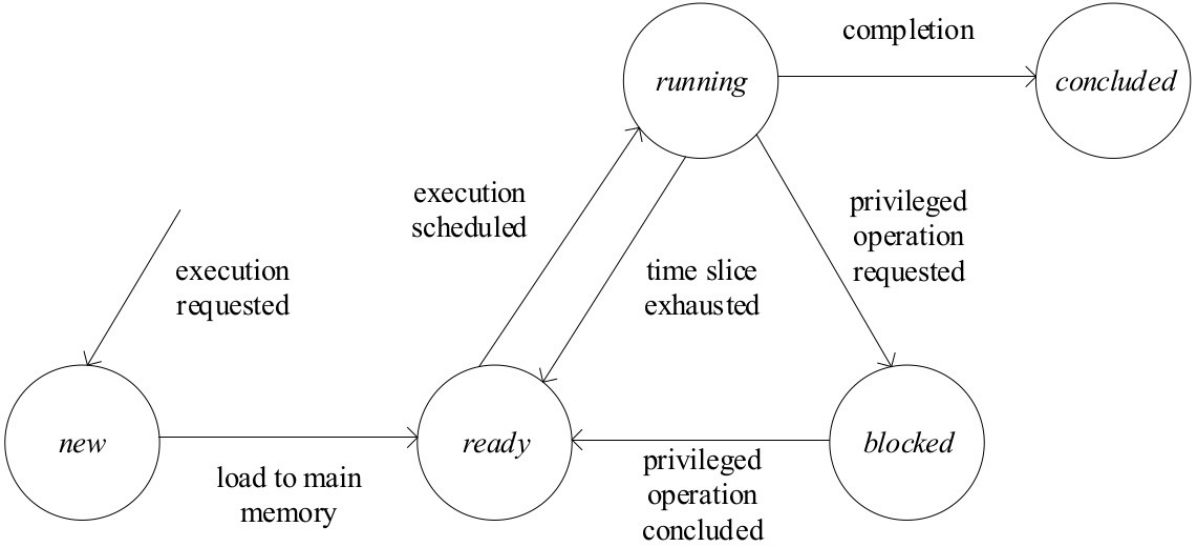
Sistemas Operativos

Processos

- Organização da Memória de um Processo
- Relações entre Processos
- Identificação de Processos (getpid e getppid)
- Criação e Terminação de Processos (fork e exit)

Programa vs. Processo

- Um programa é uma sequência de instruções
- **Um processo é um contexto de execução do programa (“ é um programa em execução)**
- Ao longo da sua vida, um processo percorre vários estados



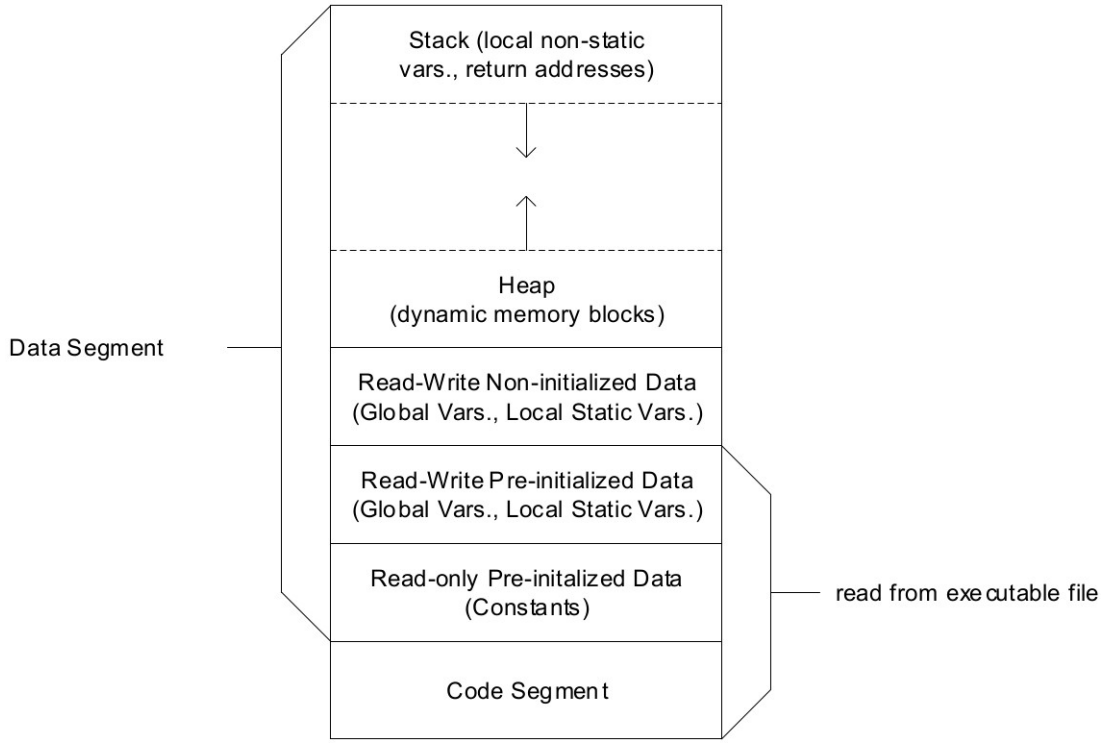
Organização da Memória de um Processo

- A memória (virtual) atribuída a um processo organiza-se em duas zonas
  - Zona de **Nível Utilizador**: a única zona acessível em Modo Utilizador
  - Zona de **Nível Supervisor**: apenas acessível em Modo Supervisor

Organização da Memória de um Processo

- A Zona de **Nível Supervisor** (ou **Bloco de Controlo** de um Processo) só é acessível em Modo de execução Supervisor/Privilegiado
- Nesta zona está a informação de gestão do processo, durante o seu tempo de vida:
  - o estado do processo
  - o conteúdo dos registos da CPU (usado na comutação de processos)
  - informação de escalonamento (prioridade, fatia de tempo de CPU, etc.)
  - informação de gestão de memória (localização e dimensão das suas zonas)
  - informação de dos recursos consumidos
  - tabelas de descritores de ficheiros, etc.

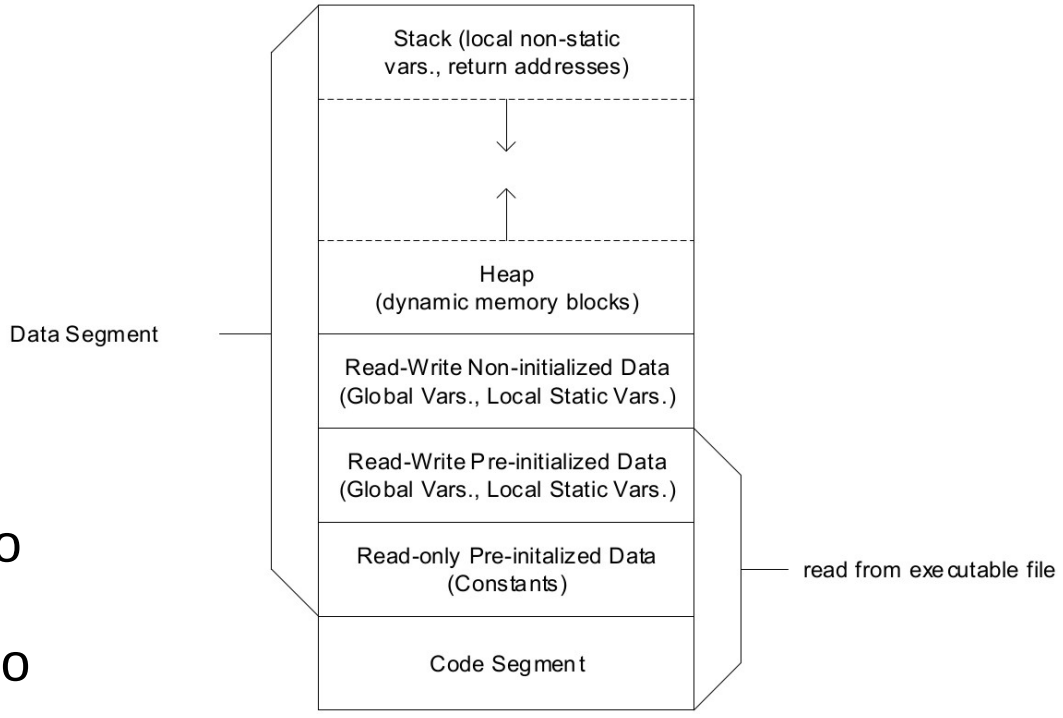
Organização da Memória de um Processo



A Zona de Nível Utilizador consiste no **Segmento de Código** e no **Segmento de Dados**

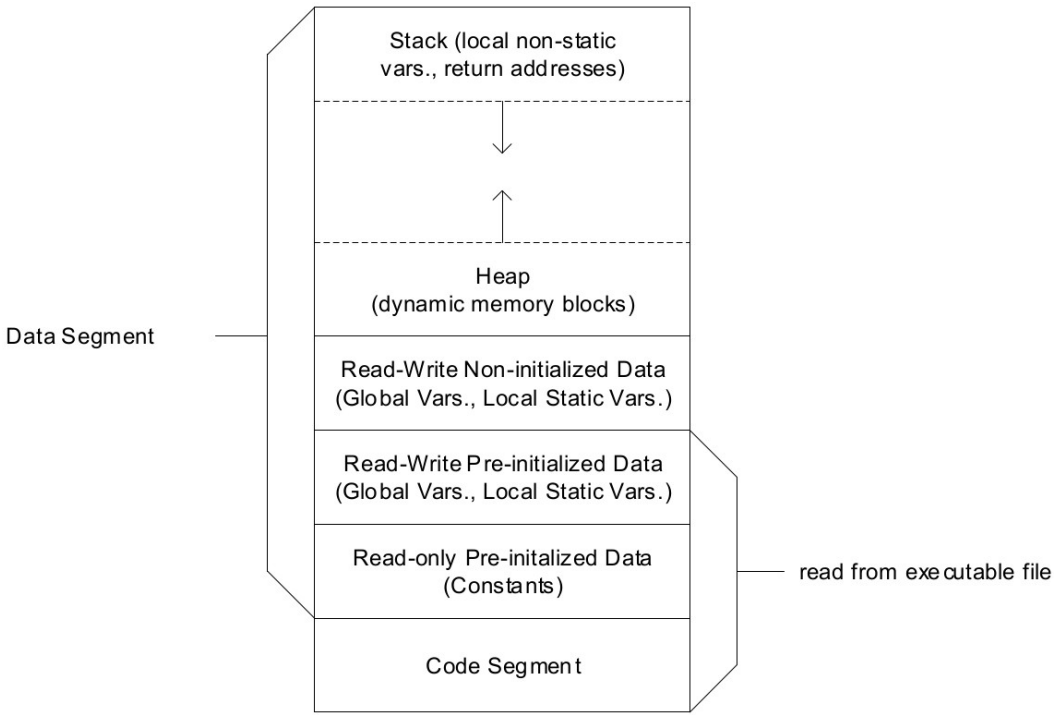
Organização da Memória de um Processo

- **Segmento de Código**
  - sequência de instruções do programa a executar
  - carregada a partir do ficheiro executável do programa
  - é apenas de leitura (o programa não se pode auto-modificar)
  - assim possibilitando a partilha do mesmo segmento de código por vários processos que executam o mesmo programa



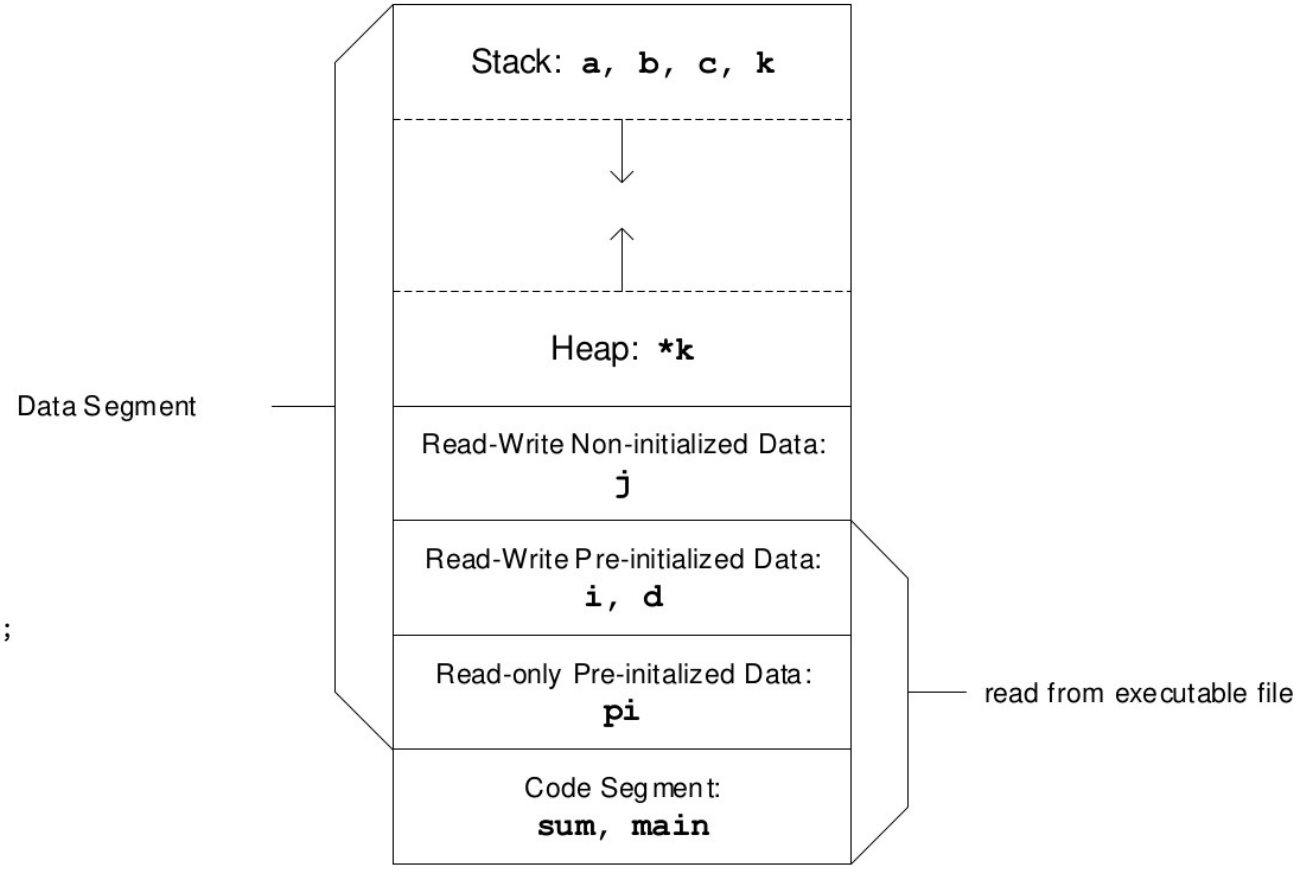
Organização da Memória de um Processo

- **Segmento de Dados**
  - **Stack/Pilha**: guarda o endereço de retorno (valor que o Program Counter deve assumir após a execução da função), o conteúdo dos registos da CPU imediatamente antes da invocação, os parâmetros, as variáveis locais e eventual valor de retorno
  - **Heap**: para os pedidos de memória dinâmica
  - **Dados de Leitura e Escrita, Não-Inicializados**: variáveis globais e variáveis estáticas, não-inicializadas explicitamente pelo programa
  - **Dados de Leitura e Escrita, Inicializados**: variáveis globais e variáveis estáticas, inicializadas
  - **Dados apenas de Leitura, Inicializados**: constantes



Distribuição na memória dos objetos de um programa

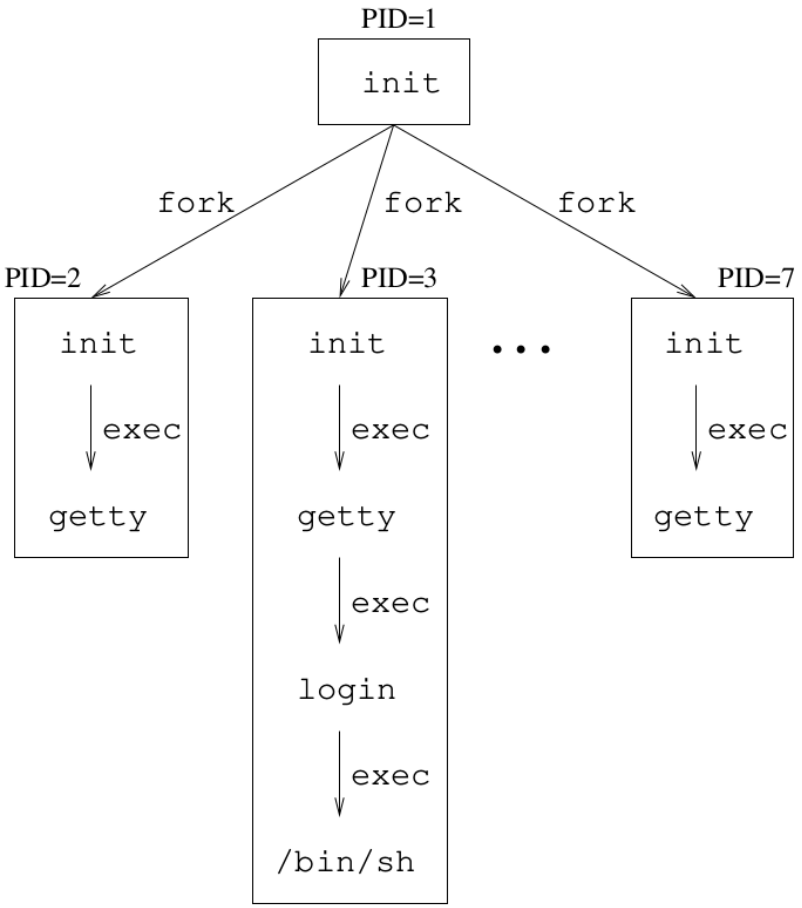
```
#define pi 3.1415927

float i=pi, j;

float sum(float a, float b)
{
  float c=0; static float d=0;
  d=a+b+c+d; return(d);
}

int main()
{
  float *k;

  j=sum(i,pi);
  k=(float*)malloc(sizeof(float)); *k=j; free(k);
  return(0);
}
```



Relações entre Processos

- Os processos são organizados numa hierarquia em árvore:
  - Pai
  - Filho
- O processo **init** é o primeiro processo em execução (geralmente criado pelo kernel, após o arranque do sistema operativo)
- A partir deste processo, são criados processos clonados que
  - herdam o mesmo programa do pai, ou
  - substituem o programa herdado

Árvore de Processos



O '**getty**' é responsável por exibir um prompt de login no terminal, onde o utilizador insere o seu nome de utilizador. Após o utilizador pressionar Enter, o 'getty' passa o controle para o processo de '**login**', que é responsável por autenticar o utilizador e solicitar a palavra-passe. Após um login bem-sucedido, o interpretador de comandos '**sh**' é lançado para permitir a interação com o sistema

Bibliotecas <unistd.h> e <sys/types.h>

- **<unistd.h>**: Biblioteca para interação com o SO
- Inclui funções como **fork()**, **getpid()**, **getppid()**, entre outras, para controle de processos e manipulação de descritores de ficheiros
- **<sys/types.h>**: Biblioteca relacionada aos tipos de dados usados em chamadas do sistema e manipulação de ficheiros
- Introduz tipos como o **pid_t** (tipo de dados correspondente a int) que especifica identificadores de processo

Identificação de Processos

- Cada processo é identificado por um número inteiro único (Process ID ou PID)
- Para identificação de processos usamos:

  #include <unistd.h>

  pid_t **getpid**(void); pid_t **getppid**(void);
- **getpid** retorna o PID do processo invocador (o PID próprio)
- **getppid** retorna PID do processo pai do processo invocador
- Exemplo: Programa 1.2 (p. 10)

Programa 1.2 (p. 10)

```
#include <stdio.h>
#include <unistd.h>

int main() {
 printf("my PID: %d\n", getpid());
 printf("my parent's PID: %d\n", getppid());
 return(0);
}
```

Criação de Processos

- A única forma de criar um novo processo (com excepção do init), designado de filho, é através da criação de uma cópia de um outro, designado de pai
- Os processos filho podem executar o programa herdado do pai ou, sendo conveniente, substituir esse programa por outro(s)
- O filho herda uma cópia do Segmento de Dados do pai (ou seja, **as variáveis não são partilhadas**)
- Como o Segmento de Código é apenas de leitura, o pai e o filho partilham-na, sem que seja feita uma cópia
- Um processo filho também herda do seu pai certos atributos de Nível Supervisor: diretoria de trabalho, cópia dos descritores dos ficheiros abertos pelo pai antes da criação do filho, etc.

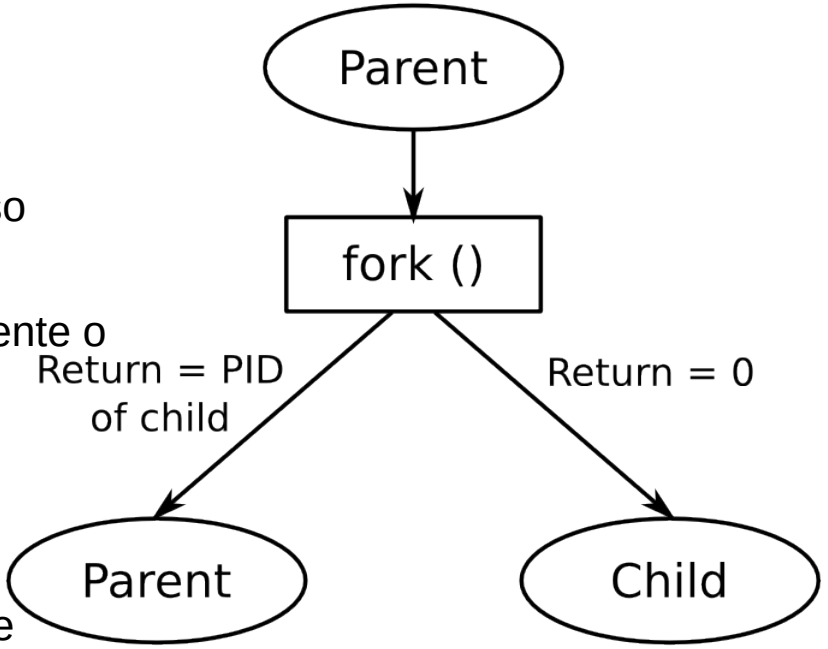
Primitiva fork

- #include <unistd.h>

  pid_t fork(void);
- man 2 fork
- É usada para criar uma cópia (o filho) do processo invocador (o pai)
- Após uma invocação bem sucedida a execução do programa prossegue na instrução a seguir a fork, mas em duplicado, de forma concorrente, pois passam a existir duas instâncias do mesmo programa, mas alojadas em processos diferentes (pai e filho)

Retorno na Chamada fork()

- Quando a chamada fork() é realizada, ambos o processo pai e o processo filho recebem um valor de retorno
- O processo pai:
  - Recebe o PID (Identificador de Processo) do processo filho recém-criado
  - O PID é um número inteiro que identifica exclusivamente o processo filho
- No Processo Filho:
  - Recebe o valor de retorno 0 (zero)
  - Isso permite ao processo filho identificar que ele é, de fato, o processo filho
- Exemplo: Programa 1.3 (p. 12)



Programa 1.3 (p. 12)

```
#include <stdio.h>
#include <unistd.h>

int main()
{
 pid_t ret=-1;

 fork(); // (1)
 // ret=fork();  // (2)
 printf("ret: %d\n", ret);
 return(0);
}
```

Primitiva exit

- #include <stdlib.h>

  void exit(int status);
- man 3 exit
- Quando um processo termina, deve invocar a primitiva exit com um **parâmetro inteiro**
- O código de saída reflete as condições em que o processo terminou
- O código de saída fará parte do estado de terminação podendo ser inspecionado pelo processo pai
- Convenção: status = 0 (sucesso), status = 1 … 255 (insucesso)
- O valor inteiro poderá ser o da variável global **errno** (#include <errno.h>)
- Exemplo: Programa 1.5 (p. 15)

Programa 1.5 (p. 15)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
 pid_t ret;

 ret=fork();
 if (ret<0) { perror("fork"); exit(errno); }
 if (ret==0) {
     printf("I am the CHILD %d of the PARENT %d\n", getpid(), getppid());
     exit(0);
 }
 printf("I am the PARENT %d of the CHILD %d\n", getpid(), ret);
 return(0);
}
```

Exercício 1 da Secção 1.12 (p. 27)

- Desenvolva e execute os seguintes programas, monitorizando a utilização da CPU no **htop** (instale com **sudo apt install -y htop** se ainda não estiver disponível):
  - a) programa onde um processo pai e um processo filho executem ambos um ciclo infinito, apresentando-se, por cada iteração do ciclo, o PID do processo em execução, e o valor de um contador inteiro que é incrementado de uma unidade por cada iteração
  - b) variante de a) em que, por cada iteração, também se invoca sleep(1), de forma a visualizar, pausadamente, o output dos processos
  - c) variante de a) em que o corpo do ciclo é vazio (ou seja, os processos são CPU-bound)
  - d) variante de c), com apenas um processo (neste cenário é mais fácil observar a migração do processo entre diferentes CPUs/núcleos; se necessário, gere carga adicional para favorecer a migração: **`sudo apt install -y stress-ng fio iperf3 sysstat; taskset -c $((RANDOM % 2)) stress-ng --cpu 1 --timeout 15s`**)