2.3 or later. It is also possible that syntax could change slightly.

SystemC process control constructs are proposed as methods in the **`sc_process_handle`** class. Since spawning processes could occur in a hierarchical manner, there are options for the controls to affect either just the specified process, or the process and all of its descendants. This is accomplished with the enumeration shown in Fig. 7.14.

```
enum sc_descendant_inclusion_info {
  SC_NO_DESCENDANTS,
  SC_INCLUDE_DESCENDANTS
};
```

**Fig. 7.14** Specifying descendants

Given the above, there are nine control constructs as shown in Fig. 7.15. Each of these takes a descendants argument that defaults to **`SC_NO_DESCENDANTS`**.

```
// Add "& resume" to sensitivity while suspended
void sc_process_handle::suspend(descend);
void sc_process_handle::resume(descend);

// Ignore sensitivity while disabled
void sc_process_handle::disable(descend);
void sc_process_handle::enable(descend);

// Complete remove process
void sc_process_handle::kill(descend);

// Asynchronously restart a process
void sc_process_handle::reset(descend);

// Reset process on every resumption event
void sc_process_handle::sync_reset_on(descend);
void sc_process_handle::sync_reset_off(descend);

// Throw an exception in the specified process
template<typename T>
  void sc_process_handle::throw_it(
                          const T&,descend);
```

**Fig. 7.15** Process control constructs

**`Suspend`** and **`resume`** are used in situations where it is desirable to continue to collect events that the process is sensitive to even while suspended. This is likely to be used for high-level modeling. Keep in mind that the process might start evaluating immediately after resuming.

**`Disable`** and **`enable`** cause the process to ignore events in the sensitivity until re-enabled. For example, one might disable a process sensitive to the clock rather than resume it. Disabling is done because **`resume`** will start processing immediately if an event occurred while suspended. For clock synchronized processes **`resume`** is probably not desirable since **`resume`** might happen at a time not on the clock boundary.

The reset and synchronous reset methods will finally make it possible to deprecate the **`SC_CTHREAD`** process[2]. Resetting simplifies SystemC by reducing the number of process types. The idea is that a reset is issued whenever returning from a **`wait`** and the **`sync_reset_on`** is in effect.

Lastly, the **`throw_it`** method enables the concept of an interrupt. This of course requires use of the C++ exception handling mechanism **`try`**`{CODE}`**`catch`** `(TYPE){HANDLER}`.

Look in the downloadable examples for this edition of the book for more information.

## 7.7 Exercises

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

**Exercise 7.1:** Rewrite the *`turn_of_events`* example in Chapter 6 using dynamic simulations processes for *`turn_knob_thread`* and *`stop_signal_thread`*.

**Exercise 7.2:** Modify your dynamic simulation process version of the *`turn_of_events`* example in Chapter 6 so that the turn_knob_thread and stop_signal_thread do not rely on the event names *`signal_stop`*, *`signal_off`*, *`stop_indicator_on`*, and *`stop_indicator_off`*. (Hint: Pass a reference to the appropriate events to the *`turn_know_thread`* and the *`stop_signal_thread`* simulation processes.)

**Exercise 7.3:** As an advanced exercise, see if you can devise a classical software interrupt handler for a hardware environment that has two interrupts, a 1 us timer, and an incoming data packet. Assume data arrives randomly with separations of 500 ns to 2 us. This exercise is as much an exercise in planning as it is an exercise in using the syntax. Feel free to use the process controls if you have access to a SystemC version that supports it.

[2] **`SC_CTHREAD`** is discussed in a later chapter on Additional Topics.