# Capítulo 5

# Implementação de tarefas

Como visto no capítulo anterior, uma tarefa é a unidade básica de atividade dentro de um sistema operacional. Tarefas podem ser implementadas de várias formas, como processos, *threads*, transações e *jobs*. Neste capítulo são descritos os problemas relacionados à implementação do conceito de tarefa em um sistema operacional típico. São descritas as estruturas de dados necessárias para representar uma tarefa e as operações necessárias para que o processador possa comutar de uma tarefa para outra de forma transparente e eficiente.

## 5.1 Contextos

Na Seção 4.2 vimos que uma tarefa possui um estado interno bem definido, que representa sua situação atual: a posição de código que ela está executando, os valores de suas variáveis e os recursos que ela utiliza, por exemplo. Esse estado se modifica conforme a execução da tarefa evolui. O estado de uma tarefa em um determinado instante é denominado **contexto**. Uma parte importante do contexto de uma tarefa diz respeito ao estado interno do processador durante sua execução, como o valor do contador de programa (PC - *Program Counter*), do apontador de pilha (SP - *Stack Pointer*) e demais registradores. Além do estado interno do processador, o contexto de uma tarefa também inclui informações sobre os recursos usados por ela, como arquivos abertos, conexões de rede e semáforos.

A cada tarefa presente no sistema é associado um *descritor*, ou seja, uma estrutura de dados no núcleo que representa essa tarefa. Nessa estrutura de dados são armazenadas as informações relativas ao seu contexto e os demais dados necessários à sua gerência, como prioridades, estado, etc. Essa estrutura de dados é geralmente chamada de TCB (do inglês *Task Control Block*) ou PCB (*Process Control Block*). Um TCB tipicamente contém as seguintes informações:

- identificador da tarefa (pode ser um número inteiro, um apontador, uma referência de objeto ou um algum outro identificador);
- estado da tarefa (nova, pronta, executando, suspensa, terminada, etc.);
- informações de contexto do processador (valores contidos nos registradores);
- lista de áreas de memória usadas pela tarefa;

- listas de arquivos abertos, conexões de rede e outros recursos usados pela tarefa (exclusivos ou compartilhados com outras tarefas);
- informações de gerência e contabilização (prioridade, usuário proprietário, data de início, tempo de processamento já decorrido, volume de dados lidos/escritos, etc.).

Dentro do núcleo, os descritores das tarefas são geralmente organizados em listas ou vetores de TCBs. Por exemplo, normalmente há uma lista de tarefas prontas para executar, uma lista de tarefas aguardando acesso ao disco rígido, etc. Para ilustrar concretamente o conceito de TCB, o Apêndice A apresenta o TCB do núcleo Linux em sua versão 1.0.

## 5.2 Trocas de contexto

Para que o sistema operacional possa suspender e retomar a execução de tarefas de forma transparente (sem que as tarefas o percebam), é necessário definir operações para salvar o contexto atual de uma tarefa em seu TCB e restaurá-lo mais tarde no processador. Por essa razão, o ato de suspender uma tarefa e reativar outra é denominado uma **troca de contexto**.

A implementação da troca de contexto é uma operação delicada, envolvendo a manipulação de registradores e *flags* específicos de cada processador, sendo por essa razão geralmente codificada em linguagem de máquina. No Linux, as operações de troca de contexto para processadores Intel de 64 bits estão definidas nos arquivos `arch/x86/kernel/process_64.c` e `arch/x86/entry/entry_64.S` dos fontes do núcleo (versão 4.*).

Durante uma troca de contexto, existem questões de ordem mecânica e de ordem estratégica a serem resolvidas, o que traz à tona a separação entre mecanismos e políticas já discutida anteriormente (vide Seção 1.2). Por exemplo, o armazenamento e recuperação do contexto e a atualização das informações contidas no TCB de cada tarefa são aspectos mecânicos, providos por um conjunto de rotinas denominado **despachante** ou **executivo** (do inglês *dispatcher*). Por outro lado, a escolha da próxima tarefa a receber o processador a cada troca de contexto é estratégica, podendo sofrer influências de diversos fatores, como as prioridades, os tempos de vida e os tempos de processamento restante de cada tarefa. Essa decisão fica a cargo de um componente de código denominado **escalonador** (*scheduler*, vide Seção 6). Assim, o despachante implementa os mecanismos da gerência de tarefas, enquanto o escalonador implementa suas políticas.

A Figura 5.1 apresenta um diagrama temporal com os principais passos envolvidos em uma troca de contexto:

1. uma tarefa *A* está executando;
2. ocorre uma interrupção do temporizador do hardware e a execução desvia para a rotina de tratamento, no núcleo;
3. a rotina de tratamento ativa o despachante;
4. o despachante salva o estado da tarefa *A* em seu TCB e atualiza suas informações de gerência;

5. opcionalmente, o despachante consulta o escalonador para escolher a próxima tarefa a ativar ($B$);

6. o despachante resgata o estado da tarefa $B$ de seu TCB e a reativa.

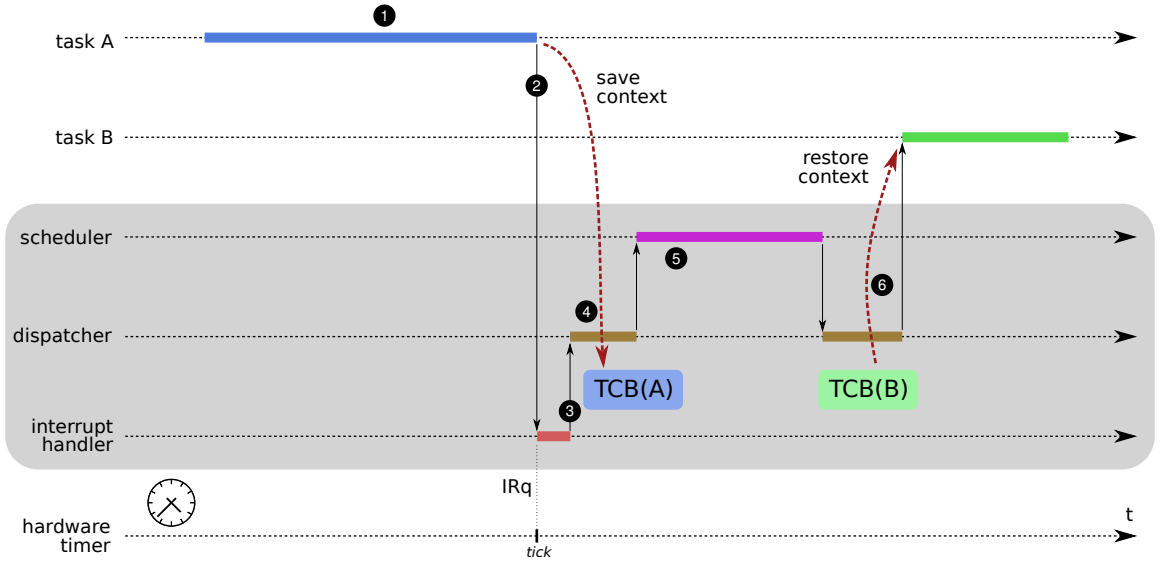


Figura 5.1: Passos de uma troca de contexto.

É importante observar que uma troca de contexto pode ser provocada pelo fim do *quantum* atual (através de uma interrupção de tempo), por um evento ocorrido em um periférico (uma interrupção do respectivo controlador), por uma chamada de sistema emitida pela tarefa corrente (uma interrupção de software) ou até mesmo por algum erro de execução que possa provocar uma exceção no processador.

A frequência de trocas de contexto tem impacto na eficiência do sistema operacional: quanto menor o número de trocas de contexto e menor a duração de cada troca, mais tempo sobrará para a execução das tarefas em si. Assim, é possível definir uma **medida de eficiência** $\mathcal{E}$ do uso do processador, em função das durações médias do *quantum* de tempo $t_q$ e da troca de contexto $t_{tc}$:

$$\mathcal{E} = \frac{t_q}{t_q + t_{tc}}$$

Por exemplo, um sistema no qual as trocas de contexto duram $100\mu s$ e cujo *quantum* médio é de $10ms$ terá uma eficiência $\mathcal{E} = \frac{10ms}{10ms+100\mu s} = 99\%$. Caso a duração do quantum seja reduzida para $1ms$, a eficiência cairá para $\mathcal{E} = \frac{1ms}{1ms+100\mu s} = 91\%$. A eficiência final da gerência de tarefas é influenciada por vários fatores, como a carga do sistema (mais tarefas ativas implicam em mais tempo gasto pelo escalonador, aumentando $t_{tc}$) e o perfil das aplicações (aplicações que fazem muita entrada/saída saem do processador antes do final de seu *quantum*, diminuindo o valor médio de $t_q$).

Nos sistemas atuais, a realização de uma troca de contexto, envolvendo a interrupção da tarefa atual, o salvamento de seu contexto e a reativação da próxima tarefa, é uma operação relativamente rápida (alguns microssegundos, dependendo do hardware e do sistema operacional). A execução do escalonador, entretanto, pode ser bem mais demorada, sobretudo se houverem muitas tarefas prontas para executar. Por

esta razão, muitos sistemas operacionais não executam o escalonador a cada troca de contexto, mas apenas periodicamente, quando há necessidade de reordenar a fila de tarefas prontas. Nesse caso, o despachante sempre ativa a primeira tarefa dessa fila.

## 5.3 Processos

Há diversas formas de implementar o conceito de tarefa. Uma forma muito empregada pelos sistemas operacionais é o uso de **processos**, apresentado nesta seção.

### 5.3.1 O conceito de processo

Historicamente, um processo era definido como sendo uma tarefa com seus respectivos recursos, como arquivos abertos e canais de comunicação, em uma área de memória delimitada e isolada das demais. Ou seja, um processo seria uma espécie de "cápsula" isolada de execução, contendo uma tarefa e seus recursos. Essa visão é mantida por muitos autores, como [Silberschatz et al., 2001] e [Tanenbaum, 2003], que apresentam processos como equivalentes a tarefas.

De fato, os sistemas operacionais mais antigos, até meados dos anos 80, suportavam somente um fluxo de execução em cada processo. Assim, as unidades de execução (tarefa) e de recursos (processo) se confundiam. No entanto, quase todos os sistemas operacionais atuais suportam e existência de mais de uma tarefa em cada processo, como é o caso do Linux, Windows, iOS e os sistemas UNIX mais recentes.

Hoje em dia o processo deve ser visto como uma *unidade de contexto*, ou seja, um contêiner de recursos utilizados por uma ou mais tarefas para sua execução: áreas de memória (código, dados, pilha), informações de contexto e descritores de recursos do núcleo (arquivos abertos, conexões de rede, etc). Um processo pode então conter várias tarefas, que compartilham esses recursos. Os processos são isolados entre si pelos mecanismos de proteção providos pelo hardware (isolamento de áreas de memória, níveis de operação e chamadas de sistema), impedindo que uma tarefa do processo $p_a$ acesse um recurso atribuído ao processo $p_b$. A Figura 5.2 ilustra o conceito de processo, visto como um contêiner de recursos.

Os sistemas operacionais atuais geralmente associam por *default* uma única tarefa a cada processo, o que corresponde à execução de um programa sequencial (iniciado pela função `main()` de um programa em C, por exemplo). Caso se deseje associar mais tarefas ao mesmo processo (para construir o navegador Internet da Figura 4.1, por exemplo), cabe ao desenvolvedor escrever o código necessário para solicitar ao núcleo a criação dessas tarefas adicionais, usualmente sob a forma de *threads* (apresentadas na Seção 5.4).

O núcleo do sistema operacional mantém descritores de processos, denominados PCBs (*Process Control Blocks*), para armazenar as informações referentes aos processos ativos. Um PCB contém informações como o identificador do processo (PID – *Process IDentifier*), seu usuário, prioridade, data de início, caminho do arquivo contendo o código executado pelo processo, áreas de memória em uso, arquivos abertos, etc. A listagem a seguir mostra um conjunto de processos no sistema Linux.

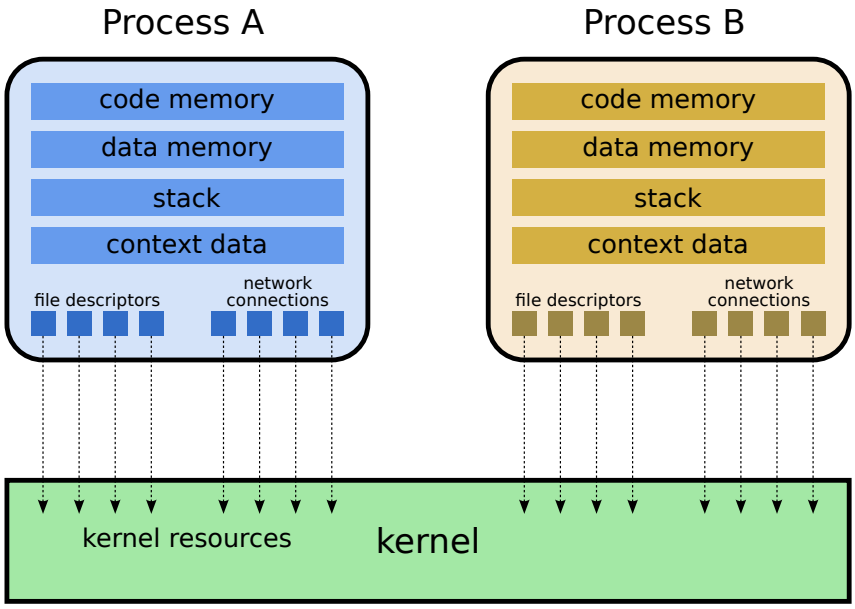


Figura 5.2: O processo visto como um contêiner de recursos.

```
top - 16:58:06 up  8:26,  1 user,  load average: 6,04, 2,36, 1,08
Tarefas: 218 total,   7 executando, 211 dormindo,   0 parado,   0 zumbi
%Cpu(s): 49,7 us, 47,0 sy,  0,0 ni,  3,2 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem : 16095364 total,  9856576 free,  3134380 used,  3104408 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11858380 avail Mem

  PID USUÁRIO   PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32703 maziero   20   0 2132220 432628 139312 S  44,8  2,7   0:53.64 Web Content
 2192 maziero   20   0 9617080 686444 248996 S  29,8  4,3  20:01.81 firefox
11650 maziero   20   0 2003888 327036 129164 R  24,0  2,0   1:16.70 Web Content
 9844 maziero   20   0 2130164 442520 149508 R  17,9  2,7   1:29.18 Web Content
11884 maziero   20   0   25276   7692   3300 S  15,5  0,0   0:37.18 bash
20425 maziero   20   0   24808   7144   3212 S  14,4  0,0   0:08.39 bash
 1782 maziero   20   0 1788328 235200  77268 S   8,7  1,5  24:12.75 gnome-shell
...
```

Associando-se tarefas a processos, o descritor (TCB) de cada tarefa pode ser bastante simplificado: para cada tarefa, basta armazenar seu identificador, os registradores do processador e uma referência ao processo onde a tarefa executa. Observa-se também que a troca de contexto entre duas tarefas dentro do mesmo processo é muito mais simples e rápida que entre tarefas em processos distintos, pois somente os registradores do processador precisam ser salvos/restaurados (pois as áreas de memória e demais recursos são comuns a ambas). Essas questões são aprofundadas na Seção 5.4.

### 5.3.2 Gestão de processos

Durante a vida do sistema, processos são criados e destruídos. Essas operações são disponibilizadas às aplicações através de chamadas de sistema; cada sistema operacional tem suas próprias chamadas para a gestão de processos. O quadro 5.1 traz exemplos de algumas chamadas de sistema usadas na gestão de processos.

| Ação | Windows | Linux |
|---|---|---|
| Criar um novo processo | `CreateProcess()` | `fork()`, `execve()` |
| Encerrar o processo corrente | `ExitProcess()` | `exit()` |
| Encerrar outro processo | `TerminateProcess()` | `kill()` |
| Obter o ID do processo corrente | `GetCurrentProcessId()` | `getpid()` |

Tabela 5.1: Chamadas de sistema para a gestão de processos.

No caso dos sistemas UNIX, a criação de novos processos é feita em duas etapas: na primeira etapa, um processo cria uma réplica de si mesmo, usando a chamada de sistema `fork()`. Todo o espaço de memória do processo inicial (pai) é copiado para o novo processo (filho), incluindo o código da(s) tarefa(s) associada(s) e os descritores dos arquivos e demais recursos associados ao mesmo. A Figura 5.3 ilustra o funcionamento dessa chamada.

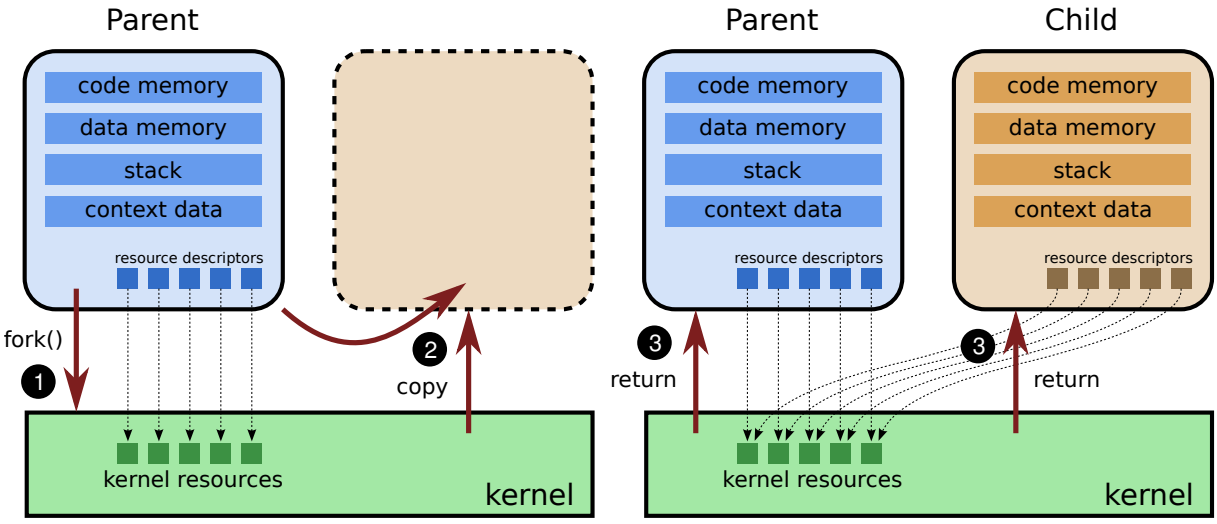


Figura 5.3: Execução da chamada de sistema `fork()`.

A chamada de sistema `fork()` é invocada por um processo, mas dois processos recebem seu retorno: o *processo pai*, que a invocou, e o *processo filho*, que acabou de ser criado e que possui o mesmo estado interno que o pai (ele também está aguardando o retorno da chamada de sistema). Ambos os processos acessam os mesmos recursos do núcleo, embora executem em áreas de memória distintas.

Na segunda etapa, o processo filho usa a chamada de sistema `execve()` para carregar um novo código binário em sua memória. Essa chamada substitui o código do processo que a invoca pelo código executável contido em um arquivo informado como parâmetro. A listagem a seguir apresenta um exemplo de uso conjunto dessas duas chamadas de sistema:

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[], char *envp[])
{
   int pid ;                        // identificador de processo

   pid = fork () ;                  // replicação do processo

   if ( pid < 0 )                   // fork funcionou?
   {
      perror ("Erro: ") ;           // não, encerra este processo
      exit (-1) ;
   }
   else                             // sim, fork funcionou
      if ( pid > 0 )                // sou o processo pai?
         wait (0) ;                 // sim, vou esperar meu filho concluir
      else                          // não, sou o processo filho
      {
         // carrega outro código binário para executar
         execve ("/bin/date", argv, envp) ;
         perror ("Erro: ") ;        // execve não funcionou
      }
   printf ("Tchau !\n") ;
   exit(0) ;                        // encerra este processo
}
```

A chamada de sistema `exit()` usada no exemplo acima serve para informar ao núcleo do sistema operacional que o processo em questão não é mais necessário e pode ser destruído, liberando todos os recursos a ele associados (arquivos abertos, conexões de rede, áreas de memória, etc.). Processos podem solicitar ao núcleo o encerramento de outros processos, mas essa operação só é aplicável a processos do mesmo usuário, ou se o processo solicitante pertencer ao administrador do sistema.

Outro aspecto importante a ser considerado em relação a processos diz respeito à comunicação. Tarefas associadas ao mesmo processo podem trocar informações facilmente, pois compartilham as mesmas áreas de memória. Todavia, isso não é possível entre tarefas associadas a processos distintos. Para resolver esse problema, o núcleo deve prover às aplicações chamadas de sistema que permitam a comunicação interprocessos (IPC – *Inter-Process Communication*). Esse tema será estudado em profundidade no Capítulo 8.

### 5.3.3 Hierarquia de processos

Na operação de criação de processos do UNIX aparece de maneira bastante clara a noção de **hierarquia** entre processos: processos são filhos de outros processos. À medida em que processos são criados, forma-se uma *árvore de processos* no sistema, que pode ser usada para gerenciar de forma coletiva os processos ligados à mesma aplicação ou à mesma sessão de trabalho de um usuário, pois irão constituir uma sub-árvore de processos. Por exemplo, quando um processo encerra, seus filhos são informados sobre isso, e podem decidir se também encerram ou se continuam a executar.

Por outro, nos sistemas Windows, todos os processos têm o mesmo nível hierárquico, não havendo distinção entre pais e filhos. O comando `pstree` do sistema Linux permite visualizar a árvore de processos do sistema, como mostra a listagem de exemplo a seguir[1].

```
init-+-cron
     |-dhcpcd
     |-getty
     |-getty
     |-ifplugd
     |-ifplugd
     |-lighttpd---php-cgi-+-php-cgi
     |                    `-php-cgi
     |-logsave
     |-logsave
     |-ntpd
     |-openvpn
     |-p910nd
     |-rsyslogd-+-{rsyslogd}
     |          `-{rsyslogd}
     |-sshd---sshd---sshd---pstree
     |-thd
     `-udevd-+-udevd
             `-udevd
```

## 5.4 Threads

Conforme visto na Seção 5.3, os primeiros sistemas operacionais suportavam uma única tarefa por processo. À medida em que as aplicações se tornavam mais complexas, essa limitação se tornou bem inconveniente. Por exemplo, um editor de textos geralmente executa tarefas simultâneas de edição, formatação, paginação e verificação ortográfica sobre os mesmos dados (o texto em edição). Da mesma forma, processos que implementam servidores de rede (de arquivos, bancos de dados, etc.) devem gerenciar as conexões de vários usuários simultaneamente, que muitas vezes requisitam as mesmas informações. Essas demandas evidenciaram a necessidade de suportar mais de uma tarefa operando sobre os mesmos recursos, ou seja, dentro do mesmo processo.

### 5.4.1 Definição de *thread*

Uma *thread* é definida como sendo um fluxo de execução independente. Um processo pode conter uma ou mais *threads*, cada uma executando seu próprio código e compartilhando recursos com as demais *threads* localizadas no mesmo processo[2]. Cada *thread* é caracterizada por um código em execução e um pequeno contexto local, o chamado *Thread Local Storage* (TLS), composto pelos registradores do processador e uma área de pilha em memória, para que a *thread* possa armazenar variáveis locais e efetuar chamadas de funções.

[1] Listagem obtida de um sistema *Raspberry Pi* com SO Linux *Raspbian* 8.

[2] Alguns autores usam também o termo *processo leve* (*lightweigth process*) como sinônimo de *thread*.

*Threads* são também utilizadas para implementar fluxos de execução dentro do núcleo do SO, neste caso recebendo o nome de *threads de núcleo* (em oposição às *threads* dos processos, denominadas *user threads*). Além de representar as *threads* de usuário dentro do núcleo, as *threads* de núcleo também incluem atividades internas do núcleo, como rotinas de *drivers* de dispositivos ou tarefas de gerência. A Figura 5.4 ilustra o conceito de *threads* de usuário e de núcleo. Na figura, o processo A tem várias *threads*, enquanto o processo B é sequencial (tem uma única *thread*).

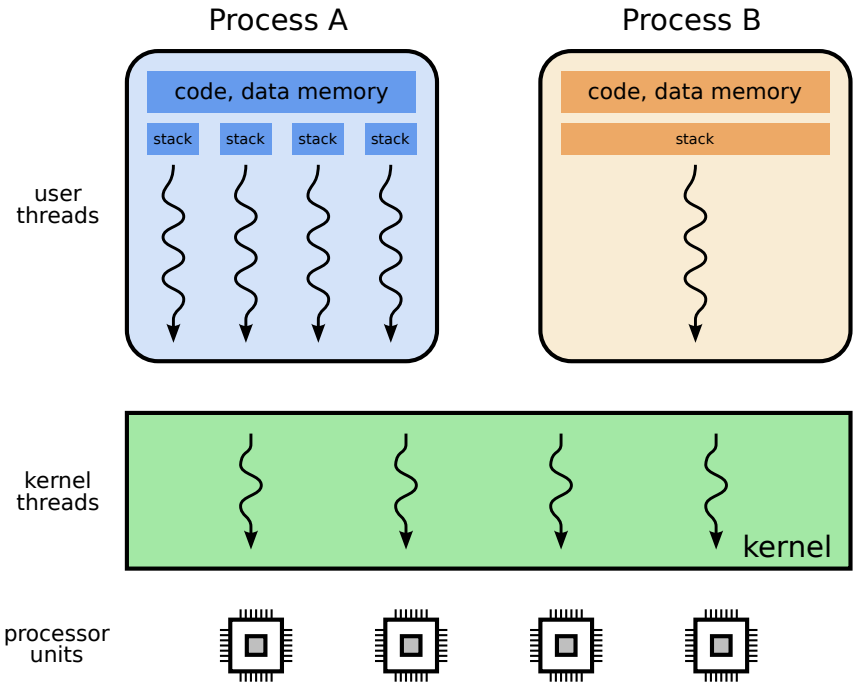


Figura 5.4: *Threads* de usuário e de núcleo.

### 5.4.2 Modelos de *threads*

As *threads* contidas nos processos, definidas no espaço de usuário, devem ser gerenciadas pelo núcleo do sistema operacional. Essa gerência pode ser feita de diversas formas, conforme os modelos de implementação de *threads* apresentados nesta seção.

#### O modelo N:1

Os sistemas operacionais mais antigos suportavam apenas processos sequenciais, com um único fluxo de execução em cada um. Os desenvolvedores de aplicações contornaram esse problema construindo bibliotecas para salvar, modificar e restaurar os registradores da CPU dentro do processo, permitindo assim criar e gerenciar vários fluxos de execução (*threads*) dentro de cada processo, sem a participação do núcleo.

Com essas bibliotecas, uma aplicação pode lançar várias *threads* conforme sua necessidade, mas o núcleo do sistema irá sempre perceber (e gerenciar) apenas um fluxo de execução dentro de cada processo (ou seja, o núcleo irá manter apenas uma *thread* de núcleo por processo). Esta forma de implementação de *threads* é denominada **Modelo de *Threads* N:1**, pois N *threads* dentro de um processo são mapeadas em uma única *thread* no núcleo. Esse modelo também é denominado *fibers* ou ainda *green threads*. A Figura 5.5 ilustra o modelo N:1.

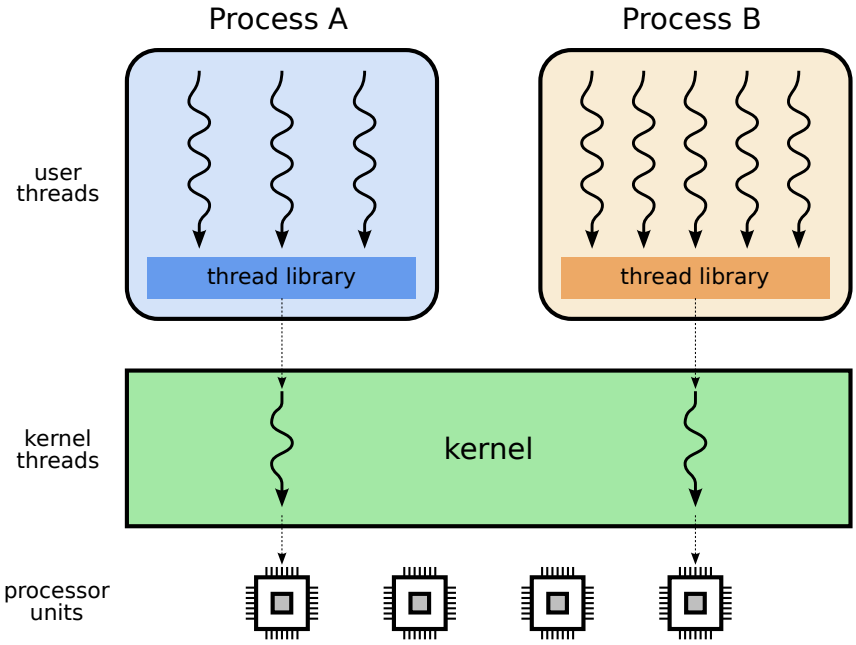


Figura 5.5: O modelo de *threads* N:1.

O modelo de *threads* N:1 é muito utilizado, por ser leve e de fácil implementação. Como o núcleo somente vê uma *thread*, a carga de gerência imposta ao núcleo é pequena e não depende do número de *threads* dentro da aplicação. Essa característica torna este modelo útil na construção de aplicações que exijam muitos *threads*, como jogos ou simulações de grandes sistemas[3]. Exemplos de implementação desse modelo são as bibliotecas *GNU Portable Threads* [Engeschall, 2005], *User-Mode Scheduling* (UMS, Microsoft) e *Green Threads* (Java).

Por outro lado, o modelo de *threads* N:1 apresenta alguns problemas:

- as operações de entrada/saída são realizadas pelo núcleo; se uma *thread* de usuário solicitar uma leitura do teclado, por exemplo, a *thread* de núcleo correspondente será suspensa até a conclusão da operação, bloqueando todas as *threads* daquele processo;
- o núcleo do sistema divide o tempo de processamento entre as *threads* de núcleo. Assim, um processo com 100 *threads* irá receber o mesmo tempo de processador que outro com apenas uma *thread*, ou seja, cada *thread* do primeiro processo irá receber 1/100 do tempo que recebe a *thread* única do outro processo;
- *threads* do mesmo processo não podem executar em paralelo, mesmo se o computador dispuser de vários processadores ou *cores*, porque o núcleo somente escalona as *threads* de núcleo nos processadores.

**O modelo 1:1**

A necessidade de suportar aplicações *multithread* levou os desenvolvedores de sistemas operacionais a incorporar a gerência dos *threads* dos processo no núcleo

[3] A simulação detalhada do tráfego viário de uma cidade grande, por exemplo, pode exigir uma *thread* para cada veículo, podendo portanto chegar a milhões de *threads*.

do sistema. Para cada *thread* de usuário foi então associado um *thread* correspondente dentro do núcleo, suprimindo com isso a necessidade de bibliotecas de *threads*. Essa forma de implementação é denominada **Modelo de *Threads* 1:1**, sendo apresentada na Figura 5.6. Este é o modelo mais frequente nos sistemas operacionais atuais, como o Windows NT e seus descendentes, além da maioria dos UNIXes.

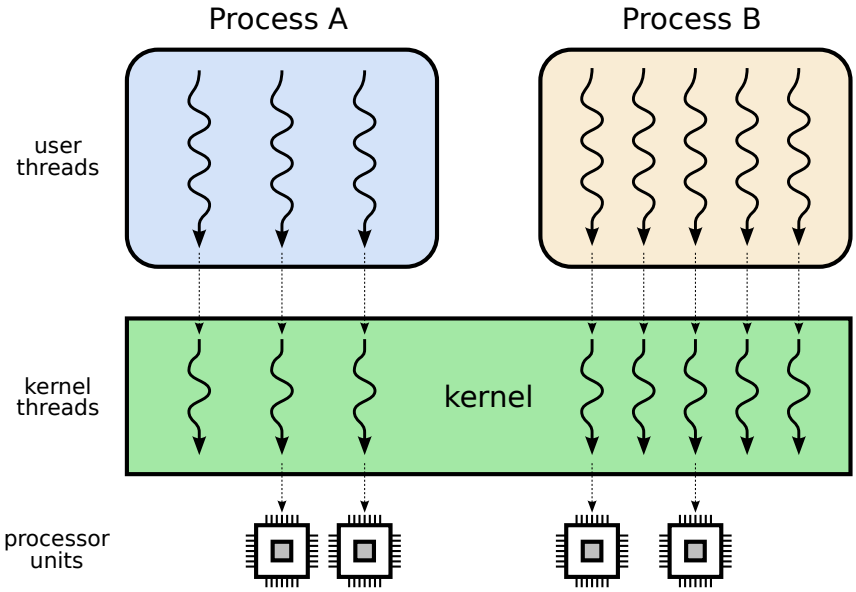


Figura 5.6: O modelo de *threads* 1:1.

O modelo 1:1 resolve vários problemas: caso uma *thread* de usuário solicite uma operação bloqueante, somente sua *thread* de núcleo correspondente será suspensa, sem afetar as demais *threads* do processo. Além disso, a distribuição de processamento entre as *threads* é mais justa e, caso o hardware tenha mais de um processador, mais *threads* do mesmo processo podem executar ao mesmo tempo, o que não era possível no modelo N:1.

O modelo de *threads* 1:1 é adequado para a maioria das situações usuais e atende bem às necessidades das aplicações interativas e servidores de rede. No entanto, é pouco escalável: a criação de um número muito grande de *threads* impõe uma carga elevada ao núcleo do sistema, inviabilizando aplicações com muitas tarefas (como grandes servidores Web e simulações de grande porte).

### O modelo N:M

Para resolver o problema de escalabilidade da abordagem 1:1, alguns sistemas operacionais implementam um modelo híbrido, que agrega características dos modelos anteriores. Nesse novo modelo, uma biblioteca gerencia um conjunto de $N$ *threads* de usuário (dentro do processo), que é mapeado em $M < N$ *threads* no núcleo. Essa abordagem, denominada **Modelo de *Threads* N:M**, é apresentada na Figura 5.7.

O conjunto de *threads* de núcleo associadas a um processo, denominado *thread pool*, geralmente contém uma *thread* para cada *thread* de usuário bloqueada, mais uma *thread* para cada processador disponível; esse conjunto pode ser ajustado dinamicamente, conforme a necessidade da aplicação.

O modelo N:M é implementado pelo Solaris e também pelo projeto KSE (*Kernel-Scheduled Entities*) do FreeBSD [Evans and Elischer, 2003] baseado nas ideias apresentadas

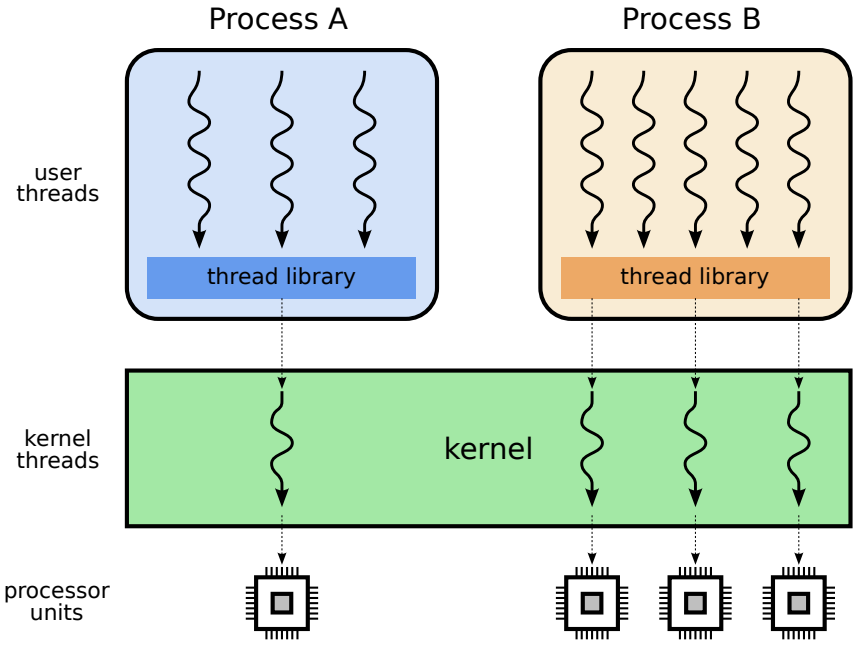


Figura 5.7: O modelo de *threads* N:M.

em [Anderson et al., 1992]. Ele alia as vantagens de maior interatividade do modelo 1:1 à maior escalabilidade do modelo N:1. Como desvantagens desta abordagem podem ser citadas sua complexidade de implementação e maior custo de gerência dos *threads* de núcleo, quando comparados ao modelo 1:1.

#### Comparação entre os modelos

A Tabela 5.2 resume os principais aspectos dos três modelos de implementação de *threads* e faz um comparativo entre eles.

### 5.4.3 Programando com *threads*

No passado, cada sistema operacional definia sua própria interface para a criação de *threads*, o que levou a problemas de portabilidade das aplicações. Em 1995 foi definido o padrão *IEEE POSIX 1003.1c*, mais conhecido como *POSIX Threads* ou simplesmente *PThreads* [Nichols et al., 1996], que define uma interface padronizada para o uso de *threads* na linguagem C. Esse padrão é amplamente difundido e suportado hoje em dia. A listagem a seguir, extraída de [Barney, 2005], exemplifica o uso do padrão *PThreads*:

| **Modelo** | **N:1** | **1:1** | **N:M** |
|---|---|---|---|
| Resumo | N *threads* do processo mapeados em uma *thread* de núcleo | Cada *thread* do processo mapeado em uma *thread* de núcleo | N *threads* do processo mapeados em M< N *threads* de núcleo |
| Implementação | no processo (biblioteca) | no núcleo | em ambos |
| Complexidade | baixa | média | alta |
| Custo de gerência | baixo | médio | alto |
| Escalabilidade | alta | baixa | alta |
| Paralelismo entre *threads* do mesmo processo | não | sim | sim |
| Troca de contexto entre *threads* do mesmo processo | rápida | lenta | rápida |
| Divisão de recursos entre tarefas | injusta | justa | variável, pois o mapeamento *thread*→ processador é dinâmico |
| Exemplos | GNU Portable Threads, Microsoft UMS | Windows, Linux | Solaris, FreeBSD KSE |

Tabela 5.2: Comparativo dos modelos de *threads*.

```c
// Exemplo de uso de threads Posix em C no Linux
// Compilar com gcc exemplo.c -o exemplo -lpthread

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

#define NUM_THREADS 5

// cada thread vai executar esta função
void *print_hello (void *threadid)
{
   printf ("%ld: Hello World!\n", (long) threadid);
   sleep (5) ;
   printf ("%ld: Bye bye World!\n", (long) threadid);
   pthread_exit (NULL);     // encerra esta thread
}
```

```
19 // thread "main" (vai criar as demais threads)
20 int main (int argc, char *argv[])
21 {
22    pthread_t thread[NUM_THREADS];
23    long status, i;
24
25    // cria as demais threads
26    for(i = 0; i < NUM_THREADS; i++)
27    {
28       printf ("Creating thread %ld\n", i);
29       status = pthread_create (&thread[i], NULL, print_hello, (void *) i);
30
31       if (status) // ocorreu um erro
32       {
33          perror ("pthread_create");
34          exit (-1);
35       }
36    }
37
38    // encerra a thread "main"
39    pthread_exit (NULL);
40 }
```

Nessa listagem pode-se perceber que o programa inicia sua execução com apenas uma *thread* (representada pela função `main()`, linha 20); esta, por sua vez, cria as demais *threads* (linha 29) e indica o código a ser executado pelas mesmas (no exemplo, todas executam a mesma função `print_hello()`, linha 12).

*Threads* podem ser utilizadas em diversas outras linguagens de programação, como Java, Python, Perl, etc. O código a seguir traz um exemplo simples de criação de *threads* em Java (extraído da documentação oficial da linguagem):

```
1  // save as MyThread.java; javac MyThread.java; java MyThread
2
3  public class MyThread extends Thread {
4     int threadID;
5
6     private static final int NUM_THREADS = 5 ;
7
8     MyThread (int ID) {
9        threadID = ID;
10    }
11
12    // corpo de cada thread
13    public void run () {
14       System.out.println (threadID + ": Hello World!") ;
15       try {
16         Thread.sleep(5000);
17       }
18       catch (InterruptedException e) {
19         e.printStackTrace();
20       }
21       System.out.println (threadID + ": Bye bye World!") ;
22    }
```

```
23    public static void main (String args[]) {
24
25       MyThread[] t = new MyThread[NUM_THREADS] ;
26
27       // cria as threads
28       for (int i = 0; i < NUM_THREADS; i++) {
29          t[i] = new MyThread (i);
30       }
31
32       // inicia a execução das threads
33       for (int i = 0; i < NUM_THREADS; i++) {
34          t[i].start () ;
35       }
36    }
37 }
```

## 5.5 Uso de processos *versus threads*

Neste capítulo foram apresentadas duas formas de implementar tarefas em um sistema operacional: os processos e as *threads*. No caso do desenvolvimento de sistemas complexos, compostos por várias tarefas interdependentes executando em paralelo, qual dessas formas de implementação deve ser escolhida? Um navegador de Internet (*browser*), um servidor Web ou um servidor de bancos de dados atendendo vários usuários simultâneos são exemplos desse tipo de sistema.

A implementação de sistemas usando **um processo para cada tarefa** é uma possibilidade. Essa abordagem tem como grande vantagem a robustez, pois um erro ocorrido em um processo ficará restrito ao seu espaço de memória pelos mecanismos de isolamento do hardware. Além disso, os processos podem ser configurados para executar com usuários (e permissões) distintas, aumentando a segurança do sistema. Por outro lado, nessa abordagem o compartilhamento de dados entre os diversos processos que compõem a aplicação irá necessitar de mecanismos especiais, como áreas de memória compartilhada, pois um processo não pode acessar as áreas de memória dos demais processos. Como exemplos de sistemas que usam a abordagem baseada em processos, podem ser citados o servidor web Apache (nas versões 1.*) e o servidor de bancos de dados PostgreSQL.

Também é possível implementar todas as tarefas em um único processo, usando **uma *thread* para cada tarefa**. Neste caso, todas as *threads* compartilham o mesmo espaço de endereçamento e os mesmos recursos (arquivos abertos, conexões de rede, etc), tornando mais simples implementar a interação entre as tarefas. Além disso, a execução é mais ágil, pois é mais rápido criar uma *thread* que um processo. Todavia, um erro em uma *thread* pode se alastrar às demais, pondo em risco a robustez da aplicação inteira. Muitos sistemas usam uma abordagem multi-*thread*, como o servidor Web IIS (Microsoft *Internet Information Server*) e o editor de textos LibreOffice.

Sistemas mais recentes e sofisticados usam uma abordagem híbrida, com o **uso conjunto de processos e *threads***. A aplicação é composta por vários processos, cada um contendo diversas *threads*. Busca-se com isso aliar a robustez proporcionada pelo isolamento entre processos e o desempenho, menor consumo de recursos e facilidade de programação proporcionados pelas *threads*. Esse modelo híbrido é usado, por exemplo, nos navegadores Chrome e Firefox: cada aba de navegação é atribuída a um novo

processo, que cria as *threads* necessárias para buscar e renderizar aquele conteúdo e interagir com o usuário. O servidor Web Apache 2.* e o servidor de bases de dados Oracle também usam essa abordagem.

A Figura 5.8 ilustra as três abordagens apresentadas nesta seção; a Tabela 5.3 sintetiza a comparação entre elas.

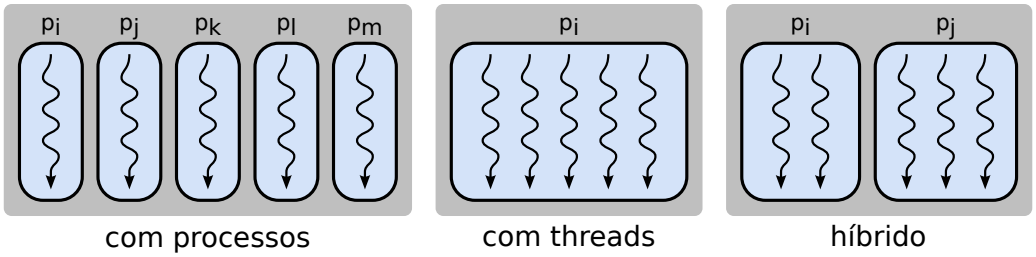


Figura 5.8: Implementação de sistemas usando processos, *threads* ou híbrido.

| **Característica** | **Com processos** | **Com *threads* (1:1)** | **Híbrido** |
|---|---|---|---|
| Custo de criação de tarefas | alto | baixo | médio |
| Troca de contexto | lenta | rápida | variável |
| Uso de memória | alto | baixo | médio |
| Compartilhamento de dados entre tarefas | canais de comunicação e áreas de memoria compartilhada. | variáveis globais e dinâmicas. | ambos. |
| Robustez | alta, um erro fica contido no processo. | baixa, um erro pode afetar todas as *threads*. | média, um erro pode afetar as *threads* no mesmo processo. |
| Segurança | alta, cada processo pode executar com usuários e permissões distintas. | baixa, todas as *threads* herdam as permissões do processo onde executam. | alta, *threads* que necessitam as mesmas permissões podem ser agrupadas em um mesmo processo. |
| Exemplos | Servidor Apache (versões 1.*), SGBD PostGres | Servidor Apache (versões 2.*), SGBD MySQL | Navegadores Chrome e Firefox, SGBD Oracle |

Tabela 5.3: Comparativo entre uso de processos e de *threads*.

# Exercícios

1. Explique o que é, para que serve e o que contém um TCB - *Task Control Block*.
2. Desenhe o diagrama de tempo da execução do código a seguir, informe qual a saída do programa na tela (com os valores de $x$) e calcule a duração aproximada de sua execução.

```c
int main()
{
   int x = 0 ;

   fork () ;
   x++ ;
   sleep (5) ;
   wait (0) ;
   fork () ;
   wait (0) ;
   sleep (5) ;
   x++ ;
   printf ("Valor de x: %d\n", x) ;
}
```

3. Indique quantas letras "X" serão impressas na tela pelo programa abaixo quando for executado com a seguinte linha de comando:

   `a.out 4 3 2 1`

   O comando `a.out` resulta da compilação do programa a seguir:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   pid_t pid[10];
   int i;

   int N = atoi(argv[argc-2]);

   for (i=0; i<N; i++)
      pid[i] = fork();
   if (pid[0] != 0 && pid[N-1] != 0)
      pid[N] = fork();
   printf("X");
   return 0;
}
```

4. O que são *threads* e para que servem?

5. Quais as principais vantagens e desvantagens de *threads* em relação a processos?

6. Forneça dois exemplos de problemas cuja implementação *multi-thread* não tem desempenho melhor que a respectiva implementação sequencial.

7. Associe as afirmações a seguir aos seguintes modelos de *threads*: a) *many-to-one* (N:1); b) *one-to-one* (1:1); c) *many-to-many* (N:M):

   (a) Tem a implementação mais simples, leve e eficiente.

(b) Multiplexa os *threads* de usuário em um pool de *threads* de núcleo.

(c) Pode impor uma carga muito pesada ao núcleo.

(d) Não permite explorar a presença de várias CPUs pelo mesmo processo.

(e) Permite uma maior concorrência sem impor muita carga ao núcleo.

(f) Geralmente implementado por bibliotecas.

(g) É o modelo implementado no Windows NT e seus sucessores.

(h) Se um *thread* bloquear, todos os demais têm de esperar por ele.

(i) Cada *thread* no nível do usuário tem sua correspondente dentro do núcleo.

(j) É o modelo com implementação mais complexa.

8. Considerando as implementações de *threads* N:1 e 1:1 para o trecho de código a seguir, a) desenhe os diagramas de execução, b) informe as durações aproximadas de execução e c) indique a saída do programa na tela. Considere a operação `sleep()` como uma chamada de sistema (*syscall*).

   A chamada `thread_create` cria uma nova *thread*, `thread_exit` encerra a *thread* corrente e `thread_join` espera o encerramento da *thread* informada como parâmetro.

```
1  int y = 0 ;
2
3  void threadBody
4  {
5     int x = 0 ;
6     sleep (10) ;
7     printf ("x: %d, y:%d\n", ++x, ++y) ;
8     thread_exit();
9  }
10
11 main ()
12 {
13    thread_create (&tA, threadBody, ...) ;
14    thread_create (&tB, threadBody, ...) ;
15    sleep (1) ;
16    thread_join (&tA) ;
17    thread_join (&tB) ;
18    sleep (1) ;
19    thread_create (&tC, threadBody, ...) ;
20    thread_join (&tC) ;
21 }
```

# Referências

T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

B. Barney. POSIX threads programming. http://www.llnl.gov/computing/tutorials/pthreads, 2005.

R. Engeschall. The GNU Portable Threads. http://www.gnu.org/software/pth, 2005.

J. Evans and J. Elischer. Kernel-scheduled entities for FreeBSD. http://www.aims.net.au/chris/kse, 2003.

B. Nichols, D. Buttlar, and J. Farrell. *PThreads Programming*. O'Reilly Media, Inc, 1996.

A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.