МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»**

**(БГТУ им. В. Г. Шухова)**



Кафедра программного обеспечения вычислительной

техники и автоматизированных систем

**Лабораторная работа №1**

по дисциплине: «Операционные системы»

на тему: «Системные вызовы. Базовая работа с процессами в ОС Linux (Ubuntu)»

Выполнил: ст. группы ПВ-223

Дмитриев Андрей Александрович

Проверили:

доц. Островский Алексей Мичеславович,

асс. Четвертухин Виктор Романович

Белгород, 2024

**Цель работы:** Изучить основы работы с системными вызовами и процессами в операционной системе Linux (Ubuntu).

**Условие индивидуального задания:**

2. Изучить основы работы команды ulimit -u, которая ограничивает максимальное количество процессов, запущенных от имени одного пользователя. Добиться ситуации, когда порождено множество зомби-процессов такой мощности, что таблица процессов заполнилась полностью и это мешает созданию новых процессов (контролировать ошибки типа EAGAIN). В протоколе подробно описать поведение Linux (Ubuntu) в такой ситуации. Корректно завершить все зомби процессы. Провести эксперименты в виртуальной машине для разного объёма ОЗУ

**Ход выполнения работы**

**Текст программы** на языке С с комментариями.

#include <stdio.h>    // Подключаем стандартную библиотеку для работы

                      // с функциями ввода и вывода (printf, scanf и т.д.)

#include <stdlib.h>   // Подключаем библиотеку для работы с различными

                      // функциями стандартной библиотеки, например,

                      // для работы с памятью и функцией exit()

#include <unistd.h>   // Подключаем библиотеку для работы с системными

                      // вызовами UNIX, такими как fork(), getpid(), sleep()

#include <time.h>     // Подключаем библиотеку для работы с временем,

                      // используем для генерации случайных чисел и

                      // отображения системного времени

#include <sys/wait.h> // Подключаем библиотеку для работы с процессами,

                      // в частности, для ожидания завершения порожденных

                      // процессов с помощью waitpid()

#include <signal.h>   // Подключаем библиотеку для работы с сигналами

                      // в процессе (pause(), управление сигналами)

#include <errno.h>    // Библиотека для получения доступа к глобальной

                      // переменной последней ошибки

// настройка эксперимента

#define amount\_cicles -1    // количество эксперементов. Если значение меньше 0,

                            // то процессов будет максимальное количество

#define amount\_cubes 4      // количество кубов в пуле

#define amount\_rolls 100000 // количество бросков в эксперементе

// настройка кубика

#define cube\_sides 6                                    // количество граней

#define max\_value\_on\_cube 2                             // максимальное значение на кубике

                                                        // (нужно, чтобы определить максимальную сложность)

const int cube\_values[cube\_sides] = {0, 0, 0, 1, 1, 2}; // значения на кубах

#define max\_value\_on\_cubes max\_value\_on\_cube \*amount\_cubes

void roll\_test();

int main()

{

    int status; // Используется в функциях waitpid() для отслеживания

                // завершения процессов

    // pid главного процесса

    pid\_t core\_proc\_pid = getpid();

    printf("core proc pid: %d\n", core\_proc\_pid);

    // вывод заголовка таблицы

    for (int i = 1; i <= max\_value\_on\_cubes; i++)

        printf("%d\t", i);

    printf("n\tpid\n");

    // главный "обработчик" процессов

    pid\_t fork\_proc\_pid = core\_proc\_pid; // хранит оперируемый pid

    int is\_break = 0;

    int cicles = amount\_cicles;

    int proc\_number\_by\_order = 0;

    while (1)

    {

        switch (fork\_proc\_pid)

        {

        case -1: // error

            if (errno == EAGAIN)

            {

                fprintf(stderr, "try again make fork\n");

                sleep(5);

                break;

            }

            fprintf(stderr, "other errors\n");

            is\_break = 1;

            break;

        case 0: // child do ...

            roll\_test(proc\_number\_by\_order);

            return 0;

        default: // parent do ...

            if (cicles >= 0 && cicles-- == 0)

            {

                is\_break = 1;

                break;

            }

            proc\_number\_by\_order++;

            fork\_proc\_pid = fork();

            break;

        }

        if (is\_break)

            break;

    }

    pause();

    // sleep(15);

    // while (wait(&status) != -1)

    // {

    //     printf("call wait\n");

    //     // sleep(1);

    // }

    return 0;

}

void roll\_test(int proc\_number)

{

    srand(getpid());

    // хранит количества бросков которые преуспели в сложности

    int complexities[max\_value\_on\_cubes];

    for (size\_t i = 0; i < max\_value\_on\_cubes; i++)

        complexities[i] = 0;

    // расчёт успехов и запись в complexities

    int rolls = amount\_rolls;

    while (rolls--)

    {

        // результат за бросок пула кубов

        int cubes = amount\_cubes;

        int sum\_results = 0;

        while (cubes--)

            sum\_results += cube\_values[rand() % cube\_sides];

        // подсчитываем сложности, которые можно пройти

        // с полученным кол-вом успехов

        while (sum\_results--)

            complexities[sum\_results]++;

    }

    // вывод результатов

    for (size\_t i = 0; i < max\_value\_on\_cubes; i++)

        printf("%.2f%%\t", complexities[i] \* 100.0f / amount\_rolls);

    printf("%d\t%d\n", proc\_number, getpid());

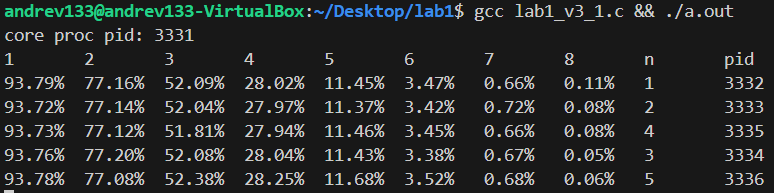
    exit(0);

}

**Протоколы, логи, скриншоты, графики** (обуславливаемые логикой выполнения лабораторной работы и служащие для пояснения принципов авторского решения, демонстрации поведения операционной системы, контроля расхода ресурсов и т.д.)

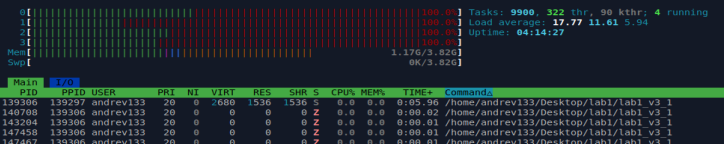
Программа подсчитывает статистику выпадения кубиков на основе множеств экспериментов. Бросок может быть успешным или неуспешным, об этом далее. Куб имеет значения: 0, 0, 0, 1, 1, 2 – бросается несколько кубов (это константа amount\_cubes), формируя пул, результат суммируется. К броску подставляются сложности от 1 до произведения макс. значения на кубе и кол-ва кубов. Если сумма броска больше или равна сложности, значит бросок успешный.

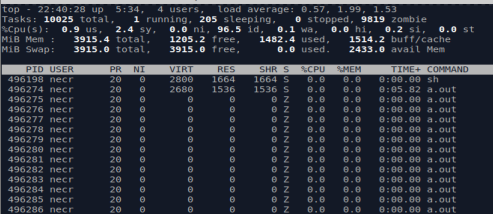
Вывод программы:



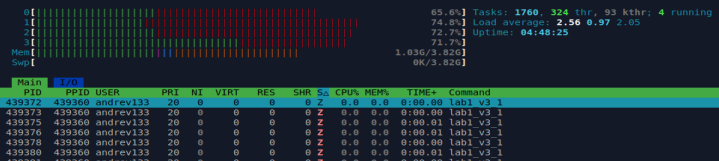
Программа может запускать ограниченное кол-во экспериментов, а может запускать бесконечно.

При ulimit -u: 15384:





При ulimit -u 2000:



Команда ulimit -u ограничивает максимальное количество процессов для текущей сессии shell.

Итак, из-за того, что программа не завершается, всё ещё остаются «зомби» процессы. Они не позволят породить новые процессы, запустить какие-либо программы, но всё ещё будет возможность взаимодействовать с некоторой частью системы.

При уменьшении объёма оперативной памяти уменьшается количество возможных процессов. При 1гб получилось 3839 возможных процессов за место 15384 при 4гб.

**Выводы**

В ходе лабораторной работы было выполнено индивидуальное задание. Были применены программы для отслеживания состояний процессов, как top и htop. Получена практика работы с процессами и функциями для управления ими.