**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 34

# Análise dos Resultados 36

# Conclusão 37

**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.

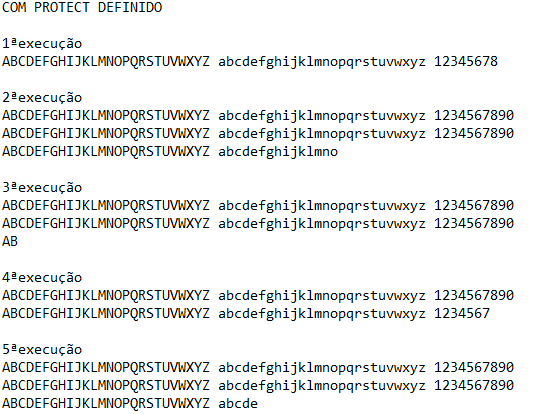


Figura - Resultados do programa exemplo com protect definido



Figura -Resultados do programa exemplo sem protect definido

**Resultados da execução do programa modificado**

Abaixo são mostrados alguns dos resultados da execução da tarefa 2 (programa modificado). Foram feitas 10 rodadas de teste, sendo 5 rodadas com o mecanismo de exclusão mútua, e outras 5 rodadas sem o mecanismo.

Nessa tarefa será colocado apenas o resultado da primeira execução de cada uma das rodadas.

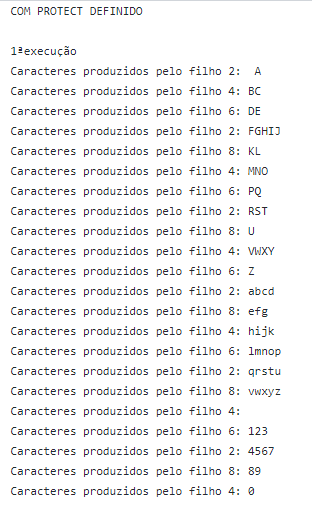


Figura 3 - Resultado da primeira execução com protect definido (parte 1)

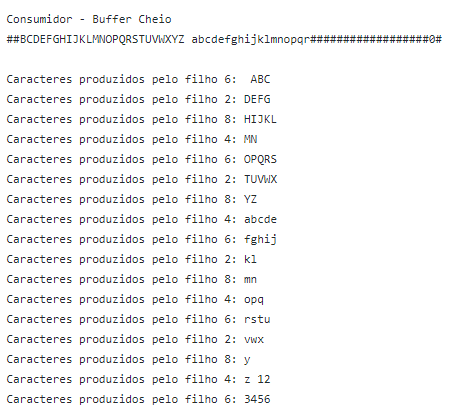


Figura 4 - Resultado da primeira execução com protect definido (parte 2)

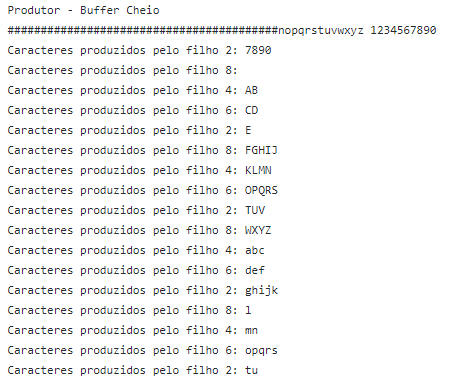


Figura 5 - Resultado da primeira execução com protect definido (parte 3)

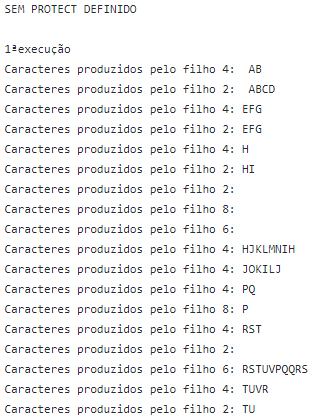


Figura 6 - Resultado da primeira execução sem protect definido (parte 1)

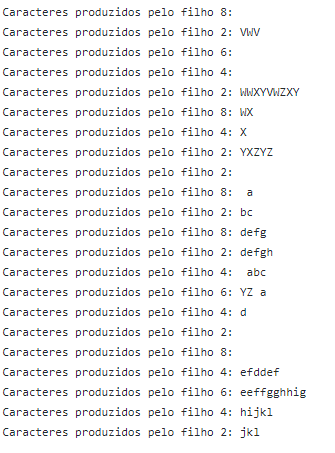


Figura 7 - Resultado da primeira execução sem protect definido (parte 2)

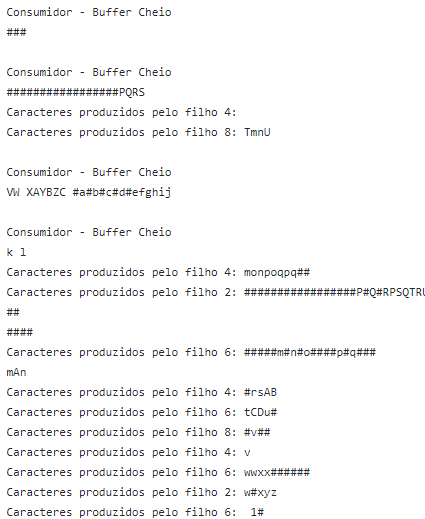


Figura 8 - Resultado da primeira execução sem protect definido (parte 3)

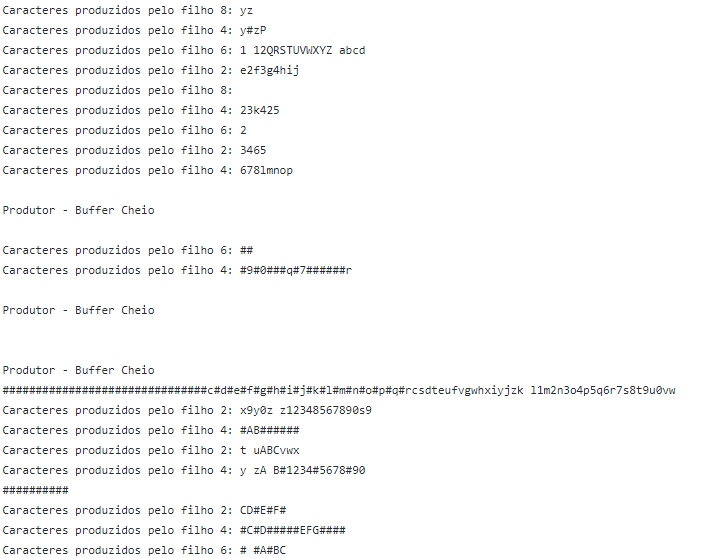


Figura 9 - Resultado da primeira execução sem protect definido (parte 4)



Figura 10 - Resultado da primeira execução sem protect definido (parte 5)

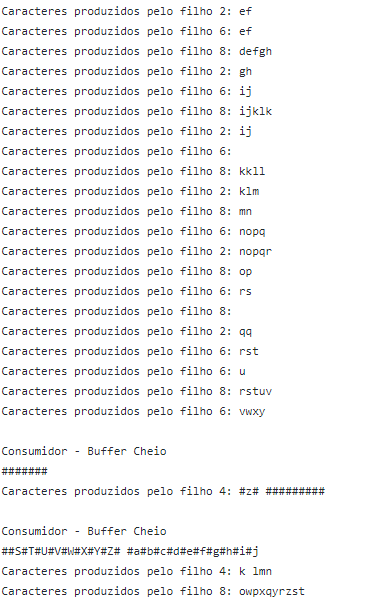


Figura 11 - Resultado da primeira execução sem protect definido (parte 6)

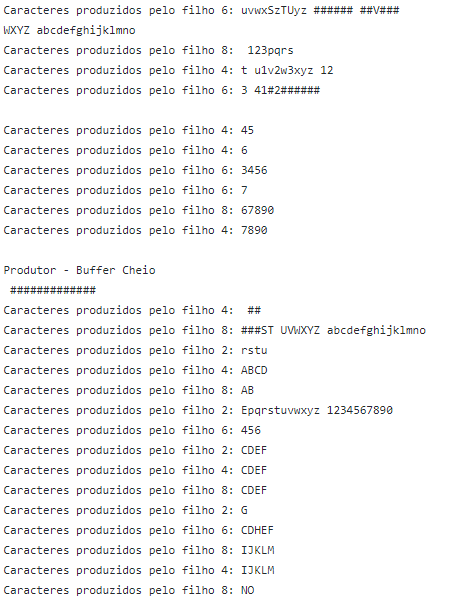


Figura 12 - Resultado da primeira execução sem protect definido (parte 7)

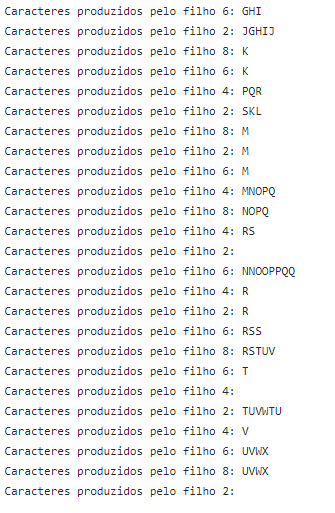


Figura 13 - Resultado da primeira execução sem protect definido (parte 8)

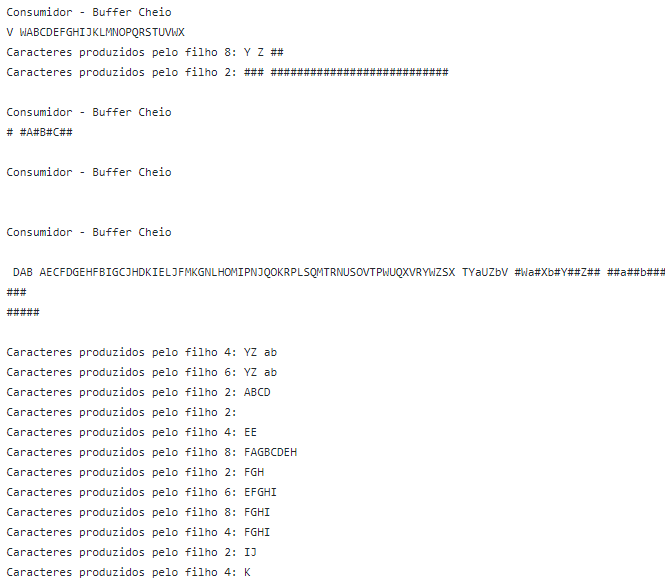


Figura 14 - Resultado da primeira execução sem protect definido (parte 9)

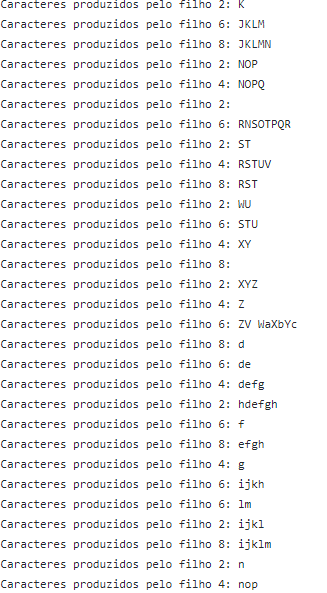


Figura 15 - Resultado da primeira execução sem protect definido (parte 10)

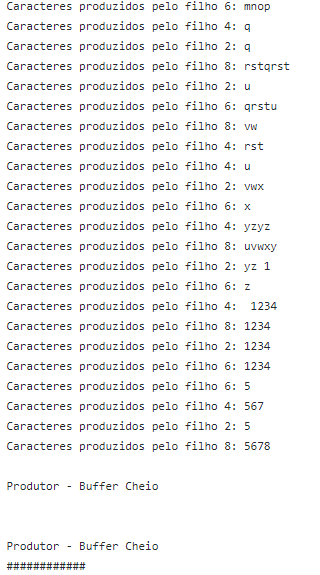


Figura 16 - Resultado da primeira execução sem protect definido (parte 11)

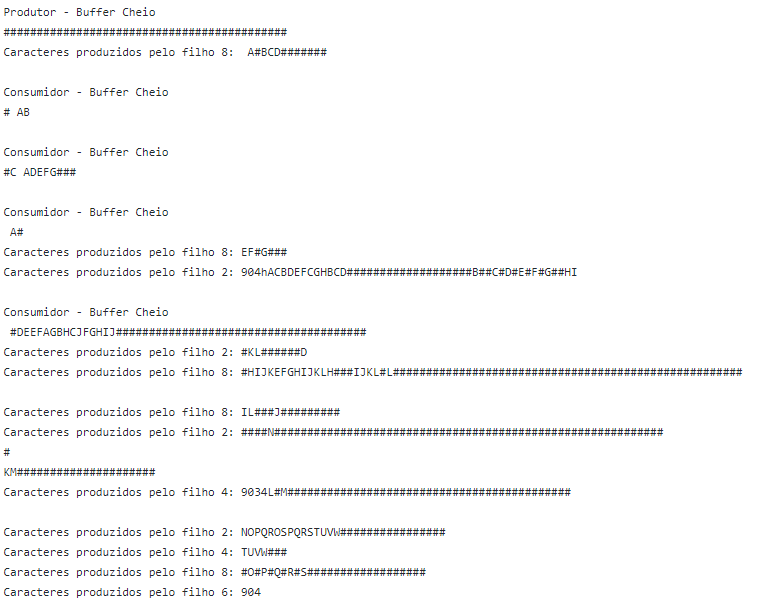


Figura 17 - Resultado da primeira execução sem protect definido (parte 12)

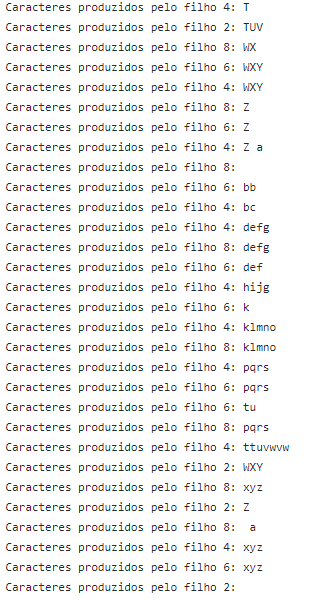


Figura 18 - Resultado da primeira execução sem protect definido (parte 13)

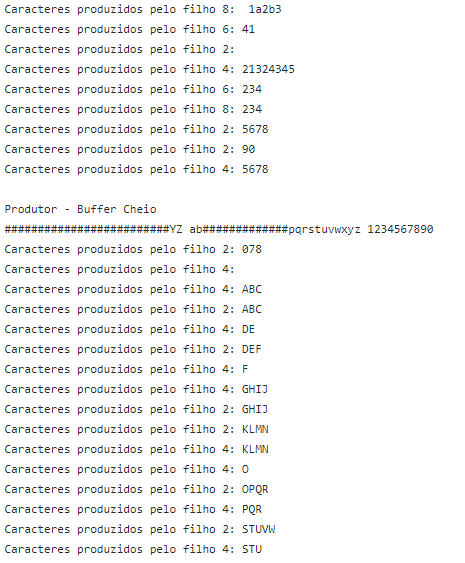


Figura 19 - Resultado da primeira execução sem protect definido (parte 14)

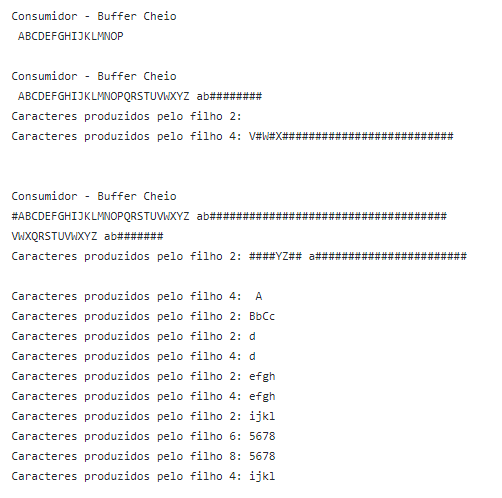


Figura 20 - Resultado da primeira execução sem protect definido (parte 15)



Figura 21 - Resultado da primeira execução sem protect definido (parte 16)

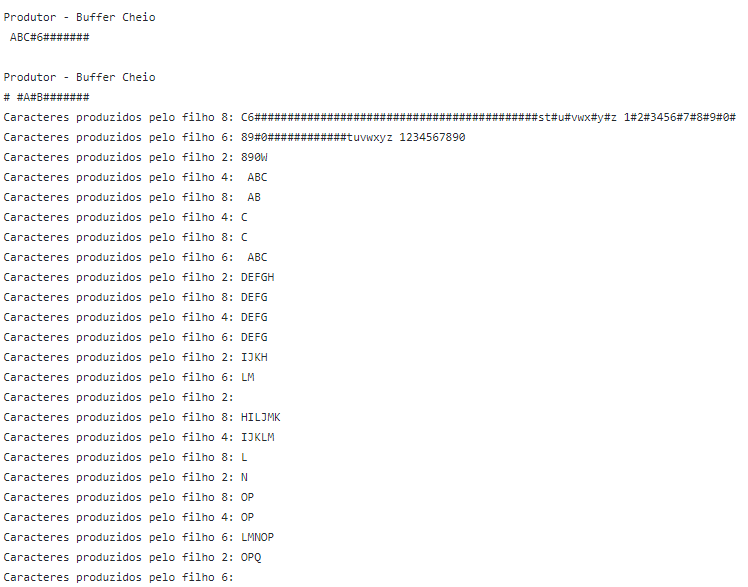


Figura 22 - Resultado da primeira execução sem protect definido (parte 17)

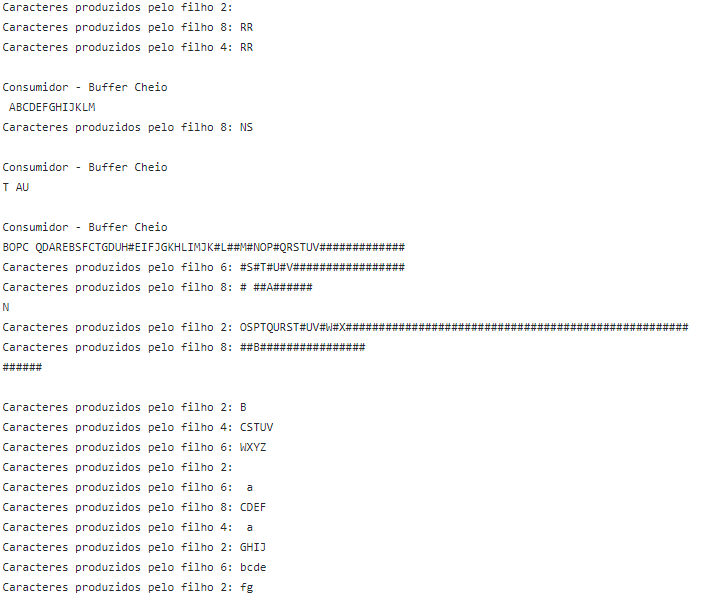


Figura 23 - Resultado da primeira execução sem protect definido (parte 18)

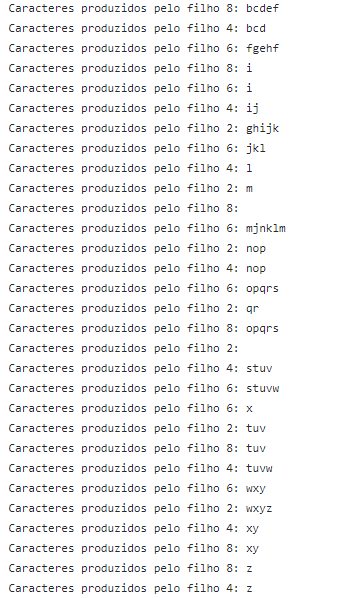


Figura 24 - Resultado da primeira execução sem protect definido (parte 19)

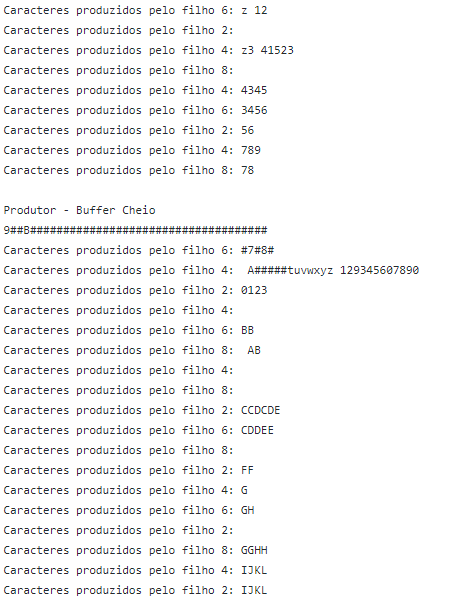


Figura 25 - Resultado da primeira execução sem protect definido (parte 20)

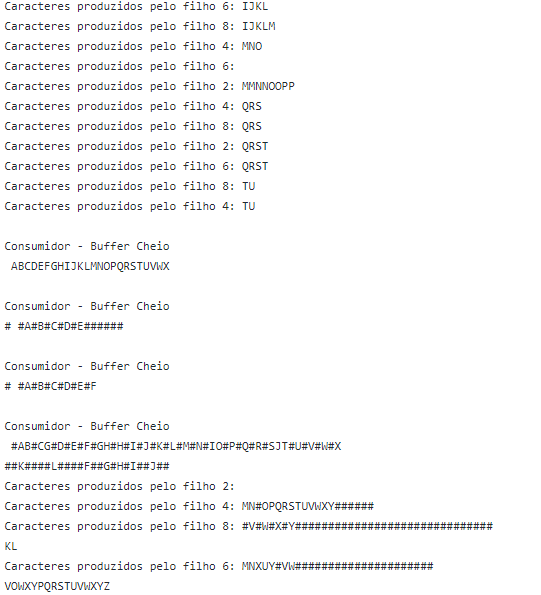


Figura 26 - Resultado da primeira execução sem protect definido (parte 21)

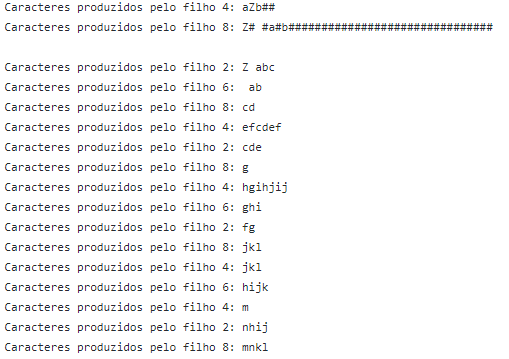


Figura 27 - Resultado da primeira execução sem protect definido (parte 22)

**Respostas das perguntas**

**Perguntas do relatório**

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

**Resposta:** Não. Para que a região tenha a exclusão mútua, é necessário o programador implementar algum método, como semáforo (usado no experimento).

***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)**

A tarefa 1 possibilitou o melhor entendimento sobre o uso de semáforos e sua importância. Antes de discutirmos o efeito do semáforo no programa, é necessário a discussão do conceito de race condition, exclusão mútua e região crítica.

Race condition (condição de corrida), ocorre quando dois ou mais processos querem acessar o mesmo recurso e o resultado depende de quem foi escalonado primeiro. Portanto, dependendo da ordem de escalonamento, o resultado obtido não será o esperado. Para evitar que processos concorrentes acessem o mesmo recurso, é necessário algum mecanismo de sincronização. A exclusão mútua garante que, se um processo esteja acessando uma variável ou um arquivo compartilhado, os outros processos estarão bloqueados, garantindo acesso exclusivo de um único processo. A região (ou seção) crítica, é onde se localiza os recursos compartilhados.

Como já citado, o recurso compartilhado é um inteiro, usado para acessar uma string de caracteres. O uso do semáforo é necessário para que a impressão ocorra de forma correta, como apresentada na seção de resultados.

Percebemos que quando o protect está definido, a impressão ocorre de forma correta, as letras e números são imprimidos exatamente na ordem em que estão na string. Portanto, podemos concluir que o uso do protect é essencial para que o semáforo faça sua função corretamente.

Já quando o protect não está definido, a impressão não ocorre da maneira esperada. Isso se deve ao fato de que vários processos tentaram acessar a variável compartilhada, mudando o valor da mesma, de maneira incorreta, a cada execução, fazendo com que a impressão ocorresse de maneira errada.

**Tarefa 2 (programa modificado)**

Os mesmos conceitos de race condition, exclusão mutua e região crítica valem para esta tarefa. Também foi observado como o uso do protect é necessário para evitar que dois ou mais processos acessem a região crítica ao mesmo tempo.

Como foi observado nos resultados desta tarefa, quando usamos o protect, a impressão ocorreu de forma correta, mostrando o que cada filho produziu e o buffer quando estivesse cheio. Mais uma vez, verificamos que o uso de semáforos nessa tarefa foi essencial.

O não definição do protect, assim como na tarefa 1, levou a uma impressão completamente desordenada. Também observamos a extensão dos resultados (podemos observar que os resultados colocados aqui no relatório, foram divididos em vinte e duas partes). Isso se deve ao fato de que os processos filhos (8 nessa tarefa), não são bloqueados pelos semáforos, e como cada processo é independente, estes executam até o pai executar o comando kill.

**Conclusão**

Através deste experimento, foi observado o funcionamento de semáforos e da memória compartilhada. Conceitos como exclusão mútua, região crítica e condições de corrida, também foram discutidos e o entendimento de cada um ficou mais fácil.

Também foi possível verificar o funcionamento de algumas funções par manipulação de semáforos e memoria compartilhada, como: semget(), semop(), semctl(), shmget(), shmat() e shmctl(). As três primeiras, são relacionadas à semáforos e as três últimas à memória compartilhada (o comando man do Linux auxilia no entendimento de cada uma).