Лабораторная работа № 2
по курсу "Операционные системы":

Выполнил студент группы 08-208 МАИ *Куликов Алексей*.

Цель работы

Приобретение практических навыков в:

1. Управление процессами в ОС
2. Обеспечение обмена данных между процессами посредством каналов

Задание

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решения задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe).

В качестве конкретного варианта задания предлагается написание собственного простого целочисленного калькулятора с операцией "* "/". В дочернем процессе должны происходить вычисления выражений. В родительском процессе должны происходить вывод/ввод и передача их дочернему процессу (вариант 12).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Описание программы

Программа использует два процесса: один для вычисления выражений, другой – для созданий первого, получения входных данных и вывода результата.

Для организации «общения» между каналами используются однонаправленные каналы межпроцессного взаимодействия (pipe). Т. к. оба процесса взаимодействуют друг с другом, то их понадобится 2.

Далее при помощи системного вызова `fork()` дочерний процесс.

В родительском процессы происходит считывание из стандартного входного потока математического выражения, состоящего из целочисленных операндов и операторов целочисленного деления и умножения. Далее родительский процесс посредствам канала посылает строку-выражение на вычисление дочернему процессу и, дождавшись его завершения, получает от него результат и выводит его в стандартный выходной поток программы.

Дочерний процесс, получив от родительского строку-выражение, начинает его вычисление. Вычисление делится на 2 части: получение из строки-выражения операндов и оператора и самих вычислений.

Получение операндов происходит с помощью функции `fetch_int`. Ей на вход передается указатель на еще необработанную часть строки и указатель на переменную, в которую будет записан только что считанный результат. Сама функция возвращает число обработанных в результате вызова символов. Результат функци используется для «сдвига» указателя на обработанную часть строки.

Вычисление выражения происходит без использования дерева выражений и рекурсивных спусков т.к. операции умножения и деления имеют одинаковый приоритет. Вычисления аккумулируются в переменной и, когда выражение будет полностью обработано, отправляется по каналу обратно родительскому процессу.

Листинг

```
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define MAX_EXPR_L 128 // max length of expression
#define MAX_DIGS_C 11 // max digits count for int4

int fetch_int(char *str, int *num);

int main (void)
{
        int p1[2], // parent --> child
                p2[2]; // parent <-- child

        if(pipe(p1)){
                perror("pipe::p1");
        exit(-1);
        }
        if(pipe(p2)){
                perror("pipe::p2");
        exit(-1);
        }
    while(1){
        pid_t child = fork();
            if(child == -1){
                perror("fork");
                exit(-1);
            }
            else if(child > 0){
            /* Parent process */
                    char expr[MAX_EXPR_L];
                        printf("%s\n", "Please enter an expression with /or * operations only");
                        scanf("%s", expr);
                        printf("Input expression: %s\n", expr);
                    int len =  strlen(expr)+1;
                    if(write(p1[1], &len, sizeof(int)) != sizeof(int)){
                        perror("pipe::p1::write");
                        exit(-1);
```

```
        }
        if(write(p1[1], expr, len) != len){
            perror("pipe::p1::write");
            exit(-1);
        }
        int status;
        waitpid(child, &status, 0);
        if (WIFSIGNALED(status)){
            perror("child::signalled");
            fprintf(stderr, "signal: %d\n", WTERMSIG(status));
            exit(-1);
        }
        else if(WIFEXITED(status)){
            char reason = WEXITSTATUS(status);
            if(reason != 0){
                    perror("child::exited");
                    fprintf(stderr, "status: %d\n", reason);
                        exit(-1);
                    }
        }
        int res;
        if(read(p2[0], &res, sizeof(int)) != sizeof(int)){
            perror("pipe::p2::read");
            exit(-1);
        }
            printf("Result: %d\n", res);
}
else{
    /* Child process */
    char expr2comp[MAX_EXPR_L];
    int len;
    if(read(p1[0], &len, sizeof(int)) != sizeof(int)){
                    perror("pipe::p1::read::str_length");
            exit(-1);
    }
    if(read(p1[0], expr2comp, MAX_EXPR_L) != len){
            perror("pipe::p1::read::str");
            exit(-1);
    }
    close(p1[0]);

    int res = 0;
    int i = 0;
    int operand;
    char sign;

    i += fetch_int(expr2comp + i, &res);
    while(expr2comp[i] != '\0'){
            sign = expr2comp[i];
            ++i;
            i += fetch_int(expr2comp + i, &operand);

            if(sign == '*'){
                    res *= operand;
            }
            else{
                    if(operand == 0){
                            perror("computation::division_by_zero");
                            exit(-1);
                    }
                    res /= operand;
            }
    }
    if(write(p2[1], &res, sizeof(int)) != sizeof(int)){
            perror("pipe::p2::write");
```

```
                        exit(-1);
                }
                return 0;
            }
        }
        close(p1[0]);
        close(p1[1]);
        close(p2[0]);
        close(p2[1]);
        return 0;
}

/* fetch_int fetches int value from str to num and returns count
    of digs in num + 1 (it's neccessary for further fetches)*/
int fetch_int(char *str, int *num){
        int k = 0;
        char temp[MAX_DIGS_C] = { 0 };

        while(isdigit(str[k]) || str[k] == '-'){
                temp[k] = str[k];
                ++k;
        }
        *num = atoi(temp);
        return k;
}
```

Демонстрация работы

```
Please enter an expression with / or * operations only
12*3/4/7*8*5*8
Input expression: 12*3/4/7*8*5*8
Result: 320
Please enter an expression with / or * operations only
145/6*87*587*930/120/47
Input expression: 145/6*87*587*930/120/47
Result: 202102
```

Strace

```
3640  pipe([3, 4])                      = 0
3640  pipe([5, 6])                      = 0
3640  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fea4e0447d0) = 3641
...
3640  write(4, "\3\0\0\0", 4)           = 4
3640  write(4, "12\0", 3)               = 3
3640  wait4(3641, 0x7ffd6eb93f30, 0, NULL) = ? ERESTARTSYS
(To berestarted if SA_RESTART is set)
3641  <... nanosleep resumed> {tv_sec=8, tv_nsec=319581947})
= ?ERESTART_RESTARTBLOCK (Interrupted by signal)
3640  --- SIGCONT {si_signo=SIGCONT, si_code=SI_USER, si_pid=3498,
si_uid=1000} ---
3640  wait4(3641,  <unfinished ...>
3641  --- SIGCONT {si_signo=SIGCONT, si_code=SI_USER,
si_pid=3498, si_uid=1000} ---
```

```
3641  restart_syscall(<... resuming interrupted nanosleep ...>) = 0
3641  read(3, "\3\0\0\0", 4)            = 4
3641  read(3, "12\0", 128)              = 3
3641  close(3)                          = 0
3641  write(6, "\f\0\0\0", 4)           = 4
3641  exit_group(0)                     = ?
3641  +++ exited with 0 +++
3640  <... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s)
== 0}], 0, NULL) = 3641
3640  --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
si_pid=3641, si_uid=1000, si_status=0, si_utime=0, si_stime=0} ---
3640  read(5, "\f\0\0\0", 4)            = 4
```

Выводы

Процесс – экземпляр программы и необходимые ему сопутствующие данные: исполняемый код, счетчик инструкций, память процесса, pid, ppid, таблица открытых файлов и, собственно, некоторая часть памяти, в которой все это располагается включяая стек, кучу, bss и т.д.

Процессы позволяют выполнять программы параллельно (псевдо- в случае однопроцессорной системы) или создавать илллюзию параллельности.

Это используется повсеместно. Как пример любая интерактивная ОС. В ней параллельно или псевдопараллельно выполняется большое количество процессов от системных демонов, до неользовательских процессов. Например во время выполнения операционная система должна по прежнему обрабатывать пользовательский ввод, так же должна продолжаться загрузка файла с торрентов, и пиликать уведомления из соцсетей и т.д.

Параллельность достигается распределением процессорного времени между процессами. Т.е. в зависимости от стратегии планировщика всем процессам по очереди выделяется какой-то промежуток времени для выполнения, после чего запоминается текущий прогресс выполнения, текущий процесс переходит в состояние готовности к выполнению, а процессор поступает в распоряжение следующего процесса и т.д.

Порождение процессов происходит при помощи системного вызова `fork`. Создается практически полная копия родительского процесса (в том чсиле таблица файловых дескрипторов, включая pipe'ы). Таким образом далее программа разделяется на две ветви: ветвь родительского процесса и ветвь дочернего. Ветви могут общаться друг с другом.

Разделение возможно из-за того, что `fork` возвращает 0 в дочерний процесс и ненулевое знаечение в родительский (pid дочернего).

Также дочерний процесс может, совершив некоторые действия (например перенаправив свои стандартные потоки ввода/вывода), запустить сторонний исполняемый файл при помощи системного вызова `exec`.

Процесс может быть приостановлен в ожидании завершения дочернего процессса при помощи системного вызова wait. Если дочерний процесс завершается не по собственной воле, родитель получает сигнал о завершении из которого можно извлечь информацию о коде завершения WIFSIGNALED и WTERMSIG.

Процесс так же может завершиться добровольно, но с каким либо кодом ошибки информацию о которой можно так же получить при помощи комбинации макросов WIFEXIT и WEXITSTATUS.

Если процесс завершился, то имеет смысл дальнейшая обработка и производится в родительском процессе.

Данные в канале организованы по принципу FIFO (первый пришел, первый ушел).

Существуют анонимные и именовынные каналы. Анонимные существуют для внутренних взаимодействий и доступны только процессу и его дочерним процессам. Именованные каналы доступны извне и могут быть использованы так же для связи программ, изначально не преднозначенных для совместного использования.

Существует два режима чтения/записи блокирующий и неблокирующий.

Если при чтении из пустого канала процесс блокируется до появления оных в канале или когда запись осуществляется в переполненный канал, то это блокирующий режим.

Неблокирующий – когда блокировок не происходит, но выбрасываются сигналы, которые должны быть перехвачены процессами, участвующими в обмене данными.

Емкость канала в старых системах равна объему страницы памяти, в новых – 64кб.

Открытые анонимные каналы доступны так же и порожденным дочерним процессам т.к. порожденный процесс наследует таблицу файловых дескрипторов.

Открытый анонимный канал представляет собой некий файл в оперативной памяти и пару дескрипторов: по одному для чтения и записи. Для взаимного «общения» процессам необходимимо два канала (т.к. одно): первый пишет, второй – читает и наоборот.

Анонимный канал создается при помощи системного вызова pipe, принимающего на вход массив из двух чисел, в который будут записаны дескрипторы концов канала: в [0] на чтение, в [1] – на запись.

Каналы – это удобный (на мой, пока не умудренный опытом, взгляд), но не единственный способ организации обмена данными между процессами и, вероятно, не самый лучший. Другие способы еще предстоит изучить.

Итак, в ходе работы над данной задачей, были получены знания и практические навыки в управлении процессами UNIX-подобных операционных системах и организации обмена данными между ними посредством анонимных каналов.