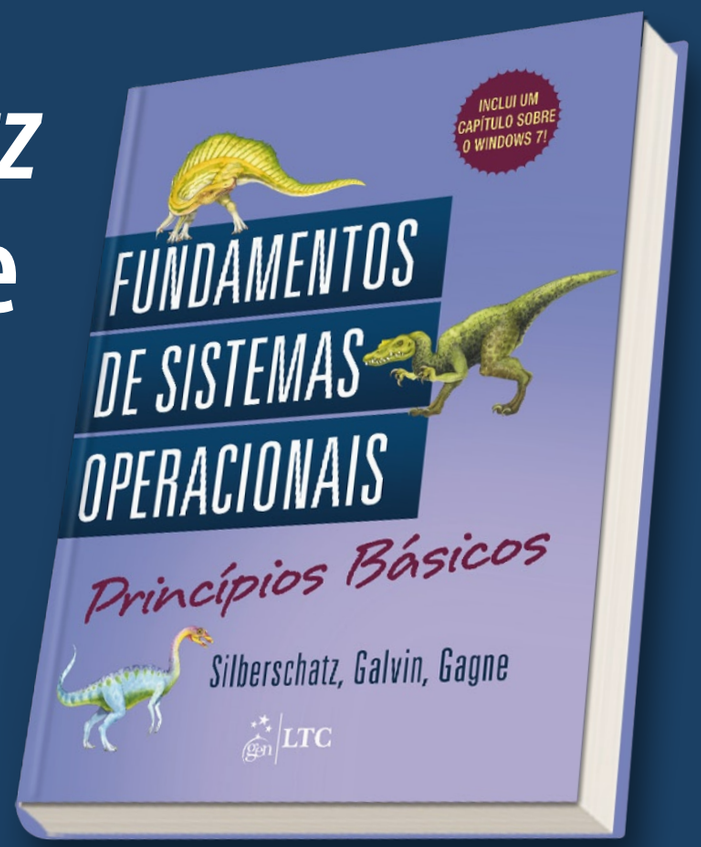
Abraham Silberschatz

Fundamentos de Sistemas Operacionais

Princípios Básicos



Grupo Editorial Nacional

LTC EDITORA

GEN | Informação Online

www.**grupogen**.com.br

http://**gen-io.grupogen**.com.br

Capítulo 3
PROCESSOS

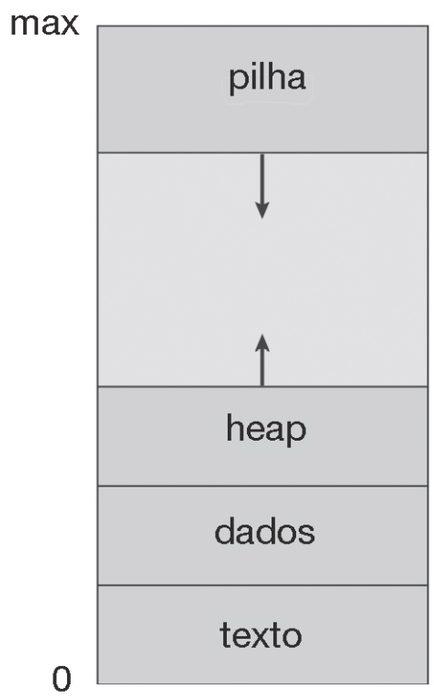


Figura 3.1 Processo na memória.

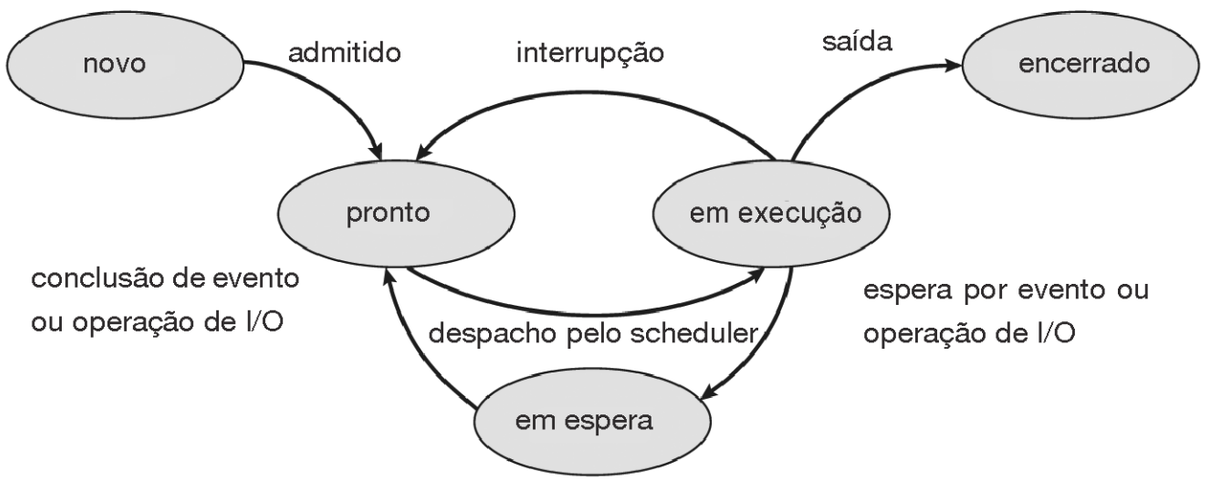
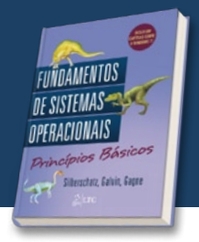


Figura 3.2 Diagrama de estado do processo.



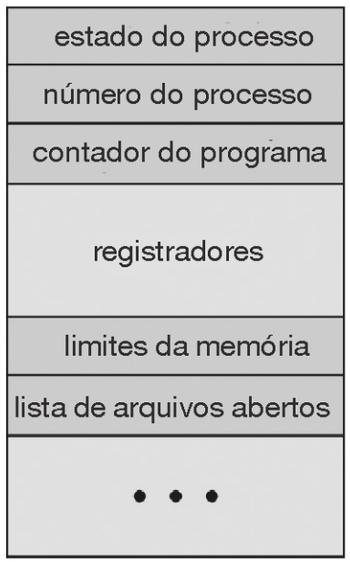


Figura 3.3 Bloco de controle de processo (PCB).

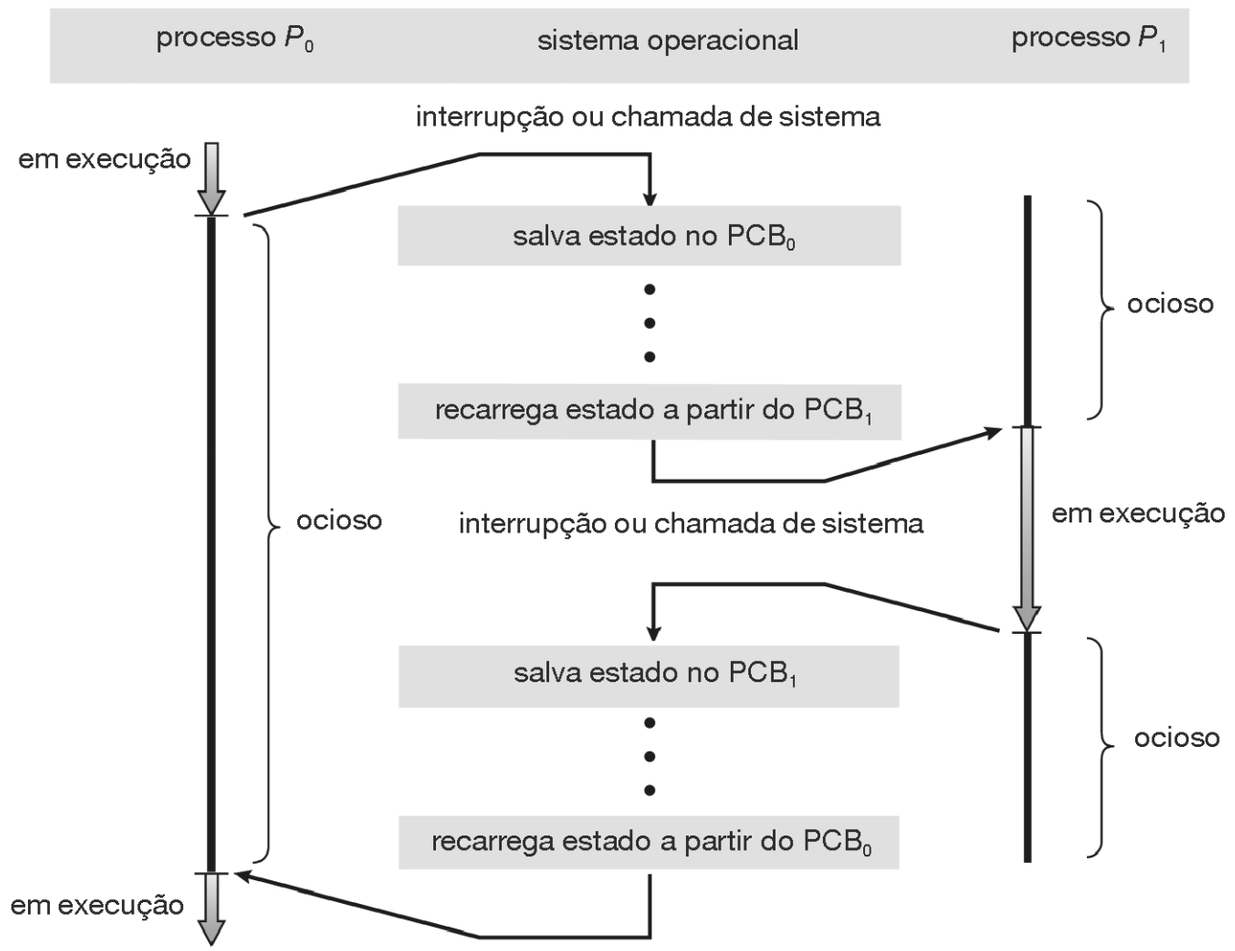


Figura 3.4 Diagrama mostrando a alternância de CPU de um processo para outro.

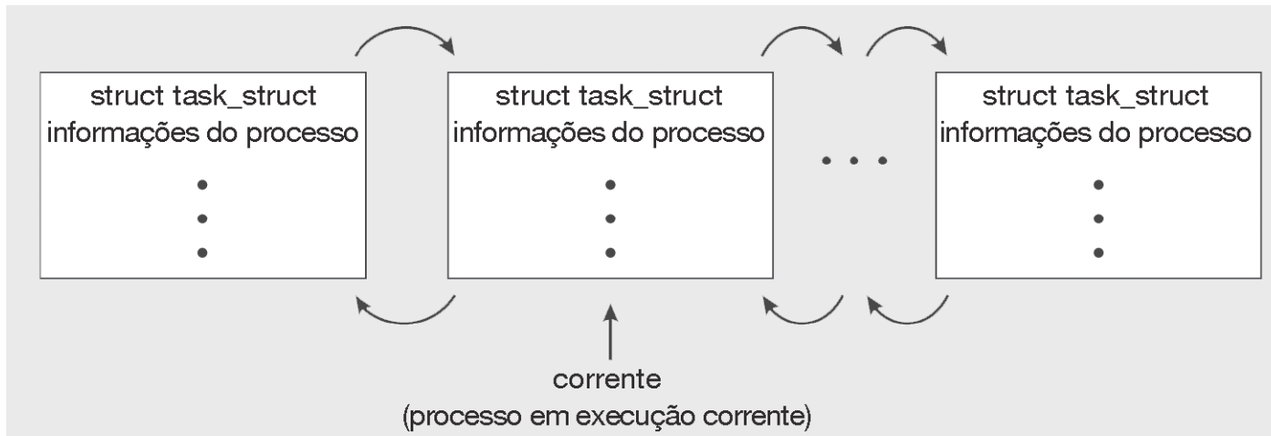


Figura 3.5 Processos ativos no Linux.

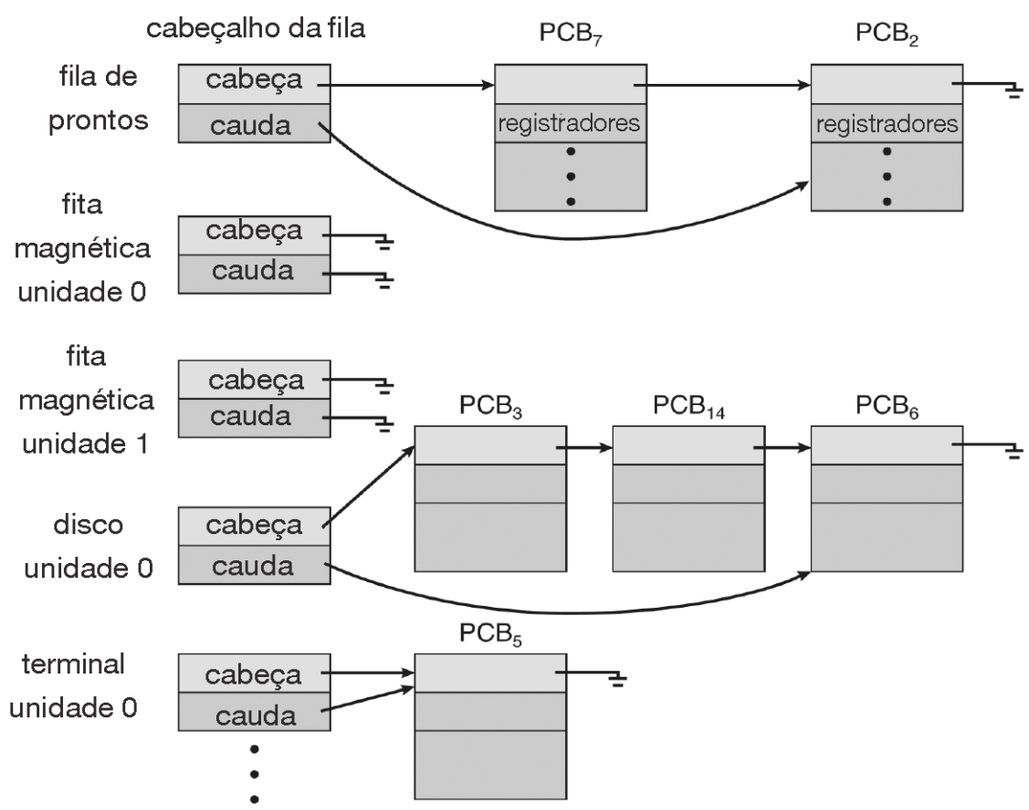


Figura 3.6 A fila de prontos e várias filas de dispositivos de I/O.

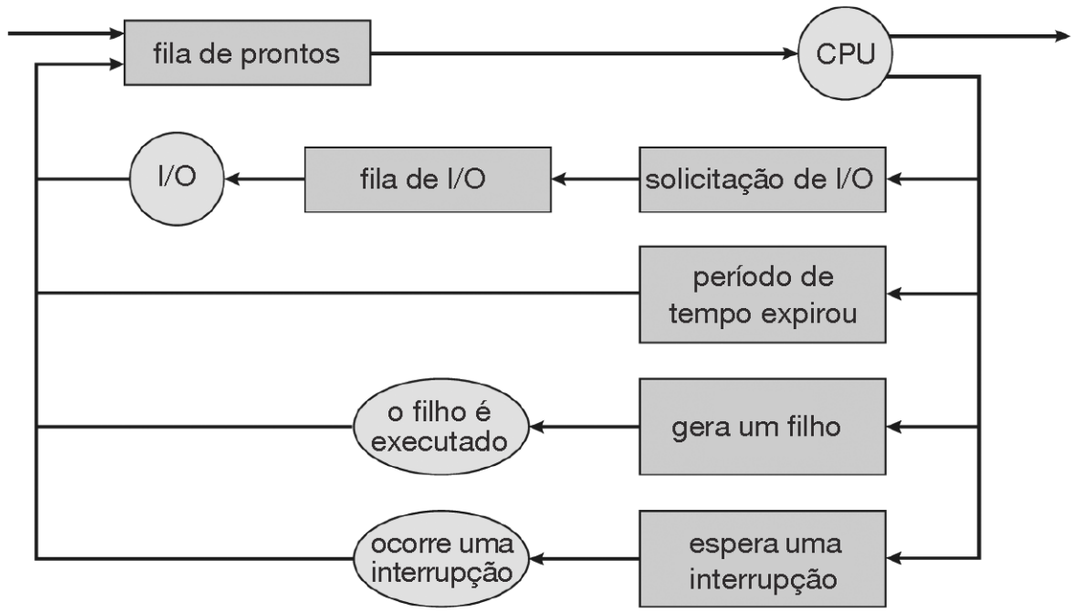


Figura 3.7 Representação do processo do scheduling de processos em diagrama de enfileiramento.

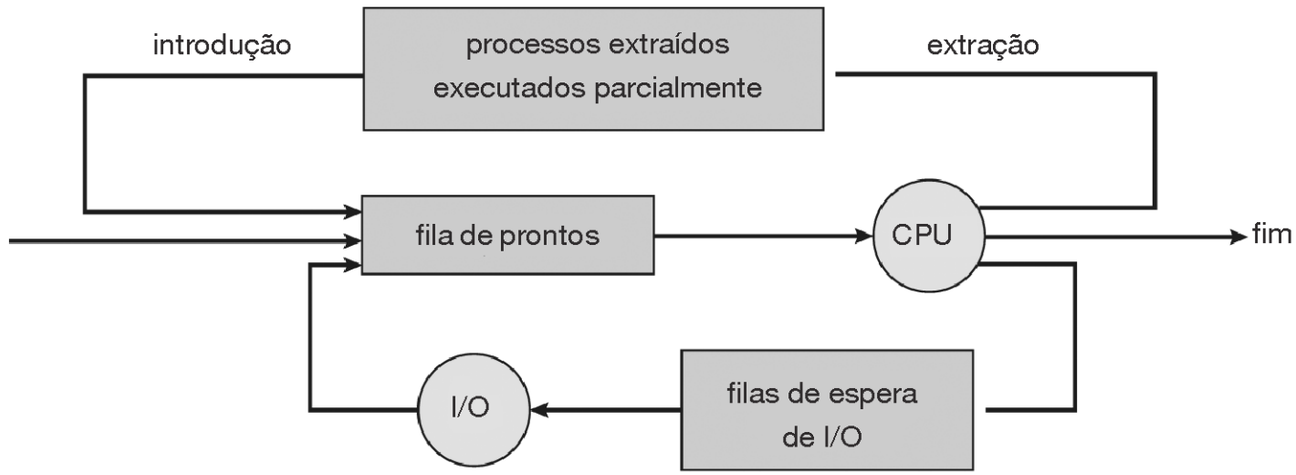


Figura 3.8 Inclusão de scheduling de médio prazo no diagrama de enfileiramento.

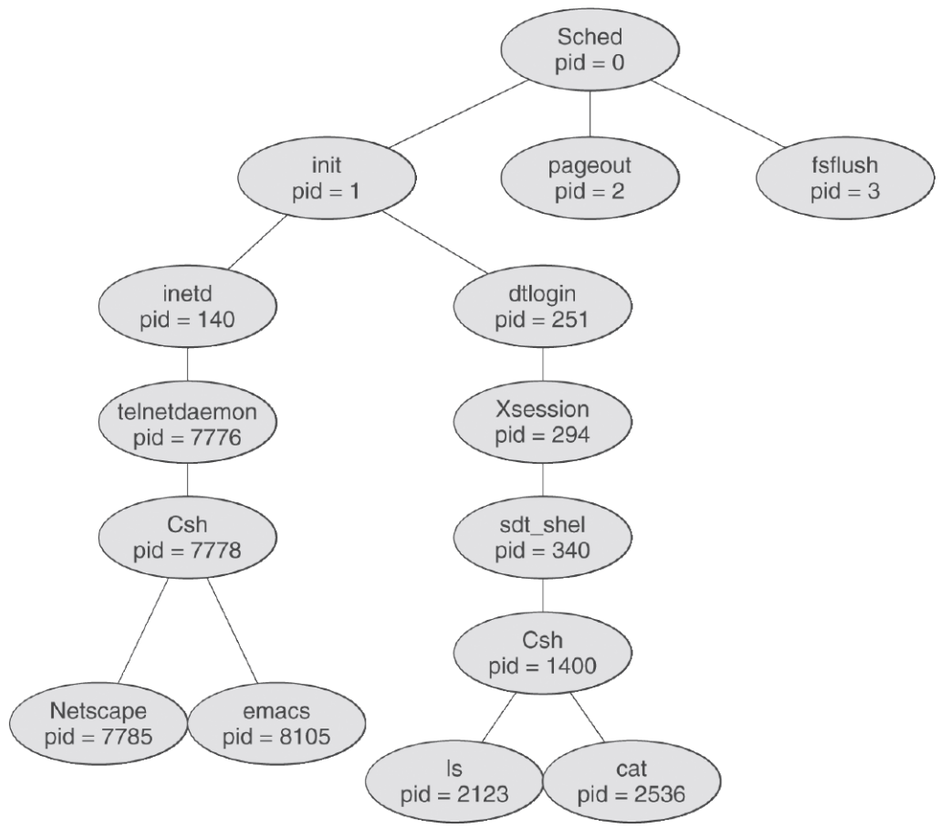
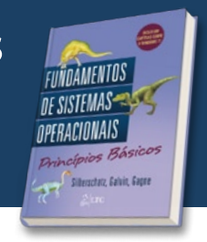


Figura 3.9 Uma árvore de processos em um sistema Solaris típico.




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

int main()
{
pid_t pid;

   /* gera um processo filho */
   pid = fork();

if (pid < 0) {/* um erro ocorreu */
   fprintf(stderr, "Fork Failed");
   return 1;
}
else if (pid == 0) {/* processo filho */
   execlp("/bin/ls","ls",NULL);
}
else { /* processo pai */
  /* o pai esperará o filho ser concluído */
   wait(NULL);
   printf("Child Complete");
}
```

Figura 3.10 Criando um processo separado usando a chamada de sistema `fork()` do UNIX.

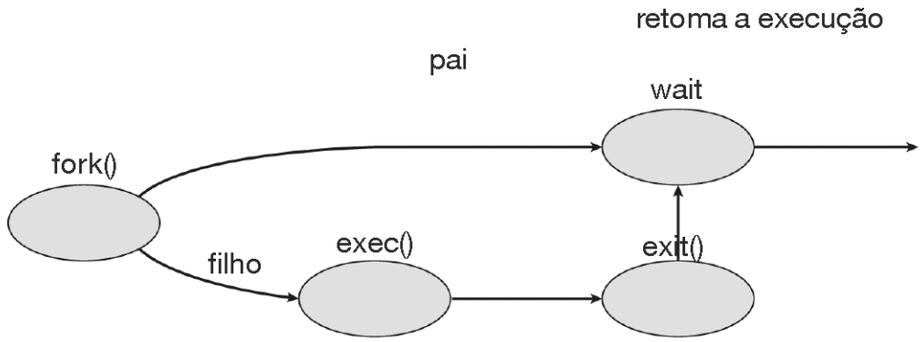


Figura 3.11 Criação de processo com o uso da chamada de sistema `fork()`.


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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // aloca memória
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // cria processo filho
    if (!CreateProcess(NULL, // usa linha de comando
     "C:\\WINDOWS\\system32\\mspaint.exe",//linha de comando
     NULL, // não herda manipulador do processo
     NULL, // não herda manipulador do thread
     FALSE, // disativa a herança de manipuladores
     0, // sem flags de criação
     NULL, // usa o bloco de ambiente do pai
     NULL, // usa o diretório existente do pai
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    // o pai esperará o filho ser concluído
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // fecha manipuladores
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figura 3.12 Criando um processo separado usando a API Win32.

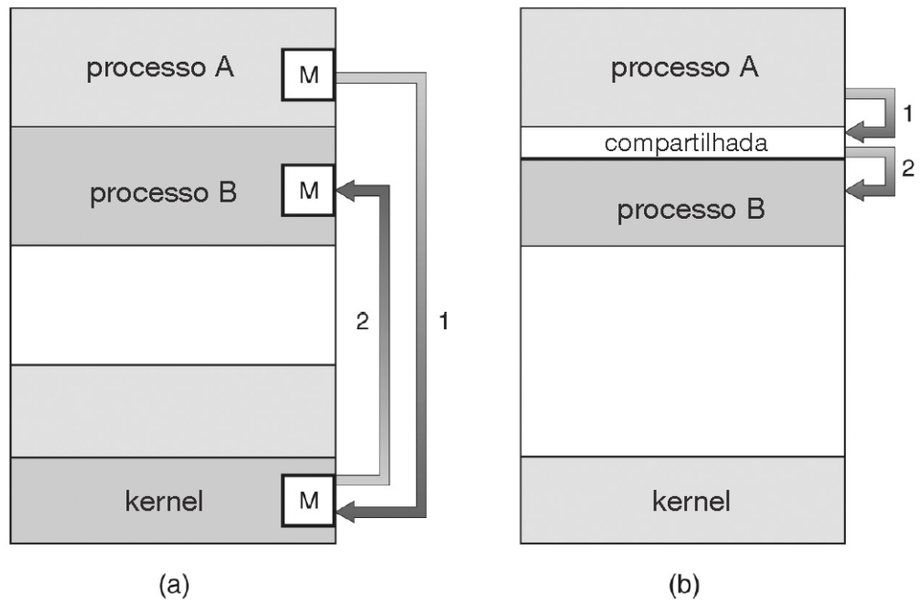
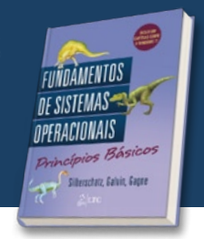


Figura 3.13 Modelos de comunicação. (a) Transmissão de mensagem. (b) Memória compartilhada.




```
while (true) {
        /* produz um item em nextProduced */
        while (((in + 1) % BUFFER_SIZE) == out)
           ; /* não faz coisa alguma */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Figura 3.14 O processo produtos.


```
item nextConsumed;

while (true) {
     while (in == out)
        ; // não faz coisa alguma

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consome o item em nextConsumed */
}
```

Figura 3.15 O processo consumidor.


```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* o identificador do segmento de memória compartilhada */
int segment_id;
/* um ponteiro para o segmento de memória compartilhada */
char *shared_memory;
/* o tamanho (em bytes) do segmento de memória compartilhada */
const int size = 4096;

    /* aloca um segmento de memória compartilhada */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* anexa o segmento de memória compartilhada */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* grava uma mensagem no segmento de memória compartilhada */
    sprintf(shared_memory, "Hi there!");

    /* agora, exibe a cadeia de caracteres a partir da memória compartilhada */
    printf("*%s\n", shared_memory);

    /* desanexa o segmento de memória compartilhada */
    shmdt(shared_memory);

    /* remove o segmento de memória compartilhada */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Figura 3.16 Programa em C ilustrando a API POSIX de memória compartilhada.

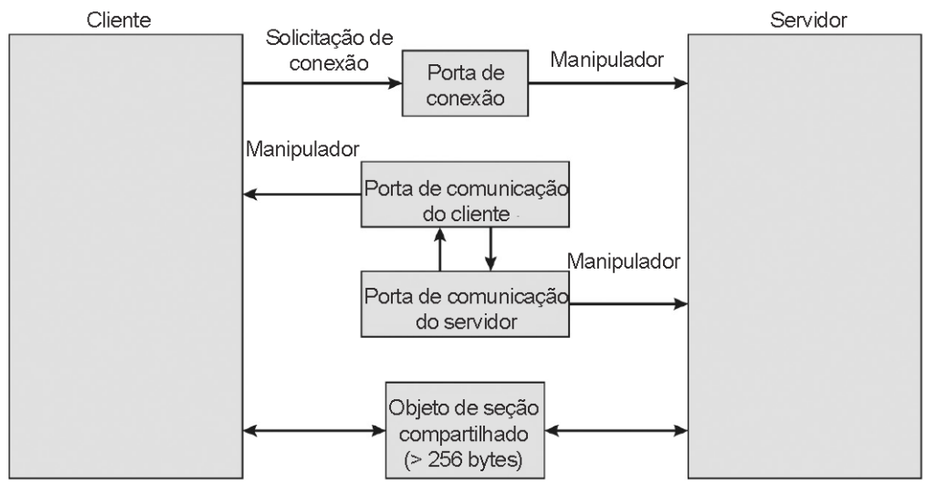


Figura 3.17 Chamadas de procedimento locais no Windows.

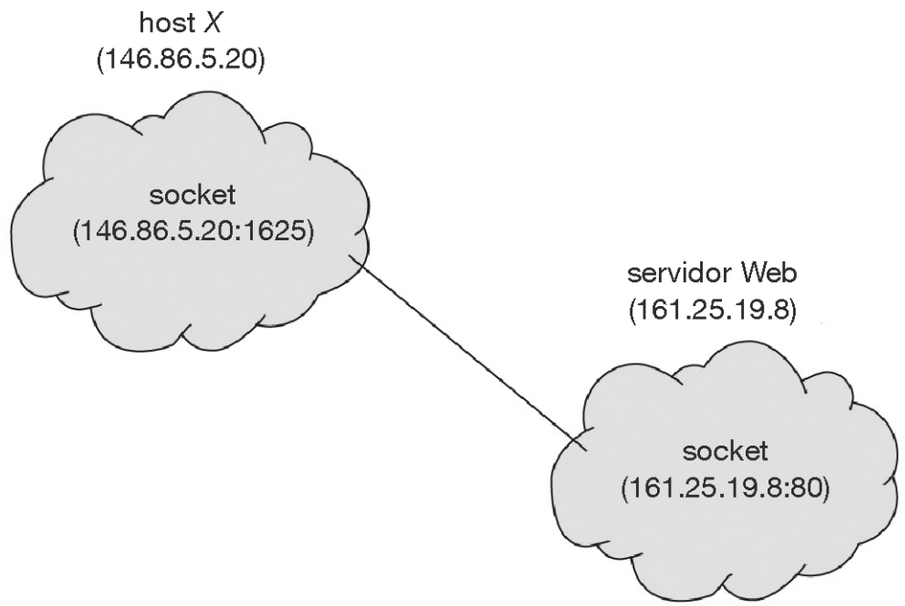


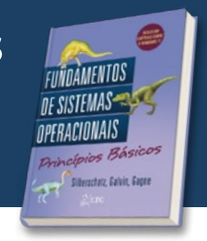
Figura 3.18 Comunicação com o uso de sockets.


```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // agora, espera conexões
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // grava a data no socket
            pout.println(new java.util.Date().toString());

            // fecha o socket e volta
            // a escutar conexões
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figura 3.19 Servidor de data.


```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         //estabelece conexão com o socket do servidor
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // lê a data no socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // fecha a conexão com o socket
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figura 3.20 Cliente do servidor de data.



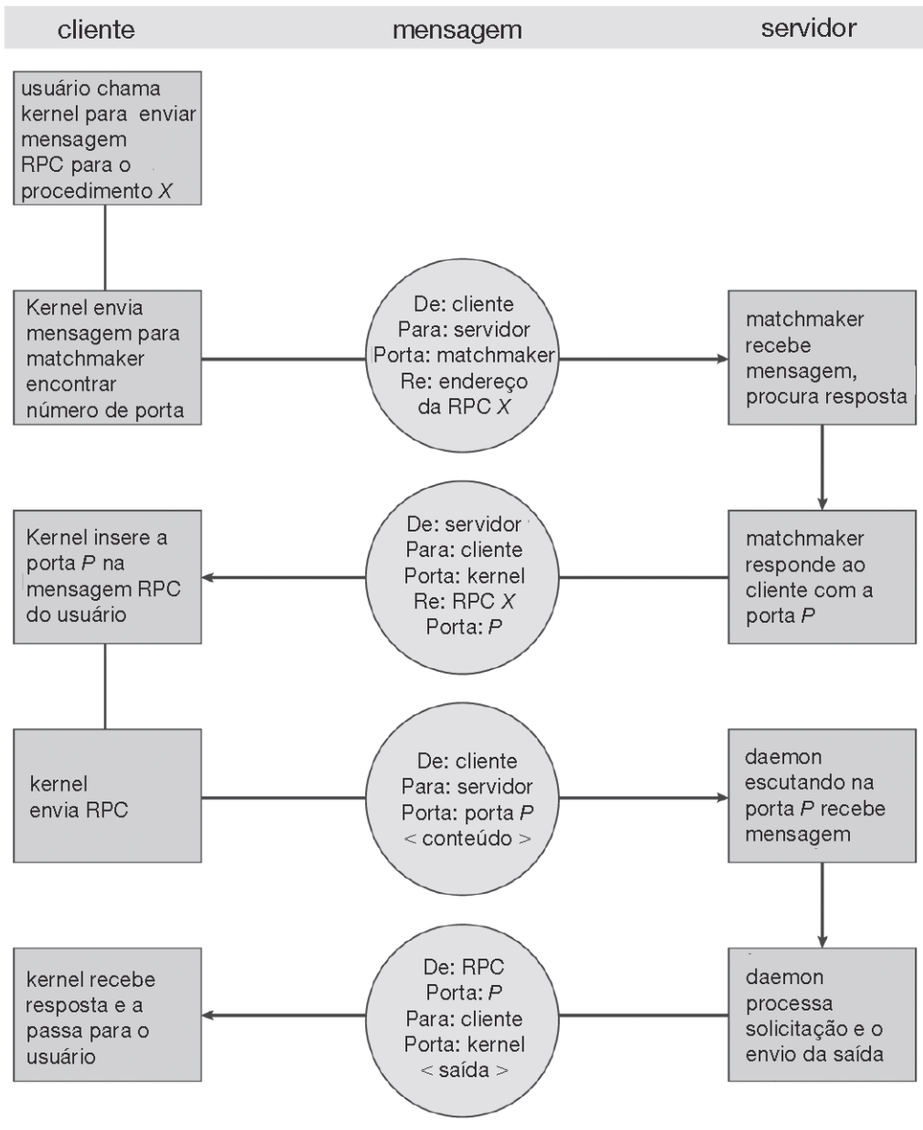
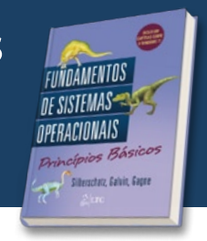
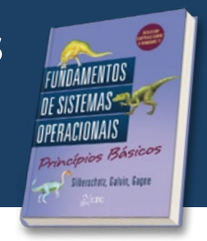


Figura 3.21 Execução de uma chamada de procedimento remota (RPC).



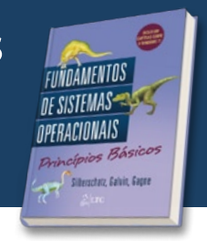

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* gera um processo filho */
    fork();

    /* gera outro processo filho */
    fork();

    /* e gera ainda mais um */
    fork();

    return 0;
}
```

Figura 3.22 Quantos processos são criados?




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

int main()
{
pid_t pid, pid1;

   /* gera um processo filho */
   pid = fork();

   if (pid < 0) {/* um erro ocorreu */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) {/* processo filho */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* processo pai */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
   }

   return 0;
}
```

Figura 3.23 Quais são os valores de pid?




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

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) {/* processo filho */
      value += 15;
      return 0;
   }
   else if (pid > 0) {/* processo pai */
      wait(NULL);
      printf("PARENT: value = %d",value);/* LINHA A */
      return 0;
   }
}
```

Figura 3.24 Que saída teremos na linha A?