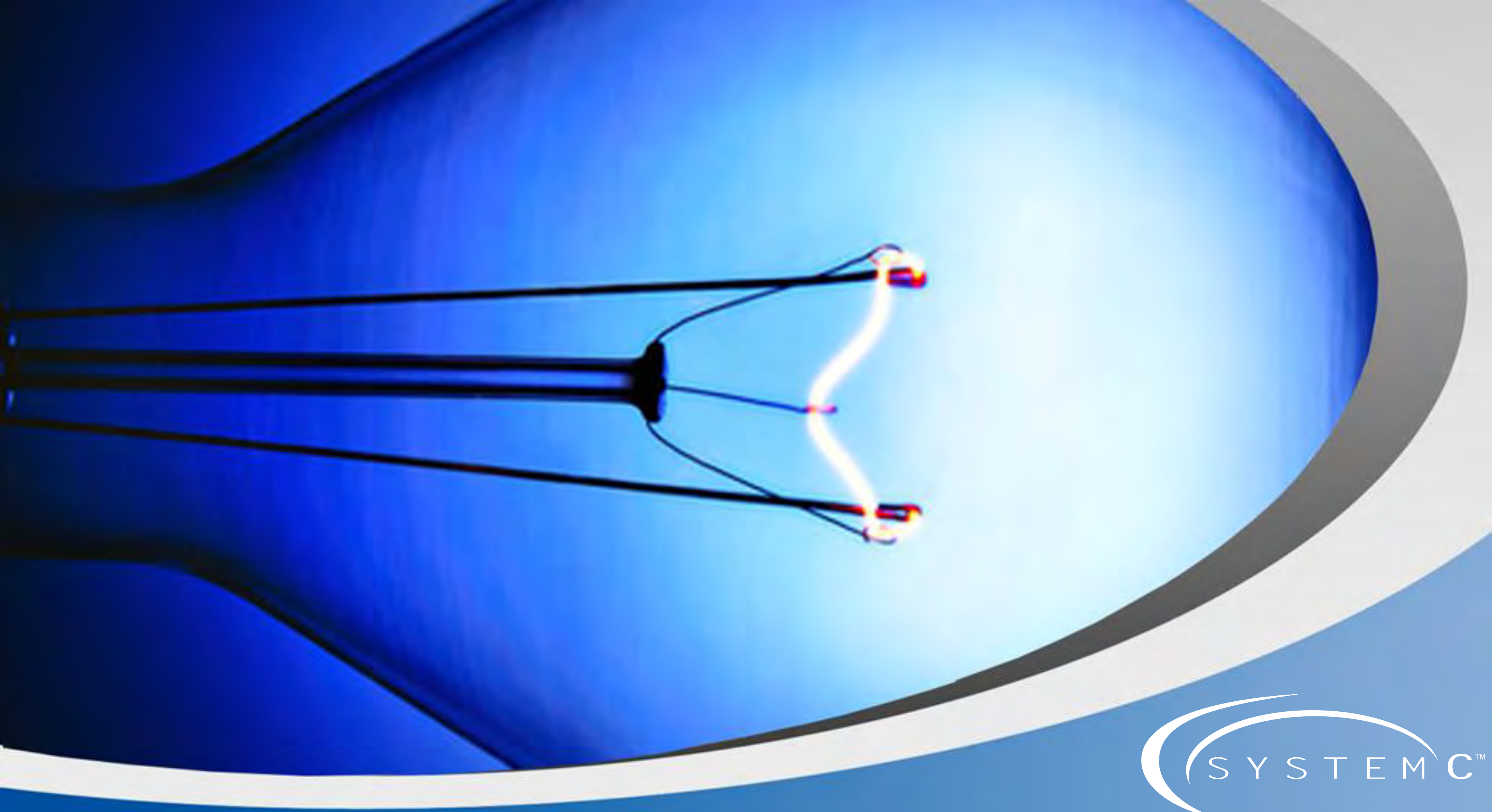


New Features of IEEE Std 1666-2011 SystemC

John Aynsley, Doulos

Introduction

This presentation briefly describes all of the significant new features introduced in IEEE Std 1666-2011, the SystemC Language Reference Manual, and implemented in the Accellera Systems Initiative proof-of-concept simulator version 2.3.x

This presentation was first given at DVCon, San Jose, in February 2012.

John Aynsley, Doulos, 9-May-2012

Contents

Process Control

- suspend
- resume
- disable
- enable
- sync_reset_on
- sync_reset_off
- reset
- kill
- throw_it

- reset_event
- sc_unwind_exception
- sc_is_unwinding

- reset_signal_is
- async_reset_signal_is

Framework for Examples

```
struct M: sc_module
{
  M(sc_module_name n)
  {
    SC_THREAD(calling);
    SC_THREAD(target);
  }

  void calling()
  {
    ...
  }

  void target()
  {
    ...
  }

  SC_HAS_PROCESS(M);
};
```

```
int sc_main(int argc, char* argv[])
{
  M m("m");
  sc_start(500, SC_NS);
  return 0;
}
```

Events

```
M(sc_module_name n)
{
  SC_THREAD(calling);
  SC_THREAD(target);
}
```

```
sc_event ev;
```

```
void calling()
{
  ev.notify(5, SC_NS);
}
```

```
void target()
{
  while (1)
  {
    wait(ev);
    cout << sc_time_stamp();
  }
}
```

5

Process Handles

```
M(sc_module_name n)
{
  SC_THREAD(calling);
  SC_THREAD(target);
    t = sc_get_current_process_handle();
}
```

```
sc_process_handle t;
```

```
void calling()
{
  assert( t.valid() );
  cout << t.name();
  cout << t.proc_kind();
}
```

```
m.target  2
```

```
void target()
{
  while (1)
  {
    wait(100, SC_NS);
    cout << sc_time_stamp();
  }
}
```

```
100 200 300 400
```

suspend & resume

```
void calling()
{
  wait(20, SC_NS);
  t.suspend();          at 20
  wait(20, SC_NS);
  t.resume();           at 40

  wait(110, SC_NS);
  t.suspend();          at 150
  wait(200, SC_NS);
  t.resume();           at 350
}
```

```
void target()
{
  while (1)
  {
    wait(100, SC_NS);
    cout << sc_time_stamp();
  }
}
```

100 350 450

suspend & resume

```
void calling()
{
  wait(20, SC_NS);
  t.suspend();
  wait(20, SC_NS);
  t.resume();

  wait(110, SC_NS);
  t.suspend();
  wait(200, SC_NS);
  t.resume();
}
```

at 20

at 40

at 150

at 350

```
void tick() {
  while (1) {
    wait(100, SC_NS);
    ev.notify();
  }
}
```

```
void target()
{
  while (1)
  {
    wait(ev);
    cout << sc_time_stamp();
  }
}
```

100 350 450

disable & enable

```
void calling()
{
  wait(20, SC_NS);
  t.disable();          at 20
  wait(20, SC_NS);
  t.enable();           at 40

  wait(110, SC_NS);
  t.disable();          at 150
  wait(200, SC_NS);
  t.enable();           at 350
}
```

```
SC_THREAD(target);
  sensitive << clock.pos();
```

```
void target()
{
  while (1)
  {
    wait();
    cout << sc_time_stamp();
  }
}
```

100 400

accellera SYSTEMS INITIATIVE

suspend versus disable

```
void calling()
{
   ...
   t.suspend();
   ...
   t.resume();
   ...
}
```

- Clamps down process until resumed
- Still sees incoming events & time-outs
- Unsuitable for clocked target processes
- Building abstract schedulers

```
void calling()
{
   ...
   t.disable();
   ...
   t.enable();
   ...
}
```

- Disconnects sensitivity
- Runnable process remains runnable
- Suitable for clocked targets
- Abstract clock gating

An Abstract Scheduler

```
M(sc_module_name n)
{
  SC_THREAD(scheduler);
  for (int i = 0; i < n; i++)
    task_handle[i] = sc_spawn(sc_bind(&M::task, this , i));
}
```

```
sc_process_handle task_handle[n];
```

```
void scheduler() {
  for (int i = 0; i < n; i++)
    task_handle[i].suspend();
  while (1)
    for (int i = 0; i < n; i++) {
      task_handle[i].resume();
      wait(timeslot);
      task_handle[i].suspend();
    }
}
```

```
void task(int number)
{
  while (1)
  {
    ...
    sc_time busy_for;
    wait(busy_for);
    ...
  }
}
```

Abstract Clock Gating

```
M(sc_module_name n)
{
  SC_CTHREAD(calling, clk.pos());
  SC_CTHREAD(target,  clk.pos());
    t = sc_get_current_process_handle();
}
```

```
void calling()
{
  while (1)
  {
    wait();
    t.disable();

    wait();
    t.enable();

    wait();
  }
}
```

q = 0

q = 1

q = 1

```
int q;
```

```
void target()
{
  int q = 0;
  while (1)
  {
    wait();
    ++q;
  }
}
```

Scheduling

```
void calling1()
{
  t.suspend();
}
```

Target suspended immediately

```
void target()
{
  while (1)
  {
    wait(ev);
    ...
  }
}
```

```
void calling2()
{
  t.resume();
}
```

Target runnable immediately, may run in current eval phase

```
void calling3()
{
  t.disable();
}
```

Sensitivity disconnected immediately, target may run in current eval phase

```
void calling4()
{
  t.enable();
}
```

Sensitivity reconnected immediately, never itself causes target to run

Self-control

```
M(sc_module_name n)
{
  SC_THREAD(thread_proc);
    t = sc_get_current_process_handle();
  SC_METHOD(method_proc);
    m = sc_get_current_process_handle();
}
```

```
void thread_proc()
{
  ...
  t.suspend();      Blocking
  ...
  t.disable();      Non-blocking
  wait(...);
  ...
}
```

```
void method_proc()
{
  ...
  m.suspend();      Non-blocking
  ...
  m.disable();      Non-blocking
  ...
}
```

sync_reset_on/off

```
SC_THREAD(calling);
SC_THREAD(target);
  t = sc_get_current_process_handle();
```

```
void calling() {
  wait(10, SC_NS);
  ev.notify();                ++q

  wait(10, SC_NS);
  t.sync_reset_on();

  wait(10, SC_NS);
  ev.notify();                q = 0

  wait(10, SC_NS);
  t.sync_reset_off();

  wait(10, SC_NS);
  ev.notify();                ++q
}
```

```
void target()
{
  q = 0;
  while (1)
  {
    wait(ev);
    ++q;
  }
}
```

Wakes at 10 30 50

Interactions

```
void calling()
{
  t.suspend();
  ...
  t.sync_reset_on();
  ...
  t.suspend();
  ...
  t.disable();
  ...
  t.sync_reset_off();
  ...
  t.resume();
  ...
  t.enable();
  ...
  t.resume();
}
```

3 independent flags

- ○ disable / enable (highest priority)
- ● suspend / resume
- ● sync_reset_on / off (lowest priority)

```
void target()
{
  q = 0;
  while (1)
  {
    wait(ev);
    ++q;
  }
}
```

Forbidden Interactions

- Suspend does not play with disable
- Suspend does not play with sync_reset_on
- Suspend does not play with clocked threads
- Disable does not play with time-outs

- All implementation-defined

- Disable and sync_reset_on play together

Process Control

- suspend
- resume
- disable
- enable
- sync_reset_on
- sync_reset_off
- reset
- kill
- throw_it

- reset_event
- sc_unwind_exception
- sc_is_unwinding

- reset_signal_is
- async_reset_signal_is

reset and kill

```
SC_THREAD(calling);
SC_THREAD(target);
  t = sc_get_current_process_handle();
```

```
void calling()
{
  wait(10, SC_NS);
  ev.notify();

  wait(10, SC_NS);
  t.reset();

  wait(10, SC_NS);
  ev.notify();

  wait(10, SC_NS);
  t.kill();
}
```

++q

q = 0

++q

```
void target()
{
  q = 0;
  while (1)
  {
    wait(ev);
    ++q;
  }
}
```

Wakes at 10 20 30

Terminated at 40

reset and kill are Immediate

```
void calling()
{
  wait(10, SC_NS);
  ev.notify();
  assert( q == 0 );

  wait(10, SC_NS);
  assert( q == 1 );

  t.reset();
  assert( q == 0 );

  wait(10, SC_NS);
  t.kill();
  assert( t.terminated() );
}
```

++q

q = 0

Forever

```
int q;
```

```
void target()
{
  q = 0;
  while (1)
  {
    wait(ev);
    ++q;
  }
}
```

Cut through suspend, disable

Disallowed during elaboration

Unwinding the Call Stack

```
void target()
{
  q = 0;
  while (1)
  {
    try {
      wait(ev);
      ++q;
    }
    catch (const sc_unwind_exception& e)
    {



    }
    ...
```

reset()

kill()

Unwinding the Call Stack

```
void target()
{
  q = 0;
  while (1)
  {
    try {
      wait(ev);
      ++q;
    }
    catch (const sc_unwind_exception& e)
    {
      sc_assert( sc_is_unwinding() );
      if (e.is_reset()) cout << "target was reset";
      else              cout << "target was killed";



    }
    ...
```

reset()

kill()

Unwinding the Call Stack

```
void target()
{
  q = 0;
  while (1)
  {
    try {
      wait(ev);
      ++q;
    }
    catch (const sc_unwind_exception& e)
    {
      sc_assert( sc_is_unwinding() );
      if (e.is_reset()) cout << "target was reset";
      else              cout << "target was killed";
      proc_handle.reset();

      throw e;
    }
    ...
```

reset()

kill()

Resets some other process

Must be re-thrown

reset_event

```
SC_THREAD(calling);
SC_THREAD(target);
  t = sc_get_current_process_handle();

SC_METHOD(reset_handler);
  dont_initialize();
  sensitive << t.reset_event();

SC_METHOD(kill_handler);
  dont_initialize();
  sensitive << t.terminated_event();
```

```
void calling()
{
  wait(10, SC_NS);
  t.reset();
  wait(10, SC_NS);
  t.kill();
  ...
```

```
void target()
{
  ...
  while (1)
  {
    wait(ev);
    ...
  }
}
```

Suicide

```
void target()
{
  q = 0;
  while (1)
  {
    wait(ev);
    ++q;
    if (q == 5)
    {
      handle = sc_get_current_process_handle();
      handle.kill();
      assert( false );
    }
  }
}
```

Never executes this line

throw_it

std::exception recommended

```
std::exception ex;
```

```
void calling()
{
  ...
  t.throw_it(ex);
  ...
}
```

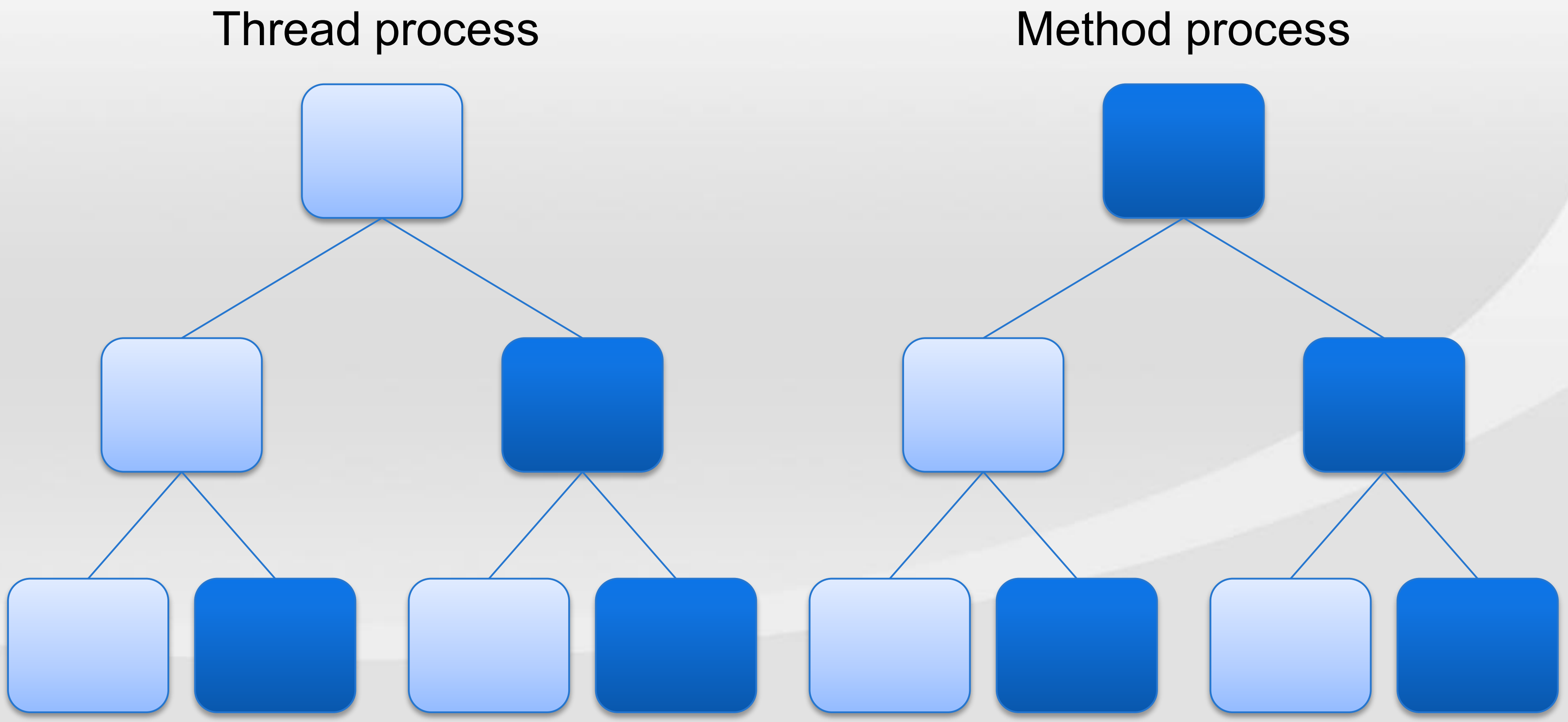
Immediate - 2 context switches

```
void target()
{
  q = 0;
  while (1) {
    try {
      wait(ev);
      ++q;
    }
    catch (const std::exception& e)
    {
      if (...)
        ; // wait(ev);
      else
        return;
    }
  ...
```

Must catch exception

May continue or terminate

Include Descendants



Include Descendants

```
M(sc_module_name n)
{
  SC_THREAD(calling);
  t = sc_spawn(sc_bind(&M::child_thread, 3));
  m = sc_spawn(sc_bind(&M::child_method, 3), "m", &opt);
}
```

```
void child_thread(int level)
{
  if (level > 0) {
    sc_spawn(sc_bind(&M::child_thread, level - 1));
    sc_spawn(sc_bind(&M::child_method, level - 1), "m", &opt);
  }
  while (1)
  {
    wait(ev);
    ...
  }
}
```

Include Descendants

```
void calling()
{
   wait(10, SC_NS);
   t.suspend();

   wait(10, SC_NS);
   t.suspend(SC_INCLUDE_DESCENDANTS);

}
```

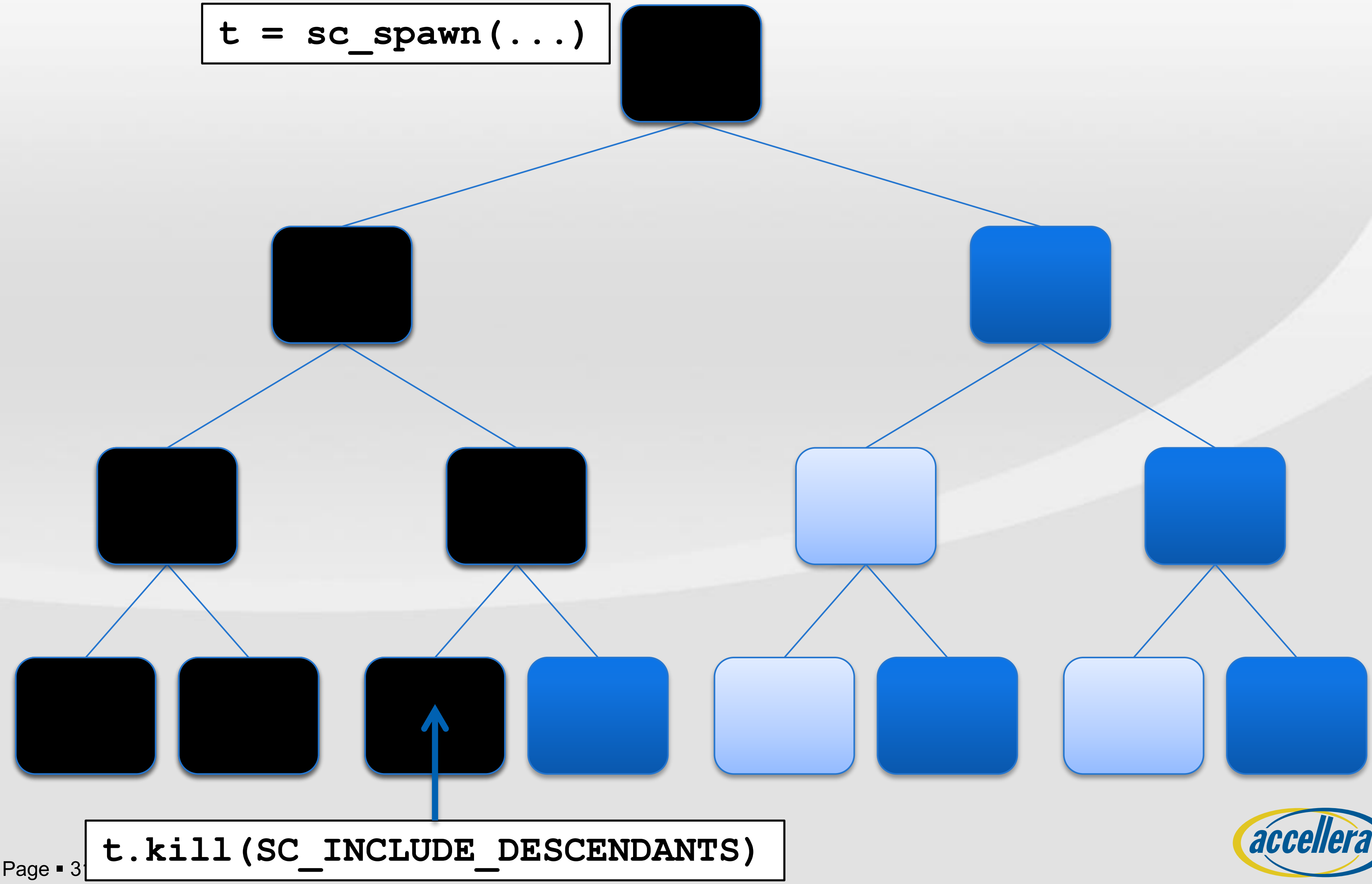
Null action on t itself

```
void child_thread(int level)
{
   ...
   if (...)
      t.kill(SC_INCLUDE_DESCENDANTS);
   ...
}
```

Kills itself!

SYSTEMS INITIATIVE

Attempted Genocide



```
t = sc_spawn(...)
```

```
t.kill(SC_INCLUDE_DESCENDANTS)
```

Process Control

- suspend
- resume
- disable
- enable
- sync_reset_on
- sync_reset_off
- reset
- kill
- throw_it

- reset_event
- sc_unwind_exception
- sc_is_unwinding

- reset_signal_is
- async_reset_signal_is

Styles of Reset

	asynchronous	sync reset state
`handle.reset();`	●	
`handle.sync_reset_on();`		●
`...`		
`handle.sync_reset_off();`		
`SC_THREAD(target);`		
`reset_signal_is(reset, active_level);`		●
`async_reset_signal_is(reset, active_level);`	●	●
`sc_spawn_options opt;`		
`opt.reset_signal_is(reset, active_level);`		●
`opt.async_reset_signal_is(reset, true);`	●	●

Styles of Reset

```
SC_THREAD(target);
  sensitive << ev;
  reset_signal_is(sync_reset, true);
  async_reset_signal_is(async_reset, true);
```

Effectively

```
t.reset();                          t.reset();
t.sync_reset_on();
...
ev.notify();                        t.reset();
...
t.sync_reset_off();
sync_reset = true;
...
ev.notify();                        t.reset();
sync_reset = false;
...
async_reset = true;                 t.reset();
..
ev.notify();                        t.reset();
```

Processes Unified!

```
SC_METHOD(M);
  sensitive << clk.pos();
  reset_signal_is(r, true);
  async_reset_signal_is(ar, true);
```

```
SC_THREAD(T);
  sensitive << clk.pos();
  reset_signal_is(r, true);
  async_reset_signal_is(ar, true);
```

```
SC_CTHREAD(T, clk.pos());
  reset_signal_is(r, true);
  async_reset_signal_is(ar, true);
```

```
void M() {
  if (r|ar)
    q = 0;
  else
    ++q
}
```

```
void T() {
  if (r|ar)
    q = 0;
  while (1)
  {
    wait();
    ++q;
  }
}
```

Reset Technicalities

- Can have any number of sync and async resets

- Reset clears dynamic sensitivity and restores static sensitivity
- Reset wipes the slate clean for resume

- Method process called when reset
  - Synchronous reset resets sensitivity
  - *else* can *only* mean clock

- Clocked threads not called during initialization
- Clocked threads sensitive to one clock

```
void M() {
  if (reset)
    q = 0;
  else
    ++q
}
```

```
void T() {
  if (reset)
    q = 0;
  while (1)
    ...
}
```

Processes in Containers

has operator< and swap

```
#include <map>
typedef std::map<sc_process_handle, int> proc_map_t;
proc_map_t all_procs;
```

```
SC_THREAD(proc);
  handle = sc_get_current_process_handle();
  all_procs[handle] = ++num;
```

```
proc_map_t::iterator it;
for (it = all_procs.begin(); it != all_procs.end(); it++)
  cout << it->first.name() << " in "
       << it->first.get_parent_object()->name() << endl;
```

Contents

Stepping Simulation

```
int sc_main(...)
{
  Top top("top");
  sc_start(10, NS);
  ...
  sc_start(0, SC_NS);
  ...
  sc_start();
  ...
  sc_start();
  ...
}
```

Simulation time = 10ns?

Did anything happen?

Simulation time = max time?

Nothing left to do?

Event Starvation

```
int sc_main(...)
{
  Top top("top");
  sc_time period(10, SC_NS);

  sc_start(period);

  sc_start(period, SC_RUN_TO_TIME);

  sc_start(period, SC_EXIT_ON_STARVATION);
  ...
  sc_start();
  sc_start();
}
```

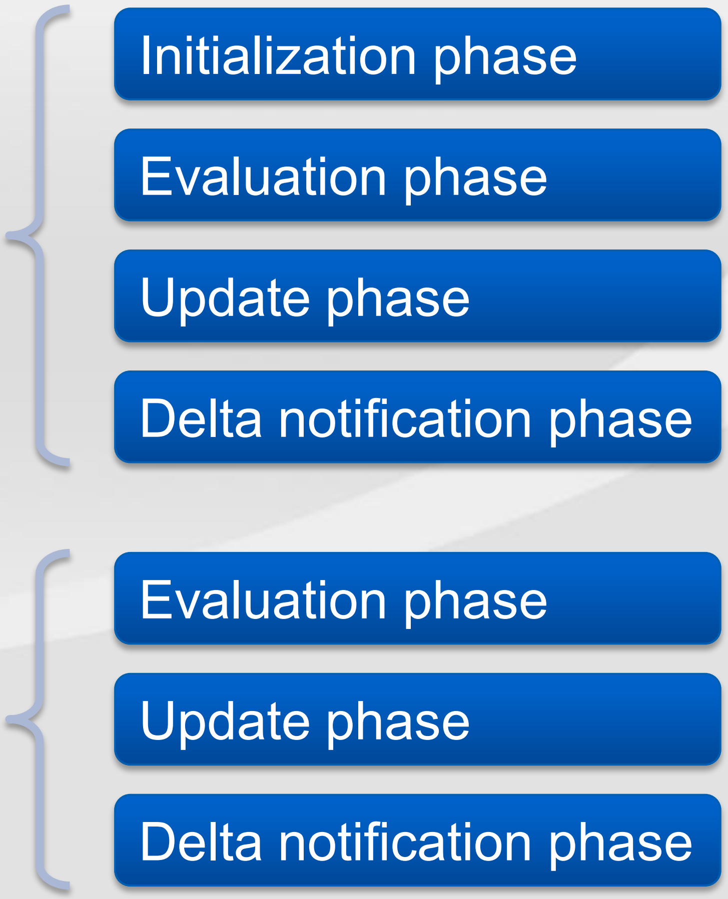
Time = end time

Don't run processes at end time

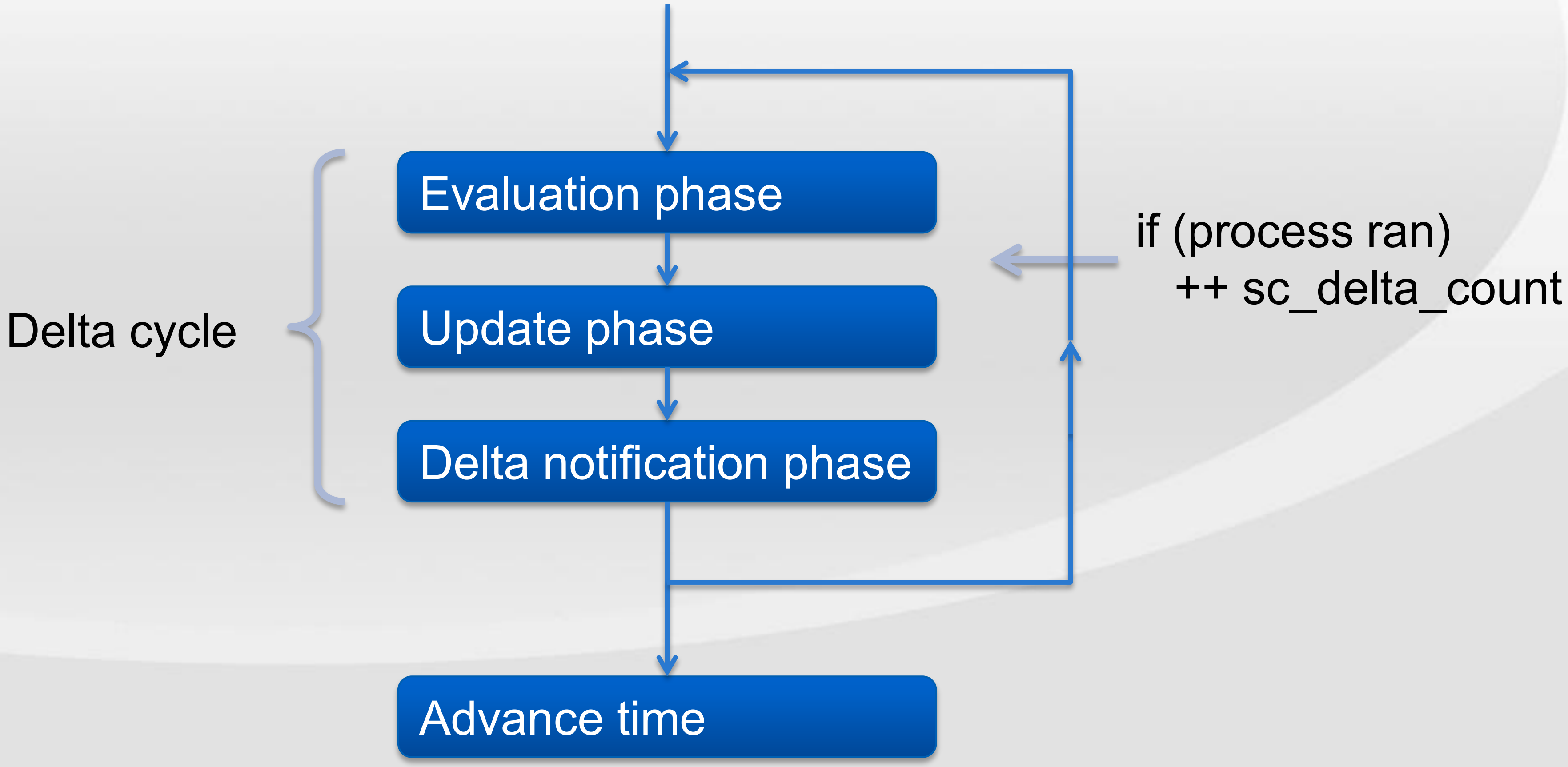
Time = latest event

sc_start(0)

```
int sc_main(...)
{
  Top top("top");
  sc_start(0, SC_NS);


  ...



  sc_start(0, SC_NS);
}
```



The Delta Cycle



Pending Activity

Pseudo-code

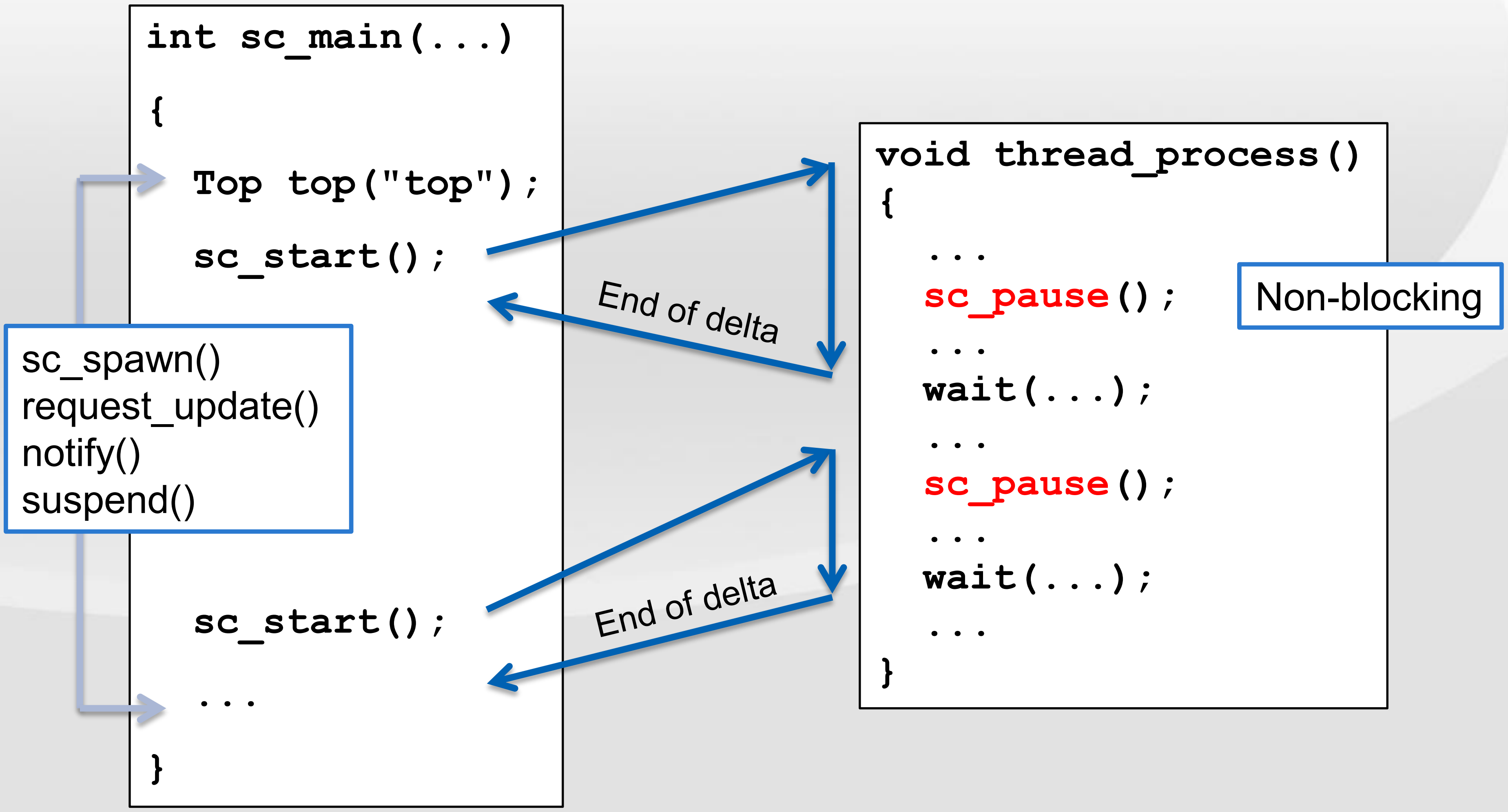
```
sc_time sc_time_to_pending_activity()

{

  if ( sc_pending_activity_at_current_time() )

    return SC_ZERO_TIME;

  else if ( sc_pending_activity_at_future_time() )

    return (time of earliest event) - sc_time_stamp();

  else

    return sc_max_time() - sc_time_stamp();

}
```

Single Stepping the Scheduler

```
int sc_main(...) {
  Top top("top");


  ...   Create some activity


  while (sc_pending_activity())
    sc_start(sc_time_to_pending_activity());
}
```

- Either run one delta cycle at current time
- or advance simulation time but don't run any processes

Pausing Simulation



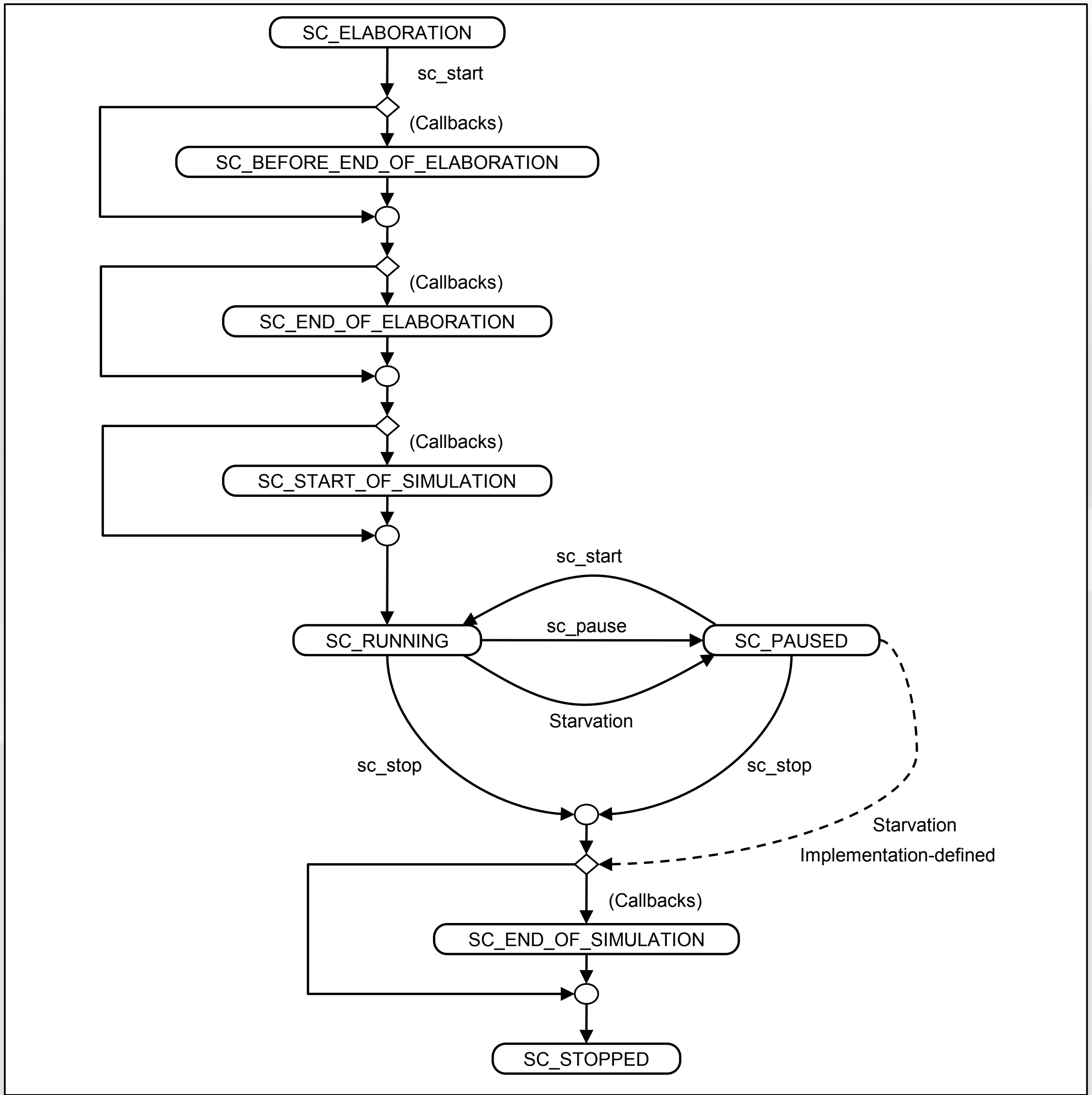
```
int sc_main(...)
{
  Top top("top");
  sc_start();



  sc_start();
  ...
}
```

```
void thread_process()
{
  ...
  sc_pause();
  ...
  wait(...);
  ...
  sc_pause();
  ...
  wait(...);
  ...
}
```

Simulation Status

```
int sc_main(...)
{
  Top top("top");
  assert( sc_get_status() == SC_ELABORATION );

  sc_start();
  assert( sc_get_status() == SC_PAUSED );
  ...
  sc_start();
  ...
  sc_start();
  assert( sc_get_status() == SC_STOPPED );
}
```

Simulation Status



Immediate Notification

```
SC_THREAD(target);
  sensitive << ev;
```

```
void target()
{
  assert( sc_delta_count() == 0 );
  wait(SC_ZERO_TIME);
  assert( sc_delta_count() == 1 );
  ev.notify(5, SC_NS);
  assert( sc_time_to_pending_activity()
                 == sc_time(5, SC_NS) );
  wait(ev);
  ev.notify();
  wait(ev);
  sc_assert( false );
}
```

Assuming!

Process does not awake

Contents

Array of Ports or Signals

```
struct Child: sc_module
{
  sc_in<int> p[4];
  ...
```

Ports cannot be named

```
struct Top: sc_module
{
  sc_signal<int> sig[4];
  Child* c;

  Top(sc_module_name n)
  {
    c = new Child("c");
    c->p[0].bind(sig[0]);
    c->p[1].bind(sig[1]);
    c->p[2].bind(sig[2]);
    c->p[3].bind(sig[3]);
  }
  ...
```

Signals cannot be named

Array or Vector of Modules

```
struct Child: sc_module
{
  sc_in<int> p;
  ...
```

```
struct Top: sc_module
{
  sc_signal<int> sig[4];
  std::vector<Child*> vec;

  Top(sc_module_name n) {
    vec.resize(4);
    for (int i = 0; i < 4; i++)
    {
      std::stringstream n;
      n << "vec_" << i;
      vec[i] = new Child(n.str().c_str(), i);
      vec[i]->p.bind(sig[i]);
    }
  }
  ...
```

Modules not default constructible

sc_vector of Ports or Signals

```
struct Child: sc_module
{
  sc_vector< sc_in<int> > port_vec;

  Child(sc_module_name n)
  : port_vec("port_vec", 4)
  { ...
```

Elements are named

```
struct Top: sc_module
{
  sc_vector< sc_signal<int> > sig_vec;
  Child* c;

  Top(sc_module_name n)
  : sig_vec("sig_vec", 4)
  {
    c = new Child("c");
    c->port_vec.bind(sig_vec);
  }
  ...
```

Size passed to ctor

Vector-to-vector bind

sc_vector of Modules

```
struct Child: sc_module
{
  sc_in<int> p;
  ...
```

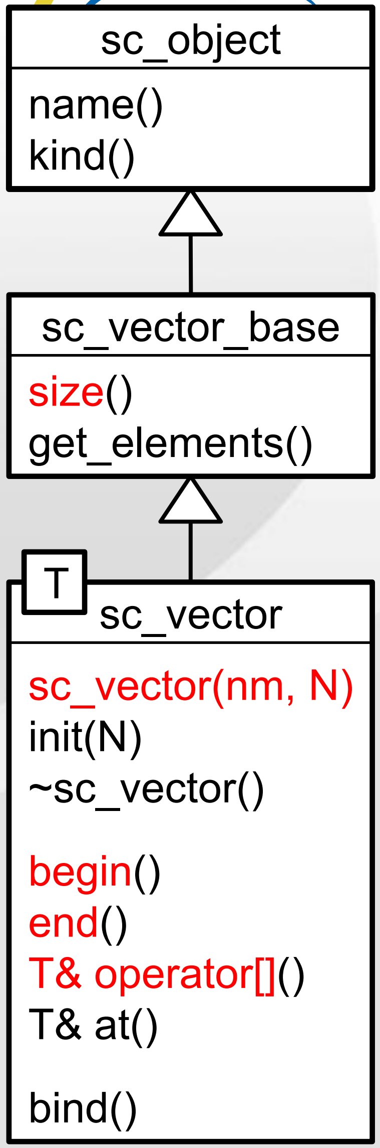
```
struct Top: sc_module
{
  sc_vector< sc_signal<int> > sig_vec;
  sc_vector< Child > mod_vec;

  Top(sc_module_name n)
  : sig_vec("sig_vec")
  , mod_vec("mod_vec")
  {
    sig_vec.init(4);
    mod_vec.init(4);
    for (int i = 0; i < 4; i++)
      mod_vec[i]->p.bind(sig_vec[i]);
  }
  ...
```

Elements are named

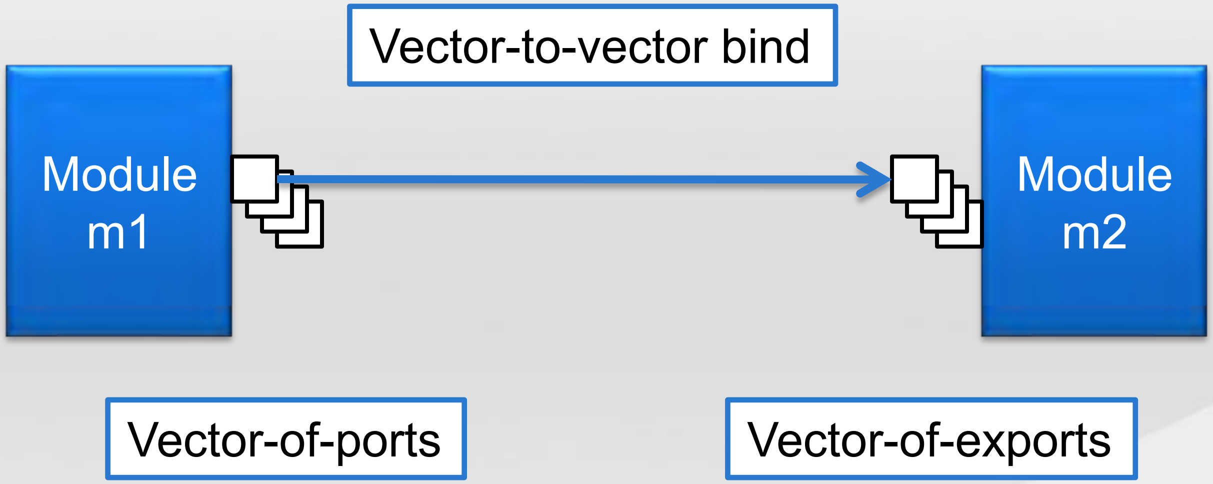
Size deferred

sc_vector methods

```
struct M: sc_module
{
  sc_vector< sc_signal<int> > vec;

  M(sc_module_name n)
  : vec("vec", 4) {
    SC_THREAD(proc)
  }
  void proc() {
    for (unsigned int i = 0; i < vec.size(); i++)
      vec[i].write(i);

    wait(SC_ZERO_TIME);

    sc_vector< sc_signal<int> >::iterator it;
    for (it = vec.begin(); it != vec.end(); it++)
      cout << it->read() << endl;
    ...
```

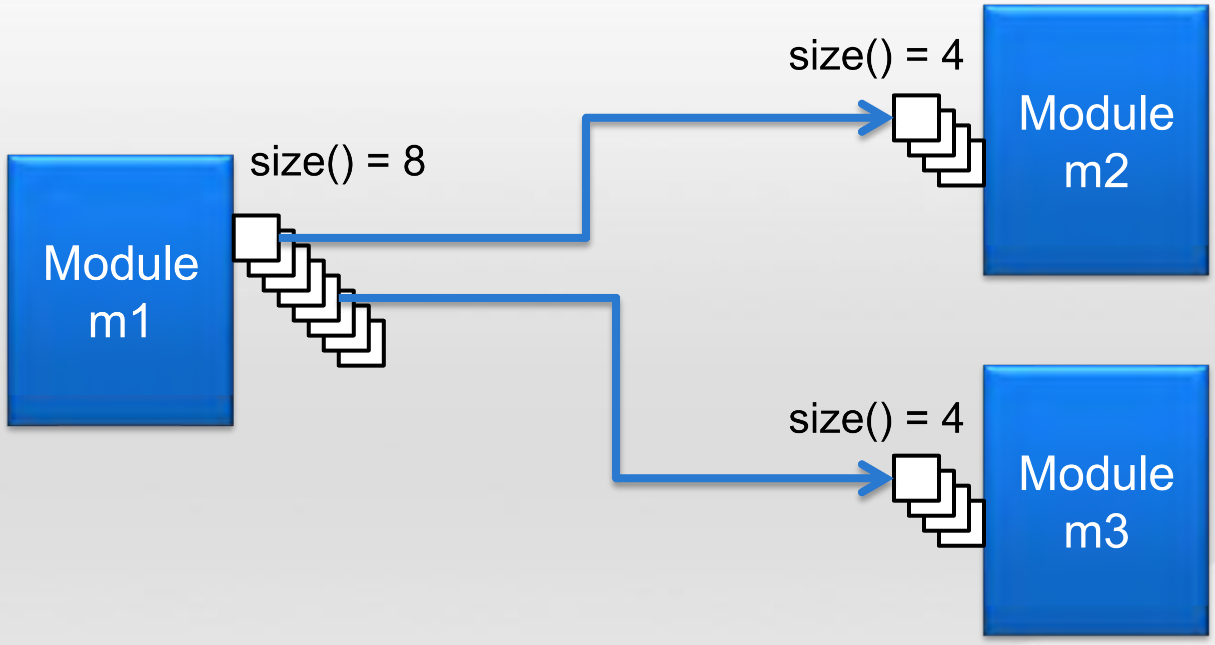


Binding Vectors



```
m1->port_vec.bind( m2->export_vec );
```

Partial Binding

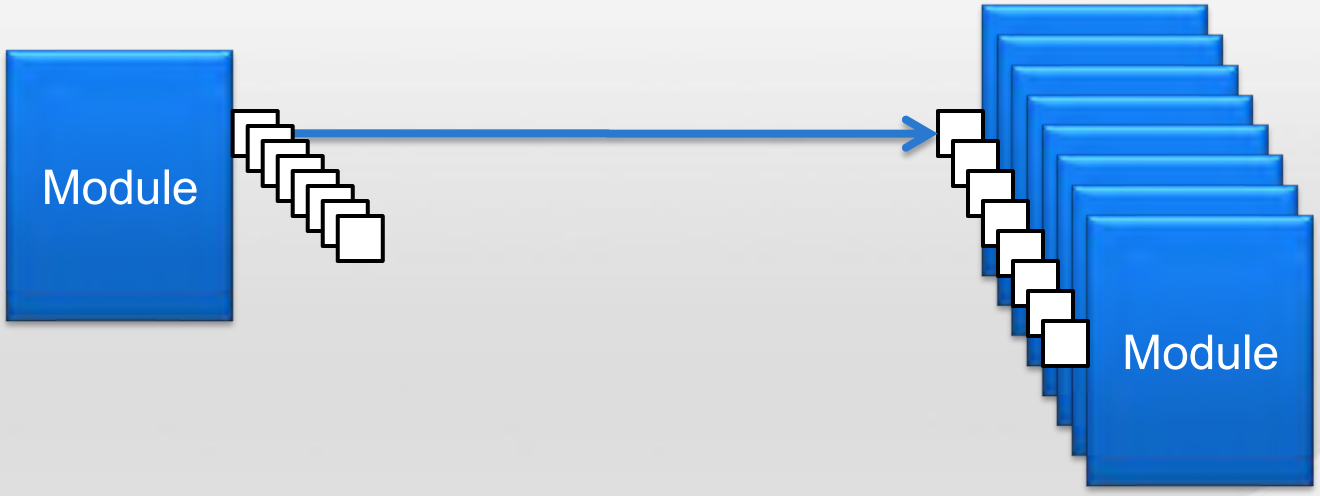


```
sc_vector<sc_port<i_f> >::iterator it;
it = m1->port_vec.bind( m2->export_vec );

it = m1->port_vec.bind( m3->export_vec.begin(),
                        m3->export_vec.end(),
                        it );
```

1st unbound element

Start binding here

sc_assemble_vector



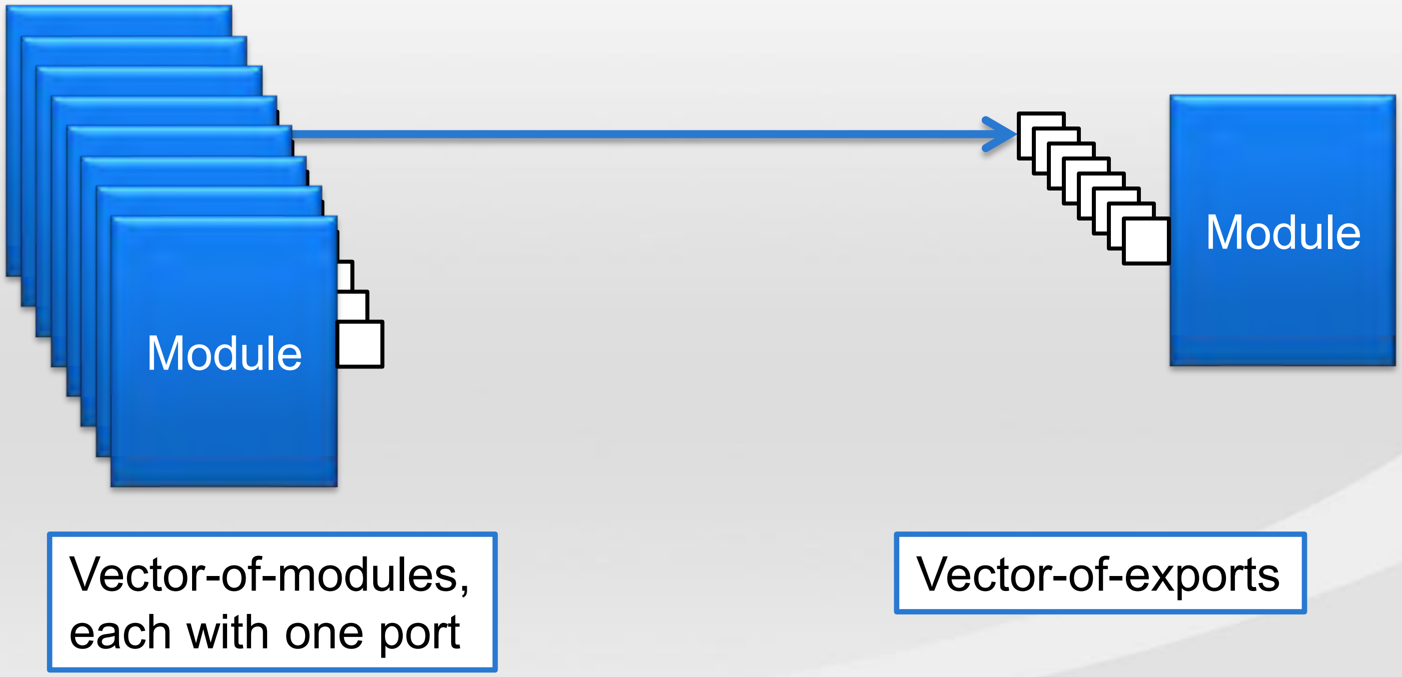
Vector-of-ports

Vector-of-modules, each with one export

```
init->port_vec.bind(
    sc_assemble_vector(targ_vec, &Target::export) );
```

Substitute for a regular vector

sc_assemble_vector



```
sc_assemble_vector(init_vec, &Init::port).bind(
                                    targ->export_vec);
```

Constructor Arguments

```
struct M: sc_module
{
  M(sc_module_name n, int a, bool b);
  ...
```

```
sc_vector<M> mod_vec;
```

```
static M* creator_func( const char* name, size_t s )
{
  return new M(name, 3, true);
}
```

Pass args to constructor

```
mod_vec.init(4, creator_func);
```

Fancy Variant 1

```
struct M: sc_module
{
  M(sc_module_name n, int a, bool b);
  ...
```

```
sc_vector<M> mod_vec;
```

```
M* creator_func( const char* name, size_t s )
{
  return new M(name, 3, true);
}
```

Member fn

Pass args to constructor

```
mod_vec.init(4,
        sc_bind(&top::creator_func, this, _1, _2));
```

Fancy Variant 2

```
struct M: sc_module
{
  M(sc_module_name n, int a, bool b);
  ...
```

```
sc_vector<M> mod_vec;
```

Function object

```
struct creator {
  creator( int a, bool b ) : m_a(a), m_b(b) {}
  int  m_a;
  bool m_b;
  M* operator() (const char* name, size_t) {
    return new M(name, m_a, m_b );
  }
};
```

Pass args to constructor

```
mod_vec.init(4, creator(3, true));
```

sc_vector Restrictions

- Restricted to sc_vector<derived_from_sc_object>
- Elements become children of vector's parent
- Cannot be resized
- Cannot be copied or assigned

Contents

Odds and Ends

- Event List Objects
- Named Events
- sc_writer_policy
- Verbosity
- Virtual Bind
- Other Enhancements

Waiting on a List of Events

```
sc_port<sc_signal_in_if<int>, 0> port;
...

void thread_process()
{
  wait(port[0] | port[1] | port[2] | ...);
  ...
}
```

Multiport

Not expressible in SystemC

Event List Objects

Multiport

```
sc_port<sc_signal_in_if<int>, 0> port;
...

void thread_process()
{
  sc_event_or_list or_list;

  for (int i = 0; i < port.size(); i++)

    or_list |= port[i]->default_event();

  wait(or_list);
  ...
}
```

Event List Technicalities

```
sc_event ev1, ev2, ev3, ev4;
```

```
sc_event_or_list  or_list;
sc_event_and_list and_list = ev1;
assert( or_list.size()  == 0 );
assert( and_list.size() == 1 );

or_list = ev1;
or_list = or_list | ev2 | ev3;
or_list |= ev4;
assert( or_list.size() == 4 );

and_list &= ev2 & ev2 & ev2;
assert( and_list.size() == 2 );

wait(or_list);
wait(and_list);
```

Can't mix them up

Duplicates don't count

List must be valid when process resumes

Named Events

```
struct M: sc_module
{
  sc_event my_event;

  M(sc_module_name n)
  : my_event("my_event")
  {
    assert( my_event.in_hierarchy() );
    assert( my_event.get_parent_object() == this );

    assert( sc_find_event("top.my_event") == &my_event );

    std::vector<sc_event*> vec = this->get_child_events();
    assert( vec.size() == 1 );
    ...
```

Events created during elab are named

Events are not sc_objects

Run-Time Events

```
struct M: sc_module
{
  M(sc_module_name n) { SC_THREAD(proc); }

  void proc()
  {
    sc_event ev1("ev1");
    assert( ev1.in_hierarchy() );

    sc_event ev2;
    assert( !ev2.in_hierarchy() );
    cout << ev2.name();
    ...
```

Implementation-defined for performance

Kernel Events

```
struct M: sc_module
{
  sc_event          my_event;
  sc_signal<bool> my_sig;

  M(sc_module_name n)
  : my_event("my_event")
  , my_sig("my_sig")
  {
    cout << my_sig.default_event().name();


    assert( sc_hierarchical_name_exists("m.my_event") );
    assert( sc_hierarchical_name_exists("m.my_sig") );
    ...
```

Kernel events not hierarchically named

m.$$$$kernel_event$$$$__value_changed_event

sc_object and sc_event share the same namespace

sc_writer_policy

```
struct M: sc_module
{
  sc_signal<int> sig1;
  sc_signal<int, SC_MANY_WRITERS> sig_many;

  M(sc_module_name n) {
    SC_THREAD(proc1);
    SC_THREAD(proc2);
  }
```

Default SC_ONE_WRITER

```
void proc1()
{
  sig1.write(1);
  wait(1, SC_NS);
  sig_many.write(3);
  wait(1, SC_NS);
  sig_many.write(4);
}
```

OK

```
void proc2()
{
  sig_many.write(2);
  wait(1, SC_NS);
  sig1.write(4);
  wait(1, SC_NS);
  sig_many.write(6);
}
```

Error

Error

sc_writer_policy/b_transport

```
sc_signal<int, SC_MANY_WRITERS> interrupt;
```

```
void b_transport( tlm::tlm_generic_payload& trans,
                  sc_time& delay )
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64    adr = trans.get_address();
  ...
  if ( cmd == tlm::TLM_WRITE_COMMAND && adr == 0xFFFF)
    interrupt.write(level);

  ...
  trans.set_response_status( tlm::TLM_OK_RESPONSE );
}
```

Called from several initiators

Verbosity Filter for Reports

```
enum sc_verbosity {
  SC_NONE   = 0,
  SC_LOW    = 100,
  SC_MEDIUM = 200,
  SC_HIGH   = 300,
  SC_FULL   = 400,
  SC_DEBUG  = 500
};
```

Sets a global maximum

```
sc_report_handler::set_verbosity_level( SC_LOW );
```

```
SC_REPORT_INFO("msg_type", "msg");

SC_REPORT_INFO_VERB("msg_type", "msg", SC_LOW);
```

Default is SC_MEDIUM

Ignored if argument > global maximum

virtual bind

```
template<typename IF>
struct my_port: sc_core::sc_port<IF> {
  typedef sc_core::sc_port<IF> base_port;
  virtual void bind( IF& iface ) {
    ...
    base_port::bind( iface );
  }
  using base_port::bind;
};
```

Relevant to all specialized ports

Do something special

Don't override operator()

```
struct M: sc_module
{
  my_port< sc_fifo_in_if<int> > my_fifo_in;
  ...
```

```
sc_fifo<int> my_fifo;
M m("m");
m.my_fifo_in(my_fifo);
```

Call sc_port<IF>::operator()

Other Enhancements

- Certain fixed-point constructors made explicit
- Preprocessor macros to return SystemC version
- sc_mutex and sc_semaphore no longer primitive channels
- Asynchronous update requests for primitive channels

Contents

Versions

1666-2011 requires SystemC 2.3 and TLM-2.0.2

```
#define SC_DISABLE_VIRTUAL_BIND
#include <systemc>
using namespace sc_core;
#include <tlm.h>
```

To run SystemC 2.3 with TLM-2.0.1

1666-2011 allows #include <tlm>

```
int sc_main(int argc, char* argv[])
{
  #ifdef IEEE_1666_SYSTEMC
    cout << SC_VERSION << endl;
    cout << SC_VERSION_RELEASE_DATE <
  #endif
  cout << TLM_VERSION << endl;
  cout << TLM_VERSION_RELEASE_DATE <<
  ...
  sc_start();
  return 0;
}
```

2.3.0_pub_rev_20111121-OSCI
20111121

2.0.1_-TLMWG
20090715

TLM-2.0 Compliance

- TLM-2.0-compliant-implementation
- TLM-2.0-base-protocol-compliant
- TLM-2.0-custom-protocol-compliant

Generic Payload Option

Attribute	Transport	DMI	Debug
Command	Yes	Yes	Yes
Address	Yes	Yes	Yes
Data pointer	Yes	No	Yes
Data length	Yes	No	Yes
Byte enable pointer	Yes	No	No
Byte enable length	Yes	No	No
Streaming width	Yes	No	No
DMI hint	Yes	No	No
Response status	Yes	No	No
Extensions	Yes	Yes	Yes

Enabled using gp_option

Backward compatible with pre-IEEE version

set/get_gp_option

Initiator

```
trans->set_gp_option(TLM_FULL_PAYLOAD);
trans->set_streaming_width(4);
socket->transport_dbg( *trans );
```

Target

```
if (trans.get_gp_option() == TLM_FULL_PAYLOAD)
{
  trans.set_gp_option(TLM_FULL_PAYLOAD_ACCEPTED);
  trans.set_response_status( TLM_OK_RESPONSE );
}
```

Initiator

```
if (trans->get_gp_option() == TLM_FULL_PAYLOAD_ACCEPTED )
  if (trans->is_response_error())
    ...
```

gp_option Technicalities

- TLM_MIN_PAYLOAD
  - Default, backward compatible
  - All components ignore optional attributes
- TLM_FULL_PAYLOAD
  - Set by initiator for DMI and Debug only
  - Set all attributes to proper values
- TLM_FULL_PAYLOAD_ACCEPTED
  - Set by target
  - DMI & Debug – response status used
  - Debug – byte enables, streaming, and DMI hint used

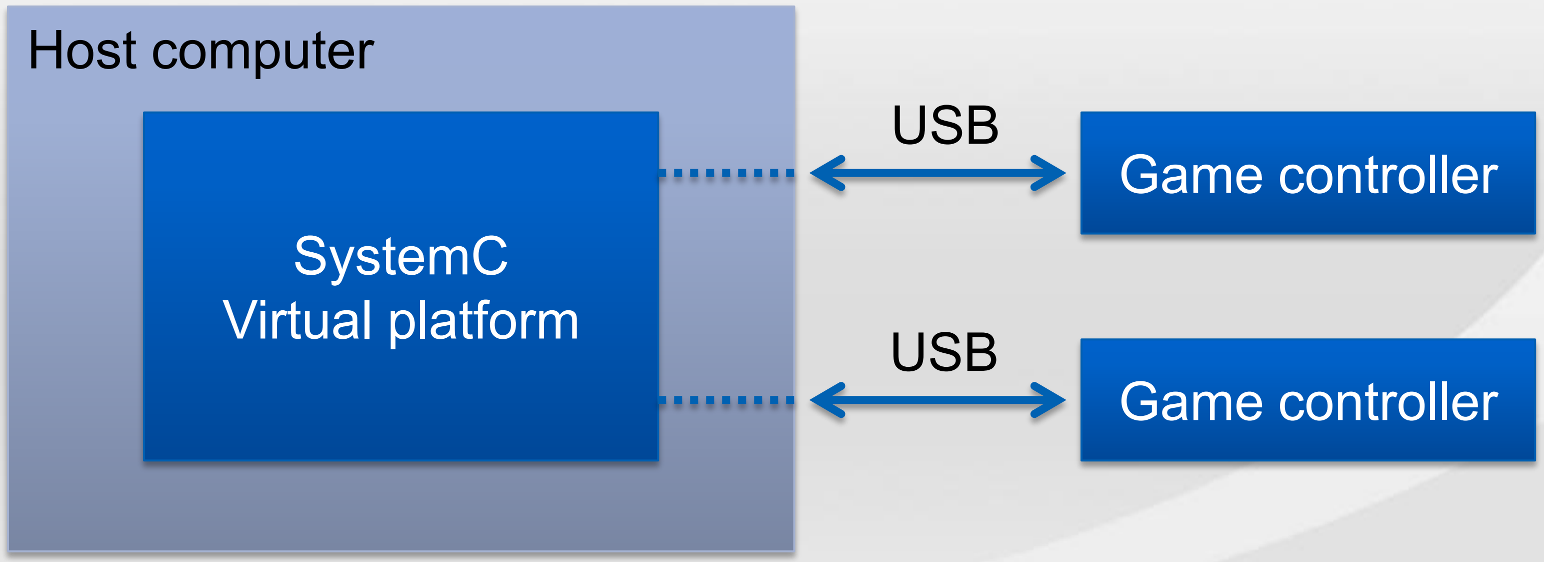
Other Changes

- TLM_IGNORE_COMMAND used for custom commands
- Generic payload data array pointer may now be null
- Target may now return any value from transport_dbg

- Macro DECLARE_EXTENDED_PHASE is deprecated
- Renamed to TLM_DECLARE_EXTENDED_PHASE

Contents

One Motivation



Expect near-real-time responsiveness

Co-operative Multitasking

```
SC_THREAD(thread1);
SC_THREAD(thread2);
```

```
void thread1()
{
  wait(0, SC_NS);









}
```

```
void thread2()
{
  while (1) {
    wait(ev1);








}
```

Co-operative Multitasking

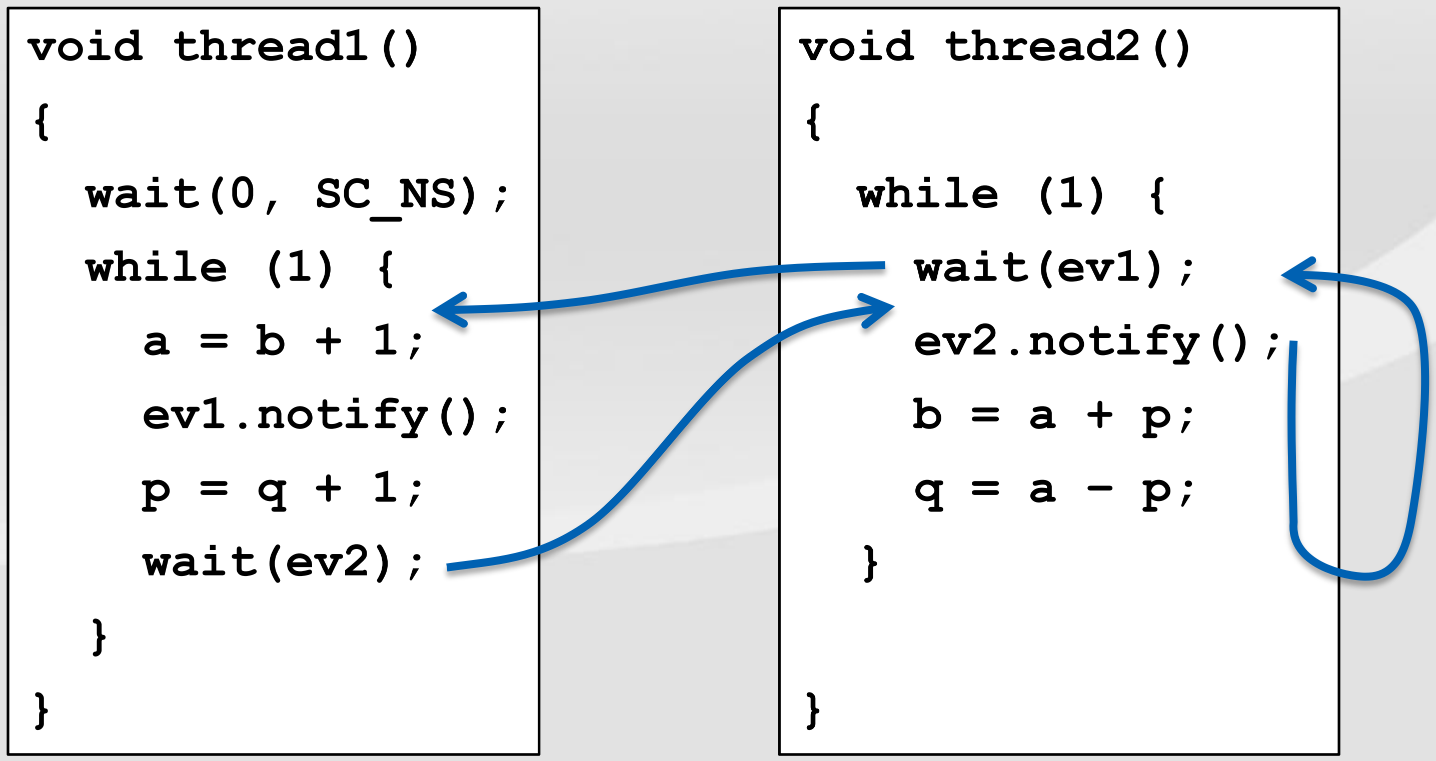
```
SC_THREAD(thread1);
SC_THREAD(thread2);
```

```
void thread1()
{
  wait(0, SC_NS);
  while (1) {
    a = b + 1;
    ev1.notify();
    p = q + 1;
    wait(ev2);
  }
}
```

```
void thread2()
{
  while (1) {
    wait(ev1);




}
```

Co-operative Multitasking

```
SC_THREAD(thread1);
SC_THREAD(thread2);
```

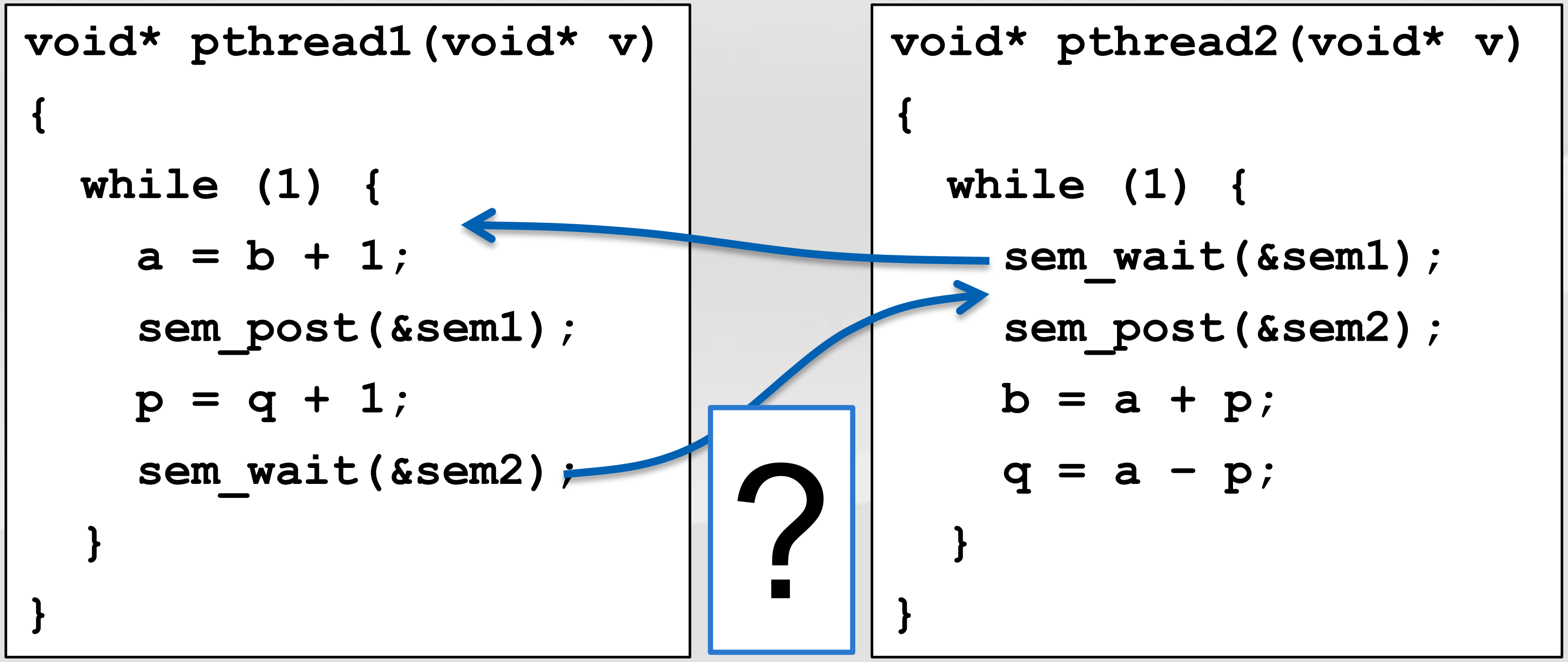


```
void thread1()
{
  wait(0, SC_NS);
  while (1) {
    a = b + 1;
    ev1.notify();
    p = q + 1;
    wait(ev2);
  }
}
```

```
void thread2()
{
  while (1) {
    wait(ev1);
    ev2.notify();
    b = a + p;
    q = a - p;
  }
}
```

Pre-emption

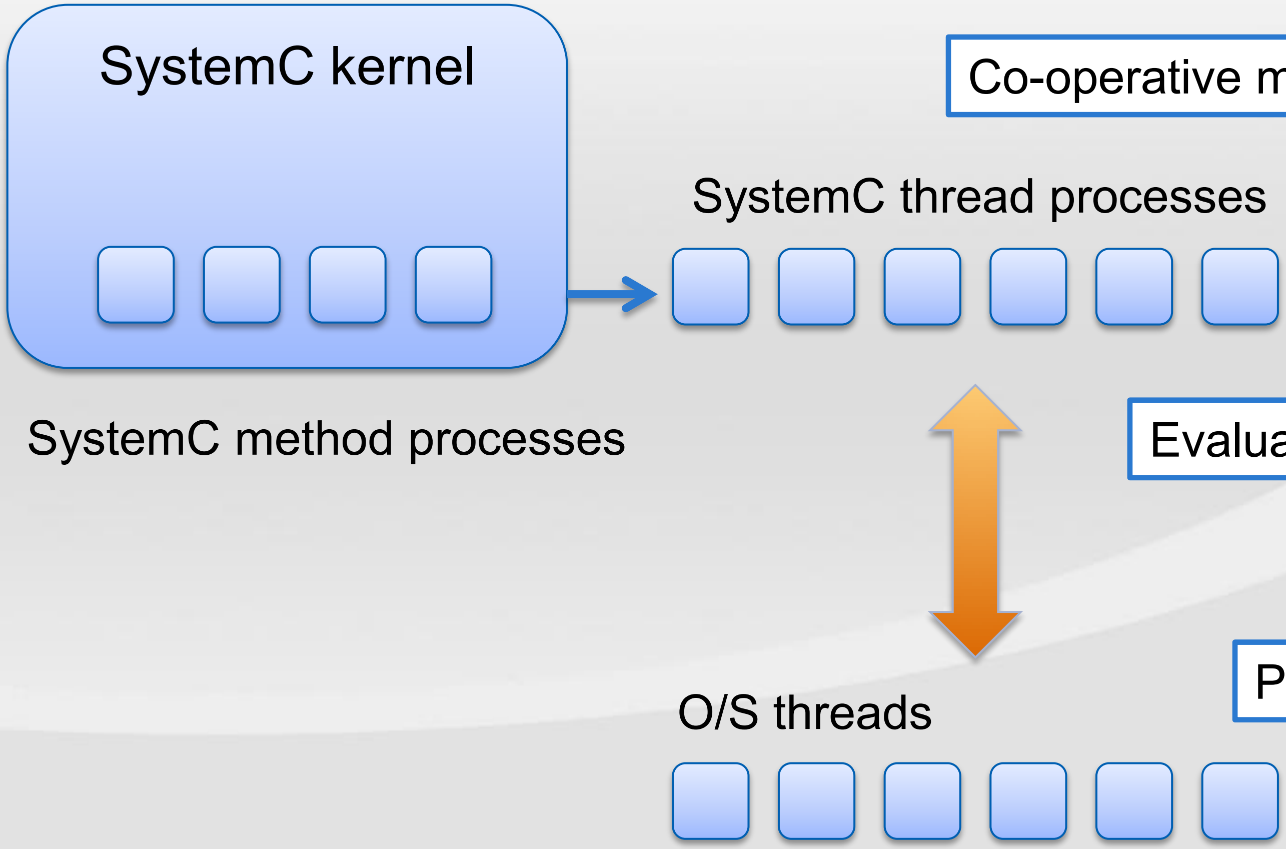
```
status = pthread_create(&p1, NULL, pthread1, NULL);
status = pthread_create(&p2, NULL, pthread2, NULL);
```



```
void* pthread1(void* v)
{
  while (1) {
    a = b + 1;
    sem_post(&sem1);
    p = q + 1;
    sem_wait(&sem2);
  }
}
```

```
void* pthread2(void* v)
{
  while (1) {
    sem_wait(&sem1);
    sem_post(&sem2);
    b = a + p;
    q = a - p;
  }
}
```

SystemC and O/S Threads



Creating a pthread

```
#include <pthread.h>

struct M: sc_module
{
  pthread_t pthread;

  M(sc_module_name n)
  {
    int status;
    status = pthread_create(&pthread, NULL, pth, this);

    SC_THREAD(scth);

    sem_init(&empty, 0, 1);
    sem_init(&full,  0, 0);
  }

  ~M() { pthread_join( pthread, NULL ); }
  ...
```

pthread and SC_THREAD

pthread - producer

```
void* pth(void* ptr)
{
  for (int i = 0; i < 8; i++)
  {
    rendezvous_put(i);
  }
  return NULL;
}
```

SC_THREAD - consumer

```
void scth()
{
  for (int i = 0; i < 8; i++)
  {
    cout << rendezvous_get() << endl;
    wait(1, SC_NS);
  }
}
```

Synchronization

```
#include <semaphore.h>
sem_t empty;
sem_t full;
int data;
```

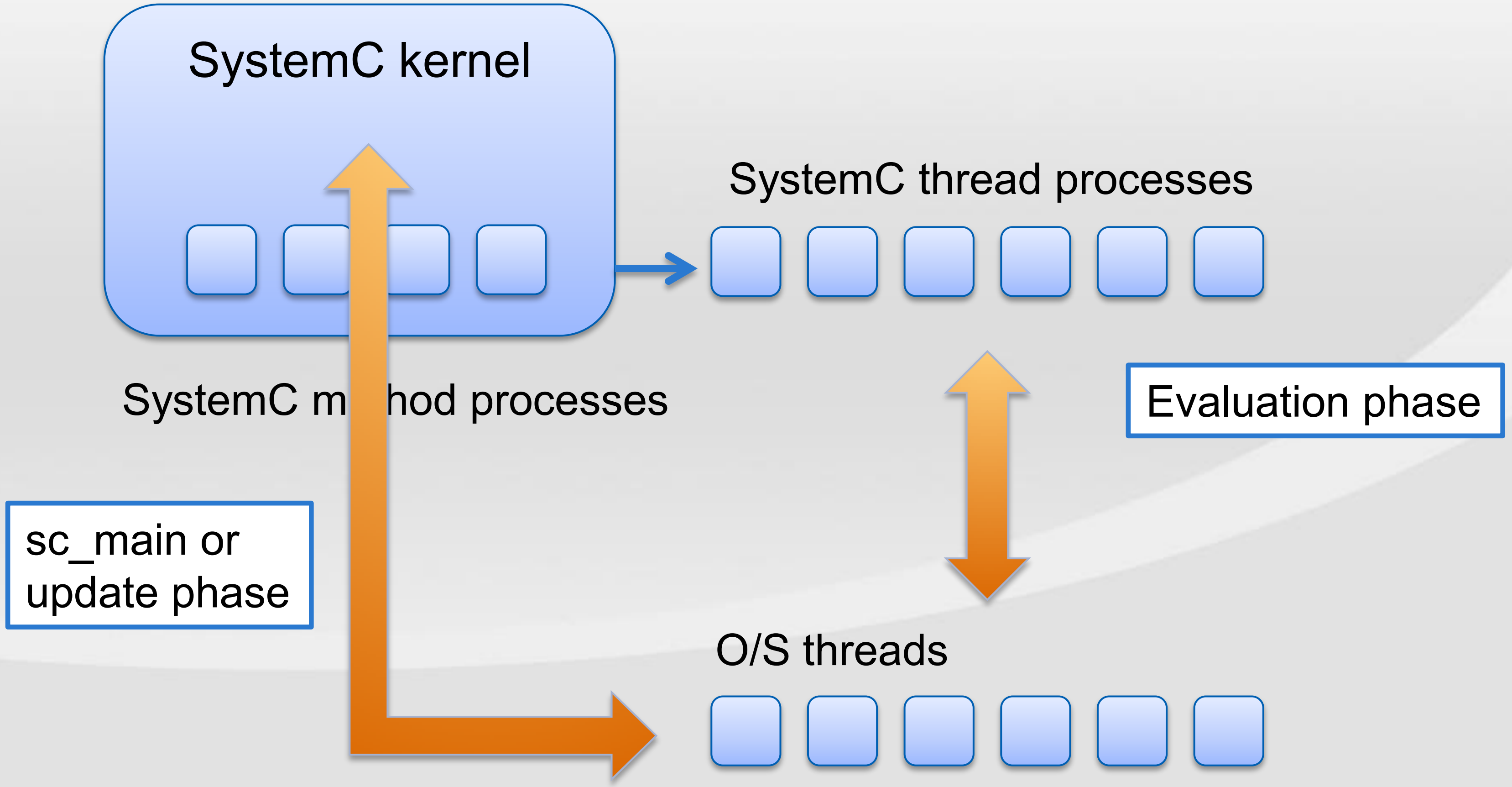
Cannot use sc_semaphore

```
sem_init(&empty, 0, 1);
sem_init(&full,  0, 0);
```

```
void rendezvous_put(int _data)
{
  sem_wait(&empty);
  data = _data;
  sem_post(&full);
}
```

```
int rendezvous_get()
{
  int result;
  sem_wait(&full);
  result = data;
  sem_post(&empty);
  return result;
}
```

Sync with Kernel



Thread-Safe Primitive Channel

```
struct thread_safe_channel: sc_prim_channel, IF
{
  thread_safe_channel(const char* name);

  virtual void write(int value);
  virtual int  read();
  virtual const sc_event& default_event() const;

protected:
  virtual void update();

private:
  int m_current_value;
  int m_next_value;
  sc_event m_value_changed_event;
};
```

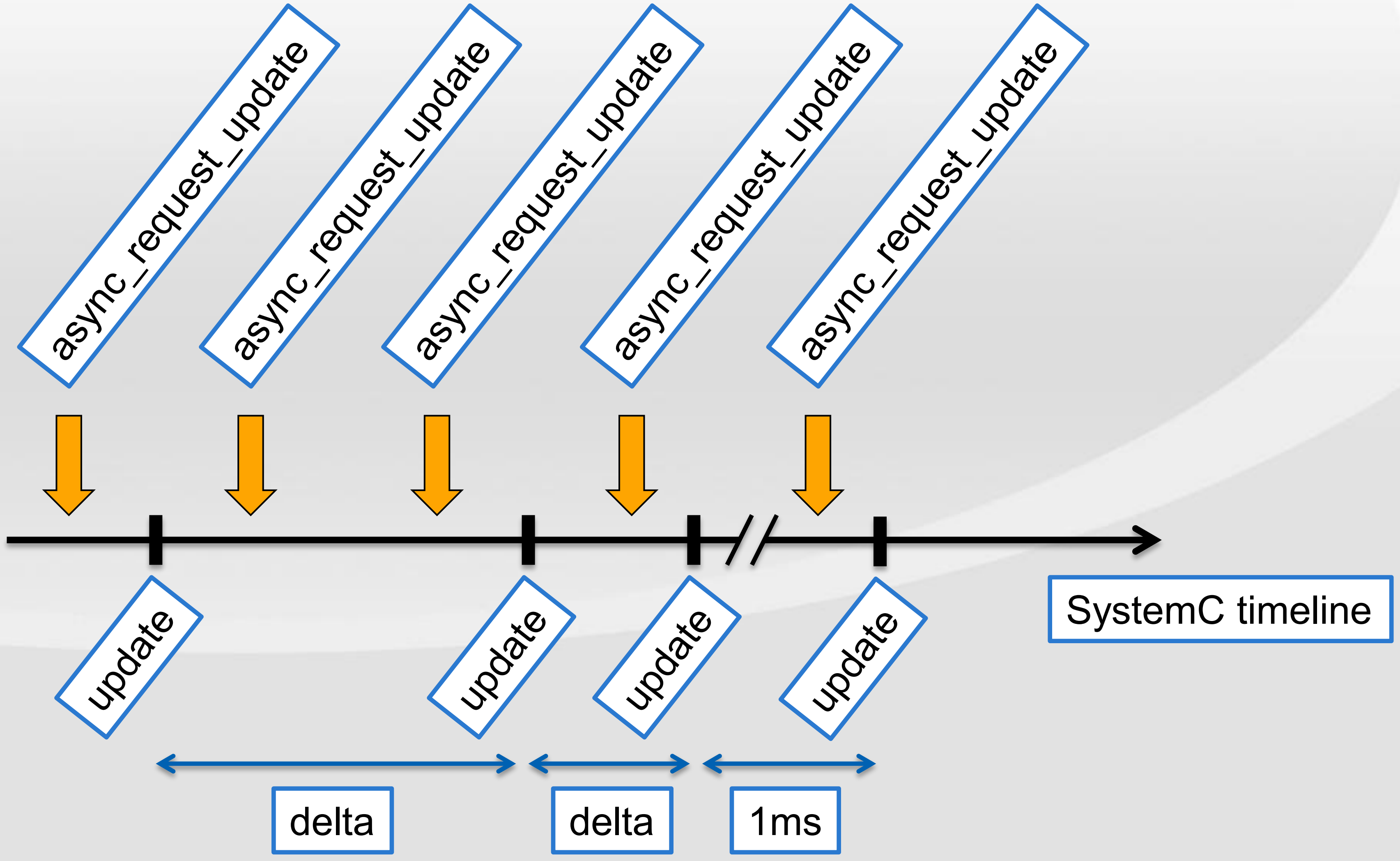
Callable from external threads

async_request_update

```
virtual void write(int value)
{
  ...
  m_next_value = value;
  async_request_update();
  ...
}
```

```
virtual void update()
{
  ...
  if (m_next_value != m_current_value)
  {
    m_current_value = m_next_value;
    m_value_changed_event.notify(SC_ZERO_TIME);
  }
  ...
}
```

async_request_update



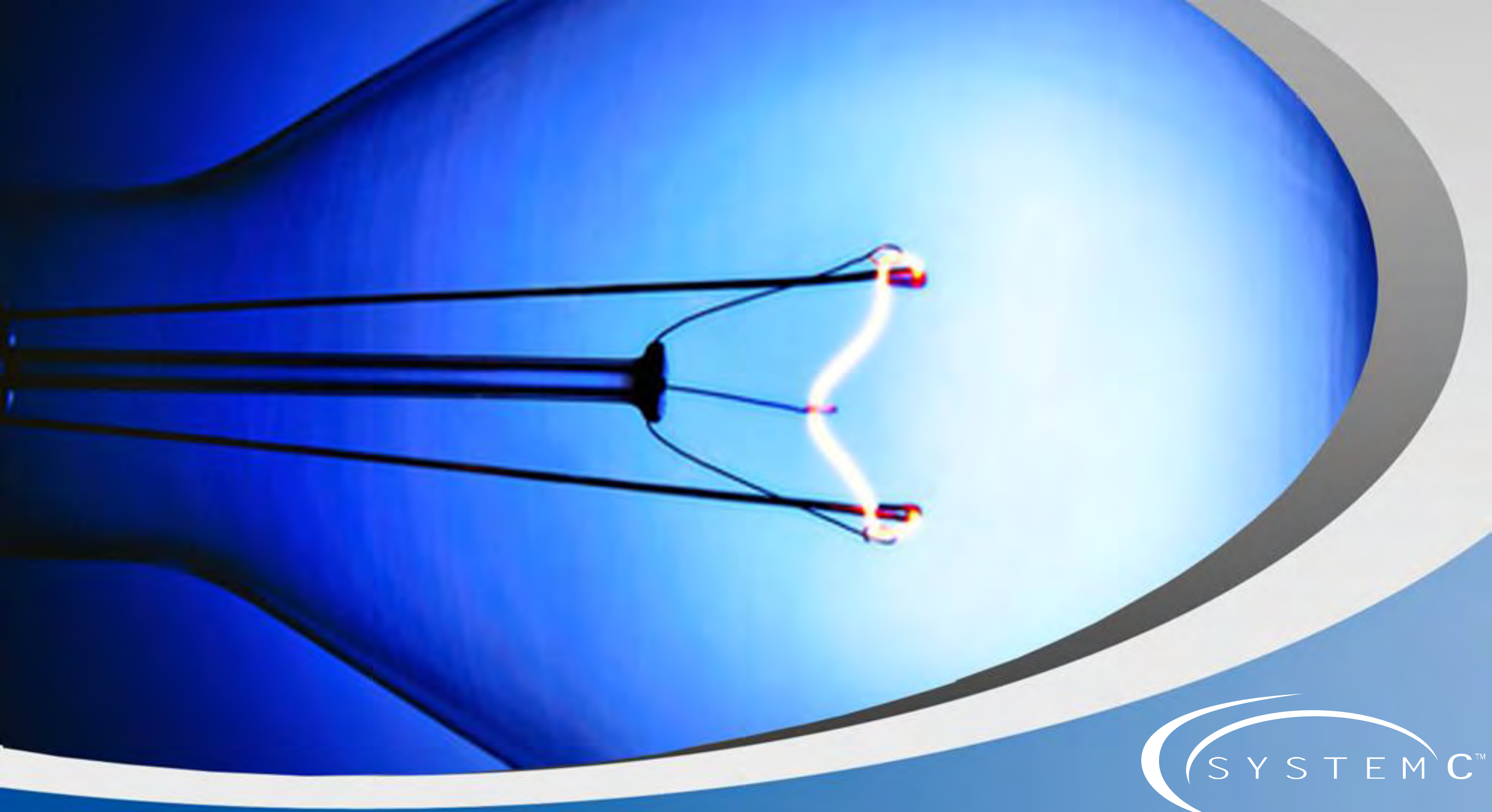
Shared Memory

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
virtual void write(int value) {
  pthread_mutex_lock(&mutex);
  m_next_value = value;
  async_request_update();
  pthread_mutex_unlock(&mutex);
}
```

Cannot use sc_mutex

```
virtual void update() {
  pthread_mutex_lock(&mutex);
  if (m_next_value != m_current_value) {
    m_current_value = m_next_value;
    m_value_changed_event.notify(SC_ZERO_TIME);
  }
  pthread_mutex_unlock(&mutex);
}
```



THE END

SYSTEMS INITIATIVE