

UNIVERSIDADE FEDERAL DO PARANÁ

DIEGO RENAN GARZARO

GRR20172364

**TRABALHO 2 - CONTROLE DE EXECUÇÃO DE THREADS NA IMPRESSÃO DE
SEQUÊNCIA REPETIDA DE CARACTERES**

LINK VÍDEO YOUTUBE: <https://youtu.be/UzSnHpQGgjs>

LINK GITHUB: <https://github.com/DiegoGarzaro/EmbeddedOperationalSystems>

CURITIBA

2020

SUMÁRIO

1 INTRODUÇÃO	3
2 DESENVOLVIMENTO	4
2.1 CONFIGURAÇÃO DA MÁQUINA VIRTUAL	4
2.2 DESENVOLVIMENTO DO CÓDIGO EM C	6
2.3 RESULTADOS	10
3 CONCLUSÃO	13
4 REFERÊNCIAS BIBLIOGRÁFICAS	14

1 INTRODUÇÃO

Esta atividade avaliativa da matéria de Sistemas Operacionais Embarcados visa consolidar os conhecimentos lecionados sobre threads e semáforos, através da implementação de um semáforo para controlar o fluxo de execução de múltiplas threads em um programa que deve imprimir na tela do terminal uma sequência de caracteres repetidos pré definidos pelo professor. Onde, utilizando as funções da biblioteca *semaphore.h*, é possível controlar a execução entre threads de forma a imprimir apenas os caracteres desejados mesmo com todas as threads sendo executadas de forma simultânea.

2 DESENVOLVIMENTO


2.1 CONFIGURAÇÃO DA MÁQUINA VIRTUAL

No desenvolvimento deste trabalho, está sendo utilizado uma máquina virtual, onde está sendo executado o Sistema Operacional Ubuntu 20.04. Na figura abaixo é possível visualizar os detalhes da Virtual Box:

	Geral	
	Nome	ubuntu
	Sistema Operacional	Ubuntu (64-bit)
	Sistema	
	Memória Principal	4096 MB
	Ordem de Boot	Óptico, Disco Rígido
	Aceleração	VT-x/AMD-V, Paginação Aninhada, Paravirtualização KVM
	Tela	
	Memória de Vídeo	16 MB
	Controladora Gráfica	VMSVGA
	Servidor de Desktop Remoto	Desabilitado
	Gravação	Desabilitado
	Armazenamento	
	Controladora: IDE	
	IDE Secundário Master	[Disco Óptico] Vazio
	Controladora: SATA	
	Porta SATA 0	ubuntu.vdi (Normal, 10,00 GB)
	Áudio	
	Driver do Hospedeiro	Windows DirectSound
	Controladora	ICH AC97
	Rede	
	Adaptador 1	Intel PRO/1000 MT Desktop (NAT)
	Portas Seriais	
	Desabilitado	
	USB	
	Controladora USB	OHCI
	Filtros de Dispositivo	0 (0 ativos)
	Pastas Compartilhadas	
	Nenhum	

Figura 1 - Detalhes do Sistema Operacional Linux na Virtual Box

Nas imagens a seguir, serão exibidos as configurações de instalação definidas no Ubuntu, dentro da máquina virtual:



Device Name	diego-VirtualBox >
Memory	3,8 GiB
Processor	Intel® Core™ i7-7700 CPU @ 3.60GHz
Graphics	llvmpipe (LLVM 10.0.0, 256 bits)
Disk Capacity	10,7 GB
OS Name	Ubuntu 20.04.1 LTS
OS Type	64-bit
GNOME Version	3.36.3
Windowing System	X11
Virtualization	Oracle
Software Updates	>

Figura 2 - Informações do Ubuntu instalado na Virtual Box

```
diego@diego-VirtualBox: ~/Desktop/SO$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:         39 bits physical, 48 bits virtual
CPU(s):                1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Stepping:              9
CPU MHz:               3599.994
BogoMIPS:              7199.98
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32 KiB
L1i cache:             32 KiB
L2 cache:              256 KiB
L3 cache:              8 MiB
NUMA node0 CPU(s):     0
Vulnerability Itlb multihit: KVM: Vulnerable
Vulnerability L1tf:      Mitigation; PTE Inversion
Vulnerability Mds:       Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
Vulnerability Meltdown:  Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2: Mitigation; Full generic retpoline, STIBP disabled, RSB filling
Vulnerability Srbds:     Unknown: Dependent on hypervisor status
Vulnerability Tsx async abort: Not affected
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mm
                        x fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good noopl xtopology nonst
                        op_tsc cpuid tsc_known_freq pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse4_2 x2
                        apic movbe popcnt aes xsave avx rdrand hypervisor lah_f_lm abm 3dnowprefetch invpcid
                        d_single pti fsgsbase avx2 invpcid rdseed clflushopt flush_l1d
```

Figura 3 - Informações de Hardware e Software: comando ***lscpu***

2.2 DESENVOLVIMENTO DO CÓDIGO EM C

Após a configuração da máquina virtual, deu-se início ao desenvolvimento do código para realizar o controle de execução das threads no programa para impressão dos caracteres no terminal, de acordo com a ordem solicitada pelo professor. O programa que realiza este controle e imprime as letras será explicado com mais detalhes nos próximos parágrafos.

A partir deste momento, será apresentado e explicado a parte do código, para que o programa funcionasse corretamente, foi necessário incluir algumas bibliotecas, entre elas estavam a *lpthread.h* e a *semaphore.h*, principais bibliotecas para que fosse possível desenvolver o código proposto. E também foram definidos algumas constantes, tal como o número de threads que iriam ser utilizadas, além da estrutura criada para melhor organização do código, e facilitar no trabalho com os dados dentro do código:

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

#define NUM_THREADS 6

typedef struct Letter_Sequence
{
    char letter;
    int quantity;
} Letter_Sequence;
```

Após isso, foram declaradas algumas variáveis globais, a primeira delas é do tipo “*sem_t*” representando o semáforo *my_sem* que será utilizado para controlar as threads no programa. Então foi declarado um vetor de estrutura *Letter_Sequence* com o nome de *sequence*, este vetor tem a principal função de armazenar a ordem dos caracteres definidos pelo professor, juntamente com a quantidade de vezes que

cada caractere deve ser impresso na tela. A variável global do tipo inteiro, apelidada de *count* que será responsável por auxiliar na varredura do vetor *sequence*. E, por fim, o vetor de caractere *thread_letter*, que armazena as letras que cada thread deve imprimir, ou seja:

Número da Thread	Letra de Impressão de cada Thread
thread 0	a
thread 1	b
thread 2	c
thread 3	d
thread 4	e
thread 5	f

Tabela 1 - Ordem das threads e seus respectivos caracteres

Assim, este bloco do código ficou da seguinte forma:

```
sem_t my_sem;
int count = 0;

Letter_Sequence sequence[6] = {{'d', 6}, {'c', 9}, {'a', 11}, {'c',
17}, {'e', 8}, {'d', 28}};
char thread_letters[NUM_THREADS] = {'a', 'b', 'c', 'd', 'e', 'f'};
```

Neste momento, será dado início a explicação da função genérica que será executada por cada thread. A função irá receber como argumento a identidade de cada thread, ou seja, a identidade nada mais é do que um número de 0 a 5, de forma a identificar a thread que estará entrando na função, pois apenas a thread correspondente a letra da sequência de caracteres poderá imprimir o caractere, de forma a assegurar a integridade do programa. Desta forma, quando uma thread começa a executar a função, é setado a flag do semáforo por meio do comando *sem_wait(&my_sem)*, de forma que o semáforo indica que já tem uma thread sendo executada, impedindo que qualquer outra thread possa executar a função

`thread_handler(void *arg)` em paralelo. Depois disso, é feita uma verificação, se a thread que está executando a função possui a letra correspondente a da variável `sequence` na posição do contador `count`, caso a comparação dos dois caracteres seja verdadeira, então, um laço `for` vai garantir que seja printado o número correto de vezes que a letra se repete, e então, incrementar a variável `count`; caso a condição seja falsa, o programa apenas irá ignorar este bloco de instruções, e executará o restante do código, que seria um `usleep`, acompanhado do comando para liberar o semáforo `sem_post(&my_sem)` e então sair da função da thread.

```
void *thread_handler(void *arg)
{
    // Hold semaphore
    int *t_letter = (int *)arg;
    // Wait
    sem_wait(&my_sem);

    // Verify if the correct thread was called
    if (sequence[count].letter == thread_letters[*t_letter])
    {
        int j;
        // Print the letter
        for (j = 0; j < sequence[count].quantity; j++)
        {
            printf("- %c -> %d\n", thread_letters[*t_letter], j+1);
        }
        count++;
    }

    usleep(1000);
    // Release semaphore
    sem_post(&my_sem);
    pthread_exit(NULL);
}
```

Agora, por fim, será explicado a função `main`. Primeiramente, é declarado um vetor de threads com seis posições, então, é printado no terminal as informações do código da disciplina, informações do trabalho, e o nome do aluno, juntamente com o GRR do mesmo. Depois disso, é realizado a inicialização do semáforo por meio do

comando `sem_init(&my_sem, 0, 0)`, e atribuído o resultado deste comando para a variável `create_sem` que vai fazer a verificação se o semáforo foi criado corretamente ou retornar um se houver algum erro durante sua inicialização.

No próximo momento, é iniciado um loop *do while*, que vai permanecer fechado até que o contador *count* percorra todo o vetor *sequence*, de forma a imprimir no terminal todas as letras em ordem. Dentro deste laço, foram utilizados dois laços *for*, um seguido do outro. O primeiro deles é responsável por criar as threads e passar como parâmetro o valor da variável auxiliar *i* que vai estar percorrendo por todos os números da quantidade de threads, começando pelo número zero, e executando o loop até chegar no número seis. Caso haja algum erro durante a criação das threads, o erro será apontado e o programa se encerrará com o retorno do valor um. Após a varredura do primeiro laço *for*, é executado o comando para liberar o semáforo `sem_post(&my_sem)`, e então é executado o segundo *for*, sendo este responsável por finalizar todas as threads criadas. Este loop irá se repetir até que todas as letras da sequência sejam impressas e repetidas de acordo com o enunciado disponibilizado pelo professor.

```
int main()
{
    pthread_t threads[NUM_THREADS];

    // Header
    printf("TE355 - Trabalho 2 - Semaphore\n");
    printf("Aluno: Diego R. Garzaro - GRR20172364\n");

    // Create semaphore
    int create_sem = sem_init(&my_sem, 0, 0);
    // Verify if semaphore was created successfully
    if (create_sem < 0)
    {
        perror("Failed to create semaphore\n");
        return 1;
    }

    // Loop
    do
    {
```

```

    int i;
    for (i = 0; i < NUM_THREADS; i++)
    {
        int *parameter;
        parameter = calloc(1, sizeof(int *));
        *parameter = i;
        int rc = pthread_create(&threads[i], NULL,
thread_handler, parameter);
        if (rc)
        {
            perror("Failed to create thread\n");
            return 1;
        }
    }

    sem_post(&my_sem);

    for (i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }
} while (count < 6);

return 0;
}

```

2.3 RESULTADOS

Na imagem abaixo será possível visualizar o correto funcionamento do programa de acordo com a proposta solicitada pelo professor. Na imagem abaixo, será mostrado o código funcionando com algumas alterações, será impresso ao lado de cada caractere o número de repetições, de forma a auxiliar na contagem de cada letra no momento de verificar o funcionamento do programa:

```

TE355 - Trabalho 2 - Semaphore
Aluno: Diego R. Garzaro - GRR20172364

- d -> 1
- d -> 2
- d -> 3
- d -> 4
- d -> 5
- d -> 6
- c -> 1
- c -> 2
- c -> 3
- c -> 4
- c -> 5
- c -> 6
- c -> 7
- c -> 8
- c -> 9
- a -> 1
- a -> 2
- a -> 3
- a -> 4
- a -> 5
- a -> 6
- a -> 7
- a -> 8
- a -> 9
- a -> 10
- a -> 11
- c -> 1
- c -> 2
- c -> 3
- c -> 4
- c -> 5
- c -> 6
- c -> 7
- c -> 8
- c -> 9
- c -> 10

- c -> 10
- c -> 11
- c -> 12
- c -> 13
- c -> 14
- c -> 15
- c -> 16
- c -> 17
- e -> 1
- e -> 2
- e -> 3
- e -> 4
- e -> 5
- e -> 6
- e -> 7
- e -> 8
- d -> 1
- d -> 2
- d -> 3
- d -> 4
- d -> 5
- d -> 6
- d -> 7
- d -> 8
- d -> 9
- d -> 10
- d -> 11
- d -> 12
- d -> 13
- d -> 14
- d -> 15
- d -> 16
- d -> 17
- d -> 18
- d -> 19
- d -> 20
- d -> 21
- d -> 22
- d -> 23
- d -> 24
- d -> 25
- d -> 26
- d -> 27
- d -> 28

```

Figura 4 - Saída do terminal parte 1 e parte 2

* Saída:		
- d	1	- c 13
- d	2	- c 14
- d	3	- c 15
- d	4	- c 16
- d	5	- c 17
- d	6	- e 1
- c 1		- e 2
- c 2		- e 3
- c 3		- e 4
- c 4		- e 5
- c 5		- e 6
- c 6		- e 7
- c 7		- e 8
- c 8		- d 1
- c 9		- d 2
- a 1		- d 3
- a 2		- d 4
- a 3		- d 5
- a 4		- d 6
- a 5		- d 7
- a 6		- d 8
- a 7		- d 9
- a 8		- d 10
- a 9		- d 11
- a 10		- d 12
- a 11		- d 13
- c 1		- d 14
- c 2		- d 15
- c 3		- d 16
- c 4		- d 17
- c 5		- d 18
- c 6		- d 19
- c 7		- d 20
- c 8		- d 21
- c 9		- d 22
- c 10		- d 23
- c 11		- d 24
- c 12		- d 25
		- d 26
		- d 27
		- d 28

Figura 5 - Proposta do trabalho para o meu GRR parte 1 e 2

3 CONCLUSÃO

Durante o desenvolvimento deste trabalho, explorando a implementação de semáforos de forma a gerenciar a execução de múltiplas threads na linguagem C, tornou-se evidente como o controle de execução das threads pode ditar o funcionamento de um programa. Onde sem a utilização desta metodologia não há como controlar a ordem de execução das threads em sua função genérica, podendo resultar erros na saída de programas mais críticos devido a execução das threads na ordem incorreta.

4 REFERÊNCIAS BIBLIOGRÁFICAS

MAZIERO, CARLOS A. Sistemas Operacionais: Conceitos e Mecanismos. Curitiba: DINF UFPR, 2019.