Universidade Presbiteriana Mackenzie  
Ciência da Computação – 05P11  
Computação Paralela  
20/02/2025

Alan Meniuk Gleizer  
RA 10416804

Atividade – Monitoramento de Processos

**Código Fonte**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int lerInt() {

    // funcao da biblioteca propria para ler um int sem os problemas de scanf

    char buffer[32]; // buffer para input

    int num;

    if (fgets(buffer, sizeof(buffer), stdin) == NULL) {

        return 0; // se fgets é NULL, houve erro de leitura ou EOF

    }

    if (sscanf(buffer, "%d", &num) != 1) {

        return 0; // verificar se entrada realmente é int

    }

    return num;

}

int main() {

    int totalFilhos = 0;

    do {

        printf("Informe a quantidade de filhos: ");

        totalFilhos = lerInt();

    } while (totalFilhos < 1);

    int numFilhos = 0;

    // Criar todos os filhos primeiro

    for (int i = 0; i < totalFilhos; i++) {

        pid\_t pid = fork();

        if (pid < 0) {

            printf("Erro ao criar processo filho");

            return (-1);

        }

        if (pid == 0) {

            // no processo filho

            srand(time(NULL) + getpid());

            int sleep\_time = rand() % 10 + 1;

            sleep(sleep\_time);

            return (sleep\_time);

        }

    }

    // aqui estamos no processo pai, pois todos os filhos deram return acima

    // precisamos de dois loops! no primeiro fazemos os fork. aqui, esperamos todos os filhos terminarem

    for (int i = 0; i < totalFilhos; i++) {

        int status;

        pid\_t pidFilho = wait(&status);

        /\*

        wait() bloqueia execução do pai até que QUALQUER filho termine

        retorno de wait é o PID do filho que terminou.. por isso não precisamos armazenar o PID

        o SO gerencia isso automaGicamente

        \*/

       // apos cada filho terminar, pidFilho tem o pid, e WEXITSTATUS nos da o valor de retorno

        if (WIFEXITED(status)) {

            printf("No processo pai: processo filho: %d, status de saída do filho: %d\n", pidFilho, WEXITSTATUS(status));

        } else {

            printf("No processo pai: processo filho: %d terminou anormalmente.\n", pidFilho);

        }

    }

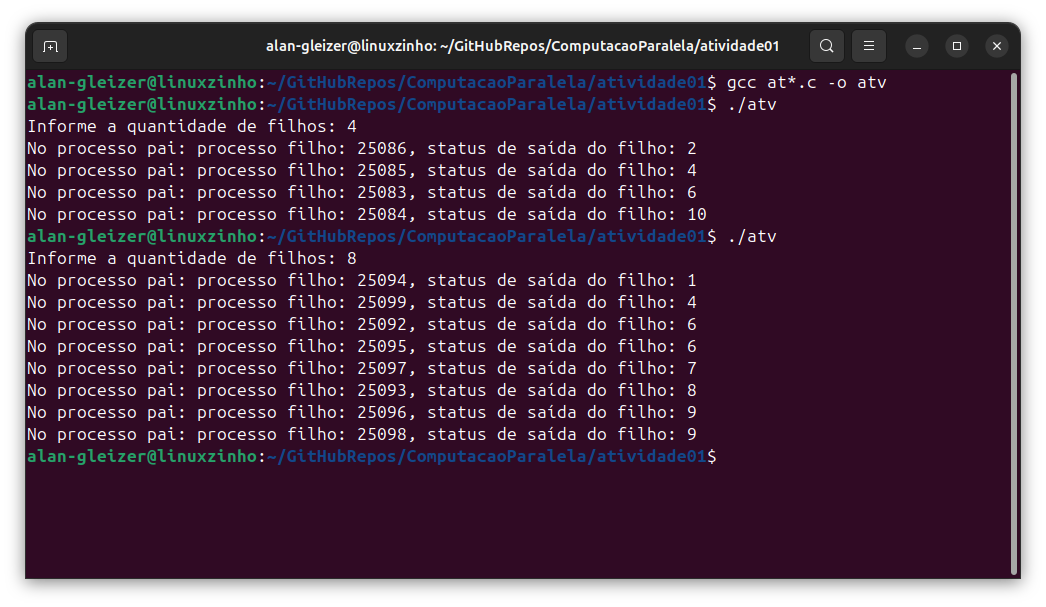
    return 0;

}

**Comentários**

O uso de fork() e wait(), junto com algumas das funções relacionadas como WIFEXITED, é desafiador na medida que o funcionamento dessas funções não é intuitivo em múltiplos sentidos. Em primeiro lugar, o desenvolvimento de um programa com execução de processos paralelos requer uma nova abordagem que evidentemente requer bastante prática. Na primeira tentativa, havia apenas um loop para fork(), sleep() e wait(). Isto é, um único loop no qual os processos filhos eram criados, executados, e no qual o pai esperava a execução terminar para imprimir o retorno do filho. Foi difícil perceber o erro: nesse caso, a execução de todos os filhos era sequencial, sem nenhum tipo de paralelismo. Também é necessário ter em mente quais partes do código serão executadas por pai ou filho, e até que ponto isso deve ocorrer.

Em segundo lugar, o funcionamento de muitas das funções citadas é um tanto obscuro, e requer consultas frequentes à documentação. Nos exemplos estudados em sala de aula, e no código desenvolvido, wait() recebe como parâmetro &status, mas a variável status é declarada e nunca é explicitamente alterada. Pela documentação, ficou claro que o conteúdo de status é alterado e gerenciado pelo kernel. Da mesma forma, o funcionamento de WIFEXITED e funções alternativas requer familiaridade com o SO e as bibliotecas associadas.

**Print de Execução**