

Questão 01. Mude o valor na chamada *criandothreadsbin* para 20. Qual foi o novo valor devolvido para *f*?

R:

f = 210

Questão 02. O que faz a função *funcaof*?

R:

A função *funcaof* realiza a soma dos números de 1 até o valor especificado pelo parâmetro *ultimo* e armazena o resultado na variável global *f*. Essa função é executada em uma thread criada no programa principal.

Questão 03. Crie uma nova função chamada *funcaok* e a variável global booleana *k*. Esta função deverá executar em uma nova thread, denominada, *thread2*. *funcaok* apresenta mesma assinatura de *funcaof*. No entanto, *funcaok* avalia se o valor passado como parâmetro é primo. Caso seja primo, *k* = true, caso contrário, *k* = false. Informe no método main (através de print) se o valor associado a *k* após execução de *funcaok* é true ou false.

R:

f = 210

k = false (O valor não é primo)

```
[8] %writefile criandothreads.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int f; // Dado compartilhado entre as threads
bool k; // Variável global booleana

void *funcaof(void *param); // Assinatura da função que será executada pela thread
void *funcaok(void *param);

int main(int argc, char *argv[]){
    if (argc != 2 && atoi(argv[1]) < 0) {
        fprintf(stderr, "sintaxe: ./criandothreads <valor inteiro maior que 0>\n");
        return -1;
    }

    pthread_t thread1, thread2; // cria variável do tipo thread
    pthread_attr_t attr, attr2; // cria variável do tipo atributo de thread
    pthread_attr_init(&attr); // inicializa attr com valores padroes
    pthread_attr_init(&attr2); // inicializa attr com valores padroes

    pthread_create(&thread1, &attr, funcaof, argv[1]); // cria a thread
    pthread_create(&thread2, &attr2, funcaok, argv[1]); // cria a thread

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("f = %d\n", f);

    if (k) {
        printf("k = true (O valor é primo)\n");
    } else {
        printf("k = false (O valor não é primo)\n");
    }

    return 0;
}

void *funcaof(void *param) {
    int i, ultimo = atoi(param);
    f = 0;

    if (ultimo > 0)
        for (i = 1; i <= ultimo; i++)
            f += i;

    pthread_exit(0);
}

void *funcaok(void *param) {
    int num = atoi(param);
    k = true; // Assumimos inicialmente que o número é primo

    if (num <= 1) {
        k = false; // 0 e 1 não são primos
    } else {
        for (int i = 2; i <= num / 2; i++) {
            if (num % i == 0) {
                k = false; // não é primo
                break;
            }
        }
    }

    pthread_exit(0);
}
```

Questão 04. Na caixa de código abaixo, execute o comando ```!unimize```, necessário para realização da Questões 05 e 06.

R:

Feito.

Questão 05 - Na caixa de código a seguir, utilize o comando `man` para `pthread_create` e informe em que casos uma nova thread é finalizada.

R:

- Chamando **`pthread_exit`** com um valor de status.
- Retornando da função **`start_routine`**, equivalente a chamar **`pthread_exit`** com o valor de retorno.
- Sendo cancelada por outra thread usando **`pthread_cancel`**.
- Se uma das threads no processo chamar **`exit`**, ou a thread principal retornar do `main`, todas as threads no processo serão encerradas.

Questão 06. Investigue em <https://docs.oracle.com/cd/E19455-01/806-5257/attrib-34610/index.html> os valores associados por padrão pelo método `pthread_attr_init` a variáveis do tipo `pthread_attr_t`. Apresente o significado de cada um deles.

R:

Atributo	Valor	Resultado
<i>scope</i>	PTHREAD_SCOPE_PROCESS	O novo thread está desvinculado - não está permanentemente anexado ao LWP.
<i>detachstate</i>	PTHREAD_CREATE_JOINABLE	O status de saída e o thread são preservados após o término do thread.
<i>stackaddr</i>	NULO	O novo thread possui endereço de pilha alocado pelo sistema.
<i>stacksize</i>	1 megabyte	O novo thread tem tamanho de pilha definido pelo sistema.
<i>priority</i>		O novo thread herda a prioridade do thread pai.
<i>inheritsched</i>	PTHREAD_INHERIT_SCHED	O novo thread herda a prioridade de agendamento do thread pai.
<i>schedpolicy</i>	SCHED_OTHER	Política de escalonamento padrão

Questão 07. Na caixa de código a seguir, utilize o comando `man pthread_join` e informe para que serve este comando.

R:

É usado para visualizar o manual da função “`pthread_join`” no terminal.

A função `pthread_join` permite que o programa principal obtenha o status de saída da thread concluída e, se desejado, libere os recursos associados a essa thread.

Questão 08. O que foi impresso ao final da execução de `./clonebin 1 2` ? Na linha de código abaixo faça uma nova execução, chamando o programa da seguinte forma: `./clonebin 5 6 7 8`. O que foi impresso?

R:

```
! ./clonebin 1 2

Foi alocado espaço para (2) threads
Threads filhas finalizadas.
Executando na tarefa filha 2
  thread(2) = 2 * 1 = 2
  thread(2) = 2 * 2 = 4
  thread(2) = 2 * 3 = 6
  thread(2) = 2 * 4 = 8
  thread(2) = 2 * 5 = 10
  thread(2) = 2 * 6 = 12
  thread(2) = 2 * 7 = 14
  thread(2) = 2 * 8 = 16
  thread(2) = 2 * 9 = 18
  thread(2) = 2 * 10 = 20

Executando na tarefa filha 1
  thread(1) = 1 * 1 = 1
  thread(1) = 1 * 2 = 2
  thread(1) = 1 * 3 = 3
  thread(1) = 1 * 4 = 4
  thread(1) = 1 * 5 = 5
  thread(1) = 1 * 6 = 6
  thread(1) = 1 * 7 = 7
  thread(1) = 1 * 8 = 8
  thread(1) = 1 * 9 = 9
  thread(1) = 1 * 10 = 10
```

```
! ./clonebin 5 6 7 8

Foi alocado espaço para (4) threads
Executando na tarefa filha 5
  thread(5) = 5 * 1 = 5
  thread(5) = 5 * 2 = 10
  thread(5) = 5 * 3 = 15
  thread(5) = 5 * 4 = 20
  thread(5) = 5 * 5 = 25
  thread(5) = 5 * 6 = 30
  thread(5) = 5 * 7 = 35
  thread(5) = 5 * 8 = 40
  thread(5) = 5 * 9 = 45
  thread(5) = 5 * 10 = 50

Threads filhas finalizadas.
Executando na tarefa filha 6
  thread(6) = 6 * 1 = 6
  thread(6) = 6 * 2 = 12
  thread(6) = 6 * 3 = 18
  thread(6) = 6 * 4 = 24
  thread(6) = 6 * 5 = 30
  Executando na tarefa filha 8
  thread(6) = 6 * 6 = 36
    thread(8) = 8 * 1 = 8
    thread(6) = 6 * 7 = 42
    thread(8) = 8 * 2 = 16
  thread(6) = 6 * 8 = 48
    thread(8) = 8 * 3 = 24
  thread(6) = 6 * 9 = 54
    thread(8) = 8 * 4 = 32
  thread(6) = 6 * 10 = 60

  thread(8) = 8 * 5 = 40
  thread(8) = 8 * 6 = 48
  thread(8) = 8 * 7 = 56
  thread(8) = 8 * 8 = 64
  thread(8) = 8 * 9 = 72
  thread(8) = 8 * 10 = 80

Executando na tarefa filha 7
  thread(7) = 7 * 1 = 7
  thread(7) = 7 * 2 = 14
  thread(7) = 7 * 3 = 21
  thread(7) = 7 * 4 = 28
  thread(7) = 7 * 5 = 35
  thread(7) = 7 * 6 = 42
  thread(7) = 7 * 7 = 49
  thread(7) = 7 * 8 = 56
  thread(7) = 7 * 9 = 63
  thread(7) = 7 * 10 = 70
```

Questão 09. Execute por mais vezes a chamada `./clonebin 5 6 7 8`. As respostas de cada thread sempre são impressas na mesma ordem ou há mistura? Caso ocorra mistura, qual o motivo que leva a essa mistura?

R:

Sim, houve mistura. O tempo em que uma thread está executando e outra executa na mesma hora, sem antes finalizar a thread que já havia iniciado.

Questão 10. Explique o que está sendo feito na função `int thread_code(void *arg)` (linhas 11-19).

R:

1. Converte o argumento da thread para um número inteiro.
2. Imprime uma mensagem indicando a identificação da thread.
3. Executa um loop de 1 a 10, imprimindo resultados de operações de multiplicação.
4. Retorna 0 como código de saída da thread.

Essa função é executada por cada thread criada, e a mistura nas saídas ocorre quando as threads são executadas concorrentemente.

Questão 11. Na caixa de código a seguir, execute o comando `man` para a função `clone`. Extraia do manual de `clone`:

Para que serve a função `clone`?

Para que serve as constantes `CLONE_VM`, `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND` usadas pelo código na linha 43.

R:

A função `clone` é usada para criar um novo processo ou thread.

`CLONE_VM`: Indica que o novo processo compartilhará o mesmo espaço de endereço virtual que o processo pai.

`CLONE_FS`: Indica que o novo processo compartilhará o mesmo sistema de arquivos (estrutura de diretório).

`CLONE_FILES`: Indica que o novo processo compartilhará a mesma tabela de descritores de arquivo que o processo pai.

`CLONE_SIGHAND`: Indica que o novo processo compartilhará o mesmo conjunto de manipuladores de sinal (tratamento de sinais) que o processo pai.

Questão 12. Para que serve a região de memória denominada `stack`? Threads possuem `stack`. Elas são individuais ou compartilhadas, justifique.

R:

A região de memória chamada "stack" é usada para armazenar dados temporários durante a execução de um programa. Threads possuem stacks individuais para garantir independência e isolamento entre elas. Isso permite que cada thread tenha suas próprias variáveis locais e evita interferência direta entre threads. Existem modelos de implementação com stacks próprias (individual) ou compartilhadas, sendo a abordagem individual mais comum devido a benefícios como isolamento e segurança.

Questão 13. Em `clone.c` qual o tamanho da stack adotada por cada thread? Explique o que acontece nas linhas [22-38] do código.

R:

No código, o tamanho alocado para cada thread é definido pela constante `STACK_SIZE`, que tem o valor de 65536 bytes. Portanto, cada thread terá uma stack com um tamanho de 65536 bytes.

1. Alocação dinâmica de stacks para cada thread, usando **`malloc`**.
2. Criação de threads com a função **`clone`**, especificando o código da thread, o topo da stack e algumas flags de clonagem.
3. Aguarda o término de cada thread com a função **`wait`**.
4. Liberação da memória alocada para as stacks após o término das threads.

Questão 14. Porque o processo pai e o processo filho apresentam valores diferentes para a variável global *valor*?

R:

Tanto processo pai, como processo filho apresentam valores diferentes para a variável global *valor* pois, eles têm espaços de memória separados. Quando o processo filho é criado usando a função `fork()`, ele duplica o espaço de endereçamento do processo pai, porém, os dois processos executam de forma independente a partir desse ponto. A variável *valor* é inicializada como 5 no espaço de memória do processo pai. Quando o processo filho é criado, ele herda o valor atual da variável *valor* do pai. No entanto, o processo filho modifica sua própria cópia da variável, aumentando em 15 unidades. Enquanto o processo filho tem o valor modificado ($5 + 15 = 20$), o processo pai mantém sua própria cópia original de *valor* (5).

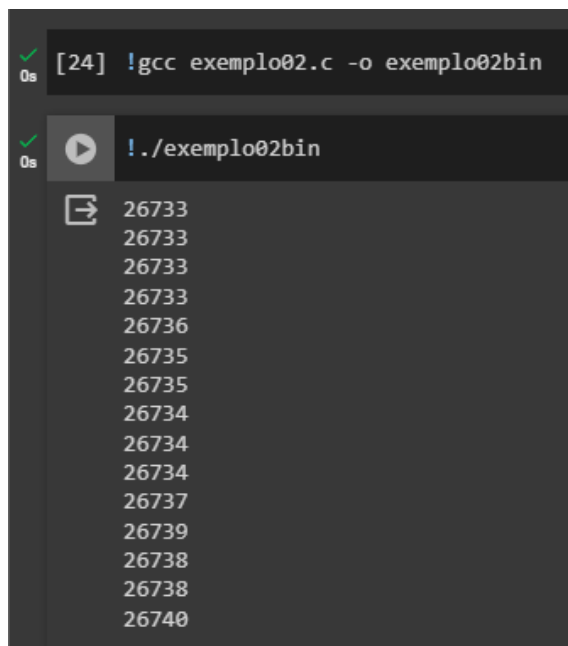
Questão 15. Na caixa de código abaixo, use o comando `man` para consultar a função `fork`. Qual o valor que `fork` devolve para o processo pai e para o processo filho?

R:

No processo pai, o valor retornado é o ID do processo filho (PID). cada PID é único, portanto, o valor de retorno no processo pai é o PID do processo recém-criado. Já no processo filho, o valor retornado é 0. Isso indica que o processo é o filho recém-criado.

Questão 16. `getpid` é uma função que devolve o process id (identificador único) do processo em execução. Baseado nessa informação, execute as seguintes caixas de código para geração do arquivo `exemplo02.c`, compilação e execução.

R: Feito.



```
[24] !gcc exemplo02.c -o exemplo02bin
!./exemplo02bin
26733
26733
26733
26733
26736
26735
26735
26734
26734
26734
26737
26739
26738
26738
26740
```

Questão 17. Baseado na execução da Questão 16, o que significam os números que foram impressos? Quantos números diferentes temos após a execução do código? Quantos processos foram criados durante a execução desse código?

R:

Os números que foram impressos, são os PIDs. São oito processos em execução, onde cada um imprime seu próprio PID, resultando em oito linhas diferentes, cada uma representando um PID diferente e 12 processos diferentes foram criados durante a execução do código.

Questão 18. É possível associar uma fórmula que informe a quantidade de processos criados a depender da quantidade de forks? Considere n , como sendo a quantidade de forks e nps , como sendo a quantidade de processos criados.

R:

Sim, Cada chamada **fork** duplica o número de processos existentes. Então, se você faz n chamadas **fork**, a quantidade total de processos criados será 2^n . Pegando como exemplo o código que executamos anteriormente, nele contém três chamadas **fork**, então, seriam criados $2^3 = 8$ processos.

Questão 19. Use as caixas de código a seguir para modificar, compilar e executar **10processos.c**. Ao final da execução de **10processos.c**, devem ser gerados 10 processos.

R:

```
[2] !gcc 10processos.c -o 10processosbin

0s

[2] !./10processosbin

0s

593
593
593
593
596
595
595
597
594
594
594
599
598
598
600
```

modificado:

```
[13] }

Overwriting 10processos.c

[14] !gcc 10processos.c -o 10processosbin

[14] !./10processosbin

ID do processo: 1787, ID do processo pai: 170
ID do processo: 1787, ID do processo pai: 170
ID do processo: 1787, ID do processo pai: 170
ID do processo: 1789, ID do processo pai: 1787
ID do processo: 1787, ID do processo pai: 170
ID do processo: 1789, ID do processo pai: 1787
ID do processo: 1788, ID do processo pai: 1
ID do processo: 1791, ID do processo pai: 1
ID do processo: 1788, ID do processo pai: 1
ID do processo: 1790, ID do processo pai: 1
ID do processo: 1788, ID do processo pai: 1
ID do processo: 1793, ID do processo pai: 1
ID do processo: 1792, ID do processo pai: 1
ID do processo: 1792, ID do processo pai: 1
ID do processo: 1794, ID do processo pai: 1
```