

Logotipo, nome da empresa

Descrição gerada automaticamente

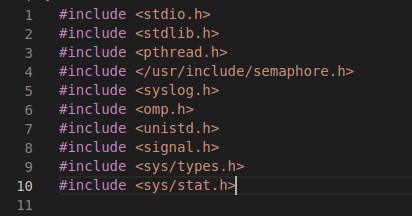
1. Introdução

Neste documento será descrito o funcionamento do código na versão com “extra”, ou seja, com adições e melhorias sugeridas, bloco a bloco, do algoritmo de produtor-consumidor realizado como elemento de composição da nota final da disciplina de Sistema de Computação.

Os extras realizados foram as questões 1,3,4 e 5.

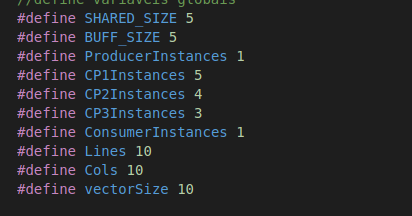
1. Bibliotecas e Declarações

Inicia-se o código com a inclusão das bibliotecas necessárias:



* *stdio.h,* sendo o cabeçalho padrão de entrada e saída;
* *stdlib.h,* sendo outra biblioteca padrãoenvolvendo alocação de memória, controle de processos, conversões e outras;
* *pthread.h,* sendo utilizada para criar e manipular threads e;
* */usr/include/semaphore.h,* para manipular os semáforos que permitem a utilização de memória compartilhada;
* syslog é a biblioteca necessária para podermos trabalhar com os logs;
* omp.h é a biblioteca que usamos para trabalhar com o open mp;
* as demais foram adicionadas para conseguirmos trabalhar com o daemon.

Em seguida, declara-se as variáveis globais:



Sendo:

* SHARED\_SIZE define o tamanho da memória compartilhada;
* BUFF\_SIZE define o tamanho do buffer;
* ProducerInstances define a quantidade de threads de Produtores;
* CP1Instances define a quantidade de threads de consumidores/produtores do primeiro processo;
* CP2Instances define a quantidade de threads de consumidores/produtores do segundo processo;
* CP3Instances define a quantidade de threads de consumidores/produtores do terceiro processo;
* ConsumerInstances define a quantidade de threads de consumidores;
* Lines define a quantidade de linhas da matriz;
* Cols define a quantidade de colunas da matriz;
* vectorSize define o tamanho do vetor que será utilizado para armazenar o resultado.

Continuando com as declarações, define-se as *structs* (S) conforme imagem a seguir:

Texto

Descrição gerada automaticamente

A primeira estrutura é definida por S e contém as seguintes propriedades:

* Caractere Name com até 100 elementos;
* Três matrizes (A, B e C) com o número de linhas e colunas conforme definidas no bloco anterior;
* um vetor V que cria um vetor com o tamanho especificado no bloco anterior;
* E, por fim, a variável E que armazenará o resultado final desta estrutura.

Ainda criando estruturas, cria-se outra *struct* chamada *shared\_t* que controlará o *buffer* da seguinte forma:

* A variável S receberá o ponteiro do buffer, com tamanho definido no bloco anterior;
* Utiliza dois contadores inteiros (in e out) para controlar o número de elementos que entram e saem da região de buffer;
* Cria-se duas variáveis de semáforo para indicar se a região está cheia ou vazia. Uma terceira, chamada mutex é responsável por gerenciar as outras duas variáveis, fazendo o “lock” quando necessário.

Após finalizar a etapa de declarações e definições, o código concentra todas as chamadas de função de matriz, arquivos e array em um único bloco. Estas funções serão descritas com maiores detalhes adiante.

Texto

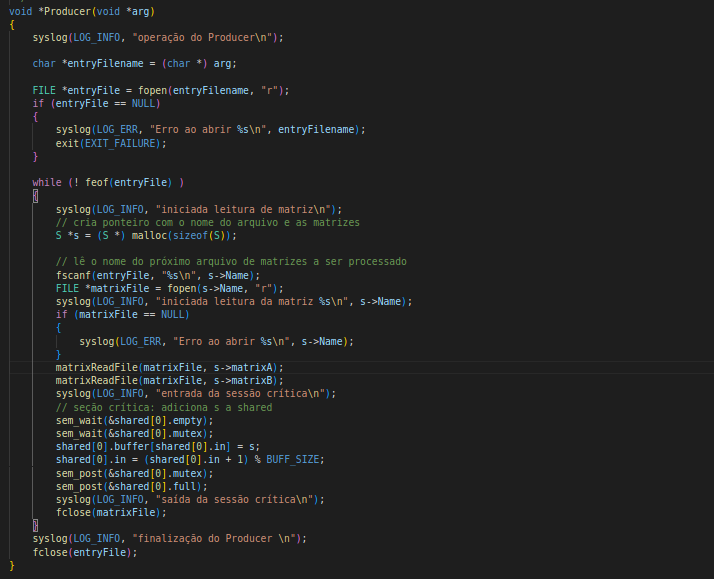
Descrição gerada automaticamente

1. Agentes do Código

Nesta etapa do código, entra, talvez, a parte mais importante, onde cria-se as funções de produtor, consumidor/produtor e consumidor. Como existem blocos com linhas de código iguais, após a primeira explicação será ressaltado apenas as mudanças de um bloco para o outro.

* 1. Produtor

Começando com o Produtor, como o nome sugere, é a parte do código que produz conteúdo – leia-se, dados - e insere na memória. Sendo assim, temos:



Uma função *void* que trabalha com ponteiros inicia com o comando *printf* informando que o processo foi inicializado. Em seguida, atribui-se ao ponteiro de dado de entrada o arquivo “entrada.in” através do comando *fopen* com permissão para leitura. Para impedir erros de leitura, adiciona-se um *printf* para sinalizar o erro em caso de arquivo em branco.

Para a leitura completa dos dados, utiliza-se o comando w*hile* com a função *feof*, que verifica se o indicador de fim do arquivo do fluxo retorna um valor diferente de zero. Em seguida, cria-se um ponteiro para cada linha de leitura, de forma a alocar na memória os dados correspondentes.

Como podemos ver, já foi adicionado aqui a lógica de não ter limite de arquivos escritos, pois o feof irá ler enquanto houver o que ler no arquivo.

Continuando, o *fscanf* aponta para a próxima matriz a ser lida e informa caso haja erro ao encontrar os dados. Como o arquivo de leitura incorpora duas matrizes, a linha de *matrixReadFile* aponta para as duas matrizes, chamadas de MatrixA e MatrixB.

Ao acessar a seção crítica, foi utilizado a instrução *sem\_wait* para coordenar o semáforo, sinalizando a “parada” de instruções para que haja a escrita e, posteriormente, a “abertura” para prosseguimento das demais instruções. Isso é feito através da adição de s – o ponteiro criado anteriormente – dentro da região *shared*, iniciando o semáforo com *sem\_wait* alterando os valores de *empty* (informa se está vazio) e do *mutex* (coordena o semáforo, abrindo e fechando as ações de escrita/leitura).

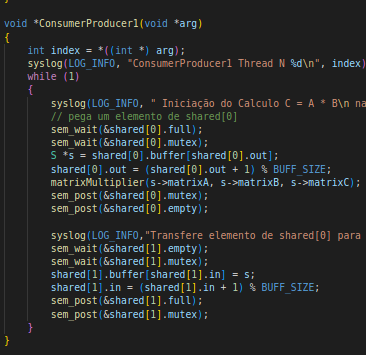
De forma um pouco mais explicativa, o sem\_wait vai validar se a região crítica está ou não vazia, usando o “empty”, conseguindo entrar, ele irá chamar o mutex, fechando o sinal e utilizando a região. Após o final de seu processamento, no caso, alocando S na região compartilhada, ele chama o sem\_post, liberando o mutex, permitindo que outro processo atue na região crítica.

Por fim, o comando *fclose* fecha os arquivos.

Vale ressaltar que ja podemos ver aqui os nossos primeiros retornos em log, para maior controle da aplicação, informando o passo do programa e os casos de erro.

* 1. Consumidor/Produtor

Nesta etapa do processo tem-se o agente Consumidor Produtor (CP) responsável por consumir a informação do agente anterior - Produtor ou Consumidor/Produtor anterior - e, em seguida, produzir uma nova informação. Esta parte do código é dada por:

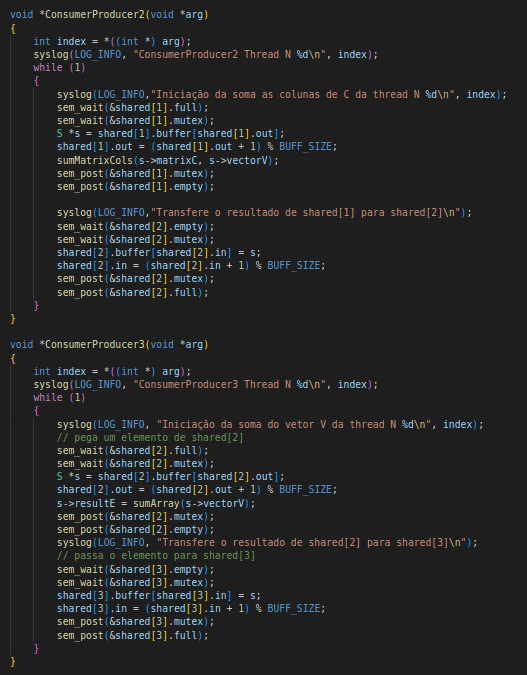


O código inicia-se com um print na tela informando o número da *Thread* a ser executada. A seguir, o código executa a multiplicação de matrizes sob a mesma lógica do Produtor, parando o semáforo com os comandos *sem\_wait* e começando com o “*full*” para verificar se existe outro processo utilizando a região crítica. Caso haja algum processo utilizando a região crítica, o algoritmo espera até que o sinal seja atualizado permitindo o acesso.

Continuando, o algoritmo cria um ponteiro s que coleta o dado de *shared[0],* realiza a operação de multiplicação de matrizes através da chamada de função *matrixMultiplier*, multiplicando a matrixA pela matrixB e armazenando o resultado na matrixC. Em seguida, libera-se o semáforo após a operação com o comando *sem\_post*.

A parte final do código é responsável por transferir (produzir) o elemento de *shared[0]* para o *shared[1]* utilizando-se do mesmo mecanismo explicado nos processos anteriores para controle de semáforo com o *sem\_wait* e *sem\_post*.

A partir daqui os códigos do segundo e terceiro Consumidor/Produtor possuem praticamente a mesma arquitetura, alterando-se somente a ação a ser tomada quando há o acesso na região crítica, conforme ilustração a seguir:

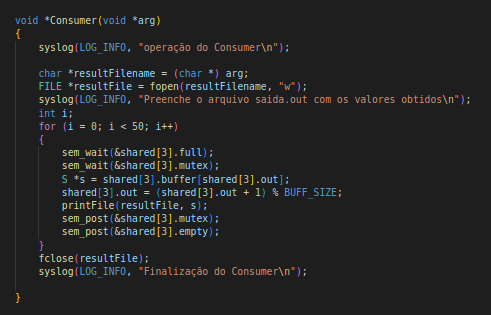


O segundo Consumidor/Produtor acessa a região crítica quando há permissão, coleta o resultado de *shared [1],* realiza o somatório das colunas das matrizes e armazena no *vectorV* através da função *SumMatrixCols*. Enquanto a segunda chamada de *sem\_wait* marca o início da parte produtora ao acessar a região crítica e armazenar os valores em *shared[2].*

O terceiro Consumidor/Produtor acessa a região critica quando há permissão, realiza o somatório de *array vectorV* através da chamada de função *sumArray*. Após o consumo e processamento dos dados, o código chama novamente a função *sem\_wait* e inicia a parcela produtora ao armazenar o resultado de *shared[2]* em *shared[3].*

* 1. Consumidor

Este agente é responsável por “consumir” ou processar os dados da região critica. Diferentemente dos demais agentes, este não escreve dados na memória.



Esta parte do código é responsável por escrever os resultados no arquivo saida.out utilizando a função *fopen* com permissão para escrita. Para isso, o código espera as demais etapas do processo serem concluídas através da variável *index*, que será utilizada no laço *While* como parâmetro lógico.

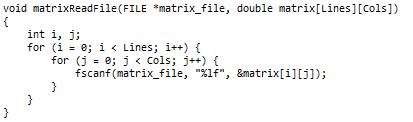
Caso os processos já tenham finalizado, o algoritmo espera a abertura do semáforo com o *sem\_wait*, cria um ponteiro com os valores de *shared[3]* e insere no arquivo com o comando *printfile*. Adicionalmente, incrementa um valor na variável *index* para continuar o looping de escrita. Por fim, o comando *sem\_post* libera o semáforo para os outros processos.

1. Funções

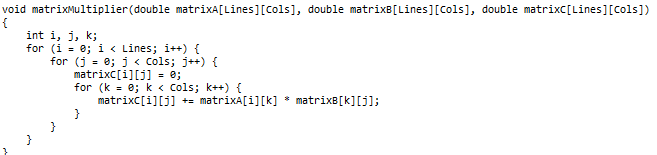
Nesta seção será descrito o funcionamento das funções utilizadas nos algoritmos de produtor, produtor/consumidor e consumidor.

4.2. Matrizes

A primeira função redigida no código é a *MatrixReadFile*.

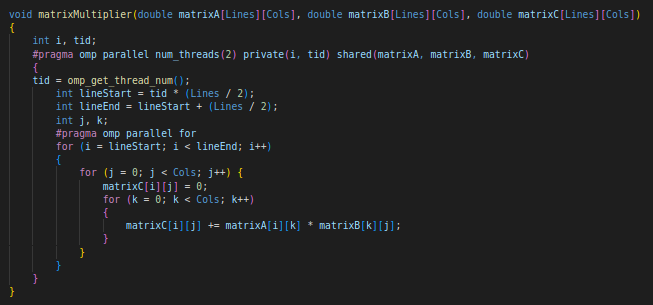


O objetivo desta função é receber como dado de entrada o ponteiro do arquivo da matriz e as informações de linhas e coluna e, a partir disso, realizar um loop em todas as linhas e colunas aplicando o comando *fscanf* para ler cada um dos elementos da matriz que foi apontada.



Sob a mesma lógica da função anterior de “varrer” a matriz utilizando looping, a função *matrixMultiplier* recebe os parâmetros (linhas e colunas) das 3 matrizes, A, B e C e escreve em C a multiplicação dos elementos de A e B.

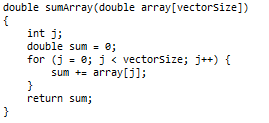
Conforme solicitado nas questões extras, a matriz faz a multiplicação utilizando duas threads usando o open mp.



Ainda com a utilização de looping, a função *printMatrix* possui um único parâmetro de entrada, que se refere à matriz que será exibida ao usuário, utilizando o laço criado pelo comando for para printar cada um dos elementos.

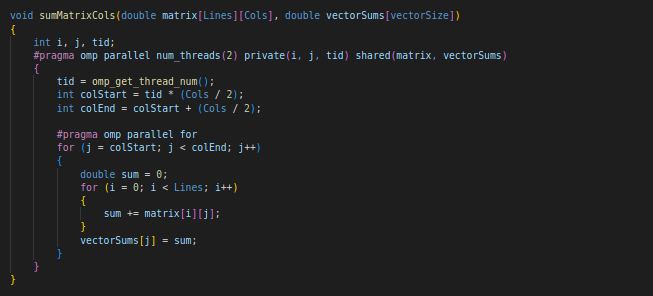
4.2. Soma

A primeira função de soma é a *sumArray*:



Esta função exige como parâmetro de entrada um *array*, onde será realizado a soma de cada um dos elementos retornando este único valor na variável *sum*.

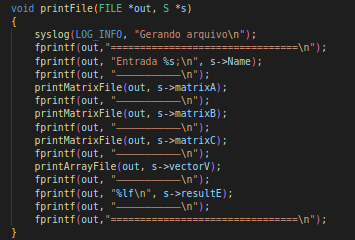
Conforme solicitado, a função utiliza duas threads e open mp para fazer sua operação.



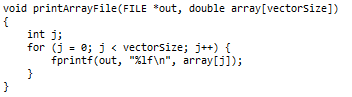
Com funcionamento parecido, a função *sumMatrixCols* soma os elementos das colunas das matrizes, retornando um *array* com os respectivos valores.

4.3. Print

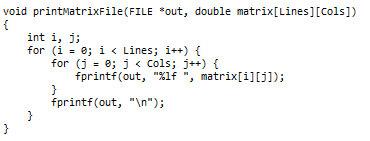
A função *printFile* possui dois argumentos de entrada, sendo o ponteiro \**out* e \**s*.



O objetivo desta função é printar os elementos processados anteriormente de todas as matrizes e variáveis.



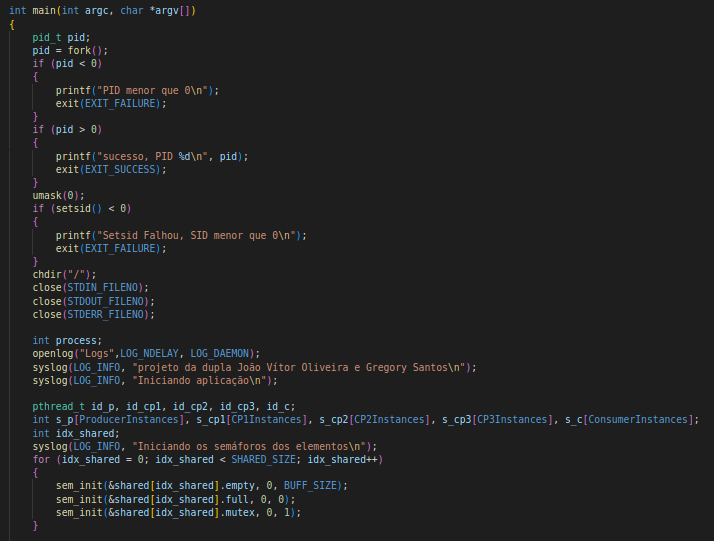
A função *printArrayFile* utiliza a função for para printar cada elemento do *Array* de entrada.



Com a mesma lógica anterior, a função *printMatrixFile* printa todos os elementos da matriz utilizando dois comandos de for para varrer cada linha e coluna.

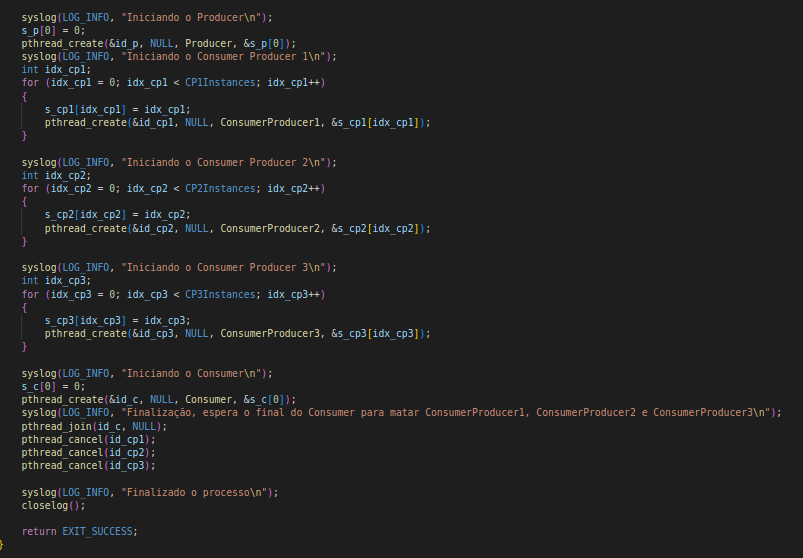
1. Main

A parte do código que conectará tudo o que foi dito até aqui é realizado sob a função **main**.



A função é iniciada com transformando o proesso em daemon, fazendo fork() e validando o pid. Um comando de *print*, que informa o início do algoritmo, seguido da declaração das *threads* com o comando *pthread\_t*. Além disso, declara-se as varáveis dos processos (P, CP e C). Em seguida, há a inicialização dos parâmetros iniciais dos semáforos, como podemos ver.

O comando *sem\_init* é responsável por inicializar um semáforo com o endereço apontado pela região *shared*, atribuindo no último argumento o valor que será armazenado. Neste caso, o *full* inicia com o valor 0 e o *mutex* com o valor 1.



Em seguida, inicia-se uma estrutura de chamada de cada um dos agentes do processo. O primeiro é o Produtor, atribuindo à sua variável o valor zero e chamando a função *pthread\_create*.

Continuando o processo, as chamadas dos Consumidores/Produtores é feita atribuindo um valor inteiro para a variável de controle de cada um dos agentes seguido de um comando for que cria as threads controlando as instâncias de CP1, CP2 e CP3.

Por fim, é criado a *thread* do consumidor da mesma maneira, finalizando a criação de *threads*.

A última parte do código consiste num print da tela informando ao usuário o status do processo e utilizando o comando *pthread\_cancel* para forçar o término dos demais processos.

1. Inicialização

Para iniciar o projeto, é necessário apenas clicar no arquivo Projeto e, poderá ser visto que irá gerar uma saída, seguindo o padrão esperado, caso deseje adicionar alguma matriz, será necessário criar um novo arquivo conforme especificado no trabalho, cada arquivo deve conter duas matrizes 10x10 uma sob a outra.

Aqui, nesse caso, é esperado também que se execute o comando para rodar o daemon:

“gcc projeto.c -o projetoDaemon -lpthread -fopenmp”

para que o projeto daemon funcione, é necessário criar alterar o caminho do arquivo entrada.in para a entrada das matrizes visto que o daemon opera com caminhos absolutos.

Após esses ajustes, só precisamos rodar “./projetoDaemon $PWD/entrada.in $PWD/saida.out”

Opcionalmente, pode-se executar o comando abaixo para criar um executavel e executar o projeto:

“gcc -Wall -g -o projetoExec projeto.c -lpthread -fopenmp”

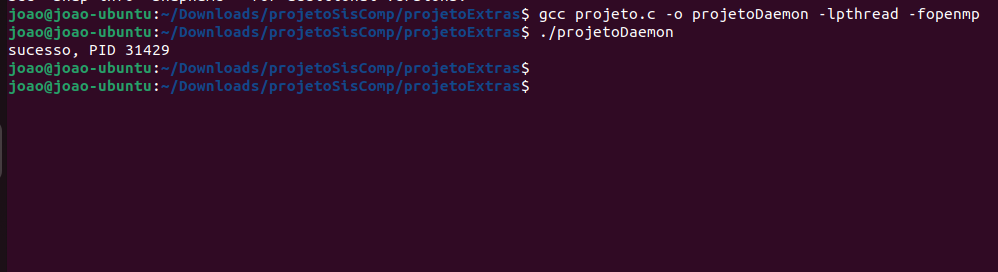
para executar executar, escreva: “./projetoExec”

Para visualizar os logs, basta digitar o comando

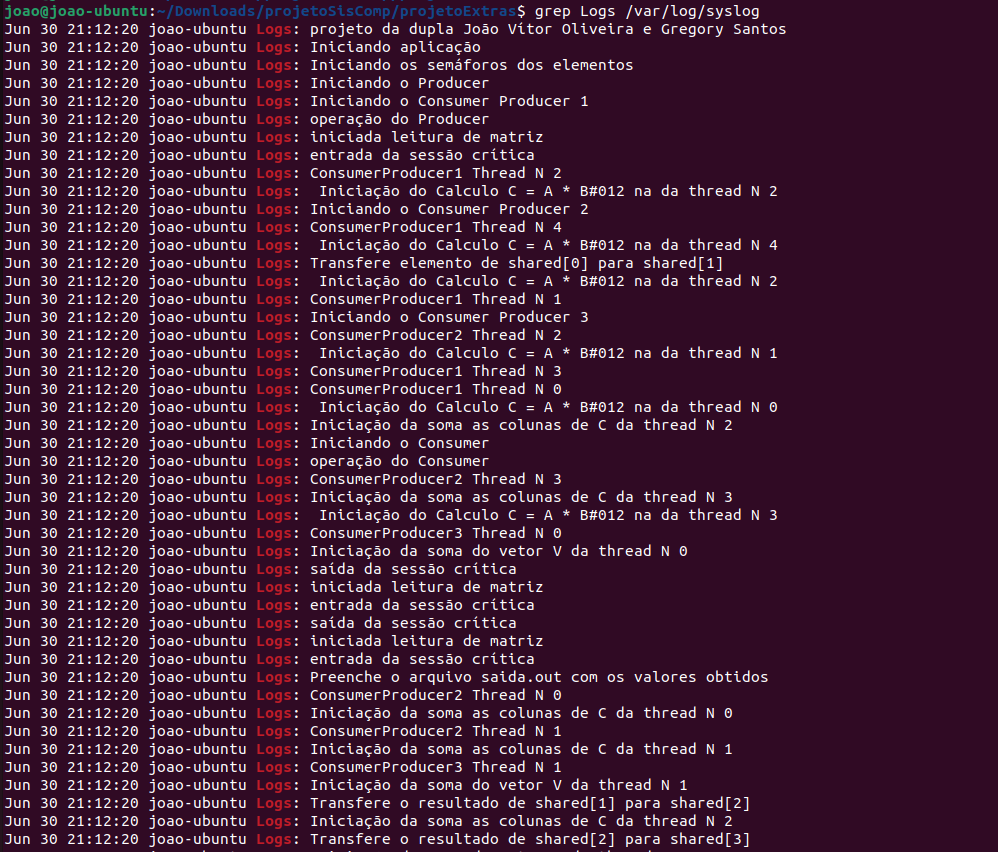
“grep Logs /var/log/syslog”

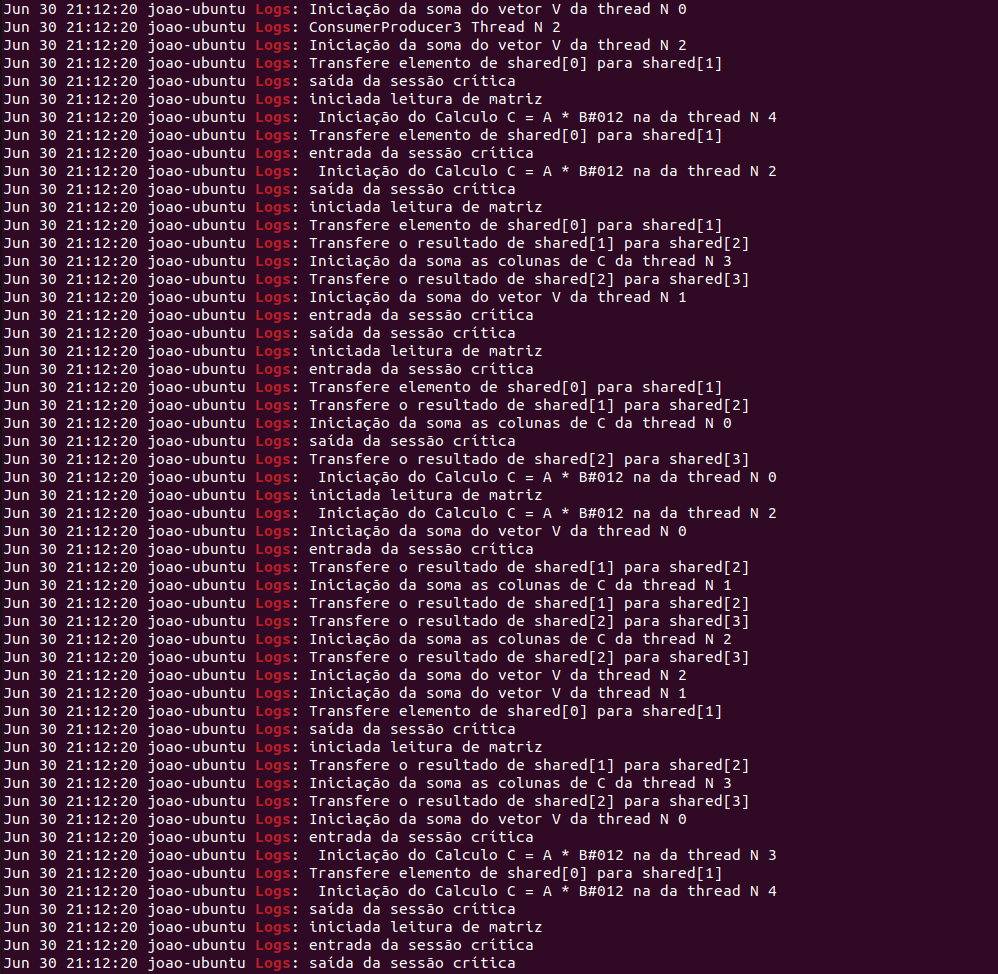
1. Imagens

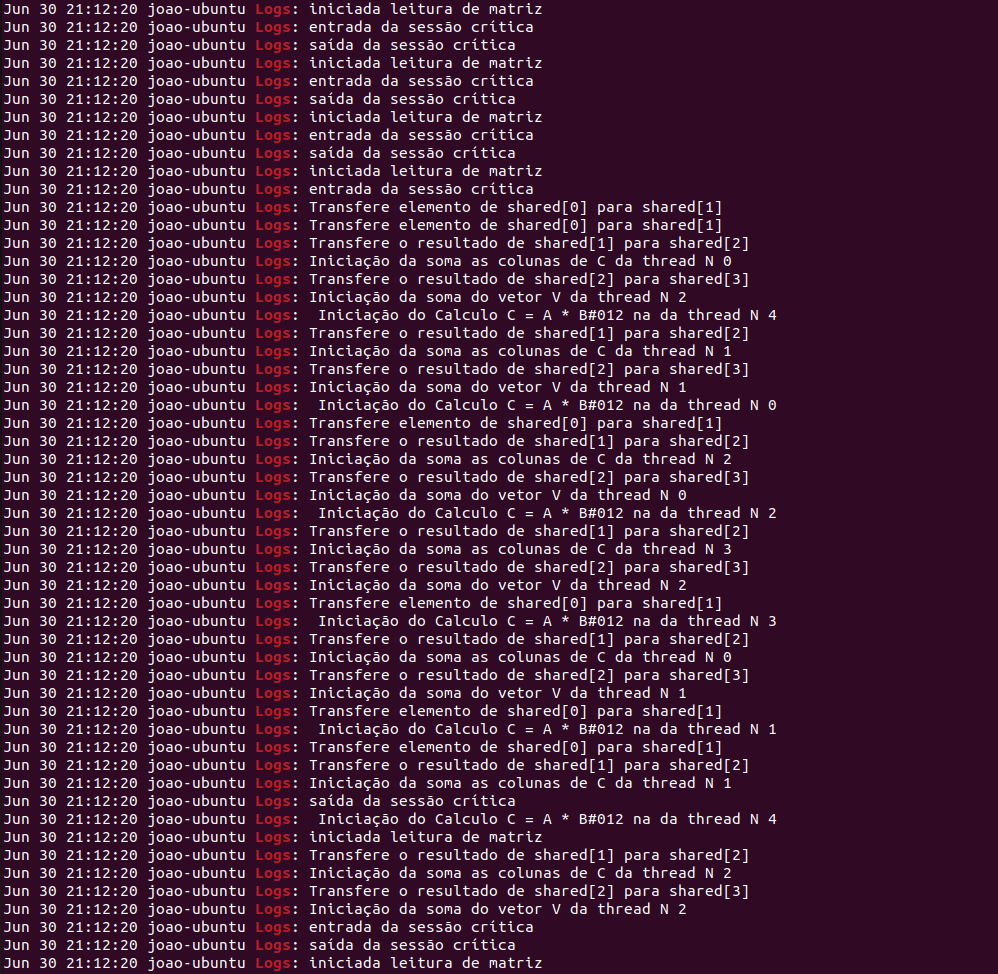
Nessa sessão seguirá prints do console quando o projeto é executado:



Abaixo temos um exemplo do comando log com algumas das mensagens retornadas:







Abaixo temos o entrada.in, seguido do saida.out, no caso focando apenas no resultado da matriz\_1 e, consecutivamente, o arquivo da matriz 1.

Optamos por colocar apenas o exemplo um para não termos diversas páginas de conteúdo repetido.

