Lista1 – Processos

1. Descreva a funcionalidade das 5 principais estruturas de dados utilizadas na implementação dos sistemas operacionais: listas, pilhas, filas, função hash e bitmap
   - Escolha uma opção entre lista, pilha ou fila e escrever um exemplo de implementação em código. A escolha da linguagem é livre (e.g., Go, Rush, C/C++, Java, Python)

2. O que é uma API, qual é a sua utilidade dentro do sistema operacional e que relação que guarda com as chamadas de sistema?

3. Qual é o nome da API que utilizam os sistemas Windows, Unix (e.g. Solaris, Linux, MacOS X) e Java Virtual Machine (JVM) e quais são as suas principais características?

4. Descreva 5 chamadas de sistema utilizadas nos sistemas Unix detalhando sintaxe e operação que executam. Cada função deverá pertencer a uma categoria diferente entre Process control, File management, Device management, Information Maintenance, Communication and Protection.
   Sugestão: Leia a "man page" da função Unix e o livro-texto Seção 2.3.3 Types of System Calls
   Exemplo: $ man fork (fornece informações da função "fork")

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

5. O que é uma máquina virtual? Cite exemplos de uso na atualidade
6.
7. Descreva o princípio de funcionamento do programa bootstrap e da memória cache

8. Escreva o programa `arvore-processos.c` que execute o programa mostrado na Tabela 1. O programa cria a estrutura de árvore de processos `P_A -> P_B -> P_C`, onde o processo `P_A` é pai de `P_B`, quem por sua vez é pai de `P_C`. A solução deverá

incluir um diagrama de processos e chamadas de sistema similar ao da Fig. 3.9 do livro-texto:

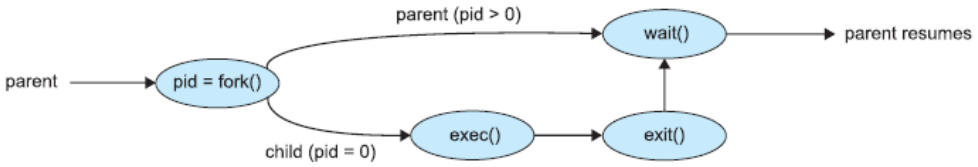

Figure 3.9 Process creation using the `fork()` system call.

N.B.: `PID` interno é o identificador que recebe o processo pai dentro do seu corpo de programa após a criação de um processo filho. `PID` é o identificador de valor único e irrepetível que o SO outorga a cada processo.

Sugestões:

- Revise o código da `Figure 3.8` do livro-texto
- Revise o manual `man` das seguintes chamadas de sistema:

`fork`	Cria um processo filho. Retorna o valor `PID` no final da execução
`wait`	Permite aguardar pela terminação de um processo
`execlp`	Executa um comando no sistema
`getpid`	Retorna o `PID` do processo corrente
`getppid`	Retorna o `PID` do pai do processo corrente
`printf`	Imprime texto formatado no terminal

Tabela 1 - Programa para criação da árvore de processos `P_A->P_B->P_C`

Linha	**`gcc -o q1.out q1.c; ./q1.out`** `# Compile & Run`
1	`Sou P_A com PID 1028, filho de PID 8`
2	`Eu P_A criei P_B!`
3	`Sou P_B com PID 1029, PID interno 0, filho do PID 1028`
4	`Eu P_B criei P_C!`
5	`Sou P_C com PID 1030, PID interno 0, filho do PID 1029`
6	`Eu P_C executei: ps`

7	`  PID TTY          TIME CMD`
	`    8 tty1     00:00:01 bash`
	`1028 tty1     00:00:00 q1.out`
	`1029 tty1     00:00:00 q1.out`
	`1030 tty1     00:00:00 ps`

8	
9	`Eu P_B aguardei P_C terminar!`
10	`Eu P_B executei: ps`

11	`  PID TTY          TIME CMD`
	`    8 tty1     00:00:01 bash`
	`1028 tty1     00:00:00 q1.out`
	`1029 tty1     00:00:00 ps`

12	
13	`Eu P_A aguardei P_B terminar!`
14	`Eu P_A executei: ps`

15	PID TTY TIME CMD 8 tty1 00:00:01 bash 1028 tty1 00:00:00 ps

9. Crie um programa que simule a árvore de processos da Figura 3.7

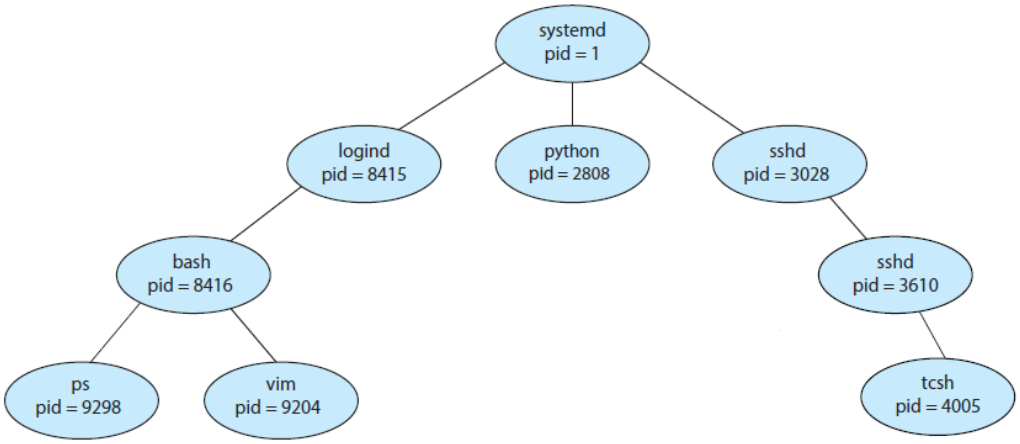

Figure 3.7 A tree of processes on a typical Linux system.

10. Escreva o programa `zombie.c` que execute o programa mostrado na Tabela 2. O programa cria um filho e deixa ele como zombie, isto é, não aguarda o filho terminar. Após um tempo de `X` segundos o pai executa o comando `ps` e termina. A saída no terminal apresenta o processo `zombie.x` no estado `<defunct>` indicando que `zombie.x` com `PID 1192`, filho de `PID 1191` virou órfão e, após ser reconhecido pelo SO, virou um "órfão defunto", isto é, foi terminado pelo SO.
    Crie um diagrama de estados do processo e explique todos os estados pelos quais passa o processo filho, desde a sua criação até a sua terminação.
    Sugestões:
    - Utilizar o código `fig3-8.c` como referência
    - Estudar o conteúdo das aulas sobre processos no `gdrive`
    - Revisar o manual `man` das seguintes chamadas de sistema:

    `sleep(X)` Tempo de espera de X segundos

Tabela 2 - Processo zombie

Linha	**$ gcc -o zombie.x zombie.c; ./zombie.x** # Compile & Run
1	PID TTY TIME CMD 8 tty1 00:00:02 bash 1191 tty1 00:00:00 ps 1192 tty1 00:00:00 zombie.x <defunct>

11. Escreva o programa `userprog.c` que permita o usuário entrar com um comando e seus parâmetros, e execute o comando. Faça uso das chamadas `fork` e `exec`.

12. Escreva o programa `ordena-array.c` que possui uma variável do tipo array contendo 10 números desordenados. Esse processo `main` deve criar um processo filho usando a função `fork`. Em seguida o `main` deve ordenar o array utilizando a função de ordenamento `bubbleSort`, enquanto o filho deve utilizar `quickSort`.
    Sugestões:
    - Utilize a biblioteca `sortlib.c`

13. Crie um diagrama da árvore de processos, mostrando os processos e seus filhos, gerada com a execução do código `fork4.c` e discuta o resultado. Adicionalmente, anexe uma imagem da árvore de processos criada utilizando o programa `pstree` e compare ambos resultados.

14. Compare e contraste duas técnicas de IPC: Memória compartilhada e passagem de mensagens. Dê um exemplo cada de uma situação onde uma seria mais apropriada do que a outra. Cite a função POSIX de cada técnica incluindo uma descrição dos seus parâmetros.

15. Defina o que é pipe na comunicação entre processos. O que ocorre quando a comunicação entre dois processos (processos A e B) querem conversar usando pipe?