Gerador de Números Aleatórios Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article contains the implementation of several random number generator algorithms in literary programming and is intended to be used with Weaver Game Engine. However, it can also be used as an independent API in other projects. All different algorithms are encapsulated in the same API letting programmers to easily change the algorithm using macro definitions. We also describe here how the algorithms perform in some statistical tests and in benchmarks running in Linux, OpenBSD, Windows and Web Assembly.*

Resumo: *Este artigo contém a implementação de vários algoritmos geradores de números pseudo-randômicos em programação literária e ele foi feito para ser usado junto com o Motor de Jogos Weaver. Contudo, ele também pode ser usado como uma API independente em outros projetos. Todos os diferentes algoritmos são encapsulados usando a mesma API de modo que é possível mudar o algoritmo usado apenas por meio de definição de macros. Também descrevemos aqui como os algoritmos se saem diante de diferentes testes estatísticos e em medidores de desempenho rodando em Linux, OpenBSD, Windows e Web Assembly.*

1. Introducão

1.1. Geradores de Números Aleatórios em Jogos Eletrônicos

Um gerador de números aleatórios é útil para os seguintes cenários em um jogo eletrônico:

a) Simulação: Isso se aplica tanto à simulação de fenômenos naturais que podem ser aleatórios ou mesmo fenômonos sociais. Isso é usado exaustivamente em jogos que são simuladores, como simuladores de cidades como SimCity. Mas pode ser usado para simular variações no clima, direção do vento ou no humor de personagens em quase qualquer tipo de jogo.

b) Amostragem: Podem existir muitos tipos diferentes de oponentes que podem ser gerados. Gerando as características variáveis aleatoriaente estamos de fato fazendo uma amostragem de um tipo de personagem específico dentre o conjunto de todos os possíveis. Por exemplo, em um jogo de Pokémon, determinada espécie de criatura pode possuir muitas variações diferentes de atributos possíveis a depender de sua genética e os tipos de ataques que ela tem também podem mudar. Gerando uma criatura aleatória, obtemos uma amostragem de um indivíduo dentre vários outros possíveis. Há o caso de simulações de mundos inteiros gerados de maneira procedural.

c) Programação: Durante o desenvolvimento do jogo, o programador poderia usar o gerador para simular escolhas aleatórias que um jogador poderia fazer para ver se alguma das combinações causa problemas ou falhas. Ou gerar fases aleatórias somente para checar se algum bug envolvendo o motor do jogo pode ser encontrado.

d) Estética: Adicionar aleatoriamente pequenas aleatoriedades em padrões de imagens que de outra forma seriam regulares pode aumentar o apelo estético de uma imagem. A aleatoriedade pode tornar também efeitos de transição de telas

mais interessantes.

e) Entretenimento: A diversão de um jogo pode estar inteiramente na aleatoriedade. Este é o caso de jogos envolvendo dados, roletas, embaralhamento de cartas.

Com tantos casos de uso diferentes para geradores de números aleatórios em jogos eletrônicos, o gerador ideal deve ser um projetado para propósito geral. Nunca um que apresenta um bom desempenho em só alguns dos casos de uso acima.

Geralmente há uma tolerância maior para quando um gerador de números aleatórios gera resultados previsíveis em um jogo. Muitos jogos antigos usavam geradores que consistiam apenas em uma sequência de números fixos, e mesmo assim o viés do gerador não era aparente. Mas existem casos em que as consequências de um gerador ruim são bastante severas. Por exemplo, um simulador que se propõe a ser realista pode falhar em treinar alguém para uma atividade real. Um gerador ruim pode ser explorado em jogos envolvendo apostas ou competições sérias.

Nosso objetivo aqui será definir uma API para um gerador de números aleatórios e programar diferentes opções de algoritmos para instanciá-la. Um usuário poderá escolher qual algoritmo será usado, mas se nenhuma escolha será feita, a biblioteca que será definida fará sua própria escolha sobre o algoritmo.

1.2. Programação Literária e Notação Usada no Artigo

Este artigo utiliza a técnica de "Programação Literária" para desenvolver a API de gerador de números aleatórios. Esta técnica foi apresentada em [Knuth, 1984] e tem por objetivo desenvolver *softwares* de tal forma que um programa de computador a ser compilado é exatamente igual a um documento escrito para pessoas detalhando e explicando o código. O presente documento não é algo independente do código, mas sim consiste no próprio código-fonte do projeto. Ferramentas automáticas são utilizadas para extrair o código deste documento, colocá-lo na ordem correta e produzir o código que é passado para o compilador.

Por exemplo, neste artigo serão definidos dois arquivos diferentes: `random.c` e `random.h`, os quais podem ser inseridos estaticamente em qualquer projeto, ou compilados como uma biblioteca compartilhada. O conteúdo de `random.h` é:

Arquivo: `src/random.h`**:**

```
#ifndef WEAVER_RANDOM
#define WEAVER_RANDOM
#ifdef __cplusplus
extern "C" {
#endif
#include <stdint.h>
#include <stdbool.h>
#if defined(__unix__) || defined(__APPLE__)
#include <pthread.h>
#endif
          <Seção a ser Inserida: Escolhe Algoritmo Padrão do RNG>
                    <Seção a ser Inserida: Estrutura RNG>
           <Seção a ser Inserida: Declarações de Gerador Aleatório>
#ifdef __cplusplus
}
#endif
#endif
```

As duas primeiras linhas assim como a última são macros que impedem que garantem que as funções e variáveis declaradas ali serão inseridas no máximo uma só vez em cada unidade de compilação. Também colocamos macros para checar se

estamos compilando o código como C ou C++. Se estivermos em C++, avisamos o compilador que estamos definindo tudo como código C e garantimos que não vamos modificar nada usando sobrecarga de operadores. O código poderá ser armazenado de maneira mais compacta.

A parte vermelha no código acima mostra que código será inserido ali no futuro. Em "Declarações de Gerador Aleatório", por exemplo, nós colocaremos declarações das funções.

Cada trecho de código tem um título, que no caso acima é `random.h`. O título indica onde o código será inserido. No caso acima, o código irá para um arquivo. Em trechos de código futuros, haverá um com o título "Declarações de Gerador Aleatório" com o código que irá nesta parte vermelha indicada.

1.3. Funções de API a serem Definidas

Nosso gerador de números aleatórios deverá fornecer um total de 3 novas funções.

A primeira função gera e inicializa um gerador retornando um ponteiro para ele. Para inicializá-lo, precisamos de uma função de alocação de memória passada como primiero argumento (pode ser o `malloc` da biblioteca padrão ou qualquer outro alocador personalizado). Em seguida nossa semente será um valor com um tamanho de um vetor e o .petor propriamente dito de números de 64 bits que assumimos terem sido obtidos de maneira aleatória e uniforme.

Dessa forma, nossa função de inicialização é suficientemente genérica para que o usuário possa fornecer um número personalizado de entropia ao gerador. Caso ele seja usado em um jogo que simula o embaralhamento de 54 cartas, por exemplo, o ideal é que ele passe um número de bits aleatório n tal que 2^n não seja muito menor que $54!$. Do contrário, não seremos capazes de simular a maioria dos embaralhamentos possíveis. Por outro lado, em algumas máquinas ou ambientes pode ser difícil conseguir uma quantidade muito grande de bits aleatórios para inicialização. Então, a API permite que um número menor seja passado.

Seção: Declarações de Gerador Aleatório:

```
struct _Wrng *_Wcreate_rng(void *(*alloc)(size_t), size_t size, uint64_t *seed);
```

Alguns algoritmos podem, contudo, ignorar qualquer valor maior que o recomendado ou podem não dar qualquer garantia de qualidade se receberem uma quantidade menor de bits aleatórios em sua inicialização. Justamente por isso, vamos fazer com que a nossa API avise a quantidade recomendada para o tamanho do vetor da semente, deixando a região abaixo a ser definida em breve:

Seção: Declarações de Gerador Aleatório:

<Seção a ser Inserida: Avisa Tamanho Ideal da Semente>

A segunda função é a que efetivamente é usada para nos dar o próximo número aleatório da sequência, que será um elemento de 64 bits:

Seção: Declarações de Gerador Aleatório (continuação):

```
uint64_t _Wrand(struct _Wrng *);
```

A última função será apenas para finalizar o uso de um gerador de números aleatórios. Ela recebe também um ponteiro para função que desalocará o gerador (ou NULL). Se receber NULL, o gerador não será desalocado (alguns alocadores podem ter seu próprio coletor de lixo sem fornecerem função de desalocação). De qualquer forma, ele não poderá mais ser usado depois de finalizado:

Seção: Declarações de Gerador Aleatório (continuação):

```
bool _Wdestroy_rng(void (*free)(void *), struct _Wrng *);
```

1.4. Suporte à Threads

O código adicionado aqui servirá para garantir que mais de uma thread possa usar o gerador de números randômicos sem problemas após a sua inicialização. Para

isso precisamos de um cabeçalho adequado para declarar os tipos que iremos usar nos ambientes em que for suportado:

Seção: Incluir Cabeçalhos Necessários (continuação):

```
#if defined(__unix__) || defined(__APPLE__)
#include <pthread.h>
#endif
#if defined(_WIN32)
#include <windows.h>
#endif
```

Tudo o que será preciso fazer é, para cada gerador, definir um mutex a ser colocado dentro do gerador (dentro do `struct _Wrng`):

Seção: Declaração de Mutex:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_t mutex;
#endif
#if defined(_WIN32)
CRITICAL_SECTION mutex;
#endif
```

Se temos um ponteiro para `struct _Wrng` chamado de `rng`, podemos inicializar seu mutex com:

Seção: Inicialização de Mutex:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_init(&(rng -> mutex), NULL);
#endif
#if defined(_WIN32)
InitializeCriticalSection(&(rng -> mutex));
#endif
```

Para pedirmos o uso de um Mutex, fazemos:

Seção: Mutex:WAIT:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_lock(&(rng -> mutex));
#endif
#if defined(_WIN32)
EnterCriticalSection(&(rng -> mutex));
#endif
```

E o código para liberarmos o uso do Mutex:

Seção: Mutex:SIGNAL:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_unlock(&(rng -> mutex));
#endif
#if defined(_WIN32)
LeaveCriticalSection(&(rng -> mutex));
#endif
```

O principal motivo de precisarmos de uma função para finalizar nosso gerador, a função `_Wdestroy_rng`, é que precisaremos finalizar o uso do mutex, além de desalocar o gerador se necessário. Como é só isso, podemos já declarar esta função nesta parte, pois já temos o necessário para defini-la:

Seção: Definição de _Wdestroy_rng:

```
bool _Wdestroy_rng(void (*free)(void *), struct _Wrng *rng){
  bool ret;
```

```
#if defined(__unix__) || defined(__APPLE__)
  ret = pthread_mutex_destroy(&(rng -> mutex));
#elif defined(_WIN32)
  DeleteCriticalSection(&(rng -> mutex));
  ret = true; // Sempre é bem-sucedida segundo a documentação
#endif
  if(free != NULL)
    free(rng);
  return ret;
}
```

2. Algoritmos Geradores de Números Pseudo-Randômicos

2.1. Gerador Linear Congruente (LGC)

Um Gerador Linear Congruente (também chamado pela sigla inglesa de *Linear Congruent Generator*) é simplesmente um gerador de sequências de valores aleatórias $(x_0, x_1, \ldots)$ definido com ajuda de constantes inteiras a, c e m tal que:

$$x_{i+1} = ax_i + c \ (mod\ m)$$

A qualidade dos geradores varia enormemente de acordo com os parâmetros escolhidos. Como queremos gerar sempre valores de 64 bits com nossa API, vamos escolher $m = 2^{64}$, o que irá facilitar bastante a operação. Não será necessário usar o operador de módulo explicitamente.

Com relação ao multiplicador a, nós usaremos o valor 0xfa346cbfd5890825 devido a este valor ser um dos listados entre multiplicadores com uma qualidade boa para ser usado com este módulo m no artigo [Steele, 2021].

Com relação ao valor de c, um requisito para sua qualidade é que ele seja ímpar (primo em relação à m), mas fora isso não há impacto na qualidade se c for algum valor ímpar. J'o valor inicial, também deve ser sempre um número ímpar, mas sem outros requisitos.

Sendo assim, podemos usar os primeiros 64 bits da semente para inicializar x_0, apenas forçando o bit menos significativo a ser 1. Se existir mais um valor de 64 bits na semente, usamos ele como nossa constante c, apenas ajustando o bit menos significativo para um. Do contrário apenas usamos $c = 1$.

Isso nos dá os seguintes tamanhso recomendados para nossa semente:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_LCG
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  2
#endif
```

A estrutura do nosso gerador é bastante simples, só precisa armazenar o último valor gerado e o valor da constante c:

Seção: Estrutura RNG:

```
#ifdef W_RNG_LCG
struct _Wrng{
  uint64_t last_value, c;
                <Seção a ser Inserida: Declaração de Mutex>
};
#endif
```

E ela é inicializada com o código:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_LCG
```

```
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    // Se não temos semente nenhuma, usamos uma constante aleatória como semente:
    rng -> last_value = ((size > 0)?(seed[0]):(0x1c3b9d10b1d41adc));
    rng -> c = ((size > 1)?(seed[1]):(1));
    rng -> last_value |= (1u);
    rng -> c |= (1u);
                <Seção a ser Inserida: Inicialização de Mutex>
  }
  return rng;
}
#endif
```

E finalmente, podemos agora definir a função que gera os próximos valores pseudo-randômicos:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_LCG
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
  rng -> last_value = 0xfa346cbfd5890825 * rng -> last_value + rng -> c;
  ret = rng -> last_value;
                    <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

O período esperado para esse gerador é 2^{64}, depois dessa quantidade os valores começam a se repetir.

2.2. SFMT

O SFMT é o "SIMD-Oriented Fast Mersenne Twister", proposto pela primeira vez em [Saito, 2006]. Ele tem esse nome porque o período deste gerador é projetado para ter valores bastante grandes, que são primos de Mersenne ($2^n - 1$ para algum n inteiro positivo). As implementações mais usadas deste algoritmo suportam períodos de $2^{19937} - 1$ sem que haja repetição. O que geralmente é mais que suficiente (ou mesmo um exagero) para qualquer caso de uso. Já o "SIMD" é um conjunto de instruções para valores vetorizados de 128 bits que CPUs mais recentes suportam. Tais instruções, se suportadas, são usadas para tornar o desempenho mais rápido quando o compilador é esperto o bastante para usá-las ou se a usamos explicitamente com assembly.

O SFMT funciona por meio da seguinte recursão onde cada valor X_i tem 128 bits:

$$X_{i+n} = g(X_i, \ldots, X_{i+n-1})$$

No caso, o gerador usa como n o número 156. O que significa que nós sempre precisamos memorizar os últimos 156 valores da sequência (cada um com 128 bits). Isso explica porque este gerador precisa de uma quantidade tão maior de memória quando comparado a outros.

A função g funciona concatenando toda a sua entrada na forma de um vetor binário com 156×128 elementos. Então, ela multiplica uma matriz binária por este vetor binário. A matriz binária tem 156×128 colunas e 128 linhas. Exceto que ao invés de trabalharmos usando o corpo convencional de números racionais

ou reais, nós trabalhamos no corpo finito GF(2) composto somente pelos números 0 e 1 (o vetor e a matriz são binários). Isso significa que o que entendemos por "multiplicação" aqui é um AND lógico e a adição é um XOR lógico. E também escolhemos uma matriz conveniente para que não seja necessário representar ela explicitamente. Poderemos representar a multiplicação da matriz por um vetor por meio de um número compacto de operações.

Na estrutura de nosso gerador, precisamos armazenar todos os valores anteriores necessários e também um *offset* que armazena a posição do próximo valor de 64 bits a ser retornado. Se este for um valor par, assumimos que temos que gerar um novo valor de 128 bits e retornar seus primeiros 64 bits. Se for um valor ímpar, não precisaremos gerar um novo valor, apenas retornaremos os 64 bits restantes do último valor.

Seção: Estrutura RNG:

```
#ifdef W_RNG_MERSENNE_TWISTER
struct _Wrng{
  char w[128 * 156 / 8]; // Todos N valores gerados, cada um com _W bits
  int offset;          // Índice para o próximo valor retornado
                <Seção a ser Inserida: Declaração de Mutex>
};
#endif
```

O padrão ISO da linguagem C não suporta variáveis que garantidamente tenham 128 bits. Contudo, versões recentes do compilador GCC e Clang as suportam. Usando elas, podemos computar a função g que computa o próximo elemento da sequência (ou seja, a multiplicação de nosso vetor por uma matriz em F_2) por meio da seguintes operações:

Seção: SFMT: Computa próximo elemento:

```
#ifdef __SIZEOF_INT128__
unsigned __int128 result, tmp;
uint32_t aux[4];
int i, index = rng -> offset / 2;
result = ((unsigned __int128 *) (rng -> w))[index];
result = result << 8;
result = result ^ ((unsigned __int128 *) (rng -> w))[index];
i = (index + 122) % 156;
aux[0] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 0);
aux[1] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 1);
aux[2] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 2);
aux[3] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 3);
aux[0] = (aux[0] >> 11) & 0xDFFFFFEF; // 0xBFFFFFF6
aux[1] = (aux[1] >> 11) & 0xDDFECB7F; // 0xBFFAFFFF
aux[2] = (aux[2] >> 11) & 0xBFFAFFFF; // 0xDDFECB7F
aux[3] = (aux[3] >> 11) & 0xBFFFFFF6; // 0xDFFFFFEF
memcpy(&tmp, aux, 16);
result = result ^ tmp;
i = (index + 156 - 2) % 156;
result = result ^ (((unsigned __int128 *) (rng -> w))[i] >> 8);
i = (index + 156 - 1) % 156;
aux[0] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 0);
aux[1] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 1);
aux[2] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 2);
aux[3] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 3);
aux[0] = (aux[0] << 18);
aux[1] = (aux[1] << 18);
```

```
aux[2] = (aux[2] << 18);
aux[3] = (aux[3] << 18);
memcpy(&tmp, aux, 16);
result = result ^ tmp;
((unsigned __int128 *) (rng -> w))[index] = result;
#else
#error "Mersenne Twister unsupported without 128 bit integer support."
#endif
```

Note que se estivermos em um compilador que não suporta variáveis de tamanho fixo com 128 bits, nós simplesmente retornamos um erro e não compilaremos.

O código acima representa a multiplicação de uma matriz por um vetor. Note que a matriz usada é bastante esparsa. Se nós queremos computar o valor x_i, nós o obtemos consultando apenas os valores anteriores x_{i-1}, x_{i-2}, x_{i-156} e x_{i-34}.

Se sabemos como computar o próxim elemento x_i, o código da função que nos dá o próximo valor de 64 bits será:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_MERSENNE_TWISTER
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
  if(rng -> offset % 2 == 0){
            <Seção a ser Inserida: SFMT: Computa próximo elemento>
  }
  ret = ((uint64_t *) (rng -> w))[rng -> offset];
  rng -> offset = (rng -> offset + 1) % (128 * 156 / 64);
                   <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

Para inicializar o nosso gerador, precisamos preencher nosso histórico de v156 valores anteriores com ajuda de nossa semente. Como a semente provavelmente será menor do que isso, aplicamos a ela um mini-gerador de números pseudo-randômicos só para esticar a aleatoriedade inicial e preencher os valores. Os primeiros dois componentes de nosso mini-gerador são:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_MERSENNE_TWISTER
static uint32_t f1(uint32_t x){
  return (x ^ (x >> 27)) * (uint32_t) 1664525UL;
}
static uint32_t f2(uint32_t x){
  return (x ^ (x >> 27)) * (uint32_t) 1566083941UL;
}
#endif
```

O modo de inicializar nossa estrutura é definido pela implementação de referência do algoritmo. Assim, tal como na referência, usamos um mini-gerador baseado em números de 32 bits para preencher o estado inicial:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_MERSENNE_TWISTER
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
```

```
    uint32_t *dst = (uint32_t *) (rng -> w), *origin = (uint32_t *) seed;
    size_t size_dst = 128 * 156 / 32, size_origin = size * 2;
    int count, r, i, j, mid = 306, lag = 11;
    // Preenchemos inicialmente tudo com 0x8b:
    memset(rng -> w, 0x8b, 128 * 156 / 8);
    count = ((size_origin + 1 >= size_dst)?(size_origin + 1):(size_dst));
    r = f1(dst[0] ^ dst[mid] ^ dst[size_dst - 1]);
    dst[mid] += r;
    r += size * 2;
    dst[mid + 11] += r;
    dst[0] = r;
    count --;
    for(i = 1, j = 0; j < count && j  < size_origin; j ++){
      r = f1(dst[i] ^  dst[(i + mid) % size_dst] ^
             dst[(size_dst - 1 + i) % size_dst]);
      dst[(i + mid) % size_dst] += r;
      r += origin[j] + i;
      dst[(i + mid + 11) % size_dst] += r;
      dst[i] = r;
      i = (i + 1) % size_dst;
    }
    for(; j < count; j++){
      r = f1(dst[i] ^ dst[(i + mid) % size_dst] ^
             dst[(i + size_dst - 1) % size_dst]);
      dst[(i + mid) % size_dst] += r;
      r += i;
      dst[(i + mid + 11) % size_dst] += r;
      dst[i] = r;
      i = (i + 1) % size_dst;
    }
    for (j = 0; j < size_dst; j++) {
      r = f2(dst[i] + dst[(i + mid) % size_dst] +
             dst[(i + size_dst - 1) % size_dst]);
      dst[(i + mid) % size_dst] ^= r;
      r -= i;
      dst[(i + mid + lag) % size_dst] ^= r;
      dst[i] = r;
      i = (i + 1) % size_dst;
    }
    rng -> offset = 0;
```

<Seção a ser Inserida: SFMT: Garante Período>

<Seção a ser Inserida: Inicialização de Mutex>

```
  }
  return rng;
}
#endif
```

Entretanto, antes de retornar o código acima é preciso checar se o estado que geramos realmente tem o período desejado de $2^{19937} - 1$. O modo de fazer isso é por meio da checagem de paridade para alguns bits específicos de nossa semente inicial. Se a paridade nao estiver correta, apenas ajustamos ela sem precisar mudar os demais valores gerados:

Seção: SFMT: Garante Período:

```
{
```

```
  // Máscara de bits a serem checados:
  uint32_t parity = (dst[0] & 0x00000001U) ^ (dst[3] & 0xc98e126aU);
  // Checagem de paridade:
  for (i = 16; i > 0; i >>= 1)
    parity ^= parity >> i;
  parity = parity & 1;
  if(parity != 1)
    dst[0] = dst[0] ^ 1;
}
```

Por fim, temos que documentar o tamanho recomendado para a semente do algoritmo. Podemos usar sem problemas uma semente com um único valor de 64 bits, já que a implementação de referência chega a suportar sementes com apenas 32 bits. Se passarmos sementes maiores, elas serão usadas para ajudar a preencher o estado inicial. Contudo, como o estado inicial é um vetor de 19968 bits, fornecer mais de 312 valores com 64 bits seria redundante:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_MERSENNE_TWISTER
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  312
#endif
```

2.3. SplitMix64

O SplitMix é um gerador de números aleatórios projetado para ser possível de ser dividido. Isso significa que existe um algoritmo tal que dada a estrutura do gerador, retorna uma nova estrutura tal que agora ambas podem gerar números pseudo-randômicos sem correlação entre si e sem a necessidade de fornecer novas sementes.

Mas o verdadeiro motivo de estarmos fornecendo o SplitMIx aqui não é esta propriedade. Não há planos de fornecer o suporte para geradores divisíveis aqui nesta API. Nosso verdadeiro interesse nete gerador é o fato de que este gerador tem uma natureza diferente dos demais definidos aqui e pelo fato dele funcionar precisando apenas de uma semente de 64 bits.

Lembre-se que no final da seção do Mersenne Twister foi necessário iicializar um estado de 312 bits mesmo que a nossa semente inicial tenha menos de 312 bits aleatórios. No caso do Mersenne Twister, usamos exatamente o método sugerido em sua implementação de referência, a qual consistia em usar outro RNG para preencher o estado inicial dada a semente que temos. Mas nes todos os geradores possuem um método padrão recomendado para preencher o estado inicial. Para os casos em que não há um método padrão, precisamos justamente de um gerador que use sementes pequenas, e publicações como [Matsumoto, 2007] mostram a importância de usarmos para este fim geradores radicalmente diferentes para evitar correlação entre sequências geradas com sementes semelhantes.

Ignorando os mecanismos de dividir a estrutura do gerador, o SplitMix é bastante simples. Ele foi criado à partir da ideia de ter um estado de 64 bits, e cada vez que um novo valor precisa ser gerado, o estado é atualizado para um novo valor e retornamos o resultado de uma função hash aplicada sobre o estado.

Exceto que por razões de performance, na prática ao invés de aplicar uma função hash inteira, usamos um misturador de bits: um componente que é parte de algumas funções hash e é responsável por evitar correlações entre valores de entrada semelhantes. O misturador de bits usado no SplaitMix64 é um qu foi definido originalmente para uma função hash chamada de MurmurHash3:

Seção: SplitMix: Misturador de Bits:

```
{
  uint64_t tmp = *state;
```

```
  tmp = (tmp ^ (tmp >> 33)) * 0xff51afd7ed558ccdl;
  tmp = (tmp ^ (tmp >> 33)) * 0xc4ceb9fe1a85ec53l;
  ret = tmp ^ (tmp >> 33);
}
```

E antes de computar o misturador de bits acima, nós também temos que atualizar o estado do RNG com a seguinte linha de código, onde `gamma` é uma constante ímpar:

Seção: SplitMix: Atualiza Estado:

```
{
  *state +=  gamma;
}
```

A construção final da função que retorna um novo número pseudo-randômico é:

Seção: Funções Auxiliares:

```
#if defined(W_RNG_SPLITMIX) || defined(W_RNG_XOSHIRO) || defined(W_RNG_PCG) || \
  defined(W_RNG_CHACHA20)
static inline uint64_t splitmix_next(uint64_t *state, uint64_t gamma){
  uint64_t ret;
            <Seção a ser Inserida: SplitMix: Misturador de Bits>
              <Seção a ser Inserida: SplitMix: Atualiza Estado>
  return ret;
}
#endif
```

Se estivermos usando o SplitMix não para inicializar outros geradores de números pseudo-randômicos, para para usar ele mesmo como nosso gerador escolhido, iremos definir a estrutura de nosso gerador com o estado e o valor gamma dele:

Seção: Estrutura RNG:

```
#ifdef W_RNG_SPLITMIX
struct _Wrng{
  uint64_t state, gamma;
                <Seção a ser Inserida: Declaração de Mutex>
};
#endif
```

E a função da API para gerar um novo número pseudo-randômico é:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_SPLITMIX
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
  ret = splitmix_next(&(rng -> state), rng -> gamma);
                  <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

Podemos inicializar a estrutura do gerador diretamente usando a semente para preencher o estado inicial. Se tiver mais bits aleatórios na semente, podemos também usá-los para escolher o valor de gamma, apenas cuidando para manter ele um valor ímpar e assim poder garantir um período de 2^{64}:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_SPLITMIX
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
```

```
                            uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size < 1)
      rng -> state = 0x32147198b5436569;
    else
      rng -> state = seed[0];
    if(size < 2)
      rng -> gamma = 0x9e3779b97f4a7c15;
    else
      rng -> gamma = (seed[1] | 1);
                <Seção a ser Inserida: Inicialização de Mutex>
  }
  return rng;
}
#endif
```

Isso significa que se usarmos o SplitMix como nosso gerador, precisamos de uma semente com 1 ou 2 números de 64 bits:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_SPLITMIX
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  2
#endif
```

2.4. Xoshiro256**

Xoshiro e Mersenne Twister são geradores com algumas similaridades. Ambos usam valores gerados do passado para produzir novos valores envolvendo uma multiplicação de matriz e vetor binários. Mas ao contrario do Mersenne Twister, cada novo valor gerado pelo Xoshiro tem 256 bits ao invés de 128. E ao invés de gerar o próximo valor levando em conta os 156 valores anteriores gerados, Xoshiro gera o próximo valor apenas à partir do valor imediatamente anterior gerado, precisando assim de menos memória.

Na estrutura do gerador nós armazenamos o número de 256 bits como uma sequência de quatro valores de 64 bits:

Seção: Estrutura RNG:

```
#ifdef W_RNG_XOSHIRO
struct _Wrng{
  uint64_t w[4];   // Valores de estado
                <Seção a ser Inserida: Declaração de Mutex>
};
#endif
```

Assim como o Mersenne Twister, para produzir o próximo valor, nós multiplicamos o valor anterior por uma matrix com operações no corpo GF(2): usamos o operador AND nos bits ao invés da multiplicação e usamos o operador XOR ao invés da soma. E também usamos uma matriz que ao mesmo tempo produz bons resultados e é conveniente o bastante para não precisar ser armazenada. Ao invés disso, a multiplicação por matriz pode ser escrita usando apenas poucas operações rápidas compostas por AND, XOR e SHIFT.

De fato, a multiplicação por matriz que atualiza nosso estado é representada apenas por esse código de 7 linhas:

Seção: Xoshiro: Multiplicação por Matriz:

```
{
```

```
  uint64_t t = rng -> w[1] << 17;
  rng -> w[2] ^= rng -> w[0];
  rng -> w[3] ^= rng -> w[1];
  rng -> w[1] ^= rng -> w[2];
  rng -> w[0] ^= rng -> w[3];
  rng -> w[2] ^= t;
  rng -> w[3] = ((rng -> w[3] << 45) | (rng -> w[3] >> 19));
}
```

Mas como estamos gerando números pseudo-randômicos usando uma simples transfrmação linear e o nosso estado é muito pequeno (ao contrário do estado do Mersenne Twister), precisamos usar um truque adicional para evitar falhar em testes estatísticos. Ao invés de retornar o número de 256 bits que geramos, nós iremos sempe produzir a saída à partir de um misturador que escolhe parte do valor total e aplica sobre ele operações que mascaram a relação entre o valor atual e os anteriores:

Seção: Xoshiro: Misturador:

```
{
  uint64_t tmp = rng -> w[1] * 5;
  ret = ((tmp << 7) | (tmp >> 57)) * 9;
}
```

E finalmente, definimos a função que retorna o próximo valor pseudo-randômico como abaixo, primeiro gerando o valor a ser retornado `ret` à partir do misturador, e então obtendo o próximo estado com a multiplicação por matriz:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_XOSHIRO
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
                <Seção a ser Inserida: Xoshiro: Misturador>
            <Seção a ser Inserida: Xoshiro: Multiplicação por Matriz>
                    <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

Por fim, a inicialização do gerador Xoshiro** consistirá em copiar para seu estado inicial a semente, caso esta tenha 256 bits ou mais. Caso contrário, copiaremos o que temos e usaremos os últimos 64 bits da semente para inicializar o estado do algoritmo SplitMix64 usando um valor gama padrão e usamos os gerador do SplitMix64 para produzir os bits restantes:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_XOSHIRO
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  int i;
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size >= 4){
      for(i = 0; i < 4; i ++)
        rng -> w[i] = seed[i];
    }
    else{
      uint64_t state = 0x32147198b5436569, gamma = 0x9e3779b97f4a7c15;
      for(i = 0; i < size - 1; i ++)
```

```
        rng -> w[i] = seed[i];
      if(size > 1)
        state = seed[i];
      for(; i < 4; i ++)
        rng -> w[i] = splitmix_next(&state, gamma);
    }
                <Seção a ser Inserida: Inicialização de Mutex>
  }
  return rng;
}
#endif
```

Devido a isso, vamos indicar como um valor entre 1 e 4 o tamanho recomendado para a semente deste gerador:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_XOSHIRO
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  4
#endif
```

2.5. PCG (*Permuted Congruential Generator*)

O gerador de números aleatórios PCG é similar ao SplitMix em sua filosofia de funcionamento. El usa uma operação simples para atualizar o estado interno. Mas ao invés de tratar cada novo estado interno como o novo número a ser retornado, ele o passa para uma função mais complicada para assim produzir cada número aleatório de saída.

A atualização simples de estado, que no SplitMix consistia em somar uma constante ao estado interno, aqui significa atualizar o estado por meio de um gerador linear congruente (LCG) como o apresentado na seção 2.1.

Contudo, o gerador linear congruente que definimos funcionava usando 64 bits. Mas o estado interno do PCG tem 128 bits e por isso precisamos de um gerador LCG com 128 bits para atualizar o estado.

A estrutura de nosso gerador é definida como:

Seção: Estrutura RNG:

```
#ifdef W_RNG_PCG
#ifdef __SIZEOF_INT128__
struct _Wrng{
  unsigned __int128 state;
  unsigned __int128 increment; // Sempre deve ser ímpar
                <Seção a ser Inserida: Declaração de Mutex>
};
#else
#error "PCG unsupported without 128 bit integer support."
#endif
#endif
```

Nós atualizamos o estado interno realizando uma operação LCG de multiplicar por um multipli9cador constante escolhido cuidadosamente a somar um incremente ímpar ao resultado:

Seção: PCG: Atualizar Estado:

```
{
  unsigned __int128 multiplier;
  multiplier = 2549297995355413924ULL;
  multiplier = multiplier << 64;
```

```
  multiplier += 4865540595714422341ULL;
  rng -> state = rng -> state * multiplier + rng -> increment;
}
```

Mas tal como no SplitMix, ao invés de retornar este estado como o próximo número da sequência, nós o passamos para algum tipo de função hash, embaralhador ou misturador. No nosso caso, nós combinamos os 128 bits em 64 por meio de uma operação de XOR e em seguida aplicamos uma função de permutação ao resultado:

Seção: PCG: Permutação:

```
{
  uint64_t xorshifted, rot;
  xorshifted = (((uint64_t)(rng -> state >> 64u)) ^ ((uint64_t) rng -> state));
  rot = rng -> state >> 122u;
  ret = (xorshifted >> rot) | (xorshifted << ((-rot) & 63));
}
```

E finalmente, a função completa que gera o próximo número aleatório combina as operações acima da seguinte forma:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_PCG
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
                <Seção a ser Inserida: PCG: Atualizar Estado>
                  <Seção a ser Inserida: PCG: Permutação>
                   <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

O que resta ser feito é inicializar o estado inicial. Se inicializarmos com 256 bits ou mais, nós poderíamos usar a própria semente para inicializar o estado inicial. Ao invés disso, para agirmos assim como a implementação de referência, nós fazemos uma mistura mais complicada com os bits de entrada tentando lidar com casos em que a inicialização não é muito aleatória.

Se passarmos uma semente com somente 128 bits, podemos usá-la diretamente para inicializar o estado inicial e podemos ajustar o incremento do LCG como sendo 1. Se tivermos 192 bits aleatórios, podemos usar os bits restantes para escolher um valor diferente para o incremento. Já se tivermos menos de 128 bits, usaremos então o SplitMix para esticar a nossa aleatoriedade inicial até 128. A função de inicialização é definida conforme se vê abaixo:

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_PCG
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size >= 4){
      unsigned __int128 multiplier;
      multiplier = 2549297995355413924ULL;
      multiplier = multiplier << 64;
      multiplier += 4865540595714422341ULL;
      unsigned __int128 initstate = seed[0], initseq;
      initstate = initstate << 64;
      initstate += seed[1];
```

```
      initseq = seed[2];
      initseq = initseq << 64;
      initseq += seed[3];
      rng->state = 0U;
      rng -> increment = (initseq << 1) | 1;
      rng->state = rng->state * multiplier + rng -> increment;
      rng -> state += initstate;
      rng->state = rng->state * multiplier + rng -> increment;
    }
    else if(size >= 2){
      rng -> state = seed[0];
      rng -> state = (rng -> state << 64);
      rng -> state += seed[1];
      if(size == 3){
        rng -> increment = seed[2];
        rng -> increment = (rng -> increment) | 1;
      }
      else
        rng -> increment = 1;
    }
    else{
      uint64_t state, gamma = 0x9e3779b97f4a7c15;
      if(size > 0)
        state = seed[0];
      else
        state = 0x32147198b5436569;
      rng -> state = splitmix_next(&state, gamma);
      rng -> state = (rng -> state << 64);
      rng -> state += splitmix_next(&state, gamma);
      rng -> increment = 1;
    }
```

<**Seção a ser Inserida:** Inicialização de Mutex>

```
  }
  return rng;
}
#endif
```

Isso significa que o tamanho recomendado para a semente é entre dois e quatro números de 64 bits aleatórios para assim não precisarmos esticar a aleatoriedae com o SplitMix:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_PCG
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  2
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  4
#endif
```

2.6. ChaCha20

O ChaCha20 é um gerador projetado para ser criptograficamente seguro, mas essa afirmação só é verdadeira se ele for alimentado com ua semente com um tamanho adequado, e se a semente realmente for obtida de maneira aleatória e uniforme e sem que ela seja reaproveitada novamente. Nossa API não garante tais coisas, aqui não estamos interessados em geradores criptograficamente seguros. Para nós o CHaCha20 será apenas um gerador como todos os outros anteriores, com o diferencial de que espera-se que ele tenha uma qualidade maior com um maior custo em

desempenho para compensar.

O algoritmo tem como primeiro componente de seu funcionamento uma função de preenchimento, que receberá como entrada 384 bits (a serem interpretados como separados em componentes de 64 bits) e gera como saída 512 bits. A ideia é que a saída seja interpretada como uma matriz 4×4 de elementos de 32 bits, mesmo que não a armazenemos neste formato.

A função de preenchimento é:

Seção: ChaCha20: Preenchimento:

```
#ifdef W_RNG_CHACHA20
static void chacha_padding(uint64_t input[6], uint32_t output[16]){
  int i, j;
  output[0] = ('e' << 24) + ('x' << 16) + ('p' << 8) + 'a';
  output[1] = ('n' << 24) + ('d' << 16) + (' ' << 8) + '3';
  output[2] = ('2' << 24) + ('-' << 16) + ('b' << 8) + 'y';
  output[3] = ('t' << 24) + ('e' << 16) + (' ' << 8) + 'k';
  for(j=4, i = 0; i < 6; i ++, j += 2){
    output[j] = (input[i] / 4294967296llu);
    output[j+1] = input[i] % 4294967296llu;
  }
}
#endif
```

Na entrada dessa função de preenchimento, os primeiros 4 valores sempre são retirados da semente inicial. O próximo é sempre um contador que no primeiro bloco gerado é 0, e a cada próximo incrementa em 1. E o último é um número que deve ser sempre usado uma só vez.

O próximo componente é a função abaixo que recebe 4 valores de 32bits e os modifica realizando uma série de operações:

Seção: ChaCha20: QuarterRound:

```
#ifdef W_RNG_CHACHA20
void quarter_round(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d){
  *a = *a + *b;
  *d = *d ^ *a;
  *d = (*d << 16) | (*d >> 16);
  *c = *c + *d;
  *b = *b ^ *c;
  *b = (*b << 12) | (*b >> 20);
  *a = *a + *b;
  *d = *d ^ *a;
  *d = (*d << 8) | (*d >> 24);
  *c = *c + *d;
  *b = *b ^ *c;
  *b = (*b << 7) | (*b >> 25);
}
#endif
```

A operação anterior é utilizada na seguinte função de permutação que recebe como entrada 16 elementos de 32 bits e os transforma:

Seção: ChaCha20: Permutação:

```
#ifdef W_RNG_CHACHA20
void chacha_permutation(uint32_t elements[16]){
  int i;
  for(i = 0; i < 10; i ++){
    quarter_round(&elements[0], &elements[4], &elements[8], &elements[12]);
```

```
    quarter_round(&elements[1], &elements[5], &elements[9], &elements[13]);
    quarter_round(&elements[2], &elements[6], &elements[10], &elements[14]);
    quarter_round(&elements[3], &elements[7], &elements[11], &elements[15]);
    quarter_round(&elements[0], &elements[5], &elements[10], &elements[15]);
    quarter_round(&elements[1], &elements[6], &elements[11], &elements[12]);
    quarter_round(&elements[2], &elements[7], &elements[8], &elements[13]);
    quarter_round(&elements[3], &elements[4], &elements[9], &elements[14]);
  }
}
#endif
```

Com esses pré-requisitos podemos começar a definir a estrutura e funções de nossa API. A estrutura do gerador deverá armazenar um vetor com 6 elementos de 64 bits (que depois serão passados para a função que gera novos números após antes passar pelo preenchimento), um espaço de 512 bits com os últimos valores gerados (se existirem) e um índice para saber qual o próximo valor que deve ser retornado. A estrutura será então:

Seção: Estrutura RNG:

```
#ifdef W_RNG_CHACHA20
struct _Wrng{
  uint64_t array[6];
  uint32_t generated_values[16];
  int index;
                <Seção a ser Inserida: Declaração de Mutex>
};
#endif
```

Quando a estrutura é alocada, os primeiros 4 valores do nosso vetor de 6 elementos são copiados da semente, os próximo valor é preenchido como 0 e o último também é removido da semente, mas se não existir pode ser tratado como 0 (ele seria um nounce). Isso significa que o ideal é que nossa semente tenha entre 4 e 5 valores:

Seção: Avisa Tamanho Ideal da Semente:

```
#ifdef W_RNG_CHACHA20
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  4
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  5
#endif
```

A questão é o que fazr se nossa semente for menor que isso. Faremos a mesma coisa que fizemos com o Xoshiro: usamos o algoritmo splitmix64 para preencher os próximos à partir do valor que temos (e usamos uma constante se não existir nenhum). Mas não se deve ter nenhuma expenctativa de que a qualidade é mantida com uma semente reduzida (não é, e é catastrófico se for usado para fins criptográficos):

Seção: Definição de _Wcreate_rng:

```
#ifdef W_RNG_CHACHA20
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  int i, j;
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    for(i = 0; i < 4; i ++){
      if(i < size - 1 || size >= 4)
        rng -> array[i] = seed[i];
      else{
```

```
        // Using SplitMix to fill the remaining values
        uint64_t state, gamma = 0x9e3779b97f4a7c15;
        if(size > 0)
          state = seed[i];
        else
          state = 0x32147198b5436569;
        for(j = i; j < 4; j ++)
          rng -> array[j] = splitmix_next(&state, gamma);
        break;
      }
    }
    rng -> array[4] = 0;
    if(size > 4)
      rng -> array[5] = seed[4];
    else
      rng -> array[5] = 0;
    rng -> index = 0;
                <Seção a ser Inserida: Inicialização de Mutex>
  }
  return rng;
}
#endif
```

Quando tivermos que gerar um novo valor, geralmente temos valores já gerados no vetor de 16 números de 32 bits. Com eles dá para gerarmos 8 saídas de 64 bits. Procedemos da seguinte forma quando não precisamos gerar novos valores:

Seção: Definição de _Wrand:

```
#ifdef W_RNG_CHACHA20
              <Seção a ser Inserida: ChaCha20: Preenchimento>
              <Seção a ser Inserida: ChaCha20: QuarterRound>
               <Seção a ser Inserida: ChaCha20: Permutação>
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Seção a ser Inserida: Mutex:WAIT>
  if(rng -> index % 16 == 0){
    rng -> index = 0;
             <Seção a ser Inserida: ChaCha20: Gera Novos Valores>
    rng -> array[4] ++;
  }
  ret = rng -> generated_values[rng -> index];
  ret = ret << 32;
  ret += rng -> generated_values[rng -> index + 1];
  rng -> index += 2;
                   <Seção a ser Inserida: Mutex:SIGNAL>
  return ret;
}
#endif
```

E finalmente a especificação fica completa mostrando como usando o seu estado interno o gerador obtém novos valores pseudo-aleatórios:

Seção: ChaCha20: Gera Novos Valores:

```
{
  int i;
  uint32_t padded_array[16], copied_array[16];
```

```
  chacha_padding(rng -> array, padded_array);
  for(i = 0; i < 16; i ++)
    copied_array[i] = padded_array[i];
  chacha_permutation(padded_array);
  for(i = 0; i < 16; i ++)
    rng -> generated_values[i] = padded_array[i] + copied_array[i];
}
```

2.6. Estrutura Final do Arquivo

O arquivo com o código-fonte de nossas funções terá a seguinte forma:

Arquivo: `src/random.c`:

```
<Seção a ser Inserida: Incluir Cabeçalhos Necessários>
#include "random.h"
#include <string.h> // memcpy
<Seção a ser Inserida: Funções Auxiliares>
<Seção a ser Inserida: Definição de "Wrand>
<Seção a ser Inserida: Definição de "Wcreate"rng>
<Seção a ser Inserida: Definição de "Wdestroy"rng>
```

3. Testes de Qualidade

Como forma de testar a qualidade de todos os algoritmos fornecidos e fazer comparações, fazemos alguns testes estatísticos empíricos sugeridos em [Knuth, 1998]. Eles são um conjunto de testes iniciais, mas mais testes podem ser adicionados à esta seção futuramente.

Uma ressalva é que muitos dos testes sugeridos lá presume o desejo de gerar números aleatórios entre 0 e um valor n, ou então valores reais entre 0 e 1. Contudo, se assumirmos isso acabaremos dando um peso maior para os bits mais significativos gerados. Mas queremos garantir a qualidade mínima dele para todos os bits. Desta forma, buscamos em diferentes testes tratar os valores gerados como uma sequência de bits (não de números de 64 bits), e escolhemos de maneira arbitrária, mas diferente em cada um dos testes, escolher um tamanho em bits diferente para cada valor gerado.

Em todos os testes, aplicamos um teste como o Chi-Quadrado como sugerido na referência [Knuth, 1998]. Repetimos as medidas 3 vezes, se uma delas um valor aleatório tiver uma chance inferior a 1% de ser tão distante ou de ser tão próximo do esperado, interrompemos os testes e consideramos uma falha. Se por duas das três vezes testadas o valor obtido for algo com apenas 5% de chance de ser tão distante ou tão próximo do esperado, também consideramos uma falha. Repetimos os três testes mil vezes e geramos assim uma porcentagem de sucesso. Como referência, a menos que seja dito o contrário, o valor esperado de sucesso é de 92,34%.

Todos os testes são feitos também para a mesma semente, escolhida de maneira elatória e uniforme. No caso, ela é a sequência de 5 valores hexadecimais: (`32147198b5436569`, `260287febfeb34e9`, `0b6cc94a91a265e4`, `c6a109c50dd52f1b`, `8298497f3992d73a`).

3.1. Teste de Equidistribuição

Basicamente neste teste geramos uma sequência de dez mil bits aleatórios e avaliamos o quão bem distribuído o valor está entre zeros e uns. A taxa de sucesso para as implementações apresentadas foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	91%
PCG	93%	`LCG`	93%
ChaCha20	91%	SplitMix64	92%

3.2. Teste Serial

O teste serial consiste em gerar sequências de d bits e verificar se cada combinação possível ocorre com uma proporção semelhante a $1/2^d$. No nosso caso escolhemos $d = 15$. O resultado obtido foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	93%	`LCG`	91%
ChaCha20	93%	SplitMix64	93%

3.3. Teste de lacuna

Este teste também foi sugerido em [Knuth, 1998]. Mas ao invés de fazer a medida de lacuna interpretando os números obtidos como entre 0 e 1 como sugerido (o que talvez desse um peso alto para os bits mais significativos), nós medimos a lacuna de bits: interpretamos os valores retornados pelo gerados como uma sequência contínua de bits. E contamos o tamanho das lacunas encontradas (sequências de bits 0 terminadas em bits 1).

Por exemplo, na sequência "100110010100001" obtemos as lacunas "1" (tamanho 0), "001" (tamanho 2), "1" (tamanho 0), "001" (tamanho 2), "01" (tamanho 1) e "00001" (tamanho 4). Em cada gerador obtemos uma amostragem de $5 \cdot 2^{20}$ lacunas e avaliamos a quantidade de cada tamanho de lacunas com um teste de chi-quadrado. Aplicamos então a mesma avaliação dos outros testes e medimos a taxa de sucesso. O resultado obtido foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	91%	`LCG`	86%
ChaCha20	91%	SplitMix64	90%

3.4. Teste de Pôquer

Neste teste interpretamos o retorno de nossos geradores como sequências de números de 0 a 15. Geramos então sucessivas tuplas com 5 elementos. E fazemos a contagem de fenômenos como tuplas em que todos os valores são diferentes, todos são iguais, há quatro valores iguais, há uma tripla e um par, há uma tripla, há dois pares ou há só um par. Medimos o quanto os valores obtidos divergem do que seria esperado com valores completamente aleatórios.

O resultado obtido foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	93%
PCG	91%	`LCG`	92%
ChaCha20	92%	SplitMix64	91%

3.5. Teste do Colecionador

Este teste é um complemento do anterior. Enquanto o anterior mostrava a equidistributividade de uma mão de pôquer, este tenta medir o quão aleatória é a ordem na qual obtemos cada valor de 4 bits. Geramos valores novos até que tenhamos gerado os 16 valores diferentes, quando encerramos. Fazemos isso um

total de 4408394 vezes. E notamos em quantas dessas tentativas paramos de gerar novos valores por já termos obtido todos após 16, 17, ..., 115 ou 116+ tentativas. O teste compara então a quantidade que cada caso ocorreu comparando com o esperado.

O resultado do teste foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	91%	Xoshiro**	92%
PCG	92%	`LCG`	00%
ChaCha20	92%	SplitMix64	93%

Este é o primeiro teste no qual um de nossos algoritmos falha. O LCG produz resultdos que são idealizados demais para serem considerados aleatórios.

3.6. Teste da Permutação

Este teste consiste em gerar números entre 0 e 7, ignorando valores repetidos, até que os oito valores diferentes apareçam. Em seguida, repetimos o teste e vamos contando todas as diferentes permutações destes oito valores assim obtida, verificando se a distribuição das $8!$ diferentes permutações está de acordo com o esperado.

O nosso resultado foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	93%	Xoshiro**	92%
PCG	93%	`LCG`	02%
ChaCha20	92%	SplitMix64	93%

3.7. Teste de Tamanho de Sequências Ascendentes

Este teste, como o anterior, consiste em gerar permutações de elementos. Mas ao invés de fazer uma contagem de todas as permutações, ele avalia elas por meio da quantidade de elementos ascendentes, onde o elemento gerado é maior que o anterior.

Para este teste escolhemos gerar permutações de números entre 0 e 8192. E fizemos uma análise estatística da quantidade de sequências ascendentes de 1, 2, 3, 4, 5 ou maiores ou iguais a 6 elementos.

O resultado obtido foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	91%
PCG	90%	`LCG`	90%
ChaCha20	90%	SplitMix64	90%

3.8. Teste do Maior de t

Este teste consiste em gerar t sequências de números entre 0 e algum valor (no nosso caso foi $2^6 - 1$) e armazenar o maior dos valores. No nosso caso foi escolhido $t = 3$. Em seguida, compara-se a distribuição dos maiores valores com o que é estatisticamente esperado.

O resultado foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	91%	`LCG`	86%
ChaCha20	92%	SplitMix64	92%

3.9. Teste de Colisões

Este teste, ao contrário dos demais, não usa o teste do Chi-Quadrado. Isso

porque ele tenta medir de maneira aproximada a ocorrência de eventos que tem uma probabilidade muito baixa para o Chi-Quadrado. Se a saída do nosso gerador de números aleatórios fôsse a saída de uma função hash, com que probabilidade ocorreria uma colisão?

Este teste assume uma saída de 20 bits para o nosso gerador e gera um total de 2^{14} saídas diferentes. Em seguida, o número de colisões que ocorreram é testada para ver se ficou muito acima ou muito abaixo do esperado. Tal teste é repetido um total de cerca de 3000 vezes, onde o número de colisões esperada é ajustada para que a saída deste teste tenha uma porcentagem de sucesso esperada semelhante aos testes anteriores com o Chi-Quadrado (que é próximo de 92%).

O resultado do teste foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	94%	Xoshiro**	93%
PCG	94%	`LCG`	92%
ChaCha20	93%	SplitMix64	92%

3.10. Teste de Espaçamento de Aniversário

Neste teste nós geramos um número $n = 512$ de valores de 25 bits. Em seguida, os ordenamos e medimos a diferença entre valores sucessivos. E contamos o número de distâncias iguais que encontramos. O resultado é comparado com o esperado em sequências aleatórias.

Este teste foi sugerido por George Marsaglia e quando proposto encontrou falhas em geradores que passavam nos testes estatísticos existentes na época.

Nosso resultado foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	92%	`LCG`	92%
ChaCha20	91%	SplitMix64	91%

3.11. Teste da Correlação Serial

Este teste gera uma sequência de n elementos (onde $n = 1000$) de 64 bits, que aqui vamos interpretar como elementos entre 0 e 1. Ou seja, dividimos o resultado por $2^{61} - 1$ para obter um número em ponto flutuante. Calculamos então o coeficiente de correlação serial entre cada um dos números gerados quando comparado com cada número anterior e medimos se ele está dentro de um intervalo considerado aceitável.

Em seguida, repetimos esta mesma comparação não apenas entre cada elemento e seu sucessor, mas também entre cada elemento e o elemento duas posições à frente. Depois três posições. Até chegar nas 500 posições de deslocamento, tentando encontrar qual nos dá o pior índice de correlação serial. A medida do pior índice é usada e comparada com valores esperados de forma que o esperado seja passar neste teste derca de 93% das vezes. E repetimos isso mil vezes para ver se a porcentagem de testes qu passou é próxima desta.

Assim como o anterior, este teste não é uma medida de Chi-Quadrado.

O resultado obtido foi:

Gerador	Sucesso	Gerador	Sucesso
SFMT	93%	Xoshiro**	93%
PCG	93%	`LCG`	93%
ChaCha20	93%	SplitMix64	93%

3.12. Generating all 32-bit values

Este teste difere dos anteriores porque ao contrário de obter uma porcentagem de sucesso, nós estamos interessados se é possível gerar todos os números de 32

bits diferentes com nossos geradores se eles executarem por tempo suficiente. Se for possível, estamos interessados na quantidade de valores que precisaram ser gerados até termos todos os possíveis. Consideramos que um gerador falha se ele não for capaz de gerar algum valor.

Nosso resultado é:

Gerador	Sucesso	Gerador	Sucesso
SFMT	98852849277	Xoshiro**	92842427748
PCG	100979563727	`LCG`	8589934581
ChaCha20	99686167785		

Todos os algoritmos conseguiram gerar todos os valores de 32 bits diferentes e o número de tentativas foi similar, exceto pelo LCG. O número de valores gerados pelo LCG foi baixo demais, praticamente metade dos valores retornados eram valores novos, não repetidos. Assim como o teste do colecionador, isso mostra um viés conhecido deste gerador de evitar produzir valores repetidos.

3.13. Conclusão dos Testes

Mós produzimos testes que conseguiram encontrar falhas no algorithmo LCG. Os resultados indicam um viés neste algoritmo contra a geração de valores repetidos.

4. Medida de Desempenho

Para medir o desempenho de cada gerador, pedimos para cada um deles gerar sequencialmente um total de 100 milhões de números aleatórios de 64 bits. Também somamos os valores gerados apenas para evitar otimizações do compilador que poderiam ignorar a computação. Testamos o tempo que levou em 4 ambientes diferentes divididos em dois computadores.

O teste abaixo foi feito em um OpenBSD 6.7 usando o Computador A (Intel Pentium B980 dual core, 2,40 GHz, 4 GB RAM) e o compilador Clang 8.0.1.

O resultado em segundos foi:

Gerador	Tempo (s)	Gerador	Tempo (s)
SFMT	9.571183	Xoshiro**	7.811493
PCG	8.133946	`LCG`	7.411282
ChaCha20	10.498551	SplitMix64	7.529885

Fazendo os testes no mesmo computador, mas usando o Windows 10 e compilando com o compilador padrão do Visual Studio (que não suporta variáveis de 128 bits e por isso não pode executar nosso código do PCG e Mersenne Twister), obtivemos como resultado:

Gerador	Tempo (s)	Gerador	Tempo (s)
SFMT	-	Xoshiro**	4.570623
PCG	-	`LCG`	3.615481
ChaCha20	8.019865	SplitMix64	4.340963

Repetindo os testes em um outro computador mais novo, um Intel i5-3210M quad core com 2,50GHz e também 4 GB de memória RAM, e desta vez utilizando um Ubuntu 20.04.2 e um compilador GCC 9.3.0, o resultado foi:

Gerador	Tempo (s)	Gerador	Tempo (s)
SFMT	2.848333	Xoshiro**	2.565788
PCG	2.292101	`LCG`	2.378609
ChaCha20	7.085438	SplitMix64	2.437079

O mais surpreendente é o PCG estar rodando mais rápido que o LCG. É inesperado tal resultado pelo fato do PCG usar internamente uma versão de gerador linear congruente para functionar. Este também foi o único ambiente no qual o PCG foi

mais rápido que outros algoritmos como o Xoshiro** e SplitMix64. O resultado não é muito diferente se executarmos no mesmo sistema, mas compilarmos com o Clang 10.0:

Gerador	Tempo (s)	Gerador	Tempo (s)
SFMT	2.601277	Xoshiro**	2.449881
PCG	2.222556	`LCG`	2.348797
ChaCha20	4.620730	SplitMix64	2.346077

Usando o mesmo computador, mas compilando usando o Emscripten 2.0.14 para produzir Web Assembly e executar em uma página de Internet no navegador Firefox 86.0, o resultado foi:

Gerador	Tempo (s)	Gerador	Tempo (s)
SFMT	1.318	Xoshiro**	0.460
PCG	1.324	`LCG`	0.349
ChaCha20	3.687	SplitMix64	0.393

O resultado surpreendente é o quão mais rápido este teste rodou quando comparado com código compilado executando nativamente. Além disso, o resultado sugere que implementações que são baseadas em variáveis de 128 bits tem um desempenho pior neste ambiente.

5. Conclusão: Escolhendo o Algoritmo Padrão

Quando o usuário não escolher qual algoritmo nossa API deve usar, devemos fazer para ele uma escolha bem-informada.

Nosso primeiro critério, é claro, será se o algoritmo como implementado é suportado no ambiente e arquitetura atual. Os algoritmos PCG e Mersenne Twister não compilarão se o compilador não suportar números de 128 bits.

O segundo critério é se o algoritmo passou em todos os nossos testes de qualidade. Aqui só o LCG é excluído e não será considerado para o terceiro critério.

O terceiro critério é a velocidade. Aqui em todos os testes o SplitMix64 foi o mais rápido, exceto quando testado em ambiente Linux. É possível que isso ocorreu devido aos compiladores padrão serem mais recentes e conseguirem otimizar melhor implementações com 128 bits.

Combinando esses critérios, terminaremos usando o SplitMix64 em todos os cenários, exceto se estivermos no Linux e com um compilador que suporta variáveis de 128 bits:

Seção: Escolhe Algoritmo Padrão do RNG:

```
#if !defined(W_RNG_MERSENNE_TWISTER) && !defined(W_RNG_XOSHIRO) && \
!defined(W_RNG_PCG) && !defined(W_RNG_LCG) && !defined(W_RNG_CHACHA20)
#if defined(__SIZEOF_INT128__) && defined(__linux__)
#define W_RNG_PCG
#else
#define W_RNG_SPLITMIX
#endif
#endif
```

Referências

Knuth, D. E. (1984) “Literate Programming”, The Computer Journal, volume 27, edição 2, p. 97–111

Knuth, D. (1998) “The Art of Computer Programming, v. 2: Seminumerical Algorithms”, Addison-Wesley Professional, terceira edição.

Saito, M.; Matsumoto M. (2006) “SIMD-oriented fast Mersenne Twister: a 128-

bit pseudorandom number generator", Monte Carlo and Quasi-Monte Carlo Methods, Springer, p. 607–622.

Steele, G.; Vigna S. (2021) "Computationally Easy, Spectrally Good Multipliers for Congruential Pseudorandom Number Generators", arXiv preprint arXiv:2001.-05304.

Matsumoto, M; Wada I.; Kuramoto A.; Ashihara, H. (2007) "Common Defects in the Initialization of Pseudrandom Number Generators", ACM Trans. Model. Comput. Simul, 17, 4.