Simpatica

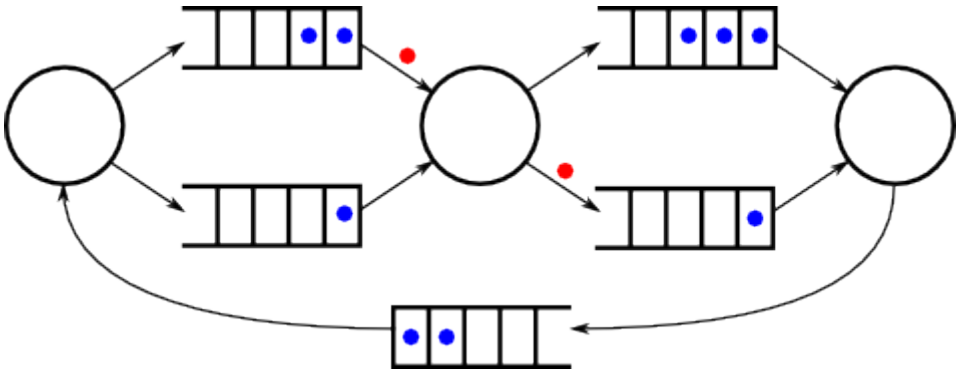
[English version]

SIMPATICA é uma pequena biblioteca de simulação a eventos discretos. Suas principais características são:

- Escrita em linguagem C ANSI.
- Estrutura interna simples, de fácil compreensão e amplamente comentada, permitindo seu uso didático.
- Extensivamente testada em ambientes Linux 32 e 64 bits.
- Baseada no paradigma atores/mensagens.
- Concebida visando um baixo consumo de memória, o que permite criar simulações com dezenas de milhares de entidades ativas (suportou 150.000 tarefas simultâneas em um computador com 4 GB de memória).
- Excelente desempenho, por usar *threads* leves implementadas pela própria biblioteca, que não dependem do escalonador do sistema operacional.
- A fila do escalonador é implementada usando um *heap* binário, o que permite alcançar um bom desempenho em simulações de larga escala.
- Software aberto de livre acesso (licença GNU).

A biblioteca SIMPATICA permite construir simulações a eventos discretos com modelos baseados no paradigma atores/mensagens. Segundo esse paradigma, um modelo de simulação é composto por um conjunto de atores ou tarefas que se comunicam por mensagens. Esta biblioteca suporta três tipos de entidades:

- **Tarefa**: uma tarefa é uma entidade ativa, que tem seu comportamento definido por uma função, da mesma forma que uma thread (threads de usuário são usadas para implementar as tarefas). Cada tarefa tem um identificador único no sistema. As tarefas podem produzir e consumir mensagens, que são transferidas através de filas.
- **Fila**: é um depósito de mensagens ordenado por data de chegada (ordem FIFO). Tarefas podem depositar mensagens nas filas ou retirar mensagens delas. Filas e tarefas são independentes: qualquer tarefa pode consumir/produzir mensagens em qualquer fila.
- **Mensagem**: mensagens são estruturas em C (`struct`) com conteúdo definido pelo programador, o que proporciona uma boa flexibilidade de modelagem.



Arquivos

- Versão 0.7

Interface

Esta biblioteca oferece um conjunto de funções em C ANSI para a construção de modelos de simulação. Por

convenção, a maioria das funções aborta a execução do modelo em caso de erro, enviando uma mensagem de erro para `stderr` e retornando um status 1 para o sistema operacional. Esta abordagem "radical" foi adotada para minimizar o risco de erros não detectados que possam interferir nos resultados da simulação. Por essa mesma razão, a maior parte das funções da biblioteca não tem valor de retorno (ou seja, retorna `void`).

```
void init_simulation (int maxTasks, int maxQueues)
```

Inicializa as estruturas internas necessárias para cada simulação. Esta função deve ser chamada somente uma vez, sempre no inicio do programa principal (função `main`) do modelo. Os parâmetros `maxTasks` e `maxQueues` definem o numero máximo de tarefas e de filas que poderão ser criadas pelo modelo (definem o tamanho das estruturas de dados a alocar).

```
void run_simulation (double maxTime)
```

Executa a simulação até que o relógio de simulação atinja o valor `maxTime`. Deve ser chamada assim que todas as tarefas tenham sido definidas. Caso esta função seja chamada novamente após a conclusão de uma simulação, a simulação é retomada a partir do ponto onde havia parado na chamada anterior; isso permite executar uma simulação por etapas:

```
... // inicializa e cria tarefas

run_simulation (1000) ; // executa a simulação no intervalo t = [0 - 1000)

... // trata resultados parciais

run_simulation (2000) ; // continua a simulação no intervalo t = [1000 - 2000)

... // trata resultados parciais
```

```
void kill_simulation ()
```

Encerra uma simulação, limpando todas as definições de tarefas, mensagens e estruturas de dados alocadas em memória. Esta chamada permite reiniciar a biblioteca e assim realizar várias simulações dentro do mesmo programa C.

```
void trace_interval (double startTime, double stopTime)
```

Gera mensagens de *tracing* quando o relógio de simulação estiver dentro do intervalo [`startTime`, `stopTime`]. As mensagens de tracing informam dados sobre os eventos processados pela biblioteca, e são enviadas para a saída padrão (`stdout`).

ATENÇÃO: a biblioteca usa as funções C `printf` e `sprintf` para gerar as mensagens de *tracing*. Essas funções podem ter um consumo elevado de memória, mais precisamente da pilha alocada para cada tarefa (pois o *tracing* é feito dentro do contexto das tarefas). Assim, caso o programa com *tracing* se torne instável, talvez seja necessário aumentar o tamanho das pilhas das tarefas.

```
uint task_create (void (*taskBody)(void *),
                  void* startArg,
                  int stackPages)
```

Cria uma nova tarefa. O parâmetro `taskBody` indica uma função que contém o código da tarefa; essa função receberá o parâmetro `startArg` ao iniciar sua execução. Cada tarefa é identificada no modelo por um ID inteiro positivo único (o retorno desta chamada). O parâmetro `stackPages` indica o número de páginas de memória a alocar para a pilha da tarefa e seu valor deve ser maior que 0 (zero). Retorna o ID da tarefa recém-criada. Os IDs são inteiros positivos criados em seqüência (1, 2, 3, ...).

É importante observar que pilhas muito pequenas podem levar a erros de acesso à memória ou comportamento errático do modelo, enquanto pilhas muito grandes consomem mais memória e assim limitam o número máximo de tarefas que o modelo pode criar. O tamanho da pilha deve ser estimado em função do comportamento da tarefa: quantidade de variáveis locais, funções chamadas pela tarefa, etc. Modelos simples geralmente funcionam bem com pilhas com 1 a 3 páginas de memória, mas isso deve ser estimado caso a caso.

Obs: a função `printf` e suas congêneres podem usar muito espaço de pilha; tarefas que imprimam muito podem precisar de pilhas maiores. Caso a funcionalidade de Tracing seja utilizada, todas as tarefas precisarão de mais espaço de pilha (esse problema deverá ser sanado em futuras versões).

```
void task_exit ()
```

Encerra a tarefa corrente (em execução), liberando seus recursos. Deve ser chamada no final do código de cada tarefa, para liberar os recursos por ela utilizados.

```
void task_destroy (int task_id)
```

Destrói a tarefa indicada como parâmetro, liberando os recursos utilizados por ela.

```
void task_sleep (double t)
```

A tarefa corrente vai dormir durante `t` unidades de tempo simulado.

```
void task_passivate ()
```

A tarefa corrente vai dormir indefinidamente, até ser acordada por outra tarefa através da chamada `task_activate` (vide abaixo).

```
void task_activate (int task_id, double waitTime)
```

Acorda a tarefa indicada por `task` dentro de `waitTime` unidades de tempo simulado, a partir do instante atual. O parâmetro `waitTime` pode ser zero, para acordar a tarefa no instante atual. Esta chamada somente acorda tarefas que foram dormir através da chamada `task_passivate`, não tendo efeito sobre as demais tarefas.

```
int task_id ()
```

Retorna o ID (identificador único) da tarefa corrente, que é um inteiro positivo (1, 2, 3, ...). Se chamada fora de uma tarefa (por exemplo, no programa principal ou no escalonador), retorna 0.

```
double time_now ()
```

Informa o valor atual do relógio simulado.

```
int queue_create (int capacity, int policy)
```

Cria uma nova fila com a capacidade (número máximo de mensagens) e política de ordenamento indicadas. Na versão atual, ambos os parâmetros são ignorados: todas as filas são ilimitadas e têm comportamento FIFO.

```
int queue_destroy (int queue_id)
```

Destrói a fila indicada, eliminando todas as mensagens nela contidas e as estruturas de dados que a representam.

```
void queue_stats (uint queue_id,
                  uint *size,
```

```
                  uint *max,
                  double *mean,
                  double *var,
                  ulong *put,
                  ulong *got)
```

Fornece informações estatísticas relativas à fila indicada, computadas a partir do início da simulação:

- `size`: número de mensagens atualmente na fila
- `max`: número máximo de mensagens na fila
- `mean`: número médio de mensagens na fila
- `var`: variância da média de mensagens na fila
- `put`: número de mensagens depositadas na fila (via `msg_put`)
- `got`: número de mensagens retiradas da fila (via `msg_get`)

Para cada informação, deve ser informado o endereço da variável que irá recebê-lo, ou NULL (0) para ignorá-la. Por exemplo, a chamada abaixo informa o número atual de mensagens na fila `q1`:

```
queue_stats (q1, &size, 0, 0, 0, 0, 0) ;
```

```
void* msg_create (short size)
```

Cria uma nova mensagem com tamanho `size` bytes. Cada mensagem criada recebe um ID único no sistema. O tipo da mensagem é definido pelo usuário, e pode conter as informações que este bem desejar. Normalmente mensagens são definidas como `struct` contendo os campos necessários ao modelo de simulação. Caso não haja necessidade de conteúdos específicos, o parâmetro `size` pode ser igual a zero. Internamente, a biblioteca mantém várias informações em cada mensagem, que podem ser consultadas através da função `msg_attr` (vide abaixo).

Retorna um ponteiro para a mensagem criada.

```
void msg_destroy (void *msg)
```

Destrói a mensagem indicada. Todas as mensagens devem ser destruidas ao encerrar sua vida útil, para liberar a memória utilizada e assim permitir simulações maiores e/ou mais longas, sem esgotar a memória do computador.

```
void msg_put (int queue_id, void* msg)
```

Coloca a mensagem indicada no final da fila indicada.

```
void* msg_get (void *msg)
```

Retira a mensagem indicada da fila onde ela se encontra, retornando um ponteiro para a mensagem.

```
void* msg_wait (int queue_id, double timeOut)
```

Espera uma mensagem na fila indicada. A tarefa fica suspensa até receber uma mensagem ou esgotar o tempo de espera `timeOut` (para esperar indefinidamente, basta informar um valor de time-out muito elevado, usando a constante `INFINITY`). Retorna um ponteiro para a mensagem recebida ou `NULL`, no caso de ocorrer um *time-out*. Importante: a mensagem recebida não é retirada da fila.

```
void* msg_first (int queue_id)
void* msg_last  (int queue_id)
void* msg_prev  (void* msg)
```

```
void* msg_next  (void* msg)
```

Permitem navegar em uma fila de mensagens. Retornam um ponteiro para uma mensagem na fila ou NULL, caso não exista a mensagem solicitada.

```
void msg_attr (void *msg,
               long *id,
               double *birth,
               double *sent,
               long *creator,
               long *sender,
               int *queue)
```

Informa os seguintes atributos da mensagem indicada:

- `id` : identificador único da mensagem (ID)
- `birth`: data de criação da mensagem
- `sent`: data de último envio da mensagem
- `creator`: ID da tarefa que criou a mensagem
- `sender`: ID da tarefa que enviou a mensagem por último
- `queue`: ID da fila onde a mensagem se encontra, ou 0 (em nenhuma fila)

Para cada atributo, deve ser informado o endereço da variável que irá recebê-lo, ou NULL para ignorá-lo (vide chamada `queue_stats`).

Forma de uso

O uso desta biblioteca é bastante simples: basta escrever um programa C, usando as funções da biblioteca para definir o modelo e os parâmetros da simulação, compilar o programa junto com a biblioteca e executar:

```
$ cc simpatica.c modelo.c
$ a.out
```

O arquivo `simpatica.c` contém a implementação das chamadas da biblioteca e o gerenciamento do escalonador. O arquivo `model.c` contém o modelo de simulação em si.

Exemplo de simulação

Eis abaixo um exemplo de simulação no qual 1000 tarefas `source` enviam mensagens a uma mesma fila `queue`; a tarefa `sink` retira as mensagens da fila, calcula o tempo decorrido entre a produção e o consumo da mensagem e a destrói. Ao final da simulação, o programa calcula o tempo médio decorrido entre a produção e o consumo das mensagens e informa o tamanho médio e desvio da fila de mensagens. O código-fonte do modelo (arquivo `modelo.c`):

modelo.c

```
#include <stdio.h>
#include <stdlib.h>
#include "simpatica.h"

int queue ;
```

```
// variaveis para o calculo do tempo medio entre geracao e consumo das msgs
long num_msgs = 0 ;
double soma_tempos = 0.0 ;

// mensagens sao structs com conteudo definido pelo programador
typedef struct msg_t
{
   int value ;
} msg_t ;

// corpo das tarefas "source"
void sourceBody (void *arg)
{
   msg_t *msg ;

   for (;;)
   {
      // cria uma nova mensagem
      msg = (msg_t*) msg_create (sizeof (msg_t)) ;

      // preenche a mensagem com um valor aleatorio
      msg->value = random ();

      // coloca a mensagem na fila "queue"
      msg_put (queue, msg) ;

      // dorme durante um tempo aleatorio
      task_sleep (15 + random() % 5) ;
   }
}

// corpo da tarefa "sink"
void sinkBody (void *arg)
{
   msg_t *msg ;
   double data_criacao ;

   for (;;)
   {
      // espera uma mensagem na fila e a retira
      msg = (msg_t*) msg_get (msg_wait (queue, INFINITY)) ;

      // obtem a data de criacao da mensagem
      msg_attr (msg, 0, &data_criacao, 0, 0, 0, 0) ;

      // simula o tempo gasto no tratamento da mensagem
      task_sleep (1) ;

      // acumula tempos
      soma_tempos += (time_now() - data_criacao) ;
      num_msgs ++ ;

      // destroi a mensagem recebida (libera recursos)
      msg_destroy (msg) ;
   }
}
```

```
int main ()
{
   int i ;
   double media, variancia ;

   // prepara a simulacao para 1001 tarefas e uma fila
   init_simulation (1001,1) ;

   // cria 1000 tarefas "source"
   for (i=0; i< 1000; i++)
     task_create (sourceBody, NULL, 2) ;

   // cria uma tarefa "sink"
   task_create (sinkBody, NULL, 2) ;

   // cria uma fila "queue"
   queue = queue_create (0, 0) ;

   // executa a simulacao ate 50000 segundos
   run_simulation (50000) ;

   // imprime resultados obtidos
   printf ("Tempo medio entre producao e consumo das mensagens: %0.3f\n",
           soma_tempos / num_msgs) ;

   // imprime o tamanho medio da fila e seu desvio padrão
   queue_stats (queue, 0, 0, &media, &variancia, 0, 0) ;
   printf ("Tamanho da fila: media %0.3f, variancia %0.3f\n", media,
variancia) ;

   // libera os recursos da simulacao
   kill_simulation () ;

   exit(0) ;
}
```

A compilação do modelo é feita através da seguinte linha de comando:

```
$ cc simpatica.c modelo.c
```

A execução do modelo gera os seguintes resultados:

```
$ a.out
-- Simulation initialized, Simpatica version 0.7, 06/out/2007 (mem: 16Kb)
-- Simulation in interval t=[0.000, 50000.000), 1001 tasks
-- Simulation time:  5000.000,     299593 events,  10% done in     1 secs (mem:
43693Kb)
-- Simulation time: 10000.000,     598584 events,  20% done in     1 secs (mem:
66271Kb)
-- Simulation time: 15000.000,     897715 events,  30% done in     2 secs (mem:
88859Kb)
-- Simulation time: 20000.000,    1196897 events,  40% done in     3 secs (mem:
111451Kb)
-- Simulation time: 25000.000,    1496007 events,  50% done in     3 secs (mem:
```

```
134038Kb)
-- Simulation time: 30000.000,    1795129 events,  60% done in     4 secs (mem:
156626Kb)
-- Simulation time: 35000.000,    2094218 events,  70% done in     4 secs (mem:
179211Kb)
-- Simulation time: 40000.000,    2393259 events,  80% done in     5 secs (mem:
201792Kb)
-- Simulation time: 45000.000,    2692314 events,  90% done in     6 secs (mem:
224375Kb)
-- Simulation time: 50000.000,    2991445 events, 100% done in     6 secs (mem:
246963Kb)
-- Simulation completed in 6 seconds (mem: 246963Kb)
Tempo medio entre producao e consumo das mensagens: 24583.696
Tamanho da fila: media 1445994.474, variancia 696500121941.198
-- Simulation killed (mem: 0Kb)
```

No exemplo acima, as linhas que iniciam com "`--`" contém mensagens de acompanhamento da simulação geradas automaticamente pela biblioteca. Todas as mensagens geradas pela biblioteca são enviadas para o arquivo de saída de erro padrão `stderr`, e podem ser desviadas usando redireções de shell. Além disso, uma opção de compilação permite inibir a geração dessas mensagens (vide abaixo).

Opções de compilação

Existem algumas opções de compilação que permitem ativar testes e mensagens adicionais. Essas opções são úteis para a depuração das funcionalidades e mecanismos internos da biblioteca em si, não tendo muita utilidade para a depuração dos modelos de simulação.

- `HEAPCHECK`: verifica a integridade da fila do escalonador a cada operação.
- `HEAPDEBUG`: gera mensagens na tela sobre as operações efetuadas na fila do escalonador.
- `STACKCHECK`: verifica a integridade da pilha da tarefa corrente a cada operação.
- `STACKDEBUG`: gera mensagens detalhadas sobre uso da pilha da tarefa corrente.
- `NOTESTS`: inibe todos os testes de consistência. Pode ser usado para melhorar um pouco o desempenho da simulação, assim que ela estiver plenamente testada e sem erros.
- `QUIET`: inibe a geração de mensagens de acompanhamento em `stderr`.

A forma de uso dessas opções é a seguinte:

```
$ cc -DQUIET -DNOTESTS simpatica.c modelo.c
```

Concorrência

Em sistemas com tarefas executando simultaneamente, existe um risco de operações concorrentes sobre as mesmas estruturas de dados apresentarem condições de corrida, produzindo erros ou resultados inconsistentes. Nesta biblioteca, as tarefas são implementadas como threads leves cooperativas. De acordo com essa técnica de implementação, a execução de uma thread só é suspensa em pontos específicos do código, quando estritamente necessário.

Nesta biblioteca, uma tarefa em execução só perde o processador quando solicita uma operação que possa fazer avançar o tempo simulado, ou seja: `task_sleep` e `msg_wait`. Todas as demais operações são realizadas sem bloqueio ou perda do processador, portanto sem risco de concorrência com as outras tarefas do modelo. Portanto, o estado (conteúdo) das filas e de variáveis globais só poderá mudar durante a execução dessas duas operações, e nunca no restante do código.

From:
https://wiki.inf.ufpr.br/maziero/ - **Prof. Carlos Maziero**

Permanent link:
https://wiki.inf.ufpr.br/maziero/doku.php?id=software:simulacao

Last update: **2020/08/18 19:48**