Сигналы

Сигналы

- Сигнал – это асинхронное уведомление процесса о каком-либо событии.
- Если во время выполнения системного вызова приходит сигнал, то вызов прерывается (программное прерывание).
- **`kill -l`** список доступных сигналов
- **`kill [-номер] pid`** посылка сигнала процессу
- `man 7 signal`
- `signal.h`

Сигналы

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- **pid > 0** - сигнал sig посылается процессу с идентификатором pid.
- **pid = 0** - sig посылается каждому процессу, который входит в группу текущего процесса.
- **pid = -1** - sig посылается каждому процессу, за исключением процесса с номером 1 (init), но в Linux kill(-1, sig) не посылает сигнал текущему процессу.
- **pid < -1** - sig посылается каждому процессу, который входит в группу процесса pid.
- **sig = 0** - никакой сигнал не посылается, только выполняется проверка на ошибку.

Сигналы POSIX (управление)

Название	Код	Действие	Описание
SIGINT	2	Завершение	Сигнал прерывания (Ctrl-C) с терминала
SIGQUIT	3	Завершение с дампом памяти	Сигнал «Quit» с терминала (Ctrl-\)
SIGABRT	6	Завершение с дампом памяти	Сигнал, посылаемый функцией abort()
SIGKILL	9	Завершение	Безусловное завершение
SIGTERM	15	Завершение	Сигнал завершения (сигнал по умолчанию для утилиты kill)
SIGTSTP	20	Остановка процесса	Сигнал остановки с терминала (Ctrl-Z)
SIGSTOP	19	Остановка процесса	Остановка выполнения процесса

Сигналы POSIX (управление)

Название	Код	Действие	Описание
SIGCONT	18	Продолжить выполнение	Продолжить выполнение ранее остановленного процесса
SIGTTIN	21	Остановка процесса	Попытка чтения с терминала фоновым процессом
SIGTTOU	22	Остановка процесса	Попытка записи на терминал фоновым процессом

Сигналы POSIX (исключение)

Название	Код	Действие	Описание
SIGILL	4	Завершение с дампом памяти	Недопустимая инструкция процессора
SIGFPE	8	Завершение с дампом памяти	Ошибочная арифметическая операция
SIGBUS	7	Завершение с дампом памяти	Неправильное обращение в физическую память
SIGSEGV	11	Завершение с дампом памяти	Нарушение при обращении в память
SIGSYS	31	Завершение с дампом памяти	Неправильный системный вызов

Сигналы POSIX (исключение)

Название	Код	Действие	Описание
SIGXCPU	24	Завершение с дампом памяти	Процесс превысил лимит процессорного времени
SIGXFSZ	25	Завершение с дампом памяти	Процесс превысил допустимый размер файла

Сигналы POSIX (уведомление)

Название	Код	Действие	Описание
SIGHUP	1	Завершение	Закрытие терминала
SIGPIPE	13	Завершение	Запись в разорванное соединение (пайп, сокет)
SIGALRM	14	Завершение	Сигнал истечения времени, заданного alarm()
SIGCHLD	17	Игнорируется	Дочерний процесс завершен или остановлен
SIGURG	23	Игнорируется	На сокете получены срочные данные
SIGPOLL (SIGIO)	29	Завершение	Изменение состояния асинхронного ввода-вывода, выполняющегося в фоне
SIGVTALRM	26	Завершение	Истечение «виртуального таймера»

Сигналы POSIX

Название	Код	Действие	Описание
Отладка			
SIGTRAP	5	Завершение с дампом памяти	Ловушка трассировки или брейкпоинт
SIGPROF	27	Завершение	Истечение таймера профилирования
Пользовательский			
SIGUSR1	10	Завершение	Пользовательский сигнал № 1
SIGUSR2	12	Завершение	Пользовательский сигнал № 2

Сигналы

- Процесс может назначить свой обработчик сигнала или настроить свою сигнальную маску для игнорирования сигналов (кроме SIGKILL и SIGSTOP).
- Процесс (или пользователь) с реальным UID, не равным 0, может посылать сигналы только процессам с тем же реальным UID.
- Сигналы не могут быть посланы завершившемуся процессу, находящемуся в состоянии «зомби».

Сигналы

- Отправка сигнала текущему процессу:
  `raise(int signum); // kill(getpid(),signum);`

- Отправка сигнала SIGABRT текущему процессу:
  `void abort(void);`
  Если сигнал перехватывается или игнорируется, то после возвращения из обработчика abort() завершает программу.

- Отправка сигнала SIGALRM себе через time секунд:
  `unsigned int alarm(unsigned int seconds);`
  - Возвращает 0, если ранее alarm не был установлен или число секунд остававшихся до срабатывания предыдущего alarma.
  - Установка нового alarm'а отменяет старый.
  - Параметр seconds==0 позволяет получить оставшееся до alarm'а время без установки нового.

Группы процессов

- Группа процессов - инструмент для доставки сигнала нескольким процессам, а также способ арбитража при доступе к терминалу.
- **pgid** – идентификатор группы. Равен pid процесса, создавшего группу (лидер группы).
- В рамках одного сеанса могут существовать несколько групп процессов.
- Процесс может переходить из группы в группу внутри одного сеанса или переводить из группы в группу другие процессы сеанса.
- Перейти в группу другого сеанса нельзя, но можно создать свой собственный сеанс из одного процесса со своей группой в этом сеансе.

Группы процессов

- Включить процесс pid в группу pgid:
  `int setpgid(pid_t pid, pid_t pgid);`
  pid=0 - текущий процесс,
  pgid=0 - pgid=pid текущего процесса,
  pid=pgid=0 - создание новой группы с pgid=pid текущего процесса и переход в эту группу.

- Получить номер группы процесса pid:
  `pid_t getpgid(pid_t pid);`

- Создание группы, эквивалент setpgid(0.0):
  `int setpgrp(void);`

- Запрос текущей группы, эквивалент getpgid(0):
  `pid_t getpgrp(void);`

Задания

- **3.1** (2 балла). Написать программу, бесконечно выводящую в файл 1 раз в секунду значение внутреннего счетчика (1 2 3 …). Запустить ее в фоновом режиме (myprogram &). Узнать идентификатор процесса и протестировать команды управления (SIGINT, SIGQUIT, SIGABRT, SIGKILL, SIGTERM, SIGTSTP, SIGSTOP, SIGCONT).

Для сдачи задания нужно прислать исходный код программы и скриншоты с комментариями (что тестируется, какое результат вы ожидаете и что фактически получилось).

Системный вызов signal()

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- Изменение реакции процесса на какой-либо сигнал.
- signum – номер сигнала, обработку которого нужно изменить.
- handler – новый способ обработки сигнала:
  - указатель на пользовательскую функцию;
  - SIG_DFL для восстановления реакции на signum по умолчанию;
  - SIG_IGN чтобы игнорировать сигналы с номером signum.
- Системный вызов возвращает предыдущее значение обработчика сигнала или SIG_ERR при ошибке.

Системный вызов signal()

```
#include <signal.h>
#include <conio.h>
#include <stdio.h>

static wait = 1;

void listener(int sig) {
    //очищаем буфер
    while (getchar() != '\n');
    printf("listener: stop");
    wait = 0;
    _getch();
}
```

```
void main() {
    signal(SIGINT, listener);
    // signal(SIGINT, SIG_IGN);

    do {
        //...
    } while (wait);

    _getch();
}
```

Задания

- **3.2** (1 балл). Изменить программу из п. 3.1 так, чтобы она игнорировала сигнал SIGINT.

- **3.3** (1 балл). Изменить программу из п. 3.1 так, чтобы она завершалась после третьего сигнала SIGINT.

- **3.4** (1 балл). Изменить программу из п. 3.1 так, чтобы она печатала сообщение о получении сигнала SIGINT или SIGQUIT. Использовать одну функцию-обработчик сигнала.

Задания

- **3.5** (1 балл). Изменить программу из п. 3.4 так, чтобы она игнорировала сигнал SIGINT и SIGQUIT, если в это время идет работа с файлом.

- **3.6** (3 балла). Изменить программу из п. **2.7** так, чтобы дочерний процесс выводил информацию из файла. Если родительский процесс собирается изменить файл, то он отправляет сигнал SIGUSR1 (блокировка доступа к файлу). Когда родительский процесс завершил модификацию файла, он отправляет сигнал SIGUSR2 (разрешение доступа к файлу). Дочерний процесс отправляет новое число родительскому после того, как прочитал файл.

Системный вызов sigaction()

#include <signal.h>

- int **sigaction**(int signum, const struct sigaction *act,
  struct sigaction *oldact);
  Изменение действий процесса при получении сигнала signum (кроме SIGKILL и SIGSTOP).

Системный вызов sigaction()

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask; // Маска блокируемых сигналов
  int sa_flags; // Флаги для управления обработкой сигналов
  void (*sa_restorer)(void); // Устарел, нет в POSIX
}
```

Не используйте sa_handler и sa_sigaction вместе!

Системный вызов sigaction()

- int **sigprocmask**(int how, const sigset_t *set, sigset_t *oldset);
  Изменение списка блокированных в данный момент сигналов.

- int **sigpending**(sigset_t *set);
  Проверка наличия ожидающих сигналов (полученных заблокированных сигналов).

- int **sigsuspend**(const sigset_t *mask);
  Временное изменение маски блокировки сигналов процесса на указанное в mask. Затем приостановка работы процесса до получения соответствующего сигнала.

System V и POSIX

Очередь сообщений, семафоры, разделяемая память

- Очередь сообщений – это связный список, находящийся в адресном пространстве ядра. Каждая очередь имеет свой уникальный идентификатор IPC.
- Семафор – это механизм, который позволяет конкурирующим процессам и потокам работать с общими ресурсами и помогает в решении различных проблем синхронизации таких как гонки, дедлоки (взаимные блокировки) и неправильное поведение потоков.
- Разделяемая память – область в памяти, доступная одновременно нескольким процессам.

System V

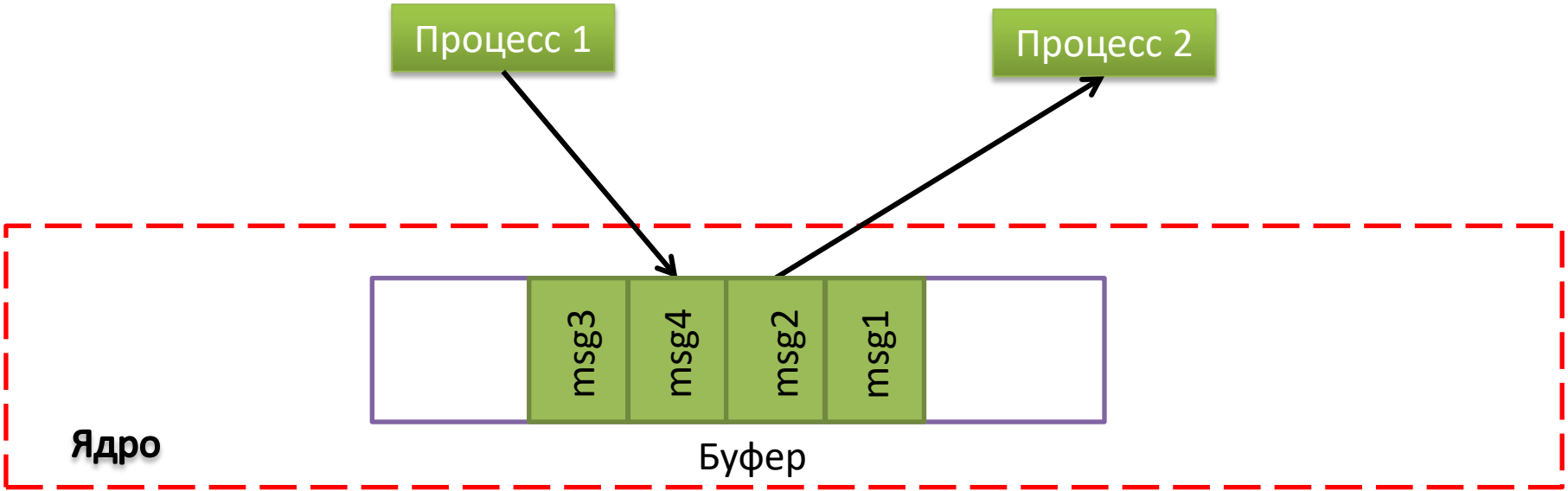
	Очереди сообщений	Семафоры	Общая память
Заголовочный файл	sys/msg.h	sys/sem.h	sys/shm.h
Создание / открытие	msgget	semget	shmget
Операции управления	msgctl	semctl	shmctl
Операции IPC	msgsnd, msgrcv	semop	shmat, shmdt

POSIX

	Очереди сообщений	Семафоры	Общая память
Заголовочный файл	mqueue.h	semaphore.h	sys/mman.h
Создание / открытие	mq_open	sem_open	shm_open
Операции управления	mq_gerattr, mq_setattr	-	fstat
Операции IPC	mq_send, mq_receive	sem_wait, sem_post	mmap, munmap

Очереди сообщений

Очереди сообщений



Очереди сообщений

- Очередь с приоритетом.
- Можно извлечь любое сообщение.
- Для создания и присоединения к очереди нужен уникальный ключ.

Очереди сообщений (System V)

- Ключ IPC – целочисленное значение типа key_t.
- Преобразование имени и идентификатора в значение типа key_t:
  key_t **ftok**(const char *pathname, int proj_id);
- Создание очереди:
  int **msgget**(key_t key, int msgflg);
  - key=IPC_PRIVATE означает создание новой очереди с ключом, который не совпадает со значением ключа ни одной из уже существующей очередей и который не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.
  - Флаги указываем только при создании очереди.

Очереди сообщений (System V)

- Отправка сообщения:
  int **msgsnd**(int msqid, struct msgbuf *msgp, size_t msgsz,
  int msgflg);
  - Если для очереди недостаточно пространства, то по умолчанию для msgsnd будет блокировка, до тех пор, пока не станет достаточно места.
  - Если IPC_NOWAIT установлено в msgflg то вызов вместо этого выдаст ошибку EAGAIN.
- Получение сообщения из очереди:
  ssize_t **msgrcv**(int msqid, struct msgbuf *msgp, size_t msgsz,
  long msgtyp, int msgflg);

Очереди сообщений (System V)

- **msqid** – идентификатор очереди
- struct **msgbuf** {
  long mtype; /* тип сообщения, должен быть > 0 */
  char mtext[N]; /* содержание сообщения */
  };
- **msgsz** – размер массива mtext
- **msgtyp**:
  msgtyp == 0 - первое сообщение из очереди;
  msgtyp > 0 - из очереди берется первое сообщение типа msgtyp;
  msgtyp < 0 - из очереди берется первое сообщение со значением, меньшим, чем абсолютное значение msgtyp.
- **msgflg**:
  IPC_NOWAIT - немедленный возврат из функции, если в очереди нет сообщений необходимого типа. Системный вызов возвращает ошибку, устанавливая значение errno равным ENOMSG;
  MSG_EXCEPT - если msgtyp > 0, то читается первое сообщение, тип которого не равен msgtyp;
  MSG_NOERROR - сократить текст сообщения, размер которого больше msgsz байтов.

Очереди сообщений (System V)

- Выполнить управляющие операции над сообщениями:
  int **msgctl**(int msqid, int cmd, struct msqid_ds *buf);
- cmd:
  - IPC_STAT - скопировать информацию из структуры данных очереди сообщений, ассоциированных с msqid в структуру с адресом buf (у вызывающего должны быть права на чтение очереди сообщений).
  - IPC_SET - записать значения некоторых элементов структуры msqid_ds , адрес которой указан в buf, в структуру данных из очереди сообщений, обновляя при этом его поле msg_ctime.
  - IPC_RMID - немедленно удалить очередь сообщений и связанную с ним структуру данных, «разбудив» все процессы, ожидающие записи или чтения этой очереди (при этом функция возвращает ошибку, а переменная errno приобретает значение EIDRM).

Очереди сообщений (System V)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc, char *argv[])
{
  int i, msqid;
  for (i=0;i<10;i++) {
    msqid=msgget(IPC_PRIVATE, IPC_CREAT);
    printf("msqid = %d\n", msqid);
    msgctl(msqid, IPC_RMID, NULL);
  }
  exit(0);
}
```

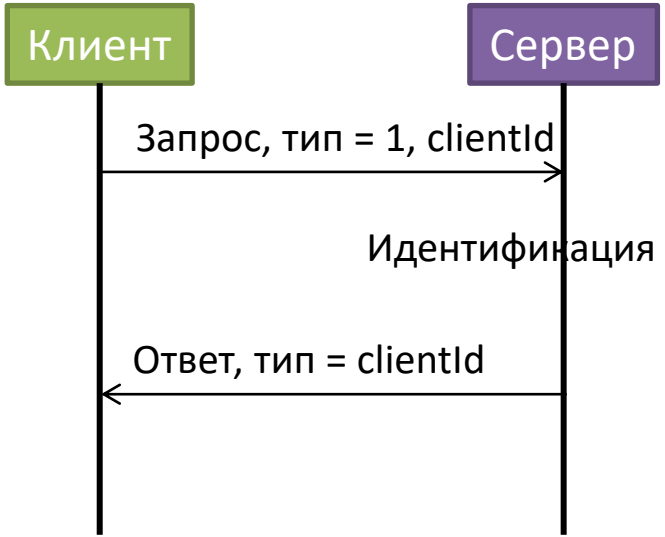
Задания

- **3.7** (4 балла). Написать две программы, использующие очередь сообщений: первая отправляет строки в очередь, а вторая – считывает и выводит на экран. Отправляемые строки можно читать из файла или генерировать в программе (например, случайные числа). Для завершения обмена данными первая программа должна отправить сообщение с типом 255.

- **3.8** (3 балла). На основе решения задачи 3.7 сделать программу для двухстороннего обмена сообщениями (персональный чат).

Мультиплексирование

- Мультиплексирование сообщений: получение сообщений одним процессом от множества других через одну очередь сообщений. Ответы отправляются через ту же очередь сообщений.
- Сервер получает запросы от других процессов - клиентов - на выполнение некоторых действий и отправляет им результаты обработки запросов.
- Сервер работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
- Сервер ждет запроса от клиентов, инициатором же взаимодействия выступает клиент.
- Клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступить запросы от нескольких клиентов.
- Клиент должен знать, как обратиться к серверу (например, какого типа сообщения он воспринимает) перед началом организации запроса к серверу, в то время как сервер может получить недостающую информацию о клиенте из пришедшего запроса.



Задания

- **3.9** (4 балла). На основе решения задачи 3.8 сделать групповой чат. Клиенты подключаются к серверу, и сервер готов пересылать сообщения. Клиент отправляет сообщение серверу. Сервер дополняет текст сообщения идентификатором клиента, и рассылает всем это сообщение.

Очереди сообщений (POSIX)

- mqd_t **mq_open**(const char *name, int oflag);
- mqd_t **mq_open**(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
- int **mq_close**(mqd_t mqdes);
- int **mq_unlink**(const char *name);
- int **mq_send**(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
- ssize_t **mq_receive**(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);

Link with **-lrt**.

Очереди сообщений (POSIX)

```
/* Отправка сообщения в очередь */
#include <stdio.h>
#include <string.h>
#include <mqueue.h>
#define QUEUE_NAME "/my_queue"
#define PRIORITY   1
#define SIZE       256
int main(){
  mqd_t ds;
  char text[] = "Hello Posix World";
  struct mq_attr queue_attr;
  queue_attr.mq_maxmsg = 32;
  queue_attr.mq_msgsize = SIZE;
  if ((ds = mq_open(QUEUE_NAME,
           O_CREAT | O_RDWR , 0600,
           &queue_attr)) == (mqd_t)-1){
    perror("Creating queue error");
    return -1;
  }
```

```
if (mq_send(ds, text, strlen(text), PRIORITY) == -1){
    perror("Sending message error");
    return -1;
  }
  if (mq_close(ds) == -1)
    perror("Closing queue error");
  return 0;
}
```

Очереди сообщений (POSIX)

```
/* Прием сообщения из очереди */
#include <stdio.h>
#include <mqueue.h>
#define QUEUE_NAME "/my_queue"
#define PRIORITY   1
#define SIZE       256

int main(){
  mqd_t ds;
  char new_text[SIZE];
  struct mq_attr attr, old_attr;
  int prio;
  if ((ds = mq_open(QUEUE_NAME,
           O_RDWR | O_NONBLOCK, 0600,
           NULL)) == (mqd_t)-1){
    perror("Creating queue error");
    return -1;
  }
```

```
attr.mq_flags = 0; /* set !O_NONBLOCK */
if (mq_setattr(ds, &attr, NULL)){
  perror("mq_setattr");
  return -1;
}
if (mq_getattr(ds, &old_attr)) {
  perror("mq_getattr");
  return -1;
}
if (!(old_attr.mq_flags & O_NONBLOCK))
  printf("O_NONBLOCK not set\n");
```

Очереди сообщений (POSIX)

```
 if (mq_receive(ds, new_text, SIZE, &prio) == -1){
    perror("cannot receive");
    return -1;
  }
  printf("Message: %s, prio = %d\n", new_text, prio);
  if (mq_close(ds) == -1)
    perror("Closing queue error");
  if (mq_unlink(QUEUE_NAME) == -1)
    perror("Removing queue error");
  return 0;
}
```

- **Задания 3.10 – 3.12:** решить задачи 3.7 – 3.9 с использованием очередей POSIX.