ACS. OS. HW8

Горбачева Маргарита Валерьевна | БПИ-245

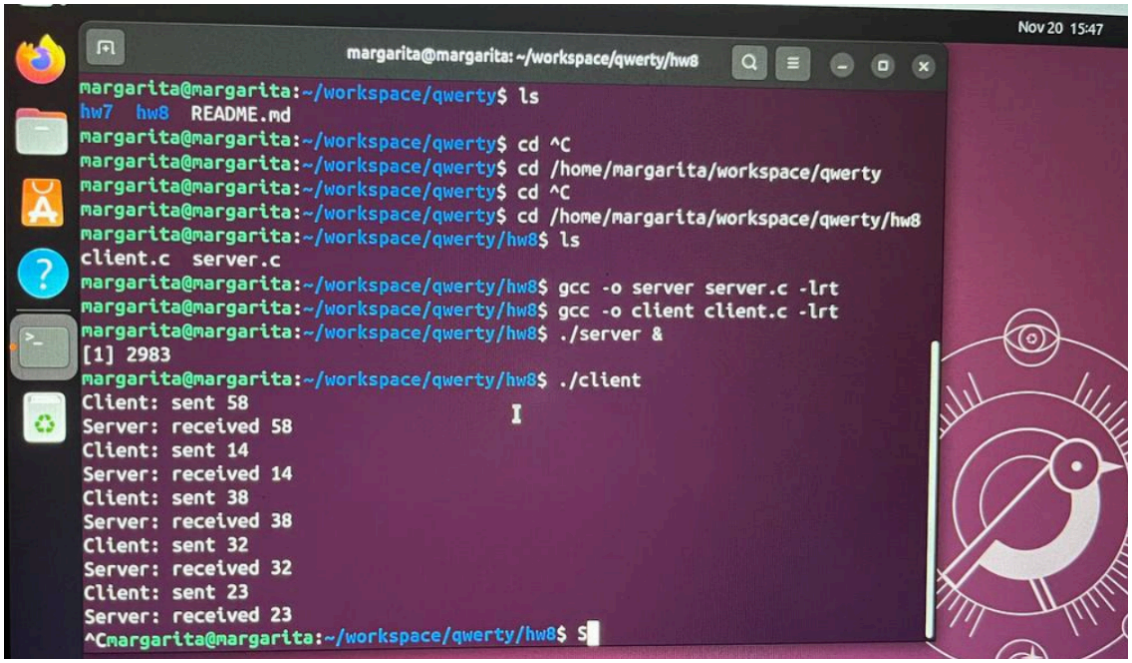
Задание:

Разработать независимые программы клиента и сервера, взаимодействующие через разделяемую память с использованием функций POSIX. Клиент в автоматическом режиме генерирует случайные числа в том же диапазоне, что и клиент, рассмотренный на семинаре. Сервер осуществляет вывод данных из разделяемой памяти. Предполагается (но специально не контролируется и, следовательно, не реализуется), что запускаются только один клиент и один сервер. Необходимо обеспечить корректное завершение работы для одного клиента и одного сервера, при котором удаляется сегмент разделяемой памяти и оба завершаются процесса при попытке завершить один из них. Предложить и реализовать свой вариант корректного завершения. Описать этот вариант в отчете.

Решение: решение данного домашнего задания лежит в папке GitHub репозитория по ссылке: https://github.com/Misss-Lacoste/ACS-OS/tree/main/hw8

NotaBene!! *Прошу заметить, что комментарии в кодах написаны на английском языке, т.к. задание выполнялось на виртуальной машине линукса, которая не поддерживает русский язык.*

Демонстрация работы программы:



Как работает программа:

1. Нам необходимо разработать 2 программы – для клиента, который будет генерировать какое-либо число и отправлять его серверу, и сам сервер, который будет получать то самое число и выводить нам его. И клиент, и сервер должны взаимодействовать не напрямую, а используя один и тот же участок разделяемой памяти, используя функции POSIX (*Portable Operating System Interface — переносимый интерфейс операционных систем — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов*[Wikipedia]).
2. В кодах программ клиента и сервера находится одна и та же общая структура данных, которая и создает(имитирует, не знаю, как лучше сказать) участок разделяемой памяти.

   ```
   typedef struct {
       volatile int stop_flag;
       volatile int number;
       volatile int start_reading;
   } SharedData;
   ```

   То есть:
   Тут stop_flag - флажок завершения операций. Number - генерируемое число, а start_reading - флажок, означающий готовность числа к чтению.

3. Рассмотрим код сервера.

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

#define Shared_Memory "/Shared_Memory"

//here we will set the structure for shared (razdelyaemaya) memory
typedef struct {
    volatile int stop_flag; //volatile - peremennye may differ from process to process, flag to stop the process
    volatile int number; //generated number
    volatile int start_reading; //flag that means that number is ready to be read
} SharedData;

void cleanup(int sig) { //process the signal to stop the program using
```

```c
ctrl+c
    shm_unlink(Shared_Memory); //deletes the segment of a shared memory
    _exit(0);
}

int main() {
    signal(SIGINT, cleanup); //here we set the cleanup as a signal; handler
sigint

    int fd = shm_open(Shared_Memory, O_CREAT | O_RDWR, 0666); //opens the
share memory space and starts to work with it, gives the access rights(to
read, write and all in one)
    if(fd == -1) {
        perror("shm_open (server)");
        return 1;
    }

    if (ftruncate(fd, sizeof(SharedData)) == -1) { //sets the
segment(SharedData) in bytes(its size)
        perror("ftruncate");
        return 1;
    }

    SharedData *data = mmap(NULL, sizeof(SharedData), PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0); //mmap sets the segment to the address
space of a process and also checks the rights
    if(data == MAP_FAILED) {
        perror ("mmap");
        return 1;
    }
    data -> stop_flag = 0; //here we initialize the inital values in shared
memory
    data -> start_reading = 0;

    while (!data -> stop_flag) { //checks if we have to kill the process?
if the stop_flag == 1
        if (data -> start_reading) {
            printf("Server: received %d\n", data->number);
            data->start_reading = 0;
        }
        usleep(100000); //in order to give a processor a chance to relax
for a while
    }
```

```
    munmap(data, sizeof(SharedData)); //separates shared memory from the
address space
    close(fd);
    shm_unlink(Shared_Memory); //deletes the segment of a memory
    return 0;
}
```

Стоит заметить, что строка, в которой определяем сегмент памяти, именовать будем таким образом, чтобы клиент и сервер находили этот один общий участок памяти, тк в линуксе он хранится в директории /dev/shm/Shared_Memory

- Функция cleanup() вызывается, когда мы нажимаем сигнал прерывания Ctrl+C. Тут shm_unlink удаляет сегмент разделяемой памяти из системы, а exit(0) завершает процесс.
- signal(SIGINT, cleanup) - устанавливаем обработчик, то есть когда мы нажимаем Ctrl+C это преобразуется в SIGINT, и вызывается функция cleanup().
- Далее создаем сегмент разделяемой памяти. Файловому дескриптору присваиваем функцию из POSIX, которая создает или открывает сегмент с именем /Shared_Memory, далее в параметрах задаем права доступа. Иначе завершаемся аварийно.
- Устанавливаем размер сегмента. ftruncate() устанавливает размер сегмента, и размер структуры, которую мы хотим поместить, а это как раз наша структура SharedData.
- Отображаем сегмент в память. mmap() - отображает сегмент в адресное пространство процесса. Null нужен для того, чтобы система сама выбрала адрес в адресное пространство, даем также права доступа. MAP_SHARED для того, чтобы изменения были видны также другим процессам, которые используют этот сегмент. 0 - для смещения в байтах.
- Инициализируем начальные значения.
- Основной цикл сервера: цикл работает, пока флажок остановки процесса не будет = 1. Если start_reading = 1 - это означает, что клиент записал число, а сервер читает number и выводит его. Далее start_reading снова сбрасывается до 0, и клиент снова может “придумывать и записывать число”. Sleep() - даём немножко процессору релаксnуть.
- Завершаем работу. То есть munmap() отключает сегмент от адресного пространства, далее закрываем дескриптор, удаляем сегмент из памяти.

4. Теперь рассмотрим код клиента.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/types.h>


#define Shared_Memory "/Shared_Memory"
typedef struct {
    volatile int stop_flag;
    volatile int number;
    volatile int start_reading;
} SharedData;

void cleanup(int sig) { //process the signal to stop the prigram using ctrl+c
    exit(0);
}

int main() {
    signal(SIGINT, cleanup); //here we set the cleanup as a signal; handler sigint

    int fd = shm_open(Shared_Memory, O_RDWR, 0666); //opens the real segment
    if (fd == -1) {
        perror("shm_open(client)");
        return 1;
    }

    SharedData *data = mmap(NULL, sizeof(SharedData), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0); //reflects the segment to the memory
    if(data == MAP_FAILED) { //checks if mistakes occur
        perror("mmap");
        return 1;
    }

    srand(time(NULL)); //here we will generate the random values according
```

```
to the task

    while (!data -> stop_flag) { //checks the stop flag
        if (!data-> start_reading) {
            data -> number = rand() % 100; //generated value is written to
the peremennaya number
            data -> start_reading = 1;
            printf("Client: sent %d\n", data->number);
        }
        sleep(1);
    }
    munmap(data, sizeof(SharedData)); //frees the resourses, closes process
    close(fd);
    return 0;
}
```

- Также имеется cleanup() - обработчик сигнала аналогично, как и в коде сервера
- Аналогичная установка обработчика(в случае, если вызывается Ctrl+C)
- Открытие существующего сегмента памяти, но тут без параметра O_CREAT, тк сервер нам его уже создал.
- Отображение сегмента в память
- Вот тут новенькое: инициализация генератора случайных чисел методом srand(time(Null) - текущее время в секундах, чтобы числа были разными при каждом запуске.
- Основной цикл клиента, цикл также работает, пока флажок остановки не станет равным единице. Если start_reading = 0, значит сервер прочитал предыдущее число и клиенту нужно снова генерировать новое значение, тогда start_reading становится = 1 и все начинается заново 🙂, пока, конечно, я не нажму Ctrl+C...
- Ну и также завершаем работу

5. Таким образом, синхронизация клиента и сервера происходит через флажок start_reading!