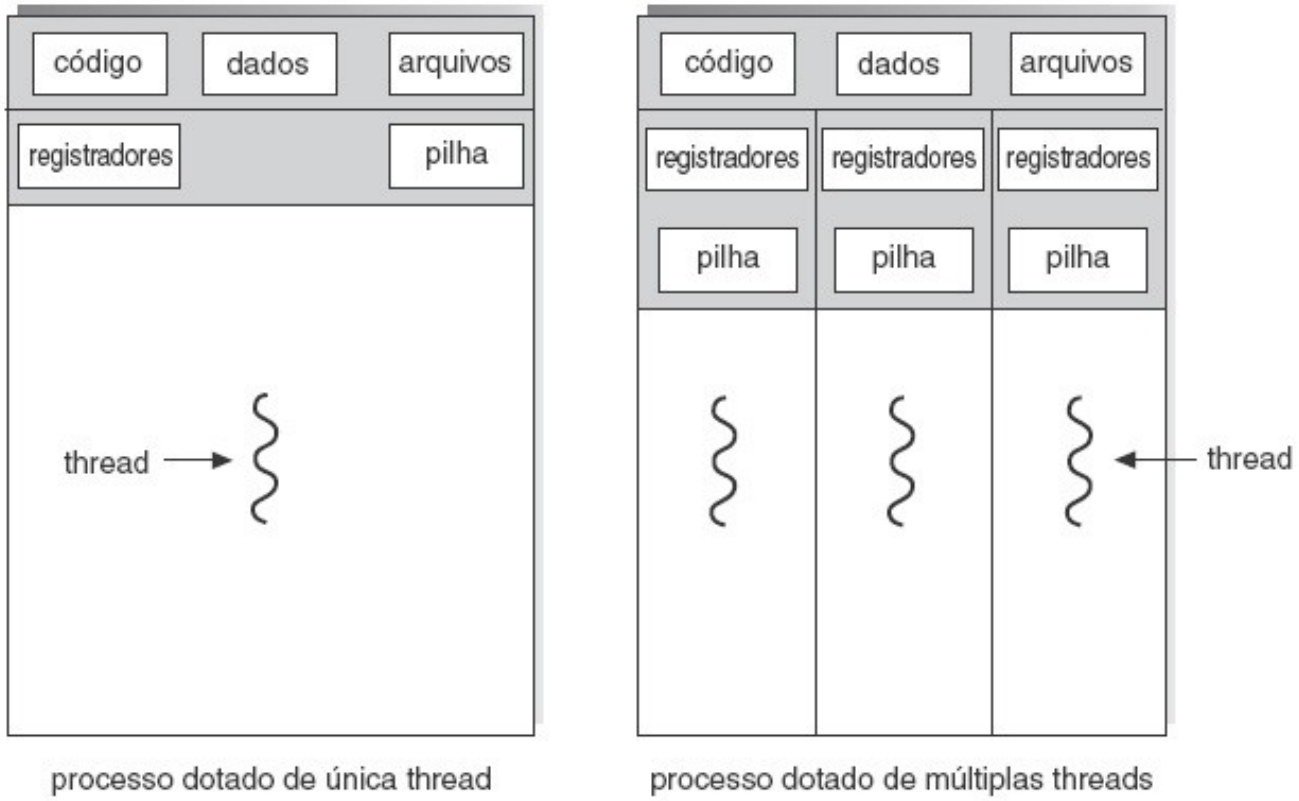
Threads

- Visão geral
- Modelos multithreads
- Aspectos do uso de threads
- Pthreads
- Threads no Linux

Threads

- Uma **thread** ou processo leve (*LPW – lightweight process*) é uma unidade básica da utilização da CPU; ela consiste em:
  - ID
  - contador de programa e registradores
  - pilha
- Compartilha com outras threads pertencentes ao mesmo processo:
  - seção de código e de dados
  - recursos do sistema operacional (arquivos abertos)
- Um processo tradicional, ou pesado, tem uma única thread de controle
- Um processo multithreaded pode realizar várias tarefas ao mesmo tempo

Processos com Thread Única e Multithread



processo dotado de única thread

processo dotado de múltiplas threads

Threads

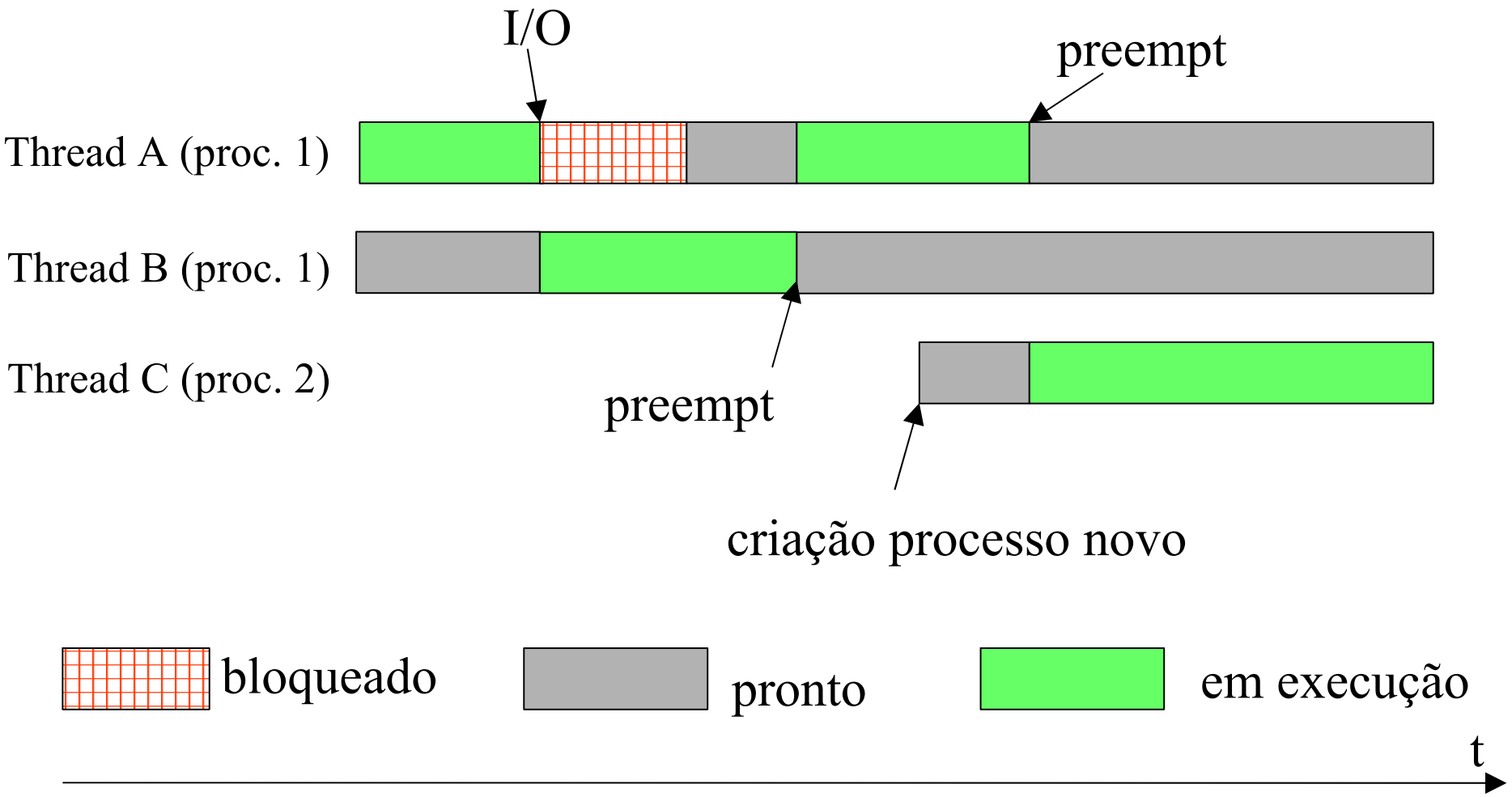
- Em uma tarefa com múltiplas threads, enquanto uma thread servidora está bloqueada e esperando, uma segunda thread da mesma tarefa pode ser executada
  - Múltiplas threads na mesma tarefa confere maior *throughput* e melhor desempenho
  - As aplicações que precisam compartilhar um buffer comum (ou seja, produtor-consumidor) tiram proveito da utilização de threads

- Threads permitem que processos seqüenciais façam chamadas de sistema bloqueantes ao mesmo tempo em que obtêm paralelismo

Vantagens

- Responsividade
- Compartilhamento de recursos
- Economia (custo de criação e troca de contexto)
- Utilização de arquiteturas multiprocessadas

Exemplo


I/O
preempt
Thread A (proc. 1)
Thread B (proc. 1)
Thread C (proc. 2)
preempt
criação processo novo
bloqueado
pronto
em execução
t

Threads de Usuário

- Suportado acima do kernel, através de um conjunto de chamadas de biblioteca no nível do usuário (User Level Threads)

- Três principais bibliotecas de threads:
  - POSIX Pthreads
  - Threads Java
  - Threads Win32

Threads de kernel

- Suportado pelo kernel através de uma biblioteca implementada no nível do kernel (Kernel Level Threads). A chamada de uma função resulta em uma chamada de sistema

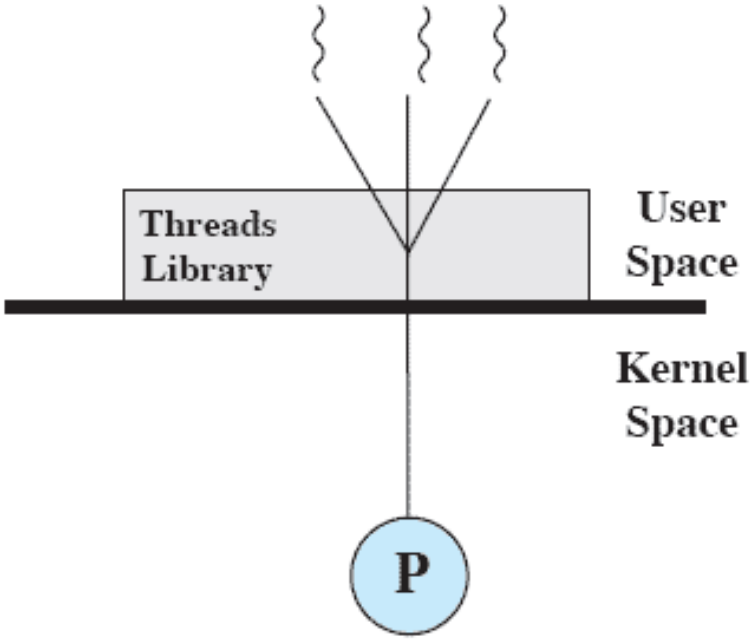
- Exemplos
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Modelos Multithreads

Relação entre threads de usuário e de kernel:

- Muitos-para-um
- Um-para-um
- Muitos-para-muitos

Modelo Muitos-para-um

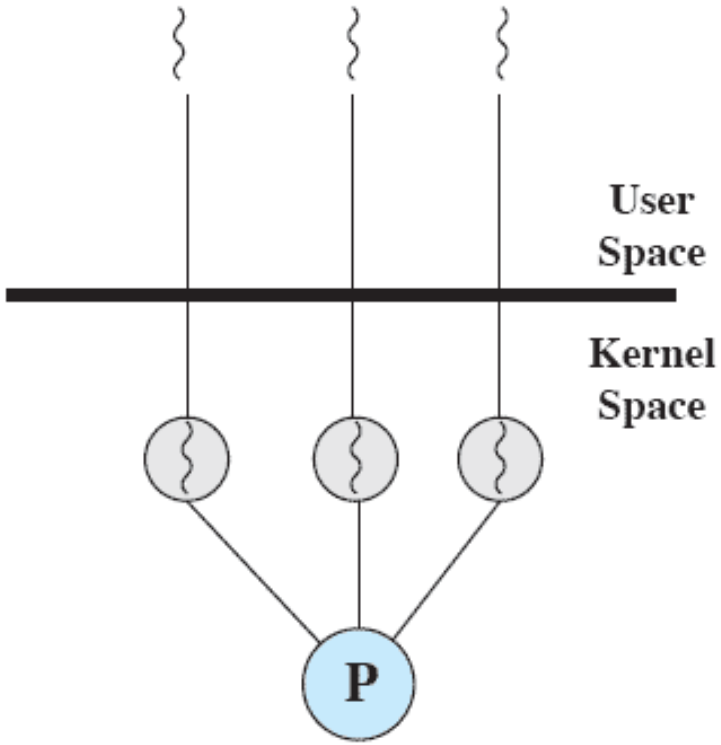


(a) Pure user-level

Muitos-para-um

- Muitas threads no nível do usuário associadas a uma única thread de kernel
- Gerenciamento através de uma biblioteca no espaço de usuário
- É eficiente, mas o processo inteiro é bloqueado quando uma thread faz uma chamada de sistema bloqueante
- Não aproveita arquiteturas multiprocessadas
- Exemplos
  - Solaris Green Threads
  - GNU Portable Threads

Modelo Um-para-Um

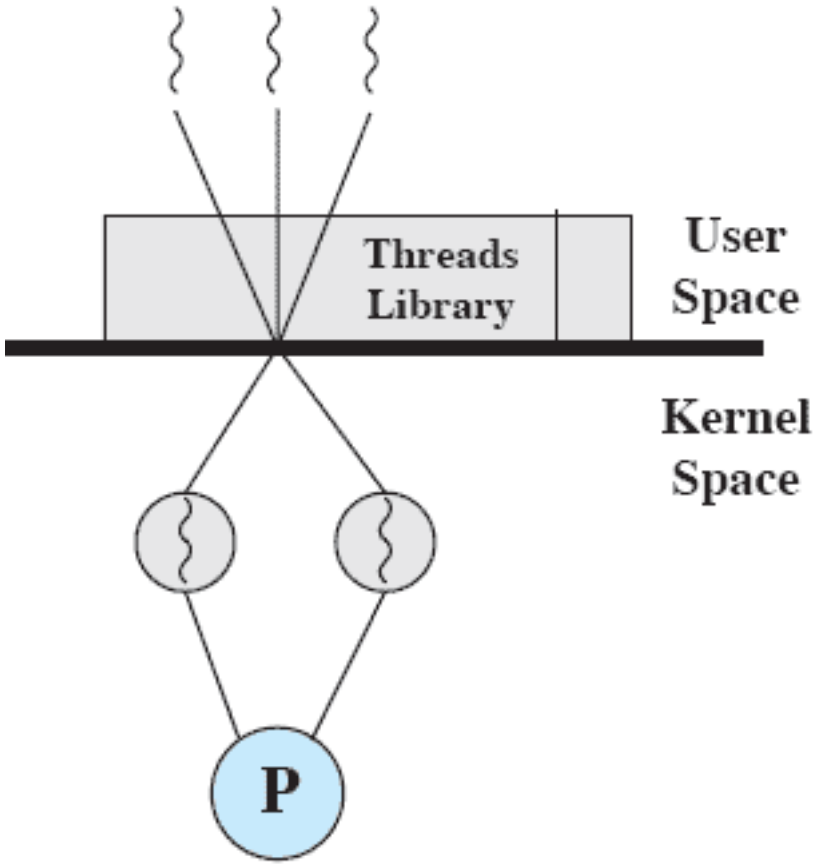


(b) Pure kernel-level

Um-para-um

- Cada thread do usuário associada a uma thread de kernel
- Permite o escalonamento de outra thread quando uma faz uma chamada de sistema bloqueante
- Threads executam em paralelo em multiprocessadores
- Menos eficiente devido ao custo das chamadas de sistema, limitando o número de threads
- Exemplos
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 e acima

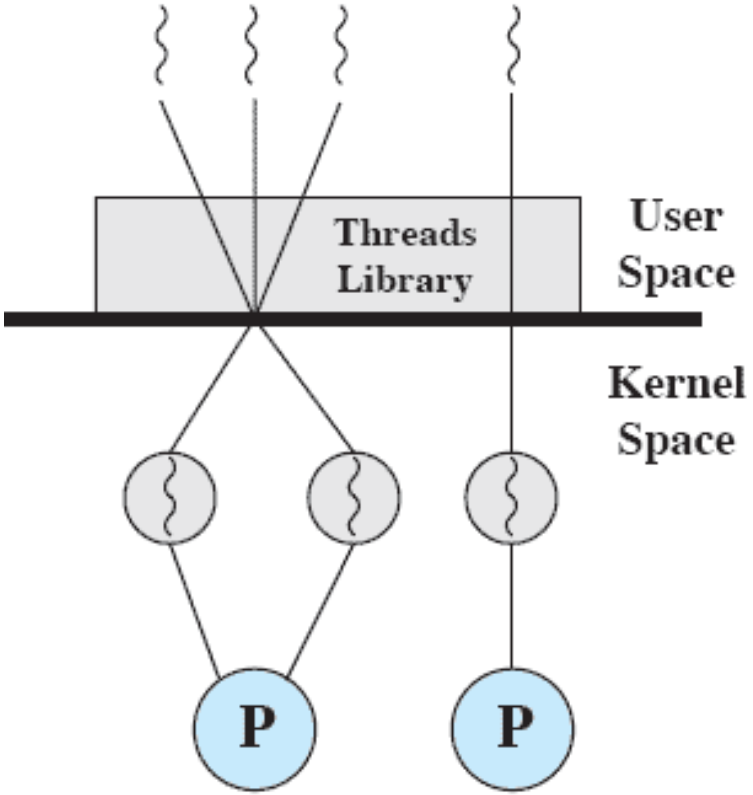
Modelo Muitos-para-Muitos



Muitos-para-muitos

- Permite que muitas threads no nível do usuário sejam associadas a muitas threads no nível do kernel
- Permite ao usuário criar quantas threads forem necessárias, pois o sistema operacional irá criar um número suficiente de threads de kernel
- Exemplos
  - Solaris antes da versão 9
  - Windows NT/2000 com o pacote *ThreadFiber*

Modelo em Dois Níveis



(c) Combined

Modelo em Dois Níveis

- Semelhante ao modelo M:M, exceto que permite que uma thread do usuário seja associada a thread do kernel

- Exemplos
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris até a versão 8

Aspectos do Uso de Threads

- Semântica das chamadas de sistema **fork()** e **exec()**
- Cancelamento da thread
- Tratamento de sinais
- Bancos de threads

Semântica do fork() e exec()

- **fork()** duplica apenas a thread que o chama ou todas as threads do processo?
- Alguns sistemas oferecem duas versões de fork()
- **exec()** substitui o processo inteiro (todas as threads) pelo programa especificado

Cancelamento da Thread

- Termina uma thread antes que tenha sido completada

- Dois métodos gerais:
  - **Cancelamento assíncrono** – termina a thread-alvo imediatamente
  - **Cancelamento adiado** – permite que a thread-alvo verifique periodicamente se deve terminar

Tratamento de Sinais

- Sinais são usados nos sistemas UNIX para notificar um processo de que um evento específico ocorreu
- Um sinal é recebido de forma síncrona ou assíncrona
- Um **tratador de sinal** é usado para processar sinais
  1. Um sinal é gerado por um evento em particular
  2. O sinal é entregue a um processo
  3. O sinal é tratado

Tratamento de Sinais

- Tratador de sinal padrão ou tratador de sinal definido pelo usuário
- Em qual thread um sinal deve ser entregue?
  - Entregar o sinal à thread que o sinal se aplica (sinais síncronos)
  - Entregar o sinal a cada thread do processo (ex.: término de processo)
  - Entregar o sinal a certas threads do processo (definir os sinais aceitos e os bloqueados)
  - Atribuir uma thread específica para receber todos os sinais para o processo

Bancos de Threads

- Uma série de threads são criadas em um banco, onde esperam para atuar
- Vantagens:
  - Em geral, é um pouco mais rápido atender a uma requisição com uma thread existente do que criar uma nova thread
  - Permitem que o número de threads da(s) aplicação (ções) seja limitado ao tamanho do banco

Pthreads

- Uma API do padrão POSIX (IEEE 1003.1c) para a criação e sincronismo de threads
- A API especifica o comportamento da biblioteca de threads; a implementação cabe ao desenvolvimento da biblioteca
- Comuns nos sistemas operacionais UNIX (Solaris, Linux, Mac OS X)

Pthreads

- Toda thread possui uma thread ID (**TID**), que tem sentido no contexto do processo.
- TID é representado pela estrutura de dados **pthread_t**
- A função **pthread_equal** é usada para comparar dois TIDs.
- Uma thread pode obter seu TID chamando a função **pthread_self**.

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
pthread_t pthread_self(void);
```

Pthreads

```
#include <pthread.h>

int pthread_create(pthread_t *tidp,
                   const pthread_attr_t *attr,
                   void *(*start_rtn)(void), void *arg);
```

- Um programa inicia sua execução como um processo com uma única thread de controle. Threads adicionais podem ser criadas chamando a função **pthread_create**
- **tidp** aponta para o TID da nova thread criada.
- O argumento **attr** é usado para customizar vários atributos. Para utilizar os atributos default, este parâmetro deve ser NULL.

Pthreads

```
#include <pthread.h>

int pthread_create(pthread_t *tidp,
                   const pthread_attr_t *attr,
                   void *(*start_rtn)(void), void *arg);
```

- A nova thread inicia executando a função **start_rtn**. Esta função recebe um único argumento (**arg**). Caso seja necessário passar mais do que um argumento, deve-se armazená-los em uma estrutura e passar o endereço da estrutura em **arg**.

Pthreads

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
pthread_t ntid;

void printids(const char *s)
{
    pid_t      pid;
    pthread_t  tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}
```

Pthreads

```
void *thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0) {
        perror("can't create thread”);
        exit(1);
    }
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Pthreads

- Se qualquer thread de um processo chamar exit, ou _exit, o processo inteiro termina.
- Uma thread pode terminar seu fluxo de controle sem terminar o processo inteiro de três formas:
  1. A thread pode retornar da rotina inicial (o valor de retorno é o código de saída da thread)
  2. A thread pode ser cancelada por outra thread do mesmo processo.
  3. A thread pode chamar pthread_exit.

Pthreads

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
int pthread_join(pthread_t thread, void **rval_ptr);
int pthread_cancel(pthread_t tid);
```

- O **rval_ptr** é um ponteiro disponibilizado a outras threads do processo através da chamada à função **pthread_join**.
- Ao chamar **pthread_join**, a thread é bloqueada até a thread especificada terminar.
- Se a thread retornar da sua rotina inicial, **rval_ptr** irá conter o código de retorno. Se a thread foi cancelada, será retornado o valor PTHREAD_CANCELED.
- Uma thread pode cancelar outra thread do mesmo processo através da função **pthread_cancel**.

Pthreads

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
pthread_t ntid;

void *thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}
```

Pthreads

```
Int main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0) {
        perror("can't create thread 1”); exit(1); }
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0) {
        perror("can't create thread 2"); exit(1); }
    err = pthread_join(tid1, &tret);
    if (err != 0) {
        perror("can't join with thread 1"); exit(1); }
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0) {
        perror("can't join with thread 2"); exit(1); }
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

Pthreads

```
int sum; /* esses dados são compartilhados pela(s) Thread(s)  */
void *runner(void *param);          /* a thread*/

int main(int argc, char **argv)
{
    pthread_t tid;                          /* o identificador da Thread */
    pthread_attr_t attr;    /* conjunto de atributos para a Thread */

    pthread_attr_init(&attr);           /* obtém os atributos padrão */
    pthread_create(&tid,&attr,runner,argv[1]); /* cria a thread  */
    /* agora espera que o fluxo termine  */
    pthread_join(tid,NULL);
    printf("soma = %d\n",sum);
    exit(0);
}
```

Pthreads

```
void *runner(void *param)
{
    int upper = atoi(param);
    int i;

    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
        sum += i;
    }
    pthread_exit(0);
}
```

Threads no Linux

- O Linux refere-se a elas como *tarefas*, em vez de *threads*
- A criação de thread é feita através da chamada de sistema **clone()**
- **clone()** permite que uma tarefa filha compartilhe o espaço de endereços da tarefa pai (o processo)

Exercício

1. Fazer um programa que cria uma thread e espera por sua finalização. A thread criada deve conter um loop infinito. Use os comandos abaixo enquanto o programa está executando:

   # ps -aLf

   # top -H -p <pid>

2. Faça um programa que declara uma variável global e cria (fork) um outro processo. O processo filho lê esta variável global e a altera. Imprima o valor desta variável no processo pai, depois do término do filho.
3. Faça uma versão do programa anterior usando threads.
4. Faça um programa para somar os elementos de um vetor usando threads. Crie 4 threads.

Sincronização em Pthread: Mutexes

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

- Uma variável mutex é do tipo **pthread_mutex_t** e pode ser inicializada atribuindo a constante PTHREAD_MUTEX_INITIALIZER (alocação estática) ou através da função **pthread_mutex_init**
- Para inicializar um mutex com os atributos default, o argumento **attr** deve ser NULL

Mutexes

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Se a variável for alocada dinamicamente, é necessário chamar a função **pthread_mutex_destroy** antes de desalocá-la

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Ao chamar **pthread_mutex_lock**, se o **mutex** já estiver “locked”, a thread será bloqueada até que o **mutex** seja “unlocked”
- Se o **mutex** estiver “unlocked” quando **pthread_mutex_trylock** é chamado, o **mutex** será “locked” e a função retornará 0. Caso contrário, a função retornará EBUSY não bloqueando a thread

ReaderWriter Locks

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- Para inicializar um readerwriter lock com os atributos default, chama-se a função **pthread_rwlock_init** com o argumento **attr** em NULL
- Se o lock for alocado dinamicamente, é necessário chamar a função **pthread_rwlock_destroy** antes de desalocá-lo

ReaderWriter Locks

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- **pthread_rwlock_rdlock** adquire o lock em modo de leitura (modo compartilhado)
- **pthread_rwlock_wrlock** adquire o lock em modo de escrita (modo exclusivo)
- Independente de como o lock foi adquirido, usa-se **pthread_rwlock_unlock** para liberá-lo

ReaderWriter Locks

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

- Quando o lock está livre para ser adquirido, estas funções retornam 0. Caso contrário, elas retornam EBUSY.

Trabalho (entrega - 29/10)

- Implementar dois programas paralelos iterativos usando processos e threads e comparar o overhead de execução
  - Um programa seqüêncial iterativo é um que usa loops (for ou while) para ler dados e calcular resultados
  - Um programa paralelo iterativo contém dois ou mais processos/threads iterativos. Cada processo/thread calcula os resultados para um subconjunto de dados e então os resultados são combinados

Multiplicação de Matrizes

```
double  a[n][n], b[n][n], c[n][n];
for (i=0; i < n; i++) {
    for (j = 0; j < n; j++ ) {
        c[i,j] = 0.0;
        for (k = 0; k < n; k++)
            c[i,j] += a[i,k] * b[k,j];
    }
}
```

Multiplicação de Matrizes

- Aplicação variando-se o número de threads ou processos de 1 ao número de processadores
- Variar o tamanho da matriz
- Executar em modo exclusivo
- Tirar uma média do tempo de execução
- Entregar relatório com 2 gráficos: um com o tempo de execução para diversos tamanhos de matriz comparando a execução com processos e threads; e outro com o ***speedup*** (tempo exec. 1 processador / tempo exec. em n procs.) variando o número de processadores.