Universidade Federal de Itajubá – UNIFEI
Instituto de Engenharia de Sistemas e Tecnologias da Informação – IESTI
CCO 004 – Sistemas Operacionais – Prof. Edmilson Marmo Moreira

Capítulo 5 – *Threads*

> *"Os estudos dão motivação e gosto à alegria, amansam e consolam a tristeza, refreiam os ímpetos loucos da mocidade, aliviam o pesar da velhice."*
>
> **Jean Luis Vives**

1 Introdução

Uma *Thread,* às vezes chamada de **processo leve** (*lightweight process*), é uma unidade básica de utilização de CPU; compreende um ID de *thread*, um contador de programa, um conjunto de registradores e uma pilha.

Um processo tradicional, ou **pesado** (*heavyweight*), tem um único fluxo de controle. Entretanto, existem situações em que é interessante ter múltiplos fluxos de controle (*threads*) que compartilhem um único espaço de endereçamento, mas executem em concorrência.

Uma *Thread* compartilha com outras *threads,* pertencentes ao mesmo processo, sua seção de código, seção de dados e outros recursos do sistema operacional, tais como arquivos abertos e sinais.

Assim, as *threads* compartilham o mesmo contexto de software e espaço de endereçamento com as demais *threads,* porém cada *thread* possui seu contexto de hardware individual.

As *Threads* são implementadas internamente através de uma estrutura de dados denominada *bloco de controle de threads (Thread Control Block – TCB).*

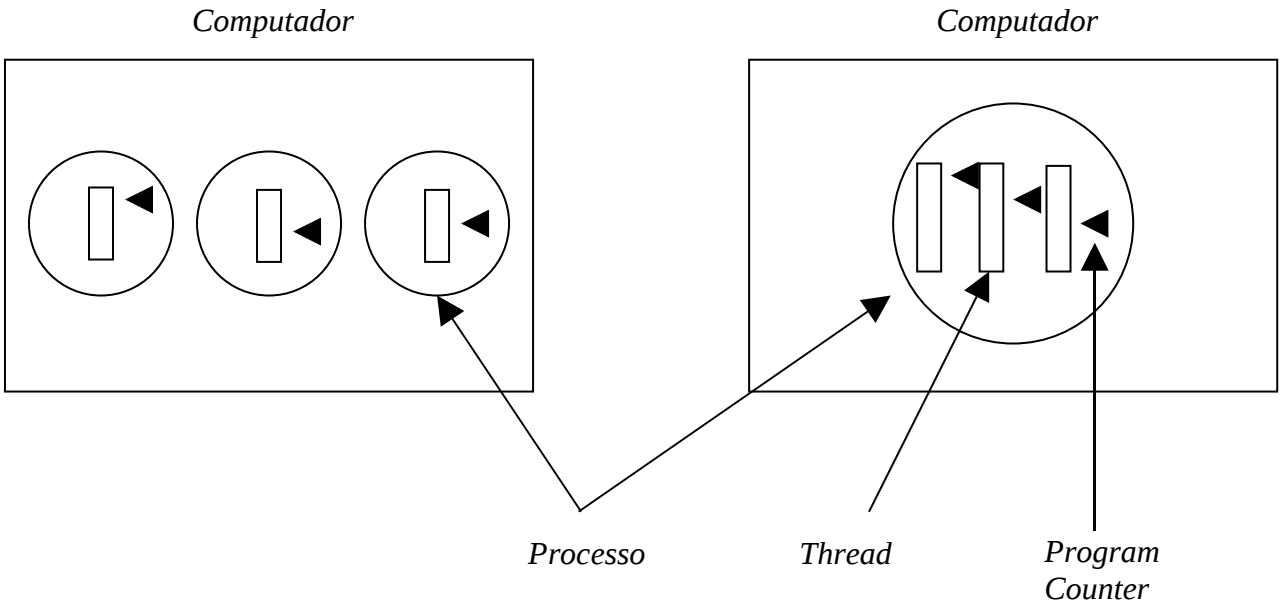
O TCB armazena, além do contexto de hardware, mais algumas informações relacionadas exclusivamente a *thread,* como prioridade, estado de execução e bits de estado.

Exemplo:

Considerando um sistema de arquivos que tem que se bloquear ocasionalmente aguardando pelo disco.

Se o servidor possuir muitas *threads* de controle, uma segunda *thread* pode executar enquanto a primeira fica bloqueada.

O resultado é um melhor aproveitamento dos recursos e melhoria do desempenho do sistema. Este resultado não é possível de alcançar com a utilização de dois servidores de arquivo independentes porque eles devem compartilhar *buffers,* que precisam estar no mesmo espaço de endereçamento.



Nesta figura, é possível identificar duas máquinas. A primeira apresenta 3 processos. Cada processo possui o seu próprio *program counter,* sua pilha e seu conjunto de registradores.

A outra máquina possui apenas um processo em execução. Este processo contém 3 *threads* de controle.

As *threads* são como mini-processos. Cada *thread* executa de forma estritamente seqüencial e possui o seu próprio *program counter* e sua própria pilha, para localizar a sua posição.

Threads compartilham a CPU, da mesma forma que um processo faz (*timesharing*). Somente em máquinas com vários processadores elas executam em paralelo.

Threads podem criar novas *threads* e ser bloqueadas.

O Sistema Operacional é responsável em escalonar o uso do processador entre as diversas *threads,* dando a cada uma, alternadamente, uma fatia de tempo.

Cada processo possui o seu próprio espaço de endereçamento. Isso impede que uma *thread* de um processo possa escrever na área de endereçamento de outro processo.

Threads compartilham o processador da mesma maneira que processos e passam pelas mesmas mudanças de estado (execução, espera e pronto).

Vantagens da Utilização de *Thread:*

1. **Capacidade de Resposta:** A *multithreading* de uma aplicação interativa pode permitir que um programa continue executando mesmo se parte dele estiver bloqueado ou executando uma operação demorada, aumentando a capacidade de resposta para o usuário.

2. **Compartilhamento de Recursos:** As *Threads* compartilham a memória e os recursos do processo aos quais pertencem.

3. **Economia:** Alocar memória e recursos para a criação de processos é caro. Como as *threads* compartilham recursos do processo aos quais pertencem, é mais econômico criar e realizar a troca de contexto das *threads.*

4. **Utilização de Arquiteturas Multiprocessador:** Os benefícios da *multithreading* podem ser ampliados em uma arquitetura multiprocessador, na qual cada *thread* pode estar executando em paralelo em um processador diferente.

Latência de processos e *Threads*

Implementação	**Tempo de Criação** (μ s)	**Tempo de Sincronização** (μ s)
Processo	1700	200
Thread	52	66

2 Arquitetura e Implementação

O conjunto de rotinas disponíveis para que uma aplicação utilize as facilidades das *threads* é chamado pacote de *threads*.

Existem diferentes abordagens na implementação deste pacote em um sistema operacional, o que influenciará no desempenho, na concorrência e na modularidade das aplicações *multithread.*

Existem suporte a *threads* em nível de usuário, *threads* de usuário, pelo *kernel, threads* de *kernel,* por uma combinação de ambos, modo híbrido, ou por um modelo conhecido como *scheduler activations.*

Ambientes	Arquitetura
Distributed Computing Environment (DCE)	Modo usuário
Compaq Open VMS versão 6	Modo usuário
Microsoft Windows 2000	Modo kernel
Compaq Unix	Modo kernel
Compaq Open VMS versão 7	Modo kernel
Sun Solares versão 2	Modo híbrido
University of Washington FastThreads	Scheduler activations

3 *Threads* em Modo Usuário

As *threads* de usuário são suportadas acima do *kernel* e são implementadas por uma biblioteca de *threads* em nível do usuário.

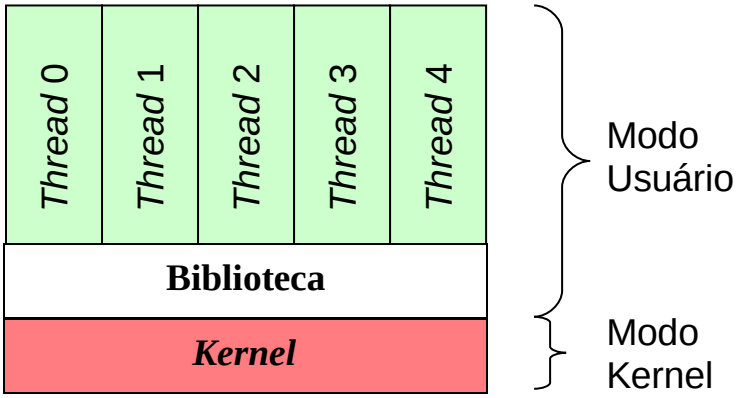
A biblioteca fornece suporte à criação, escalonamento e gerência de *threads,* sem suporte do *kernel.*

Como o *kernel* não está a par das *threads* de usuário, todas as atividades de criação e escalonamento de *threads* são feitas no espaço de usuário, sem necessidade da intervenção do *kernel.*

A vantagem deste modelo é a possibilidade de implementar aplicações *multithreads* mesmo em sistemas operacionais que não suportam *threads*.

Além disso, as *threads* de usuário são rápidas de criar e gerenciar; no entanto, também apresentam desvantagens. Por exemplo, se o *kernel* tiver uma única *thread* de usuário realizando uma chamada bloqueante causará o bloqueio de todo o processo, mesmo se houver outras *threads* disponíveis para execução na aplicação.

Outra desvantagem é a impossibilidade de executar as *threads* em paralelo, se houver múltiplos processadores, isso ocorre porque o sistema operacional identifica apenas um processo e escalona esse processo para executar em um processador.

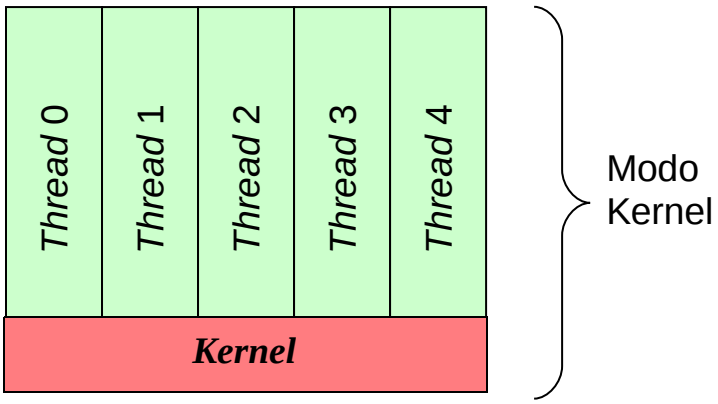


3 *Threads* em Modo *Kernel*

No modo *kernel* as *threads* são suportadas diretamente pelo sistema operacional: a criação, o escalonamento e a gerência de *threads* são feitos pelo *kernel.*

O sistema operacional sabe da existência de cada *thread* e pode escaloná-las individualmente. Desta forma, as *threads* de kernel são geralmente mais lentas para criar e gerenciar do que as *threads* de usuário.

No entanto, se uma *thread* realizar uma chamada bloqueante ao sistema, o *kernel* poderá escalonar outra *thread* na aplicação para execução.



4 *Threads* em Modo Híbrido

A arquitetura de *threads* em modo híbrido combina as vantagens de *threads* implementadas em modo usuário e *threads* em modo kernel.

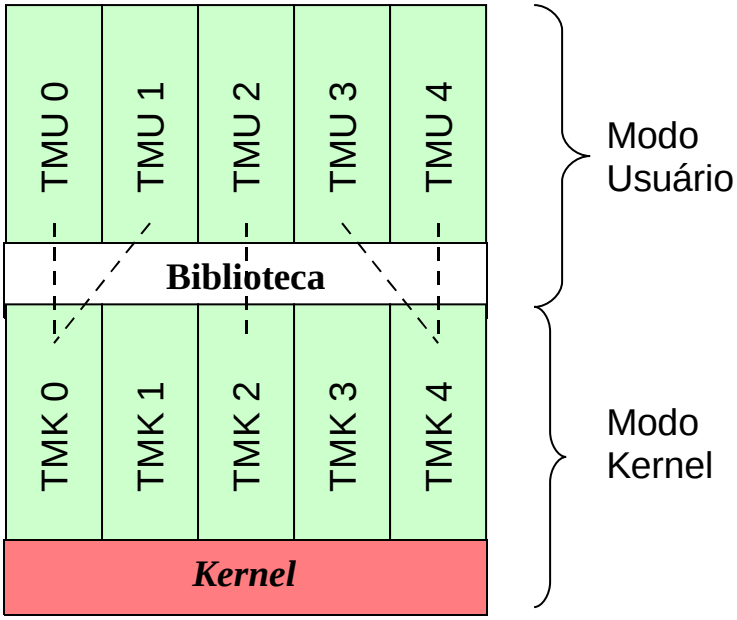
Um processo pode ter várias *threads* em modo *kernel* e, cada *thread* em modo *kernel*, pode possuir várias *threads* em modo usuário. O núcleo do sistema reconhece as *threads* em modo *kernel* e pode escaloná-las individualmente.

Uma *thread* em modo usuário pode ser executada em uma *thread* em modo *kernel*, em um determinado momento, e no instante seguinte ser executada em outra *thread.*

O programador desenvolve a aplicação em termos de *threads* em modo usuário e especifica quantas *threads* em modo *kernel* estão associadas ao processo.

As *threads* em modo usuário são mapeadas para a *threads* em modo *kernel* enquanto o processo está em execução. O programador pode utilizar apenas *threads* em modo *kernel*, *threads* em modo usuário ou uma combinação de ambos.

O modo híbrido, apesar da maior flexibilidade, apresenta problemas herdados de ambas as implementações. Por exemplo, quando uma *thread* em modo *kernel* realiza uma chamada bloqueante, todos as *threads* em modo usuário são colocadas no estado de espera. *Threads* em modo usuário que desejam utilizar vários processadores devem utilizar diferentes *threads* em modo kernel, o que influenciará no desempenho.



5 *Scheduler Activations*

Os problemas apresentados no pacote de *threads* em modo híbrido existem devido à falta de comunicação entre as *threads* em modo usuário e em modo kernel.

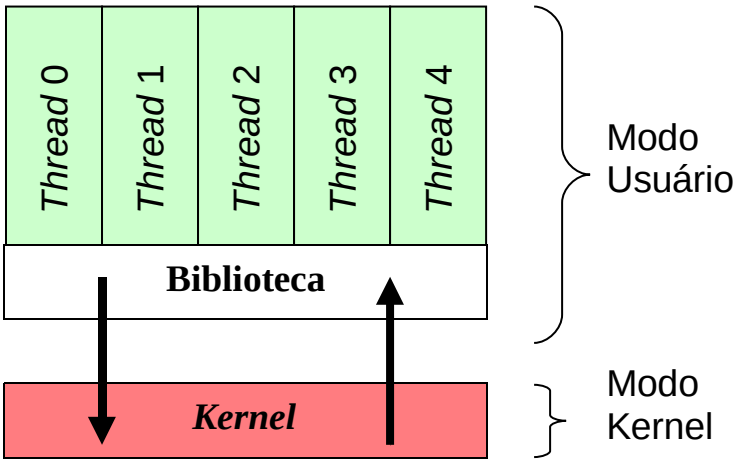
O modelo ideal deveria utilizar as facilidades do pacote em modo kernel com o desempenho e flexibilidade do modo usuário.

Introduzido no início da década de 1990 na Universidade de Washington, este pacote combina o melhor das duas arquiteturas, mas em vez de dividir as *threads* em modo usuário entre os de modo *kernel,* o núcleo do sistema troca informações com a biblioteca de *threads* utilizando uma estrutura de dados chamada *scheduler activations.*

A maneira de alcançar um melhor desempenho é evitar as mudanças de modos de acesso desnecessárias (usuário-kernel-usuário). Caso uma *thread* utilize uma chamada ao sistema que a coloque no estado de espera, não é necessário que o *kernel* seja ativado. A própria biblioteca, em modo usuário, escalona outra *thread.*

Isto é possível porque a biblioteca em modo usuário e o *kernel* se comunicam e trabalham de forma cooperativa.

Cada camada implementa seu escalonamento de forma independente, porém trocando informações quando necessário.



6 Estudo de Caso 1: *Pthreads*

Pthreads refere-se ao padrão POSIX (IEEE 1003.1c) que define uma API para a criação e sincronismo de *threads.*

Essa é uma especificação para o comportamento da *thread,* e não uma implementação.

Os projetistas de sistemas operacionais podem implementar a especificação como desejarem. Diversos sistemas implementam a especificação *Pthreads,* incluindo Solaris, Linux, Tru64 UNIX e Mac OS X.

Também existem implementações *shareware* em domínio público para os diversos sistemas operacionais da família Windows.

A API *Pthreads* é dividida em 3 grandes categorias:

- **Gerenciamento de *threads:*** nesta classe há rotinas e estruturas para criar, configurar, escalonar, etc., as *threads.*
- **Mutexes:** são funções e estruturas utilizadas para impor exclusão mútua entre as *threads.*
- **Variáveis condicionais:** utilizadas na comunicação entre *threads* que compartilham mutexes.

O exemplo a seguir cria 5 *threads* com a função `pthread_create()`. Cada *thread* imprime a mensagem “Hello World!” e termina com uma chamada a `pthread_exit()`.

Exemplo:

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}
```

```
int main ()
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d\n", t);
      rc=pthread_create(&threads[t], NULL,
                        PrintHello, (void *)t);
      if (rc){
       printf("ERROR; return code from pthread_create()
               is %d\n", rc);
       return -1;
      }
   }
   pthread_exit(NULL);
}
```

A função `pthread_create()` possui os seguintes argumentos:

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

- `thread` : é um identificador único para a nova *thread.*
- `attr` : um objeto que pode ser usado para setar atributos da *thread.*
- `start_routine` : função que deverá ser executada pela *thread.*
- `arg` : um único argumento que pode ser passado para a função *start_routine.* Deve ser passado por referência como um ponteiro do tipo `void`. `NULL` pode ser usado se nenhum argumento for passado.

A função `pthread_create()` retorna 0, se funcionar, ou um valor indicando o erro, caso contrário.

Uma *thread* termina nas seguintes situações:

- A *thread* retorna da função que a originou (`start_routine`);
- A *thread* chama `pthread_exit()`;
- A *thread* é cancelada por outra *thread* através da função `pthread_cancel()`;
- O processo inteiro termina.

É importante observar que se a *thread* mãe termina retornando da sua função principal, suas filhas morrem. Entretanto, se a *thread* mãe termina com `pthread_exit()`, suas filhas não morrem.

O próximo exemplo cria uma *thread* separada para o somar os *n* primeiros números inteiros não negativos. Como foi visto no exemplo anterior, em um programa `Pthread`, *threads* separadas iniciam a execução em uma função específica.

Neste programa esta função é `runner()`. Quando esse programa é iniciado, uma única *thread* de controle é iniciada em `main()`. A função *main* cria uma segunda *thread* que inicia o controle da função `runner()`.

As duas *threads* compartilham a soma de dados da variável global `sum`. Após a criação da segunda *thread,* a *thread* `main()` esperará até que a função *thread* `runner()` termine, através da chamada da função `pthread_join()`.

A *thread* `runner()` terminará quando chamar a função `pthread_exit()`. Quando a *thread* `runner()` tiver retornado, a *thread* `main()` informará o valor da soma dos dados compartilhados.

Exemplo:

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

int sum; /* variavel compartilhada */

void *runner(void *param)
{
   int i, upper = atoi((char *)param);
   sum = 0;
   if (upper > 0) {
     for (i=1; i<=upper; i++)
       sum += i;
   }
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t tid;
   pthread_attr_t attr;
   if (argc != 2) {
```

```
    fprintf(stderr, "uso: a.out <valor inteiro>\n");
    return 0;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr, "%d precisa ser >= 0\n", atoi(argv[1]));
    return 0;
  }
  pthread_attr_init(&attr);
  pthread_create(&tid, &attr, runner, argv[1]);
  pthread_join(tid, NULL);
  printf("soma = %d\n", sum);
  return 0;
}
```

A função `pthread_join()` permite realizar a sincronização entre *threads.* Uma *thread* também pode ser *"detached"*, ou seja, uma *thread* que não pode ser sincronizada com esta função.

Ser *"detached"* é um atributo da *thread* que pode ser definido no momento da sua criação.

Para criar uma *thread detached* é preciso declarar uma variável `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 de criação.

A função `pthread_attr_setdetachstate()` tem os seguintes argumentos:

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

O parâmetro `detachstate` pode ser:

- `PTHREAD_CREATE_DETACHED`: criada como *"detached"*
- `PTHREAD_CREATE_JOINABLE`: pode ser sincronizada com *join* (padrão).

Além desta opção, o tipo da *thread* pode ser alterado para que fique como *"detached"* através da função `pthread_detach ()`.

Variáveis Mutexes

Mutex é uma abreviação de "*mutual exclusion".* Variáveis mutexes são uma das formas de implementar sincronização e proteger áreas compartilhadas.

Uma variável do tipo mutex é um *"lock"* que protege dados compartilhados pelas *threads.*

O princípio básico é que apenas uma *thread* pode ter efetuado um *lock* em uma variável do tipo mutex em um dado instante.

Mesmo que diversas *threads* tentem efetuar o *lock,* apenas uma delas será bem sucedida e nenhuma outra *thread* poderá efetuar o *lock* antes que a primeira *thread* o libere.

Quando diversas *threads* competem por um mutex, aquelas que não conseguiram efetuar o *lock* ficam bloqueadas na chamada da função de *lock.*

A especificação `Pthread` possui as seguintes rotinas para criar e destruir mutexes:

- `ptread_mutex_init (mutex, attr)`
- `ptread_mutex_destroy (mutex)`
- `ptread_mutexattr_init (attr)`
- `ptread_mutexattr_destroy (attr)`

Existem 4 tipos de mutexes:

- **Fast:** a *thread* fica bloqueada para sempre.
- **Recursive:** a *thread* retorna da função de *lock* imediatamente como se tivesse conseguido efetuar o *lock.* O número de vezes que a função de *lock* foi chamada para o mutex fica armazenado na estrutura do mutex.
- **Error checking:** a *thread* retorna da função de *lock* imediatamente com o código de erro EDEADLK.
- **Timed** (padrão): A *thread* fica bloqueada por um determinado tempo. Para utilizar o mutex dessa forma, a função `pthread_mutex_timedlock()` deve ser chamada no lugar de `pthread_mutex_lock()`.

O tipo do mutex afeta a forma como ele é tratado quando uma *thread* que já efetuou o *lock* no mutex tenta fazê-lo novamente.

Variáveis mutexes devem ser declaradas com o tipo:

```
pthread_mutex_t
```

E devem ser inicializadas antes que possam ser utilizadas. Existem duas formas para se inicializar uma variável mutex:

1. Estaticamente, quando ela é declarada:

```
pthread_mutex_t meuMutex = PTHREAD_MUTEX_INITIALIZER;
```

2. Dinamicamente, através da função `pthread_mutex_init()`.

Na inicialização estática é possível atribuir os seguintes valores contantes:

- `PTHREAD_MUTEX_INITIALIZER`: timed
- `PTHREAD_RECURSIVE_MUTEX_INITIALIZER`: recursive
- `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER`: error check
- `PTHREAD_MUTEX_ADAPTIVE_NP`: fast

O mutex é inicializado desbloqueado. O objeto *attr* é usado para estabelecer as propriedades para o mutex e deve ser do tipo:

```
pthread_mutexattr_t
```

Travando e destravando Mutexes

A especificação `Pthread` três funções para o travamento (*lock*) e destravamento (*unlock*) do mutex:

- `ptread_mutex_lock (mutex)`
- `ptread_mutex_trylock (mutex)`
- `ptread_mutex_unlock (mutex)`

A função `ptread_mutex_lock()` é usada por uma *thread* para adquirir o travamento do mutex. Se o mutex já estiver travado por outra *thread,* a chamada a esta função irá bloquear a *thread* até que o mutex seja destravado.

A função `ptread_mutex_trylock()` tentará travar o mutex. Entretanto, se o mutex já estiver travado, a função retorna com um código de erro. Esta rotina pode ser utilizada na prevenção de *deadlocks.*

A função `ptread_mutex_unlock()` irá destravar o mutex. Esta função retorna erro se o mutex já estiver liberado ou outra *thread* tiver feito *lock* no mutex.

O próximo exemplo ilustra o uso de variáveis mutex em um programa. Os principais dados estão disponíveis para todas as *threads* através de uma estrutura global. Cada *thread* trabalha em uma parte diferente dos dados. A *thread* aguarda por todas as *threads* para completar sua execução e imprime, em seguida, os resultados do somatório.

Exemplo:

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

/*
Estrutura que contém as informações necessárias para permitir
que a função "dotprod" acesse seus dados de entrada
localizando-os em uma estrutura de saída.
*/
struct DOTDATA {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 };

/* Variáveis e definições globais */
#define NUMTHRDS 4
#define VECLEN 100
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

void *dotprod(void *arg)
{
   int i, start, end, offset, len ;
   double mysum, *x, *y;
   offset = (int)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
```

```
    x = dotstr.a;
    y = dotstr.b;
    mysum = 0;
    for (i=start; i<end ; i++)
       mysum += (x[i] * y[i]);


    pthread_mutex_lock (&mutexsum);
    /* Inicio da Região Crítica */
      dotstr.sum += mysum;
    /* Fim da Região Crítica */
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

int main()
{
    int i;
    double *a, *b;
    int status;
    pthread_attr_t attr;

    a = new double[NUMTHRDS*VECLEN];
    b = new double[NUMTHRDS*VECLEN];

    for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
       a[i]=1.0;
       b[i]=a[i];
    }

    dotstr.veclen = VECLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    pthread_mutex_init(&mutexsum, NULL);

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0; i<NUMTHRDS; i++)
      pthread_create(&callThd[i], &attr, dotprod, (void *)i);

    pthread_attr_destroy(&attr);

    for(i=0; i<NUMTHRDS; i++)
      pthread_join( callThd[i], (void **)&status);
```

```
    printf ("Sum =  %f \n", dotstr.sum);
    delete [] a;
    delete [] b;
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Variáveis Condicionais

Variáveis condicionais provêem outra forma de sincronizar *threads.* Enquanto mutexes implementam sincronização controlando as *threads* no acesso aos dados, variáveis condicionais sincronizam as *threads* com base nos valores atuais dos seus dados.

Uma variável condicional é sempre utilizada em conjunto com um mutex.

As seguintes rotinas criam e destróem variáveis condicionais:

- `ptread_cond_init (condition, attr)`
- `ptread_cond_destroy (condition)`
- `ptread_condattr_init (attr)`
- `ptread_condattr_destroy (attr)`

Variáveis condicionais devem ser declaradas com o tipo:

`pthread_cond_t`

Existem duas formas de iniciar variáveis condicionais:

1. Estaticamente, quando ela é declarada:

```
pthread_cond_t meuCond = PTHREAD_COND_INITIALIZER;
```

2. Dinamicamente, através da função `pthread_cond_init()`.

Waiting e *Signaling* em Variáveis Condicionais

As seguintes rotinas são usadas com as variáveis condicionais para bloquear e liberar *threads*:

- `ptread_cond_wait (condition, mutex)`
- `ptread_cond_signal (condition)`
- `ptread_cond_broadcast (condition)`

A função `pthread_cond_wait()` bloqueia a *thread* chamadora até que uma condição seja sinalizada. Esta rotina deve ser chamada enquanto um mutex está bloqueado e irá automaticamente liberar o mutex enquanto a *thread* aguarda.

Após o sinal ser recebido e a *thread* voltar a executar, o mutex irá automaticamente ser bloqueado para o uso da *thread.*

O programador é responsável por liberar o mutex quando a *thread* terminar de utilizá-lo.

A função `pthread_cond_signal()` é usada para sinalizar as outras *threads* que a variável condicional está no estado *waiting.* Esta rotina deve ser chamada após o mutex ser bloqueado, e deve destravar o mutex para que a rotina `pthread_cond_wait()` complete sua execução.

A função `pthread_cond_broadcast()` deve ser usada se mais de uma *thread* estiver no estado de bloqueado.

O próximo exemplo demonstra o uso destas rotinas. A função principal cria três *threads.* Duas *threads* modificam a variável `count`. A terceira *thread* espera até que a variável `count` alcance o valor especificado.

Exemplo:

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  int *my_id = (int *)idp;

  for (i=0; i<TCOUNT; i++) {
```

```
    pthread_mutex_lock(&count_mutex);
    count++;
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count():thread %d, count = %d  Threshold
              reached.\n", *my_id, count);
    }
    printf("inc_count(): thread %d, count = %d, unlocking
            mutex\n",  *my_id, count);
    pthread_mutex_unlock(&count_mutex);

   /* Realiza algum processamento */
    for (j=0; j<1000; j++)
      result = result + (double)random();
  }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  int *my_id = (int *)idp;

  printf("Starting watch_count(): thread %d\n", *my_id);

  pthread_mutex_lock(&count_mutex);
  if (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
            received.\n", *my_id);
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main ()
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count,
                 (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count,
                 (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count,
                 (void *)&thread_ids[2]);
```

```
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n",
          NUM_THREADS);

  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

7 Estudo de Caso: *Threads* na Linguagem Builder C++

O Builder C++ provê a classe `TThread` e também um *wizard* que acrescenta uma unit ao projeto corrente, com o código já estruturado para a implementação de uma nova *thread.*

Usando a classe `TThread` pode-se criar *threads* sem a necessidade do uso direto da Win32 (função `CreateThread`).

Cada objeto *TThread* (ou alguma classe derivada) corresponde a um *thread.*

Todas as *threads* de um mesmo processo possuem acesso às mesmas variáveis globais, aos mesmos objetos, e também ao mesmo código da VCL (*Visual Component Library*)

Não se usa a classe `TThread` diretamente, porque ela é uma classe abstrata (uma classe com um método virtual abstrato).

Desta forma, é necessário criar classes derivadas usando os recursos da classe base. A classe `TThread` possui um construtor com um único parâmetro que permite escolher se o processo deve iniciar imediatamente ou ficar suspenso até mais tarde:

```
__fastcall TThread(bool CreateSuspended);
```

Há também alguns métodos públicos para sincronização entre *threads*:

```
void __fastcall Resume(void);
void __fastcall Suspend(void);
void __fastcall Terminate(void);
```

```
int __fastcall WaitFor(void);
```

As propriedades públicas incluem `Priority`, `Suspended`, e dois valores de baixo nível de somente-leitura: `Handle` e `ThreadID`.

A classe `TThread` também propicia uma interface protegida que inclui dois métodos principais para as subclasses dos processos:

```
virtual void __fastcall Execute(void) = 0;

typedef void __fastcall (__closure *TThreadMethod)(void);
void __fastcall Synchronize(TThreadMethod &Method);
```

O método `Execute()`, declarado como um métdo virtual puro, precisa ser redefinido em cada classe. Ele deverá conter o código principal da *thread*.

O método `Synchronize()` é usado para evitar acesso simultâneo aos componentes da VCL. Ele funciona como um monitor evitando que o programador seja forçado a utilizar outros mecanismos de sincronização como semáforos.

Exemplo:

Este exemplo usa o método `Synchronize()`. O programa usa um processo para desenhar a superfície de um formulário.

A classe do processo, a `TPainterThread`, sobrescreve o método `Execute()` e define um método particular chamado `Paint()`. O método `Paint()` é usado para acessar os objetos da VCL, de modo que estes são chamados somente de dentro do método `Synchronize()`.

Como o método `Paint()` não pode ter parâmetros diretamente e deve ser um argumento compatível com o método `Synchronize()`. A classe requer alguns dados privados. Eis a declaração da classe:

```
class TPainterThread : public TThread
{
private:
        int X, Y;
        TColor Cl;
protected:
        void __fastcall Execute();
        void __fastcall Paint();
public:
```

```
        __fastcall TPainterThread(TColor C);
};
```

Foi acrescentado um construtor à classe para passar um valor inicial de cor ao processo:

```
__fastcall TPainterThread::TPainterThread(TColor C)
  : TThread(true)
{
  Cl = C;
}
```

O construtor inicializa os dados privados e depois chama o construtor da classe base, criando o processo em um estado suspenso.

O método `Execute()` do processo simplesmente examina cada linha da tela, desenhando cada *pixel* com a cor fornecida:

```
void __fastcall TPainterThread::Execute()
{
  randomize();
  do {
    X = random(300);
    Y = random (Form1->ClientHeight);
    for (int i=0; i<10000; i++);
    Synchronize (Paint);
  } while (!Terminated);
}
```

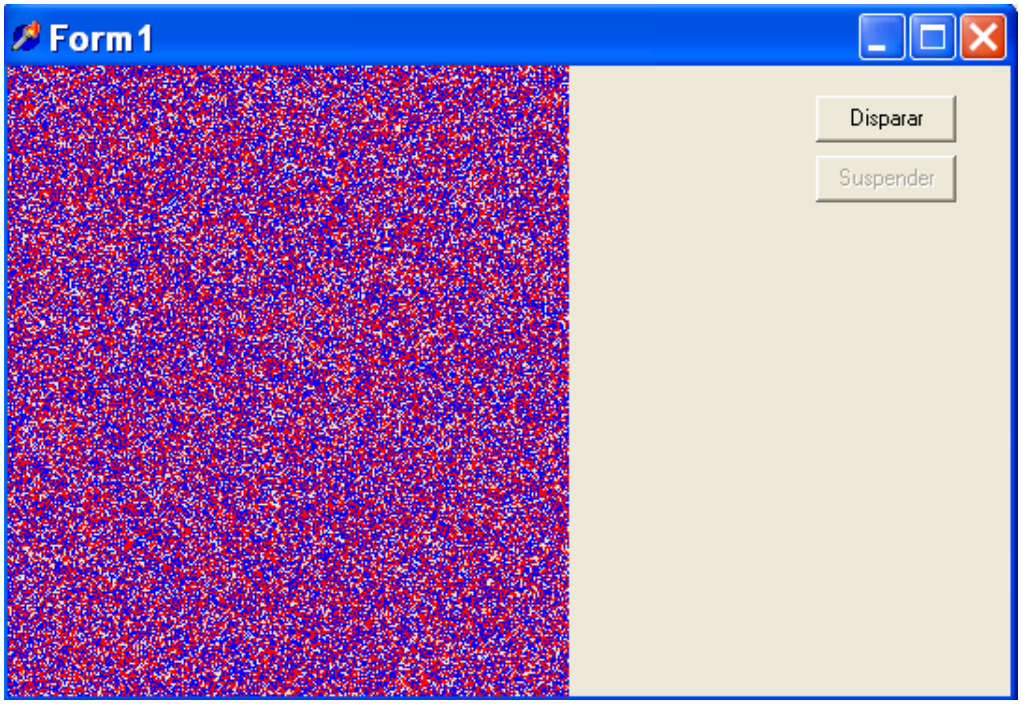
O método `Paint()` desenha *pixels* no formulário:

```
void __fastcall TPainterThread::Paint()
{
  Form1->Canvas->Pixels[X][Y] = Cl;
}
```

O formulário principal possui um vetor para guardar três objetos *thread:*

```
private:
  TpainterThread *PT[3];
```

São criadas 3 *threads*, quando o formulário é criado, cada uma colore os *pixels* de uma cor diferente:



Técnicas de Sincronização do Windows

As funções da API do Windows oferecem muitas outras técnicas de sincronização (disponíveis nas plataformas Win32):

- ***Seções Críticas****:* São parcelas do código fonte que não podem ser executadas por dois processos ao mesmo tempo.

- ***Os mutexes***: são objetos globais que podem ser usados para serializar o acesso a um recurso.

- ***Semáforos***: são semelhantes aos mutexes, mas são contados. Um *mutex* é semelhante a um semáforo com contagem máxima de 1. Além disso, podem ser manipulados por *threads* de processos distintos.

- ***Eventos***: podem ser usados como um meio de sincronizar um processo com os eventos do sistema, como nas operações com arquivos do usuário. Também podem ser usados para ativar vários processos ao mesmo tempo.

Exemplo:

Supondo que haja dois processo operando uma string, ambos usando seu valor de algum modo e depois atualizando a string como resultado da operação.

Supondo, também, que o valor atual da string é compartilhado pelos dois processos. No exemplo, a string contém inicialmente 20 caracteres *A*, depois é atualizado para conter 20 caracteres *B*, e assim por diante.

Cada processo simplesmente calcula o valor seguinte da string e depois o envia para sua própria caixa de lista.

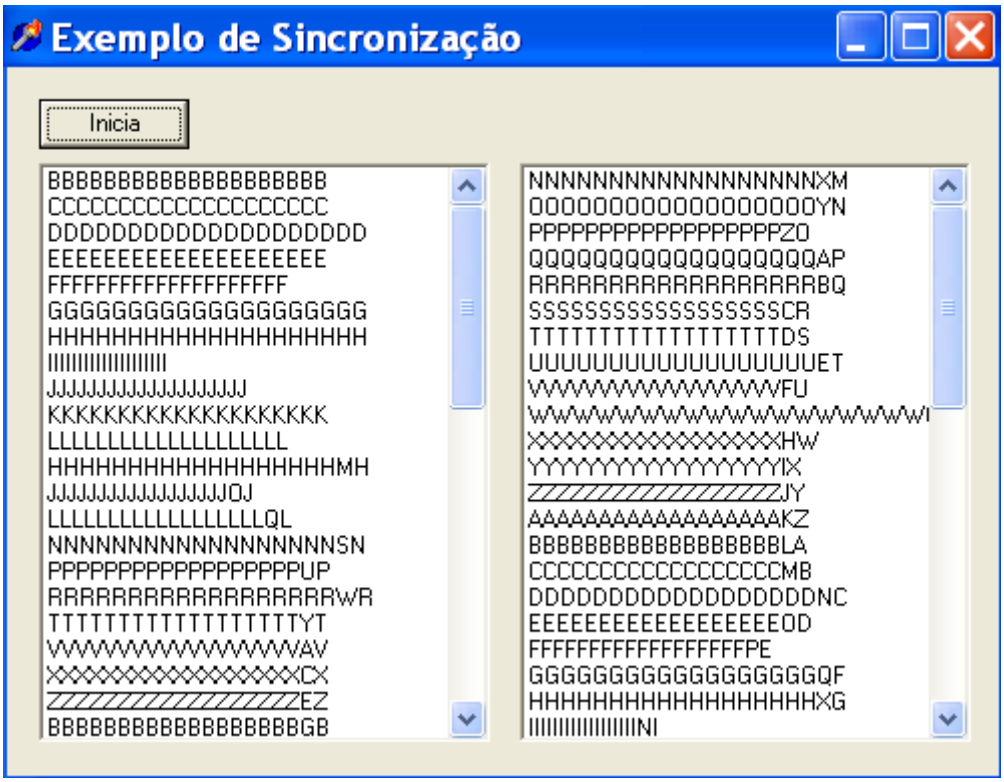
Eis a classe do processo e seu método `Execute()` em uma primeira versão mais simples:

```
class TListThread : public TThread
{
private:
  AnsiString Str;
protected:
        void __fastcall Execute();
        void __fastcall AddToList();
public:
        TListBox *LBox;
        __fastcall TListThread(TListBox *LB);
};

void __fastcall TListThread::Execute()
{
  for (int i=0; i<50; i++)
  {
    for (int j=1; j<20; j++)
      for (int k=0; k<2600; k++)
        if (Letters[j] != 'Z')
          Letters[j]++;
        else
          Letters[j] = 'A';
    Str = Letters;
    Synchronize(AddToList);
  }
}
```

O método `AddToList()` simplesmente adiciona a `Str` à caixa de lista conectada com o processo.

O método `Execute()` é artificialmente longo, aumentando cada letra 2600 vezes em vez de uma: o efeito é o mesmo, mas assim há mais chances de ocorrer conflito entre dois processos, ou seja, condição de disputa.



Usando Seções Críticas

Uma forma de sincronizar é utilizando seções críticas. Para fazer isso deve-se acrescentar a declaração de outra variável global para a seção crítica:

```
TRTLCriticalSection RCritica;
```

Essa variável é inicializada quando o formulário é criado e é destruída ao final, com duas chamadas da API:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  InitializeCriticalSection(&RCritica);
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  DeleteCriticalSection(&RCritica);
}
```

Pode-se, então, usar as seções críticas para serializar partes específicas do código, como o código que atualiza cada letra da string. Eis como foi atualizado o código do método `Execute()`:

```
void __fastcall TListThread::Execute()
{
  for (int i=0; i<50; i++)
  {
    EnterCriticalSection(RCritica);
    for (int j=1; j<20; j++)
      for (int k=0; k<2600; k++)
        if (Letters[j] != 'Z')
          Letters[j]++;
        else
          Letters[j] = 'A';
    Str = Letters;
    LeaveCriticalSection(RCritica);
    Synchronize(AddToList);
  }
}
```

Usando Mutex

Pode-se escrever o mesmo código, mas usando um mutex em vez de uma seção crítica. O efeito é o mesmo. As modificações no código são apresentadas abaixo:

```
  Thandle *Semaforo;

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Semaforo = (Thandle *)CreateSemaforo(NULL, 1, 1, NULL);
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  CloseHandle(Mutex);
}

void __fastcall TListThread::Execute()
{
  for (int i=0; i<50; i++)
  {
    WaitForSingleObject(Mutex, INFINITE);
    for (int j=1; j<20; j++)
      for (int k=0; k<2600; k++)
        if (Letters[j] != 'Z')
          Letters[j]++;
        else
```

```
            Letters[j] = 'A';
      Str = Letters;
      ReleaseMutex(Mutex);
      Synchronize(AddToList);
    }
  }
```

Usando Semáforos

A seguir o mesmo código utilizando semáforos:

```
  Thandle *Mutex;

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Mutex = (Thandle *)CreateMutex(NULL, false, NULL);
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  CloseHandle(Mutex);
}

void __fastcall TListThread::Execute()
{
  for (int i=0; i<50; i++)
  {
    WaitForSingleObject(Semaforo, INFINITE);
    for (int j=1; j<20; j++)
      for (int k=0; k<2600; k++)
        if (Letters[j] != 'Z')
          Letters[j]++;
        else
          Letters[j] = 'A';
    Str = Letters;
    ReleaseMutex(Semaforo, 1, NULL);
    Synchronize(AddToList);
  }
}
```

Bibliografia:

DEITEL, H. M., DEITEL, P. J. & CHOFFNES, D. R.
Sistemas Operacionais
3a. Edição: Person (2005) – São Paulo / SP

MACHADO, F. B. & MAIA, L. P.
Arquitetura de Sistemas Operacionais
3a. Edição: LCT (2002) – Rio de Janeiro / RJ

SILBERSCHATZ, ABRAHAM & GALVIN, PETER B.
Operating System Concepts
5th Edition: John Wiley (1999) – Massachusetts

STALLINGS, WILLIAM
Operating Systems
2nd Edition: Prentice-Hall (1995)

TANENBAUM, ANDREW S.
Sistemas Operacionais Modernos
2ª. Edição. Person (2003) – São Paulo / SP

TANENBAUM, ANDREW S. & WOODHULL, ALBERT S.
Operating Systems: Design and Implementation
2nd Edition: Prentice-Hall (1997) – Upper Saddle River / NJ

TOSCANI, S. S.; OLIVEIR, R. S. & CARISSIMI, A. S.
Sistemas Operacionais e Programação Concorrente
Editora Sagra-Luzzatto (2003) – Porto Alegre / RS

http://www.llnl.gov/computing/tutorials/pthreads/