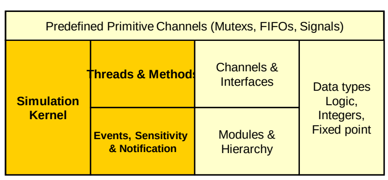
Plan

Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique



Concurrency

- Introduction
- Threads
- Methods
- Clocked Threads
- Dynamic Processes

Processes & Events

- SystemC uses processes to model concurrency
  - based on event-driven simulator (sc_event)
  - concurrency is NOT true concurrent execution
  - the concurrency is NOT preemptive
- 3 types of processes
  - SC_THREAD
  - SC_METHOD
  - SC_CTHREAD (used by behavioral synthesis tools)

Graphical notations

sc_event → process_name : SC_METHOD or SC_THREAD or SC_CTHREAD

Concurrency is NOT preemptive Example

```
SC_MODULE(simple_process)
{
    //  sc_out< ...
    enum defstate {IDLE, RUN, STOP};
    defstate state;

    SC_CTOR(simple_process) : state(IDLE)
    {
        SC_THREAD(p1_thread);
    }
    void p1_thread(void)
    {
        while (true)
        {
            wait(99, SC_MS);
            switch (state)
            {
                case IDLE : cout << "idle : ";
                    // ...
                    state = RUN;
                    break;
                case RUN : cout << "run : ";
                    // ...
                    state = STOP;
                    break;
                case STOP : cout << "stop : ";
                    // ...
                    state = IDLE;
                    break;
            }
            cout << sc_time_stamp() << endl;
        }
    }
};
```

releases control to kernel

test_concurrency

```
int sc_main(int argc, char* argv[])
{
    simple_process my_instance("my_instance");
    sc_start(1, SC_SEC);
    return 0;
}
```

1 sec of simulation

```
idle : 0 s
run : 0 s
stop : 0 s
idle : 0 s
run : 0 s
stop : 0 s
idle : 0 s
run : 0 s
…
```

time don't change !!

CTRL C to stop the simulation !!

solution: add wait(99, SC_MS) to change process

```
idle : 99 ms
run : 198 ms
stop : 297 ms
idle : 396 ms
run : 495 ms
stop : 594 ms
idle : 693 ms
run : 792 ms
stop : 891 ms
idle : 990 ms
Press any key to continue
```

Triggering Events : notify()

- Events are key to an event-driven simulator
- Events are no value, no duration
- Events happen at a single point in time
- Processes wait for event
  - dynamic sensitivity
  - static sensitivity

sc_event class

Declaration sc_event ev;

Methods & Operators

```
void notify();
void notify( const sc_time& );
void notify( double , sc_time_unit );
void cancel();

sc_event_or_list& operator| ( const sc_event& ) const;
sc_event_and_list& operator& ( const sc_event& ) const;
```

The classes *sc_event_and_list* and *sc_event_or_list* provide the & and | operators used to construct the event lists passed as arguments to the functions wait (SC_THREAD) and next_trigger (SC_METHOD)

Examples

```
sc_event action;
sc_time now(sc_time_stamp());

// immediately action
action.notify();
// schedule new action for 20 ms from now
action.notify(20, SC_MS);
// reschedule action for 2 ns from now
action.notify(2, SC_NS);
// reschedule action for next delta cycle
action.notify(SC_ZERO_TIME);
// cancel action entirely
action.cancel();
```


Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

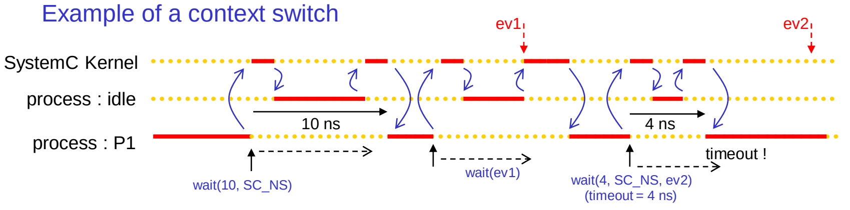
Concurrency

- Introduction
- Threads
- Methods
- Clocked Threads
- Dynamic Processes

Dynamic Sensitivity

- SC_THREAD processes rely on the wait() method to suspend their execution
- Wait() method is supplied by the sc_module class
- When wait() executes, the state of the current thread is saved (context switch)

Example of a context switch



Wait methods

```
void wait();
void wait( const sc_event& );
void wait( sc_event_or_list& );
void wait( sc_event_and_list& );
void wait( const sc_time& );
void wait( double v , sc_time_unit tu );
```

wait(ev1 | ev2)
wait(ev1 & ev2)

Wait methods with Timeout

```
void wait( const sc_time& , const sc_event& );
void wait( double , sc_time_unit , const sc_event& );
void wait( const sc_time& , sc_event_or_list& );
void wait( double , sc_time_unit , sc_event_or_list& );
void wait( const sc_time& , const sc_event_and_list& );
void wait( double , sc_time_unit , sc_event_and_list& );
```


Wait Method
Example 1 – Header File

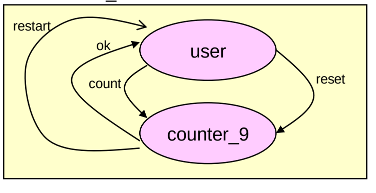
archi_ex.h

thread_example1

```
#ifndef ARCHI_EX_H
#define ARCHI_EX_H

#include <iostream>
#include <systemc.h>

SC_MODULE(archi_ex)
{
    sc_event count,ok, restart, reset;
    enum defmess {RESTART, OK, RESET, COUNT};
    defmess from_usermess, from_countermess;
    int n;

    SC_CTOR(archi_ex) : n(0)
    {
        SC_THREAD(user_thread);
        SC_THREAD(counter_9_thread);
    }

    void user_thread(void);
    void counter_9_thread(void);
};

#endif
```

Declaration of SC_THREAD

archi_ex



Wait Method
Example 1 – CPP File

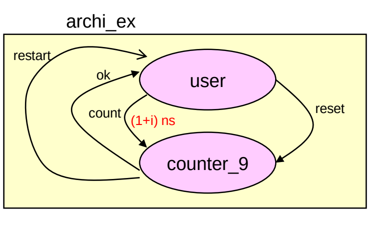
archi_ex.cpp *thread_example1*

```
#include "archi_ex.h"

void archi_ex::user_thread(void)
{
    for (int i = 0; i <= 5; i++)
    {
        cout << sc_time_stamp() << " user: notified count event (" << i <<")" << endl;
        from_usermess = COUNT;
        count.notify(1+i, SC_NS);
        cout << sc_time_stamp() << " user: waiting ok or restart events" << endl;
        wait(ok | restart);
        if (from_countermess == OK)
            cout << sc_time_stamp() << " user: ok !" << endl;
        else
            cout << sc_time_stamp() << " user: restart !" << endl;
    }

    cout << "-----------------------------------------------"  << endl;
    wait(20, SC_NS);
    cout << sc_time_stamp() << " user: reset" << endl;
    from_usermess = RESET;
    reset.notify();

    cout << sc_time_stamp() << " user: waiting restart event" << endl;
    wait(restart);
}
```

```
void archi_ex::counter_9_thread(void)
{
    while (true)
    {
        cout << sc_time_stamp() << " counter: waiting ";
        cout << "count or reset event" << endl;
        wait(count | reset);
        cout << sc_time_stamp() << " counter: receiving"
        cout << "count or reset event" << endl;
        if (from_usermess == RESET)
            n = 0;
        else
            n += 1 % 10;

        if (n == 0)
        {
            from_countermess = RESTART;
            restart.notify();
        }
        else
        {
            from_countermess = OK;
            ok.notify();
        }
    }
}
```


Wait Method
Example 1 - Execution

thread_example1




```
SystemC 2.1_oct_12_04.beta --- Apr 10 2005 17:53:24
        Copyright (c) 1996-2004 by all Contributors
                ALL RIGHTS RESERVED
Starting simulation
0 s user: notified count event (0)
0 s user: waiting ok or restart events
0 s counter: waiting count or reset event
1 ns counter: receiving count or reset event
1 ns counter: waiting count or reset event
1 ns user: ok !
1 ns user: notified count event (1)
1 ns user: waiting ok or restart events
3 ns counter: receiving count or reset event
3 ns counter: waiting count or reset event
3 ns user: ok !
3 ns user: notified count event (2)
3 ns user: waiting ok or restart events
6 ns counter: receiving count or reset event
6 ns counter: waiting count or reset event
6 ns user: ok !
6 ns user: notified count event (3)
6 ns user: waiting ok or restart events
10 ns counter: receiving count or reset event
10 ns counter: waiting count or reset event
10 ns user: ok !
10 ns user: notified count event (4)
10 ns user: waiting ok or restart events
15 ns counter: receiving count or reset event
15 ns counter: waiting count or reset event
15 ns user: ok !
15 ns user: notified count event (5)
15 ns user: waiting ok or restart events
21 ns counter: receiving count or reset event
21 ns counter: waiting count or reset event
21 ns user: ok !
-----------------------------------------------
41 ns user: reset
41 ns user: waiting restart event
41 ns counter: receiving count or reset event
41 ns counter: waiting count or reset event
Exiting simulation
Press any key to continue
```

Wait Method Example 2

```
const sc_time t1 = sc_time(10, SC_NS);
const sc_time t2 = sc_time(5, SC_NS);
const sc_time t3 = sc_time(15, SC_NS);

void test::Process_A()
{
    cout << sc_time_stamp() << " Process_A : State 1" << endl;
    wait(t1);
    cout << sc_time_stamp() << " Process_A : State 2" << endl;
    wait(t2);
    cout << sc_time_stamp() << " Process_A : State 3" << endl;
    wait(t3);
}

void test::Process_B()
{
    cout << sc_time_stamp() << " Process_B : State 1" << endl;
    wait(t1);
    cout << sc_time_stamp() << " Process_B : State 2" << endl;
    wait(t2);
    cout << sc_time_stamp() << " Process_B : State 3" << endl;
    wait(t3);
}

void test::Process_C()
{
    cout << sc_time_stamp() << " Process_C : State 1" << endl;
    wait(t1);
    cout << sc_time_stamp() << " Process_C : State 2" << endl;
    wait(t2);
    cout << sc_time_stamp() << " Process_C : State 3" << endl;
    wait(t3);
}

void test::Process_D()
{
    cout << sc_time_stamp() << " Process_D : State 1" << endl;
    wait(t1);
    cout << sc_time_stamp() << " Process_D : State 2" << endl;
    wait(SC_ZERO_TIME);
    cout << sc_time_stamp() << " Process_D : State 3" << endl;
    wait(t3);
}
```

thread_example2

```
Starting simulation
0 s Process_A : State 1
0 s Process_B : State 1
0 s Process_C : State 1
0 s Process_D : State 1
10 ns Process_A : State 2
10 ns Process_D : State 2   ←
10 ns Process_C : State 2
10 ns Process_B : State 2
10 ns Process_D : State 3   ←
15 ns Process_A : State 3
15 ns Process_B : State 3
15 ns Process_C : State 3
Exiting simulation
Press any key to continue
```


Static Sensitivity

- SystemC provides another type of sensitivity called Static Sensitivity
- establishes during elaboration phase
- static sensitivity parameters cannot be changed
- possible to override (dynamic sensitivity)

```
SC_CTOR(M)
{
    SC_THREAD(test_thread);
      sensitive << event1 << event2 …;
    or
      sensitive(event1, event2, …);
}

void test_thread()
{
    …
}
```

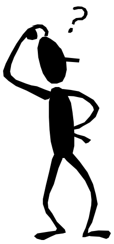
"<<" streaming style

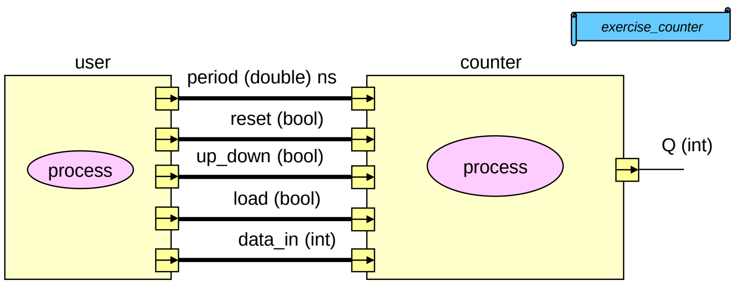
functional style

```
SC_MODULE(Mod)
{
  sc_signal<bool> A, B, C, D, E;
  SC_CTOR(Mod)
  {
      sensitive << A;      // Has no effect. Poor coding style
    SC_THREAD(M_thread);
      sensitive << B << C; // Thread process M is made sensitive to B and C.
      f();                 // Method process M is made sensitive to D.
      sensitive(E);        // Method process M is made sensitive to E
  }

void f()
{
  sensitive << D;
 }

void M_thread();
…
};
```

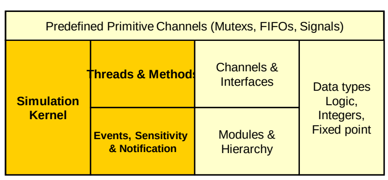
Training

- Write a counter modulo 10
  - Period
  - up/down (up:true, down:false)
  - load
  - asynchronous reset





(tip: use wait method with timeout)



Concurrency

- Introduction
- Threads
- Methods
- Clocked Threads
- Dynamic Processes

SC_METHOD

- Simpler than the SC_THREAD
- More efficient than SC_THREAD
- More difficult to use for some modeling style
- Difference SC_THREAD / SC_METHOD ?

SC_METHOD(p1_method)

execute code and return

```
void p1_method()          local variable (3)
{
    int a = 0;
    while (a < 10)        Error (2)
    {
      wait(10, SC_NS);  // RUNTIME ERROR
      cout << "a = " << a << endl;
       a += 1;
    }
}
```

SC_THREAD(p1_thread)

```
void p1_thread()          persistent local variables (3)
{
    int a = 0;
    while (a < 10)
    {
      wait(10, SC_NS); (2)
      cout << "a = " << a << endl;
      a += 1;
    }
}
```

exit after 10 x 10 ns = 100 ns

Dynamic Sensitivity

- SC_METHOD processes rely on the next_trigger() method to trig their execution

void **next_trigger**(); → Re-establish static sensitivity

void **next_trigger**(const sc_event&);
void **next_trigger**(*sc_event_or_list&*); → any of these event
void **next_trigger**(*sc_event_and_list&*); → all of these event required

void **next_trigger**(const sc_time&); → next_trigger(t1);
void **next_trigger**(double v , sc_time_unit tu); → next_trigger(25, SC_MS);
void **next_trigger**(const sc_time& , const sc_event&); → trig after the given time OR notified event

void **next_trigger**(double , sc_time_unit , const sc_event&);
void **next_trigger**(const sc_time& , *sc_event_or_list&*);
void **next_trigger**(double , sc_time_unit , *sc_event_or_list&*);
void **next_trigger**(const sc_time& , const *sc_event_and_list&*);
void **next_trigger**(double , sc_time_unit , *sc_event_and_list&*);

} same methods with time out

Static Sensitivity

- Same as SC_THREAD

streaming style

```
SC_CTOR(M)
{
    SC_METHOD(test_method);
      sensitive << event1 << event2 …;
}

void test_method()
{
    …
}
```

functional style

```
SC_CTOR(M)
{
    SC_METHOD(test_method);
      sensitive(event1, event2, …);
}

void test_method()
{
    …
}
```

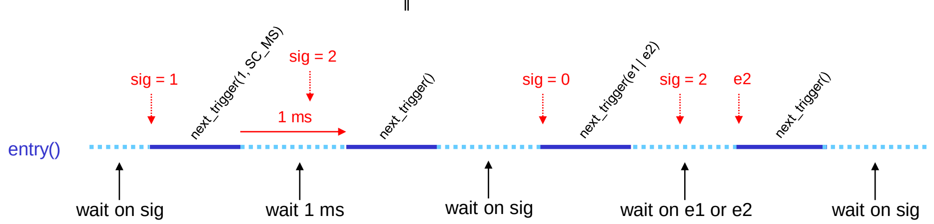
don't remember !

next_trigger() (without argument) re-establishes the static sensitivity

Example

```
SC_MODULE(M)
{
    sc_signal<int> sig;
    sc_event e1, e2;

    SC_CTOR(M)
    {

    SC_METHOD(entry_method);
        sensitive << sig;
    }
```

```
void entry_method() // Run first at initialization.
{
    if (sig.read() == 0)
        next_trigger(e1 | e2); // Trigger on event e1 or event e2 next time
    else if (sig.read() == 1)
        next_trigger(1, SC_MS); // Time-out after 1 millisecond.
    else
        next_trigger(); // Trigger on signal sig next time.
    }
    ...
};
```



Don't initialize

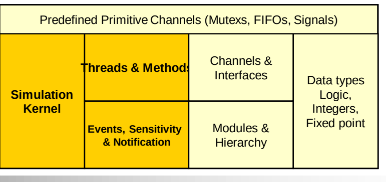
- Sometimes, it becomes necessary to specify some processes that are not initialized
- use dont_initialize() method

```
SC_MODULE(Mod)
{
      sc_signal<bool> B, C;
      SC_CTOR(Mod)
      {
            SC_METHOD(M_method);
                  sensitive << B << C;     // Thread process M is made sensitive to B and C.
                  dont_initialize();
      }

      void M_method()
      {
      }

      ...
};
```

Method M will not be initialized

Ecole polytechnique de l'université de Nice-Sophia Antipolis
Département Electronique

Concurrency



- Introduction
- Threads
- Methods
- Clocked Threads
- Dynamic Processes

Predefined Process : sc_clock (1/3)

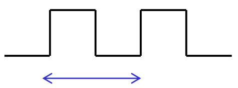
- Clocks represent a common hardware behavior
- TLM Level
  - Bus Cycle Accurate model (BCA)
  - Cycle Accurate model (BA)
- RTL Level

Constructors

```
sc_clock( const char* name_, const sc_time& period, double duty_cycle_ = 0.5, const sc_time& start_time = SC_ZERO_TIME,
          bool posedge_first_ = true );
sc_clock( const char* name_, double period_v_, sc_time_unit period_tu_, double duty_cycle_ = 0.5 );
sc_clock( const char* name_, double period_v_, sc_time_unit period_tu_, double duty_cycle_, double start_time_v_,
          sc_time_unit start_time_tu_, bool posedge_first_ = true );
```

default value   sc_clock clk1("clk1")

clk1



Duty cycle = 50%
StartTime = 0 sec
Posedge_first = true

Methods

```
const sc_time& period() const;
double duty_cycle() const;
const sc_time& start_time() const;
bool posedge_first() const;
```


Predefined Process : sc_clock (2/3)

- Clocks can slow simulation
  - add many events
  - much resulting activity
  - prefer wait

one event ! (fast)

```
wait(N * t_PERIOD);
```

many events ! (slow)

```
for (i=1, i<=N; i++)
    wait(clk->posedge_event());
```

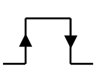
- Connect clock to module
  - use "sc_in_clk" or sc_in<bool>

```
SC_MODULE(M)
{
    sc_in<bool> clk_in;
    …
```

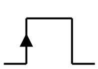
equals

```
SC_MODULE(M)
{
    sc_in_clk clk_in;
    …
```

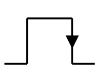
- Sensitive pos/neg edge



```
SC_METHOD(counter);
    sensitive << clk;
```



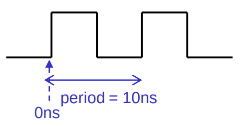
```
SC_METHOD(counter);
    sensitive_pos << clk;
    OR
    sensitive << clk.pos();
```



```
SC_METHOD(counter);
    sensitive_neg << clk;
    OR
    sensitive << clk.neg();
```

Predefined Process : sc_clock (3/3) Examples

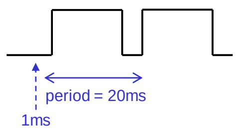
clk1



sc_clock("clk1", 10, SC_NS);

Period = 10 ns
Duty cycle = 50% (default)
StartTime = 0 s (default)
Posedge_first = true (default)

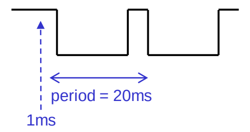
clk2



sc_clock("clk2", 20, SC_MS, 0.75, 1, SC_MS, true);

Period = 20 ms
Duty cycle = 75%
StartTime = 1 ms
Posedge_first = true

clk3



sc_clock("clk3", 20, SC_MS, 0.25, 1, SC_MS, false);

Period = 20 ms
Duty cycle = 25%
StartTime = 1 ms
Posedge_first = false

Counter Example

```
int sc_main(int argc, char* argv[])
{
      sc_clock clk1("clk1", 10, SC_MS);  // Period = 10 ms
      counter my_counter("counter1");
            my_counter.clk(clk1);

      sc_start(200, SC_MS);
      return 0;
}
```

```
SC_MODULE(counter)
{
   sc_in_clk clk;
   int count;
   SC_CTOR(counter) : count(0)
   {
      SC_METHOD(counter_method);
         sensitive << clk.pos();
   }
   void counter_method()
   {
      cout << "count = " << count++ << endl;
   }
};
```


SC_CTHREAD

- Popular for behavioral synthesis tools
- Triggered by clock (synchronous thread)
- Reset possible

cthread_systemc2_1

sc_event_finder type pos() or neg()

asynchronous reset

normal operation

```
SC_MODULE(counter)
{
   sc_in<bool> reset;
   sc_in_clk clk;
   int count;
   SC_CTOR(counter) : count(0)
   {
      SC_CTHREAD(counter_p_cthread, clk.pos());
         reset_signal_is(reset, true);
   }
   void counter_p_cthread()
   {
      if (reset->read() == true)
      {
         cout << sc_time_stamp() << " : ";
         cout << "RESET ..." << endl;
         count = 0;
      }
      while (true)
      {
         cout << sc_time_stamp() << " : ";
         cout << "count = " << count << endl;
         count++;
         wait (SC_ZERO_TIME);
      }
   }
};
```

```
int sc_main(int argc, char* argv[])
{
   sc_clock clk1("clk1", 10, SC_MS);  // Period = 10 ms
   sc_signal<bool> rst;
   counter my_counter("counter1");
         my_counter.reset(rst);
         my_counter.clk(clk1);
   rst .write(true);
   sc_start(1, SC_MS);
   rst= false;
   sc_start(100, SC_MS);
   rst = true;
   sc_start(12, SC_MS);
   rst = false;
   sc_start(30, SC_MS);

  return 0;
}
```

```
0 s : RESET ...
0 s : count = 0
10 ms : count = 1
20 ms : count = 2
30 ms : count = 3
40 ms : count = 4
50 ms : count = 5
60 ms : count = 6
70 ms : count = 7
80 ms : count = 8
90 ms : count = 9
100 ms : count = 10
110 ms : RESET ...
110 ms : count = 0
120 ms : count = 1
130 ms : count = 2
140 ms : count = 3
```

SC_CTHREAD : watching()

- Don't use the watching() method, it's deprecated !
  - Macro: W_BEGIN, W_DO, W_ESCAPE, W_END are also deprecated
  - wait_until() method is deprecated

cthread_systemc2_1

New Code !! SystemC 2.1

```
SC_MODULE(counter)
{
  …
  SC_CTOR(counter) : count(0)
  {
    SC_CTHREAD(counter_p_cthread, clk.pos());
    reset_signal_is(reset, true);
  }
  void counter_p_cthread()
  {
    if (reset->read() == true)
    {
      cout << sc_time_stamp() << " : ";
      cout << "RESET ..." << endl;
      count = 0;
    }
    while (true)
    {
      cout << sc_time_stamp() << " : ";
      cout << "count = " << count << endl;
      count++;
      wait (SC_ZERO_TIME);
    }
  }
};
```

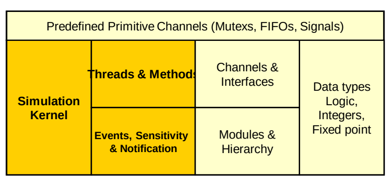
cthread_systemc2_0_1

Deprecated Code !!

```
SC_MODULE(counter)
{
  …
  SC_CTOR(counter) : count(0)
  {
    SC_CTHREAD(counter_p_cthread, clk.pos());
    watching(reset.delayed());
  }
  void counter_p_cthread()
  {
    while (true)
    {
      W_BEGIN
        watching(reset->delayed());
      W_DO
        cout << sc_time_stamp() << " : ";
        cout << "count = " << count << endl;
        count++;
      W_ESCAPE
        if (reset->read() == true)
        {
          cout << sc_time_stamp() << " : ";
          cout << "RESET ..." << endl;
          count = 0;
        }
      W_END
      wait (SC_ZERO_TIME);
    }
  }
};
```


Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique



Concurrency

- Introduction
- Threads
- Methods
- Clocked Threads
- Dynamic Processes

Why dynamic threads ?

- Ability to perform temporal checks
  - PSL, Sugar, Vera …
  - Bus protocol
    - split transactions and timing requirements
    - track the completion of that transaction from a verification point of view
    - each transaction will require a separate thread to monitor
- Modeling software tasks
  - Some tasks are dynamic
    - creation
    - running
    - killing
- Reconfigurable hardware
  - Some parts of a SoC have a FPGA areas
  - Modeling hardware tasks like software tasks
- Using SystemC 2.1

Declaration

Creation of a process : sc_spawn

```
template <typename T>
sc_process_handle sc_spawn(T object , const char* name_p = 0 , const sc_spawn_options* opt_p = 0 );

template <typename T>
sc_process_handle sc_spawn(typename T::result_type* r_p ,T object ,const char* name_p = 0 ,
                           const sc_spawn_options* opt_p = 0 );
```

sc_process_handle() class

hierarchical name of the underlying process instance.

```
bool valid() const;
const char* name() const;
sc_curr_proc_kind proc_kind() const;
const std::vector<sc_object*>& get_child_objects() const;
sc_object* get_parent_object() const;
bool dynamic() const;
bool terminated() const;
```

sc_spawn_options () class

```
void spawn_method();
void dont_initialize();
void set_stack_size( int sz);

void set_sensitivity( const sc_event* );
void set_sensitivity( sc_port_base* );
void set_sensitivity( sc_interface* );
void set_sensitivity( sc_event_finder* );
```

```
sc_process_handle* m_owner = sc_get_current_process_handle();
```

Use SystemC 2.1 October 2004 !

sc_process_b, sc_get_curr_process_handle() : (SystemC 2.1 October 2004)
sc_process_handle, sc_get_current_process_handle (SystemC 2.1 October 2005)

Example (1/2)

dynamic_threads

Don't Forgotten !

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

int spawned_thread();
int spawned_method();
int h(int a, int &b, const int& c);

SC_MODULE(simple_spawn)
{
   sc_in_clk clk;

   SC_CTOR(simple_spawn)
   {
      SC_THREAD(main_thread);
   }

  // Process declarations
  void main_thread(void);

  // Process Member Function
  void g();
};
```

Dynamic Non Member Processes

Static Member Process

Dynamic Member Process

```
struct Functor
{
   typedef int result_type;
   result_type operator() ();
};
Functor::result_type Functor::operator() ()
{
   return spawned_thread();
}

int spawned_thread()
{
   cout << sc_time_stamp() << " : INFO: spawned_thread() Starting "  << endl;
   wait(70,SC_NS);
   cout << sc_time_stamp() << " : INFO: spawned_thread() Exiting "  << endl;
   return 0;
}

int spawned_method()
{
   cout << sc_time_stamp() << " : INFO: spawned_method() Starting "  << endl;
   return 0;
}

int h(int a, int &b, const int& c)
{
   cout << sc_time_stamp() << " : INFO: h() Starting "  << endl;
   b = a + 1;
   return 0;
}

void simple_spawn::g()
{
   cout << sc_time_stamp() << " : INFO: g() Starting " << endl;
}
```

to catch return value

Example (2/2)

```
void simple_spawn::main_thread()
{
   // Spawn a function without arguments and discard any return value.
   cout << sc_time_stamp() << " : Create 1 : spawned_thread" << endl;
   sc_spawn(&spawned_thread);

   // Spawn a similar process and create a process handle.
   cout << sc_time_stamp() << " : Create 2 : spawned_thread" << endl;
   sc_process_handle handle = sc_spawn(&spawned_thread);
   Functor fr;
   int ret;
   sc_spawn(&ret, fr); // Spawn a function object and catch the return value.

   // Spawn a method process named "f1", sensitive to sig, not initialized.
   cout << sc_time_stamp() << " : Create 1 : spawned_method" << endl;
   sc_spawn_options opt;
      opt.spawn_method();
      opt.set_sensitivity(&clk);
      opt.dont_initialize();
   sc_spawn(spawned_method, "g1", &opt);

   // Spawn a similar process named "f2" and catch the return value.
   cout << sc_time_stamp() << " : Create 4 : spawned_thread" << endl;
   sc_spawn_options opt2;
      opt2.set_sensitivity(&clk);

   sc_spawn(&ret, fr, "f2", &opt2);

   // Spawn a member function using Boost bind.
   cout << sc_time_stamp() << " : Create 2 : g()" << endl;
   sc_spawn(sc_bind(&simple_spawn::g, this));

   // Spawn a function using Boost bind, pass arguments
   // and catch the return value.
   cout << sc_time_stamp() << " : Create 1 : h()" << endl;
   int A = 0, B, C;
   sc_spawn(&ret, sc_bind(&h, A, sc_ref(B), sc_cref(C)));
   wait(500, SC_NS);
}
```

using boost library

```
START SIMULATION ...
0 s : Create 1 : spawned_thread
0 s : Create 2 : spawned_thread
0 s : Create 1 : spawned_method
0 s : Create 4 : spawned_thread
0 s : Create 2 : g()
0 s : Create 1 : h()
0 s : INFO: spawned_thread() Starting
0 s : INFO: spawned_thread() Starting
0 s : INFO: spawned_thread() Starting
0 s : INFO: g() Starting
0 s : INFO: h() Starting
0 s : INFO: spawned_method() Starting
0 s : INFO: spawned_thread() Starting
50 ns : INFO: spawned_method() Starting
70 ns : INFO: spawned_thread() Exiting
70 ns : INFO: spawned_thread() Exiting
70 ns : INFO: spawned_thread() Exiting
70 ns : INFO: spawned_thread() Exiting
100 ns : INFO: spawned_method() Starting
150 ns : INFO: spawned_method() Starting
200 ns : INFO: spawned_method() Starting
250 ns : INFO: spawned_method() Starting
STOP SIMULATION ...
```

clk period = 100 ns
simulation = 300 ns

FORK / JOIN Macros

- The spawned process instances shall be thread processes, No Method process !
- Control leaves the fork-join construct when all the spawned process instances have terminated

```
SC_MODULE(M)
{
    SC_CTOR(M)
    {
        C_THREAD(fork_thread);
    }
    void fork_thread()
    {
        SC_FORK
            sc_spawn( "p1" ) ,
            sc_spawn("p2" ) ,
            sc_spawn("p3" )
        SC_JOIN
    }
};
```

comma