Threads e Processos

AMANDA LOPES
LÍVIA HIPÓLITO
ADRYELE OLIVEIRA
FILIPE CORRÊA
FELIPE ALVES

1 – Escreva um programa com 1 avô, 1 pai e 1 filho

```c
etapa 1
//Escreva um programa com 1 Avo, 1 Pai e 1 Filho

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

int main() {
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        // duplicacao de pid1
        pid2 = fork();
        if (pid2 == 0) {
            // processo avo
            printf("Eu sou o processo avo com PID %d\n", getpid());
        } else {
            // processo filho
            printf("Eu sou o processo filho com PID %d\n", getpid());
        }
    } else {
        // processo pai
        printf("Eu sou o processo pai com PID %d\n", getpid());
    }
    return 0;
}

```

- Utilizando a função "**Fork**" para duplicar o processo.

- Função "**Printf**" para mostrar os pids do processo pai, filho e avô.

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/8$ ./oito
Eu sou o processo avo com PID 159221
Eu sou o processo pai com PID 159222
Eu sou o processo filho com PID 159223
```

2 – Escreva um programa com 1 pai e 2 filhos

```c
int main() {
    int pid1, pid2;
//pid1 e pid2 sao identificadores de variaveis
    pid1 = fork();
    //duplicacao de processo
    if (pid1 == 0)
    //verificandp se o processo atual e o processo filho
    {
        printf("Filho 1: Meu PID é %d. O PID do meu pai é %d.\n", getpid(), getppid());
        exit(0);
//getpid retorna o identificador do processo atual
//getpidd retorna o identificador do pai do processo atual
    }
    pid2 = fork();
    if (pid2 == 0) {
        printf("Filho 2: Meu PID é %d. O PID do meu pai é %d.\n", getpid(), getppid());
        exit(0);
    //filho2 retorna seu pid e o pid de seu pai
    }

    printf("Pai: Meu PID é %d. Os PIDs dos meus filhos são %d e %d.\n", getpid(), pid1, pid2);
    //pid pai ira printar na tela seu pid e o pid de seus dois filhos
    return 0;
}
```

- Utilização das funções "**getpid**" e "**getppid**" que retornam o número pid do processo.

- Observa-se que ambos os filhos obtêm o mesmo número pid do processo pai, provando que eles foram gerados do mesmo processo pai.

Resultado:

```
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$ gcc atividade2.c -o 2
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$ ./2
Pai: Meu PID é 19793. Os PIDs dos meus filhos são 19794 e 19795.
Filho 1: Meu PID é 19794. O PID do meu pai é 19793.
Filho 2: Meu PID é 19795. O PID do meu pai é 19793.
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$
```

3 – Eliminação do processo pai

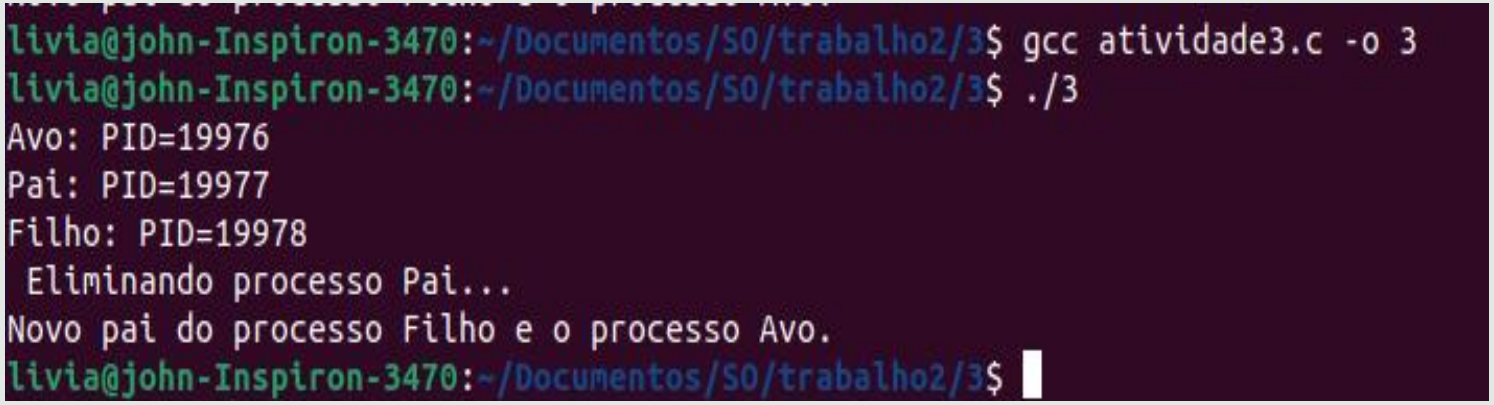
```c
int main() {
    pid_t pid, pid2;

    pid = fork();
    //pid pai criando o filho

    if (pid == 0) {
        printf("Pai: PID=%d\n", getpid());
        exit(0);
        // se o pid for igual a 0, entao ele e o processo filho e mostrara seu pid
    } else if (pid > 0) {
        // se o pid for maior que 0, entao e o processo pai
        pid2 = fork();
        //processo pai se tornando avo
        if (pid2 == 0)
          {
            printf("Filho: PID=%d\n", getpid());
            exit(0);

        } else if (pid2 > 0) {
            printf("Avo: PID=%d\n ", getpid());
            sleep(10);
            //apos a duplicacao, encontra-se o processo pai e avo
            //ha uma espera para a finalizacao do processo
```

```c
            printf("Eliminando processo Pai...\n");
            kill(pid2, SIGKILL);
            sleep(10);
            //processo pai e eliminado pela funcao kill
            //ha uma espera de 10s para que o pai seja morto
            //entao, encontra-se o novo pai do processo filho
            printf("Novo pai do processo Filho e o processo Avo.\n");
            //Quando um processo morre, seus filhos sao
            //adotados pelo processo init, que tem PID = 1

        } else {
            perror("fork");
            exit(1);
        }
    } else {
        perror("fork");
        exit(1);
    }

    return 0;
}
```

- O processo pai irá ser eliminado através da função "**kill**" da biblioteca de sistema **signal.h.**

- Na tela irá imprimir que o novo pai do processo filho será o processo avô.

3 – Eliminação do processo pai – Resultado



4 – Threads ABC

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


#define NUM_THREADS 3
volatile int turn = 0;


void* imprime_A(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 0){ //loop q continua quando turn for igual a 0. Faz a thread esperar sua vez para printar A
        printf("A"); // imprime A na tela
        turn = 1; // muda o valor, passa para proxima thread
        }

    }
}

void* imprime_B(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 1){ //loop q continua quando turn for igual a 1. Faz a thread esperar sua vez para printar B
        printf("B"); // imprime B na tela
        turn = 2; // muda o valor, passa para proxima thread
        }

    }
}

void* imprime_C(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 2){ //loop q continua quando turn for igual a 2. Faz a thread esperar sua vez para printar C
        printf("C\n"); // imprime C na tela
        turn = 0; // muda o valor, passa para proxima thread
        }

    }
}
```

- Nas funções das threads, a variável turn muda e passa para a próxima thread.
- No main, as threads são criadas e é designado suas funções

```c
int main() {
    pthread_t thread[NUM_THREADS]; //cria array, cada elemento é um identificador da thread


    pthread_create (&thread[0], NULL, imprime_A, NULL); //cria thread 1 e define sua função "impreme_A"
    pthread_create (&thread[1], NULL, imprime_B, NULL); //cria thread 2 e define sua função "impreme_B"
    pthread_create (&thread[2], NULL, imprime_C, NULL); //cria thread 3 e define sua função "impreme_C"

    for(int i=0; i< NUM_THREADS; i++){
        pthread_join(thread[i], NULL); //não permite que a função principal termine antes de todas as threads forem executadas
    }

    return 0;
}
```

4 – Threads ABC – Resultado

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/4$ ./quatro
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
```

5 – Filho altera variável que o Pai imprime – Resultado

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/5$ ./cinco
PID Pai: 1
PID do Filho: 5
Valor final do Pai: 1
```

5 – Filho altera variável que o Pai imprime

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

int main() {

    //criando a variavel com o pid do pai
    int pid = 1;


    //pid do pai antes de executar o comando fork
    printf("PID Pai: %d\n", pid);

    //comando fork sendo executado (criacao de um filho)
    int ret = fork();


    //se o numero de fork for igual a 0, é o processo filho
    if (ret == 0) {

        //alterando o valor do pid
        pid = 5;

        //valor do pid do filho antes do comando exit
        printf("PID do Filho: %d\n", pid);

        //comando exit ()
        _exit(0);
    } else if (ret > 0) { //se o numero do fork for maior que 0, é o processo pai

        // Processo pai espera o processo filho finalizar
        waitpid(NULL);

        //valor final do pid do pai
        printf("Valor final do Pai: %d\n", pid);
    }
    return 0;
```

- Pid do Processo Pai é criado e mostrado na tela
- Processo Filho é criado pelo comando fork() e altera a variável pid
- A variável pid que pertence ao Processo Pai não muda, independentemente da alteração feita pelo Filho

6 – Filho ordena o vetor e Pai imprime

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

void sort(int* vet, int n) { //codigo de ordenacao por ordem crescente
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (vet[j] > vet[j+1]) {
                int temp = vet[j];
                vet[j] = vet[j+1];
                vet[j+1] = temp;
            }
        }
    }
}
```

- Função com código de ordenação, ordem crescente
- Criado vetor de 100 posições

```c
int main() {

    //cria um vetor de 100 posicoes
    int vet[100];

    //adiciona valor a cada posicao do vetor
    for (int i = 0; i < 100; i++) {
        vet[i] = rand() % 100;
    }

    //exibe o vetor 100 antes da criacao do Filho
    printf("Valor do Pai antes do Filho ordenar: \n");
    for (int i = 0; i < 100; i++) {
        printf("%d ", vet[i]);
    }
    printf("\n\n");

    //chama comando fork para copiar processo Pai
    int pid = fork();
```

6 – Filho ordena o vetor e Pai imprime

```
38      if (pid == 0) {
39
40          //Processo Filho ordena o vetor 100
41          sort(vet, 100);
42
43          printf("Valor do vetor ordenado pelo Filho depois do fork: \n");
44          for (int i = 0; i < 100; i++) {
45              printf("%d ", vet[i]);
46          }
47          printf("\n\n");
48
49          //finalizacao do processo Filho
50          _exit(0);
```

- Criado Processo Filho, ele usa a função para ordenar o vetor
- Independente do Pai, o Filho altera a copia do vetor que pertence a ele.

```
51      } else if (pid > 0) {
52
53          //processo Pai espera a finalizacao do processo Filho
54          waitpid(NULL);
55
56
57          //processo Pai imprime na tela o vetor 100
58          printf("Valor do vetor do Pai depois do Filho ordenar: \n");
59          for (int i = 0; i < 100; i++) {
60              printf("%d ", vet[i]);
61          }
62          printf("\n");
63      }
64      return 0;
```

6 - Filho ordena o vetor e Pai imprime - Resultado

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/6$ ./seis
Valor do Pai antes do Filho ordenar:
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 2
9 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87
8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39

Valor do vetor ordenado pelo Filho depois do fork:
2 3 5 5 8 11 11 12 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 26 27 27 29 29 29 29 30 32 34 35 35 36 36 37 39 39 40
 42 43 45 46 49 50 51 54 56 56 57 58 59 60 62 62 62 63 64 67 67 67 67 68 68 69 70 70 72 73 73 76 76 77 78 80 81 82 82 83
 84 84 84 86 86 86 87 88 90 91 92 93 93 94 95 96 98 99

Valor do vetor do Pai depois do Filho ordenar:
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 2
9 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87
8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39
```

7 – Threads X Processos

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

#define LIMITE_PROCESSO 100000000
#define LIMITE_THREAD 100000000

void* funcao_thread(void* arg){
    pthread_exit(NULL);
}
```

- Definido um limite de criação dos processos e threads
- Criado uma função para thread
- Em um loop é criado repetidamente os processos

```c
int main()
{

    //Processos

    printf("Teste do limite de criacao dos Processos: \n");

    int i, pid;

    for(i=1; i <= LIMITE_PROCESSO; i++){
        pid = fork();
        if (pid < 0){
            printf("Limte de Processos excedido: %d\n", i - 1);
            break;
        }
        if(pid == 0){
            exit(0);
        }
    }

    if(pid > 0){
        for(int j=0; j < i; j++){
            wait(NULL);
        }
        printf("Teste processos concluido\n");
    }
```

7 – Threads X Processos

- Para as threads é realizado o mesmo processo
- No emulador mostra quantos processos e threads foram criados até dar erro
- Erro por falta de recurso ou limite excedido

```
41      //Threads
42
43
44      printf("\nTeste do limite de criacao de Threads: \n");
45
46      pthread_t thread;
47
48      for(int i = 1; i <= LIMITE_THREAD; i++){
49          if(pthread_create(&thread, NULL, funcao_thread, NULL) != 0){
50              printf("Limite de Threads excedido: %d\n", i+1);
51              break;
52          }
53      }
54
55      if(i == LIMITE_THREAD + 1){
56          printf("Teste threads concluido\n");
57      }
58
59      return 0;
60  }
```

7 - Threads X Processos - Resultado

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/7$ ./sete
Teste do limite de criacao dos Processos:
Limte de Processos excedido: 15281

Teste do limite de criacao de Threads:
Limite de Threads excedido: 2
```

8 - Desempenho de Fork e Threads

```c
#define NUM_SIMULATIONS 20

void* funcaot(void* threadid) { // uma thread vai rodar essa função
    wait(NULL);
}

int main() {
    pid_t pid;
    int status;
    pthread_t thread;
    clock_t start_time, end_time;
    double total_time_fork = 0.0, total_time_thread = 0.0;

    // Fork simulations
    for (int i = 0; i < NUM_SIMULATIONS; i++) {
        start_time = clock();
        pid = fork();
        if (pid == 0) {
            // Child process
            exit(0);
        } else if (pid > 0) {
            // Parent process
            wait(&status);
            end_time = clock();
            total_time_fork += ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
        } else {
            printf("Erro ao criar o processo filho\n");
            exit(1);
        }
    }

    // Thread simulations

    for (int i = 0; i < NUM_SIMULATIONS; i++) {
        start_time = clock();
        pthread_create(&thread, NULL, funcaot, NULL);
        pthread_join(thread, NULL);
        end_time = clock();
        total_time_thread += ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    }



    printf("Tempo médio por Processos: %f por segundos\n", total_time_fork / NUM_SIMULATIONS);
    printf("Tempo médio por Threads: %f por segundos\n", total_time_thread / NUM_SIMULATIONS);

    return 0;
}
```

- Utiliza-se a função "**clock**" que retorna o tempo de processamento
- Estrutura de repetição **for** até obter 20 simulações
- Calcula o tempo médio por segundos das 20 simulações

8 – Desempenho de Fork e Threads – Resultado

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO$ ./t2
Tempo médio por Processos: 0.000412 por segundos
Tempo médio por Threads: 0.000118 por segundos
```