

Sistemas Operacionais Threads

Profª Drª Thaína Aparecida Azevedo Tosta
tosta.thaina@unifesp.br

Aula passada

- O modelo de processo
- Criação de processos
- Término de processos
- Estados de processos
- Implementação de processos

Sumário

- Utilização de threads
- O modelo de thread clássico
- Implementando threads no espaço do usuário
- Implementando threads no núcleo
- Implementações híbridas
- Threads pop-up
- Convertendo código de um thread em código multithread

Objetivo: conhecer threads e suas implementações, e questão de fixação.

Utilização de threads

As threads são miniprocessos com uso motivado por:

1. Decomposição de uma aplicação em múltiplos threads com execução quase em paralelo → modelo de programação simples;
2. Compartilhamento do espaço de endereçamento e de todos os dados (vs processos com espaços separados);
3. Mais fáceis e rápidos de criar e destruir que processos;
4. Melhor desempenho pela sobreposição de atividades;
5. Permite um paralelismo real em sistemas com múltiplas CPUs.

Utilização de threads

Exemplo: escrita de um livro sem linhas incompletas no início e no fim das páginas, mantendo o livro inteiro em um único arquivo (facilita busca por tópicos e substituições globais).

- Autor(a) apaga uma frase da página 1 do livro de 800 páginas;
- Autor(a) faz modificação na primeira linha da página 600;
- Implicações: o processo precisa reformatar até a página 600 → atraso para exibir a página 600 a ser modificada.

Por que não três processos?
Compartilhamento

Thread 2: reformatação em segundo plano

Four score and seven years ago, our fathers brought forth upon this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that	union, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.	lives that this nation might live. It is altogether fitting and proper that we should do this.	who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated	here to the unfinished work which they who fought here have thus far so nobly advanced. It is our duty, as well as our honor, to complete the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain, that this nation, under God, shall have a new birth of freedom, and that the government of the people, by the people, for the people
--	---	--	---	--

Thread 1: interação com usuário

Núcleo

Disco

Thread 3: backups da RAM para o disco periodicamente

Utilização de threads

Exemplo: Planilha eletrônica com dados fornecidos pelo usuário e outros calculados com base nos dados de entrada usando fórmulas potencialmente complexas.

Pessoa altera um dado → recálculo de outros.

Thread 1: thread interativo;

Thread 2: recálculo;

Thread 3: backups periódicos para o disco.

Utilização de threads

Exemplo: servidor web com thread despachante (a - lê requisições de trabalho que chegam da rede) e thread operário (b - mudança de estado bloqueado para pronto quando recebe solicitação).

- Quando o operário desperta, ele verifica o cache da página da web (acesso de todos os threads) e bloqueia até nova solicitação;
- Se não, uma operação read consegue a página do disco e bloqueia o operário até a conclusão → outro thread é escolhido para ser executado;
- Despachante: laço infinito para requisições de trabalho e entregá-las a um operário;
- Operário: laço infinito que aceita uma solicitação de um despachante.

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

Servidor com thread única?

O modelo de thread clássico

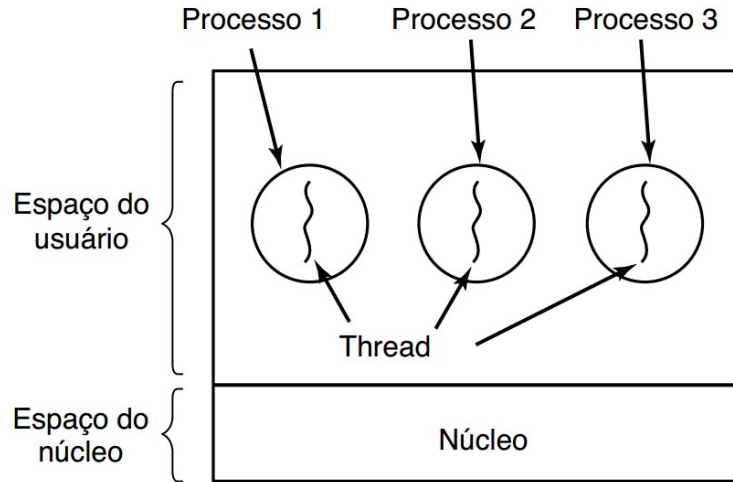
O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução. Às vezes é útil separá-los; é onde os threads entram (múltiplas execuções independentes no mesmo ambiente).

Processo	Thread/ Processos leves
<p>Modo para agrupar recursos relacionados; Contém:</p> <ul style="list-style-type: none">• Espaço de endereçamento (código + dados de programa);• Recursos (arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais, informações sobre contabilidade, etc).	<p>Linha de execução; Contém:</p> <ul style="list-style-type: none">• Contador de programa (qual a próxima instrução?);• Registradores (para variáveis de trabalho atuais);• Pilha (histórico de execução).

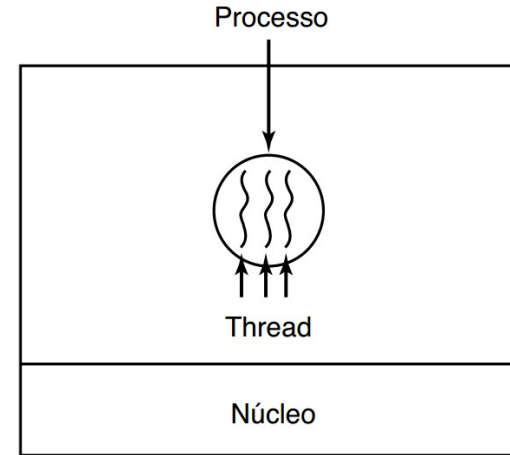
O modelo de thread clássico

Cada thread opera em um espaço de endereçamento.

Threads compartilham o mesmo espaço de endereçamento.



(a)



(b)

- Quando um processo multithread é executado em um sistema de CPU única, os threads se revezam executando (chaveamento com ilusão de execução em paralelo), como os processos.

O modelo de thread clássico

- Threads diferentes em um processo não são tão independentes quanto processos diferentes;
- Eles têm o mesmo espaço de endereçamento (mesmas variáveis globais) dentro do espaço de endereçamento do processo → um thread pode ler, escrever, ou mesmo apagar a pilha de outro thread;
- Não há proteção entre threads.

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

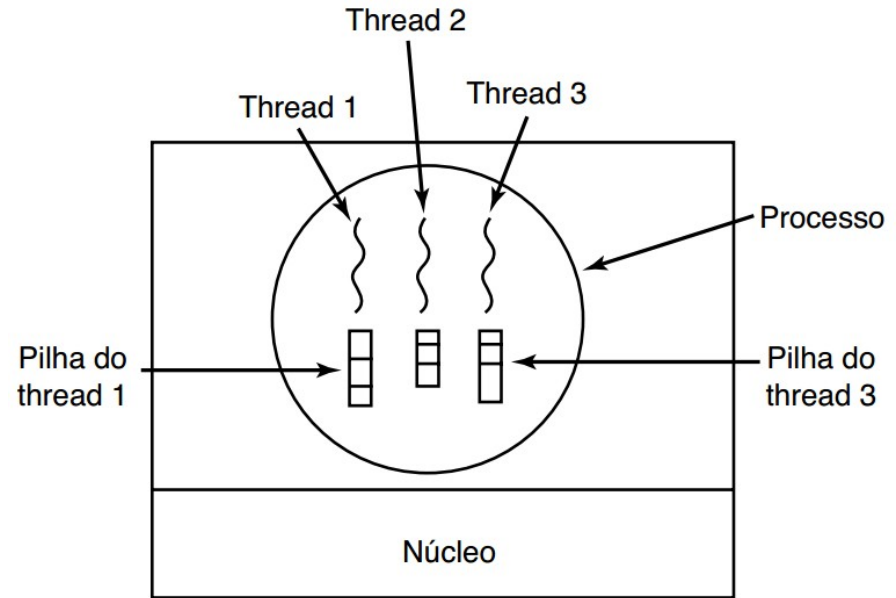
O modelo de thread clássico

Um thread pode estar em qualquer um de vários estados (como os processos):

- Em execução: está ativo e com a CPU;
- Bloqueado: esperando por algum evento externo ou outro thread para desbloqueá-lo;
- Pronto: programado para ser executado e o será tão logo chegue a sua vez;
- Concluído.

O modelo de thread clássico

- É importante perceber que **cada thread tem a sua própria pilha**, contendo uma estrutura para cada rotina chamada mas ainda não retornada;
- Essa estrutura contém as variáveis locais da rotina e o endereço de retorno para usar quando a chamada de rotina for encerrada;
- Cada thread geralmente chamará rotinas diferentes e desse modo terá uma história de execução diferente.



O modelo de thread clássico

- Com o multithreading, os processos normalmente começam com um único thread presente com a capacidade de criar novos (`thread_create`);
- Quando um thread tiver terminado o trabalho, pode concluir sua execução com `thread_exit`;
- Em alguns sistemas, um thread pode esperar pela saída de um thread (específico) com `thread_join` (envolve bloqueios);
- `thread_yield` permite que um thread abra mão voluntariamente da CPU para deixar outro thread ser executado.
 - Não há uma interrupção de relógio para forçar a multiprogramação.

O modelo de thread clássico

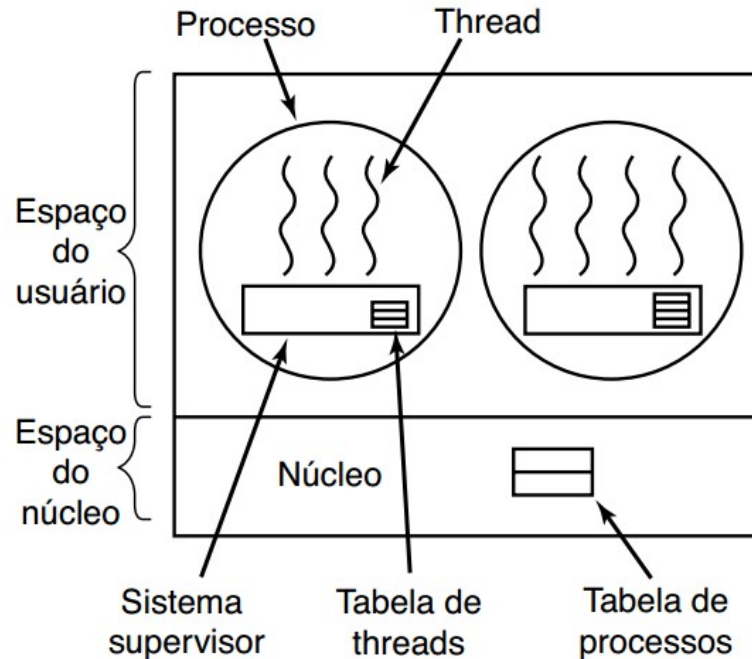
Embora threads sejam úteis, eles também introduzem complicações no modelo de programação:

- Se o processo filho possuir tantos threads quanto o pai, o que acontece se um thread no pai estava bloqueado em uma chamada read de um teclado? Dois threads estão agora bloqueados no teclado? Quando uma linha é digitada, qual recebe uma cópia?
- O que acontece se um thread fecha um arquivo enquanto outro ainda está lendo dele?
- Se um thread começa a alocar mais memória e um outro também, a memória provavelmente será alocada duas vezes.

Implementando threads no espaço de usuário

- Até onde o núcleo sabe, ele está gerenciando processos de um único thread;
- Cada processo precisa da sua própria tabela de threads.
- Vantagens:
 - Possível em SOs sem suporte a threads por bibliotecas (comum);
 - Chaveamento e escalonamento de thread mais rápidos que pelo núcleo;
 - Cada processo tem seu próprio algoritmo de escalonamento.
- Desvantagens:
 - Implementação de chamadas bloqueantes (um thread lê de um teclado antes que quaisquer teclas tenham sido acionadas → bloqueio do processo);
 - Fim voluntário das execuções.

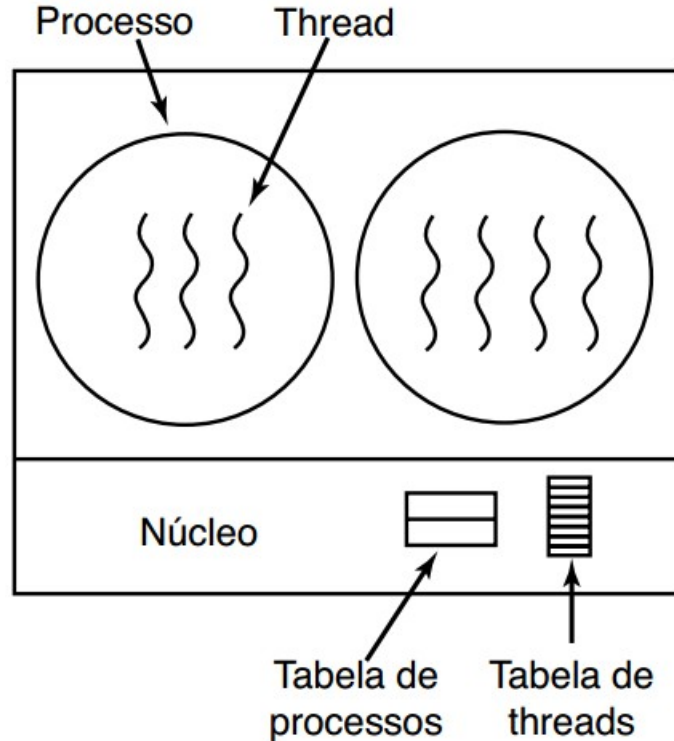
Implementando threads no espaço de usuário



Implementando threads no núcleo

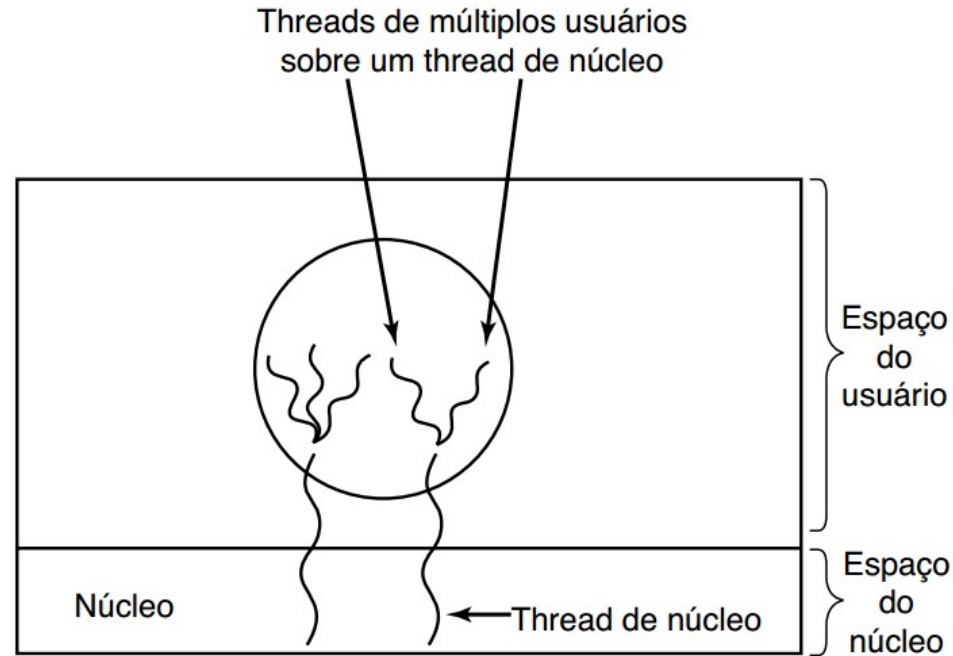
- O núcleo sabe sobre os threads e os gerencia;
- Criar e destruir um thread → chamada de núcleo, atualizando a tabela de threads do núcleo;
- Vantagens:
 - Alguns sistemas reciclam seus threads (threads destruídos sem efeitos nos dados de núcleo);
 - Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes.
- Desvantagens:
 - Todas as chamadas que bloqueiam um thread são implementadas como chamadas de sistema;
 - O que acontece quando um processo com múltiplos threads é bifurcado? O novo processo tem tantos threads quanto o antigo, ou possui apenas um?
 - Os sinais são enviados para os processos, então qual thread deve cuidar deles?

Implementando threads no núcleo



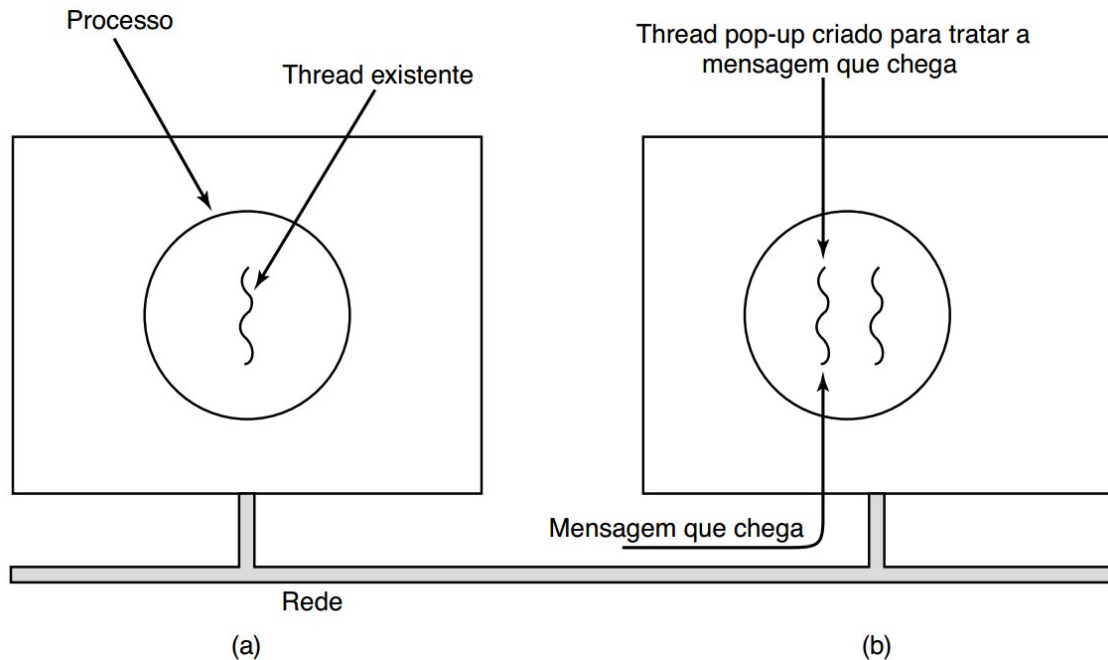
Implementações híbridas

- Combinar vantagens de threads de usuário com threads de núcleo:
- Usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles;
- Máxima flexibilidade: quem programa determina quantos threads de núcleo usar e quantos threads de usuário multiplexar para cada um.



Threads pop-up

- Threads costumam ser úteis em sistemas distribuídos;
 - Tratamento de mensagens/requisições de serviços.
- Abordagem tradicional: processo ou thread bloqueado na chamada recebe esperando;
- Abordagem alternativa: thread pop-up criada para lidar com a mensagem.



Threads pop-up

- **Vantanges:**

Threads novos → sem restauração de estruturas (registradores, pilha, etc) → criação mais rápida → latência entre a chegada da mensagem e o começo do processamento pode ser encurtada.

- **Planejamento:**

Em qual processo o thread é executado? Se o sistema dá suporte a threads sendo executados no contexto núcleo, o thread pode ser executado ali.

Vantagens	Desvantagens
<ul style="list-style-type: none">• Execução mais fácil e mais rápida do que no espaço do usuário;• Fácil acesso a todas as tabelas do núcleo e aos dispositivos de E/S (necessários para interrupções).	<p>Se erro, mais danos que no espaço de usuário (se ele for executado por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos para sempre).</p>

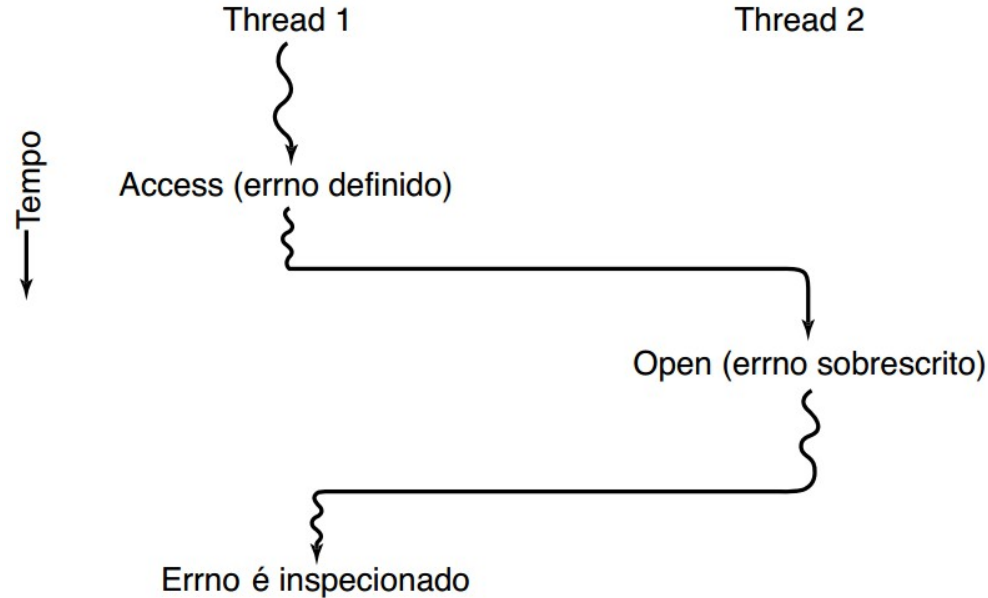
Convertendo código de um thread em código multithread

- Muitos programas existentes foram escritos para processos monothread;
- O código de um thread em geral consiste em múltiplas rotinas, com variáveis locais, variáveis globais e parâmetros.
- Tornar um código multithread envolve questões com:
 - Variáveis globais;
 - Chamadas reentrantes;
 - Sinais;
 - Gerenciamento de pilha.

Convertendo código de um thread em código multithread

Variáveis globais

1. O thread 1 executa a chamada de sistema `access`;
2. O sistema operacional retorna a resposta na variável global `errno` (armazena código de erro);
3. O escalonador decide que o thread 1 teve tempo de CPU suficiente;
4. O escalonador troca para o thread 2;
5. O thread 2 executa uma chamada `open` que falha, o que faz que `errno` seja sobrescrito.

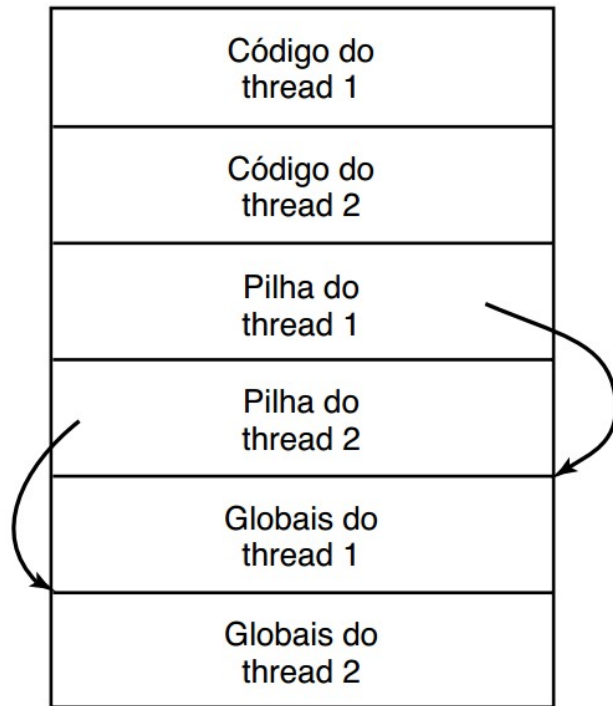


Convertendo código de um thread em código multithread

Variáveis globais

Possíveis soluções:

- Proibir completamente variáveis globais;
- Designar a cada thread as suas próprias variáveis globais privadas (cópia privada de erro e outras variáveis globais) → novo escopo → maior complexidade para linguagens de programação;
- Passar variável global como parâmetro para cada rotina → funcional mas deslegante.



Convertendo código de um thread em código multithread

Variáveis globais

Possíveis soluções:

Novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais restritas ao thread:

```
create_global("bufptr");  
set_global("bufptr", &buf);  
bufptr = read_global("bufptr").
```

Convertendo código de um thread em código multithread

Chamadas reentrantes

- Muitas rotinas de biblioteca não foram projetadas para ter uma segunda chamada enquanto uma anterior ainda não foi concluída;
- Exemplos:
 - Um thread montou a sua mensagem no buffer para envio posterior, então, uma interrupção de relógio força um chaveamento para um segundo thread que imediatamente sobrescreve o buffer com sua própria mensagem;
 - Enquanto malloc está ocupada atualizando uma lista encadeada de pedaços de memória disponíveis, ela pode temporariamente estar em um estado inconsistente, com ponteiros que apontam para lugar nenhum.

Convertendo código de um thread em código multithread

Chamadas reentrantes

Possíveis soluções:

Reescrever toda a biblioteca → não trivial → possível introdução de erros sutis;

Proteção que altera um bit para indicar que a biblioteca está sendo usada → bloqueio de threads que tentam usar rotina de biblioteca enquanto a chamada anterior não tiver sido concluída → elimina muito o paralelismo em potencial.

Convertendo código de um thread em código multithread

Sinais

- Quando threads são implementados inteiramente no espaço de usuário, o núcleo mal pode dirigir o sinal para o thread certo;
- Sinais como uma interrupção de teclado não são específicos aos threads: Quem deveria pegá-los? Um thread designado? Todos os threads? Um thread pop-up recentemente criado?
- Se um thread mudar os tratadores de sinal sem contar para os outros threads?

Em geral, sinais já são suficientemente difíceis para gerenciar em um ambiente de um thread único.

Convertendo código de um thread em código multithread

Gerenciamento de pilha

- Em muitos sistemas, quando a pilha de um processo transborda, o núcleo apenas fornece àquele processo mais pilha automaticamente;
- Se o núcleo não tem ciência de todas as pilhas de um processo com múltiplas threads, isso não é possível.

Convertendo código de um thread em código multithread

Como lidar com tantos problemas então?

- Introduzir threads em um sistema existente sem uma alteração substancial do sistema não vai funcionar;
- No mínimo, as semânticas das chamadas de sistema talvez precisem ser redefinidas e as bibliotecas reescritas;
- Alterações devem ser compatíveis com programas já existentes com processo de um thread.

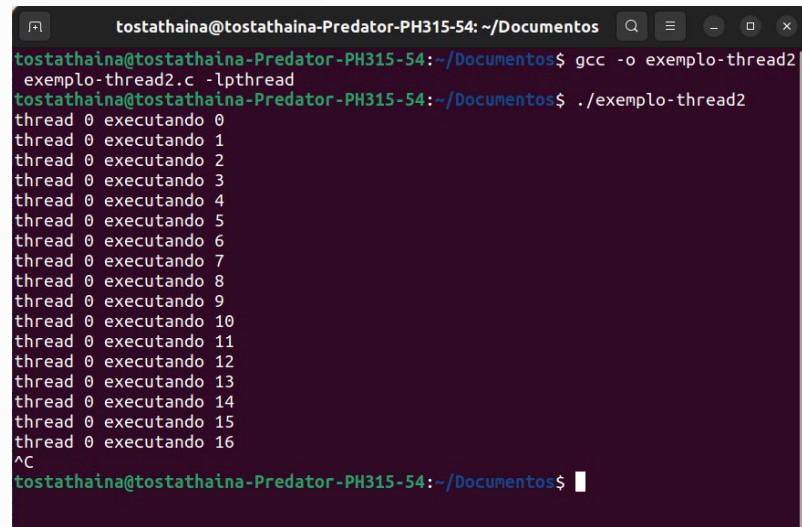
Prática - Thread única

```
1#include <stdio.h>
2#include <pthread.h>
3#include <stdlib.h>
4
5void *t1(void *args){
6    int i=0;
7    int *ret;
8    ret = calloc(1, sizeof(int));
9    *ret = 10;
10   while(1) {
11       printf("t1 executando %d\n", i++);
12       if (i == 1000) pthread_exit(ret);
13   }
14   pthread_exit(ret); //https://man7.org/linux/man-pages/man3/pthread_exit.3.html
15}
16
17int main(int argc, char **argv){
18    pthread_t tid;
19    int *ret_val;
20    pthread_create(&tid, NULL, t1, NULL); //https://man7.org/linux/man-pages/man3/pthread_create.3.html
21    pthread_join(tid, (void*)&ret_val); //https://man7.org/linux/man-pages/man3/pthread_join.3.html
22    printf("main ret_val = %d \n", *ret_val);
23    return 0;
24}
```

```
tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-thread1
exemplo-thread1.c -lpthread
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-thread1
t1 executando 0
t1 executando 1
t1 executando 2
t1 executando 3
t1 executando 4
t1 executando 5
t1 executando 6
t1 executando 7
t1 executando 8
t1 executando 9
t1 executando 10
t1 executando 11
t1 executando 12
t1 executando 13
t1 executando 14
t1 executando 15
t1 executando 16
t1 executando 17
t1 executando 18
t1 executando 19
t1 executando 20
```

Prática - Thread com vários parâmetros

```
1#include <stdio.h>
2#include <pthread.h>
3#include <stdlib.h>
4#include <unistd.h>
5struct targs {
6    int id;
7    int usecs;
8};
9typedef struct targs targs_t;
10
11void *t1(void *args){
12    targs_t ta = *(targs_t *)args;
13    int i=0;
14    int *ret;
15    ret = calloc(1, sizeof(int));
16    *ret = 10;
17    while(1) {
18        printf("thread %d executando %d\n", ta.id, i++);
19        usleep(ta.usecs);
20    }
21    pthread_exit(ret); //https://man7.org/linux/man-pages/man3/pthread_exit.3.html
22}
23
24int main(int argc, char **argv){
25    pthread_t tid;
26    targs_t ta = {0, 1000000};
27    int *ret_val;
28    pthread_create(&tid, NULL, t1, (void*)&ta); //https://man7.org/linux/man-pages/man3/pthread_create.3.html
29    pthread_join(tid, (void**)&ret_val); //https://man7.org/linux/man-pages/man3/pthread_join.3.html
30    printf("main ret_val = %d \n", *ret_val);
31    return 0;
32}
```



The terminal window shows the compilation and execution of the program. The command `gcc -o exemplo-thread2 exemplo-thread2.c -lpthread` is used to compile the program. The output shows 17 threads (0 to 16) executing sequentially, each printing its ID and the number of iterations (0 to 16). The program is executed with `./exemplo-thread2`.

```
tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-thread2
exemplo-thread2.c -lpthread
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-thread2
thread 0 executando 0
thread 0 executando 1
thread 0 executando 2
thread 0 executando 3
thread 0 executando 4
thread 0 executando 5
thread 0 executando 6
thread 0 executando 7
thread 0 executando 8
thread 0 executando 9
thread 0 executando 10
thread 0 executando 11
thread 0 executando 12
thread 0 executando 13
thread 0 executando 14
thread 0 executando 15
thread 0 executando 16
^C
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$
```


Prática - Múltiplas threads

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #define MAX_T 2
6 struct targs {
7     int id;
8     int usecs;
9 };
10 typedef struct targs targs_t;
11 void *func_thread(void *args){
12     targs_t ta = *(targs_t *)args;
13     int i=0;
14     int *ret;
15     ret = calloc(1, sizeof(int));
16     *ret = 10;
17     while(1) {
18         printf("thread %d executando %d\n", ta.id, i++);
19         usleep(ta.usecs);
20     }
21 }
22 int main(int argc, char **argv){
23     pthread_t tid[MAX_T];
24     targs_t ta[MAX_T];
25     int *ret_val[MAX_T];
26     int i;
27     for (i=0; i<MAX_T; i++) {
28         ta[i].id = i;
29         if (i%2) ta[i].usecs = 100000;
30         else ta[i].usecs = 10000;
31         pthread_create(&tid[i], NULL, func_thread, (void*)&ta[i]);
32     }
33     for (i=0; i<MAX_T; i++) {
34         pthread_join(tid[i], (void**)&ret_val[i]);
35         printf("main ret_val[%d] = %d \n", i, *ret_val[i]);
36     }
37     return 0;
38 }
```

```
tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-thread3
exemplo-thread3.c -lpthread
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-thread3
thread 0 executando 0
thread 1 executando 0
thread 0 executando 1
thread 0 executando 2
thread 0 executando 3
thread 0 executando 4
thread 0 executando 5
thread 0 executando 6
thread 0 executando 7
thread 0 executando 8
thread 0 executando 9
thread 1 executando 1
thread 0 executando 10
thread 0 executando 11
thread 0 executando 12
thread 0 executando 13
thread 0 executando 14
thread 0 executando 15
thread 0 executando 16
thread 0 executando 17
thread 0 executando 18
thread 0 executando 19
thread 1 executando 2
```

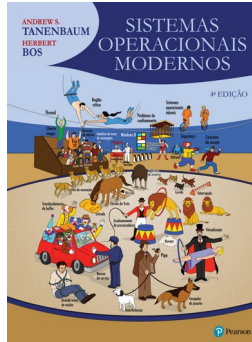
Objetivo: conhecer threads e suas implementações, e questão de fixação.

QUESTÃO 45 – Considere o programa abaixo escrito em linguagem C. No instante da execução da linha 5, ter-se-á uma hierarquia composta de quantos processos e threads, respectivamente?

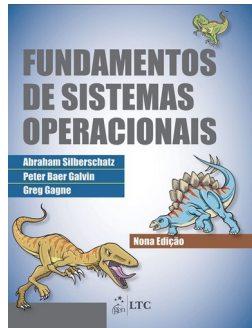
```
1    main(){  
2    int i;  
3    for(i=0;i<3;i++)  
4        fork();  
5    while(1);  
6    }
```

- A) 1 e 0.
- B) 3 e 0.
- C) 4 e 1.
- D) 7 e 7.
- E) 8 e 8.

Referências



TANENBAUM, Andrew S.; BOS, Herbert. Sistemas operacionais modernos. 4. edição. São Paulo: Pearson, 2016. xviii, 758 p. ISBN 9788543005676.



SILBERSCHATZ, Abraham.; GALVIN, Peter Baer.; GAGNE, Greg. Fundamentos de sistemas operacionais. 9. edição. Rio de Janeiro: LTC, 2015.

O modelo desta apresentação foi criado pelo Slidesgo.

Agradeço ao Prof. Bruno Kimura da Universidade Federal de São Paulo pelo material disponibilizado.