1 - Crie um programa que recebe um número inteiro **n** como argumento e cria **n** threads. Cada uma dessas threads deve executar uma função (void **foo()**). Esta função deve fazer duas coisas: 1) sortear um número aleatório (através da função **int rand()**); 2) dormir por um tempo em segundos igual ao número aleatório sorteado (use a função **sleep(int seconds)** para isso). A thread mãe deve criar as **n** threads filhas através das função (**create_thread(func, arg)**).

A thread mãe deve esperar que todas as filhas terminarem de executar suas funções **foo**. Após as thread-filhas terminarem seu trabalho, a thread mãe deve retornar o maior e o menor valores entre os valores sorteados pelas threads filhas. Implemente a função **main** e a função **foo** indicadas.

2 - Considere a implementação de um Pilha, que internamente usa vetores, com pelo menos a seguinte interface. Para este problema, considere que a pilha tem capacidade máxima definida em 1000 elementos.

```
//cria uma pilha
new Stack();

//verifica se a pilha está vazia
boolean isEmpty();

//remove e retorna o item no topo da pilha
int pop();

//adiciona um novo item no topo da pilha
void push(int value);
```

Sua implementação deve ser correta considerando que um número arbitrário de threads usará o heap concorrentemente. Corretude é mais importante do que desempenho (embora desempenho também seja importante, ou seja, não proteja regiões desnecessariamente). Você deve usar semáforos para desenvolver sua solução concorrente segura.

3 - Considere a implementação de um sistema de processamento de requisições que funciona no modo **master/worker.** Neste tipo de sistemas, temos duas entidades distintas: 1) um master (chamado abaixo de broker); e 2) workers. O broker é a entidade para a qual requisições são submetidas. Os workers são entidades que processam requisições. A API do broker tem duas funções, listadas abaixo:

```
type Broker
     void submitRequest(Request r);
     Request getWork();
```

Não precisa com threads

Considere que o Broker tem capacidade finita N. Threads diferentes podem chamar a função ***submitRequest****. O Broker mantém as requests submetidas em um vetor de tamanho N. Workers são threads que retiram requests do Broker através da função* **getWork.** *Caso tenha sido submetido muitos requests, a ponto de o vetor de requests encher sua capacidade, threads que submeter requests devem bloquear até que requests tenham sido retiradas. Por sua vez, workers devem bloquear caso não tenhamos requests disponíveis. Uma request submetida precisa ser processada por um único worker.*

SENHA: PROVA1@PC

Considere a seguinte API para o worker:

```
class Worker(Broker broker);

run();
void exec(Request req)
```

No método run, o worker deve executar em um loop infinito. O worker precisa retirar uma Request do broker (chamando **getWork**) e depois executar o request com a função **exec** (que você não precisa implementar).

Considere que:

1. threads serão criadas para executar os workers, portanto não se preocupe com isso;
2. Você deve inicializar semáforos no main criado por você. Os semáforos podem ser passados no construtor dos objetos (você pode mudar os construtores para tanto);
3. A constante N é conhecida;
4. Você deve implementar todo o resto do código necessário para o sistema funcionar (pelo menos submitRequest e ~~stealRequest~~ getWork do Broker bem como run do Worker)

API Semáforos
Semaphore (int initialValue) wait() / signal()

p.s Das abstrações de concorrência vista em sala, você só pode usar semáforos. Qualquer abstração mais complexa (p.ex barreira etc) precisa ser implementada para ser usada.

submitRequest
getWork
run