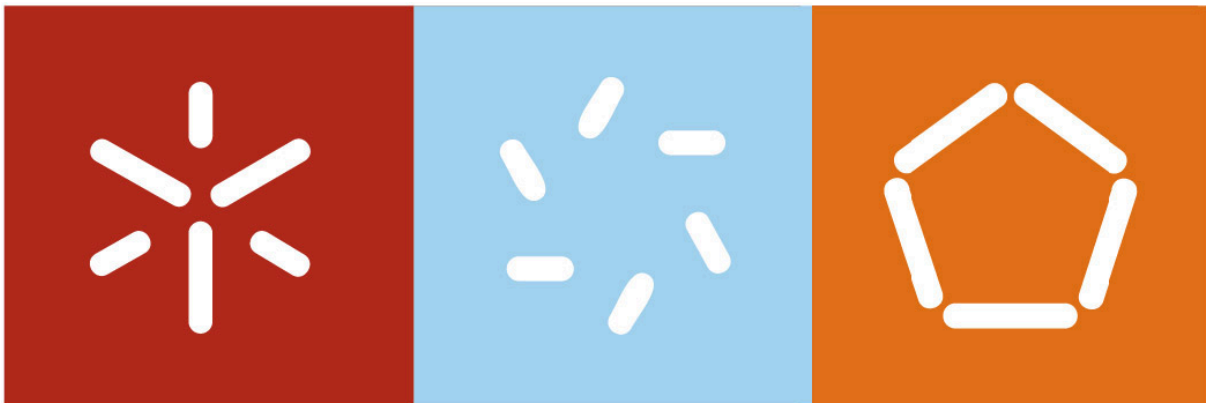


Universidade do Minho

Licenciatura em Ciências da Computação

Sistemas Operativos
Trabalho Prático - Orquestrador de Tarefas
Grupo 15

Simão Martins - A102877
Renato Garcia - A101987
Vitor Pereira - A102515

Índice

1 Introdução

Este relatório tem como objetivo descrever o desenvolvimento do trabalho prático da UC “Sistemas Operativos”.

Este projeto tinha o propósito de desenvolver um serviço de orquestração de tarefas tendo por base o sistema operativo Linux e o uso da Linguagem de Programação C. O serviço facilita a execução e o escalonamento de tarefas definidas por um utilizador através de uma arquitetura cliente-servidor, onde o utilizador, através do programa Client envia solicitações de tarefas ao programa Server, que orquestra o escalonamento e a execução dessas tarefas.

O servidor utiliza diferentes políticas de escalonamento que se baseiam em indicadores como a duração da tarefa para otimizar a utilização de recursos e minimizar o tempo de espera do utilizador.

O serviço também apresenta funcionalidades como o encadeamento de programas e multi-processamento, o que amplia a aplicação do serviço, tornando-o capaz de lidar eficientemente com um maior volume de tarefas em simultâneo.

2 Estrutura do Projeto

2.1 Servidor e Cliente

Foi desenvolvido um programa cliente que permite o envio de um pedido de execução de tarefas ao servidor, com o uso da opção execute. Além disso, tal programa também oferece a opção status que lista a situação atual das tarefas em execução, das tarefas terminadas e das tarefas que ainda serão executadas.

Foi também desenvolvido um servidor responsável por escalonar e executar as tarefas dos utilizadores. Este comunica com o programa cliente através de pipes com nome (FIFOs).

2.2 Funcionalidades básicas

O serviço de orquestração de tarefas desenvolvido oferece uma variedade de opções para gerir e monitorizar a execução de tarefas eficientemente.

2.2.1 Pedidos de execução

Os utilizadores podem enviar pedidos de execução de tarefas para o servidor, especificando os detalhes de tal tarefa como a duração estimada, o programa a executar e, possivelmente, argumentos para o input. O serviço suporta execuções individuais de programas e também execução de pipelines, permitindo ao utilizador definir o fluxo da execução.

2.2.2 Escalonamento de tarefas e Execução

Ao receber pedidos de execução, o servidor aplica uma política de escalonamento que prioriza a execução das tarefas de forma específica. A política escolhida, independentemente de ser o FCFS, o SJF ou o LJF, tem como objetivo otimizar a execução das tarefas.

2.2.3 Monitorização de Execução de tarefas

O utilizador pode, através do programa cliente, monitorizar o “status” dos seus pedidos, em tempo real. O serviço oferece informações das tarefas em execução, tarefas em espera para executar e também tarefas já executadas.

2.2.4 Resultados e output

O serviço, através do servidor, redireciona informações produzidas pelas tarefas para um ficheiro próprio cujo nome corresponde ao identificador da tarefa (task ID) com recurso a dups.

2.2.5 Terminação do programa

O utilizador pode terminar a sua interação com o serviço de orquestração de tarefas quando os seus pedidos de execução concluem ou quando já não necessitam mais do serviço. A opção quit do programa cliente é o que permite o utilizador terminar as suas interações com a interface cliente, e o uso de tal opção é comunicada ao servidor para que tal também seja terminado.

3 Arquitetura de processos

3.1 Estrutura

No servidor, as tarefas são colocadas na struct Program.

```
typedef struct Program {
    int taskid;
    char mode[2][30];
    char command[500];
    pid_t pid;
    int tempo_exec;
} Progam;
```

Essa struct contém informações relevantes sobre a tarefa a ser executada, guardada nas variáveis que podem ser vistas acima.

Além disso, essas tarefas contidas na struct Program, são guardadas na lista ligada definida pela struct FCFS_Task, o que permite que as tarefas sejam processadas na ordem que são recebidas.

```
typedef struct FCFS_Task {
    Progam task;
    struct FCFS_Task *next;
} FCFS_Task;
```

Para tarefas já executadas, a sua representação é feita através da struct Finished_task contendo informações relevantes para monitorização de execuções através da opção status no programa cliente e também para o output dos resultados.

```
typedef struct Finished_task {
    int taskid;
    char command[300];
    long tempo_exec;
} Finished_task;
```

3.2 Gestão de tarefas

Quando um pedido de execução é efetuado pelo programa cliente, o servidor insere a nova tarefa na lista ligada FCFS_Task, com base na política de escalonamento selecionada (FCFS, SJF ou LJF). Portanto, temos a definição das seguintes funções no código do servidor:

```
void enqueue(FCFS_Task **queue, Progam task)
void enqueue_sjf(FCFS_Task **queue, Progam task)
void enqueue_ljf(FCFS_Task **queue, Progam task)
```

Além disso, o servidor mantém a sua disponibilidade ao cliente através do while loop

```
while (read(fd, args, sizeof(Progam)) > 0)
```

Neste loop, o servidor lê continuamente pedidos de execução efetuados pelo programa cliente e processa-os como dito anteriormente, garantindo que o mesmo se mantenha disponível para receber e lidar com novos pedidos.

3.3 Execução e remoção da queue

À medida que as tarefas são executadas, as tarefas concluídas são retiradas da lista ligada (dequeue) através da seguinte função. Tal permite que subsequentes tarefas possam ser executadas de acordo com a política escolhida.

```
Progam dequeue(FCFS_Task **queue)
```

4 Mecanismos de comunicação

Para facilitar a comunicação entre cliente e servidor, o serviço utiliza pipes com nome, que fornecem um canal de comunicação bidirecional entre processos. Os FIFOs garantem uma troca de dados assíncrona e persistente, permitindo que o servidor envie atualizações de status das tarefas de volta ao cliente e também permitem o acesso por diferentes processos mesmo após o término dos processos originais.

4.1 Implementação dos FIFOs

No servidor, uma pipe com nome é criada com o uso da função mkfifo().

```
mkfifo(SERVER, 0666);
```

No cliente, o pipe com nome é aberto para inserção de dados (write), enquanto o servidor o abre para a leitura (read). Isso permite um fluxo unidirecional de dados do cliente para o servidor. Por exemplo, no código do cliente temos:

```
int fd = open(fifopath, O_WRONLY);
```

A transmissão dos dados é feita através das funções write() e read().

```
write(fd, args, sizeof(Progam));
```

4.2 Vantagens dos FIFOs

4.2.1 Simplicidade e facilidade de uso

A simplicidade de criar pipes com nome no servidor e abri-las no cliente faz com que a sua implementação seja mais fácil de entender, reduzindo a complexidade do código.

4.2.2 Assíncrona

Os FIFOs permitem que o cliente e o servidor troquem dados de maneira assíncrona, o que permite o envio de pedidos de execução concorrentes por múltiplos clientes.

4.2.3 Integridade de dados

As pipes com nome operam na base first-in, first-out (FIFO), garantindo que os dados são transmitidos e recebidos na ordem em que foram inseridos (write). Este comportamento dos FIFOs mantém a integridade dos pedidos de execução, prevenindo corrupção de dados.

5. Avaliação das Políticas de Escalonamento

5.1 Políticas de escalonamento implementadas

5.1.1 FCFS

As tarefas são executadas na ordem em que são enviadas ao servidor, sem considerar o seu tempo de execução.

5.1.2 SJF

As tarefas são executadas tendo como base o tempo de execução, priorizando tarefas com tempos mais curtos antes das mais longas.

5.1.3 LJF

Tal como o SJF, as tarefas são executadas com base no tempo de execução, porém, priorizando tarefas com tempos mais longos antes das mais curtas.

5.2 Avaliação da eficiência das políticas através de testes

Para a avaliação da eficiência das diferentes políticas de escalonamento foram efetuados três testes. Estes testes têm como objetivo avaliar o desempenho e a eficácia das políticas FCFS, SJF e LJF.

5.2.1 Primeiro teste

Neste primeiro teste, o servidor foi sempre executado com 3 tarefas a serem executadas simultaneamente. Os seguintes pedidos foram feitos para cada política de escalonamento (FCFS, SJF e LJF) através da opção execute no programa cliente.

```
./client execute 1000 -p "ps aux | grep firefox | echo 'Ola Mundo' | ls -a | wc -l | date | ../../Progs_exemplo/hello 1"
./client execute 3000 -p "ls -R /usr | grep 'bin' | wc -l | ../../Progs_exemplo/void 3"
./client execute 5000 -p "../../Progs_exemplo/hello 5 | wc -l"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 5000 -p "../../Progs_exemplo/void 5| ../../Progs_exemplo/hello 5"
```

Após a execução destes pedidos com as diferentes políticas de escalonamento e após a análise dos diferentes outputs, podemos chegar à conclusão que no cenário dado, onde tarefas de diferentes comprimentos e complexidades precisam de ser executadas de forma eficiente, a política que parece ser mais adequada é a SJF. A SJF prioriza tarefas mais curtas, maximizando o uso do CPU, permitindo que esta seja usada de maneira mais eficiente. No entanto, visto que temos poucas tarefas a serem executadas, podemos observar pelos tempos que qualquer dos algoritmos podia ser escolhido e ser adequado.

Tempo total SJF: 27732 ms (FCFS: 27780 ms , LJF: 27770 ms)

```
1 ps aux | grep firefox | echo 'Ola Mundo' | ls -a | wc -l | date | ../../Progs_exemplo/hello 1 1147 ms
5 ../../Progs_exemplo/hello 1 1242 ms
2 ls -R /usr | grep 'bin' | wc -l | ../../Progs_exemplo/void 3 3002 ms
3 ../../Progs_exemplo/hello 5 | wc -l 6109 ms
6 ../../Progs_exemplo/void 5| ../../Progs_exemplo/hello 5 6231 ms
4 ../../Progs_exemplo/void 10 10001 ms
```

(output com a política SJF)

5.2.2 Segundo teste

Neste segundo teste, o servidor foi sempre executado com 5 tarefas a serem executadas simultaneamente. Os seguintes pedidos foram feitos para cada política de escalonamento (FCFS, SJF e LJF) através da opção execute no programa cliente.

```
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
```

Neste caso, podemos chegar a uma conclusão diferente do teste anterior, uma vez que com uma combinação de tarefas de curta e longa execução, juntamente com uma maior frequência de execuções, faz com que uma política de escalonamento como o SJF não seja a mais adequada. Neste caso, a SJF tem como objetivo maximizar o throughput executando tarefas mais curtas primeiro. No entanto, se tarefas longas são requisitadas constantemente, a SJF não consegue maximizar o throughput.

Portanto, assim sendo, a escolha de uma política como o FCFS seria mais eficiente, neste caso.

Tempo total FCFS: 87360 ms. (SJF: 89809 ms , LJF: 90141 ms)

```
2 ../../Progs_exemplo/hello 5 5863 ms
4 ../../Progs_exemplo/hello 5 5897 ms
1 ../../Progs_exemplo/void 10 10002 ms
3 ../../Progs_exemplo/void 10 10001 ms
5 ../../Progs_exemplo/void 10 10001 ms
6 ../../Progs_exemplo/hello 5 5534 ms
11 ../../Progs_exemplo/void 1 1002 ms
12 ../../Progs_exemplo/hello 1 1098 ms
13 ../../Progs_exemplo/void 1 1002 ms
8 ../../Progs_exemplo/hello 5 5606 ms
10 ../../Progs_exemplo/hello 5 5659 ms
14 ../../Progs_exemplo/hello 1 1218 ms
7 ../../Progs_exemplo/void 10 10001 ms
15 ../../Progs_exemplo/void 1 1001 ms
17 ../../Progs_exemplo/void 1 1001 ms
16 ../../Progs_exemplo/hello 1 1246 ms
18 ../../Progs_exemplo/hello 1 1226 ms
9 ../../Progs_exemplo/void 10 10002 ms
```

(output com a política FCFS)

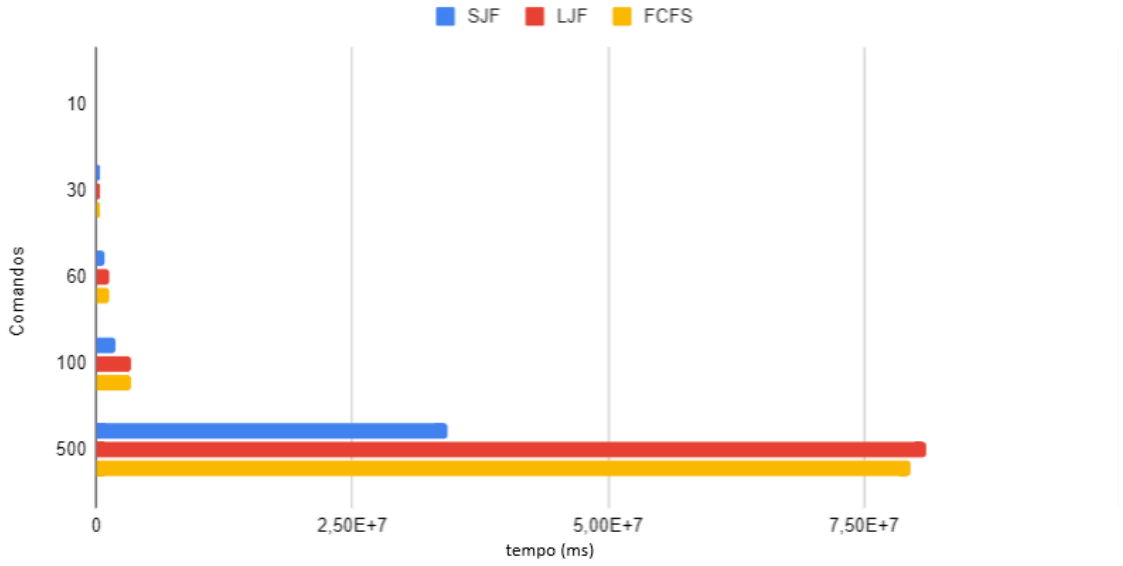
5.2.3 Terceiro teste

```
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 10000 -u "../../Progs_exemplo/void 10"
./client execute 5000 -u "../../Progs_exemplo/hello 5"
./client execute 5000 -p "../../Progs_exemplo/hello 5 | wc -l"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 5000 -p "../../Progs_exemplo/hello 5 | wc -l"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 5000 -p "../../Progs_exemplo/hello 5 | wc -l"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
./client execute 1000 -u "../../Progs_exemplo/void 1"
./client execute 1000 -u "../../Progs_exemplo/hello 1"
```

Baseado no anterior script, presente também no github, fizemos uma nova experiência. Visto que os comandos são idênticos ao do teste dois, fomos ver se a tendência verificada no segundo teste se manteve.

Testamos os diferentes algoritmos para testes com 10, 30, 60, 100 e 500 comandos. O que acontece é que podemos verificar que quantos mais comandos, menos o algoritmo SJF se mostra adequado. O que podemos constatar é que quando chegamos aos 500 comandos, o LJF acaba até por ser mais indicado do que o FCFS. Para gerar os 500 comandos, usamos um código presente nos anexos.

Quantos mais comandos temos e havendo mistura de processos curtos e longos, o LJF executa os mais longos primeiros e vai libertar recursos mais cedo para os mais curtos, que vai reduzir o tempo de espera geral. Temos também de ter em atenção sempre a ordem dos nossos inputs, sendo essa uma das razões possíveis para que o FCFS e o LJF estejam relativamente perto. Abaixo temos a evolução do tempo.

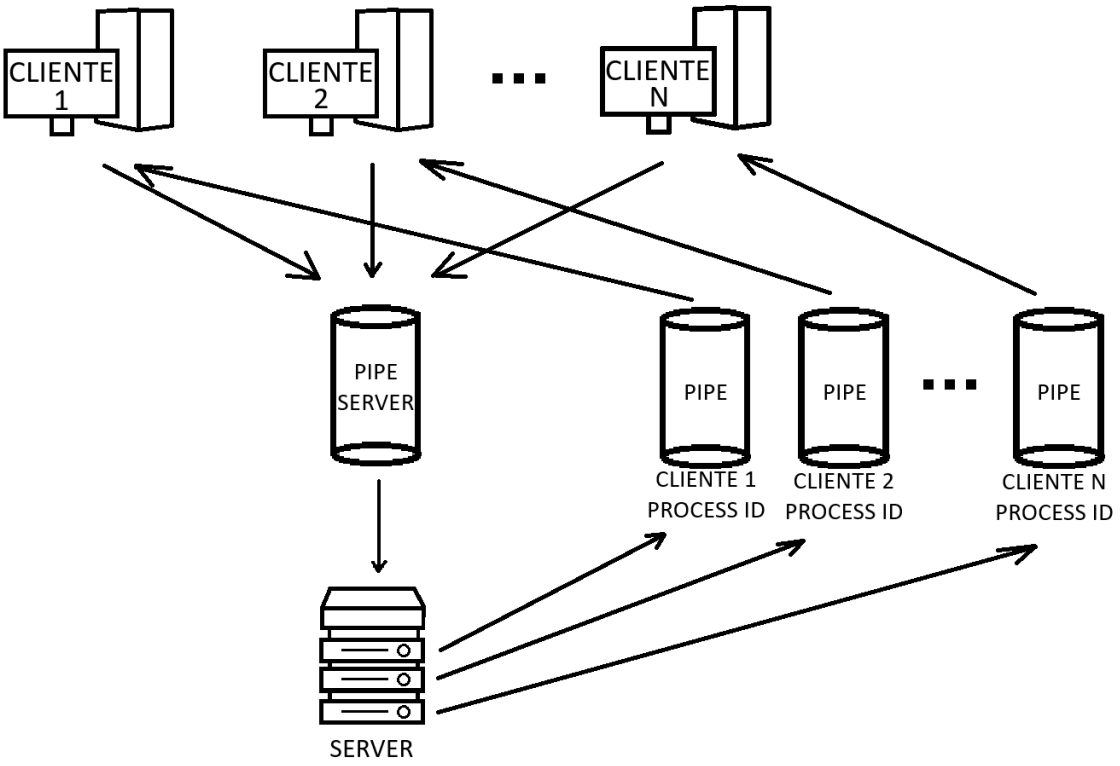


Tempo das diferentes políticas de escalonamento.

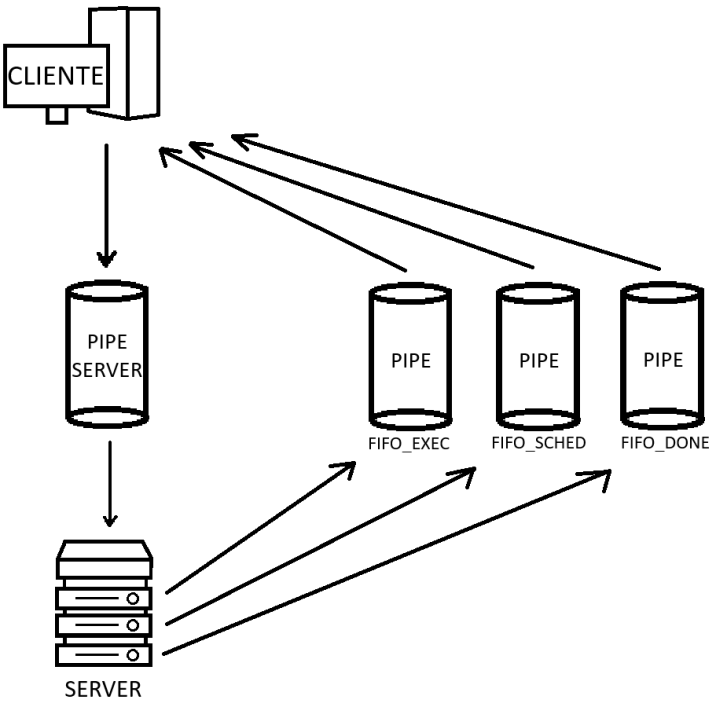
6 Conclusão

Em conclusão, tendo em conta os objetivos do projeto, achamos que todos os objetivos foram alcançados com sucesso. Enfrentando alguns desafios ao longo do processo, pensamos que os conseguimos superar sempre de forma cuidadosa e analítica. Conseguimos desenvolver e implementar com sucesso um sistema capaz de gerir tarefas de forma eficiente, dando um melhor entendimento no uso de FIFOs, na implementação de políticas de escalonamento e na estruturação de processos. Os resultados obtidos mostram não apenas a nossa compreensão dos conteúdos da Unidade Curricular, mas também a nossa capacidade de lidar com desafios complexos de maneira sistemática e criteriosa.

Anexos



Comunicação cliente-servidor



Comunicação cliente-servidor no pedido da opção status

```
execute_task(){
    local task="$1"
    local tempo=$((task * 1000))
    echo "./client execute $tempo -u "../../Progs_exemplo/hello $task"" >> scripts.txt
    echo "./client execute $tempo -u "../../Progs_exemplo/void $task"" >> scripts.txt
}

if [ "$#" -ne 1 ]; then
    echo "Erro nos argumentos"
    exit 1
fi

quantas=$1
max_time=10

for ((i=1;i<= quantas; i++)); do
    random_time=$((RANDOM % max_time + 1))

    execute_task "$random_time"
done
```

Script em bash utilizado para gerar os comandos para o terceiro teste