Trabalho Prático 1

Data Publicação: 12-03-2023 **Data Conclusão: 03-04-2023**

Entrega: A entrega da resolução é realizada na página da disciplina no Moodle juntando, num único ficheiro compactado, o **código fonte**, o **Makefile**. Existirão aulas práticas para a realização parcial deste trabalho com entrega na respetiva aula.

Objetivo: Familiarização com as ferramentas de desenvolvimento em ambiente UNIX/LINUX C (`gcc`, `make`, `gdb`) e de um ambiente integrado (vscode); Conceção de programas com a criação de processos (`fork`, `getpid`, `getppid`, `wait`, `waitpid`, `sleep`), execução de programas (família de funções `exec`), mecanismo de comunicação *pipe* (`pipe`) e redireccionamento de I/O (dup, dup2); Consolidação da programação ao nível de sistema.

Livro: A resolução deste trabalho pressupõe a leitura dos capítulos 1 a 6 do livro R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, August, 2018.

I. Realize os seguintes exercícios

Durante a realização dos exercícios propostos utilize, no terminal, o comando `man` de forma a esclarecer dúvidas sobre as chamadas de sistema e as funções C, como por exemplo, quais os seus argumentos, os valores de retorno e como verificar as situações de erro.

Para a resolução de cada questão comece por criar uma pasta contendo os ficheiros C com a resolução do exercício e um ficheiro `Makefile` (Incluindo, entre outras, as regras **`all`**, e **`clean`**) que permita a compilação da solução.

Na sua resolução considere o tratamento de erros das chamadas de sistema utilizadas.

1. Realize um programa `cpu_stress` com o objetivo de simular uma carga de processamento no(s) CPU(s) disponíveis no sistema através da criação de múltiplos processos.

   Este programa recebe pelos argumentos da linha de comando o número de processos auxiliares (processos filho) a serem criados. Cada um dos processos auxiliares executa a função `process_work`. Na Figura 1 ilustra-se a utilização desta função num programa sequencial.

```
void process_work (long niter) {
    for (long i = 0; i < niter; i++) {
        sqrt (rand ());
    }
}
int main () {
    process_work(1e9);
    return 0;
}
```

Figura 1

   O processo pai só deve terminar depois de garantir que todos os processos filho terminaram a sua execução.

   Utilize os programas `htop`, `top` e `ps` num terminal para acompanhar a execução do seu programa

2. Considere o programa seguinte.

```
int main ()
{
  printf("MSG 1\n");

  for (int i = 0; i < 3; ++i) {
    if (fork() == 0) {
      printf("MSG 2\n");
    }
  }

  printf("MSG 3\n");

  return 0;
}
```

A execução deste código irá originar além do processo principal mais 6 processos.	
A execução deste código irá originar além do processo principal mais 4 processos.	
Este código pode originar processos órfãos, mas não processos zombies (*defuncts*).	
Este código pode originar processos órfãos e processos zombies (*defuncts*).	

3. Realize um programa (`count_words`) que recebe através da linha de argumentos o nome de um ficheiro e imprima no standard de output o número de palavras desse ficheiro. A determinação do número de palavras é realizada executando o programa externo `wc` com a opção `-w`.

4. Considere o programa seguinte.

```
int main ()
{
  for (int i = 0; i < 10; ++i) {

    execlp("date", "date", NULL);

    sleep(1);
  }

  return 0;
}
```

Este programa executa o programa `date`, para imprimir a data e hora no *standard de output*, em intervalos de 1 segundo durante 10 segundos.	
São criados 10 processos auxiliares para executarem o programa `date` ficando esses processos no estado zombie.	
Este código apenas executa o programa `date` 1 única vez.	
Este código deveria esperar pelos processos filho através da utilização da chamada de sistema `waitpid()`.	

II. Processamento estatístico de vetores com múltiplos processos

5. Pretende-se uma função que extraia, de um vetor (`array`) de números inteiros, um subvetor com os elementos que encontrem num determinado intervalo de valores. Esta operação será utilizada em vetores com dimensões elevadas, por exemplo, da ordem dos 256M elementos.

   A função deve respeitar a seguinte assinatura:

   `int vector_get_in_range (int v[], int v_sz, int sv[], int min, int max, int n_processes);`

   A função recebe o vetor de valores a analisar no primeiro parâmetro (`v`) seguido da sua dimensão (`v_sz`), o terceiro parâmetro é indicado o vetor onde será guardado o subvetor e o número de processos (`n_processes`) a utilizar na operação. A função devolve o número de elementos colocados no subvetor sv ou um valor negativo em caso de erro.

   A estratégia a adotar é a seguinte:

   - O processo principal vai criar um determinado número de processos filhos auxiliares, por exemplo, podemos dividir o processamento do vetor em quatro partes o que implica a necessidade de quatro processos auxiliares.
   - O processo principal envia aos processos filho o subvetor que lhe são destinadas. Neste caso, deve tirar partido do facto da chamada de sistema `fork()` criar um processo através de clonagem do processo pai incluindo todo o seu espaço de endereçamento.
   - Os processos filhos, no fim da computação, enviam os resultados ao processo principal para que este construa o vetor resultado (`sv`).
   - Os processos filhos utilizam `pipes` para enviarem ao processo pai o resultado calculado (ver Figura 2).

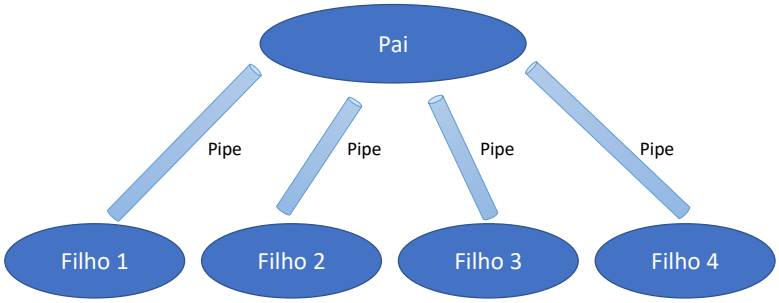

Figura 2 - Arquitetura geral da solução baseada em múltiplos processos

Sugere-se que o programa a realizar (`vector_stat_proc`) receba por argumentos o número de elementos do vetor seguindo-se o número de processos a usar no processamento, i.e.:

```
$> ./vector_stat_proc <vector dimension> <number processes>
```

Neste exercício será valorizado a generalidade da solução, a justificação das opções tomadas e a qualidade do código, bem como os testes realizados para comprovar a correção da solução apresentada. Teste a sua solução usando vetores com dimensões superiores a 256 mega valores.

```
$> ./vector_stat_proc 256000000 2
```

Como apoio é disponibilizado o projeto `vector_stat_seq.zip` que contém o código sequencial do pretendido. Este código deve ser estendido de forma a responder aos requisitos do enunciado.

Avalie os tempos de execução, entre a versão sequencial e a versão baseada em múltiplos processos, variando a dimensão das matrizes e o número de processos. Numa primeira fase o tempo de execução pode ser obtido, através do comando time:

```
time ./vector-stat-seq
```

Para obter o tempo de execução de um troço de código, por exemplo a multiplicação da matriz, pode utilizar o código apresentado na Figura 3.

```
#include <sys/time.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    struct timeval t1,t2;
    gettimeofday(&t1, NULL);

    // code to evaluate

    gettimeofday(&t2, NULL);
    long elapsed = ((long)t2.tv_sec - t1.tv_sec) * 1000000L + (t2.tv_usec - t1.tv_usec);
    printf ("Elapsed time = %6li us\n", elapsed);
    return 0;
}
```

Figura 3

Notas

- **As funções deste trabalho devem ser desenvolvidas <u>recorrendo às chamadas de sistema estudadas: open, read, write, close, fork, wait, exec, pipe e dup</u>, não usando as funções system, popen, etc.**

- **A solução de cada exercício tem de conter, obrigatoriamente, um Makefile** com as regras para a geração dos executáveis e uma regra para a eliminação dos ficheiros desnecessários da pasta do projeto.
- Esta proposta de trabalho deixa em aberto algumas questões que constituem opções a serem tomadas pelos alunos e que serão alvo de apreciação na avaliação do trabalho.
- Os testes realizados na verificação da correta implementação do trabalho fazem parte da avaliação.
- Valorize o seu trabalho tornando o código robusto através da validação das operações, deteção de erros, fecho dos descritores de ficheiros abertos, eliminação de processos *zombie*, etc.
- Nesta fase não se pretende um relatório formal, mas apenas uma pequena descrição da arquitetura das soluções e das opções realizadas.
- O último exercício será utilizado nos trabalhos seguintes. Assim, deve realizar a sua solução de forma modular criando uma camada de abstração que facilite a sua utilização e a integração nos trabalhos seguintes.

Bibliografia de suporte

1) Bibliografia de suporte disponível na página comum do Moodle:
   a) Slides utilizados nas aulas.
   b) Exemplos fornecidos.
   c) Exemplos realizados nas aulas.
2) R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, august, 2018 [Ch 1 - 6]. Available: http://pages.cs.wisc.edu/~remzi/OSTEP/. [Accessed: 06-04-2022].

Bom trabalho,
Diogo Cardoso, *Nuno Oliveira*