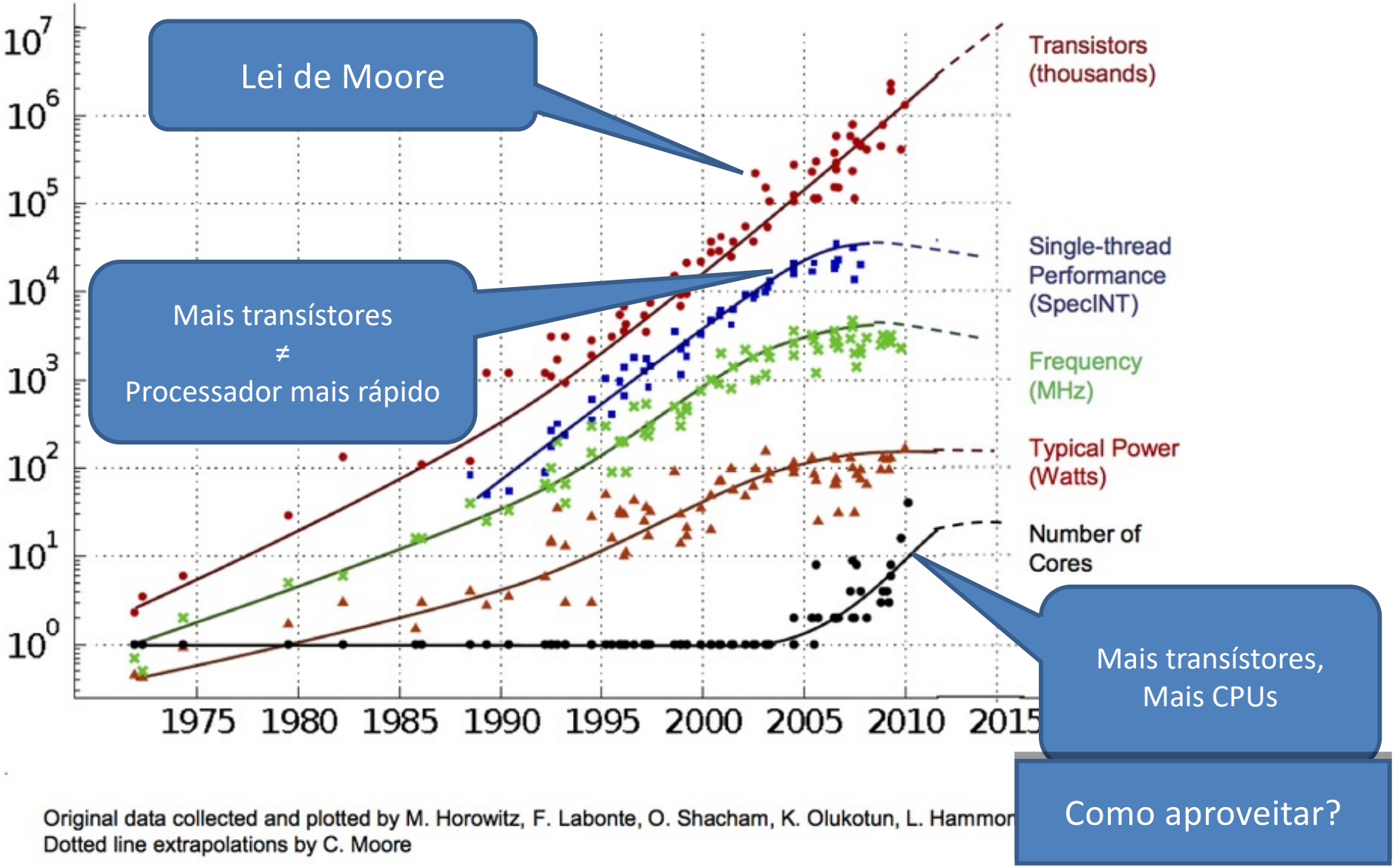
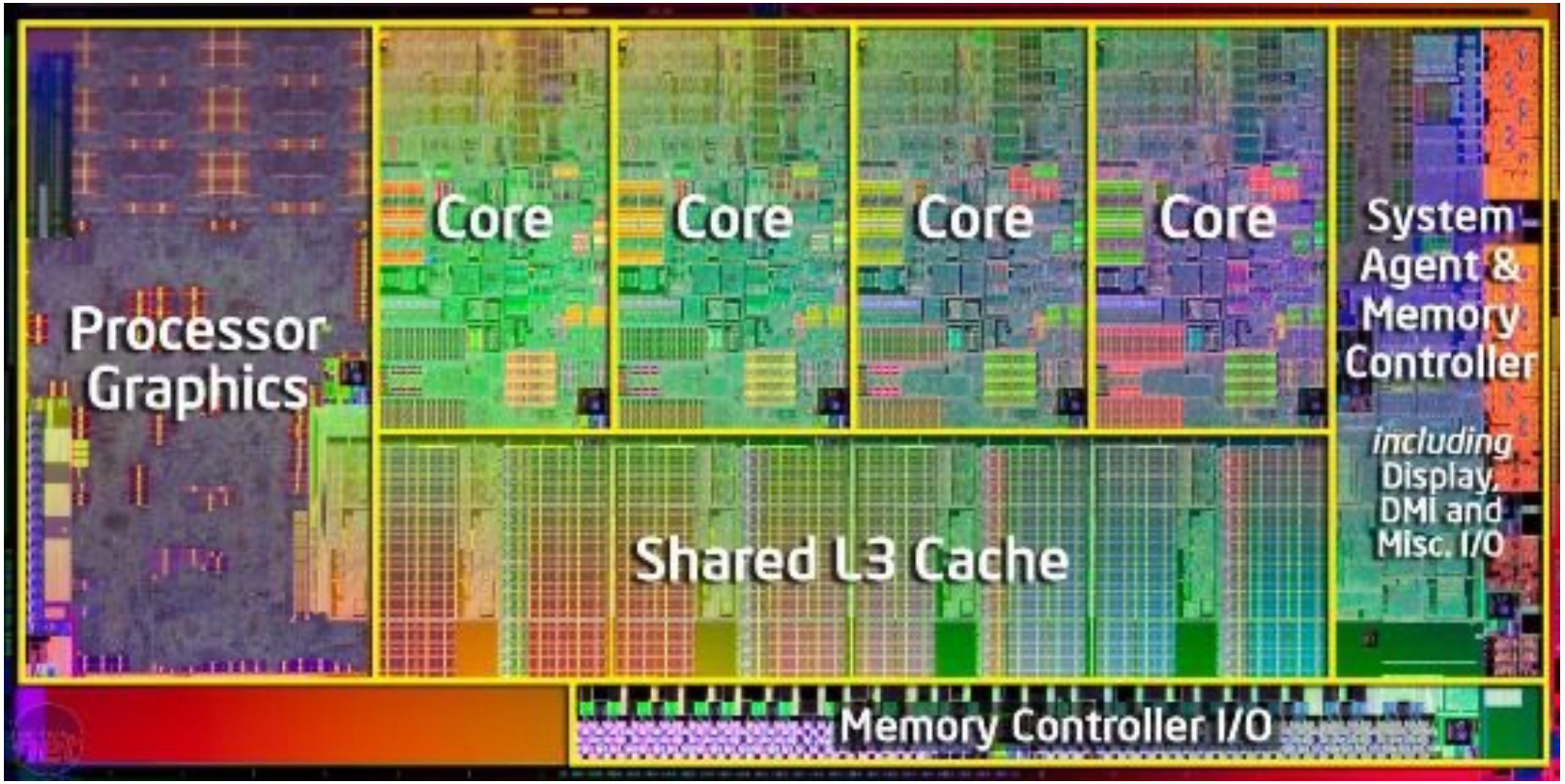
O fim dos “almoços gratis”



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-cpu!*

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

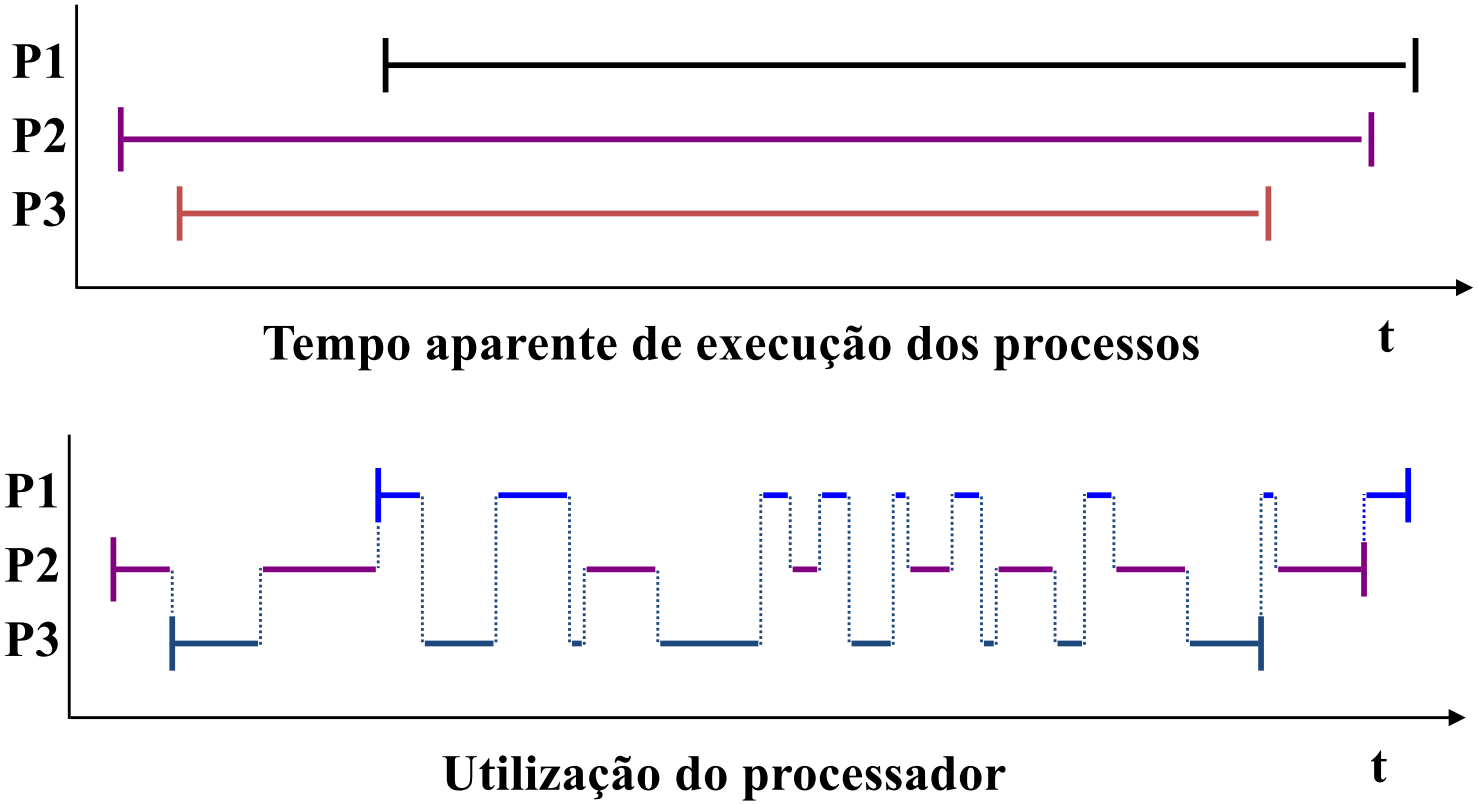
Introdução à programação com processos

Multiprogramação

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

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

Pseudoconcorrência



Processo = Programa?

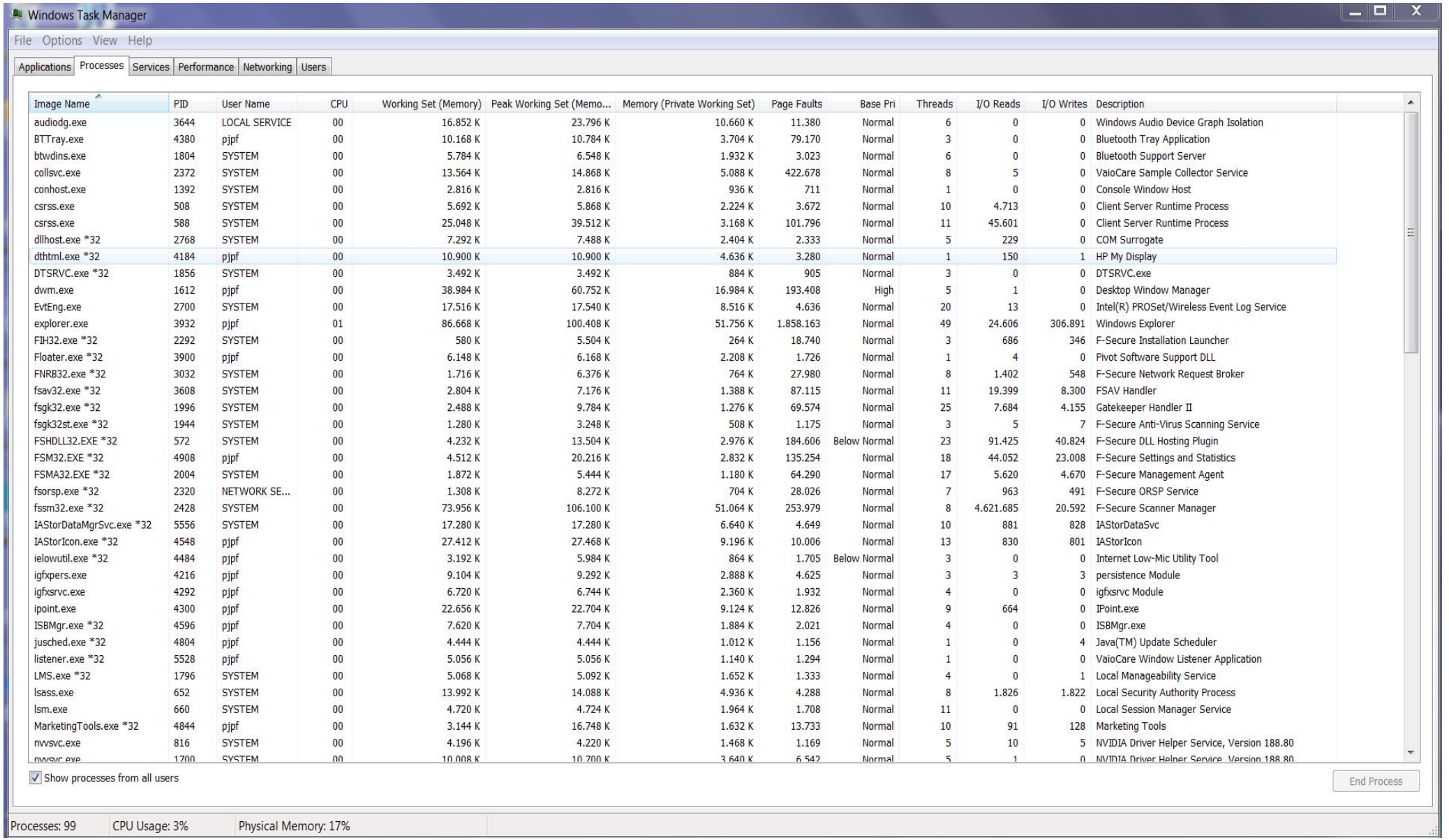
- Programa = Fich. executável (sem actividade)
- Um processo é um objecto do sistema operativo que suporta a execução dos programas
- Um processo pode, durante a sua vida, executar diversos programas
- Um programa ou partes de um programa podem ser partilhados por diversos processos
  - Ex.: biblioteca partilhadas DLL no Windows

Exemplo: Unix

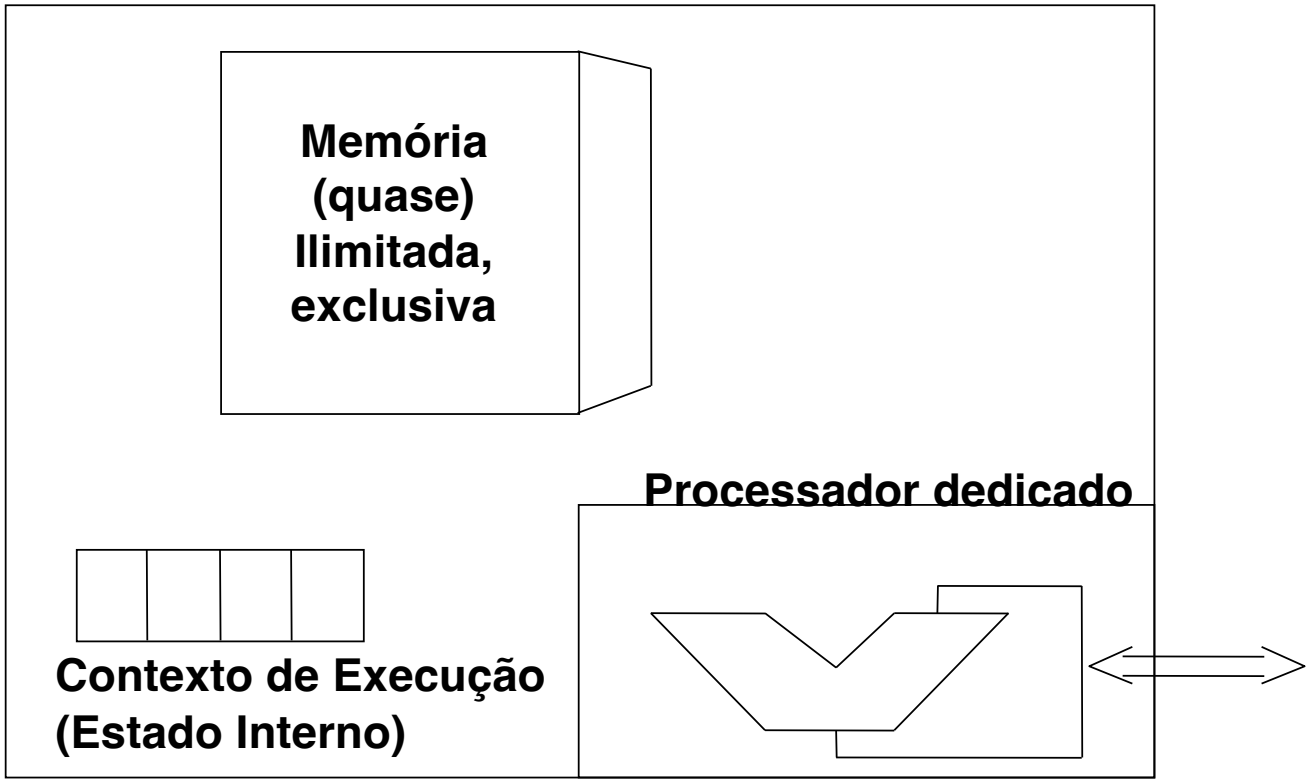
```
ps -el | more
   UID   PID  PPID  C    STIME TTY  TIME CMD
  root     0     0  0   Sep 18 ?    0:17 sched
  root     1     0  0   Sep 18 ?    0:54 /etc/init -
  root     2     0  0   Sep 18 ?    0:00 pageout
  root     3     0  0   Sep 18 ?    6:15 fsflush
  root   418     1  0   Sep 18 ?    0:00 /usr/lib/saf/sac -t 300
daemon   156     1  0   Sep 18 ?    0:00 /usr/lib/nfs/statd
```

```
ps      displays information about a selection of the
active processes.
e       select all processes
l       long format
```

Exemplo: Windows



Processo como uma Máquina Virtual



Elementos principais da máquina virtual que o SO disponibiliza aos processos

Processo como uma Máquina Virtual

- Tal como uma máquina real, um processo tem:
  - Espaço de endereçamento (virtual):
    - Conjunto de posições de memória acessíveis
    - Código, dados e pilha
    - Dimensão variável
  - Reportório de instruções:
    - As instruções do processador executáveis em modo utilizador
    - As funções do sistema operativo
  - Contexto de execução (estado interno):
    - Toda a informação necessária para retomar a execução do processo
    - Memorizado quando o processo é retirado de execução

Modelo: Objecto “Processo”

- Propriedades
  - Identificador
  - Programa
  - Espaço de Endereçamento (codigo, dados, pilha)
  - Prioridade
  - Processo pai
  - Canais de Entrada Saída, Ficheiros,
  - Quotas de utilização de recursos
  - Contexto de Segurança

- Operações – Funções sistema que actuam sobre os processos
  - Criar
  - Eliminar
  - Esperar pela terminação de subprocesso

Programação com processos em Unix

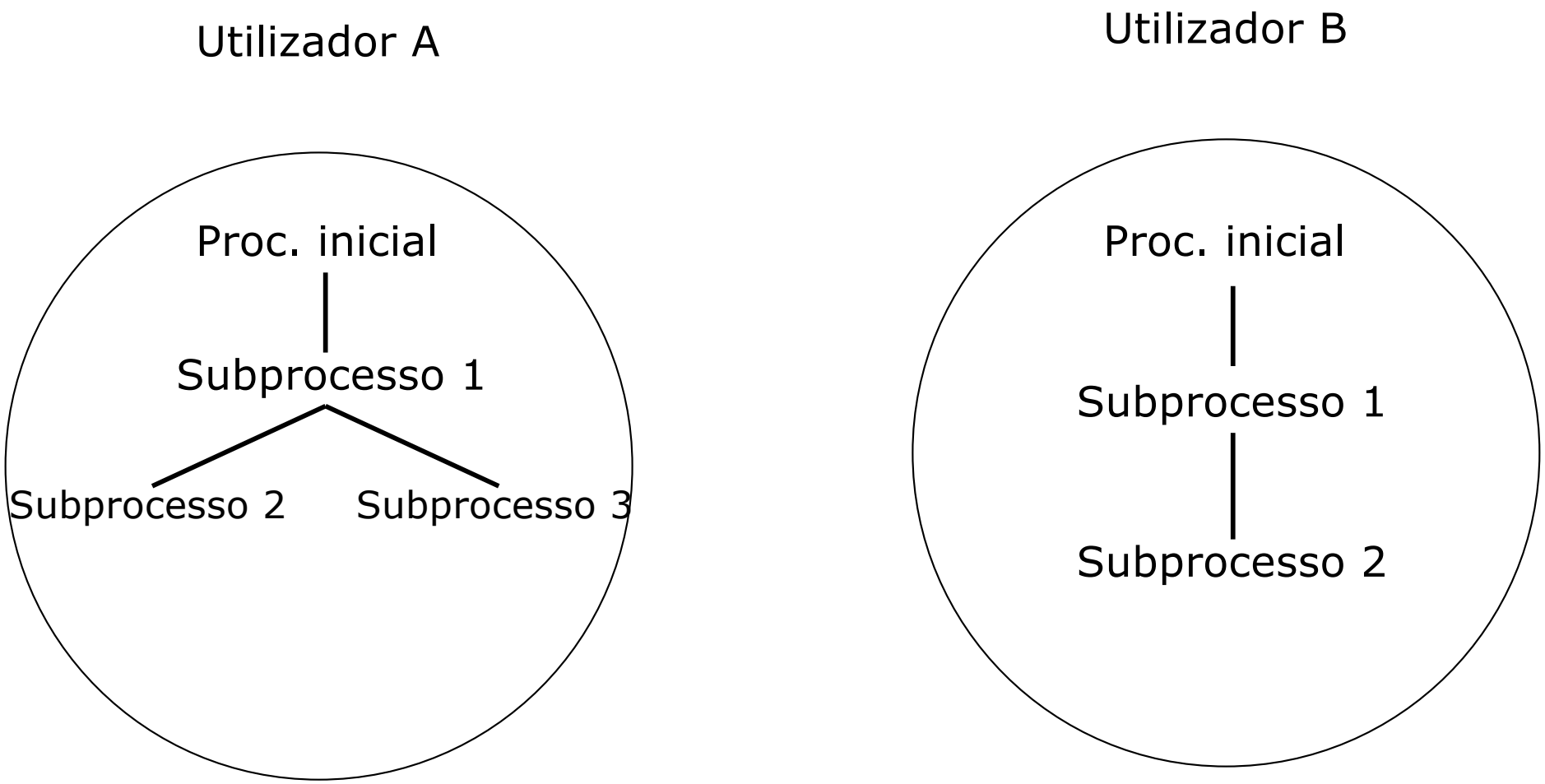
Processos em Unix

- Processos identificados por inteiro (PID)
- Alguns identificadores estão pré atribuídos:
  - Processo 0 é o *swapper* (gestão de memória)
  - Processo 1 init é o de inicialização do sistema

Hierarquia de processos

- Processos relacionam-se de forma hierárquica
- Novo processo herda grande parte do contexto do processo pai
- Quando o processo pai termina os subprocessos continuam a executar-se
    - São adoptados pelo processo de inicialização (pid = 1)

Hierarquia de Processos



Certas propriedades são herdadas

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?

Como ter o filho a executar um programa diferente?

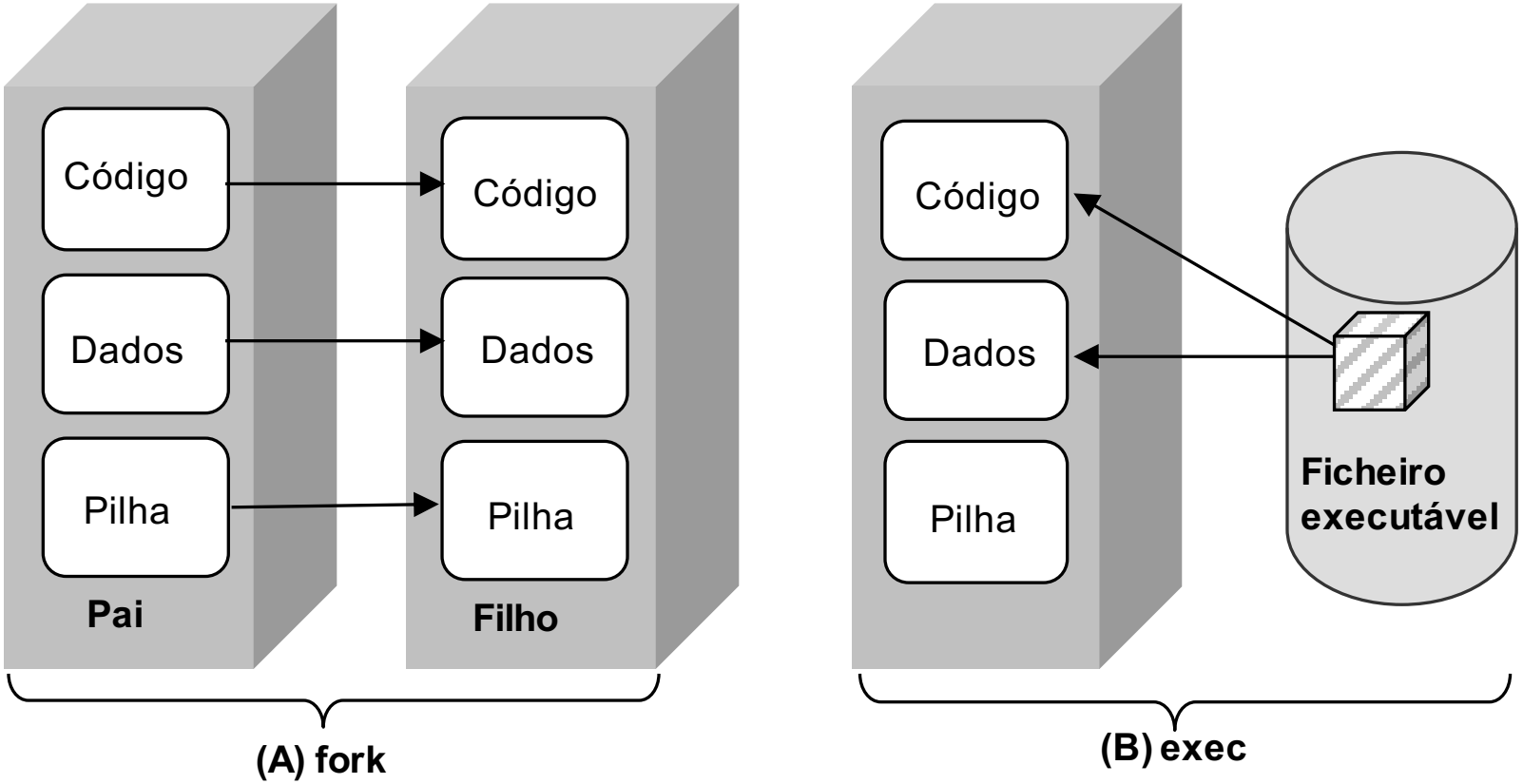
```
int execl(char* ficheiro, char* arg0, char* argl,…, argn,0)

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**

Comparação *fork* e *exec*



Como ter filho a executar programa diferente?

- A função `exec` **substitui** o espaço de endereçamento do processo onde é invocada por aquele contido num ficheiro executável:
  - Programa substituído pelo programa no ficheiro
  - Dados substituídos pelos dados iniciais do ficheiro
  - Pilha é esvaziada
  - *Instruction pointer* aponta para a 1ª instrução do *main* do novo programa

Como ter o filho a executar um programa diferente?

- Quando exec tem sucesso, não há retorno
  - Porquê?
- Restante contexto de execução mantém-se intacto
  - Exemplos de atributos que se mantêm?
  - Como fazer caso se queira que alguns desses atributos não sejam herdados pelo filho?

Exemplo de Exec

```
main ()
{
     int pid;

     pid = fork ();
     if (pid == 0) {
          execl ("/usr/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

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);
    }
}
```

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 UID/GID efectivo (*effective*)
  - Restrições caso sejam chamadas por processo sem o privilégio Sistema

Mudar de UID/GID

- Segunda via:
  - Utilizando o `exec`
  - Ficheiro executável com **com bit setUID/setGID posicionado e outro UID/GID**
  - Processo que execute esse executável (chamando `exec`) adquire UID/GID do dono do ficheiro

Exemplo de utilização do setUID

```
$  which whoami
     /usr/bin/whoami
$  cp /usr/bin/whoami .    #copio o executável para a directoria corrente
$  ls –l whoami
     -rwxr-xr-x  1 paolo  staff  23264 Dec  5 10:12 whoami
$  ./whoami
     paolo
$  sudo chown root whoami
$  ls –l whoami
   -rwxr-xr-x  1 root   staff  23264 Dec  5 10:12 whoami
$  sudo chmod u+s whoami
$  ls –l whoami
   -rwsr-xr-x  1 root   staff  23264 Dec  5 10:12 whoami
$   ./whoami
    root
```

Exemplo de utilização do setUID (2)

- No linux, as passwords (criptadas) são armazenadas no ficheiro /etc/shadow

> ls –l /etc/shadow

-rw-r—– 1 root shadow 1824 Oct 18 19:49 /etc/shadow

- O commando *passwd* permite que os utilizadores (não superusers) possam alterar as próprias passwords
- Problema: *passwd* precisa aceder /etc/shadow para mudar as passwords

> ls -l /bin/passwd

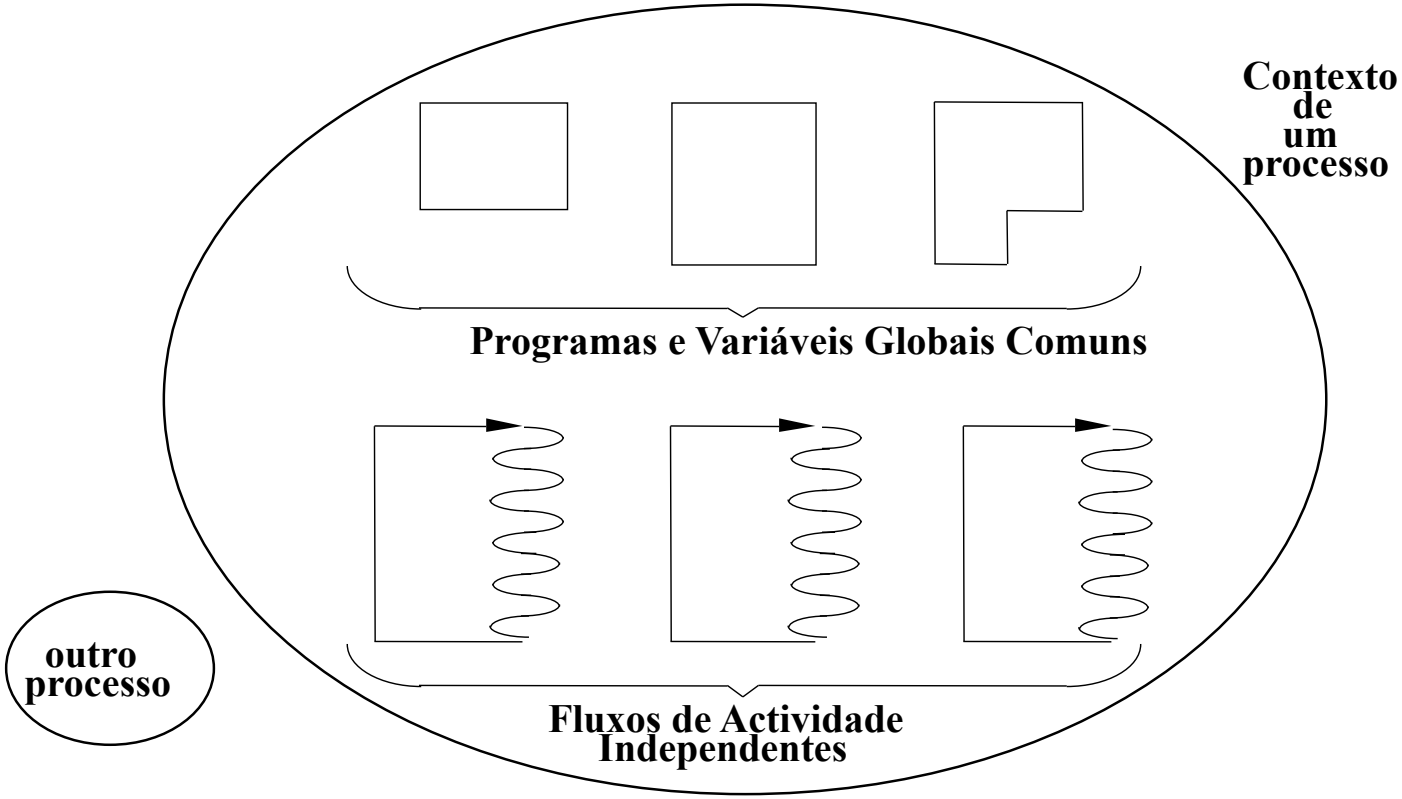
-rw**s**r-xr-x 1 root root 68208 May 28 01:37 /bin/passwd

- Solução: dono root e user sticky bit ativo!

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

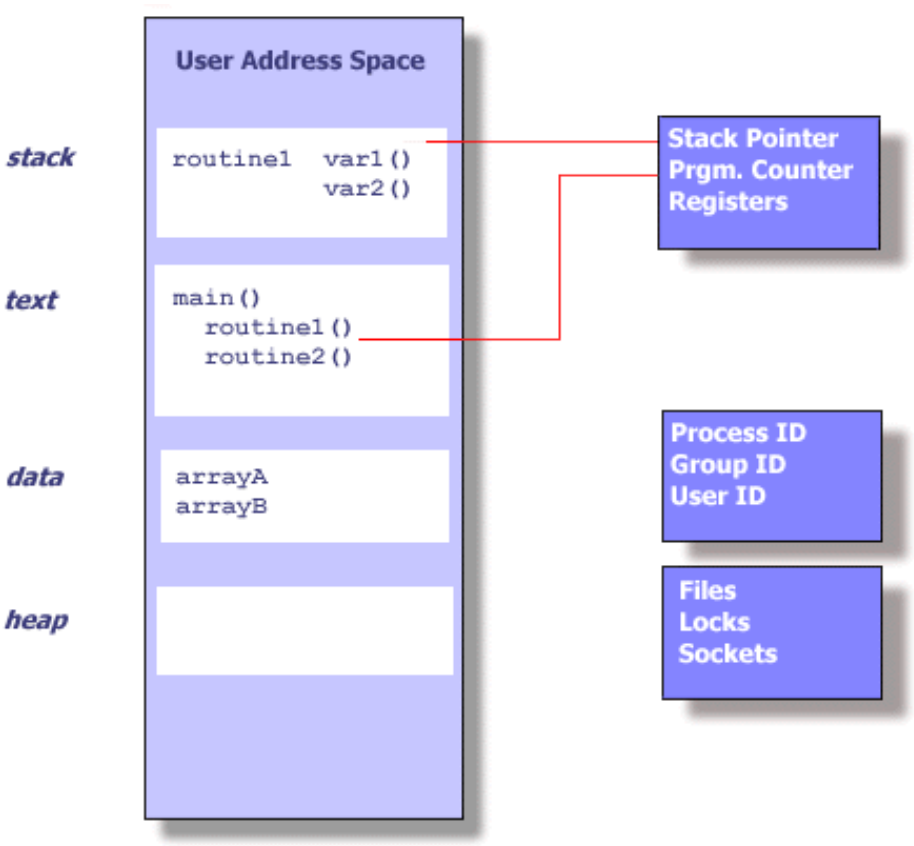
- O código
- Amontoado (heap)
  - Variáveis globais
  - Variáveis dinamicamente alocadas
- Atributos do processo

  (Veremos mais tarde)

O que não é partilhado entre tarefas do mesmo processo

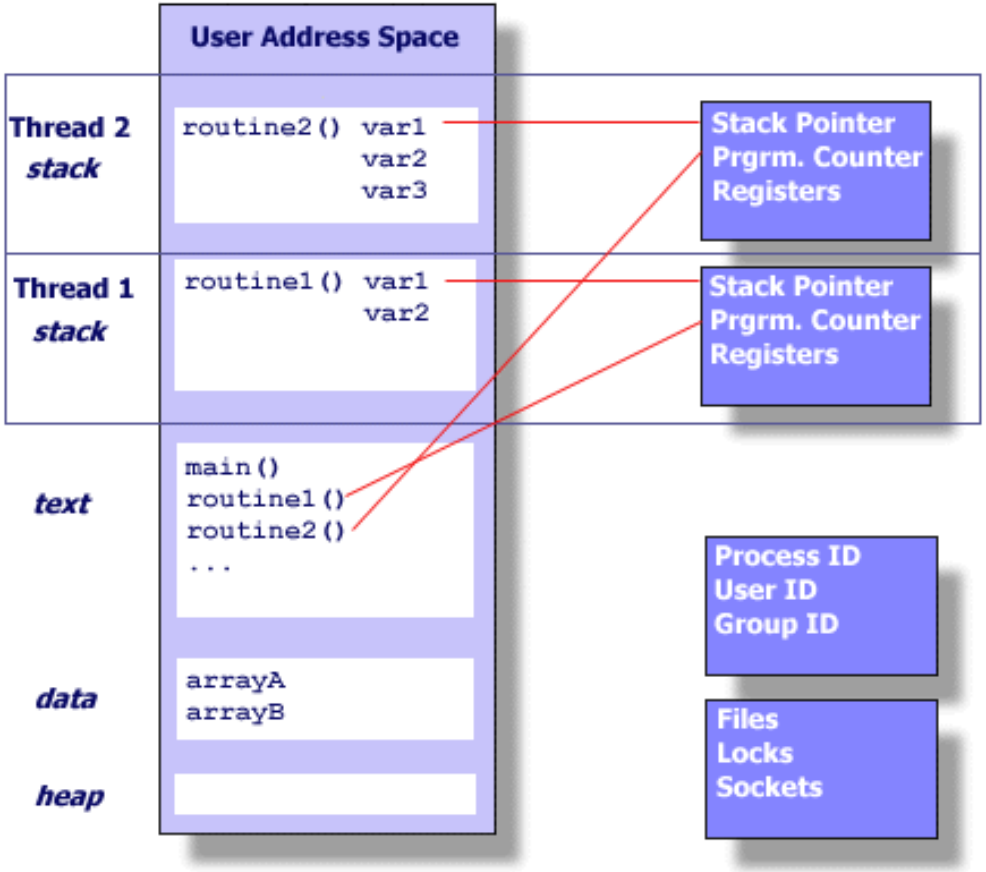
- Pilha (stack)
  - Mas atenção: não há isolamento entre pilhas!
  - *Bugs* podem fazer com que uma tarefa aceda à pilha de outra tarefa
- Estado dos registos do processador
  - Incluindo *instruction pointer*
- Atributos específicos da tarefa
  - Thread id (tid)
  - Etc (veremos mais tarde)

Um processo Unix



em Posix Programming threads - https://computing.llnl.gov/tutorials/pthreads/

Um processo unix com duas tarefas (threads)



em Posix Programming threads - https://computing.llnl.gov/tutorials/pthreads/

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

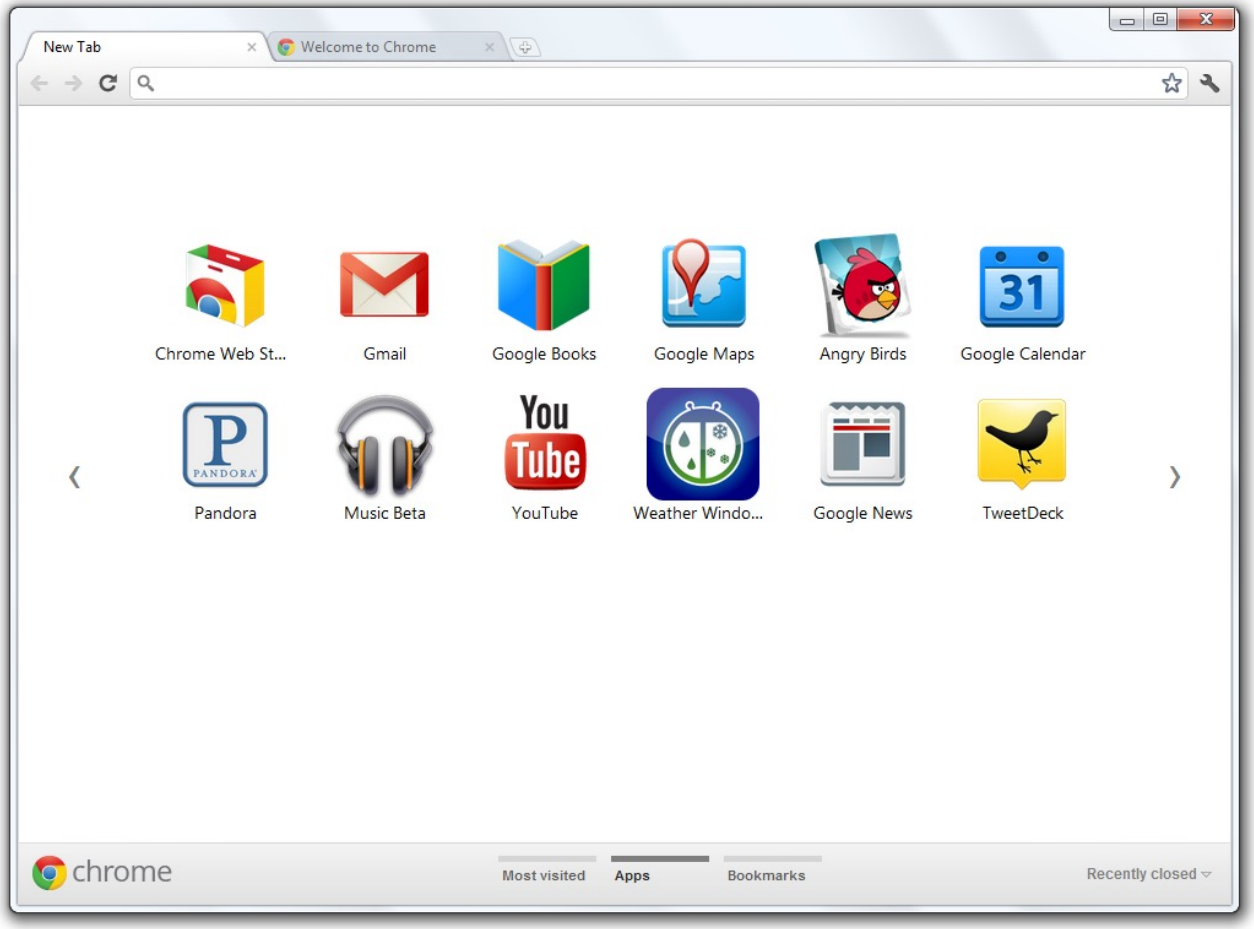
Quando faz sentido paralelizar um projeto com cada alternativa?

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?



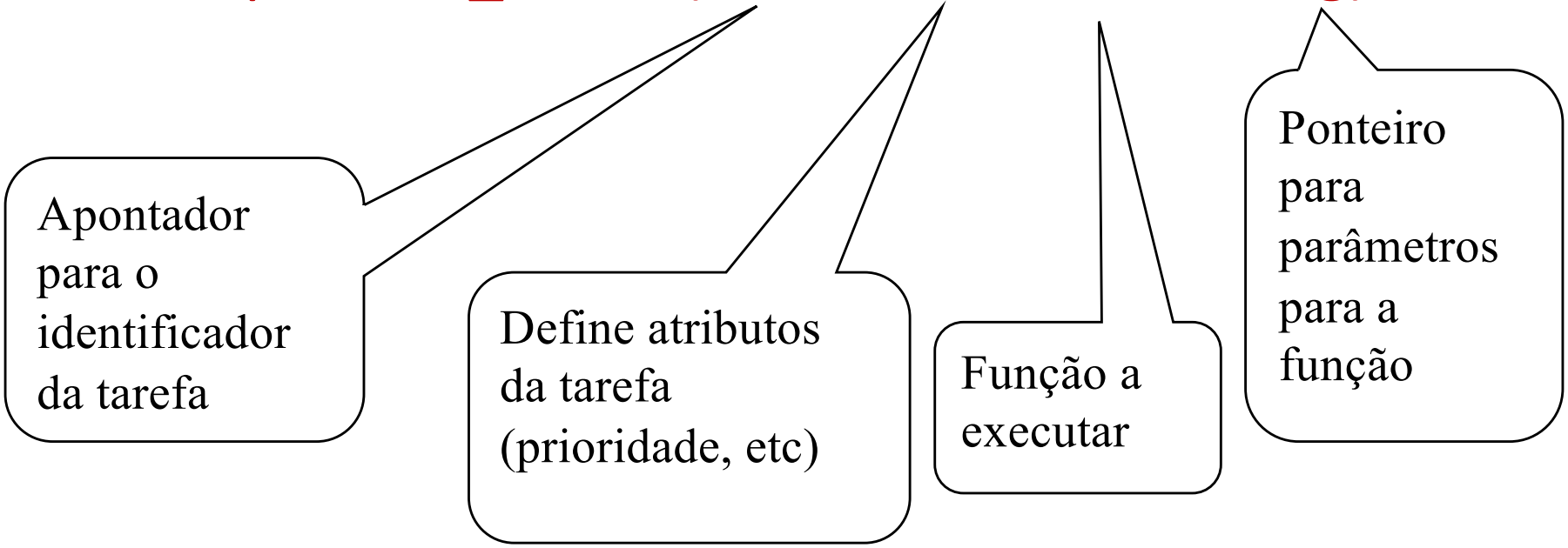
Exemplo de uso de tarefas: IST-KVS!

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)

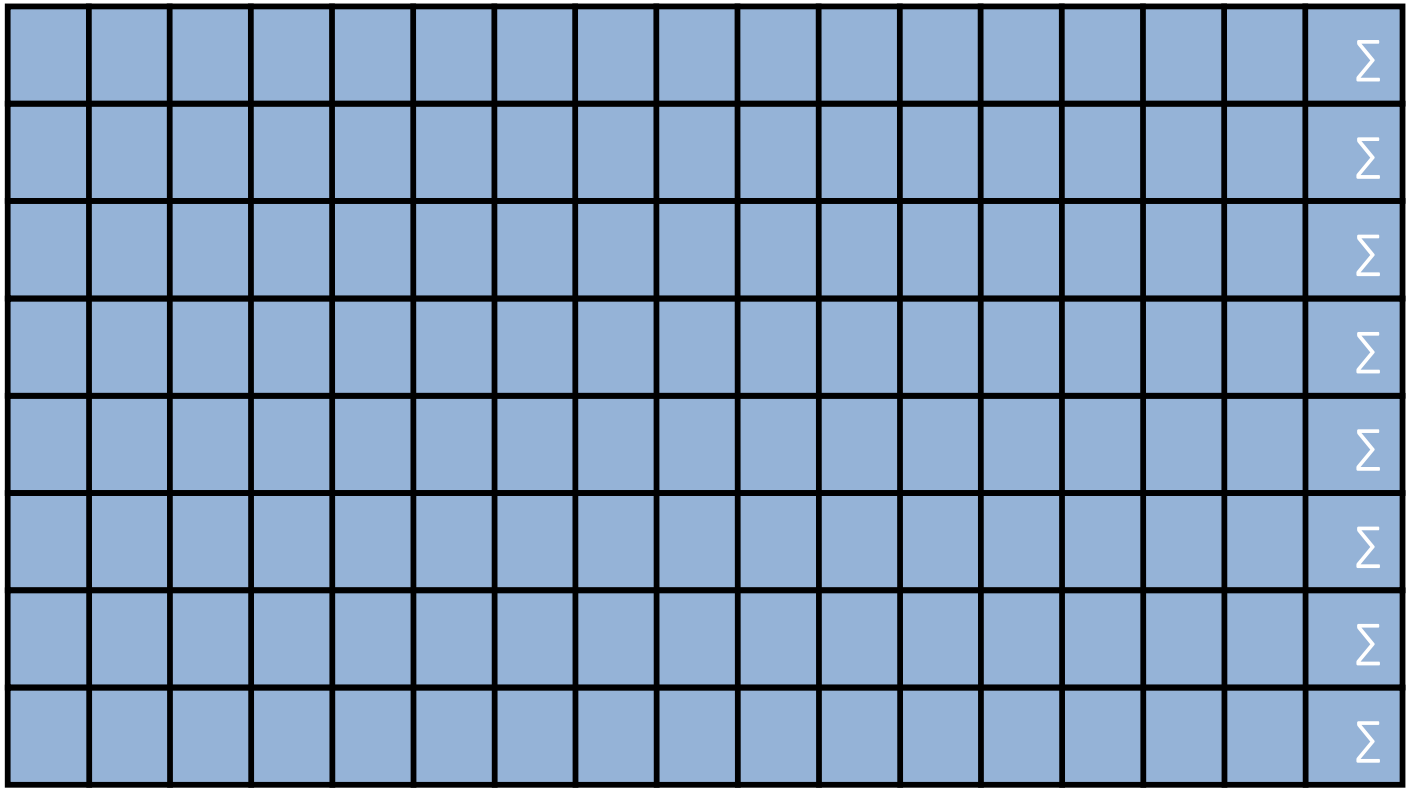
- Tarefa chamadora espera até a tarefa indicada ter terminado
- O ponteiro retornado pela tarefa terminada é colocado em (*value_ptr)

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

Exemplo: somar linhas de matriz

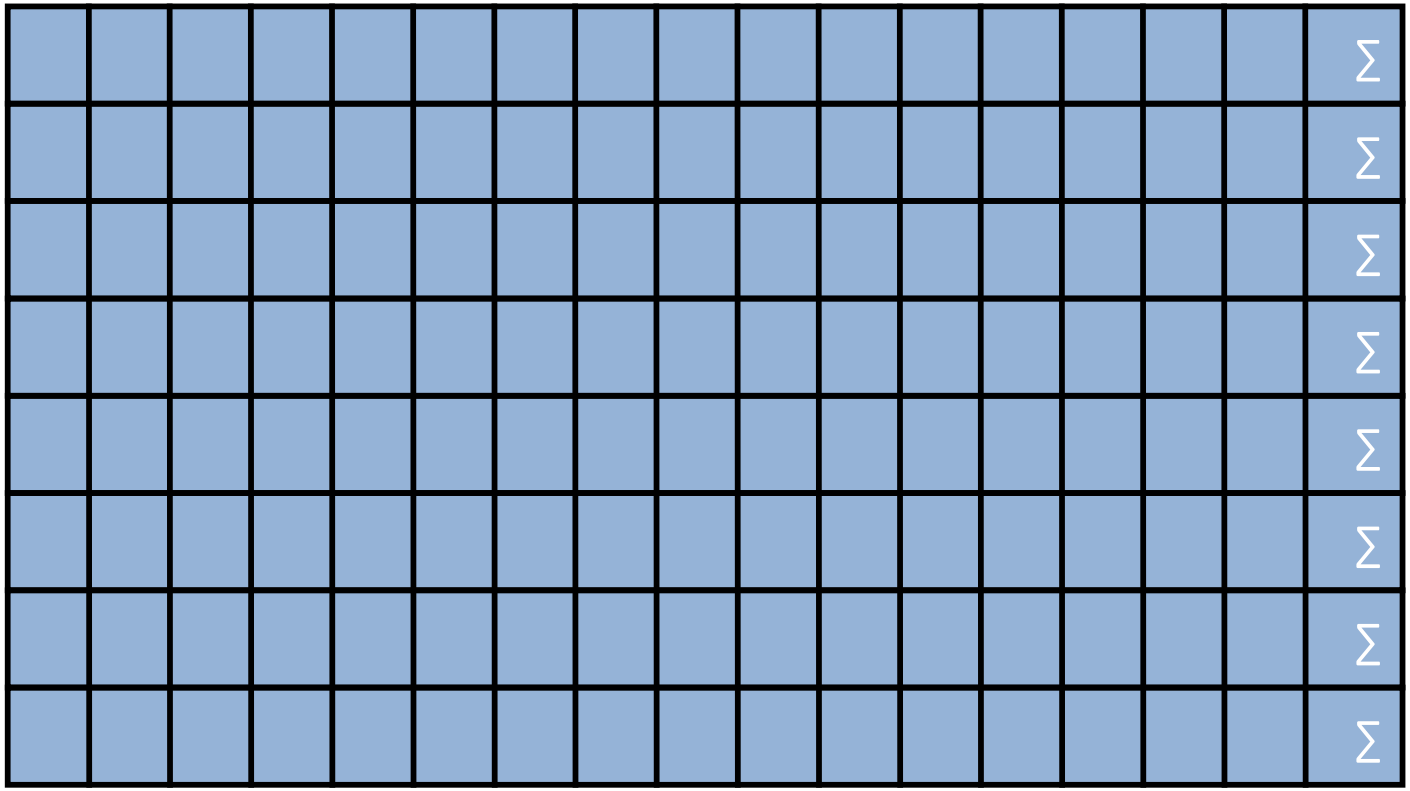


Solução sequencial

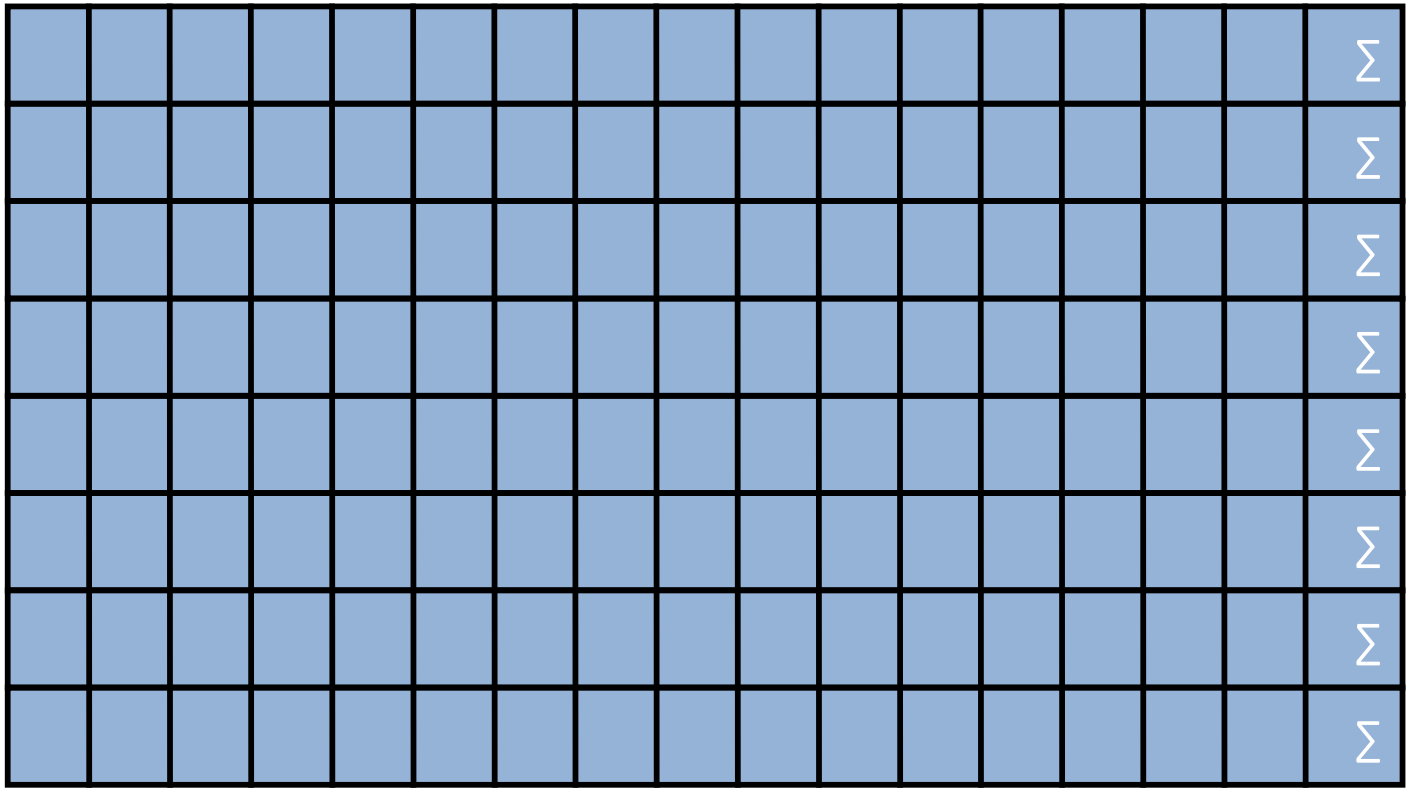
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

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)

```
#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;
}
```

```
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?