Лабораторная работа 2. Процессы.

Цели работы: изучение способов создания и уничтожения процессов в UNIX системах, изучение способов организации межпроцессного взаимодействия (без использования разделяемой памяти и сокетов).

1 Теоретическая часть

1.1 Определения

Многозадачность — возможность ОС запускать на выполнение сразу несколько программ.

Вытесняющая многозадачность — вид многозадачности, при котором ОС автоматически выполняет переключение между процессами (одни процессы останавливаются, другие — запускаются на выполнение взамен остановленных). Выбор процессов для запуска/остановки осуществляется планировщиком ОС.

Процесс (pid) – системная сущность, соответствующая независимому экземпляру исполняемой программы.

Идентификатор процесса — целое число, однозначно идентифицирующее незавершенный процесс.

Адресное пространство процесса — уникальный для каждого процесса способ сопоставления данных и их адреса. Механизм виртуальной памяти гарантирует, что адресные пространства разных процессов не пересекаются (изоляция процессов), за исключением пространства ядра и явно созданных общих областей.

Приоритет процесса — параметр, определяющий, насколько часто планировщик ОС должен выбирать данный процесс для продолжения выполнения.

Процесс-родитель — процесс, который создал текущий процесс вызовом fork (или аналогичным по функционалу системным вызовом). Процессы в UNIX образуют дерево процессов, корнем которого является процесс init, являющийся предком всех процессов.

Идентификатор родителя (ppid) — идентификатор процесса-родителя.

Процесс-сирота — процесс, родитель которого завершился раньше, чем данный процесс-потомок.

Процесс-зомби — завершившийся процесс, данные которого все еще присутствуют в системной таблице процессов. До тех пор, пока процесс-родитель не получит данные о завершенном процессе вызовами wait/waitpid/waitid или не откажется от отслеживания процессов-потомков блокировкой сигнала SIGCHLD, в системной таблице процессов будет

сохраняться запись о фактически не существующем процессе-потомке (процесс формально «жив», но фактически «мертв», потому - «зомби»).

Процесс init – первый процесс ОС (pid == 1), корень всего дерева процессов.

Сигнал — уведомление процесса о наступлении какого-либо события.

Обработчик сигнала — функция, вызываемая при получении определенного сигнала.

Диспозиция сигнала — действие по умолчанию для сигнала без заданного обработчика.

Блокировка сигнала – запрет получения сигналов определенного типа.

Маска сигналов — структура, определяющая, какие сигналы заблокированы в данный момент.

Реентерабельная функция – функция, выполнение которой может быть прервано выполнением этой же функции без влияния на результат исходного вызова. Реентерабельность – более сильное требование, чем потокобезопасность.

Стандартные сигналы — сигналы, назначением которых является только извещение о наступлении определенного состояния/события. Если несколько одинаковых стандартных сигналов были посланы, когда прием сигналов процессом был временно заблокирован, после снятия блокировки процессу будет доставлен только один сигнал.

Сигналы реального времени — сигналы с номерами из интервала [SIGRTMIN, SIGRTMAX], определяемого соответствующими константами ОС. В отличие от стандартных сигналов, сигналы реального времени могут нести с собой дополнительные данные. Если несколько одинаковых сигналов реального времени были посланы, когда прием сигналов процессом был временно заблокирован, после снятия блокировки процессу будут доставлены все сигналы в порядке их отправки.

Сигнал SIGKILL – специальный сигнал, получение которого всегда приводит к уничтожению процесса. Данный сигнал не может быть ни перехвачен обработчиком, ни заблокирован.

Неименованный канал — объект ОС, реализующий однонаправленную передачу информации с одного конца канала на другой конец канала.

Именованный канал — канал, дополнительно имеющий отражение в файловой системе в виде специального файла (следовательно, поддерживает вызов open).

Очередь сообщений — объект ОС, имеющий отражение в виде файла в файловой системе и предназначенный для обмена порциями данных (сообщениями).

1.2 Полезные системные вызовы и библиотечные функции

1.2.1 Процессы

1.2.1.1 Идентификаторы процесса

```
pid_t pid = getpid(); //получить идентификатор процесса

pid_t ppid = getppid(); //получить идентификатор процесса-родителя
```

1.2.1.2 Создание процесса

Новый процесс может быть создан системным вызовом `fork` (англ. вилка).

```
pid_t pid = fork(); //клонирование текущего процесса
```

Процесс возвращает идентификатор созданного процесса для процесса-родителя. В созданном процессе-потомке результатом `fork` будет 0. В случае ошибки `fork` вернет -1.

Новый процесс является полной копией процесса родителя. В том числе все открытые родительским процессом файлы будут также открыты и в новом процессе. Стандартные потоки также будут общими для обоих процессов.

Пример приведен в **приложении 1**.

1.2.1.3 Завершение процесса

Нормальное завершение работы процесса происходит либо при завершении функции `main`, либо при вызове функции `exit` (не является системным вызовом).

```
void exit(int status); //завершить процесс
```

В ходе нормального завершения процесса буферы всех стандартных потоков записываются по назначению (потери данных в буферах не происходит). Данное ограничение не касается буферов открытых файлов: они не сбрасываются автоматически, пользователь должен сам их освободить.

Замечание: в случае C++ в результате `exit()` будут вызваны только деструкторы глобальных переменных. Деструкторы локальных переменных вызваны не будут, т. к. не будет инициирован процесс раскрутки стека → нельзя полагаться на идиому RAII.

Если перед непосредственно завершением процесса необходимо совершить дополнительные действия (например, сбросить буферы открытых файловых потоков через `fflush`, записать что-то в лог или вывести итоговое сообщение на экран), можно зарегистрировать функцию, которая выполнит необходимые действия, с помощью функции `atexit` (см. **приложение 2**)

```
/*зарегистрировать функцию, которая будет выполнена в ходе нормального завершения процесса*/

     int atexit(void (*function)(void));
```

Функция возвращает 0 в случае успешной регистрации, и -1 — в случае ошибки. С помощью данного вызова можно зарегистрировать до `ATEXIT_MAX` функций, которые будут выполнены в порядке, обратном порядку регистрации.

Обратите внимание, что нет возможности дерегистрировать функцию.

Вызова функций-обработчиков можно избежать, вызвав вместо функции exit системный вызов `_exit`.

```
    void _exit(int status); //завершить процесс немедленно
```

В результате вызова `_exit` процесс будет завершен немедленно. Это означает, что даже буферы стандартных потоков не будут записаны по назначению → возможна неконтролируемая потеря данных.

1.2.1.4 Приостановка потока

Если процесс имеет только один поток, то процесс приостанавливается при приостановке этого потока.

Поток процесса может автоматически приостанавливаться в случае некоторых блокирующих вызовов (например, read из заблокированного файла или пустого канала). Если необходимо временно приостановить поток на определенное время, можно воспользоваться функцией `sleep`.

```
 #include <unistd.h>

/*приостановить поток процесса на заданное количество секунд*/
unsigned int left = sleep(unsigned int seconds);
```

Функция возвращает 0, если поток был приостановлен на заданное время или более (т. е., поток «проспал» столько, сколько необходимо). Если поток был возобновлен раньше, чем требовалось, функция возвращает число секунд, которые оставались до момента планируемого возобновления (т. е., сколько поток «недоспал»). Причиной более раннего пробуждения обычно является сигнал.

Для более точного контроля над временем можно использовать вызов `nanosleep` (см. ***man nanosleep***)

1.2.1.5 Ожидание завершения процесса-потомка

Дождаться завершения процесса-потомка можно вызовами `wait/waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>

/*дождаться завершения любого из процессов-потомков*/
pid_t wait(int *wstatus);

/*дождаться изменения состояния процесса-потомка*/
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Оба системных вызова возвращают pid процесса-потомка, который завершился или изменил состояние. В случае ошибки возвращается -1. Вызов `waitpid()` может вернуть 0 — число, не являющее идентификатором ни одного процесса (см. ниже).

Вызов `wait()` блокирует поток-родитель до тех пор, пока не завершится один из его потомков, что может быть использовано для синхронизации.

Вызов `waitpid()` по умолчанию (при `options` == 0) дожидается завершения процесса, pid которого указан в параметре `pid`. Вызов `waitpid` может также отслеживать изменение состояния процесса: завершение, приостановку и продолжение выполнения. За модификацию поведения `waitpid()` отвечает параметр `options` (подробнее см***. man waitpid***).

Параметр `wstatus` может содержать в себе указатель на переменную, в которую будет помещен код выхода завершившегося процесса или др. информация о его статусе. Для этого параметра допустимо значение `NULL`. Проверить причину завершения процесса можно макросами `WIFEXITED(wstatus)` (возвращает >0, если процесс был завершен нормально),

`WIFSIGNALED(wstatus)` (возвращает >0, если процесс был уничтожен неожиданным сигналом). Код выхода может быть получен макросом `WEXITSTATUS(wstatus)`, номер сигнала — макросом `WTERMSIG(wstatus)`.

Пример — в **приложении 3**.

1.2.2 Сигналы

Сигналы являются базовым способом коммуникации ОС с потоками, а также потоков между собой. Пример работы с сигналами приведен **в приложении 4**.

1.2.2.1 Общие замечания

Сигналы являются асинхронными событиями, которые потенциально могут прийти в любой момент времени. Если для сигнала установлена функция-обработчик, она также может быть вызвана в любой момент времени. Более того, один обработчик потенциально может прервать другой обработчик. Как следствие, функция-обработчик обязана удовлетворять следующим основным ограничениям:

1. любая функция, вызываемая внутри обработчика, обязана быть или реентерабельной или явно помеченной, как безопасная для использования в обработчиках сигналов (список можно найти в ***man signal-safety***);
2. любые операции с внешними переменными (т.е., определенными вне обработчика) обязаны быть атомарными.

Если функция использует кучу, глобальные неконстантные переменные или вызывает нереентерабельные функции, то она не является реентерабельной.

Функции `malloc()/free()` и операторы `new/delete` **нереентерабельны**.

Атомарной является операция, которая с точки зрения внешнего наблюдателя не имеет промежуточного состояния (т.е., внешний наблюдатель видит или состояние до, или состояние после операции). В случае с обработчиками сигналов гарантированно атомарным является присваивание или чтение (и ничего другого!) значения переменной типа `volatile sig_atomic_t`. Сложение/вычитание и прочие арифметические операции по умолчанию неатомарны, поскольку состоят из 3 операций – чтения, изменения и записи – между которыми поток может быть прерван.

1.2.2.2 Отправка сигналов

Послать сигнал выбранному процессу можно вызовом `kill`:

```
#include <sys/types.h>
#include <signal.h>
/*послать сигнал процессу (и потенциально «убить» его) */
int kill(pid_t pid, int sig);
```

Вызов возвращает -1 при ошибке.

Аргумент `pid` содержит pid процесса, которому посылается сигнал. Допустимы также значения 0 и <0 (см. ***man 2 kill***).

Номер сигнала в аргументе `sig` должен быть одной из именованных констант, начинающихся на `SIG` (например, `SIGALRM`).

1.2.2.3 Маска сигналов

Временно запретить доставку процессу некоторых сигналов можно вызовом `sigprocmask`. При этом модифицируется маска сигналов процесса.

```
#include <signal.h>
/*задать маску сигналов */
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

int sigemptyset(sigset_t *set); // инициализировать пустую маску

int sigfillset(sigset_t *set); // добавить все сигналы

int sigaddset(sigset_t *set, int signum); // добавить сигнал

int sigdelset(sigset_t *set, int signum); // удалить сигнал
```

Вызов возвращает -1 при ошибке.

Параметр `set` указывает на устанавливаемую маску сигналов. Предыдущая маска возвращается по указателю в `oldset` (допустимо передать NULL).

Параметр `how` должен содержать одну из констант `SIG_BLOCK`, `SIG_UNBLOCK` или `SIG_SET`, указывающих, как изменяется маска сигналов:

- `SIG_BLOCK` – сигналы в аргументе set должны быть заблокированы;

- `SIG_UNBLOCK` – что сигналы должны быть разблокированы;

- `SIG_SETMASK` – что маска должна в точности быть равна set.

Если заблокированный сигнал посылался процессу несколько раз, после разблокировки процессу будет доставлен только 1 сигнал (это не касается сигналов реального времени).

Замечание: нельзя запретить доставку сигналов `SIGKILL` и `SIGSTOP`.

1.2.2.4 Обработка сигналов

Установить функцию, которая будет вызвана при получении сигнала, можно системным вызовом `sigaction`.

```
#include <signal.h>

struct sigaction {
          void     (*sa_handler)(int);
          void     (*sa_sigaction)(int, siginfo_t *, void *);
          sigset_t   sa_mask;
          int        sa_flags;
      };

/* задать реакцию на сигнал */
int sigaction(int signum, const struct sigaction *act,
                struct sigaction *oldact);
```

Номер сигнала, для которого задается обработчик, определяется аргументом `signum`.

Данные об обработчике сигнала передаются в аргументе `act`.

Данные о предыдущем обработчике (или о текущем, если `act==NULL`) можно записать по указателю в `oldact`.

Структура `sigaction` служит для передачи информации об обработчике.

Поле `sa_handler` содержит указатель на функцию обработчик (или константу диспозиции, см. след пункт). Аргументом функции-обработчика является номер пришедшего сигнала → один и тот же обработчик может использоваться для нескольких сигналов и различать их по значению аргумента.

В качестве значения `sa_handler` могут быть переданы 2 именованные константы. Константа `SIG_DFL` сбрасывает установленный обработчик сигнала (сигнал будет вызывать действие по умолчанию). Константа `SIG_IGN` заставляет программу *игнорировать* сигнал. Игнорирование отличается от блокировки, т. к. блокировка лишь откладывает обработку сигнала, а игнорирование — полностью отменяет всякую реакцию на сигнал.

Поле `sa_mask` содержит маску сигналов, которые блокируются на время исполнения обработчика.

Поле `sa_flags` содержит дополнительные флаги. Если в данном поле указан флаг `SA_SIGINFO`, то обработчик берется из поля `sa_sigaction`. При этом стоит отметить изменившуюся сигнатуру — помимо номера сигнала в функцию передается адрес структуры `siginfo_t` и указатель на контекст.

```
union sigval_t{
    int   sival_int;
    void* sival_ptr;
}

struct siginfo_t {
           int      si_signo;     /* Номер сигнала */
           int      si_errno;     /* Значение errno */
           int      si_code;      /* Доп. информация о сигнале */

           pid_t    si_pid;       /* ИД процесса-отправителя */
           uid_t    si_uid;       /* Реальный ИД пользователя */

           int      si_status;    /* Код выхода или номер сигнала */

           sigval_t si_value;     /* Передаваемое значение */
           void    *si_addr;      /* Адрес ошибки */
}
```

В данной структуре особый интерес представляет поле **si_value**, в котором сигналы реального времени могут передать небольшую порцию данных — целое число либо указатель.

Для информации об остальных полях смотрите ***man 2 sigaction***.

1.2.2.5 Синхронизация по сигналу

Остановить процесс до получения любого из заданных сигналов можно вызовом `sigsuspend`.

```
#include <signal.h>

/* подождать получения сигнала из маски */
int sigsuspend(const sigset_t *mask);
```

Данная функция всегда возвращает -1 с `errno==EINTR`.

Данный вызов атомарно изменяет маску сигналов, блокирует процесс до получения любого из незаблокированных сигналов, а затем возвращает старую маску сигналов.

Если требуется выполнить некоторое действие по истечении заданного времени, необходимо задать обработчик сигнала `SIGALRM`, а затем вызвать функцию `alarm`:

```
#include <unistd.h>

/*Послать себе сигнал SIGALRM по истечении времени*/
unsigned int alarm(unsigned int seconds);
```

Вызов возвращает число секунд, остававшихся до окончания таймера, установленного предыдущим вызовом `alarm.` Иными словами, `alarm` всегда отменяет предыдущий таймер.

Замечание: на некоторых системах функция `sleep()` реализована через `alarm()`, следовательно, рекомендуется не использовать эти функции вместе, во избежание сбоев таймера.

1.2.2.6 Сигналы реального времени

Обычные сигналы не несут с собой дополнительной информации. Для передачи информации между процессами можно использовать специальные сигналы реального времени. Номера сигналов реального времени находятся в интервале [`SIGRTMIN, SIGRTMAX`]. Послать сигнал реального времени вместе с доп. информацией можно вызовом `sigqueue`.

```
#include <signal.h>

// послать сигнал реального времени
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Вызов возвращает -1 при ошибке.

Pid процесса-цели передается в `pid`. Номер сигнала передается в `sig`. Значение, передаваемое вместе с сигналом, передается в `value`.

Замечание: если сигнал реального времени был заблокирован, и во время блокировки посылался процессу N раз, то все N сигналов будут поочередно доставлены процессу.

1.2.4 Межпроцессное взаимодействие

Процессы имеют собственные и по умолчанию непересекающиеся адресные пространства, следовательно, взаимодействие процессов между собой и передача данных между процессами должно организовываться специальными методами (помимо сигналов).

1.2.4.1 Каналы

Каналы позволяют организовать *однонаправленную* передачу данных между двумя или более процессами.

Неименованный канал создается вызовом `pipe`. Пример приведен в **приложении 5**.

```
#include <unistd.h>

// создать канал и вернуть ассоциированные дескрипторы
int pipe(int pipefd[2]);
```

Вызов создает 2 дескриптора. Первый дескриптор – `fd[0]` – предназначен только для чтения из канала, второй – `fd[1]` – только для записи.

Если требуется организовать двустороннее взаимодействие между процессами, потребуется 2 канала — один для передачи данных от А к Б, второй — для передачи данных от Б к А.

Если требуется предоставить доступ к каналу любым процессам, можно создать именованный канал функцией mkfifo. Пример работы с именованным каналом приведен в **приложении 6.**

```
#include <sys/types.h>
#include <sys/stat.h>

//создать именованный канал с заданными правами доступа
int mkfifo(const char *pathname, mode_t mode);
```

Данная функция создаст специальный файл, представляющий собой канал. Для дальнейшей работы с каналом он должен быть открыт вызовом open, после чего с каналом можно будет работать, как с обычным файлом (см**. *man 2 read/write***).

Доступ к каналу для процессов других пользователей определяется параметром mode (см. ***man 2 open***, константы `S_IRWXU`, ...) аналогично обычным файлам. Одновременно несколько процессов могут открыть канал на чтение и на запись (например, несколько процессов-писателей поставляют данные для обработки одному процессу-читателю).

В ядре ОС каждому каналу соответствует буфер размером, как минимум, PIPE_BUF (константа, значение которой можно получить через pathconf). Запись в канал данных размером менее PIPE_BUF производится атомарно. При чтении из канала место в буфере освобождается. Если буфер полон, то поток-писатель будет приостановлен при попытке записи до освобождения места в нем.

Чтение данных блоками размером до PIPE_BUF будет производиться атомарно. Если данных в канале недостаточно, поток-читатель блокируется при попытке чтения до появления новых данных.

У канала должны быть открыты оба конца (для записи и для чтения). Запись в канал, у которого не открыт конец для чтения, будет завершаться с ошибкой EPIPE. Чтение из канала, у которого закрыт конец для записи, будет возвращать 0 — признак конца файла (если конец для записи открыт, но в канале нет данных, то вызов read заблокирует процесс до появления данных).

1.2.4.2 Очереди сообщений

Очереди сообщений предоставляют возможность несколько более упорядоченно обмениваться информацией между процессами. Элементарной единицей в этом случае является сообщение, содержащее в себе данные и их длину. Пример работы с очередью сообщение приведен в **приложении 7**.

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>

//открыть очередь
mqd_t mqdes = mq_open(const char *name, int oflag, mode_t mode,
                  struct mq_attr *attr);

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

//прочитать сообщение в буфер
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);

//закрыть очередь
int mq_close(mqd_t mqdes);

//удалить очередь
int mq_unlink(const char *name);
```

Все вызовы возвращают -1 при ошибке.

Вызов `mq_open` во многом аналогичен `open` с теми же параметрами (***man 2 open***). Последний параметр может содержать атрибуты создаваемой очереди (в т.ч. максимальный размер сообщения), но может быть равен `NULL`.

Запись сообщения в очередь производится функцией `mq_send`. В очереди одновременно может быть записано ограниченное количество сообщений (по умолчанию — 10). Максимальный размер сообщения также ограничен (по умолчанию - 8КБ). Если очередь заполнена, то процесс по умолчанию блокируется до освобождения места в очереди.

Параметр `msg_prio` содержит приоритет сообщения. Сообщения с б*о*льшим приоритетом доставляются быстрее.

Чтение сообщения из очереди производится функцией `mq_receive`. Функция возвращает размер принятого сообщения. Буфер и его размер передаются в качестве 2 и 3 аргументов. Буфер не может быть меньше, чем размер сообщения очереди, иначе возникнет ошибка `EMSGSIZE`. Чтение из пустой очереди по умолчанию блокирует поток до появления следующего сообщения.

Очередь может быть закрыта функцией `mq_close` и удалена функцией `mq_unlink` (очередь продолжит существовать, пока открыта хотя бы в 1 процессе). Очереди уничтожаются при перезагрузке.

По умолчанию чтение/запись в очередь может заблокировать поток, если очередь пуста/полна. Открыть очередь в неблокирующем режиме можно путем передачи флага `O_NONBLOCK` во 2 аргументе `mq_open`, при этом попытка чтения/записи из пустой/полной очереди будет завершаться с ошибкой `EAGAIN` вместо блокировки потока.

2. Задание на лабораторную работу.

Написать программу, которая играет сама с собой в «угадай число». Число находится в диапазоне от 1 до N (параметр программы, передается как аргумент или вводится сразу после запуска).

В игре участвуют 2 процесса, представляющих собой игроков. Второй процесс создается через `fork()`.

Первый игрок загадывает число от 1 до N и извещает об этом второго игрока.

Второй игрок начинает угадывать число. Попытки и результат выводятся на экран.

После того, как второй игрок угадывает число, на экран выводится статистика игры – число попыток и, опционально, время игры, после чего игроки меняются местами, и игра начинается заново (число циклов может быть конечным или бесконечным – по желанию, но не менее 10).

Задание 1. Решить задачу с помощью сигналов. Передачу предположений (целых чисел) организовать с помощью сигналов реального времени. Извещение угадывающего процесса организовать с помощью стандартных сигналов `SIGUSR1` (угадал) и `SIGUSR2` (не угадал).

Задание 2. Решить задачу с помощью одного из средств межпроцессного взаимодействия:

Вариант 1: именованный канал;

Вариант 2: очередь сообщений;

Вариант 3: неименованный канал.

Запрещается использовать sleep/nanosleep для синхронизации (т.е., если корректность работы программы зависит от наличия sleep – задача решена неверно).

Легкий уровень: выполнить задание 1 и задание 2 без дополнительных требований.

Средний уровень: завершение одного из процессов должно приводить к корректному завершению второго процесса (для этого необходимо исключить возможность вечного ожидания сигнала/сообщения из очереди).

Сложный уровень: выполните второе задание в 2 вариантах - с помощью очереди сообщений и одного из каналов. Завершение одного из процессов должно приводить к корректному (неаварийному) завершению второго процесса во всех случаях.

Приложение 0. Файл check.hpp

```
#ifndef CHECK_HPP
#define CHECK_HPP 1

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

inline void error(const char* file, int line) {
	auto tmp = errno;//fprintf may fail, so we preserve errno
	fprintf(stderr, "%s (line %d) :", file, line);
	errno = tmp;
	perror(NULL);
	exit(EXIT_FAILURE);
}

inline int xcheck(int p, const char* file, int line) {
	if (p < 0) error(file, line);
	return p;
}

template<typename T>
inline T* xcheck(T* p, const char* file, int line) {
	if (p == nullptr)  error(file, line);
	return p;
}

#define check(x) xcheck(x, __FILE__, __LINE__ )


#endif // !CHECK_HPP
```

Приложение 1. Простой пример создания процесса (без синхронизации вывода)

```
#include <unistd.h>
#include <sys/types.h>
#include <iostream>

int main() {
	auto id = fork();

	//попробуйте раскомментировать :)
	//std::cout << std::unitbuf; // отключить буферизацию

	if (id>0) { /*id > 0, выполняется процесс-родитель*/
		std::cout << getpid() << ">> I am your father!" << std::endl;
	}
	else {	/*id == 0, выполняется процесс-потомок*/
		std::cout << getpid() << ">> Noooooooooo!" << std::endl;
	}
}
```

Приложение 2. Выполнение действий при завершении процесса

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void on_end1() {
	printf("But I'm not dead yet");
}

void on_end2() {
	printf("Oh no, I'm dying...");
}

int main() {
	if (atexit(on_end1)) {
		perror("Failed to register function 1");
		_exit(EXIT_FAILURE);
	}
	if (atexit(on_end2)) {
		perror("Failed to register function 2");
		_exit(EXIT_FAILURE);
	}
	exit(EXIT_SUCCESS); // необязательно
}
```

Приложение 3. Ожидание потомков процессом-родителем

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>


int spawn(int num) {
	pid_t id = fork();
	if (id < 0) {
		perror("Failed to create child");
		_exit(EXIT_FAILURE);
	}
	if (id)		// id!= 0,
		return id; //we are in the parent process, return child ID

	printf("Child%d >> I'm born!\n", num);
	sleep(5);
	printf("Child%d >> I'm done!\n", num);
	exit(10);
}

int main() {
	pid_t ch1 = spawn(1);

	pid_t ch2 = spawn(2);

	sleep(1);
	kill(ch2, SIGABRT); //kill the second child

	int stat;
	waitpid(ch1, &stat, 0);
	printf("Has child 1 exited? %s Exit code is %d\n"
		, WIFEXITED(stat) ? "yes" : "no"
		, WEXITSTATUS(stat));

	waitpid(ch2, &stat, 0);
	printf("Was child 2 teminated? %s, process was killed by %s\n"
		, WIFSIGNALED(stat) ? "yes" : "no"
		, strsignal(WTERMSIG(stat))); // strsignal gives description
}
```

Приложение 4. Пример работы с сигналами

```
#include "check.hpp"
#include <pthread.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>

volatile sig_atomic_t last_sig;
volatile sig_atomic_t sig_val;

void sig_handler(int signo) {
	//can not call printf here cause it’s not reentrant
	last_sig = signo;
}

void rtsig_handler(int signo, siginfo_t* si, void* ctx) {
	//can not call printf here cause it's not reentrant
	last_sig = signo;
	sig_val = si->si_value.sival_int;
}

void child() {
	struct sigaction abrt_action {}
			, quit_action{}
			, rt_action{}
			, kill_action{};

	sigset_t set;
	sigemptyset(&set);
	//block SIGTERM
	sigaddset(&set, SIGTERM);
	check(sigprocmask(SIG_SETMASK, &set, NULL));

	//ignore SIGABRT
	abrt_action.sa_handler = SIG_IGN;
	check(sigaction(SIGABRT, &abrt_action, NULL));

	//handle SIQUIT
	quit_action.sa_handler = sig_handler;
	check(sigaction(SIGQUIT, &quit_action, NULL));

	//handle SIGRTMAX
	rt_action.sa_sigaction = rtsig_handler;
	rt_action.sa_flags = SA_SIGINFO;
	check(sigaction(SIGRTMAX, &rt_action, NULL));


	printf("I want to do something bad >:>\n");
	while (true) {
		sigsuspend(&set); //wait for any non-blocked signal
```

```
            printf("I've got a signal %d ", last_sig);
            if(last_sig >= SIGRTMIN)
                  printf("with a value %d. Nice present :)", sig_val);
            printf("\n");
      }

}

bool exists(pid_t p) {
      int stat;
      return waitpid(p, &stat, WNOHANG) == 0;
}

void try_kill(pid_t p, int sig) {
      if (sig < SIGRTMIN)
            check(kill(p, sig));
      else
            check(sigqueue(p, sig, sigval{ 1024 }));
      sleep(1);
      if (!exists(p)) {
            printf("Убил наконец...\n");
            exit(EXIT_SUCCESS);
      }
      printf("Выжил, однако...");
}

void parent(pid_t child) {
      sleep(2);
      printf("Йа тебя породил, йа тебя и убью!\n");

      printf("Попробуем SIGTERM\n");
      try_kill(child, SIGTERM);

      printf("Попробуем SIGABRT\n");
      try_kill(child, SIGABRT);

      printf("Попробуем SIGQUIT\n");
      try_kill(child, SIGQUIT);

      printf("Попробуем SIGRTMAX\n");
      try_kill(child, SIGRTMAX);

      printf("SIGKILL точно убьет\n");
      try_kill(child, SIGKILL);
}

int main() {
      pid_t child_id = check(fork());
      if (child_id) parent(child_id);
      else          child();
}
```

Приложение 5. Пример работы с неименованным каналом

```
#include "check.hpp"
#include <unistd.h>
#include <wait.h>

int child(int pipe_fd) {
	int buffer, r;
	do {
		r = check(read(pipe_fd, &buffer, sizeof(int)));
		if (r > 0)
			printf("Child >> Got %d from the pipe\n", buffer);
	} while (r > 0);
}

int parent(int pipe_fd) {
	int x;
	for (int i = 0; i < 10; ++i) {
		printf("Parent >> Writing %d into the pipe\n", i);
		x = check(write(pipe_fd, &i, sizeof(i)));
		sleep(1);
	}
}

int main() {
	int fd[2];
	check(pipe(fd)); //make pipe
	int pid = check(fork());
	if (pid) {
		check(close(fd[0])); //close the unused pipe end
		parent(fd[1]);       //use writing descriptor

		// close the writing pipe end.
		// otherwise the child's read() will block forewer
		check(close(fd[1]));
		int stat;
		wait(&stat);
	}
	else {
		check(close(fd[1])); //close the unused pipe end
		child(fd[0]);        //use reading descriptor
	}
}
```

Приложение 6. Пример работы с именованным каналом

```
#include "check.hpp"
#include <unistd.h>
#include <wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>

int child(int pipe_fd) {
	int buffer, r;
	do {
		r = check(read(pipe_fd, &buffer, sizeof(int)));
		if (r > 0)
			printf("Child >> Got %d from the pipe\n", buffer);
	} while (r);
}

int parent(int pipe_fd) {
	int x;
	for (int i = 0; i < 10; ++i) {
		printf("Parent >> Writing %d into the pipe\n", i);
		x = check(write(pipe_fd, &i, sizeof(i)));
		sleep(1);
	}
}

int main() {
	// create pipe
	check(mkfifo("/tmp/myfifo", S_IRUSR|S_IWUSR));
	int pid = check(fork());
	if (pid) {
		int fd = open("/tmp/myfifo", O_WRONLY); //open for writing
		parent(fd);

		// close the pipe end.
		// otherwise the child's read() will block
		check(close(fd));
		int stat;
		wait(&stat);
		check(unlink("/tmp/myfifo"));// delete the pipe
	}
	else {
		int fd = open("/tmp/myfifo", O_RDONLY); //open for reading
		child(fd);
	}
}
```

Приложение 7. Пример работы с очередью сообщений

```
#include "check.hpp"
#include <unistd.h>
#include <wait.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

int child(mqd_t mqd) {
	int* buf = new int[8 * 1024 / sizeof(int)];
	int r;
	do {
		r = check(mq_receive(mqd, (char*)buf, 8 * 1024, NULL));
		printf("Child >> Got %d from the queue\n", buf[0]);
	} while (r > 0 && *buf != -1);
	delete[] buf;
}

int parent(mqd_t mqd) {
	int i = 0;
	for (i = 0; i < 10; ++i) {
		printf("Parent >> Writing %d into the queue\n", i);
		check(mq_send(mqd, (char*)&i, sizeof(i), 0));
		sleep(1);
	}
	i = -1;
	//send end message
	check(mq_send(mqd, (char*)&i, sizeof(i), 0));
}

int main() {
	{ //ensure the message queue exists
		auto d = check(mq_open("/mymq", O_CREAT, S_IRUSR | S_IWUSR,
NULL));
		check(mq_close(d)); //free descriptor
	}
	pid_t p = check(fork());
	if (p) {
		mqd_t mqd = check(mq_open("/mymq",O_WRONLY)); //open for write
		parent(mqd);
		int stat;
		wait(&stat);
		mq_unlink("/mymq"); //remove the queue from the filesystem
	}
	else {
		mqd_t mqd = check(mq_open("/mymq", O_RDONLY)); //open for read
		child(mqd);
	 }
}
```