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 C с комментариями.

```c
#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
```

```c
// настройка кубика
#define cube_sides 6                                    // количество граней
#define max_value_on_cube 2                             // максимальное значение
на кубике
                                                        // (нужно, чтобы опреде-
лить 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.)

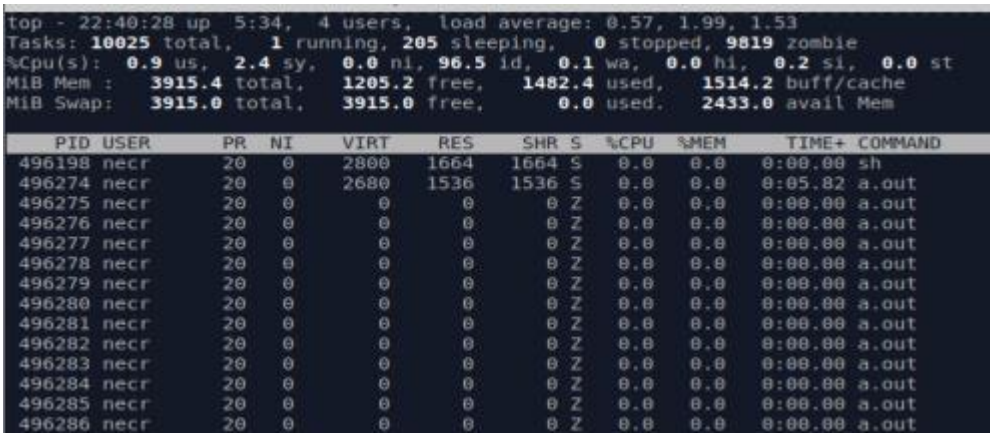
Программа подсчитывает статистику выпадения кубиков на основе множеств экспериментов. Бросок может быть успешным или неуспешным, об этом далее. Куб имеет значения: 0, 0, 0, 1, 1, 2 – бросается несколько кубов (это константа amount_cubes), формируя пул, результат суммируется. К броску подставляются сложности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

Вывод программы:

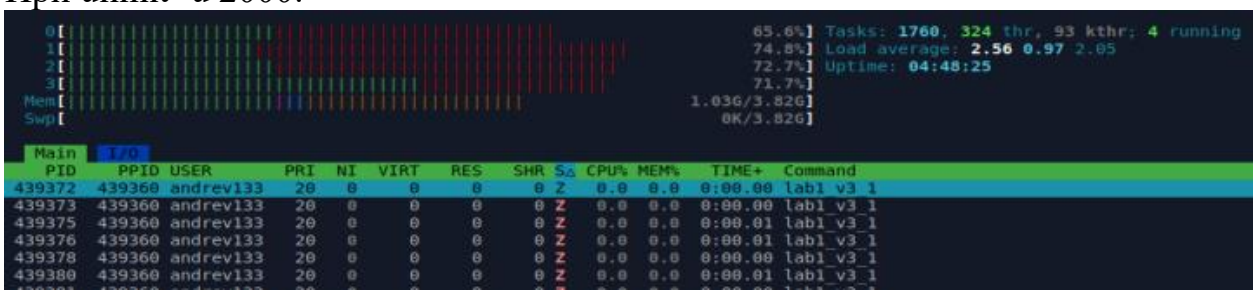
```
andrev133@andrev133-VirtualBox:~/Desktop/lab1$ gcc lab1_v3_1.c && ./a.out
core proc pid: 3331
1       2       3       4       5       6       7       8       n       pid
93.79%  77.16%  52.09%  28.02%  11.45%  3.47%   0.66%   0.11%   1       3332
93.72%  77.14%  52.04%  27.97%  11.37%  3.42%   0.72%   0.08%   2       3333
93.73%  77.12%  51.81%  27.94%  11.46%  3.45%   0.66%   0.08%   4       3335
93.76%  77.20%  52.08%  28.04%  11.43%  3.38%   0.67%   0.05%   3       3334
93.78%  77.08%  52.38%  28.25%  11.68%  3.52%   0.68%   0.06%   5       3336
```

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.