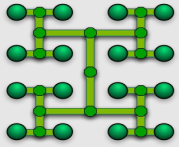
Universidade de Brasília

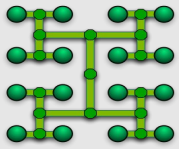
Canais

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



LAICO

Universidade de Brasília



LAICO

Canais Primitivos

- Canais primitivos não contém módulos, derivam de ***sc_prim_channel***
- Canais hierárquicos podem conter uma estrutura de módulos, derivam de sc_channel
- Quando usar:
  - use canais primitivos quando for necessário o esquema *request_update/update* para simular paralelismo
  - use canais hierárquicos se seus canais contém módulos, processos, ou outros canais
  - havendo escolha, prefira canais primitivos pois consomem menos recursos de memória

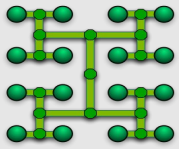
sc_mutex

- *sc_mutex* é um exemplo de canal primitivo que não utiliza o protocolo *request_update/update*
- Usado para obter acesso exclusivo a recursos
- Um mutex é um objeto da classe sc_mutex e pode estar em um de dois estados mutualmente exclusivos: *locked* ou *unlocked*
  - Apenas um processo pode travar o mutex
  - Apenas o processo que travou pode destravar o mutex
  - Uma vez liberado, o mutex pode ser travado por outro processo

sc_mutex

- Métodos:
  - `virtual int lock()`:
    - Se o mutex está liberado, `lock()` trava o mutex e retorna
    - Se o mutex estiver travado, `lock()` suspende o processo que o chamou até a liberação do mutex
    - Uma vez liberado o mutex, todos os processos suspensos através de `lock()` são reativados
    - O processo que executar primeiro deverá obter o *lock* e travar o mutex
    - Caso múltiplos processos tentem travar o mutex no mesmo ciclo delta, a escolha do processo que obterá o *lock* é indeterminada

Universidade de Brasília



LAICO

sc_mutex

- Métodos:
  - `virtual int trylock()`:
    - Função não bloqueante que testa o lock
    - Se o mutex está liberado, trava o mutex e retorna 0
    - Se o mutex está travado, retorna -1
  - `virtual int unlock()`:
    - Se o mutex está liberado, retorna -1
    - Se o mutex foi travado por outro processo, retorna -1
    - Senão, destrava o mutex e retorna 0
      - Todos os processos suspensos pelo mutex são reativados e o primeiro a executar adquire o mutex, sendo que os outros irão suspender novamente
      - Ordem de reativação aleatória

sc_mutex

- Um exemplo aplicação de mutex em hardware é no controle de acesso a um barramento.
- Ex:

```
class bus {
   sc_mutex bus_access;
   ...
   void write(int addr, int data) {
      bus_access.lock();
      // realiza a escrita
      bus_access.unlock();
   }
   ...
}; //endclass
```

sc_mutex

- Com *trylock()* é possível um sc_method acessar o mutex:

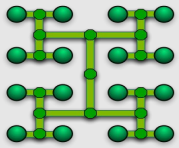
```
void grab_bus_method() {
   if (bus_access.trylock()) {
      /* access bus */
   } //endif
}
```

  - uma limitação do mutex é a falta de um evento que sinalize sua liberação
  - se um sc_method chamar repetidamente *trylock*, não libera a CPU e a simulação trava

sc_semaphore

- sc_semaphore é uma extensão de sc_mutex para um conjunto de recursos compartilhados
- Ao criar um sc_semaphore, é especificada a quantidade de recursos a serem compartilhados
- Ao adquirir um recurso, o número de recursos é decrementado
- Ao liberar um recurso, seu número é incrementado

Universidade de Brasília

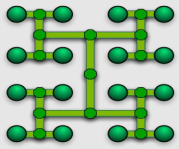


LAICO

sc_semaphore

- Métodos:
  - `sc_semaphore nome(int n);`
    - cria um semáforo com *n* recursos
  - `virtual int wait():`
    - se o valor de semáforo for maior que zero, decrementa e retorna
    - se o valor do semáforo for zero, o processo que chamou wait suspende até que um recurso seja liberado
    - wait() retorna sempre zero
  - `virtual int trywait():`
    - `similar ao wait, não bloqueante`
    - retorna -1 se não existem recursos disponíveis

Universidade de Brasília



LAICO

sc_semaphore

- Métodos:
  - `virtual int post()`
    - incrementa o valor do semáforo. Se existem processos suspensos aguardando recursos, apenas um processo deverá ter acesso ao novo recurso, decrementando o valor para zero de novo. Os outros processos devem suspender novamente
    - retorna sempre zero
  - `virtual int get_value()`
    - retorna o número de recursos total do semáforo

sc_semaphore

- Um exemplo de aplicação em hardware seria o controle de acesso a uma memória multiporta.

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  …
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, int data) (
    write_ports.lock();
    // perform write
    write_ports.unlock();
  }
  …
}; //endclass
```

um semáforo pode ser utilizado para indicar
o número de portas de leitura e escrita simultâneas

sc_fifo

- Canal que modela uma fila para transmissão de dados
  - dados são escritos e lidos na mesma ordem
- Por default tem tamanho 16
  - tamanho pode ser definido no construtor
- Acessos bloqueante e não-bloqueante
  - fila cheia bloqueia operação de escrita
  - fila vazia bloqueia operação de leitura

sc_fifo

- Construtores:
  - `sc_fifo<T> nome(int size)`
    - cria uma fila com size elementos do tipo T
    - é gerado um nome default
  - `sc_fifo<T> nome(const char * n, int size = 16)`
    - cria uma fila com o nome indicado e size elementos do tipo T
    - se omitido, o tamanho default da fila é 16 elementos

sc_fifo

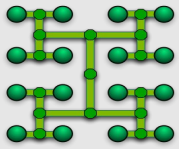
- Métodos:
  - `virtual void read(T&);`
    `virtual T read();`
    `virtual bool nb_read(T&);`
    - retornam o valor mais antigo escrito na fila e o removem da fila
    - *read()* é bloqueante, se a fila estiver vazia, a thread que chamou *read()* é suspensa até que haja uma escrita
    - *nb_read(&T)* é não-bloqueante. Retorna **false** se não há elementos na fila e **true** caso contrário

sc_fifo

- Métodos:
  - `virtual void write(const T&);`
    `virtual bool nb_write(const T&);`
    - escrevem o valor passado como parâmetro na fila.
    - *write()* é bloqueante. Se a fila estiver cheia, a *thread* que chamou *write()* é suspensa até que haja uma leitura
    - *nb_write(&T)* é não-bloqueante. Retorna **false** se a fila está cheia e não foi escrito o dado nela, **true** caso contrário
  - `operator=`
    - o operador atribuição é redefinido de forma a funcionar como:

```
sc_fifo<T>& operator= (const T&a )
      { write(a); return *this; }
```

Universidade de Brasília

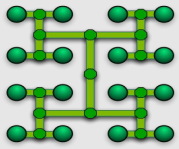


LAICO

sc_fifo

- Métodos:
  - `virtual int num_available() const;`
    - retorna o número de elementos disponíveis para leitura na fila, ignora novos valores escritos durante o ciclo delta corrente, mas considera valores lidos no mesmo
  - `virtual int num_free() const;`
    - retorna o número de locais disponíveis para escrita na fila. Ignora valores lidos no ciclo D atual, mas considera valores escritos no ciclo D corrente
  - `virtual void dump(std::ostream& = std::cout) const;`
  - `virtual void print(std::ostream& = std::cout) const;`
    - dump imprime o nome e conteúdo da fila
    - print apenas o conteúdo

Universidade de Brasília



LAICO

Exemplo sc_fifo

```
SC_MODULE(M) {
sc_fifo<int> fifo;

SC_CTOR(M) : fifo(4) {
SC_THREAD(T);  }
void T() {
int d;
fifo.write(1);
fifo.print(std::cout); //1
fifo.write(2);
fifo.print(std::cout); //1 2
fifo.write(3);
fifo.print(std::cout); //1 2 3
std::cout << fifo.num_available(); //0 values
available to read
std::cout << fifo.num_free();  //1 free slot
fifo.read(d); // read suspends and returns in
the next delta cycle
fifo.print(std::cout); // 2 3
std::cout << fifo.num_available(); // 2
values available to read
std::cout << fifo.num_free(); // 1 free slot
fifo.read(d);
fifo.print(std::cout); // 3
fifo.read(d);
fifo.print(std::cout); // empty
std::cout << fifo.num_available(); // 0
values available to read
std::cout << fifo.num_free(); // 1 free slot
wait(SC_ZERO_TIME);
std::cout << fifo.num_free(); // 4 free slots
  }
};
```

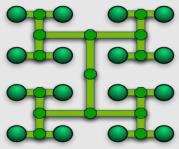
Exemplo *simple_fifo*

- Simple_fifo é uma implementação de uma fila de comunicação que interliga um produtor com um consumidor

```
#include <systemc.h>
class write_if : virtual public sc_interface {
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};
class read_if : virtual public sc_interface {
  public:
    virtual void read(char &) = 0;
    virtual int num_available() = 0;
};
```

declaração das interfaces para acesso à fila

Universidade de Brasília



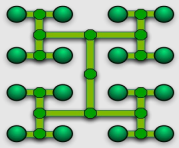
LAICO

Exemplo *simple_fifo*...

```
class fifo : public sc_channel, public write_if, public read_if {
  public:
    fifo(sc_module_name name) : sc_channel(name), num_elements(0),
    first(0) {}
    void write(char c) {
      if (num_elements == max)
        wait(read_event);

      data[(first + num_elements) % max] = c;
      ++ num_elements;
      write_event.notify();
    }
```

Universidade de Brasília

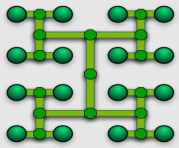


LAICO

Exemplo *simple_fifo*...

```
void read(char &c) {
    if (num_elements == 0)   wait(write_event);
    c = data[first];
    -- num_elements;
    first = (first + 1) % max;
    read_event.notify();
  }
  void reset() { num_elements = first = 0; }
  int num_available() { return num_elements;}
private:
  enum e { max = 10 };
  char data[max];
  int num_elements, first;
  sc_event write_event, read_event;
};
```

Universidade de Brasília

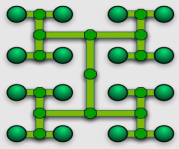


LAICO

simple_fifo::producer

```
class producer : public sc_module  {
  public:
    sc_port<write_if> out;
    SC_HAS_PROCESS(producer);
    producer(sc_module_name name) : sc_module(name)   {
      SC_THREAD(main);
    }
    void main()  {
      const char *str =
       "Visit www.systemc.org and see what SystemC can do for you today!\n";
      while (*str)
        out->write(*str++);
    }
};
```

Universidade de Brasília



LAICO

simple_fifo::consumer

```
class consumer : public sc_module {
  public:
    sc_port<read_if> in;
    SC_HAS_PROCESS(consumer);
    consumer(sc_module_name name) : sc_module(name)    {
      SC_THREAD(main);
    }
    void main()      {
      char c;
      cout << endl << endl;

      while (true) {
        in->read(c);
        cout << c << flush;
        if (in->num_available() == 1)   cout << "<1>" << flush;
        if (in->num_available() == 9)   cout << "<9>" << flush;
      }
    }
};
```