

## AULA 06 - Threads

### Idéia básica

*Processos*: programa em execução que contém um único fluxo de execução.

*Threads*: programa em execução com múltiplos fluxos de execução.

Em sistemas tradicionais, cada processo tem seu espaço de endereçamento individual e apenas um fluxo de execução (*thread*). No entanto, algumas vezes desejamos ter vários fluxos de execução no mesmo espaço de endereçamento e uma execução paralela.

### Visão Geral

**Thread** ou **linha de execução** é uma das maneiras utilizadas por um processo para dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente, em geral, em arquiteturas multiprocessadas. O suporte à *thread* é fornecido pelo próprio sistema operacional (SO), no caso da Kernel-Level Thread (KLT), ou implementada através de uma biblioteca de uma determinada linguagem, no caso de uma User-Level Thread (ULT) [1].

Um exemplo pode ser dado através de um jogo onde o mesmo pode ser modelado com linhas de execução diferentes, sendo uma para desenho de imagem e outra para áudio. Nesta aplicação, teríamos uma *thread* para tratar rotinas de desenho e outra *thread* para tratar instruções do áudio. Para o usuário, a imagem é desenhada ao mesmo tempo em que o áudio é emitido pelos auto-falantes. No entanto, para sistemas com uma única CPU, cada linha de execução é processada por vez [1].

Assim, as *threads* permitem um paralelismo de *granularidade* mais fina (paralelismo de instruções). Por granularidade mais fina, subentende-se que a unidade de cálculo realizada de maneira concorrente é menor do que a unidade de cálculo associada a um processo (paralelismo de aplicações, por exemplo). A granularidade de uma *thread* pode ser, por exemplo, um método ou um conjunto de instruções dentro de um programa. Para isso ser possível, cada *thread* mantém um contador de programa (**PC – Program Counter**) e um registrador de instruções (**IR- Instruction Register**) para dizer qual instrução é a próxima a ser executada e qual está sendo processada, respectivamente. Além

de registradores que contém suas variáveis atuais de trabalho, pilha com o histórico das instruções executadas e o estado de cada variável. Para que uma thread possa ser executada ela deve pertencer a algum processo, ou seja, um processo deve ser criado anteriormente. Dessa forma, podemos dizer que processos são usados para agrupar recursos (espaço de endereçamento com o código do programa, variáveis globais, etc) enquanto as threads são as entidades escalonadas pela CPU para a execução de tarefas.

**Threads** são conhecidas como *processos leves*. Basicamente, esse atributo se deve ao menor tempo gasto em atividades de criação e escalonamento de threads, se comparadas aos processos. O compartilhamento de memória entre as threads maximiza o uso dos espaços de endereçamento e torna mais eficiente o uso destes dispositivos.

Como todas as threads tem exatamente o mesmo espaço de endereçamento, eles também compartilham as mesmas variáveis globais. Assim, como cada thread pode acessar qualquer posição de memória dentro do espaço de endereçamento do processo, é possível a um thread ler, escrever ou até apagar informações usadas por outra thread. Não existe um meio de proteção para isso, fica sob responsabilidade do usuário este cuidado já que normalmente criamos threads para cooperar e não competir. O escalonamento entre threads acontece da mesma forma que entre processos.

Portanto, cada thread tem o mesmo contexto de software e compartilha o mesmo espaço de memória de um mesmo processo pai. Contudo, o contexto de hardware de cada fluxo de execução é diferente. Conseqüentemente, o tempo “perdido” com o escalonamento das threads é muito menor do que o escalonamento de processos. Além disso, não há acesso protegido a memória nativamente (a implementação é de responsabilidade do programador) devido ao compartilhamento do espaço de memória [1].

Uma das vantagens do uso de threads está no fato do processo poder ser dividido em mais de uma linha de execução. Quando uma thread está esperando determinado dispositivo de I/O ou qualquer outro recurso do sistema, o processo como um todo não precisa ser bloqueado, pois quando uma thread entra no estado de bloqueio uma outra *thread* do mesmo processo está aguardando na fila de prontos para dar continuidade a execução do programa [1].

Normalmente, criamos um processo com um único thread e a partir deste cria-se novas threads através de comandos do tipo *thread\_create( )*. Geralmente passamos como parâmetro um nome de um procedimento para informar o que a thread deve executar. Como resultado do comando temos um

identificador/nome para a thread criada. Ao finalizar seu trabalho a thread pode chamar uma função do tipo *thread\_exit* e desaparecer para não ser mais escalonável.

Assim como os processos, as *threads* possuem estados durante o ciclo de vida. Os estados atingidos são os mesmos discutidos em processos. Uma tabela de *threads* deve então ser mantida para armazenar informações individuais de cada fluxo de execução. Essa tabela é denominada TCB e contém:

1. o endereço da pilha;
2. o contador de programa;
3. registrador de instruções
4. registradores de dados, endereços, flags;
5. endereços das *threads* filhas;
6. estado de execução.

Resta para o processo, como um todo, informações do tipo endereço da área de trabalho, variáveis globais, apontadores para informações de arquivos abertos, endereços de processo filhos, informações sobre *timers*, sinais, semáforos e de contabilização.

*Threads* podem comunicar-se através das variáveis globais do processo que as criou. A utilização destas variáveis pode ser controlada através de primitivas de sincronização (monitores, semáforos, ou construções similares). Primitivas existem para bloqueio do processo que tenta obter acesso a uma área da memória que está correntemente sendo utilizada por outro processo. Primitivas de sinalização de fim de utilização de recurso compartilhado também existem. Estas primitivas podem “acordar” um ou mais processos que estavam bloqueados.

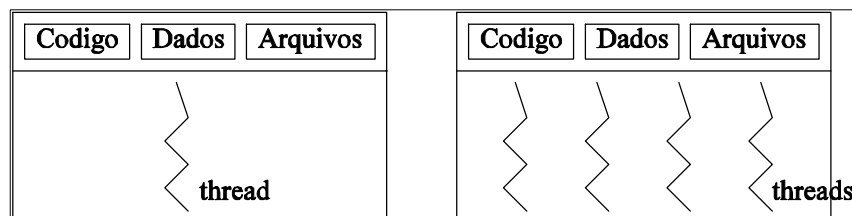


Figura 1. Processos com um e com múltiplos threads.

## **Alguns Benefícios**

1. *Velocidade de Criação das Threads*: as threads são mais fáceis de criar e destruir que os processos pois elas não tem quaisquer recursos associados a elas. Em alguns sistemas criar um thread pode ser cem vezes mais rápido do que criar um processo. No Solaris, criar um processo é aproximadamente 30 vezes mais lento do que criar um thread e a troca de contexto é aproximadamente 5 vezes mais lento.
2. *Capacidade de Resposta*: a utilização do multithreading pode permitir que um programa continue executando e respondendo ao usuário mesmo se parte dele está bloqueada ou executando uma tarefa demorada. Por exemplo, enquanto um navegador Web carrega uma figura ele permite a interação com o usuário.
3. *Compartilhamento de Recursos*: todos os recursos alocados e utilizados pelo processo aos quais pertencem são compartilhados pelos threads.
4. *Economia*: como os threads compartilham recursos dos processos aos quais pertencem, é mais econômico criar e realizar a troca de contexto de threads.
5. *Utilização de Arquiteturas Multiprocessadas*: é possível executar cada uma das threads criadas para um mesmo processo em paralelo (usando processadores diferentes). Isso aumenta bastante os benefícios do esquema multithreading.
6. *Desempenho*: obtido quando há grande quantidade de computação e E/S, os threads permitem que essas atividades se sobreponham e, logo, melhore o desempenho da aplicação.

Em arquiteturas monoprocessadas o paralelismo obtido na execução de threads CPU-Bound é aparente, pois a CPU fica alternando entre cada thread de forma tão rápida que cria a ilusão de paralelismo real. Portanto, neste caso, a execução das threads acontecem de forma *concorrente* assim como os processos. Entretanto, o paralelismo ocorre entre threads do tipo CPU-Bound e I/O-Bound.

**Exemplo** do uso de *threads* em **Java**:

**Exemplo 1:**

Neste exemplo duas threads são criadas com igual prioridade. Cada uma imprime os 100 primeiros inteiros seguidos da frase TERMINOU SANTOS! ou TERMINOU SÃO PAULO! Uma thread se chama Santos, e a outra, São Paulo.

Após cada linha impressa, a thread "dorme" um número aleatório de milissegundos - entre 0 e 399 - para evitar que o laço de impressão termine antes de se esgotar a fatia de tempo da thread (delta t). O nome é dado pelo argumento passado ao construtor da thread. O método getName() da classe Thread retorna a string com o nome:

```
public class ThreadSimples extends Thread {
    public ThreadSimples(String str) {
        super(str);
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 400));
            }
            catch (InterruptedException e) {}
        }
        System.out.println("TERMINOU " + getName()+"!");
    }
}

public class TesteDuasThreads {
    public static void main (String[] args) {
        ThreadSimples santo = new ThreadSimples("SANTOS");
        santo.setPriority(4);
        ThreadSimples sampa =new ThreadSimples("SÃO PAULO");
        sampa.setPriority(6);
        Thread.currentThread().setPriority(1);
        santo.start();
        sampa.start();
        System.out.println("Main terminado!");
    }
}
```

**Exemplo 2:**

```
import java.lang.Thread;

public class SequencialA extends Object {

    public static void main(String args[]) throws Exception {
        int i;
        exemplo ex2 = new exemplo();

        exemplo TTA = new exemplo();
        Thread thA = new Thread(TTA);
        thA.start();

        for (i=0; i<10; i++) {
            ex2.f("normal");
            Thread.sleep(1000);
        }
    }
}

class exemplo implements Runnable {
    private int y;
    public void run() {
        f("run");
    }
    public void f(String x) {
        y++;
        System.out.println(x+" Valor de y: "+y);
    }
}
```

**Pergunta:** O que o programa do exemplo 2 faz?

---

### **Exemplo** do uso de *threads* em UNIX:

```
#include <stdio.h>
#include <pthread.h>
int global;
void *thr_func(void *arg);
int main(void)
{
    pthread_t tid;
    global = 20;
    printf("Thread principal: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_join(tid, NULL);
    ("Thread principal: %d\n", global);
    return 0;
}

void *thr_func(void *arg)
{
    global = 40;
    printf("Novo thread: %d\n", global);
    return NULL;
}
```

### **Exercícios**

1. O que é uma thread e quais as vantagens em sua utilização?
2. Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre threads de um mesmo processo?
3. Dê exemplos do uso de threads no desenvolvimento de aplicativos.
4. Quais os benefícios do uso de threads em ambientes cliente-servidor?
5. Escreva um programa em java que faça uso de duas threads para somar os elementos de um vetor de inteiros e calcular a média de maneira colaborativa. Cada thread deve gerar a média de metade dos elementos. Use variáveis de memória compartilhada e monitores para sincronização das tarefas.

### ***Bibliografia Básica***

SILBERSCHATZ, A. et al. **Sistemas Operacionais: conceitos e aplicações**. Campus, 2001. (Cap. 5).

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. Prentice-Hall, 2ª Edição, 2003. (Cap. 2).

[1] Site: [http://pt.wikipedia.org/wiki/Thread\\_\(ciência\\_da\\_computação\)](http://pt.wikipedia.org/wiki/Thread_(ciência_da_computação))