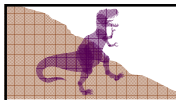


## Pthreads API

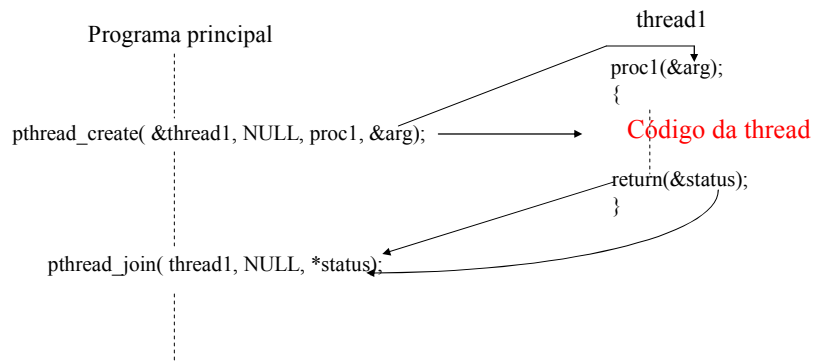
The [POSIX 1003.1-2001](#) standard defines an application programming interface (API) for writing multithreaded applications. This interface is known more commonly as *pthread*s.

6.1

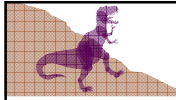


## Criação de Threads: usando Pthreads

Interface portátil do sistema operacional, POSIX, IEEE



6.2

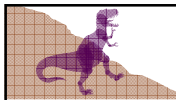


## Pthread Join

- A rotina *pthread\_join()* espera pelo término de uma thread específica

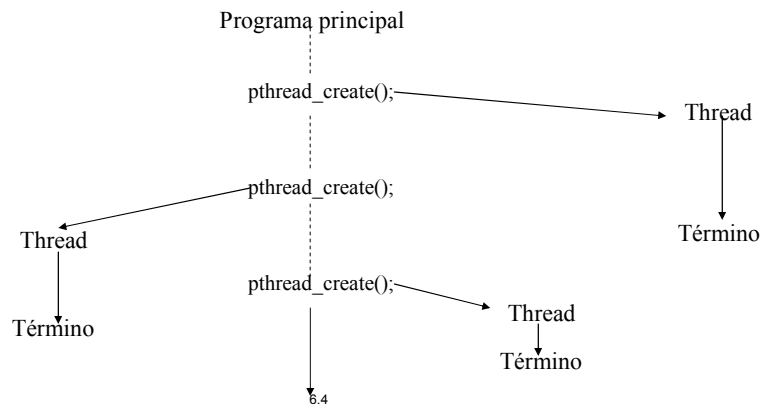
```
for (i = 0; i < n; i++)  
    pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);  
  
...thread mestre  
...thread mestre  
  
for (i = 0; i < n; i++)  
    pthread_join(thread[i], NULL);
```

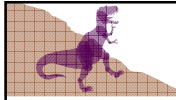
6.3



## 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 Variáveis Globais

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

Código nestas condições diz-se “**uma secção crítica**”

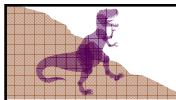
Por exemplo, quando dois ou mais threads podem simultaneamente incrementar uma variável x

```
/* código – Secção Crítica */
```

```
x = x + 1 ;
```

- Uma secção crítica pode ser protegida utilizando-se *pthread\_mutex\_lock()* e *pthread\_mutex\_unlock()*

6.5



## 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()*

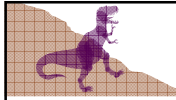
- Protegindo uma secção crítica utilizando *pthread\_mutex\_lock()* e *pthread\_mutex\_unlock()*

```
pthread_mutex_lock ( &mutex1 );
```

```
.  
seção crítica
```

```
pthread_mutex_unlock( &mutex1 );
```

6.6



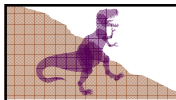
## Variáveis de condição

- Frequentemente, uma seção crítica deve ser executada caso exista uma condição específica, por exemplo, o valor de uma variável chegou a um determinado valor
- Utilizando-se locks, a variável global tem que ser frequentemente examinada dentro da seção crítica (*busy-wait cycle*)
- Consome tempo de CPU e pode não funcionar bem!

```
while (FLAG)
{
    lock () //O trinco protege qualquer variavel
            necessario para verificar condição
    if verificar_condicao
        FLAG = FALSE
    unlock()
}
```

6.7

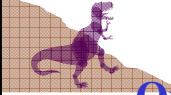
```
fazer algum trabalho()
lock()
condição a verificar = true
unlock()
```



## Variáveis de condição

- **Solução -Variáveis de condição - Três Operações Atomicas:**
  - wait(cond\_var) - espera que ocorra a condição
  - signal(cond\_var) - sinaliza que a condição ocorreu
  - status(cond\_var) - retorna o número de processos esperando pela condição
- A rotina wait() libera o lock ou semáforo e pode ser utilizada para permitir que outro processo altere a condição
- Quando o processo que chamou a rotina wait() é liberado para seguir a execução, o semáforo ou lock é trancado novamente

6.8



## Operações para variáveis de condição (Exemplo)

- Considere um ou mais processos (ou threads) designados a executar uma determinada operação quando um contador  $x$  chegue a zero
- Outro processo ou thread é responsável por periodicamente decrementar o contador

```

action()
{
    lock();

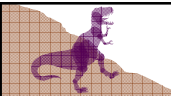
    while ( x != 0 )
        wait(s);
    unlock();
    take_action() ;
}
        
```

```

contador()
{
    lock();
    x--;
    if (x == 0)
        signal (s);

    unlock();
}
        
```

6.9



## Variáveis de condição em Pthreads

- Associadas a uma variável mutex específica
- Declarações e inicializações:
 

```

pthread_cond_t cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
            
```
- Utilização das rotinas de wait e signal:
 

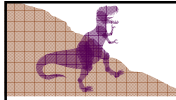
```

action()
{
    pthread_mutex_lock(&mutex1);
    while (c != 0 )
        pthread_cond_wait(&cond1, &mutex1);
    pthread_mutex_unlock(&mutex1);
    take_action();
}
            
```

```

contador()
{
    pthread_mutex_lock(&mutex1);
    c--;
    if (c == 0) pthread_cond_signal(&cond1);
    pthread_mutex_unlock(&mutex1);
}
            
```

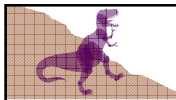
Signals are *not* remembered - threads must already be waiting for a signal to receive it.



## Exemplos

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

6.11



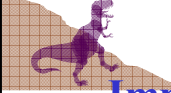
## Exemplo: SOMAR

- **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 problema será implementado usando, multi-processos com unix IPC multi-thread com pthreads e multi-threads com java threads.

6.12

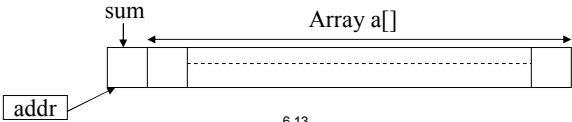


## Implementação utilizando dois processos UNIX Inter Process Communication (IPC)


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

<p>Processo 1</p> <pre>sum1 = 0 for (i = 0; i &lt; 1000; i = i+2)     sum1 = sum1 + a[i]</pre>	<p>Processo 2</p> <pre>sum2 = 0 for (i = 1; i &lt; 1000; i = i+2)     sum2 = sum2 + a[i]</pre>
--	--

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



6.13



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>

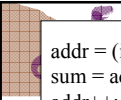
#define array_size 1000

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);
}
```

6.14



```

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);

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

```

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

```

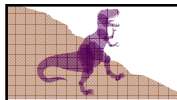
//esperar pela terminação do filho
if (pid == 0) exit (0); else wait(0);

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;
}

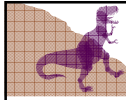
```

```

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;
}

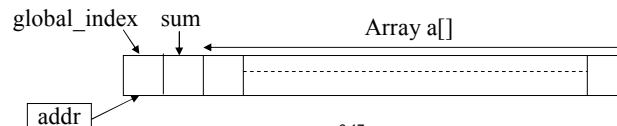
```





## 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



6.17

```
#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)
}
```

6.18



```

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 é "+sum);
    }
}

```

6.19

## 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

```

6.20

```

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);
    }
}

```