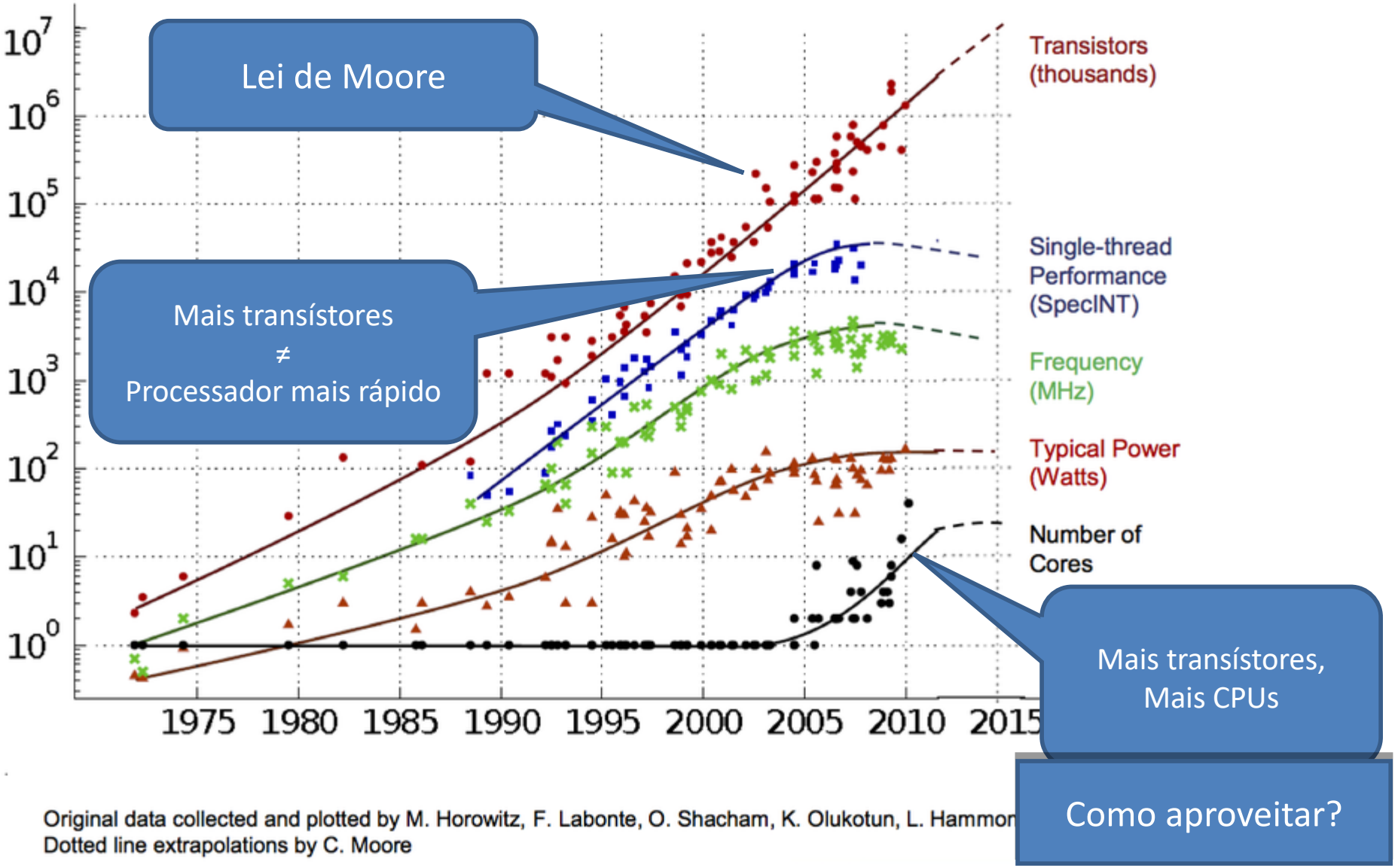
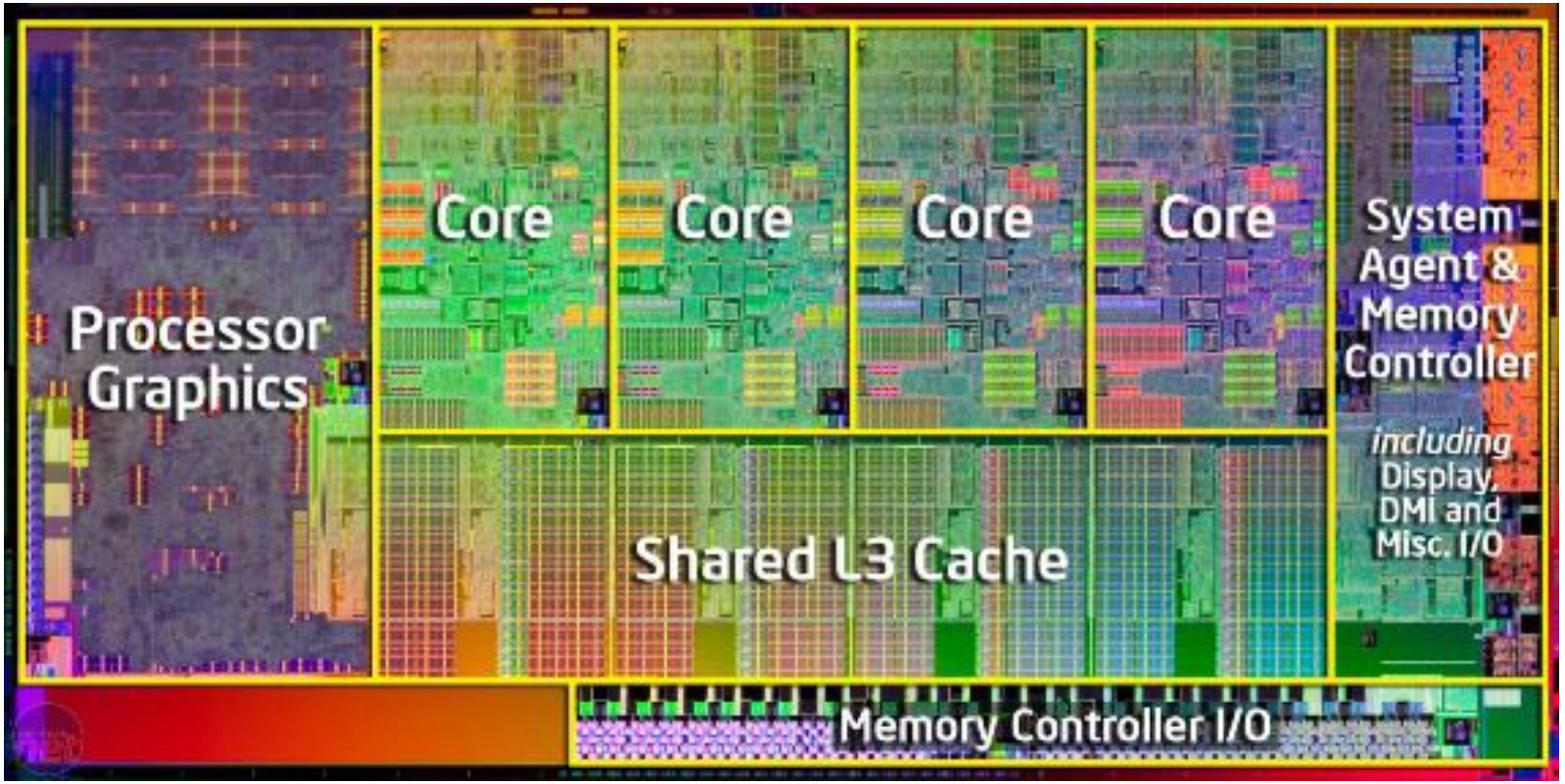
# O fim dos “almoços gratis”



# Programação paralela

- Permite explorar processadores múltiplos
  - Incluindo os dual-cores, quad-cores, etc.



# Outras razões para optar por programação paralela?

- Interação com periféricos lentos
  - Enquanto periférico demora a responder a um fluxo de execução, outro fluxo paralelo pode continuar a fazer progresso
- Idem para programas interativos
  - Enquanto um fluxo de execução espera por ação do utilizador, outros podem progredir em fundo

Ou seja, programação paralela faz sentido mesmo em máquinas *single-core!*

# Próximas aulas:
# Dois níveis de programação paralela (processos e tarefas)

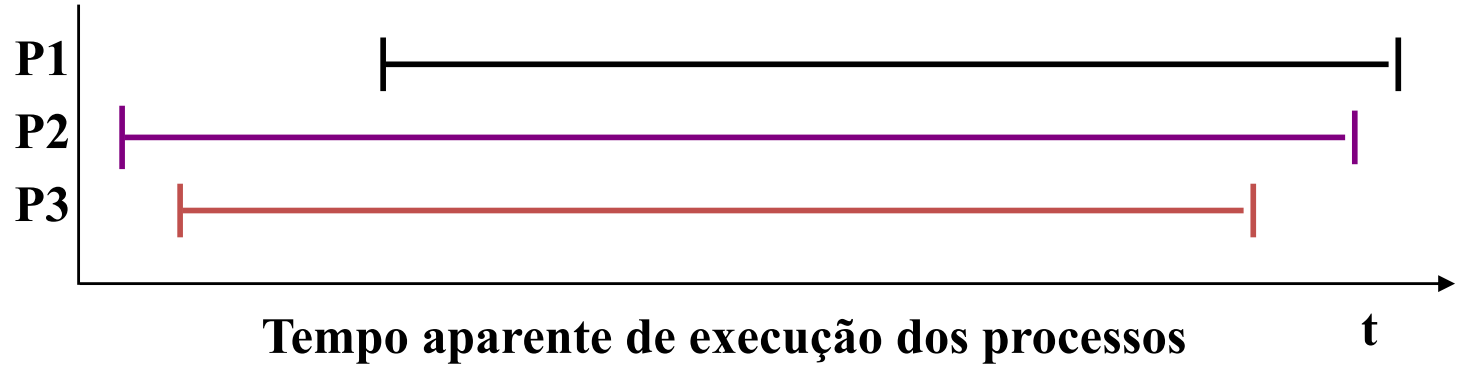
# Introdução à programação com processos

# Multiprogramação

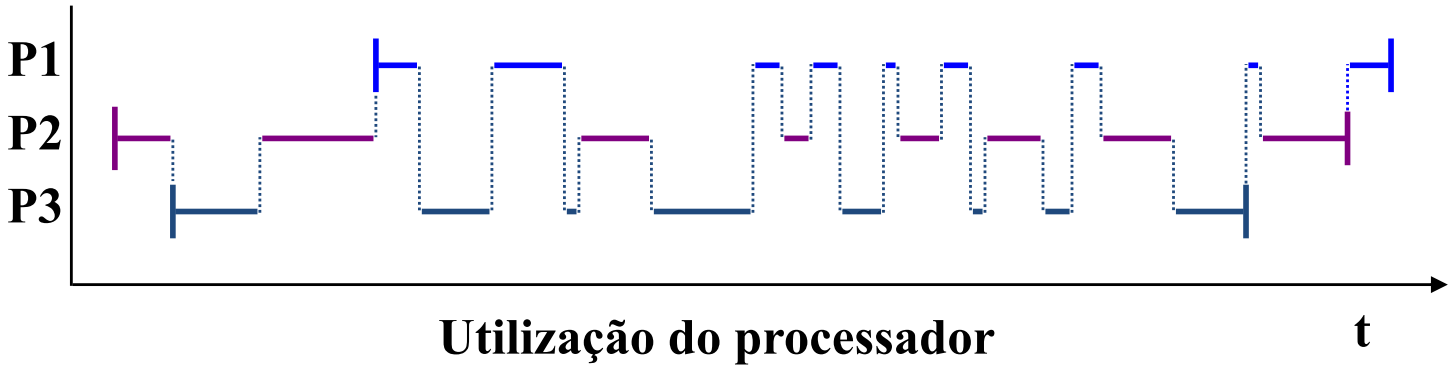
- Execução, em paralelo, de múltiplos programas na mesma máquina

- Cada instância de um programa em execução denomina-se um **processo**

# Pseudoconcorrência

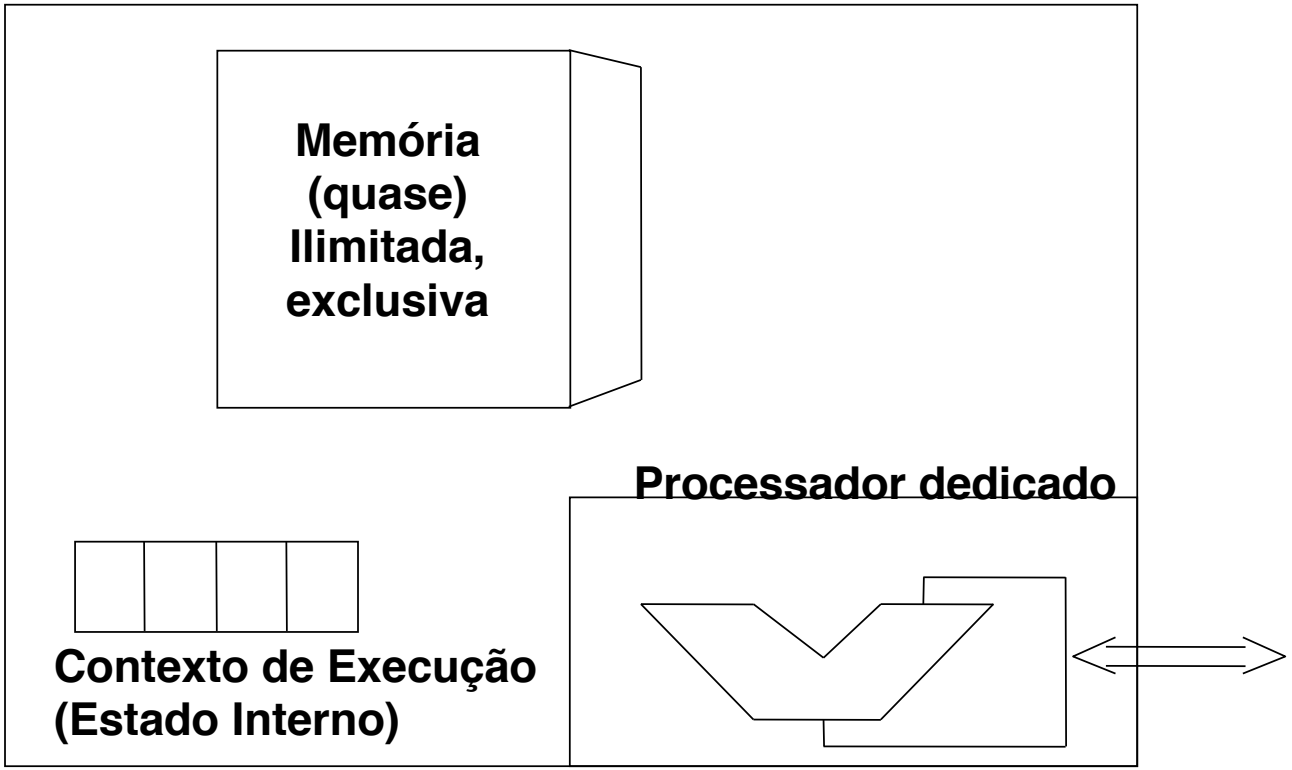



Tempo aparente de execução dos processos



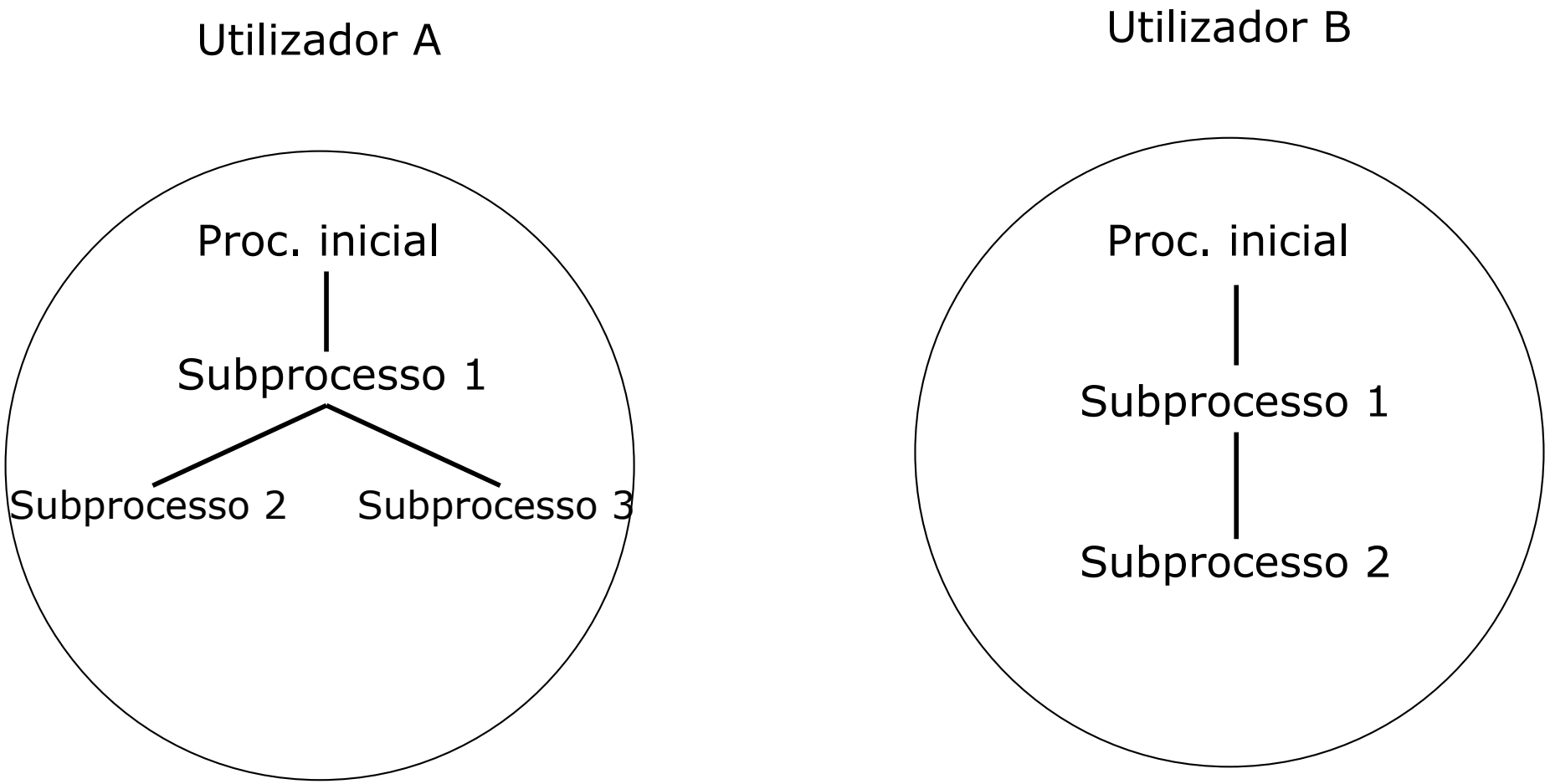

Utilização do processador

# Processo como uma Máquina Virtual




Elementos principais da máquina virtual que o SO disponibiliza aos processos

# Hierarquia de Processos




Certas propriedades são herdadas

# Programação com processos em Unix

# Criação de um Processo

id = fork()

**A função não tem parâmetros, em particular o ficheiro a executar.**

**Processo filho é uma cópia do pai:**

**- O espaço de endereçamento é copiado**

**- Contexto de execução é copiado**

Então que atributos diferem entre filho e pai?

Estas cópias são pesadas?
Se acontecessem literalmente, seriam.
Na verdade, a chama a fork é muito rápida.
Veremos mais tarde qual o segredo.

**A função retorna o PID do processo.**

**Este parâmetro assume valores diferentes consoante o processo em que se efectua o retorno:**

- ao processo pai é devolvido o "pid" do filho
- ao processo filho é devolvido 0
- -1 em caso de erro

Retorno de uma função com valores diferentes!
Nunca visto em programação sequencial

# Exemplo de fork

```
main() {
    pid_t pid;
    pid = fork();
    if (pid == -1) /* tratar o erro */
    if (pid == 0) {
        /* código do processo filho */
    } else {
        /* código do processo pai */
    }

    /* instruções seguintes */
}
```

# Terminação do Processo

- Termina o processo, liberta todos os recursos detidos pelo processo, ex.: os ficheiros abertos
- Assinala ao processo pai a terminação

```
void exit (int status)
```

**Status é um parâmetro que permite passar ao processo pai o estado em que o processo terminou.**

**Normalmente um valor negativo indica um erro**

# E se a main terminar com return em vez de exit?

- Até agora, nunca chamámos exit para terminar programas
- Terminação de programa feita usando return (int) na função main do programa
- Qual a diferença?
- Nenhuma, pois o compilador assegura que return da main resulta em chamada a exit!

```
main_aux(argc, argv) {
   int s = main(argc, argv);
   exit(s);
}
```

Função main do programador

# Terminação do Processo

- Em Unix existe uma função para o processo pai se sincronizar com a terminação de um processo filho
- Bloqueia o processo pai até que um dos filhos termine

```
int wait (int *status)
```

**Retorna o pid do processo terminado. O processo pai pode ter vários filhos sendo desbloqueado quando um terminar**

**Devolve o estado de terminação do processo filho que foi atribuído no parâmetro da função exit**

# Como usar o estado de terminação

**man wait**

[...]

If status is not NULL, wait() and waitpid() store status information in the int to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in wait() and waitpid()!):

WIFEXITED(status)

returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().

WEXITSTATUS(status)

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

WIFSIGNALED(status)

returns true if the child process was terminated by a signal.

WTERMSIG(status)

returns the number of the signal that caused the child process to terminate. This macro should only be employed if WIFSIGNALED returned true.

WCOREDUMP(status)

returns true if the child produced a core dump. This macro should only be employed if WIFSIGNALED returned true. This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Only use this enclosed in #ifdef WCOREDUMP ... #endif.

[...]

Processo filho nem sempre termina normalmente (com exit)!

Quando termina com exit, inteiro retornado deve ser obtido usando esta macro

Há várias razões para terminação sem exit

# Exemplo de Sincronização entre o Processo Pai e o Processo Filho

```
main () {
    int pid, estado;

    pid = fork ();
    if (pid == 0) {
        /* algoritmo do processo filho  */
        exit(0);
    } else {
        /* o processo pai bloqueia-se à espera da
           terminação do processo filho  */
        pid = wait (&estado);
    }
}
```

# *Exit* elimina todo o estado do processo?

- São mantidos os atributos necessários para quando o pai chamar *wait*:
  - Pid do processo terminado e do seu processo pai
  - Status da terminação
- Entre exit e wait, processo diz-se *zombie*
- Só depois de wait o processo é totalmente esquecido

# O minimalismo da função fork

- O fork apenas permite lançar processo com o mesmo código
  - Prós e contras?

# Vamos aprender passo por passo...

1. O meu primeiro programa paralelo
2. Pai e filho a executarem trabalhos diferentes
   - Pai executa fnPai(), filho executa fnFilho()
   - https://tinyurl.com/so-fork-1
3. Pai espera pelo resultado do filho
   - Filho: termina devolvendo o retorno de FnFilho()
   - Pai: depois de executar fnPai(), aguarda até saber o resultado do filho, e imprime soma de ambos
   - https://tinyurl.com/so-fork-2

# Como ter filho a executar programa diferente?

```
int execl(char* ficheiro, char* arg0, char* argl,…, argn,0)

int execv(char* ficheiro, char* argv [])
```

**Caminho de acesso ao ficheiro executável**

**Argumentos para o novo programa. Podem ser passado como apontadores individuais ou como um array de apontadores.**
**Estes parâmetros são passados para a função main do novo programa e acessíveis através do argv**

**NOTA:**
*** execl() e execv() são "front-ends" mais simples para execve() que é a função principal com mais parâmetros**

# Exemplo de Exec

```
main ()
{
    int pid;

    pid = fork ();
    if (pid == 0) {
        execl ("/bin/who", "who", 0);
        /* controlo deveria ser transferido para o novo
              programa  */
        printf ("Erro no execl\n");
        exit (-1);
    } else {
        /* algoritmo do proc. pai */
    }
}
```

**Por convenção o arg0 é o nome do programa**

# Vamos implementar uma shell?

- Ciclo infinito, em que cada iteração:
  - Imprime mensagem
  - Lê comando
  - Cria novo processo filho
  - Processo filho deve executar outro programa (indicado no comando lido)
  - Entretanto, o processo da shell bloqueia-se até filho terminar
  - Volta à próxima iteração

https://tinyurl.com/so-fork-3

# Um exemplo completo: Shell

- O shell constitui um bom exemplo da utilização de fork e exec (esqueleto muito simplificado)

```
while (TRUE){
    prompt();
    read_command (command, params);

    pid = fork ();
    if (pid < 0) {
        printf ("Unable to fork");
         continue;
    }
    if (pid !=0) {
        wait(&status)
    } else{
        execv (command, params);
    }
}
```

# Modelo de segurança

# Modelo de Segurança

- Cada utilizador no sistema identificado por **User IDentifier (UID)**
  - *superuser* (ou *root*) tem UID especial (zero)
- Para facilitar a partilha, o utilizador pertence a um ou mais grupos de utilizadores
  - Cada grupo identificado por um **Group Identifier (GID)**

# Autenticação de processos

- Cada processo corre em nome de um utilizador (UID)/grupo (GID)
- UID/GID atribuídos ao **primeiro processo** criado quando o utilizador se autentica (*log in*)
  - Obtidos do ficheiro /etc/passwd no momento do login
- Processos filho **herdam UID/GID** do pai

# Autenticação de processos (II)

- Na verdade, há:
  - Real UID e real GID
    - » Normalmente nunca mudam
  - Effective UID e effective GID
    - » Podem mudar temporariamente
  - *E também há o saved UID/GID (fora da matéria)*
- Quando o processo faz chamada sistema: o SO **consulta o EUID/EGID** para determinar se tem permissão

# Controlo dos Direitos de Acesso

1. Processo pede para executar operação sobre objecto gerido pelo núcleo
2. Núcleo valida se o EUID/EGID do processo tem direitos para executar aquela operaçao sobre aquele objeto
3. Se sim, núcleo executa operação; se não, retorna erro

# Controlo dos Direitos de Acesso

- Conceptualmente, a autorização baseia-se numa Matriz de Direitos de Acesso

| | | **Objectos** | |
|---|---|---|---|
| **Utilizadores** | **1** | **2** | **3** |
| **1** | Ler | - | Escrever |
| **2** | - | Ler/ Escrever/Execu ta | - |
| **3** | - | - | Ler |

- Colunas designam-se Listas de Direitos de Acesso (ACL)
- Linhas designam-se por Capacidades

# Exemplo: ACL em Unix

```
$ ls -l
total 7
-rw-r--r-- 1 jpbarreto 197121 113 nov 20  2017 0SXW49AO.cookie
-rw-r--r-- 1 jpbarreto 197121   0 nov 15  2017 container.dat
-rw-r--r-- 1 jpbarreto 197121  91 jan 30  2018 deprecated.cookie
drwxr-xr-x 1 jpbarreto 197121   0 fev 20  2018 DNTException/
drwxr-xr-x 1 jpbarreto 197121   0 jul 15 12:04 ESE/
-rw-r--r-- 1 jpbarreto 197121 119 jan 25  2018 F0RK22C9.cookie
drwxr-xr-x 1 jpbarreto 197121   0 ago 16 23:36 Low/
-rw-r--r-- 1 jpbarreto 197121  95 nov 15  2017 N036J89L.cookie
-rw-r--r-- 1 jpbarreto 197121 470 dez  7  2017 OCPOPE1W.cookie
drwxr-xr-x 1 jpbarreto 197121   0 nov 15  2017 PrivacIE/
-rw-r--r-- 1 jpbarreto 197121 155 jan 26  2018 URPQ8UQ1.cookie
-rw-r--r-- 1 jpbarreto 197121 112 dez  7  2017 ZDCLU5P1.cookie
```

**Se tudo começa num primeiro processo a correr em nome do root, como pode haver processos (filhos) a correr em nome de outros utilizadores?**

# Mudar de UID/GID

- Primeira via: funções setuid(int) e setgid(int)
  - Mudam effective UID/GID
  - Normalmente só podem ser chamadas por processo a correr em nome do root
    - Exemplo: processo que controla a autenticação do utilizador na máquina

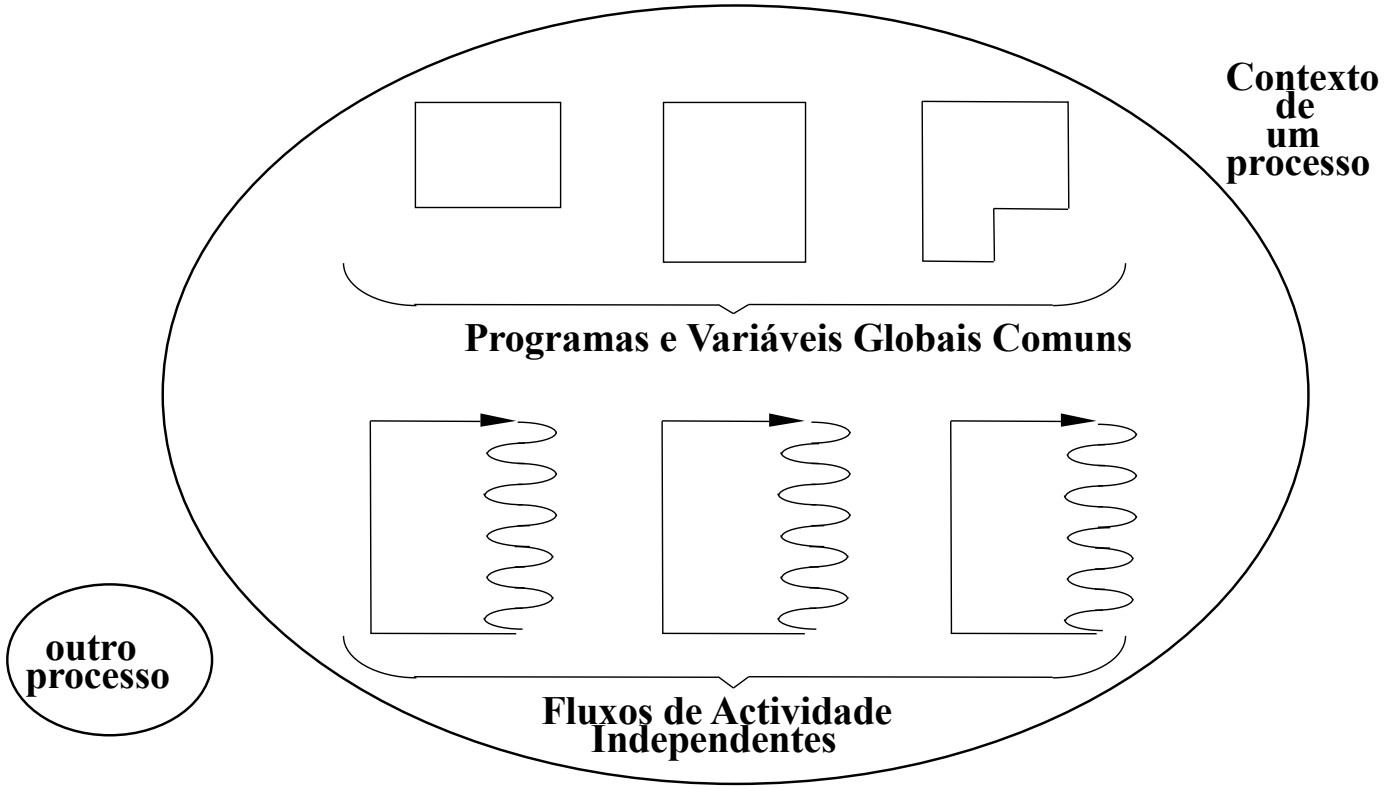
# Mudar de UID/GID

- Segunda via:
  - Executável de outro UID/GID com bit setUID/setGID ativo
  - Processo que execute esse executável (chamando exec*) adquire EUID/EGID do dono do ficheiro
- De que forma é útil?

# Introdução à programação com tarefas (*threads*)

# Tarefas

- Mecanismo simples para criar fluxos de execução independentes, partilhando um contexto comum



# O que é partilhado entre tarefas do mesmo processo

- O código
- Amontoado (heap)
  - Variáveis globais
  - Variáveis dinamicamente alocadas
- Atributos do processo

  (Veremos mais tarde)

# O que não é partilhado entre tarefas do mesmo processo

- Pilha (stack)
  - Mas atenção: não há isolamento entre pilhas!
  - *Bugs* podem fazer com que uma tarefa aceda à pilha de outra tarefa
- Estado dos registos do processador
  - Incluindo *instruction pointer*
- Atributos específicos da tarefa
  - Thread id (tid)
  - Etc (veremos mais tarde)

# Paralelismo com múltiplos processos vs. múltiplas tarefas (no mesmo processo)

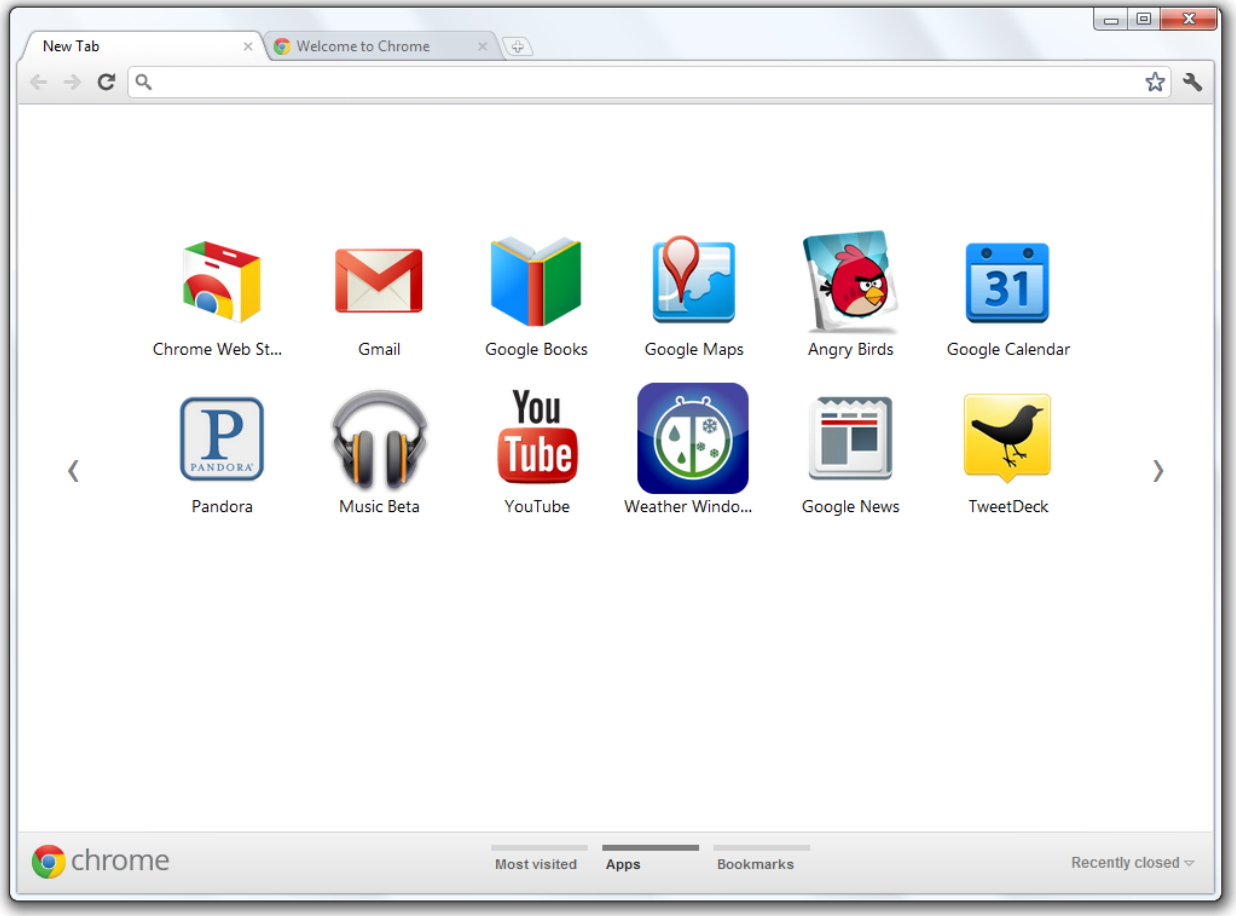
## Quando faz sentido paralelizar um projeto com cada alternativa?

# Paralelizar com processos vs. tarefas (no mesmo processo)

- Vantagens de multi-tarefa:
  - Criação e comutação entre tarefas do mesmo processo mais leves (vs. entre processos)
  - Tarefas podem comunicar através de memória partilhada
    - Comunicação entre processos mais limitada, mas vamos aprender mais no futuro
- Vantagens de processos:
  - Podemos executar diferentes binários em paralelo
  - Isolamento: confinamento de *bugs*
  - Outras (veremos mais tarde)

# Exemplo de uso de processos: Chrome

- No browser Chrome, criar um novo separador causa chamada a fork
- Processo filho usado para carregar e executar scripts dos sites abertos nesse separador
- Porquê?



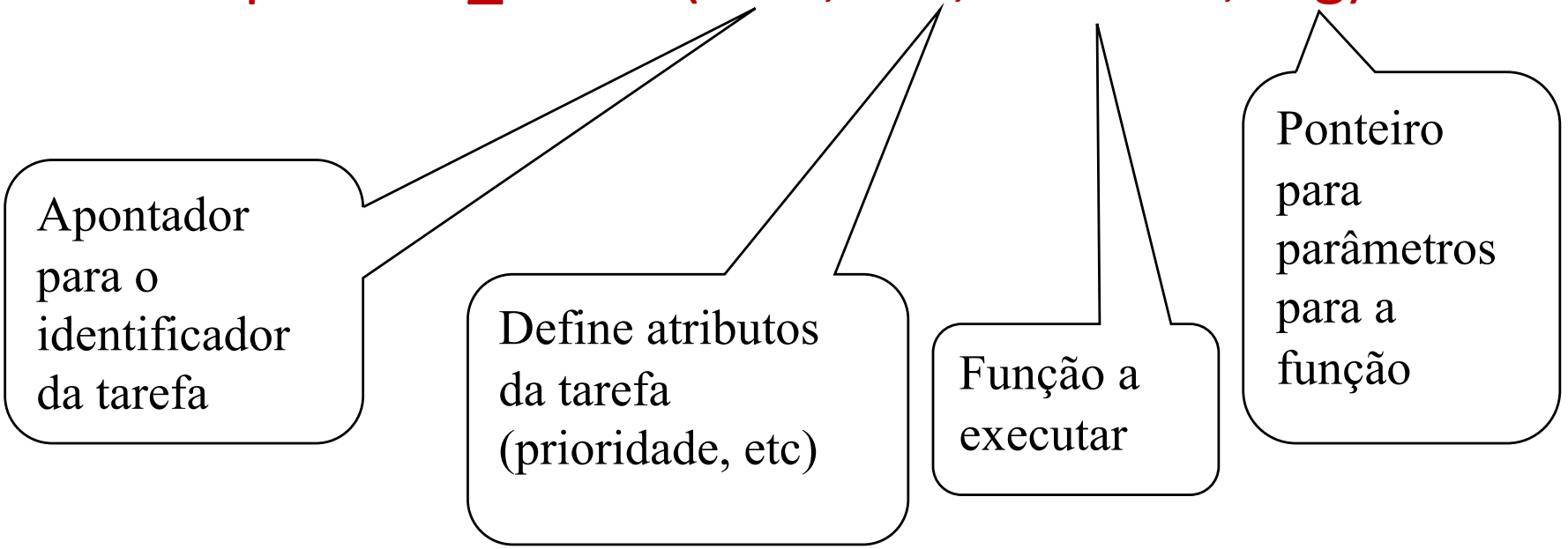
# Exemplo de uso de tarefas: TecnicoFS!

# Programação de processos multi-tarefa em Unix/etc

## Interface POSIX

# Interface POSIX: criar tarefa

pthread_create(&tid, attr, function, arg)



# Interface POSIX: terminação de tarefa

pthread_exit(void *value_ptr)

- Tarefa chamadora termina
- Retorna ponteiro para resultados

int pthread_join(pthread_t thread,
void *value_ptr)

- Tarefa chamadora espera até a tarefa indicada ter terminado
- O ponteiro retornado pela tarefa terminada é colocado em (*value_ptr)

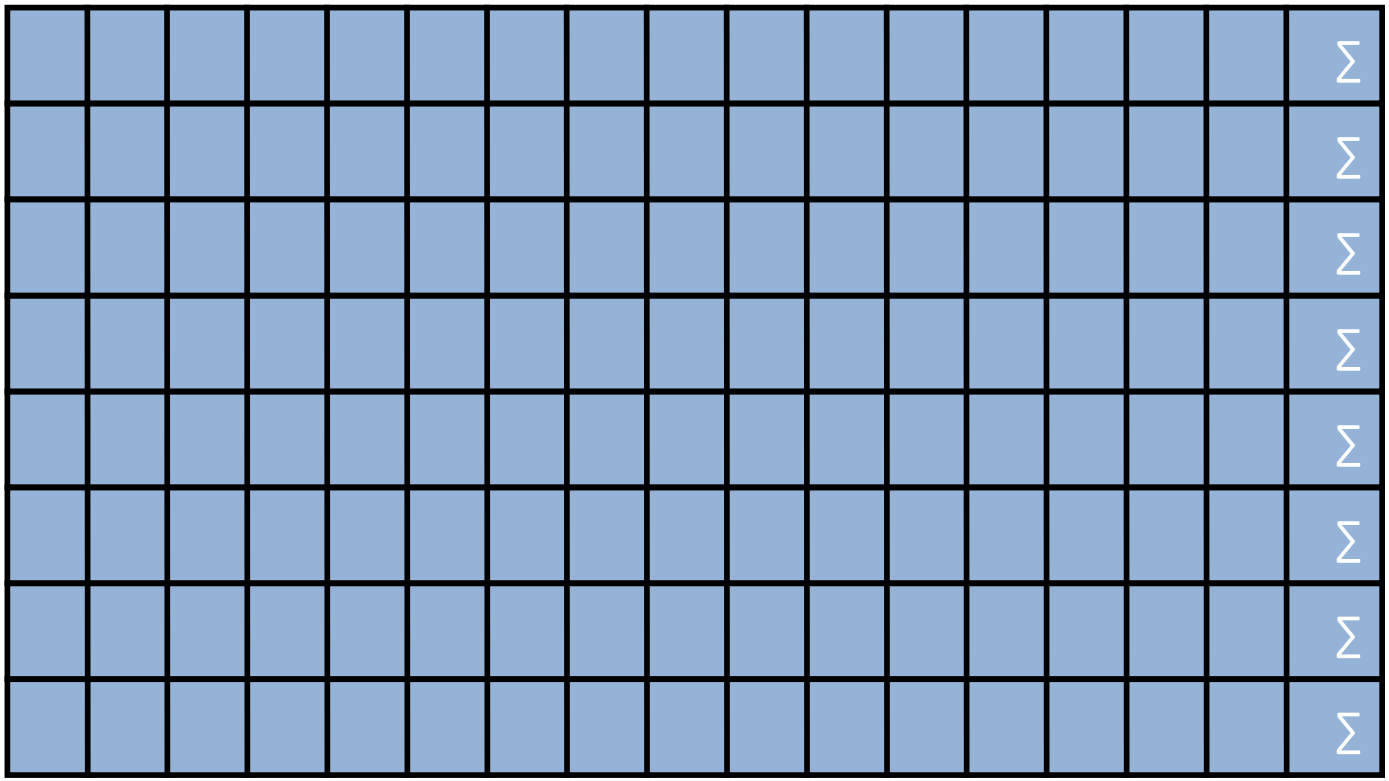
# Regra de ouro

- O núcleo oferece a ilusão de uma máquina com número infinito de processadores, sendo que cada tarefa corre no seu processador
- No entanto, as velocidades de cada processador virtual podem ser diferentes e não podem ser previstas
  - Porquê?
  - Consequências para o programador?

**Esta regra também se aplica a programação com processos paralelos**

https://tinyurl.com/so-threads-1

# Exemplo: somar linhas de matriz

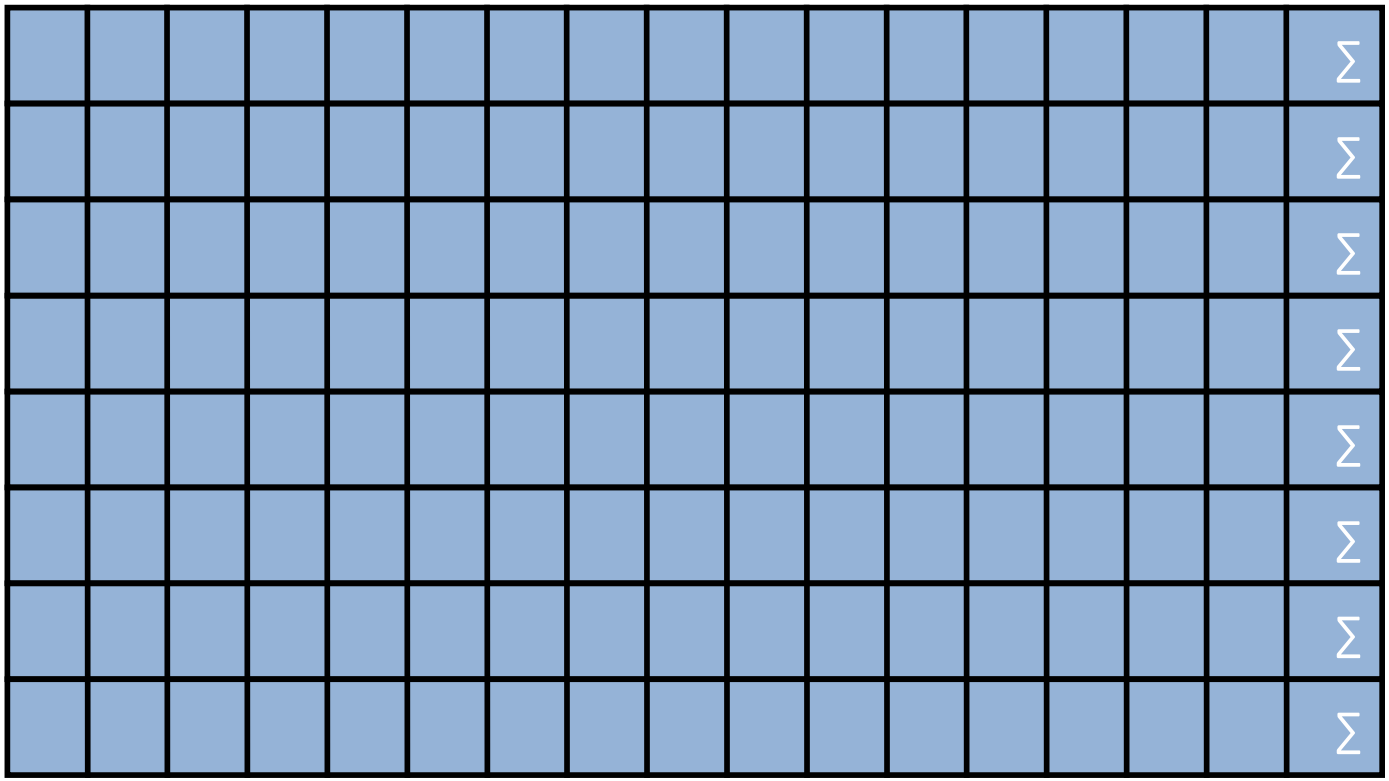


# Solução sequencial

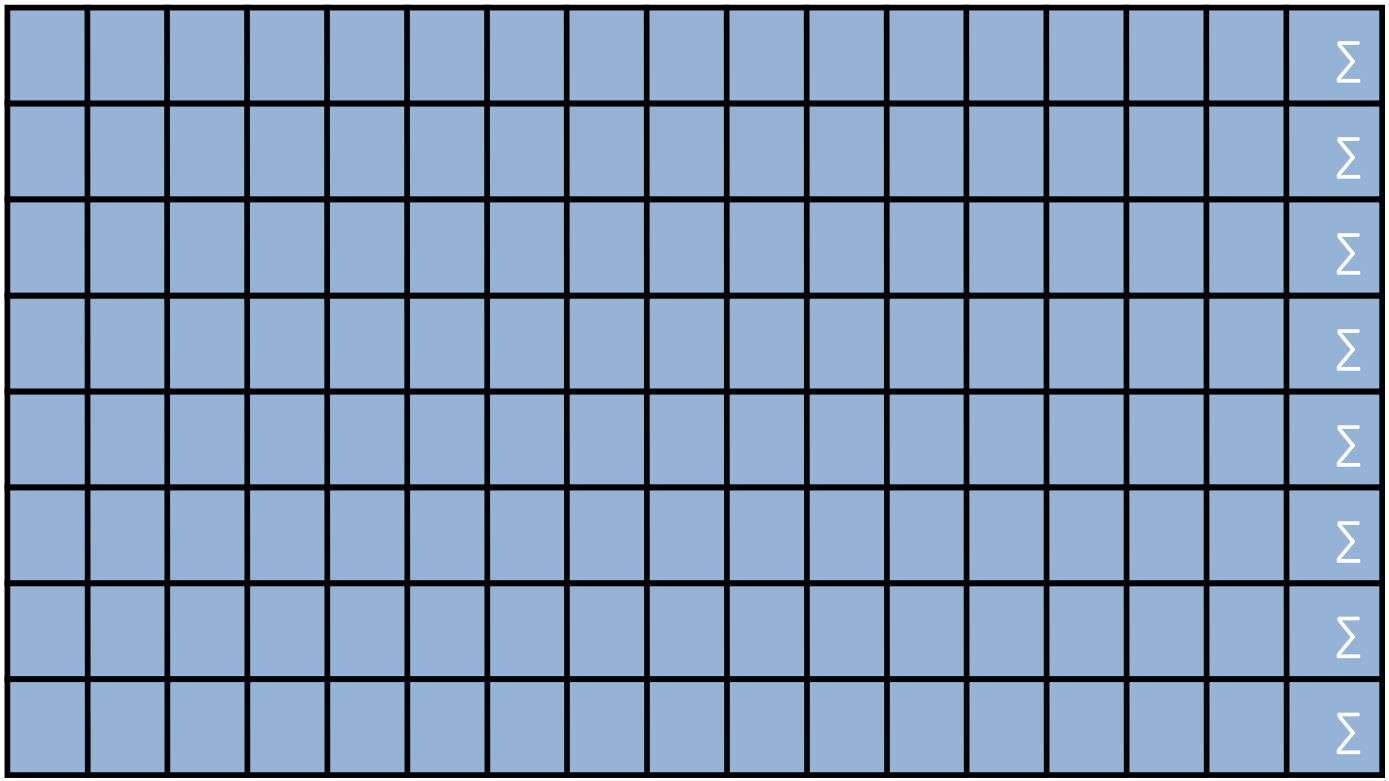
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];

void *soma_linha (int *linha) {
  int c, soma=0;
  int *b = linha;
  for (c = 0; c < TAMANHO - 1; c++) {
    soma += b[c];
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```

```
int main (void) {
  int i,j;

  inicializaMatriz(buffer(N, TAMANHO);

  for (i=0; i< N; i++)
   soma_linha(buffer[i]);


  imprimeResultados(buffer);

  exit(0);
}
```

# Execução sequencial



# Execução em N tarefas paralelas



# Exemplo (paralelo)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];

void *soma_linha (int *linha) {
  int c, soma=0;
  int *b = linha;
  for (c = 0; c < TAMANHO - 1; c++) {
    soma += b[c];
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```

# Exemplo (paralelo)

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];

void *soma_linha (void *linha) {
  int c, soma=0;
  int *b = (int*) linha;

  for (c=0;c<TAMANHO-1;c++) {
    soma += b[c];
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```

```c
int main (void) {
  int i,j;
  pthread_t tid[N];

  inicializaMatriz(buffer, N, TAMANHO);
  for (i=0; i< N; i++){
    if(pthread_create (&tid[i], 0, soma_linha,
                       buffer[i])== 0) {
      printf ("Criada a tarefa %d\n", tid[i]);
    }
    else {
      printf("Erro na criação da tarefa\n");
      exit(1);
    }
  }

  for (i=0; i<N; i++){
    pthread_join (tid[i], NULL);
  }
  printf ("Terminaram todas as threads\n");

  imprimeResultados(buffer);

  exit(0);
}
```

# Criação de tarefa: Como passar/receber parâmetros?

- Parâmetros podem ser de **qualquer** tipo, passados por referência *opaca* (void*)
- Parâmetro de entrada para a nova tarefa:
  - Através do argumento de *pthread_create*
  - Nova tarefa recebe parâmetro no argumento único da sua função
- Parâmetro de saída devolvido pela nova tarefa
  - Função da tarefa retorna ponteiro para o parâmetro
  - Tarefa criadora recebe esse ponteiro através de *pthread_join* (por referência)

# Atenção!
# Qual o problema neste programa?

```
void *threadFn(void *arg) {
 MyStruct *s = (MyStruct*) arg;
 printf("Nova thread criada com user %d:%s\n",
        s->userId, s->userName);
 ...
}


int main (void) {

 MyStruct s;

 for (i=0; i< N; i++){

   //Prepara argumentos da próxima thread
   s.userId = i;
   s.userName = getUserName(i);

   //Cria thread, passando-lhe um user novo
   if(pthread_create (&tid[i], 0, threadFn, &s)== 0) {
      printf ("Criada a tarefa %d\n", tid[i]);
   }
   else ...
 }
 ...
}
```

# E neste programa?

```
void *threadFn(void *arg) {
AnotherStruct r;
...
 r.x = ...;
 r.y = ...;
 return &r;
}


int main (void) {
AnotherStruct *s2;

//Cria thread, passando-lhe um user novo
 if (pthread_create (&tid, 0, threadFn, NULL) < 0) {
  ...
 }
 ...
 pthread_join(tid, &s2);
 printf("Tarefa devolveu: %d, %d\n", s2->x, s2->y);
}
```

# Então e se o programa não for embaraçosamente paralelo?