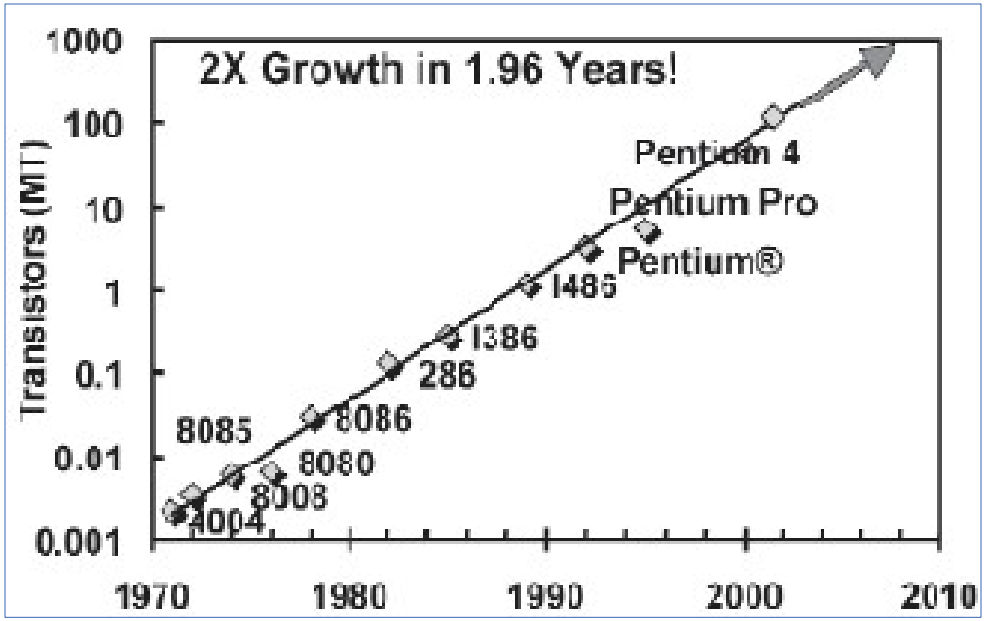
Programação Sistemas

Fios de Execução

Porquê o paralelismo? (1)

1. Crescentes necessidades de computação têm sido satisfeitas com aumento do número de transístores nos "chip". Gordon Moore, co-fundador da Intel previu em 1965 que o número de transístores duplicaria em cada 18 meses.

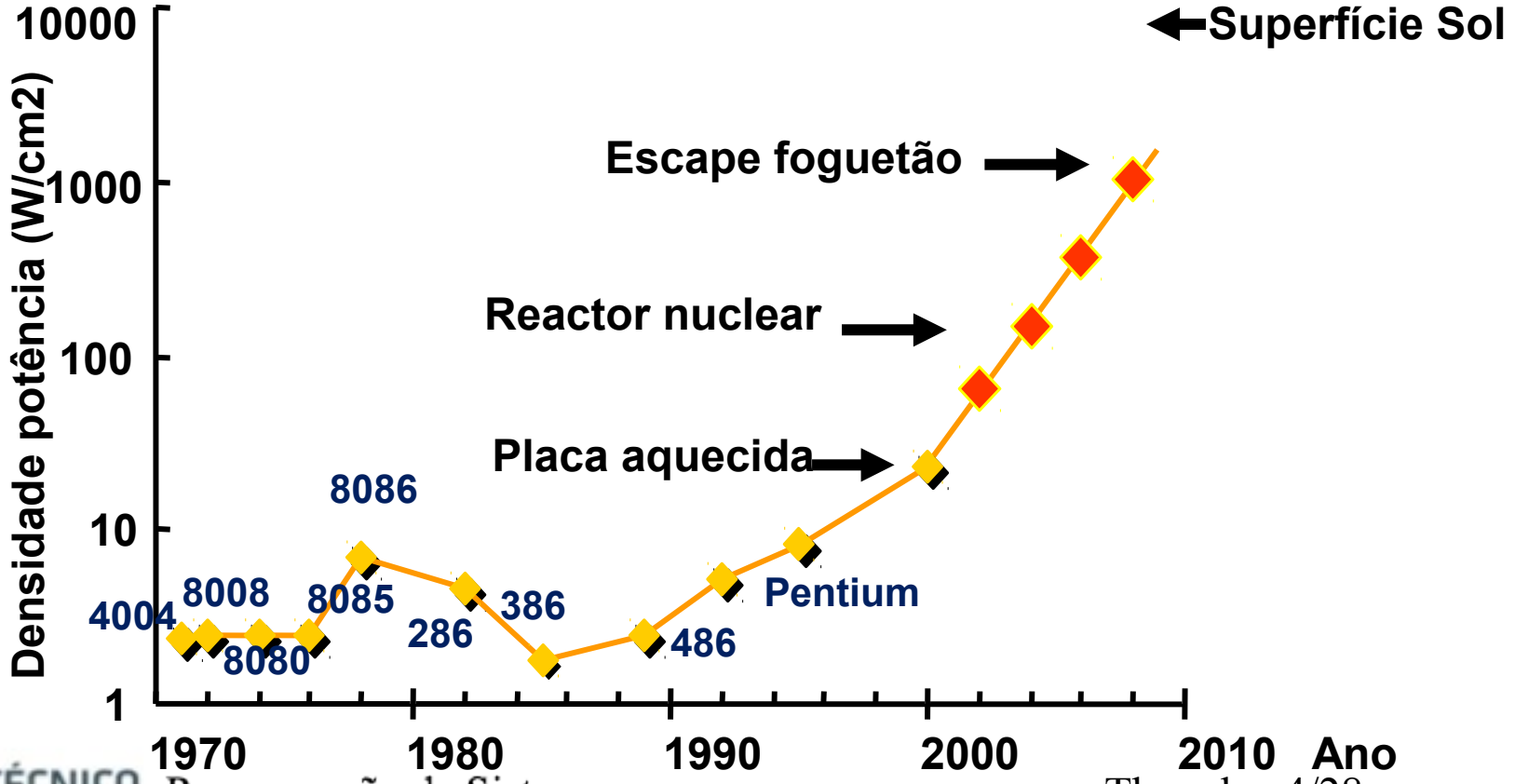


Porquê o paralelismo? (2)

2. A velocidade do relógio aumentou imenso entre anos 70 e 2000.
   - 8008 @ 200 Khz em Abr 1972
   - 8080 @ 2 Mhz em Dec 1974
   - 8085 @ 5 Mhz em Ago 1976
   - 8086 @ 10 Mhz em Set 1978
   - 80286 @ 12 Mhz em Fev 1982
   - 80386 @ 16 Mhz em Out 1985
   - 80486 @ 25 Mhz em Abr 1989
   - Pentium @ 60 Mhz em Mar 1993, @ 120 Mhz em Mar 1995
   - Pentium II @ 300 Mhz em Mai 1997
   - Pentium III @ 600 Mhz em Ago 1999, @ 1 Ghz em Mar 2000
   - Pentium 4 @ 1.5 Ghz em Nov 2000, @ 3.4 Ghz em Fev 2004
- Actualmente a velocidade do relógio estabilizou à volta de 3GHz.

Porquê o paralelismo? (3)

3. Com a actual tecnologia, o aquecimento constitui a barreira impeditiva do aumento da frequência de relógio.



Porquê o paralelismo? (4)

- Várias estratégias têm sido exploradas no processamento paralelo de informação:
  - Processadores múltiplos.
  - Paralelismo a nível de instrução (ILP-Instruction Level Paralelism ex: pipeline).
  - Paralelismo a nível de fios de execução (TLP-Thread Level Paralelism).

Nota: tópico central nas disciplinas
Sistemas Operativos Distribuídos
e
Computação Paralela e Distribuída

Introdução (1)

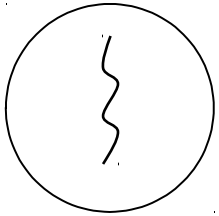
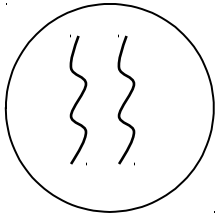
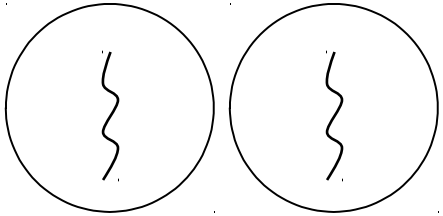
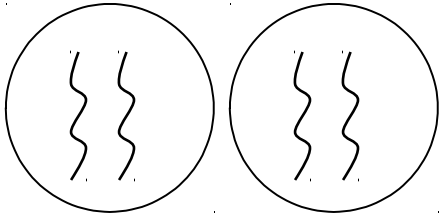
- Diversas unidades de execução partilham acesso aos mesmos recursos (ex: ficheiros).
  No entanto a gestão dos processos é muito pesada. A solução é ter diversas unidades de execução, partilhando código, dados globais e ficheiros.
- **[Def] Fio de execução**: unidade de execução dum processo, com os valores individuais de
  - Registos de processador
  - Pilha

Nota1: também usada designação de processo leve LWP-"lightweight process" e "task" no Linux.

Nota2: um processo lançado pelo `fork()` possui um único fio de execução.

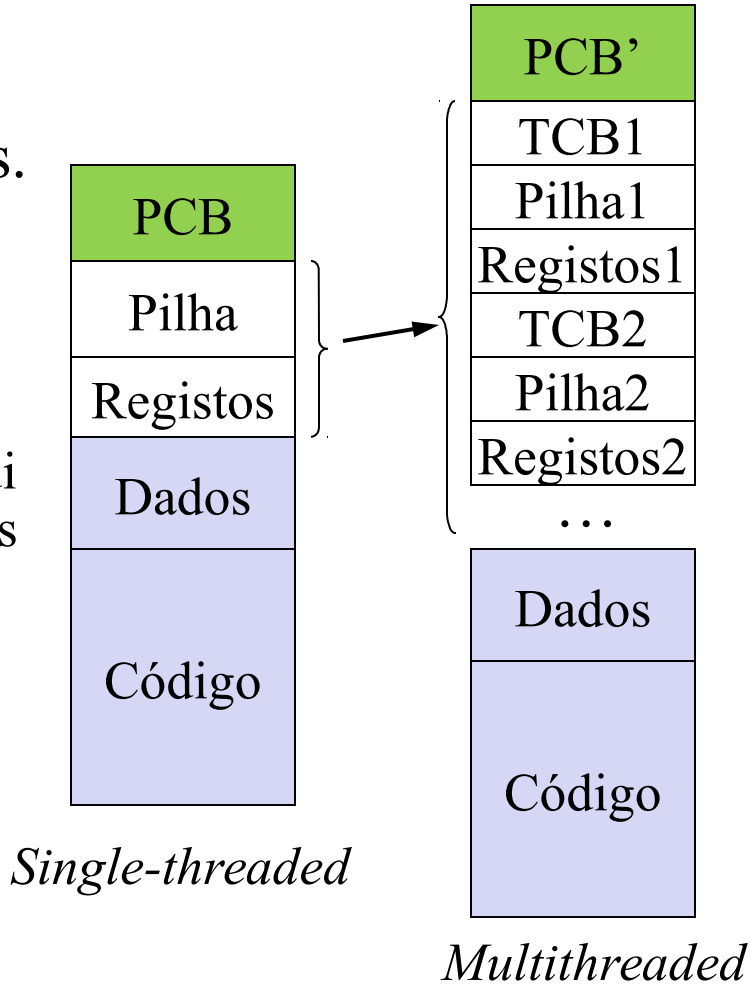
Introdução (2)

- Os sistemas que permitem vários fios de execução para o mesmo processo são denominados “multithreading”.
- Os SO podem ser catalogados em:

	Single-threaded	Multithreading
Uniprogramação	 Ex:MSDOS	 Ex:JavaVirtualMachine
Multiprogramação	Ex:UNIX iniciais 	Ex:Linux,Windows 

Introdução (3)

- Com fios de execução, a distribuição em memória do processo no Linux sofre alterações.
  - Cada fio de execução possui o seu bloco de controlo TCB-Thread Control Block.
  - Em vez de uma única pilha para o processo, cada fio de execução possui a sua própria pilha. as variáveis locais são pertencentes a cada fio de execução.
- Os fios de execução partilham
  - código,
  - variáveis globais,
  - recursos do SO (sinais, ficheiros)



Introdução (4)

- Ao contrário dos processos,
  - os fios de execução não são hierarquizados numa árvore,
  - o espaço de dados não é independente (apenas a pilha e os registos são individuais às *threads*).
- Tipicamente, o controlo dos processos (criação e ceifa/ ”reap”) é mais pesado que nos fios de execução (entre 20 e 240 vezes, dependendo da máquina e da implementação dos fios de execução).

Introdução (5)

- Várias APIs para fios de execução foram criadas
  - Wind32 threads.
  - C-Threads
  - Pthreads : norma POSIX IEEE 1003.1c, publicada em 1995, disponível em muitas implementações do Unix e adoptada nesta disciplina.
- O POSIX define funções de gestão de fios de execução, na extensão THR (cerca de 60).
  - As funções possuem o prefixo `pthread_`
  - Definições disponíveis no ficheiro `pthread.h`
  - Edição de ligações ("link") deve indicar arquivo `-lpthread`

Introdução (6)

Nota: convenção adoptada pelo Pthreads para os identificadores

- Tipos de dados : `pthread[_objecto]_t`
- Funções : `pthread[_objecto]_acção`
- Constantes e macros: `PTHREAD_OBJECTIVO`

Exemplos:

`pthread_t` : referência a fio de execução.

`pthread_create()` : lançamento de novo fio de execução.

`pthread_mutex_t` : referência a ferrolho de fios de execução.

`pthread_mutex_lock()` : fechar ferrolho.

`PTHREAD_CREATE_DETACHED` : macro

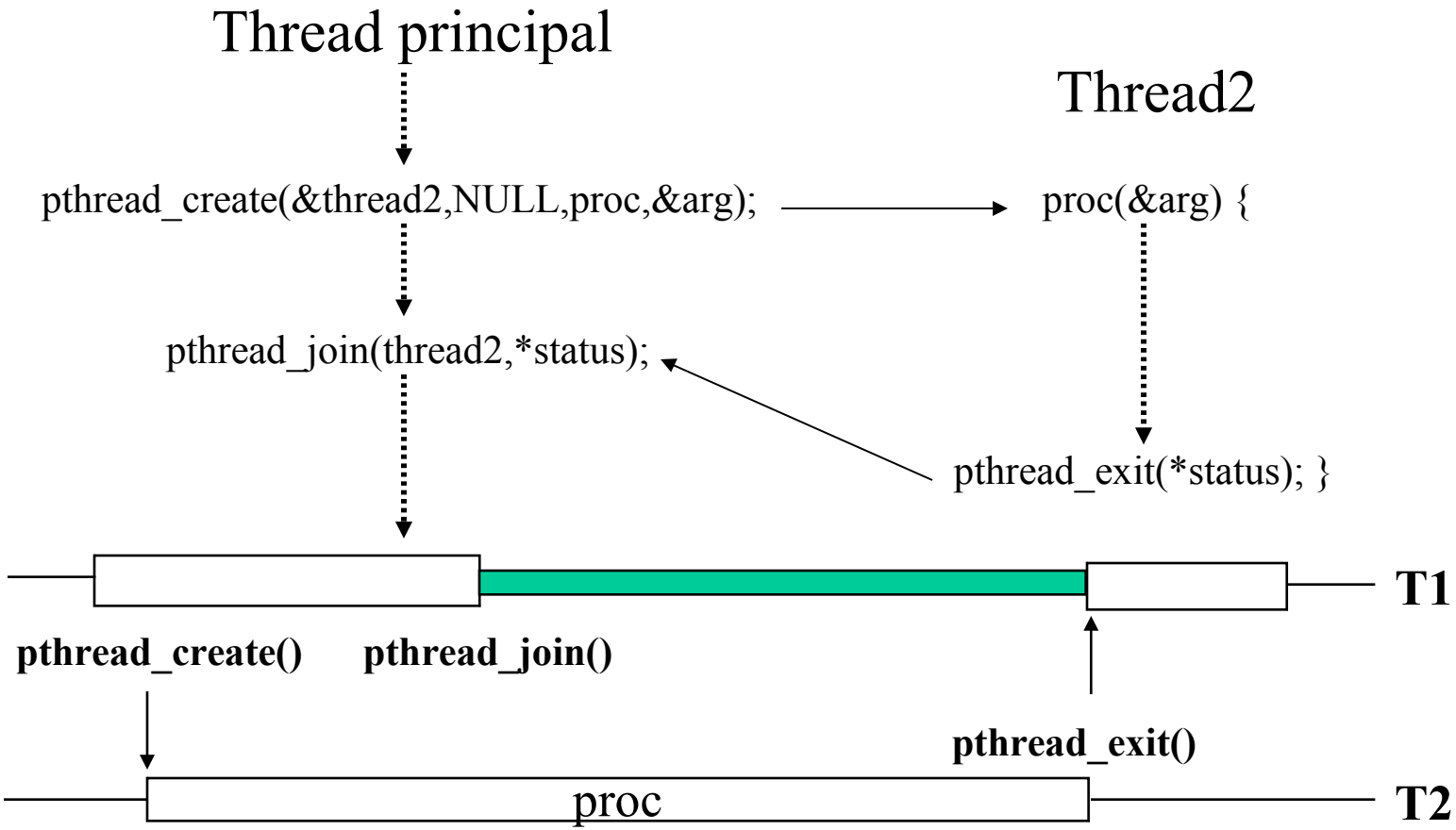
Introdução (7)

- Listagem das funções mais relevantes

Função POSIX	Descrição
`pthread_cancel()`	Solicita terminação a outro fio de execução
`pthread_create()`	Cria fio de execução
`pthread_detach()`	Determina libertação de recursos no fim
`pthread_equal()`	Testa igualdade em 2 fios de execução
`pthread_exit()`	Sai do fio de execução
`pthread_kill()`	Envia sinal para o fio de execução
`pthread_join()`	Espera pela terminação de um fio de execução
`pthread_self()`	Identifica o próprio ID

Introdução (8)

- Tipicamente um programa possui as seguintes instruções



Introdução (9)

- Casos favoráveis à utilização de fios de execução:
  - Multiplice (“multiplex”), com comunicação de mensagens de tipos distintos pelo mesmo canal: ex-servidores.
  - Espera sincronizada: ex-clientes.
  - Notificação de eventos: ex-simulações, sistemas gráficos.
  - Sistemas com desempenho crítico.
- Casos desfavoráveis à utilização de fios de execução:
  - Sistemas cuja execução é limitada por um recurso único. (ex: contar número de ficheiros num disco)
  - Quando o custo de lançamento e gestão da thread não o justificarem
  - Quando a complexidade do desenvolvimento não o justificar

Identificação

- Os fios de execução possuem um identificador de tipo `pthread_t` (uma estrutura).
- Um fio de execução conhece o seu identificador por

POSIX:THX
```
#include <pthread.h>
pthread_t pthread_self();
```

- Uma vez que `pthread_t` é uma estrutura, a comparação de dois identificadores é feita pela função

POSIX:THX
```
int
pthread_equal(pthread_t,pthread_t)
```

Nota: para imprimir o identificador do terminal, usar formatação `%lu`.

Lançamento (1)

- Um fio de execução é lançado pela instrução

  POSIX:THX `int pthread_create(`

  ```
  pthread_t *,
  pthread_attr_t *,
  void *(*function)(void *),
  void *);
  ```

  - 1º parâmetro: endereço da localização onde é armazenado identificador do novo fio de execução.
  - 2º parâmetro: endereço da localização onde são indicados os atributos do novo fio de execução. O valor `NULL` indica atributos por omissão.
  - 3º parâmetro: função C que executa código, obrigatoriamente de assinatura `void * foo(void*)`.
  - 4º parâmetro: parâmetro da função C (se houver vários parâmetros, eles devem ser embrulhados numa tabela ou numa estrutura).
- Em caso de sucesso, `pthread_create()` retorna 0.

Lançamento (2)

- Os parâmetros e valores de retorno podem ser transmitidos por dias vias:
  - Variáveis globais: opção desaconselhada, porque os fios de execução chamador e lançado podem alterar as variáveis em simultâneo.
  - Pelo 4º parâmetro da função `pthread_create`.
- Os dois fios de execução trabalham em paralelo.
- O número máximo de fios de execução lançados depende do espaço ocupado pela pilha (tipicamente 300).

Nota: um processo lançado dentro de uma thread, por *fork()*, contém uma única cópia da thread que chamou o *fork()*.

Lançamento (3)

Nota: se a mesma thread for lançada várias vezes, não usar a mesma variável para parâmetro se ela for modificada – as threads podem ficar com o mesmo valor!

```
int i;          /* ordem de lançamento da thread */
int numb;       /* número de threads a lançar */
pthread_t *tID;

tID = (pthread_t *)malloc( numb*sizeof(pthread_t) );
for( i=0;i<numb;i++ ) {
      int *pos = (int *)malloc( sizeof(int) );
      *pos = i;
      if( pthread_create( &(tID[i]),NULL,foo,(void *)pos ) ) {
        printf( "Erro no lancamento da thread %d\n",i );
        exit( 2 ); }
   }
```

Espera por uma thread

- Um fio de execução espera pela conclusão de outro fio de execução pela instrução

  POSIX:THX `int pthread_join(pthread_t, void **);`

  - 1º parâmetro: identificador do fio de execução a esperar.
  - 2º parâmetro: endereço da localização onde reside o ponteiro para a localização de tipo `int` (onde reside código de saída).

Nota1: apenas um fio de execução ("thread") pode esperar pela conclusão de outro.

Nota2: os recursos de uma thread (ID,pilha) são libertos APENAS depois do *join*.

Eliminação (1)

- Um fio de execução elimina-se a si próprio pela instrução

  POSIX:THX `int pthread_exit(void *);`

  – 1º parâmetro: endereço da localização onde reside código de saída. A localização, de tipo `int`, deve ter existência fora da função que executa o código, o que pode ser feito por:
    - Variável global,
    - Localização indicada pelo fio de execução à espera.
    - Criada dinamicamente.

Nota 1: fazer `return(cod)` na rotina do fio de execução leva compilador a gerar automaticamente o `pthread_exit()`

Nota 2: depois do `pthread_exit`, os recursos só são libertos depois de outro fio de execução ter feito o `pthread_join()`.

A libertação dos recursos é feita automaticamente se for executado

POSIX:THX `int pthread_detach(pthread_t);`

Neste caso não pode ser feito o `pthread_join()`.

Eliminação (2)

- Um fio de execução sinaliza outro fio de execução pela instrução

  POSIX:THX `int pthread_kill(pthread_t, int);`

  - 1º parâmetro: identificador do fio de execução a eliminar
  - 2º parâmetro: código do sinal

Nota 1: se um processo terminar com `exit()`, todos os fios de execução lançados por ele terminam igualmente, excepto se os fios de execução lançados tiverem executado `pthread_detached()` .

Nota 2: enviar SIGKILL elimina todos os fios de execução do processo, não apenas a thread enviada.

2º EXERCÍCIO TEORICO-PRATICO

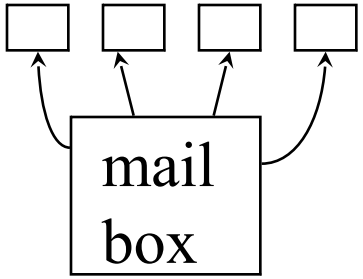
Calcule a soma dos elementos de uma tabela, contendo uma série de inteiros com crescimento linear a partir de 1, delegando o cálculo num fio de execução.

- A soma deve ser impressa no fio de execução e no programa principal.
- O fio de execução deve conter 3 argumentos:
  - Endereço base da tabela.
  - Dimensão da tabela.
  - Localização onde a soma é guardada.

Distribuição de pedidos (1)

- Chegado um pedido de processamento, como distribuí-lo pelos fios de execução existentes?
- Existem vários modelos de distribuição dos pedidos

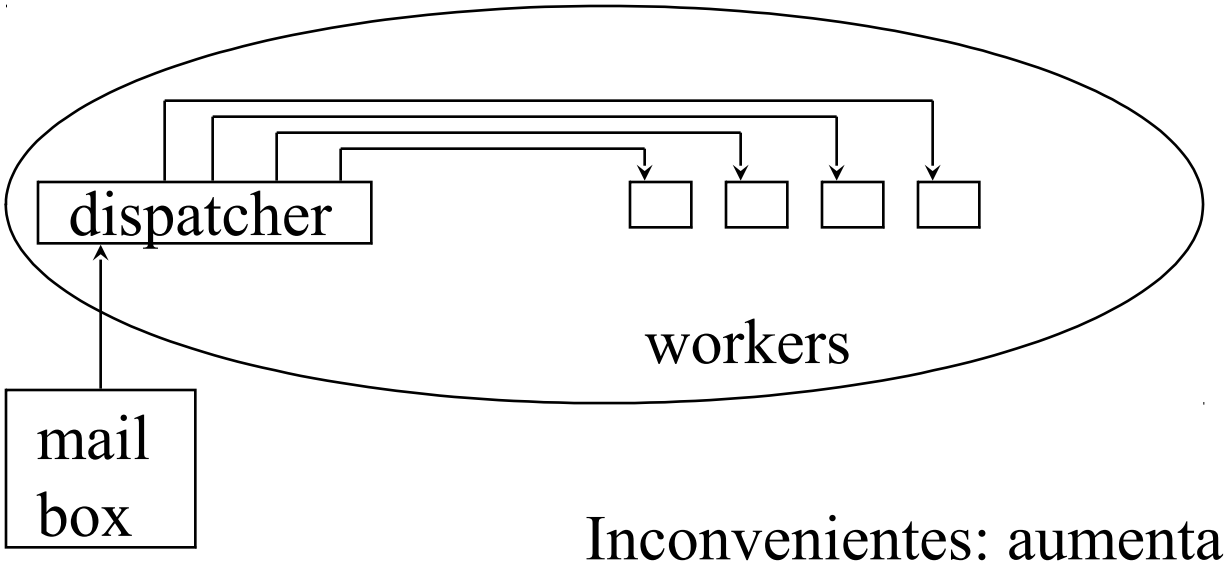
1. Modelo por equipa: cada "thread" fica à espera da chegada de um pedido



Inconveniente: especialização das "threads" dificultada

Distribuição de pedidos (2)

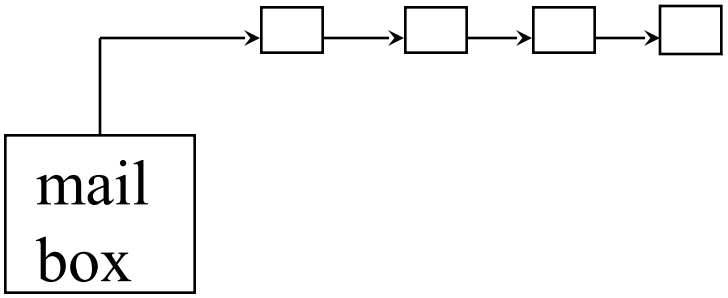
2. Modelo por servidores: um despacho lê a mensagem e entrega-a ao fio de execução apropriado, ou disponível



Inconvenientes: aumenta o número de threads, atrasa processamento do pedido.

Distribuição de pedidos (3)

3. Modelo por pipeline: a mensagem é processada numa sequência de threads



Inconvenientes: processamento do pedido tem de ser dividido por etapas.

Criação de threads

- Para cada nova tarefa pode ser criada uma thread (task) nova
  - Pode ser dispendioso se a execução da thread for curta (tarefa rapidamente executada)
- E se estivermos perante uma aplicação que está constantemente a executar novas tarefas?

  Ex: um servidor que está constantemente a servir pedidos
  - Estar constantemente a criar e destruir/terminar novas threads é ineficiente
    - pthread_exit() logo seguido de pthread_create() ?
- Uma solução é usar uma pool de threads
  - Não terminar as threads e "reutilizá-las"
    - Mas a manutenção das threads activas ocupa recursos

Thread Pool

- É uma forma de *multithreading* em que as tarefas, criadas em bloco no início ou sucessivamente quando necessárias, são mantidas numa fila para serrem activas quando necessário
  - Implica gerir a fila
  - Necessário definir o número de threads a ter na pool
    - Pool pode ser dinâmica (número de threads na pool varia) ou estática
  - Necessário gerir cuidadosamente a activação das threads da pool
    - Evitar situações de *deadlock*: todas à espera de outras que estão à espera de executar
  - Ganho de eficiência pode ser grande se o número de tarefas a executar ao longo do tempo for grande
  - Reutilização é mais eficiente que criação + terminação

Relação thread/processos (1)

- No Linux, as threads e processos (ambos designados por tarefas-"tasks") são lançados pela chamada de sistema `clone()`.
- Os recursos partilhados pelas duas tarefas, que levam à distinção entre threads e processos, é determinada por bandeiras ("flags").

Bandeira	Descrição
`CLONE_FS`	Sistema de ficheiros partilhado
`CLONE_VM`	Memória partilhada
`CLONE_SIGHAND`	Tratamento partilhado de sinais
`CLONE_FILES`	Partilhados ficheiros abertos

Relação thread/processos (2)

- Criação de tarefas efectuada pela chamada de sistema

```
#include <sched.h>
int clone(int (*fn)(void *),
   void *, int, void *,...)
```

O processo filho partilha parte do contexto de execução com processo ascendente (memória, tabela de descritores de ficheiros,...)

1. Questão: Se um processo for suspenso, o que acontece às threads?
   - Resposta: todas as threads são suspensa, porque elas partilham o espaço de dados.

Relação thread/processos (3)

2. Questão: Se um processo for terminado, o que acontece às threads?
   - Resposta: todas as threads são igualmente terminadas.

3. Questão: Se um fio de execução executar `fork()`, quantas threads são duplicadas?
   - Resposta: depende da versão do Unix! algumas possuem duas versões do fork(), uma duplica todas as LWP, outra apenas duplica a LWP chamadora.
   - `exec()` normalmente substitui todas as LWP no processo antigo.