Processos e Threads

Por Hermes Senger / Hélio Crestana Guardia

De modo geral, os sistemas operacionais tratam as aplicações (programas) que se encontram em execução como **processos**. Em uma definição bem simples, processos são instâncias de programas em execução.

Para criar um processo, é preciso gerar um código executável apropriado para a arquitetura (processador) e o sistema operacional do computador onde se deseja executá-lo. Isso pode ser feito usando comandos do repertório de instruções em linguagem *assembly* do processador, ou, mais comumente, usando uma linguagem de alto nível, que depois é compilada e traduzida para comandos executáveis.

Em alguns casos, como ocorre com a linguagem *java*, é possível criar programas que são traduzidos para um formato que pode ser executado (ou interpretado) em diferentes sistemas computacionais. Para isso, usa-se uma máquina virtual no ambiente alvo.

Usando ambientes gráficos, como *Gnome* e *KDE* (entre muitos outros) num ambiente Unix, ou as interfaces gráficas de sistemas Windows, a ativação de um processo pode ser feita selecionando-se ícones ou itens listados em gerenciadores de arquivos. Quando interagindo com um *shell* interpretador de comandos, como o *bash* em Unix, ou *command.com* em sistemas Windows, a criação de processo é feita diretamente em linha de comando. Também é possível criar um processo a partir de um processo já em execução, fazendo chamada a serviços específicos do sistema operacional.

Para listar quais processos estão sendo executados no computador, num *shell*, digite o comando ***ps –ef*** (ou ***ps aux***, ou simplesmente **ps**) caso esteja usando um sistema do tipo UNIX[1]. Se estiver no Windows execute o ***task manager*** (ou gerente de tarefas – digite ***Ctrl-Shift-Esc***) e você verá algo parecido com isto que se vê na Figura 1.

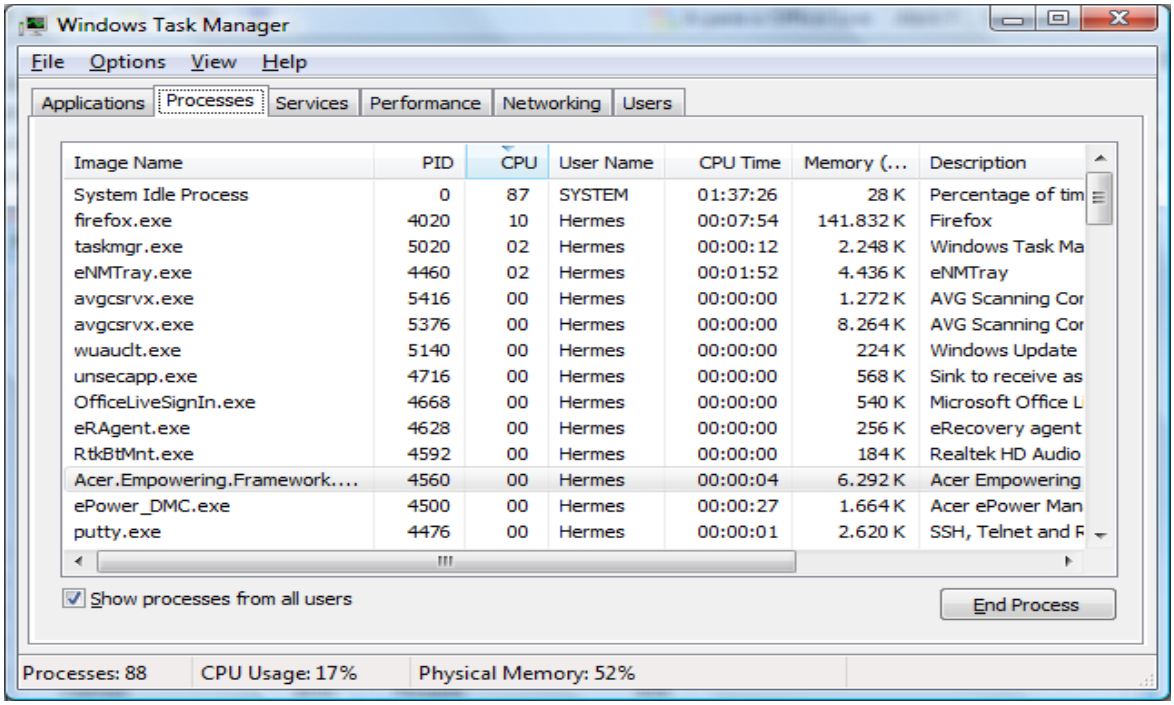

Figura 1 – Lista de processos no gerenciador de tarefas.

[1] De agora em diante, quando utilizarmos o termo UNIX queremos nos referir as diversas versões e implementações de sistemas inspirados no UNIX, tais como o Linux, Solaris, FreeBSD, IRIX, HPUX, etc.

Essa aplicação mostra uma pequena lista (com parte) dos processos que estão em execução. Em sistemas UNIX, também é possível usar o programa **top**, ativado a partir de um *shell*, para mostrar uma visão interativa dos processos em execução.

Quando iniciamos a execução de um programa qualquer, como um editor de texto, o sistema operacional lê do disco o seu código executável e o carrega na memória principal. Isso é necessário pois as instruções a executar devem estar na memória. Para isso, o SO aloca espaço na memória principal para fazer a carga do **código** e também para as áreas de **dados**, que incluem: um espaço para os valores constantes e as variáveis estáticas, um espaço para alocação dinâmica de memória, chamado *heap*, e uma área para uma pilha (*stack*).

Além do código e das áreas de dados, diversas informações de controle precisam ser mantidas pelo SO para um processo em execução. Para isso, o SO cria um descritor de processo, também chamado de **PCB** (*process control block,* ou bloco de controle de processo). Um PCB contém diversas informações importantes, tais como o número **identificador do processo** (o PID que aparece na Figura 1), o **estado do processo**, um **contador de programa** (que indica o endereço da próxima instrução a executar para esse processo), valores de **prioridade** para escalonamento, credenciais do usuário ao qual o processo está associado, indicações sobre o processo pai desse processo, ponteiros para processos filhos, ponteiros para os endereços na memória que contêm os dados e instruções do processo, ponteiros para recursos alocados, tais como *buffers*, arquivos abertos e mecanismos de comunicação entre processos, entre outros. Parte desse conjunto de informações, incluindo o estado dos **registradores** do hardware no momento em que o processo teve sua execução interrompida, é comumente chamado de **contexto do processo**.

Depois de criado, um processo pode assumir um dentre vários estados. Logo após a sua criação, o processo entra em um estado chamado de **pronto**. Isso significa que ele está pronto para ser executado, tão logo chegue a sua vez. Como normalmente há mais processos a executar do que processadores (CPUs) disponíveis, os processos normalmente alternam-se no uso do(s) processador(es), de acordo com políticas definidas pelo SO. Todos os processos prontos (ou melhor, que estão no estado pronto) ficam em uma fila de execução, ou **fila de processos prontos**. Na verdade, os PCBs desses processos é que ficam ligados nessa fila [2]. Quando o processador fica livre, ele é atribuído ao primeiro processo da fila de prontos, ou o mais prioritário, que passará para o estado de execução. O ato de atribuir um processador ao processo da vez é chamado de **escalonamento** e é executado por um componente do sistema chamado de **escalonador** (*scheduler*), ou **despachante** (*dispatcher*).

Um processo permanece no estado **em execução** até que algum evento faça com que ele mude de estado. Um exemplo de evento que pode provocar a mudança de estado é a solicitação de uma operação de E/S ao SO pelo processo, como uma leitura de um registro de um arquivo em disco. Nesse momento, o processo "perderá" o uso do processador e passará para o estado **bloqueado**. A sua PCB será então armazenada em uma **fila de processos bloqueados** que aguardam uma leitura em disco, e a CPU será atribuída para a execução de outro processo. Como operações de E/S são tipicamente muito mais demoradas do que o acesso à memória para execução de instruções, nesse caso, o SO aproveita para sobrepor as transferências de dados de/para dispositivos com a execução de instruções pelo processador. Esse é o princípio básico da **multiprogramação**. Isso é possível porque os controladores de dispositivos de E/S, como os controladores de disco, possuem pequenos processadores dedicados, que conseguem transferir dados entre a memória do computador e os dispositivos, sem (ou com pouca) interferência da CPU. Quando terminam de realizar uma operação solicitada, esses controladores normalmente notificam a CPU desse evento através de uma interrupção. Ao saber que a operação de E/S está concluída, o SO move o processo bloqueado de volta para a fila de prontos.

[2]Em alguns SOs, a fila de processos prontos pode ser uma estrutura complexa, formada por diferentes filas ou outras estruturas, organizadas de acordo com prioridades.

Mesmo que um processo não solicite nenhuma operação de E/S, existe outro tipo de evento que pode tirá-lo do estado de execução. Para melhorar a alternância do uso do processador, de forma que todos os processos prontos sejam executados, o SO estabelece uma fatia de tempo (*time-slice*) para que cada processo tenha suas instruções executadas, criando rodadas de execução. Ao final desse tempo, uma **interrupção** gerada por um dispositivo temporizador *(timer)* avisa sobre o término do *time-slice*, e o SO muda o estado do processo de execução para o estado de pronto. Nesse momento sua PCB é recolocada no final da fila de execução. A Figura 2 ilustra as diversas filas de processos típicas de um sistema.

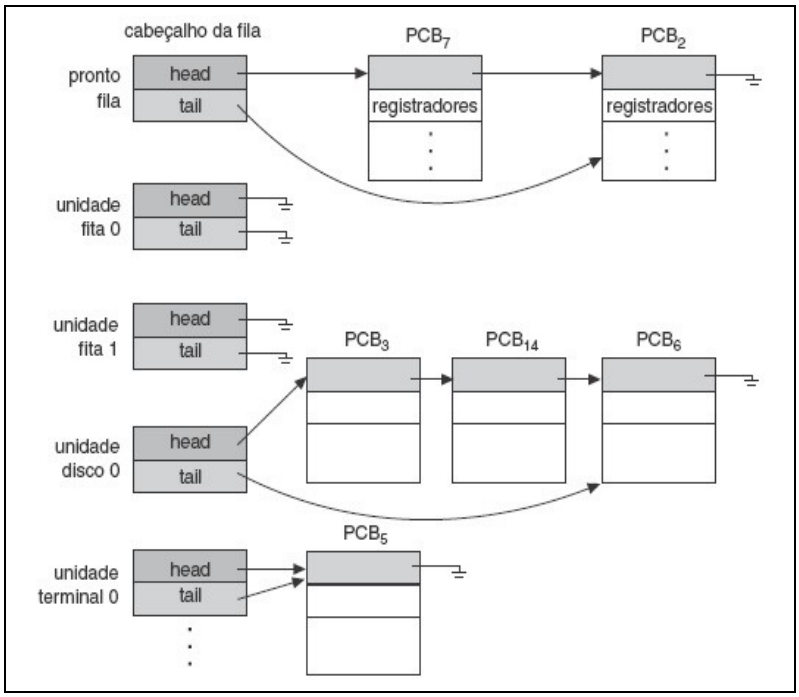

Figura 2 - Fila de prontos e as várias filas de processos bloqueados (Fonte: Silberschatz, 2008)

Quando a operação de E/S termina, o processo retorna ao estado pronto e a PCB do processo requisitante é recolocada na fila de prontos, até que chegue novamente a sua vez de executar. O ato de chavear o processador de um processo para outro é chamado de **troca de contexto**. Por exemplo, suponha que um processo P1 esteja em execução e P2 seja o próximo da fila. O chaveamento nesse caso envolve o **salvamento do contexto** de P1 e a **restauração do contexto** do processo P2. Salvar o contexto significa armazenar algumas informações do PCB que são alteradas durante a execução de um processo, como o conteúdo dos registradores do hardware. Restaurar o contexto é a ação inversa, ou seja, recuperar as informações do PCB do processo a ser executado e ajustá-las nos registradores do hardware. Como dito anteriormente, para impedir que um processo permaneça com o processador indefinidamente, o SO estipula um *quantum* de tempo máximo (duração do *time-slice*) para cada processo ser executado numa rodada, após o qual o processador será chaveado para outro processo. É possível que o processador seja chaveado antes do término do *time-slice*, por exemplo, se o processo em execução requisitar uma operação de E/S que requer algum tempo para ser realizada.

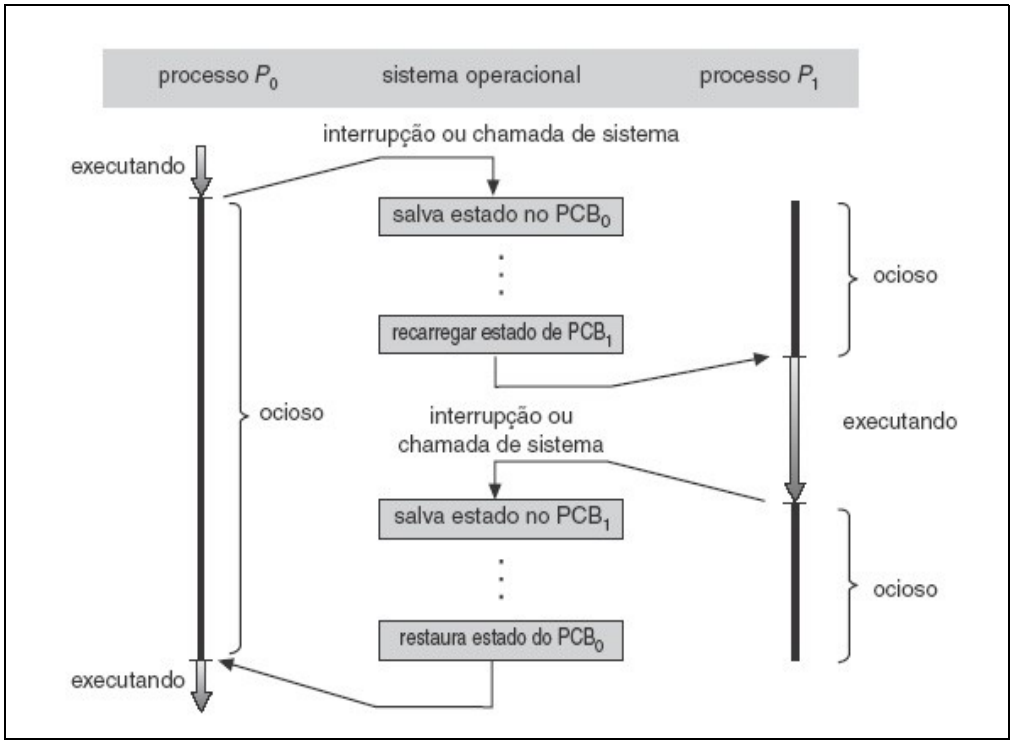

Figura 3 – Esquematização de uma troca de contexto envolvendo dois processos P0 e P1. (Fonte: Silberschatz, 2008)

Um mecanismo muito importante para os SOs são as interrupções. Uma **interrupção** é um sinal de hardware recebido pelo processador, indicando que um determinado evento ocorreu. Esse mecanismo é útil, de maneira geral, para evitar que o SO tenha que ficar testando periodicamente o estado de cada dispositivo em operação. Por exemplo, o dispositivo temporizador pode enviar tais sinais ao processador, indicando o término do *time-slice* de um processo. Os controladores de dispositivos também podem gerar interrupções, indicando que uma operação de E/S foi concluída e que o SO pode mudar o estado do processo que aguardava essa operação de **bloqueado** para **pronto**.

Quando o **processador** recebe um sinal de interrupção, ele é imediatamente forçado a desviar o fluxo de execução para uma **sub-rotina de tratamento de interrupção** (*interrupt service routine*), específica daquela interrupção. Isso é feito automaticamente pelo processador, independentemente de qual trecho de código estiver sendo executado, seja ele um código do SO ou de um processo de usuário. Para tanto, após concluir a execução da instrução corrente, o **processador salva** numa pilha o conteúdo do ponteiro de instruções e, comumente, os *flags* que indicam o *status* da última operação. O número correspondente à interrupção ocorrida é usado, então, como um índice para um **vetor de endereços** em memória, que contém os endereços das **rotinas** de tratamento de interrupções. Esses endereços são **ajustados** pelo **SO**, mas são consultados automaticamente pelo processador, quando as interrupções ocorrem. Deste modo, o endereço apropriado é copiado do vetor para o registrador que indica a próxima instrução a executar. Essas instruções correspondem às rotinas de tratamento definidas pelo SO para cada caso específico. Ou seja, na ocorrência de uma interrupção, uma rotina apropriada do SO é executada automaticamente. O chaveamento de processos, por exemplo, pode ser programado dentro da sub-rotina que trata da interrupção do relógio (término do *time-slice*).

Um efeito importante da ocorrência de interrupções, é que elas permitem que o SO retome o controle do processador depois de tê-lo atribuído à execução de um processo. Isso também vale

no caso em que há diversos processadores. Sempre que há uma interrupção, uma rotina do SO é ativada. Isso vale para os diferentes tipos de interrupção: interrupções **externas**, geradas por controladores de dispositivos, ***traps***, que indicam que uma situação anormal ocorreu durante a execução de uma instrução, como divisão por zero ou acesso inválido à memória, e a **instrução** de interrupção (*int*), usada explicitamente para chamada de serviços do SO. Alguns processadores atuais da plataforma PC também possuem instruções específicas para que os programas realizem chamadas de sistema, como **sysenter** e **syscall**, presentes em processadores Intel e AMD, respectivamente.

Frequentemente, processos que executam em um mesmo computador precisam se comunicar. Por exemplo, se um processo estiver escrevendo um registro de dados no disco, então, para evitar inconsistências, outro processo que necessite escrever no mesmo registro terá de esperar. Se ambos escreverem sobre o mesmo registro simultaneamente, haverá um conflito e os dados poderão ser corrompidos. O mesmo pode ocorrer se dois processos tentarem, simultaneamente, escrever sobre uma mesma posição da memória ou acessar o mesmo dispositivo. Além disso, processo podem cooperar na realização de atividades, dividindo-se para usar mais processadores. Por isso, os SOs tiveram de implementar **mecanismos de comunicação interprocessos** (IPC – *interprocess communication*). SOs modernos normalmente oferecem diversos mecanismos de IPC, tais como os **semáforos**, **filas de mensagens**, e áreas de **memória compartilhada**. Outros mecanismos de comunicação entre processos incluem ***pipes***, ***sockets***, além de **monitores,** ***spinlocks***, **eventos** e **sinais**.

Processos no UNIX

No sistema operacional UNIX, todos os processos têm um conjunto de endereços de memória, que é chamado de **espaço de endereçamento virtual**. Mais detalhes sobre o gerenciamento das áreas de memória serão vistos posteriormente no curso. O ***kernel*** (núcleo) do UNIX mantém o PCB de cada processo em uma região protegida de memória, de forma que os demais processos (processos de usuários) não podem acessar essa área. Entre outras coisas, o PCB do UNIX tipicamente armazena informações tais como o conteúdo dos registradores dos processos, o identificador do processo (PID), o contador do programa, a pilha do sistema, entre outros. Uma relação de todos os processos existentes é comumente mantida numa **tabela de processos**.

Quando um processo necessita interagir com o sistema operacional, ele invoca uma **chamada ao sistema** (*system call*), que é uma espécie de sub-rotina de serviço executada pelo próprio SO. Isso ocorre porque, por razões de segurança, um processo não pode acessar os dispositivos diretamente e nem manipular áreas de memória fora do seu espaço de endereçamento. Além disso, há diversos serviços que o SO pode prover aos processos, como criar outros processos ou oferecer mecanismos para comunicação e sincronização entre esses processos. Por exemplo, no UNIX, um processo pode gerar outro processo através a chamada *fork*, que cria uma cópia do processo que realiza essa chamada. O novo processo criado, chamado de processo-filho, recebe uma cópia de tudo o que processo-pai possui, como por exemplo, as variáveis, as informações dos arquivos abertos, o contador de programas e os demais registradores, etc.

Também há chamadas de sistema para enviar sinais a outros processos, para abrir e fechar arquivos, para ler ou gravar em arquivos, para realizar comunicações com outros processos locais ou remotos, através da rede, entre outras tantas operações.

Processos possuem **prioridades** diferenciadas de execução. No UNIX há um valor de prioridade associado aos processos, chamado **nível de *nice***. Essas prioridades são indicadas por números inteiros que variam de –20 a 19, sendo que um valor numérico de prioridade mais **baixo** indica uma prioridade de escalonamento mais **alta**. Tanto o usuário (com restrições) quanto o próprio sistema podem alterar a prioridade de um processo. De maneira geral, processos são

inciados com valor de nice 0, o que corresponde à prioridade normal de processos interativos. Para processos que requerem bastante processamento mas têm pouca ou nenhuma interatividade com os usuários, é possível aumentar seus valores de nice, de forma que eles tenham menor prioridade que outros processos interativos.

Outras informações associadas aos processos em ambientes Unix incluem um vetor de arquivos abertos, estruturas para comunicação entre processos (IPC) e para tratamento de sinais.

O trecho de programa na figura **4** ilustra a criação de um processo-filho por um programa em linguagem C.

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

int main(int argc, char **argv)
{
      pid_t result;

      /* cria outro processo */
      result = fork();
      if (result < 0) { /* ocorreu erro na execução do Fork */
            fprintf(stderr, "Falha no Fork.\n");
            exit(-1);
      }
      if (result == 0) {            /* processo filho */
            execlp("/bin/ls","ls",NULL);
            exit(1);
      } else { /* processo pai */
            wait(NULL); /* pai espera o término do filho */
            printf("Filho terminou.\n");
            exit(0);
      }
}
```

Figura 4 – Programa em C que ilustra o uso da chamada fork.

Processos leves: *threads*

A esta altura, você já deve ter percebido que processos, seus estados e a troca de contexto são instrumentos-chave para implementar a multiprogramação e o *time-sharing*, discutidos no Ciclo anterior. Imagine que um sistema operacional concede um *quantum* de tempo de 16 ms (milissegundos – milésimo de segundo) a cada processo, e que uma troca de contexto leva 4 milissegundos (ou seja, 2 ms para a fazer o salvamento do contexto do processo que perde o processador, mais 2 ms para a recuperação de contexto do próximo a executar). Esse sistema desperdiça nada menos que 20% do tempo do processador fazendo trocas de contexto. Assim, esse tempo gasto em trocas de contexto pode se tornar um problema.

Suponha agora que esse mesmo sistema executa um programa bem conhecido nos dias de hoje, um servidor *web*, que hospeda páginas (por exemplo o servidor Moodle da UAB) e que pode receber dezenas ou centenas de requisições de clientes (*browsers* de alunos, professores, tutores) por minuto, solicitando uma página HTML, uma imagem, um arquivo em PDF, etc. Para atender a essas requisições, o processo servidor *web* cria dezenas (ou centenas) de processos-filhos, um para atender a cada cliente. Você já se perguntou quanto tempo o SO gastará para criar todos esses processos? Como o número de usuários solicitando páginas simultaneamente pode ser grande, esta pode se tornar uma operação muito custosa e lenta. A criação de processos-filhos envolve a criação

de um novo PCB, a alocação de áreas na memória, bem como a criação e a cópia de várias estruturas de dados do processo-pai para o processo-filho. Portanto, o tempo gasto na criação de um grande número de processos-filhos pode se tornar mais um problema.

Outro aspecto importante a observar aqui é o fato de todas essas instâncias da aplicação executam o mesmo código, ou uma mesma parte dele. Como código não é comumente alterável em tempo de execução, parece razoável pensar que o compartilhamento de áreas de memória poderia levar a economias de espaço e de tempo para alocação. Além disso, é comum que essas instâncias também compartilhem informações, como variáveis e outras estruturas de dados, como arquivos abertos e mecanismos de comunicação.

Para resolver os problemas de lentidão na criação de processos e também na troca de contexto, além de tornar uso de memória mais eficiente e simplificar o compartilhamento de dados, foi desenvolvido o conceito de processos "leves", com menos informações de contexto, que são chamados de ***threads,*** ou **linhas de execução**. Com um contexto mais reduzido, a criação de *threads* e a troca de contexto entre elas pode ser muito mais rápida. Com isso, o processador terá mais tempo para executar o trabalho realmente útil das aplicações. Aplicações que executam múltiplas atividades simultaneamente poderão executar mais rápido se utilizarem *threads* em vez de processos. Além disso, o compartilhamento de memória e a comunicação entre as *trheads* de um processo é mais simples. Do ponto de vista dos programas, essa facilidade de compartilhamento de dados entre diferentes linhas de execução desse programa talvez seja o maior incentivo para o uso de *threads*.

Pensando no desenvolvimento de um programa de aplicação, é comum que existam várias atividades que podem ser executadas simultaneamente. Para um programa que realiza comunicação em rede, por exemplo, é possível receber pedidos que chegam pela interface de comunicação, ao mesmo tempo em que os dados de um arquivo são lidos do disco para atender um pedido anterior. Já um programa navegador WWW pode estar buscando diferentes aquivos de servidores remotos enquanto processa arquivos já recebidos, formatando-os e adequando-os para exibição na tela. Todas essas atividades relativamente independentes podem ser executadas em paralelo, se houver mais de um processadores disponíveis. Além disso, o uso de entidades de execução paralela num programa, mesmo que o computador em uso possua apenas um processador, pode ser interessante para evitar que esse programa fique completamente bloqueado à espera de uma informação da rede ou do disco, enquanto há outras atividades a processar.

Processos e *threads* podem ser usados no desenvolvimento de programas que exploram o paralelismo de execução. Para serem gerenciados pelo Sistema Operacional, contudo, tanto processos quanto *threads* precisam ter informações de contexto, mantidas em PCBs [3]. Processos são independentes e podem comunicar-se, usando mecanismos providos pelo SO, ou áreas de memória compartilhadas explicitamente. Quando o modelo de *threads* é usado, contudo, criam-se *threads* **associadas a um processo**. Assim, as *threads* de um mesmo processo **compartilham** sua área de memória onde está armazenado o código a executar, e grande parte da área de dados. Embora o SO precise manter um PCB para cada *thread*, com informações do contexto de execução de cada uma delas, várias informações contidas no PCB do processo ao qual as *threads* estão associadas são compartilhadas entre elas.

Nos SOs atuais, é comum que cada processo tenha ao menos uma *thread.* Quando se desenvolve programas na linguagem C, por exemplo, há uma *thread* associada à função *main()*.

Conforme podemos ver na Figura 5, *threads* pertencem a um processo e cada processo possui pelo menos uma *thread*. *Threads* de um mesmo processo compartilham parte do seu contexto (contexto do processo), e possuem também o seu contexto individual (bem menor). Assim, a troca de contexto entre *threads* de um mesmo processo é mais rápida que a troca de

[3]No Linux, a estrutura de dados usada para manter informações dos processos é chamada *task_struct*.

contexto entre *threads* de processos distintos. Por esse motivo, *threads* também são chamados de processos leves (*lighweight process* – LWP).

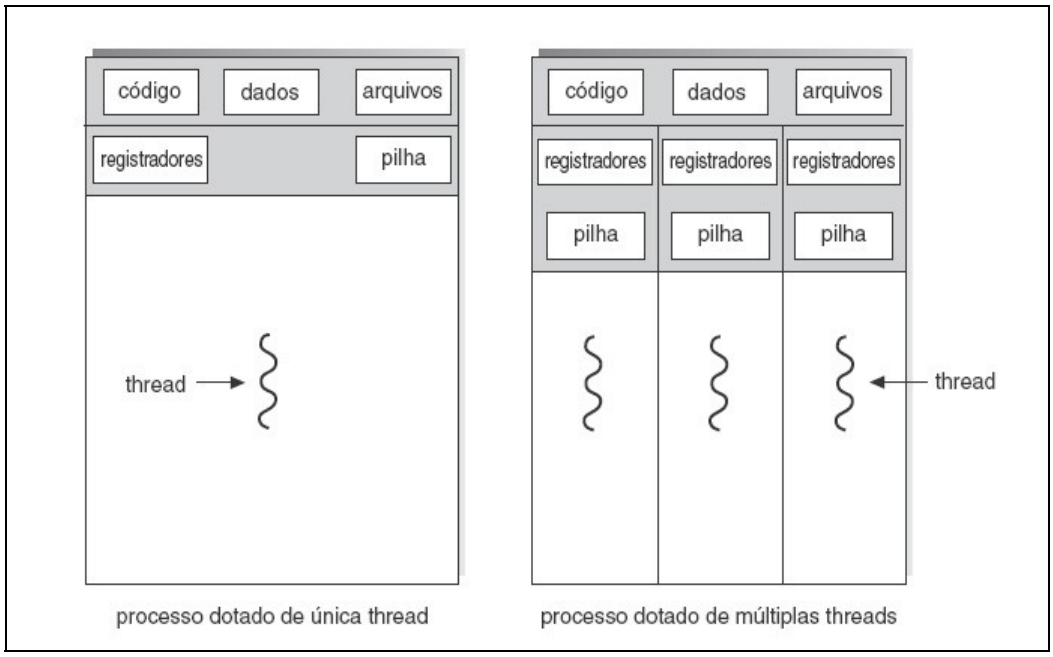

Figura 5 – Contexto de processos e de threads (extraído de Silberschatz 2008).

Atualmente, *threads* são indispensáveis para os desenvolvedores de aplicações. Por exemplo, quando abrimos uma página da web, cheia de objetos gráficos, *banners*, imagens que se movimentam, normalmente há múltiplos *threads* executando simultaneamente, seja para baixar esses objetos (arquivos HTML, imagens, animações, etc) ou para exibi-los na tela. Um simples editor de texto normalmente é um programa **multithread**, pois enquanto uma *thread* coleta os dados que digitamos, outra *thread* pode fazer a repaginação do texto, e uma terceira *thread* pode, ainda, fazer a gravação de segurança a cada intervalo de tempo estabelecido. Boa parte das aplicações importantes que utilizamos são *multithreaded*, tais como jogos interativos, servidores em geral e muitos outros.

Muitos sistemas operacionais e linguagens de programação atualmente implementam *threads*. Para o SO, há duas maneiras de se implementar *threads*, sendo uma no próprio núcleo do SO e a outra por meio de bibliotecas no nível dos programas de usuário. Como exemplo de SOs que possuem *threads* nativos, isto é, que possuem *threads* suportados pelo *kernel* podemos citar o Windows NT/XP/2000/Vista e sucessores, e sistemas Unix, como Linux, MAC OS X e sistemas baseados no BSD.

Quando o SO não possui *threads* nativos, é possível fazer uso de bibliotecas de *threads* implementadas no nível dos programas de usuário. Nesse caso, o SO enxerga cada aplicação como um processo com uma única linha de execução. Cabe à biblioteca de *threads*, então, dividir as fatias de tempo alocadas pelo SO ao processo entre as *threads* criadas pelo programador. Uma das desvantagens dessa forma de implementação de *threads*, é a ausência de paralelismo efetivo entre as *threads* de um programa, já que o SO não as vê separadamente, como entidades que podem ser escalonadas para execução. Por outro lado, mesmo esse tipo de implementação de *threads* pode ser vantajoso para a aplicação, uma vez que, enquanto uma *thread* pode ficar bloqueada à espera de alguma condição, outra *thread* dessa aplicação pode continuar ativa, usando as fatias de tempo passadas ao processo pelo SO. Nesse caso, cabe à biblioteca de *threads* implementar os mecanismos de alternância do uso do processador entre as *threads* do processo.

Para facilitar o desenvolvimento de programas que usam *threads*, há um padrão que define chamadas de função para o uso e o gerenciamento de *threads*. Chamado de *pthreads* (POSIX Threads) esse padrão é suportado nativamente por sistemas UNIX e há também implementações que permitem que programas que utilizam funções definidas nesse padrão sejam executadas em ambientes Windows.

Outras linguagens também oferecem mecanismos para criação de *threads* como Threads Java, que implementa *threads* para máquinas virtuais Java, e o Threads Win32 (para sistemas Windows antigos). Além disso, linguagens como C#, Visual C++.Net e VisualBasic.Net também implementam *threads* e permitem que programadores desenvolvam programas *multithread* com facilidade.

Entre as vantagens do uso de *threads*, podemos citar uma melhor responsividade das aplicações, já que é possível usar *threads* para cada uma das atividades concorrentes do processo. O melhor compartilhamento de recursos também é comumente observado, uma vez que *threads* de um mesmo processo compartilham vários recursos e estruturas de dados de controle. O uso de *threads* também permite uma melhor utilização de arquiteturas multiprocessadas (ou multicore).

Programa-exemplo: Threads em Java

A seguir, apresenta-se um exemplo de programa Java que utiliza *threads*.

O programa abaixo manipula um vetor de 10 elementos. Para tanto, 2 *threads* java são criadas, sendo que cada uma soma metade dos elementos do vetor e deposita o seu resultado (soma parcial) em um elemento do vetor v_resultado.

```
/**
 * ThreadExemplo1 - Exemplo para criação de programas multithread.
 * Descricao:
 *    Este programa ilustra a criacao de threads utilizando a
      interface Runnable (há outra forma que cria uma subclasse
      Thread).
 */
import java.io.*;
class WorkerThread implements Runnable {
      private int inicio, fim, meu_numero;
      private double [] V;
      private double [] resultado;

      WorkerThread (int val_inic, int val_fim,  int val_meunum,
            double [] val_V, double [] val_result) {
            this.inicio = val_inic;
            this.fim = val_fim;
            this.meu_numero = val_meunum;
            this.V = val_V;
            this.resultado = val_result;
      }

      public void run() {
            // calcula a soma de uma parte do vetor V[inicio..fim]
            int i;
            double Acum;
            // calcula soma de uma parte do vetor
            Acum = 0;
            for (i=inicio; i<fim; i++)
                  Acum += V[i];
            // deposita o resultado na sua posicao do vetor
            this.resultado[meu_numero] = Acum;
      }
}
```

Figura 6 – Código-fonte da classe WorkerThread.
O construtor recebe e atribui os valores de trabalho da classe.

```
public class ThreadExemplo1
{
   public static void main(String[] args) throws IOException {
      int i, N;
      double total;

      // Tamanho do vetor
      N = 10;
      // inicializa vetor de números
      double []V = new double [N];
      for (i=0; i<N; i++) {
            V[i] = Math.random();
            System.out.println(" V["+i+"]="+V[i]);
      }

      // inicializa vetor de resultados
      double []v_resultado = new double [2];

      long t1;
      t1 = System.currentTimeMillis();
      // criacao dos threads
      Thread [] workerThread = new Thread[2];
      workerThread[0] = new Thread(new WorkerThread(
                  0, 5, 0, V, v_resultado));
      workerThread[1] = new Thread(new WorkerThread(
                  5, 10, 1, V, v_resultado));

      // Inicia execucao dos threads
      workerThread[0].start();
      workerThread[1].start();

      // Faz thread principal aguardar threads filhos
      try {
            workerThread[0].join();
            workerThread[1].join();
      } catch (java.lang.InterruptedException e) {
            System.out.println("Erro no join!");
      }

      long t2 = System.currentTimeMillis() - t1;
      System.out.println("Resultado ");

      total = 0.0;
      for (i=0; i<2; i++)
            total += v_resultado[i];
      System.out.println("Valot total = "+total);
      System.out.println("\nTempo gasto = "+t2+" ms");
   }
}
```

Figura 7 – Código-fonte da classe que implementa o thread principal.

Exercícios resolvidos:

1) Por que não existe um consenso sobre o conceito de processos na literatura técnica (livros sobre o assunto, manuais, revistas) ?
Resposta: Processos são implementados de diferentes formas pelos desenvolvedores dos diversos sistemas operacionais existentes. Portanto, podemos dizer que processo é na verdade uma "abstração", ou seja, uma idéia que utilizamos para entender e para poder explicar o funcionamento dos sistemas.

2) Qual é o mecanismo utilizado para notificar o sistema a respeito de eventos assíncronos importantes, como por exemplo para avisar que expirou o *quantum* de tempo de um processo? Explique resumidamente o funcionamento do mecanismo quando isso acontece.
Resposta: É o mecanismo de interrupção. Interrupções podem ser acionadas por hardware ou por software. Quando o processador recebe uma interrupção vinda do temporizador (circuito do relógio), ele interrompe a execução atual, salva o conteúdo dos registradores, desvia o fluxo de execução para uma rotina que irá salvar todo o contexto do processo que foi interrompido no seu respectivo PCB e, eventualmente, numa área de pilha na memória. Em seguida, o escalonador do SO é ativado para selecionar, entre os prontos, o próximo processo a executar. O contexto desse processo é restaurado no hardware, ajustando os valores dos registradores e dos ponteiros mantidos pelo SO para o processo em execução. A CPU é, então direcionada para a execução do código desse processo.

3) Considere uma aplicação (por exemplo, um servidor web) que cria muitos processos-filhos, que permanecem usando o processador em média por 20 ms. Considere que cada troca de contexto entre processos leva 5 ms, e que para criar um processo-filho leve 100 ms.
   a. Quanto tempo do processador é desperdiçado com trocas de contexto?
   b. Quanto tempo de processador seria desperdiçado supondo que a aplicação crie em média 3 processos-filhos por segundo?
   c. Considere agora a mesma aplicação, porém implementada com múltiplos *threads*. Ou seja, ela cria *threads* em vez de processos-filhos. De quanto seria o desperdício de tempo do processador, se para trocar de contexto entre *threads* de um mesmo processo levasse apenas 1 ms?
   d. Quanto tempo de processador seria desperdiçado, supondo que agora a aplicação cria 3 *threads* por segundo, e que o tempo de criação de *threads* é de apenas 1 ms?

Resposta:
a) Desperdício com troca de contexto = tempo de troca / tempo total
= 5 / (20 + 5) = 5/25= 20%.
b) Desperdício = tempo de criação / tempo total = 3 * 100 / 1000 = 30%.
c) Desperdício = tempo de troca / tempo total = 1 / (20 + 1) = 1/21= 4.76%.
d) Desperdício = tempo de troca / tempo total = 3 * 1 / 1000 = 0.3 %.

3) Um programa de folha de pagamento precisa ler o registro do próximo funcionário para calcular o seu salário. Para isso, o processo faz uma chamada ao sistema (*system call*) passando parâmetros como a posição lógica do bloco do disco a ser lido e o endereço do *buffer* onde os dados lidos devem ser armazenados. Explique resumidamente o que acontecerá com o processo que requisitou a operação de leitura.
Resposta: A chamada ao sistema ativa a execução de uma função do SO. Esse serviço do SO envia uma requisição à controladora de disco, e coloca o processo requisitante em espera até que a transferência solicitada esteja pronta. Para isso, o estado do processo é ajustado para bloqueado e este deixa a fila de prontos e é inserido numa fila de processos que aguardam E/S associada ao dispositivo envolvido. Quando a operação for concluída, uma interrupção será gerada pelo dispositivo controlador de disco. Ao tratar essa interrupção, o processador salva informações sobre a próxima instrução a executar, e desvia o fluxo de execução para o endereço indicado pelo SO para essa interrupção através do vetor de interrupções. A rotina de

tratamento, que é outro serviço do SO, irá acordar o processo que estava bloqueado à espera da transferência recém-concluída, e mover o seu PCB de volta para a fila de prontos para que ele volte a executar.

Leitura complementar

Para complementar a aprendizagem sobre processos recomenda-se a leitura do Capítulo 3 do livro-texto (disponível online para a UAB através do link http://ufscar.bvirtual.com.br). Para complementar a aprendizagem sobre *threads*, indicamos o Capítulo 4 do mesmo livro. Ambos os capítulos possuem exercícios resolvidos e exercícios propostos interessantes.