**Nome:** Diogo Miguel Quintino Rogado Cardoso Fonseca

**Nº:** 79858

**Ex. I.**

1. São iniciados 4 processos: primeiro há um fork, onde o processo é dividido em 2, depois na execução desses dois processos, cada um vai para a linha seguinte onde há outro fork, dividindo esses processos em 2 também. No final vão haver 4 processos. Os processos serão o Parent process (p1 > 0, p2 > 0), o primeiro child process (p1 = 0, p2 > 0), o segundo child process (p1 > 0, p2 = 0) e finalmente o child process do primeiro child process (p1 = 0, p2 = 0)
2. O valor do inteiro “a” é alterado pelo parent process e pelo child process independentemente, e embora ele tenha o mesmo memory address, ele tem valores diferentes. Isto dá-se porque ao fazer o fork, a memoria virtual inteira do processo é copiada para o filho, logo a variável “a” embora tenha o mesmo address nos dois processos, refere-se a duas variáveis diferentes.
3. 5 processos são gerados. No primeiro fork são gerados 2 processos, parent e child, desses 2, o parent gera outro na continuação da lógica do &&, sendo este child2 do parent process. Para além disso, o child2, como falha a lógica do &&, entra na parte da lógica do ||, fazendo outro fork, fazendo um child process da child2 (podemos chamar child2child). Por outro lado, indo pelo ramo da primeira child do parent process, ela falha logo a lógica do && e vai para a lógica do ||, fazendo um fork, tendo então outro child process (podemos chamar a este childchild). No total são 5: (embora haja outros parent processes relativos, chamo parent ao processo parent de todos estes processos, o original) parent, child, childchild, child2, child2child.  
   Com isto em mente como logo depois dos forks temos um wait, eles vão esperar por um processo filho. Todos os processos esperam pelo seu filho, exceto o processo parent, visto que tem dois filhos, ele só espera por um deles. Isto pode ser visto no output já que logo depois do wait é chamado o “ps”. Ao analisarmos o output do ps, vemos que há dois processos que não mostram nada, isto seria o child2child e o childchild, pois ambos não têm children. Para além disso conseguimos observar que o processo parent não espera por um dos filhos, pois quando ele chama o ps vê-se um child process “defunct”.
4. Para corrigir o programa, seria necessário o processo parent esperar por todos os seus filhos. Corrigindo isto obtemos este código:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

    pid\_t pid = (fork() && fork() || fork());

    while (wait(&pid) > 0);

    system("ps");

    exit(0);

}

**Ex. II.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

void terminateOnError()

{

    printf("Fork error\n");

    exit(0);

}

void child()

{

    for (int i = 0; i < 5; i++)

        printf("Eu sou o fiho, meu pai é %d\n", getppid());

}

void parent()

{

    for (int i = 0; i < 3; i++)

        printf("Eu sou o pai, minha identificação é %d\n", getpid());

    while(wait(NULL) > 0);

}

int main()

{

    pid\_t p = fork();

    if (p < 0)

        terminateOnError();

    if (p == 0)

        child();

    else

        parent();

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

void terminateOnError()

{

    printf("Fork error\n");

    exit(0);

}

void child()

{

    for (int i = 0; i < 5; i++)

        printf("Eu sou o fiho, meu pai é %d\n", getppid());

}

void make\_children(int *n*)

{

    for (int i = 0; i < *n*; i++)

    {

        pid\_t p = fork();

        if (p < 0)

            terminateOnError();

        if (p == 0)

        {

            child();

            break;

        }

    }

}

void first\_child()

{

    child();

    make\_children(2);

    while(wait(NULL) > 0);

}

void parent()

{

    for (int i = 0; i < 3; i++)

        printf("Eu sou o pai, minha identificação é %d\n", getpid());

    make\_children(3);

    while(wait(NULL) > 0);

}

int main()

{

    pid\_t p = fork();

    if (p < 0)

        terminateOnError();

    if (p == 0)

        first\_child();

    else

        parent();

}

**Ex. III.**

1. Em i) os programas who e ps estão escritos com um & à frente, logo vão ser executados em child processes do terminal, executando em paralelo. Em ii) os 3 programas são executados um após o outro de forma iterativa e síncrona.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

void who()

{

    execlp("who", "who", (void \*)NULL);

}

void ps()

{

    execlp("ps", "ps", (void \*)NULL);

}

void ll()

{

    execlp("ls", "ls", "-l", (void \*)NULL);

}

int main()

{

    pid\_t p0;

    if (p0 = fork() == 0)

        who();

    if (p0 = fork() == 0)

        ps();

    ll();

    exit(0);

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

void who()

{

    execlp("who", "who", (void \*)NULL);

}

void ps()

{

    execlp("ps", "ps", (void \*)NULL);

}

void ll()

{

    execlp("ls", "ls", "-l", (void \*)NULL);

}

int main()

{

    pid\_t p0;

    if (p0 = fork() == 0)

        who();

    wait(NULL);

    if (p0 = fork() == 0)

        ps();

    wait(NULL);

    ll();

    exit(0);

}

**Ex. III.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

    char \*msg = "Hello, it's me";

    int fd[2];

    if (pipe(fd) < 0)

    {

        printf("An error ocurred creating the pipe.\n");

        return 0;

    }

    write(fd[1], msg, strlen(msg) + 1);

    close(fd[1]);

    char msg\_buffer[strlen(msg) + 1];

    read(fd[0], msg\_buffer, strlen(msg) + 1);

    printf("%s\n", msg\_buffer);

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

int child(int *fd*[])

{

    dup2(*fd*[1], 1);

    close(*fd*[0]);

    printf("Hello Hi");

    close(*fd*[1]);

}

int wc(char \**str*)

{

    int result = 0;

    int len = strlen(*str*);

    int wasLastSpace = 1;

    for (int i = 0; i < len; i++)

    {

        if (*str*[i] != ' ')

        {

            if (wasLastSpace)

                result++;

            wasLastSpace = 0;

        }

        else

        {

            wasLastSpace = 1;

        }

    }

    return result;

}

int parent(int *fd*[])

{

    char message[100];

    close(*fd*[1]);

    read(*fd*[0], message, 100 \* sizeof(char));

    printf("words: %d\n", wc(message));

    close(*fd*[0]);

    wait(NULL);

}

int main()

{

    int fd[2];

    if (pipe(fd) < 0)

    {

        printf("An error ocurred creating the pipe.\n");

        exit(0);

    }

    pid\_t p = fork();

    if(p < 0)

    {

        printf("An error ocurred forking.\n");

        exit(0);

    }

    if(p == 0)

        child(fd);

    else

        parent(fd);

    exit(0);

}

**Referências:**

<https://www.geeksforgeeks.org/fork-system-call/>

<https://www.geeksforgeeks.org/wait-system-call-c/>

<https://www.includehelp.com/c/process-identification-pid_t-data-type.aspx>

<https://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.neutrino_lib_ref/e/execlp.html>

<https://www.geeksforgeeks.org/pipe-system-call/>

<https://www.geeksforgeeks.org/dup-dup2-linux-system-call/>