**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS – PUC- CAMPINAS**

***Experimento 3 Sistemas Operacionais A***

|  |  |
| --- | --- |
| **ALUNO** | **RA** |
| Beatriz Morelatto Lorente | 18071597 |
| Cesar Marrote Manzano | 18051755 |
| Fabricio Silva Cardoso | 18023481 |
| Pedro Ignácio Trevisan | 18016568 |

# Sumário

# Introdução 3

# Apresentação dos erros do programa exemplo e suas soluções 4

# Resultados da execução do programa exemplo 9

# Resultados da execução do programa modificado 10

# Respostas das perguntas

# Análise dos Resultados

# Conclusão

**Introdução**

O experimento realizado permitiu o entendimento do uso de semáforos do System V e do uso de memória compartilhada. O experimento foi dividido em duas tarefas, com o objetivo de imprimir um vetor (que continha todas as letras maiúsculas e minúsculas e todos os números), com e sem o uso do mecanismo de exclusão mútua.

Na primeira tarefa foi executado um programa exemplo, no qual três processos filhos chamavam a mesma função para imprimir o vetor. No programa há um recurso compartilhado, um inteiro usado como índice para acessar o vetor de caracteres, e os processos filhos, por sua vez, tentam exibir os caracteres ao mesmo tempo, por isso o uso do semáforo é necessário.

Na segunda tarefa, o programa foi modificado de forma que houvesse 8 filhos e metade seriam produtores de caracteres e os outros consumidores de caracteres. Os produtores acessavam o vetor e colocavam os caracteres em um buffer compartilhado. Um consumidor substituía um caractere produzido pelo caractere ‘#’. Quando o buffer estivesse cheio, era necessário que um produtor e um consumidor exibissem o conteúdo de todo o buffer. Cada filho produtor também mostrava os caracteres produzidos.

**Apresentação dos erros do programa exemplo e suas soluções**

Ao compilar o programa pela primeira vez, foi mostrado no prompt alguns erros de sintaxe e lógica de programação. Os problemas estão listados abaixo, seguidos de suas soluções (as correções estão destacadas em negrito e itálico).

**Problema 1**

/\* #define PROTECT \*/

**Problema corrigido:**

A definição PROTECT estava comentada, não possibilitando que a exclusão mútua ocorresse. Para corrigirmos, apenas retiramos o comentário.

***#define PROTECT***

**Problema 2**

#include <errno.h>

#inlcude <sys/time.h>

#include <unistd.h>

#include <sys/types.h>

#inlcude <sys/wait.h>

#include <signal.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

**Problema corrigido:**

Faltava a biblioteca <stdlib.h> para o uso da função exit().

***#include <stdlib.h>***

#include <errno.h>

#inlcude <sys/time.h>

#include <unistd.h>

#include <sys/types.h>

#inlcude <sys/wait.h>

#include <signal.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

**Problema 3**

g\_sem\_op1[0].sem\_num = 0;

g\_sem\_op1[0].sem\_op = -1;

g\_sem\_op1[0].sem\_flg = 0;

g\_sem\_op1[0].sem\_num = 0;

g\_sem\_op1[0].sem\_op = 1;

g\_sem\_op1[0].sem\_flg = 0;

**Problema corrigido:**

A mesma estrutura estava sendo usada para travar e destravar os semáforos.

g\_sem\_op1[0].sem\_num = 0;

g\_sem\_op1[0].sem\_op = -1;

g\_sem\_op1[0].sem\_flg = 0;

***g\_sem\_op2[0].sem\_num = 0;***

***g\_sem\_op2[0].sem\_op = 1;***

***g\_sem\_op2[0].sem\_flg = 0;***

**Problema 4**

if( semop( g\_sem\_id, g\_sem\_op1, 1 ) == -1 ) {

fprintf(stderr,"chamada semop() falhou, impossivel inicializar o semaforo!");

exit(1);

}

**Problema corrigido:**

O semáforo necessita começar a sua execução destravado e não travado, como ocorre.

if( semop( g\_sem\_id, ***g\_sem\_op2***, 1 ) == -1 ) {

fprintf(stderr,"chamada semop() falhou, impossivel inicializar o semaforo!");

exit(1);

}

**Problema 5**

if( (g\_shm\_id = shmget( SHM\_KEY, sizeof(int), IPC\_CREAT | 0000)) == -1 ) {

fprintf(stderr,"Impossivel criar o segmento de memoria compartilhada!\n");

exit(1);

}

**Problema corrigido:**

A permissão para criar a memória compartilhada está errada, usamos a permissão 0666.

if( (g\_shm\_id = shmget( SHM\_KEY, sizeof(int), IPC\_CREAT | ***0666***)) == -1 ) {

fprintf(stderr,"Impossivel criar o segmento de memoria compartilhada!\n");

exit(1);

}

**Problema 6**

rtn = 1;

for( count = 0; count < NO\_OF\_CHILDREN; count++ ) {

if( rtn != 0 ) {

pid[count] =rtn = fork();

} else {

exit

}

}

**Problema corrigido:**

Na diretiva else, há a palavra ‘exit’, que deveria ser a função exit(). Para corrigirmos o problema, usamos o break.

rtn = 1;

for( count = 0; count < NO\_OF\_CHILDREN; count++ ) {

if( rtn != 0 ) {

pid[count] =rtn = fork();

} else {

***break;***

}

}

**Problema 7**

kill(pid[0], SIGKILL);

kill(pid[1], SIGKILL);

kill(pid[2], SIGKILL);

kill(pid[3], SIGKILL);

kill(pid[4], SIGKILL);

**Problema corrigido:**

O pai matava os filhos, porém no programa só havia 3 filhos e não 5. Para deixar a tarefa mais funcional, matamos os filhos com um for.

***int child;***

***for(child = 0; child < NO\_OF\_CHILDREN; child++){***

***kill(pid[child], SIGKILL);***

***}***

**Problema 8**

for( i = 0; i < number; i++ ) {

if( ! (tmp\_index + i > sizeof(g\_letters\_and\_numbers)) ) {

fprintf(stderr,"%f7", g\_letters\_and\_numbers[tmp\_index + i]);

fputc(g\_letters\_and\_numbers[tmp\_index + i], arq);

usleep(1);

}

}

**Problema corrigido:**

Ao imprimir o vetor, estava sendo usado o formato para float. Trocamos para o formato char.

for( i = 0; i < number; i++ ) {

if( ! (tmp\_index + i > sizeof(g\_letters\_and\_numbers)) ) {

***fprintf(stderr,"%c", g\_letters\_and\_numbers[tmp\_index + i]);***

fputc(g\_letters\_and\_numbers[tmp\_index + i], arq);

usleep(1);

}

}

**Problema 9**

#ifdef PROTECT

if( semop( g\_sem\_id, g\_sem\_op1, 1 ) == -1 ) {

fprintf(stderr,"chamada semop() falhou, impossivel liberar o recurso!");

exit(1);

}

#endif

**Problema corrigido:**

Ao liberarmos o recurso usando o semáforo, este estava sendo trancado e não liberado, para corrigir usamos o g\_sem\_op2.

#ifdef PROTECT

if( semop( g\_sem\_id, ***g\_sem\_op2***, 1 ) == -1 ) {

fprintf(stderr,"chamada semop() falhou, impossivel liberar o recurso!");

exit(1);

}

#endif

**Resultados da execução do programa exemplo**

Abaixo são mostrados os resultados da execução da tarefa 1 (programa exemplo). Foram feitas 10 rodadas de teste, sendo 5 rodadas com o mecanismo de exclusão mútua, e outras 5 rodadas sem o mecanismo.

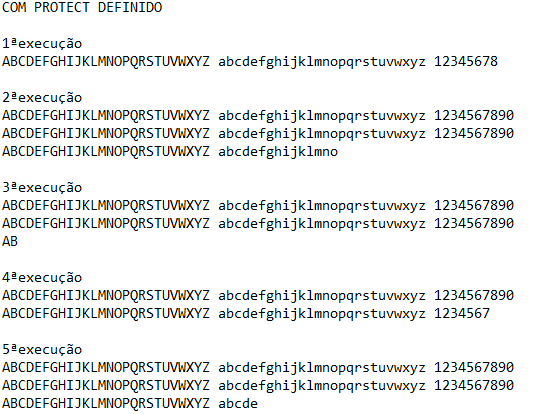


Figura 1 - Resultados do programa exemplo com protect definido



Figura 2 -Resultados do programa exemplo sem protect definido

**Resultados da execução do programa modificado**

**Respostas das perguntas**

**Perguntas do relatório**

***Pergunta 1: Uma região por ser crítica tem garantida a exclusão mútua? Justifique.***

**Resposta:** Sim, para que o processo seja atômico, ou seja, para manter a consistência dos dados, é necessário que tenha exclusão mútua na região crítica.

***Pergunta 2: É obrigatório que todos os processos que acessam o recurso crítico tenham uma região crítica igual?***

**Resposta:** Não, basta que estejam compartilhando o mesmo recurso, a mesma variável para que tenham uma região crítica.

***Pergunta 3: Porque as operações sobre semáforos precisam ser atômicas***

**Resposta:** Porque no momento de escalonamento um processo B pode acessar uma variável que o processo A ainda não terminou de usá-la. Sendo assim o dado desta variável foi manipulado incorretamente, gerando uma inconsistência e para isto não ocorrer é necessário que o momento de manipulação de uma variável compartilhada seja atômico, uma vez iniciado não será interrompido até sua finalização, só pode acorrer por inteiro.

***Pergunta 4: O que é uma diretiva ao compilador?***

**Resposta:** As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, executado pelo compilador antes da execução do processo de compilação propriamente dito.

***Pergunta 5: Porque o número é pseudo aleatório e não totalmente aleatório?***

**Resposta:** Porque não existem funções que gerem números genuinamente aleatórios, eles são formados por operações matemáticas, e uma vez que a semente da operação se repetir, toda a sequência irá se repetir também.

**Perguntas do programa**

***Pergunta 1: Se usada a estrutura g\_sem\_op1 terá qual efeito em um conjunto de semáforos?***

**Resposta:** A estrutura g\_sem\_op1 é usada para travar o semáforo, na hora de entrar em uma região critica.

***Pergunta 2: Para que serve esta operação semop(), se não está na saída de uma região crítica?***

**Resposta:** A operação serve para que o semáforo comece a execução do programa destravado.

***Pergunta 3: Para que serve essa inicialização da memória compartilhada com zero?***

**Resposta:** A variável \*g\_shm\_addr é inicializada com zero, porque ela é um ponteiro inteiro que aponta para o segmento de memória compartilhada, portanto é necessário inicializar no início da memória.

***Pergunta 4: Se os filhos ainda não terminaram, semctl e shmctl, com o parâmetro IPC-RMID, não permitem mais o acesso ao semáforo / memória compartilhada?***

**Resposta:** No caso do programa exemplo, os filhos são mortos antes que os semáforos e a memória compartilhada sejam excluídos, logo não ocorrerá o acesso. Porém, se forem removidos antes, os filhos não terão acesso.

***Pergunta 5: Quais os valores possíveis de serem atribuídos a number?***

**Resposta:** 1, 2 ou 3.

**Análise dos Resultados**

**Tarefa 1 (programa exemplo)**

**Tarefa 2 (programa modificado)**

A tarefa 2 exigia uma certa modificação do código exemplo, como aumentar o número de filhos para oito (quatro produtores e quatro consumidores), criar dois conjuntos de semáforos, de forma que eles protejam o consumidor e o produtor de entrar na memória compartilhada, e quando o consumidor for consumir trocar o carácter consumido do vetor de caracteres pelo '#', para sabermos o que está acontecendo. Após várias execuções do programa, pudemos perceber uma grande diferença de quando temos o conjunto de semáforos ativo e de quando este conjunto está desativado. Igual a tarefa 1, a exibição dos caracteres sem o semáforo foi completamente desordenada e pela quantidade de consumidores e produtores, percebemos a grande importância que os semáforos tem para obter o planejado, nesse caso , a exibição dos caracteres e quão errado sai os valores.

**Conclusão**

Através desse experimento foi discutido como funciona a exclusão mútua de uma área da memória, discutir esse conceito, perceber como funciona quando temos um semáforo para gerar essa exclusão mútua e quando não temos um semáforo para que aconteça essa exclusão mútua e ver os resultados.

Também foi possível aprender novas funções como: semget(), semop(), shmget(), shmat(). A primeira gera um id para o semáforo, a segunda tranca ou destranca o semáforo,a terceira criar uma memória compartilhada e a quarta acopla o processo na memória compartilhada.

O comando ipcs do shell do Linux nos ajudou a arrumar alguns erros que tínhamos, e saber se estava sendo criado de forma correta a memória compartilhada e os semáforos.

O ipcs mostra a fila de mensagens, a memória compartilhada e os semáforos.