ЛАБОРАТОРНАЯ РАБОТА №1

Тема: Системные вызовы. Базовая работа с процессами в ОС Linux (Ubuntu).

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

Цель работы обуславливает постановку и решение следующих **задач**:

1) Изучить основные концепции системных вызовов.
2) Получить основное понятие о процессе. Изучить базовые системные вызовы POSIX для работы с процессами, такими как: fork(), waitpid().
3) Ознакомиться с механизмом системных вызовов в языке программирования C.
4) Выполнить индивидуальное задание, связанное с использованием POSIX-системных вызовов, для закрепления знаний на практике. Подготовить соответствующую программу на языке C. **Внимание! По возможности следует воздержаться от использования в основной логике программы специальных парадигм взаимодействия между процессами (например, сигналов). Эти парадигмы выходят за рамки настоящей лабораторной работы и ждут еще своего часа.**

Ход выполнения лабораторной работы

1. Для изучения особенностей работы с процессами напишем код на языке C для порождения цепочки из 4-х процессов «родитель-потомок»: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

1.1. Процессы 2 и 3 будут нагружать процессор бесконечным циклом.

1.2. 4-й процесс печатает свой PID и немедленно завершает свою работу.

1.3. 2-й процесс ожидает завершения 3-го процесса. В свою очередь 3-й процесс с вероятностью 1/3 завершает свою работу. Если 3-й процесс не завершает работу в контексте итерации, то 2-й процесс печатает сообщение: «Ну, погоди!».

1.4. После завершения 3-го процесса, 2-й процесс ждет 5 секунд и завершает свою работу.

1.5. После завершения 2-го процесса, 1-й печатает сообщение с текущим системным временем и ждет команды в терминале для завершения через Ctrl+C.

Текст программы на языке C, соответствующий пункту 1.

```
#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,
```

```c
                    // в частности, для ожидания завершения порожденных
                    // процессов с помощью waitpid()
#include <signal.h>   // Подключаем библиотеку для работы с сигналами
                    // в процессе (pause(), управление сигналами)

int main() {

    // PID используется для идентификации процессов в операционной системе
    pid_t process_1_pid_global = -1, process_2_pid_global = -1,
          process_3_pid_global = -1, process_4_pid_global = -1;

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

    // Функция getpid() возвращает идентификатор текущего процесса
    process_1_pid_global = getpid();
    printf("Процесс 1 (PID: %d)\n", process_1_pid_global);

    // Порождаем новый процесс 2 с помощью системного вызова fork()
    // fork() создает копию текущего процесса (процесс 1),
    // которая становится новым процессом (процессом 2)
    // Возвращаемое значение:
    // - В родительском процессе (процесс 1) fork() возвращает PID
    // нового процесса (процесса 2)
    // - В процессе-потомке (процессе 2) fork() возвращает 0

    process_2_pid_global = fork();

    // Проверяем, находимся ли мы в процессе-потомке (процессе 2)
    if (process_2_pid_global == 0) { // Это процесс 2

        // Функция getppid() возвращает PID родительского процесса
        // (в данном сучае процесс 1)
        // Этот идентификатор используется для указания, какой
        // процесс породил текущий процесс (процесс 2)
        // В контексте этого кода, getppid() вернет PID процесса 1
        // для процесса 2

        printf("Процесс 2 (PID: %d) создан процессом 1 (PID: %d)\n",
               getpid(), getppid());

        // Порождаем процесс 3
        process_3_pid_global = fork();
        if (process_3_pid_global == 0) { // Это процесс 3
            printf("Процесс 3 (PID: %d) создан процессом 2 (PID: %d)\n",
                   getpid(), getppid());

            // Порождаем процесс 4
            process_4_pid_global = fork();
            if (process_4_pid_global == 0) { // Это процесс 4
                printf("Процесс 4 (PID: %d) создан процессом 3 (PID: %d). "
                       "Завершает свою работу.\n", getpid(), getppid());
                exit(0);
            } else {
                // Процесс 3 ожидает завершения процесса 4
                // Ожидаем завершения процесса 4 с помощью waitpid()
                // process_4_pid_global — это PID процесса 4
                // waitpid() приостанавливает выполнение текущего
                // процесса (процесс 3),
```

```
            // до тех пор, пока процесс 4 не завершится
            // Аргументы:
            // - process_4_pid_global: PID процесса
            // - &status: указатель на переменную, куда будет
            // записан статус завершения процесса 4
            // - 0: процесс будет приостановлен до завершения
            // дочернего процесса (ожидание завершения)
            waitpid(process_4_pid_global, &status, 0);
            srand(time(NULL) ^ (getpid() << 16));
            while (1) {
                sleep(1); // Ждать 1 сек.
                if (rand() % 3 == 0) {
                    printf("Процесс 3 (PID: %d) завершает работу "
                           "(срабатывание 1/3)\n", getpid());
                    exit(0);
                    // Завершает процесс с кодом 0,
                    // указывая на успешное завершение
                }
            }
        }
    } else {
        while (1) {
            sleep(1);
            // Проверяем, завершился ли процесс 3 с помощью
            // waitpid()
            // process_3_pid_global — это PID процесса 3
            // waitpid() не блокирует выполнение текущего
            // процесса, так как используется флаг WNOHANG
            // Аргументы:
            // - process_3_pid_global: PID процесса
            // - &status: указатель на переменную для получения
            // статуса завершения процесса
            // - WNOHANG: флаг, который указывает, что waitpid()
            // не должен блокировать выполнение,
            //   если процесс 3 ещё не завершился; функция
            // вернет 0, если процесс всё ещё активен

            pid_t result = waitpid(process_3_pid_global, &status, WNOHANG);
            if (result == 0) {
                printf("Процесс 3 (PID: %d) еще работает... "
                       "Ну, погоди!\n", process_3_pid_global);
            } else if (result == process_3_pid_global) {
                printf("Процесс 3 завершил свою работу.\n");
                printf("Процесс 2 ждет 5 сек.\n");
                sleep(5);
                printf("Процесс 2 завершает свою работу.\n");
                exit(0);
            }
        }
    }
} else {
    // Процесс 1 (родительский процесс) ожидает завершения
    // процесса 2
    waitpid(process_2_pid_global, &status, 0);
    // Проверяем, завершился ли процесс нормально с помощью
    // макроса WIFEXITED
    // WIFEXITED(status) возвращает ненулевое значение (true),
    // если процесс завершился
    // нормально (через вызов exit())
```

```
        // Если процесс завершился с ошибкой или был прерван сигналом,
        // WIFEXITED вернет 0 (false)
        if (WIFEXITED(status)) {
            printf("Процесс 2 завершил свою работу.\n");
        }

        printf("Процесс 1 завершает свою работу и ждёт Ctrl+C\n");

        time_t current_time = time(NULL);
        printf("Текущее время: %s", ctime(&current_time));

        while (1) {
            pause();
            // pause() приостанавливает выполнение процесса до
            // получения любого сигнала, который не игнорируется.
            // Процесс возобновит свою работу, как только
            // будет получен сигнал. В данном случае это
            // позволяет процессу 1 "ждать" завершения
            // программы, пока его не прервут вручную (например,
            // с помощью Ctrl+C).
        }

        printf("Процесс 1 завершил свою работу.\n");
    }

    return 0;
}
```

Протокол работы программы в консоли Linux (Ubuntu):

user@user:~/progs$ ./lab1.o
Процесс 1 (PID: 11784)
Процесс 2 (PID: 11785) создан процессом 1 (PID: 11784)
Процесс 3 (PID: 11786) создан процессом 2 (PID: 11785)
Процесс 4 (PID: 11787) создан процессом 3 (PID: 11786). Завершает свою работу.
Процесс 3 (PID: 11786) еще работает...Ну, погоди!
Процесс 3 (PID: 11786) еще работает...Ну, погоди!
Процесс 3 (PID: 11786) завершает работу (срабатывание 1/3)
Процесс 3 завершил свою работу.
Процесс 2 ждет 5 сек.
Процесс 2 завершает свою работу.
Процесс 2 завершил свою работу.
Процесс 1 завершает свою работу и ждёт Ctrl+C
Текущее время: Sun Sep 15 13:55:03 2024
^C

2. Выполнить индивидуальное задание, закрепляющее на практике полученные знания (номер задания соответствует номеру студента по журналу; если этот номер больше, чем максимальное число заданий, тогда вариант задания вычисляется по формуле: номер по журналу % максимальный номер задания, где % — остаток от деления).

3. Подготовить отчёт по работе, который должен включать: краткое описание всех использованных системных вызовов; программы, написанные в ходе выполнения лабораторной

работы; протоколы выполнения программ; скриншоты с демонстрацией поведения Linux (Ubuntu); выводы.

Индивидуальные задания

1. Создать путем порождения процессов двоичное дерево из 7-ми вершин (процессов) со связями «родитель-потомок» путем последовательных вызовов функции fork(). В этом дереве каждый процесс (кроме листьев) должен порождать двух потомков. Превратить дерево в граф, путем замещения одного листа корнем. Корректно завершить все процессы. Осуществлять проверку программы путем мониторинга процессов через утилиты (ps или top).

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.

3. Породить один процесс. Аккуратно клонировать его до тех пор, пока имеются в ОС свободные ресурсы. Найти критическое значение для мощности порожденных клонов, когда дальнейшее увеличение числа процессов неприемлемо. Корректно завершить все процессы. Описать поведение Linux (Ubuntu). Провести эксперименты в виртуальной машине для разного объёма ОЗУ.

4. Породить цепочку из 5 процессов «родитель-потомок». В бесконечном цикле последовательно по цепочке печатать PID процессов (каждый процесс печатает свой PID). Программа должна завершать свою работу по сочетанию Ctrl+C.

5. Мониторинг и ограничение использования процессорного времени процессами. Изучить способ ограничения использования процессорного времени с помощью команды ulimit -t. Породить цепочку из 5 процессов «родитель-потомок». Каждый потомок должен ограничивать свою производительность на 10% по сравнению со своим родителем (необходимо использовать другие способы для снижения производительности нежели с помощью команды ulimit -t). В протоколе описать влияние ограничений на выполнение программы в целом. Корректно завершить процессы.

6. Породить 5 процессов-демонов с разными приоритетами, которые периодически (через каждые 5 секунд) записывают информацию о своем состоянии (PID и время работы) в специальный файл (например, /var/log/my_daemon.log). Для этого можно использовать системные вызовы, такие как open() и write() для записи в файл. Убедиться, что процессы работают корректно в фоновом режиме, используя команды ps или top. Правильно завершить работу процессов-демонов, обеспечив освобождение всех занятых ресурсов.

7. Создать два процесса, которые в бесконечном цикле обмениваются между собой правом печати сообщений в консоль. 1-й процесс должен печатать PID 2-го процесса, а 2-й процесс PID 1-го процесса. Обеспечить корректное завершение процессов.

Контрольные вопросы

1. Что такое процесс в контексте операционной системы? Чем процесс отличается от потока?
2. Что такое системный вызов в контексте операционной системы?
3. Что такое POSIX?
4. Какой системный вызов используется для создания нового процесса в Linux?
5. Что такое идентификатор процесса (PID), и как его можно получить в программе на языке C?
6. Какой системный вызов позволяет родительскому процессу ожидать завершения своего потомка, и как он работает?
7. Что произойдет, если родительский процесс завершится раньше своих потомков? Как это отразится на процессе потомков в операционной системе Linux?
8. Как можно программно контролировать завершение процесса в бесконечном цикле с вероятностью завершения 1/3?
9. Что делает системный вызов waitpid() и как его использовать для управления процессами в отношениях «родитель-потомок»?
10. Что происходит, если вызвать waitpid() с флагом WNOHANG, и чем это отличается от вызова без этого флага?
11. Как можно использовать системный вызов sleep() для организации пауз между выполнением операций в процессе?
12. В чем отличие между системными вызовами fork() и exec(), и как они используются при работе с процессами?
13. Как можно использовать команды ps или top для мониторинга состояния процессов в Linux и анализа их поведения?