Лабораторная работа № 4
по курсу "Операционные системы":

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

Цель работы

Приобретение практических навыков в:

1. Освоение принципов работы с файловыми системами
2. Обеспечение обмена данных между процессами посредством технологии «File mapping»

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 решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).

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 на этот раз используются средства синхронизации (POSIX-семафоры) и отображаемая совместно оспользуемая память.

Вначале в родительском процессе происходит инициализация: выделяется разделяемая память при помощи вызова `shm_open`, при помощи функции `ftruncate` в нем выделяется для выражения и двух семафоров (описаны далее). Далее разделяемый файл отображается на виртуальную память при помощи вызова `mmap`.

Далее в начале разделяемой памяти резервируется два места для семафоров. Вся оставшаяся память считается буффером.

Очень важен тот факт, что они будут находиться в разделяемой памяти и будут проинициализированы при помощи `sem_init`, с установленной в 1 переменной `pshared`. Это позволит использовать семафоры разными процессами.

Далее, собственно, происходит инициализация семафоров `ready_to_print`, `ready_to_comp` нулем.

Далее при помощи системного вызова `fork` дочерний процесс.

Т.к. семафоры были инициализированы нулем выполнение дочернего процесса сразу после запуска остановится в ождании данных от родительского. Когда данные записаны в буффер, родительский процесс "отпускает"семафор `ready_to_comp`, позволяя дочернему начать вычисление выражения. Сам же родительский процесс останавливается в ожидании сообщения о готовности от дочернего процесса. Когда дочерний процесс закончит вычисление выражения, он разблокирует семафор `ready_to_print`. Дочерний процесс печатает данные, и все повторяется заново т.к. семафоры вернулись к состояниям как после инициализации.

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>
#include <sys/mman.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <signal.h>

#define SHM_FILE_NAME "/tmp_memory"
#define MAX_EXPR_L 128
#define MAX_DIGS_C 11
```

```
#define SIZE MAX_EXPR_L+2*sizeof(sem_t)

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

int main(void)
{
        int fd = shm_open(SHM_FILE_NAME, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
        if(fd == -1){
                perror("shm::open_fail");
                exit(-1);
        }
        if(ftruncate(fd, SIZE) == -1){
                perror("trucate::fail");
                exit(-1);
        }

        char *mapped_memory = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

        if(mapped_memory == MAP_FAILED){
                perror("mmap::mapping_fail");
                fprintf(stderr, "%p", mapped_memory);
                exit(-1);
        }
        close(fd);

        int shift = 0;
        sem_t *ready_to_print = (sem_t*)(mapped_memory + shift);
        shift += sizeof(sem_t);
        sem_init(ready_to_print, 1, 0);

        sem_t *ready_to_comp = (sem_t *)(mapped_memory + shift);
        shift += sizeof(sem_t);
        sem_init(ready_to_comp, 1, 0);

        char *buffer = (char *)(mapped_memory + shift);

        while (1)
        {
                pid_t child = fork();

                if (child == -1)
                {
                        perror("fork");
                        exit(-1);
                }
                else if (child > 0)
                {
                        char expr[MAX_EXPR_L];
                        printf("%s\n", "Please␣enter␣an␣expression␣with␣/␣or␣*␣operations␣only");
                        scanf("%s", expr);
                        if(!strcmp(expr, "exit")){
                                kill(child, SIGTERM);
                                waitpid(child, NULL, 0);
                                break;
                        }
                        printf("Input␣expression:␣%s\n", expr);

                        int len = strlen(expr) + 1;
                        int res;

                        sprintf(buffer, "%s", expr);
                        sem_post(ready_to_comp);
                        sem_wait(ready_to_print);
                        sscanf(buffer, "%d", &res);
```

```
                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);
                        }
                }
                printf("Result: %d\n", res);
        }
        else
        {
                char expr2comp[MAX_EXPR_L];

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

                sem_wait(ready_to_comp);
                sscanf(buffer, "%s", expr2comp);

                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;
                        }
                }
                sprintf(buffer, "%d", res);
                sem_post(ready_to_print);
                return 0;
        }
}
if (shm_unlink(SHM_FILE_NAME)){
        perror("shm::unlink::fail");
        exit(-1);
}
if(munmap(mapped_memory, SIZE)){
        perror("mmap::munmap_failed");
        exit(-1);
```

```
        }
        sem_destroy(ready_to_comp);
        sem_destroy(ready_to_print);

        return 0;
}

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

```
openat(AT_FDCWD, "/dev/shm/tmp_memory", O_RDWR|O_CREAT|O_TRUNC|
O_NOFOLLOW|O_CLOEXEC, 0600) = 3
ftruncate(3, 192)                       = 0
mmap(NULL, 192, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0) = 0x7fd9e5e94000
close(3)                                = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fd9e5e9ea10) = 18027
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 2), ...}) = 0
brk(NULL)                               = 0x55a94e589000
brk(0x55a94e5aa000)                     = 0x55a94e5aa000
write(1, "Please enter an expression with "..., 55) = 55
write(1, "Enter 'exit' to exit\n", 21)  = 21
fstat(0, {st_mode=S_IFREG|0644, st_size=19, ...}) = 0
read(0, "12/7*8*74/41/7\nexit", 4096)   = 19
write(1, "Input expression: 12/7*8*74/41/7"..., 33) = 33
futex(0x7fd9e5e94020, FUTEX_WAKE, 1)    = 1
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
si_pid=18027, si_uid=1000, si_status=0, si_utime=0, si_stime=0} ---
```

```
wait4(18027, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 18027
write(1, "Result: 2\n", 10)             = 10
unlink("/dev/shm/tmp_memory")           = 0
munmap(0x7fd9e5e94000, 1000)            = 0
```

Выводы

Разделяемая память и файловые отображения

Принцип работы файлового отображения довольно прост.

Файл либо непрерывный кусочек файла из дискового простанства отображается в виртуальную память процесса. Т.е. каждой ячейке в файле ставится некоторый адрес в виртуальной памяти. При этом чтение с этих адресов приводит к чтению соответствующих данных в файле, а запись – к изменению данных в файле. При этом отображение не приводит к копированию всего файла в виртуальную память, а лишь отображение страниц файла, с которыми производятся действия в программе. Т.е. используется demand paging – обращение за новой порцией данных по востребованию. Т.е. обращение к диску для копирования производится только тогда, когда происходит обращение к странице, пока не находящейся в памяти(page-fault).

Не все файлы могут быть отображены в память. Нельзя отображать, например, сокеты и терминалы.

Для работы с отображаемой памятью существует набор функций.

Отображение файла(или объекта разделяемой памяти) осуществляется при помощи вызова `mmap(addr, size, prot, mode, fd, offset)`, где:

- `addr` – желаемый адрес для начала отображения. Используется ядром всего лишь как намек на то, куда следует разместить отображение, если не указано `MAP_FIXED` в поле `mode`. Ядро может постараться разместить ближе к этому адресу. Обычно указывается `NULL` предоставляя право выбора системе.

- `size` – размер отображения в байтах.

- `size` – размер отображения в байтах

- `prot` – опции защиты файла. С помощью нее устанавливаются права на чтение/-запись/исполнение и т.д.

- `mode` – режим открытия отображения. С помощью нее устанавливается является ли отображение общим или локальным для процесса. Если отображение общее (опция `MAP_SHARED`), все изменения в отображенной области будут видны другим процессам, которым доступно отображение, а так же если в основе отображения лежит реальный файл на диске, то изменения отануться и в нем. Если отображение локальное (опция `MAP_PRIVATE`), то все изменения видны тольк для данного процесса, и не сохранятся на диск в случае отобраежния реального файла.

- `fd` – файловый дескриптор отображаемого файла.
- `offset` – смещение отображаемого "окошка"относительно начала файла в байтах.

После работы с отображаемыми файлами, как и с другими ресурсами, вообще говоря, следует их закрывать. Это можно сделать с помощью вызова `munmap(addr, size)`, где `addr` – адрес начало отображения, `size` – размер отображения.

Достоинства:

- Отображение файлов могут использоваться как для межпроцесссного взаимодействия, так и в качестве замены последовательному вводу/выводу.
- Разделяемая память является наиболее быстрым способом межпроцессного взаимодействия. Т.к. после ее отображения на адресное пространство проессов для их общения не требуются участие ядра. В некоторых случаях это гораздо выгоднее, а в некоторых совсем не оправдано и может даже замедлить программу. Все зависит от цели работы программы.
- Удобство использования – работа как с простой памятью процесса. После отображения файла/разделяемой памяти с ней можно обращаться как с простой памятью процесса. Т.е. дозволен произвольный доступ, вместо последовательного, как при операциях `read`/`write`.

Но есть и свои минусы.

Недостатки:

- Чуть более сложная организация межпроцессного взаимодействия. Приходится использовать средства синхронизации такие как мьютексы, семаафоры и др.
- Размер отображаемых файлов ограничен. Файл больший, чем адресуемое пространство можно обрабатывать только порциями.

Но, с другой стороны, это более гибкий способ.

Семафоры

Семафор – еще один способ синхронизации процессов/потоков. Главное отличие от мьютексов – может быть разблокирован любым процессом, а не только тем, кто заблокировал. Так же немного отличается цель использования. Мьютексы используются для блокирования доступа к ресурсу для работы именно с актуальными данными, чтобы никто другой не мог их менять.

Семафоры же несут другую идею. Здесь процесс/поток либо ожидает сигнала от другого, либо сам сигналит. Но не то и другое вместе.

Каналы, как оказалось – это далеко не самый удобный способ организации обмена данными между процессами и, вероятно, не самый быстрый. Выяснилось, что гораздо

удобнее и порой гораздо быстрее использовать отображаемую память. Причем как для работы с реальными файлами, так и для межпроцессного взаимодействия.

Итак, в ходе работы над данной задачей, были дополнены знания и практические навыки в управлении процессами UNIX-подобных операционных системах и организации обмена данными между ними разделяемой памяти и отображений.