ESCOLA DE ARTES CIÊNCIAS E HUMANIDADES - USP

RELATÓRIO DO EXERCÍCIO DE PROGRAMAÇÃO I

SISTEMAS OPERACIONAIS - ACH2044

Nome: Guilherme José da Silva Nascimento
NUSP:12543252

Nome: Guilherme Cavalanti Gomes
NUSP:11844788

São Paulo - SP

09/2022

SUMÁRIO

INTRODUÇÃO

Este relatório consiste no desenvolvimento comentado de três problemas computacionais relacionados à Programação Concorrente dentro da Matéria de Sistemas Operacionais, matéria ministrada pela Professora Gisele Craveiro.

Nesse Relatório encontram-se três seções de desenvolvimento detalhado dos programas que solucionem cada problema, uma seção dedicada às conclusões gerais sobre a tarefa e uma seção final com referências bibliográficas utilizadas durante o desenvolvimento deste trabalho.

Os problemas em questão são:

1- Criação de um processo pai, “hello world”, fork processo filho, “hello world”, finalização de ambos. Desenvolvido em Python 3.10.

2 – Criação de um processo, criação de várias Java threads, “hello world” de todos, finalização. Desenvolvido em Java SE 11.

3 - Criação de um processo, criação de várias POSIX threads, “hello world” de todos, finalização. Desenvolvido em C.

DESENVOLVIMENTO DO PROGRAMA 1

Problema: Criação de um processo pai, “hello world”, fork processo filho, “hello world”, finalização de ambos.

Implementação: Para execução deste programa usaremos a linguagem interpretada Python 3.10.4, com o auxílio da biblioteca “os” que faz com que o programa interaja até certo nível com o Sistema Operacional da Máquina.

A priori iremos criar uma função que apenas fará um “fork” de um processo:

```
import os
def create_a_fork_1():
  "Função que cria um fork do processo a ser executado"
   n = os.fork()
   print(n)
```

Executando essa função teríamos o seguinte output:

```
bcrnoc@bcrnoc-Aspire-A315-23:~/so-ep1$ /bin/python3 /home/bcrnoc/so-ep1/ex1.py
62925
0
```

Percebe-se que mesmo havendo apenas um print na função create_fork_1( ) tivemos 2 resultados impressos em nosso terminal, o método fork( ) criou um processo filho idêntico ao processo pai, no primeiro print temos o “id” do processo pai e no segundo apenas a execução do processo filho, ele retorna 0 para caso a criação do Filho seja bem sucedida e -1 para caso tenha ocorrido algum erro.

Aplicando isso na execução de um print(“Hello World”) teríamos o seguinte código:

```
import os

def create_a_fork_1():
   "Função que cria um fork do processo a ser executado"
   n = os.fork()
   if n > 0:
       "Execução do Processo Pai"
       print("Hello World")
   elif n == 0:
       "Execução do Processo Filho"
       print("Hello World")
   else:
       print("Criação do Filho Falhou")
```

Na execução desse código temo o print duplicado da mesma mensagem “Hello World”, ambos processos são finalizados juntos ao fim do programa:

```
bcrnoc@bcrnoc-Aspire-A315-23:~/so-ep1$ /bin/python3 /home/bcrnoc/so-ep1/ex1.py
Hello World
Hello World
```

Uma maneira interessante de visualizarmos que são dois processos diferentes sendo executados seria visualizar os ID de ambos os processos, elemento este que podemos implementar usando o método os.getpid( ).

Este método em questão nos retorna o ID do processo executado neste caso teríamos a seguinte implementação de código:

```
import os


def create_a_fork_2():
   "Função que cria um fork do processo exibindo o id do processo"
   n = os.fork()
   if n > 0:
       "Execução do Processo Pai"
       print("Hello World", os.getpid())
   elif n == 0:
       "Execução do Processo Filho"
       print("Hello World", os.getpid())
   else:
       print("Criação do Filho Falhou")
```

Com esta implementação teremos o ID do processo impresso ao lado de cada Hello World, percebe se no Output abaixo que o processo Pai tem um ID menor que o processo filho pois seu processo foi gerado antes no Interpretador:

```
bcrnoc@bcrnoc-Aspire-A315-23:~/so-ep1$ /bin/python3 /home/bcrnoc/so-ep1/ex1.py
Hello World 64001
Hello World 64002
```

64001 --> Processo Pai
64002 --> Processo Filho

A execução continua desta segunda função desenvolvida gerará sempre dois IDs diferentes sendo o segundo ID a iteração do primeiro ID. (Ex: Processo Pai 00001, Processo FIlho 00002)

DESENVOLVIMENTO DO PROGRAMA 2

Problema: Criação de um processo, criação de várias Java threads, “hello world” de todos, finalização.

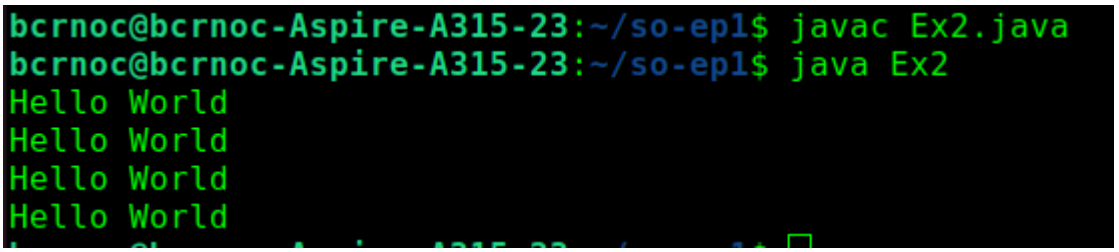
Implementação: Para o desenvolvimento deste programa foi utilizado a Linguagem JAVA SE em sua versão 11, nesta linguagem podemos implementar o sistema concorrente de Threads usando biblioteca nativa “Java Lang Thread", instanciando uma Classe Thread para cada Thread que desejarmos criar.

Para a resolução do problema apresentado iremos criar 4 Threads que irão apenas imprimir Hello World:

```
import java.lang.Thread;

public class Ex2 {
   public static void main(String[] args) {
       //Booleano para sincronizar a execução das Threads
       boolean[] next = { false };

       //Criação da primeira Thread
       Thread t1 = new Thread(() -> {
           synchronized (next) {
               System.out.println("Hello World");
       }});

       //Criação da segunda Thread
       Thread t2 = new Thread(() -> {
           synchronized (next) {
               System.out.println("Hello World");
       }});

       //Criação da terceira Thread
       Thread t3 = new Thread(() -> {
           synchronized (next) {
               System.out.println("Hello World");
       }});

       //Criação da quarta Thread
       Thread t4 = new Thread(() -> {
           synchronized (next) {
               System.out.println("Hello World");
```

```
        }});

        // Chamada das 4 Threads
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Ao executarmos este código no terminal o Resultado esperado são quatro “Hello Worlds” impressos na tela:



Em Java ao criarmos Threads podemos utilizar os métodos start() e sleep(), para manipularmos elas da melhor forma possível, o comando start executa o que foi programado para Thread realizar enquanto o comando sleep coloca Thread para “dormir” dando espaço para que outro processo possa ser executado. No exemplo por se tratar apenas de print não tivemos lentidão perceptível na concorrência das Threads talvez no orquestramento de processos mais custosos tal implementação não seria a mais eficiente.

DESENVOLVIMENTO DO PROGRAMA 3

Problema: Criação de um processo, criação de várias POSIX threads, “hello world” de todos, finalização

Implementação: Para execução deste programa usaremos a linguagem C, compilador gcc versão 6.3.0, utilizando as biblioteca ‘pthread.h’ e ‘unistd.h’ que permitem a criação de POSIX threads.

A primeira parte é definir a função que vai ser executada, ela recebe um parâmetro ‘threadid’ e imprime o número da thread e “Hello World”, após isso ela termina a thread.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *PrintHello(void *threadid){//função a ser executada quando uma
//thread for criada
  long id;
  id = (long)threadid;//conversão de void pra long
  printf("Thread %ld: Hello World\n", id);
  pthread_exit(NULL); //finalização da thread
}
```

A próxima parte é a função main, primeiro criamos duas threads usando o tipo “pthread_t”. Depois, usando a função “pthread_create” e passando como parâmetro o endereço das threads e a função “PrintHello”, as threads são iniciadas. Depois a função “pthread_join” garante que a thread já foi executada antes de terminar ela.

```c
int main(){
    pthread_t t1, t2; //criando t1 e t2 do tipo thread
    int create; //variável que recebe o retorno da função
//pthread_create()
    long num; //identificador da thread

    num = 1;
    create = pthread_create(&t1, NULL, PrintHello, (void *)num);
//criando thread 1

    num = 2;
```

```
    create = pthread_create(&t2, NULL, PrintHello, (void *)num);
//criando thread 2
    pthread_join(t1,NULL); //Espera a t1 acabar
    pthread_join(t2,NULL); // Espera a t2 acabar

    return 0;
}
```

Ao executar o código, a saída são 2 “Hello World”, 1 de cada thread como esperado:

```
C:\Users\W10\Desktop\so-ep1>thread.exe
Thread 1: Hello World
Thread 2: Hello World
```

CONCLUSÃO

Durante o desenvolvimento deste relatório pudemos exercitar muito melhor os conceitos de concorrência e orquestramento de múltiplos processos com três atividades práticas. Todas as atividades conseguiram o resultado esperado, foi determinante para o bom desempenho delas o desenvolvimento guiado pela documentação das linguagens escolhidas.

A primeira atividade foi a mais simples, a criação de processos filhos é algo muito bem desenvolvido na biblioteca “os” utilizada tornando a solução pequena e clara com a possibilidade de identificar os processos com IDs.

A segunda atividade foi uma que demandou uma leitura de documentação mais profunda para entender completamente o funcionamento da Classe Thread e de seus métodos para orquestrar a concorrência dos processos. Seu resultado cumpriu as expectativas esperadas de maneira mais fácil do que o imaginado.

Na última atividade tivemos algumas dificuldades na execução dos comandos no Windows, mas ao utilizar máquinas virtuais linux e ler mais a fundo a Documentação Oficial das “lpthreads” obtivemos um resultado extremamente semelhante ao da segunda atividade, fator que nos indica um resultado satisfatório.

Ao fim pudemos concluir que a manipulação de Threads é uma ferramenta poderosa e complexa que é essencial não somente para o bom funcionamento de um sistema operacional como também é um ótimo mecanismo para lidar com programas com processos muito grandes e repetitivos, deve ser muito útil para realizar migrações extensas de dados.

REFERÊNCIAS

SILBERSCHATZ, ABRAHAM Sistemas Operacionais com Java 8a. Edição, Editora LTC

Linux Tutorial: POSIX Threads, Acessado em 29/09/2022 em Link: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

Java Threads W3schools, Acessado em 28/09/2022 em Link Web: https://www.w3schools.com/java/java_threads.asp