Системное программирование

Лекция 5

Межпроцессное взаимодействие

Файлы (пример)

Файловая система является общей для всех процессов –> взаимодействие процессов можно осуществлять через файлы. Для синхронизации доступа можно использовать вызов `flock()` [см. лекцию 2].

Основным минусом такого подхода является производительность.

Примечание: вызовы `read()/write()` не являются атомарными – из-за этого, например, попытка одновременной записи может привести «перемешиванию» (и итоговому повреждению) данных.

Отображение файлов в память

Для отображения файлов в память используется системный вызов mmap().

```
void* mmap(void* addr, size_t length, int prot, int flags,
                                      int fd, off_t offset);
```

Параметры:

addr – адрес, по которому *желательно* выделить память [опционален];
length – размер выделяемой памяти (должен быть кратен размеру страницы);
prot – режим доступа (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE);
flags – флаги (**MAP_PRIVATE, MAP_SHARED**, MAP_FIXED);
fd, offset – см. следующий слайд.

Вызов возвращает указатель на выделенную память или NULL.

Доп. вызовы:

```
int munmap(void* addr, size_t length);          //удалить отображение
int mprotect(void* addr, size_t length, int prot); //изменить права
```

См. также: флаг MAP_ANONYMOUS, файл /proc/<PID>/maps.

Отображение файлов в память

```
void* mmap(void* addr, size_t length, int prot, int flags,
                                      int fd, off_t offset);
```

Область файла `[offset, offset+length)` будет отображена в область памяти размера `length`.

Если в `flags` указан флаг MAP_SHARED, то чтение/запись в пределах этой области памяти будет соответствовать чтению/записи в область файла.

Значения в `offset` и `length` должны быть кратны размеру страницы.

Изменения будут переноситься в файл не мгновенно. Для синхронизации состояния в памяти и на диске используется вызов `msync()`.

```
int msync(void *addr, size_t length, int flags);
```

См.также: `mprotect()`, `mlock()`

Guard Pages (пример)

С помощью вызовов `mmap()` и `mprotect()` можно создавать т.н. **guard pages**.

Такие страницы, расположенные до или после защищаемых данных, можно использовать для обнаружения переполнения – попытка переполнения данных приводит к получению сигнала SIGSEGV.

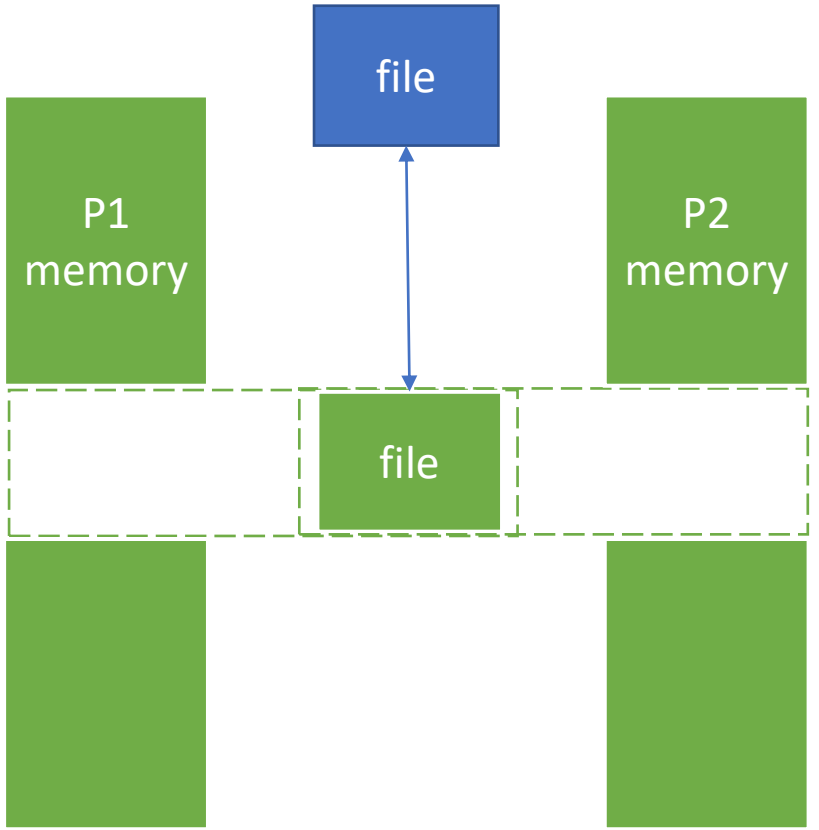
```
void* data = mmap(NULL, 3*PAGE_SIZE, PROT_NONE,
                                    MAP_PRIVATE|MAP_ANONYMOUS, -1, 0 );

mprotect((char*)data+PAGE_SIZE, PAGE_SIZE, PROT_READ|PROT_WRITE);
```

Разделяемая память (пример)

Если несколько процессов отображают один и тот же файл* в свое адресное пространство с флагом MAP_SHARED, то ОС выделит один общий регион памяти и добавит его в адресное пространство всех процессов.

Содержимое данного региона памяти будет доступно всем участвующим процессам.

Отображение файлов в память сохраняется после `fork()`.



**точнее, одну и ту же часть одного и того же файла*

Файлы разделяемой памяти (пример)

POSIX позволяет создавать файлы, расположенные целиком в оперативной памяти. Отображение подобных файлов в память процесса просто добавляет соответствующую область памяти к адресному пространству процесса.

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

Параметры и их назначение совпадают с параметрами `open()` и `unlink()`, за исключением требования, что имя файла должно начинаться с / и не должно содержать других / (т.е. «/имя», не «/каталог/имя»).

Вызов `shm_open()` возвращает дескриптор открытого файла.

Семафоры

Семафор – примитив синхронизации, представляющий собой атомарный счетчик, значение которого всегда неотрицательно (*man 7 sem_overview*).

- При захвате семафора его значение уменьшается на 1;
- Если значение семафора == 0, то при попытке блокировки поток приостановится, пока значение семафора не станет положительным.
- В UNIX семафор является файлом специального вида.

Семафоры

Для создания/открытия семафора используется функция:

```
sem_t* sem_open(const char* name, int flags, mode_t mode,
                                              unsigned int value);
```

Параметры:

name	–	имя семафора (должно начинаться с /);
flags, mode	–	см. open();
value	–	начальное значение создаваемого семафора.

Доп. функции:

```
int sem_close(sem_t* sem);          /* закрыть семафор*/
int sem_unlink(const char* name); /* удалить семафор*/
```

Семафоры (пример)

Для уменьшения и увеличения значения семафора определен ряд функций

```
int sem_post(sem_t* sem);       /* ++sem */
int sem_wait(sem_t* sem);       /* --sem или блокировка*/
int sem_timedwait(sem_t* sem,   /* --sem или errno==ETIMEDOUT */
        const struct timespec *abs_timeout); /
int sem_trywait(sem_t* sem);    /* sem-- или errno==EAGAIN */
```

Каналы

Каналы (pipe)– объекты ядра, служащие для однонаправленной передачи данных между процессами (*man 7 pipe*).

- Каналы автоматически создаются оболочкой для соединения процессов в конвейере (*ls | grep*).
- Работа с каналами не отличается от работы с файлами (`read/write/close`).
- Каналы имеют внутренний буфер (по умолчанию, 64КБ).
- Если буфер канала заполнен, то при попытке записи в канал вызов `write()` заблокируется, пока в буфере не появится место. Место освобождается при чтении.
- Если в канале нет данных – при чтении поток также блокируется.
- Если канал не открыт для записи ни в одном процессе, чтение из канала будет возвращать 0 (новых данных не будет -> канал «закончился»).
- Если канал не открыт для чтения ни в одном процессе, то при попытке записи процессу будет послан сигнал SIGPIPE; если процесс игнорирует сигнал – вызов `write()` завершится с ошибкой EPIPE.

Неименованные каналы (пример)

Неименованные каналы создаются вызовом `pipe()`:

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

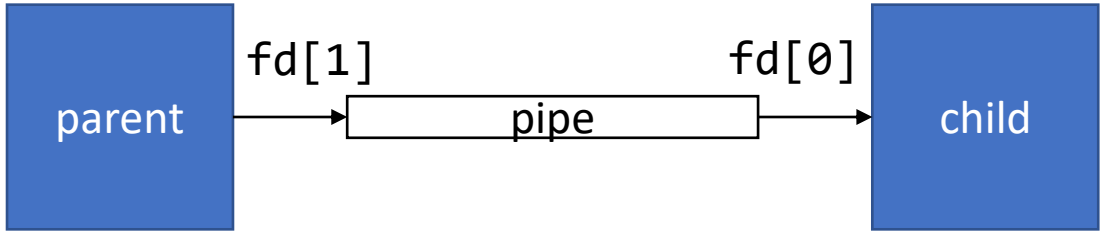
Вызов записывает в массив `pipefd` 2 файловых дескриптора.

Дескриптор в `pipefd[0]` является выходным концом канала (из него можно только читать), дескриптор в `pipefd[1]`– входным концом канала (в него можно только писать).

Обычно вызов `pipe()` выполняется перед вызовом `fork()` – в этом случае оба процесса (родительский и дочерний) будут иметь доступ к открытому каналу.

Неименованные каналы

```
int fd[2];
pipe(fd);
if (fork()) { //parent
  close(fd[0]);
  /*use fd[1] to send */
}
else {//child
  close(fd[1]);
  /*use fd[0] to receive */
}
```



Именованные каналы (пример)

Именованные каналы создаются функцией `mkfifo()` и утилитой *`mkfifo`*

```
int mkfifo(const char* pathname, mode_t mode);
```

Параметры:

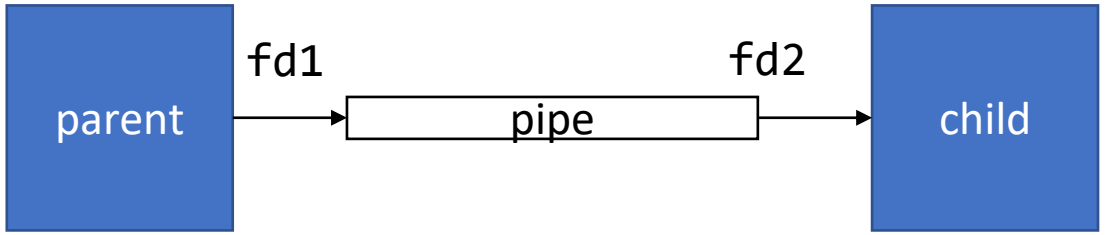
```
pathname    - путь  к создаваемому каналу;
mode        - права доступа к каналу.
```

Работать с таким каналом можно обычными вызовами `open/read/write/close`.

По умолчанию, при открытии канала `open()` заблокирует поток до тех пор, пока канал не будет открыт кем-то еще.

Именованные каналы

```
mkfifo("named_pipe", S_IWUSR | S_IRUSR);

if (fork()) { //parent
   int fd1 = open("named_pipe", O_WRONLY);
   /*use fd1 to send */
}
else {//child
   int fd2 = open("named_pipe", O_RDONLY);
   /*use fd2 to receive */
}
```



Очереди сообщений

Очередь сообщений (message queue) – средство межпроцессного взаимодействия, предназначенное для обмена отдельными порциями данных (сообщениями) (*man mq_overview*).

- Сообщению в канале может быть присвоен приоритет.
- В отличие от канала, очередь явно разделяет данные, записанные в разное время (в канале данные идут друг за другом без всяких разделителей).
- Очереди сообщений активно используются в Python (multiprocessing.Queue).
- Очередь сообщений может уведомлять процесс о поступлении новых сообщений (*man mq_notify*).

Очереди сообщений

Для создание/открытия очереди используется вызов mq_open():

```
mqd_t mq_open(const char* name, int oflag, mode_t mode,
                                           mq_attr* attr);
```

Параметры:

name – имя очереди (должно начинаться с /);
oflag, mode – см. open();
attr – указатель на атрибуты создаваемой очереди [опционален].

В oflag можно указать O_NONBLOCK, в этом случае операции с очередью не будут блокировать поток (будут завершаться с ошибкой EAGAIN).

Доп. функции:

```
int mq_close(mqd_t mqdes);          /*закрыть очередь*/
int mq_unlink(const char* name); /*удалить очередь*/
```

Атрибуты очередей

```
struct mq_attr {
  long mq_flags;       /* Flags: 0 или O_NONBLOCK */
  long mq_maxmsg;      /* Макс. кол-во сообщений в очереди */
  long mq_msgsize;     /* Макс. размер сообщения*/
  long mq_curmsgs;     /* Количество сообщений в данный момент */
};
```

Функции получения/изменения атрибутов:

```
int mq_getattr(mqd_t mqdes, struct mq_attr* attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr* attr);
```

Очереди сообщений

```
int mq_send(mqd_t mqdes, const char* msg_ptr,
                         size_t msg_len, unsigned int msg_prio);
```

Параметры:

`mqdes`	–	дескриптор очереди;
`msg_ptr`	–	указатель на данные;
`msg_len`	–	размер данных;
`msg_prio`	–	приоритет сообщения.

Если в очереди уже содержится максимальное число непрочитанных сообщений (*man mq_overview*), то вызывающий поток блокируется до тех пор, пока в очереди не появится место.

Очереди сообщений

```
ssize_t mq_receive(mqd_t mqdes, char* msg_ptr,
                   size_t msg_len, unsigned int* msg_prio);
```

Параметры:

`mqdes` – дескриптор очереди;
`msg_ptr` – буфер для данных;
`msg_len` – размер буфера;
`msg_prio` – буфер для приоритета сообщения.

Функция возвращает размер прочитанного сообщения.

Если в очереди нет сообщений, то поток блокируется до появления новых сообщений.

Размер буфера `msg_len` должен быть не меньше максимального размера сообщения очереди.

Очереди сообщений (пример)

Для исключения «вечной» блокировки удобно использовать функции с ограничением времени ожидания:

```
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                 size_t msg_len, unsigned int msg_prio,
                 const struct timespec *abs_timeout);


ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned int *msg_prio,
                        const struct timespec *abs_timeout);
```