

Threads POSIX

API e exemplo

Mutexes

API

Este ficheiro destina-se a ser projectado e, portanto, tem um formato e tamanho de letra diferente do habitual.

Tópicos

1. Compilação de programas com threads POSIX
2. Ficheiros header relacionados
3. Estruturas de dados usadas com threads e mutexes
4. Resumo da funcionalidade de threads e mutexes
5. API detalhado – threads e mutexes
6. Questões relativas ao controlo das threads
7. Exemplo - Threads

1. Compilação de programas com *threads* posix:

`gcc etcetc.c -pthread`

2. Ficheiros header directamente envolvidos

`pthread.h`

3. Estruturas de dados mais usadas

- ID de thread

`pthread_t`

- Atributos de thread

`pthread_attr_t`

- Mutex para sincronização entre threads

`pthread_mutex_t`

4. API (Resumo)

>> Threads

- Criação de uma thread

`pthread_create`

- Obter o ID da (própria) thread

`pthread_self`

- Terminar a (própria) thread

`pthread_exit`

- Terminação de (outra) thread

`pthread_cancel`

- Modificar o estado cancel-state (da própria thread)

`int pthread_setcancelstate`

- Modificar o tipo de cancel-state (da própria thread)

`int pthread_setcanceltype`

- Esperar que uma thread termine

`pthread_join`

- Enviar um sinal a uma *thread* do mesmo processo

`pthread_kill`

>> Mutexes

Operações fundamentais

- Criação / Inicialização

`int pthread_mutex_init (pthread_mutex_t *,`

```
const pthread_mutexattr_t *restrict attr)
```

- **Eliminação**

```
int pthread_mutex_destroy (pthread_mutex_t *)
```

- **Esperar, bloqueando**

```
int pthread_mutex_lock(pthread_mutex_t *)
```

- **Tentar esperar, sem bloquear (usar só em caso bem justificado)**

```
int pthread_mutex_trylock(pthread_mutex_t *)
```

- **Libertar**

```
int pthread_mutex_unlock(pthread_mutex_t *)
```

5. API - Detalhado

>> Threads

Criação de uma thread

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

- **thread**

Ponteiro para o ID da thread

- **attr**

Atributos iniciais da thread

- **start_routine**

Ponteiro para a função da thread

- **void *arg**

Argumento a passar para a função da thread

A *thread* criada ficará imediatamente a executar.

Pode-se passar qualquer valor para o seu argumento desde que ocupe o mesmo número de bits que o ponteiro para *void*. Se for necessário passar mais informação, então passar-se-á um ponteiro para a estrutura com a informação

Obter o ID da (própria) thread

```
pthread_t pthread_self(void)
```

O ID da *thread* tem a mesma utilidade que o PID de um processo.

Terminar a thread (a própria *thread*)

```
void pthread_exit(void *retval);
```

- **retval**

Valor a usar como código de terminação da thread

Este valor pode ser obtido com *pthread_join*

Terminar (outra) *thread*

```
int pthread_cancel(pthread_t thread)
```

- **thread**

ID da thread a terminar

A thread alvo terminará ou não consoante o seu estado de cancel-state

-> Usar apenas em último caso e desde que bem justificado

Deve evitar-se a terminação abrupta de *threads* com esta função. É melhor usar a estratégia geral da variável de condição: uma variável que é consultada periodicamente pela *thread* em questão que terminará naturalmente quando detecta que o seu valor mudou (modificada por uma outra *thread* qualquer do programa)

Modificar o estado *cancel-state* (da própria thread)

```
int pthread_setcancelstate(int state, int *oldstate)
```

- **state**

Estado desejado

- **oldstate**

Ponteiro para armazenar o estado anterior

PTHREAD_CANCEL_ENABLE

PTHREAD_CANCEL_DISABLE

Modificar o tipo de *cancel-state* (da própria thread)

```
int pthread_setcanceltype(int type, int *oldtype)
```

- **state**
Estado desejado
- **oldstate**
Ponteiro para armazenar o estado anterior

PTHREAD_CANCEL_DEFERRED (*default*)

PTHREAD_CANCEL_ASYNCHRONOUS

Esperar que uma *thread* termine

```
int pthread_join(pthread_t thread, void **retval)
```

- **thread**
ID da thread a esperar
- **retval**
Ponteiro para o valor de retorno usado pela thread

Enviar um sinal a uma *thread* do mesmo processo

```
int pthread_kill(pthread_t thread, int sig)
```

- **thread**
Thread alvo (dentro do mesmo processo)
- **sig**

Sinal a enviar

Efeito

- Envia o sinal *sig* à thread com o ID indicado no primeiro parâmetro.
- O tratamento de sinais é geral ao processo. No entanto, se o sinal estiver a ser tratado por uma função, essa função será executada no contexto da thread indicada.

Usos típicos

- Lidar com situações em que se deseja que uma thread processe imediatamente uma determinada situação mesmo que esteja bloqueada em algo (ex., um read), sem, no entanto, forçar a thread a interromper imediatamente

>> Mutex

Inicializar uma variável *mutex*

```
int pthread_mutex_init (pthread_mutex_t * pmutex,  
                        const pthread_mutexattr_t * attr)
```

- **pmutex**
Ponteiro para a variável *mutex*
- **attr**
Atributos de inicialização do *mutex*.
NULL faz com que sejam usados os atributos *default*

É necessário inicializar o *mutex* antes de o utilizar

Alternativamente o *mutex* pode ser inicializado com a atribuição:

```
var_mutex = PTHREAD_MUTEX_INITIALIZER
```


O *mutex* corresponde a um semáforo binário de utilização simplificada no contexto de *threads*

“Des-inicializar” uma variável *mutex*

```
int pthread_mutex_destroy (pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Não corresponde a uma “destruição”: a variável *mutex* continuará a existir, mas já não poderá ser usada (a não ser que seja novamente inicializada).

Esta função é chamada quando já não se deseja voltar a usar o *mutex*. Só deve ser chamada quando o *mutex* não está ocupado.

Aguardar e obter a posse do *mutex*

```
int pthread_mutex_lock(pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Esta função bloqueia até que o *mutex* esteja livre.

Ao avançar, o *mutex* fica ocupado (pela *thread* que invocou a função)

Tentar obter a posse do mutex sem bloquear

```
int pthread_mutex_trylock(pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Esta função tenta obter a posse do *mutex*. Se o *mutex* estivesse livre, fica ocupado por esta *thread*. Se o *mutex* já estivesse ocupado, a função retorna logo com o código EBUSY

Esta função remete para uma lógica não bloqueante e torna-se menos interessante porque complica as tarefas do programador. Poderá, no entanto, ser útil quando uma *thread*, por alguma razão, necessite de garantir que não fica bloqueada.

Libertar a posse do mutex

```
int pthread_mutex_unlock(pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Esta função liberta o *mutex*. Uma das *threads* que estiver à espera do *mutex* poderá avançar (obtendo essa *thread* a posse dele).

6. Algumas questões relativas ao controlo das threads

- Muitas vezes é necessário indicar a uma thread que deve terminar. A forma correcta de o fazer é de indicar através de uma variável de controlo que a thread deverá terminar assim que possível (evitar `pthread_cancel`).
- No entanto, a thread pode estar ocupada em algo e não consultar rapidamente (ou de todo) essa variável. Exemplos:
 - Pode estar bloqueada numa leitura de teclado
 - Pode estar bloqueada numa leitura de named pipe
 - Pode estar bloqueada em algo

Como resolver a questão sem usar `pthread_cancel`?

-> A solução é dependente do cenário presente em cada caso e remete o programador (terá que “ser engenheiro/a”)

Exemplos:

- Se a *thread* estiver bloqueada numa leitura de pipe, então a parte do programa que está a indicar à thread que deve terminar pode, por exemplo enviar uma informação *mock-up* para esse pipe de forma a que a *thread* saia do *read* e consulte a variável de controlo. A informação enviada para o *pipe* pode precisamente ter o significado “thread deve terminar”.
- **Este é apenas um exemplo. Haverá muitas outras estratégias**

Outra ideia, mais genérica e mais facilmente reutilizável em diversos cenários:

- Usar a função `pthread_kill`
Esta função permite a um programa enviar um sinal a uma thread **do mesmo programa**. O sinal é atendido (em código, se estiver a ser tratado) no

contexto da *thread* alvo. O tratamento do sinal é feito nos moldes habituais, com as propriedades “desbloqueantes” já vistas anteriormente.

7. Exemplo:

Este exemplo ilustra:

- Criação de *threads*
- Ordenar a terminação de *threads* sem usar *pthread_cancel*
- “junção” de *threads* = “uma *thread* aguardar que outra termine”

```
#include <stdio.h>
#include <string.h>
#include <pthread.h> // Notar este include
#include <stdlib.h>
#include <unistd.h>

/* dados de controlo para cada thread */
/* definidos pelo programador - conforme o que for preciso */

typedef struct {
    int continua;
    char character;
    int vezes;
    pthread_t tid; /* ID da thread */
    void * retval; /* Código de terminação */
} TDados;

/* função de suporte à(s) thread(s) */

void * imprime(void * arg) {
    int i;
    TDados * dados = (TDados *) arg;

    for (i=0; i<dados->vezes; i++) {
        printf(" %c ", dados->character);
        fflush(stdout); /* para o character aparecer logo */
        sleep(1);
        if (dados->continua == 0)
            break;
    }
    printf(" thread %c terminou ", dados->character);
```

```

    return NULL;
}

#define NUMTHR 3

int main() {
    int res, i;
    char temp[30];
    TDados workers[NUMTHR];

    for (i=0; i<NUMTHR; i++) {
        workers[i].continua = 1;
        workers[i].vezes = (i+1)*10;
        workers[i].caracter = 'A'+i;
        res = pthread_create(
            & workers[i].tid, // & variavel p/ ID da thread
            NULL,             // atributos default
            imprime,          // funcao da thread
            (void *) &workers[i]); // argumento -> ptr para
                                // dados da thread ( melhor: workers + i )
        if (res != 0) {
            perror("\nErro na criação da thread");
            exit(1);
        }
    }

    printf("\nmain: estou a trabalhar. "
        "Escreve coisas, sair para terminar\n");
    while (1) {
        fgets(temp, 30, stdin);
        if (temp[strlen(temp)-1] == '\n')
            temp[strlen(temp)-1] = '\0';
        printf("disseste: [%s]\n", temp);
        if (strcmp(temp,"sair")==0)
            break;
    }

    /* indicar às threads para terminarem */
    for (i=0; i<NUMTHR; i++)
        workers[i].continua = 0; // nunca terminar thread à força

```



```

/* esperar que as threads terminem mesmo */
/* sleep(1) -> Nunca usar sleeps para esperar fim de threads*/

/* forma correcta = pthread_join */
for (i=0; i<NUMTHR; i++)
    pthread_join(workers[i].tid, & workers[i].retval);

printf("\ntodas as threads terminaram\n");
printf("\nmain a encerrar\n");
return 0;
}

```

Acerca do exemplo

- Não inclui sincronização (mutexes, semáforos, etc.)
 - É bastante simples – apenas o suficiente para mostrar criação e junção de *threads*
 - As tarefas das duas *threads* são tão parecidas que poderiam ser mais logicamente feitas pela mesma função em vez de duas funções separadas como está agora
- TPC: melhorar este exemplo para usar apenas uma única função para ambas as *threads*

Outros exemplo com threads

(exclusivamente em aulas ou também em outros documentos)

- Leitura simultânea de dados de várias fontes com **threads**
- Exemplo de algoritmo paralelizado e demonstração de ganho de performance (determinação PI - método Montecarlo)
- Identificação de valores com e sem Mutex – demonstração da necessidade de sincronização
- Exemplo de variáveis condicionais, que usa **threads** e **mutexes**

Nota. Este documento destina-se a ser projetado como se fossem slides. Por essa razão o formato é diferente do habitual, incluindo o tamanho da letra e organização do texto.