INF1316 - Laboratório 06 - Semáforos em sistemas Unix

Nome:

- Guilherme Riechert Senko
- Pedro de Almeida Barizon

Matrícula:

- 2011478
- 2211350

Professor: Luiz Fernando Seibel

Turma: 3WA

Data: 27/10/2024

Objetivo: implementar e compreender semáforos entre processos em sistemas Unix com auxílio da biblioteca padrão POSIX `semaphore.h`.

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 … fonten.c` e executados com `./programa arg1 arg2 … argn`. Caso haja particularidades, estas serão abordadas ao longo do relatório.
- Por uma questão de acessibilidade e de possível uso em portfólio pessoal, os códigos fonte foram comentados em inglês.

Exercício 1.a) Faça um programa que utilize a memória compartilhada para trocar 128 mensagens entre 2 processos/programas independentes. Use semáforos para sincronizar a aplicação da seguinte forma:

a) Processos síncronos:
Processo 1 escreve mensagem na memória compartilhada, processo 2 lê a mensagem da memória compartilhada e informa ao processo 1 que pode enviar nova mensagem. As mensagens são strings de, no máximo, 15 caracteres e têm o seguinte formato:
"mensagem 1",
"mensagem 2", etc.
As mensagens são tratadas uma de cada vez.

1.a.1 Estrutura

Criou-se apenas um módulo, `ex1.c`, que continha as funções:

- **`main`**: executa o programa.

- **`error`**: exibe mensagem de erro e encerra com `exit(EXIT_FAILURE)`.

- **`sem_open_wrapper`**: *wrapper* da função `sem_open`, que aloca um semáforo e inicializa-o com um valor dado como parâmetro.

- **`shm_alloc_at`**: *wrapper* das funções `shmget` e `shmat`, que alocam e anexam memória compartilhada, respectivamente.

- **`writing_finished`**: *handler* de `SIGINT` que escreve uma última mensagem, desconecta-se dos recursos alocados e encerra com `exit(EXIT_SUCCESS)`.

- **`dettach_resources`**: desanexa processo atual de todos os recursos alocados.

- **`free_resources`**: libera todos os recursos alocados na memória.

1.a.2 Solução

Inicialmente, alocam-se um segmento de memória compartilhada e dois *mutexes* `has_read` e `has_written` com o auxílio de `sem_open_wrapper`, sendo estes inicializados, respectivamente, com 1 e com 0. `has_read`, quando aberto (= 1), indica que o processo leitor já realizou sua leitura, o que autoriza a escrita de uma nova mensagem pelo escritor. Similarmente, `has_written`, quando aberto (= 1), indica que o escritor já realizou sua escrita, o que autoriza a leitura de uma nova mensagem pelo leitor. Como se deseja que o escritor, chamado no código de `sender`, escreva primeiro, e apenas depois o `receiver` a leia, deve ser que `has_written` comece em 0 e `has_read` comece em 1, conforme descrito acima. Feito isso, o processo inicial cria `receiver` e `sender` com `fork` e por estes espera com `wait`.

Em um laço com 128 repetições, `sender`, antes de escrever, aciona `sem_wait(has_read)` (*down*), isto é, espera até que `receiver` tenha lido. Em seguida, escreve sua mensagem na memória compartilhada com o auxílio de `sprintf`. Por fim, indica ter terminado a escrita com `sem_post(has_written)` (*up*). Na última mensagem, emite via `kill` um `SIGINT`, a fim de avisar o `receiver` que o conteúdo enviado é o derradeiro. Por fim, desaloca seus recursos e encerra.

Já `receiver` fica em um *loop* eterno com `sem_wait(has_written)`, `printf` do conteúdo lido da memória compartilhada, e `sem_post(has_read)`. Quando recebe `SIGINT`, aciona `writing_finished`, que faz escrever a última mensagem e encerra o processo.

Como nenhum dos dois processos recebe qualquer `SIGSTOP` ou equivalente e como ambos acessam a mesma memória compartilhada, garante-se aí a concorrência. Por outro lado, o uso dos dois *mutexes* garante a sincronicidade.

Com os dois filhos terminados, o pai encerra a `main` ao retornar 1. A saída do programa encontra-se abaixo:

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab06_seibel/06ex1$ ./ex1
mensagem 1
mensagem 2
mensagem 3
mensagem 4
mensagem 5
mensagem 6
mensagem 7
mensagem 8
mensagem 9
mensagem 10
mensagem 11
mensagem 12
mensagem 13
mensagem 14
mensagem 15
mensagem 16
mensagem 17
mensagem 18
mensagem 19
mensagem 20
mensagem 21
mensagem 22
mensagem 23
mensagem 24
mensagem 25
mensagem 26
mensagem 27
mensagem 28
mensagem 29
mensagem 30
mensagem 31
mensagem 32
mensagem 33
mensagem 34
mensagem 35
mensagem 36
mensagem 37
mensagem 38
mensagem 39
mensagem 40
mensagem 41
mensagem 42
mensagem 43
mensagem 44
mensagem 45
mensagem 46
mensagem 47
mensagem 48
mensagem 49
mensagem 50
mensagem 51
mensagem 52
mensagem 53
mensagem 54
mensagem 55
mensagem 56
mensagem 57
mensagem 58
mensagem 59
mensagem 60
mensagem 61
```

Figura 1.a.1 - Saída de `ex1` até a mensagem 61

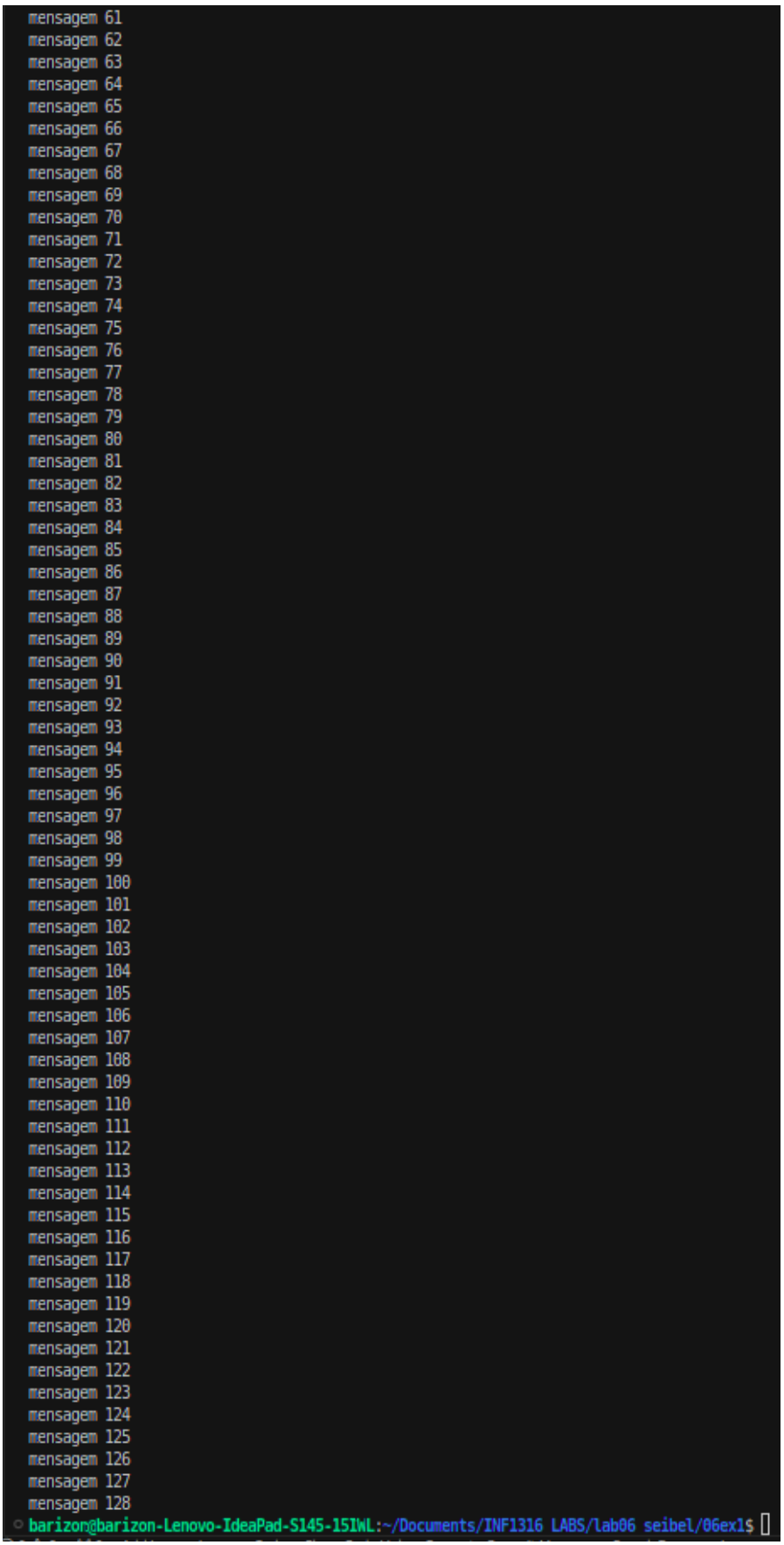

Figura 1.a.2 - Saída de `ex1` da mensagem 61 até a 128

1.a.3 Observações e conclusões

Conforme visível na saída, o programa obteve êxito, uma vez que todas as 128 mensagens foram exibidas e na ordem adequada, prova cabal da devida sincronização dos processos escritor e leitor.

Cabe mencionar a semelhança deste problema com aquele do *consumidor e do produtor*: o leitor pode ser interpretado como o *consumidor*; e o escritor, como o *produtor*. Neste caso, porém, o limite de itens produzidos mantém-se em 1, o que permitiu o uso de *mutexes* e tornou a implementação ainda mais simples.

Por fim, devemos ressaltar a facilidade de uso da biblioteca `semaphore.h` em detrimento da `sys/sem.h`, cuja interface mais complexa a torna difícil de lidar.

Exercício 1.b) **Faça um programa que utilize a memória compartilhada para trocar 128 mensagens entre 2 processos/programas independentes. Use semáforos para sincronizar a aplicação da seguinte forma:**

b) Processos assíncronos:
Processo 1 escreve as mensagens para serem tratadas pelo processo 2. O buffer de comunicação entre os processos, que fica na memória compartilhada, é capaz de armazenar apenas 8 mensagens. As mensagens são strings e são exibidas a fim de mostrar o que está ocorrendo na aplicação. Deve ser informado ainda que a mensagem foi enviada pelo processo x e que foi recebida pelo processo y. Os processos devem, obrigatoriamente, ser concorrentes. As mensagens são strings de, no máximo, 15 caracteres e têm o seguinte formato:
"mensagem 1",
"mensagem 2", etc.

1.b.1 Estrutura

Criou-se apenas um módulo, `ex2.c`, que continha funções análogas às do módulo `ex1.c` — salvo `writing_finished`, cujo uso foi dispensado. Por concisão, não serão aqui repetidas.

1.b.2 Solução

O funcionamento e a nomenclatura são idênticos ao do módulo anterior, com a exceção de que agora não se usam dois *mutexes*, mas sim dois semáforos cujos valores máximos correspondiam ao número máximo de mensagens armazenáveis no *buffer* de comunicação, definido como 8. Assim `has_written` novamente começava em 0 (*buffer* vazio), porém `has_read` começava em 8 (sender ainda pode escrever oito mensagens).

À medida que escrevia, `sender` fazia *down* em `has_read` (espera `receiver` ter lido ao menos uma mensagem) e *up* em `has_written` (avisa ter escrito nova mensagem), ao passo que `receiver` fazia *down* em `has_written` (espera ao menos uma nova mensagem ter sido escrita) e *up* em `has_read` (avisa ter lido uma mensagem).

Para garantir coordenação de leitura e escrita, ambos os processos percorrem ciclicamente o *buffer* com saltos regulares de 20 *bytes*, o que evita sobreposição de escrita, uma vez que o tamanho máximo de uma mensagem é 16

bytes (15 caracteres + `'\0'`). A saída do programa foi idêntica à do primeiro. Por concisão, não será repetida.

Por fim, como apenas `sender` escrevia na memória compartilhada e como somente `receiver` exibia na console, o programa informa que todas as mensagens foram mandadas de `sender` a `receiver`.

1.b.3 Observações e conclusões

Conforme visível na saída, o programa obteve êxito. Tratou-se, nada mais nada menos, de uma generalização do primeiro exercício, isto é, do caso geral *produtor-consumidor*, cuja sincronização pode ser feita com dois semáforos, consoante descrito acima.

2 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. Ademais, conforme se observa na saída do programa, todos os testes obtiveram êxito. Por fim, devemos reconhecer a importância deste laboratório, uma vez que nos permitiu vislumbrar a relevância dos semáforos na sincronização de processos concorrentes que acessam uma mesma região crítica.