



INF1316 - Laboratório 02 - Memória compartilhada

Nome:

- Guilherme Riechert Senko
- Pedro de Almeida Barizon

Matrícula:

- 2011478
- 2211350

Professor: Luiz Fernando Seibel

Turma: 3WA

Data: 09/09/2024

Objetivo: implementar e compreender processos com memória compartilhada em Unix, com foco nos conceitos de paralelismo, concorrência e troca de mensagens.

Observações preliminares

- Para mais detalhes, todos os códigos fonte comentados encontram-se em anexo. Por isso, para evitar redundâncias, não serão aqui exibidos.
- Todos os arquivos fonte foram compilados com auxílio do *GNU C Compiler* (GCC) fazendo `gcc -Wall -o programa fonte1.c fonte2.c...` e executados com `./programa arg1 arg2...`. Caso haja particularidades, estas serão abordadas ao longo do relatório.

Exercício 1) Criar um vetor `a` de 10.000 posições inicializado com valor 5.

Criar 10 processos trabalhadores que utilizam áreas diferentes do vetor para multiplicar a sua parcela do vetor por 2 e somar as posições do vetor retornando o resultado para um processo coordenador que irá apresentar a soma de todas as parcelas recebidas dos trabalhadores.

Obs: O 1º trabalhador irá atuar nas primeiras 1.000 posições, o 2º trabalhador nas 1.000 posições seguintes e assim sucessivamente.

1.1 Estrutura

Para que fosse enxuto, criou-se apenas um módulo, o `ex1.c`, que continha somente a função `main`, responsável pela execução do programa.

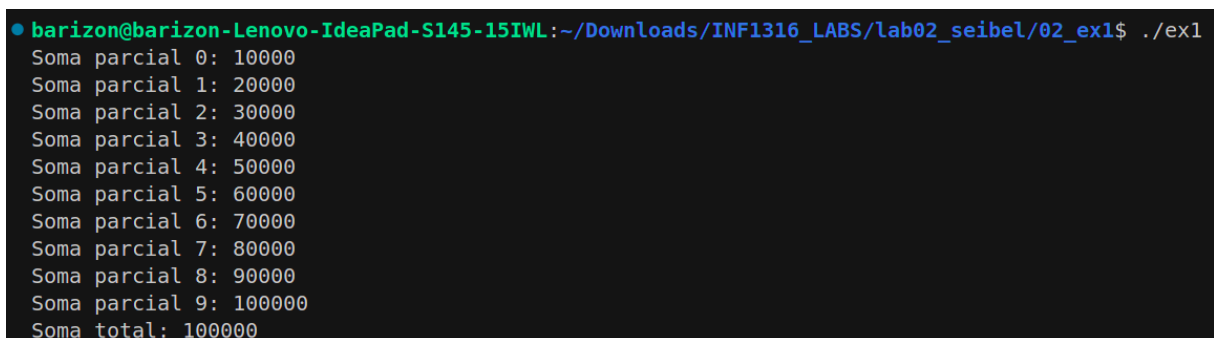
1.2 Solução

Utilizamos a função `shmget` para alocar um bloco de memória compartilhada suficiente para armazenar o vetor de 10.000 posições e mais uma posição extra para armazenar a soma total. A memória compartilhada permite que todos os processos (pai e filhos) acessem o mesmo espaço de memória. Antes de criar os processos filhos, o vetor é preenchido com o valor 5, e a posição extra, que armazenará a soma, é inicializada com 0.

Com um laço, criamos 10 processos filhos usando `fork()`. Cada processo filho é responsável por manipular uma parte específica do vetor (1.000 posições), multiplicando os valores por 2 e somando os resultados ao valor total armazenado na última posição da memória compartilhada. Após a criação de cada processo filho, o processo pai espera a conclusão do filho com `wait`. Durante o processo, o valor parcial da soma é impresso na tela a cada iteração.

Quando todos os processos filhos desanexam-se da memória compartilhada e terminam suas tarefas, o processo pai imprime a soma total acumulada no vetor. Em seguida, a *shared memory* é desanexada pelo pai com `shmdt` e liberada com `shmctl` atrelado à *flag* `IPC_RMID`.

A saída do programa encontra-se abaixo:



```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex1$ ./ex1
Soma parcial 0: 10000
Soma parcial 1: 20000
Soma parcial 2: 30000
Soma parcial 3: 40000
Soma parcial 4: 50000
Soma parcial 5: 60000
Soma parcial 6: 70000
Soma parcial 7: 80000
Soma parcial 8: 90000
Soma parcial 9: 100000
Soma total: 100000
```

Figura 1.1 - Saída de `ex.1`

1.3 Observações e conclusões

O uso de um processo coordenador — no caso, o pai de todos — atrelado ao uso de `wait` permitiu que os dez trabalhadores atuassem harmoniosamente, de modo que a saída do programa correu como o esperado, isto é, as somas parciais incrementaram-se em 10.000, que corresponde justamente a $(5 \times 2) \times 1.000$, o valor somado por cada trabalhador.

Exercício 2) Considere o vetor de 10.000 posições inicializado com o valor 5.

Crie 2 trabalhadores, ambos multiplicam por 2 e somam 2 em todas as posições do vetor. Verifique automaticamente se todas as posições têm valores iguais e explique o que ocorreu.

2.1 Estrutura

Para que fosse enxuto, criou-se apenas um módulo, o `ex2.c`, que continha somente a função `main`, responsável pela execução do programa.

2.2 Solução

Usamos memória compartilhada para garantir que ambos os trabalhadores (processos filhos) tenham acesso ao mesmo vetor de 10.000 posições. Utilizamos o comando `fork()` para criar dois processos filhos, que operam concorrentemente sobre o vetor. Cada trabalhador realiza, respectivamente, a operação de multiplicar por dois e somar dois para todas as posições do vetor. Como ambos os processos estão tentando modificar o vetor ao mesmo tempo (sem sincronização), uma dada posição pode ser multiplicada por dois por um dado processo e, antes de que este possa incrementá-la em dois, seja multiplicada pelo outro processo. Isso tenderá a produzir valores diferentes. Afinal,

$$(((5 \times 2) + 2) \times 2) + 2 = 26 \neq 24 = (((5 \times 2) \times 2) + 2 + 2)$$

Cabe ressaltar que, para aumentar a probabilidade de descoordenação, introduziu-se um *sleep* de 100 milissegundos entre as duas operações. Após a execução de ambos os trabalhadores, o processo pai verifica se todas as posições do vetor possuem o mesmo valor. Como visto, dependendo da concorrência entre

os processos, pode haver inconsistências nos valores, conforme mostra a saída do programa abaixo:

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex2$ ./ex2
# VALOR DE REFERENCIA: 24

> Na posicao 182, valor 26 difere
> Na posicao 183, valor 26 difere
> Na posicao 184, valor 26 difere
> Na posicao 185, valor 26 difere
> Na posicao 186, valor 26 difere
> Na posicao 187, valor 26 difere
> Na posicao 188, valor 26 difere
> Na posicao 189, valor 26 difere
> Na posicao 190, valor 26 difere
> Na posicao 191, valor 26 difere

! Para evitar saturacao do log, nao serao exibidos os proximos divergentes

* TOTAL DE DIVERGENCIAS: 9623
```

Figura 2.1 - Saída de ex. 2

2.3 Observações e conclusões

Como os dois processos estão manipulando o vetor ao mesmo tempo sem controle de acesso, pode haver inconsistências no resultado final (valores diferentes em diferentes posições do vetor). Para resolver esse problema, seria necessário usar algum tipo de mecanismo de sincronização, como semáforos, para evitar que os processos atropelassem as mudanças uns dos outros.

Em muitos casos, devido à condição de corrida, a verificação final provavelmente indicará que os valores no vetor não são iguais, a menos que a execução dos dois processos seja perfeitamente sequencial (o que é improvável em um ambiente de concorrência).

Exercício 3) Faça um programa que: Leia da entrada uma mensagem do dia. Crie uma memória compartilhada com a chave 7000. Salve a mensagem na memória compartilhada.

Faça um outro programa que utilize o mesmo valor de chave da memória compartilhada e dê attach na memória gerada pelo programa anterior. Em seguida este processo deve exibir a mensagem do dia para o usuário e deve liberar a memória compartilhada.

3.1 Estrutura

A solução consiste em dois programas em C que utilizam memória compartilhada para trocar informações entre processos distintos. A comunicação é realizada por meio de um segmento de memória compartilhada, identificado por uma chave específica (`KEY_ID = 7000`), em que o primeiro programa (`escritor.c`) grava uma mensagem, e o segundo programa (`leitor.c`) recupera essa mensagem para exibi-la.

Tanto o `escritor.c` quanto o `leitor.c` possuem cada qual apenas a rotina principal, isto é, a `main`, responsável pela execução do programa.

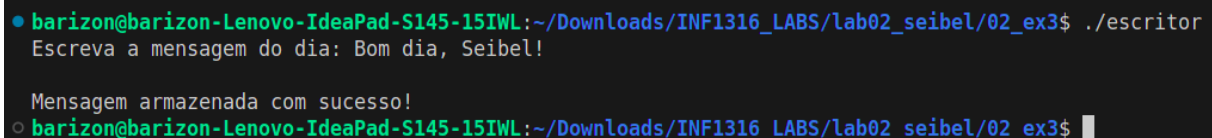
3.2 Solução

3.2.1 Fase de escrita

O `escritor.c` cria um segmento de memória compartilhada usando a função `shmget`, com uma chave identificadora (`KEY_ID = 7000`) e um tamanho de 4096 *bytes*. Essa memória é alocada com permissões para leitura, escrita e execução apenas para o dono do processo.

Após a criação, o segmento é anexado ao processo por meio da função `shmat`, o que permite ao programa manipular os dados da memória como se fossem parte do espaço de endereçamento local. O programa, então, solicita ao usuário que insira uma mensagem, que é armazenada diretamente no segmento de memória compartilhada.

Ao final, o segmento de memória é desanexado do processo com `shmdt`, mas a memória continua alocada até ser explicitamente removida. A saída encontra-se abaixo:

A terminal window with a dark background and light-colored text. The prompt is 'barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex3\$'. The user enters './escritor'. The program outputs 'Escreva a mensagem do dia: Bom dia, Seibel!'. The user enters a blank line. The program outputs 'Mensagem armazenada com sucesso!'. The prompt returns.

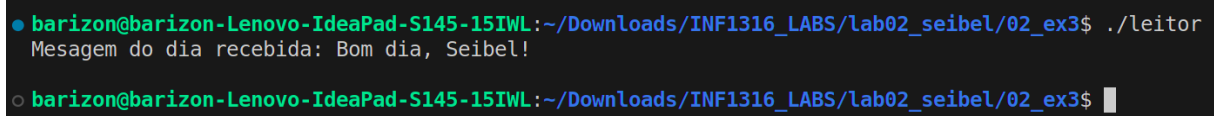
```
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex3$ ./escritor
Escreva a mensagem do dia: Bom dia, Seibel!

Mensagem armazenada com sucesso!
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex3$
```

Figura 3.1 - Saída de `escritor.c`

3.2.2 Fase de leitura

O `leitor.c` acessa o segmento de memória compartilhada criado anteriormente, utilizando a mesma chave identificadora (`KEY_ID = 7000`). Como o segmento já existe, o programa o recupera com `shmget` sem passar `IPC_CREAT` e anexa a memória com `shmat`. A mensagem armazenada na memória é lida e exibida na saída padrão. Por fim, o programa desanexa o segmento com `shmdt` e remove a memória do sistema com `shmctl` ao passar `IPC_RMID`, para liberar os recursos. A saída encontra-se abaixo:



```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex3$ ./leitor
Mensagem do dia recebida: Bom dia, Seibel!
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Downloads/INF1316_LABS/lab02_seibel/02_ex3$
```

Figura 3.2 - Saída de `leitor.c`

3.3 Observações e conclusões

A memória compartilhada criada pelo `escritor.c` persiste no sistema mesmo após o término do processo que a criou. Isso permite que outros processos (como o `leitor.c`) acessem a mesma área de memória posteriormente. Essa característica facilita a comunicação entre processos não simultâneos, mas requer que a memória seja explicitamente removida para evitar desperdício de recursos.

Embora este exemplo seja simples e não envolva acessos simultâneos, em sistemas mais complexos que utilizam memória compartilhada, pode ser necessário o uso de mecanismos de sincronização (como semáforos), para evitar condições de corrida. Neste caso, como um programa grava e o outro lê de forma independente, a sincronização não foi necessária.

4 Conclusões finais

Os programas foram compilados sem erro usando-se o GCC, o ambiente de desenvolvimento do Visual Studio Code e o sistema operacional Ubuntu. Além disso, conforme se observa nas saídas dos programas, todos os testes obtiveram êxito. Por fim, devemos reconhecer a importância deste laboratório, uma vez que nos permitiu vislumbrar o grande poder do compartilhamento de memória entre processos, mas igualmente a responsabilidade. Afinal, como nos mostram as condições de corrida, quando vários processos manipulam uma mesma área

descoordenadamente, o resultado será imprevisível, o que pode ser extremamente prejudicial.