PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

ENGENHARIA DE COMPUTAÇÃO

SISTEMAS OPERACIONAIS A

Carlos Henrique Vieira Marques – 18720367

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

João Henrique Pereira – 18712919

Leonardo Sanavio – 18054395

EXPERIMENTO #3

EXCLUSÃO MÚTUA, IMPLEMENTAÇÃO DE SEMÁFOROS E MEMÓRIA COMPARTILHADA

CAMPINAS

Março 2020

SUMÁRIO

1. INTRODUÇÃO 3

2. PERGUNTAS 4

2.1. REFERENTES AOS SISTEMAS UNIX (SYSTEM-V) 4

2.2. REFERENTES AO PROGRAMA EXEMPLO 6

3. ERROS DO PROGRAMA EXEMPLO 7

4. RESULTADOS 11

4.1. REFERENTES AO PROGRAMA EXEMPLO 11

4.2. REFERENTES AO PROGRAMA MODIFICADO 13

5. ANÁLISE DOS RESULTADOS 13

5.1. PROGRAMA EXEMPLO 14

5.2. PROGRAMA MODIFICADO 14

6. CONCLUSÃO 14

1. INTRODUÇÃO

O experimento realizado tem como objetivo o estudo e aprofundamento sobre um recurso crítico e o motivo pelo qual é necessário protegê-lo de acessos “simultâneos”. Além disso, visa-se esclarecer o conceito de exclusão mútua, e entender sobre a aplicação de semáforos. Foi estudado a fundo como é a criação e fechamento de um semáforo, sua utilização e comportamento diante da respectiva atividade realizada.

Na primeira parte do experimento, foi utilizado um programa exemplo fornecido pelo professor, nele é usado um índice para acessar uma string de caracteres com as quais vários processos exibem letras do alfabeto e dígitos cooperativamente, de maneira que não ocorra repetição destas, sendo que, algumas vezes de maneira desordenada pois depende da execução de cada filho. O programa recebido havia erros de lógica e de sintaxe, que foram corrigidos para compilar e realizar os respectivos testes solicitados nessa parte do experimento.

Em seguida, na segunda parte do programa, o código foi modificado para que existam agora oito filhos, sendo quatro dos quais são iguais aos existentes no exemplo, ou seja, são produtores de caracteres, e quatro sendo consumidores. Sendo que, cada vez que se vai produzir ou consumir, sorteia-se um número de caracteres que serão envolvidos, com um total de cinco. Também, cada filho produtor deve mostrar quais caracteres produziu e tanto um produtor como um consumidor deve exibir o conteúdo de todo o buffer. E assim realizado os testes, semelhante à primeira parte, executou-se o programa dez vezes sendo uma metade com e outra sem proteção, o que promoveu o esclarecimento da funcionalidade do semáforo construído e sua aplicação que como pedido, posteriormente foi analisado e comparado.

2. PERGUNTAS

2.1. REFERENTES AOS SISTEMAS UNIX (SYSTEM-V)

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

Resposta: Uma região crítica necessariamente precisa de uma exclusão mútua, para que não ocorra nenhum tipo de incoerência nos dados, quando dois processos acessam dados concorrentemente. Não se pode esquecer que Região Crítica é a área de um código ou recurso compartilhado que depende expressivamente que o acesso à mesma seja realizado de maneira sequencial. Porém, sabendo da possibilidade de condição de corrida, essa região deverá ser tratada de maneira a evitar/tratar condições de corrida.

Com isso, faz-se necessário a implementação da Exclusão Mútua que, por sua vez, consiste em métodos que podem ser criados (via implementação de semáforos, monitores, flags e afins) a fim de evitar o problema de condição de corrida e, de certa maneira, a possibilidade de outros processos invadirem uma região crítica.

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

Resposta: Depende, se nenhum processo está executando dentro de sua seção crítica e existem alguns processos que desejam adentrar suas seções críticas, então a seleção do próximo processo que irá entrar em sua seção crítica não pode ser adiado. Ou seja, se os processos são de recursos compartilhados entre si eles possuem mesmo recurso crítico, já os não compartilhados, não dividem o mesmo recurso crítico, pois os dados estão em memórias diferentes.

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

Resposta: As operações de incrementar e decrementar devem ser operações atômicas, ou indivisíveis, ou seja, 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, além de garantir o bem funcionamento e não "quebrar" a

ordem do semáforo, não interromper as ações que devem ser executadas da respectiva maneira. Essa obrigação evita condições de disputa entre vários processos, visando com que a exclusão mútua não seja violada.

Pergunta 4: O que é uma diretiva ao compilador?

Resposta: Em C, existem comandos que são processados durante a **compilação** do programa. Estes comandos são genericamente chamados de *diretivas de compilação*. Estes comandos informam ao compilador do C basicamente quais são as **constantes simbólicas** usadas no programa e quais **bibliotecas** devem ser anexadas ao programa executável. Existem diversas diretivas como:

Compilação Condicional - Usando diretivas de pré-processamento especiais, você pode incluir ou excluir partes do programa de acordo com várias condições;

Controle de Linha - Se você usa um programa para combinar ou rearranjar os arquivos de origem em um arquivo intermediário, que será compilado, você pode usar o controle de linha para informar Compilação Condicional - Usando diretivas de pré-processamento especiais, você pode incluir ou excluir partes do programa de acordo com várias condições;

Controle de Linha - Se você usa um programa para combinar ou rearranjar os arquivos de origem em um arquivo intermediário, que será compilado, você pode usar o controle de linha para informar o compilador de onde veio cada linha;

Relatório de Erros e de advertência - A diretiva '#error' faz com que o pré-processador Informe um erro fatal e a diretiva '#warning' faz com que a o pré-processador emita um aviso e continue pré-processamento. Compilação Condicional - Usando diretivas de pré-processamento especiais, você pode incluir ou excluir partes do programa de acordo com várias condições;

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

Resposta: O número é pseudoaleatório pois visa-se um padrão na variação de alocação de mensagem e no tempo gasto para a troca de dados. E m gerador de número pseudoaleatório é um algoritmo que gera uma sequência de números, os quais são aproximadamente independentes um dos outros. A saída da maioria dos geradores de números aleatórios não é verdadeiramente aleatória, ela somente aproxima algumas das propriedades dos números aleatórios.

2.2. REFERENTES AO PROGRAMA EXEMPLO

Pergunta 1: Se usada a estrutura g_sem_op1 terá qual efeito em um conjunto de semáforos?

Resposta: A estrutura g_sem_op1 serve para realizar o controle das operações de semáforos. Ela é uma struct com os recursos capazes de guardar o identificador do semáforo (g_sem_op1[].sem_num), o estado do semáforo (g_sem_op1[].sem_flg) como fechado ou aberto, e por fim a operação do semáforo (g_sem_op1[].sem_op), que é a necessidade dele liberar um processo ou fechar sua região crítica para que outro processo não acesse simultaneamente o que este está acessando.

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

Resposta: A função semop() permite efetuar operações sobre os semáforos identificados por semid, caso o semop() esteja fora de uma região crítica significa que o semáforo está aguardando um recurso de um novo processo.

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

Resposta: A inicialização da memória com zero acontece quando é possível gerar o segmento de memória compartilhada. Dessa forma, é endereçado um segmento de memória lógico com valor 0 para posteriormente ser utilizado no segmento de memória físico.

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: Sim, caso ocorra a chamada das funções semctl e shmctl os semáforos e o acesso a memória compartilhada serão destruídos. Dessa

forma, quando os filhos forem acessar os dados na memória não haverá essa estrutura de controle para evitar a ocorrência de race condition.

Pergunta 5: quais os valores possíveis de serem atribuídos a number?
Resposta: Os valores possíveis a serem atribuídos varia de 1 a 5.

3. ERROS DO PROGRAMA EXEMPLO

A seguir, os trechos contendo os erros e a nossa solução.

Trecho #1:

```
/*
 * Construindo a estrutura de controle do semaforo
 */
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;
```

Neste trecho temos duas situações problema: A primeira, na atribuição do "*g_sem_op1[0].sem_op*" como "-1" (sinal para trancar o semáforo), visto que desejamos que o op1 abra-o. A segunda, em logo após atribuirmos "g_sem_op1", reatribuirmos alterando seu valor. Nisso, não estamos inicializando a segunda estrutura de controle do semáforo (g_sem_op2), nem utilizando-a.

Solução:

```
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;
```

Definimos a estrutura de controle de maneira que a primeira seja para a abertura do semáforo e a segunda para o fechamento, isso nos permitirá corrigir erros futuros mais à frente do código.

Trecho #2:

```
/*
 * Criando o segmento de memoria compartilhada
 */
if((g_shm_id = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0000)) == -1) {
      fprintf(stderr,"Impossivel criar o segmento de memoria comparti-
lhada!\n");
      exit(1);
}
```

O erro se encontra na atribuição de permissão para os grupos (dono, grupos, outros), onde a permissão “0000”, é o mesmo que permissão “para nada” para nenhum grupo.

Solução:

```
if ((g_shm_id = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666)) == -1) {
```

Altera-se a permissão para 0666, garantindo direitos de leitura e escrita “rw” para qualquer entidade no sistema.

Trecho #3

```
/*
 * Criando os filhos
 */
rtn = 1;
for(count = 0; count < NO_OF_CHILDREN; count++) {
      if(rtn != 0) {
            pid[count] = rtn = fork();
      } else {
            exit
      }
}
```

Mesmo erro presente nos experimentos anteriores, em primeiro lugar, a função “void exit(int status)” está sendo invocada de maneira incorreta, e ainda

que fosse da maneira correta, não é o que desejamos, visto que ocasionará na terminação imediata de todo processo filho, e nós apenas queremos que o processos filho saia do laço de repetição.

Solução:

```
if (rtn != 0) {
      pid[count] = rtn = fork();
} else {
      break; // Corrigido exit() -> break
}
```

A solução dispensa maior explicação, apenas utilizamos um “break” para sairmos do laço de repetição quando cairmos no else com um processo filho.

Trecho #4

```
usleep(15000);

/*
 * Matando os filhos
 */
kill(pid[0], SIGKILL);
kill(pid[1], SIGKILL);
kill(pid[2], SIGKILL);
kill(pid[3], SIGKILL);
kill(pid[4], SIGKILL);
```

O tempo de sleep, conforme constatado em nossos testes, acaba não sendo suficiente para que todos os filhos executem devidamente ao menos uma vez; além disso, nesta primeira parte do experimento, temos apenas três filhos sendo produzidos pelo processo pai, o que faz com que o comando kill seja executado no vetor em posições de memória não alocadas para ele, podendo ocasionar algum eventual erro.

Solução:

```
// Quanto mais tempo o pai demora para matar os filhos, mais printa
usleep(30000);

/*
 * Matando os filhos
 */
kill(pid[0], SIGKILL);
kill(pid[1], SIGKILL);
kill(pid[2], SIGKILL);
```

Alteramos o valor para 30000 para garantir que todos os filhos executem antes que o pai os mate, e apaga-se as duas linhas extras “matando” filhos inexistentes.

Trecho #5

```
#ifdef PROTECT
      if(semop(g_sem_id, g_sem_op1, 1) == -1) {
            fprintf(stderr,"chamada semop() falhou, impossivel fe-
      char o recurso!");
            exit(1);
      }
#endif
```

Este trecho está logo antes a região crítica de código, onde haverá a manipulação do índice presente no segmento de memória compartilhada entre os processos filhos, isso significa que não podemos passar como parâmetro para a função “semop” a estrutura de controle 1 (g_sem_op1), que está definida para realizar a abertura do semáforo. Além de já estar aberto, o que devemos fazer é, na verdade, fechá-lo.

Solução:

```
if (semop(g_sem_id, g_sem_op2, 1) == -1) {
```

A solução, tendo em visto que desejamos fechar o semáforo para bloquear os outros processos, está em trocar de g_sem_op1 para g_sem_op2, que contém o sinal para o fechamento (sem_op = -1).

Trecho #6

```
/*
 * Repita o numero especificado de vezes, esteja certo de nao
 * ultrapassar os limites do vetor, o comando if garante isso
 */
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]);
            usleep(1);
      }
}
```

Erro semântico no segundo parâmetro de “fprintf”, a notação “%f7” não atende ao resultado desejado por nós.

Solução:

```
fprintf(stderr, "%c", g_letters_and_numbers[tmp_index + i]);
```

Apenas troca-se para “char” o tipo de saída que a função deve executar.

4. RESULTADOS

Abaixo, os resultados das execuções no terminal do sistema operacional Manjaro KDE Plasma 19.0.2 baseado em Arch Linux.

4.1. REFERENTES AO PROGRAMA EXEMPLO

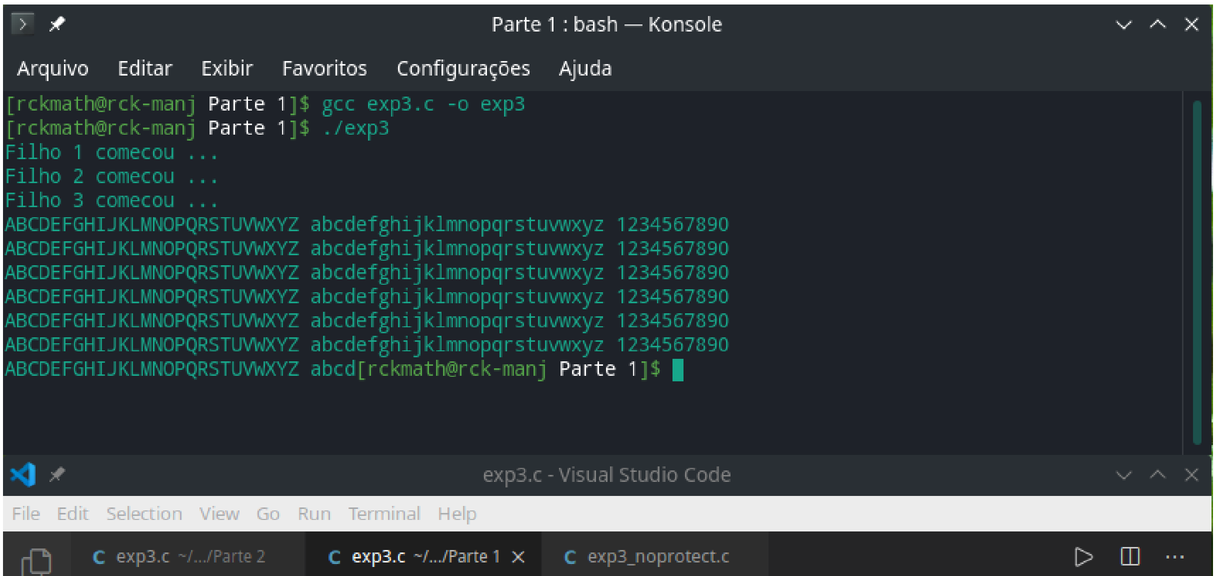
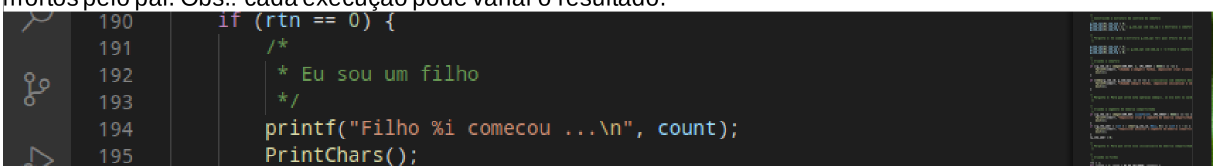


Figura 1 – Mostra que, com um tempo de 30000 µs, todos os processos filhos executam antes de serem mortos pelo pai. Obs.: cada execução pode variar o resultado.



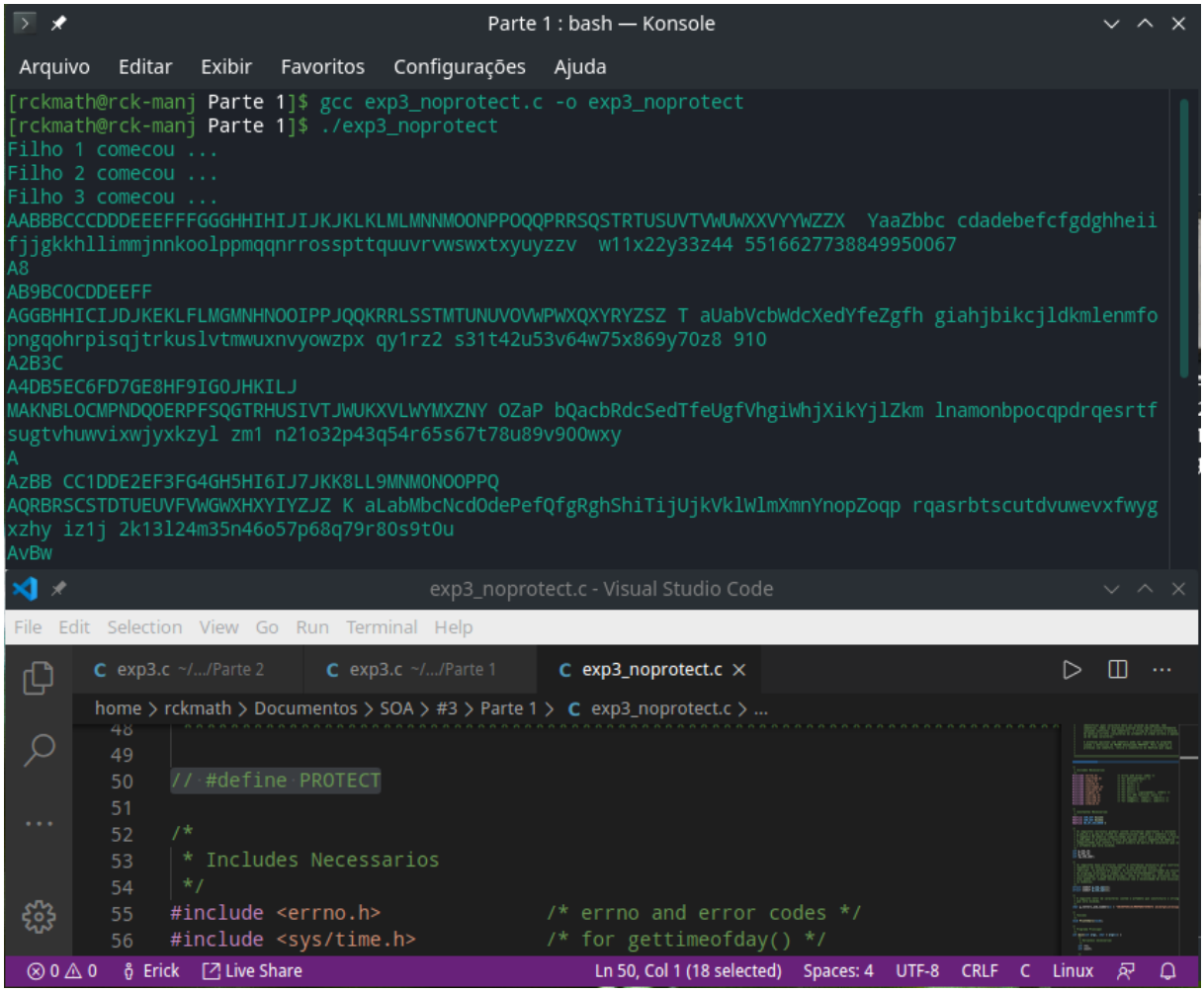


Figura 2 – Nos mostra como a saída se torna “bagunçada” quando removemos a proteção da Região Crítica com o uso de semáforos.

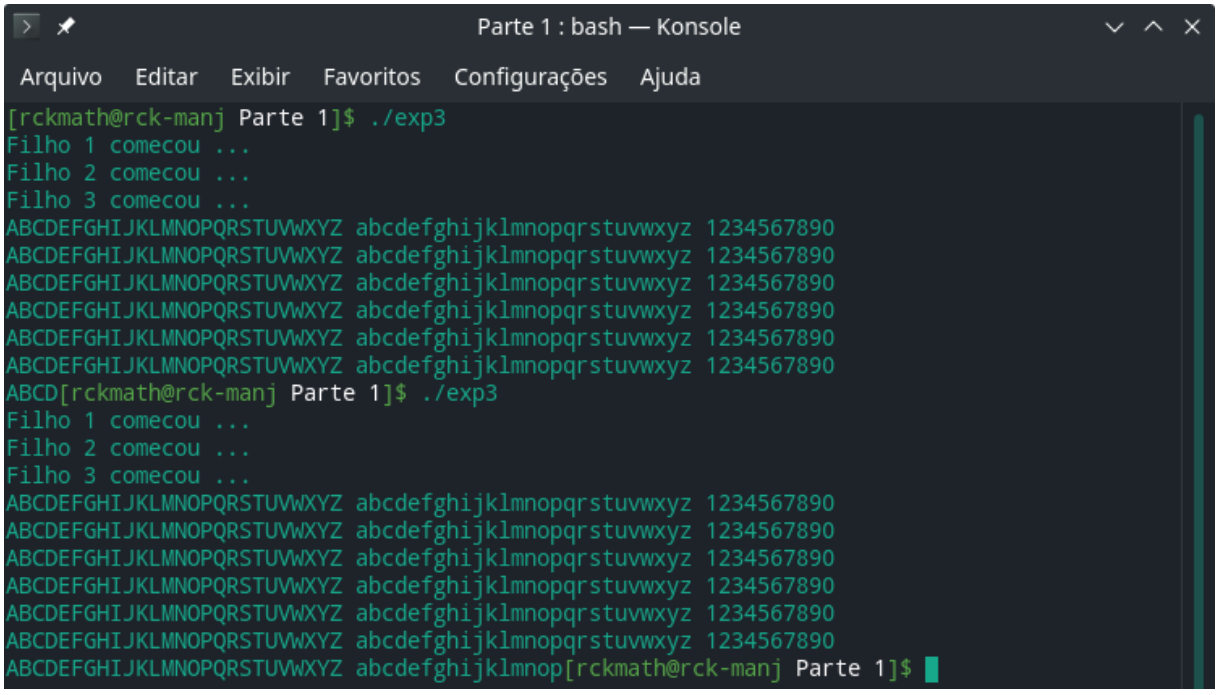


Figura 3 – Resultado com PROTECT mostrando a variação que podemos presenciar em cada execução. Na primeira execução.

4.2. REFERENTES AO PROGRAMA MODIFICADO

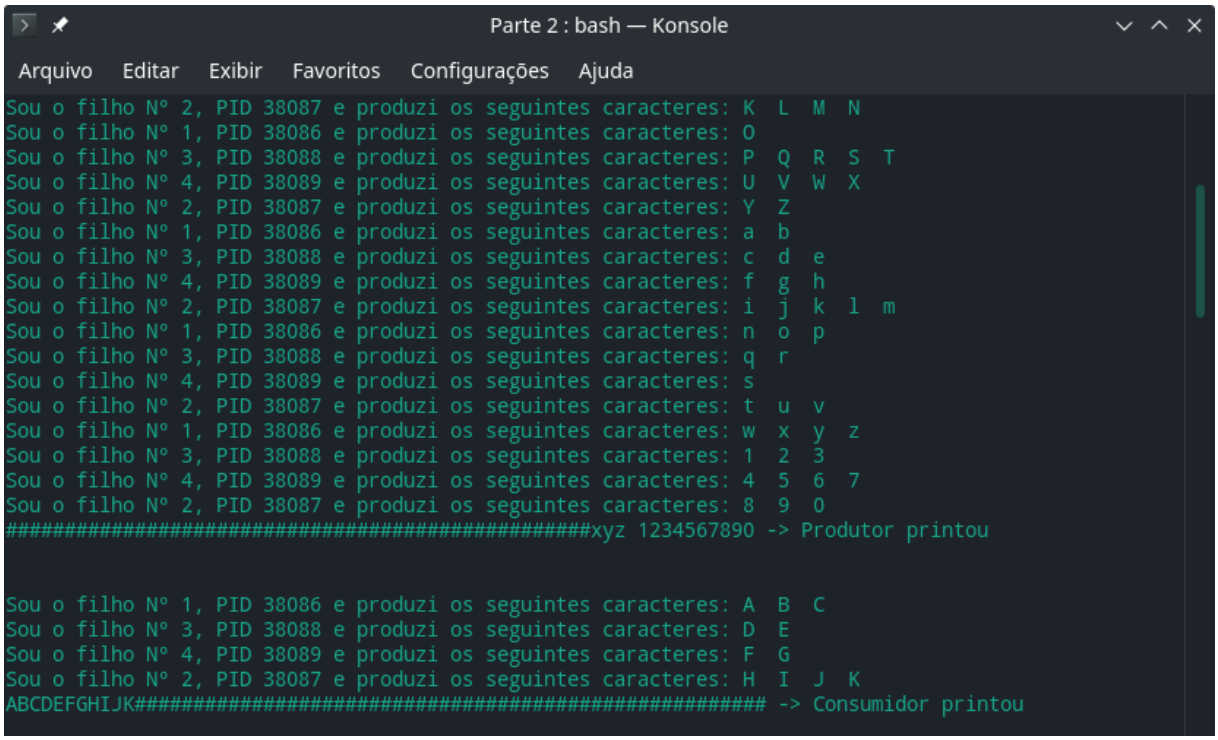


Figura 4 – Mostra o resultado do programa modificado, exibindo cada caractere inserido no buffer por cada filho, bem como o resultado do buffer após um produtor ou consumidor chegar ao final do índice.

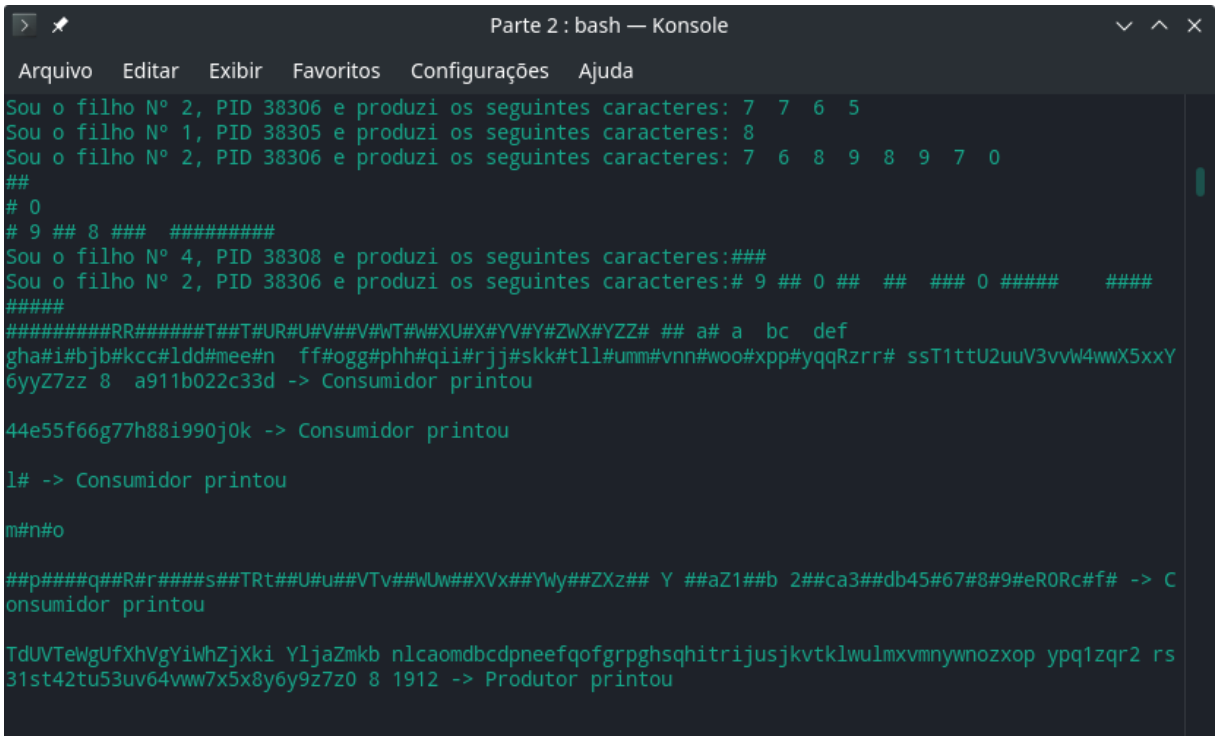


Figura 5 – Resultado ao retirarmos o controle exercido pelo semáforo, permitindo que todos os 8 filhos acessem o buffer e os índices ao mesmo tempo, apresentando um resultado completamente inconsistente.

5. ANÁLISE DOS RESULTADOS

5.1. PROGRAMA EXEMPLO

O programa exemplo demonstrou efetivamente a estrutura e o comportamento dos semáforos em um processo. Foi visto que com a presença dos semáforos a região crítica não é violada por outros processos evitando o problema de race condition. Tal quesito é demonstrado quando os filhos vão printar o vetor de char. Os testes realizados com a proteção do semáforo fizeram com que cada filho printasse corretamente as letras que a eles foram passadas. Porém, sem o uso dessa proteção gerou-se inconsistência nos dados o que consequentemente fez com que mais de 1 filho acessasse a mesma região crítica devido a uma imprevisibilidade nos dados. Com este acesso mútuo, no final da aplicação foi perceptível em tela uma printagem com diversos caracteres desconexos.

5.2. PROGRAMA MODIFICADO

No programa modificado realizou-se o compartilhamento de um segmento de memória dividindo-o em cinco partes, sendo elas da seguinte forma: índice para os filhos consumidores (4 bytes), índice para os filhos produtores (4 bytes), contador de total de caracteres produzidos (4 bytes), contador de total de caracteres consumidos (4 bytes) e o tamanho do buffer de carácter (65 bytes).

Utilizando semáforos para fazer o controle das regiões críticas, evitou-se o problema de race condition e erro no uso do índice global para os 4 filhos produtores e para os 4 filhos consumidores. Visando evitar o erro de quando o produtor estiver zerado, mas ainda havendo dados para serem consumidos, foi utilizado um contador auxiliar para que dessa forma o consumidor possa pegar os dados ou informações ainda presentes no buffer. Finalizado o Buffer tanto ao produtor quanto o consumidor foi aplicado testes com e sem proteção de semáforos, assim, os erros que ocorreram foram semelhantes ao da primeira tarefa, ressaltando a importância do uso de semáforo em processos com memória compartilhada.

6. CONCLUSÃO

A partir do experimento realizado foi possível entender sobre o funcionamento do compartilhamento de memória e assim identificar a necessidade

de um mecanismo de controle para o acesso aos dados. Dessa forma, teve-se como objetivo evitar a ocorrência de inconsistências de dados o que consequentemente gera uma imprevisibilidade nesses acessados. Para implementar esse mecanismo baseou-se na aplicação de semáforos para controlar o acesso dos filhos à memória compartilhada evitando concorrência de dados. Por fim, na parte 1 do experimento houveram problemas como inconsistência na memória compartilhada e também na passagem de operação dos semáforos (abrir/fechar). As soluções encontradas foram a alteração na permissão de acesso a memória (0666) e adição de outra variável para receber a segunda operação do semáforo(g_sem_op2). Na segunda parte enfrentamos o desafio de como estruturar e acessar a memória compartilhada de maneira correta e ao mesmo tempo garantir a consistência dos, a solução, além do uso do mesmo esquema de semáforo da parte 1, foi manipular o segmento através de um endereço base + offset, conhecendo anterior a posição (em bytes) de cada item presente na estrutura – os índices, os contadores e o próprio buffer.