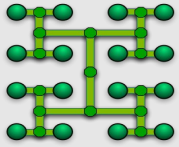
Universidade de Brasília

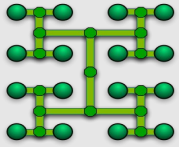
Concorrência

Ricardo Jacobi
Departamento de Ciência da Computação
Universidade de Brasília
jacobi@unb.br



LAICO

Universidade de Brasília



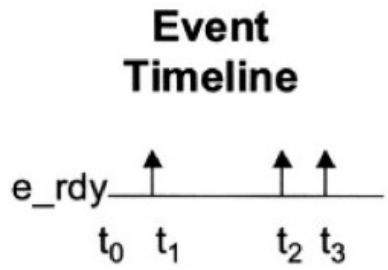
LAICO

Concorrência

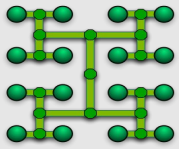
- O modelo de simulação de SystemC é baseado em eventos
- Uma simulação é composta por processos concorrentes que se sincronizam por eventos
- A concorrência segue o modelo de multi-tarefas cooperativo
  - Um SC_METHOD executa até o final e retorna o controle para o simulador
  - Uma SC_THREAD deve voluntariamente liberar o processador para simulador

Eventos

- Um evento é uma mensagem sem conteúdo nem duração que ocorre em determinado instante de tempo
- Um evento não deixa traços nem transmite outras informações além de sua própria ocorrência
- Em SystemC, eventos são gerados por objetos do tipo sc_event
  - Um sc_event pode gerar vários eventos ao longo do tempo



Universidade de Brasília

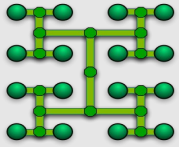


LAICO

Eventos

- Eventos devem ser **observados** para provocar algum resultado
- Processos em SystemC podem suspender sua execução e esperar a ocorrência de um evento para serem reativados
- SystemC permite associar eventos a processos de maneira **estática** ou **dinâmina**
- Declaração:
  - sc_event evento1_evt, evento2_evt, … ;

Universidade de Brasília

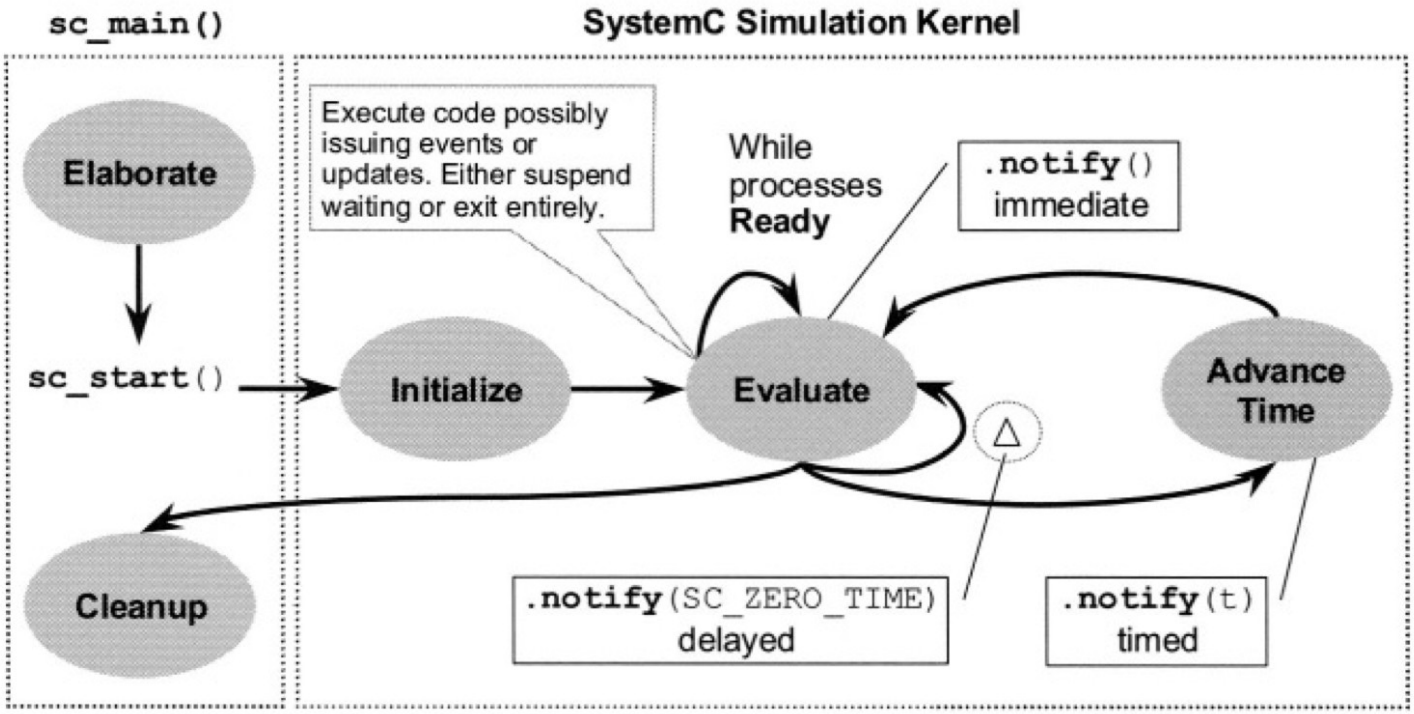


LAICO

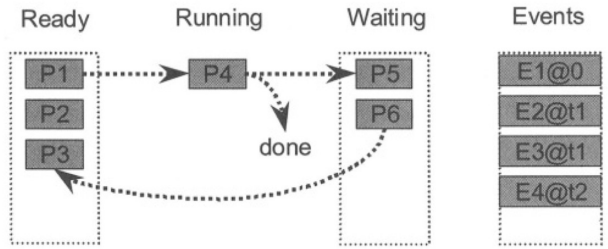
Eventos

- Sensitividade dinâmica
  - Especifica em tempo de execução qual evento deve reativar um processo suspenso
    - Comandos *wait* (*sc_threads*) e *next_trigger* (*sc_methods*)
- Sensitividade estática
  - Especifica em tempo de elaboração qual evento deve reativar um processo suspenso
    - sensitive << evento1 << evento2 << … ;
    - Deve ser especificado imediatamente após a declaração do processo
    - Pode ser temporariamente sobrescrita pela sensitividade dinâmica

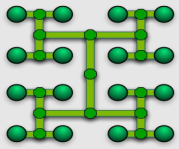
Núcleo de Simulação



Processos e eventos:



Universidade de Brasília

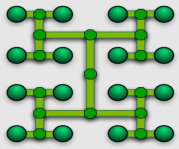


LAICO

Núcleo de Simulação

- Elaboração:
  - Construção do sistema, realizada em tempo de execução
  - sc_start():
    - início de simulação
    - Executa todos os processos (configurável)
  - Avaliação:
    - Processos são retirados aleatoriamente de *Ready*
    - Executam até encerrar ou suspender (vai para *Waiting*)
  - Notificação:
    - Processos são reativados por eventos
      - Imediatos
      - *Delta delay*: processo volta para Ready
      - *Timed*: escalonado para ser reativado em tempo t

Universidade de Brasília

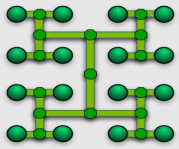


LAICO

SC_THREAD

- Linha de execução chamada apenas uma vez pelo simulador
- Pode retornar o controle de execução para o simulador de duas formas:
  - Encerrando a execução (ex: *return*)
  - Suspendendo a execução com *wait*
    - *wait* pode ocorrer no código da *sc_thread* ou em códigos chamados a partir dela (ex: fifo)
    - *wait* especifica um evento que reativa a *sc_thread*
    - Ao ser reativada, continua execução a partir da instrução seguinte ao *wait*

Universidade de Brasília



LAICO

SC_THREAD...

- Sensitividade dinâmica
  - Eventos que reativam a sc_thread são definidos no comando wait:
    - wait(time); // espera até o tempo time
    - wait(evento); // espera a ocorrência de evento
    - wait(ev1 & ev2 & …); // conjunção de eventos: todos eles
    - wait(ev1 | ev2 | …); // disjunção de eventos: qualquer um
    - wait(timeout, evento); // o que ocorrer antes
    - wait(timeout, ev1 & ev2 & …); // o que ocorrer antes
    - wait(timeout, ev1 | ev2 | …); // o que ocorrer antes
    - wait(); // sensitividade estática

Exemplo

- Considerar 4 processos
  - Inicial em t0
  - Chamam wait() em t1, t2 (menos D) e t3

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

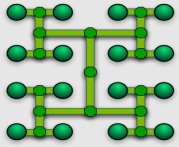
```
Process_B() {
  //@ t0
  stmtB1;
  stmtB2;
  wait(t1);
  stmtB3;
  stmtB4;
  wait(t2);
  stmtB5;
  stmtB6;
  wait(t3);
}
```

```
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);
}
```

** extraído de SystemC from the Ground Up*

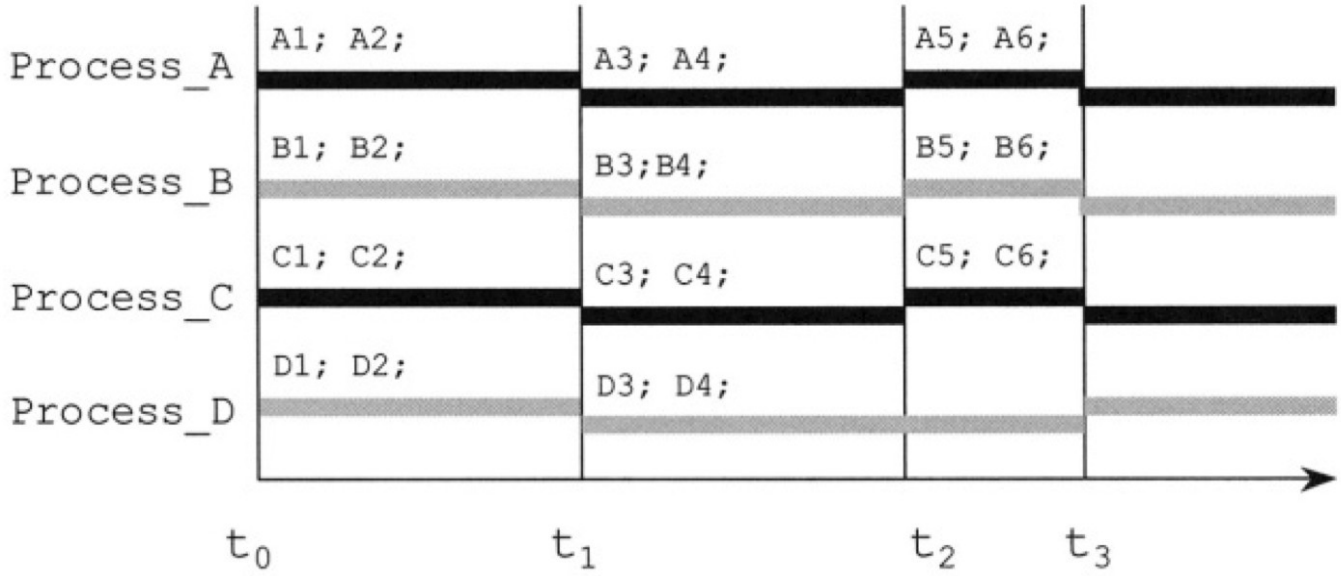
Universidade de Brasília



LAICO

Exemplo...

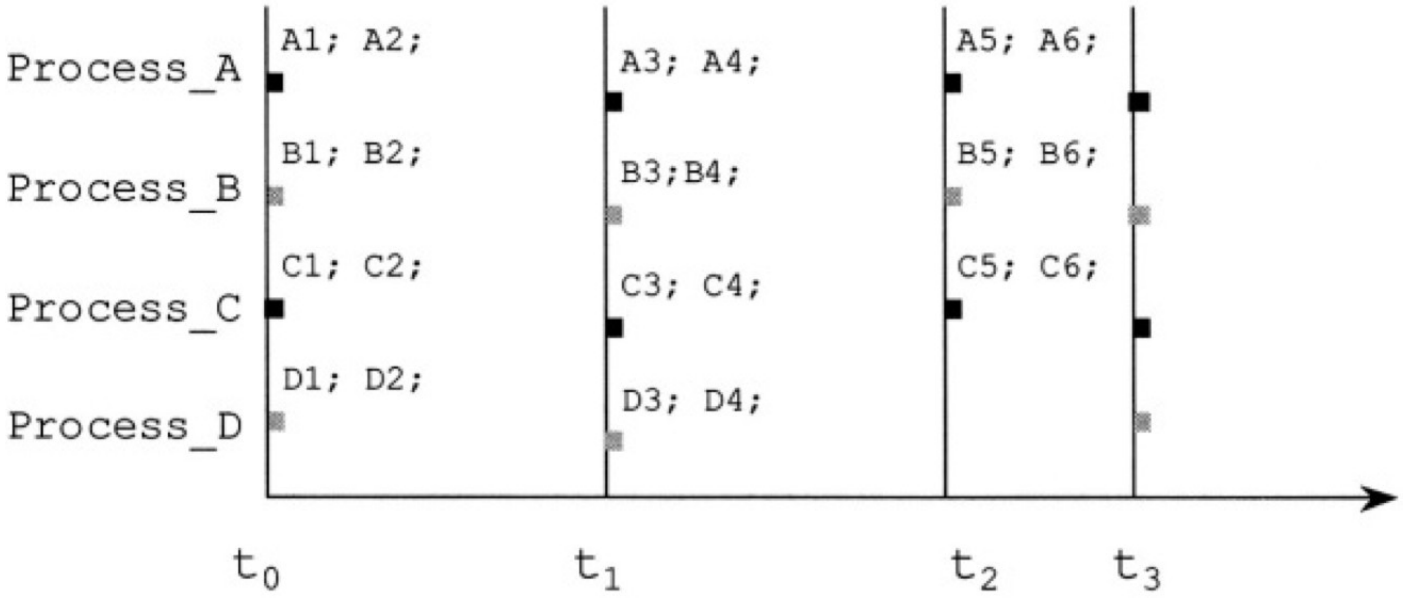
- Atividade percebida
  - Descontinuidades indicam chamadas wait()



** extraído de SystemC from the Ground Up*

Exemplo...

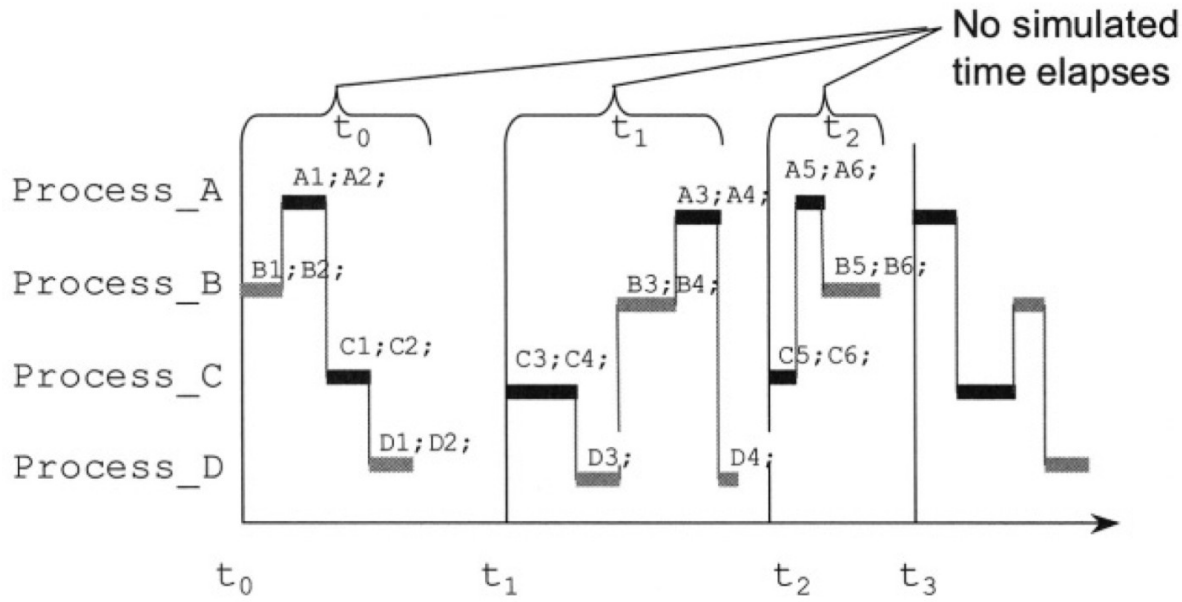
- Atividade simulada
  - Comandos executados instantaneamente, sem avanço no tempo de simulação



** extraído de SystemC from the Ground Up*

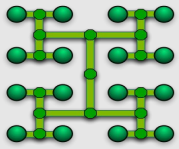
Exemplo...

- Expandindo a linha de tempo:
  - Comandos executados em ordem aleatória sem avanço no tempo de simulação



** extraído de SystemC from the Ground Up*

Universidade de Brasília

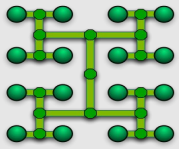


LAICO

Acionando Eventos

- São lançados através do método notify()
- Estilo orientado a objetos:
  - evento.notify(); // notificação imediata
  - evento.notify(SC_ZERO_TIME); // delta ciclo
  - evento.notify(time); // temporal
- Estilo chamada de funções:
  - notify(evento); // notificação imediata
  - notify(evento, SC_ZERO_TIME); // delta ciclo
  - notify(evento, time); // temporal

Universidade de Brasília

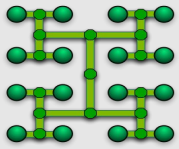


LAICO

Acionando Eventos

- Cada objeto **sc_event** armazena apenas um evento
  - Se um evento mais cedo é criado, cancela o anterior
  - Eventos posteriores ao armazenado são descartados
  - O método **.cancel()** pode ser utilizado para explicitamente cancelar o evento escalonado

Universidade de Brasília



LAICO

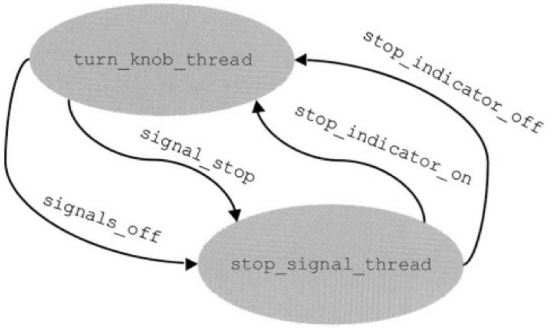
Exemplo notify()

```
...
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 next delta 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 20 femtosecond from now
action.notify(20,SC_FS);
```

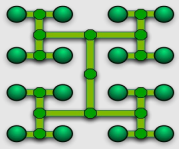
** extraído de SystemC from the Ground Up*

Turn_of_events

- Simular um sistema de acionamento de sinais de um carro
  - turn_left:
    - Envia sinal para ligar o pisca esquerdo
  - turn_rigth:
    - Envia sinal para ligar o pisca direito
  - stop_on:
    - Liga luz de freio
  - turn_off:
    - Desliga todos os sinais



Universidade de Brasília



LAICO

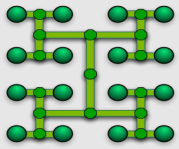
Turn_of_events

```
SC_MODULE(turn_of_events) {
  SC_CTOR(turn_of_events) {
    SC_THREAD(turn_knob_thread);
    SC_THREAD(left_signal_thread);
    SC_THREAD(stop_signal_thread);
    SC_THREAD(right_signal_thread);
  }
    sc_event signal_right, signal_left, signal_stop, signals_off;
    sc_event right_blinking, right_off;
    sc_event left_blinking, left_off;
    sc_event stop_blinking, stop_off;
    void turn_knob_thread(void);
    void right_signal_thread(void);
    void left_signal_thread(void);
    void stop_signal_thread(void);
};//endclass turn_of_events
```

Uma thread para cada controle

Eventos para troca de informações e sincronização

Universidade de Brasília



LAICO

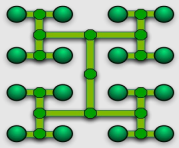
Turn_of_events

```
void turn_of_events::turn_knob_thread(void) {
  enum directions {LEFT='l', RIGHT='r', OFF='f', STOP='s', QUIT='q'};
  bool left_on, right_on, stop_on;
  char direction;
  wait(SC_ZERO_TIME);
  std::cout << "Commands are (case sensitive):" << endl
            << "   l => Left turn signal on" << endl
            << "   r => Right turn signal on" << endl
            << "   s => Stop signal on" << endl
            << "   f => turn all signals oFF" << endl
            << "   q => Quit program" << endl
            << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
  for(;;) {
    std::cout << "Signal command [l/r/s/f/q]: ";
    std::cin >> direction;
```

Thread de acionamento dos sinais

Leitura dos comandos do usuário

Universidade de Brasília

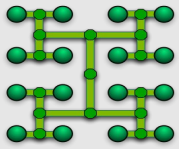


LAICO

Turn_of_events

```
switch (direction) {
 case RIGHT:
   signal_right.notify();
   if (!right_on) wait(right_blinking);
   else std::cout << "INFO: Right signal is already on" << endl;
   right_on = true;
   break;
 …  // case LEFT e case STOP similares
 case OFF:
   signals_off.notify();
   if (right_on && left_on && stop_on) wait(right_off & left_off & stop_off);
   else if (right_on && left_on) wait(right_off & left_off);
   else if (right_on && stop_on) wait(right_off & stop_off);
   else if (left_on  && stop_on) wait(left_off  & stop_off);
   else if (right_on) wait(right_off);
   else if (left_on)  wait(left_off);
   else if (stop_on)  wait(stop_off);
   right_on = left_on = stop_on = false;
   break;
 case QUIT:
   std::cout << "Quiting" << std::endl;
   sc_stop();
   return;
   break;
  }//endswitch
 }//endforever
}//end turn_knob_thread()
```

Universidade de Brasília



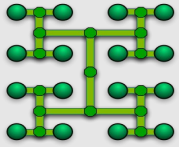
LAICO

Turn_of_events

```
void turn_of_events::right_signal_thread(void) {
  for(;;) {
    wait(signal_right);
    std::cout << "Turning right ->->->" << std::endl;
    right_blinking.notify();
    wait(signals_off);
    std::cout << "Right off     ------" << std::endl;
    right_off.notify();
  }   //endforever
}  //end right_signal_thread()

void turn_of_events::stop_signal_thread(void) {
  for(;;) {
    wait(signal_stop);
    std::cout << "Stopping      !!!!!!" << std::endl;
    stop_blinking.notify();
    wait(signals_off);
    std::cout << "Stop off      ------" << std::endl;
    stop_off.notify();
  }  //endforever
}  //end right_signal_thread()
```

Universidade de Brasília

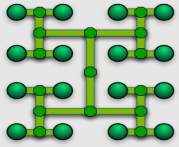


LAICO

Turn_of_events

- Questão de sincronização:
  - Qual a ordem com que os processos são iniciados afeta o funcionamento do programa
  - Se uma notificação é enviada antes que um processo esteja esperando por ela, ela é perdida
  - Verificar exemplo no código

Universidade de Brasília

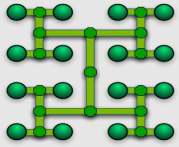


LAICO

SC_METHOD

- Similares a métodos em orientação a objetos
- Ciclo de execução:
  - Depois de iniciado, entra na *pool* de espera
  - Quando um evento estático ou dinâmico ocorrer, o simulador o coloca na pool de processos prontos para executar
  - Executa até o final, sem interrupção
  - Volta ao final para a *pool* de espera

Universidade de Brasília

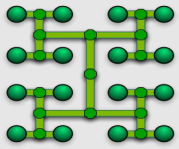


LAICO

SC_METHOD

- Não podem chamar *wait()*
- Dados locais:
  - Em sc_thread os dados locais persistem até o final
  - Em sc_method os dados locais são voláteis
  - Tipicamente, sc_method utiliza dados e sinais definidos no módulo, enquanto sc_thread usa dados locais

Universidade de Brasília



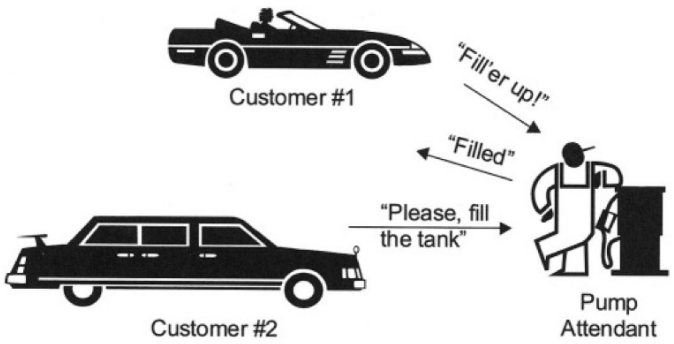
LAICO

SC_METHOD

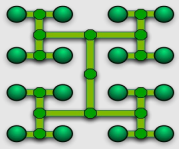
- Sensitividade dinâmica implementada a partir da chamada **`next_trigger()`**
  - Temporariamente mascara a sensibilidade estática
- Sintaxe idêntica ao wait():
  - `next_trigger(time);  // espera até o tempo time`
  - `next_trigger(evento);  // espera a ocorrência de evento`
  - `next_trigger(ev1 & ev2 & …); // conjunção: todos eles`
  - `next_trigger(ev1 | ev2 | …);  // disjunção: qualquer um`
  - `next_trigger(timeout, evento);  // o que ocorrer antes`
  - `next_trigger(timeout, ev1 & ev2 & …);// o que ocorrer antes`
  - `next_trigger(timeout, ev1 | ev2 | …);// o que ocorrer antes`
  - `next_trigger(); // sensitividade estática`

Sensitividade Estática

- Exemplo *Gas Station*
  - Versão simplificada:
    - Um atendente
    - Dois clientes
    - Atendente acionado por eventos de requisição
    - Um cliente responde ao evento tanque cheio
    - Outro cliente sem sensitividade estática



Universidade de Brasília



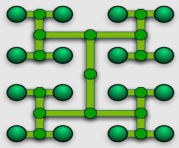
LAICO

Gas Station

- Declaração do módulo

```
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();
};
```

Universidade de Brasília



LAICO

Gas Station

```
…
void gas_station:: customer1_thread() {
  for (;;) {
    wait(EMPTY_TIME);
    cout << "Customer1 needs gas" << endl;
    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()
```

Exercício: analisar o exemplo *gas_station* do capítulo 7 do livro

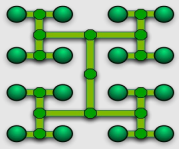
dont_initialize

- Método que previne a execução de um processo durante a fase de inicialização
- Ex:

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

Obs: usar com cuidado, pois um processo pode não ser chamado nunca.
=> O uso de sensitividade estática garante sua chamada

Universidade de Brasília



LAICO

sc_event_queue

- Permite o escalonamento de diversos eventos ao longo do tempo
  - Em contraste com sc_event, que permite apenas um evento por variável
  - Não suporte notificação imediata
  - .cancel_all() remove todos os eventos da fila
  - Exemplos =>

```
…
sc_event_queue  action;
sc_time now(sc_time_stamp()); //observe current time
action.notify(20, SC_MS);//schedule for 20 ms from now
action.notify(1.5,SC_NS);//another for 1.5 ns from
                         //now
action.notify(1.5,SC_NS);//another identical action
action.notify(3.0,SC_NS);//another for 3.0 ns from
                          //now
action.notify(SC_ZERO_TIME);//for next delta cycle
action.notify(1,SC_SEC);//for 1 sec from now
action.cancel_all(); // cancel all actions entirely
…
```