# Лабораторная работа 8. ИСПОЛЬЗОВАНИЕ ОБЩИХ ФАЙЛОВ, БУФЕРИЗАЦИЯ, СИНХРОНИЗАЦИЯ

**Цель**. Освоить механизм взаимодействия между процессами на основе использования общей памяти.

# Краткие теоретические сведения.

**Перечень рассматриваемых вопросов**

* Отображение файла в оперативной памяти.
* Освобождение памяти, выделенной для отображения файла.
* Синхронизация данных при использовании отображения файлов в оперативной

В языках высокого уровня этот механизм реализован как **буферизация** (C#, Java).

**Системные вызовы** для использования общих файлов в оперативной памяти и синхронизации данного механизма межпроцессного взаимодействия объявлены в заголовочном файле **sys/ mmap.h** .

# Размещение файла в памяти и освобождение памяти

Системный вызов mmap() позволяет частично или целиком отображать в оперативной памяти содержимое файла и может предназначаться в Linux для различных целей.

Однако применение его для межпроцессного взаимодействия представляет особый интерес.

void \* **mmap** (void \* ADDRESS, size\_t LEN, int PROT, int FLAGS, int FD, off\_t OFFSET);

При успешном завершении mmap() возвращает **адрес** отображаемой области памяти. В случае ошибки возвращается указатель MAP\_FAILED.

Системный вызов **munmap()** освобождает отображаемую область памяти.

int **munmap** (void \* ADDRESS, size\_t LEN);

Данный системный вызов освобождает буфер отображаемой памяти, расположенный по адресу ADDRESS и имеющий размер LEN. При успешном завершении munmap() возвращает 0. В случае ошибки возвращается –1.

Для понимания работы механизма разделяемой памяти проделайте следующие упражнения.

## Упражнение 1

Программа демонстрирует использование системного вызова mmap(), перезаписывая содержимое файла "задом наперед".

**Программа mmap1.c**

#include <stdio.h>

#include <string.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#define FLENGTH 256

void reverse (char \* buf, int size)

{

int i;

char ch;

for (i = 0; i < (size/2); i++)

{

ch = buf[i];

buf[i] = buf[size-i-1];

buf[size-i-1] = ch;

}

}

int main (int argc, char \*\* argv)

{

int fd;

char \* buf;

if (argc < 2) {

fprintf (stderr, "Too few arguments\n");

return 1;

}

fd = open (argv[1], O\_RDWR);

if (fd == -1) {

fprintf (stderr, "Cannot open file (%s)\n", argv[1]);

return 1;

}

buf = mmap (0, FLENGTH, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

if (buf == MAP\_FAILED) {

fprintf (stderr, "mmap() error\n");

return 1;

}

close (fd);

reverse (buf, strlen (buf));

munmap (buf, FLENGTH);

return 0;

}

Проверяем:

$ gcc -o mmap1 mmap1.c

$ echo "" > myfile

$ echo -n LINUX >> myfile

$ ./mmap1 myfile

$ cat myfile

XUNIL

Объясните принцип работы данной программы.

# Синхронизация с использование общих файлов

**Синхронизация** разделяет среди двух или более процессов (потоков) **общий ресурс**, который может быть *одновременно доступен только одному потоку*.

Данные файла, отображаемые в памяти, на самом деле существуют отдельно от самого файла.

Если отображаемая область изменяется, то сброс данных файла на носитель осуществляется только при вызове munmap(). Однако использование отображаемых в памяти файлов при реализации межпроцессного взаимодействия зачастую требует осуществить сброс данных на носитель (синхронизацию) без отключения отображаемой области. Это позволяет делать системный вызов msync().

int **msync** (void \* ADDRESS, size\_t LEN, int FLAGS);

При реализации межпроцессного взаимодействия в третьем аргументе msync() обычно фигурирует флаг MS\_SYNC, который не дает системному вызову завершиться до тех пор, пока данные не будут синхронизированы.

## Упражнение 2

Пример межпроцессного взаимодействия на основе механизма отображения файлов в памяти. Для разделения доступа к общему файлу применяется семафор.

**Программа mmap2-owner.c**

#include <stdio.h>

#include <sys/sem.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#define SEM\_KEY 2007

#define MM\_FILENAME "message"

union semnum {

int val;

struct semid\_ds \* buf;

unsigned short \* array;

} sem\_arg;

int main (int argc, char \*\* argv)

{

int fd, sem\_id, msg\_len;

struct sembuf sb[1];

unsigned short sem\_vals[1];

char \* mbuf;

if (argc < 2) {

fprintf (stderr, "Too few arguments\n");

return 1;

}

msg\_len = strlen (argv[1]);

fd = open (MM\_FILENAME, O\_RDWR | O\_CREAT | O\_TRUNC, 0600);

if (fd == -1) {

fprintf (stderr, "Cannot open file\n");

return 1;

}

mbuf = (char \*) malloc (msg\_len);

if (mbuf == NULL) {

fprintf (stderr, "malloc() error\n");

return 1;

}

mbuf = memset (mbuf, 0, msg\_len);

write (fd, mbuf, msg\_len);

lseek (fd, 0, SEEK\_SET);

free (mbuf);

mbuf = mmap (0, msg\_len, PROT\_WRITE | PROT\_READ, MAP\_SHARED, fd, 0);

if (mbuf == MAP\_FAILED) {

fprintf (stderr, "mmap() error\n");

return 1;

}

close (fd);

strncpy (mbuf, argv[1], msg\_len);

msync (mbuf, msg\_len, MS\_SYNC);

sem\_id = semget (SEM\_KEY, 1, 0600 | IPC\_CREAT | IPC\_EXCL);

if (sem\_id == -1) {

fprintf (stderr, "semget() error\n");

return 1;

}

sem\_vals[0] = 1;

sem\_arg.array = sem\_vals;

if (semctl (sem\_id, 0, SETALL, sem\_arg) == -1) {

fprintf (stderr, "semctl() error\n");

return 1;

}

sb[0].sem\_num = 0;

sb[0].sem\_flg = SEM\_UNDO;

sb[0].sem\_op = -1;

semop (sem\_id, sb, 1);

sb[0].sem\_op = -1;

semop (sem\_id, sb, 1);

semctl (sem\_id, 1, IPC\_RMID, sem\_arg);

munmap (mbuf, msg\_len);

unlink (MM\_FILENAME);

return 0;

}

**Программа mmap2-** **user.c**

#include <stdio.h>

#include <sys/sem.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

#define SEM\_KEY 2007

#define MM\_FILENAME "message"

int **main** (void)

{

int fd, sem\_id, buf\_len;

struct sembuf sb[1];

char \* mbuf;

struct stat st;

sem\_id = semget (SEM\_KEY, 1, 0600);

if (sem\_id == -1) {

fprintf (stderr, "semget() error\n");

return 1;

}

fd = open (MM\_FILENAME, O\_RDONLY);

if (fd == -1) {

fprintf (stderr, "Cannot open file\n");

return 1;

}

if (fstat (fd, &st) == -1) {

fprintf (stderr, "fstat() error\n");

return 1;

}

buf\_len = st.st\_size;

mbuf = mmap (0, buf\_len, PROT\_READ, MAP\_SHARED, fd, 0);

if (mbuf == MAP\_FAILED) {

fprintf (stderr, "mmap() error\n");

return 1;

}

close (fd);

write (1, mbuf, buf\_len);

printf ("\n");

munmap (mbuf, buf\_len);

sb[0].sem\_num = 0;

sb[0].sem\_flg = SEM\_UNDO;

sb[0].sem\_op = 1;

semop (sem\_id, sb, 1);

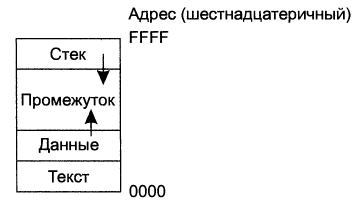
return 0;

}

Объясните принцип работы данной программы. Результаты выполнения подтвердите *скриншотами*.

# Динамическое распределение памяти

Каждый процесс работает в отдельном адресном пространстве оперативной памяти. Под процессы отводится три сегмента: текст, динамические данные (кучи) и стек (статические данные).



Сегмент данных растет снизу вверх, а стек увеличивается сверху вниз, как показано на рисунке.

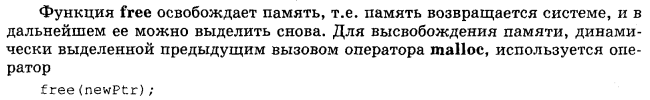
Создание и использование **динамических структур данных** требует динамического распределения памяти – возможности получать в процессе использования дополнительную память для хранения новых узлов и освобождать блоки памяти, ставшие ненужными.

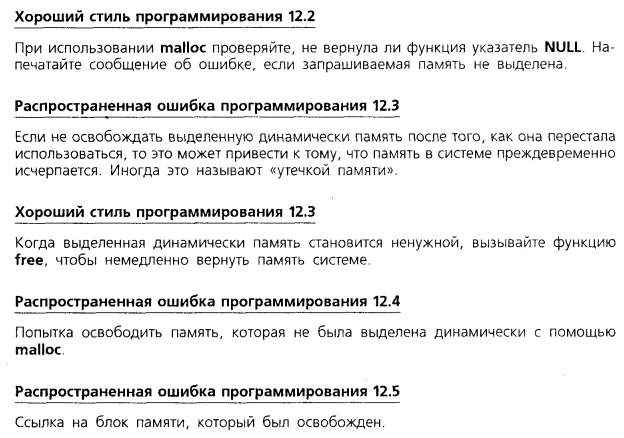
Динамического распределения памяти требуют такие структуры данных как связные списки, стеки, очереди, деревья и структуры, ссылающиеся на себя.

Максимальный размер динамически выделяемой определяется доступной физической памятью компьютера или доступным виртуальным адресным пространством в системе с виртуальной памятью. Однако часто эти размеры значительно меньше, потому что память разделяется между многими пользователями (задачами).

Для динамического выделения процессу дополнительной памяти, в системе предусмотрены четыре библиотечные функции языка С, объявленные в заголовочном файле **stdlib.h**. :

* malloc( ). Выделяет указанное количество байт памяти. Выделенная память не инициализируется, т.е. может содержать произвольные данные.
* calloc( ). Выделяет память для размещения заданного количества объектов определенного размера. Все байты выделенной памяти инициализируются нулями.
* realloc( ). Изменяет размер сегмента ранее выделенной памяти, как в большую, так и в меньшую стороны. Адрес вновь выделяемой памяти, как правило, не совпадает с прежним адресом. Прежнее содержимое памяти по возможности (насколько позволяет размеры сегментов) копируется во вновь выделенную память.
* free( ). Освобождает память, выделенную одной из первых трех функций.



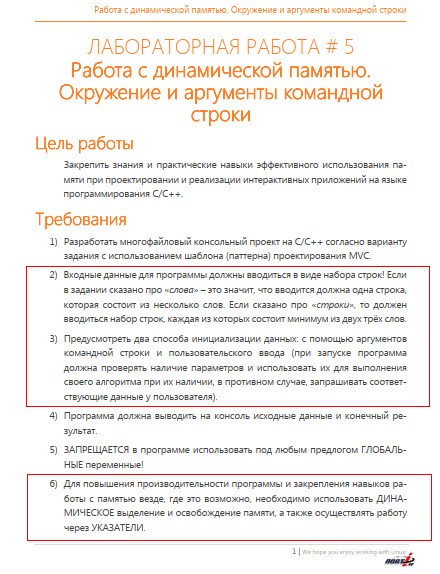


# Задания

*Выполните упражнения, объясните принцип работы программы, ответьте на контрольные вопросы.*

Выполнить задание лабораторной работы №5. Особое внимание уделите выделенным пунктам.

1. Создайте двух клиентов и один сервер, передайте данные с помощью маппинга файла.
2. Создайте два сервера и одного клиента и передайте данные через маппинг файла. Клиент должен определить от какого сервера приходит сообщение.

# Контрольные вопросы

1. Что такое **разделяемая память**?
2. Что такое **синхронизация процессов**?
3. Какие системные вызовы используются для работы с разделяемой памятью? Какие атрибуты они имеют, какие значения возвращают? Какие библиотеки необходимы для их подключения,
4. Что такое **семафоры**, для чего они используются?
5. Какие системные вызовы используются для работы с семафорами? Какие атрибуты они имеют, какие значения возвращают?
6. Опишите примерный сценарий работы с разделяемой памятью с использованием семафоров.
7. Что Вы знаете о синхронизации и семафорах в языках более высокого уровня **C#** или **Java**?
8. Что такое **динамически изменяемые структуры данных**, приведите примеры?
9. Какими средствами языка С обеспечивается динамическое выделение памяти?
10. Что такое «**утечка памяти**» и из-за чего она происходит?
11. Что такое «**Сборщик мусора**» в языках более высокого уровня **C#** или **Java**, по какому принципу (алгоритму) он работает.

## Дополнительная информация

<https://www.opennet.ru/docs/RUS/xtoolkit/x-1.html#x-1-7-3-3>

The OpenNET Project

1.7.3.3. Разделяемая память.

1.7.3.2. Семафоры.

1.8. Распределение памяти.