Capitulo 6 Threads

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

Conceito dum Processos

O conceito dum Processo pode ser dividido em dois :

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

Por exemplo :

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

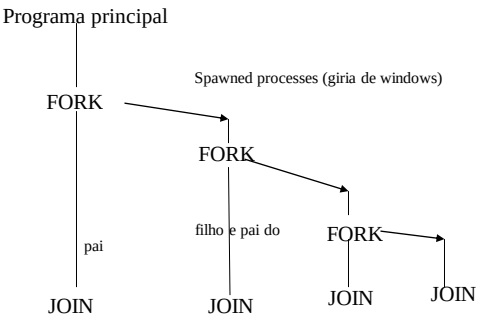
2 Uma linha ou contexto de execução, chamada "Thread"

- Uma thread tem um programa counter (pc) que guarde informação sobre a próxima instrução a executar
- Registadores – valores das variáveis actuais
- Stack – contem a história de execução com um "frame" para cada procedimento chamado mas não terminado

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

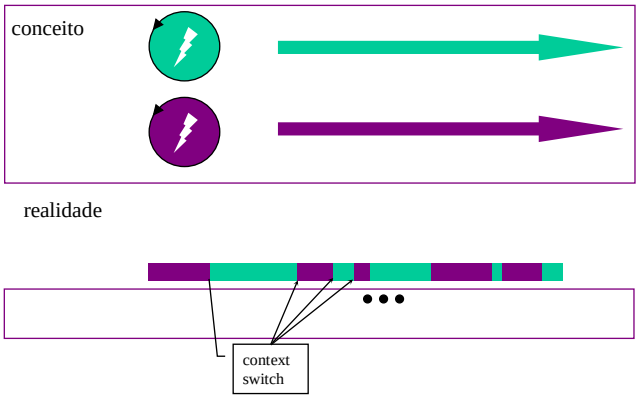
O processo filho é uma **cópia exacta** do processo que chama a rotina *fork*(), sendo a única diferença um identificador de processo (pid) diferente

```
pid = fork()
if (pid == 0)
   código executado pelo filho
else
   código executado pelo pai
if (pid == 0)
   exit (0);
else
   wait (0); //esperar pelo filho .. JOIN
```



Inter Process Communication – Pipes, Shared Memory, Signal etc.

Dois Processos partilhando um CPU



O Modelo de Threads

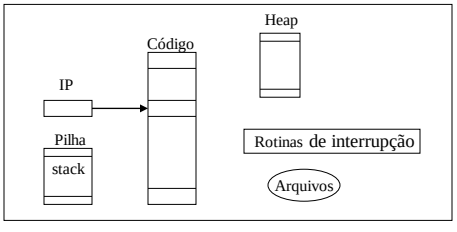
Os dois conceitos necessários para a execução duma programa, "conjunto de recursos" e "contexto de execução" podem ser separados:

Assim

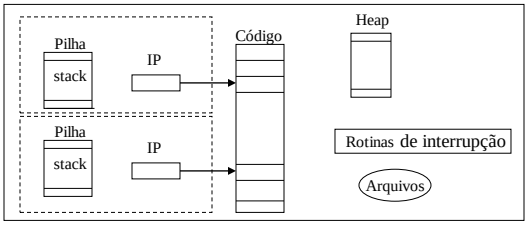
1. Processos são usados para agrupar recursos.
2. Threads são as entidades escalonadas para execução no CPU.

Processos versus Threads

Heavyweight Process:
programas completamente separados com suas próprias variáveis, pilha e alocação de memória



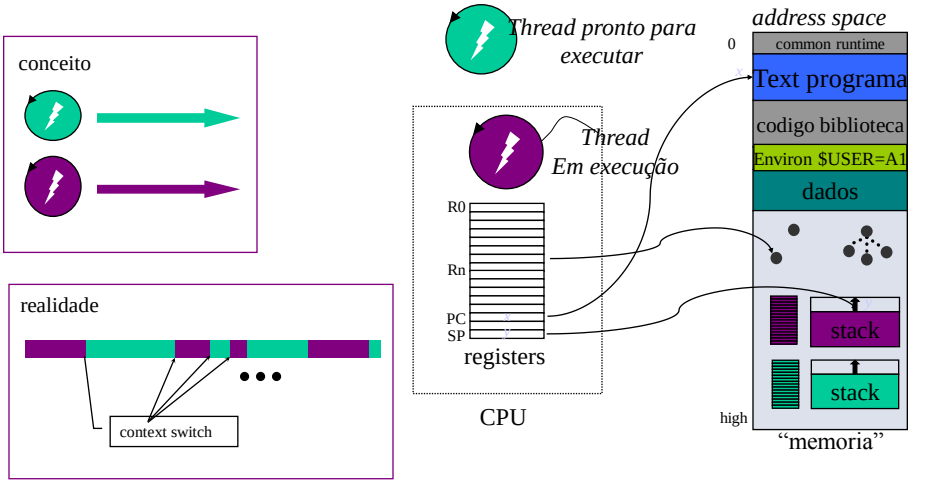
Threads: lightweight processes
rotinas compartilham o mesmo espaço de memória e variáveis globais



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 • Signals • Semaphores • Accounting Information

Duas Threads partilhando um CPU

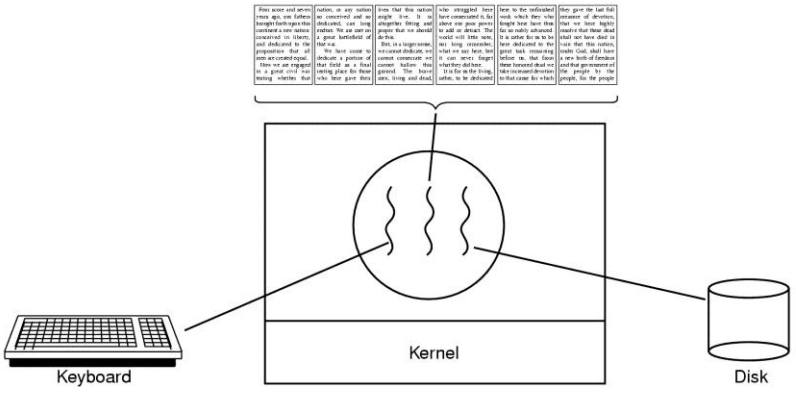


Beneficios de Threads vs Processos

- 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 duas threads é mais rápido do que entre dois processos.
- A comunicação entre threads é mais rápida do que a comunicação entre processos - porque as threads compartilham tudo: espaço de endereçamento, variáveis globais etc.
- Multi-programação usando o modelo de threads é mais simples e mais portável do que multi-programação usando múltiplos processos.

6.9

Thread Usage (1) Office

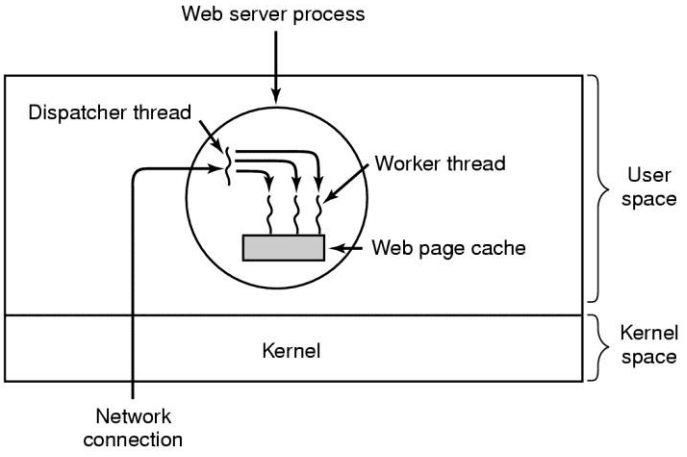


Processador de Texto com três threads.

Quantos threads tem “microsoft word” no SO windows ?

6.10

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 );
   Iniciar_Worker_Thread( buf );
}
```

(b) Worker thread

```
Enquanto (verdade) {
   Esperar_Trabalho ( &buf );
   Look_for_page_cache(&buf,&page);
   If (page_not_in_cache (&page)
      Read_page_from_disk( &buf, &page);
   Return _page( &page );
}
```

Threads múltiplas podem estar a executar simultaneamente para aproveitar arquitecturas como "hyperthreading", "multi-core" e "smp" etc. 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). Aqui, para cada nova ligação através dum programa cliente tipo putty o programa faz "fork"

Thread Usage (3) : Matrizes

- Multiplicação de matriz : A.B=C
- O calculo dum elemento da Matriz C, envolve a multiplicação duma linha de Matrizx A com a coluna da Matriz B.

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

Repare que cada elemento da Matriz C pode ser calculado indepenedentemente dos outros, portanto podem ser facilmente calculados por threads diferentes.

Links

- **Very shortly, threads are "lightweight" processes (see the definition of a process in the Linux Kernel book) that can run in parallel. This is a lot different from standard UNIX processes that are scheduled and run in sequential fashion. More threads information can be found here or in the excellent Linux Parallel Processing HOWTO by Professor Hank Dietz.**
- **From The linux-kernel mailing list FAQ**
- **(REG) When people talk about threads, they usually mean kernel threads i.e. threads that can be scheduled by the kernel. On SMP hardware, threads allow you to do things truly concurrently (this is particularly useful for large computations). However, even without SMP hardware, using threads can be good. It can allow you to divide your problems into more logical units, and have each of those run separately. For example, you could have one thread read blocking on a socket while another reads something from disk. Neither operation has to delay the other. Read "Threads Primer" by Bill Lewis and Daniel J. Berg (Prentice Hall, ISBN 0-13-443698-9).**
- **Bibliografia acerca de threads (linux kernel developers)**
- **http://liinwww.ira.uka.de/bibliography/Os/threads.html**
- **http://software.intel.com/en-us/articles/boost-game-performance-with-threads/?cid=sw:kmlnews3930&CTP=4K0V7T4F**

Implementação de Threads

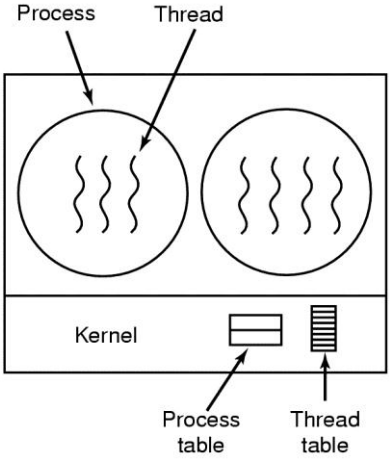
Existem duas abordagens principais para a implementação de threads.

- **Kernel-level Threads -- System Calls.**
- **User-Level Threads -- Thread Libraries.**

Existem vantagens e desvantagens em ambos os casos.

Também existe sistemas operativos que implementam threads usando uma abordagem mista dos dois métodos principais, por exemplo Solaris da Sun.

Implementação de Threads no Kernel do SO



threads geridas pelo kernel

Implementação deThreads no Kernel (KLT)
Vantagens e Desvantagens

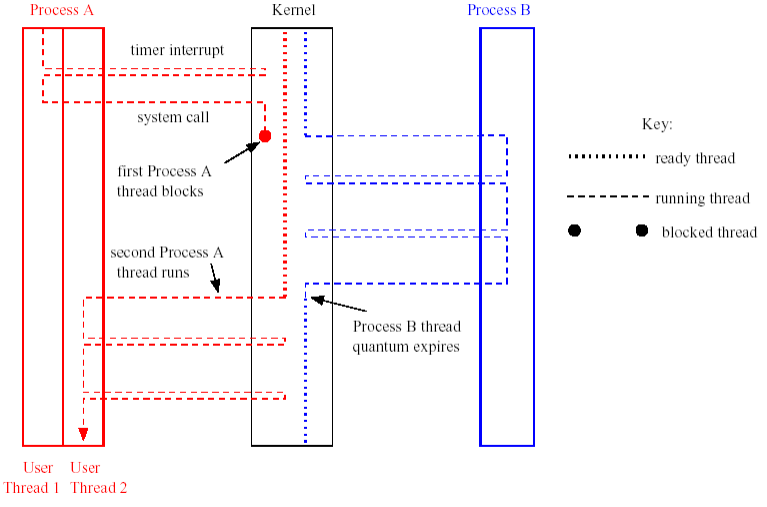
Vantagens

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

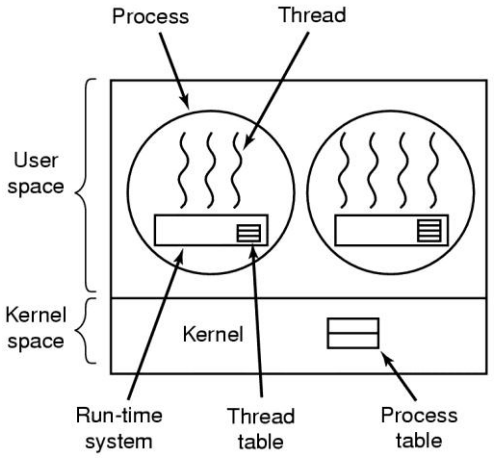
Desvantagens:

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

Execução de KLT's



Implementação de Threads no User Space



Uma biblioteca de threads implementada ao nível de utilizador.

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

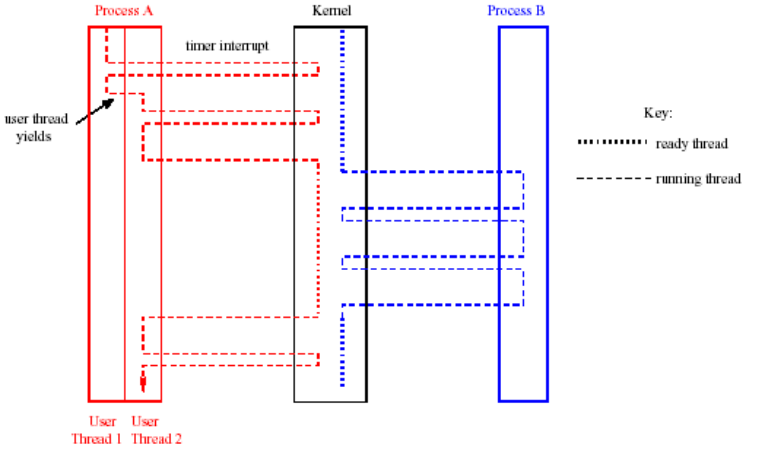
Vantagens

- 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

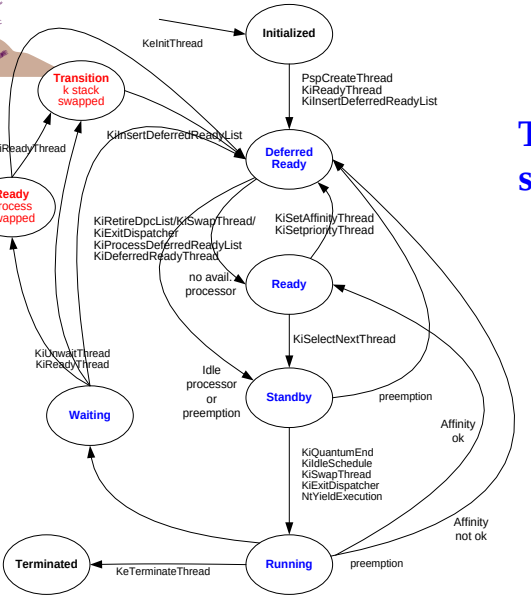
Desvantagens

- 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 arquitectura com múltiplas processadores

Execução de ULT's



Thread scheduling states on Windows



Kernel Thread Transition Diagram
DavePr@Microsoft.com
2003/04/06 v0.4b

Pthreads API

The POSIX 1003.1-2001 standard defines an application programming interface (API) for writing multithreaded applications. This interface is known more commonly as *pthreads*.

For more information, see the LinuxThreads README

and the LinuxThreads Frequently Asked Questions.

Getting Started With POSIX Threads Tutorial
http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html

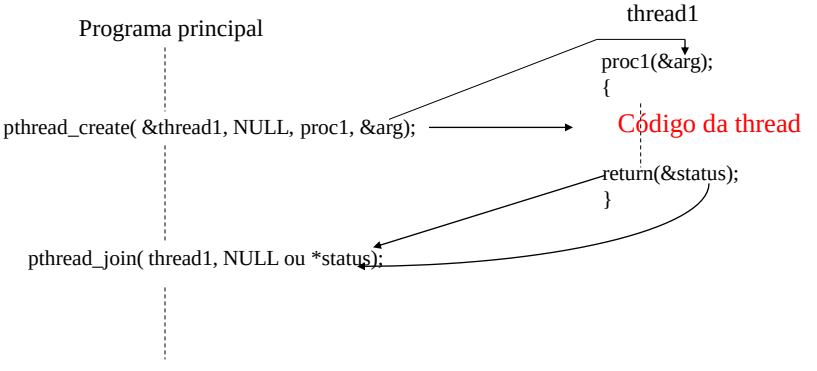
numa maquina linux

>more /usr/include/pthread.h

> man pthread_create etc

Criação de Threads: usando Pthreads

Interface portável do sistema operacional, POSIX, IEEE



Pthread Join

- **A rotina *pthread_join*() espera pelo término de uma thread específica**

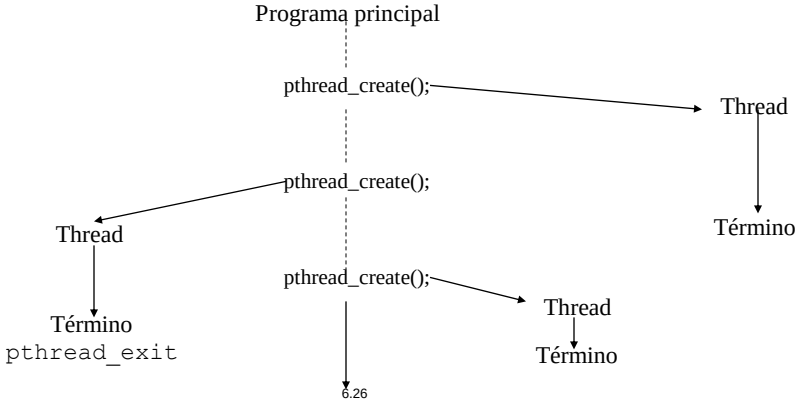
```
for (i = 0; i < n; i++)
    pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);

...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 término de uma outra por ela criada, então não executará a operação de união. Neste caso diz-se que o thread criado é *detached* (desunido da thread pai progenitor)



Acesso Concorrente às Variaveis Globais

Quando dois ou mais threads podem simultaneamente alterar às mesmas variaveis globais poderá ser necessário sincronizar o acesso a este variavel para evitar problemas.

Código nestas condição diz-se "uma secção critica"

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

/* codigo – Secção Critica */

x = x +1 ;

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

6.27

Rotinas de lock para Pthreads

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

  declaração : pthread_mutex_t mutex1;

  inicialização : pthread_mutex_init (&mutex1, NULL);
- NULL specifies a default attribute for the mutex.
- Uma variável mutex criada dinamicamente (via malloc) deverá ser destruída com a função *pthread_mutex_destroy*()

- **Protegindo uma secção critica utilizando *pthread_mutex_lock*() e *pthread_mutex_unlock*()**

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

6.28

Exemplos

- **Visualizar threads**
  - Ferramentas de visualização
  - windows – process explorer
  - linux / macintosh – comando ps
- **Multiplicação de Matrizes**
- **Somar um vector**

Exemplo: SOMAR

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

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

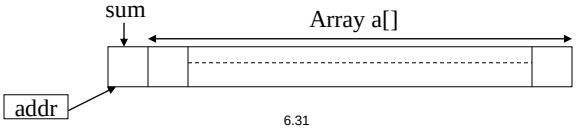
O problemas será implementada usando, multi-processos com unix IPC multi-thread com pthreads e multi-threads com java threads.

Implementação utilizando dois processos UNIX 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**



Shared Memory Needed : Array_size * sizeof(int) + 1 int (global sum) // + Semaforo

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

#define array_size 1000

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 ;
```

```
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);
}
```

```
shmid =
shmget(IPC_PRIVATE,((array_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
addr++;
a = addr;                //vector at others
*sum = 0;
//vamos arranjar alguns dados para somar
for (i = 0; i < array_size; i++)    *(a+i) = i+1;

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

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);
}

}
```

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

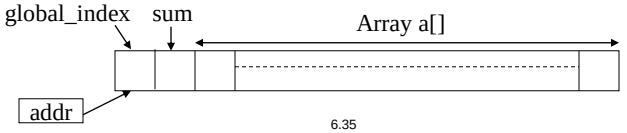
"s" é um inteiro- reside em memoria partilhada

```
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;
}
```

Implementação utilizando Pthreads

- **São criadas *n* threads, cada uma obtém os números de uma lista, 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 de *a***
- **Após a leitura do índice, ele é incrementado para preparar para a leitura do próximo elemento**
- **Estrutura de dados utilizada**



6.35

```
#define array_size 1000
#define no_threads 10

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;

void * slave ( 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 < array_size)
        partial_sum += *(a+local_index);
  } while (local_index < array_size);

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

```
main()
{
  int i;
  pthread_t  thread [no_threads] ;

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

 for (i = 0; i < no_threads; i++)
  if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
   {
      perror("Pthread_create falhou");
      exit(1);
   }

for (i = 0; i < no_threads; i++)
    if (pthread_join(thread[i], NULL) != 0)
    {
      perror("Pthread_join falhou");
      exit(1);
    }

 printf("A soma é %d \n", sum)
}
```

6.36

```
public class Adder
{
   public int [] array;
   private int sum = 0;
   private int index = 0;
   private int number_of_threads = 10;
   private int threads_quit;

   public Adder ()
   {
       threads_quit = 0;
        array = new int[1000];
       initializeArray();
       startThreads()
   }

   public synchronized int getNextIndex()
   {
       if (index < 1000) return (index ++); else return (-1);
   }

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

Implementação em Java

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

```
   private void initializeArray()
   {
      int i;
      for (i = 0; i < 1000; i+=) array[i] = i;
   }

   public void startThreads ()
   {
     int i = 0;
     for (i = 0; i < 10; i++)
      {
         AdderThread at = new adderThread(this, i);
         at.start();
      }
   }
   public static void main(String args[])
   {
     Adder a = 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();
    }
    parent.addPartialSum(partial_sum);

    System.out.println
      ('Soma parcial da thread " + number +
         "é "+ partial_sum);
  }
}
```