

# PROCESSOS e *THREADS*

- Conceito de processo
- Estados de um processo
- Transições de estado
- Descrição de processos
- Estruturas de controlo de processos
- Operações sobre processos
- *Threads*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Noção de Processo

### Processo

- programa em execução

### Processo $\neq$ Programa

- programa - entidade passiva (conteúdo de um ficheiro)
- processo - entidade activa

### Um processo engloba

- código + dados
- conteúdo do *program counter*, registos, *stack*, ...
- recursos



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Programas e Processos

Um programa torna-se num processo através de um procedimento de carregamento ( *loading* ).

O ficheiro contendo o programa compilado é lido e a memória do novo processo é inicializada com o conteúdo do ficheiro

- código do programa
- dados inicializados.

O S.O. cria um novo Bloco de Controlo de Processo (*PCB - Process Control Block*)

- estrutura de dados contendo informação acerca do processo.

O processo inicia a execução no ponto de partida do programa quando o S.O. o despacha para execução.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Multiprogramação

Execução sequencial de programas  $\Rightarrow$

- desperdício de recursos
  - » as operações de I/O são muito mais lentas do que a execução de instruções por parte da CPU

**Solução:**

- interlaçar os cálculos com a I/O para aumentar a eficiência
- execução concorrente (multiprogramação)

**Multiprogramação:**

- técnica que sobrepõe operações de I/O e de cálculo de diversos processos em execução.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto


## Multiprogramação

A ideia básica é permitir que múltiplos processos residam em memória ao mesmo tempo e

- quando um processo bloqueia à espera de uma operação de I/O a CPU executa outro processo;
- quando este bloqueia, a CPU executa um 3º processo, ...etc...
- quando a operação de I/O, de que o 1º processo estava à espera, termina, o S.O. marca o processo que estava bloqueado como pronto a executar.

O processo pode ser obrigado a ceder a CPU antes de bloquear.

Multiprogramação sem preempção

- Os processos decidem quando devem ceder a CPU. 

Multiprogramação com preempção

- O S.O. decide quando um processo deve ceder a CPU.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

### Uniprogramação

O S.O. permite

- só um processo em execução
- só um processo algures entre o início e o fim de execução

### Multiprogramação

O S.O. permite

- só um processo em execução
- múltiplos processos algures entre o início e o fim de execução

### Multiprocessamento

O S.O. (e o hardware) permite

- múltiplos processos em execução
- múltiplos processos algures entre o início e o fim de execução.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Exercício

Programa A	Programa B	Programa C
100 instruções	ler 1 sector	1000 instruções
escrever 1 sector	100 instruções	escrever 1 sector
100 instruções	escrever 1 sector	

Ler/Escrever 1 sector = 0.0020 segundos  
executar 100 instruções = 0.0001 segundos

Com uniprogramação e multiprogramação

- Quanto tempo leva a executar cada programa ?  
(considerar uma chegada quase simultânea pela ordem A,B,C)
- Quanto tempo é que a *CPU* está inactiva ?



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Dificuldades da multiprogramação

- Necessidade de proteger os recursos atribuídos a cada processo, nomeadamente, proteger e controlar o acesso a:
  - áreas de memória
  - certas instruções do processador
  - periféricos de I/O
- Isto requer que o *hardware* possua certas características especiais, por exemplo
  - dois modos de funcionamento (utilizador e supervisor)
  - registos especiais usados na protecção de memória
- Necessidade de comunicação e sincronização entre processos interdependentes.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Funções de administração de processos num S.O.

- Criação e remoção de processos.
- Interlaçamento da execução dos processos e controlo do seu progresso garantindo o avanço da sua execução pelo sistema.
- Actuação por ocasião da ocorrência de situações excepcionais (erros aritméticos, ... ).
- Alocação dos recursos de *hardware* aos processos.
- Fornecimento dos meios de comunicação de mensagens e sinais entre os processos.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Processos e *Threads*

Um processo tem duas características

- posse de recursos
  - » Ficheiros, memória, ... detidos pelo processo
- uma sequência / *thread* de execução
  - » Informação sobre o que é e onde está o processo (*PC*, *PSW* e outros registos)

Os S.O.'s modernos usam o conceito de *thread* ou *lightweight process* (LWP).

Processo / tarefa → posse de recursos

*Thread* / LWP → sequência de execução

Múltiplas *threads* podem estar associadas a um processo.



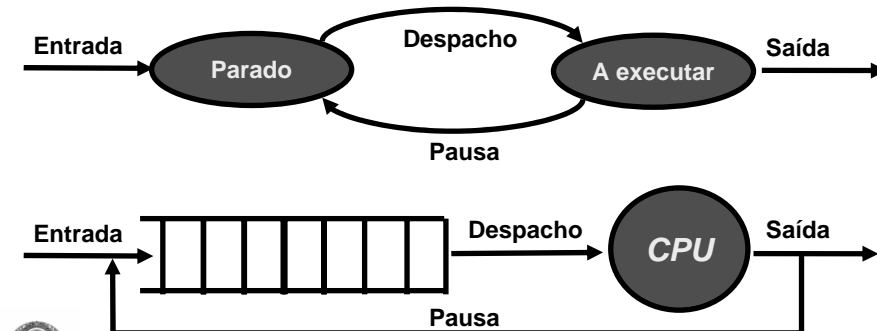
FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Estados de um processo

À medida que um processo executa, muda de estado.

### Modelo de 2 estados



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Estados de um processo

Alguma informação que é necessário guardar:

- estado actual do processo
- sua posição na memória
- lista de processos à espera de execução

A lista de processos à espera de execução pode conter 2 tipos de processos:

- processos prontos a correr
- processos bloqueados (à espera de I/O)

Surge assim o modelo de 5 estados.

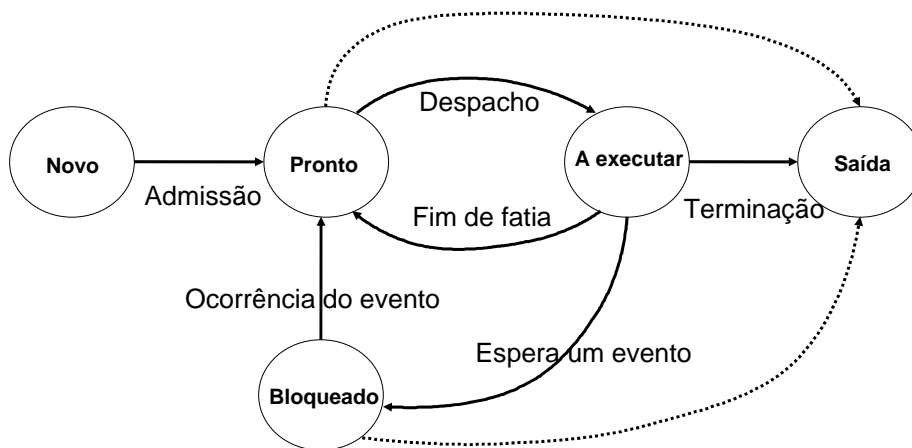
Poderá existir  
uma fila de processos prontos e uma fila de processos bloqueados  
ou mesmo  
uma fila de processos prontos por cada nível de prioridade  
e uma fila de processos bloqueados por cada evento (dispositivo).



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

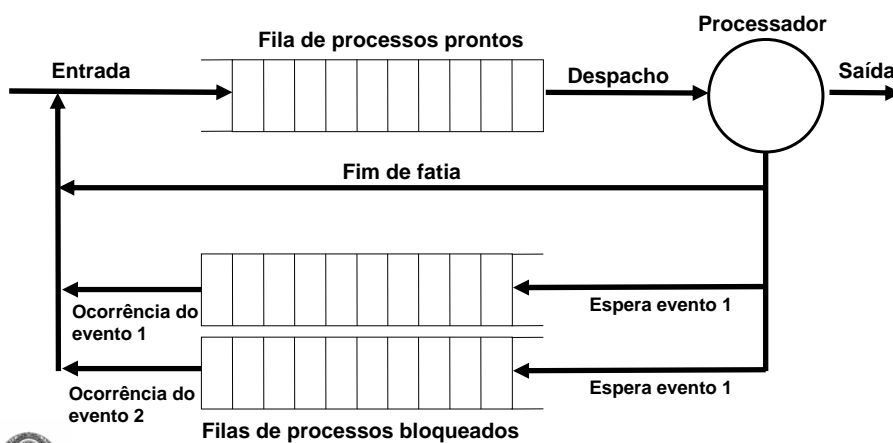
## Modelo de 5 estados



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Modelo de 5 estados



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Modelo de 5 estados

### Estados:

- **Novo**

- » o processo acaba de ser definido, mas ainda não está em execução

- **Pronto**

- » o processo está à espera que lhe seja atribuída a *CPU*

- **A executar**

- » as instruções estão a ser executadas

- **Bloqueado**

- » o processo está à espera da ocorrência de um acontecimento

- **Terminado**

- » o processo terminou a execução (normalmente ou abortou)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Modelo de 5 estados

### Transições de estado:

- **Novo → Pronto**

- » q.do um processo é criado e inicializado

- **Pronto → A Executar**

- » q.do a um processo é atribuída a *CPU*

- **A Executar → Pronto**

- » q.do uma fatia de tempo expira  
( multiprogramação com preempção )

- **A Executar → Bloqueado**

- » q.do um processo bloqueia à espera de um acontecimento  
( operação de I/O, acesso a ficheiro, serviço do S.O. ,  
comunicação c/outro processo, ... )

- **A Executar → Terminado**

- » q.do um processo termina a execução

- **Bloqueado → Pronto**

- » q.do o acontecimento ocorre

- **Pronto, Bloqueado → Terminado**

- » q.do o processo é forçado a terminar por outro processo



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto



## Estados de um processo

### Num sistema sem memória virtual

- cada processo a executar tem de estar totalmente carregado em memória principal
- todos os processos, de todas as filas de espera, têm de estar em memória principal

### Problema

- como o processador é muito mais rápido que a I/O será comum acontecer que todos os processos em memória estejam à espera de I/O.

### Solução

- *Swapping* - deslocar parte de ( / todo) um processo para o disco.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Swapping



Quando nenhum dos processos em mem. principal está pronto o S.O. desloca um dos processos bloqueados para o disco e coloca-o numa fila de processos suspenso (modelo de 6 estados).

A activação do processo (Suspenso → Pronto) só deve ser feita quando acontecer o evento que deu origem a que o processo fosse suspenso  
⇒ preferível dividir o estado Suspenso em 2 estados:  
Bloqueado Suspenso e Pronto Suspenso  
(modelo de 7 estados)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Processos suspensos

Razões para a suspensão de um processo:

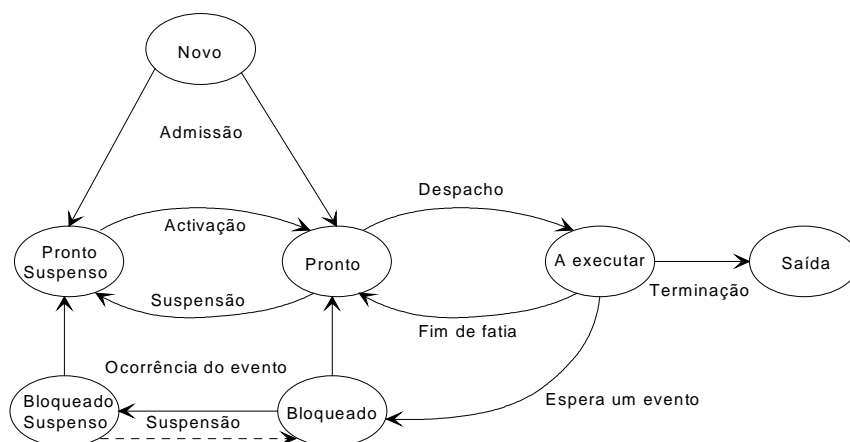
- *swapping*
- pedido interactivo do utilizador
- pedido do processo-pai
- temporização  
(ex.: processo executado periodicamente)
- outra razão do S.O.  
(ex.: processo que corre em *background*)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Modelo de 7 estados



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Modelo de 7 estados

### Transições de estado (algumas notas):

- **Pronto → Pronto Suspenso**
  - » em geral será pouco comum;
  - » será preferível suspender um processo bloqueado;
  - » mas pode acontecer p/libertar memória.
- **Bloqueado Suspenso → Bloqueado**
  - » q.do o processo BS tem maior prioridade do que qualquer um dos que está no estado Pronto Suspenso e o SO presume que o motivo do bloqueio desaparecerá em breve
- **A Executar → Pronto, Suspenso**
  - » o S.O. recorre à preempção (retirar a CPU) de um processo quando um processo de prioridade mais elevada fica Pronto.
- Podem acontecer várias transições de diversos estados para Terminação.

### Preempção

- acto de retirar o processador a um processo sem ser por ele estar bloqueado ou ter terminado.

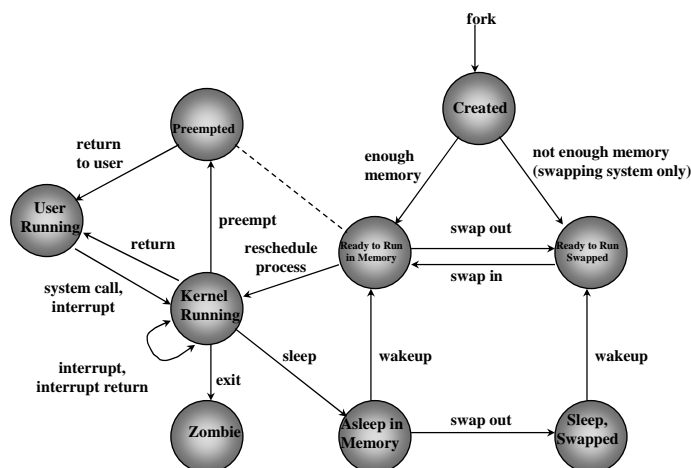


FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Estados dos Processos no UNIX System V



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Estados dos Processos no UNIX System V

- 9 estados
- 2 estados “A Executar”
  - modo núcleo / supervisor
    - » como resultado de :
      - chamada ao sistema
      - interrupção do relógio
      - interrupção de I/O
  - modo utilizador
- Os estados *Ready to Run in Memory* e *Preempted* são essencialmente o mesmo. Existe uma única fila de espera para ambos.
- A transição *sleep* corresponde a bloqueamento
- A preempção só pode ocorrer na ocasião em que um processo que está a executar em modo supervisor (*kernel running*) vai passar a executar em modo utilizador (*user running*).



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Estados dos Processos no UNIX System V

### Estado *asleep*

- O processo está à espera de um determinado acontecimento (operação de I/O, à espera num semáforo, ...)

### Estado *zombie*

- Um processo que termina não pode deixar o sistema até que o seu processo-pai aceite o seu código de retorno.
- Se o processo-pai estiver “vivo” mas nunca executar um *wait( )* o código de retorno do processo-filho nunca será aceite e este ficará *zombie*.
- Um processo *zombie* não tem código, nem dados, nem *stack*, mas continua a constar da tabela de processos.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Descrição de Processos

Para gerir e controlar os processos o S.O. deve saber:

- onde cada processo está colocado
  - » bloco contíguo de memória ou
  - » blocos separados (paginação, segmentação)  
podendo alguns não estar em memória (mem. virtual)
- os atributos do processo

O S.O. mantém uma tabela, a tabela de processos, com uma entrada por cada processo, contendo toda a informação relevante para a gestão dos processos.

A informação relativa a cada processo é mantida no respectivo Bloco de Controlo do Processo.

Os processos da tabela de processos poderão estar organizados em várias listas consoante o seu estado (pronto, a executar, ...).



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Bloco de Controlo do Processo ( *Process Control Block - PCB* )

Estrutura de dados  
contendo informação associada ao processo.

É no PCB que é guardado o estado de um processo por ocasião da comutação de processos.

Inclui

- identificação do processo ( =*Process ID / PID*, do processo, do pai )
- estado do processo (pronto, bloqueado, suspenso, ...)
- registos do processo ( *program counter*, flags, registos da *CPU*, ... )
- informação de escalonamento da *CPU* (prioridade, ...)
- informação de gestão da memória (lim.s da zona de memória,...)
- informação de contabilidade (tempo de *CPU* gasto,...)
- informação de estado da *I/O* ( fich.s abertos, operaç.s pendentes, ... )



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Estruturas de controlo do S.O.

Outras estruturas de dados  
necessárias para a gestão dos processos:

### Tabelas de memória

- alocação da memória principal e secundária
- protecções de acesso
- informação para a gestão de memória virtual

### Tabelas de I/O

- estado das operações de I/O
- localização dos dados de origem e de destino

### Tabelas de ficheiros

- ficheiros existentes
- posição em memória secundária
- estado actual
- outros atributos



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Gestão de processos

Operações típicas do núcleo (*kernel*):

- criação e terminação de processos
- escalonamento e despacho
  - » *scheduller* - implementa a política global de gestão da *CPU*  
( selecciona o próximo processo a executar )
  - » *dispatcher* - dá o controlo da *CPU* ao processo seleccionado
- ⇒
  - comutar de contexto
  - comutar para modo utilizador
  - saltar para o endereço adequado do programa
- sincronização e suporte para intercomunicação entre processos
- gestão dos *PCB's*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação e Terminação de processos

### Criação de um processo (estado Novo)

- O sistema operativo
  - » cria as estruturas de dados necessárias p/gerir o processo
  - » aloca o espaço de endereçamento a ser usado pelo processo

### Terminação de um processo

- O processo é retirado em 2 etapas
  - » 1 – É-lhe retirado o processador (normalmente ou abortou).
  - » 2 – A informação associada ao processo é apagada
    - Esta informação é mantida no sistema depois de o processador lhe ter sido retirado para que outros processos possam extrair informação relativa ao processo que acabou (ex.: tempo de CPU, recursos usados)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação e Terminação de processos

### Razões para a criação de processos:

- novo "batch job"
- "log on" interactivo
- criado pelo SO p/fornecer um serviço (ex: impressão)
- criado por outro processo

### Razões para a terminação de um processo (\*):

- completção normal
  - tempo limite excedido
  - memória indisponível
  - violação dos limites de memória
  - erro de protecção
  - erro aritmético
  - tempo de espera excedido
  - falha de I/O
  - instrução inválida
  - instrução privilegiada
  - intervenção do operador ou do SO
  - terminação do processo-pai
  - pedido do processo-pai
  - ...
- (\*) - algumas podem não ser aplicáveis em alguns S.O.'s



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação de um processo

### Etapas:

- Atribuir um identificador ao processo
- Reservar espaço para o processo
  - » para todos os elementos da imagem do processo
    - programa + dados + *stack* + *PCB*
- Inicializar o *PCB*
- Colocar o processo na lista de processos Prontos
- Criar / actualizar outras estruturas de dados  
(ex.: dados de contabilidade do sistema)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Comutação de Contexto

Sempre que um processo bloqueia e outro processo passa a ser executado ocorre uma comutação de contexto:

- salvaguarda do estado actual do processo (registos, ...)
- restauro do estado, previamente guardado, do próximo processo a executar
- passagem do controlo do processador para o novo processo

Comutação de contexto  $\Rightarrow$  perda de tempo

Redução do tempo de comutação de contexto

- máquinas onde existe mais do que um conjunto de registos;
- utilização de *threads*.

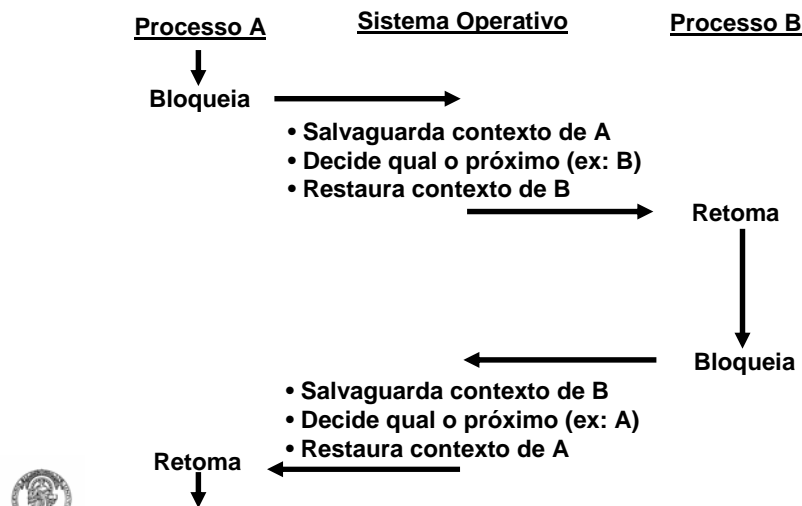


FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto



## Comutação de contexto



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação de processos em UNIX

fork() cria um novo processo (processo-filho) que obtém uma cópia de toda a memória do processo-pai e partilha os ficheiros que o processo-pai estiver a usar.

Os 2 processos (pai e filho) executam concorrentemente.

Não é carregado nenhum programa novo.

Os 2 processos correm o mesmo programa.

O processo divide-se em 2 cópias, ambas resultantes da chamada a fork(), com todo o estado anterior em comum.

Existe um conjunto de chamadas exec(), ( de facto execXX(), em que XX depende da chamada ) para fazer o carregamento de um programa novo. O código do programa que invocar exec() é substituído pelo código do programa que for indicado como argumento de exec().

(ver apontamentos sobre API do UNIX)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação de processos em UNIX (cont.)

fork() cria simultaneamente

- um novo processo
- um novo espaço de endereçamento

Espaço de endereçamento

- a memória em que um processo é executado

É possível distinguir o processo-pai do processo-filho testando o valor retornado por fork():

- = 0  $\Rightarrow$  é o processo-filho
- > 0  $\Rightarrow$  é o processo-pai e o valor retornado é o identificador do filho
- = -1  $\Rightarrow$  a chamada falhou



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação de processos em UNIX (cont.)

### EXEMPLO:

```
main()
{
    int pid;
    if ((pid = fork())== -1)
        return(-1);
    else
        if (pid==0)
        {
            printf("Eu sou o filho !\n");
            exit(1);
        }
    printf("Eu sou o pai\n");
}
```

Qual o resultado  
deste programa ?

E se a instrução `exit(1)`  
fosse retirada ?



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Criação de processos em UNIX (cont.)

**EXERCÍCIO:** Qual o resultado do seguinte programa ?

```
main()
{ printf("1\n");
  printf("2\n");
  fork();
  printf("3\n");
  printf("4\n");
}
```

### Resposta:

Tanto pode ser	→	como →
1		1
2		2
3		3
4		3
3		4
4		4

Tudo depende de como  
a sequência dos 2 programas  
for interlaçada,  
devido à multiprogramação.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Criação de processos em UNIX (cont.)

Para criar um novo processo, executando um programa diferente usa-se uma função execXX().

Uma função execXX() é uma chamada à biblioteca do C que por sua vez chama uma rotina de sistema, execve().

### EXEMPLO:

```
...
switch (pid=fork()) {
  case 0 : /* Este é o filho */
    execl("/bin/ls", "ls", "-l", NULL);
    /* se chegar aqui, o exec falhou */
    exit(1);
  case -1 : /* o fork() falhou */
    exit(2);
  default : /* Este é o pai */
    /* executa concorrentemente com o filho */
    ...
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Threads

Um(a) *thread* é um processo “leve” (*Lightweight Process*), com um estado reduzido.

A redução de estado é conseguida fazendo com que um grupo de *threads* (do mesmo processo) partilhe recursos como memória, ficheiros, dispositivos de I/O, ...

Nos sistemas baseados em *threads*,

- um processo pode ter vários *threads*;
- os *threads* tomam o lugar dos processos como a mais pequena unidade de escalonamento ;
- se a implementação for *kernel-level* enquanto um *thread* está bloqueado, outro pode estar a executar
- o processo serve como o ambiente p/ a execução dos *threads*.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Threads

Processo tradicional (*Heavyweight Process*)  
⇔ processo c/um único *thread*.

Um *thread* partilha c/ os outros *threads* do mesmo processo:

- a secção de código
- a secção de dados
- os recursos do S.O.

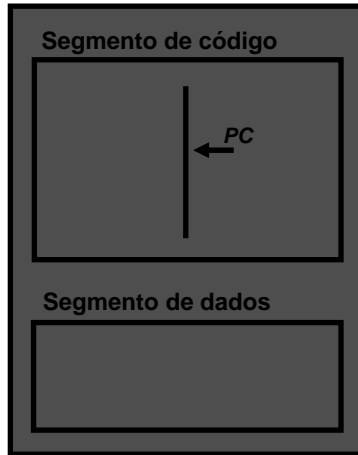
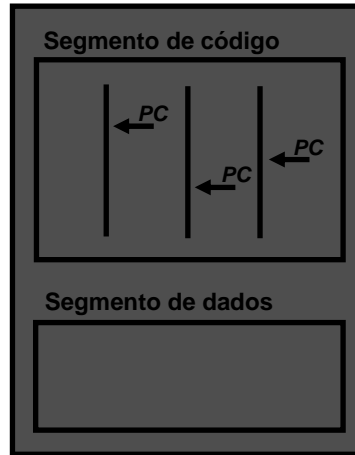
⇒ a comutação entre *threads* do mesmo processo é muito menos pesada do que entre processos tradicionais



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

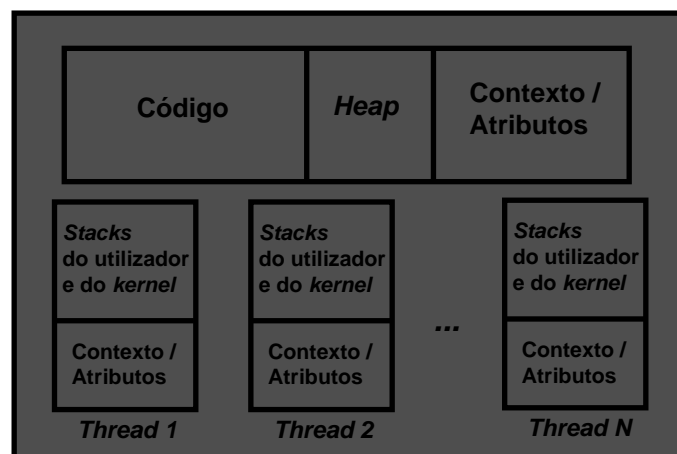
## Processo tradicional

Processo *c/ threads*

FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Processo



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Processos Tradicionais vs. *Threads*

### *Threads*

#### Semelhanças c/ os processos

- têm um estado (pronto, a executar, bloqueado, ...)
- partilham a *CPU* entre si  
(em cada instante apenas um *thread* está a executar, num sistema uniprocessador)
- cada *thread* de um processo executa sequencialmente
- cada *thread* tem associado
  - » um *program counter*
  - » um *stack pointer*
  - » um *Thread Control Block*  
(c/ conteúdo dos registos da *CPU*, estado do *thread*, prioridade,...)
- um *thread* pode criar *threads-filho*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Processos Tradicionais vs. *Threads*

### *Threads*

#### Algumas características importantes:

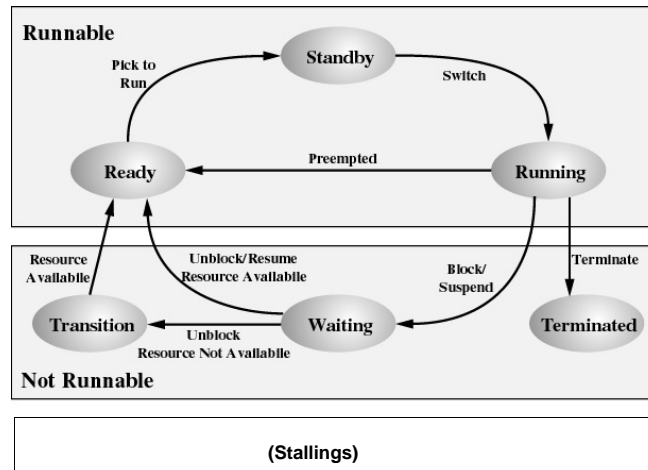
- Não existe protecção entre *threads* do mesmo processo
  - » desnecessária (!) ;  
os *threads* são concebidos para cooperarem numa tarefa comum
- Qualquer alteração das variáveis globais de um processo é visível em todos os seus *threads*
  - » Em alguns SOs é possível um *thread* criar variáveis globais cujo conteúdo depende do *thread* que refere essa variável; estas variáveis não são acedidas directamente, mas através de chamadas a funções específicas de acesso. Esta facilidade é conhecida por *TLS-Thread Local Storage*.
- Suspensão (*swapping*) de um processo  $\Rightarrow$  suspensão dos seus *threads*
- Terminação de um processo  $\Rightarrow$  terminação dos seus *threads*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Estados dos *threads* no Windows 2000



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Estados dos *threads* no Windows 2000

- Ready
  - o *thread* pode ser escalonado para execução
- Standby
  - o *thread* foi seleccionado p/executar a seguir; espera neste estado até o processador estar disponível (que o *thread* a executar bloqueie ou a sua fatia de tempo expire); se a prioridade deste *thread* for superior à do *thread* que está a correr este pode sofrer preempção
- Running
  - a executar até sofrer preempção, expirar a sua fatia de tempo, bloquear ou terminar; nos 2 primeiros casos volta p/ o estado Ready
- Waiting
  - bloqueado num evento (ex: I/O) ou
  - à espera de um acontecimento de sincronização ou
  - recebeu ordem de suspensão
- Transition
  - está pronto a correr mas os recursos ainda não estão disponíveis (ex: a *stack* do *thread* foi colocada em disco, em consequência da paginação)
- Terminated
  - terminou normalmente ou foi terminado p/outro *thread* ou o proc.-pai terminou



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## ***Threads***

### **Algumas vantagens de utilização:**

- Economia e velocidade
  - » menos tempo p/ criar, comutar e terminar
- Aumento da rapidez de resposta percebida pelo utilizador
  - » ex: um *thread* lê comandos, outro executa-os; permite ler o próximo comando enq. o anterior é executado
- Eficiência de comunicação
  - » recorrendo à memória partilhada não é necessário invocar o *kernel*
- Utilização de arquitecturas multiprocessador
  - » cada *thread* pode executar em paralelo num processador diferente

### **Dificuldade**

- garantir a sincronização entre os *threads* quando manipulam as mesmas variáveis



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## ***User-level e Kernel-level Threads***

### **User-level threads**

- O *kernel* "não sabe" da existência de *threads*.
- Toda a gestão dos *threads* é feita pela aplicação usando uma biblioteca de funções apropriada.
- A comutação entre *threads* não requer privilégios de *kernel mode*.
- O escalonamento depende da aplicação.

### **Kernel-level threads**

- Toda a gestão dos *threads* é feita pelo *kernel*.
- Não existe uma biblioteca de *threads* mas uma *API* de *threads*.
- A comutação entre *threads* requer a intervenção do *kernel*.
- O escalonamento é feito sobre os *threads*.
- O *kernel* mantém informação de escalonamento sobre os processos e sobre os *threads*.

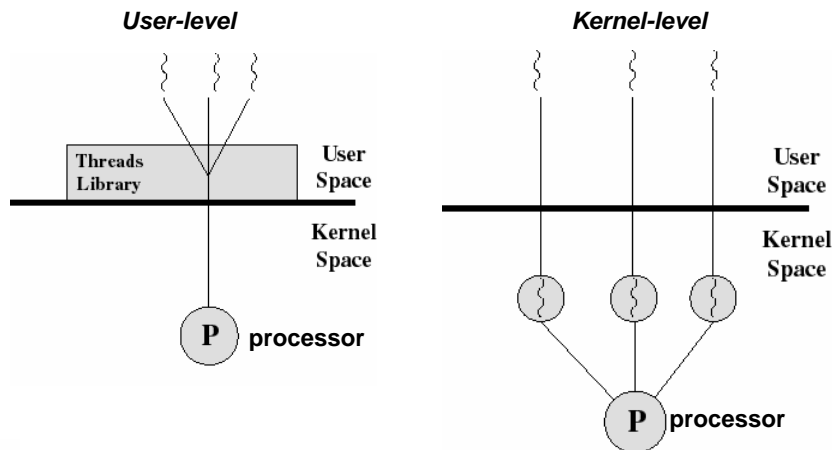


FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto



## User-level e Kernel-level Threads



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## User-level Threads

### Vantagens

- A comutação entre *threads* não envolve o *Kernel*: não implica comutação para *Kernel mode*.
- O escalonamento pode ser específico de uma aplicação: possível escolher o algoritmo mais adequado.
- Podem ser usados em qualquer S.O. . Basta que se disponha de uma biblioteca adequada.

### Inconvenientes

- Quando uma chamada ao sistema implica um bloqueio (ex: I/O) todos os *threads* do processo ficam bloqueados
- O *kernel* só pode atribuir processadores aos processos. Dois *threads* do mesmo processo não podem correr em simultâneo em dois processadores.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Kernel-level Threads

### Vantagens

- O *kernel* pode escalonar os diversos *threads* de um mesmo processo para executarem em diferentes processadores.
- O bloqueamento é feito ao nível dos *threads*. Quando um *thread* bloqueia, outros *threads* do mesmo processo podem continuar a executar.
- As rotinas do *kernel* podem ser *multithreaded*.

### Inconvenientes

- A comutação entre *threads* do mesmo processo envolve o *kernel*. (2 comutações: *user mode* → *kernel mode* e *kernel mode* → *user mode*)
- Isto resulta numa comutação mais lenta.



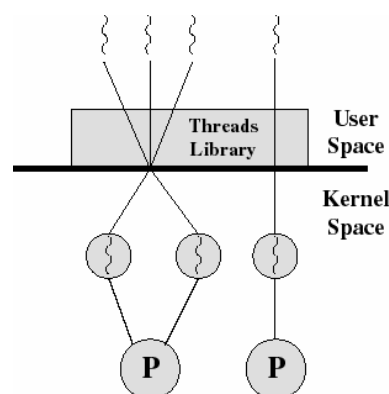
Sistemas operativos: Windows NT/2000/XP, Linux, Solaris

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Aproximação mista

- A criação de *threads* é feita no espaço do utilizador.
- A maior parte do escalonamento e sincronização também são feitos no espaço do utilizador.
- As *User-level Threads* são mapeadas em *Kernel-level Threads* ( $n^{\circ}$  de KLTs  $\leq$   $n^{\circ}$  de ULTs).
- O utilizador pode ajustar o número de *Kernel-level Threads*.
- Permite combinar as vantagens de *ULTs* e *KLTs*.
- Exemplo: Solaris 2.x



MIEIC  
Faculdade de Engenharia da Universidade do Porto

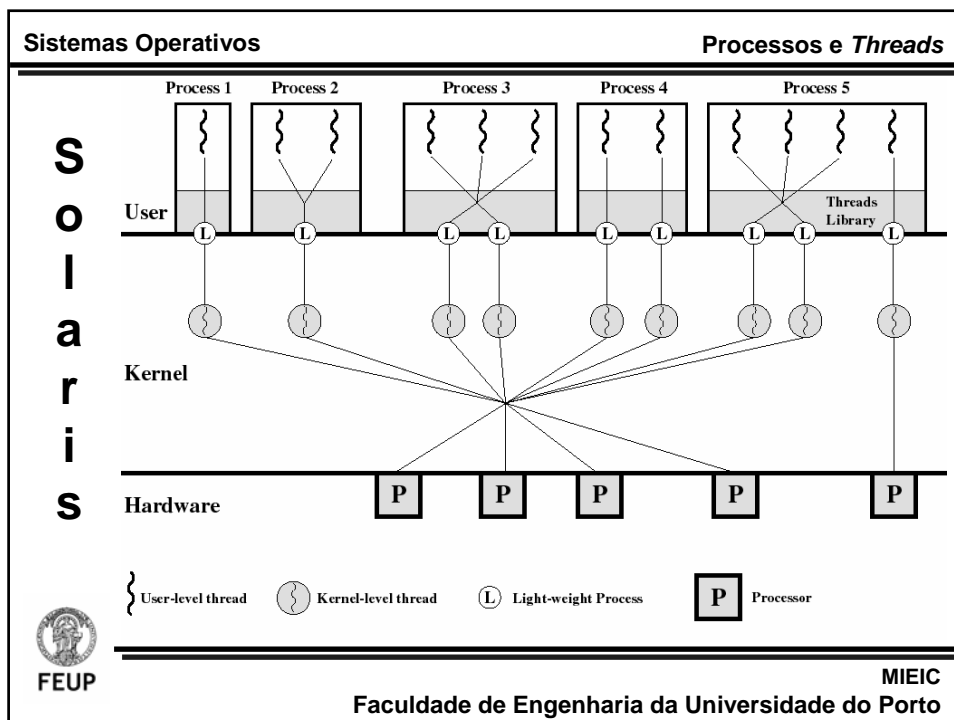
## Solaris 2.x

- Um processo inclui o espaço de endereçamento do utilizador, a *stack* e o *PCB*.
- *User-level threads (threads library)*
  - » invisível para o S.O.
- *Kernel threads*
  - » a unidade sujeita a despacho num processador
- *Lightweight processes (LWP)*
  - » cada *LWP* suporta um ou mais *ULTs* e mapeia-os exactamente num *KLT* (fig. seguinte).



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto



## Interface Pthreads (*Posix threads*)

Funções (~60) :

- Criar e esperar por *threads*
  - pthread\_create
  - pthread\_join
- Terminar *threads*
  - pthread\_cancel
  - pthread\_exit
  - exit() - termina todos os *threads*;  
return - termina o *thread* corrente
- Determinar a ID de um *thread*
  - pthread\_self
- Sincronizar o acesso a variáveis partilhadas
  - pthread\_mutex\_init
  - pthread\_mutex\_lock, pthread\_mutex\_unlock
  - ...



(ver apontamentos sobre API do UNIX)

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Posix threads

**EXEMPLO** (criação de um *thread*) :

```
#include <stdio.h>
#include <pthread.h>
...
char message[] = "Hello world !";

void *thread_function(void *arg) {
    printf(" Thread function is running. Argument is %s\n", (char *) arg);
    ...
    return NULL;
}

int main(void) {
    int res;
    pthread_t thread_id;

    res = pthread_create(&thread_id, NULL, thread_function, (void *) message);
    if (res != 0) { /* ERROR */ }
    ...
}
```

compilação: \$ cc thrprog.c -o thrprog -lpthread



(ver apontamentos sobre API do UNIX)

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Win32 *API*

### Primitivas da Win32 *API* :

- *CreateProcess*
- *CreateThread*
- *SuspendThread*
- *ResumeThread*
- *ExitThread*
- *TerminateThread*
- ...

### *CreateProcess()*

- cria um novo espaço de endereçamento a partir de um ficheiro executável e cria um único *thread* executando no ponto de entrada do programa

### *CreateThread()*

- cria um novo *thread* dentro do espaço de endereçamento do *thread* original



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto