Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук

# ЛАБОРАТОРНА РОБОТА № 4

з дисципліни «Операційні системи»

Тема: «Процеси»

Виконав:

студент 3 курсу

групи КС-32

Безрук Юрій Русланович

Перевірив: Споров О.Є.

Харків – 2020

Целью данной работы является ознакомление с полноценным созданием процессов в языке С, загрузкой и исполнением нового образа в память для выполнения процессом, ожидание завершения потомков и т.д. В ходе выполнения работы будут написаны нескольких программ, использующих эти функции.

# ХОД РАБОТЫ

**Задание 1**

В задании один требуется написать программу, которая выводит в стандартный поток вывода информацию о процессе. Кроме того, было выполнено дополнительное задание: предусмотрен аргумент, управляющий выводимой информацией.

Варианты работы программы были разбиты на 3 режима, описанных в отдельных функциях:

«Обычный» - Выводит идентификатор процесса, идентификатор родительского процесса и идентификатор группы процесса.

«Пользовательский» - Выводит всю информацию обычного режима, а также идентификатор пользователя и группы пользователя.

«Эффективный» - Выводит всю вышеперечисленную информацию, а также «эффективные» идентификаторы пользователя и группы, к которой он принадлежит.

Вызов этих режимов осуществляется при помощи соответствующих опций, которые имеют несколько вариантов, а именно: числовой (-0 – обычный, -1- пользовательский и -2 - эффективный); и буквенный (-d – обычный, -u пользовательский и -e - эффективный), который в некоторых случаях более удобен со стороны мнемоники.

При передаче опции –h выводится справка о программе.

По умолчанию (если не было передано никаких опций), программа запускается в обычном режиме.

Вся информация получается при помощи соответствующих функций заголовочного файла unistd.h.

**Задание 2**

В задании 2 нужно продемонстрировать «непредсказуемость» алгоритма переключения процессов, считая, сколько итераций цикла сделает процесс-родитель и процесс-потомок за указанное количество секунд. Количество секунд можно задать программе с консоли при запуске (по умолчанию - 3). Процессы засекают время при помощи функции time(). На следующем рисунке видно, что при одних и тех же начальных условия больше итераций получает то один, то другой процесс:

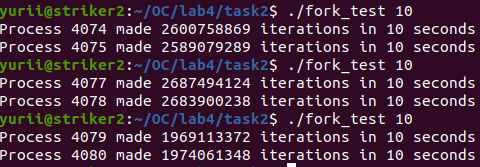


Рисунок 1 – непредсказуемость переключения процессов

**Задание 3б**

В задании 3 был предоставлен выбор задания из двух вариантов. В данной работе был реализован вариант б, в котором следовало создать некоторое количество дочерних процессов (число из клавиатуры), которые завершаются или уходят в бесконечный цикл, а процесс – родитель выводит информацию о каждом из них и удаляет (при помощи консольных команд ps и kill). Было решено для наглядности продемонстрировать оба варианта состояния дочерних процессов.

Программа считывает и обрабатывает количество дочерних процессов, которое ей необходимо создать, после чего циклически создает их, сохранияя их идентификаторы в массив. В каждом дочернем процессе при помощи генератора псевдослучайных чисел по зерну текущего времени в наносекундах случайно определяется одно из двух состояний, в котором будет находится дочерний процесс – завершится (тем самым перейдя в состояние «зомби»), или уйдёт в бесконечный цикл («продолжит работу»).

Родительский же процесс формирует строки вызовов и при помощи функции system() выводит информацию о процессах и удаляет их.

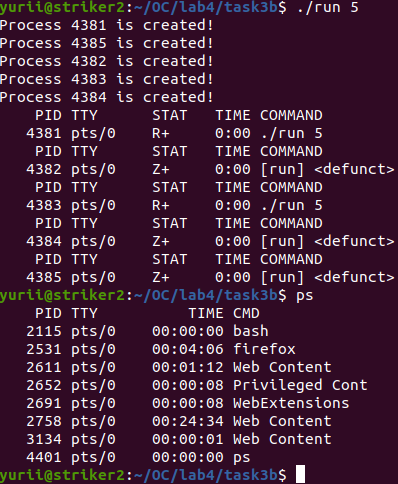


Рисунок 2 – созданные и удалённые дочерние процессы

На рисунке 2 видно, что созданные процессы находятся в одном из двух описанных выше состояниях во время вывода информации о них родительским процессом. Далее при помощи команды ps выполняем проверку, что процессы больше не существуют.

**Задание 4**

В необязательном для выполнения задании 4 нужно создать собственные аналоги функций execvp и execlp (выполняющие загрузку образа программ используя путь, указанный относительно путей, хранящихся в переменной окружения PATH), используя другие вызовы семейства exec. Таким образом, в программе реализованы функции my\_execvp(char\* file, char\*\* args) и my\_execlp(char\* file, char\* arg0, …), выполняющие вызов переданной программы с ее аргументами по пути, который был установлен в PATH, при помощи вызова execv().

Функция my\_execvp() считывает значение переменной PATH, разбивает его на массив каталогов и при помощи execv() пытается запустить программу в каждом из каталогов с массивом аргументов, переданным ей в параметрах.

Функция my\_execlp() устроена сложнее, т.к. она является функцией с переменным количеством аргументов. Вначале она, так же как и предыдущая, считывает и разбивает значение переменной PATH, а затем, при помощи специальных макросов, получает все аргументы, переданные ей среди необязательных аргументов используя переменную типа va\_list. Все аргументы сохраняются в массив аргументов и дальнейший вызов происходит по аналогии с предыдущей функцией.

Файл something.c – это исходный код программы smth, которая в целях эксперимента располгается в корне лабораторной работы (папка lab4). Именно ее программа exec (исходный код в main.c) и пытается запустить.

**Задание 5**

В задании 5 необходимо при помощи функций fork, exec, wait создать упрощенный аналог функции system().

Для более наглядной демонстрации работы был использован и доработан код из прошлой лабораторной работы – задание с упрощенным командным процессором, обрабатывающим и выполняющим команды, передаваемые в консоль. По большей части код программы идентичен, за исключением того, что вместо функции system() используется функция my\_system(), которая реализована следующим образом: при помощи реализованной в этом же файле функции split\_by() она разбивает переданную строку с командами по разделителю «;» на отдельные команды, а затем каждую из них в цикле по « » - на аргументы. Под каждую команду создается процесс-потомок, в котором выполняется функция execve() (если передано имя конкретного исполняемого файла с полным или относительным путём) или execvp() (в случае, если execve вернула код ошибки, возможно, пользователь ввел простую консольную команду Linux, поэтому выполняем запуск по переменной PATH). Если же и второй вызов не переключил программу, выводится ошибка и функция завершается с кодом -1. Процесс-родитель ожидает завершения своего потомка, а затем продолжает работу.

Программа test (исходный код в файле test.c) создана для пробных запусков системы.

**Задание 6**

В задании 6 требуется написать программу, создающую процесс-зомби и показывающую его наличие в системе, а так же его исчезновение после вызова функции wait().

Программа очень проста: она создаёт один процесс потомок, который выводит информацию о себе и завершается, переходя в состояние «зомби». Родительский же процесс в это время засыпает на 5 секунд. Выводит информацию о зомби-потомке, завершает его вызовом.wait() и показывает что среди работающих на данный момент процессов «зомби» нет. Работа программы продемонстрирована на следующем скриншоте:

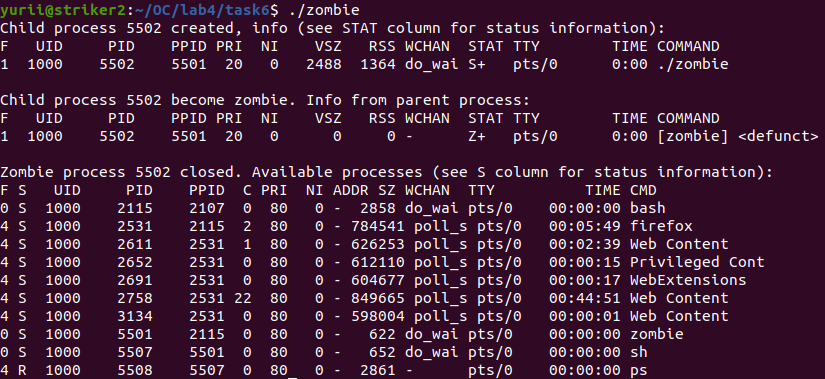


Рисунок 3 - пример процесса-зомби

**Задание 7**

Задание 7 (вычисление площади круга по методу Монте-Карло) реализовано при помощи взаимодействия процессов.

В файле sibling.c находится исходный код программы point\_count, запускаемой в дочернем процессе, которая возвращает через результат выполнения количество бросков, попавших в круг заданного радиуса, вписанный в квадрат всевозможных случайных значений. Через соответствующие опции он получает количество точек и радиус круга (по умолчанию радиус равен 1, а количество точек - 10).

Далее в функции generate\_points(), массив структур типа точка заполняется случайными значениями в заданном диапазоне (который определяется как (0-радиус; 0+радиус)) при помощи функции get\_random(double min, double max), которая является оберткой функции rand() для более удобного использования (зерно задается функцией seed() на основании текущего времени в наносекундах). После того, как точки сгенерированы, программа просматривает их и считает количество точек, попадающих в круг (расстояние до которых меньше либо равно полученному радиусу) (расстояние высчитывается функцией get\_distance() по теореме пифагора).

Программа передает количество точек как код завершения.

В файле main.c находится исходный код программы monte\_carlo. Программа принимает на вход радиус круга, количество точек и (по дополнительному заданию) количество процессов-потомков которые необходимо создать (по умолчанию 1). Вначале происходит обработка входных данных и расчёт площади квадрата, описанного вокруг круга. Далее циклично создаются процессы потомки в указанном количестве. Каждый процесс-потомок переходит в программу point\_count и выполняет её. Родитель же, после создания каждого потомка, блокируется в ожидании завершения его работы, после чего проверяет, завершился ли он про простому выходу из программы функцией exit(), и, если это так, то получает его код завершения, который является количеством точек из заданного, которые попали в круг. Этот результат подставляется в формулу, по которой считается площадь, и искомая величина записывается в массив результатов.

После завершения всех дочерних процессов, родительский процесс усредняет результаты, которые он получил путем вычислений с участием бросков каждого из потомков и выводит ответ на экран. Дополнительно оценивается аппроксимированное значение числа Пи, и значение абсолютной ошибки по отношению к реальному.

Пример работы программы изображен на скриншоте:

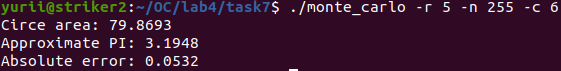


Рисунок 4 - площадь круга по методу Монте-Карло

# ВЫВОДЫ

Таким образом, в ходе выполнения данной работы были рассмотрены функции для создания дочерних процессов, переключения текущей исполняемой программы и ожидания завершения потомков. Соответствующие файлы сгруппированы в архив вместе со скриптами сборки в makef-файлах.