



**UNIVERSIDADE FEDERAL DE UBERLÂNDIA**  
**FACULDADE DE COMPUTAÇÃO**  
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

# **SISTEMAS OPERACIONAIS**

Prof. Rivalino Matias Jr

# **Unidade II:**

# **Modelo de Processos/Threads**

# Agenda

## ❑ MODELO DE PROCESSO

- ✓ Programa vs. Processo;
- ✓ Características básicas;
  - Estados, sub-processos, hierarquia, identificação e credenciais;

## ❑ IMPLEMENTAÇÃO INTERNA

- ✓ PCB;
- ✓ Contextos (Hardware/Software);

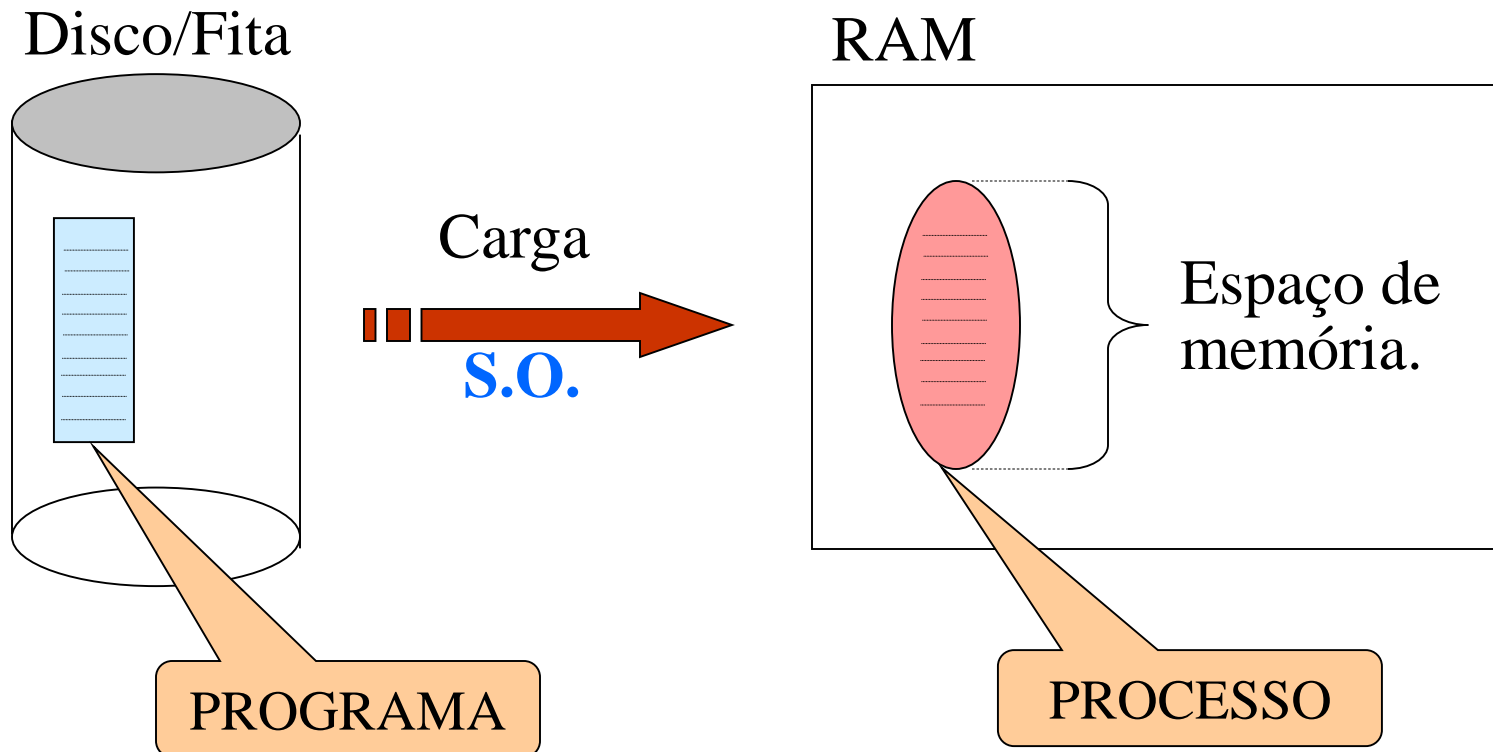
## ❑ MODELO DE *THREADS*

- ✓ Conceito
- ✓ Vantagens e Desvantagens;
- ✓ Paralelismo;

## ❑ EXERCÍCIOS

# Definição

- **Processo:** É uma instância de um programa em execução.



# Programa vs. Processo

## prog01.c

```
#include <stdio.h>
main()
{
    int vet[10000];
    vet[0]=1;
    getchar();
}
```

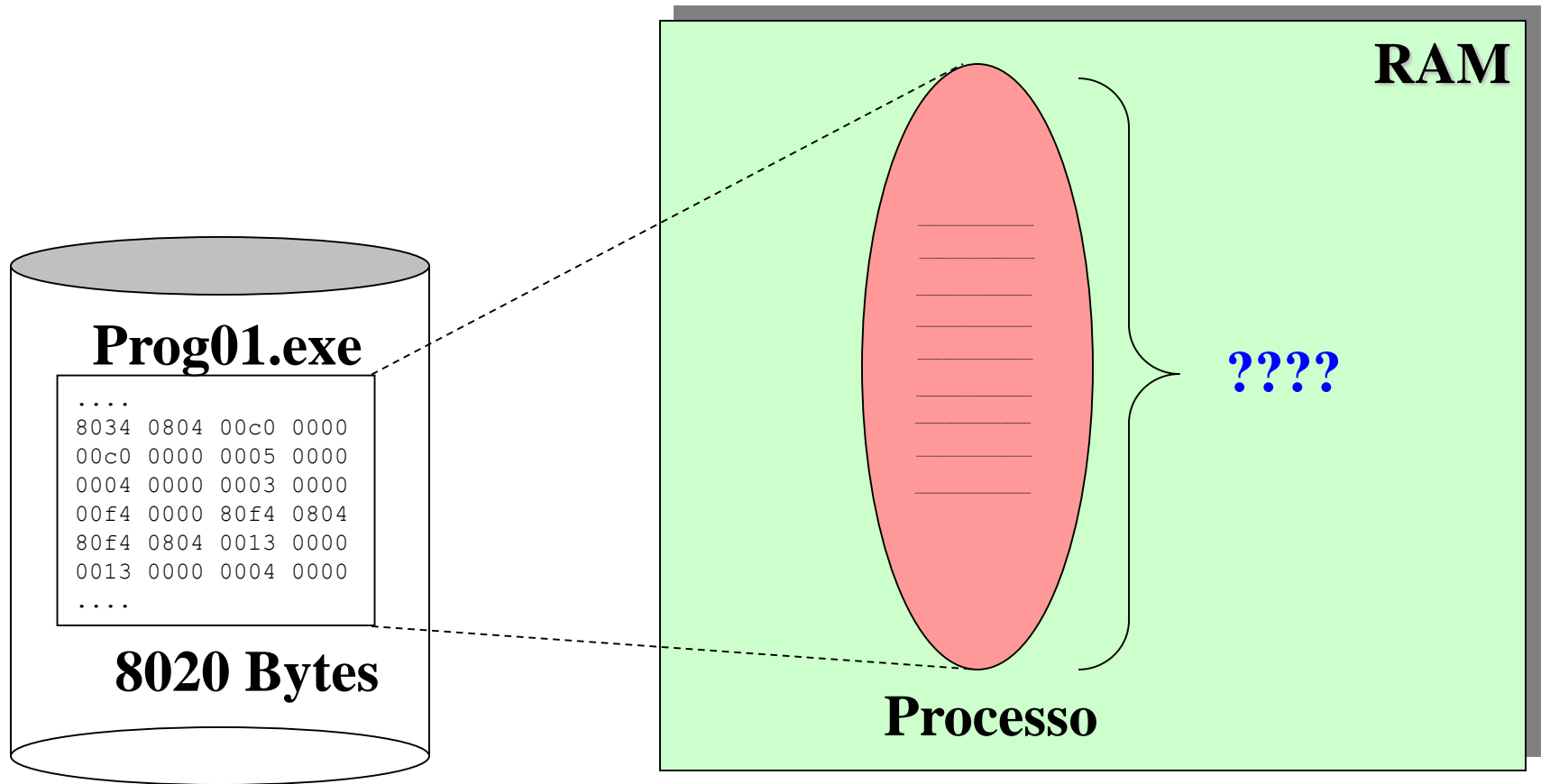
## prog01.s

```
...
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $40008, %esp
    andl     $-16, %esp
    movl     $0, %eax
    addl     $15, %eax
    addl     $15, %eax
    shrl     $4, %eax
    sall     $4, %eax
    subl     %eax, %esp
    movl     $1, -40008(%ebp)
    call     getchar
    leave
    ret
...
```

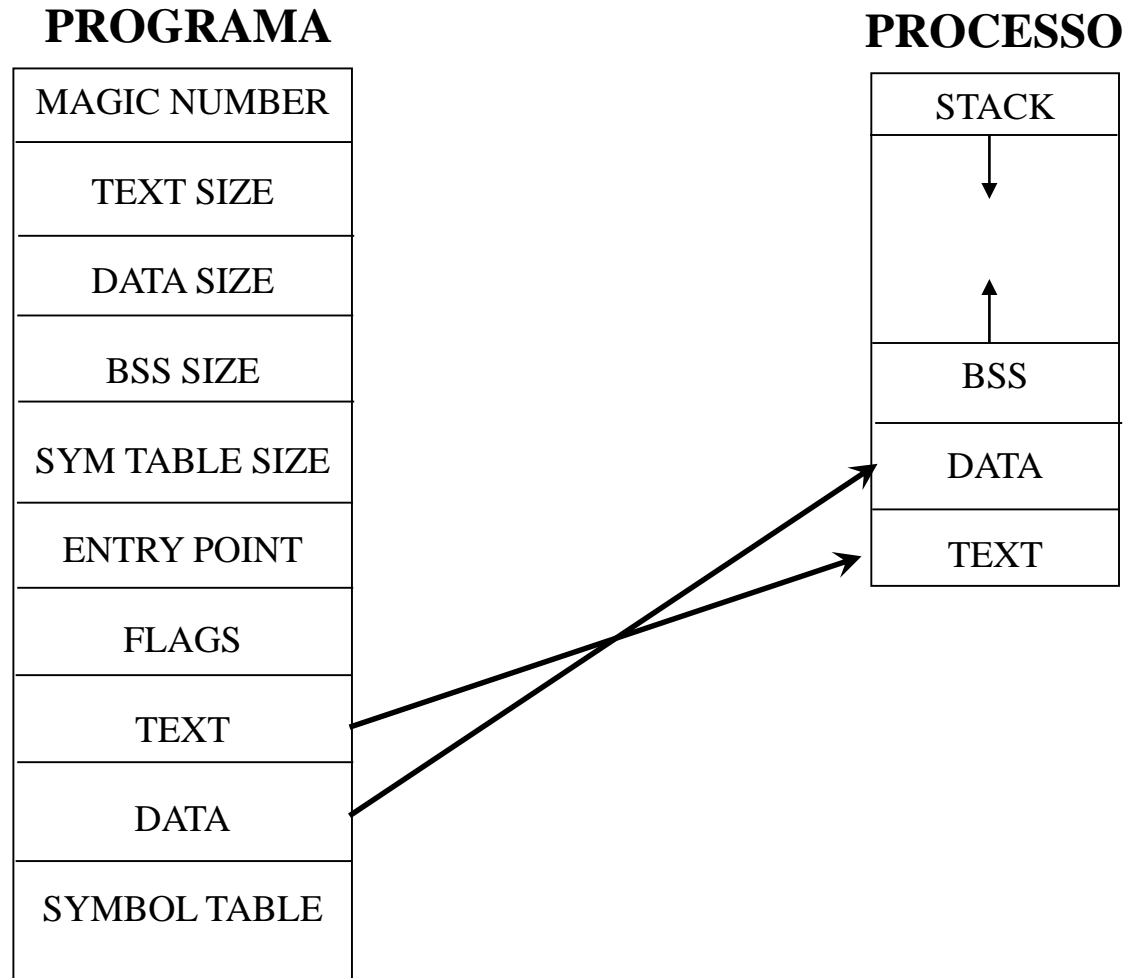
## prog01.exe

```
....
8034 0804 00c0 0000
00c0 0000 0005 0000
0004 0000 0003 0000
00f4 0000 80f4 0804
80f4 0804 0013 0000
0013 0000 0004 0000
....
```

# Programa vs. Processo



# Programa vs. Processo

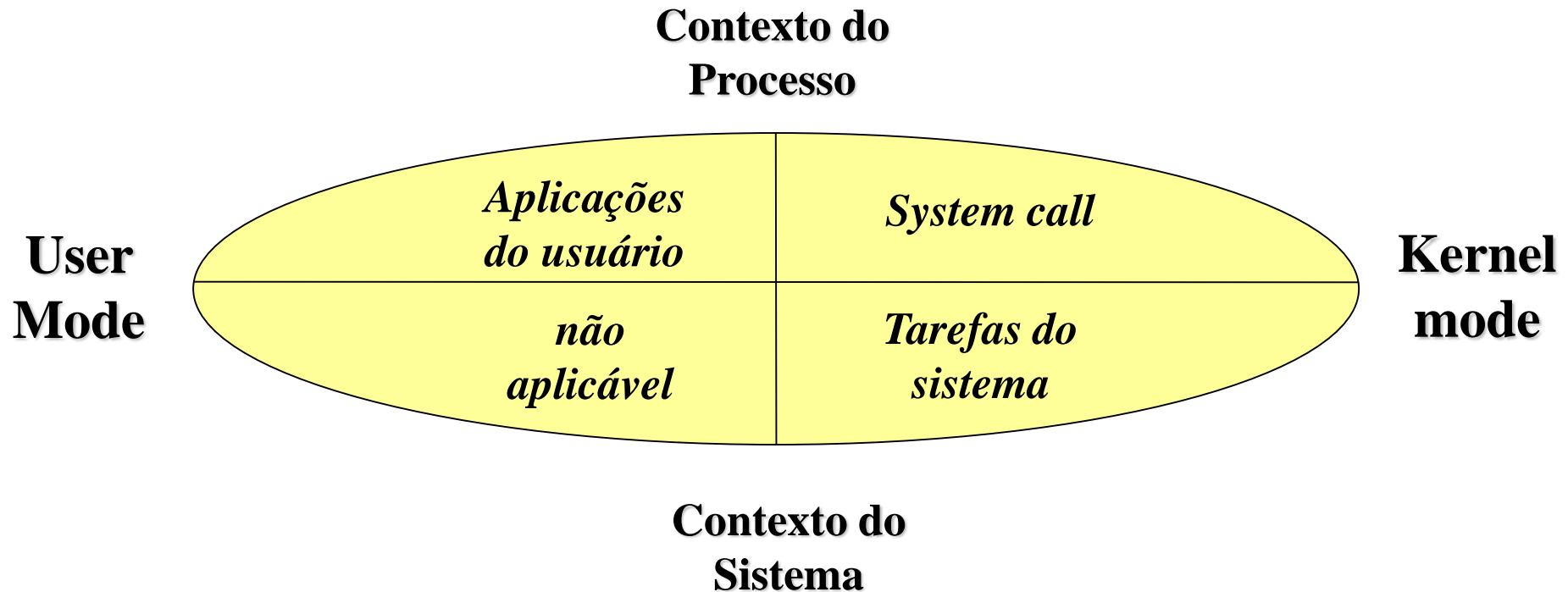


# Processo

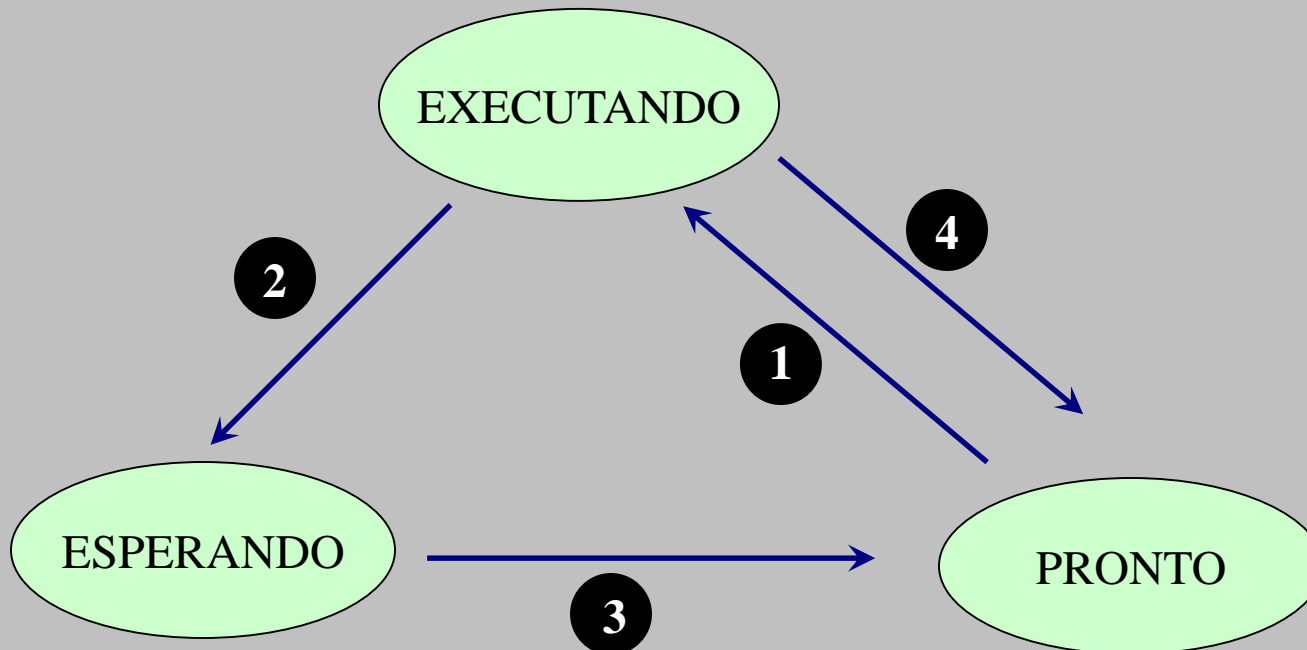
- Principais Características:
  - Modos de Execução
    - Usuário (*user mode*)
    - Kernel (*kernel mode*)
  - Estados
    - Pronto, Esperando, Executando
  - Hierarquia
    - Processo, sub-processo
  - Identificação
    - PID, credenciais



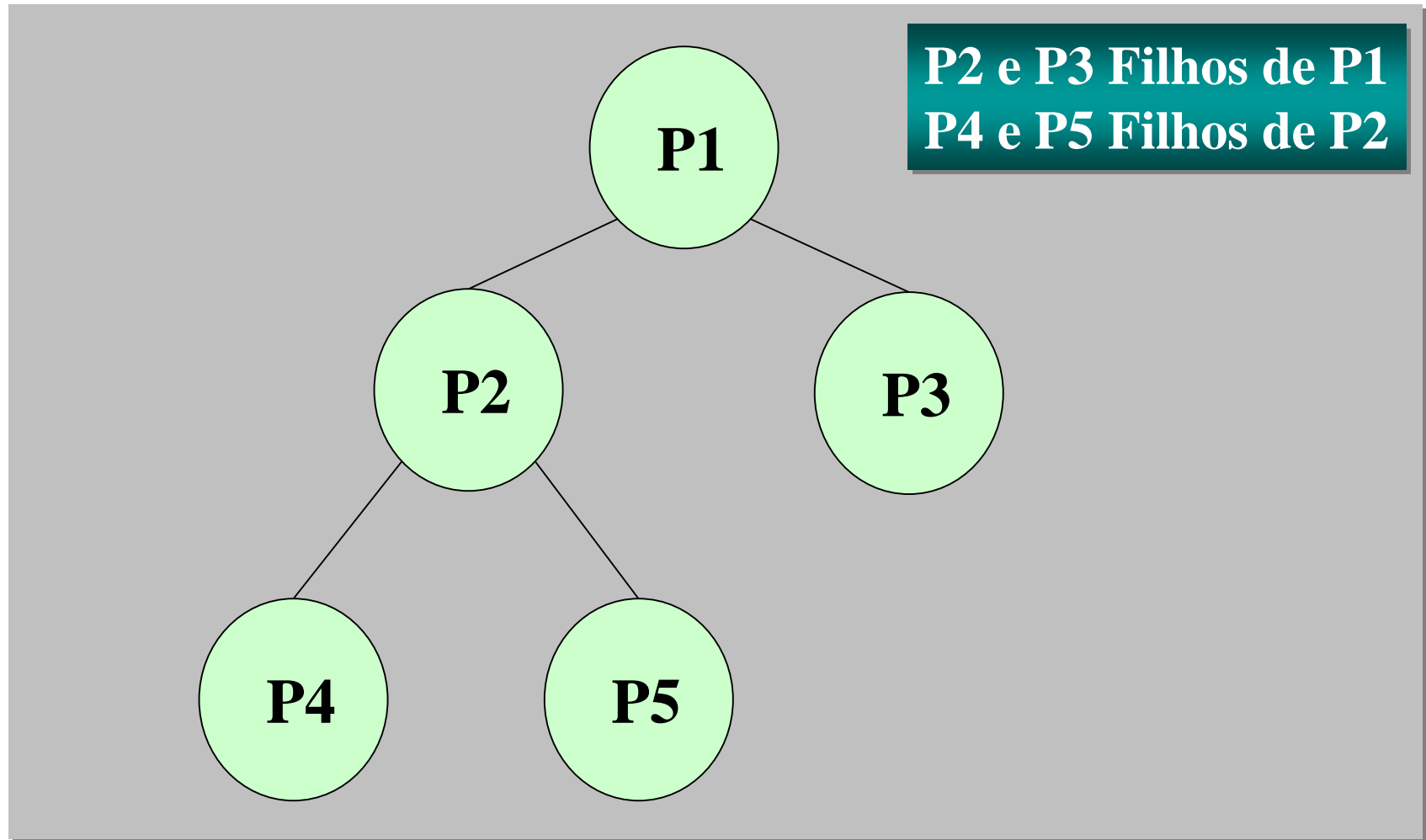
# Modos de Execução/Acesso



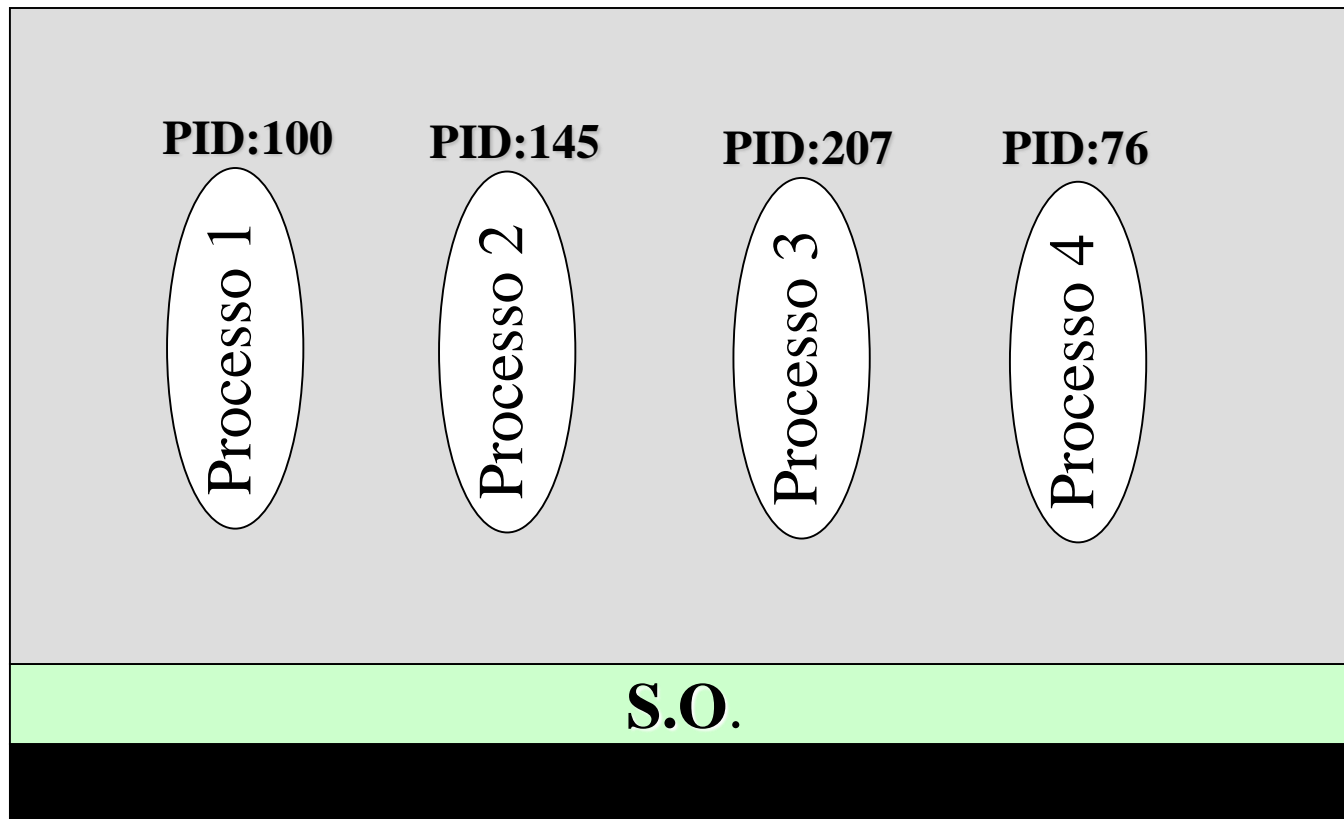
# Estados



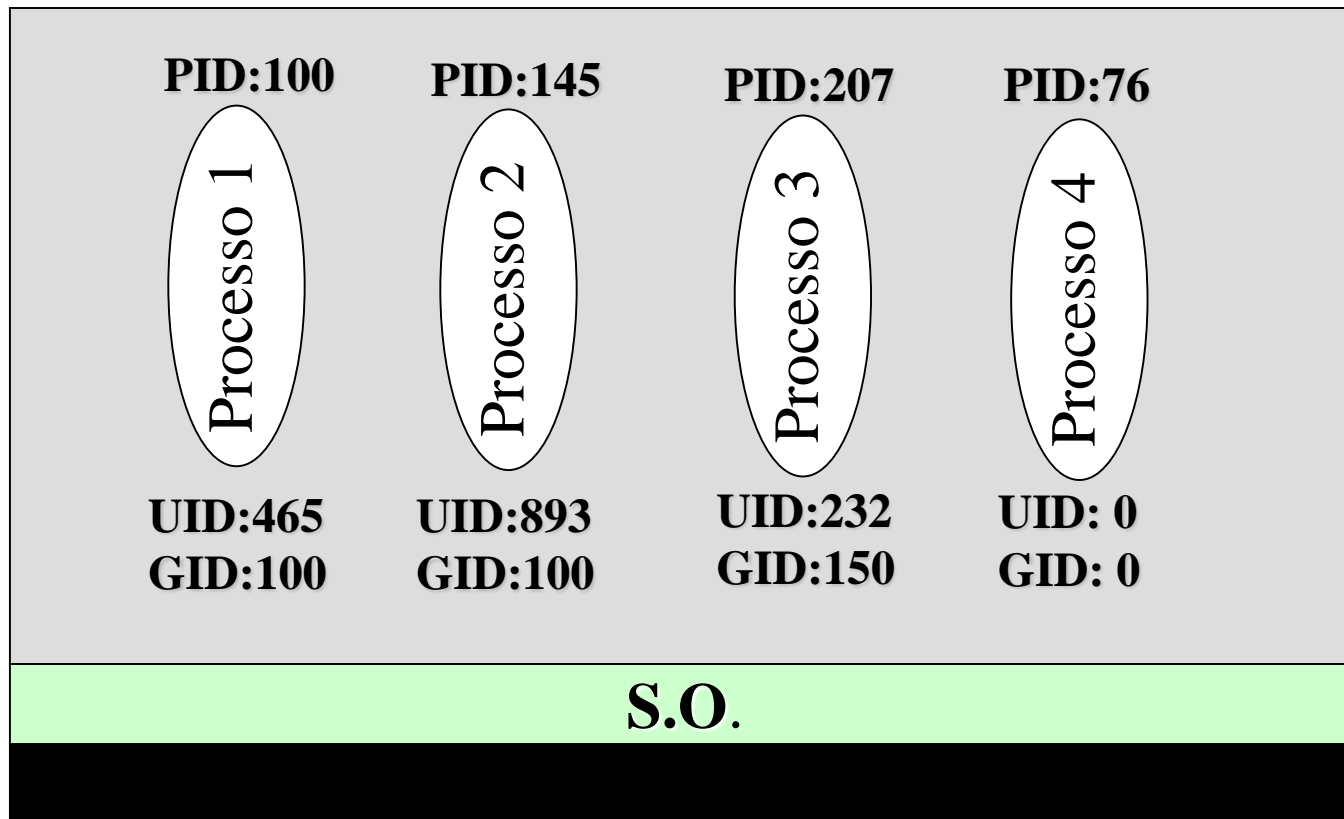
# Hierarquia



# Identificação



# Credenciais



# Gerenciamento

- Identificação+Credenciais:
  - Controle de Acesso;
  - Contabilização de recursos;
  - Quotas de utilização;
  - Logs de auditoria;
  - outros....

# Implementação Interna

- Bloco de controle de processos (PCB)
  - Estrutura interna ao sistema operacional;
  - Armazena todas informações a respeito do processo;
  - Normalmente implementado como uma lista ligada ou vetor de estruturas;
  - Acessível somente pelas rotinas do *kernel*;

# PCB (PROCESS CONTROL BLOCK)

....
Estado do processo
Argumentos da linha de comando
PID, UID, GID,...
Registradores
Limites de memória
Lista de arquivos abertos
....

**O PCB materializa o conceito de processo.**



# PCB (PROCESS CONTROL BLOCK)

PCB#1

....
Estado do processo
Argumentos da linha de comando
PID, UID, GID,...
Registradores
Limites de memória
Lista de arquivos abertos
....

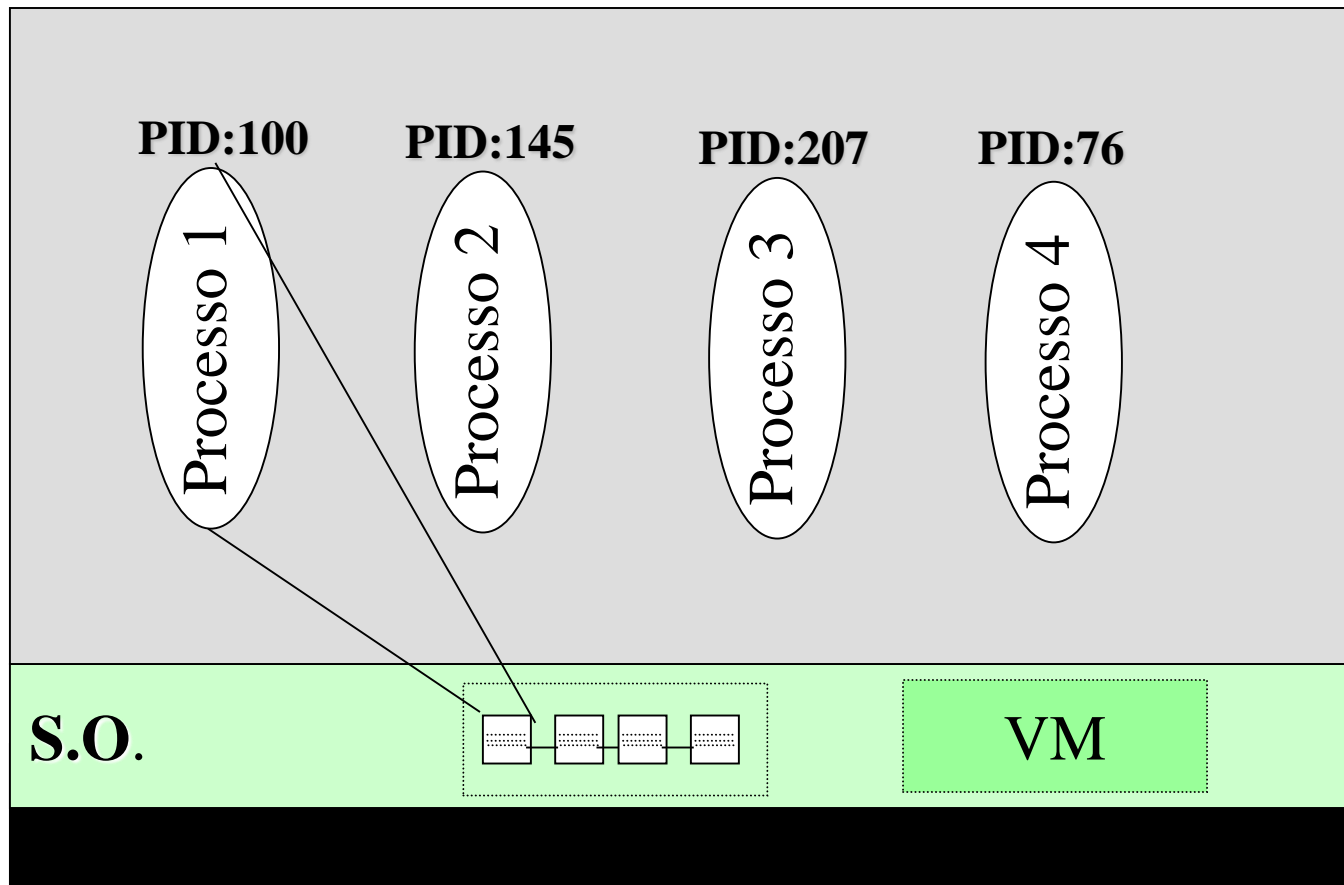
PCB#2

....
Estado do processo
Argumentos da linha de comando
PID, UID, GID,...
Registradores
Limites de memória
Lista de arquivos abertos
....

....



# PCB (PROCESS CONTROL BLOCK)

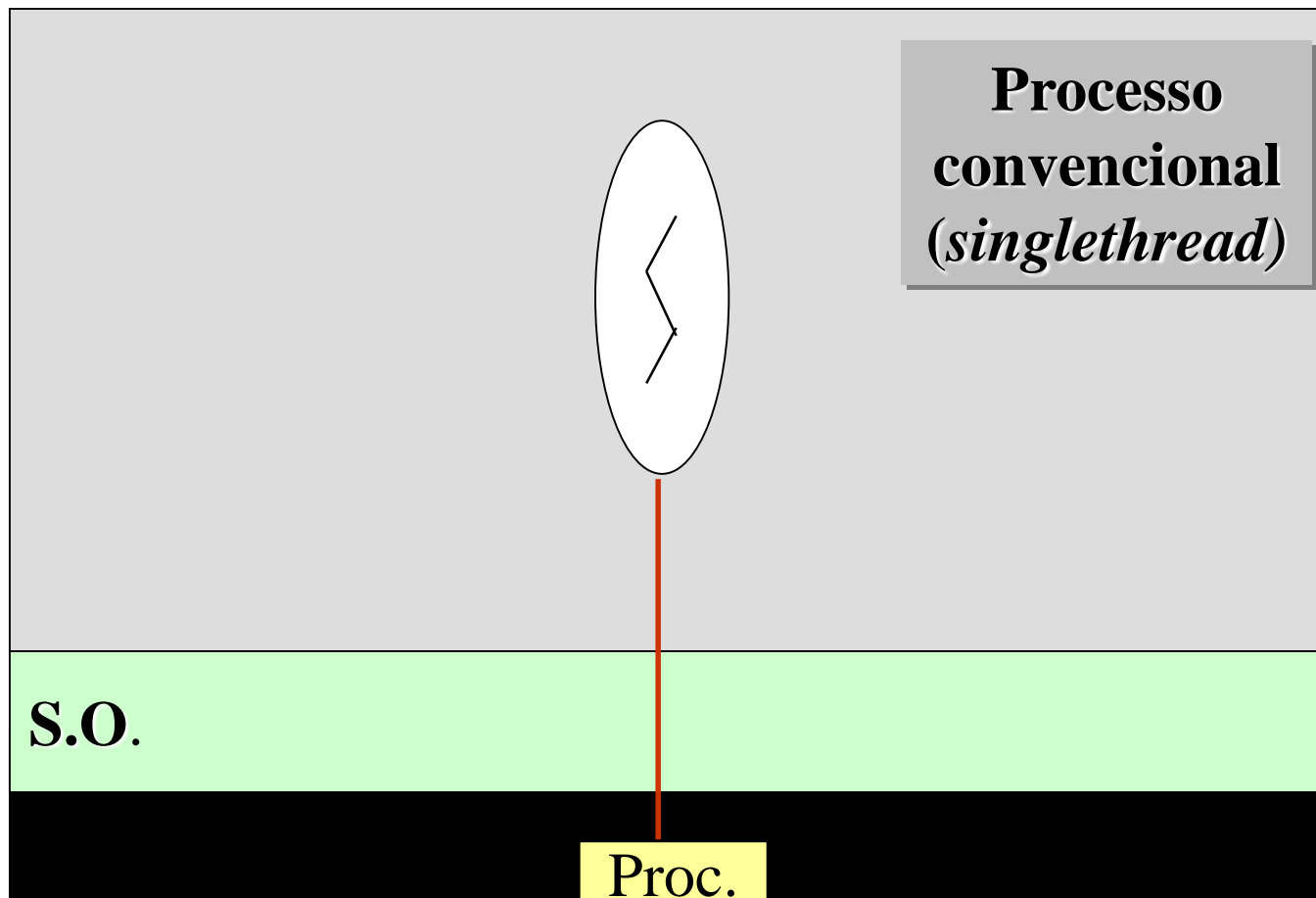


# Contextos

- **HARDWARE**
  - Conteúdo dos registradores:
    - PC (*program counter*);
    - *Stack pointer*;
    - *Flags* de estado;
    - demais registradores;
- **SOFTWARE**
  - Recursos/Atributos do Processo:
    - Arquivos abertos;
    - Tamanho dos *Buffers* de I/O;
    - Tempo de execução;
    - Identificação;
    - Credenciais;
    - outros....

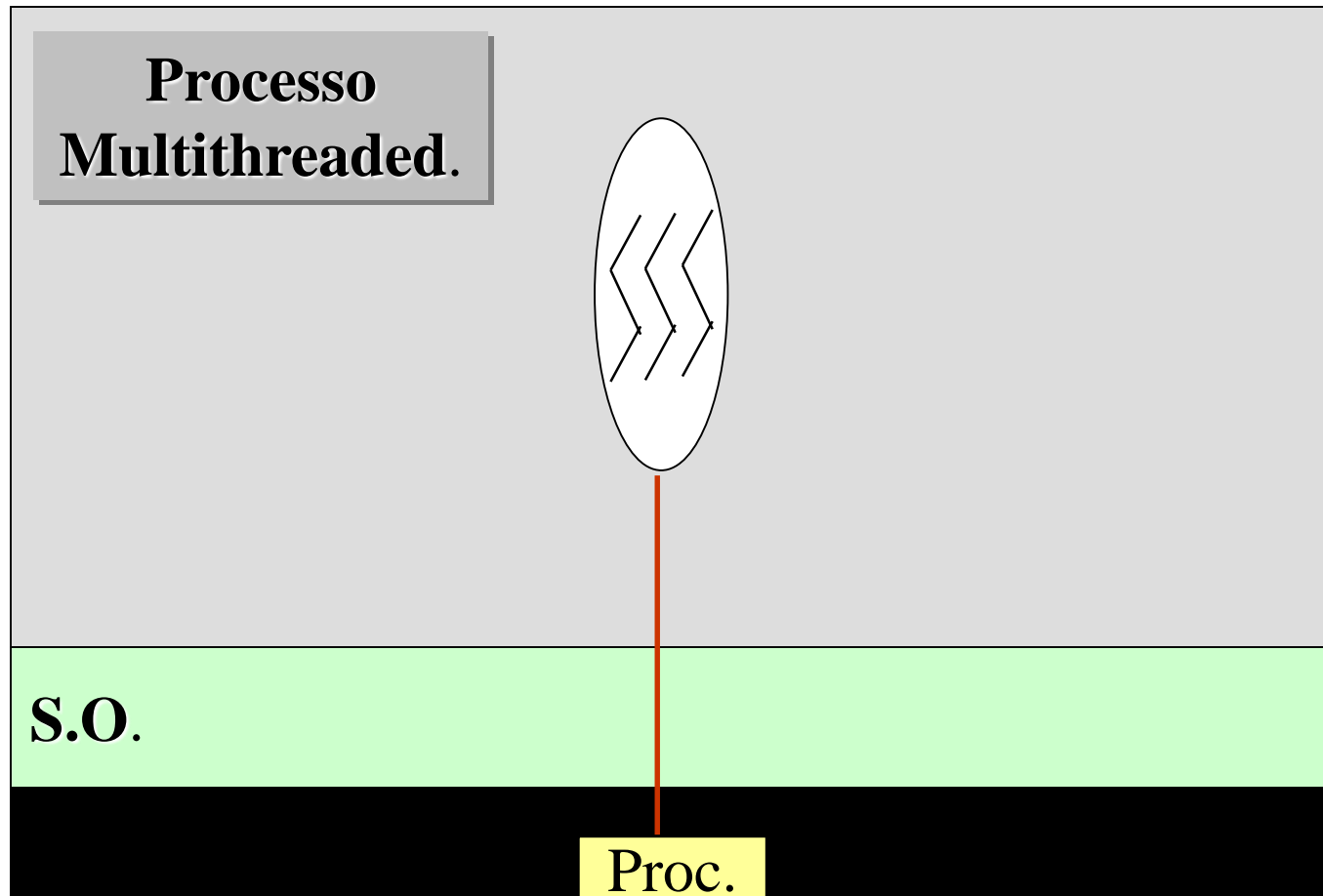
# Definição

- Thread: É um fluxo de execução interno a um processo.



# Definição

- **Multithread:** Processo com mais de uma *thread* de controle.



# Vantagens e Desvantagens

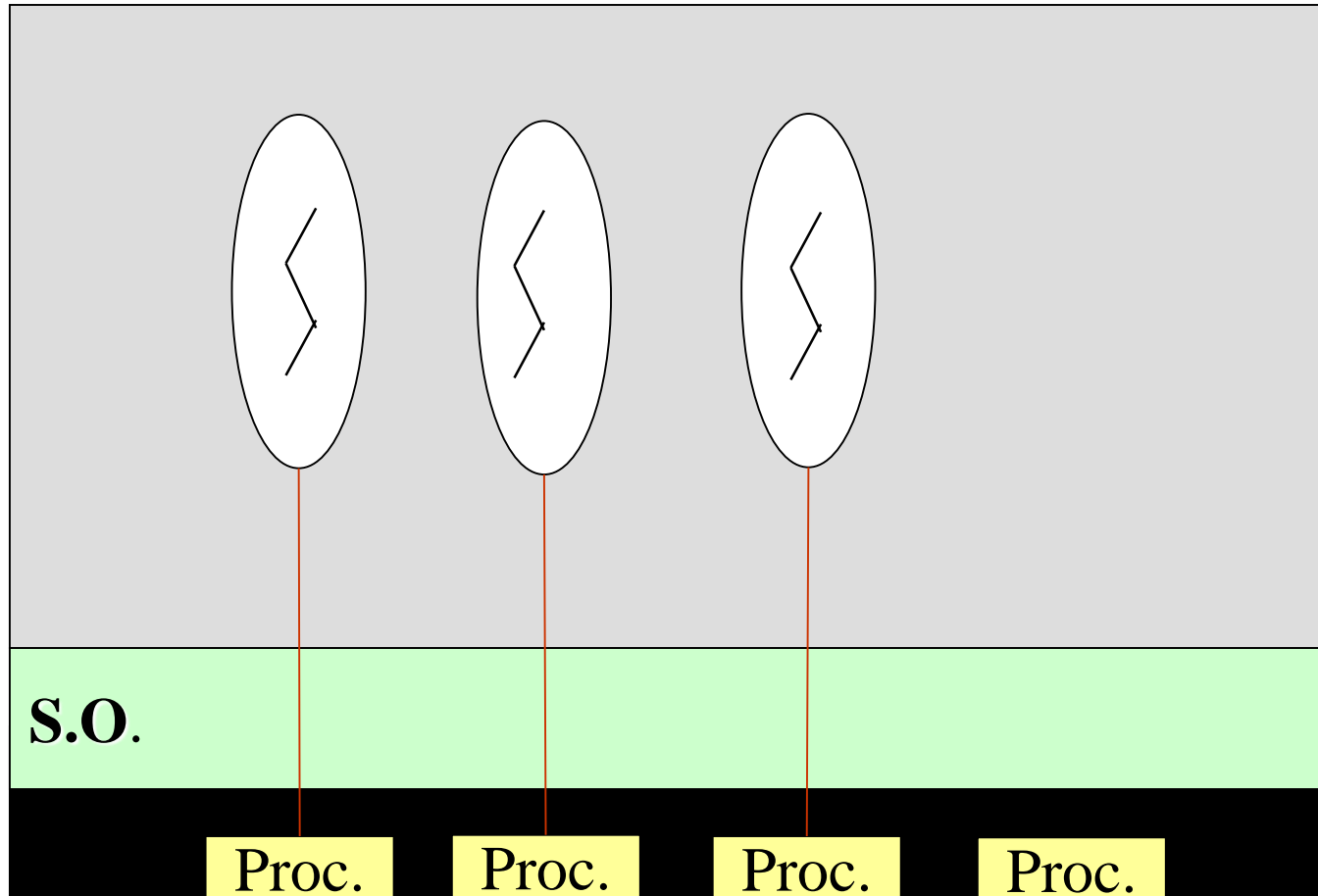
- Vantagens
  - Economia de recursos do sistema;
  - Melhor organização para aplicações com requisitos de concorrência;
  - Uso eficiente de multiprocessadores (\*).

# Vantagens e Desvantagens

- Desvantagens
  - Introduz maior complexidade;

# Paralelismo

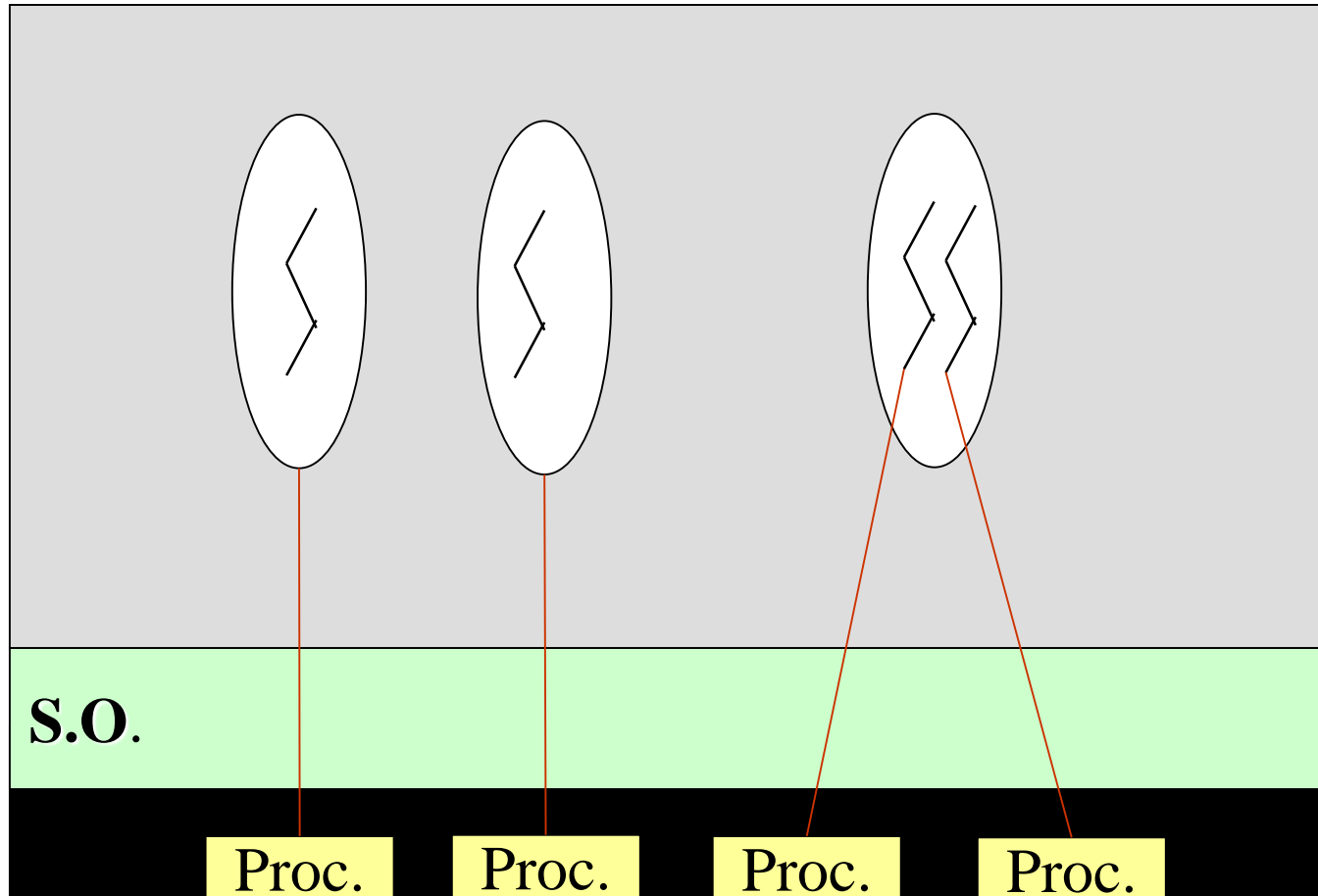
## Processos *SingleThreaded*





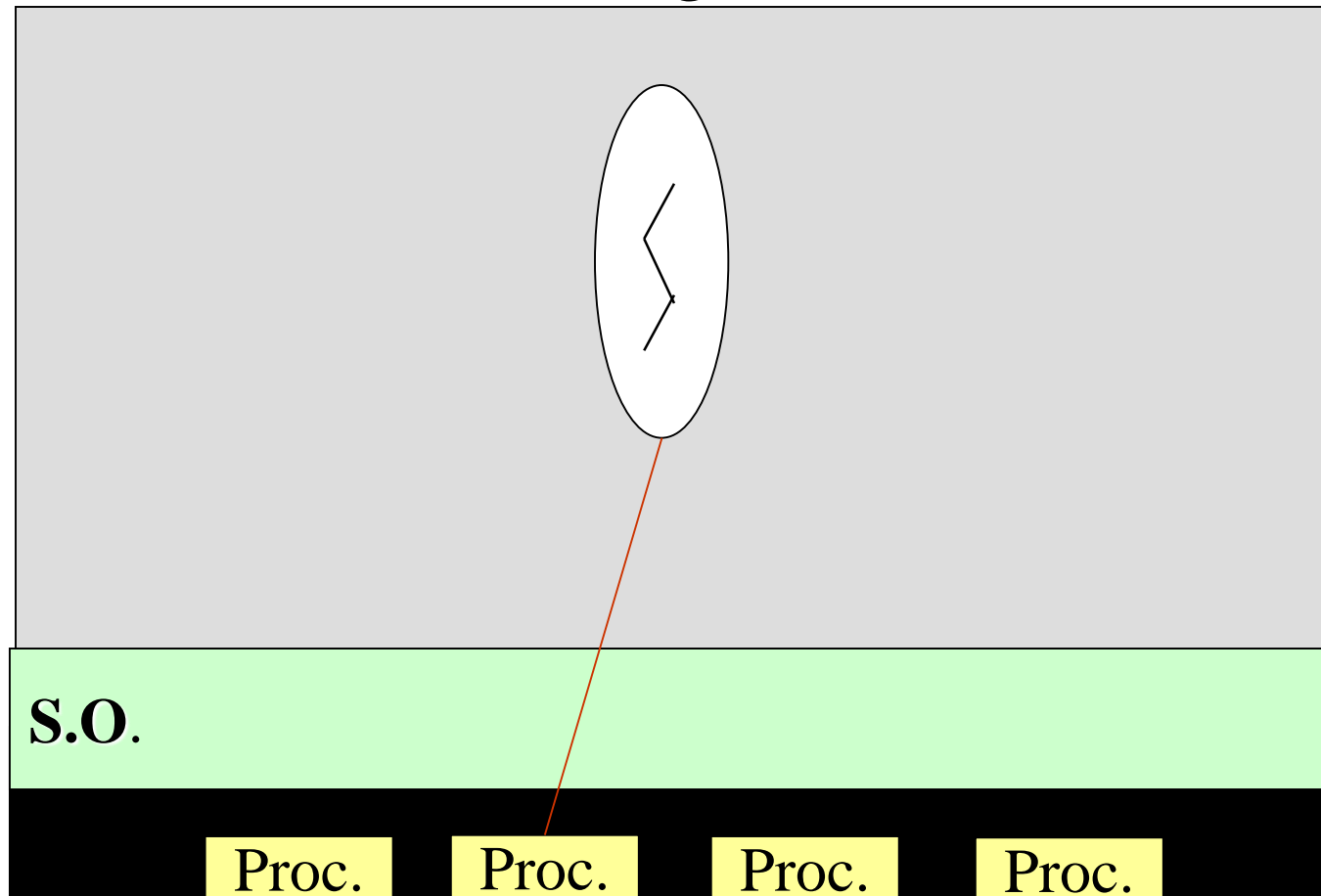
# Paralelismo

## Processos *Multithreaded*



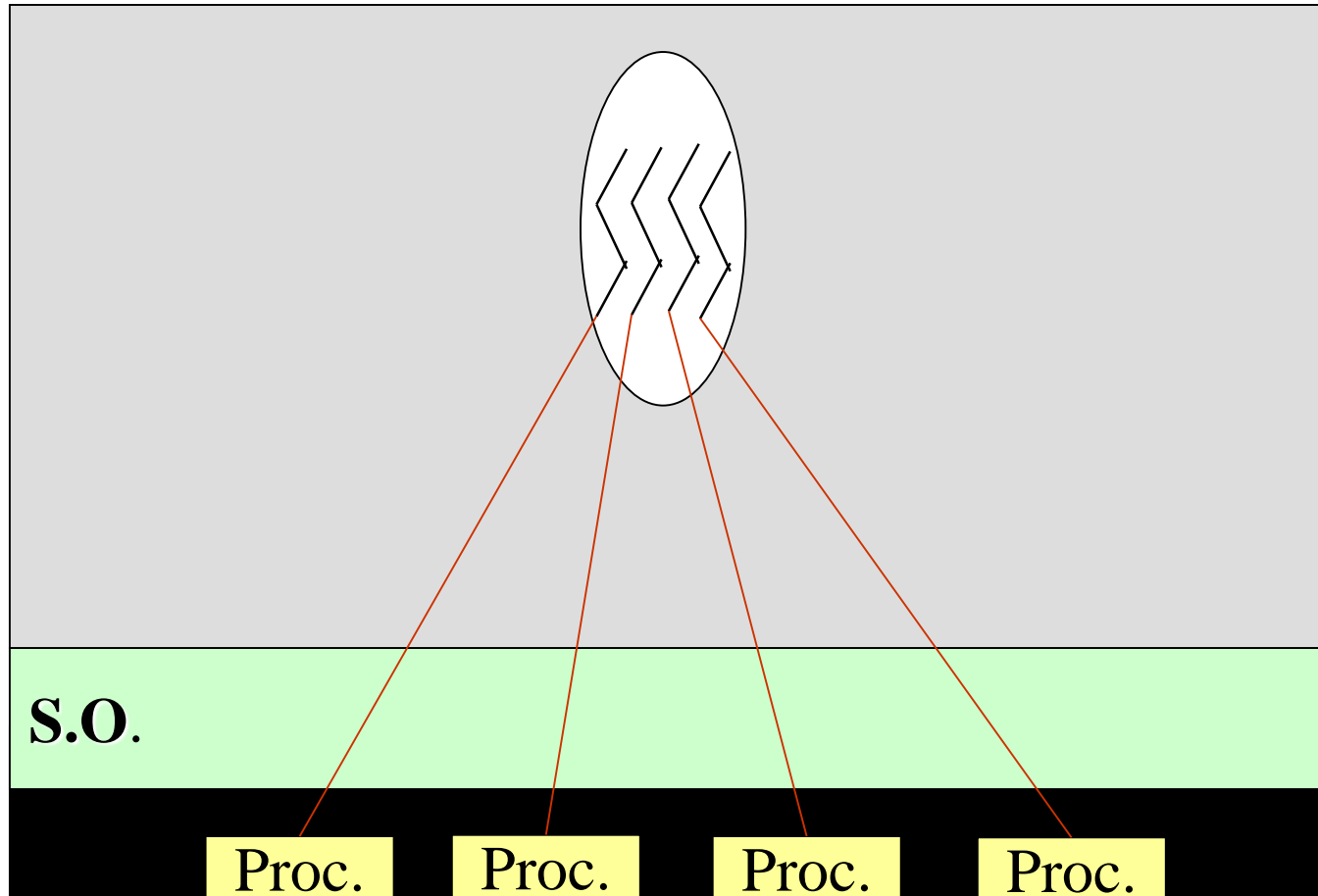
# Paralelismo

## Processo *SingleThreaded*



# Paralelismo

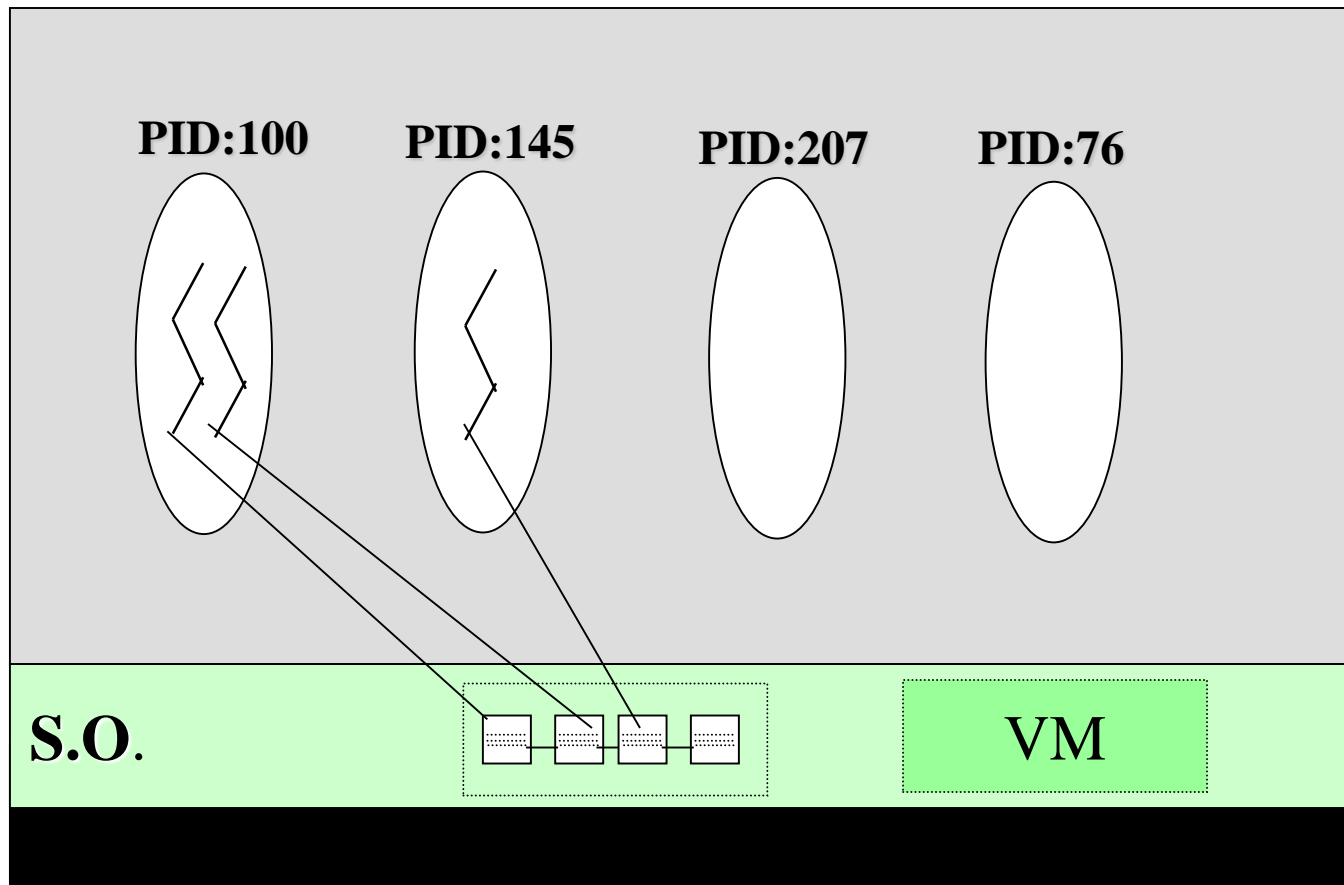
## Processo *Multithreaded*



# TCB (THREAD CONTROL BLOCK)

- Thread ID (tid);
- Registradores de estado (inclui PC e SP);
- Pilha;
- Máscara de sinais;
- Prioridade;
- Armazenamento privado
- Outros...

# TCB (THREAD CONTROL BLOCK)

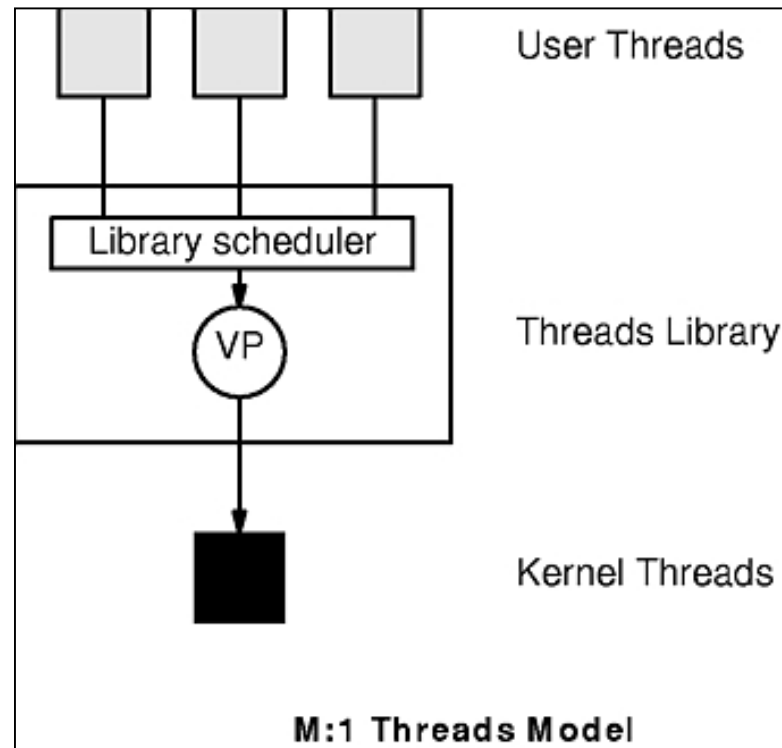


# Modelos de Threads

- Os principais modelos de implementação de threads são
  - User-space threads ( $M \times 1$ )
  - Kernel-space threads ( $1 \times 1$ )
  - User-space and kernel-space threads ( $M \times N$ )
  - POSIX threads (pthreads)
    - *pode ser usado com qualquer modelo acima.*

# Modelos de Threads

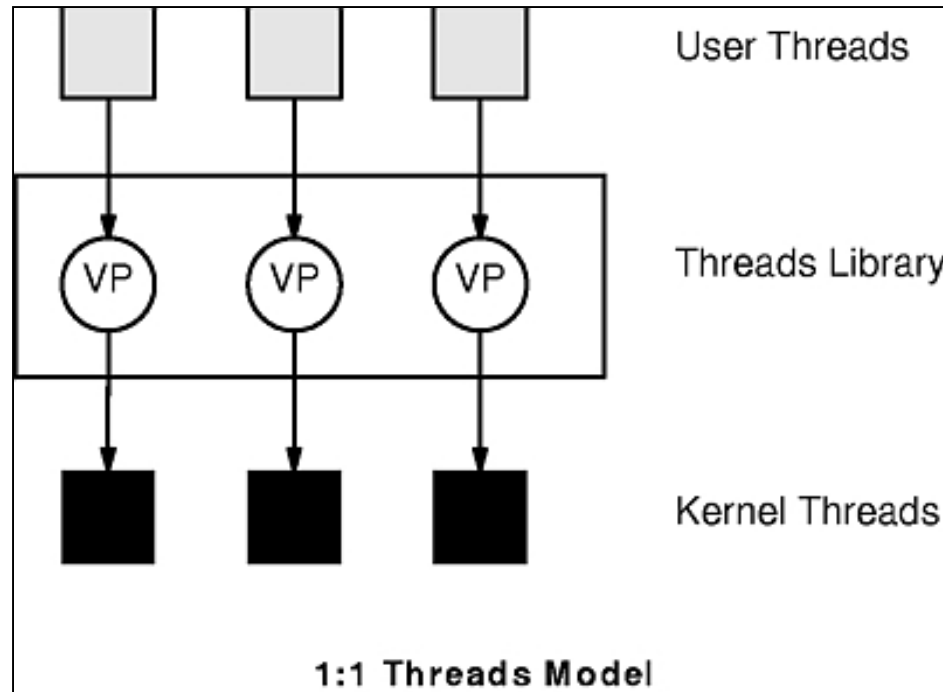
- User-space threads (M x 1)



VP (virtual processor) é também chamado de LWP (*light-weight process*)

# Modelos de Threads

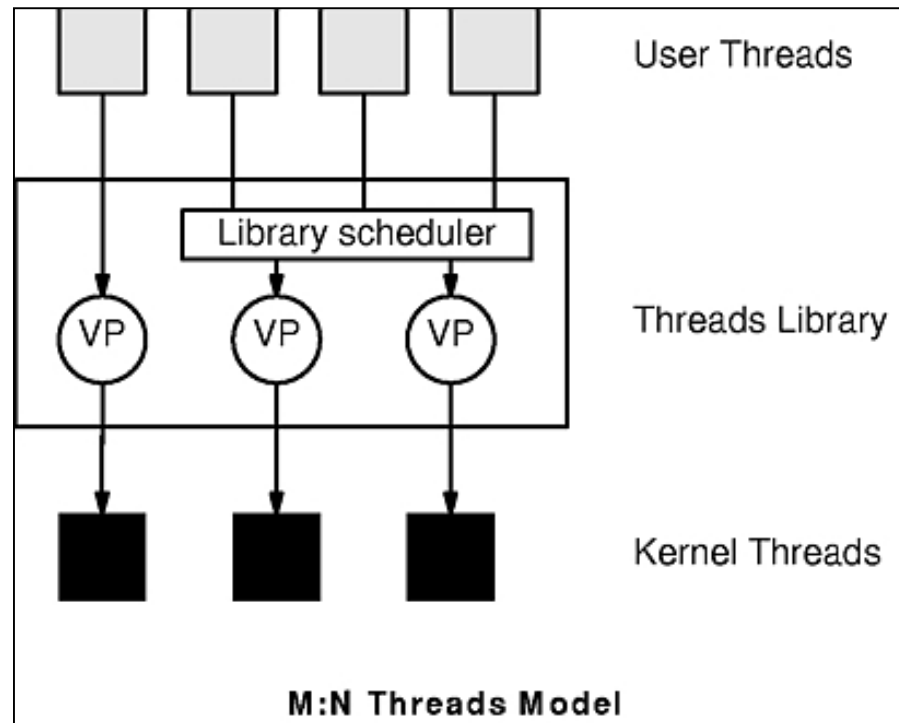
- Kernel-space threads (1x1)





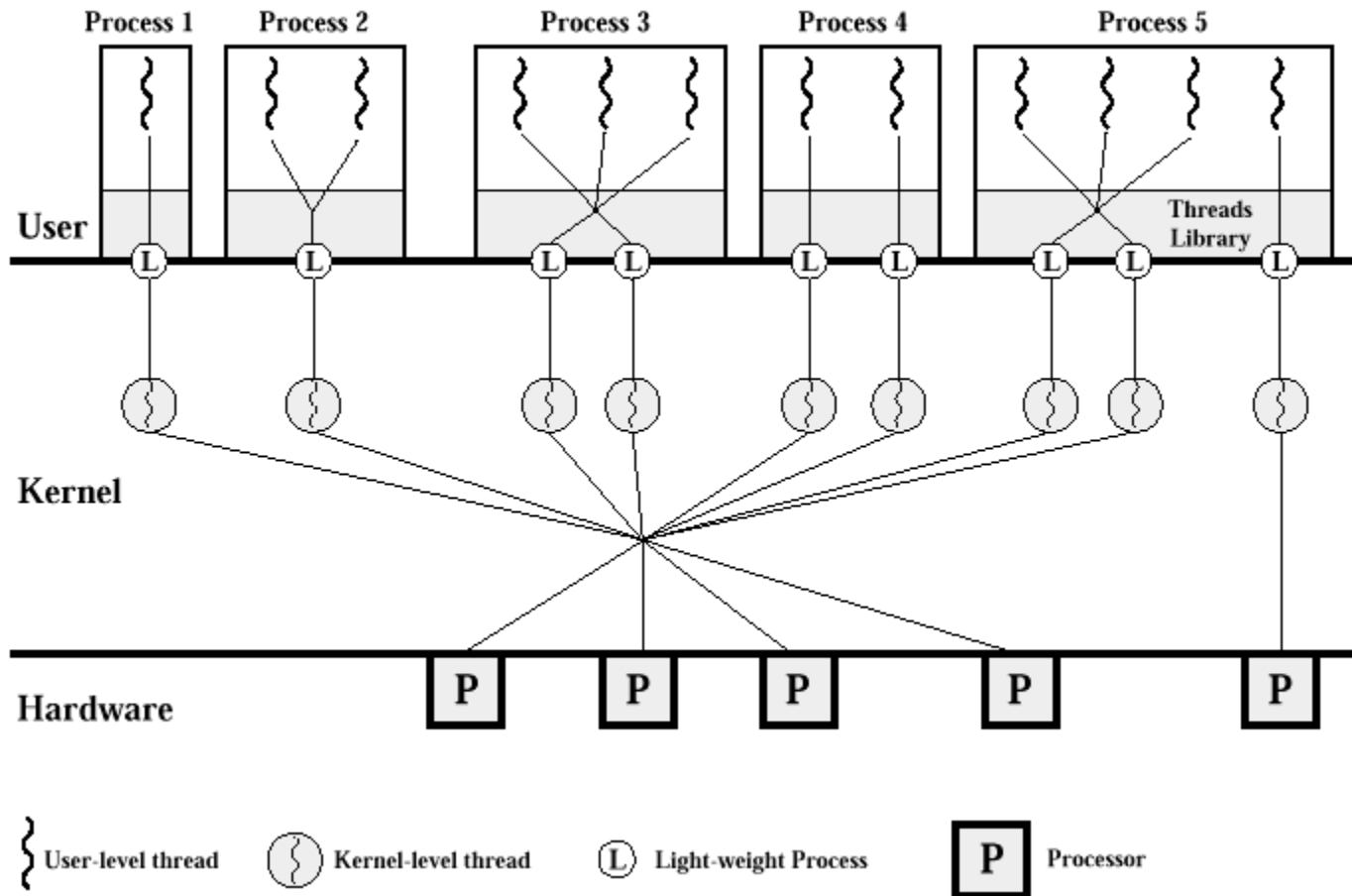
# Modelos de Threads

- User-space and Kernel-space threads (MxN)



# Modelos de Threads

- Modelo de Threads no UNIX Solaris



# **Unidade II:**

## **Exemplo de Criação de Processos/Threads**

# Criando um Processo (fork)


- Uso: `pid_t fork(void);`
- A chamada de sistema `fork()` retorna um valor zero para o filho e o valor do PID do filho para o Pai.
- O pai normalmente espera o filho terminar antes de encerrar (usando `wait()` ).
- O filho retorna um status de término para o Pai.

```
int pid;
int status = 0;
pid = fork();
if (pid > 0) {
    /* Pai */
    ....
    pid = wait(&status);
}
else if (pid == 0) {
    /* Filho */
    ....
    exit(status);
}
else {
    printf("Erro\n");
}
```

# Criando um Processo

- Após ser criado, um processo filho pode executar um novo programa (diferente do pai) por chamar `exec()`.
  - `fork()` e `exec()` são usadas em conjunto para executar um programa qualquer.

```
int pid;
int status = 0;
pid=fork();
if (pid > 0 ) {
    /* Pai */
    ....
    pid = wait(&status);
} else if ( pid == 0)
    /* Filho*/
    execve("prog",...);
    ...
}
```

 retorna -1

# Criando um Processo

- Outra chamada de sistemas usada para criação de processos é o **vfork()**.
  - *Possui mesma sintaxe de utilização do fork().*
- vfork() bloqueia o processo Pai até que o filho chame exec().
- É uma chamada criada para evitar o *overhead* de um fork() para processos que irão invocar exec() imediatamente após serem criados.

# Criando um Processo

## Exercícios Práticos (1)

1. Comparar o tempo de execução de dois programas que criam 500 processos filhos. O primeiro usa **fork()** e o segundo **vfork()**.
2. Faça um programa que crie 5 processos filhos, onde cada um execute um programa externo. O programa externo executado deve imprimir na tela os valores pares de 1 até 5000.
3. Faça um programa que crie 5 processos filhos. Antes de finalizar o Pai e os filhos veja como está a hierarquia de processos no sistema (comando: `ps tree`).
4. Repita o exercício anterior, porém mantenha os processos filhos executando e finalize o processo Pai. Veja como fica a hierarquia de processos nesse caso.

# Estudo de caso

- Considere o Programa abaixo:

```
int main()
{
    pid_t pids[2];
    int i, status;

    printf("PID=%d: Processo Pai\n",getpid());

    for(i=0; i < 2; i++ )
    {
        pids[i] = fork();
        printf("PID=%d: i=%d\n",getpid(), i);
    }
    printf("Finalizando - PID=%d: i=%d\n",getpid(),i);
}
```

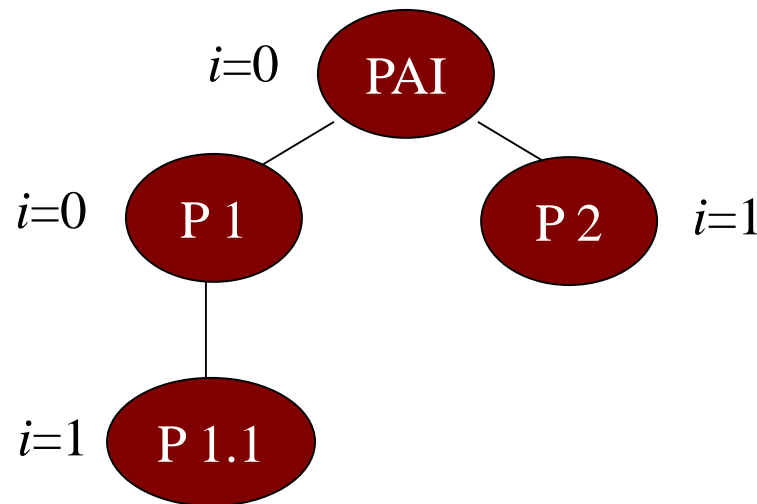


# Estudo de caso

```
int main()
{
    pid_t pids[2];
    int i, status;

    printf("PID=%d: Processo Pai\n",getpid());

    for(i=0; i < 2; i++ )
    {
        pids[i] = fork();
        printf("PID=%d: i=%d\n",getpid(), i);
    }
    printf("Finalizando - PID=%d: i=%d\n",getpid(),i);
}
```

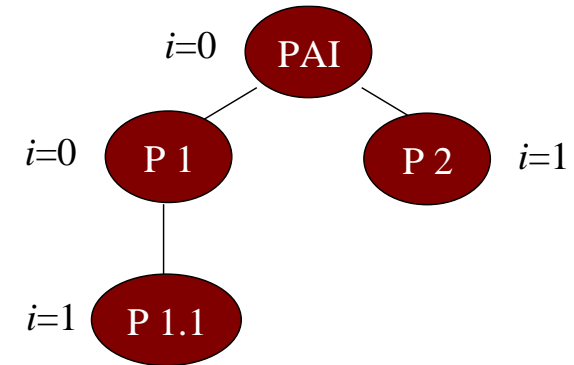


# Estudo de caso

```
int main()
{
    pid_t pids[2];
    int i, status;

    printf("PID=%d: Processo Pai\n",getpid());

    for(i=0; i < 2; i++ )
    {
        pids[i] = fork();
        printf("PID=%d: i=%d\n",getpid(), i);
    }
    printf("Finalizando - PID=%d: i=%d\n",getpid(),i);
}
```



```
PID=5578: Processo Pai
PID=5579: i=0
PID=5578: i=0
PID=5581: i=1
Finalizando - PID=5581: i=2
PID=5578: i=1
Finalizando - PID=5578: i=2
PID=5580: i=1
Finalizando - PID=5580: i=2
PID=5579: i=1
Finalizando - PID=5579: i=2
```

```
PID=5582: Processo Pai
PID=5583: i=0
PID=5584: i=1
Finalizando - PID=5584: i=2
PID=5583: i=1
Finalizando - PID=5583: i=2
PID=5582: i=0
PID=5585: i=1
Finalizando - PID=5585: i=2
PID=5582: i=1
Finalizando - PID=5582: i=2
```

# Criando um Processo

Comando ps(3) – dicas de uso.

Imprime os processos criados pelo usuário

**# pstree -c -p rivalino**

```
bash(19111)-+-prog1(19839)-+-prog1(19840)---prog1(19841)
              |
              `--prog1(19842)---prog1(19843)
                  `--pstree(23829)
```

# Criando Threads (pthreads)

- Uso:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,  
void * (*start_routine)(void *), void *arg);
```

**thread** – retorna o thread id

**attr** – NULL para os valores default.

**start\_routine** – ponteiro para a função que será executada pela thread.

**arg** – ponteiro para o argumento da função. Para passar múltiplos argumentos, use um ponteiro para estrutura (*struct*)

# Criando Threads (pthreads)

- Uso:

```
#include <pthread.h>
```

```
int pthread_join(pthread_t th, void **thread_return);
```

**th** –thread chamadora suspensa até que a thread identificada por **th** termina.

**thread\_return** – Se o retorno da thread **th** é diferente de NULL, então o valor de retorno é armazenado em **thread\_return**.

# Criando Threads (pthreads)

- Uso:

```
#include <pthread.h>
```

```
void pthread_exit(void *retval);
```

**retval** – valor de retorno da thread.

- Essa rotina encerra a thread, portanto, ela nunca retorna.

# Criando Threads (pthreads)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *print(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Ola', Eu sou a thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int r, i;
    for(i=0; i<NUM_THREADS; i++)
    {
        printf("Criando thread nr%ld\n", i);
        r = pthread_create(&threads[i], NULL, print, (void *)i);
        if (r){
            printf("ERRO; o return code da pthread_create() eh %d\n", r);
            exit(-1);
        }
    }
    getchar();
}
```

# Criando Threads (pthreads)

```
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10
void *thread_function(void *);

int counter = 0;
main(){
    pthread_t thread_id[NTHREADS];
    int i, j;
    for(i=0; i < NTHREADS; i++)
    {
        pthread_create( &thread_id[i], NULL, thread_function, NULL );
    }

    for(j=0; j < NTHREADS; j++)
    {
        pthread_join( thread_id[j], NULL);
    }

    printf("Valor final do contador: %d\n", counter);
}

void *thread_function(void *ptr){
    printf("Thread nr. %ld\n", pthread_self());
    counter++;
    pthread_exit(NULL);
}
```



# Threads

## Exercícios Práticos (2)

1. Comparar o tempo de execução de um programa para criar 500 processos e 500 threads.
2. Faça um programa *Singlethread* para contar o número de caracteres '@' existe nos arquivos Arq1, Arq2, ..., Arq10. Compare o tempo de execução do programa *Singlethread* com um programa *Multithread*, onde cada arquivo é processado por uma *thread*.

# Kernel Preemptivo (*Preemptive kernel*)

- Nos primeiros sistemas UNIX, o kernel não era preemptivo (*non-preemptive kernel*).
  - *ou seja, um processo/thread em kernel mode não poderia perder o processador por preempção*
- Nos sistemas operacionais mais modernos (inclusive Linux 2.6.x) o kernel passou a ser preemptível
  - *Um processo/thread pode sofrer preempção a qualquer momento.*

# Kernel Preemptivo (*Preemptive kernel*)

