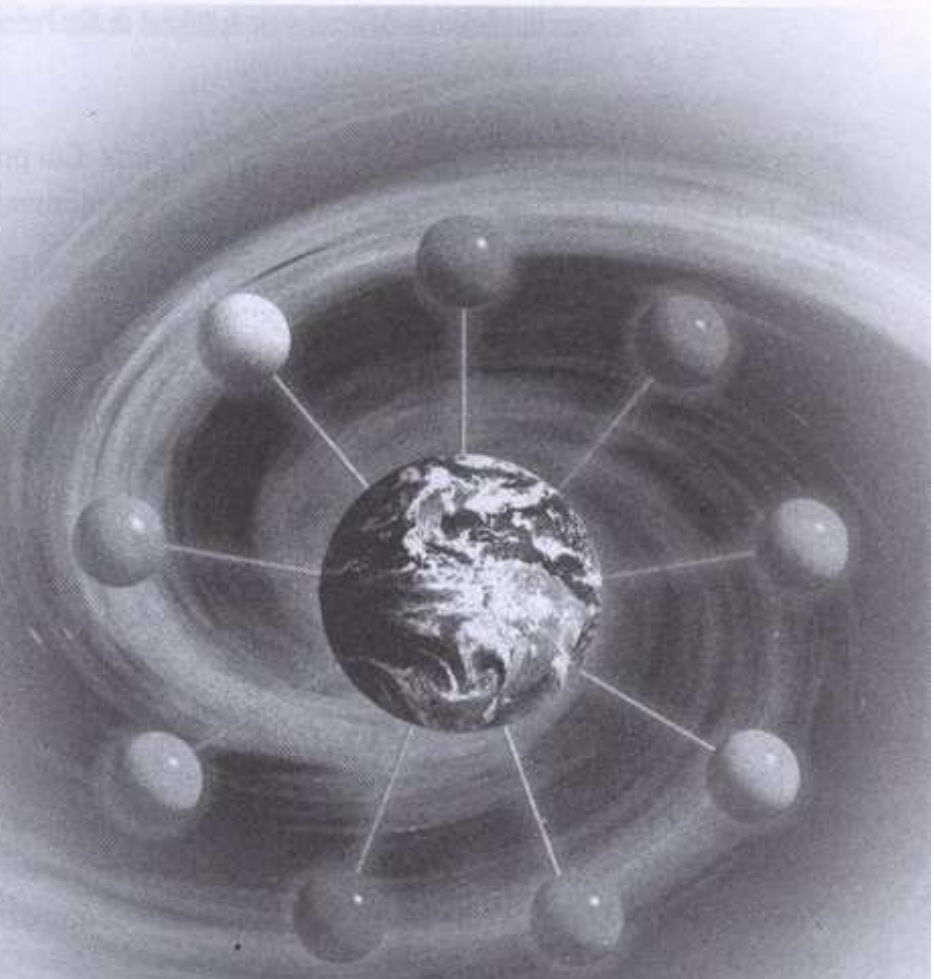
CAPÍTULO



4

Processos Leves (Threads)

Os processos leves são mais conhecidos como *threads*. Em poucas palavras, um *thread* é uma parte de um programa que é executado concorrentemente.

A maioria dos programas corresponde somente a um único fluxo de controle. Uma única função principal – *void main( )* – que executa todas as instruções. Imagine que as tarefas exercidas por este programa possam ser separadas em diferentes funções e que estas possam ser executadas concomitantemente com a função principal. Desta forma, cada uma das funções corresponderia a um fluxo de controle independente e a isto se dá o nome de processos leves.

O uso de processos leves permite que o programador estruture o seu código num conjunto de atividades independentes, de forma que elas possam ser executadas em concorrência, permitindo o paralelismo da aplicação. Este conceito também é utilizado na primitiva *fork( )*. A diferença básica é a seguinte:

Um processo é composto de cinco partes: segmento de código, segmento de dados, pilha, segmento de entrada e saída e tabela de sinais. Quando um processo sofre uma cópia através do *fork( )* uma quantidade enorme de dados precisa ser copiada na memória e muitos ciclos de CPU são necessários. A única parte compartilhada entre o processo pai e filho é o segmento de código, onde ficam as instruções. Outra desvantagem é que os processos pai e filho só podem se comunicar através de *pipes*, memória compartilhada, sinais e *sockets*. A interação entre eles é difícil.

As *threads* reduzem o gasto desnecessário de memória e processamento, uma vez que o processo principal e os processos leves compartilham, além do segmento de código, o segmento de dados e espaço de endereçamento. Elas possuem apenas sua própria pilha de execução e o seu próprio programa *counter*. Cada uma roda um código seqüencial, definido no programa de aplicação do processo pesado no qual foram definidas.

Portanto, quando uma *thread* altera um valor compartilhado ou abre um descritor de arquivo, todos os outros processos leves ou mesmo o processo pesado podem ter acesso a eles. As *threads* podem até mesmo criar outros processos leves filhos. E a comunicação fica mais simples, pois tudo é compartilhado.

"A simplicidade é o último degrau da sabedoria."

Gibran

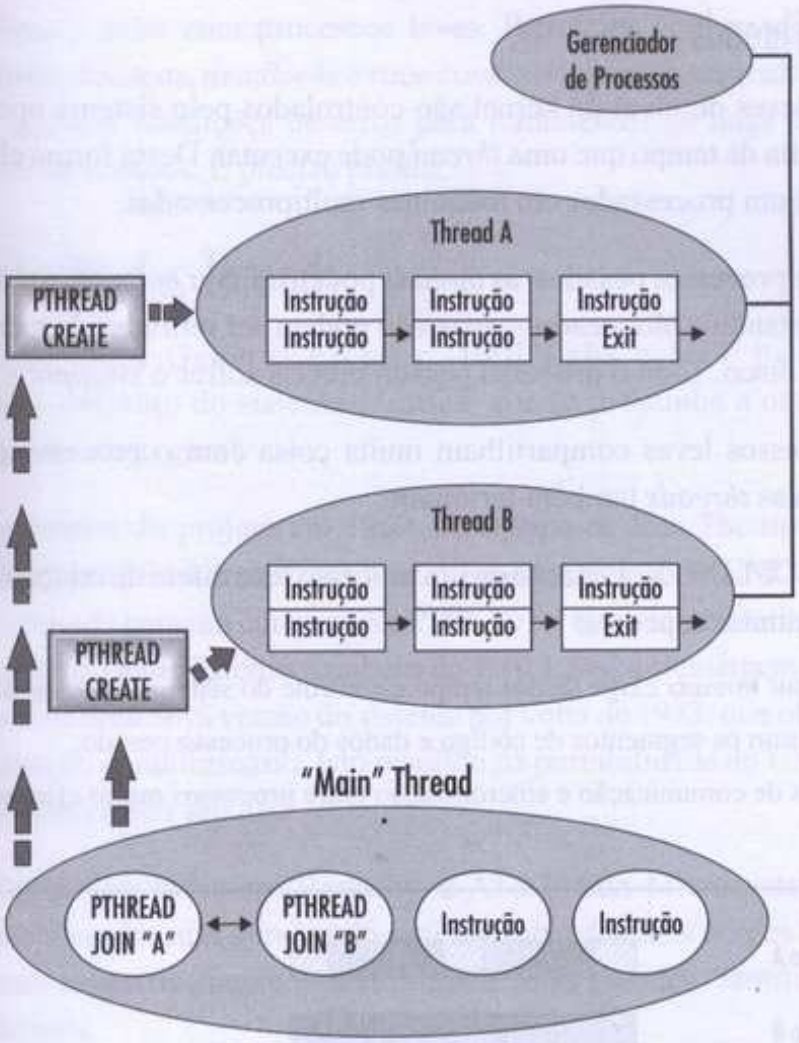


Figura 4.1: Esquema de como funcionam as threads.

Basicamente existem dois tipos de processos leves: nível usuário – *user-level* – e o nível de kernel – *kernel-level*.

User-Level Threads

Os processos leves de nível de usuário são independentes do kernel e são capazes de definir quanto tempo ficarão no controle da CPU. São conhecidos como multitarefa cooperativa. O problema com este tipo é que uma simples *thread* pode monopolizar o processador e não há como obter as vantagens de uma máquina multiprocessada.

Kernel-Level Threads

Os processos leves de nível de kernel são controlados pelo sistema operacional. É ele que define a fatia de tempo que uma *thread* pode executar. Desta forma elas podem fazer uso de mais de um processador em máquinas multiprocessadas.

Assim como os processos pesados, as *threads* podem alterar entre os estados de execução – pronto, executando e bloqueado –, mas não podem ser retiradas da memória principal e gravadas em disco. Todo o processo pesado precisa sofrer o *swapping*.

Como os processos leves compartilham muita coisa com o processo pesado, se este terminar, todas as *threads* também terminam.

No caso do UNIX/LINUX, a criação de um processo leve difere da criação de um processo pesado nos seguintes aspectos:

- Criar e destruir *threads* exige menor tempo e controle do sistema operacional;
- As *threads* usam os segmentos de código e dados do processo pesado;
- As operações de comunicação e sincronização entre processos muito eficazes.

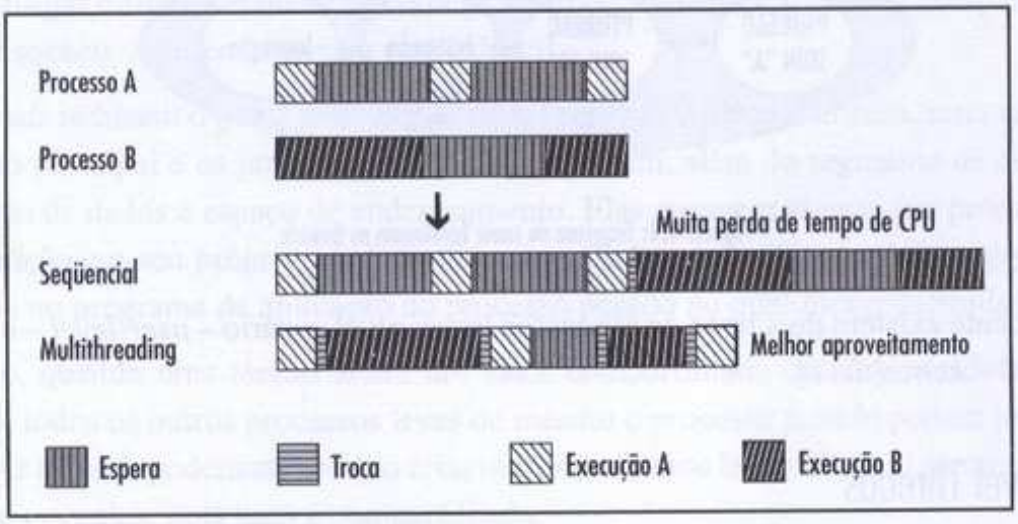


Figura 4.2: Gráfico comparativo do ganho de processamento com as threads.

As *threads* são mais leves, mais rápidas e se comunicam melhor do que os processos pesados. Mas nem tudo são rosas, pois o programador precisa estar consciente da

complexidade que é lidar com processos leves. Pesadelos com processos travados, compartilhamentos malucos, *deadlocks* e *race conditions* podem ser mais comuns. Além disso, testar e duplicar condições de erros para tratamentos de bugs também podem causar mais cabelos brancos. É preciso prática.

A Implementação das Threads

Entre 1965 e 1969, a General Electric, Bell Labs e MIT participaram do desenvolvimento conjunto do sistema Multics, que se propunha a oferecer serviços de time-sharing para uso geral.

O Bell Labs se retirou do projeto em 1969 e a equipe de Ken Thompson começou a trabalhar por conta própria num sistema bem menos ambicioso: o UNIX. Sua primeira versão foi desenvolvida para um computador PDP-7 e foi totalmente escrita em linguagem assembly. Criada por Denis Ritchie, também do Bell Labs, a linguagem C foi usada no desenvolvimento de uma nova versão do sistema por volta de 1973, que oferecia recursos de multiprogramação e multiusuário. Isto resultou na portabilidade do UNIX para outras plataformas, disseminando seu uso.

A Universidade da Califórnia em Berkeley, a AT&T, Sun Microsystems e a própria Microsoft desenvolveram suas versões do sistema sob diferentes nomes – BSD, System V, SunOS e Xenix respectivamente. Daí resultaram duas grandes "famílias" de sistemas UNIX incompatíveis.

Para padronizar as interfaces do UNIX e permitir a portabilidade das aplicações para as várias versões do sistema, o Instituto de Engenheiros Elétricos e Eletrônicos – IEEE – definiu o padrão POSIX – Portable Operating System Interface. Esse padrão define o conjunto de funções que as bibliotecas devem oferecer e permitindo que as aplicações possam ser executadas em qualquer sistema operacional que siga suas recomendações.

A biblioteca que define os processos leves foi criada no padrão POSIX e por isto leva o seu nome *pthreads* – POSIX Threads – e definida em *pthread.h*. Ela permite que a criação de uma *thread* seja feita com a primitiva *pthread_create( )*.

A primitiva *pthread_create()*

A primitiva *pthread_create* permite que uma função do programa seja executada em um processo leve concorrentemente ao processo principal.

```
#include <pthread.h>
int pthread_create(*identificador, *atributo, função, argumentos);
```

Seus argumentos são:

- *identificador*: É um ponteiro para uma variável do tipo *pthread_t* responsável pela identificação da *thread* recém-criada. Ele irá conter no retorno da chamada o identificador do processo leve;
- *atributo*: É um ponteiro para os atributos que são armazenados em uma variável do tipo *pthread_attr_t*. Ele permite que o programador defina alguns atributos especiais como política de escalonamento, seu escopo de execução, dentre outros. Para utilizar o padrão do sistema, deve ser passado NULL;
- *função*: É um ponteiro para a função que será executada pela *thread;*
- *argumentos*: É um void * que é passado como argumento para rotina. Eles contêm o endereço de memória dos dados para a thread criada.

Em C/C++, o conjunto de instruções a ser executado por uma *thread* é definido no corpo de uma função. O novo processo leve será disparado imediatamente após a primitiva *pthread_create()* e termina somente ao fim da função ou pela chamada *pthread_exit(*ret)* que é análoga à função *exit()*.

Na implementação pura do POSIX, as *threads* são rotinas mínimas que contêm reminiscências do processo original – dados, pilha, E/S e sinais – o que significa rapidez, mas também alguns problemas. O que aconteceria se um processo leve executasse um *fork()* ou um *execv()*?

A implementação dos processos leves em Linux tradicionalmente difere um pouco das definições clássicas do POSIX. Toda vez que a função *pthread_create()* é chamada para criar um *thread*, o Linux cria um novo processo para executar aquela determinada função. No entanto, este novo processo não é igual ao processo criado com o *fork()*, uma vez que ele compartilha o mesmo espaço de endereçamento e recursos do processo original.

Desta forma, uma *thread* no Linux é chamada de "contexto de execução", o que significa que os processos pesados e leves utilizem a mesma tabela e o mesmo escalonador, simplificando o trabalho do sistema operacional.

Isto permite que as *threads* no Linux permaneçam rápidas e que o nível de compartilhamento das partes do processo possa ser identificado. Não há problemas com as chamadas *execv( )* e *fork( )*.

Veja esta implementação no exemplo abaixo:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void* funcao_thread(void* arg)
{
        printf("Nova thread criada com o PID %d.\n", getpid());
        sleep(20);
        return NULL;
}
int main ()
{
        pthread_t thread;
        printf("Processo pesado PID %d.\n", getpid());
        pthread_create (&thread, NULL, &funcao_thread, NULL);
        printf("Identificador do thread %d\n",thread);
      sleep(20);
        exit(0);
}
```

Para compilar este programa é necessário incluir a biblioteca *pthread.h* no momento de ligação para fazer a junção da função externa *pthread_create( )* no arquivo objeto.

O comando é:

```
# gcc -lpthread  exemplo_pthread.c -o exemplo_pthread
```

Para observar o comportamento das *threads*, é interessante executar este programa em segundo plano.

```
# ./exemplo_pthread &
[1] 14417
~/sistemas # Processo pesado PID 14417.
Nova thread criada com o PID 14419.
Identificador do thread 16386
```

Enquanto o programa fica em execução durante os 20 segundos de *sleep(20)*, veja os processos em execução com o comando:

```
# ps
PID TTY          TIME     CMD
26119 pts/61     00:00:00 bash
14417 pts/61     00:00:00 pthread    <- Processo pesado
14418 pts/61     00:00:00 pthread    <- Gerenciador de Thread
14419 pts/61     00:00:00 pthread    <- Processo leve
14592 pts/61     00:00:00 ps
```

Neste exemplo existem três processos em execução. O primeiro, PID 14417, é o processo original pesado definido na função *main( )*.

O segundo processo é o gerenciador de *thread*. Ele é criado toda vez que um processo pesado cria um processo leve e somente um é necessário, indiferentemente do número de *threads* criadas. O gerenciador de *thread* é responsável por criar e coordenar todos os processos leves criados por um processo pesado. Este modelo simplifica um pouco as coisas para o Kernel, mas tem como desvantagem o aumento do processamento necessário para criar e destruir *threads*, e oferece algumas limitações quanto ao número de processos leves que o sistema pode lidar.

O terceiro, PID 14419, é a *thread* em execução definida pela função *funcao_thread( )*. Cada *thread* tem o seu próprio *Process ID*.

A Primitiva *pthread_join( )*

A primitiva *pthread_join( )* suspende o processamento da *thread* que a chamou até que a *thread* identificada na função termine normalmente através da função *pthread_exit()* ou seja cancelada.

DICA

A função pthread_join() é análoga à waitpid().

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

Esta chamada de sistema recebe como parâmetro a identificação da *thread* que vai ser esperada e o valor de retorno desta. Se o parâmetro *thread_return* for NULL, o valor de retorno é descartado. Caso contrário, o valor de retorno será um dos argumentos da função *pthread_exit( )*.

Quando uma *thread* comum termina, os recursos de memória utilizados – descritores e pilha – não são liberados até que todas as *threads* em execução do processo sofram a junção feita pelo *pthread_join( )*. Desta forma, esta primitiva precisa ser executada para cada processo leve criado para evitar vazamento de memória.

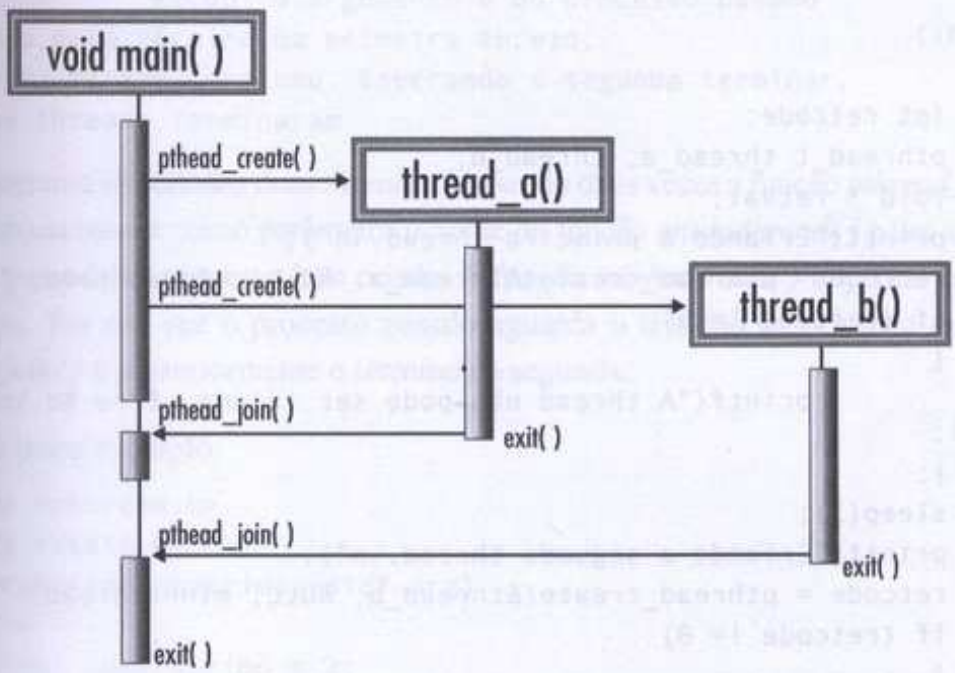


Figura 4.3: Esquema pthread_create() e pthread_join().

A *thread* que vai ser esperada não pode estar desconectada do processo pesado pela função *pthread_detach( )*. Este recurso é empregado para garantir que a memória utilizada por uma *thread* seja liberada imediatamente após o seu término. Infelizmente a desconexão impede o sincronismo entre os processos leves. A função *pthread_create( )* permite que um processo leve seja criado já desconectado com o atributo *PTHREAD_CREATE_DETACHED*.

Se não houver necessidade de sincronização, deve-se desconectar uma *thread* com a função *pthread_detach( )* para liberar os recursos imediatamente.

Veja o exemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void * minhathread(void * arg)
{
int i;
        printf("\t\tIniciando a thread.\n", arg);
        printf("\t\tRecebi o argumento %s do processo pesado\n", arg);
        sleep(3);
}
int main()
{
        int retcode;
        pthread_t thread_a, thread_b;
        void * retval;
        printf("Criando a primeira thread.\n");
        retcode = pthread_create(&thread_a, NULL, minhathread, "a");
        if (retcode != 0)
        {
                printf("A thread não pode ser criada. Erro %d.\n",
retcode);
        };
        sleep(1);
        printf("Criando a segunda thread.\n");
        retcode = pthread_create(&thread_b, NULL, minhathread, "b");
        if (retcode != 0)
        {
                printf("A thread não pode ser criada. Erro %d.\n",
retcode);
        };
        sleep(1);
        printf("Esperando pelo término da primeira thread.\n");
        retcode = pthread_join(thread_a, &retval);
        printf("A primeira thread terminou. Esperando a segunda
terminar.\n");
        retcode = pthread_join(thread_b, &retval);
        printf("Ambas as threads terminaram\n");
        exit(0);
}
```

Para compilar este programa salvo como "*pthread_join.c*":

```
# gcc -lpthread pthread_join.c -o pthread_join
```

O resultado da execução será:

```
# ./pthread_join
Criando a primeira thread.
                Iniciando a thread.
                Recebi o argumento a do processo pesado
Criando a segunda thread.
                Iniciando a thread.
                Recebi o argumento b do processo pesado
Esperando pelo término da primeira thread.
A primeira thread terminou. Esperando a segunda terminar.
Ambas as threads terminaram
```

Neste programa são criadas duas *threads* chamando duas vezes a função *pthread_create()*. Esta chamada recebe como parâmetro o nome da função *minhathread()* e um argumento que será passado ao processo leve criado. A função *minhathread()* imprime na tela este argumento. Por sua vez o processo pesado aguarda o término da primeira *thread* com *pthread_join()* e posteriormente o término da segunda.

Veja este outro exemplo:

```
#include <pthread.h>
#include <stdio.h>
void* procuranumeroprimo(void* arg)
{
        int numeroprimo = 2;
        int totalnumeros = *((int*) arg);
        printf("Vou calcular os números primos até chegar no
%do.\n\n",totalnumeros);
        printf("São primos: ");
        sleep(1);
        while (1) {
                int fatordivisao;
                int flagnumeroprimo = 1;
                for (fatordivisao = 2; fatordivisao < numeroprimo;
++fatordivisao)
              if (numeroprimo % fatordivisao == 0) /* se não for primo */
                {
```

```
                        flagnumeroprimo = 0;
                        break;
                }
                if (flagnumeroprimo) /* É primo, pois só divide por 1
e por ele mesmo. */
                {
                        printf(" %d ",numeroprimo);
                        if (--totalnumeros == 0)
                        return (void*) numeroprimo;
                }
                ++numeroprimo;
        }
        return NULL;
}
int main ()
{
pthread_t thread;
int total = 10;
int primo;
pthread_create (&thread, NULL, &procuranumeroprimo, &total);
pthread_join (thread, (void*) &primo);
printf("\n\nO %do número primo é %d.\n", total, primo);
return 0;
}
```

Para compilar este programa salvo como "*numero_primo.c*":

```
# gcc -lpthread numero_primo.c -o numero_primo
```

O resultado da execução é:

```
# ./numero_primo
```

Vou calcular os números primos até chegar no 10º.

São primos: 2 3 5 7 11 13 17 19 23 29

O 10º número primo é 29.

"Nunca desista de alguém. Milagres acontecem todos os dias."

S. Brown, escritor norte-americano

Neste exemplo o programa pesado dispara uma *thread* com o parâmetro de até qual número primo deve ser calculado pela função *procuranumeroprimo( )*. Esta função faz a divisão dos números até encontrar o décimo número primo.

Veja agora o programa a seguir que demonstra o uso destas primitivas para a implementação de *threads* concorrentes.

```
#include <pthread.h>
#include <stdio.h>
void * ProcessoLeve1()
{
    int i;
    for(i=1;i<100;i++)
        printf("\tThread 1 - %d\n",i);
}
void * ProcessoLeve2()
{
    int i;
    for(i=100;i<200;i++)
        printf("\t\tThread 2 - %d\n",i);
}
main(void)
{
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, ProcessoLeve1,NULL) ;
    pthread_create(&thread2, NULL, ProcessoLeve2,NULL) ;
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    printf("FIM\n");
}
```

Para compilar este programa salvo como "*concorrentes.c*":

```
# gcc -lpthread concorrentes.c -o concorrentes
```

O resultado da execução é:

```
# ./concorrentes
        Thread 1 - 1
        Thread 1 - 2
        Thread 1 - 3
       (...)
        Thread 1 - 98
        Thread 1 - 99
                Thread 2 - 100
```

```
            Thread 2 - 101
      (...)
            Thread 2 - 198
            Thread 2 - 199
FIM
```

Neste exemplo duas novas *threads* são criadas. A primeira irá contar de 0 a 99 e a segunda irá contar de 100 a 199. A *thread main* cria uma nova *thread* ProcessoLeve1 com a chamada da primitiva *pthread_create( )*. A seguir a *thread main* cria a *thread* ProcessoLeve2 e se bloqueia com a primitiva *pthread_join(ProcessoLeve1,NULL)*. Esta primitiva fica à espera de que ProcessoLeve1 termine. Após o término, a *thread main* se bloqueia novamente com a operação *pthread_join(ProcessoLeve2,NULL)* até que esta acabe.

Neste exemplo, o uso da primitiva *pthread_join( )* é indispensável, pois impede que o processo pesado – *thread main* – continue a execução e termine. Se isso acontecesse, os processos leves seriam eliminados antes da hora.

Se comentarmos as linhas *pthread_join( )* do programa, o resultado seria o seguinte:

```
# ./concorrente
      Thread 1 - 1
      Thread 1 - 2
FIM
```

Uma maneira de contornar isso é desconectar as *threads ProcessoLeve1* e *ProcessoLeve2* incluindo as linhas abaixo antes do final da função *main( )*:

```
pthread_detach(thread1);
pthread_detach(thread2);
```

Uma vez desconectadas, as *threads* não dependem mais do processo pesado e não precisam da função *pthread_join( )*. O resultado voltará a ser o esperado.

Resumidamente, as principais funções relacionadas com a gestão de processos leves são:

- Criação de um processo leve: *pthread_create( )*
- Término de um processo leve: *pthread_exit( )*
- Espera por um processo leve: *pthread_join( )*
- Desconexão de um processo leve: *pthread_detach( )*

O Next Generation POSIX Threads

A IBM introduziu um novo modelo de processos leves para o Linux chamado Next Generation POSIX Threads – NGPT. Esta nova implementação oferece mais conformidade com o modelo clássico do POSIX e melhor performance.

Este novo software introduzido a partir do kernel 2.5 foi a base necessária para a construção de aplicações corporativas como banco de dados, servidores de web e e-mail de alta capacidade de processamento.

Para que as aplicações possam fazer uso das novas possibilidades do NPTL é necessário estar executando o kernel 2.6. Algumas talvez necessitem ser recompiladas.

Mas, nem sempre migrar para o kernel 2.6 significa utilizar o NTPL. Ainda é necessário conferir a versão da biblioteca de *threads* utilizada pelo sistema. O comando *getconf* disponível no pacote *glibc* pode determinar isso examinando a variável *GNU_LIBPTHREAD_VERSION*:

```
# getconf GNU_LIBPTHREAD_VERSION
linuxthreads-0.10
```

No exemplo acima o sistema está usando a implementação normal de *threads* da biblioteca *linuxthreads*. Se o sistema estiver usando o NPTL, o comando irá retornar a versão da biblioteca *nptl*:

```
# getconf GNU_LIBPTHREAD_VERSION
nptl-0.60
```

Para habilitar o NPTL é preciso compilar a versão mais recente das bibliotecas C *glibc* no kernel 2.6 com a versão mais recente do GCC – 3.3.2 para Intel e 3.4 para PPC, na data de fechamento deste livro – e o pacote *binutils* que suporte a diretiva Call Frame Information – versão 2.14.90.0.7 ou mais nova.

A opção de configuração *—enable-add-ons=nptl* permite que o NPTL seja utilizado.

Características do NPTL

As mudanças realizadas pelo NPTL melhoraram muito o suporte a *threads* do Linux. tais como:

- Suporte MUTEX entre as threads
- Simplifica o compartilhamento de recursos
- Prevenção de conflitos
- Melhor tratamento de sinais entre processos/thread e threads/threads
- Aumento do paralelismo
- Performance melhor

Veja o gráfico comparativo do tempo de criação de *threads* entre as implementações LinuxThreads e NPTL:

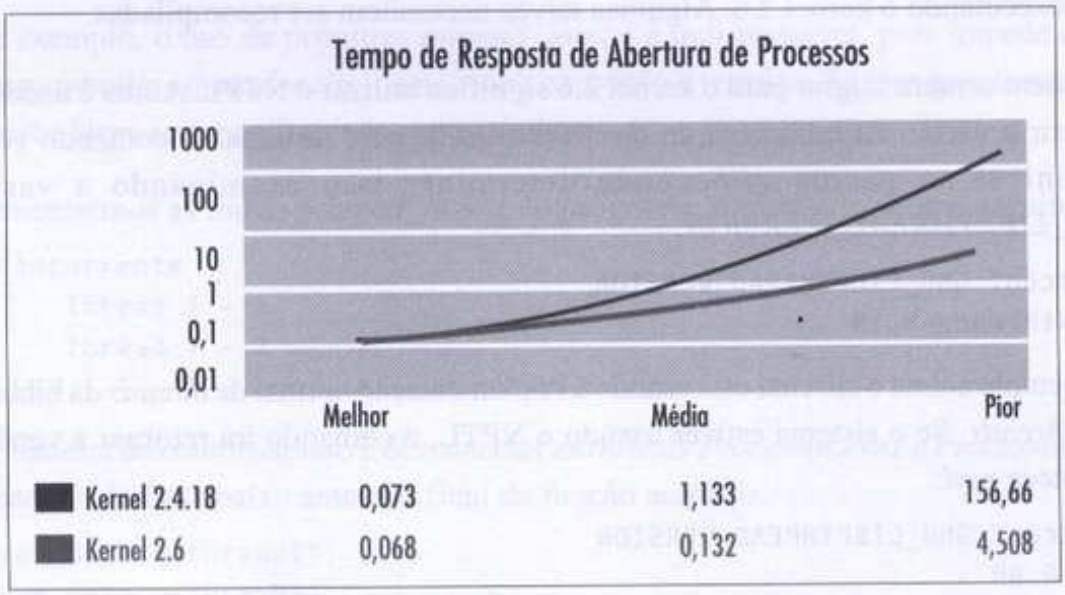


Figura 4.4: Gráfico comparativo Linuxthreads e NPTL.

Fonte: RedHat.

As Alterações de Código Requeridas Pelo NPTL

O NPTL pode requerer mudanças no código-fonte das aplicações no tocante à identificação das *threads*. Na implementação antiga, cada processo leve tem o seu próprio PID. No novo modelo, cada *thread* compartilha o PID do processo pesado que a criou.

Desta forma, a função *getpid( )* executada por uma *thread* retorna o PID do processo pesado e os processos leves precisam ser identificados somente pelo *Thread ID*.

Alterar o kernel do sistema operacional não é necessariamente uma tarefa difícil, e portanto possível de ser feita, mas foge do escopo deste livro. O estudo de *threads* e as chamadas de sistema associadas funcionam para ambas as implementações de processos leves.

Hyperthreading

Os novos processadores com a tecnologia HyperThreading – HT – desenvolvida pela Intel consagram a programação paralela e uso dos *threads*. Segundo a fabricante, o HT oferece um aumento de desempenho de até 30% dependendo da configuração do sistema.

A tecnologia HyperThreading simula em um único processador físico dois processadores lógicos. Cada processador lógico recebe seu próprio controlador de interrupção programável e conjunto de registradores. O cachê de memória, a unidade de execução, a unidade lógica e aritmética, a unidade de ponto flutuante e o barramento são compartilhados pelos processadores lógicos.

Isso permite que o sistema operacional envie processos para os processadores lógicos como se estivesse enviando para dois processadores físicos, tal como em um sistema multiprocessado real.

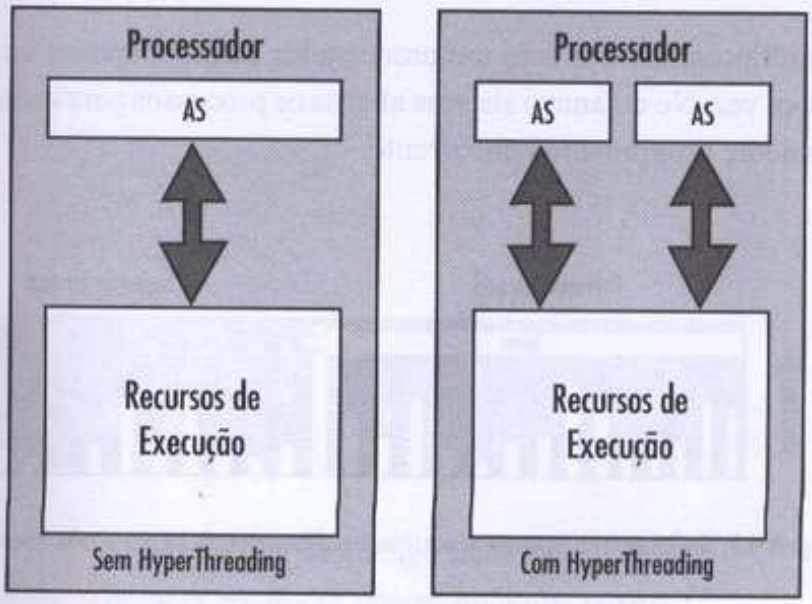


Figura 4.5: Diagrama dos processadores comum e com HT.

Neste diagrama os registradores e o controlador de interrupção são representados pela sigla "AS". Na área denominada de "recursos de execução" estão todos os recursos de que o processador necessita para executar as instruções.

O processador da esquerda não suporta a tecnologia HyperThreading, enquanto o processador da direita suporta, duplicando os registradores e controladores dos processos.

Um processador comum é capaz de processar apenas uma thread em um dado tempo. Nos sistemas operacionais monoprogramados este processo fica em execução até que termine e outro possa ser executado. O resultado pode ser representado no seguinte gráfico de linha de tempo.

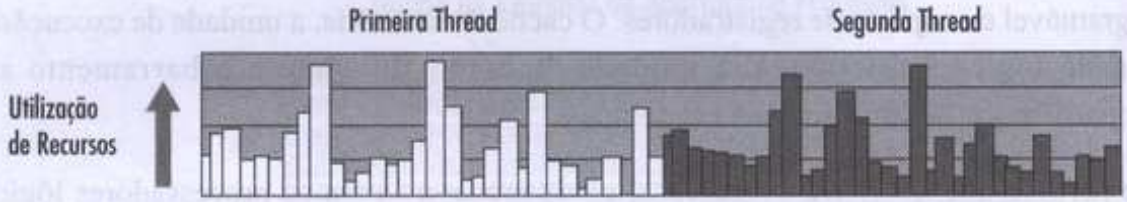


Figura 4.6: Gráfico de linha de tempo de um sistema monoprogramado em processador comum.

Como observado, os recursos computacionais não são utilizados ao máximo.

Nos sistemas multiprogramados com um processador comum, apenas uma *thread* pode ser processada por vez. No entanto o sistema alterna os processos para otimizar as tarefas, criando um ambiente programado concorrente.

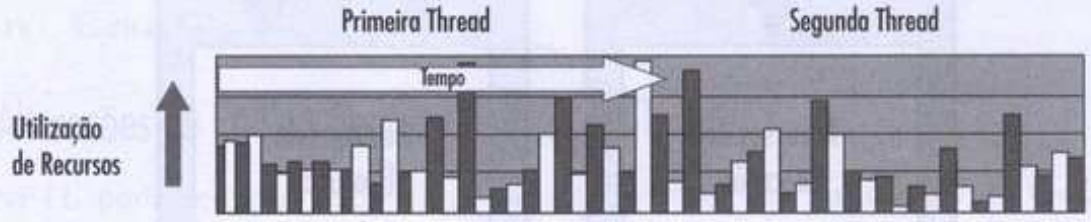


Figura 4.7: Gráfico de linha de tempo de um sistema multiprogramado em processador comum.

Neste modelo os processos são concorrentes, pois concorrem pelo uso da CPU. Ainda assim os recursos computacionais não são utilizados ao máximo.

Com a tecnologia HyperThreading, o processador poderá processar duas *threads* simultaneamente. Este recurso somente será utilizado em sua totalidade se o sistema e as aplicações executadas fizerem uso intensivo dos *threads*. O resultado será um ganho de tempo de processamento. Observe o gráfico:

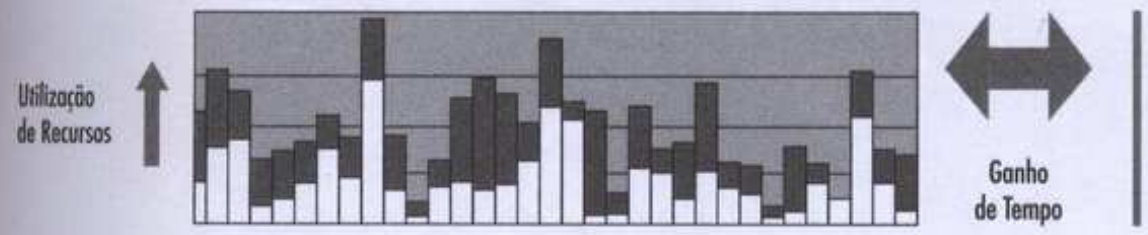


Figura 4.8: Gráfico de linha de tempo de um sistema multiprogramado em processador Hyperthreading.

"O homem de bem exige tudo de si próprio;
o homem mediocre espera tudo dos outros."

Confúcio