

EA872 Laboratório de Programação de Software Básico
Atividade 10



Vinícius Esperança Mantovani

RA 247395

Entrega (limite): 01/11/2023, 13:00.

Exercício 1)

1.1) Analisando o programa e suas saídas, percebe-se que sua ordem de execução está descoordenada, de maneira que, por vezes a thread principal passa à frente de algumas threads-filhas, conforme se nota a seguir:

Unset

Thread 140158272583488. Criei a thread 140158272579264 na iteracao 0 ...
Thread 140158272579264 - Criada na iteracao 0.

Thread 140158272583488. Criei a thread 140158264055488 na iteracao 1 ...

Thread 140158272583488. Criei a thread 140158255662784 na iteracao 2 ...
Thread 140158264055488 - Criada na iteracao 1.
Thread 140158255662784 - Criada na iteracao 2.

Este é o texto de saída para o caso de termos 3 threads sendo criadas. Nele, podemos notar que a terceira mensagem impressa pela thread principal acabou sendo impressa anteriormente à segunda mensagem impressa por threads-filhas (pela segunda thread filha, neste caso). Este caso pode servir para a análise de casos com menos de 7 threads.

Essa desordem ocorre por conta de existirem diferentes threads imprimindo mensagens indiscriminadamente, ou seja, a primeira thread (thread-mãe) não espera adequadamente pela impressão feita por uma thread-filha, logo, temos uma inversão na ordem de impressão.

Analizando um exemplo de saída para um caso com 10 threads, temos o seguinte:



Unset

Thread 140553142986560. Criei a thread 140553142982336 na iteracao 0 ...
Thread 140553142982336 - Criada na iteracao 0.

Thread 140553142986560. Criei a thread 140553134458560 na iteracao 1 ...

Thread 140553142986560. Criei a thread 140553126065856 na iteracao 2 ...
Thread 140553134458560 - Criada na iteracao 1.

Thread 140553142986560. Criei a thread 140553117673152 na iteracao 3 ...
Thread 140553117673152 - Criada na iteracao 3.
Thread 140553126065856 - Criada na iteracao 2.

Thread 140553142986560. Criei a thread 140553109280448 na iteracao 4 ...
Thread 140553109280448 - Criada na iteracao 4.

Thread 140553142986560. Criei a thread 140553100887744 na iteracao 5 ...
Thread 140553100887744 - Criada na iteracao 5.

Thread 140553142986560. Criei a thread 140553092495040 na iteracao 6 ...
Thread 140553092495040 - Criada na iteracao 6.

Thread 140553142986560. Criei a thread 140553084102336 na iteracao 7 ...
Thread 140553084102336 - Criada na iteracao 7.

Thread 140553142986560. Criei a thread 140553006085824 na iteracao 8 ...
Thread 140553006085824 - Criada na iteracao 8.

Thread 140553142986560. Criei a thread 140552997693120 na iteracao 9 ...
Thread 140552997693120 - Criada na iteracao 9.



Neste caso, notamos, além da inversão entre mensagens da mãe e uma thread-filha, ocorre também, inversões de mensagens impressas por diferentes threads-filhas, como se percebe entre as mensagens:

Thread 140553117673152 - Criada na iteracao 3.

Thread 140553126065856 - Criada na iteracao 2.

em que se pode observar uma inversão entre uma thread criada na 2º iteração e na 3º iteração.

Em suma, vemos que, entre threads, há uma desordem na execução do programa, no sentido de que elas não estão devidamente sincronizadas de forma a permitir a correta sequência de impressão.

1.2) Analisemos a fundo o programa. De início, a thread-mãe inicia a execução do programa na “main” declarando variáveis importantes para a lógica de funcionamento deste programa. Em seguida, temos um bloco “if” responsável por imprimir uma mensagem de explicação sobre o modo de uso do programa para o caso em que o usuário fez uso incorreto. Seguindo, entramos no bloco “for” do programa, responsável por loopar de “n” vezes, sendo “n” o valor passado pelo usuário como argumento na chamada do programa.

No interior do loop “for”, logo de início, é criada uma nova thread por meio da chamada “pthread_create”, com argumento passados sendo: um objeto da lista de objetos do tipo “pthread_t”



declarada no início da “main”; um “NULL” no lugar de um possível ponteiro de objeto de atributos de thread; a rotina de início da thread, neste caso “PrintHello” e; o valor de “i” (índice das iterações do loop “for”) com devido “cast” feito. Assim, a thread criada começará a executar o código chamando a função “PrintHello”, printando, portanto, uma mensagem avisando que foi criada na iteração “i” e, sendo, imediatamente encerrada pela chamada “pthread_exit(NULL)”. Simultaneamente a isso, a thread-mãe continua sua execução e, no caso de erro na chamada de criação de thread, imprime uma mensagem de erro e encerra o programa, caso contrário, imprime uma mensagem afirmando que criou uma thread, passando seu id e o id da thread criada.

Por fim, assim termina o bloco “for” e, é feita uma chamada “pthread_exit(NULL)” para encerrar a thread-mãe.

1.3) A função “PrintHello” é responsável por imprimir uma mensagem de criação de thread e encerrar a thread que a chamou. Porém, para além disso, ela serve como rotina de início da thread criada na linha “rc = pthread_create(&thread_id[i], NULL, PrintHello, (void*)(intptr_t)i);”. Ou seja, ela tem como função ser a rotina pela qual as threads criadas começarão a execução do código.

1.4) Como o sleep atua apenas em valores de “i” que sejam múltiplos de três, o efeito é mais destacado próximo das mensagens mostradas pelas threads associadas a tais valores. Desse modo, podemos ver, na prática, a diferença entre casos com e sem a linha destacada:

Com a linha:

Unset

```
Thread 140460036929344. Criei a thread 140460036925120 na iteracao 0 ...  
Thread 140460036925120 - Criada na iteracao 0.  
  
Thread 140460036929344. Criei a thread 140460028401344 na iteracao 1 ...  
Thread 140460028401344 - Criada na iteracao 1.
```

Sem a linha:

Unset

```
Thread 140121750337344. Criei a thread 140121750333120 na iteracao 0 ...  
  
Thread 140121750337344. Criei a thread 140121741940416 na iteracao 1 ...  
Thread 140121750333120 - Criada na iteracao 0.  
Thread 140121741940416 - Criada na iteracao 1.
```

Neste caso, temos um exemplo com duas threads, no qual podemos notar que, já na segunda thread temos uma diferença entre os casos com e sem a linha “sleep”. Isso porque, sem a espera para múltiplos de três, não ocorre a espera no caso de “i=0”, logo, a thread-filha 0 não tem o tempo devido para escrever a mensagem que deveria, sendo, portanto, escrita a mensagem da thread-mãe a respeito da nova thread-filha 1 antes da mensagem impressa pela thread-filha 0.

Em suma, é possível notar que, embora já estivesse sendo desordenada a sequência de impressões, com a retirada da linha destacada, temos uma desordem ainda maior.

Exercício 2)


2.1) Inicialmente, analisemos casos de saída para 2 e 10 threads conforme os textos abaixo:

Para o caso com 2 threads, temos o seguinte de resposta no terminal:

```
Unset
Main: criando thread 0
Main: criando thread 1
Comecando thread 0 ...
Comecando thread 1 ...
Thread 1 terminada. Resultado = 1.074530e+15
Thread 0 terminada. Resultado = 1.072815e+15
Main: completou join com thread 0 com status 0
Main: completou join com thread 1 com status 1
Programa ./prog2 terminado
```

Já de começo, notamos a impressão de duas mensagens de dentro da “main”, anunciando a criação das duas threads. Isso se explica pelo bloco de código:


```
for(t = 0; t < n; t++) {
    printf("Main: criando thread %d\n", t);
    rc = pthread_create(&thread[t], &attr, WorkerThread, (void *) (intptr_t)t);
    if (rc) {
        printf("\n ERRO: codigo de retorno de pthread_create eh %d \n", rc);
        exit(1);
    }
}
```



Neste trecho, temos um loop como o número de threads passado como argumento na chamada do programa e, temos uma linha “rc = pthread_creat(...);” em que são criadas, para cada iteração, uma thread, sendo impresso, antes disso, pela chamada “printf” da linha anterior, a mensagem que vemos no terminal para cada uma das threads criadas.

Seguindo, na saída amostrada no terminal, temos duas mensagens anunciando o início de cada uma das threads. Isso se dá, por conta do início de cada uma das threads, criadas na “main” pelo bloco exposto acima, ser feito partindo da função “WorkerThread” por ser esta a função passada como parâmetro de rotina de início na chamada “pthread_creat”. Isso porque, essa função converte o argumento “t” passado para um “intptr_t”, sendo, em seguida, este valor impresso junto da mensagem de aviso de começo de cada uma de suas respectivas threads. Assim ocorre a impressão das linhas:

```
Comecando thread 0 ...
Comecando thread 1 ...
```



Em seguida, temos mais duas mensagens:

```
Thread 1 terminada. Resultado = 1.074530e+15
Thread 0 terminada. Resultado = 1.072815e+15
```

as quais são também impressas pela função “WorkerThread” imediatamente antes do fim de cada uma das threads e anunciam esses encerramentos.

Por fim, a “main” anuncia a completude de cada uma das threads criadas por ela, por meio das mensagens:

```
Main: completou join com thread 0 com status 0
```

```
Main: completou join com thread 1 com status 1
```

afirmando, ainda nestas mensagens, que completou uma chamada “pthread_join” para cada uma das duas threads, o que significa que ela aguardou o término de cada uma para poder imprimir estas mensagens finais respectivas a cada uma. Finalmente, temos ainda uma mensagem impressa pela “main” anunciando o encerramento do programa imediatamente antes de ele ser efetivamente encerrado pela chamada “pthread_exit(NULL)”.

Agora, analisemos o texto de saída do caso com 10 threads a seguir:

Unset

```
Main: criando thread 0
Main: criando thread 1
Main: criando thread 2
Main: criando thread 3
Comecando thread 0 ...
Comecando thread 1 ...
Main: criando thread 4
Comecando thread 3 ...
Comecando thread 2 ...
Main: criando thread 5
Comecando thread 4 ...
Main: criando thread 6
Comecando thread 5 ...
Comecando thread 6 ...
Main: criando thread 7
Main: criando thread 8
Main: criando thread 9
Comecando thread 7 ...
Comecando thread 8 ...
Comecando thread 9 ...
Thread 3 terminada. Resultado = 1.073815e+15
Thread 2 terminada. Resultado = 1.073454e+15
Thread 1 terminada. Resultado = 1.074106e+15
Thread 0 terminada. Resultado = 1.073427e+15
Main: completou join com thread 0 com status 0
Main: completou join com thread 1 com status 1
Main: completou join com thread 2 com status 2
Main: completou join com thread 3 com status 3
Thread 6 terminada. Resultado = 1.073865e+15
Thread 5 terminada. Resultado = 1.073133e+15
Thread 9 terminada. Resultado = 1.073054e+15
Thread 8 terminada. Resultado = 1.073451e+15
Thread 7 terminada. Resultado = 1.074801e+15
Thread 4 terminada. Resultado = 1.075032e+15
```



```
Main: completou join com thread 4 com status 4
Main: completou join com thread 5 com status 5
Main: completou join com thread 6 com status 6
Main: completou join com thread 7 com status 7
Main: completou join com thread 8 com status 8
Main: completou join com thread 9 com status 9
Programa ./prog2 terminado
```

Aqui, tomando como base a análise feita para o caso com duas threads, podemos compreender que, embora as mensagens impressas pela “main” sejam perfeitamente ordenadas entre si, aquelas que são impressas por outras threads, que não a mãe, podem não ser ordenadas em relação às de outras threads, sejam estas outras a thread-mãe ou mesmo alguma das outras threads-filhas. Isso ocorre porque, antes de uma thread-filha ser iniciada e sua mensagem de começo, impressa pela rotina de início “WorkerThread”, ser impressa, as mensagens de outras threads, sejam elas a thread-mãe ou não, podem ser impressas sem qualquer sincronização, como se nota a seguir:

```
Comecando thread 3 ...
Comecando thread 2 ...
(thread 3 “comecando” antes da 2);

Main: criando thread 5
Comecando thread 4 ...
(thread 4 “comecando” depois da “main” criar a thread 5);

Thread 3 terminada. Resultado = 1.073815e+15
Thread 2 terminada. Resultado = 1.073454e+15
(thread 3 “terminando” antes da thread 2);
```

Apesar disso, nota-se que as mensagens de completude de “join” com as mais diversas threads, impressas pela “main”, sucedem as respectivas mensagens de término das respectivas threads, impressas por estas próprias threads. Ou seja, uma mensagem impressa pela “main” anunciando a completude de “join” com uma thread de número “k” nunca será sucedida pela mensagem de término desta thread “k”, mas sempre antecedida por ela. Isso ocorre, pois a “main” faz uma chamada de “pthread_join” na “main” para cada uma das threads-filhas, o que impõe que a mãe aguarde o término de uma thread-filha para continuar seu fluxo, sendo, portanto, necessário que a thread-filha acabe para que a mãe possa continuar o fluxo de execução e começar o “join” com a próxima thread ou, caso não haja mais threads-filhas, seguir para a impressão da mensagem de término do programa.

Assim, temos, por fim, que a mensagem de término do programa é **SEMPRE** a última a ser impressa!

2.2) A chamada “pthread_join” ocasiona a espera da “main” (thread-mãe) pelo encerramento das demais threads. Desse modo, a thread-mãe é suspensa até que a thread alvo da chamada “pthread_join” seja encerrada, retornando à atividade apenas quando este encerramento ocorre. Deste modo, no caso do programa que temos, ela é responsável por sincronizar a mensagem de término impressa por uma thread-filha com a mensagem de completude da sincronização de “join” impressa

pela “main” da thread-mãe, sendo a mensagem da thread-filha impressa anteriormente à da thread-mãe.

Quanto à chamada “pthread_attr_init”, temos como função inicializar um objeto de atributos de thread. Portanto, é responsável por permitir que um objeto de atributos de thread tenha seus valores alterados e manipulados. Sendo assim, no programa a nós atribuído, tem como finalidade permitir que seja, na sequência, chamada “pthread_attr_setdetachstate” alterando o atributo do objeto de atributos de thread que se incumbe de permitir ou não sincronização por “join”, permitindo, desse modo, que seja posteriormente definida a possibilidade de sincronização por “join” que será feitas várias vezes na main da thread-mãe.

Seguindo, a chamada “pthread_attr_setdetachstate” tem como função alterar o atributo de estado de “detach” do objeto de atributos de threads. Deste modo, tem como finalidade permitir ou não a sincronização por “join” da thread criada e, no caso do programa que estamos analisando, a chamada em questão é responsável por permitir que façamos chamadas de “pthread_join” na sequência da thread-mãe, sendo as threads criadas com esse estado de “joinable” as threads alvo dessas chamadas. Assim, podemos sincronizar como se deseja no programa, a thread-mãe às threads-filhas para a impressão ordenada das mensagens necessárias.

Por fim, temos a chamada “pthread_attr_destroy” que finaliza (destrói) um objeto de atributos de thread, tendo, portanto, como função, impedir que este objeto seja utilizado para a criação de novas threads na sequência do código. Assim, é responsável, tanto neste programa quanto em qualquer outro, por impedir o uso do objeto destruído por threads que sejam criadas após a sua destruição, mostrando-nos, ainda, que as threads já criadas não são afetadas por essa destruição.

2.3) Essa diferença na ordem de término de threads com relação à ordem de sua criação se dá por conta, tanto (principalmente) da operação feita por elas como por conta de oscilações mínimas nos tempos de criação, início de execução e finalização de diferentes threads. Isso porque, como as operações feitas por elas são dadas por números aleatórios (doubles geradas aleatoriamente) e o número de iterações é extremamente grande, temos uma diferença considerável entre os tempos de execução de uma e outra thread. Deste modo, uma thread que foi iniciada depois de outra pode ter um conjunto de operações (neste caso, de soma) cuja complexidade seja menor ou suficiente comparada à complexidade da thread criada antes, de modo que essa thread iniciada posteriormente tenha um tempo necessário para execução menor que a iniciada anteriormente. Isso ocasiona que a thread criada depois seja finalizada antes.

Exercício 3)

3.1) O que ocorre se retiramos a chamada “pthread_join” da “main” é que o programa é encerrado quase que instantaneamente após seu início. Isso ocorre, porque ela é a chamada responsável por fazer com que a thread-mãe aguarde pelo término da thread-produtora até que esta seja encerrada para que a thread-mãe possa continuar sua execução. Deste modo, se tiramos a chamada “pthread_join”, ocorre que a thread-mãe, que executa a “main”, não aguarda o processamento das demais threads, encerrando, assim, o programa como um todo muito rapidamente.

3.2) Se tiramos as delimitações de regiões críticas e as sincronizações entre produtor e consumidor, notamos um comportamento esquisito do programa, contendo itens de índice supostamente negativos sendo removidos e inseridos no buffer do programa, conforme se nota na saída do terminal em um certo tempo de execução do programa:

Unset

Consumidor iniciando...

Produtor iniciando...

Nº de itens a tratar no consumidor =4

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item -1 removido (valor=0)

Nº de itens a tratar no produtor =7

item -1 inserido (valor=77)

item 0 inserido (valor=15)

item 0 removido (valor=15)

item 0 inserido (valor=93)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item 0 removido (valor=93)

item 0 inserido (valor=35)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item 0 removido (valor=35)

item 0 inserido (valor=86)

item 1 inserido (valor=92)

Nº de itens a tratar no consumidor =10

item 1 removido (valor=92)

item 2 inserido (valor=21)

item 1 removido (valor=92)

item 0 removido (valor=86)

Nº de itens a tratar no produtor =3

item 0 inserido (valor=27)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item 0 removido (valor=27)

item 0 inserido (valor=90)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item 0 removido (valor=90)

item 0 inserido (valor=59)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item 0 removido (valor=59)

Buffer vazio -- consumidor aguardando...

Consumidor reiniciando...

item -1 removido (valor=77)

item -2 removido (valor=0)

item -3 removido (valor=0)

Nº de itens a tratar no produtor =4




```
item -3 inserido (valor=26)
item -3 removido (valor=26)
item -3 inserido (valor=40)
item -2 inserido (valor=26)
item -1 inserido (valor=72)
Nº de itens a tratar no consumidor =17

Buffer vazio -- consumidor aguardando...
Consumidor reiniciando...
item -1 removido (valor=72)
item -2 removido (valor=26)
```



Este comportamento ocorre porque, como não são sincronizados produtor e consumidor, eles ambos iteram indefinidamente, de modo a operarem em regiões críticas simultaneamente. Assim supondo que não existem itens para serem consumidos pelo consumidor, neste caso, a variável “pointer” seria 0. Porém, se o produtor ainda não inseriu um item no buffer e o consumidor continua sua execução até a linha “remove_item();”, ocorre que o consumidor reduzira “pointer” para -1 e, como o “if” no consumidor trata apenas do caso em que “pointer” == 0, quando pointer fica menor que 0, não caímos mais no “if” e não é impressa mais a mensagem de falta de produtos a serem consumidos, enquanto que as mensagens de remoção e inserção continuam ocorrendo. Desse modo, temos ambas as threads filhas operando com índices negativos e, conseqüentemente, proporcionando como saída o que notamos acima, um padrão esquisito de mensagens.



3.3) Abaixo, temos a saída no terminal para um caso em que foi causado uma situação de “deadlock”:

```
Unset
Produtor iniciando...
Consumidor iniciando...
Nº de itens a tratar no produtor =4

item 0 inserido (valor=86)
item 1 inserido (valor=77)
item 2 inserido (valor=15)
item 3 inserido (valor=93)
Nº de itens a tratar no consumidor =16

item 3 removido (valor=93)
item 2 removido (valor=15)
item 1 removido (valor=77)
item 0 removido (valor=86)
Buffer vazio -- consumidor aguardando...
Nº de itens a tratar no produtor =7
```



```
item 0 inserido (valor=92)
item 1 inserido (valor=49)
item 2 inserido (valor=21)
item 3 inserido (valor=62)
item 4 inserido (valor=27)
item 5 inserido (valor=90)
item 6 inserido (valor=59)
Nº de itens a tratar no produtor =4

item 7 inserido (valor=26)
item 8 inserido (valor=40)
item 9 inserido (valor=26)
item 10 inserido (valor=72)
Nº de itens a tratar no produtor =17

item 11 inserido (valor=11)
item 12 inserido (valor=68)
item 13 inserido (valor=67)
item 14 inserido (valor=29)
item 15 inserido (valor=82)
item 16 inserido (valor=30)
item 17 inserido (valor=62)
item 18 inserido (valor=23)
item 19 inserido (valor=67)
Buffer cheio -- produtor aguardando...
```

Nota-se, neste caso, que o programa entrou em situação de “deadlock” quando o buffer está cheio. Isso ocorre porque a alteração feita foi que as linhas de chamada de “pthread_cond_signal” foram comentadas, deste modo, poderia acontecer tanto de o “deadlock” ocorrer quando o buffer está cheio como quando ele está vazio, pois no caso de o buffer estar cheio ou vazio, ou o consumidor ou o produtor vão aguardar o outro inserir ou remover, respectivamente, itens no buffer, mas, como as chamadas “pthread_cond_signal” foram retiradas do código, então a thread em execução não informará que a outra pode assumir a execução. Dessa forma, entramos em uma situação em que uma das threads não permite que a outra execute, mas também não faz mais nada, pois sua lógica de execução não permite que ela faça algo significativo sem que a outra lhe permita (informe por meio da chamada “pthread_cond_signal”). Assim, entramos em uma situação de “deadlock” e, por consequência, o programa continua eternamente no mesmo ponto, tendo seu fluxo comprometido.

