

# Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte  
M<sup>a</sup>. Cecília Gomes

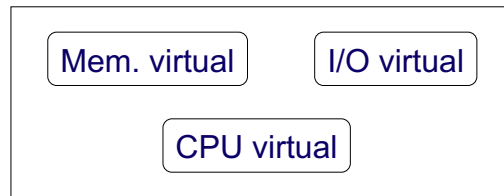
1

## Aula 7

- Processos leves: fluxos de execução (*execution threads*)
- OSTEP: cap. 26.0, 26.1, 26.2, 27.1, 27.2

2

## Máquina virtual de um processo



- O SO gere e mapeia na máquina real
  - Mem. Virtual → mapa de memória
  - CPU virtual → *time-sharing*
  - I/O virtual → exemplo: canais de I/O

## Multiplas tarefas num programa

- Um programa pode compreender várias tarefas:
  - Facilidade no desenvolvimento (Modularidade)
  - Estas podem ser operações independentes (ou quase)
    - Não têm ordem entre elas
    - Se uma bloquear outras podem continuar (Multiprogramação interna)
  - Aceleração das computações (paralelismo)
    - Tirando partido de vários CPUs
- Mas o estado e recursos devem ser comuns:
  - Com uso de processos a comunicação entre estes será enorme
  - Os recursos deviam manter-se partilhados: memória, I/O, etc.

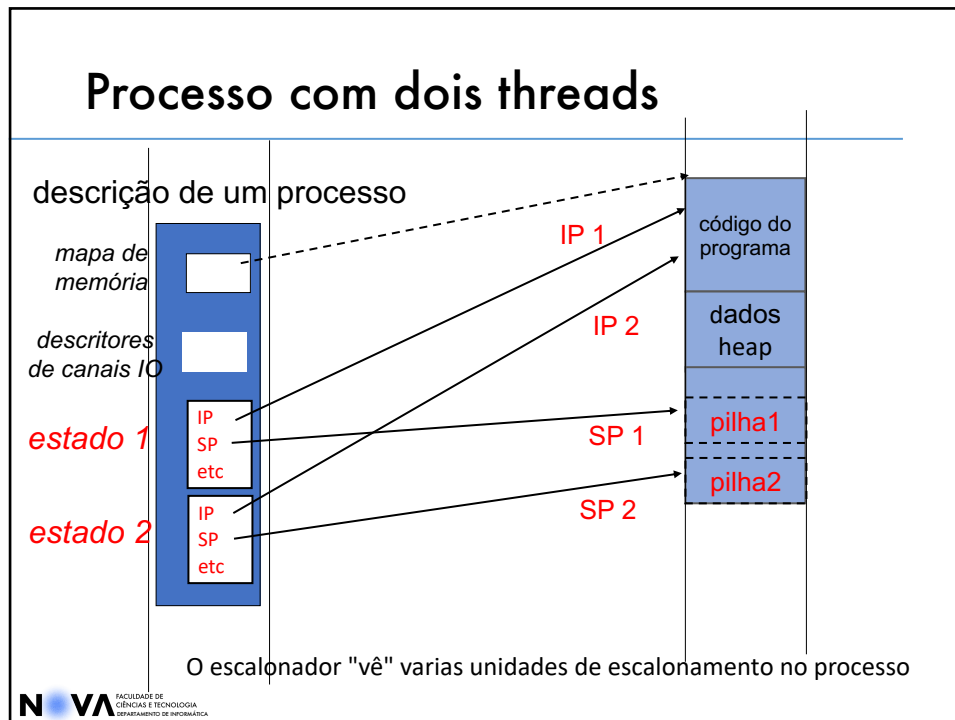
## Processos

- Multiprocessamento → processos
  - Múltiplos programas
  - Instâncias do mesmo programa
- Unidade de gestão de recursos
  - memória: dados, código e *stack*
    - estado da memória: mapa de memória e seu conteúdo
  - comunicação (I/O): ficheiros, IPC, outras abstrações...
    - estado das comunicações: canais de I/O, etc...
- Unidade de execução concorrente
  - sequência (fio ou fluxo) de execução de instruções (*execution thread*)
    - estado de execução: estado do CPU, contexto no *Stack*
- *Porque não ter vários estados de execução num processo?*

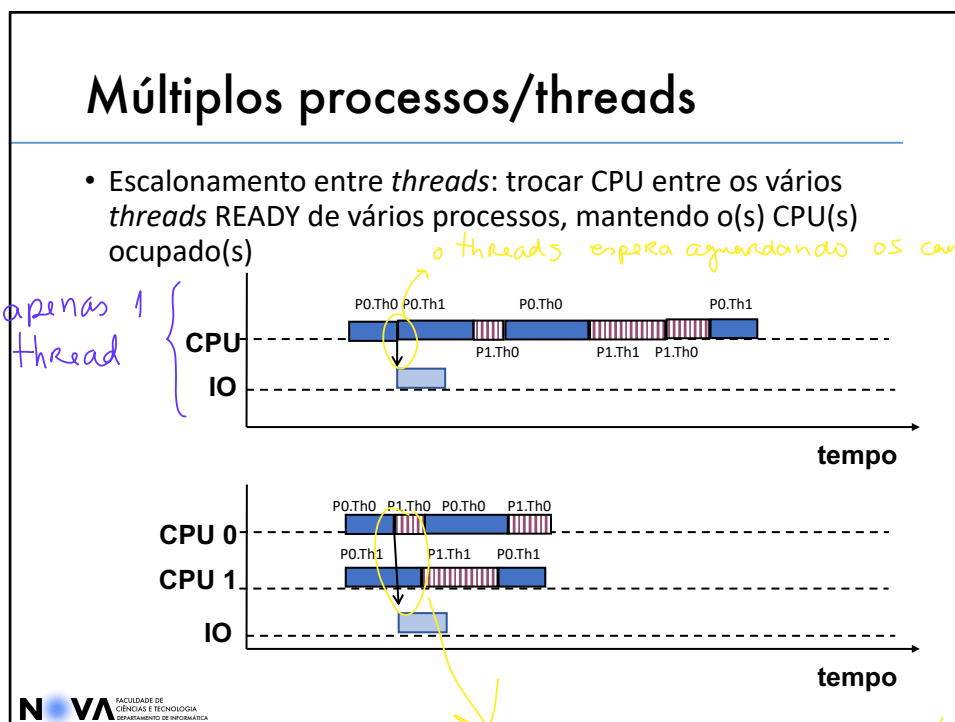
## Threads concorrentes

- Cada processo é criado com um fio/fluxo de execução (*execution thread*)
  - este executa a partir do `fork()`
- Multiprocessamento interno a um processo
  - criando novos fios de execução
  - ou seja, várias execuções dentro do mesmo processo
    - vários estados de CPU (vários IP, etc)
    - vários Stacks → chamada de funções e variáveis locais por thread
  - todos os recursos do processo se mantêm partilhados
    - memória, I/O, etc.

6 cada thread filho tem uma pilha associada → cada um tem o seu stackpointer  
↓  
mesmo código



7

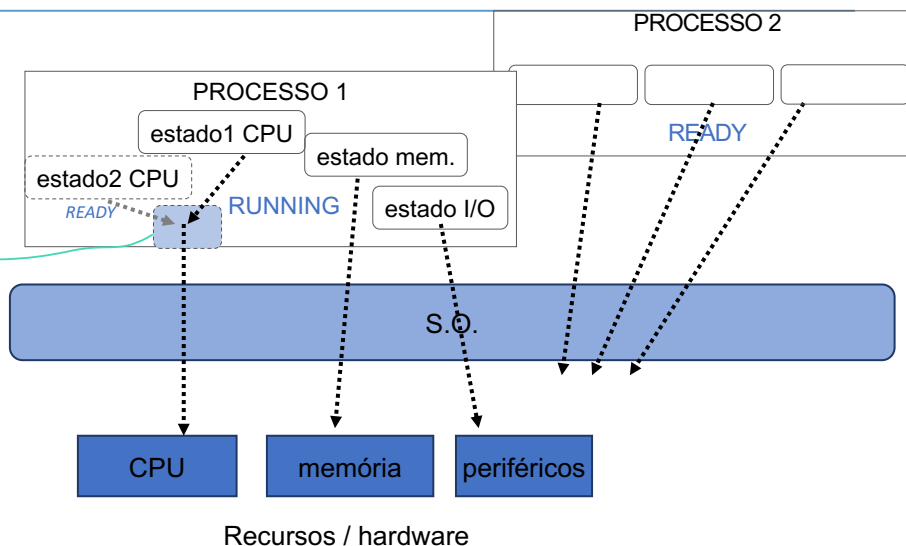


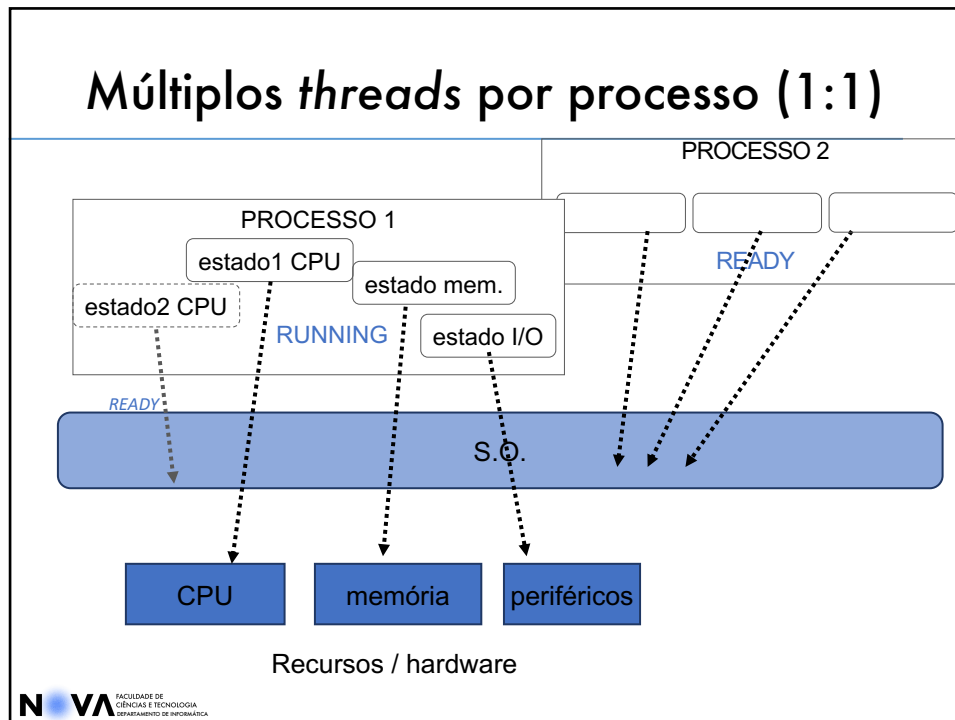
8

## Modelos de implementação

- Nível do processo / biblioteca (N:1)
  - o código do processo **passa a incluir um mini-sistema que suporta os threads desse processo**
  - o escalonamento de threads é **interno ao processo sem usar o SO, assim como as sincronizações, etc...**
- Nível do kernel (1:1)
  - o **SO suporta a noção de threads em processo**
  - o **escalador do SO também gere os threads**
- Híbrido (N:M)
  - vários threads do kernel (LWP), cada um suportando vários threads de nível do processo
  - possivelmente com o mapeamento configurável

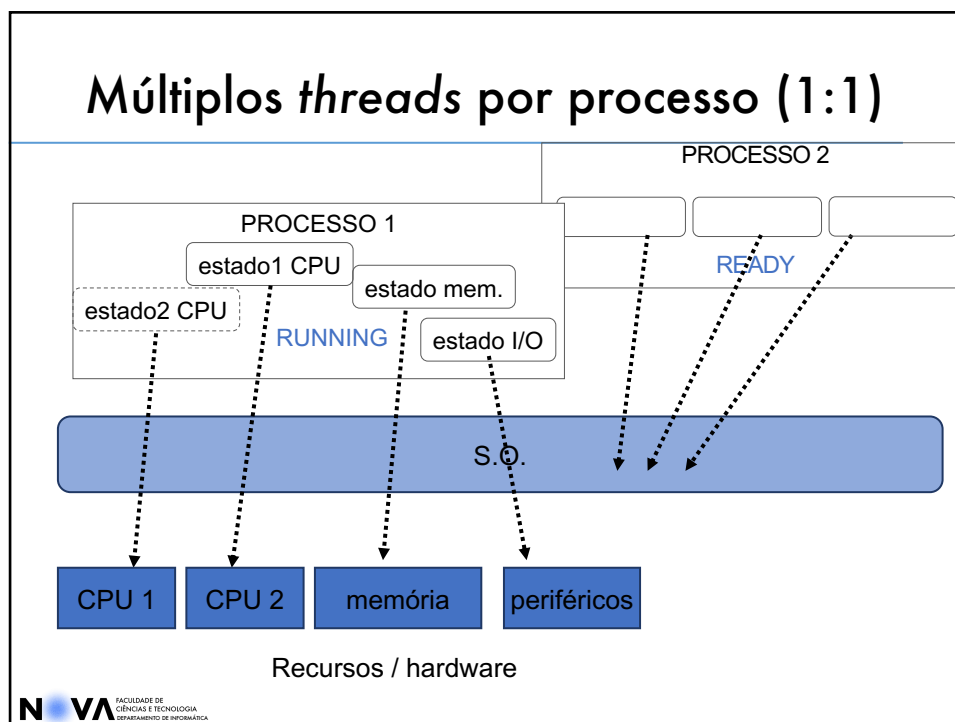
## Múltiplos threads por processo (N:1)





O próprio processo sabe de todos os estados e o SO gerencia os processos e os threads

11



Se houver + que 1 CPU o SO gere qual processo/thread vai para qual

12

## Comparação threads vs processos

- Criação de um novo thread
  - muito mais fácil e rápida
- Comutação de contexto entre threads no mesmo processo
  - muito mais fácil e rápido
- Partilha de informação
  - toda a memória e I/O pode ser partilhado
  - muito mais fácil e rápido
- Concorrência
  - é necessário gerir a concorrência (agora tudo é partilhado...)
  - pode ser mais difícil... *mas pode ser rápido...*

13

## API Posix: Criar novos *threads*

- Nota: API *threads* pode não ser sempre chamadas ao SO

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

- Criar um novo *thread*:

```
int pthread_create( pthread_t *tid,  
                  pthread_attr_t *attr,  
                  void *(*start)(void *),  
                  void *arg )
```

- novo thread começa chamando a função **start(arg)**
  - novo estado de CPU e novo Stack seguido de call **start**
- retorna 0 ou nº de erro (não segue convenção do SO)

14

identificação do  
novo thread (nº índices)

o novo thread  
começa por executar  
a função start  
com o argumento em  
\*arg

## Argumentos do novo *thread*

- Exemplo, criar um *thread* e passar-lhe um `int`:

```
void *thmain( int *arg )
{...}
...
int arg=10;
...
pthread_create(&td, NULL, thmain, &arg)
...
```

- `td` fica com identificador do novo thread
- O novo *thread* começa a execução como:  
`thmain(&arg)`



15

## Terminação/retorno do thread

```
void *thmain( int *arg )
{ long *result = malloc(sizeof(long));
  . . .
  pthread_exit( result );
  // ou return result;
}
...
• aguardando a terminação de thmain:
long *th_result;
pthread_join( tid, &th_result );
...
```

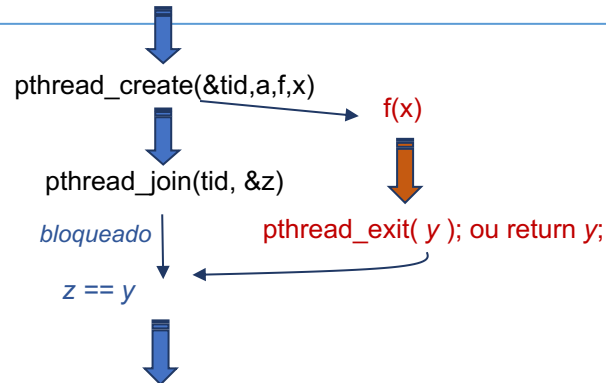
o thread em que chama esta função espera que o thread (tid) termine para continuar — join fica à espera



16



## Vida de um thread



- **pthread\_join()** aguarda pela terminação de um *joinable thread* (e pode receber um valor de retorno)

## Atributos (pthread\_attr\_t)

- Definem um conjunto de características:
  - tipo de escalonamento, prioridade, tamanho do stack, "detach"...
  - variável do tipo pthread\_attr\_t representa um conjunto de atributos e é manipulada com funções:

```
pthread_attr_init(),
pthread_attr_destroy()
pthread_attr_set*(), pthread_attr_get*()
```

- Exemplo, para o atributo *detached state*:

```
int pthread_attr_setdetachstate(set, val)
int pthread_attr_getdetachstate(set)
```

## exemplo - *thread detached*

- Não se pretende e não será necessário fazer o join ao thread

```
pthread_attr_t att;
```

```
pthread_attr_init( &att );
```

```
pthread_attr_setdetachstate(&att,  
                             PTHREAD_CREATE_DETACHED);
```

```
pthread_create(&td, &att, thmain, &arg);
```

```
pthread_attr_destroy( &att );
```

- alternativa para este exemplo, após a criação:

```
pthread_detach(td)
```

## Compilação

- Estas funções **não são sempre chamadas ao SO** (a implementação varia com cada sistema)
  - **Não reportam** os erros via errno (depois de retornar -1), mas diretamente pelo retorno da função, sendo 0 se não há erro

- Podem não fazer parte do **libc**

- podem estar numa **libpthread**

- Exemplo de compilação:

```
cc -o prog prog.c -pthread
```

(liga com biblioteca de threads e escolhe uma implementação alternativa de algumas funções do libc)

*percor( ),*

*quando se compila tem que se informar que queremos compilar o programa com todas as funcionalidades dos threads*

## Chamadas p/ threads vs processos

- |                  |   |
|------------------|---|
| • pthread_create | • fork/execve                           |
| • "attributes"   | • <i>environment e outros atributos</i> |
| • pthread_exit   | • exit                                  |
| • ou return      | • ou return                             |
| • pthread_join   | • waitpid                               |
| • pthread_self   | • getpid                                |

## Threads e variáveis

- Todo o espaço de memória do processo é de todos os *threads*
- Tipicamente:
  - As variáveis globais, estáticas ou criadas dinamicamente (heap), são partilhadas por todos os threads
  - As variáveis locais, criadas no frame da função (stack), são do respectivo thread

## Exemplo (ostep 26.3)

```

int max;
volatile int counter=0;

void *
mythread(void *arg){
    int i;
    for (i=0; i<max; i++){
        counter = counter+1;
    }
    return NULL;
}

int
main(int argc, char *argv[]){
    max = atoi(argv[1]);
    pthread_t p1, p2;
    pthread_create(&p1, NULL,
                  mythread, "A");
    pthread_create(&p2, NULL,
                  mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n
           [counter: %d]\n
           [should: %d]\n",
           counter, max*2);
    return 0;
}

```

NVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

23

## Problema

- Detalhe (objdump -dS):

```

8048829: a1 48 a0 04 08
804882e: 83 c0 01
8048831: a3 48 a0 04 08

```

**counter = counter + 1;**

```

mov    0x804a048,%eax
add    $0x1,%eax
mov    %eax,0x804a048

```

Código executado dentro dos threads

- Um escalonamento possível:

Thread A

```
mov    0x804a048,%eax
```

Thread B

```

mov    0x804a048,%eax
add    $0x1,%eax
mov    %eax,0x804a048

```

```

1→2 { add    $0x1,%eax
      mov    %eax,0x804a048
      ...

```

não está atualizado com o valor modificado pelo outro thread

perdeu-se um incremento, em vez de se passar de 1→3, passou-se de 1→2

NVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

24

## O que pode correr mal?

- Existem ações concorrentes . . .
  - Devido a interrupções
  - Devido ao escalonamento ou paralelismo
  - Devido a sinais
- Produzem resultados errados se partilharem recursos
  - Variáveis, IO, ...
- O SO é o primeiro exemplo que um sistema com ações concorrentes
  - Muitos dos problemas e suas soluções ocorrem na sua implementação

## Ações concorrentes

- Threads/processos concorrentes: dados dois threads P e Q:
  - uma vez iniciado P, pode-se iniciar Q, sem ter de esperar pelo fim de P, ou vice-versa.
  - A execução concorrente de P com Q define um conjunto de **várias sequências possíveis**. Exemplo considerando apenas um CPU:

P: { **p1**; **p2** } e Q: { **q1**; **q2** }

**p1**; **p2**; **q1**; **q2** ← equivale a P; Q

**p1**; **q1**; **p2**; **q2**

**p1**; **q1**; **q2**; **p2**

**q1**; **q2**; **p1**; **p2** ← equivale a Q; P

**q1**; **p1**; **q2**; **p2**

**q1**; **p1**; **p2**; **q2**

## Correção de um programa concorrente

- Programa concorrente: especifica múltiplos processos ou threads concorrentes
- A sua execução tem de produzir resultados corretos em **TODAS** as execuções possíveis.
  - Fácil: se threads INDEPENDENTES.
  - Difícil: se os threads DEPENDEM uns dos outros
    - Comunicam ou sincronizam-se
  - *Ao repetir a execução de um MESMO programa com os MESMOS dados, os resultados podem ser DIFERENTES?*
  - *Se nunca vir um resultado diferente não significa que o programa esteja correto!*

## Regiões Críticas

- **Regiões Críticas**: Regiões de código que envolvem recursos partilhados executadas concorrentemente
- É necessário sincronizar as ações concorrentes que interferem
  - Impor uma ordem ou impedir certas ordens
- Recorre-se à programação de protocolos de acesso à região crítica
  - Recorrendo ao auxílio de entidades externas (p.e. SO)
  - Recorrendo a algoritmos e/ou instruções específicas
  - Exemplo: optar pela **exclusão mútua** no uso do recurso, criando operações indivisíveis ou **atómicas**