PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA

CHRISTOPHER OLIVEIRA	**RA:18726430**
GIULIANO SANFINS	**RA:17142837**
MATHEUS MORETTI	**RA:18082974**
MURILO ARAUJO	**RA:17747775**
VICTOR REIS	**RA:18726471**

SISTEMAS OPERACIONAIS A - EXPERIMENTO 3

CAMPINAS
2020

SUMÁRIO

1. INTRODUÇÃO

O experimento tem como objetivo observar e entender por que um recurso crítico deve ser protegido contra acessos simultâneos, esclarecer o conceito de exclusão mútua, e aprender a como utilizar a ferramenta de semáforos. Foi corrigido o código que foi entregue, tanto logicamente, quanto sintaticamente e então alterado para execução da segunda atividade. Sendo o foco do experimento a utilização correta dos semáforos e entender o que ocorre quando não se utiliza essa ferramenta, quando há acesso simultâneo de um mesmo espaço da memória.

Após o entendimento do conteúdo que envolve de forma direta o experimento e de modificar o código, foi realizado uma série de testes em relação às saídas do programa, registrados os resultados para analisá-los de forma coerente.

2. DISCUSSÃO

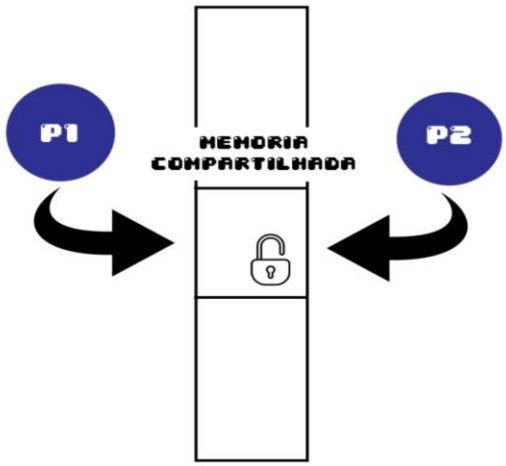

Figura 1: Memória compartilhada

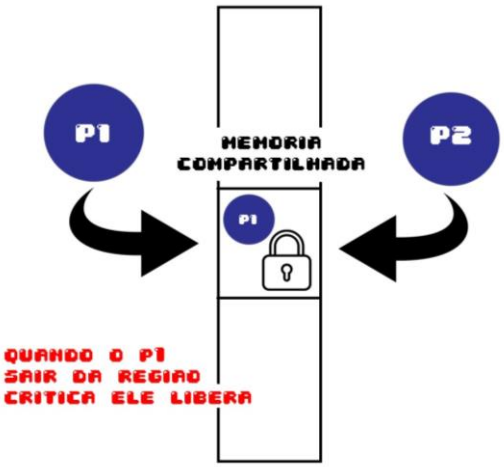

Figura 2: Memória compartilhada e região crítica

Se 2 processos querem acessar e alterar uma mesma região de memória, para que não ocorra as chamadas *race-condition*, é necessário fazer uso de métodos que garantam exclusão mútua, ou seja, uma forma de restringir os acessos a dados compartilhados caso um processo esteja executando dentro de sua região crítica, e nesse exemplo foi utilizado os semáforos. Semáforos funcionam da seguinte maneira, inicialmente o semáforo está aberto, quando um processo entra na sua região crítica o semáforo é trancado impedindo que outros processos acessem a mesma região da memória, ao acabar ele libera o semáforo permitindo que outro processo consiga acessá-lo.

A respeito do código, inicialmente foram feitas as correções necessárias de lógica e sintaxe, depois para continuação do exercício foram os adicionados dois novos buffers, um para guardar o index do consumidor, que conforme os itens são consumidos o index é alterado, e o outro para guardar a mensagem produzida (que ao final será impressa na tela), já existia outro buffer que guarda o index do produtor. Fora criado um novo semáforo para controlar o index do consumidor, pois por existir mais de um consumidor (existem 4 consumidores e 4 produtores) é necessário controlar quem vai acessá-lo para que não haja problemas de inconsistência nos dados. As funções referentes aos consumidores e produtores foram separadas para modularização do código.

O programa funciona basicamente da seguinte maneira, caso o processo do consumidor acesse a memória compartilhada, ele trava o semáforo, gera um número randômico entre 1 e 5 para ver quantos itens ele irá consumir, porém antes de consumir ele checa se o index do produtor é maior que o index do consumidor mais o número randômico gerado, caso não seja ele não consome nada e libera o semáforo, mas caso seja ele então consome esses itens (e os troca por #), então atualiza o seu index e libera o semáforo; Caso o processo do produtor acesse a memória compartilhada, ele travará o semáforo, gerar um número de 1 a 5 para saber quantos itens irá produzir, então os produz (modificando a memória compartilhada) então atualiza seu index e libera o semáforo, e o programa é

finalizado quando o *wait* do processo pai acaba, e então todos os filhos, memórias compartilhadas e semáforos são finalizados, e o programa é encerrado.

Fizemos uso de algumas funções pré-fabricadas como *semop(), semctl(), semget(),* e suas respectivas variações para *shm*, memória compartilhada. Para isso, tivemos que incluir as seguintes bibliotecas: *types.h, ipc.h, sem.h, shm.h*.

A função ***semop()*** permite de efetuar operações sobre os semáforos identificado por *semid*, sendo especificado se a operação será de acrescentar um semáforo ou subtrair do mesmo.

Valor de retorno: valor do último semáforo manipulado ou -1 em caso do erro.

A função ***semctl()*** é utilizada para examinar e controlar os valores de cada um dos componentes de um conjunto de semáforos. Ela executa as ações de controle definidas em comando no conjunto de semáforos identificado por *semid*.

Existem vários valores possíveis para *cmd*, todavia, dois são os mais comuns:

SETVAL: utilizado para inicializar o semáforo com um valor conhecido.

IPC_RMID: utilizado para remover um semáforo que não é mais necessário.

Valor de retorno: Depende do valor do argumento *cmd*:

A função ***semget()*** é utilizada para criar um novo conjunto de semáforos, ou para obter o ID de um conjunto de semáforos já existentes.

Valor de retorno: o ID do conjunto de semáforos *semid*, e -1 em caso de erro.

A função ***shmget()*** é encarregada de buscar o elemento especificado (pela chave de acesso *key*) na estrutura *shmid_ds* e, caso esse elemento não exista, de criar um novo segmento de memória compartilhada, com tamanho em bytes igual a *size*.

Valor de retorno: o identificador do segmento de memória compartilhada *shmid*, ou -1 em caso de erro.

A função ***shmat()*** faz a ligação ao segmento através do identificador *shmid*.

Valor de retorno: identificador do segmento se OK, -1 em caso de erro.

A função ***shmctl()*** elimina um segmento de memória partilhada, assumindo que nenhum processo está a ele ligado.

Valor de retorno: 0 se sucesso, -1 em caso de erro.

3. SOLUÇÃO DE ERROS DE SINTAXE

3.1 Modificação no sem_op de g_sem_op1 e 2.

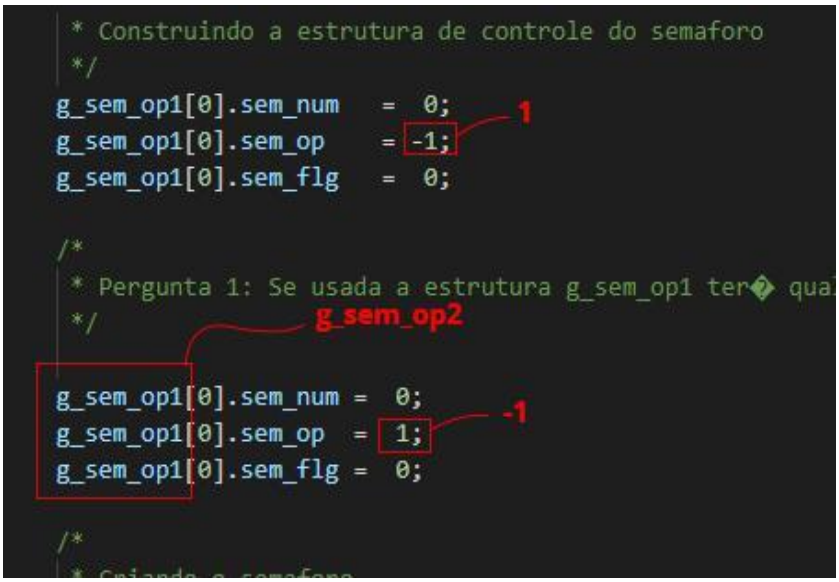

Figura 3: Modificando sem_op de g_sem_op1 e g_sem_op2

3.2 Erro na permissão para leitura e escrita.

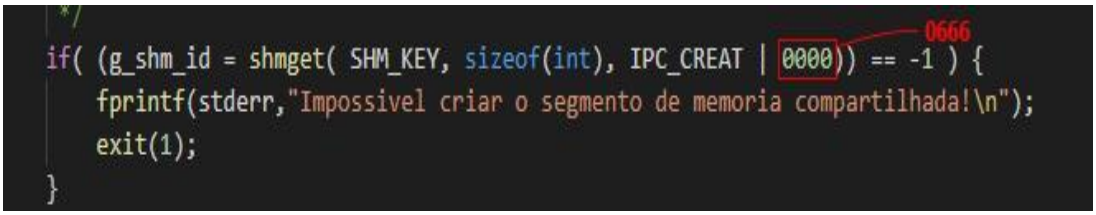

Figura 4: Alterando a erro na permissão (0000 para 0666)

3.3 Erro no exit.

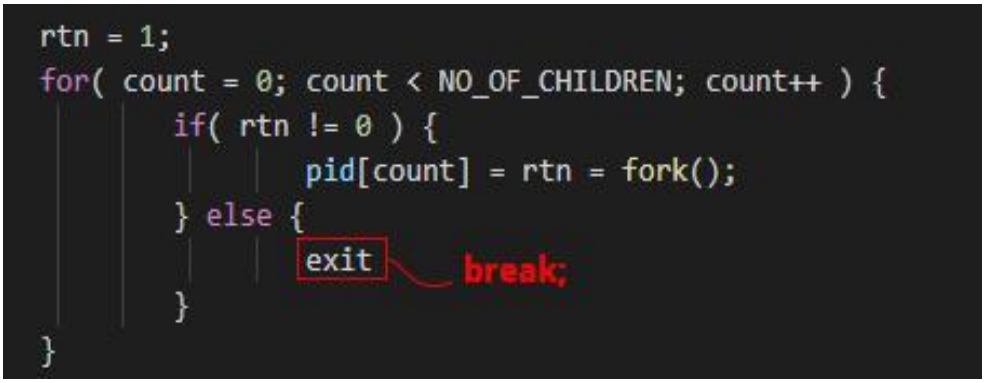

Figura 5: Alterando *exit* por *break*

3.4 Número extra de filhos ao matá-los.

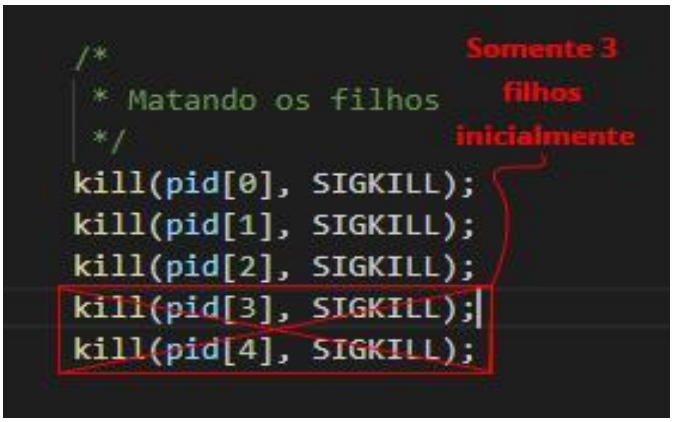

Figura 6: Somente 3 filhos removendo o sinal para 3 e 4

3.5 Tipo da variável na impressão.

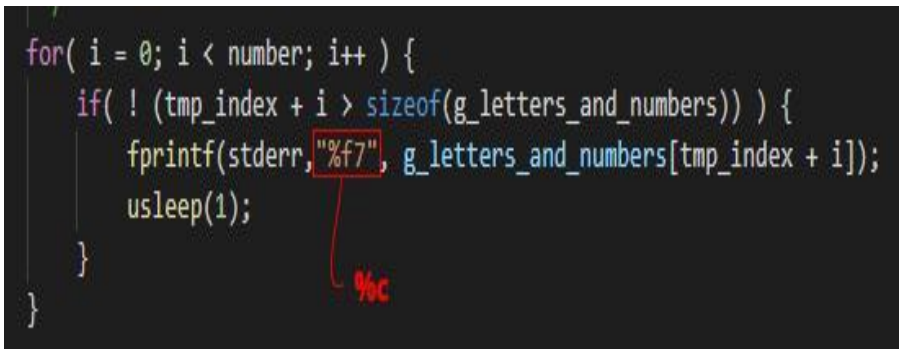

Figura 7: Alterando %f por %c

4 PERGUNTAS

4.1 Pergunta 1

Uma região por ser crítica tem garantida a exclusão mútua? Justifique.

R: Não, pois a região crítica trata-se apenas da região do código onde se faz o uso dos recursos compartilhados, e somente isso não garante exclusão mútua, deve-se usar métodos como o semáforo nessas regiões para atingir a exclusão mútua.

4.2 Pergunta 2

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

R: Não é necessário que a região crítica seja igual.

4.3 Pergunta 3

Por que as operações sobre semáforos precisam ser atômicas?

R: Enquanto um processo estiver executando uma dessas duas operações, nenhum outro processo pode executar outra operação sob o mesmo semáforo, devendo esperar que o primeiro processo encerre a sua operação sob o semáforo. Essa obrigação evita condições de disputa entre vários processos.

4.4 Pergunta 4

O que é uma diretiva do compilador?

R: Uma diretiva de pré-processador é prefixada com o símbolo # e aparece em uma linha por si só. Ele é interpretado pelo pré-processador, que é executado antes do próprio compilador. Como *#if, #else* e *#endif*.

4.5 Pergunta 5

Porque o número é pseudoaleatório e não totalmente aleatório?

R: Não é possível criar um número totalmente aleatório, pois sempre é necessária uma semente para gerar um número aleatório e caso a semente seja repetida o número gerado é igual.

4.6 Pergunta 1

Se usada a estrutura g_sem_op1 terá qual efeito em um conjunto de semáforos?

R: Ela aumenta o valor interno do semáforo, e o aumento desse valor interno serve para destrancar o semáforo.

4.7 Pergunta 2

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

R: *Semop()* realiza as operações de trancamento e destrancamento do semáforo. Sendo -1 para trancamento, 1 para destrancamento do semáforo (o 0 não faz nada, ou dependendo da situação serve para testar o semáforo).

4.8 Pergunta 3

Para que serve essa inicialização da memória compartilhada com zero?

R: Para ter um maior controle dos seus índices e também uma organização maior para os filhos ou threads desse processo acessarem essa memória compartilhada, sabendo incrementar os índices corretamente em relação aos outros índices utilizados para a mesma memória compartilhada.

4.9 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?

R: Não permite mais o acesso nem o semáforo nem a memória compartilhada, pois esse comando funciona como *kill* destas estruturas criadas, para que não haja problemas futuros é necessário que ao fim do programa elas sejam destruídas, e neste caso ao fim do processo pai ele as destrói e mesmo que algum filho deseje utilizá-la não consegue.

4.10 Pergunta 5

Quais os valores possíveis de serem atribuídos a *number*?

R: Em relação ao enunciado e o que foi solicitado pelo docente, os valores possíveis de serem atribuídos a *number* faz parte de um intervalo de 1 a 5, sendo um número pseudoaleatório gerado em relação ao tempo da máquina.

5. ANÁLISE DOS RESULTADOS

Verificar amostras de vários testes do mesmo programa experimental é necessário para analisar comportamentos e padrões, com uma maior precisão, desta forma, ao verificar os resultados obtidos, vê-se o comportamento da concorrência sem controle de processos, perdendo totalmente o sentido, consistência e contexto de uma variável, aos devidos múltiplos acessos não oportuno dos processos, portanto é necessário um meio deste problema não ocorrer, para isso é utilizado semáforos, onde o controle de acesso a regiões críticas são efetivado. Feito isto é possível entrever uma consistência muito maior dos dados obtidos.

5.1 Parte 1

Nesta primeira parte do experimento, um programa é utilizado para exemplificar as diferenças entre a utilização e a não utilização de semáforo, em tentativa de um mesmo objetivo, este qual o de percorrer um vetor de caracteres, com vários processos acessando-o concorrentemente, assim todos acessam um índice compartilhado, onde todos começam a percorrer o vetor a partir de tal índice.

Protect desativado:

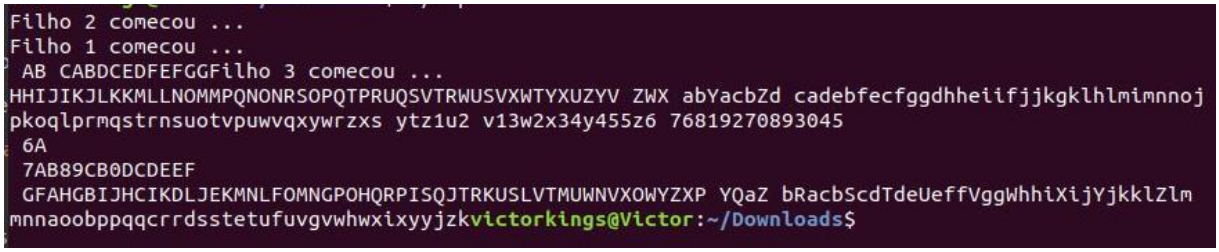

Figura 8: Exemplo da saída de uma execução com o *protect* desativado

Protect Ativado:

Figura 9: Exemplo da saída de uma execução com o *protect* desativado

É possível visualizar que quando não há uma proteção para região crítica compartilhada (Protect desativado), onde existe o índice, os acessos múltiplos dos processos, gera uma inconsistência nos dados, podendo ser verificada a partir da repetição de caracteres. Quando um processo estava acessando o índice, antes de terminar de usá-lo outro processo o acessou também, começando a percorrer o vetor do mesmo ponto que o processo anterior, ocasionando uma saída incoerente.

Porém quando usado semáforo, ou seja, a proteção é ativada (Protect Ativado), vê-se uma saída lógica, seguindo a sequência correta de caracteres. Isso se deve ao bloqueio da região crítica quando um processo a está utilizando, neste caso o índice do vetor, impedindo de que outros processos a acessem ao mesmo tempo.

5.2 Parte 2

Na segunda parte do experimento é implementado um exemplo de produtor/consumidor. Onde alguns processos produzirão caracteres armazenando-os em um buffer compartilhado, e outros processos consumirão o que foi produzido, como antes na parte 1, haverá índice compartilhado, porém agora terá índices separado para o cliente e o servidor.

Protect desativado:

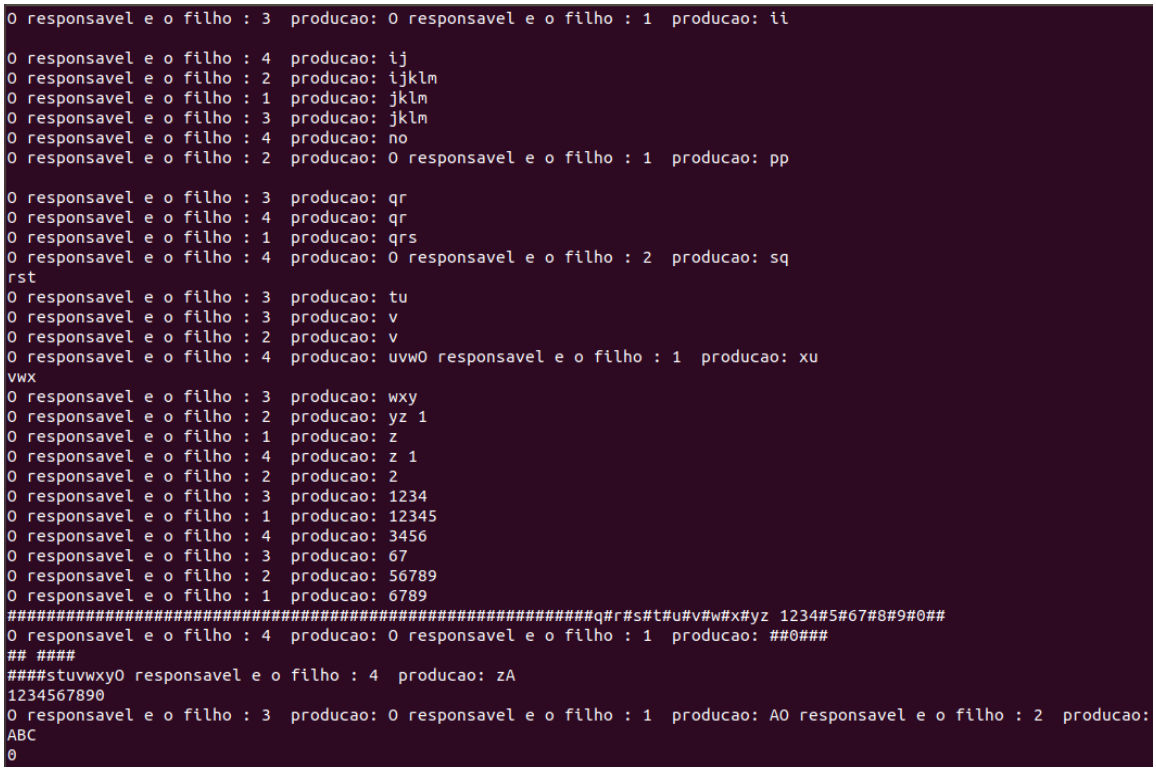

Figura 10: Exemplo da saída de uma execução com o *protect* desativado na parte 2

Protect ativado:

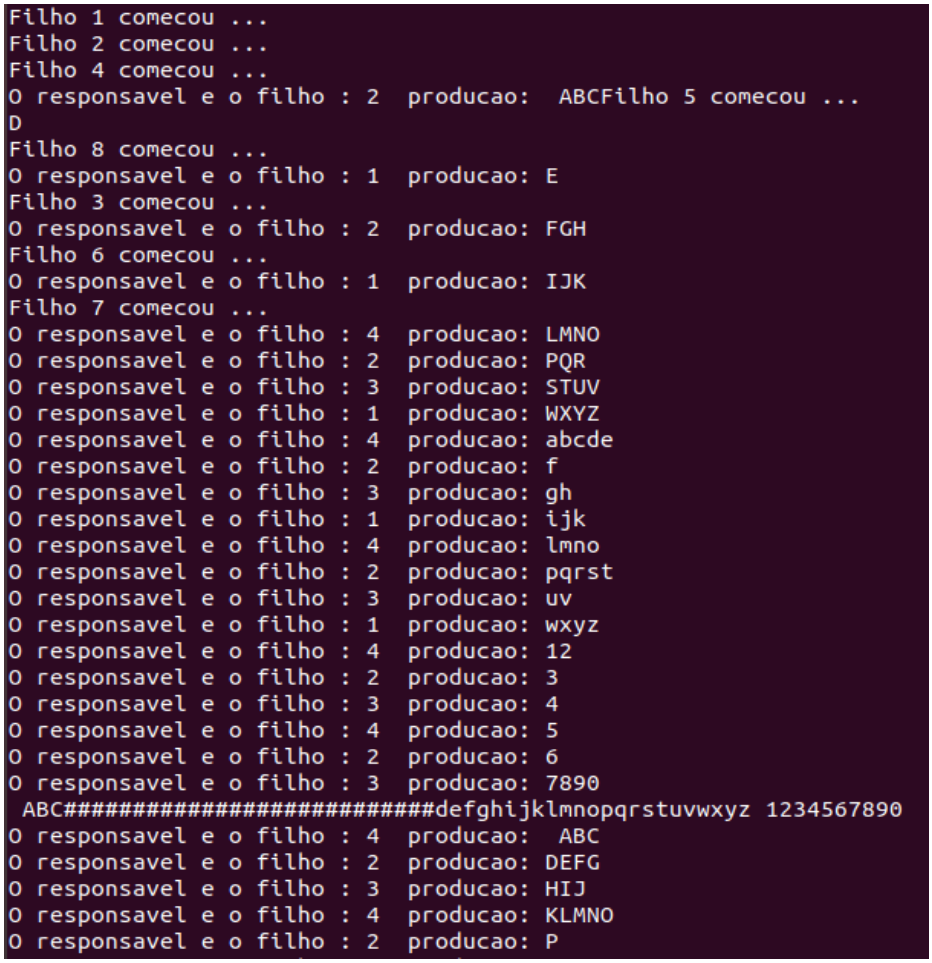

Figura 11: Exemplo da saída de uma execução com o *protect* desativado

Quando não há semáforo (Protect desativado) para o controle do acesso aos índices e buffer, pode ver tanto uma inconsistência na geração dos dados por parte do produtor, quanto na utilização dos dados pelo consumidor.

A produção é feita de modo que há repetição de caracteres, devido ao acesso concorrente de processos ao índice, e também há casos onde os caracteres são sobrescritos, as vezes impedindo do vetor chegar no tamanho mínimo para a impressão na parte do consumo (65, mesmo tamanho do vetor base de caracteres). Outro erro verificado é que na impressão no produtor houve “#”, isso é causado por algum acesso de consumidor, quando este produtor estava utilizando o buffer.

O consumo é também feito de forma errônea, onde os processos não continuam consumindo de onde o anterior terminou, a concorrência para o acesso

a região crítica do índice do consumidor causa esta inconsistência no acesso aos dados.

Quando utilizado semáforo (Protect ativado) nas regiões críticas, tanto no índice de produtor como de consumidor e para o buffer, é possível verificar uma grande consistência nos dados, não há mais uma produção incorreta, sem repetições, sem subscrições, onde os dados seguem a ordem lógica de caracteres do vetor proposto. Os consumidores também funcionam perfeitamente, sem nenhuma falha no consumo, continuando exatamente de onde o último consumo foi feito.

6. CONCLUSÃO

Este terceiro experimento realizado teve como propósito o aprendizado e aprofundamento de um importantíssimo conceito em sistemas operacionais, sendo ele o semáforo. Foi utilizado para aprendizagem um exemplo elaborado na linguagem C, onde foi possível aprender a manipular e utilizar as seguintes funções em relação ao conceito proposto, a *semget(), semop()* e a *semctl()*.

Também com esse trabalho foi possível aprender mais sobre como analisar programas que envolvem semáforos e a sua “saída”, podendo verificar a diferença de se utilizar ou não, entendendo melhor a memória compartilhada do programa e como funciona e o acesso dos filhos nessa região dentro de um processo repetitivo.

Com o auxílio dessas três principais funções semget (), semop () e a semctl (), que utilizamos, facilitou bastante na nossa aprendizagem, pois foi possível compreender melhor a utilização em si do semáforo e todo o seu conceito, em vez de focar em como ele é criado e implementado do zero, ou seja, entendendo melhor e tendo um foco maior no seu funcionamento na prática.

Destarte o objetivo almejado foi alcançado com êxito, correspondendo a todas expectativas de aprendizado sobre o assunto em questão, tornando nossos conhecimentos sobre processos cada vez mais diversificado, e nos ajudando a compreender como realmente tudo acontece no sistema operacional, mesclando ideias antes soltas, em algo muito mais consistente, pois tivemos a oportunidade de estar aplicando todos essas ideias e conceitos na prática em problemas dinâmicos.