Laboratório de Programação Concorrente

Lab2
Warmup Again - 25.2

Objetivo

Dominar a sintaxe básica para **criar, nomear e iniciar Threads** em **Python e C**, observando o comportamento de concorrência simples (intercalação de execução) e aplicando as boas práticas de encapsulamento de tarefas.

Cenário Básico: O Processo de Correção e Alocação de Notas de Provas

A cada início de período letivo, os professores retornam renovados e cheios de energia para desempenhar suas atividades. Um cenário comum na UFCG é que muitas disciplinas possuem mais de uma turma (nossa disciplina é um desses exemplos), com múltiplos professores lecionando o mesmo conteúdo. Neste novo semestre, a UFCG definiu que todas as turmas de uma mesma disciplina sempre devem receber as notas de cada prova juntas, ou seja, de forma sincronizada. Dessa forma, apesar de todos os professores de uma mesma disciplina já trabalharem de forma concorrente para corrigir e atribuir notas para as provas de sua turma, vai ser necessário uma sincronização entre esses professores para a divulgação dessas notas só ocorrer após todos finalizarem esta tarefa.

Durante o processo de correção, cada professor precisa:

1. Selecionar os alunos da sua turma.
2. Corrigir as provas de cada aluno.
3. Registrar as notas no sistema.

As duas primeiras tarefas são **independentes** o suficiente para serem executadas concorrentemente – por exemplo, através de threads ou processos paralelos. No entanto, o registro das notas no sistema deve acontecer apenas após todos os professores da disciplina em questão finalizarem a correção das provas de suas respectivas turmas.

Neste laboratório vamos simular a coordenação e sincronização de tarefas concorrentes, considerando o cenário de correção e alocação de notas das provas. Nesse caso, *n* threads trabalham concorrentemente para corrigir as provas (representando os professores e suas respectivas turmas), mas a *thread* principal deve

aguardar todas finalizarem antes de prosseguir com a etapa final (divulgação das notas).

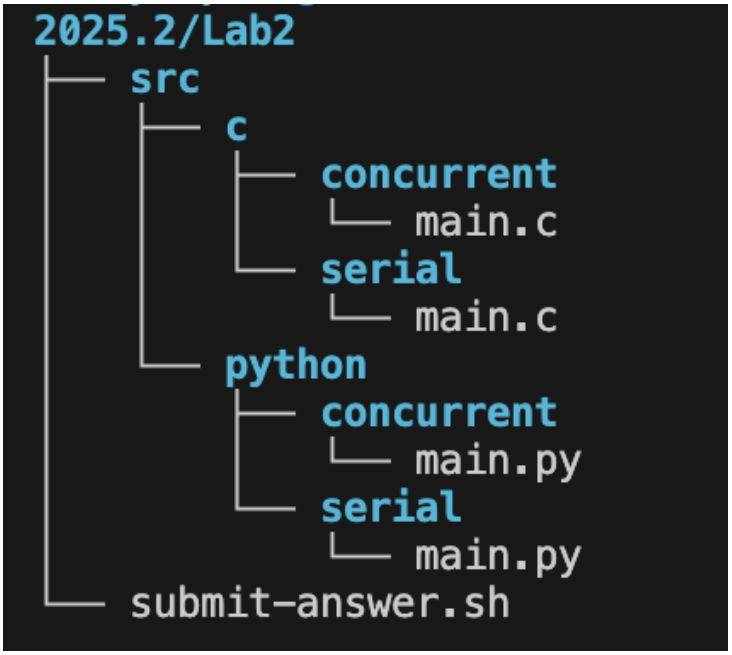
Visão geral do código base

No código base vocês encontrarão implementações serial e concorrente em em duas linguagens de programação: Python e C. Considere que todas as turmas possuem a mesma **quantidade de alunos**, e, em ambas as soluções, o usuário deve informar a **quantidade de turmas** e a quantidade de alunos por turma. No entanto, é importante destacar que as soluções concorrentes disponibilizadas estão **incompletas.**

A entrega, detalhada nas seções seguintes, envolverá o código fonte e análises de execução. Iremos avaliar tanto as possibilidades de plágio entre os alunos quanto a geração automática de código.

https://github.com/giovannifs/fpc/tree/master/2025.2/Lab2

O código está organizado na seguinte hierarquia:



Preparação

1. Clone o repositório do código base

   git clone [link do repositório]

2. Execute a versão serial da solução. Para isso, você deve ir até o diretório

   a. 2025.2/Lab2/src/python

      bash run.sh S <qtd_turmas> <qtd_alunos_por_turma>

   b. 2025.2/Lab2/src/c

      bash build.sh S

      bash run.sh <qtd_turmas> <qtd_alunos_por_turma>

   Entendendo o output do script run.sh:

   - real: o tempo total decorrido
   - user: o tempo total que o processo gastou utilizando a CPU em modo usuário
   - sys: o tempo total que o processo gastou utilizando recursos do kernel

   Interpretação

   - **real: é o tempo que você veria em um cronômetro**
   - user + sys: representa o tempo efetivamente gasto pela CPU no processamento

   Se o programa usar múltiplas threads em um sistema com vários núcleos, o valor de user pode ser maior que real, já que múltiplas threads podem trabalhar simultaneamente.

Execução da solução serial

Execute algumas vezes a solução serial (para ambas as linguagens) e verifique seu comportamento através dos logs. Houve alguma alteração na ordem de execução das correções entre as diferentes execuções?

Crie o diretório **comments** dentro do diretório **Lab2/src,** e, dentro do novo diretório, crie o arquivo **comments1.txt** com sua análise sobre a execução da solução serial.

Desenvolvendo uma solução concorrente

O mecanismo de gerenciamento das threads tem três etapas iniciais:

1) a definição do código a ser executado pela thread;

2) a criação da thread; e, por fim,

3) a inicialização da thread.

Desenvolvendo as soluções concorrentes

Python

Para implementar a concorrência em python, utilizamos a biblioteca ***threading***, que é a biblioteca padrão do Python para trabalhar com threads. Ela nos permite criar threads de execução paralela usando a função ***Thread()***.

Exemplo de uso:

A função ***Thread()*** cria um objeto thread que pode executar uma função específica. Ela recebe dois argumentos:

1. **target**: a função que a thread irá executar
2. **args**: uma tupla contendo os argumentos que serão passados para a função.

Sintaxe:

```
process = threading.Thread(target=func, args=(arg1func, arg2func,))
// A última vírgula nos argumentos da função pode ser necessária.
```

Assim como fizemos no Lab anterior em Java, após criar as threads precisamos iniciá-las. Para isso, utilizamos a função ***start().*** Esta função é que de fato inicia uma thread para executar a sua tarefa atribuída.

Ao precisar sincronizar o fluxo de execução de diferentes threads, podemos usar a função ***join().*** Esta função tipicamente é usada para fazer uma Thread (mãe) esperar por uma thread (filha). Considerando que t1 é uma thread criada e iniciada em um trecho de código

anterior, o exemplo abaixo representa como fazer a thread (mãe) esperar até t1 ser finalizada:

```
print("Waiting for thread t1...")
t1.join()
print("Thread t1 finished!")
```

O código ***print("Thread t1 finished!")*** vai executar apenas quando t1 terminar sua tarefa, ou seja, a thread mãe executa o ***t1.join()*** e espera até que t1 finalize.

Considerando o passo a passo acima, complemente a solução concorrente disponibilizada neste laboratório de tal forma que as correções das atividades sejam realizadas de forma concorrente e que as notas sejam divulgadas após a finalização de toda correção.

Para executar a versão concorrente de Python, você deve ir ao diretório 2025.2/Lab2/src/python/concurrent e executar:

```
bash run.sh C <qtd_turmas> <qtd_alunos_por_turma>
```

C

Para implementar concorrência em C, utilizamos a biblioteca **POSIX Threads (pthreads)**, que é a biblioteca padrão para criação e controle de threads em sistemas Unix-like.

Ela nos permite criar threads de execução paralela usando a função pthread_create(). Por exemplo:

A função pthread_create() cria uma nova thread e executa uma função específica. Sua assinatura é:

```int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);```

- **thread**: ponteiro para a variável do tipo pthread_t que irá identificar a thread criada.
- **attr**: atributos da thread (geralmente usamos NULL para os padrões).
```

- **start_routine**: ponteiro para a função que será executada pela thread.
- **arg**: argumento passado para a função (precisa ser um ponteiro void *).

Após criar as threads, precisamos iniciá-las com **pthread_create()** e, posteriormente, aguardar que elas terminem com **pthread_join()**.
A função pthread_join() bloqueia a thread principal até que a thread especificada termine sua execução.

Exemplo completo:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>  // para sleep()

void* func(void* arg) {
    char* name = (char*)arg;
    sleep(2);
    printf("%s finished\n", name);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // Criação das threads
    pthread_create(&t1, NULL, func, "Thread 1");
    pthread_create(&t2, NULL, func, "Thread 2");

    printf("Waiting for threads...\n");

    // Espera pelas threads terminarem
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("All threads finished\n");
    return 0;
}
```

O código printf("All threads finished") só será executado após as duas threads finalizarem, pois a thread principal bloqueia até o término de ambas com pthread_join().

Considerando o passo a passo acima, complemente a solução concorrente disponibilizada neste laboratório de tal forma que as correções das atividades sejam realizadas de forma concorrente e que as notas sejam divulgadas após a finalização de toda correção.

Para executar a versão concorrente de C, você deve ir ao diretório 2025.2/Lab2/src/c e executar:

```
bash build.sh C
bash run.sh <qtd_turmas> <qtd_alunos_por_turma>
```

Evoluindo as soluções

Usando a mesma estratégia das soluções concorrentes, evolua suas soluções em Python e C para, na medida que o professor computa as notas dos alunos, a nota mais alta da turma também seja identificada. Ao final das correções, além de exibir os registros de todas as notas, a thread principal também deve exibir a nota mais alta de cada uma das turmas. Note que este processamento não deve acontecer na thread principal, esta vai apenas exibir o que já foi computado pelas threads filhas, como já acontece com as notas registradas.

Questionamentos

Em suas soluções você sincronizou a execução das threads de tal forma que a thread principal esperasse a finalização de suas threads filhas. Ao implementar esta solução, você chegou a compartilhar algum espaço na memória (variável) entre as diferentes threads filhas e mãe? Se sim, qual? Isso pode levar a condições corridas? As múltiplas threads estão atualizando o mesmo espaço na memória? Temos algum problema na solução desenvolvida? Por que sim ou por que não?

Dentro do diretório **comments**, crie o arquivo **comments2.txt** com sua análise e comentários sobre esses questionamentos relacionados com a solução concorrente.

Prazo

28/10/2025 às 16h00

Entrega

Você deve criar e manter um repositório privado no GitHub com a sua solução. No entanto, a entrega do laboratório deverá ser realizada por meio de submissão online utilizando o script submit-answer.sh, disponibilizado na estrutura de arquivos do próprio laboratório. Uma vez que você tenha concluído sua resposta, seguem as instruções:

1) Crie um arquivo lab2_matr1_matr2.tar.gz somente com o “src” do repositório que vocês trabalharam. Para isso, supondo que o diretório raiz de seu repositório privado chama-se lab2_pc, você deve executar:

   ```
   tar -cvzf lab2_matr1_matr2.tar.gz lab2_pc/src
   ```

2) Submeta o arquivo lab2_matr1_matr2.tar.gz usando o script submit-answer.sh, disponibilizado no mesmo repositório do laboratório:

   ```
   bash submit-answer.sh lab2 path/lab2_matr1_matr2.tar.gz
   ```

Lembre-se que você deve manter o seu repositório privado no GitHub para fins de comprovação em caso de problema no empacotamento ou transmissão online. Alterações no código realizadas após o prazo de entrega não serão analisadas.