

Aula 10: Processos em sistemas operacionais

Apresentação

Iremos usar a linguagem de programação C para estudar como um programador pode ordenar que várias tarefas (threads) simultâneas/paralelas sejam criadas. Você irá conhecer também as formas de sincronização de threads usando duas das principais técnicas disponíveis. Além disso, estudaremos outra opção de programação paralela: a criação de processos filhos usando a chamada *fork* da linguagem C.

Objetivos

- Definir o conceito de *threads* e as vantagens alcançadas com seu uso;
- Demonstrar as principais armadilhas a serem evitadas em programação com *threads*;
- Descrever outra forma de programar ações paralelas usando *fork* para criação de processos filhos.

Conceitos básicos

As bibliotecas de tarefas/threads POSIX são uma API baseada em padrões para as linguagens de programação C e C++. Esta API permite gerar um novo fluxo de processos simultâneos/paralelos, que são chamados de tarefas, linhas de execução ou threads.

Saiba mais

O uso de threads é mais eficaz em sistemas com vários processadores ou vários núcleos/cores de processamento, nos quais a *thread* pode ser agendada para ser executada em outro processador, ganhando velocidade por meio de processamento paralelo ou distribuído.

Os encadeamentos requerem menos sobrecarga do que bifurcar ou gerar um novo processo filho usando a chamada *fork*, porque o sistema operacional (S.O.) não inicializa um novo espaço e ambiente de memória virtual do sistema para o processo.

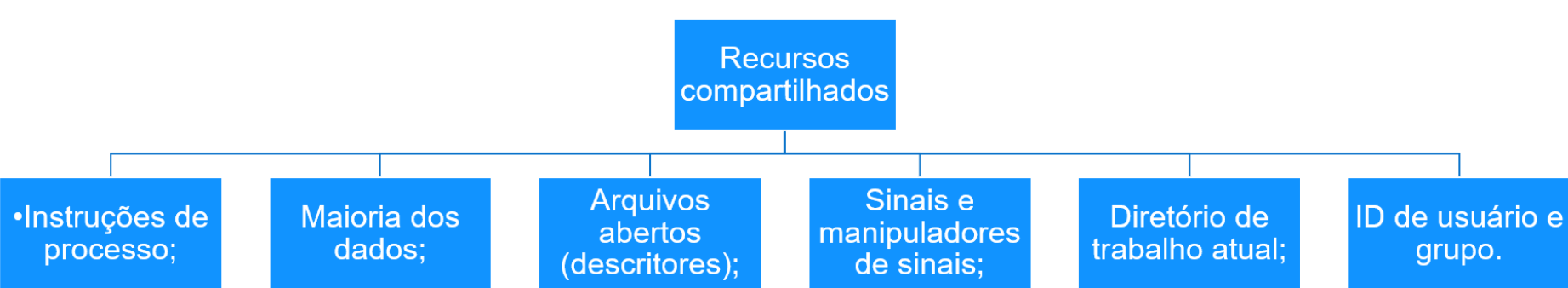
Embora sejam mais eficazes em um sistema multiprocessador, também ocorrem ganhos em sistemas uniprocessadores, que exploram a latência em E/S (entrada e saída) e em outras funções do sistema que podem interromper a execução do processo. Por exemplo, uma *thread* pode ser executada enquanto outra aguarda E/S ou alguma outra latência do sistema.

Tecnologias de programação paralela, como MPI – *Message Passing Interface* – e PVM – *Parallel Virtual Machine* –, são usadas em um ambiente de computação distribuído, enquanto as *threads* são limitadas a um único sistema de S.O. Todos as threads em um processo compartilham o mesmo espaço de endereço.

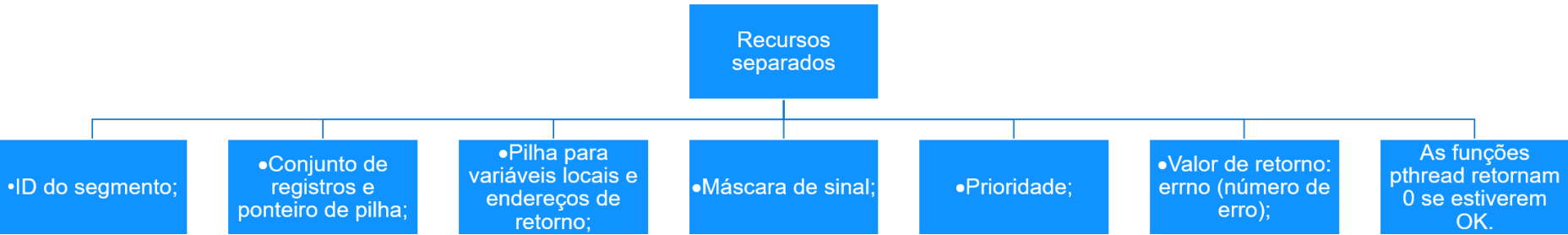
Uma *thread* é gerada ao se **definir uma função e seus argumentos** que serão processados na nova linha de execução. A principal vantagem do uso da biblioteca de *threads* POSIX é executar o software mais rapidamente, através da execução de tarefas em paralelo. É por isso que seu uso se torna muito interessante em sistemas **multicore**.

As operações de *threads* incluem criação, encerramento, sincronização (junções e bloqueio), agendamento, gerenciamento de dados e interação de processos. Uma *thread* não mantém uma lista de *threads* e nem conhece o processo que a criou. Todas essas tarefas compartilham o mesmo espaço de endereço em um processo.

Threads de um mesmo processo compartilham recursos como:



Por outro lado, cada *thread* de um processo possui recursos separados como:



Veja a seguir um exemplo de criação e término de threads:

Exemplo: primeiraThread.c

```
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>

void * imprime_mensagem (void * ptr);

int main()
{
    pthread_t threadUm, threadDois;
    char * mensagem1 = "threadUm: Frase 1";
    char * mensagem2 = "threadDois Frase 2";
    int retorno 1, retorno 2;

    /* Crie threads independentes, cada uma deles executará a função. */

    retorno1 = pthread_create (&threadUm, NULL, imprime_mensagem, (void *) mensagem1);
    retorno2 = pthread_create (&threadDois, NULL, imprime_mensagem, (void *) mensagem2);

    /* Aguarde até que as threads sejam concluídas antes que o main continue. */

    pthread_join (threadUm, NULL);
    pthread_join (threadDois, NULL);

    printf ("A threadUm retornou: %d \n", retorno1);
    printf ("A threadDois retornou: %d \n", retorno2);
    exit(0);
}

void * imprime_mensagem (void * ptr)
{
    char * mensagem;
    mensagem = (char *) ptr;
    printf (" %s \n", mesnagem);
}
```

Como compilar este exemplo:

Compilador C: cc -pthread primeiraThread.c -o primeiraThread

Como executar este exemplo:

Execução: ./primeiraThread

Exemplo de resultado:

| | | |
|---------|----------|------------|
| Frase | | 1 |
| Frase | | 2 |
| Linha | 1 | retorno: 0 |
| Linha 2 | retorno: | 0 |

Pode-se perceber que a função (imprime_mensagem) é usada por ambas as threads, porém, os argumentos são diferentes (mensagem1 e mensagem2). É importante que você tenha em mente que as funções não precisam ser as mesmas.


- Ao final de sua execução, uma *thread* pode ocorrer por meio de uma das formas listadas a seguir:
- Chamar explicitamente pthread_exit();
- Permitir que a função retorne;
- Chamar para a saída da função que encerrará o processo, incluindo qualquer *thread* em execução.

No código anterior, usamos a chamada pthread_create para criar threads. O protótipo dessa função, disponível na API pthreads POSIX, pode ser visto abaixo.

```
int pthread_create (pthread_t * thread, const pthread_attr_t * attr, void * (* start_routine) (void *), void * arg);
```

Sincronização de Threads

A biblioteca de threads fornece três mecanismos de sincronização:

 Clique nos botões para ver as informações.

Mutexes



Termo originado do inglês *mutual exclusion*, que significa bloqueio de exclusão mútua. Essa ferramenta bloqueia o acesso de outras threads a variáveis e impõe o acesso exclusivo de um encadeamento a uma variável ou conjunto de variáveis. É imprescindível que haja cuidado do desenvolvedor para evitar o problema de deadlock;

Junções(join).



Faz uma thread aguardar até que outras sejam finalizadas;

Variáveis de condição



Tipo de dados pthread_cond_t.

Mutexes

São usados para evitar inconsistências de dados devido ao acesso concorrente. Esse, geralmente, ocorre quando duas ou mais *threads* precisam executar operações na mesma área de memória, mas os resultados dos cálculos dependem da ordem em que essas ações são executadas.

São usados também para serializar recursos compartilhados. Sempre que um recurso global é acessado por mais de uma *thread*, o recurso deve ter um *mutex* associado a ele. Pode-se aplicar um *mutex* para proteger um segmento de memória (região crítica) de outras *threads*. Esta solução pode ser aplicada apenas a *threads* em um único processo e não funcionam entre processos, como os semáforos. Observe o exemplo a seguir.

Exemplo: meuMutex.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * funcaoC ();
pthread_mutex_t meuMutex = PTHREAD_MUTEX_INITIALIZER;
contador int = 0;

main ()
{
    int retorno1, retorno2;
    pthread_t threadUm, threadDois;

    / * Crie threads independentes. Cada uma delas executará funcaoC. * /

    if ((retorno1 = pthread_create (&threadUm, NULL, &funcaoC, NULL)))
    {
        printf ("Falha na criação do thread: %d \n", retorno1);
    }

    if ((retorno2 = pthread_create (&threadDois, NULL, &funcaoC, NULL)))
    {
        printf ("Falha na criação do thread: %d \n", retorno2);
    }

    / * Aguarde até que as threads sejam concluídas antes de continuar com o main(). Caso não
    esperemos, corremos o risco de terminar o main() antes das threads. Com isso, elas seriam
    finalizadas antes de terminarem seu processamento/tarefa. */
    pthread_join (threadUm, NULL);
    pthread_join (threadDois, NULL);

    exit(0); /* Finaliza o processo e todas as threads em execução.*/
}

void * funcaoC ()
{
    pthread\_mutex\_lock (&mutex1);
    contador++;
    printf ("Valor do contador: %d \n", contador);
    pthread\_mutex\_unlock (&meuMutex);
}
```

Compilar: cc -pthread meuMutex.c -o meuMutex

Execução: ./meuMutex

Exemplo de Resultados:

Valor do contador: 1

Valor do contador: 2

Quando um *thread*, como o threadA, tenta manipular um bloqueio de um *mutex* que já foi bloqueado por outra *thread*, o threadA é bloqueado até que o *mutex* seja liberado.

Quando um *thread* termina, o *mutex* não é terminado, a menos que seja explicitamente desbloqueado. Nada acontece por padrão. Se o desenvolvedor não tomar o devido cuidado e permitir que um thread termine sem desbloquear um *mutex* bloqueado, então, outros *threads* podem ficar indefinidamente esperando o desbloqueio, o que gerará um *deadlock*.

Junções (Join)

São realizadas quando se deseja aguardar o término de um *thread*. Uma rotina de chamada de *thread* pode iniciar vários *sub threads* e esperar que eles terminem para obter os resultados. A chamada pthread_join é usada para ordenar a espera pela conclusão dos threads criados.

Note o comentário, no código-fonte de meuMutex.c, acima das chamadas pthread_join, para os *threads* threadUm e threadDois. Conforme explicado no comentário, não usamos *join* para esperar o término de *threads*, pois corremos o risco de a função main() finalizar todo o processo antes dos *threads* mencionados. Com isso, eles seriam finalizados antes de terminarem seu processamento/tarefa.

Exemplo de uso de pthread_join:

Exemplo de uso de pthread_join:

```
#include <stdio.h>
#include <pthread.h>

#define MAXTHREADS 10
void *funcaoC (void *);
pthread_mutex_t meuMutex = PTHREAD_MUTEX_INITIALIZER;
contador int = 0;

main ()
{
    pthread_t identificacaoDaThread [ MAXTHREADS ];
    int i, j;

    for (i = 0; i<MAXTHREADS; i ++ )
    {
        pthread_create (&identificacaoDaThread[ i ], NULL, funcaoC, NULL);
    }

    for (j = 0; j<MAXTHREADS; j ++ )
    {
        pthread_join (identificacaoDaThread [j], NULL);
    }
    /* Agora que todas as threads estão completas, posso imprimir o resultado final. Sem o uso da
    junção, o printf abaixo poderia ser chamado antes de todas as threads sejam concluídas e imprimir o
    valor final incorreto. * /
```

```
printf ("Valor final do contador: %d \n", contador);
}

void *funcaoC(void *meuPTR)
{
    printf ( "Número desta thread: %ld \n", pthread\_self\(.\) );
    pthread_mutex_lock( &meuMutex );
    contador++;
    pthread_mutex_unlock( &meuMutex );
}
```

Compilar: cc -pthread meuJoin.c -o meuJoin

Execução: ./meuJoin

Exemplo de Resultados:

Número desta thread: 1561

Número desta thread: 2603

Número desta thread: 3654

Número desta thread: 4643

Número desta thread: 5578

Número desta thread: 6007

Número desta thread: 7001

Número desta thread: 8991

Número desta thread: 9554

Número desta thread: 10941

Valor final do contador: 10

Agora, suponha que o processamento realizado dentro da função `funcaoC` leve um tempo considerável. Então, caso não usemos o *join*, o *main()* pode terminar antes das *threads*.

Você deve ter observado que a constante `MAXTHREADS` foi configurada para 10 e, por isso, dez threads são executadas. No exemplo de resultados impressos na tela, há dez linhas iniciando com “Número desta thread” e, ao final, o valor final do contador é igual a 10.

Caso o *join* não fosse usado e o *main()* terminasse antes das threads, você notaria, nos resultados, menos de dez linhas iniciando com “Número desta thread”, e o valor final do contador seria menor do que 10.

Este é um bom exercício para você mesmo fazer em seu computador: remova os *joins*, aumente o valor de `MAXTHREADS` e execute o programa final até que você note a ocorrência do fenômeno em questão.

Condições de corrida

Embora o código possa aparecer na tela na ordem de execução que você deseja, as *threads* são agendadas pelo S.O. e executadas aleatoriamente. Não se pode presumir que elas sejam executadas na ordem em que são criadas. Também podem ser executadas em velocidades diferentes.

Quando as *threads* estão em execução, ou seja, competindo para finalizar, elas podem fornecer resultados inesperados, como a condição de corrida. *Mutexes* e junções (*joins*) devem ser utilizados para obter uma ordem e resultado previsíveis de execução.

Código de segurança de thread

As rotinas de *thread* devem chamar funções que são do tipo *thread-safe*. Isso significa que não há variáveis estáticas ou globais que outras *threads* possam escrever ou ler, assumindo que seriam uma operação única.

Se variáveis estáticas ou globais forem usadas, os *mutexes* deverão ser aplicados ou as funções deverão ser reescritas para evitar o uso dessas variáveis. Em C, variáveis locais são alocadas dinamicamente na pilha. Portanto, qualquer função que não use dados estáticos ou outros recursos compartilhados é segura para *threads*.

As funções inseguras podem ser usadas por apenas uma *thread* de cada vez em um programa e a exclusividade deve ser garantida. Muitas funções não-reentrantes retornam um ponteiro para dados estáticos. Isso pode ser evitado retornando dados alocados dinamicamente ou usando o armazenamento fornecido pelo chamador. Um exemplo de uma função não-segura para threads é a **strtok**, usada para manipular strings. A versão segura dessa função é a reentrante *strtok_r*.

Deadlock do mutex

Um *deadlock* ocorre quando um bloqueio/*lock* é aplicado a um *mutex*, mas não ocorre desbloqueio, fazendo com que outra thread, que aguarda pela liberação do *mutex*, espere indefinidamente.

Deadlocks também podem ser causados pela má aplicação de *mutexes* ou junções. Tenha cuidado ao aplicar dois ou mais deles a uma seção do código. Se o primeiro bloqueio for aplicado e o segundo falhar, devido ao *mutex* já estar bloqueado por outra thread, o primeiro pode eventualmente impedir que todas as threads acessem dados, incluindo a que contém o segundo *mutex*.

Threads podem esperar indefinidamente pela liberação do recurso, causando conflito. É melhor testar e, se ocorrer uma falha, libere os recursos e pare para refletir e ajustar o código-fonte, antes de tentar novamente.

Criando processos filhos usando o fork

Uma outra abordagem para programação de ações paralelas é através de criação de processos filhos usando a chamada *fork*. Quando o *fork* é chamado, o S.O. cria um processo: atribui uma nova entrada na tabela de processos e clona as informações da atual. Todos os descritores de arquivo abertos no pai serão abertos no filho. A imagem da memória executável também é copiada.

Assim que a chamada da bifurcação, ou seja, a criação de um processo filho, retornar, o pai e o filho estarão em execução paralela no mesmo ponto do programa. A única diferença é que o processo filho obtém um valor de retorno diferente do *fork*. Enquanto o processo pai obtém o ID do processo do filho que acabou de ser criado, esse obtém um retorno de 0. Se a bifurcação retornar -1, significa que o S.O. não pôde criar o processo.

Atenção

O ponto crucial a ser observado com o *fork* é que nada é compartilhado após sua chamada. Mesmo que o filho esteja executando as mesmas linhas de instrução e tenha os mesmos arquivos abertos, ele mantém seus próprios ponteiros de busca (posições no arquivo) e possui sua própria cópia de toda a memória. Se um processo filho alterar um local de memória, os pais não verão a alteração e vice-versa, já que possuem área de memória distinta.

Exemplo de uso de fork

Exemplo: meuFork.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char ** argv)
{</head>}
    int IdentificacaoDoProcesso; /*PID - Identificação do processo junto ao S.O.*/
    IdentificacaoDoProcesso = fork ();

    if( IdentificacaoDoProcesso == 0 ) /*Este é o processo Filho*/
    {
        printf ("Eu sou o processo filho: PID = %d \n", getpid() );
    }
    else if ( IdentificacaoDoProcesso > 0 ) /* Este é o processo pai*/
    {
        printf ("Eu sou o processo pai com PID = %d, o PID do filho é = %d \n", getpid(),
            IdentificacaoDoProcesso);
    }
    else
    {
        printf( "Erro ao tentar criar processo filho\n" );
        perror ( "fork" );
        exit( 1 );
    }
    exit( 0 );
}
```

Compilar: cc meuFork.c -o meuFork

Execução: ./meuFork

Exemplo de Resultados:

Eu sou o processo pai com PID = 24550, o PID do filho é = 24551

Eu sou o processo filho: PID = 24551

Atividade

1. Marque a opção que representa corretamente a principal vantagem obtida ao programar paralelismo em um software:

- a) O resultado do processamento é mais rapidamente alcançado, principalmente, em sistemas com várias CPUs, como CPU *multicore*.
 - b) O processamento é mais seguro, em especial, em sistemas monoprocessados ou de CPUs com apenas um core.
 - c) O resultado do processamento é mais preciso, sobretudo, em sistemas com várias CPUs ou em CPUs *multicore*.
 - d) O resultado do processamento é alcançado mais rapidamente, em geral, em sistemas com CPU única ou *single-core*.
 - e) O processamento é mais seguro, principalmente, em sistemas com várias CPUs, como CPU *multicore*.
-

2. Dos recursos apresentados abaixo, identifique os que são compartilhados por duas *threads* rodando em paralelo no mesmo processo. Depois, some os valores que se encontram à frente de cada opção e marque a alternativa correspondente:

- Instruções de processo
- Conjunto de registros e ponteiro de pilha
- Maioria dos dados
- Arquivos abertos (descritores)
- Sinais e manipuladores de sinais
- Diretório de trabalho atual
- Conjunto pilha para variáveis locais e endereços de retorno
- ID de usuário e grupo

- a) 103
 - b) 254
 - c) 63
 - d) 127
 - e) 189
-

3. Considerando as principais armadilhas do tratamento de sincronização de *threads*, o que ocorre quando a **thread1** espera pelo desbloqueio de um *mutex*, que estava bloqueado pela **thread2**, sendo que essa finalizou seu processamento sem efetuar a liberação do *mutex*?
- a) O sistema operacional desbloqueia automaticamente o *mutex* quando a *thread*, que o bloqueava, finaliza.
 - b). O thread1 deve chamar a função *join*, usada para desbloquear forçadamente o *mutex*.
 - c) O processo pai deve detectar a situação e desbloquear o *mutex* em questão.
 - d) Não há como contornar a questão sem ajustes no código-fonte, pois somente a thread2 poderia realizar a liberação.
 - e) O programa será finalizado pelo sistema operacional com a mensagem: *sementation fault*.
-

4. Quando se trata de programação com *threads*, o termo condições de corrida é o nome dado:
- a) Ao ganho de desempenho alcançado pela execução paralela de *threads*.
 - b) À concorrência por um recurso, como por exemplo, uma variável, que pode ocorrer quando *threads* rodam paralelamente. Com isso, podem ocorrer resultados inesperados pelo programador.
 - c) Ao fato de que a forma como o S.O. escalona a execução de *threads* é, aos olhos do programador, aleatória. Com isso, podem ocorrer resultados, inicialmente, inesperados pelo programador.
 - d) À espera do programa principal (*main()*) pela finalização da execução das *threads* que foram demandadas. Desta forma, o programador evita resultados inesperados, como o término do programa principal antes que as *threads* finalizem suas tarefas.
 - e) Não será fornecida nenhuma garantia.
-

5. Uma outra forma de se programar a execução paralela de tarefas é o *fork*. Qual é a principal diferença de prática quanto ao uso dessa solução, e não de *threads*?
- a) Mesmo com os processos rodando de forma independente e em regiões distintas da memória, o programador terá a mesma preocupação com questões de sincronização usando, por exemplo, o *mutex*.
 - b) Como o *fork* cria processos filhos que rodam independentemente dos processos pais, o programador não precisará se preocupar em programar com detalhes de sincronização, como ocorre no uso de *threads*.
 - c) O uso de *forks* é mais interessante do ponto de vista de quantidade de memória RAM usada, dado que processos pai e filhos compartilham a mesma região da memória.
 - d) O uso de *fork* gera *threads* contidas em um mesmo processo, mas delega ao Sistema Operacional a tarefa de sincronização.
 - e) O uso de *fork* gera *threads* em diferentes processos e delega ao Sistema Operacional a tarefa de sincronização.
-

Notas

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

Referências

TANENBAUM, A. S.; BOS, H. **Sistema Operacionais Modernos**. Cap. 1, 3ª ed.

STEVENS, W. R.; FENNER, B.; RUDOFF, A. M. **Programação de Rede Unix**. 3ª ed. Porto Alegre: Bookman, 2008.

Próxima aula

Explore mais

Assista ao vídeo:

- [Depurando com múltiplas threads](#).

Leia o texto:

- [Depurando com a Ferramenta GDB](#).