Relatório EP1 - ACH2044 Sistemas Operacionais - Turma 94

Felipe Furquim - 11208030
Stefany Ramos - 10843552

Exercício 1)

Esse exercício foi feito de duas maneiras diferentes:

A. Simplesmente usando o fork() em C
B. Usando o fork(), porém chamando funções diferentes para o processo e o processo filho em C

A) A chamada de sistema fork() é usada para criar um novo processo, chamado de processo filho, que é executado simultaneamente com o processo que faz a chamada fork() (processo pai). Depois que um novo processo filho é criado, ambos os processos executarão a próxima instrução após a chamada de sistema fork(). Um processo filho usa o mesmo PC (program counter), mesmos registradores de CPU e os mesmos arquivos abertos que usam no processo pai.

Então basicamente ao usar o fork(), o programa irá criar um novo processo filho que executará tudo que vem depois do fork(). Nesse caso, o processo pai e o processo filho executarão exatamente a mesma coisa: Imprimir “hello word”:

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


int main() {

  fork();
  printf("hello world\n");

  return 0;
}
```

Para compilar esse código, foi inserido as seguintes linhas de comando em uma distribuição Linux:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ gcc exe1A.c -o 1Aexec -lpthread
```

Executando:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ ./1Aexec
hello world
hello world
```

Então note que, mesmo com somente um comando para imprimir “hello word”, o programa imprime duas vezes. Isso ocorre por conta do fork(), cada processo imprime “hello word” uma vez.

B) Fizemos o processo pai e o processo filho imprimir “hello word”. Porém eles estavam executando exatamente a mesma coisa. E se quiséssemos executar coisas diferentes?
Para isso, temos que ver o fork() um pouco mais a fundo. O fork() retorna um inteiro e se esse inteiro for

Um valor negativo - A criação do processo filho não foi bem sucedida;
Zero - Valor retornado ao processo filho recém criado;
Um valor positivo - Valor retornado para o processo pai. Esse valor é o ID do processo filho recém criado.

Sabendo disso, basta armazenarmos esses valores e verificá-los ao chamar a função do processo pai e do processo filho:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void parentProcess(){
  printf("hello world\n");
}

void childProcess(){
  printf("hello world\n");
}

int main() {

    int p = fork();

    if(p == 0){
      childProcess();
      kill(p,SIGKILL);
    }

    else if(p > 0)
      parentProcess();

    return 0;
}
```

Análise do código:
O valor de retorno do fork(0 foi armazenado na variável p.
Caso p seja 0, executa a função do processo filho e encerra os dois processos. Caso p seja maior que 0, a função do processo pai é executada.
As duas funções fazem a mesma coisa (imprime “hello word”) pois foi pedido assim no enunciado. Depois será mostrado essas funções fazendo coisas diferentes de forma mais evidente.

Para compilar:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ gcc exe1B.c -o 1Bexec -lpthread
```

Executando:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ ./1Bexec
hello world
hello world
```

Agora as funções parentProcess() e childProcess() serão alteradas da seguinte maneira:

```
void parentProcess(){
  printf("hello parent\n");
}

void childProcess(){
  printf("hello child\n");
}
```

Compilando e executando:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ ./1Bexec
hello parent
hello child
```

Agora sim dá para notar bem as tarefas diferentes que cada processo está fazendo.

Exercício 2)

Para esse exercício, o código foi compilado e executado no Windows 10.
Para criar threads em Java, é necessário usar as classes Runnable e Thread. No código, foi criado uma classe interna anônima, sobrescrevendo o comportamento do método run(). Foi utilizado uma classe interna anônima pois não será usada em outra classe além da classe principal e simplifica o código. Depois, basta executar uma nova Thread com a instância da classe Runnable passada como parâmetro, chamando o método start():

```java
public class Exe2 {
        public static void main(String[] args) {
                for(int i = 0; i < 5; i++)
                        new Thread(threadHello).start();
        }

        private static Runnable threadHello = new Runnable(){
                public void run(){
                        System.out.println("hello world");
                }
        };
}
```

Compilando e executando:

```
C:\Users\fvfurq\Desktop>javac Exe2.java && java Exe2
hello world
hello world
hello world
hello world
hello world
```

Pode-se notar que foi criado 5 threads imprimindo “hello word”.

Exercício 3)

Para criar threads em C, é necessário incluir pthread.h. Então, foi preciso criar um array do tipo pthread_t (não precisa necessariamente ser um array, só preferimos fazer dessa forma). O endereço de cada elemento desse array será passado como variável do método pthread_creat(), além do método print() e uma string com casting para ponteiro de void. O método print() será o método executado nas threads, que recebe um ponteiro para void contendo uma sequência de caracteres. Essa sequência representa a mensagem que deve ser impressa. É possível analisar o código em mais detalhes aqui:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* print(void *ptr);

int main(){

  pthread_t thread[5];

  char* hello = "hello word";

  int i;
  for(i = 0; i < 5; i++)
    pthread_create( &thread[i], NULL, print, (void*) hello);

  for(i = 0; i < 5; i++)
    pthread_join(thread[i], NULL);

  exit(0);
  return 0;
}

void* print(void* msg){

  printf("%s\n", msg);

  return msg;
}
```

Para compilar esse código no Linux, basta digitar o seguinte comando no terminal Linux:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ gcc -pthread exe3.c -o 3exec -lpthread
```

Executando:

```
istari@DESKTOP-UCHIV5N:~/Desktop$ ./3exec
hello word
hello word
hello word
hello word
hello word
```

Por fim, pode-se notar que cada thread imprimiu “hello word”. Daria para fazer com que cada thread fizesse uma coisa diferente alterando a função passada como parâmetro do pthread_create() para cada posição do array de pthread_t.