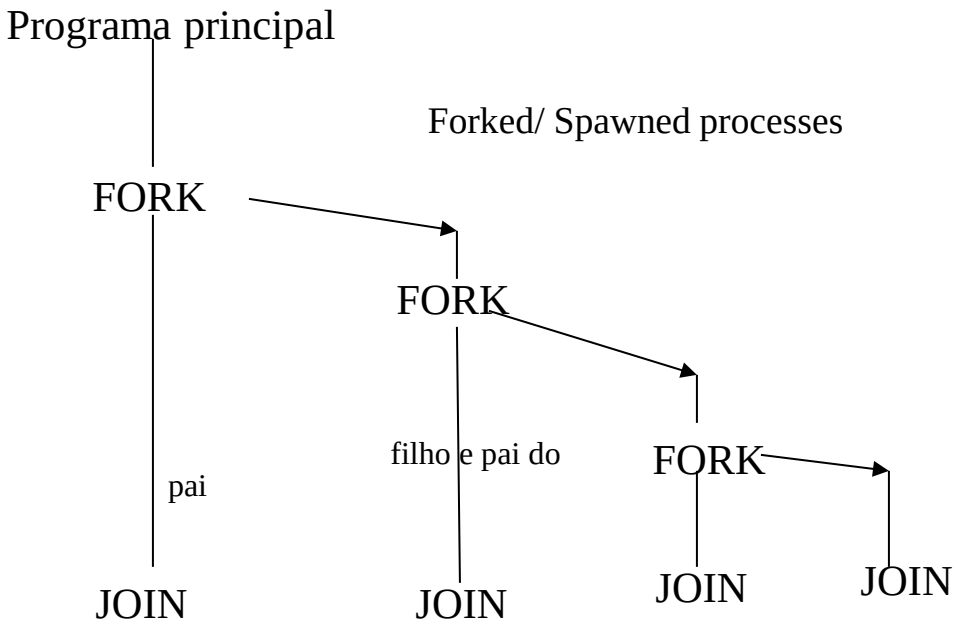
Capitulo 6 Threads

- Processos versus Threads
- Utilidade de Threads
- Implementação de Threads
- Posix Threads
    - Criação
    - Attached and Detached Threads
    - Locks
    - Exemplos

Criação de processos concorrentes utilizando a construção *fork-join*

O processo filho é uma **cópia exata** do processo que chama a rotina *fork*(), sendo a única diferença um identificador de processo (pid) diferente

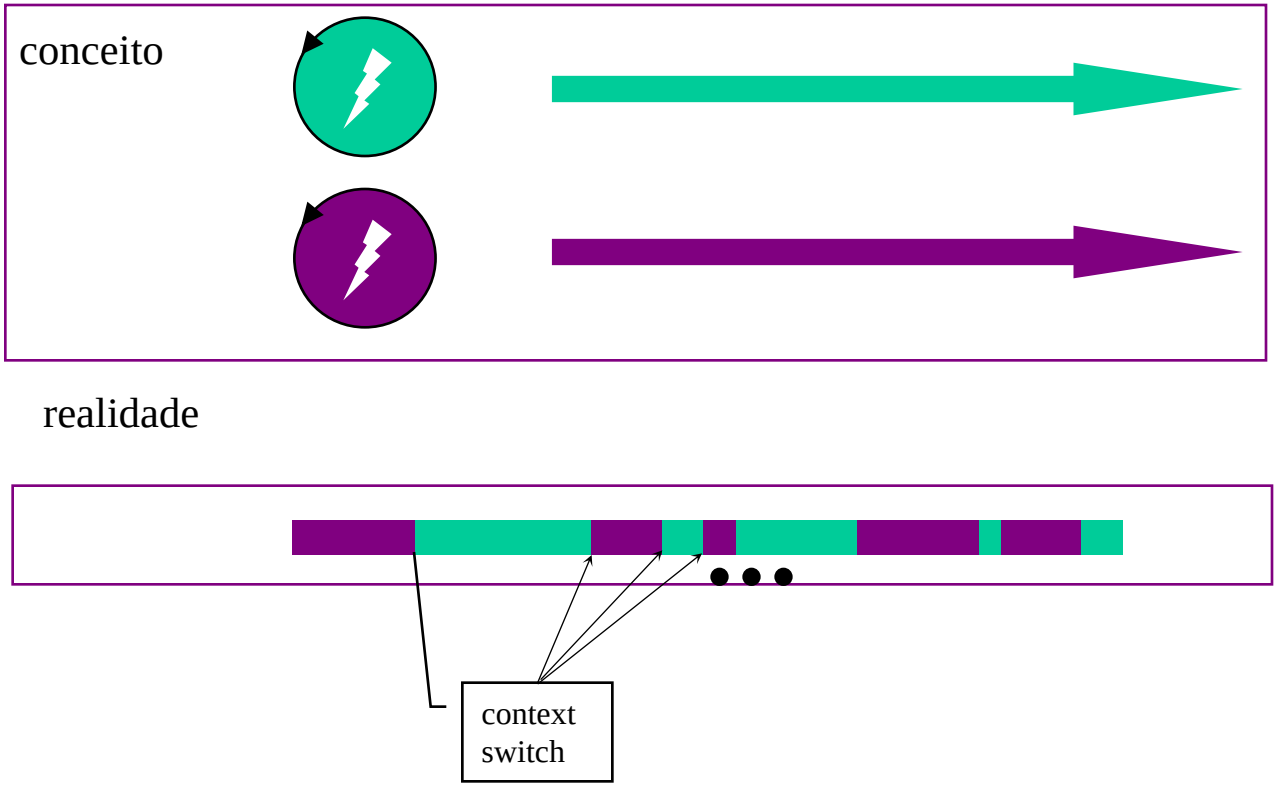
```
pid = fork()
if ( 0 == pid)
      código executado pelo filho
else
      código executado pelo pai
Codigo executado pelos dois
if ( 0 == pid )   ….  exit (n);
else  //join
   wait (&status0); //esperar pelo filho
```



Inter Process Communication

Pipes, Shared Memory, Signals, Sockets etc.

Dois Processos partilhando um CPU



O Modelo de Threads/Light-Weight Processes

Os 2 conceitos necessários para a execução dum programa

"conjunto de recursos" e

"contexto de execução"

podem ser separados:

Conceito dum Processo

O conceito dum Processo pode ser dividido em dois :

1: Um conjunto de recursos necessários para a execução duma programa.

- um espaço de endereçamento (virtual address space) que contêm o **texto** do programa e os seus **dados (globais**)
- uma tabela de ***Descritores de*** **Ficheiros abertos** pelo processo
- informação sobre processos filhos
- código para tratar de sinais (signal handlers)
- informação sobre o próprio (pid, Permissões, Nome do Utilizador, Inventário) etc.

2 : Uma linha ou contexto de execução, chamada "Thread"

- Uma thread tem um progr**ama counter** (pc) que guarde informação sobre a próxima instrução a executar
- **Contexto** (Registadores) – valores das variáveis usados nas instruções atuais
- **Stack** – contem a história de execução com um "frame" para cada procedimento chamado mas não terminado

O Modelo de Threads

Os 2 conceitos necessários para a execução dum programa

"conjunto de recursos" e

"contexto de execução"

podem ser separados:

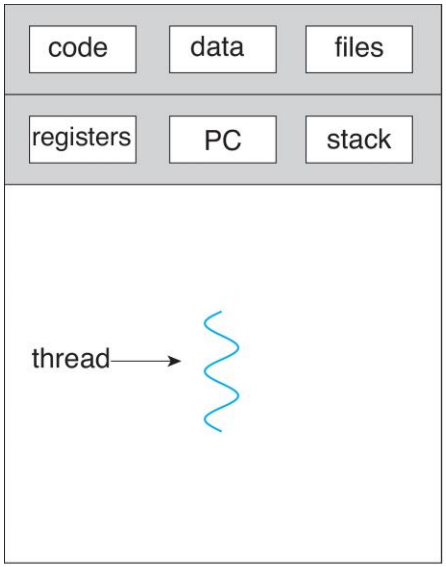
Assim

1. Processos são usados para agrupar recursos.

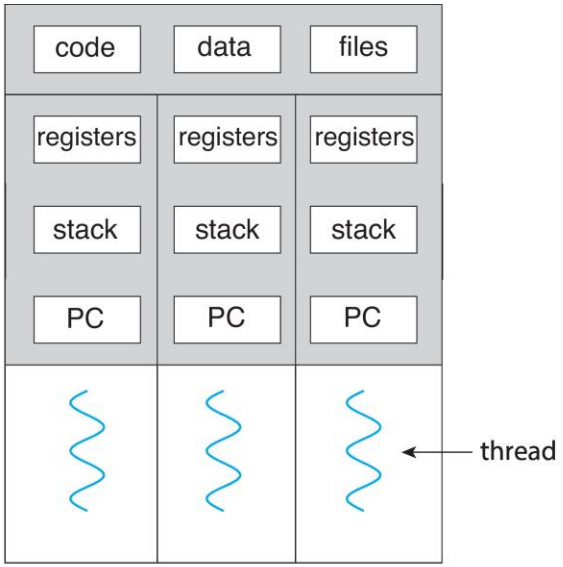
2. Threads são as entidades **escalonadas** para execução no CPU.

Processos versus Threads

Per Thread Items	Per Process Items
• Program Counter • Stack • Register Set • Child Threads • State (ready,blocked ..)	• Address Space • Global Variables • Open Files • Child Processes • Timers and signals • Semaphores • Accounting Information

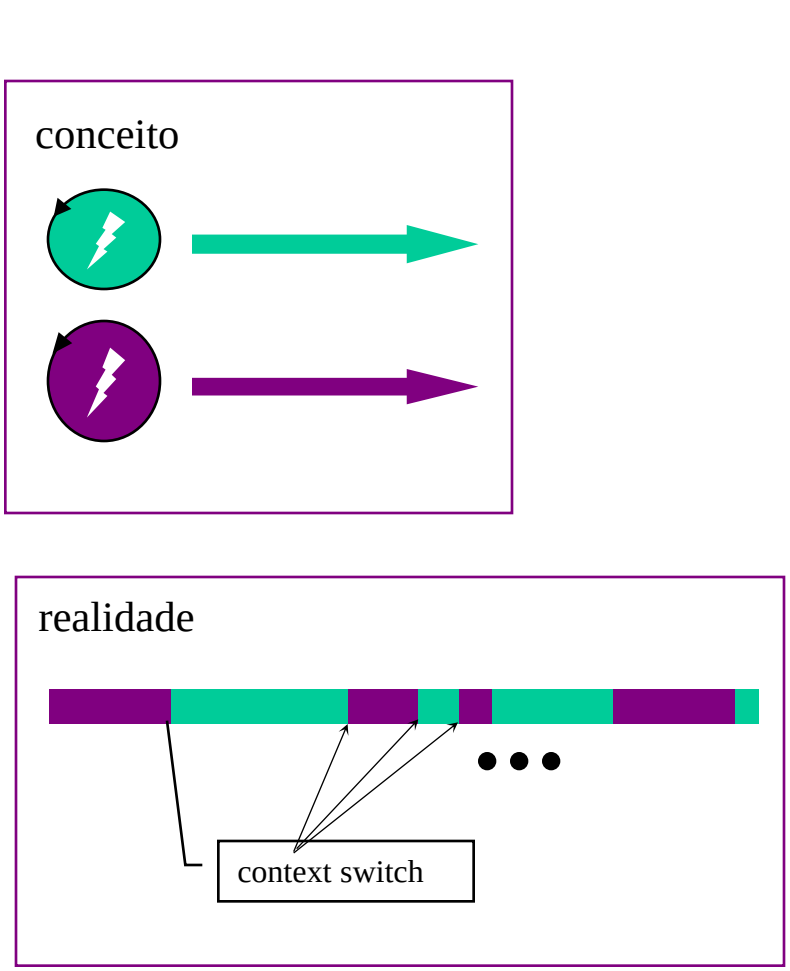


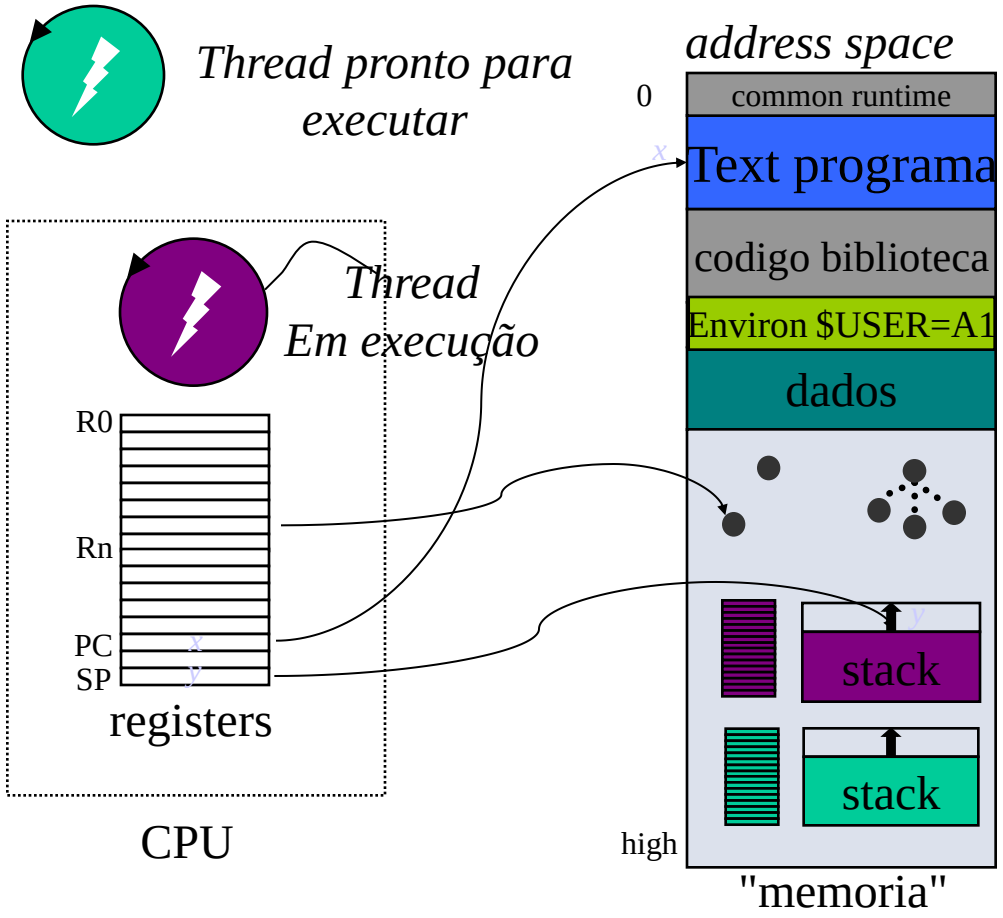
single-threaded process



multithreaded process

Duas Threads partilhando um CPU



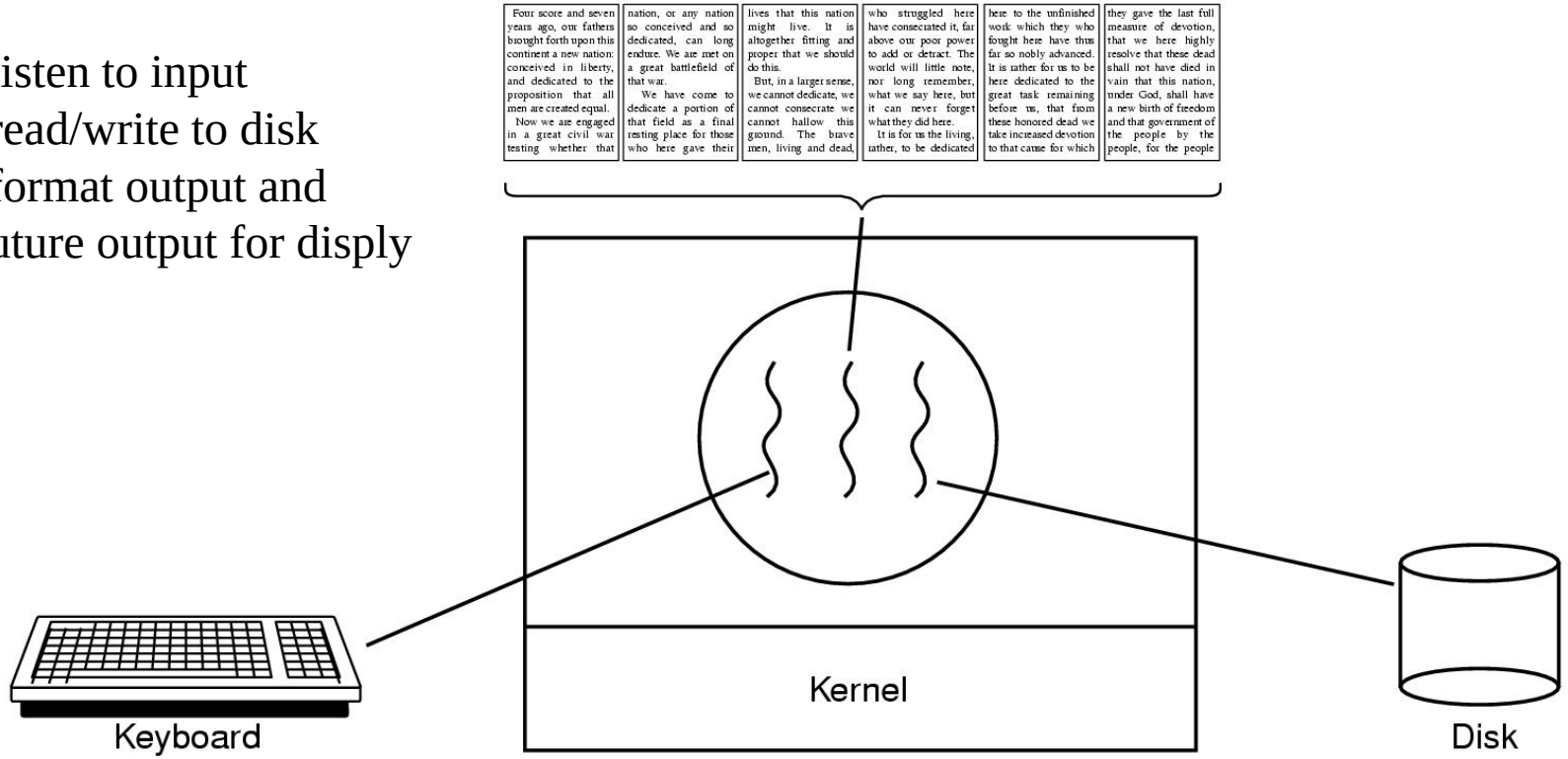


Threads vs Processos

- **Rapidez**
  - A criação e terminação duma thread nova é em geral mais rápida do que a criação e terminação dum processo novo.
  - A comutação de contexto entre 2 threads é mais rápido do que entre 2 processos.
- **Partilha de Recursos**
  - A comunicação entre threads é mais rápida do que a comunicação entre processos usando memoria partilhada e passagem e mensagens
  - threads compartilham tudo como as variáveis globais que podem ser usadas para comunicar informação
- **Criação de User Interfaces (UI)**
  - Varias threads podem tratar de partes diferentes dum UI. Assim não bloquear partes do UI
- **Escalabilidade**
  - Multi-programação usando o modelo de threads é mais simples e mais portável do que multi-programação usando múltiplos processos.
- **Desvantagens :**
  - Fácil de Introduzir Erros – concorrência / condições de corrida
  - Problemas de Segurança

Thread Usage (1) Office

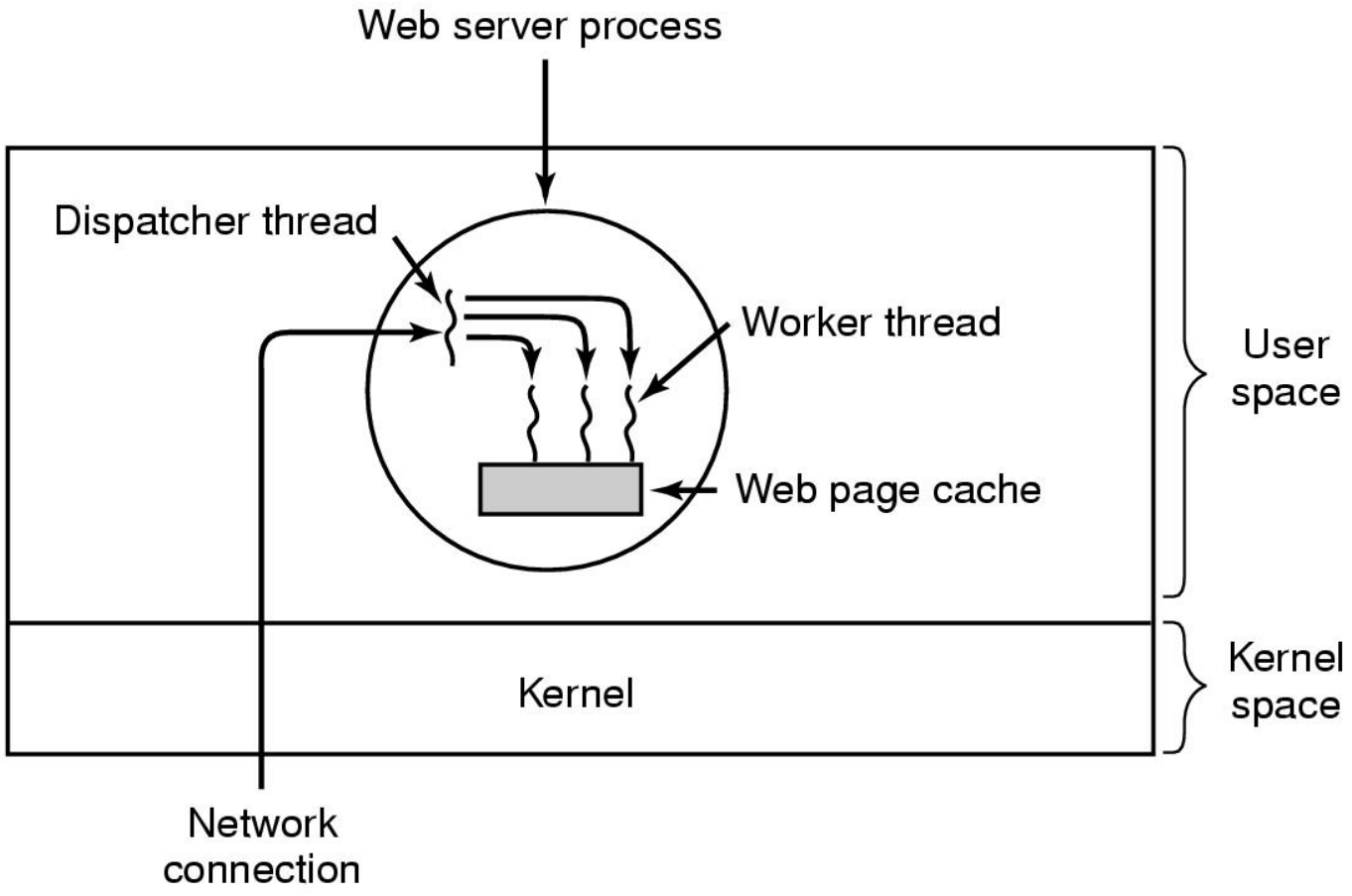
-listen to input
-read/write to disk
-format output and
future output for disply



Processador de Texto com três threads.

Quantos threads tem "microsoft word" no SO Windows ?

Thread Usage (2) : Web Server



A multithreaded Web server

Thread Usage (2)
Esboço do Código para um Web Server

(a) Dispatcher thread

```
Enquanto (VERDADE) {
    Obter_proximo_pedido( &buf );
    Enviar_Worker_Thread( buf );
}
```

(b) Worker thread

```
Enquanto (verdade) {
    Esperar_Trabalho ( &buf );
    if ( ! page_in_cache ( &buf, &page )
        Read_page_from_disk( &buf, &page);
    Send_page_to_client( page );
}
```

Threads múltiplas podem estar a executar simultaneamente para aproveitar arquiteturas como "hyperthreading", "multi-cpu/core"

Pode haver um conjunto de threads **(thread pool)** prontas para executar – evitando assim o tempo de iniciar uma thread.

Compare com a funcionamento do programa "sshd" numa maquina linux (secure shell daemon). No ssh para cada nova ligação através dum programa cliente tipo putty o programa faz "fork"

Thread Usage (3)
Processamento Paralelo
Exemplo: Matrizes

Multiplicação de matriz : A×B=C

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix}=\begin{pmatrix} a.e+b.g & a.f+b.h \\ c.e+d.g & c.f+d.h \end{pmatrix}$$

Repare que cada elemento, (a,b,c,d) (que podem ser blocos) da Matriz C pode ser calculado **independentemente** dos outros, Portanto podem ser facilmente calculados por threads diferentes.

Implementação de Threads

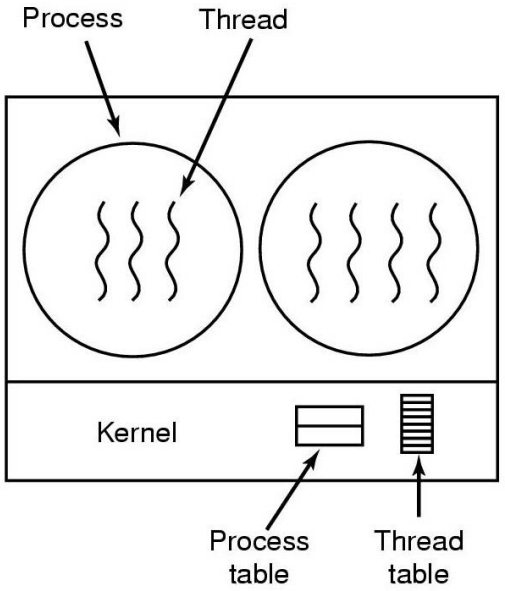
Existem duas abordagens para a implementação de threads.

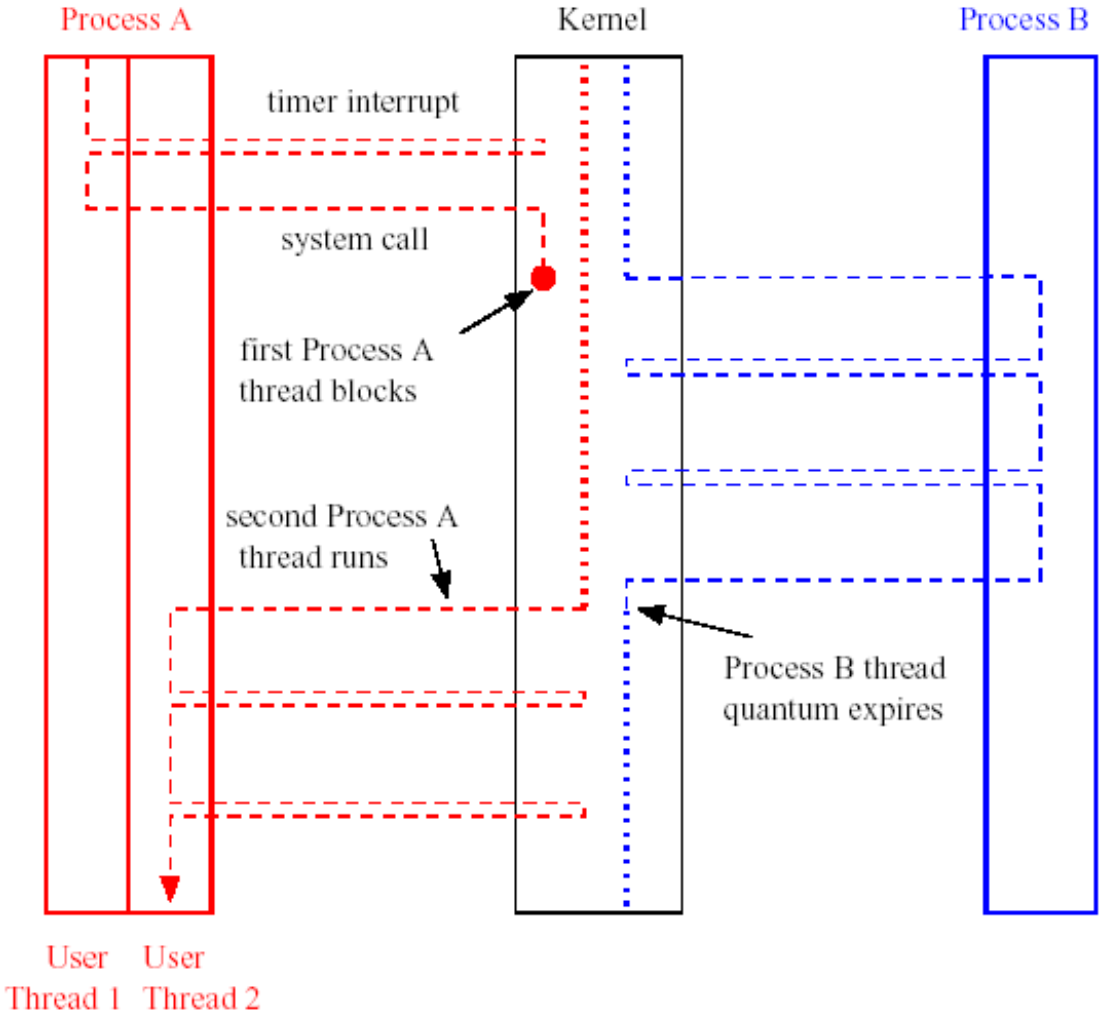
•Kernel-level Threads KLT-- System Calls.

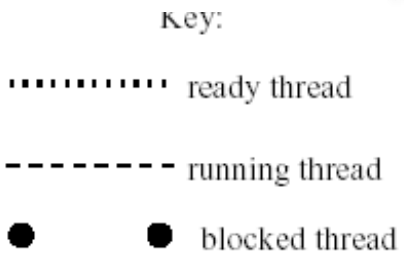
•User-Level Threads ULT -- Thread Libraries.

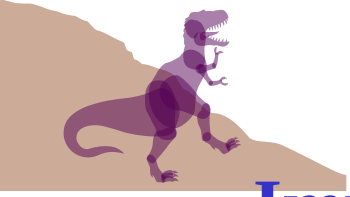
Existem vantagens e desvantagens em ambos os casos.

KLT's


Process
Thread
Kernel
Process table
Thread table


Process A
Kernel
Process B
timer interrupt
system call
first Process A thread blocks
second Process A thread runs
Process B thread quantum expires
User Thread 1
User Thread 2


Key:
ready thread
running thread
blocked thread



Implementação deThreads no Kernel (KLT) Vantagens e Desvantagens

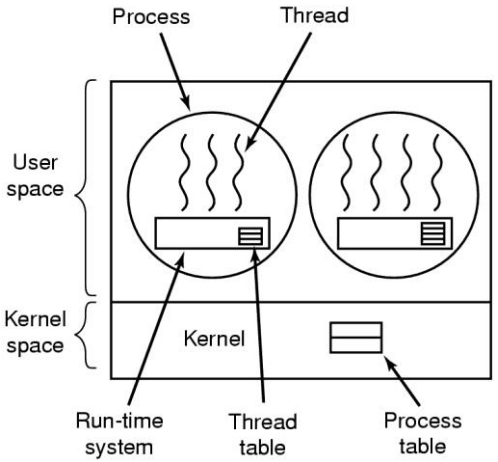
Vantagens

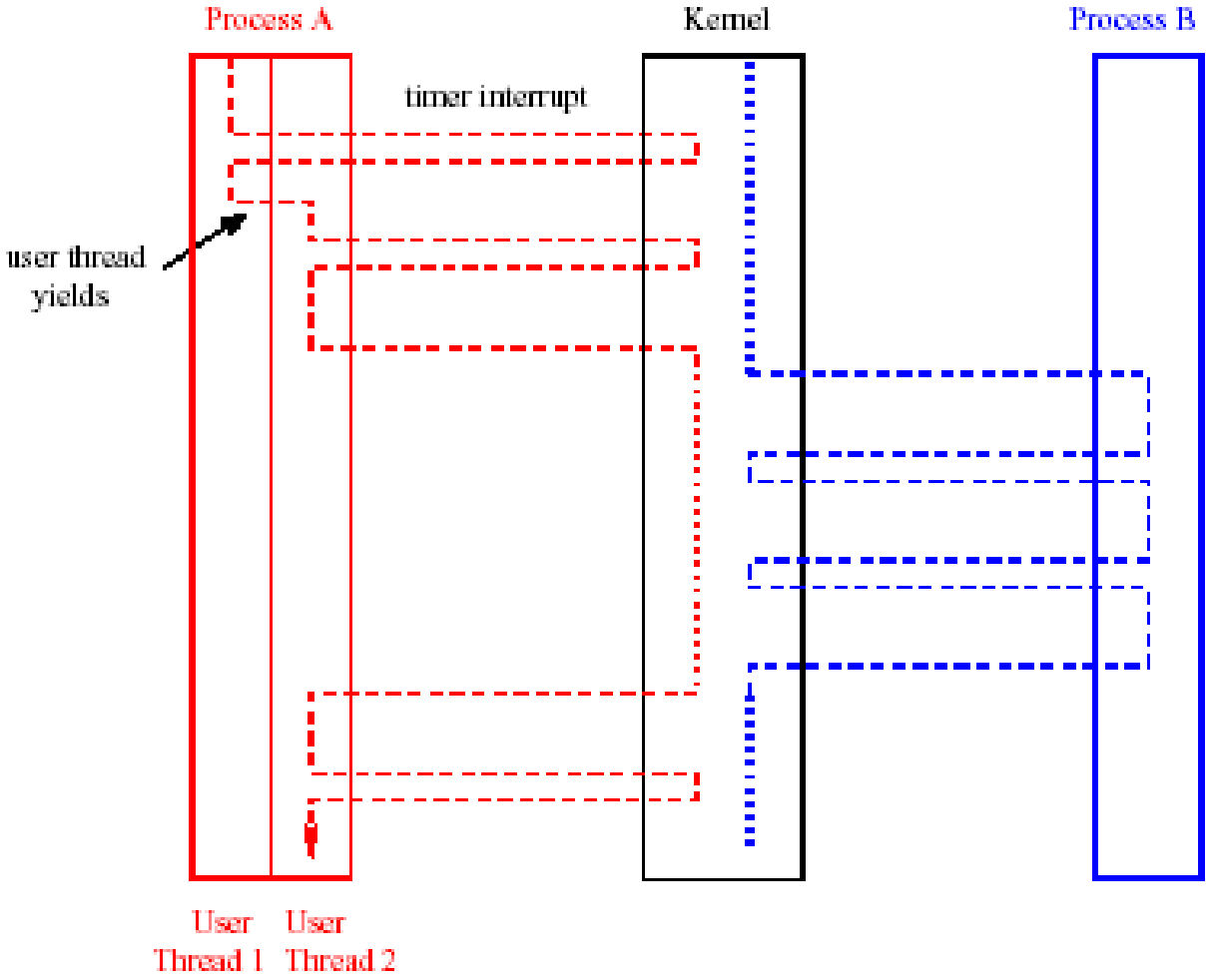
- **O kernel pode simultaneamente escalonar várias threads do mesmo processo em vários processadores (reais ou virtuais)**
- **As rotinas do próprio kernel podem aproveitar threads.**

Desvantagens:

- **A troca entre threads implica ações do kernel e isto tem um custo que pode ser significativo.**

ULT's





Key:

ready thread

running thread

Implementação de Threads no User Space Vantagens e Desvantagens de ULT (User Level Threads)

Vantagens

- A troca de Threads não envolve o kernel
  - Não há o custo adicional de execução do kernel
  - O OS não precisa de oferecer apoio para threads – portanto é mais simples.
- Escalonamento pode ser **especifico** para uma aplicação
  - Uma biblioteca pode oferecer vários métodos de escalonamento portanto uma aplicação poderá escolher o algoritmo melhor para ele.
- ULTs podem executar em qualquer SO
  - As bibliotecas de código são portáveis

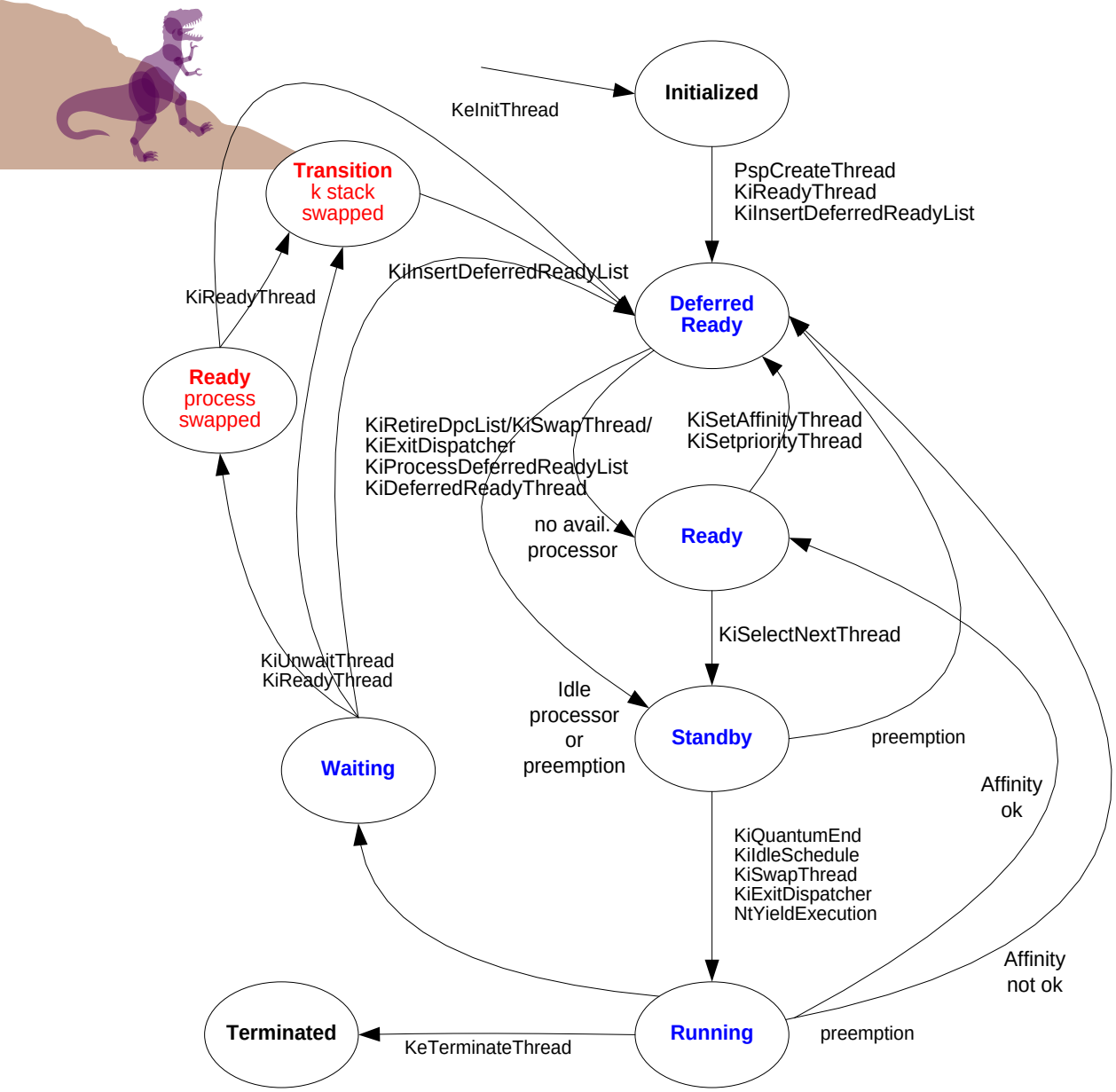
Desvantagens

- Muitas das chamadas ao sistema são "bloqueantes" e o kernel bloqueia processos – neste caso todos as threads dum processo podem ser bloqueados.
- O kernel vai atribuir o processo a apenas um CPU portanto duas threads dentro do mesmo processo não podem executar simultaneamente numa arquitetura com múltiplas processadores

ULT Exemplo : JAVA 21

Java 21 Doc

- **A JAVA platform thread is implemented as a thin wrapper around an operating system (OS) thread. A platform thread runs Java code on its underlying OS thread, and the platform thread captures its OS thread for the platform thread's entire lifetime. Consequently, the number of available platform threads is limited to the number of OS threads.**
- **Like a platform thread, A Java virtual thread is also an instance of java.lang.Thread. However, a virtual thread isn't tied to a specific OS thread. A virtual thread still runs code on an OS thread. However, when code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads.**



Thread scheduling states on Windows

Pthreads

Linux natively uses the clone() System call. However …

The POSIX 1003.1-2001 standard defines an application programming interface (API) for writing portable multithreaded applications for creating, destroying, synchronizing and scheduling threads.

This interface is known more commonly as *pthreads* and is common across many operating systems (LINUX Solaris, Mac OS X)

Programação (Linux)

- Protótipos : /usr/include/pthread.h
- Library : /usr/lib/x86_64-linux-gnu/libpthread.a OU .so

Manual.

- man pthread_create

For more information:

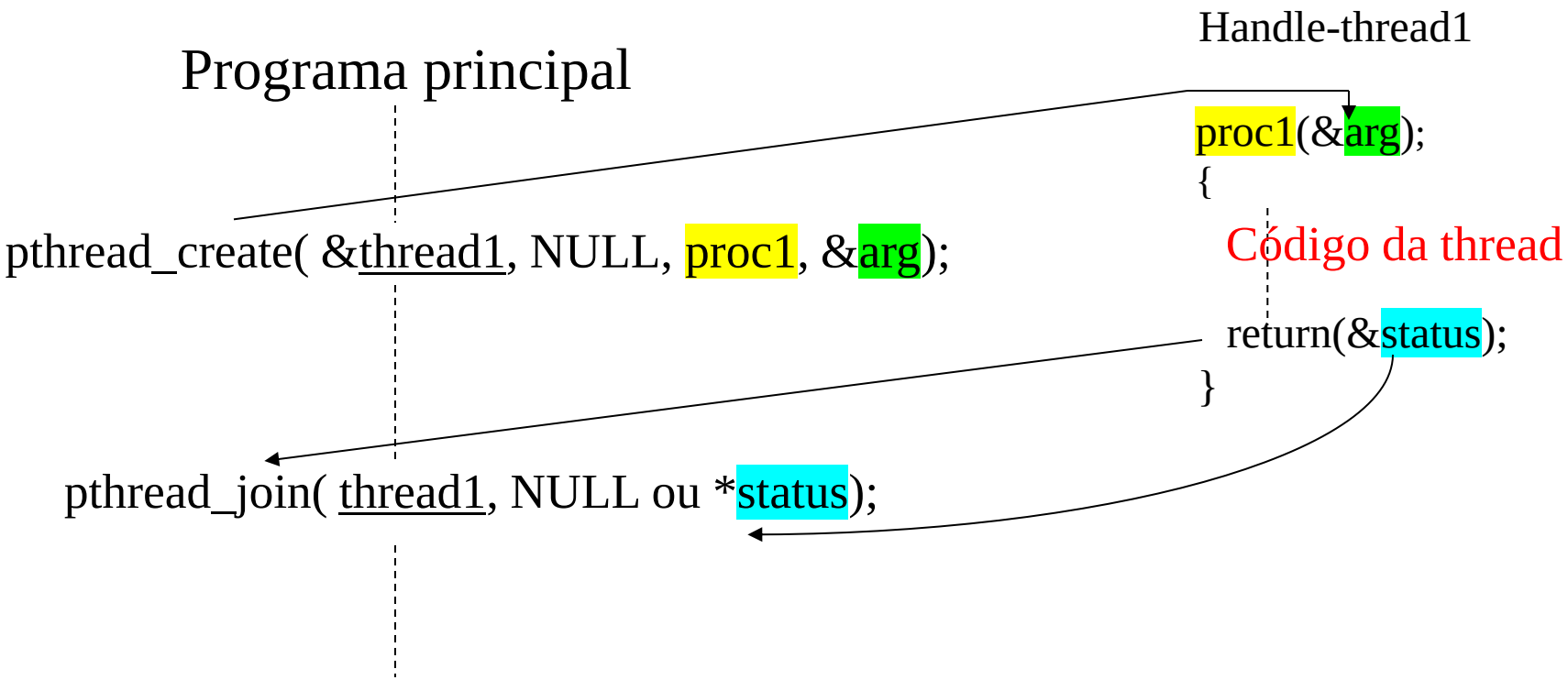
•See the wiki page for the Native Posix Thread Library

•Getting Started With POSIX Threads Tutorial, Tom Wagner

•Posix threads for windows see https://sourceforge.net/projects/pthreads4w/

Criação de Threads

https://linux.die.net/man/3/pthread_create



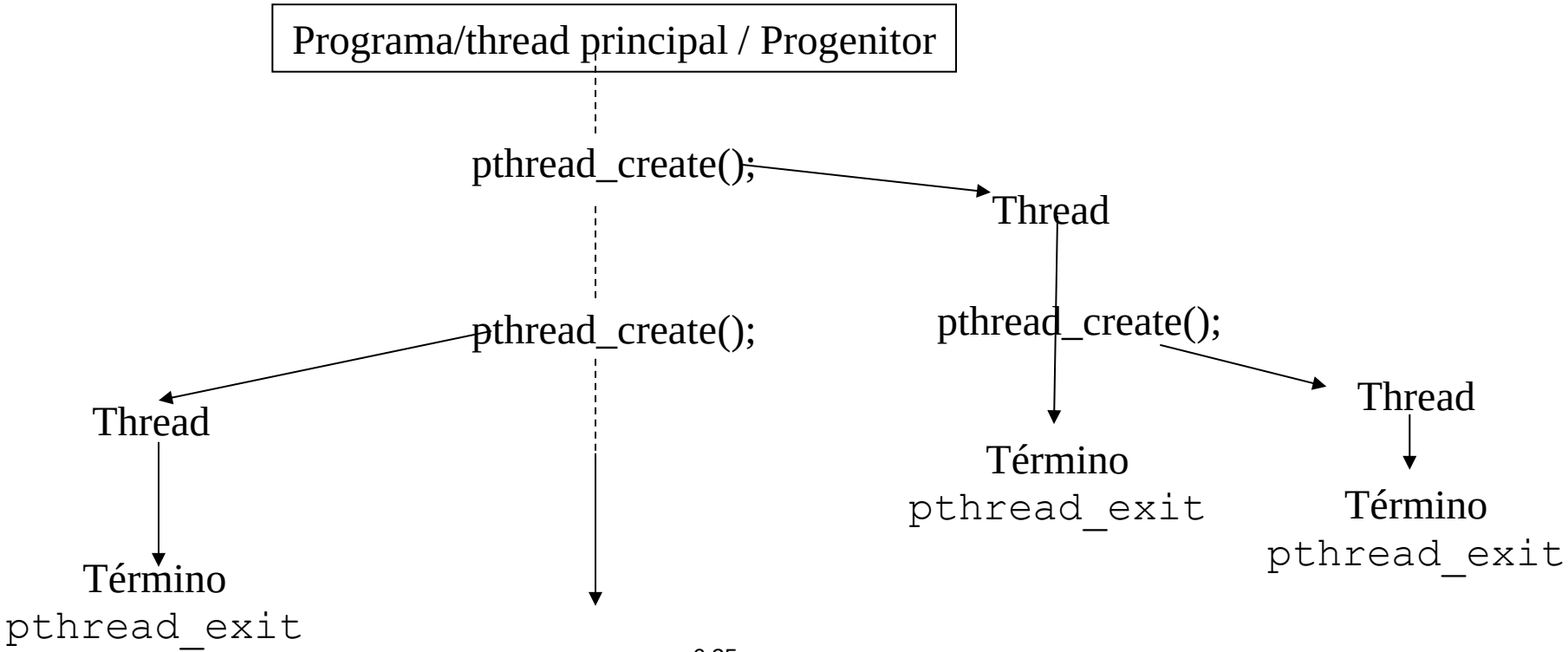
Pthread Join

- ***pthread_join*() espera pelo términação duma thread específica**

```
...thread principal

for (i = 0; i < n; i++)
        pthread_create(&thread[i], NULL, (void *) worker, (void *) &arg);

...thread mestre

for (i = 0; i < n; i++)
        pthread_join(thread[i], NULL);
```

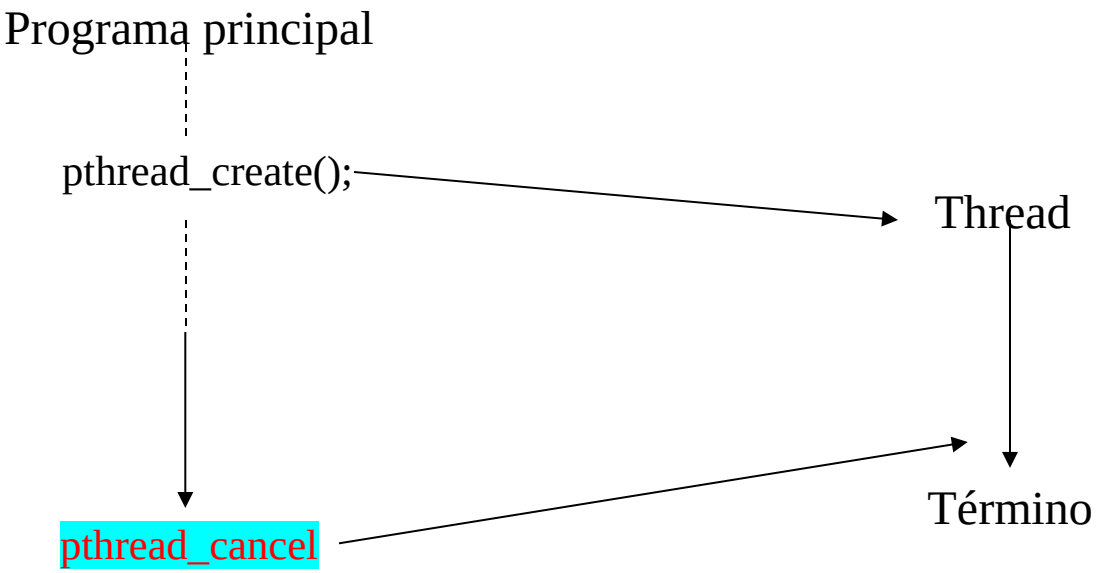
Detached Threads (desunidas)

Pode ser que uma thread não precisa saber do términação duma outra por ela criada, então não executará a operação de união.
Neste caso diz-se que o thread criado é *detached* (desunido da thread progenitor). A thread apo aterminar deve chamar pthread_exit para libertar recursos.



Terminar Threads

Pode ser que uma thread precisa de terminar outra thread – utilizar “cancel”



Visualizar informação sobre threads

- **GUI's** - p.ex windows – process explorer

- **CLI's** - linux / macintosh
  - Comando ps
  - Utilizar a opção –L ou -T
    - Opção –p $id (indicar process id)
    - Opção –f ou -F full informação
  - Comando top com opção –H
  - A pasta proc do Sistema de ficheiros tem uma lista das threads na pasta **/proc/[id]/task**
    - [id] é o proceso ID.
    - A pasta **/proc/[id]/ tem muita mais informação sobre um processo**

Exemplos

- **Compile to a.out using cc -Wall sim1.c –lpthread**
- **Execute with ./a.out**

- **sim1.c** - Simple program to print a message in each thread

- **sim2.c** In this program the worker Threads are stopped with a spin lock until the user presses enter on the main thread. This way you can inspect the situation with process management tools such as task manager, ps, ctrl-z etc
  - Exemplos: ps –Alf | grep /a.out ps –L pid

- **sim3a.c** Passing thread id argument – error
  - **./a.out | sort -n -k 3**

- **sim3b.c** Passing thread id argument – safe versions

- **demolwp.c –** native linux program using the clone() sys. call

Sincronização de Threads

Cuidados com a utilização de Threads

1. As execuções das instruções dos processos/threads individuais podem ser entrelaçadas no tempo.
2. Reordenação das instruções.
3. Acesso aos dados compartilhados.

1. Entrelaçamento de instruções

- **As execuções das instruções das threads individuais podem ser entrelaçadas no tempo – non-determinismo**
- **Exemplo:**

thread 1	thread 2
Instrução 1.1	Instrução 2.1
Instrução 1.2	Instrução 2.2
Instrução 1.3	Instrução 2.3

Possível ordem de execução no tempo:

Instrução 1.1
Instrução 1.2
Instrução 2.1
Instrução 1.3
Instrução 2.2
Instrução 2.3

Exercício : Enumerar o conjunto de possibilidades para 2 threads cada uma (i) com 2 instruções e (ii) cada um com 3 instruções utilizando um arvore de decisão

2. Optimizações do compilador/processador

- **O compilador (e até o processador em tempo de execução) pode reordenar as instruções para tentar otimizar a execução**
- **Exemplo:**
  - As instruções abaixo escritos neste ordem no código fonte

    a = b + 5;

    x = y + 4;

    podem ser compiladas e executadas em ordem contrária:

    x = y + 4;

    a = b + 5;

    O programa continua logicamente correto ? Depende !!

2. Optimizações do compilador/processador

- **Memory Barrier**
  - Para tentar evitar problemas que podem ocorrer podemos inserir uma "memory barrier"
    - asm volatile ("" ::: "memory") //compiler guarantee only
    - std::atomic_thread_fence
- **Exemplo:**
  - As instruções abaixo escritos neste ordem no código fonte

```
a = b + 5;
asm volatile ("" ::: "memory");
x = y + 4;
```

Mais Informação : https://en.wikipedia.org/wiki/Memory_barrier

3. Acessando dados compartilhados

- **Considere duas Threads que devem ler o valor de uma variável *x*, somar 1 a esse valor e escrever o novo valor de volta na variável *x***

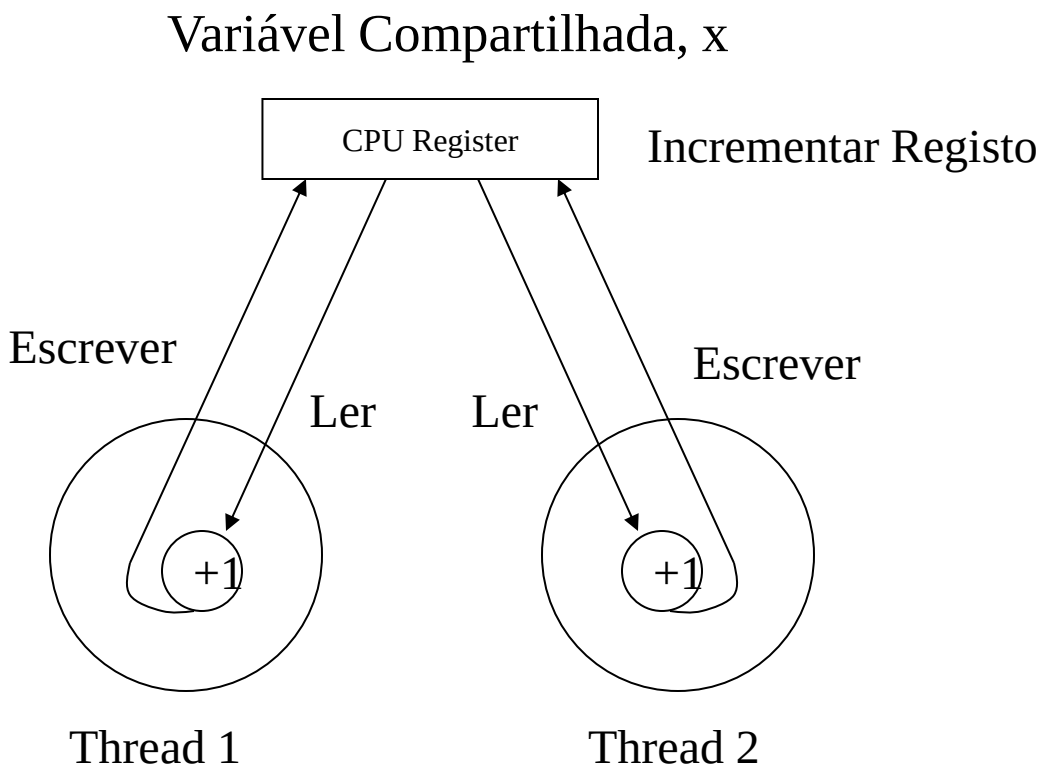
tempo

Instrução
x = x + 1

Thread 1	Thread 2
ler x de memoria	ler x de memoria
calcular x + 1	calcular x + 1
escrever x em memoria	escrever x em memoria

**Duas Threads ou dois Processos com uma variável partilhada num segmento de memoria partilhada

Conflito em acesso a dados compartilhados



Condições de Bernstein

- **Conjunto de condições suficientes para determinar se dois processos/threads (duas seções de código) podem ser executados concorrentemente.**
- **Sejam I_i o conjunto de localizações de memória lidas por um processo P_i e O_j o conjunto de localizações alteradas pelo processo P_j**
- **Três condições suficientes para que dois processos P_1 e P_2 sejam executados concorrentemente**
  - $I_1 \cap O_2 = \varnothing$
  - $I_2 \cap O_1 = \varnothing$
  - $O_1 \cap O_2 = \varnothing$

Exemplo das condições de Bernstein

- **Suponha as duas instruções em C:**

  Process.1 : a = x + y;

  Process.2: b = x + z;

- **Temos:**

  $I_1 = (x,y)$ $O_1 = (a)$

  $I_2 = (x,z)$ $O_2 = (b)$

- **Existem as três condições suficientes para que dois processos P_1 e P_2 sejam executados concorrentemente**
  - $I_1 \cap O_2 = \varnothing$
  - $I_2 \cap O_1 = \varnothing$
  - $O_1 \cap O_2 = \varnothing$

Secção crítica

- **Devem existir mecanismos para assegurar que somente um processo possa aceder a um determinado recurso durante um certo tempo**
- **Estabelecem-se sessões de código que envolvem o recurso, denominadas secções críticas, e organiza-se o acesso a elas de tal modo que somente uma delas pode ser executada por um processo de cada vez**
- **Um processo impede que outros processos acessem as secções críticas que possui um determinado recurso compartilhado quando ele estiver acessando**
- **Quando um processo finaliza a execução da seção crítica, outro processo pode executá-la**
- **Mecanismo conhecido como exclusão mútua**

Lock

- **Mecanismo mais simples de assegurar exclusão mútua para acesso a seções críticas**
- **O lock é logicamente uma variável de 1 bit que possui o valor 1 quando existe algum processo na secção crítica e 0 quando nenhum processo está na secção crítica**
- **Um processo que chega a uma porta da secção crítica e a acha aberta pode entrar, trancando-a para prevenir que nenhum outro processo entre**
- **Assim que o processo finaliza a execução da seção crítica, ele destranca a porta e sai**

Spin lock

Shared variable
Initially bit lock=0

```
while ( lock ) fazer_nada;
lock = 1;
        .
    secção crítica
        .
lock=0;
```

Processo 1

```
while ( lock )  fazer_nada;
lock =1;

secção crítica

lock=0;
```

Processo 2

```
while (lock ) fazer_nada;
        |
        v
lock = 1;

secção crítica

lock = 0;
```

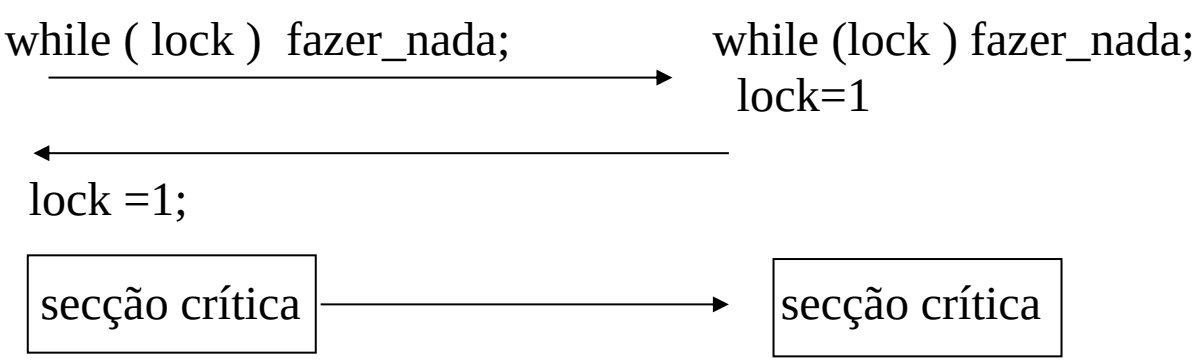
A simple non-reusable spin lock ??

Spin lock

Shared variable
Initially bit lock=0

```
while ( lock ) fazer_nada;
lock = 1;
        .
    secção crítica
        .
lock=0;
```

Processo 1 *Processo 2*



Spin lock

ATOMIC

```
while ( lock ) fazer_nada;
lock = 1;
        .
    secção crítica
        .
lock=0;
```

Shared variable
Initially bit lock=0

Processo 1

```
while ( lock )  fazer_nada;
lock =1;

secção crítica

lock=0;
```

Processo 2

```
while (lock ) fazer_nada;

lock = 1;

secção crítica

lock = 0;
```

Acesso Concorrente às Variaveis Globais

Conclusão: Quando duas ou mais *threads* podem simultaneamente alterar às mesmas variaveis globais (ou uma alterar enquanto outra está a ler) poderá ser necessário sincronizar o acesso a este variavel para evitar *problemas*.

Código nesta condição diz-se "uma secção crítica"

Por exemplo, quando dois ou mais threads podem simultaneamente incrementar uma variável x

```
/* codigo – Secção Critica */
x = x +1 ;
```

Uma seção crítica deve ser protegida utilizando-se o conceito dum trinco logico, as funções *pthread_mutex_lock*() e *pthread_mutex_unlock*()

Rotinas de lock para Pthreads

- **Os locks são implementados com variáveis lock mutuamente exclusivas : as variáveis "mutex"**

  declaração : pthread_mutex_t mutex1;

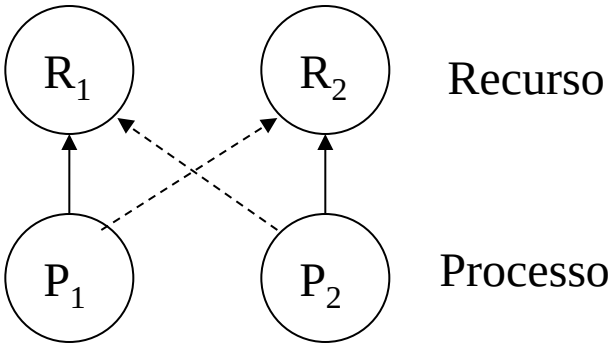
  inicialização : pthread_mutex_init (&mutex1, NULL);
- NULL specifies a default attribute for the mutex.
- Uma variável mutex pode ser destruída com *pthread_mutex_destroy*()

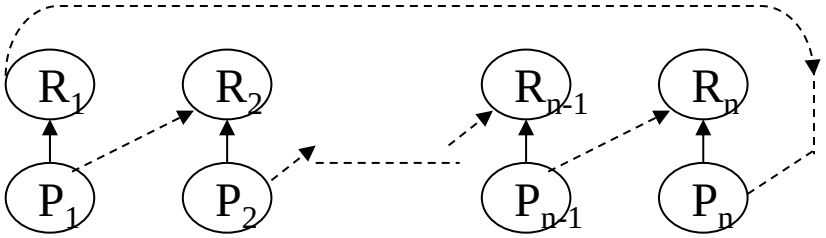
- **Uma seção crítica pode ser protegida utilizando-se *pthread_mutex_lock*() e *pthread_mutex_unlock*()**

```
pthread_mutex_lock(& mutex1);
 .
seção crítica
   .
pthread_mutex_unlock(&mutex1);
```

Deadlock

- **A rotina de lock() é bloqueante !**
- **Deadlock - Pode ocorrer com dois processos quando um deles precisa de um recurso que está com outro e esse precisa de um recurso que está com o primeiro**





Jantar dos filósofos Chineses

Rotina para evitar deadlock em Pthreads

- **pthreads oferece uma rotina para testar se uma variável está trancada sem bloquear a thread**
- **pthread_mutex_trylock()**
  - tranca uma variável *mutex* que esteja destrancada e retorna o valor 0,
  - ou retorna* o valor EBUSY caso a variável esteja trancada

```
if   (EBUSY == pthread_mutex_trylock (&mutex)  )
                Fazer algo
else
                Secção critica
```

* também pode retornar EINVAL caso que a mutex não tenha sido bem inicializado

Exemplos

- **Somar um vector – com e sem locks**

Exemplo: SOMAR

- **Somar os elementos de um array, a[1000]**

```
int sum, a[1000]
//Initialize the vector a
sum = 0;
for (i = 0; i < 1000; i++)
        sum = sum +a[i];
```

O problema será implementada numa versão multi-threaded usando,
pthreads e java threads
E para efeitos de comparação multiplos-processos com unix IPC

Implementação 1
Pthreads

- **Usando *2* threads com ponto de sincronização garantida pela operação de "join"**

Thread 1

valores pares

```
somaPar = 0
for (i = 0; i < 1000; i = i+2)
    somaPar = somaPar + a[i]
```

Thread 2

valores impares

```
somaImpar = 0
for (i = 1; i < 1000;  i= i+2)
    somaImpar = somaImpar + a[i]
```

- **Esperamos a junção das duas threads**

```
somaFinal = somaPar + somaImpar;
```

Implementação I

```
#define size 10000
//Global Variables
int a[size];
int somaPar=0, somaImpar=0;

void * par ( void  *nenhum )
{
  for (int i = 0; i < size; i=i+2)
     somaPar += a[i];
  return(NULL);
}
void * impar ( void  *nenhum )
{
  for (int i = 1; i < size; i=i+2)
     somaImpar += a[i];
  return(NULL);
}
```

```
int main()
{
   pthread_t  thread [2] ;

  for (int i = 0; i < size; i++)  //initialize some data
     a[i] = i+1;

  pthread_create(&thread[0], NULL, impar, NULL);
  pthread_create(&thread[1], NULL, par,      NULL);

   pthread_join( thread[0], NULL) ;
   pthread_join( thread[1], NULL) ;

   int sum = somaPar + somaImpar;

   printf("A soma é %d vs %d\n", sum, (size*(size+1))/2);
   return 0;
}
```

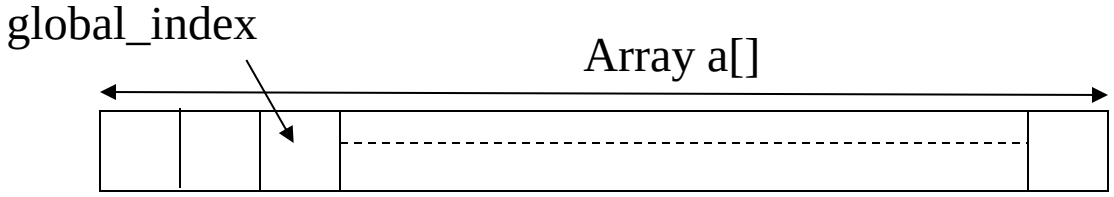
Implementação II - Generalização

```
#define size 100000
#define N 100
//Global Variables
int a[size];
int somas[N]={0};

void * worker ( void  *idPtr )
{
   int id = *(int *)idPtr;  //0 ,1,2.. N-1
   free((int*)idPtr);

   for (int i = id; i < size; i=i+N)
      somas[id] += a[i];

 return(NULL);
}
```

```
int main()
{
   pthread_t  thread [N] ;
   for (int i = 0; i < size; i++)  //initialize some data
       a[i] = i+1;

   for (int i = 0; i < N; i++)
   {
       int *m=(int*)malloc( sizeof(int) );
      *m=i;
       pthread_create(&thread[i], NULL, worker, (void*) m);
    }
    int sum = 0;
    for (int i = 0; i < N; i++){
        pthread_join( thread[i], NULL) ;
       sum += somas[i];
   }
    printf("A soma é %ld vs %d\n", sum, (size*(size+1))/2);
}
```

Implementação III utilizando n Pthreads

- **São criadas *n* threads, cada uma obtém os números de uma lista *a[]*, os soma e coloca o resultado numa variável compartilhada *sum***
- **A variável compartilhada *global_index* é utilizada por cada thread para selecionar o próximo elemento da lista**
- **Após a leitura do índice, ele é incrementado para preparar a leitura do próximo elemento**
- **Estrutura de dados utilizada**



```
void * worker ( void  *nenhum )
{
  int local_index,
      partial_sum = 0;
  do   {

      local_index = global_index;
      global_index++;

    if (local_index < size)
       partial_sum += a[local_index];

  } while (local_index < size);


      sum +=  partial_sum;

  return(NULL);
}
```

```
#define size 10000
#define NTHREADS  10

int a[size];
int global_index = 0,  sum = 0;


main()
{
  int i;
  pthread_t  thread [NTHREADS] ;


  for (i = 0; i < size; i++)
    a[i] = i+1;

 for (i = 0; i < NTHREADS; i++)
   pthread_create(&thread[i], NULL, worker, NULL);

for (i = 0; i < NTHREADS; i++)
     pthread_join(thread[i], NULL) ;

 printf("A soma é %d vs %d\n", sum, (size*(size+1))/2);
}
```

```
void * worker ( void  *nenhum )
{
  int local_index,
      partial_sum = 0;
  do  {
    pthread_mutex_lock(&mutex1);
      local_index = global_index;
      global_index++;
    pthread_mutex_unlock(&mutex1);
    if (local_index < size)
       partial_sum += a[local_index];

  } while (local_index < size);

   pthread_mutex_lock(&mutex1);
      sum +=  partial_sum;
  pthread_mutex_unlock(&mutex1);
  return(NULL);
}
```

```
#define size 10000
#define NTHREADS  10

int a[size];
int global_index = 0,  sum = 0;
pthread_mutex_t mutex1;

main()
{
  int i;
  pthread_t  thread [NTHREADS] ;

  pthread_mutex_init(&mutex1, NULL);
  for (i = 0; i < size; i++)
    a[i] = i+1;

 for (i = 0; i < NTHREADS; i++)
   pthread_create(&thread[i], NULL, worker, NULL);

for (i = 0; i < NTHREADS; i++)
     pthread_join(thread[i], NULL) ;

 printf("A soma é %d vs %d\n", sum, (size*(size+1))/2);
}
```

```
void * worker ( void  *nenhum )
{
  int local_index,
       partial_sum = 0;
  do   {
    pthread_mutex_lock(&mutex1);
      local_index = global_index;
      global_index++;
    pthread_mutex_unlock(&mutex1);
    if (local_index < size)
        partial_sum += a[local_index];

  } while (local_index < size);

   pthread_mutex_lock(&mutex2);
      sum +=  partial_sum;
  pthread_mutex_unlock(&mutex2);
  return(NULL);
}
```

```
#define size 1000
#define NTHREADS 10

int a[size];
int global_index = 0,  sum = 0;
pthread_mutex_t mutex1, mutex2;  //a little bit more effecient

main()
{
  int i;
  pthread_t  thread [NTHREADS] ;

  pthread_mutex_init(&mutex1, NULL);
  for (i = 0; i < size; i++)
    a[i] = i+1;

 for (i = 0; i < NTHREADS; i++)
   pthread_create(&thread[i], NULL, worker, NULL);

for (i = 0; i < NTHREADS; i++)
     pthread_join(thread[i], NULL) ;

 printf("A soma é %d \n", sum);
}
```

```
public class Adder
{
   private final int SIZE = 5000;
   private final int NTHREADS = 10;
   private int threads_quit = 0;
   public int [] array;
   private int sum = 0;
   private int index = 0;

   public Adder ()
   {
       threads_quit = 0;
       array = new int[SIZE];
       initializeArray();
   }

   public synchronized int getNextIndex()
   {
       if ( index < SIZE ) return (index ++); else return (-1);
   }

  public synchronized void  addPartialSum(int partial_sum)
   {
       sum = sum + partial_sum;
       if (++threads_quit == NTHREADS)
          System.out.println("a soma dos números e'"+sum);
   }
```

Java

O algoritmo de somar é igual ao algoritmo usado na implementação anterior usando pthreads – quer dizer utilize uma variável compartilhada aqui chamada index.

Usaremos a palavra synchronzied para métodos que alteram variáveis globais e que necessitam de sincronização

```
  private void initializeArray()
  {
    for (int i = 0; i < SIZE; i++) array[i] = i;
  }

  public void startThreads ()
  {
     for (int i = 0; i < NTHREADS; i++)
     {
        AdderThread at = new AdderThread(this, i);
        at.start();
      }
  }
  public static void main(String args[])
  {
       new Adder();
  }
} //end class Adder
```

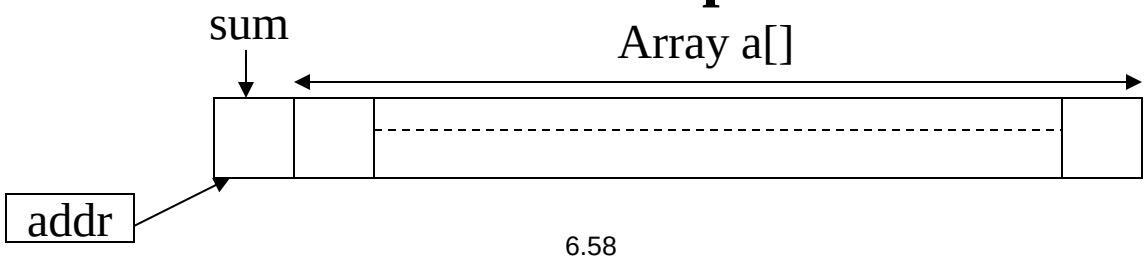
```
class AdderThread extends Thread
{
  int partial_sum = 0;
  Adder parent;
  int number;

  public AdderThread (Adder parent, int number)
  {
      this.parent = parent;
      this.number = number;
  }

  public void run ()
  {
     int index = 0;
     while ( index  !=  -1) {
         partial_sum = partial_sum + parent.array[index];
         index = parent.getNextIndex();
     }
      System.out.println(" Soma parcial thread " + number + "é "+ partial_sum);
     parent.addPartialSum(partial_sum);
  }
}
```

Implementação utilizando dois processos comUNIX Inter Process Communication (IPC)

- **O cálculo é dividido em duas partes, uma para a soma dos elementos pares e outra para os ímpares**

Processo 1	Processo 2
sum1 = 0	sum2 = 0
for (i = 0; i < 1000; i = i+2)	for (i = 1; i < 1000; i= i+2)
sum1 = sum1 + a[i]	sum2 = sum2 + a[i]

- **Cada processo soma o seu resultado (sum1 ou sum2) a uma variável compartilhada sum que acumula o resultado e deve ser ter seu acesso protegido**
- **Estrutura de dados utilizada para o vetor e soma**



Shared Memory Needed :

size * sizeof(int) + 1 * sizeof(int) (global sum) // + Semáforo

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define size 10000

void P(int *s);
void V(int *s);
int main()
{
  int shmid, s, pid;
  char *shm;
  int *a, *addr, *sum;
  int partial_sum;
  int i;
  int init_sem_value =1 ;
```

```
//setup shared semaphore for synchronization
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT));
if (s == -1) {
   perror("semget"); exit(1);
}
if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
   perror("semctl"); exit(1);
}
```

```
//setup shared memeory segment
shmid =shmget(IPC_PRIVATE,((size+1)*sizeof(int)),IPC_CREAT|0600);
if (shmid == -1)  {
    perror("shmget"); exit (1);
}
shm = (char *) shmat(shmid, NULL, 0);
if (shm == (char*)-1) {
   perror("shmat");  exit(1);
}
```

```
addr = (int *) shm;
sum = addr;              //sum at first address
*sum = 0;
addr++;
a = addr;                //vector starts at next address
//vamos arranjar alguns dados para somar
for (i = 0; i < size; i++)    a[i] = i+1;

pid = fork();
if (pid ==0)  {
   partial_sum=0;
   for (i=0; i<size; i=i+2)
       partial_sum+=*(a+i);
  }
else  {
   partial_sum=0;
   for (i=1; i<size; i=i+2)
       partial_sum+=*(a+i);
}

printf("soma parcial = %d \n",partial_sum);

//actualizar a  soma global
P(&s);
   *sum  += partial_sum;
V(&s);
```

NOW .. WAIT FOR CHILD TO FINISH, PRINT RESULT, FREE Shared Memory

```
//esperar pela terminação do filho
if (pid == 0) exit (0);
else              wait(0);

//Only the parent process gets here
printf("A soma total é %d \n", *sum);

/* remover memoria partilhada e semafor */

if (semctl(s, 0, IPC_RMID,1) == -1) {
    perror("semctl");
    exit(1);
}
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
}
return 0;
}
```

Implementação das funções "wait" and "signal " dum semaforo "s"

"s" é um inteiro- reside em memoria partilhada

```
void V(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num =0;
   sops->sem_op =   1;
   sops->sem_flg =   0;
   if  (semop(*s, sops, 1) < 0)  {
      perror ("semop");
      exit (1);
   }
   return;
}
```

```
void P(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op =  -1;
   sops->sem_flg =   0;
   if  (semop(*s, sops, 1) < 0)  {
      perror ("semop");
      exit (1);
   }
   return;
}
```