# Elixir 2023.1



# Concurrent Linked list

Abaixo, temos um esboço de implementação de lista encadeada. Esta implementação tem problemas de concorrência. Detecte e corrija os problemas detectados usando semáforos. Não proteja regiões maiores que o necessário.

```
// tipo node
typedef struct __node_t {
	int key;
	struct __node_t	*next;
} node_t;

// basic list structure
```

```
typedef struct __list_t {
	node_t *head;
} list_t;

void List_Init(list_t	*L) {
	L->head = NULL
}

int List_Insert(list_t *L, int key) {
	node_t* new = malloc(sizeof(node_t));
	if (new == NULL) {
		perror("malloc");
		return -1; // fail
	}
	new->key = key;
	new->next = L->head;
	L->head = new;
	return 0; // success
}

int List_Lookup(list_t*L, int key) {
	node_t*curr = L->head;
	while (curr) {
		if (curr->key == key){
			return 0; // success
		}
		curr =curr->next;
	}
	return -1; // failure
}
```

# Fork-sleep-join

Crie um programa que recebe um número inteiro **n** como argumento e cria **n** threads. Cada uma dessas threads deve dormir por um tempo aleatório de no máximo 5 segundos. A main-thread deve esperar todas as threads filhas terminarem de executar para em seguida escrever na saída padrão o valor de **n**. Faça a thread-mãe esperar as filhas de duas maneiras: 1) usando o equivalente à função **join** em C ou Java; 2) usando semáforos.

# Two-phase sleep

Crie um programa que recebe um número inteiro **n** como argumento e cria **n** threads. Cada uma dessas threads deve dormir por um tempo aleatório de no máximo 5 segundos. Depois que acordar, cada thread deve sortear um outro número aleatório **s** (entre 0 e 10). **Somente depois de todas** as **n** threads terminarem suas escolhas (ou seja, ao fim da primeira fase), começamos a segunda fase. Nesta segunda fase, a n-ésima

thread criada deve dormir pelo tempo **s** escolhido pela thread n - 1 (faça a contagem de maneira modular, ou seja, a primeira thread dorme conforme o número sorteado pela última). Use semáforos.

# Blocking rate-limiting

Em certos sistemas evitamos que requisições de tipos diferentes sejam executadas concorrentemente.

Considere o código abaixo:

```
func handle(Request req) {
      exec(req)
}
```

Considere que a função ***handle*** é executada por uma thread toda vez que uma requisição chega no sistema. Você não precisa se preocupar com a criação dessa thread. Uma requisição tem um tipo inteiro, ou seja, Request.type. Em nosso caso, há dois tipos (0 e 1). Altere o código da função ***handle*** para controlar a execução do sistema de maneira que não seja possível a execução concorrente pela função **exec** de requisições de tipos diferentes. Além disso, controle a quantidade máxima de execuções da função ***exec*** segundo um parâmetro **N** definido a priori. Sua solução terá pontuação máxima caso considere critérios de justiça e evite *starvation*. De qualquer maneira, uma solução sem esses critérios atendidos é aceitável (declare em sua resposta se você pretende atendê-los).

Comentários gerais:

- Não mude a função ***exec***;
- Altere o código da função ***handle*** para implementar as restrições de controle de concorrência. Crie funções auxiliares se achar necessário;
- Crie uma função ***main*** para iniciar e criar variáveis globais incluindo semáforos.

# Benchmarking

Em muitos casos, queremos entender o desempenho de sistemas quando usados concorrentemente por múltiplas threads. Digamos, por exemplo, que queiramos testar o método *put* de um *Map* em Java. Uma métrica possível para avaliação de desempenho é a duração da execução (makespan). Essa duração total considera o tempo que ***num_threads*** threads demoram, neste caso, para executar a função *put*. Implemente a função *benchmark* abaixo, que calcula o makespan, para a execução da função *put*, concorrentemente, por ***num_threads*** threads. A função deve retornar o *makespan*. Ainda, sua função deve estar atenta para duas coisas. Primeiro, você precisa manter o nível de contenção desejado em ***num_threads***. Ou seja, você deve evitar que, por exemplo, uma determinada thread (ou subgrupo de threads) seja criada e executada antecipadamente. Lembre que queremos que o método ***put*** de maneira concorrente.

Ainda, temos que esperar a última thread terminar para calcular o timestamp final. Considere algumas funções e construções utilitárias.

Para obter timestamps, use a função *long System.timestamp()*. Ou seja, para calcular o *makespan,* você irá fazer duas chamadas, e calcular a diferença entre os *timestamps*. Em algo do tipo:

```
long begin = System.timestamp()
/// …
long end = System.timestamp()
```

Para criar fluxos de execução em java, você precisa fazer duas coisas. Criar um objeto do tipo *Thread* e iniciar a thread. Ainda, você precisa definir o *Runnable* a ser executado pela thread. A criação pode ser feito como abaixo, com uma classe anônima:

```
Thread t0 = New Thread() {
   public void run() {
   ///decl. o cód. a executar pela thread. p.ex o acesso ao mapa
   }
};
```

em seguida, para que a thread seja executada, você deve fazer o seguinte:

```
t0.start()
```

Considere que os valores a serem adicionados no mapa não importam. Cuidado também para incluir na contagem do *makespan* somente o mínimo necessário, não inclua trechos inúteis.

```
public long benchmarking(int num_threads, Map<Int, Int> targetMap)
```

# Stop worrying and Learn to love prodcon

O processamento de entrada e saída, p.ex para dispositivos de rede, envolve a coordenação entre os processos que usam os dados e o sistema operacional (que de fato, interage com o dispositivo). Considere as funções ***receive*** (executada pelos processos) e ***handle*** (executada pelo sistema operacional).

```
//o processo recebe um novo pacote em um array de bytes
void receive() {
       for (;;) {
              byte[] = getDataPacket();
       }
}

//O SO lê o buffer da placa de rede. o buffer tem K pacotes
```

```
//após a leitura, injeta os pacotes na memória que então ficam disponíveis
//para os processos através da função receive
void handle() {
	for (;;) {
		byte[][] data = readNetBuffer();
		fillPackets(data);
	}
}
```

Considerando que é possível que múltiplos processos/threads (com quantidade indefinida a-priori) executam a função receive, e somente uma thread executa a função handle (que sempre lê **K** pacotes da placa de rede), altere as funções ***receive*** e ***handle***, para coordenar a execução das threads considerando as seguintes regras:

a) o SO deve enviar novos pacotes para a memória somente se ela estiver vazia;
b) uma thread não pode executar ***getDataPacket*** caso não existam pacotes na memória gerados pela função ***handle***.

No caso da memória estar vazia, qualquer thread que executa ***receive*** (do espaço do usuário) deve acordar a thread do SO. Enquanto a thread do SO não termina o seu trabalho, a thread do espaço do usuário deve dormir. Depois que a thread do SO terminar seu trabalho, deve também dormir.

Crie uma função ***main*** para inicializar os semáforos criados

# Rate limiting or admission control?

Alguns sistemas críticos controlam tanto o número máximo de requisições que podem receber, quanto a quantidade das requisições que, uma vez recebidas, podem ser atendidas. Considere que um sistema desse tipo, ao receber uma requisição, cria uma uma nova thread que executa a função ***run***. Além disso, o sistema tem uma única thread para atender as requisições, que executa a função ***handle*** em loop infinito.

```
void run();

void poison();
void encrypt()

void handle() {
	for(;;) {
		generateKey();
	}
}
```

As threads que executam ***run*** precisam verificar se o sistema já está atendendo o número máximo de **K** outras threads. Caso já esteja, a thread deve chamar a função ***poison*** que fará com que a thread termine (e, portanto, o número máximo de requisições no sistema não passe de **K**), ou seja, a chamada para ***poison*** não retorna.

Caso não exista nenhuma thread de requisição, a thread do sistema (que atende as requisições) deve bloquear. Caso a thread do sistema esteja bloqueada, uma nova requisição que tenha chegado para executar ***run*** deve acordar a thread do sistema. A thread do sistema executa ***generateKey*** para atender uma requisição. Por sua vez, a thread de requisição executa ***encrypt***. A execução de ***generateKey*** deve ser concorrente com somente uma execução de ***encrypt.***

Notem que a função ***encrypt*** executa o trabalho útil da thread de requisição. Ou seja, após terminar a execução de ***encrypt***, a thread pode encerrar a execução do método ***run*** (obviamente, não é obrigatório que ***encrypt*** seja a última linha do código) normalmente (ou seja, não é necessário executar ***poison*** para esse caso).

Considerando as restrições acima, escreva/modifique as funções run e handle acima para coordenar a execução das threads.

Crie uma função ***main*** para inicializar os semáforos criados.

# Categorical exclusion

Considere um sistema acessível por uma API REST. Essa API tem somente duas funções. Por coincidência, uma foi implementada por POST enquanto outra por GET. Este sistema consome muitos recursos, e por isso há um controle muito forte da quantidade possível de requisições concorrentes (não podem ultrapassar uma constante N, definida estaticamente) qualquer que seja o tipo das requisições. Além disso, o sistema tem apresentado problemas ainda não compreendidos pelo time de desenvolvimento quando requisições de tipos diferentes são executadas de maneira concorrente. Por esse motivo, além do controle da quantidade de requisições concorrentes, você precisa implementar um controle adicional que não permita a execução concorrente de requisições de tipos diferentes.

Considerando a especificação acima, e considerando que uma nova thread é criada na chegada de cada requisição (ou seja, não se preocupe com a criação das threads), implemente as funções abaixo:

//implementa o controle de concorrência para o tipo de requisição //que usa o método POST. Uma vez estabelecido o controle, chama //a função **do_post**.
**handle_post()**

//implementa o controle de concorrência para o tipo de requisição //que usa o método GET. Uma vez estabelecido o controle, chama a //função **do_get.**
**handle_get()**

//inicia variáveis globais e semáforos
**main()**

# Token Bucket

Em redes de computadores e em muitos sistemas distribuídos (p.ex sistemas para a web) é comum a implementação de mecanismos de limitação de taxa requisições. Esses mecanismos impedem que a frequência de requisições ultrapasse algum limite pré-estabelecido. Essa medida preventiva permite proteger os sistemas contra o uso excessivo, malicioso ou não, mantendo os níveis de disponibilidade adequados.

Considere um sistema que deve usar um controle desse tipo. Cada requisição que chega nesse sistema implica na criação de um novo fluxo que executa a função **run** (seja uma nova thread ou uma nova goroutine).

```
func run(Request req) {
      limitCap_wait(req);
      handleReq(req);
}
```

Você deve implementar a função **limitCap_wait** o comportamento especificado da função implica que qualquer thread que executa a função deve bloquear caso o limite de taxa de requisições tenha sido atingido.

O seu controle de requisições deve ser baseado numa versão de ***token bucket*** (https://en.wikipedia.org/wiki/Token_bucket). Nessa versão, você deve considerar que:

1. O bucket tem uma capacidade máxima de **B** tokens. Você pode assumir que o bucket inicia cheio;
2. Um token é adicionado ao bucket a cada 1/**R** segundos;
3. Se, ao tentar adicionar um novo token, o bucket estiver cheio, o token deve ser descartado (não deve ser adicionado ao bucket);
4. As requisições têm tamanho variável. O tamanho de uma requisição pode ser obtido através da variável **Request.size;** O atendimento de uma requisição consome **Request.size** tokens do bucket;
5. Quando uma requisição de tamanho **Request.size** tenta executar a função **run**, se pelo menos **Request.size** tokens existem no bucket, **Request.size** são removidos do bucket e a função **limitCap_wait** não bloqueia;
6. Se menos que **Request.size** tokens existem no bucket, os tokens não são removidos e a função **limitCap_wait** bloqueia (podem ser removidos posteriormente, quando novos tokens forem disponibilizados e a função desbloqueada);
7. O tamanho máximo de uma requisição é **B.**

Você pode considerar alguns funções utilitárias, tais como:

- use **sleep(m milisegundos)** - para que um fluxo de execução durma por **m** milisegundos;
- use new Thread(Runnable r), para criar um novo fluxo de execução, implementado na função run, de Runnable;

Comentários gerais:

- Não se preocupem com starvation nem com priorização de requisições;
- Implemente uma função **main** que, no mínimo, inicie os semáforos, threads, variáveis locais e globais necessários. Esses aspectos de inicialização são considerados na avaliação;
- Além da **limitCap_wait** e da **main,** implemente qualquer outra função que se faça necessária;
- Embora o design sugerido seja funcional, se achar útil, crie novos tipos/objetos para modelar o comportamento desejado;
- Crie as estruturas de dados que achar necessárias;
- Caso ache mais simples, implemente em Java, ou em pseudo-código parecido com java;
- Tenha cuidado para não inserir deadlocks. Haverá penalização alta, nesse caso;
- Tente simplificar seu código ao máximo. Você pode ser penalizado por usar soluções mais complexas do que o necessário;
- Cuidado com o uso de var. globais. Use conforme acredite que seja útil mas tenha cuidado de não coordenar as threads com var. globais (e, esperas-ocupadas) quando semáforos for uma melhor escolha

```
/// Appendix
/// o código abaixo exemplifica como usar Runnable para iniciar uma nova thread em Java.
/// lembrando que não será avaliado se o código compila ou não. Entenda o exemplo, e o use na prova, como pseudo código

public class RunnableExample implements Runnable {

/// implementacoes de Runnable pode ter construtores. como qq construtor, vcs podem //passar argumentos

public RunnableExample(String xpto) { }

/// a implementação do método run define o que será executado pela thread
public void run () {
	System.out.println(“Oi mundo”);
}

///  para que o código definido no runnable execute, eh preciso criar um objeto Thread
// e passar uma instância do Runnable como argumento
Thread myThread = new Thread(new RunnableExample(“blau”));

//  em seguida, após a criação, a thread precisa ser iniciada. eventualmente, o sistema //operacional/jvm poderá executar a thread
myThread.start()
```

# Request Control

Em certos sistemas evitamos que requisições de tipos diferentes sejam executadas concorrentemente.

Considere o código abaixo:

```
func handle(Request req) {
      exec(req)
}
```

Considere que a função ***handle*** é executada por uma thread toda vez que uma requisição chega no sistema. Você não precisa se preocupar com a criação dessa thread. Uma requisição tem um tipo inteiro, ou seja, Request.type. Em nosso caso, há dois tipos (0 e 1). Altere o código da função ***handle*** para controlar a execução do sistema de maneira que não seja possível a execução concorrente pela função **exec** de requisições de tipos diferentes. Além disso, controle a quantidade máxima de execuções da função ***exec*** segundo um parâmetro **N** definido a priori. Sua solução terá pontuação máxima caso considere critérios de justiça e evite *starvation*. De qualquer maneira, uma solução sem esses critérios atendidos é aceitável (declare em sua resposta se você pretende atendê-los).

Comentários gerais:

- Não mude a função ***exec***;
- Altere o código da função ***handle*** para implementar as restrições de controle de concorrência. Crie funções auxiliares se achar necessário;
- Crie uma função ***main*** para iniciar e criar variáveis globais incluindo semáforos.

# Joining clang

(clang) Considere a API abaixo. A função **gateway** deve criar e iniciar **nthreads** pthreads diferentes. O código executado por cada pthread deve ser o da função request**.** A função request deve sortear um número aleatório **n** e dormir **n** segundos. Após criar as pthreads, a função **gateway** deve esperar que até **wait_nthreads** terminem. Após a espera, a função gateway deve retornar a soma dos valores **n** sorteados nas funções request.

**int gateway(int nthreads, int wait_nthreads)**

**void* request(void*)**

# Your own java lock

Em sala, discutimos uma implementação de um ***lock*** justo, em clang. Esse ***lock*** usava uma fila para manter identificadores de ***pthreads*** em espera para executar a região crítica. Uma vez liberado o ***lock***, a thread que entrou primeiro na fila deve ser escolhida para executar. Faça uma implementação em java com as mesmas características, seguindo a interface listada abaixo. Sua implementação não pode

usar os métodos ***wait***, ***notify*** e ***notifyAll*** da classe Object. Use ***ArrayList*** para implementar a fila. Você não pode usar ***synchronized*** na declaração de nenhum método criado por você. Você não pode usar nenhum objeto do pacote ***java.util.concurrent*** exceto ***java.util.concurrent.locks.LockSupport.park()*** para bloquear a execução da Thread corrente e ***java.util.concurrent.locks.LockSupport.unpark(Thread thread)*** para desbloquear. Adicionalmente, você pode usar ***java.util.concurrent.atomic.AtomicInteger*** para implementar o equivalente à instrução testAndSet caso topo não usar ***synchronized*** em nenhum ponto do código (nem em um bloco interno).

# Your own Latch

CountDownLatch é um sincronizador disponível na sdk Java (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html ). Faça uma nova implementação da mesma ideia usando somente variáveis condicionais (***wait***, ***notify*** e ***notifyAll***) e a construção ***synchronized***. Se precisar usar alguma coleção, use uma LinkedList ou ArrayList. Só é necessário implementar a API abaixo:

- void await()
- void countDown()

> Caso ainda tenha dúvidas sobre a semântica da CountDownLatch após ler sua documentação, procure o professor para tirar as dúvidas.

# Yet another meeting

Aplicativos para gerenciamento de compromissos são bastantes populares (p.ex google calendar, microsoft outlook e apple calendar). Nesses aplicativos, o usuário pode cadastrar eventos. No cadastro, o usuário dá como entrada um título para o evento, o momento de início e duração do evento. Uma funcionalidade interessante desses aplicativos é a capacidade de aviso o usuário que ele tem um compromisso agendado. Considerando a API abaixo, escreva o código de um aplicativo dessa categoria (implemente a interface AppointmentManager) que relembra o usuário (usando o método notify da interface AppointmentNotifier), começando uma hora antes do compromisso, a cada 15 minutos. Obviamente, após receber uma notificação, o usuário pode desejar não ser mais notificado (haverá uma chamada cancel em AppointmentManager).

- Interface Appointment
  - int getId()
  - String getDescription()
  - long start() //start é a data de início do evento em UNIX Epoch (milisegundos desde 1 de Janeiro de 1970)
  - long duration()
- Interface AppointmentManager
  - boolean addAppointment (Appointment toAddAppointment)
  - boolean cancel(int appointmentId)
- Interface Utils

        - long milliSecondsUntil(long timeStamp)
            - retorna quantos milisegundos falta para atingir a data timeStamp (especificada em UNIX Epoch)
    - Interface AppointmentNotifier
        - void notify(Appointment app)
            - notifica a UI sobre um compromisso previamente agendado

# Java Channels

Uma abstração bastante usada em programação concorrentes são os canais. Um canal recebe mensagens enviadas por processos (threads) remetentes. Processos recipientes lêem as mensagens enviadas no canal. Mensagens devem ser lidas na ordem que entraram no canal. Uma vez lida, a mensagem não pode ser lida novamente. O canal deve ter uma capacidade máxima, ou seja, ao atingir o limite, novas mensagens não podem ser enviadas para o canal imediatamente. Mensagens não podem ser descartadas. Implemente a interface abaixo para o canal, usando quaisquer mecanismos de coordenação e controle de concorrência da linguagem Java. Considere tanto critérios de corretude quanto de eficiência (p.ex evite spin locks quando possível).

```
public interface Channel {
	public void putMessage(String message);
	public String takeMessage();
}
```

# Reliable Request

Considere um sistema que precisa consultar um site web através de uma requisição HTTP. A chamada HTTP é encapsulada por uma API com um único método:

***String request(String serverName )***

Por motivos de tolerância à falhas, há três mirrors disponíveis com a mesma informação. Nesse sistema, escreva uma função que consulta os três mirrors (cujos *servernames* são: "**mirror1.com**", "**mirror2.br**" e "**mirror3.edu**") e retorna a resposta que chegar primeiro. A função deve implementar a seguinte assinatura:

***String reliableRequest()***

Justifique as decisões importantes em sua implementação. Por exemplo, as primitivas de concorrência usadas.

a) Considere uma extensão ao sistema anterior em que você deve escrever uma nova função que retorna o resultado da execução de ***reliableRequest*** ou um erro, se a execução desta durar mais do que 2 segundos.

b) Crie uma função que executa indefinidamente a função ***reliableRequest***, definida anteriormente, enquanto espera uma sinalização de parada enviada por outro fluxo de execução.

# ConcurrentHashMap

Implemente um HashMap que seja seguro para uso concorrente. Assuma que você só precisa implementar três funções desse Map:

1) o construtor, **new HashMap()**;
2) **put(int key, int value);** e
3) **boolean containsKey(int key)**

Assuma que internamente, o Map usará uma LinkedList. Você também precisa implementar a List e ela precisa também ser segura para uso concorrente. Assuma que você só precisa implementar três funções da List (mas, assuma também que deve existir um método remove(int value) que você não precisa implementar):

1) o construtor, **new LinkedList()**;
2) **add(int value);** e
3) **boolean contain(int value)**

Lembre que internamente, um HashMap é organizado em buckets. Um bucket reúne todos os **values** os quais as **keys** relacionadas tiveram o mesmo hashcode. Em sua implementação, cada bucket deve usar uma LinkedList. Então, todos os **values** de um mesmo bucket estão na mesma LinkedList. Assuma também que a quantidade de buckets é definida a priori (e, é igual a 1024). É importante que seu código seja eficiente, então cuidado para proteger somente a região crítica (evite proteger código além da região crítica).

# BinarySearchTree

Considere uma árvore de pesquisa binária que armazena números inteiros. A árvore é representada usando nós que possuem um atributo **value** (que armazena o valor inteiro), um atributo **left** para a subárvore esquerda e um atributo **right** para a subárvore direita. Suponha que a árvore esteja inicialmente vazia e que várias threads possam acessar a árvore simultaneamente. Escreva pseudocódigo para implementar as operações de inserção e pesquisa da árvore de maneira segura para thread usando semáforos ou variáveis condicionais. Você pode usar o modelo abaixo, como referência para sua implementação.

```
class BinarySearchTree()
```

```
void func insert(int valueToInsert)
boolean func search(int valueToSearch)

class Node(int value)
      this.value = value
      this.left = None
      this.right = None
```

Você também pode modificar a pergunta pedindo aos alunos que implementem a operação de exclusão em vez de pesquisa. Como alternativa, você pode pedir aos alunos que implementem uma implementação thread-safe de uma estrutura de dados de gráfico usando semáforos ou variáveis condicionais.

# IO-SUBMIT

Considere a API abaixo, para controle requisições de IO em um sistema operacional

**func iop(Request req)**
**func submit(Request req)**

Considere que múltiplas threads podem chamar a função iop. O tipo Request é definido da seguinte forma:

```
type Request {
      //indicar se a requisição é para uma op de leitura ou escrita
      enum op_type {R, W};
      //indica o id do bloco para o qual a req será feita
      int block_id;
      //conteúdo a ser lido/escrito para cada o bloco
      byte[] buffers;
}
```

Considere que não é bacana submeter, num curto espaço de tempo (vamos chamar de MERGE_INTERVAL), operações de um determinado tipo para um mesmo bloco. Ou seja, é ruim ter duas ou mais operações de escrita para o mesmo bloco dentro de um MERGE_INTERVAL.

Escreva a função iop (e funções auxiliares) que controla o acesso à função submit e aglutina requisições de um mesmo tipo para um mesmo bloco. Ou seja, você deve juntar Requests para um mesmo bloco em uma Request única. Isso pode ser feito através da função auxiliar **merge** abaixo:

**func merge(Request one, Request two) Request**

Você pode usar também a função **now** que retorna o instante de tempo atual (tal como unix epoch, unidades de tempo desde 1970)

**func now() Timestamp**

Ou seja, se precisa calcular tempo decorrido, você pode fazer algo do tipo: **(now() - oldTimeStamp) > MERGE_INTERVAL**

Dica uma vez que você não quer que duas requests para um mesmo bloco sejam submetidas dentro de um mesmo MERGE_INTERVAL, isso significa que você pode atrasar a submissão de requests (mantendo requests em uma estrutura de dados auxiliar pelo tempo que for necessário). Ainda, você é livre para implementar a função iop de maneira bloqueante ou não. Ou seja, a thread que chama iop não precisa esperar até que a execução da request seja terminada. Você também pode criar novas threads (com a API parecida com qualquer linguagem vista em sala).

# LockOne&LockTwo

Considere os algoritmos LockOne e LockTwo. Indique exemplos de execução, para cada um dos algoritmos, em que são quebrados requisitos de segurança e/ou progresso. Explique quais os problemas que acontecem em cada caso. Use a seguinte notação para indicar os exemplos de execução (T0_5 -> T0_6 -> T1_5). Essa notação indica que a thread T0 executou as linhas 5 e 6 e depois a thread T1 executou a linha 5.

# Peterson

Explique como o algoritmo de Peterson garante que os problemas apresentados nos algoritmos LockOne e LockTwo são resolvidos.

# TTAS Lock

Explique como travas TTAS podem ter desempenho melhor que travas TAS. Sua explicação deve considerar aspectos de arquitetura de computadores.

Considere a implementação BrokenBakery. Que problemas, de progresso e segurança, essa implementação incorreta do algoritmo apresenta.

```
class BrokenBakery implements Lock {

    volatile int[] ticket;

    public Bakery (int n) {
        ticket = new int[n];
        for (int i = 0; i < n; i++) {
            ticket[i] = 0;
        }
    }

    public void lock() {
        int id = Thread.getID()
```

```
        for (int j = 0; j < n; j++)
            if (ticket[j] > ticket[id])
                ticket[id] = ticket[j];
        ticket[id]++;


        for (int j = 0; j < n; j++) {
            while ((ticket[j] != 0) && ((ticket[j] < ticket[id]));
        }
    }

    public void unlock() {
            int id = Thread.getID()
            ticket[id] = 0;
    }
}
```

# SimpleBakery

Refatore a implementação do algoritmo Bakery usando AtomicInt e AtomicBoolean. Simplifique o código ao máximo.

# Canais 101

Explique e justifique a saída esperada do programa abaixo.

```
package main
import "fmt"

func xpto(c chan int, int value) {
        c <- value
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go xpto(ch1, 42)
    go xpto(ch1, 43)

    select {
    case v1 := <-ch1:
        fmt.Println("valor recebido de ch1:", v1)
    case v2 := <-ch2:
        fmt.Println("valor recebido de ch2:", v2)
    }
}
```

# FAN-IN

Considere a API abaixo como uma função que retorna um canal no qual um número indeterminado de strings serão enviadas.

```
func request_stream() chan string
```

considere que em um programa, dois canais destes estão sendo manipulados da seguinte forma:

```
func main() {
	ch1 := request_stream()
	ch2 := request_stream()

}
```

considere que você deve incluir na sua função main uma chamada para a função

```
func ingest(in chan string)
```

que deve ser chamada como uma nova goroutine, ou seja:

```
go ingest
```

agora, o canal recebido pela função **ingest** deve conter itens disponibilizados pelos canais ch1 e ch2 na medida em que estiverem disponíveis. Ou seja, tão logo itens de ch1 e ch2 estejam disponíveis, estes podem ser enviados para o canal a ser passado para a função ingest.

# ADMISSION CONTROL & CHAN

Considere um sistema que cria requisições a serem enviadas para componentes de processamento (workers). Esse sistema é crítico e não deve **criar** mais do que **maxCapacity** requisições. Considere que uma requisição é criada com a função abaixo:

**func create_req() Request**

Considere que uma requisição é processada através da execução da função abaixo. Esse processamento demora muito, então deve ser intermediado por goroutines.

**func exec_req(req Request)**

É importante que o sistema trabalhe na maior capacidade possível. Ou seja, se for possível criar mais requisições e mandá-las para processamento, isso deve ser feito. Ainda, não deve haver limitação do ponto de vista dos workers: caso uma requisição tenha sido criada é melhor que um worker a processe tão logo possa.

Altere o código abaixo, criando goroutines, canais, estrutura de dados e funções auxiliares para resolver o problema. Assuma que o problema irá funcionar indefinidamente.

```
package main

const maxCapacity = 10 // Maximum capacity of the system

func main() {

	//loop de criação de requisições. altere para realizar o controle de criação
	for {

		var req = create_req()
	}
}
```

# FORK-SLEEP-JOIN CSP

Crie um programa que recebe um número inteiro **n** como argumento e cria **n** goroutines. Cada uma dessas goroutines deve dormir por um tempo aleatório de no máximo 5 segundos. A main-goroutine deve esperar todas as goroutines filhas terminarem de executar para em seguida escrever na saída padrão o valor de **n**.