Universidade Federal de Itajubá
CRSC02 – Sistemas Operacionais
Prof. Carlos Minoru Tamaki

Exercício Prático 02 - EP02: Threads

1 Recomendações

- Para os programas utilize das recomendações de boas práticas de programação disponível no SIGAA tópico "Apresentação da disciplina (14/08/2024)".
- Para as questões, as respostas devem ser **completas** e **justificadas**. Poderá utilizar o Word ou o Writer para as respostas, utilize o cabeçalho (acima) da proposta do exercício para o documento das respostas, não esqueça de colocar o nome e número de matrícula substituindo o nome do professor. Antes de entregar salve o documento no formato PDF.
- **TODOS** os programas fonte deverão ser entregues, inclusive os que não funcionaram nos micros do laboratório devido a segurança.
- Para a entrega você deverá compactar somente os arquivos **PDF e .C**, não compacte diretórios.
- É necessário que somente um dos componentes do grupo entregue o arquivo ZIP no SIGAA.

2 Introdução

Como visto em aula, uma thread é a unidade básica de utilização da CPU. De modo similar aos processos, threads possibilitam a uma aplicação realizar múltiplas tarefas de forma simultânea. Um processo pode possuir mais de uma thread, sendo que as threads de um processo executam de forma *concorrente*. Em arquiteturas multiprocessadas, múltiplas threads podem executar de forma *paralela*.

O Sistema Operacional Linux possui duas implementações distintas do mecanismo de threads:

- *LinuxThreads*: foi desenvolvida pelo cientista da computação francês Xavier Leroy. Por vários anos, a LinuxThreads foi o principal mecanismo de implementação de threads no Linux. No passado, este mecanismo era suficiente para suportar diversas aplicações multithreaded. Em essência, a LinuxThreads faz uso da system call `clone()` para criar threads, sendo que esta system call ainda existe no Linux. A LinuxThreads implementava parcialmente o padrão POSIX (Portable Operating System Interface) para criação de threads, mas possui diversas divergências em relação a este padrão. Por exemplo, uma chamada à função `getpid()` devolve valores distintos para cada thread.
- *NPTL* (*Native POSIX Threads Library*): foi desenvolvida na Red Hat, pelos cientistas da computação Ulrich Drepper e Ingo Molnar. Este é o mecanismo moderno (e atual) de implementação de threads no Sistema Operacional Unix (e seus derivados, como o Linux). Quando comparado com a LinuxThreads, a NPTL oferece desempenho superior, além de que, enquanto que a LinuxThreads possui vários aspectos que não seguem a especificação POSIX.1 para implementação de threads, a NPTL a segue. A geração de threads da NPTL é realizada de acordo com o modelo

1:1 (um para um), isto é, cada thread criada pelo programador é mapeada para uma thread de kernel. A NPTL provê uma API (*Application Programming Interface*) para threads chamada de *POSIX Threads*, ou, como é mais comumente conhecida, *PThreads*.

Em versões mais antigas do kernel do Linux, é possível que as duas implementações estejam disponíveis. Entretanto, estas versões de sistema tornaram-se históricas, pois desde a sua versão 2.4, a biblioteca GNU C (*glibc*) não dá mais suporte à LinuxThreads. Em sistemas onde a *glibc* possui versão 2.3.2 ou posterior, podemos usar o seguinte comando para descobrir qual mecanismo de implementação de threads é usado pelo sistema:

```
aluno@lab:~$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.35
aluno@lab:~$
```

Inicialmente, vamos estudar a criação de threads usando a system call `clone()` diretamente no código. Posteriormente, faremos uso da API PThreads, que traz maiores facilidades para o gerenciamento e sincronização de threads em programas C/C++.

3 A system call `clone()`

De maneira similar à função `fork()`, que duplica um processo, a system call `clone()` também é usada para criar um novo *processo*[1]. Entretanto, `clone()` se diferencia de `fork()` por possibilitar um controle maior da forma na qual o processo é criado.

A sintaxe do comando `clone()` é:

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Em caso de sucesso, a função `clone()` devolve o PID do processo criado. Em caso de falha, ela retorna o valor −1.

Como na função `fork()`, o novo processo criado usando `clone()` é uma cópia quase exata do processo pai. Porém, de maneira distinta de `fork()`, o filho criado não continua sua execução a partir do ponto de invocação; mas sim, inicia a execução de uma função especificada pelo argumento `func`. Esta função é chamada de *função filha*. Ao ser invocada, os parâmetros especificados por `func_arg` são passados à função filha que, através do uso de *typecast*, pode interpretar este argumento.

O processo filho criado encerra sua execução juntamente com a função `func` invocada. Além disso, o valor retornado por `func` é o valor de retorno deste processo. O processo pai pode esperar pelo filho da maneira usual, fazendo uso de `wait()`, ou funções similares (`waitpid()`).

O processo filho criado pode compartilhar o espaço de memória do pai (através de flags específicas), porém, ele deve ter seu próprio segmento de pilha (stack). Neste caso, a função que invoca `clone()`

[1] O SO Linux não faz distinção entre processos e threads. O termo *task* (tarefa) é usado para se referir a esses tipos de entidades que representam um fluxo de execução de um programa e são escalonáveis pelo kernel. A diferenciação se dá pelo compartilhamento de recursos entre eles. O PCB de um processo é representado pela estrutura `task_struct`, definida no arquivo de cabeçalho `<linux/sched.h>`.

deve alocar uma região de memória com tamanho suficiente para que o filho possa usar como pilha e passar uma referência a esta região como parâmetro `child_stack` de `clone()`. Na maioria das arquiteturas de hardware, a pilha "cresce para baixo", isto é, conforme a pilha é usada, endereços de memória menores vão sendo alocados. Assim, o ponteiro `child_stack` deve apontar para o endereço de memória mais alto do bloco alocado.

Por fim, o parâmetro `flags` possui dois propósitos. Ele armazena em seus 8 bits menos significativos o sinal de encerramento do filho. Os demais bits armazenam uma máscara de bits que controla a operação da função `clone()`. Os principais valores da máscara de bits são:

- `CLONE_FILES`: pai e filho compartilham arquivos abertos.
- `CLONE_FS`: pai e filho compartilham informações do sistema de arquivos.
- `CLONE_SIGHAND`: pai e filho compartilham a manipulação de sinais.
- `CLONE_THREAD`: inclui o filho no mesmo grupo de processos do pai.
- `CLONE_VM`: pai e filho compartilham a memória virtual.

Observe que se nenhuma flag for passada à função `clone()`, não ocorrerá compartilhamento de recursos, resultando em uma funcionalidade semelhante à fornecida pela system call `fork()`.

Para ilustrar a criação de uma thread, considere o Código 1.

No Código 1, duas threads adicionais (além da thread que executa a função principal) são criadas. Para isso, são declaradas duas variáveis inteiras (`tid1` e `tid2`), cujos valores são os TID de cada uma das threads (linha 24). Nas linhas 25 e 26, são alocados dois blocos de memória de 64 KB cada, para serem usados como região de pilha por cada thread. Embora o Código 1 não realize explicitamente esta comparação, deve-se testar o valor das variáveis `tid1` e `tid2` após a invocação de `clone()` e, em caso de falha na criação das threads (`tid1 = -1` ou `tid2 = -1`), encerrar o programa.

A invocação da função `clone()` nas linhas 30 e 33 cria as threads. Ambas as threads executam a função `run()`, definida no início do código (linha 11), recebendo, assim, uma referência a ela. Como segundo parâmetro, tem-se o endereço de memória mais alto da região de pilha criada.

O terceiro parâmetro são as flags de compartilhamento. No invocação de `clone()`, as flags são combinadas através do operador OR bit a bit (`|`). A passagem de todas as cinco flags mostradas faz com que o processo recém criado se comporte como uma thread no sistema, seguindo a teoria clássica de SO vista em aula, com compartilhamento de dados e recursos dentro do mesmo processo. Lembre-se que, sem o compartilhamento, cria-se apenas outro processo isolado no sistema.

O resultado da compilação e execução do Código 1 é mostrado a seguir:

```
aluno@lab:~$ gcc -o saida exemploclone.c
aluno@lab:~$ ./saida
Main: PID = 9026, TID = 9026
Main: TID da Thread 1 = 9027
Main: TID da Thread 2 = 9028
Thread 2: PID = 9026, TID = 9028
Thread 1: PID = 9026, TID = 9027
Thread 2 terminando...
Thread 1 terminando...

aluno@lab:~$
```

Código 1: `exemploclone.c`

```
1  #define _GNU_SOURCE // para acesso a definicoes especificas do SO Linux
2
3  #include <stdio.h>
4  #include <stdlib.h> // malloc()
5  #include <stdint.h> // intptr_t
6  #include <unistd.h> // getpid(), gettid()
7  #include <sched.h> // clone(), flags
8
9  #define STACK_SIZE 64 * 1024 // 64 KB de pilha
10
11 int run(void *arg)
12 {
13     int x = (intptr_t)arg;
14
15     printf("Thread %d: PID = %d, TID = %d\n", x, getpid(), gettid() );
16     sleep(30);
17     printf("Thread %d terminando...\n", x);
18
19     return 0;
20 }
21
22 int main()
23 {
24     int tid1, tid2;
25     void *child_stack1 = malloc(STACK_SIZE);
26     void *child_stack2 = malloc(STACK_SIZE);
27
28     printf("Main: PID = %d, TID = %d\n", getpid(), gettid() );
29
30     tid1 = clone(&run, child_stack1 + STACK_SIZE, CLONE_SIGHAND | CLONE_FS |
31 CLONE_VM | CLONE_FILES | CLONE_THREAD, (void *)1);
32
33     tid2 = clone(&run, child_stack2 + STACK_SIZE, CLONE_SIGHAND | CLONE_FS |
34 CLONE_VM | CLONE_FILES | CLONE_THREAD, (void *)2);
35
36     printf("Main: TID da Thread 1 = %d\n", tid1);
37     printf("Main: TID da Thread 2 = %d\n", tid2);
38
39     getchar();
40
41     return 0;
42 }
```

A partir do resultado da execução do Código 1, é possível observar que a ordem em que as mensagens são impressas na tela depende da ordem em que as threads foram escalonadas pelo SO. Além disso, seus valores de PID e TID também são variáveis entre máquinas e execuções distintas.

Nota-se que o Código 1 possui três threads executando (a thread do main, mais as duas criadas). Com um valor suficientemente grande na função `sleep()`, linha 16, ao executar o Código 1, abrir outra instância de terminal e listar os processos em execução, obtemos a seguinte saída:

```
aluno@lab:~$ ps -eLF | grep -e LWP -e saida
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME  CMD
aluno     9744  6361  9744  0    3   624   572   1 14:46 pts/0    00:00:00  ./saida
aluno     9744  6361  9745  0    3   624   572   1 14:46 pts/0    00:00:00  ./saida
aluno     9744  6361  9746  0    3   624   572   1 14:46 pts/0    00:00:00  ./saida
aluno@lab:~$
```

Uma vez que o Linux não faz distinção entre processos e threads, denotando ambas as entidades como *tasks*, threads criadas conforme exemplificado no Código 1 são chamadas de *Lightweight Process*. Nesse caso, o TID das threads é mostrado na quarta coluna da lista de processos, sob a sigla `LWP`. Nota-se, também, que estes são os mesmos valores impressos na tela pelo Código 1 ao invocar a função `gettid()` (linhas 15 e 28). A função `gettid()`, específica do sistema Linux, é definida no arquivo de cabeçalho `<unistd.h>` (apenas no Linux!!! Ela não é portável para outros Unix-like systems...) e retorna o número identificador da thread no SO.

Ainda no Código 1, nas linhas 30 e 33, é possível notar um último parâmetro, que é um número inteiro. Este valor foi usado para identificar a thread dentro da função filha. Por padrão, a função filha recebe um único argumento, na forma de ponteiro para `void`, como especificado na linha 11. Dessa forma, nas linhas 30 e 33 é realizado o *typecast* para este tipo. Já na função filha, em sua linha 13, realiza-se novamente o *typecast* para que o valor do parâmetro `arg` seja novamente interpretado como `int`. Nota-se, portanto, que não é possível passar mais que um parâmetro para a função filha.

Entretanto, para algumas aplicações, a função filha deve receber várias informações como parâmetros, de modo que estas informações são usadas em sua execução. A maneira pela qual isto pode ser realizado, sem violar a assinatura da função filha, é passar um ponteiro para um estrutura, onde a estrutura encapsula os vários parâmetros que devem ser recebidos como argumento. Um exemplo de passagem de uma `struct` como parâmetro é mostrado no Código 2.

No Código 2, a função filha executa um laço, imprimindo os valores da variável de controle do laço na tela. Como detalhe adicional, os limites do laço são recebidos pela função filha como parâmetro. Para isso, na linha 9, declara-se uma estrutura para encapsular os parâmetros de início e fim (respectivamente, `i` e `n`) do laço. Na função principal, as linhas 32 a 34 declaram e atribuem os valores à variável `cp`, que será passada para a função filha.

Uma vez que a função filha possui `(void *)` como parâmetro de entrada, inicialmente, na invocação de `clone()`, passa-se o endereço da variável da estrutura (`&cp`), com a realização do devido *typecast* para ponteiro de `void`. Já na linha 16, é realizado o *typecast* para o tipo de ponteiro da estrutura e, em seguida, atribui-se o seu conteúdo à variável local `cp`. A partir de então, os campos da estrutura podem ser usados normalmente pelo código da função filha através da variável `cp`, como feito no cabeçalho do laço (linha 19).

No Código 2, há uma única thread que recebe uma estrutura como parâmetro. No caso de haver mais de uma thread e os parâmetros forem distintos para cada thread, deve-se criar e atribuir valores para mais de uma variável do tipo da estrutura, e cada variável é repassada à uma chamada de `clone()`.

Exercício 1: Responda: o que a função `clone()` retorna?

Exercício 2: Responda: qual a diferença entre processos criados usando `fork()` e usando `clone()`?

Exercício 3: Programe e responda: o que acontece se a função principal encerrar sua execução, mas as threads por ela criadas não tiverem encerrado ainda?

Exercício 4: Programe e responda: o que acontece se uma thread invoca a função `fork()` durante sua execução?

Código 2: `childparams.c`

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

#define STACK_SIZE 64 * 1024

typedef struct
{
    int i, n;
}child_param;

int run(void *arg)
{
    child_param cp = *((child_param *)arg);
    int j;

    for(j = cp.i; j <= cp.n; j++)
        printf("%d ", j);

    printf("\n");

    return 0;
}

int main()
{
    int tid;
    void *child_stack = malloc(sizeof(STACK_SIZE));

    child_param cp;
    cp.i = 1;
    cp.n = 10;

    tid = clone(&run, child_stack + STACK_SIZE, CLONE_SIGHAND | CLONE_FS |
CLONE_VM | CLONE_FILES | CLONE_THREAD, (void *)&cp);

    getchar();

    return 0;
}
```

Exercício 5: Programe e responda: o que acontece se uma thread invoca a função `clone()` durante sua execução?

Exercício 6: Programe e responda: o que acontece se uma thread invoca uma das funções da família `exec()` durante sua execução?

Por fim, uma vez que a função `clone()` é específica da implementação do Linux, sua chamada direta em código é desaconselhada para programas que necessitam de portabilidade com outras variantes

do Unix (BSD, System V, etc.). Programas que necessitam ser portáveis entre distintos Sistemas Operacionais devem fazer uso das rotinas disponíveis na PThreads, mostrada a seguir.

4 A API PThreads

A *PThreads* refere-se ao padrão POSIX (IEEE 1003.1c) para criação e sincronização de threads. A PThreads é uma *especificação*, e não uma *implementação*. Isso significa que os projetistas de sistemas operacionais são livres para implementar a especificação como desejarem. Diversos sistemas operacionais implementam a PThreads, como o Unix, BSD, Linux, Solaris, macOS.

As funções que compõem a API PThreads podem ser categorizadas em quatro grupos:

- *Gerenciamento de threads*: funções que trabalham diretamente com threads, seja na criação, junção, separação, encerramento, etc. Também inclui funções para consultar atributos ou informações refentes às threads.
- *Mutexes*: funções que lidam com mecanismos de sincronização, chamados de "*mutexes*", que é a abreviação de "*mutual exclusion*" (exclusão mútua – *próximas aulas de SO*). Estas funções permitem criar, destruir, bloquear e desbloquear mutexes. Também engloba funções que definem ou modificam atributos associados a mutexes.
- *Sincronização*: funções que gerenciam bloqueios (*locks*) de leitura e/ou escrita.
- *Variáveis condicionais*: funções que tratam de comunicações entre threads que compartilham um mutex. São baseadas em condições especificadas pelo programador. Este grupo inclui funções para criar, destruir, esperar e sinalizar threads, com base em valores de variáveis especificados. Funções para definir/consultar atributos de variáveis de condição também estão incluídas.

4.1 Criação de threads

Assim que um programa é carregado na memória, o processo correspondente possui uma única thread. Todas as demais threads devem ser criadas pelo programador. A função `pthread_create()` cria uma nova thread e a torna executável no sistema. Esta função pode ser invocada em qualquer ponto do código, quantas vezes forem necessárias. Sua sintaxe é:

```
#include <pthread.h>

int pthread_create(pthread_t restrict *thr, const pthread_attr_t restrict *attr,
    void *(*start)(void*), void restrict *arg);
```

A função `pthread_create()` cria uma nova thread, com os atributos `attr`, dentro do mesmo processo. Se `attr` for setado como `NULL`, a thread é criada com os atributos padrão; e, no caso de `attr` ser modificado após a criação da thread, os atributos dela não sofrem alteração. Além disso, caso a solicitação de criação da thread seja executada com sucesso, a função `pthread_create()` armazena em `thr` um valor identificador da thread. O retorno de `pthread_create()` é zero, em caso de sucesso; em caso de falha, um valor inteiro indicando um código de erro é devolvido.

O Código 3 ilustra a criação de threads, sendo que sua compilação e resultado da sua execução são mostrados a seguir:

```
aluno@lab:~$ gcc -o saida pthrcriacao.c -pthread
aluno@lab:~$ ./saida
Criando a Thread 1
Criando a Thread 2
Criando a Thread 3
Criando a Thread 4
Criando a Thread 5
Criando a Thread 6
Criando a Thread 7
Thread 3 finalizando...
Thread 2 finalizando...
Criando a Thread 8
Thread 1 finalizando...
Thread 6 finalizando...
Thread 7 finalizando...
Thread 4 finalizando...
Thread 5 finalizando...
Criando a Thread 9
Criando a Thread 10
Main finalizando...
Thread 9 finalizando...
Thread 8 finalizando...
Thread 10 finalizando...
aluno@lab:~$
```

Códigos que fazem uso da API PThreads devem ser compilados com a flag de compilação `-pthread`. Esta opção faz com que o programa seja linkado com a biblioteca *libpthread* e habilita macros de código reentrante (*próximas aulas de SO*).

Em sua linha 18, o Código 3 declara um vetor chamado `thr` para a criação de 10 threads, através de um laço `for`. Todas as threads são criadas na linha 28, com a invocação de `pthread_create()`. Esta função recebe a referência para a variável que representa a thread (`&thr[i]`), o nome da função filha (`run`) e um ponteiro de inteiro (`pi`), com o parâmetro que será passado à função filha. O parâmetro `NULL` indica que a thread será criada com os atributos padrão do sistema.

Uma vez que as threads criadas recebem um parâmetro inteiro, usado apenas para indicar sua numeração durante a execução, as linhas 25 e 26 alocam memória para este valor inteiro e fazem sua inicialização com base no valor da variável `i`. Este valor é convertido novamente para o tipo `int` através do *typecast* realizado na linha 9.

Como pode ser observado na execução do Código 3 mostrada anteriormente, assim que as threads são criadas dentro do laço `for`, o Sistema Operacional as escalona seguindo algum critério. Dessa maneira, não é possível prever a ordem de execução das threads criadas. Nota-se, inclusive, que algumas threads finalizam sua execução enquanto outras ainda nem foram criadas. A própria thread da função principal finaliza antes de algumas outras threads.

A execução de uma thread se encerra por uma das seguintes razões:

- A função filha executa o comando `return` e devolve um valor.
- A thread invoca a função `pthread_exit()`.
- A thread é cancelada por outra thread, através da função `pthread_cancel()`.

Código 3: `pthrcriacao.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #define MAX 10
6
7  void *run(void *arg)
8  {
9      int id = *((int *)arg);
10
11     printf("Thread %d finalizando...\n", id);
12
13     pthread_exit(NULL);
14 }
15
16 int main()
17 {
18     pthread_t thr[MAX];
19
20     int ret, i, *pi;
21
22     for(i = 0; i < MAX; i++)
23     {
24         printf("Criando a Thread %d\n", i+1);
25         pi = malloc(sizeof(int *));
26         *pi = (i+1);
27
28         ret = pthread_create(&thr[i], NULL, run, (void *)pi);
29
30         if(ret)
31         {
32             fprintf(stderr, "falha na criacao da thread...\n");
33             exit(1);
34         }
35     }
36
37     printf("Main finalizando...\n");
38     pthread_exit(NULL);
39 }
```

- O processo inteiro é encerrado por meio das instruções `exec()` ou `exit()`.
- A função principal se encerra sem invocar `pthread_exit()` explicitamente como seu último comando.

A função `pthread_exit()` invocada nas linhas 13 e 38 é usada para encerrar a thread. Sua sintaxe é:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Nesta função, o parâmetro `retval` é usado para especificar o valor de retorno da thread. O valor referenciado por este parâmetro não deve estar na pilha (stack) da thread, uma vez que a stack torna-se indefinida com o encerramento da thread.

É importante lembrar que toda thread criada invocando `pthread_create()` é mapeada para uma thread de kernel do SO. De modo análogo ao Código 1, adicionando `sleep()` com um valor suficientemente grande na linha 12 do Código 3 e abrindo outra janela de terminal, ao listar os processos em execução, obtemos a seguinte saída:

```
aluno@lab:~$ ps -eLF | grep -e LWP -e saida
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME  CMD
aluno     9860  6361  9860  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9861  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9862  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9863  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9864  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9865  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9866  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9867  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9868  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9869  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno     9860  6361  9870  0   11  2124   944   1 18:16 pts/0    00:00:00  ./saida
aluno@lab:~$
```

Exercício 7: Programe e responda: o que acontece se a função principal encerrar sua execução, mas as threads por ela criadas com `pthread_create()` não tiverem encerrado ainda?

Exercício 8: Programe e responda: o que acontece se uma thread criada com `pthread_create()` invoca a função `fork()` durante sua execução?

Exercício 9: Programe e responda: o que acontece se uma thread criada com `pthread_create()` invoca a função `clone()` durante sua execução?

Exercício 10: Programe e responda: o que acontece se uma thread criada com `pthread_create()` invoca uma das funções da família `exec()` durante sua execução?

4.2 Passagem de parâmetros e valores de retorno

Como visto na definição da função `pthread_create()`, a função filha pode receber parâmetros na forma de um ponteiro de `void`. Caso seja necessário passar mais de um parâmetro, usa-se o mesmo recurso empregado no Código 2: encapsulamos todos os argumentos em uma estrutura e a repassamos à função filha.

Em muitas situações, desejamos receber respostas das threads que estão em execução. Neste caso, basta estender o mesmo artifício da estrutura, encapsulando, também, variáveis para conter os valores de retorno.

O Código 4 a seguir exemplifica a passagem de parâmetros e a obtenção do valor de um cálculo realizado pela função filha. Este código executa o chamado paralelismo de dados, onde cada thread manipula um subconjunto dos dados originais.

O Código 4 declara um vetor com 10 elementos (linha 24) e cria duas threads, para que cada uma some uma metade do vetor. Ao fim, a thread principal (`main`) soma os dois valores intermediários obtidos das threads anteriores.

Código 4: `pthrparamret.c`

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  typedef struct
5  {
6      int *vet, ini, fim, tot;
7  }child_param;
8
9  void *run(void *arg)
10 {
11     child_param *cp = ((child_param *)arg);
12     int i, soma = 0;
13
14     for(i = cp->ini; i <= cp->fim; i++)
15         soma += cp->vet[i];
16
17     cp->tot = soma;
18
19     pthread_exit(NULL);
20 }
21
22 int main()
23 {
24     int vet[] = {1,2,3,4,5,6,7,8,9,10};
25
26     child_param cp1, cp2;
27
28     cp1.vet = vet;
29     cp1.ini = 0;
30     cp1.fim = 4;
31
32     cp2.vet = vet;
33     cp2.ini = 5;
34     cp2.fim = 9;
35
36     pthread_t t1, t2;
37
38     pthread_create(&t1, NULL, run, (void *) &cp1);
39     pthread_create(&t2, NULL, run, (void *) &cp2);
40
41     pthread_join(t1, NULL);
42     pthread_join(t2, NULL);
43
44     printf("Soma = %d\n", cp1.tot + cp2.tot);
45
46     pthread_exit(NULL);
47 }
```

A estrutura definida nas linhas 4 a 7 declara uma referência para o vetor de inteiros, as posições inicial e final da soma sobre este vetor e uma variável para armazenar o total obtido. Nas linhas 26 a 34 são

criadas e inicializadas duas variáveis, com a configuração dos parâmetros correta para cada thread. Na função filha, o laço da linha 14 usa os valores recebidos por parâmetro para computar a soma da metade do vetor. Na linha 17, o resultado é armazenado na estrutura novamente. A função filha faz uso de ponteiros para a `struct`, visto que as alterações de valores devem ser persistidas na função principal, ao exibir a resposta final, na linha 45.

Ao passar um parâmetro para uma função filha, deve-se garantir que este parâmetro seja *thread-safe*, isto é, não seja modificado por nenhuma outra thread. Por exemplo, no Código 3, ao criar threads na linha 28, uma possibilidade seria passar o endereço da variável `i` para as funções filhas. Porém, isto não foi feito pelo fato de que todas as threads enxergariam a mesma variável `i` durante sua execução. Uma vez que não é possível controlar a ordem de criação e execução das threads, assim que uma delas fizesse uso da variável `i`, seu valor poderia já ter sido alterado por outra thread. Por esta razão, no Código 3, usamos o ponteiro `pi`. Na linha 25 aloca-se memória a ele e na linha 26 atribuímos o valor atual da variável `i`. Só então o ponteiro `pi` é passado como parâmetro para a função filha em `pthread_create()`. Já no Código 4, as threads recebem diretamente o endereço da variável da estrutura, porém, cada thread possui uma instância distinta de variável. O resultado da execução do Código 4 pode ser visto a seguir:

```
aluno@lab:~$ gcc -o saida pthrparamret.c -pthread
aluno@lab:~$ ./saida
Soma = 55
aluno@lab:~$
```

4.3 Threads joinable e detached

Nas linhas 41 e 42 do Código 4, é possível notar a invocação da função `pthread_join()`, utilizada para sincronizar threads. Sua sintaxe é:

```
#include <pthread.h>

int pthread_join(pthread_t thr, void **retval);
```

A função `pthread_join()` aguarda pelo encerramento da thread identificada pelo parâmetro `thr`. Além disso, caso o parâmetro `retval` seja diferente de `NULL`, o valor retornado pela terminação da thread `thr` é copiado nele. Dessa maneira, uma thread mãe pode monitorar o status de encerramento de suas threads filhas. O retorno de `pthread_join()` é zero, em caso de sucesso; em caso de falha, um valor inteiro indicando um código de erro é devolvido.

A tarefa que a função `pthread_join()` realiza em relação a threads é similar à da função `waitpid()` para processos, com as seguintes diferenças:

- Não há uma relação hierárquica entre as threads criadas. Qualquer thread pode invocar a função `pthread_join()` para esperar o encerramento de qualquer outra thread. Por exemplo: se a thread A cria uma thread B, e a thread B cria uma thread C, A pode esperar pelo encerramento de C e vice-versa.
- Não há uma maneira de esperar por "outra thread qualquer". Note que, para processos, a função `wait(NULL)` espera por outro processo qualquer finalizar.

Toda thread que pode ser sincronizada com a função `pthread_join()` é chamada de *joinable*. Contudo, uma thread também pode ser *detached*, isto é, uma thread que não pode ser sincronizada com esta função. Ser "detached" é um atributo da thread que pode ser definido no momento de sua criação.

Para criar uma thread detached é preciso declarar uma variável do tipo `pthread_attr_t` e inicializá-la através da função `pthread_attr_init()`, colocando o atributo detached nessa variável através da função `pthread_attr_setdetachstate()`. Após esta criação, a thread deve ser criada usando esta variável como parâmetro da função `pthread_create()`.

A função `pthread_attr_setdetachstate()` possui a seguinte sintaxe:

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

O parâmetro `detachstate` pode assumir os seguintes valores:

- `PTHREAD_CREATE_DETACHED`: thread criada como detached.
- `PTHREAD_CREATE_JOINABLE`: thread criada como joinable. Este é o comportamento padrão.

O Código 5 apresenta um exemplo de configuração e criação de uma thread detached (linhas 14 a 18: Método 1). Alternativamente, caso a thread já tenha sido criada como joinable, ou com a configuração padrão, a função `pthread_detach()` permite torná-la detached em tempo de execução. Ainda no Código 5, isto pode ser visto nas linhas 21 e 22 (Método 2).

Código 5: `pthrdetach.c`

```
1  #include <pthread.h>
2
3  void *run(void *arg)
4  {
5      ...
6      pthread_exit(NULL);
7  }
8
9  int main()
10 {
11     pthread_t thr;
12
13     // Metodo 1:
14     pthread_attr_t atrib;
15
16     pthread_attr_init(&atrib);
17     pthread_attr_setdetachstate(&atrib, PTHREAD_CREATE_DETACHED);
18     pthread_create(&thr, atrib, run, NULL);
19     ...
20     // Metodo 2:
21     pthread_create(&thr, NULL, run, NULL);
22     pthread_detach(thr);
23     ...
24     pthread_exit(NULL);
25 }
```

A sintaxe da função `pthread_detach()` é:

```c
#include <pthread.h>

int pthread_detach(pthread_t thr);
```

No caso da thread referenciada pelo argumento `thr` ainda não ter encerrado sua execução, ela é setada como detached e a sua execução não é interrompida. O retorno de `pthread_detach()` é zero, em caso de sucesso; em caso de falha, um valor inteiro indicando um código de erro é devolvido.

Uma das principais razões para se criar threads detached é que, uma vez encerrada a sua execução, os recursos a ela alocados podem ser liberados, de forma que o Sistema Operacional possa alocá-los a outras threads que venham a ser criadas.

Exercício 11: Usando a API PThreads, crie um programa que leia um dois inteiros `h` e `m` representando hora e minuto e os repasse para uma outra thread. A thread deve imprimir "Bom dia", "Boa tarde" ou "Boa noite", de acorddo com o valor de hora e minuto recebidos.

Exercício 12: Usando a API PThreads, crie um programa que leia um dois inteiros `a` e `b` do usuário. A seguir, crie duas funções filhas. A primeira função deve computar a soma dos valores pares no intervalo `[a, b]`. A segunda função deve computar o produto dos valores ímpares no intervalo `[a, b]`. As respostas devem ser impressas na tela pela função `main()`.

Exercício 13: Usando a API PThreads, crie um programa que declare um vetor de 20 inteiros aleatórios e o repasse para duas funções filhas. A primeira função deve computar a média dos elementos do vetor. A segunda função deve computar a mediana. As respostas devem ser impressas na tela pela função `main()`.

Exercício 14: Usando a API PThreads, crie um programa que declare um vetor de 20 inteiros aleatórios e o repasse para duas funções filhas. A primeira função deve computar a média dos elementos do vetor. A segunda função deve computar o desvio padrão. As respostas devem ser impressas na tela pela função `main()`.

4.4 Identificação da thread

Todas as threads pertencentes a um processo são univocamente identificadas por um valor de ID. Este ID é retornado ao chamador de `pthread_create()` e pode ser obtido pela thread através da função `pthread_self()`, cuja sintaxe é:

```c
#include <pthread.h>

pthread_t pthread_self(void);
```

O Código 6 mostra o uso da função `pthread_self()`.

O resultado da execução do Código 6 pode ser visto a seguir:

```
aluno@lab:~$ gcc -o saida pthrtid.c -pthread
aluno@lab:~$ ./saida
Thread ID (pthread_self) = 140145335715392
Thread ID (Linux Kernel) = 12460
aluno@lab:~$
```

É importante notar que o valor de Thread ID do POSIX não é o mesmo valor de Thread ID do kernel, retornado pela system call `gettid()`. Os valores de Thread ID do POSIX são gerenciados pela imple-

Código 6: `pthrtid.c`

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *run(void *arg)
{
    printf("Thread ID (pthread_self) = %ld\n", pthread_self());
    printf("Thread ID (Linux Kernel) = %d\n", gettid());
    pthread_exit(NULL);
}

int main()
{
    pthread_t thr;

    pthread_create(&thr, NULL, run, NULL);
    pthread_join(thr, NULL);

    pthread_exit(NULL);
}
```

mentação da NPTL. Já o valor devolvido por `gettid()` é atribuído pelo kernel do SO e, normalmente, uma aplicação não necessita de conhecer o valor de Thread ID do kernel.

Já o POSIX Thread ID é usado por várias outras funções da API PThreads, como `pthread_join()`, `pthread_detach()`, `pthread_cancel()`, `pthread_kill()`, `pthread_equal()`, etc.

Documentação

A documentação de todas as funções e bibliotecas vistas neste guia pode ser encontrada em: `https://pubs.opengroup.org/onlinepubs/9699919799/`

(No frame direto, procure pelo link *Alphabetic Index*. Use este índice para buscar pelo termo que desejar.)