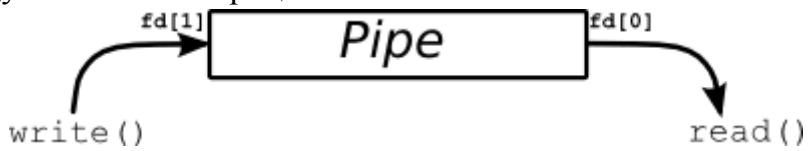
Лабораторная работа №3. Межпроцессное взаимодействие

Цель работы: изучить механизмы межпроцессного взаимодействия в ОС Linux и получить практические навыки по их использованию для организации IPC.

Теоретические сведения

Неименованные каналы (Pipes)

Неименованный канал представляет собой механизм однонаправленной передачи данных между связанными процессами.



Работа с неименованным каналом осуществляется посредством записи данных в буфер канала через точку входа и получение записанных в буфер канала данных через точку выхода. Точки входа и выхода представляются парой файловых дескрипторов, которые создаются с помощью системного вызова *pipe*.

```
int pipe_fds[2];
pipe(pipe_fds);
```

Второй из полученных дескрипторов соответствует точке входа и может быть использован в системном вызове *write* для записи данных в канал, а первый - точке выхода и может быть использован в системном вызове *read* для чтения помещенных в канал данных.

Для передачи данных между процессами один из созданных дескрипторов должен быть передан второму процессу. В данной ситуации наиболее простым и часто применяемым способом передачи нужного дескриптора является клонирование текущего процесса с помощью системного вызова *fork*. После возвращения из которого как родительский, так и дочерний процессы будут иметь свои копии дескрипторов каналов.

Наиболее частым примером использования каналов является перенаправления потоков вывода и ввода двух процессов в shell-е терминала с помощью системного вызова *dup* копирующий данный файловый дескриптор (числовой идентификатор) в первый свободный файловый дескриптор (идентификатор потока ввода/вывода).

```
ls | wc -l
```

Реализация такого механизма межпроцессного взаимодействия может быть выполнена следующим образом.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* закрыть стандартный поток вывода (stdout) */
        dup(pfds[1]);   /* задать как стандартный поток вывода pfds[1]*/
        close(pfds[0]); /* дескриптор ввода в канал не нужен */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* закрыть стандартный поток ввода (stdin) */
        dup(pfds[0]);   /* задать как стандартный поток ввода pfds[0] */
        close(pfds[1]); /* дескриптор вывода из канала не нужен */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

Именованные каналы (FIFO)

Как и в случае с неименованный каналом именованный канал представляет собой буфер для передачи данных, которому дополнительно присвоено имя. Имя канала можно интерпретировать как путь к файлу на диске, поскольку оно также обозначает узел файловой системы. В результате интерфейс для работы с именованными каналами полностью аналогичен интерфейсу для работы с обычными файлами.

Системный вызов *mknod* служит для создания узлов файловой системы.

```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

В случае с именованными каналами узел файловой системы должен быть создан с режимом комбинирующим тип узла и режим доступа к нему. Типу узла именованный канал соответствует константа *S_IFIFO*, определенная в заголовочном файле *sys/stat.h*.

Созданный узел затем может быть открыт с помощью системного вызова *open(pathname, O_WRONLY)* или *open(pathname, O_RDONLY)* для записи в канал и для чтения из канала соответственно. Причем при открытии канала системный вызов *open* блокирует исполнение потока до тех пор пока в другом потоке(процессе) не будет получен парный дескриптор канала. В случае необходимости такое поведение можно отключить, дополнив режим открытия канал значением *O_NDELAY*.

Далее запись в поток производится с помощью системного вызова *write*, а чтение - с помощью системного вызова *read*. Последний возвращает 0, когда оказываются закрыты все дескрипторы, открытые для записи. В случае же, если будут закрыты все дескрипторы открытые для чтения, процессы открывшие

канал для записи получат сигнал SIGPIPE, обработчик которого выведет сообщение “Broken pipe!” и завершает работу программы.

Очередь сообщений

В отличии от каналов фактически представляющих собой поток байт данных, очередь сообщений представляет собой поток независимых запросов/сообщений, каждое из которых может иметь свое назначение и интерпретацию при последующем использовании.

Намеревающийся использовать очередь сообщений процесс должен создать или подключиться к существующей очереди сообщений. Для этого он может использовать системный вызов *msgget*, который возвращает дескриптор очереди сообщений.

```
int msgget(key_t key, int msgflg);
```

Параметр *key* уникально идентифицирует очередь сообщений во всей системе, и в действительности является числовым значением. Параметр *msgflg* объединяет тип действия (*IPC_CREAT*) и режим доступа к очереди (полностью аналогичен режиму доступа к файлу).

Поскольку выбранное вручную числовое значение ключа в одной программе с высокой вероятностью может совпасть со значением выбранным в другой программе, общей практикой является использование функции *ftok*.

```
key_t ftok(const char *path, int id);
```

Функция принимает на вход путь к файлу *path*, к которому у данной программы есть доступ на чтение, и произвольный идентификатор *id* идентифицирующий очередь сообщений в данной программе. Затем данная функция формирует значение ключа на основе внутренней информации файловой системы, содержащей свойства файла по указанному пути, а также значению переданного идентификатора.

Само сообщение представляется в виде структуры:

```
struct actual_info {
  /* передаваемые данные */
};

struct msg_info {
  long mtype;  /* положительное значение типа */
  struct actual_info info;
};
```

Структура состоит из поля содержащего тип сообщения и имеющего тип long, и поля содержащего непосредственно передаваемые данные. В случае, если

передаваемые данные также представляют собой структурированную информацию, они оформляются в обособленную структуру, которая затем включается в структуру сообщения.
Отправка сообщения производится с помощью системного вызова *msgsnd*.

```
int msgsnd(int msqid, const void *msgp,size_t msgsz, int msgflg);
```

Параметр *msgid* соответствует дескриптору очереди сообщений. В параметры *msgp* и *msgsz* передаются указатель на структуру сообщения и размер данных сообщения (не включая размер поля типа сообщения). Параметр флагов устанавливается равным 0.
Для получения сообщения из очереди используется системный вызов *msgrcv*.

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Как можно увидеть набор параметров системного вызова *msgsnd* дополнен параметром *msgtyp*, который ограничивает тип сообщения, которое будет выбрано из очереди системным вызовом *msgrcv*. Назначение остальных параметров совпадает с одноименными параметрами системного вызова msgsnd.
Стоит отметить отличие способа интерпретации параметра *msgtyp* от его диапазона значений.

0	При выборе сообщения из очереди его тип игнорируется.
>0	Выбирается сообщение с типом *msgtyp*
<0	Выбирается сообщение с типом меньше *abs(msgtyp)*, где *abs* - абсолютное значение

Поскольку механизм очередей сообщений был одним из механизмов IPC унаследованным от System V, список созданных очередей сообщений можно просмотреть с помощью программы ***ipcs***. Ранее созданные очереди, которые более не используются, могут быть удалены с помощью команды ***ipcrm***, либо с помощью системного вызова *msgctl*.

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

В параметре *msgid* передается дескриптор очереди сообщений. Через параметр *cmd* задается тип выполняемого действия: в данном случае он равен *IPC_RMID*. Параметр *buf* устанавливается равным *NULL*.

Сегменты разделяемой памяти

Сегменты разделяемой памяти является следующим механизмом IPC, унаследованным от System V. Данный механизм предоставляет программам общий для них сегмент (участок) памяти, то есть данные записанные в этот сегмент одним процессом будут видны другим процессам. Стоит отметить, что, в отличии от ранее рассмотренных механизмов межпроцессного взаимодействия, данный механизм представляет наиболее низкоуровневый интерфейс в виде сегмента памяти содержащего "сырые" данные, где при необходимости программист сам должен разместить подходящие для решаемой задачи структуры данных, для которых определены эффективные операции.

Для использования механизма сегментов разделяемой памяти процесс должен создать или подключиться к существующему в системе сегменту разделяемой памяти с помощью системного вызова *shmget*, возвращающего дескриптор сегмента разделяемой памяти.

```
int shmget(key_t key, size_t size, int shmflg);
```

В параметр *size* передается размер запрашиваемого сегмента разделяемой памяти. Значение параметра *key* формируется по тем же принципам, что и для системного вызова *msgget*. Параметр *shmflg* также комбинирует режим доступа к ресурсу и тип действия *IPC_CREAT*.

Системный вызов *shmget* выделит страницы памяти необходимого суммарного объема, образующие сегмент разделяемой памяти. Однако этот сегмент памяти ещё должен быть отображен в адресное пространство процесса (присоединен к процессу) с помощью системного вызова *shmat*, возвращающего адрес сегмента разделяемой памяти в адресном пространстве процесса.

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

Первым параметром передается дескриптор сегмента разделяемой памяти. Затем указывается желаемый адрес сегмента в памяти процесса, который может быть задан значением *NULL*. Параметр *shmflg* может быть установлен в *SHM_RDONLY*, если процесс намерен только читать из отображенного сегмента, либо задан значением 0.

Полученный адрес сегмента памяти далее находится в полном распоряжении программы.

По окончанию использования сегмента разделяемой памяти программа должна отсоединить его от виртуального адресного пространства ее процесса с помощью системного вызова *shmdt*, принимающего в качестве единственного аргумента адрес сегмента, возвращенный системным вызовом *shmat*.

После отсоединения всех отображений в виртуальное адресное пространство сегмента разделяемой памяти, последний будет удерживать выданные ресурсы ОС (страницы памяти), которые необходимо освободить удалив сегмент

разделяемой памяти с помощью системного вызова *shmctl(shmid, IPC_RMID, NULL)*, где *shmid* - дескриптор этого сегмента.

Отображение файлов

При разработке программ возникает ситуация, когда один или несколько процессов активно модифицируют содержимое некоторого файла, например, в случае сохранения сложной структуры данных, содержащей результат работы пользователя над его проектом. В этом случае каждое изменение некоторой части структуры данных повлечет за собой необходимость перемещения каретки для записи в файл с помощью функции *fseek* и последующую запись измененной части структуры данных в соответствующий фрагмент файла.

Этих дополнительных манипуляций можно избежать за счет использования механизма отображения файла в виртуальное адресное пространство, после чего любые изменения в участке памяти, в который был отображен файл, будут немедленно отражены в этом же файле.

Для отображения файла используется системный вызов *mmap*, который возвращает адрес сегмента в виртуальном адресном пространстве, в который был отображен предварительно открытый с помощью системного вызова *open* файл.

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

В параметр *addr* здесь передается желаемый адрес, в который следует отобразить файл. В случае, если этот адрес указан некорректно (например, не кратен размеру страницы памяти) *mmap* вернет ошибку.

Следующий параметр *len* указывает размер блока данных, который необходимо отобразить в виртуальное адресное пространство. В случае если размер этого блока (или размер всего файла) не кратен размеру страницы, то отображенные данные дополняются нулевыми значениями до конца страницы, изменение которых не будет отражено в файле.

Параметр *prot* задает ограничение доступа к страницам памяти, в которые был отображен файл. Значение этого параметра комбинируется из констант *PROT_READ*, *PROT_WRITE* и *PROT_EXEC*, которые соответствуют доступу на чтение, запись и исполнение данных. Итоговое значение этого параметра должно выбираться в соответствии с режимом указанным в системном вызове *open*.

При необходимости параметр флагов *flags*, может указать, что сегмент памяти, в который был отображен файл, должен разделяться несколькими процессами (*MAP_SHARED*) либо наоборот должен содержать локальную для данного процесса копию содержимого файла (*MAP_PRIVATE*).

Далее передается дескриптор открытого файла и смещение требуемого блока данных в этом файле через параметры *fd* и *off* соответственно. Последний также должен быть кратен размеру страницы. Ее размер можно получить с помощью функции *getpagesize*.

По окончанию работы, выделенный сегмент памяти необходимо освободить передав в системный вызов *munmap* адрес возвращенный *mmap*, и его длину.

Задания

1. Выбрать согласно номеру своего варианта механизм межпроцессного взаимодействия и решить соответствующую задачу.

   Реализовать вычисление определителя квадратной матрицы с помощью разложения ее на определители меньшего порядка. При этом «ведущий» процесс рассылает задания «ведомым» процессам, последние выполняют вычисление определителей, а затем главный процесс вычисляет окончательный результат. Взаимодействие выполнить с помощью:

   1 - именованных каналов (FIFO)
   2 - очереди сообщений
   3 - сегментов разделяемой памяти
   4 - отображения файлов

   Реализовать нахождение обратной матрицы методом Гаусса, при этом задания по решению систем линейных уравнений распределяются поровну для каждого процесса. Взаимодействие выполнить с помощью:

   5 - именованных каналов (FIFO)
   6 - очереди сообщений
   7 - сегментов разделяемой памяти
   8 - отображения файлов

   Реализовать перемножение двух матриц с помощью нескольких процессов: каждый процесс выполняет перемножение строки первой матрицы на столбец второй (в соответствии с правилом умножения матриц). При необходимости процессам, выполняющим умножение, может быть отправлено несколько заданий. Взаимодействие выполнить с помощью:

   9 - именованных каналов (FIFO)
   10 - очереди сообщений
   11 - сегментов разделяемой памяти
   12 - отображения файлов

   Реализовать алгоритм блочной сортировки файла целых чисел. Каждый процесс, выполняющий сортировку, получает свою часть файла от ведущего процесса и сортирует его. Ведущий процесс выполняет упорядочивание уже отсортированных блоков. При необходимости ведомым процессам может быть выделено более одного задания на сортировку. Взаимодействие выполнить с помощью:

   13 - именованных каналов (FIFO)
   14 - очереди сообщений
   15 - сегментов разделяемой памяти
   16 - отображения файлов

   Реализовать обмен текстовыми сообщениями между несколькими процессами. Обеспечить возможность отправки сообщения сразу нескольким адресатам. Реализовать подтверждение приема сообщения

адресатом или, в случае потери сообщения, повторную его передачу. Взаимодействие выполнить с помощью:

17 - именованных каналов (FIFO)
18 - очереди сообщений
19 - сегментов разделяемой памяти
20 - отображения файлов

2. По выполненной работе нужно оформить отчет, содержащий описание ключевых аспектов реализации и демонстрацию работы написанных программ.

Ссылки

1. Beej's Guide to Unix IPC: https://web.archive.org/web/20210506163112/https://beej.us/guide/bgipc/html/single/bgipc.html