

Fundamentos de Sistemas de Operação

Unix Windows NT Network MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

Programação Concorrente
A API Pthreads

Modelo de programação Pthreads

- Uma norma POSIX (IEEE 1003.1c) para uma API para gestão de processos leves – criação, controle (prioridade, suspensão, terminar). Define também primitivas de sincronização.
- A API especifica o comportamento da biblioteca de processos leves; a realização pode ser feita de diferentes formas. Funcionalidades subdivididas em duas categorias: obrigatórias e opcionais; as obrigatórias têm de existir em todas as realizações.
- Comum no UNIX. No Windows, disponível no subsistema POSIX, embora o Win32 tenha a sua própria (diferente) API de *threads*.

API Pthreads (1)

- Pthreads: API standard com mais de 60 funções para manipular threads (só vamos abordar algumas...); “binding” com código em C.
 - Criar:
 - pthread_create
 - Determinar o thread ID:
 - pthread_self
 - Terminar threads:
 - pthread_cancel
 - pthread_exit
 - Funções “importadas” dos processos...
 - exit (termina o processo e todas as suas threads)

API Pthreads (2)

- API pthreads (continuação):
 - Sincronização dos fios de execução
 - pthread_join
 - Sincronização: mutexes
 - pthread_mutex_init, pthread_mutex_destroy
 - pthread_mutex_lock, pthread_mutex_unlock
 - Sincronização: semáforos
 - sem_init, sem_destroy
 - sem_wait, sem_post

- Sincronização: condições
 - pthread_cond_init, pthread_cond_destroy
 - pthread_cond_wait, pthread_cond_signal

Não abordados

Programação com Pthreads (1)

■ pthreads “Hello, world”:

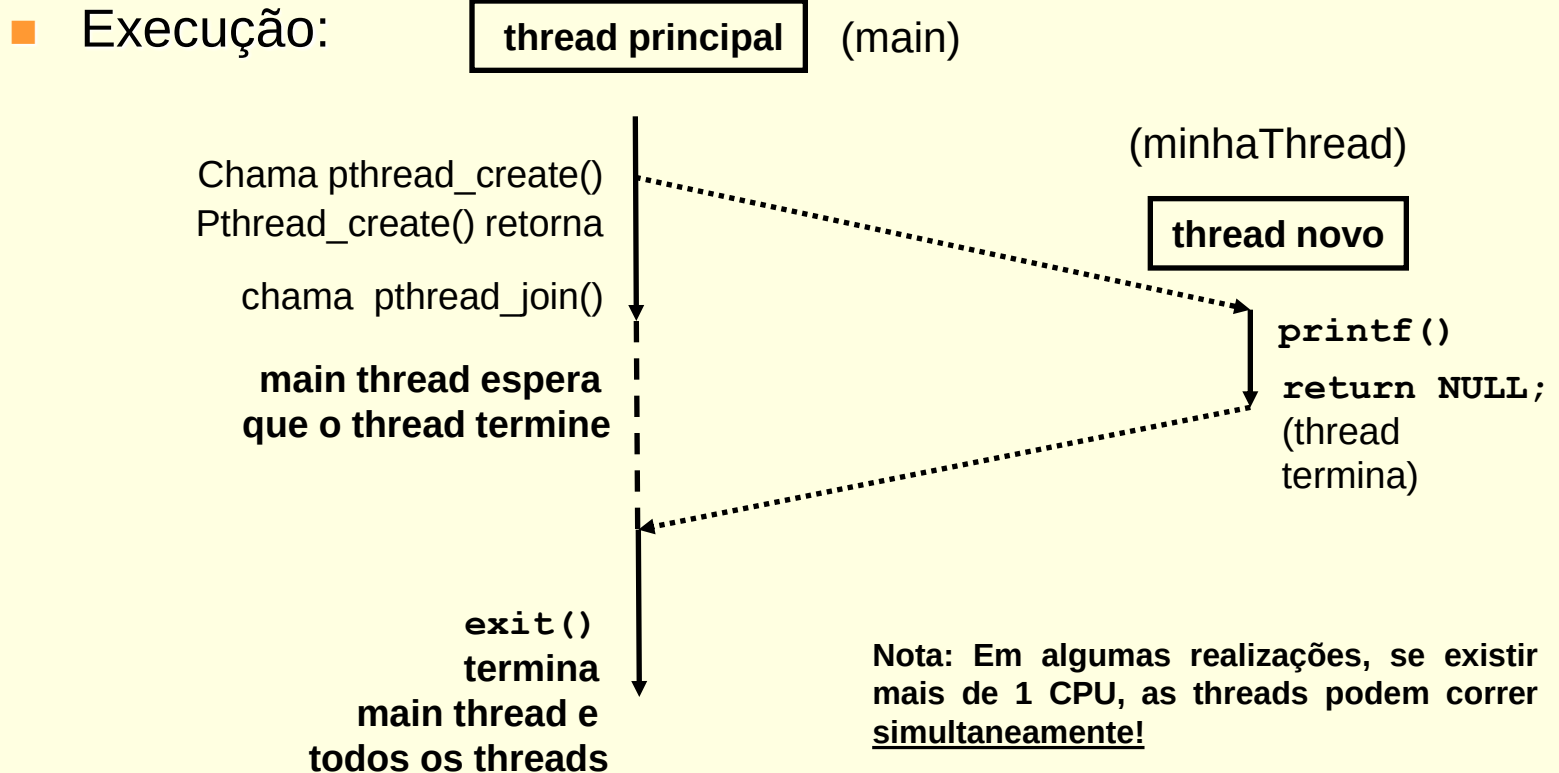
```
/*  
 * hello.c - Pthreads "hello, world" program  
 */  
#include "pthread.h"  
  
void *minhaThread(void *vargp);  
  
int main() {  
    pthread_t tid;  
  
    pthread_create(&tid, NULL, minhaThread, NULL);  
    pthread_join(tid, NULL);  
    exit(0);  
}  
  
/* thread routine */  
void *minhaThread(void *vargp) {  
    printf("Hello, world!\n");  
    return NULL;  
}
```

Thread attributes
(normalmente NULL)

argumento
(void *p)

Valor de retorno
(void **p)

Programação com Pthreads (2)

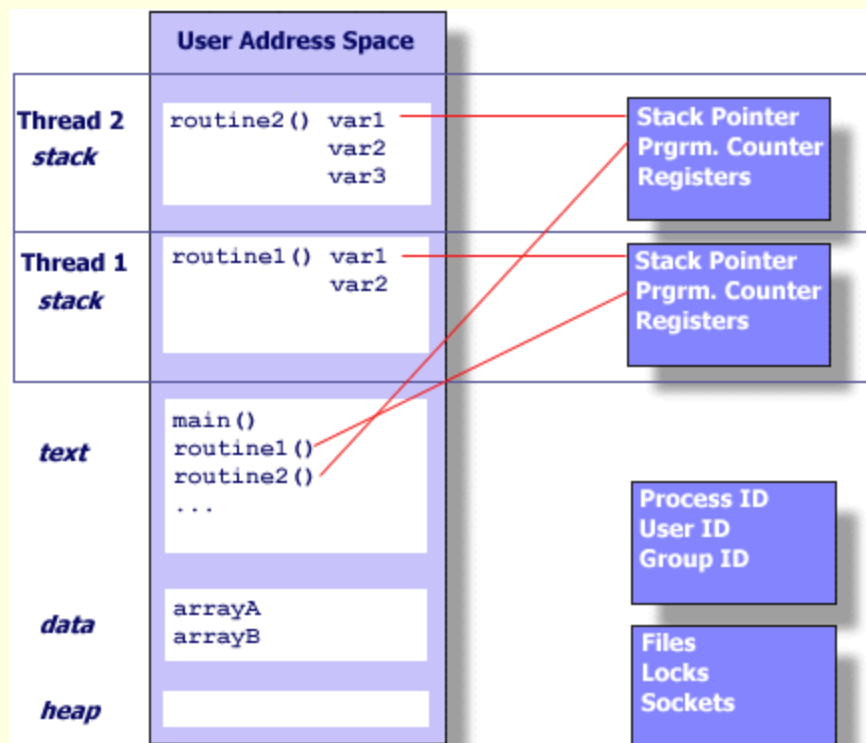


Programação com Pthreads (3)

■ Fluxos concorrentes

- Processos
 - O núcleo suporta múltiplos fluxos de controlo em espaços de endereçamento separados.
 - Controlo de processos e sinais do Unix (exº: fork, wait, kill).
 - Comunicação pelos mecanismos IPC (exº, memória partilhada)
- Threads
 - O núcleo suporta múltiplos fluxos de controlo (threads) que se executam “no interior” de um processo.
 - API dos POSIX threads.
 - Comunicação por partilha de variáveis globais.

Modelo de memória dos Threads (1)



Modelo conceptual:

- Cada *thread* corre no contexto de um processo
- Cada *thread* tem o seu próprio *thread context*
 - *Thread ID*, *stack*, *stack pointer*, *program counter*, e registos (de uso geral e de flags)
- Todo o resto do contexto do processo é partilhado por todos os *threads*
 - *Code*, *data*, *heap*, e bibliotecas partilhadas no espaço de endereços virtual do processo.
 - Ficheiros abertos e *handlers* de sinais

Fonte:

<https://www.llnl.gov/computing/tutorials/pthreads/>

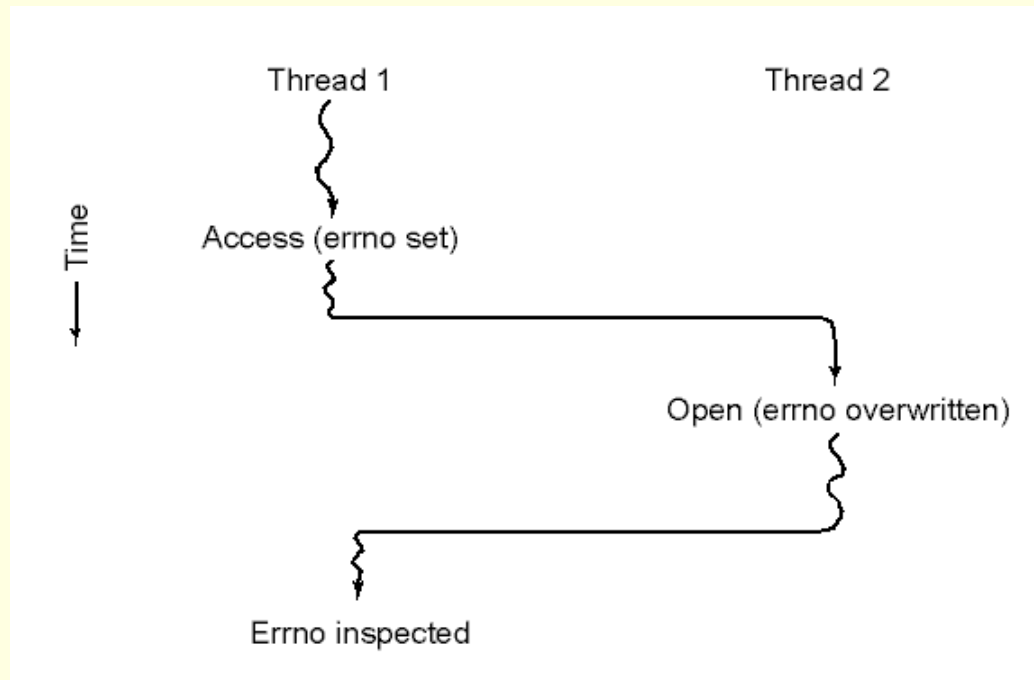
Modelo de memória dos Threads (2)

■ Modelo operacional:

- Operacionalmente, este modelo não é completamente suportado pelo SO/compilador/biblioteca de run-time:
 - Os valores dos registos são completamente separados e protegidos, mas
 - Qualquer *thread* pode ler e escrever o *stack* de outro *thread* (*por erro de programação, ignorância do espaço necessário para o stack da thread*)
 - Algumas implementações colocam o stack da thread “no meio” de 2 páginas inválidas, tentando assim “apanhar” erros de overflow e underflow do stack...
 - ... Mas um apontador “muito incorrecto” ☺ pode saltar por cima delas.

Programação com Pthreads: cuidados ⁽¹⁾

- Partilha de variáveis
 - Não óbvia!!! (a variável errno é global!)



Programação com *Pthreads*: cuidados ⁽²⁾

■ Partilha de variáveis

- Pergunta: Que variáveis, num programa *multithreaded* em C, são partilhadas?
 - A resposta não é tão simples como “variáveis globais são partilhadas” e “variáveis na pilha são privadas”.
- Requer respostas às seguintes perguntas:
 - Qual é o modelo de memória dos *threads*?
 - Como é que as variáveis “declaradas no programa” são instanciadas em memória?
 - Quantos *threads* referenciam cada uma das instâncias?

API Pthreads: controle de fluxo ⁽¹⁾

■ Criação

- `int pthread_create(pthread_t *id, pthread_attr_t atributos, void *(*função)(void *), void *argumentos);`

*Cria uma thread que executa a **função()** devolvendo o identificador da thread em **id**. A função só pode ser um argumento, mas este pode ser um “pacote” de argumentos referenciado pelo apontador - caso não se queira usar, passar **NULL**. Os **atributos** permitem especificar, por exemplo, a prioridade da nova thread, a dimensão do seu stack, etc. Para usar os atributos por omissão, usar **NULL**.*

API Pthreads: controle de fluxo (2)

■ Terminar a execução

- `void pthread_exit(void *valor);`

*Termina a thread invocadora, podendo uma outra que tenha efectuado um join recolher o valor de retorno. Para não retornar nada, usar **NULL**.*

*Quando a última thread do processo termina, o processo termina da mesma forma como terminaria se tivesse executado um **exit(0)**.*

*Quando o código da função que está a ser executada pela thread termina por invocação de um **return**, o efeito é o mesmo de ter executado um **pthread_exit()**.*

API Pthreads: sincronização ⁽¹⁾

■ *Esperar pelo fim duma thread*

- `int pthread_join(pthread_t id, void **valor);`

*Suspende a thread invocadora à espera que a thread identificada por **id** termine. Para não receber nada desta, usar **NULL**. Se a thread alvo já tinha terminado, a thread invocadora recolhe o **valor** de retorno e continua imediatamente.*

Nota: não se pode fazer join com uma thread “detached” (que é um atributo que não abordaremos aqui).

API Pthreads: sincronização (2)

■ *Mutexes: declaração e inicialização em tempo de compilação*

- `pthread_mutex_t mtx= PTHREAD_MUTEX_INITIALIZER;`

■ *Mutexes: declaração e inicialização em tempo de execução*

- `pthread_mutex_t mtx;`
- `int pthread_mutex_init(pthread_mutex_t *mtx,
pthread_mutex_attr *atributos);`

Prepara o mutex `mtx` para ser usado; para usar os atributos por omissão, passar `NULL`.

API Pthreads: sincronização (3)

■ Mutexes: remoção

- `int pthread_mutex_destroy(pthread_mutex_t *mtx);`

Liberta os recursos associados ao mutex `mtx`; falha se este não tiver sido inicializado ou estiver trancado.

API Pthreads: sincronização (4)

■ *Mutexes: aplicar o trinco*

- `int pthread_mutex_lock(pthread_mutex_t *mtx);`

Se o mutex `mtx` está trancado, a thread invocadora bloqueia; se não, obtém o trinco e continua.

■ *Mutexes: remover o trinco*

- `int pthread_mutex_unlock ( pthread_mutex_t *mtx);`

Se o mutex `mtx` foi trancado pela thread invocadora, o trinco é removido e a execução continua; se não estava trancado, ou se estava trancado por outra thread, o resultado é indefinido (muitas realizações - e.g. Linux - dão erro).

API Pthreads: miscelâneas

■ Qual é o meu ID?

- `pthread_t pthread_self(void) ;`

Devolve o ID da thread invocadora.

■ Comparação (igualdade)

- `int pthread_equal(pthread_t t1, pthread_t t2) ;`

Devolve 0 se os IDs das threads são diferentes; caso contrário devolve um valor diferente de 0.

Nota: se t1 e/ou t2 não identificam threads, o resultado é indefinido.